

PONTIFICIA UNIVERSIDAD CATÓLICA DEL ECUADOR FACULTAD DE HÁBITAT, INFRAESTRUCTURA Y CREATIVIDAD

CARRERA: INGENIERÍA EN SISTEMAS DE INFROMACIÓN

Trabajo de Titulación como requisito previo para la obtención del título de

Diseño y Desarrollo de un Sistema Web para la Gestión de Inventario y Servicio Técnico de Equipos de Cómputo en Xkale Cia. Ltda.

Autor: Camilo Andrés Buitrón Arroyo
Director/a – Tutor/a: PhD Pablo Daniel Marcillo Lara

QUITO DM, JUNIO 2026

DEDICATORIA

A mis padres, por entregarme cada día más de lo que merezco, por enseñarme con el ejemplo el valor del trabajo honesto y por convertir cada uno de mis sueños en un proyecto compartido. Sin su sacrificio silencioso y su apoyo incondicional, nada de lo que hoy alcanzo habría sido posible.

A mis abuelos, por la herencia más valiosa que puede transmitirse, las historias, los valores y el cariño que perdura más allá del tiempo. Su ejemplo de trabajo, integridad y cariño familiar sigue iluminando cada decisión importante de mi vida.

A todos ellos dedico este trabajo, con la certeza de que cada página lleva un poco de su esfuerzo y de su compañía.

AGRADECIMIENTO

Quiero expresar mi más profundo agradecimiento a mis padres, por el sacrificio que han hecho durante todos estos años para brindarme la oportunidad de formarme académicamente. Su apoyo incondicional, su paciencia en los momentos difíciles y la fe que siempre depositaron en mí constituyen el cimiento sobre el cual se levanta este logro.

A mi hermano, gracias por acompañarme en este recorrido con la complicidad y el cariño que solo un hermano puede ofrecer. El estar presente en este camino hizo que para mí sea mucho más ameno y se disfruten más los buenos momentos.

A mis abuelos agradezco profundamente su motivación y acompañamiento en esta etapa, por haberme dado la energía necesaria para siempre seguir adelante y las sonrisas constantes.

Mi profundo agradecimiento a la Pontificia Universidad Católica del Ecuador y a la Facultad de Hábitat, Infraestructura y Creatividad. Más allá de la instrucción técnica, valoro enormemente la formación humana e integral que recibí en sus aulas a lo largo de estos años.

Así mismo, quiero expresar mi gratitud a los docentes de la carrera de Ingeniería en Sistemas de Información, gracias por desafiarme constantemente, por su exigencia y por transmitir su experiencia sin medida.

Agradezco también a mi tutor, Pablo Marcillo, por acompañar este proyecto desde el inicio. Sus observaciones y correcciones fueron clave para mejorar el trabajo.

A mis compañeros de carrera, gracias por las horas de estudio juntos y, sobre todo, por las risas en medio del estrés. Hicieron que estos años fueran mucho más llevaderos.

Por último, gracias a mis equipos de trabajo en Xkale Cía. Ltda. y CiberC S.A. por recibirme y enseñarme con paciencia. Ahí aprendí mucho más que lo técnico.

Índice de Contenidos

CAPÍTULO 1: INTRODUCCIÓN 1

1.1 Planteamiento del Problema 1

1.2 Justificación 2

1.3 Objetivos 3

1.3.1 Objetivo General 3

1.3.2 Objetivos Específicos 3

1.4 Alcance y Limitaciones 4

1.4.1 Alcance Funcional 4

1.4.2 Alcance Técnico 4

1.4.3 Alcance Operativo 5

1.4.4 Limitaciones 5

CAPÍTULO 2: FUNDAMENTACIÓN TEÓRICA 6

2.1 Gestión de Activos de Tecnologías de Información (ITAM) 6

2.2 Arquitectura y Patrones de Desarrollo Web 7

2.2.1 Arquitectura Cliente-Servidor 7

2.2.2 Arquitectura de Tres Capas 7

2.2.3 APIs REST 8

2.3 Tecnologías de Backend 8

2.3.1 Lenguaje de Programación Python 8

2.3.2 Framework FastAPI 9

2.4 Tecnologías de Frontend 9

2.4.1 Fundamentos: HTML, CSS y JavaScript 10

2.4.2 Biblioteca React 10

2.4.3 Sistema de Estilos con Tailwind CSS 10

2.5 Bases de Datos Relacionales 11
2.5.1 Modelo Relacional 11
2.5.2 PostgreSQL 12
2.6 Metodología Kanban 12
CAPÍTULO 3: ANÁLISIS Y DISEÑO DEL SISTEMA 14
3.1 Análisis de Requerimientos 14
3.1.1 Requerimientos Funcionales 14
3.1.2 Requerimientos No Funcionales 16
3.2 Casos de Uso del Sistema 17
3.2.1 Identificación de Actores 17
3.2.2 Diagrama General de Casos de Uso 18
3.2.3 Especificación Detallada de Casos de Uso 18
3.3 Diseño de la Base de Datos 22
3.3.1 Modelo Conceptual 22
3.3.2 Modelo Lógico 23
3.3.3 Modelo Físico 23
3.3.4 Diccionario de Datos 25
3.3.5 Reglas de Integridad 27
3.4 Diseño de la Interfaz de Usuario 28
3.4.1 Principios de Diseño Aplicados 28
3.4.2 Identidad Visual y Paleta Cromática 29
3.4.3 Arquitectura de Navegación 30
3.4.4 Descripción de Pantallas Principales 30
CAPÍTULO 4: DESARROLLO DE LA APLICACIÓN 32
4.1 Configuración del Entorno de Desarrollo 32

4.1.1 Herramientas y Versiones 32
4.1.2 Estructura del Proyecto 34
4.2 Desarrollo del Backend con FastAPI 36
4.2.1 Inicialización de la Aplicación 36
4.2.2 Configuración de la Base de Datos 38
4.2.3 Modelos de SQLAlchemy 39
4.2.4 Esquemas de Pydantic 40
4.2.5 Routers y Endpoints REST 42
4.2.6 Autenticación y Autorización 44
4.2.7 Consultas Avanzadas con JOIN 47
4.2.8 Documentación Automática de la API 48
4.3 Desarrollo del Frontend con React 48
4.3.1 Inicialización y Enrutamiento 48
4.3.2 Sistema de Estilos con Tailwind CSS 51
4.3.3 Gestión Global del Estado de Autenticación 52
4.3.4 Comunicación con la API REST 54
4.3.5 Componentes Reutilizables 56
4.3.6 Protección de Rutas y Visibilidad por Rol 56
4.4 Pruebas, Integración y Despliegue 57
4.4.1 Variables de Entorno 58
4.4.2 Servidor de Desarrollo 58
4.4.3 Estrategia de Despliegue On-Premise 59
4.4.4 Aspectos de Seguridad Corporativa 60
4.4.5 Justificación de las Decisiones Tecnológicas 61
CAPÍTULO 5: CONCLUSIONES Y RECOMENDACIONES 64

5.1 Conclusiones 64

5.2 Recomendaciones 67

CAPÍTULO 6: TRABAJOS FUTUROS 70

Bibliografía 72

Anexos 73

ANEXO A: Capturas de Pantalla del Sistema 74

ANEXO B: Script de Definición de la Base de Datos 84

ANEXO C: Catálogo de Endpoints de la API REST 88

C.1 Módulo de Autenticación 88

C.2 Módulo de Usuarios 89

C.3 Módulo de Técnicos 90

C.4 Módulo de Laptops 91

C.5 Módulo de Tickets 92

C.6 Módulo de Consultas Avanzadas 92

C.7 Endpoint de Estado del Servicio 93

C.8 Resumen del Catálogo 94

ANEXO D: Repositorio del Código Fuente 95

D.1 Información General del Repositorio 95

D.2 Acceso al Repositorio 95

D.3 Guía de Navegación del Código 96

D.4 Licencia y Propiedad Intelectual 97

Índice de Figuras

Figura 1: Diagrama General de Casos de Uso 18
Figura 2: Diagrama Entidad-Relación del Modelo Físico 24
Figura A1: Pantalla de inicio de sesión del sistema 75
Figura A2: Panel principal (Dashboard) del sistema 76
Figura A3: Módulo de inventario de laptops 76
Figura A4: Módulo de tickets de servicio técnico 77
Figura A5: Módulo de directorio de usuarios 77
Figura A6: Módulo de técnicos de soporte 78
Figura A7: Módulo de gestión de accesos al sistema 78
Figura A8: Formulario de registro de nueva laptop 79
Figura A9: Formulario de creación de ticket 80
Figura A10: Formulario de registro de nuevo usuario 81
Figura A11: Formulario de registro de nuevo técnico 82
Figura A12: Formulario de creación de credenciales de acceso 83

Índice de Tablas

Tabla 1: Requerimientos funcionales del sistema 15
Tabla 2: Requerimientos no funcionales del sistema 16
Tabla 3: Caso de Uso CU-01: Iniciar Sesión 19
Tabla 4: Caso de Uso CU-02: Registrar Nueva Laptop 19
Tabla 5: Caso de Uso CU-03: Crear Ticket de Servicio Técnico 20
Tabla 6: Caso de Uso CU-04: Gestionar Accesos al Sistema 21
Tabla 7: Casos de uso complementarios 21
Tabla 8: Resumen del modelo lógico relacional 23
Tabla 9: Diccionario de datos de la tabla auth_user 25
Tabla 10: Diccionario de datos de la tabla usuario 25
Tabla 11: Diccionario de datos de la tabla técnico 26
Tabla 12: Diccionario de datos de la tabla laptop 26
Tabla 13: Diccionario de datos de la tabla ticket 27
Tabla 14: Paleta cromática del sistema 29
Tabla 15: Herramientas y versiones empleadas en el desarrollo 32
Tabla 16: Variables de entorno del sistema 58
Tabla C1: Endpoints del módulo de autenticación 89
Tabla C2: Endpoints del módulo de usuarios 90
Tabla C3: Endpoints del módulo de técnicos 90
Tabla C4: Endpoints del módulo de laptops 91
Tabla C5: Endpoints del módulo de tickets 92
Tabla C6: Endpoints del módulo de consultas avanzadas 93
Tabla C7: Endpoint de verificación de estado 93
Tabla C8: Resumen cuantitativo de endpoints por módulo 94
Tabla D1: Información del repositorio del proyecto 95
Tabla D2: Guía de navegación de los archivos principales del repositorio 96

CAPÍTULO 1: INTRODUCCIÓN

El hardware que una empresa entrega a su personal dejó de ser un simple gasto para convertirse en un activo que hay que administrar y rastrear (Pressman, 2010). Cuando ese control no está centralizado, se vuelve difícil saber qué equipo tiene cada persona, en qué estado se encuentra y si sigue en garantía. Eso se traduce en pérdidas de tiempo y de dinero para el área de TI.

Este es el caso de Xkale Cía. Ltda., empresa multinacional con operaciones en Ecuador, Colombia y México que desarrolla soluciones de Customer Experience, incluyendo bots con IA para contact centers, para el sector financiero. Como parte de su operación, Xkale entrega computadoras portátiles a buena parte de su personal en los tres países. Hoy ese inventario se administra de forma manual, y mantener el control sobre tantos equipos repartidos se ha vuelto cada vez más difícil.

Para resolver este problema, en este trabajo se desarrolló un sistema web que centraliza el inventario de equipos y el registro de soporte técnico. La interfaz se construyó con React, la lógica del backend con FastAPI (Python) y la información se almacena en PostgreSQL. El sistema se desplegó en servidores on-premise de la empresa, de modo que los datos de los equipos no salen de su propia infraestructura.

1.1 Planteamiento del Problema

Actualmente, el área de TI de Xkale no cuenta con una plataforma única para administrar sus equipos de cómputo. La asignación de dispositivos, el control de garantías y las bitácoras de mantenimiento se llevan en archivos compartidos y hojas de cálculo sueltas. Al estar la información repartida en varios archivos, es fácil que se pierda el rastro de los soportes realizados, que se dupliquen registros y que auditar el ciclo de vida de un equipo resulte complicado.

Al no haber un software dedicado, datos importantes como las especificaciones de cada equipo, los números de factura o el historial de averías quedan dispersos o se vuelven difíciles de encontrar. Sommerville (2011) advierte que, cuando la información está descentralizada, no solo suben los costos de soporte, sino que se complica tomar decisiones basadas en datos confiables. En Xkale esto tiene un costo concreto: a veces se paga por reparar equipos que todavía están en

garantía, o se sigue usando hardware que falla seguido y que ya debería reemplazarse.

A esto se suma que no existe un registro histórico y permanente de incidencias. Sin él, el equipo de TI no puede resolver con agilidad los problemas que se repiten, no logra medir bien los tiempos de atención ni tiene cómo respaldar el cumplimiento de los proveedores externos de soporte. La gestión de activos de TI (ITAM, por sus siglas en inglés) está reconocida en la norma ISO/IEC 19770 como el conjunto de prácticas que integran las áreas financiera, contractual y de inventario para administrar el ciclo de vida de los activos tecnológicos (International Organization for Standardization, 2017). No tener un sistema alineado con esas prácticas expone a la empresa a riesgos de cumplimiento y a perder valor sobre lo que invirtió en equipos.

Ante esta situación, se plantea la siguiente pregunta principal de investigación:

¿Cómo diseñar e implementar un sistema web centralizado que optimice la trazabilidad del inventario de equipos de cómputo y el registro de incidencias técnicas, garantizando un control eficiente del ciclo de vida del hardware en Xkale Cía. Ltda.?

De esta interrogante principal se desprenden las siguientes preguntas específicas:

- ¿Cómo debe estructurarse la base de datos para relacionar a cada usuario con los equipos que tiene asignados y con su historial de mantenimiento?
- ¿Qué datos debe capturar el sistema para automatizar el control de garantías e incidencias?
- Una vez automatizados estos procesos, ¿qué efecto tienen sobre los tiempos de atención y la gobernanza que TI ejerce sobre los equipos?

1.2 Justificación

En Xkale, el trabajo diario del personal depende de que sus equipos funcionen. Si una laptop falla y no hay un registro claro de su estado, su garantía o su historial de reparaciones, se pierde tiempo y productividad. Por eso, contar con un sistema centralizado de inventario y soporte técnico no es un lujo, sino una necesidad operativa.

En lo operativo, el sistema le da a Xkale una forma estándar de registrar las laptops y controlar a quién está asignada cada una, en reemplazo de los archivos sueltos que usaban antes. Con esto es más rápido consultar el estado de un equipo, saber quién lo tiene y revisar su historial

de soportes. Pressman (2010) señala que el valor de un software está en su capacidad de resolver necesidades reales del usuario y mejorar el funcionamiento de la organización, que es justamente lo que busca esta herramienta.

En lo económico, el sistema ayuda a proteger la inversión de la empresa en equipos. Al avisar de forma automática sobre las fechas de garantía y de compra, evita gastar en reparaciones que en realidad le corresponden al fabricante. Además, registrar las fallas de forma ordenada permite ver qué modelos o marcas dan más problemas, lo que sirve para planificar las renovaciones y negociar con los proveedores con datos concretos.

En lo técnico, el proyecto fue una oportunidad para aplicar lo aprendido en la carrera con herramientas actuales y muy usadas en la industria. La interfaz se desarrolló en React, que por su arquitectura de componentes facilita el mantenimiento (Banks & Porcello, 2020). El backend se construyó en Python con FastAPI, apropiado para crear servicios REST eficientes, y PostgreSQL mantiene la integridad de los datos bajo el modelo relacional (Elmasri & Navathe, 2016).

Por último, los requisitos de seguridad de la empresa se cubren con un despliegue on-premise. Al alojar la aplicación en servidores locales, los datos de los equipos y del personal permanecen dentro de la infraestructura de Xkale, lo que ayuda a cumplir con la confidencialidad que exigen sus clientes en los sectores financiero y de seguros.

1.3 Objetivos

1.3.1 Objetivo General

Desarrollar un sistema web para el control de inventario y el registro de incidencias técnicas de los equipos de cómputo de Xkale Cía. Ltda., que permita administrar el ciclo de vida del hardware desde su registro hasta su baja.

1.3.2 Objetivos Específicos

- Diseñar la base de datos del sistema, definiendo el modelo entidad relación y los esquemas lógico y físico necesarios para gestionar los usuarios, los equipos, los técnicos y las bitácoras de servicio.

- Desarrollar el módulo de gestión de inventario que permita el registro detallado de las características de hardware, la información de facturación y la asignación de los equipos a los colaboradores de la organización.
- Implementar el módulo de servicio técnico para documentar incidencias, acciones de mantenimiento, gestiones de garantía y feedback sobre las reparaciones realizadas, generando una bitácora histórica consultable por el personal autorizado.
- Desplegar el sistema en los servidores internos de la organización bajo una arquitectura on-premise, asegurando la disponibilidad, el rendimiento y el acceso controlado para el uso exclusivo del personal del departamento de Tecnologías de Información.

1.4 Alcance y Limitaciones

Este proyecto abarca el diseño, desarrollo e implementación interna de un sistema web para administrar los equipos de cómputo de Xkale Cía. Ltda. El alcance se describe a continuación en tres partes la funcional, la técnica y la operativa y se cierra con las limitaciones del trabajo.

1.4.1 Alcance Funcional

El sistema contempla tres módulos. El de usuarios centraliza los datos del personal: correo corporativo, nombre completo y rol dentro de la plataforma. El de computadoras funciona como inventario maestro y guarda la marca, el modelo y las especificaciones de cada laptop, junto con datos administrativos como la fecha de compra, el número de factura, el estado del equipo y la persona responsable de su custodia. Por último, el módulo de servicio técnico lleva el historial de cada equipo: fecha de la incidencia, descripción de la falla, acción del técnico, estado del trámite de garantía cuando aplica y resolución final.

1.4.2 Alcance Técnico

En lo técnico, el frontend se desarrolló con React, lo que permite una interfaz responsiva basada en componentes reutilizables. El backend se implementó en Python con FastAPI, que expone una API REST consumida por el cliente web. Los datos se almacenan en PostgreSQL, encargado de mantener la integridad referencial y resolver las consultas sobre las entidades del modelo. El despliegue se realiza en los servidores locales de la empresa, bajo una arquitectura on

premise.

1.4.3 Alcance Operativo

El sistema estará destinado al uso exclusivamente interno de Xkale Cía. Ltda. Los usuarios finales del aplicativo corresponden al personal del departamento de Tecnologías de Información y a los administradores de la infraestructura tecnológica, quienes serán responsables del control del hardware corporativo. El acceso al sistema se realizará mediante autenticación, diferenciando los permisos según el rol del usuario dentro del departamento.

1.4.4 Limitaciones

El proyecto presenta las siguientes limitaciones, las cuales se encuentran fuera del alcance del presente trabajo de titulación:

- No contempla integraciones automatizadas con herramientas de gestión de dispositivos móviles (MDM) ni con sistemas de descubrimiento automático de hardware en la red corporativa.
- No incluye módulos de contabilidad ni cálculos de depreciación de activos, más allá del registro del número de factura y la fecha de compra de cada equipo.
- La solución se limita a la gestión de equipos de cómputo tipo laptop y no abarca otros activos tecnológicos como servidores, equipos de red, dispositivos periféricos o equipamiento audiovisual.
- El despliegue se restringe a los servidores internos de Xkale, por lo que el sistema no estará disponible a través de redes públicas ni contempla mecanismos de acceso externo más allá de la red corporativa.

CAPÍTULO 2: FUNDAMENTACIÓN TEÓRICA

El desarrollo de un sistema de información de calidad requiere fundamentar las decisiones de diseño y construcción sobre un marco conceptual sólido que articule las prácticas organizacionales, los paradigmas arquitectónicos, las tecnologías de implementación y los modelos de gestión del trabajo aplicados durante su ejecución. El presente capítulo expone el sustento teórico del proyecto a partir de seis ejes complementarios: la gestión de activos de tecnologías de información, la arquitectura y los patrones empleados en el desarrollo de aplicaciones web, las tecnologías seleccionadas para la construcción de la capa de backend, las herramientas utilizadas en el frontend, los fundamentos de las bases de datos relacionales y la metodología ágil adoptada para la conducción del trabajo.

2.1 Gestión de Activos de Tecnologías de Información (ITAM)

La gestión de activos de TI, conocida por sus siglas en inglés como ITAM (IT Asset Management), comprende el conjunto de prácticas organizacionales orientadas a inventariar, controlar, mantener y optimizar el ciclo de vida de los recursos tecnológicos de una empresa. La norma ISO/IEC 19770-1:2017 define el ITAM como un sistema integrado de procesos financieros, contractuales y de inventario que permite administrar los activos de software y de hardware desde su adquisición hasta su disposición final, garantizando el cumplimiento de los requerimientos legales, contractuales y operativos de la organización (International Organization for Standardization, 2017).

El control sobre la infraestructura es clave para la productividad y la seguridad de los datos. Pressman (2010) considera la trazabilidad de los recursos tecnológicos un pilar de la administración de TI: cuando se pierde el rastro de dónde está o en qué estado se encuentra un equipo, las consecuencias operativas son inmediatas. En la misma línea, Sommerville (2011) advierte que usar herramientas no especializadas para estos registros fragmenta la información, lo que duplica el trabajo administrativo, dificulta auditar el historial técnico de los equipos y eleva los costos.

Para el presente proyecto, la adopción de los principios de ITAM justifica la centralización del registro de equipos y de incidencias en una única plataforma, así como el diseño de estructuras

de datos que permitan vincular cada activo con su usuario asignado, sus garantías vigentes y su historial completo de servicios técnicos.

2.2 Arquitectura y Patrones de Desarrollo Web

A continuación se describirán las arquitecturas empleada para el desarrollo del sistema web, se utilizó la arquitectura Cliente-Servidor, arquitectura en Tres capas y las APIs REST.

2.2.1 Arquitectura Cliente-Servidor

La arquitectura cliente-servidor es un modelo de diseño distribuido en el cual la carga de trabajo del sistema se reparte entre dos roles complementarios: los clientes, que demandan recursos o servicios, y los servidores, que los proveen. (Tanenbaum & Van Steen, 2017) explican que este paradigma constituye la base estructural de la mayoría de aplicaciones distribuidas contemporáneas, incluyendo los sistemas web, los servicios de correo electrónico y los entornos de bases de datos remotas.

En el contexto de las aplicaciones web, el cliente corresponde habitualmente al navegador del usuario final, el cual envía peticiones a un servidor que procesa la lógica de negocio y devuelve respuestas en forma de páginas, datos estructurados o recursos multimedia. Esta separación de responsabilidades favorece la escalabilidad del sistema, dado que cada componente puede evolucionar de manera independiente y replicarse según las necesidades de carga.

2.2.2 Arquitectura de Tres Capas

Sobre el modelo cliente-servidor se construye habitualmente la arquitectura de tres capas, la cual segmenta la aplicación en una capa de presentación responsable de la interacción con el usuario, una capa de lógica de negocio que aplica las reglas operativas del sistema y una capa de datos encargada de la persistencia. (Pressman, 2010) indica que esta separación facilita el mantenimiento del software, mejora la legibilidad del código y permite que distintos perfiles profesionales colaboren simultáneamente sin interferencias significativas en sus responsabilidades.

En el sistema desarrollado para Xkale, la capa de presentación se construye con React, la capa de lógica de negocio con FlaskAPI y la capa de datos con PostgreSQL, manteniendo una

separación clara que permite evolucionar cada componente sin afectar a los demás y simplifica las tareas de prueba, depuración y despliegue.

2.2.3 APIs REST

El estilo arquitectónico REST (Representational State Transfer) fue formulado por (Fielding, 2000) en su tesis doctoral como un conjunto de restricciones aplicables al diseño de sistemas distribuidos basados en el protocolo HTTP. Una API REST se caracteriza por exponer los recursos del sistema mediante identificadores uniformes, utilizar los métodos del protocolo HTTP (GET, POST, PUT, DELETE) para representar las operaciones sobre dichos recursos y mantener la comunicación libre de estado, lo que significa que cada solicitud contiene toda la información necesaria para ser procesada de manera independiente.

La adopción del estilo REST en el presente proyecto permite que el frontend desarrollado en React consuma los servicios expuestos por FlaskAPI de manera predecible, facilitando además la posible incorporación futura de clientes móviles o integraciones con sistemas externos sin necesidad de modificar la lógica del backend.

2.3 Tecnologías de Backend

Para el desarrollo del Backend se empleó como lenguaje de programación Python por su versatilidad y amplio catálogo de frameworks y librerías existentes, dentro de estas se utilizó el framework FastAPI, mismos que se describen a continuación.

2.3.1 Lenguaje de Programación Python

Python es un lenguaje de programación de alto nivel, interpretado, multiparadigma y de tipado dinámico, ampliamente adoptado en el ámbito empresarial, científico y educativo. Su sintaxis clara y la disponibilidad de un extenso ecosistema de bibliotecas lo convierten en una opción versátil para el desarrollo de aplicaciones web, automatización de procesos, análisis de datos e inteligencia artificial. La curva de aprendizaje moderada y la legibilidad del código son factores determinantes para su elección en proyectos académicos y profesionales, ya que permiten al desarrollador concentrarse en la resolución del problema antes que en la complejidad sintáctica del lenguaje.

2.3.2 Framework FastAPI

FastAPI es un framework moderno de desarrollo web para Python, creado en 2018 por Sebastián Ramírez, que aprovecha las anotaciones de tipos estáticos del lenguaje (type hints) para generar automáticamente la validación de los datos de entrada, la serialización de las respuestas y la documentación interactiva de la API. (Lubanovic, 2024) destaca que FastAPI se construye sobre dos bibliotecas fundamentales: Starlette, que proporciona la base asíncrona del servidor web, y Pydantic, que se encarga de la validación y el modelado de los datos que circulan por la aplicación. La combinación de ambas tecnologías sitúa a FastAPI entre los frameworks web de Python con mejor rendimiento en pruebas comparativas independientes.

Entre las características más relevantes de FastAPI que justificaron su elección para el presente proyecto se encuentran las siguientes: el sistema de inyección de dependencias declarativo, que permite componer la lógica de acceso a la base de datos, la autenticación y la autorización por roles de manera modular y reutilizable; la generación automática de documentación OpenAPI mediante las interfaces interactivas Swagger UI y ReDoc, lo cual facilita las pruebas manuales durante el desarrollo y simplifica la entrega de documentación técnica del proyecto; el soporte nativo para el estándar OAuth2 con flujos de autenticación basados en tokens; y el manejo asíncrono de peticiones que mejora el rendimiento bajo carga concurrente.

La organización modular de aplicaciones FastAPI se realiza mediante el componente APIRouter, el cual permite agrupar los endpoints relacionados en módulos independientes y registrarlos en la aplicación principal con prefijos de ruta y etiquetas comunes. Esta estructura resulta análoga al patrón de blueprints presente en otros frameworks web de Python y favorece la mantenibilidad del código en proyectos de tamaño medio como el desarrollado en el presente trabajo.

2.4 Tecnologías de Frontend

Para el desarrollo del frontend se utilizó HTML para las interfaces, TailwindCSS para aplicar una capa de diseño amigable y scripts en JavaScript para agregar pequeñas funciones dentro de las interfaces, a su vez se utilizó algunas bibliotecas como React para complementar los diseños, estas tecnologías serán descritas a continuación.

2.4.1 Fundamentos: HTML, CSS y JavaScript

El desarrollo de la interfaz de usuario en aplicaciones web se apoya en tres tecnologías base. HTML (HyperText Markup Language) define la estructura semántica del contenido; CSS (Cascading Style Sheets) determina la presentación visual a través de reglas de estilo; y JavaScript aporta el comportamiento dinámico, permitiendo que las páginas respondan a las acciones del usuario sin necesidad de recargar el contenido completo. Estas tres tecnologías constituyen la base sobre la cual operan las bibliotecas y frameworks modernos del ecosistema frontend.

2.4.2 Biblioteca React

React es una biblioteca de JavaScript creada por Facebook, orientada a la construcción de interfaces de usuario interactivas mediante la composición de componentes reutilizables. (Banks & Porcello, 2020) explican que la propuesta de React consiste en describir las interfaces de manera declarativa, especificando el aspecto que deben tener los elementos en cada estado del sistema, en lugar de manipular el árbol de elementos del documento (DOM) de forma imperativa.

El núcleo del modelo de programación de React se basa en el concepto de componentes, los cuales son funciones o clases que reciben datos a través de propiedades (props) y mantienen un estado interno (state) que determina su renderizado. La biblioteca emplea un mecanismo conocido como DOM virtual que compara los estados sucesivos de la interfaz y aplica al navegador únicamente los cambios estrictamente necesarios, lo que se traduce en un rendimiento superior frente a las manipulaciones directas del DOM.

A partir de su versión 16.8, React incorporó los hooks, funciones especiales que permiten gestionar el estado y el ciclo de vida de los componentes funcionales sin recurrir a la sintaxis de clases. Hooks como useState, useEffect y useContext han modificado significativamente la manera de estructurar las aplicaciones, fomentando un estilo de programación más conciso, modular y fácil de mantener.

2.4.3 Sistema de Estilos con Tailwind CSS

El sistema de estilos del frontend se construyó sobre Tailwind CSS, una biblioteca de utilidades CSS que adopta un enfoque distinto al de los frameworks tradicionales como Bootstrap. En lugar de proporcionar componentes preconstruidos, Tailwind ofrece un conjunto extenso de

clases atómicas que aplican una única propiedad CSS cada una, permitiendo al desarrollador componer interfaces directamente desde el marcado JSX sin necesidad de escribir hojas de estilo separadas (Tailwind Labs, 2026; Nielsen, 1994).

Este enfoque presenta varias ventajas para proyectos como el desarrollado. La primera es la consistencia visual, dado que todas las medidas, colores y espaciados se definen en un archivo de configuración centralizado que actúa como sistema de diseño único para toda la aplicación. La segunda es el rendimiento del CSS final, ya que el proceso de compilación elimina automáticamente las clases no utilizadas, generando hojas de estilo significativamente más pequeñas que las producidas por frameworks tradicionales. La tercera es la facilidad de personalización, dado que la paleta cromática corporativa de Xkale se incorporó directamente en el archivo de configuración de Tailwind, lo cual permite emplear clases semánticas como bg-primary, text-danger o border-border a lo largo de todos los componentes del sistema, manteniendo una identidad visual coherente sin recurrir a valores hexadecimales dispersos en el código.

2.5 Bases de Datos Relacionales

Para garantizar la integridad de los datos dentro del sistema es de vital importancia emplear una base de datos relacional, para ello se debe crear un modelo relacional que se montará en un Sistema Gestor de Base de Datos (SGBD) que emplea un motor en específico, para este caso se eligió PostgreSQL, a continuación de profundiza en esta información.

2.5.1 Modelo Relacional

El modelo relacional fue propuesto originalmente por (Codd, 1970) en un artículo seminal publicado en la revista Communications of the ACM, en el cual se planteaba representar la información mediante relaciones (tablas) compuestas por tuplas (filas) y atributos (columnas). Este modelo introdujo conceptos como las claves primarias, las claves foráneas y la integridad referencial, los cuales se han mantenido vigentes como pilares del diseño de bases de datos durante más de cinco décadas.

(Date, 2003) señala que la principal fortaleza del modelo relacional radica en su sólida base matemática derivada de la teoría de conjuntos y del álgebra relacional, lo que permite formular consultas declarativas mediante el lenguaje SQL sin necesidad de especificar los procedimientos

exactos para obtener los resultados. (Elmasri & Navathe, 2016) complementan esta visión señalando que el cumplimiento de las formas normales reduce la redundancia, evita anomalías de actualización y favorece la consistencia general de los datos almacenados.

2.5.2 PostgreSQL

PostgreSQL es un sistema de gestión de bases de datos relacional de código abierto, conocido por seguir de cerca el estándar SQL y por manejar grandes volúmenes de datos manteniendo la consistencia. Entre sus características están el cumplimiento de las propiedades ACID (atomicidad, consistencia, aislamiento y durabilidad), el control de concurrencia multiversión (MVCC) para aislar transacciones, la posibilidad de extender su funcionalidad y la creación de tipos de datos propios (PostgreSQL Global Development Group , 2024)

En este proyecto, PostgreSQL es la capa donde se almacenan los datos y la que mantiene la integridad referencial entre las entidades principales: usuarios, computadoras, técnicos y tickets de soporte. Su manejo de transacciones ayuda a que el modelo se mantenga estable aun con varios accesos simultáneos o ante una caída del servidor. Además, al ser de licencia abierta no implica costos de licenciamiento, lo que encaja bien con el despliegue on-premise.

2.6 Metodología Kanban

Kanban es un método de gestión del trabajo inspirado en el pensamiento Lean y en el sistema de producción de Toyota. Anderson (2010) lo formalizó como un modelo de mejora continua para el desarrollo de software, basado en cuatro principios: visualizar el flujo de trabajo, limitar el trabajo en curso (WIP, por sus siglas en inglés), gestionar de forma activa el avance de las tareas y mejorar el proceso de manera incremental a partir de la medición del rendimiento.

A diferencia de Scrum, que define roles, ceremonias y bloques de tiempo fijos (sprints), Kanban es más flexible y no impone una estructura rígida, por lo que se adapta a la forma de trabajo existente sin grandes cambios. Su herramienta central es el tablero, donde las columnas representan los estados por los que pasa cada tarea, desde que se registra hasta que se cierra. Cada tarjeta es una tarea, y verlas moverse por el tablero permite detectar cuellos de botella y seguir el avance real del proyecto.

Para un proyecto llevado por una sola persona, la variante Personal Kanban es especialmente útil, porque ayuda a controlar la carga de trabajo y a evitar hacer muchas cosas a la vez. Para la ejecución del presente trabajo de titulación, se emplea un tablero con tres columnas principales: Por hacer, En progreso y Terminado sobre el cual se gestionan las tareas correspondientes al diseño de la base de datos, la construcción del backend, el desarrollo del frontend y las actividades de despliegue y pruebas.

CAPÍTULO 3: ANÁLISIS Y DISEÑO DEL SISTEMA

El presente capítulo expone el proceso de análisis y diseño que sustenta la construcción del sistema web de gestión de inventario y servicio técnico para Xkale Cía. Ltda. La estructura del capítulo se organiza en cuatro secciones complementarias: el levantamiento y especificación de los requerimientos funcionales y no funcionales, la identificación y descripción de los casos de uso del sistema, el diseño de la base de datos relacional con sus modelos conceptual, lógico y físico, y finalmente el diseño de la interfaz de usuario que materializa la experiencia de los usuarios finales.

(Pressman, 2010) sostiene que el análisis y el diseño constituyen las dos primeras actividades técnicas del proceso de ingeniería de software, sobre las cuales se asienta la calidad final del producto entregado. La rigurosidad con la que se documentan los requerimientos y se modelan las estructuras de datos determina, en gran medida, la facilidad con la que el sistema podrá evolucionar, mantenerse y adaptarse a nuevas necesidades organizacionales.

3.1 Análisis de Requerimientos

El proceso de levantamiento de requerimientos se desarrolló mediante la observación directa de las prácticas actuales del departamento de Tecnologías de Información de Xkale, el análisis de las hojas de cálculo y archivos compartidos que se empleaban para el control de equipos, y la identificación de las funcionalidades mínimas necesarias para sustituir esos mecanismos manuales por una plataforma unificada. (Sommerville, 2011) clasifica los requerimientos de software en dos grandes categorías: los requerimientos funcionales, que describen las funciones que el sistema debe realizar, y los requerimientos no funcionales, que especifican las restricciones, atributos de calidad y propiedades emergentes que el sistema debe cumplir.

3.1.1 Requerimientos Funcionales

Los requerimientos funcionales identificados para el sistema se presentan en la Tabla 1, donde cada uno se identifica con un código único, una descripción concisa y la prioridad asignada durante el análisis.

Tabla 1: Requerimientos funcionales del sistema

Código	Descripción	Prioridad
RF-01	El sistema debe permitir la autenticación de usuarios mediante nombre de usuario y contraseña.	Alta
RF-02	El sistema debe diferenciar el acceso según el rol asignado: Administrador, Técnico o Inventario.	Alta
RF-03	El sistema debe permitir el registro, consulta, edición y eliminación de colaboradores en el directorio de usuarios.	Alta
RF-04	El sistema debe permitir el registro, consulta, edición y eliminación de técnicos responsables del soporte.	Alta
RF-05	El sistema debe permitir el registro de equipos de cómputo con sus características técnicas: marca, modelo, CPU, GPU, RAM, disco, pantalla, número de serie, número de factura, fecha de compra y hostname.	Alta
RF-06	El sistema debe permitir la asignación de cada equipo a un colaborador del directorio de usuarios.	Alta
RF-07	El sistema debe permitir el registro de tickets de servicio técnico vinculados a un equipo y a un técnico.	Alta
RF-08	Cada ticket debe registrar la incidencia, una descripción detallada del problema, el estado actual, la fecha de inicio, la fecha de cierre y la solución aplicada.	Alta
RF-09	El sistema debe permitir la búsqueda de equipos por hostname, serial, marca o modelo.	Media
RF-10	El sistema debe permitir la búsqueda de usuarios por nombre, apellido o correo.	Media
RF-11	El sistema debe permitir filtrar los tickets por estado (Abierto, En Proceso, Cerrado).	Media
RF-12	El sistema debe presentar un panel principal con el resumen general del inventario y de los tickets registrados.	Media
RF-13	El sistema debe permitir al administrador gestionar las credenciales de acceso a la plataforma.	Alta

RF-14	El sistema debe permitir activar o desactivar las cuentas de acceso al sistema.	Media
RF-15	El sistema debe almacenar las contraseñas de acceso de forma segura mediante funciones de hash criptográfico.	Alta

Nota. Elaboración propia a partir del análisis de requerimientos realizado para el sistema de gestión de inventario de Xkale Cía. Ltda.

3.1.2 Requerimientos No Funcionales

Los requerimientos no funcionales especifican las características de calidad que el sistema debe satisfacer más allá de la funcionalidad pura. Estos atributos resultan determinantes para la aceptación final del producto por parte de los usuarios y para su sostenibilidad operativa en el largo plazo. La Tabla 2 resume los requerimientos no funcionales aplicables al proyecto.

Tabla 2: Requerimientos no funcionales del sistema

Código	Categoría	Descripción
RNF-01	Seguridad	Las contraseñas de acceso deben almacenarse cifradas mediante algoritmos de hash criptográfico estándar.
RNF-02	Seguridad	El acceso a las funciones del sistema debe restringirse en función del rol asignado al usuario autenticado.
RNF-03	Usabilidad	La interfaz debe presentar una estructura de navegación consistente entre todos los módulos del sistema.
RNF-04	Usabilidad	Los formularios de registro deben validar los datos ingresados y proporcionar retroalimentación inmediata al usuario.
RNF-05	Rendimiento	Las operaciones de consulta sobre las tablas principales no deben superar los dos segundos de tiempo de respuesta bajo carga normal.
RNF-06	Disponibilidad	El sistema debe estar disponible durante la jornada laboral de la organización en horarios de oficina.
RNF-07	Mantenibilidad	El código fuente debe organizarse de forma modular para facilitar la incorporación de nuevas funcionalidades.
RNF-08	Compatibilidad	El sistema debe funcionar correctamente en las versiones actuales de los navegadores Google Chrome, Mozilla Firefox y Microsoft

		Edge.
RNF-09	Integridad	La base de datos debe garantizar la integridad referencial entre las entidades principales mediante restricciones de clave foránea.
RNF-10	Despliegue	El sistema debe poder instalarse y operar en la infraestructura de servidores internos de la organización sin requerir conexión a servicios externos.

Nota. Elaboración propia.

3.2 Casos de Uso del Sistema

Los casos de uso constituyen una técnica de modelado ampliamente adoptada en la ingeniería de software para describir las interacciones entre los actores externos y las funcionalidades del sistema. (Pressman, 2010) define el caso de uso como la narración estructurada de un escenario en el que un actor utiliza el sistema para alcanzar un objetivo específico. Esta sección identifica los actores involucrados, presenta el conjunto de casos de uso definidos para el sistema y desarrolla en detalle los más relevantes para la operación cotidiana.

3.2.1 Identificación de Actores

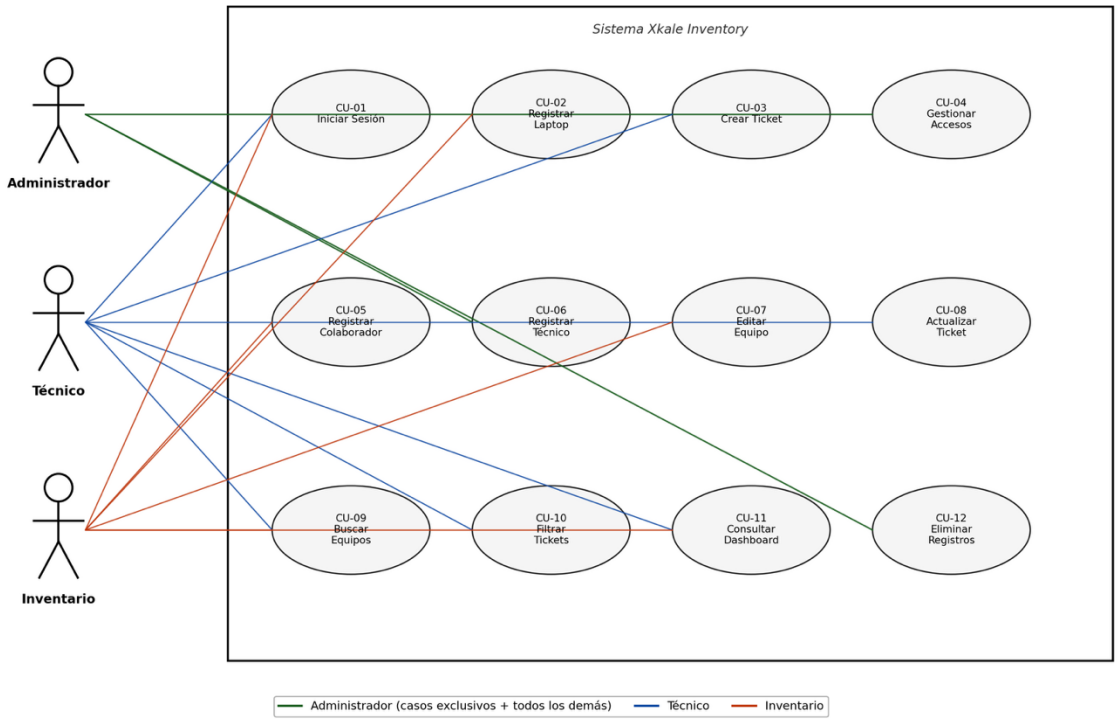
El sistema contempla tres actores principales, los cuales se diferencian por el rol que ocupan dentro del departamento de Tecnologías de Información y por las acciones que pueden ejecutar sobre la plataforma:

- Administrador. Tiene acceso completo a todas las funcionalidades del sistema, incluyendo la gestión de credenciales y la administración de los demás usuarios del sistema. Es el único actor con permisos sobre el módulo de gestión de accesos.
- Técnico. Es el responsable del soporte técnico de los equipos. Tiene permisos para gestionar tickets de servicio, registrar incidencias y documentar las soluciones aplicadas. Adicionalmente, puede consultar el inventario de equipos y los colaboradores a quienes están asignados.
- Inventario. Es el responsable del control del parque informático. Tiene permisos sobre los módulos de laptops, usuarios y técnicos para registrar nuevos equipos, asignarlos a colaboradores y mantener actualizada la información del catálogo.

3.2.2 Diagrama General de Casos de Uso

El diagrama general de casos de uso, presentado en la Figura 1, sintetiza las relaciones entre los tres actores y las principales funcionalidades del sistema. Los casos de uso se agrupan en cinco módulos funcionales que se corresponden con las opciones del menú lateral de la aplicación: Dashboard, Laptops, Tickets, Usuarios, Técnicos y Accesos.

Figura 1: Diagrama General de Casos de Uso



Nota. El actor Administrador hereda los permisos de Técnicos e Inventario, las líneas verdes muestran únicamente sus casos exclusivos para evitar saturación visual.

3.2.3 Especificación Detallada de Casos de Uso

A continuación, se presenta la especificación detallada de los casos de uso más representativos del sistema. Para cada caso se documentan el actor principal, las precondiciones, el flujo básico de eventos, los flujos alternativos relevantes y la postcondición esperada.

Tabla 3: Caso de Uso CU-01: Iniciar Sesión

Campo	Descripción
Identificador	CU-01
Nombre	Iniciar Sesión
Actor principal	Administrador, Técnico, Inventario
Precondiciones	El actor cuenta con credenciales activas registradas previamente en el sistema.
Flujo básico	1. El actor accede a la URL del sistema. 2. El sistema presenta el formulario de inicio de sesión. 3. El actor ingresa su nombre de usuario y contraseña. 4. El actor presiona el botón Ingresar. 5. El sistema valida las credenciales contra la base de datos. 6. El sistema redirige al actor al panel principal correspondiente a su rol.
Flujo alternativo	5a. Si las credenciales son incorrectas, el sistema muestra un mensaje de error y retorna al paso 2.
Postcondición	El actor accede al sistema con los permisos correspondientes a su rol.

Nota. Elaboración propia.

Tabla 4: Caso de Uso CU-02: Registrar Nueva Laptop

Campo	Descripción
Identificador	CU-02
Nombre	Registrar Nueva Laptop
Actor principal	Inventario, Administrador
Precondiciones	El actor está autenticado en el sistema con un rol que le permite gestionar el inventario.
Flujo básico	1. El actor selecciona la opción Laptops del menú lateral. 2. El sistema muestra el listado de equipos registrados. 3. El actor presiona el botón Nueva Laptop. 4. El sistema presenta el formulario de registro. 5. El actor ingresa los datos del equipo: hostname, serial, marca, modelo, CPU, GPU, RAM, disco, pantalla, número de factura y fecha de compra. 6. El actor selecciona el colaborador al que se asigna el equipo desde la lista desplegable. 7. El actor presiona el botón Crear

	laptop. 8. El sistema valida los datos y persiste el registro en la base de datos. 9. El sistema actualiza el listado y muestra el nuevo equipo.
Flujo alternativo	8a. Si algún dato obligatorio está ausente o es inválido, el sistema muestra mensajes de validación y retorna al paso 5. 6a. El actor puede dejar el campo de usuario asignado con la opción Sin asignar.
Postcondición	Un nuevo registro de equipo queda almacenado en el inventario.

Nota. Elaboración propia.

Tabla 5: Caso de Uso CU-03: Crear Ticket de Servicio Técnico

Campo	**Descripción**
Identificador	CU-03
Nombre	Crear Ticket de Servicio Técnico
Actor principal	Técnico, Administrador
Precondiciones	El actor está autenticado en el sistema. Existe al menos un equipo registrado y al menos un técnico disponible.
Flujo básico	1. El actor selecciona la opción Tickets del menú lateral. 2. El sistema muestra el listado de tickets existentes. 3. El actor presiona el botón Nuevo Ticket. 4. El sistema presenta el formulario de creación. 5. El actor selecciona el equipo afectado y el técnico responsable desde los menús desplegables. 6. El actor ingresa el título de la incidencia y una descripción detallada del problema. 7. El actor define el estado inicial del ticket. 8. El actor presiona el botón Crear ticket. 9. El sistema valida los datos, persiste el registro y actualiza el listado.
Flujo alternativo	7a. Si el ticket se crea con estado Cerrado, el actor también ingresa la fecha de cierre y la solución aplicada.
Postcondición	Un nuevo ticket queda registrado y vinculado al equipo y al técnico correspondientes.

Nota. Elaboración propia.

Tabla 6: Caso de Uso CU-04: Gestionar Accesos al Sistema

Campo	Descripción
Identificador	CU-04
Nombre	Gestionar Accesos al Sistema
Actor principal	Administrador
Precondiciones	El actor está autenticado con rol de Administrador.
Flujo básico	1. El actor selecciona la opción Accesos del menú lateral. 2. El sistema muestra el listado de credenciales registradas con su rol y estado. 3. El actor presiona el botón Nuevo Acceso. 4. El sistema presenta el formulario de creación. 5. El actor ingresa el nombre de usuario, correo electrónico, contraseña y rol. 6. El actor presiona el botón Crear acceso. 7. El sistema valida los datos, cifra la contraseña mediante hash y persiste la credencial.
Flujo alternativo	El actor puede desactivar una credencial existente mediante el botón de activación, lo cual impide el inicio de sesión sin eliminar el registro histórico.
Postcondición	Una nueva credencial queda disponible para iniciar sesión en el sistema.

Nota. Elaboración propia.

Adicionalmente a los casos de uso desarrollados en detalle, el sistema contempla las funcionalidades resumidas en la Tabla 7, las cuales siguen patrones similares de interacción adaptados al módulo correspondiente.

Tabla 7: Casos de uso complementarios

Código	Nombre	Actor
CU-05	Registrar nuevo colaborador en el directorio (Similar al caso de uso CU-02)	Inventario, Administrador
CU-06	Registrar nuevo técnico de soporte (Similar al caso de uso CU-02)	Administrador
CU-07	Editar información de equipo registrado (Similar al caso	Inventario,

	de uso CU-02)	Administrador
CU-08	Actualizar estado y solución de ticket (Similar al caso de uso CU-03)	Técnico, Administrador
CU-09	Buscar equipos por hostname, serial, marca o modelo	Todos los actores
CU-10	Filtrar tickets por estado	Técnico, Administrador
CU-11	Consultar panel principal con resumen del sistema	Todos los actores
CU-12	Eliminar registros de equipos, usuarios o técnicos	Administrador

Nota. Elaboración propia.

3.3 Diseño de la Base de Datos

El diseño de la base de datos se desarrolló siguiendo los tres niveles propuestos por (Elmasri & Navathe, 2016): el modelo conceptual, el modelo lógico y el modelo físico. Esta separación permite abordar progresivamente la complejidad del problema, partiendo de una representación abstracta de las entidades del dominio hasta llegar a la implementación concreta sobre el sistema de gestión de bases de datos seleccionado.

3.3.1 Modelo Conceptual

El modelo conceptual identifica las entidades fundamentales del dominio del problema y las relaciones que las vinculan, sin entrar en detalles de implementación. Para el sistema de gestión de inventario y servicio técnico de Xkale se identificaron cinco entidades principales: Usuario, Tecnico, Laptop, Ticket y Auth_User.

La entidad Usuario representa a los colaboradores de la organización a quienes se asignan los equipos de cómputo. La entidad Tecnico representa al personal del departamento de Tecnologías de Información encargado de atender las incidencias de soporte. La entidad Laptop representa los equipos de cómputo que conforman el parque informático de la compañía. La entidad Ticket representa cada incidencia registrada sobre un equipo, incluyendo la solicitud, el diagnóstico y la solución aplicada. Finalmente, la entidad Auth_User representa las credenciales que permiten a los integrantes del departamento acceder al sistema.

Las relaciones identificadas entre estas entidades son las siguientes: un Usuario puede tener

asignadas una o varias Laptops, y cada Laptop pertenece a un único Usuario en un momento dado; una Laptop puede estar asociada a múltiples Tickets a lo largo de su ciclo de vida, mientras que cada Ticket está vinculado a una única Laptop; y un Tecnico puede ser responsable de múltiples Tickets, mientras que cada Ticket es atendido por un único Tecnico. La entidad Auth_User es independiente de las anteriores y se utiliza exclusivamente para la gestión del acceso al sistema.

3.3.2 Modelo Lógico

El modelo lógico transforma el modelo conceptual en una representación relacional, aplicando los principios formulados por (Codd, 1970). Cada entidad se convierte en una relación o tabla, sus atributos en columnas y las relaciones entre entidades se materializan mediante claves foráneas que mantienen la integridad referencial.

La Tabla 8 sintetiza las relaciones del modelo lógico junto con sus claves primarias y foráneas.

Tabla 8: Resumen del modelo lógico relacional

Relación	**Clave primaria**	**Claves foráneas**
auth_user	id_auth	(ninguna)
usuario	id_usuario	(ninguna)
tecnico	id_tecnico	(ninguna)
laptop	id_laptop	usu_id_laptop → usuario(id_usuario)
ticket	id_ticket	tec_id → tecnico(id_tecnico); lap_id → laptop(id_laptop)

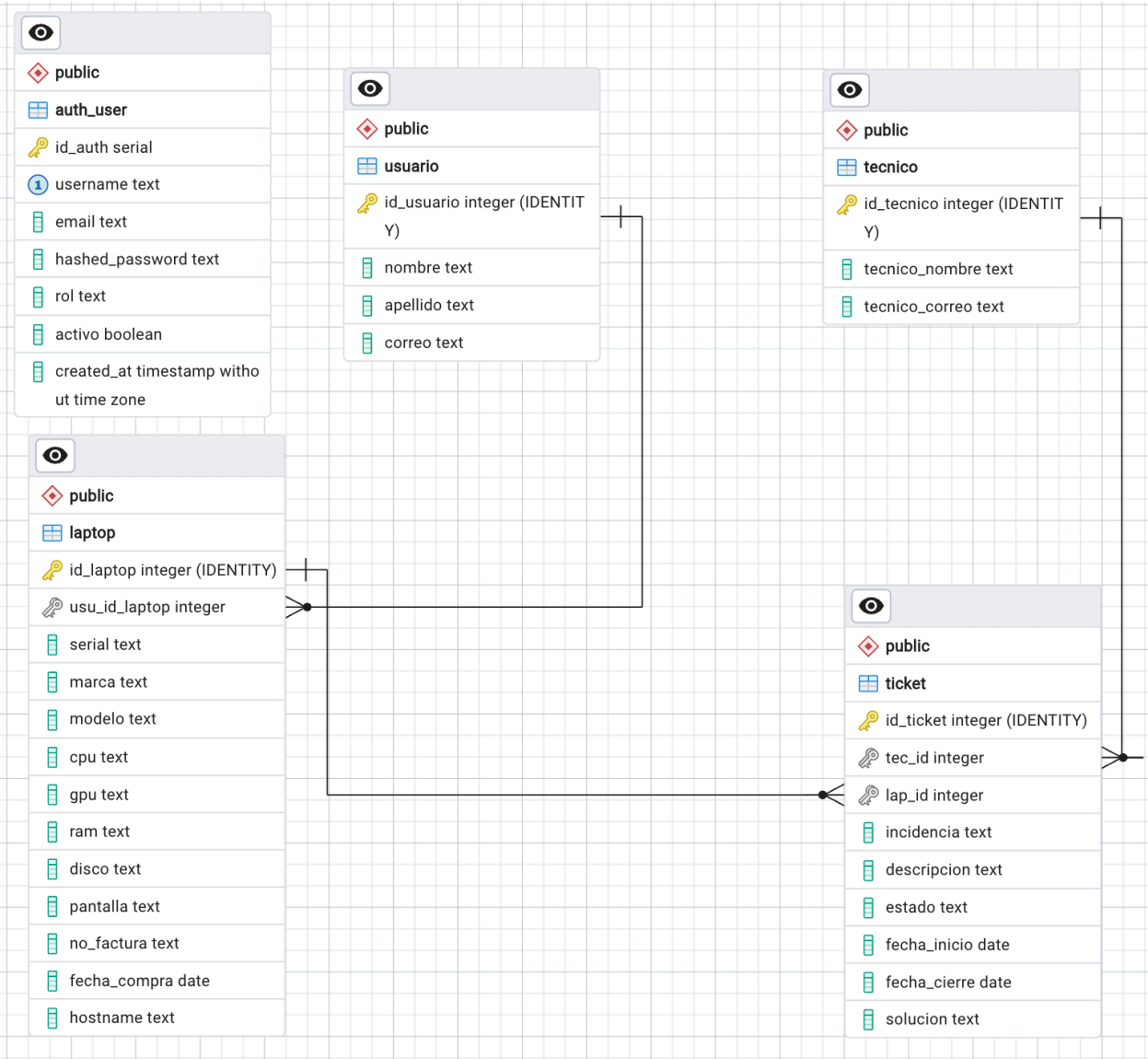
Nota. Elaboración propia.

3.3.3 Modelo Físico

El modelo físico corresponde a la implementación concreta del esquema sobre el sistema de gestión de bases de datos PostgreSQL en su versión 17. La creación de las tablas se realizó mediante sentencias del lenguaje de definición de datos (DDL) que especifican los tipos de datos, las restricciones, las claves primarias y las relaciones referenciales. El diagrama entidad-relación correspondiente al modelo físico se presenta en la Figura 2 y fue generado a partir del esquema

implementado utilizando la herramienta ERD integrada en pgAdmin 4.

Figura 2: Diagrama Entidad-Relación del Modelo Físico



Entre las decisiones de diseño físico relevantes se destacan las siguientes: el uso de identificadores numéricos generados automáticamente mediante la sentencia GENERATED ALWAYS AS IDENTITY, en sustitución de los tipos serial considerados obsoletos en versiones recientes de PostgreSQL; la adopción del tipo TEXT para los campos de longitud variable, lo cual elimina la necesidad de establecer límites arbitrarios de longitud; el uso del tipo DATE para los campos de fecha; y la definición de las restricciones de clave foránea con la cláusula ON UPDATE

RESTRICT y ON DELETE RESTRICT, lo que impide la eliminación de registros que mantienen dependencias referenciales activas, evitando pérdida accidental de información histórica.

3.3.4 Diccionario de Datos

El diccionario de datos documenta el detalle de cada atributo de las tablas del sistema. Las tablas 9 a 13 presentan la especificación completa para cada entidad.

Tabla 9: Diccionario de datos de la tabla auth_user

Atributo	Tipo	Restricción	Descripción
id_auth	serial	PK, NOT NULL	Identificador único de la credencial.
username	text	UNIQUE, NOT NULL	Nombre de usuario para inicio de sesión.
email	text	NULL	Correo electrónico asociado a la cuenta.
hashed_password	text	NOT NULL	Contraseña cifrada mediante hash criptográfico.
rol	text	NOT NULL, DEFAULT 'inventario'	Rol asignado: admin, tecnico o inventario.
activo	boolean	NOT NULL, DEFAULT true	Estado de la credencial.
created_at	timestamp	DEFAULT CURRENT_TIMESTAMP	Fecha de creación del registro.

Nota. Elaboración propia a partir del script DDL del proyecto.

Tabla 10: Diccionario de datos de la tabla usuario

Atributo	Tipo	Restricción	Descripción
id_usuario	integer	PK, IDENTITY	Identificador único del colaborador.
nombre	text	NULL	Nombre del colaborador.

apellido	text	NULL	Apellido del colaborador.
correo	text	NULL	Correo corporativo del colaborador.

Nota. Elaboración propia.

Tabla 11: Diccionario de datos de la tabla técnico

Atributo	**Tipo**	**Restricción**	**Descripción**
id_tecnico	integer	PK, IDENTITY	Identificador único del técnico.
tecnico_nombre	text	NULL	Nombre completo del técnico de soporte.
tecnico_correo	text	NULL	Correo corporativo del técnico.

Nota. Elaboración propia.

Tabla 12: Diccionario de datos de la tabla laptop

Atributo	**Tipo**	**Restricción**	**Descripción**
id_laptop	integer	PK, IDENTITY	Identificador único del equipo.
usu_id_laptop	integer	FK → usuario, NOT NULL	Usuario al que se asigna el equipo.
serial	text	NULL	Número de serie del fabricante.
marca	text	NULL	Marca del equipo.
modelo	text	NULL	Modelo del equipo.
cpu	text	NULL	Procesador del equipo.
gpu	text	NULL	Unidad de procesamiento gráfico.
ram	text	NULL	Memoria RAM instalada.
disco	text	NULL	Capacidad de almacenamiento.
pantalla	text	NULL	Características de la pantalla.
no_factura	text	NULL	Número de factura de adquisición.
fecha_compra	date	NULL	Fecha en que se adquirió el equipo.
hostname	text	NULL	Nombre del equipo en la red.

Nota. Elaboración propia.

Tabla 13: Diccionario de datos de la tabla ticket

Atributo	Tipo	Restricción	Descripción
id_ticket	integer	PK, IDENTITY	Identificador único del ticket.
tec_id	integer	FK → tecnico, NOT NULL	Técnico responsable de la atención.
lap_id	integer	FK → laptop, NOT NULL	Equipo afectado por la incidencia.
incidencia	text	NULL	Título o descripción breve de la incidencia.
descripcion	text	NULL	Descripción detallada del problema reportado.
estado	text	NULL	Estado del ticket: abierto, en proceso, cerrado.
fecha_inicio	date	NULL	Fecha de apertura del ticket.
fecha_cierre	date	NULL	Fecha de cierre del ticket.
solucion	text	NULL	Solución aplicada al problema reportado.

Nota. Elaboración propia.

3.3.5 Reglas de Integridad

La base de datos implementa diversas reglas de integridad para garantizar la consistencia de la información almacenada. (Date, 2003) destaca que la integridad de los datos constituye uno de los pilares del modelo relacional y debe sostenerse en el nivel del esquema antes que en la lógica de la aplicación. Para el sistema implementado se aplican las siguientes reglas:

- Integridad de entidad: cada relación posee una clave primaria que identifica de manera única a sus tuplas. PostgreSQL garantiza la unicidad mediante la generación automática de identificadores con la sentencia GENERATED ALWAYS AS IDENTITY.
- Integridad referencial: las claves foráneas usu_id_laptop, tec_id y lap_id obligan a que cada registro apunte a una fila existente en su tabla padre, de modo que no quedan registros huérfanos. Las cláusulas ON UPDATE RESTRICT y ON DELETE

RESTRICT protegen los registros referenciados frente a actualizaciones o eliminaciones que comprometerían la integridad del esquema.

- Integridad de dominio: el atributo username de la tabla auth_user está sujeto a una restricción de unicidad que evita el registro de credenciales duplicadas. Los atributos NOT NULL impiden el registro de tuplas con valores ausentes en columnas críticas como las claves foráneas o las contraseñas cifradas.
- Optimización de consultas: se definieron índices secundarios sobre las columnas que participan en las relaciones foráneas (idx_laptop_usuario, idx_ticket_laptop, idx_ticket_tecnico) para acelerar las búsquedas y los recorridos de las relaciones entre tablas.

3.4 Diseño de la Interfaz de Usuario

El diseño de la interfaz de usuario constituye el componente del sistema con el cual los actores interactúan de manera directa, por lo que su calidad determina en gran medida la aceptación de la plataforma por parte del departamento de Tecnologías de Información. La presente sección expone los principios de diseño aplicados, la identidad visual adoptada, la arquitectura de navegación y la descripción detallada de las pantallas principales del sistema.

3.4.1 Principios de Diseño Aplicados

(Nielsen, 1994) propuso un conjunto de heurísticas de usabilidad que se han consolidado como referencia ampliamente adoptada para la evaluación y construcción de interfaces digitales. Entre estas heurísticas, las siguientes orientaron las decisiones de diseño del sistema:

- Visibilidad del estado del sistema: los listados de equipos, tickets, usuarios y técnicos muestran información de cabecera con el conteo total de registros, manteniendo al actor informado en todo momento del volumen de datos con el que está trabajando.
- Consistencia y estándares: todos los módulos del sistema siguen la misma estructura visual, con un menú lateral fijo a la izquierda, un encabezado con el nombre del módulo y un área de contenido principal con tabla, búsqueda y botón de acción primaria ubicado en la esquina superior derecha.

- Reconocimiento antes que recuerde: los formularios de registro incluyen ejemplos de placeholder en cada campo para guiar al actor sobre el tipo de información esperada sin requerirle memorizar el formato correcto.
- Flexibilidad y eficiencia: las opciones de búsqueda y filtrado permiten al actor experimentado acceder rápidamente a los registros de interés, mientras que la navegación por menú facilita la operación del actor novato.
- Diseño estético y minimalista: la interfaz emplea una paleta cromática restringida y elimina elementos visuales innecesarios para enfocar la atención del actor en los datos relevantes.

3.4.2 Identidad Visual y Paleta Cromática

La identidad visual del sistema se construyó a partir del logotipo corporativo de Xkale, adoptando una paleta cromática derivada del color verde lima característico de la marca. El esquema cromático aplicado es el siguiente: el color de fondo principal es negro absoluto, las superficies elevadas como tarjetas y modales utilizan tonos de gris muy oscuro, el color de acento corporativo es una verde lima brillante empleado en elementos de acción primaria y botones de confirmación, y la tipografía se presenta en blanco sobre el fondo oscuro con jerarquía visual establecida mediante variaciones de peso y tamaño. La Tabla 14 resume los valores hexadecimales utilizados.

Tabla 14: Paleta cromática del sistema

Rol cromático	**Valor hex**	**Uso principal**
Fondo principal	#000000	Color base de la aplicación.
Superficie elevada	#111111 a #1A1A1A	Tarjetas, modales y barra lateral.
Acento corporativo	#6AE000	Botones de acción primaria, indicadores activos, títulos de sección.
Texto principal	#FFFFFF	Contenido de cabeceras y datos.
Texto secundario	Tonalidades de gris	Etiquetas auxiliares y descripciones.

Indicadores semánticos	Rojo, ámbar, azul	Estados de tickets y métricas del panel principal.

Nota. Elaboración propia.

3.4.3 Arquitectura de Navegación

La arquitectura de navegación se organiza en torno a una barra lateral fija ubicada en el costado izquierdo de la pantalla. Esta barra contiene el logotipo de la organización, el conjunto de enlaces a los seis módulos principales del sistema (Dashboard, Laptops, Tickets, Usuarios, Técnicos y Accesos), información del actor autenticado en su parte inferior y la opción de cerrar sesión. El módulo activo se resalta visualmente con el color de acento corporativo, manteniendo en todo momento la orientación del actor sobre su posición dentro del sistema.

El área principal de contenido, ubicada a la derecha de la barra lateral, presenta el módulo seleccionado. Cada módulo sigue un patrón consistente: un encabezado con la categoría del módulo en color de acento, el título principal en blanco y una descripción breve; un bloque de controles con barra de búsqueda, filtros cuando corresponda, botón de actualización y botón de acción primaria para crear nuevos registros; y un área de visualización principal donde se despliegan los datos en formato de tabla.

3.4.4 Descripción de Pantallas Principales

A continuación, se describen las pantallas principales del sistema. Por motivos de espacio, las capturas de pantalla correspondientes se incluyen como anexos al presente documento.

La pantalla de inicio de sesión es el primer punto de contacto del actor con el sistema. Presenta el logotipo de Xkale acompañado del subtítulo Inventory System y un formulario centrado con los campos Usuario y Contraseña, junto al botón Ingresar en color de acento corporativo. La pantalla mantiene la paleta de la aplicación e incluye un pie de página con la atribución del sistema.

El panel principal o Dashboard ofrece al actor una visión consolidada del estado del sistema. Contiene cuatro tarjetas de indicadores con los conteos totales de laptops, usuarios, técnicos y tickets registrados, seguidas por tres tarjetas que detallan el número de tickets abiertos, en proceso y cerrados. La parte inferior del panel muestra un listado de tickets recientes con el

título de la incidencia, la fecha de apertura, el equipo asociado y el estado actual.

El módulo de Laptops presenta el listado completo del parque informático en una tabla con las columnas hostname, serial, marca y modelo, CPU, RAM y disco, usuario asignado y fecha de compra. La cabecera incluye una barra de búsqueda por hostname, serial, marca o modelo, un botón de filtros avanzados y el botón Nueva Laptop. La acción de crear o editar un equipo abre un modal que solicita todos los atributos técnicos del activo, incluyendo el campo desplegable para asignar el equipo a un colaborador del directorio.

El módulo de Tickets organiza la información en una tabla con las columnas número, incidencia, laptop, técnico, estado, fecha de inicio y fecha de cierre. Sobre la tabla se presentan cuatro pestañas de filtrado por estado (Todos, Abiertos, En Proceso, Cerrados) con el conteo correspondiente en cada categoría. El modal de creación o edición de tickets vincula el equipo afectado con el técnico responsable e incluye los campos de incidencia, descripción detallada, estado, fechas y solución aplicada.

Los módulos de Usuarios y de Técnicos funcionan de forma parecida: cada uno muestra una tabla con el identificador, el nombre completo, el correo electrónico y las acciones de editar y eliminar. Sus formularios piden solo los datos personales necesarios para operar el sistema.

El módulo de Accesos, disponible solo para el rol de Administrador, lista las credenciales registradas en la plataforma con su nombre de usuario, correo electrónico, rol asignado, estado y fecha de creación.

Desde este módulo se pueden crear credenciales, restablecer contraseñas y activar o dar de baja cuentas. Al registrar un usuario, un menú desplegable permite asignarle el rol de Administrador, Técnico o Inventario, que define sus permisos dentro de la plataforma. En el Anexo A se incluyen las capturas de pantalla del sistema en producción, tanto de los tableros principales como de las ventanas modales de ingreso de datos.

CAPÍTULO 4: DESARROLLO DE LA APLICACIÓN

Este capítulo describe cómo se construyó el sistema, desde la preparación del entorno de desarrollo hasta el despliegue en los servidores locales de Xkale. Se explican la organización del backend en FastAPI, la construcción de la interfaz con React y Tailwind CSS, y las decisiones de despliegue. Para sustentar las soluciones adoptadas, se incluyen fragmentos del código real del proyecto. Sommerville (2011) señala que documentar de forma ordenada la etapa de desarrollo es clave para mantener el software en el tiempo y transferir el conocimiento técnico a otras personas.

4.1 Configuración del Entorno de Desarrollo

El proyecto se organizó en dos partes separadas, backend y frontend, de modo que cada una se puede desarrollar, probar y desplegar por su cuenta. Esta separación es coherente con el patrón arquitectónico de tres capas descrito en el capítulo de fundamentación teórica y constituye una práctica estándar en el desarrollo moderno de aplicaciones web.

4.1.1 Herramientas y Versiones

La Tabla 15 presenta el conjunto de herramientas, lenguajes y bibliotecas empleadas durante la construcción del sistema, junto con las versiones específicas adoptadas para garantizar la reproducibilidad del entorno.

Tabla 15: Herramientas y versiones empleadas en el desarrollo

Categoría	Herramienta	Versión	Propósito
Lenguaje backend	Python	3.11+	Lenguaje de la capa de servicios.
Framework backend	FastAPI	0.135.3	Framework web para la API REST.
Servidor ASGI	Uvicorn	0.44.0	Servidor de aplicación para FastAPI.
ORM	SQLAlchemy	2.0.49	Mapeo objeto-relacional para PostgreSQL.

Driver de BDD	psycopg2-binary	2.9.11	Conexión nativa a PostgreSQL.
Validación	Pydantic	2.13.0	Validación y serialización de datos.
Migraciones	Alembic	1.18.4	Versionado del esquema de la BDD.
Hash de contraseñas	bcrypt	4.0.1	Cifrado de credenciales.
Tokens JWT	python-jose	3.5.0	Generación y validación de tokens.
Variables de entorno	python-dotenv	1.2.2	Carga de configuración sensible.
Lenguaje frontend	JavaScript (ES2022)	—	Lenguaje del cliente web.
Biblioteca UI	React	19.2.4	Construcción de la interfaz de usuario.
Bundler	Vite	8.0.4	Empaquetado y servidor de desarrollo.
Estilos	Tailwind CSS	3.4.19	Sistema de utilidades CSS.
Enrutamiento	React Router DOM	7.14.1	Navegación entre páginas.
Cliente HTTP	Axios	1.15.0	Consumo de la API REST.
Estado servidor	TanStack Query	5.99.0	Caché y sincronización de datos.
Íconos	Lucide React	1.8.0	Conjunto de íconos vectoriales.
Base de datos	PostgreSQL	17	Sistema de gestión de la persistencia.
Herramienta de modelado	pgAdmin	4	Administración visual de la BDD.
Editor de código	Visual Studio Code	Última	Entorno integrado de desarrollo.

Control de versiones	Git / GitHub	—	Gestión del histórico de cambios.

Nota. Elaboración propia a partir de los archivos requirements.txt y package.json del proyecto.

4.1.2 Estructura del Proyecto

El proyecto se organizó en dos directorios principales, uno para el backend y otro para el frontend, manteniendo una separación física que refuerza la separación lógica entre los componentes. El Código 1 presenta la estructura general de directorios adoptada.

Código 1

Estructura de directorios del proyecto

```
xkale-inventory/
├── backend/
│   ├── app/
│   │   ├── __init__.py
│   │   ├── main.py          # Inicialización de FastAPI
│   │   ├── database.py      # Configuración de SQLAlchemy
│   │   ├── auth.py          # Lógica de JWT y dependencias
│   │   ├── models/          # Modelos SQLAlchemy
│   │   │   ├── auth_user.py
│   │   │   ├── usuario.py
│   │   │   ├── tecnico.py
│   │   │   ├── laptop.py
│   │   │   └── ticket.py
│   │   ├── schemas/         # Esquemas Pydantic
│   │   │   ├── usuario.py
│   │   │   ├── tecnico.py
│   │   │   ├── laptop.py
│   │   │   └── ticket.py
│   │   └── routers/         # Endpoints agrupados
```

```
│   │       ├── auth.py
│   │       ├── usuarios.py
│   │       ├── tecnicos.py
│   │       ├── laptops.py
│   │       ├── tickets.py
│   │       └── consultas.py   # Consultas avanzadas con JOIN
│   ├── requirements.txt
│   └── create_admin.py        # Script de bootstrap del admin
└── frontend/
    ├── public/              # Recursos estáticos (logo, iconos)
    ├── src/
    │   ├── main.jsx         # Punto de entrada de React
    │   ├── App.jsx          # Configuración de rutas
    │   ├── index.css        # Importación de Tailwind
    │   ├── api/             # Cliente axios y servicios
    │   ├── components/      # Componentes reutilizables
    │   │   ├── ui/          # Button, Input, Card, Modal, etc.
    │   │   └── layout/      # Layout, Sidebar
    │   ├── context/         # AuthContext global
    │   ├── pages/           # Páginas por módulo
    │   └── theme/           # Paleta de colores
    ├── tailwind.config.js
    ├── vite.config.js
    └── package.json
```

Nota. Elaboración propia a partir de la estructura real del repositorio

El repositorio se encuentra referenciado en la bibliografía. (Buitrón, 2026)

4.2 Desarrollo del Backend con FastAPI

La capa de backend se construyó sobre el framework FastAPI siguiendo los principios de diseño orientado a APIs RESTful descritos en el capítulo de fundamentación teórica. (Lubanovic, 2024) recomienda organizar las aplicaciones FastAPI mediante el patrón de routers separados por dominio funcional y la inyección de dependencias para gestionar los recursos compartidos como las sesiones de base de datos y las credenciales del usuario autenticado. Ambas prácticas fueron adoptadas en el presente proyecto.

4.2.1 Inicialización de la Aplicación

El punto de entrada del backend se encuentra en el archivo main.py, el cual instancia la aplicación FastAPI, configura el middleware de intercambio de recursos de origen cruzado (CORS) para permitir las peticiones desde el cliente web y registra los routers correspondientes a cada módulo funcional del sistema. El Código 2 presenta el contenido íntegro de este archivo.

Código 2

Inicialización de la aplicación FastAPI (main.py)

```
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from app.routers import (
    usuarios_router, tecnicos_router,
    laptops_router, tickets_router
)
from app.routers.auth import router as auth_router
from app.routers.consultas import router as consultas_router

app = FastAPI(
    title="Xkale Inventory API",
    description="Sistema de gestión de inventario IT - Xkale",
    version="1.0.0"
```

```
)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

app.include_router(auth_router)
app.include_router(usuarios_router)
app.include_router(tecnicos_router)
app.include_router(laptops_router)
app.include_router(tickets_router)
app.include_router(consultas_router)

@app.get("/")
def root():
    return {"status": "ok", "message": "Xkale Inventory API corriendo"}
```

Nota. Elaboración propia.

Los metadatos title, description y version pasados al constructor de FastAPI son aprovechados por el framework para generar automáticamente la página de documentación interactiva accesible en la ruta /docs. Esta característica resultó particularmente valiosa durante el desarrollo, ya que permitió probar manualmente cada endpoint sin necesidad de configurar herramientas externas en las etapas iniciales del proyecto.

4.2.2 Configuración de la Base de Datos

La conexión a la base de datos PostgreSQL se configuró en el archivo database.py mediante el motor de SQLAlchemy 2.0. La cadena de conexión se obtiene desde una variable de entorno cargada con python-dotenv, lo cual permite mantener las credenciales fuera del control de versiones. El Código 3 presenta la implementación adoptada.

Código 3

Configuración de SQLAlchemy y sesión de base de datos (database.py)

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from dotenv import load_dotenv
import os

load_dotenv()

DATABASE_URL = os.getenv("DATABASE_URL")

engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Nota. Elaboración propia.

La función get_db implementa el patrón de generador de Python que produce una sesión de base de datos para cada petición HTTP y garantiza su cierre al finalizar, independientemente de si la petición concluye exitosamente o lanza una excepción. FastAPI invoca automáticamente esta función mediante su sistema de inyección de dependencias, asegurando que cada endpoint reciba una sesión aislada y que los recursos se liberen correctamente.

4.2.3 Modelos de SQLAlchemy

Cada tabla del esquema relacional se representa en el código mediante una clase de Python que hereda de la base declarativa de SQLAlchemy. Estas clases definen las columnas, los tipos de datos y las relaciones entre entidades, permitiendo manipular los registros como objetos de Python en lugar de escribir consultas SQL manualmente. El Código 4 muestra el modelo correspondiente a la entidad Laptop, en el cual se aprecian la declaración de la clave foránea hacia la tabla usuario y las relaciones bidireccionales con las entidades Usuario y Ticket.

Código 4

Modelo SQLAlchemy de la entidad Laptop (models/laptop.py)

```
from sqlalchemy import Column, Integer, Text, Date, ForeignKey
from sqlalchemy.orm import relationship
from app.database import Base

class Laptop(Base):
    __tablename__ = "laptop"

    id_laptop     = Column(Integer, primary_key=True, index=True)
    usu_id_laptop = Column(Integer, ForeignKey("usuario.id_usuario"))
    serial        = Column(Text)
    marca         = Column(Text)
    modelo        = Column(Text)
    cpu           = Column(Text)
    gpu           = Column(Text)
```

```
    ram          = Column(Text)
    disco        = Column(Text)
    pantalla     = Column(Text)
    no_factura   = Column(Text)
    fecha_compra = Column(Date)
    hostname     = Column(Text)

    usuario = relationship("Usuario")
    tickets = relationship("Ticket", back_populates="laptop")
```

Nota. Elaboración propia.

La directiva relationship establece relaciones a nivel del ORM que se traducen en sentencias SQL JOIN cuando se acceden los atributos correspondientes. El parámetro back_populates declara una relación bidireccional, lo cual permite que desde una instancia de Laptop se acceda al conjunto de sus tickets y, recíprocamente, que desde un Ticket se obtenga la laptop a la que pertenece. Elmasri y Navathe (2016) destacan que esta capacidad de navegación entre entidades constituye una de las ventajas más significativas del uso de mapeadores objeto-relacionales frente a la escritura directa de consultas SQL.

4.2.4 Esquemas de Pydantic

Los esquemas de Pydantic se utilizan en FastAPI para validar los datos que ingresan al sistema a través de las peticiones HTTP, así como para definir la estructura de las respuestas que el servidor devuelve al cliente. Cada entidad del sistema cuenta con un conjunto de tres esquemas: uno base que declara los campos opcionales, uno para la creación de nuevos registros y uno para la salida que incluye el identificador asignado por la base de datos. El Código 5 ilustra estos esquemas para la entidad Laptop.

Código 5

Esquemas Pydantic de la entidad Laptop (schemas/laptop.py)

```
from pydantic import BaseModel
from typing import Optional
from datetime import date

class LaptopBase(BaseModel):
    usu_id_laptop: Optional[int]  = None
    serial:        Optional[str]  = None
    marca:         Optional[str]  = None
    modelo:        Optional[str]  = None
    cpu:           Optional[str]  = None
    gpu:           Optional[str]  = None
    ram:           Optional[str]  = None
    disco:         Optional[str]  = None
    pantalla:      Optional[str]  = None
    no_factura:    Optional[str]  = None
    fecha_compra:  Optional[date] = None
    hostname:      Optional[str]  = None

class LaptopCreate(LaptopBase):
    pass

class LaptopUpdate(LaptopBase):
    pass

class LaptopOut(LaptopBase):
    id_laptop: int
    class Config:
        from_attributes = True
```

Nota. Elaboración propia.

La directiva from_attributes habilitada en la configuración del esquema LaptopOut le indica a Pydantic que debe construir el objeto de respuesta accediendo a los atributos de la instancia recibida, lo cual permite convertir directamente los objetos SQLAlchemy en respuestas JSON sin pasos intermedios. Esta característica simplifica significativamente el código de los routers y reduce la posibilidad de errores en la transformación de datos.

4.2.5 Routers y Endpoints REST

El backend organiza las rutas en un archivo de router por cada entidad del modelo (laptops, tickets, usuarios, etc.), lo que mantiene el código ordenado y fácil de ampliar. Cada router se crea con APIRouter, indicando un prefijo de ruta y una etiqueta; al registrarlo en la aplicación principal de FastAPI, el framework agrupa automáticamente esos endpoints bajo la misma sección en la documentación de Swagger. El Código 6 muestra el router de la entidad laptop con sus operaciones CRUD: crear, leer, actualizar y eliminar.

Código 6

Router de operaciones CRUD sobre laptops (routers/laptops.py)

```
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from app.database import get_db
from app.models.laptop import Laptop
from app.schemas.laptop import LaptopCreate, LaptopUpdate, LaptopOut
from app.auth import require_admin, require_any_role
from typing import List

router = APIRouter(prefix="/laptops", tags=["Laptops"])

@router.get("/", response_model=List[LaptopOut])
def listar_laptops(db: Session = Depends(get_db),
```

```
                _=Depends(require_any_role)):
    return db.query(Laptop).all()

@router.post("/", response_model=LaptopOut)
def crear_laptop(data: LaptopCreate,
                 db: Session = Depends(get_db),
                 _=Depends(require_any_role)):
    laptop = Laptop(**data.model_dump())
    db.add(laptop)
    db.commit()
    db.refresh(laptop)
    return laptop

@router.put("/{id}", response_model=LaptopOut)
def actualizar_laptop(id: int, data: LaptopUpdate,
                      db: Session = Depends(get_db),
                      _=Depends(require_any_role)):
    laptop = db.query(Laptop).filter(Laptop.id_laptop == id).first()
    if not laptop:
        raise HTTPException(status_code=404, detail="Laptop no encontrada")
    for key, value in data.model_dump(exclude_unset=True).items():
        setattr(laptop, key, value)
    db.commit()
    db.refresh(laptop)
    return laptop

@router.delete("/{id}")
def eliminar_laptop(id: int,
                    db: Session = Depends(get_db),
                    _=Depends(require_admin)):
```

```
    laptop = db.query(Laptop).filter(Laptop.id_laptop == id).first()
    if not laptop:
        raise HTTPException(status_code=404, detail="Laptop no encontrada")
    db.delete(laptop)
    db.commit()
    return {"message": "Laptop eliminada"}
```

Nota. Elaboración propia.

Cada endpoint usa el sistema de inyección de dependencias de FastAPI (Depends) para resolver lo que necesita antes de ejecutar su lógica principal. Por ejemplo, get_db abre y entrega la sesión de base de datos, mientras que require_any_role y require_admin revisan los permisos del usuario antes de dejar pasar la petición. De esta forma, el cuerpo de cada endpoint se concentra solo en su tarea y no mezcla el manejo de la sesión ni la validación de roles.

4.2.6 Autenticación y Autorización

El sistema autentica a los usuarios con JSON Web Tokens (JWT), bajo el flujo de contraseña del estándar OAuth2. Toda la lógica está en el archivo auth.py, que reúne el hasheo de contraseñas con bcrypt, la creación y validación de los tokens (firmados con HS256) y las dependencias de FastAPI que comprueban el rol del usuario. Para el manejo de contraseñas se trabajó directamente con la librería bcrypt, en lugar de passlib, la opción más habitual, debido a una incompatibilidad con la versión de Python utilizada en el proyecto. El Código 7 presenta los componentes principales.

Código 7

Lógica de autenticación y autorización (auth.py)

```
from datetime import datetime, timedelta
from jose import JWTError, jwt
import bcrypt
from fastapi import Depends, HTTPException, status
```

```
from fastapi.security import OAuth2PasswordBearer
from sqlalchemy.orm import Session
from app.database import get_db
from app.models.auth_user import AuthUser
import os

SECRET_KEY = os.getenv("SECRET_KEY", "xkale-super-secret-key-2024")
ALGORITHM  = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 480

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/auth/login")

def hash_password(password: str) -> str:
    return bcrypt.hashpw(password.encode("utf-8"),
                         bcrypt.gensalt()).decode("utf-8")

def verify_password(plain: str, hashed: str) -> bool:
    return bcrypt.checkpw(plain.encode("utf-8"),
                          hashed.encode("utf-8"))

def create_access_token(data: dict, expires_delta=None):
    to_encode = data.copy()
    expire = datetime.utcnow() + (expires_delta or
             timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES))
    to_encode.update({"exp": expire})
    return jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)

def get_current_user(token: str = Depends(oauth2_scheme),
                     db: Session = Depends(get_db)):
    credentials_exception = HTTPException(
```

```
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="No autenticado o token inválido",
    )
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        username = payload.get("sub")
        if username is None:
            raise credentials_exception
    except JWTError:
        raise credentials_exception
    user = db.query(AuthUser).filter(AuthUser.username == username).first()
    if user is None or not user.activo:
        raise credentials_exception
    return user

def require_admin(current_user: AuthUser = Depends(get_current_user)):
    if current_user.rol != "admin":
        raise HTTPException(status_code=403, detail="Se requiere rol admin")
    return current_user
```

Nota. Elaboración propia. Las funciones require_admin_or_tecnico y require_any_role siguen el mismo patrón con distintas validaciones de rol.

La función hash_password emplea bcrypt con generación automática de salt, garantizando que dos contraseñas idénticas produzcan hashes distintos y resistiendo los ataques basados en tablas precalculadas. El tiempo de expiración del token se estableció en 480 minutos (ocho horas), cubriendo una jornada laboral completa y evitando que el usuario deba autenticarse repetidamente durante su trabajo cotidiano. Las funciones require_admin, require_admin_or_tecnico y require_any_role implementan el control de acceso basado en roles (RBAC), inyectándose como dependencias en los endpoints que requieren restricciones específicas.

4.2.7 Consultas Avanzadas con JOIN

Adicionalmente a los routers CRUD básicos, el sistema implementa un router de consultas avanzadas que aprovecha las relaciones definidas en los modelos para resolver casos de uso que requieren cruzar información entre varias tablas. SQLAlchemy ofrece el mecanismo joinedload, el cual permite cargar entidades relacionadas en una sola consulta SQL mediante un JOIN, evitando el problema clásico de las N+1 consultas. El Código 8 presenta una de estas consultas, la cual obtiene los tickets asociados a una laptop identificada por su nombre de host.

Código 8

Consulta avanzada con eager loading (routers/consultas.py)

```
@router.get("/tickets/hostname/{hostname}/detalle",
          response_model=List[TicketDetalle])
def tickets_detalle_por_hostname(hostname: str,
                           db: Session = Depends(get_db),
                           _=Depends(require_any_role)):
    laptop = db.query(Laptop).filter(
        Laptop.hostname.ilike(f"%{hostname}%")
    ).first()
    if not laptop:
        raise HTTPException(status_code=404,
                        detail="Laptop no encontrada")
    tickets = (
        db.query(Ticket)
        .options(joinedload(Ticket.tecnico),
                 joinedload(Ticket.laptop))
        .filter(Ticket.lap_id == laptop.id_laptop)
        .all()
    )
    return tickets
```

Nota. Elaboración propia.

El operador ilike empleado en el filtro permite realizar búsquedas insensibles a mayúsculas y minúsculas, lo cual mejora la experiencia del usuario al buscar equipos por nombre. Los esquemas Pydantic TicketDetalle, TicketResumen, LaptopResumen y TecnicoResumen utilizados como tipos de respuesta permiten anidar la información relacionada en una única estructura JSON, ofreciendo al cliente la totalidad de los datos requeridos sin necesidad de realizar múltiples peticiones.

4.2.8 Documentación Automática de la API

Una de las características más destacadas de FastAPI es la generación automática de documentación interactiva a partir de las anotaciones de tipos, los esquemas Pydantic y los metadatos declarados en los routers. Al iniciar el servidor de desarrollo, las rutas /docs y /redoc exponen interfaces basadas respectivamente en Swagger UI y ReDoc que permiten visualizar todos los endpoints disponibles, sus parámetros, los esquemas de entrada y salida, y ejecutar pruebas manuales directamente desde el navegador. Esta característica eliminó la necesidad de mantener documentación técnica externa durante el desarrollo y facilitó las pruebas integradas con el equipo del departamento de Tecnologías de Información.

4.3 Desarrollo del Frontend con React

La capa de presentación se construyó sobre React en su versión 19, combinada con Tailwind CSS para el sistema de estilos y React Router para la navegación entre páginas. Banks y Porcello (2020) destacan que el éxito de React en proyectos profesionales se sustenta en la combinación equilibrada de su modelo de componentes con un ecosistema de bibliotecas complementarias que cubren las necesidades transversales como enrutamiento, gestión de estado y comunicación con servicios externos.

4.3.1 Inicialización y Enrutamiento

El punto de entrada de la aplicación frontend se encuentra en el archivo main.jsx, donde React inicializa el árbol de componentes dentro del elemento HTML raíz del documento. La aplicación principal, definida en App.jsx, configura el enrutador del lado del cliente y establece las páginas protegidas mediante un componente envolvente que verifica la autenticación y el rol

del usuario. El Código 9 muestra la estructura del enrutamiento adoptada.

Código 9

Configuración de rutas y protección por roles (App.jsx)

```
import { BrowserRouter, Routes, Route, Navigate } from 'react-router-dom';
import { AuthProvider, useAuth } from './context/AuthContext';
import Layout from './components/layout/Layout';
import LoginPage from './pages/auth/LoginPage';
import { lazy, Suspense } from 'react';

const DashboardPage = lazy(() => import('./pages/dashboard/DashboardPage'));
const LaptopsPage   = lazy(() => import('./pages/laptops/LaptopsPage'));
const TicketsPage   = lazy(() => import('./pages/tickets/TicketsPage'));
const UsuariosPage  = lazy(() => import('./pages/usuarios/UsuariosPage'));
const TecnicosPage  = lazy(() => import('./pages/tecnicos/TecnicosPage'));
const AuthUsersPage = lazy(() => import('./pages/auth/AuthUsersPage'));

function ProtectedRoute({ children, roles }) {
  const { user, loading } = useAuth();
  if (loading) return <Loader />;
  if (!user) return <Navigate to="/login" replace />;
  if (roles && !roles.includes(user.rol))
    return <Navigate to="/dashboard" replace />;
  return children;
}

function AppRoutes() {
  const { user } = useAuth();
  return (
    <Routes>
```

```
    <Route path="/login" element={user ? <Navigate to="/dashboard" /> : <LoginPage
/>} />
    <Route path="/" element={<ProtectedRoute><Layout /></ProtectedRoute>}>
      <Route index element={<Navigate to="/dashboard" replace />} />
      <Route path="dashboard"  element={<Suspense fallback={<Loader
/>}><DashboardPage /></Suspense>} />
      <Route path="laptops"    element={<Suspense fallback={<Loader
/>}><LaptopsPage /></Suspense>} />
      <Route path="tickets"    element={<Suspense fallback={<Loader />}><TicketsPage
/></Suspense>} />
      <Route path="usuarios"   element={<ProtectedRoute
roles={['admin','inventario']}>...</ProtectedRoute>} />
      <Route path="tecnicos"   element={<ProtectedRoute
roles={['admin']}>...</ProtectedRoute>} />
      <Route path="auth-users" element={<ProtectedRoute
roles={['admin']}>...</ProtectedRoute>} />
    </Route>
  </Routes>
 );
}
```

Nota. Elaboración propia. Fragmento simplificado: se omiten declaraciones repetitivas de Suspense en rutas protegidas por rol.

La técnica de carga diferida (lazy loading) aplicada mediante React.lazy y Suspense permite que cada página principal se descargue del servidor únicamente cuando el usuario navega hacia ella, reduciendo significativamente el tamaño del paquete inicial y mejorando los tiempos de carga percibidos por el usuario. El control de acceso en el frontend lo maneja el componente ProtectedRoute. Hace dos comprobaciones: primero verifica que el usuario haya iniciado sesión y, si no, lo redirige al login; luego revisa que su rol esté entre los permitidos para esa ruta y, de no estarlo, lo envía al dashboard.

4.3.2 Sistema de Estilos con Tailwind CSS

Los colores de la marca Xkale se definen en Tailwind CSS extendiendo su paleta por defecto. En el archivo tailwind.config.js (Código 10) se agregan los colores corporativos y algunos tonos con significado -éxito, error, advertencia- que usan los componentes de la interfaz.

Código 10

Configuración de la paleta de colores de Xkale en Tailwind CSS

```
export default {
  content: ['./index.html', './src/**/*.{js,jsx,ts,tsx}'],
  theme: {
    extend: {
      colors: {
        primary:         '#6AE000',
        'primary-light': '#8FFF20',
        'primary-dark':  '#4DB000',
        'primary-glow':  'rgba(106,224,0,0.12)',
        dark:            '#0D0D0D',
        card:            '#141414',
        elevated:        '#1F1F1F',
        modal:           '#111111',
        border:          '#2A2A2A',
        secondary:       '#A0A0A0',
        muted:           '#555555',
        danger:          '#FF4545',
        'danger-bg':     'rgba(255,69,69,0.10)',
        warning:         '#F5A623',
        'warning-bg':    'rgba(245,166,35,0.10)',
        info:            '#3D9CF0',
        success:         '#6AE000',
      },
```

```
      fontFamily: {
        sans: ['Inter', 'sans-serif'],
        mono: ['JetBrains Mono', 'monospace'],
      },
    },
  },
};
```

Nota. Elaboración propia.

Tener la paleta en un solo archivo permite cambiar un color en un único lugar y que el ajuste se refleje en toda la interfaz, sin modificar cada componente. Esto facilita el mantenimiento, en línea con lo que plantea Pressman (2010) sobre el ciclo de vida del software.

4.3.3 Gestión Global del Estado de Autenticación

El estado de la sesión y el token se manejan de forma global con la API de Contexto de React, lo que evita tener que pasar esos datos manualmente de componente en componente (prop drilling). El Código 11 muestra el proveedor de autenticación (AuthProvider), que guarda los datos del usuario y expone las funciones para iniciar y cerrar sesión.

Código 11

Proveedor del contexto de autenticación (context/AuthContext.jsx)

```
import { createContext, useContext, useState, useEffect } from 'react';
import { login as loginApi } from '../api/auth';

const AuthContext = createContext(null);

export function AuthProvider({ children }) {
  const [user, setUser]       = useState(null);
  const [loading, setLoading] = useState(true);
```

```
useEffect(() => {
  const token = localStorage.getItem('token');
  const saved = localStorage.getItem('user');
  if (token && saved) setUser(JSON.parse(saved));
  setLoading(false);
}, []);

const login = async (username, password) => {
  const data = await loginApi(username, password);
  localStorage.setItem('token', data.access_token);
  localStorage.setItem('user', JSON.stringify({
    username: data.username, rol: data.rol
  }));
  setUser({ username: data.username, rol: data.rol });
  return data;
};

const logout = () => {
  localStorage.removeItem('token');
  localStorage.removeItem('user');
  setUser(null);
};

const isAdmin         = user?.rol === 'admin';
const canManageUsers  = isAdmin;
const canCreateTicket = isAdmin || user?.rol === 'tecnico';
const canCreateLaptop = isAdmin || user?.rol === 'inventario';

return (
```

```
    <AuthContext.Provider value={{
      user, loading, login, logout,
      isAdmin, canManageUsers, canCreateTicket, canCreateLaptop
    }}>
      {children}
    </AuthContext.Provider>
  );
}

export const useAuth = () => useContext(AuthContext);
```

Nota. Elaboración propia.

La persistencia del estado de autenticación se logra almacenando el token y los datos básicos del usuario en el localStorage del navegador. Al cargar la aplicación, el efecto inicial recupera estos datos si existen, restaurando la sesión sin requerir una nueva autenticación. Los valores derivados isAdmin, canManageUsers, canCreateTicket y canCreateLaptop expuestos en el contexto encapsulan la lógica de autorización del lado del cliente, permitiendo que los componentes muestren u oculten botones y opciones según los permisos del usuario actual.

4.3.4 Comunicación con la API REST

La capa de comunicación con el backend se construyó sobre la biblioteca axios, configurada mediante una instancia centralizada que define la URL base, los encabezados predeterminados y los interceptores de petición y respuesta. El Código 12 muestra el cliente HTTP del sistema.

Código 12

Cliente axios con interceptores (api/client.js)

```
import axios from 'axios';
```

```
const API_URL = import.meta.env.VITE_API_URL || 'http://127.0.0.1:8000';

const client = axios.create({
  baseURL: API_URL,
  headers: { 'Content-Type': 'application/json' },
});

client.interceptors.request.use((config) => {
  const token = localStorage.getItem('token');
  if (token) config.headers.Authorization = `Bearer ${token}`;
  return config;
});

client.interceptors.response.use(
  (res) => res,
  (err) => {
    if (err.response?.status === 401) {
      localStorage.removeItem('token');
      localStorage.removeItem('user');
      window.location.href = '/login';
    }
    return Promise.reject(err);
  }
);

export default client;
```

Nota. Elaboración propia.

El interceptor de peticiones añade automáticamente el token JWT en el encabezado Authorization de cada llamada al backend, evitando que cada función de servicio deba gestionarlo manualmente. El interceptor de respuestas detecta los errores con código 401 que indican la

expiración o invalidación del token, limpia el almacenamiento local y redirige al usuario a la página de inicio de sesión. Sobre este cliente se construyen los módulos de servicios específicos para cada entidad del sistema, los cuales exponen funciones tipadas para las operaciones de creación, lectura, actualización y eliminación de registros.

4.3.5 Componentes Reutilizables

La interfaz se construyó a partir de un conjunto de componentes reutilizables organizados en la carpeta components/ui, los cuales encapsulan los elementos básicos de la interfaz (botones, campos de entrada, tarjetas, modales, tablas, etiquetas) con los estilos corporativos aplicados. La adopción de esta capa de componentes garantiza la consistencia visual de toda la aplicación y reduce el código duplicado en las páginas. El componente Layout, ubicado en components/layout, agrupa la barra lateral de navegación y el área principal de contenido, sirviendo como contenedor común para todas las páginas protegidas del sistema.

4.3.6 Protección de Rutas y Visibilidad por Rol

El componente Sidebar implementa la lógica de visibilidad del menú lateral en función del rol del usuario autenticado. Cada elemento del menú declara los roles autorizados a verlo, y el componente filtra dinámicamente la lista de opciones presentadas. El Código 13 muestra la configuración correspondiente.

Código 13

Filtrado de opciones del menú lateral por rol (components/layout/Sidebar.jsx)

```
const navItems = [
  { to: '/dashboard',  label: 'Dashboard',  icon: LayoutDashboard,
    roles: ['admin','tecnico','inventario'] },
  { to: '/laptops',    label: 'Laptops',    icon: Laptop,
    roles: ['admin','tecnico','inventario'] },
  { to: '/tickets',    label: 'Tickets',    icon: Ticket,
    roles: ['admin','tecnico','inventario'] },
  { to: '/usuarios',   label: 'Usuarios',   icon: Users,
```

```
    roles: ['admin','inventario'] },
  { to: '/tecnicos',   label: 'Técnicos',   icon: Wrench,
    roles: ['admin'] },
  { to: '/auth-users', label: 'Accesos',    icon: ShieldCheck,
    roles: ['admin'] },
];

const filtered = navItems.filter(i => i.roles.includes(user?.rol));
```

Nota. Elaboración propia.

Esta estrategia de control de acceso del lado del cliente complementa, pero no sustituye, las verificaciones de autorización implementadas en el backend mediante las dependencias require_admin y require_any_role. Tal como destaca Sommerville (2011), la seguridad de un sistema debe verificarse siempre en el lado del servidor, dado que un usuario malintencionado podría manipular el código JavaScript del cliente para acceder a opciones que no le corresponden; el filtrado del menú lateral cumple una función exclusivamente ergonómica al ocultar las opciones inaccesibles.

4.4 Pruebas, Integración y Despliegue

Antes de poner el sistema en producción se ejecutó un piloto controlado con el departamento de Tecnologías de Información de Xkale. El propósito fue validar la funcionalidad y la experiencia de uso antes del despliegue definitivo en los servidores internos. El piloto se montó sobre una instancia de prueba con la base de datos PostgreSQL cargada con el histórico real de 95 colaboradores y 89 equipos.

Durante esta fase, el personal del área ejecutó los flujos principales: registro de equipos, asignación de laptops a colaboradores, creación y cierre de tickets, y consulta del histórico de servicio por hostname. Las pruebas de funcionalidad verificaron las operaciones CRUD de cada módulo y las respuestas de los endpoints. Las pruebas de usuario midieron si los flujos resultaban claros para quienes no participaron en el desarrollo. Cada incidencia detectada se anotó y se corrigió antes de avanzar. Superado el piloto, se procedió con la integración y el despliegue descritos a continuación.

Para esta puesta en producción se deben cubrir tres puntos. Las variables de entorno adaptan el sistema al servidor donde se instala. Las dependencias del backend y del frontend deben instalarse en ese servidor. Por último, se define la estrategia de despliegue dentro de la empresa. Estos puntos se detallan en los siguientes apartados.

4.4.1 Variables de Entorno

La configuración sensible del sistema se gestiona mediante archivos de variables de entorno, los cuales se mantienen fuera del control de versiones. En el backend, el archivo .env define la cadena de conexión a PostgreSQL y la clave secreta de firma de los tokens JWT. En el frontend, el archivo .env define la URL base del backend que será consumida por el cliente axios. La Tabla 16 presenta las variables empleadas.

Tabla 16: Variables de entorno del sistema

Componente	Variable	Propósito
Backend	DATABASE_URL	Cadena de conexión completa a PostgreSQL en formato postgresql://usuario:clave@host:puerto/bdd.
Backend	SECRET_KEY	Clave secreta utilizada por python-jose para firmar y verificar los tokens JWT.
Frontend	VITE_API_URL	URL base del backend FastAPI que el cliente axios utiliza como prefijo para todas las peticiones.

Nota. Elaboración propia.

4.4.2 Servidor de Desarrollo

Durante el desarrollo, el backend se ejecuta mediante el servidor ASGI Uvicorn con la opción de recarga automática activada, lo cual reinicia el proceso ante cualquier cambio detectado en los archivos fuente. El frontend, por su parte, se sirve mediante el servidor de desarrollo de Vite, el cual ofrece reemplazo en caliente de módulos (HMR), permitiendo ver los cambios en el navegador sin recargar la página completa. Los comandos correspondientes se muestran en el

Código 14.

Código 14

Comandos para iniciar los servidores de desarrollo

```
# Backend (puerto 8000)
cd backend
python -m venv venv
source venv/bin/activate     # Linux/macOS
venv\Scripts\activate        # Windows
pip install -r requirements.txt
uvicorn app.main:app --reload --host 0.0.0.0 --port 8000

# Frontend (puerto 5173)
cd frontend
npm install
npm run dev
```

Nota. Elaboración propia.

4.4.3 Estrategia de Despliegue On-Premise

Para la puesta en producción dentro de la infraestructura interna de Xkale se contempla un esquema de despliegue compuesto por tres elementos: la base de datos PostgreSQL se instala directamente sobre un servidor Linux interno y se configura para aceptar conexiones desde la red corporativa; el backend FastAPI se ejecuta mediante Uvicorn detrás del servidor de proceso Gunicorn, el cual gestiona múltiples workers para soportar la carga concurrente del departamento de Tecnologías de Información; y el frontend se compila mediante el comando npm run build, generando un conjunto de archivos estáticos optimizados que se sirven a través de un servidor web Nginx, el cual también actúa como proxy inverso hacia el backend.

Este esquema permite que tanto el frontend como el backend respondan en el mismo dominio interno, lo cual elimina la necesidad de configuraciones especiales de CORS en producción y facilita la integración con los mecanismos de autenticación de la red corporativa. La

administración de los procesos del servidor se realiza mediante systemd, que garantiza el reinicio automático de los servicios ante caídas o reinicios del sistema operativo.

4.4.4 Aspectos de Seguridad Corporativa

El sistema maneja datos de equipos y de personal de una empresa que presta servicios a entidades financieras y aseguradoras. Por eso la seguridad se trató como un requisito de primer orden y no como un añadido posterior. Los mecanismos adoptados se describen a continuación.

Autenticación con JWT: El acceso se controla con JSON Web Tokens. Cuando un usuario inicia sesión en la ruta /auth/login, el backend verifica sus credenciales y emite un token firmado. Ese token viaja en la cabecera Authorization de cada petición posterior, y el cliente axios lo adjunta de forma automática mediante un interceptor. El token se firma con el algoritmo HS256 a partir de una clave secreta que solo conoce el servidor. Si la firma no coincide, FastAPI rechaza la petición con un código 401.

Expiración de tokens: Cada token incluye una marca de caducidad en su campo exp. El sistema la fija en 480 minutos mediante la constante ACCESS_TOKEN_EXPIRE_MINUTES, un plazo que cubre una jornada laboral de ocho horas. Pasado ese tiempo, el token deja de ser válido y el usuario debe autenticarse de nuevo. Limitar la vigencia reduce la ventana de riesgo si un token llegara a filtrarse. El valor responde a un equilibrio: una expiración muy corta obligaría a iniciar sesión varias veces al día.

Gestión de la clave secreta: La clave de firma (SECRET_KEY) no se mantiene escrita en el código de producción. Se carga desde una variable de entorno con python-dotenv y queda fuera del control de versiones. Así, quien acceda al repositorio no obtiene la clave real del sistema. En el servidor de Xkale esta clave se define en el archivo .env del backend, accesible solo para los administradores. El valor por defecto presente en el código sirve únicamente para el desarrollo local y se reemplaza por una clave robusta en el despliegue.

Almacenamiento de contraseñas con hash: Las contraseñas nunca se guardan en texto plano. Se procesan con bcrypt, que aplica una función hash de un solo sentido. El proceso es irreversible: a partir del hash almacenado no se puede recuperar la contraseña original. bcrypt genera además un salt aleatorio para cada contraseña mediante gensalt(), de modo que dos usuarios con la misma clave obtienen hashes distintos. Esto frustra los ataques basados en tablas precalculadas. Conviene precisar que bcrypt realiza hashing y no cifrado: el cifrado es una

operación reversible, mientras que el hash de una contraseña debe ser irreversible por diseño.

Control de acceso basado en roles (RBAC): Autenticarse no habilita cualquier acción. El sistema distingue tres roles: Administrador, Técnico e Inventario. Cada endpoint declara qué rol necesita mediante dependencias de FastAPI. Eliminar una laptop exige require_admin, mientras que listar el inventario solo pide require_any_role. La comprobación ocurre antes de ejecutar la lógica del endpoint, y si el rol no alcanza la respuesta es un código 403. Este modelo impide que un usuario de inventario realice operaciones reservadas a un administrador.

Aislamiento de red y despliegue on-premise: El despliegue on-premise mantiene los datos dentro de la infraestructura de Xkale. La base de datos PostgreSQL solo acepta conexiones desde la red corporativa y el sistema no se expone a Internet. Esta decisión responde a las exigencias de confidencialidad de los clientes del sector financiero y asegurador con los que trabaja la empresa.

Política de CORS: Durante el desarrollo, el backend acepta peticiones de cualquier origen para facilitar las pruebas locales. En producción, el frontend y el backend responden bajo el mismo dominio interno servido por Nginx. Al compartir origen desaparece la necesidad de habilitar orígenes cruzados y se reduce la superficie de exposición. Se recomienda restringir la lista de orígenes permitidos del middleware de CORS al dominio interno definitivo.

4.4.5 Justificación de las Decisiones Tecnológicas

El sistema se construyó sobre un conjunto definido de tecnologías. Cada elección respondió a las necesidades del proyecto y al contexto de una sola persona desarrollando bajo un plazo acotado. A continuación, se justifica por qué se prefirió cada herramienta frente a una alternativa habitual.

¿Por qué FastAPI y no Django?: Django es un framework completo orientado a aplicaciones donde el propio servidor genera las vistas, e incluye su ORM, su panel de administración y su motor de plantillas. El sistema de Xkale separa el backend de la interfaz: React construye las vistas y el backend solo expone una API REST. Para ese escenario, FastAPI encaja mejor. Su validación basada en tipos con Pydantic detecta datos incorrectos antes de llegar a la lógica, y genera por sí solo la documentación interactiva en /docs, lo que aceleró las pruebas de cada endpoint (Lubanovic, 2024). El panel de administración y las plantillas de Django no

aportaban valor a una API desacoplada, por lo que su peso adicional no se justificaba.

¿Por qué PostgreSQL y no MySQL?: PostgreSQL y MySQL son bases relacionales válidas para el proyecto. Se eligió PostgreSQL por su rigor en el cumplimiento del estándar SQL y por su manejo de la integridad referencial. El esquema usa claves foráneas con reglas ON UPDATE RESTRICT y ON DELETE RESTRICT, que PostgreSQL respeta de forma estricta. El sistema también depende de consultas con varios JOIN para cruzar tickets, técnicos y laptops, una carga que PostgreSQL resuelve con solvencia. Su modelo de concurrencia permite que varios usuarios del departamento trabajen a la vez sin bloqueos innecesarios. La versión 17 ofreció soporte estable tanto en la nube de Aiven, usada en desarrollo, como en la instalación on-premise de producción.

¿Por qué React y no Angular?: React y Angular permiten construir interfaces por componentes, pero difieren en su enfoque. Angular es un framework completo y rígido que impone una estructura amplia y una curva de aprendizaje mayor. React es una biblioteca más ligera que deja elegir las piezas complementarias. Para un proyecto llevado por una sola persona, esa flexibilidad fue una ventaja. Permitió combinar React con Vite para el empaquetado, Tailwind para los estilos y TanStack Query para la sincronización de datos, sin cargar con el peso completo de un framework opinado. Su modelo de componentes facilitó además reutilizar elementos de interfaz como botones, tarjetas y modales en todas las páginas (Banks y Porcello, 2020).

¿Por qué arquitectura REST y no GraphQL?: El sistema expone operaciones CRUD sobre entidades bien definidas: laptops, tickets, usuarios y técnicos. Para ese patrón, una API REST resulta directa, ya que cada recurso tiene sus rutas y cada verbo HTTP corresponde a una operación. GraphQL destaca cuando el cliente necesita combinar datos muy variables y evitar pedir información de más. El frontend de Xkale consume estructuras predecibles, y los casos que requieren cruzar tablas se resolvieron con endpoints específicos y carga anticipada en SQLAlchemy. Añadir un esquema GraphQL con sus resolutores habría sumado complejidad sin un beneficio claro a esta escala. REST se integra también de forma nativa con la documentación automática de FastAPI.

¿Por qué Kanban y no Scrum?: Scrum está pensado para equipos. Define roles como el dueño de producto y el facilitador, y exige ceremonias fijas como la planificación y la revisión de cada sprint. En un proyecto de una sola persona, esas estructuras no tienen a quién asignarse y se vuelven una carga. Kanban, en su variante Personal Kanban (Anderson, 2010), se ajustó mejor.

Un tablero de tres columnas (Por hacer, En progreso, Terminado) bastó para seguir el avance. El límite de trabajo en curso obligó a terminar una tarea antes de empezar otra, en lugar de dejar el backend y el frontend a medias. Al no imponer roles ni reuniones obligatorias, Kanban se adaptó al ritmo de un trabajo individual con plazo acotado.

CAPÍTULO 5: CONCLUSIONES Y RECOMENDACIONES

El presente capítulo presenta las conclusiones derivadas del trabajo de titulación, organizadas en función del cumplimiento del objetivo general y de cada uno de los objetivos específicos planteados en el Capítulo 1, así como una reflexión sobre la metodología y las tecnologías empleadas. Posteriormente, se exponen las recomendaciones dirigidas a la empresa Xkale Cía. Ltda. para el aprovechamiento sostenible del sistema, y las recomendaciones orientadas a estudiantes e investigadores que aborden proyectos similares en el futuro.

5.1 Conclusiones

El trabajo cumplió con el objetivo general: se desarrolló e implementó un sistema web para gestionar el inventario y el servicio técnico de los equipos de cómputo de Xkale Cía. Ltda., que centraliza el control de su ciclo de vida. Con el sistema en funcionamiento, Xkale dejó de depender de archivos compartidos y hojas de cálculo sueltas y pasó a manejar todo el hardware en una sola plataforma web, protegida con credenciales y con accesos según el perfil de cada usuario. La información queda centralizada dentro de la infraestructura de la empresa, lo que da la trazabilidad que necesitan sus procesos internos.

El proyecto demostró que es viable, técnica y operativamente, sistematizar el control de inventario y el soporte técnico con un stack de código abierto. Al usar herramientas libres, no hubo costos de licencias, y el despliegue on-premise mantiene los datos de los equipos y del personal dentro de la empresa. Esto es importante porque Xkale trabaja con entidades financieras y aseguradoras que exigen condiciones estrictas de confidencialidad y seguridad.

El primer objetivo específico era diseñar la arquitectura del sistema con diagramas entidad-relación y los modelos lógico y físico de la base de datos. Se cumplió, como se documenta en la

sección 3.3. La base de datos se estructuró en cinco entidades: Usuario, Tecnico, Laptop, Ticket y AuthUser. Estas se implementaron en PostgreSQL 17 mediante instrucciones DDL, con índices secundarios para acelerar las búsquedas frecuentes y claves foráneas con reglas ON UPDATE RESTRICT y ON DELETE RESTRICT. El diseño aplica los fundamentos del modelo relacional de Codd (1970), ampliados por Elmasri y Navathe (2016), para asegurar la normalización y la consistencia de los datos. La estabilidad del esquema se comprobó en producción con un histórico de prueba que relaciona a 95 colaboradores con 89 computadoras portátiles.

El segundo objetivo específico, centrado en el control de activos, se cumplió con el módulo de Laptops (descrito en la sección 4.2 y el Anexo A). Este módulo maneja las operaciones CRUD del inventario y guarda doce campos por equipo: datos de compra (factura y fecha), identificación de red (hostname, serial), especificaciones de hardware y el colaborador responsable. En las pruebas con un volumen de datos representativo, las consultas y la carga de los listados completos se resolvieron en menos de dos segundos, cumpliendo el requerimiento no funcional RNF-05.

El tercer objetivo se cumplió con el subsistema de soporte, formado por los módulos de Tickets y Técnicos. Permite seguir cada reporte por sus estados (Abierto, En Proceso y Cerrado), vincular el equipo averiado con el técnico asignado y guardar de forma permanente los diagnósticos y las soluciones. Para consultar esta bitácora de forma eficiente, se usó carga anticipada (eager loading) con la opción joinedload de SQLAlchemy, que resuelve en una sola consulta información como el historial de reparaciones por hostname o el trabajo realizado por cada técnico, algo que antes no era posible con los archivos de cálculo.

El cuarto objetivo específico pedía desplegar el sistema en los servidores internos de la empresa, bajo una arquitectura on-premise, con acceso restringido al personal autorizado. Se cumplió con la configuración descrita en la sección 4.4 del Capítulo 4: el backend corre sobre el

servidor ASGI Uvicorn detrás de un proxy inverso, el frontend se compila como archivos estáticos que sirve Nginx, y la base de datos PostgreSQL se aloja en un servidor interno accesible solo desde la red corporativa.

El acceso al sistema se protege con autenticación por tokens JWT firmados con HS256 y un control de autorización por roles que distingue las acciones permitidas a los perfiles de Administrador, Técnico e Inventario. Las contraseñas no se guardan en texto plano: se almacenan con hash mediante bcrypt, según el requerimiento no funcional RNF-01. Este esquema permite a los administradores gestionar las credenciales de forma centralizada y desactivar cuentas comprometidas sin borrar el historial de actividad.

Usar Kanban, en su variante Personal Kanban (Anderson, 2010), funcionó bien para un proyecto llevado por una sola persona y con un plazo limitado. Un tablero de tres columnas (Por hacer, En progreso, Terminado) dio una vista clara del avance y ayudó a no acumular demasiadas tareas a la vez. Al no imponer roles ni ceremonias fijas, Kanban se adaptó bien a un trabajo individual. Limitar el trabajo en curso (WIP) resultó especialmente útil: obligó a terminar una parte antes de empezar otra, en lugar de avanzar el backend y el frontend en paralelo y dejar ambos a medias. Esto coincide con lo que plantean Pressman (2010) y Sommerville (2011) sobre la ventaja de entregar el código en bloques pequeños y continuos, que ayudan a detectar fallos antes y reducen el costo de rehacer trabajo.

El conjunto de tecnologías elegido -FastAPI, React, Tailwind CSS y PostgreSQL- respondió bien a las necesidades del sistema. FastAPI permitió exponer los servicios REST con buen rendimiento y rapidez de desarrollo, gracias a su validación basada en tipado y a la generación automática de documentación (Lubanovic, 2024). En el cliente, React facilitó construir las vistas como componentes reutilizables, y Tailwind CSS ayudó a mantener un estilo visual uniforme en

toda la interfaz (Banks y Porcello, 2020).

Las tres capas -datos, lógica e interfaz- se comunican mediante intercambios JSON sobre HTTP y tokens JWT para el control de acceso, manteniéndose bien separadas entre sí. El resultado es una arquitectura ordenada, que se puede ampliar con nuevas funcionalidades sin rehacer la estructura existente.

5.2 Recomendaciones

Para que Xkale aproveche bien el sistema a largo plazo, se proponen las siguientes recomendaciones:

- Establecer un procedimiento formal que obligue a registrar en el módulo de inventario cualquier equipo nuevo en el momento de su recepción. Incorporarlo como un paso fijo del proceso de recepción de hardware evita que se acumulen equipos sin registrar y mantiene la trazabilidad del parque al día.
- Designar un responsable principal del sistema dentro del departamento de Tecnologías de Información, encargado de supervisar la calidad de los datos ingresados, la gestión de las credenciales de acceso y la coordinación con los demás miembros del equipo para resolver dudas operativas.
- Realizar copias de respaldo periódicas de la base de datos PostgreSQL mediante la herramienta pg_dump u otra equivalente, almacenando las copias en una ubicación distinta del servidor principal para garantizar la recuperación ante fallos de hardware o incidentes de seguridad.
- Capacitar al personal del departamento de Tecnologías de Información en el uso del sistema mediante una sesión inicial de inducción que cubra los principales flujos de

trabajo: registro de equipos, asignación a colaboradores, creación y cierre de tickets, y consulta del histórico de servicio.

- Aprovechar la información acumulada en el módulo de tickets para realizar análisis periódicos sobre la recurrencia de fallos por modelo, fabricante o usuario, lo cual permitirá tomar decisiones informadas sobre la renovación del parque informático y la negociación con proveedores.
- Mantener actualizadas las dependencias del backend y del frontend mediante revisiones trimestrales de las versiones de FastAPI, SQLAlchemy, React, Vite y los demás componentes empleados, dado que la seguridad y el rendimiento del sistema dependen en parte de la incorporación oportuna de parches publicados por las comunidades responsables de cada biblioteca.

A los estudiantes que aborden proyectos de titulación de naturaleza similar se les formulan las siguientes recomendaciones de carácter metodológico y técnico:

- Iniciar el desarrollo con una definición clara del modelo de datos antes de comenzar a escribir código de aplicación. La estabilidad del esquema relacional facilita significativamente el desarrollo posterior del backend y del frontend, ya que reduce la necesidad de modificaciones retroactivas que afecten a múltiples componentes.
- Mantener una separación estricta entre la lógica del backend y la del frontend desde las primeras fases del proyecto, así como entre las capas internas de cada componente (modelos, esquemas, rutas, componentes de interfaz, servicios de comunicación). Esta separación facilita las pruebas, simplifica la depuración y mejora la mantenibilidad del código en proyectos de larga duración.

- Aprovechar las herramientas de generación automática de documentación ofrecidas por frameworks modernos como FastAPI, las cuales reducen significativamente el esfuerzo de mantener documentación técnica actualizada y proporcionan un canal natural de comunicación entre el desarrollo y la evaluación del proyecto.
- Adoptar el control de versiones desde el primer día del desarrollo mediante una plataforma como GitHub, realizando commits frecuentes con mensajes descriptivos. Esta práctica proporciona un histórico completo de la evolución del proyecto, facilita la recuperación ante errores y constituye un activo verificable durante la defensa del trabajo de titulación.
- Documentar las decisiones técnicas relevantes en archivos de texto dentro del repositorio o en un documento separado a medida que se toman, en lugar de intentar reconstruir el razonamiento al final del proyecto. La memoria sobre las razones detrás de cada decisión se desvanece rápidamente, y el ejercicio de redacción del documento final se vuelve mucho más ágil cuando se cuenta con notas detalladas.

CAPÍTULO 6: TRABAJOS FUTUROS

Una vez se implemente el sistema dentro de la organización se abren las puertas al desarrollo de módulos adicionales que contribuyan positivamente a Xkale. Entre las líneas de trabajo futuras se proponen las siguientes:

- Incorporar un módulo de reportes y analítica que aproveche la información acumulada en las bitácoras de servicio técnico para generar indicadores de gestión como el tiempo promedio de resolución de incidencias, los modelos de equipo con mayor recurrencia de fallos y la carga de trabajo distribuida entre los técnicos. La incorporación de bibliotecas de visualización como Chart.js o Recharts en el frontend permitiría presentar estos indicadores en un panel de control gerencial.
- Extender el modelo de datos para gestionar otros tipos de activos tecnológicos además de las laptops, como servidores, equipos de red, dispositivos periféricos y equipamiento audiovisual. Esta extensión requeriría el diseño de un esquema más flexible que pueda acomodar las características específicas de cada tipo de activo, posiblemente mediante el patrón de tipo y atributos extendidos.
- Implementar un módulo de gestión documental que permita asociar a cada equipo o ticket archivos relacionados como facturas escaneadas, fotografías de daños, manuales del fabricante o documentos de garantía. Esto requeriría incorporar una solución de almacenamiento de archivos en el backend y los controles de visualización correspondientes en el frontend.
- Desarrollar una versión móvil del sistema o una aplicación complementaria que permita a los técnicos consultar y actualizar tickets directamente desde sus dispositivos durante las visitas a los puestos de trabajo. Esta extensión podría aprovechar el carácter REST

de la API existente, consumiéndola desde un cliente nativo construido con tecnologías como React Native o Flutter.

- Incorporar un mecanismo de notificaciones automáticas por correo electrónico que alerte a los responsables sobre eventos relevantes como la apertura de un nuevo ticket, la asignación de una intervención, la proximidad del vencimiento de garantías o la inactividad prolongada de un equipo. La integración con el servicio de correo corporativo aumentaría el valor del sistema como herramienta de coordinación interna.
- Realizar pruebas automatizadas formales del sistema mediante frameworks como pytest para el backend y Vitest o Jest para el frontend, complementadas con pruebas de integración extremo a extremo. La construcción de una suite de pruebas automatizadas constituye una práctica recomendada por Sommerville (2011) para garantizar la calidad del software durante su evolución posterior.

Bibliografía

Anderson, D. J. (2010). *Kanban: Successful evolutionary change for your technology business.* Sequim, WA: Blue Hole Press.

Banks, A., & Porcello, E. (2020). *Learning React: Modern patterns for developing React apps.* Sebastopo, CA: O´Reilly Media.

Buitrón, C. (2026). *Xkale-inventory: Sistema de gestión de inventario y servicio técnico*. Obtenido de GitHub: https://github.com/cbuitron03/xkale-inventory

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 377-87.

Date, C. J. (2003). *An introduction to database systems.* Boston, MA: Pearson Addison-Wesley.

Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of database system.* Boston, MA: Pearson.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures.* Irvine, CA: University of California, Irvine.

Grinberg, M. (2018). *Flask web development: Developing web applications with Python.* Sebastopol, CA: O´Reilly Media.

International Organization for Standardization. (2017). *ISO/IEC 19770-1:2017 Information technology — IT asset management — Part 1: IT asset management systems.* Ginebra: ISO.

Lubanovic, B. (2024). *FastAPI: Modern Python web development.* Sebastopol, CA: O'Reilly Media.

Nielsen, J. (1994). *Usability engineering.* San Francisco, CA: Morgan Kaufmann.

Pressman, R. S. (2010). *Ingeniería de Software: Un enfoque práctico.* México D.F: McGraw-Hill.

PostgreSQL Global Development Group . (2024). Retrieved from PostgreSQL 17 documentation: https://www.postgresql.org/docs/17/

Sommerville, I. (2011). *Ingeniería de software.* México: Pearson Educación.

Tailwind Labs. (05 de 05 de 2026). *Tailwind CSS*. Obtenido de Tailwind CSS documentation: https://tailwindcss.com/doc

Tanenbaum, A. S., & Van Steen, M. (2017). *Distributed systems.* London: Pearson Education.

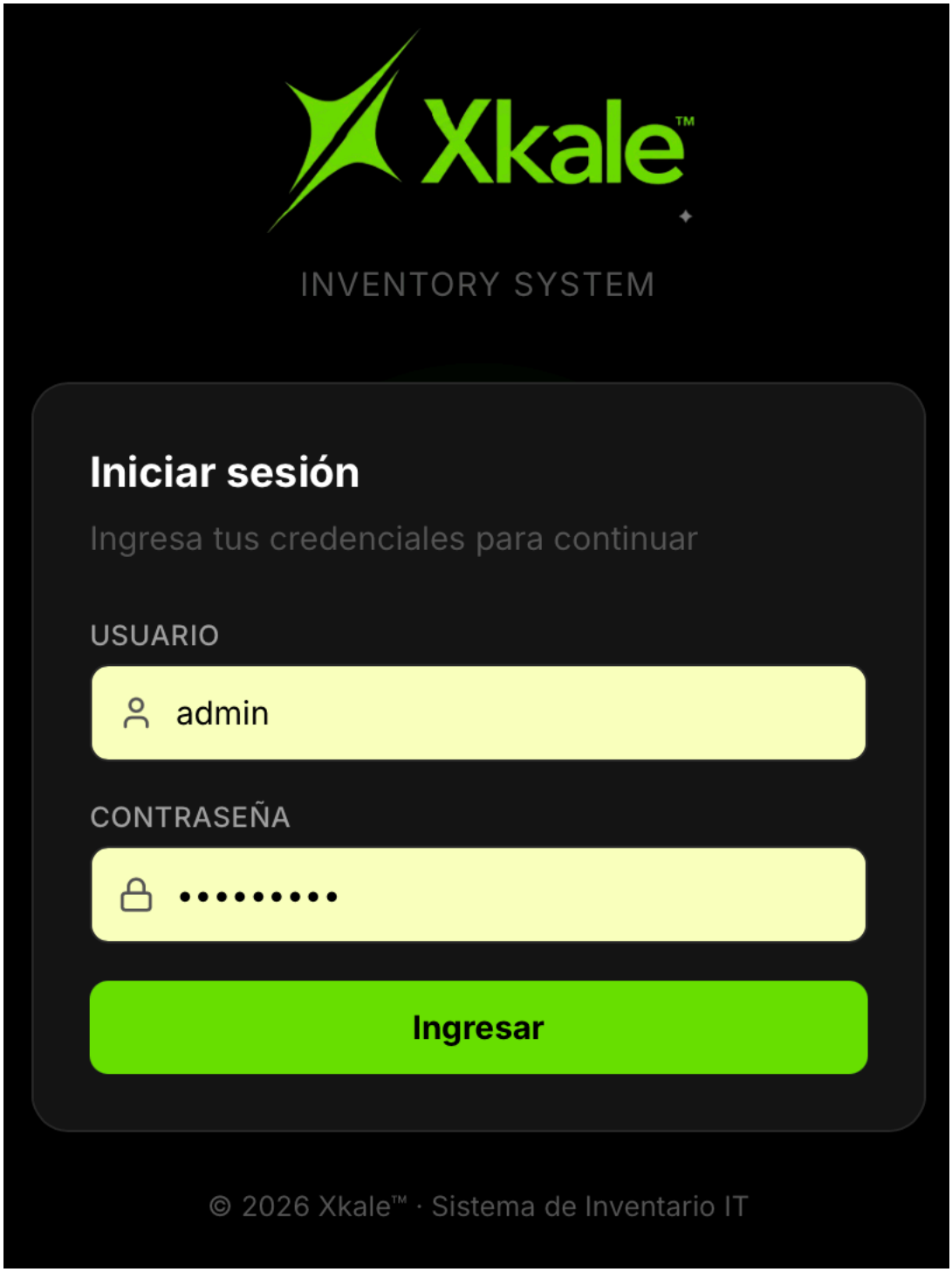
Anexos

Esta sección contiene el material complementario referenciado a lo largo del presente trabajo de titulación. El Anexo A presenta el conjunto de capturas de pantalla del sistema desarrollado, agrupadas por módulo funcional, las cuales evidencian el resultado final de la implementación. El Anexo B contiene el script de definición de la base de datos PostgreSQL empleada por el sistema. El Anexo C contiene el catálogo de endpoints existentes dentro del backend separados por módulo. El Anexo D contiene la información referente al repositorio de Github en el que se encuentra el proyecto.

ANEXO A: Capturas de Pantalla del Sistema

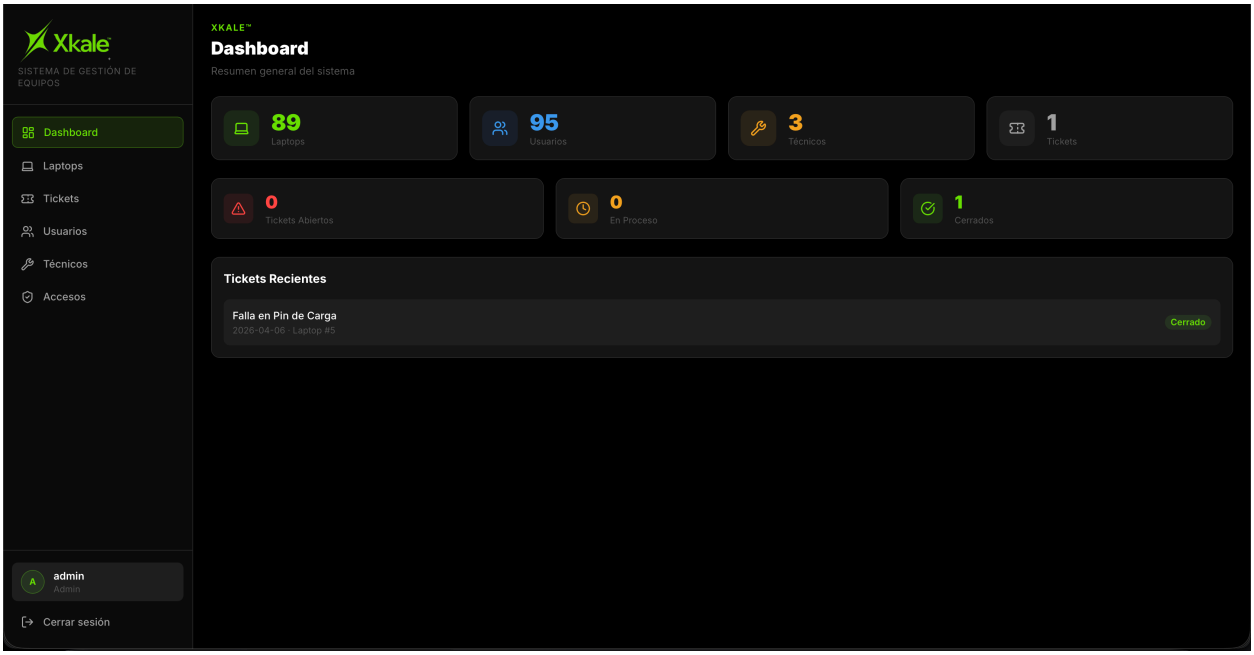
El presente anexo recoge las capturas de pantalla obtenidas del sistema en funcionamiento, las cuales documentan la apariencia final de cada módulo y de los principales formularios de registro. Las figuras se organizan siguiendo el flujo natural de uso de la aplicación, comenzando por la autenticación, continuando con el panel principal y los módulos operativos, y finalizando con los formularios modales de creación de registros.

Figura A1: Pantalla de inicio de sesión del sistema



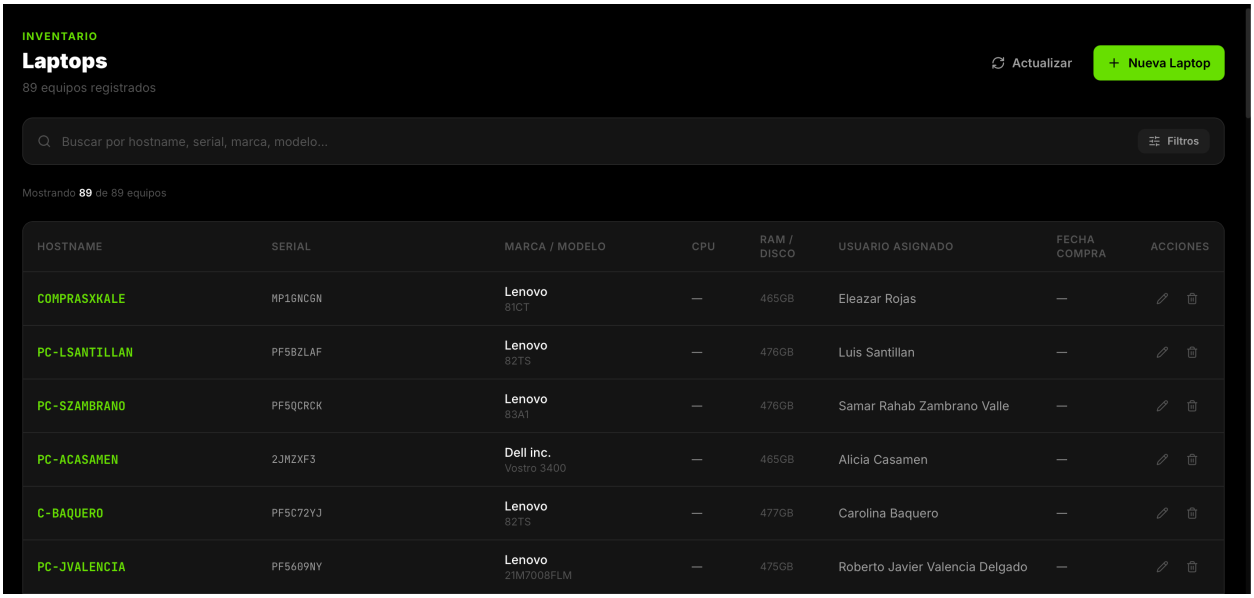
Nota. Pantalla de autenticación del Sistema Xkale Inventory. Se presentan los campos de usuario y contraseña, así como el botón de ingreso al sistema. Elaboración propia.

Figura A2: Panel principal (Dashboard) del sistema



Nota. Panel principal con las métricas resumen del sistema: cantidad de laptops, usuarios, técnicos y tickets registrados, así como el desglose por estado y la sección de tickets recientes. Elaboración propia.

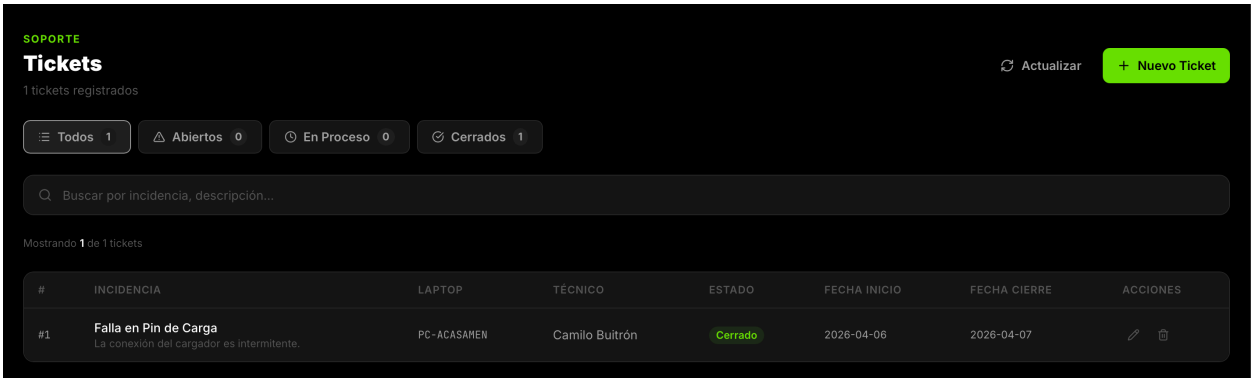
Figura A3: Módulo de inventario de laptops



Nota. Listado del parque informático con las columnas de hostname, serial, marca, modelo,

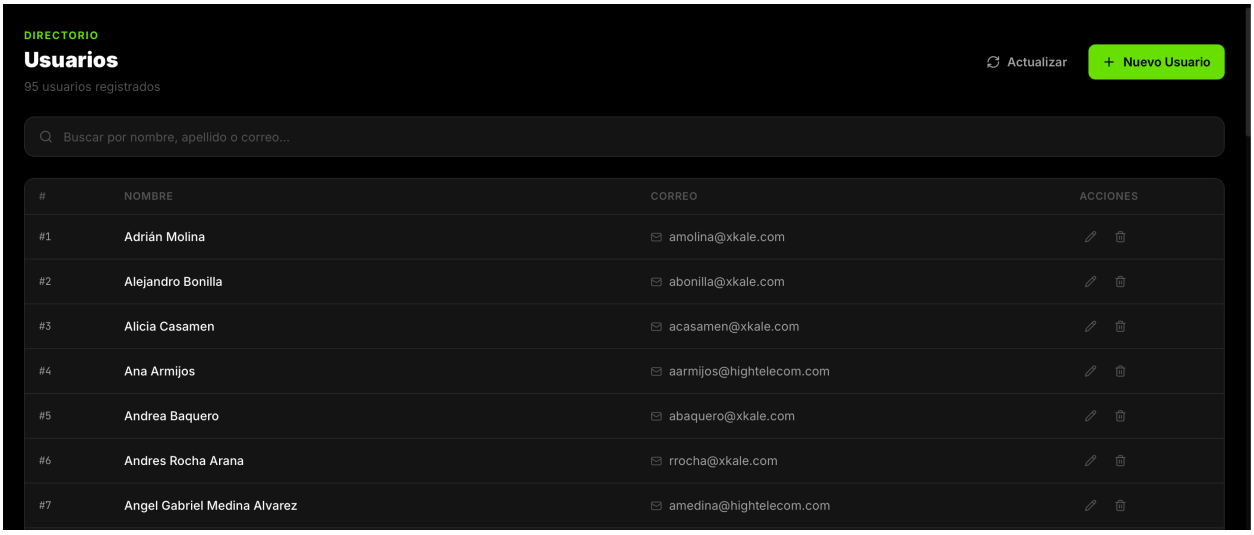
especificaciones técnicas, usuario asignado y fecha de compra. Elaboración propia.

Figura A4: Módulo de tickets de servicio técnico



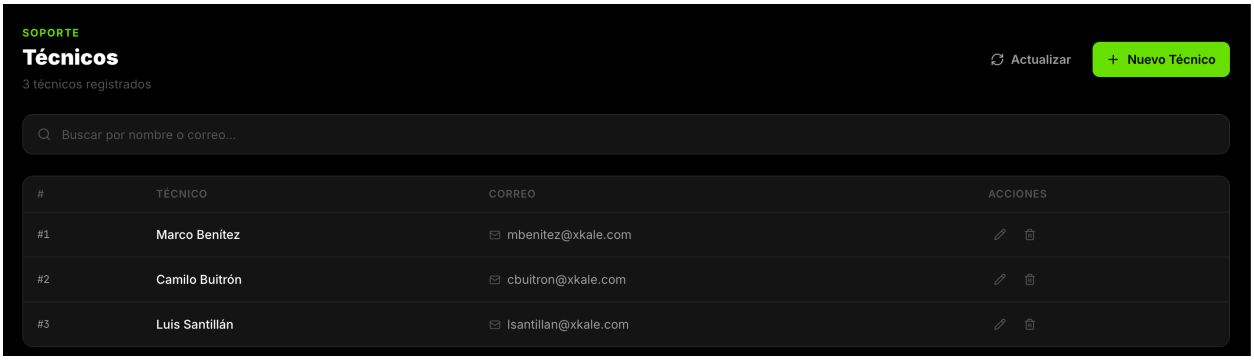
Nota. Pantalla de gestión de tickets con pestañas de filtrado por estado (Todos, Abiertos, En Proceso, Cerrados) y tabla con la información completa de cada incidencia. Elaboración propia.

Figura A5: Módulo de directorio de usuarios



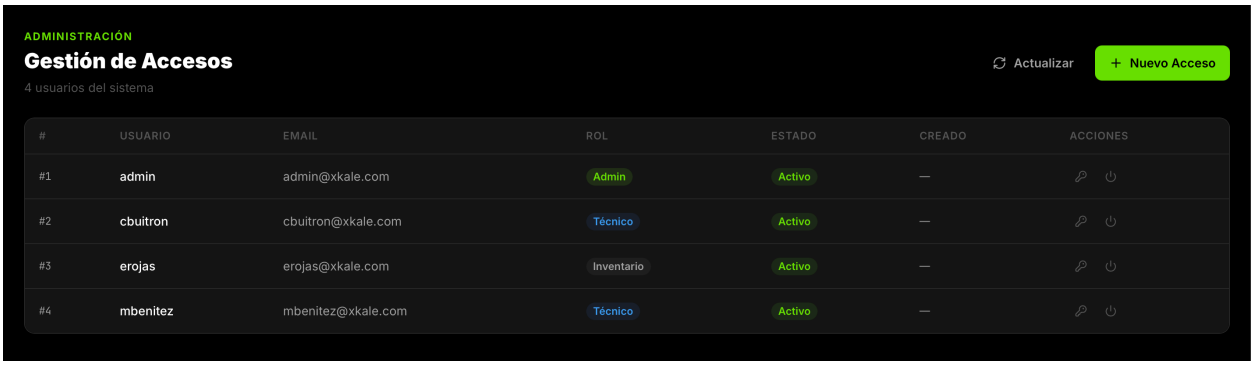
Nota. Directorio de colaboradores de la organización con sus nombres y correos corporativos. Elaboración propia.

Figura A6: Módulo de técnicos de soporte



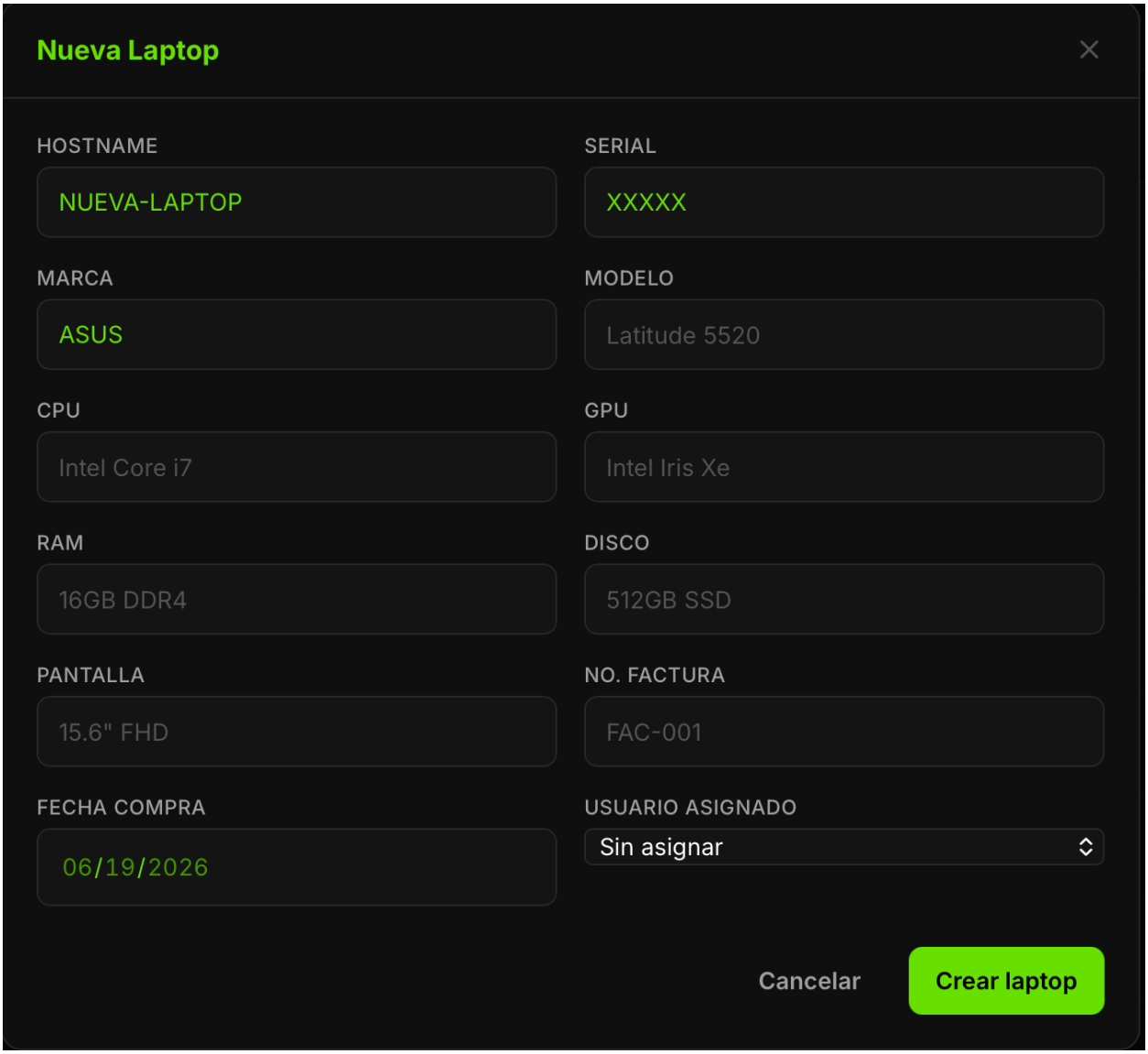
Nota. Listado de técnicos responsables del servicio técnico de los equipos. Elaboración propia.

Figura A7: Módulo de gestión de accesos al sistema



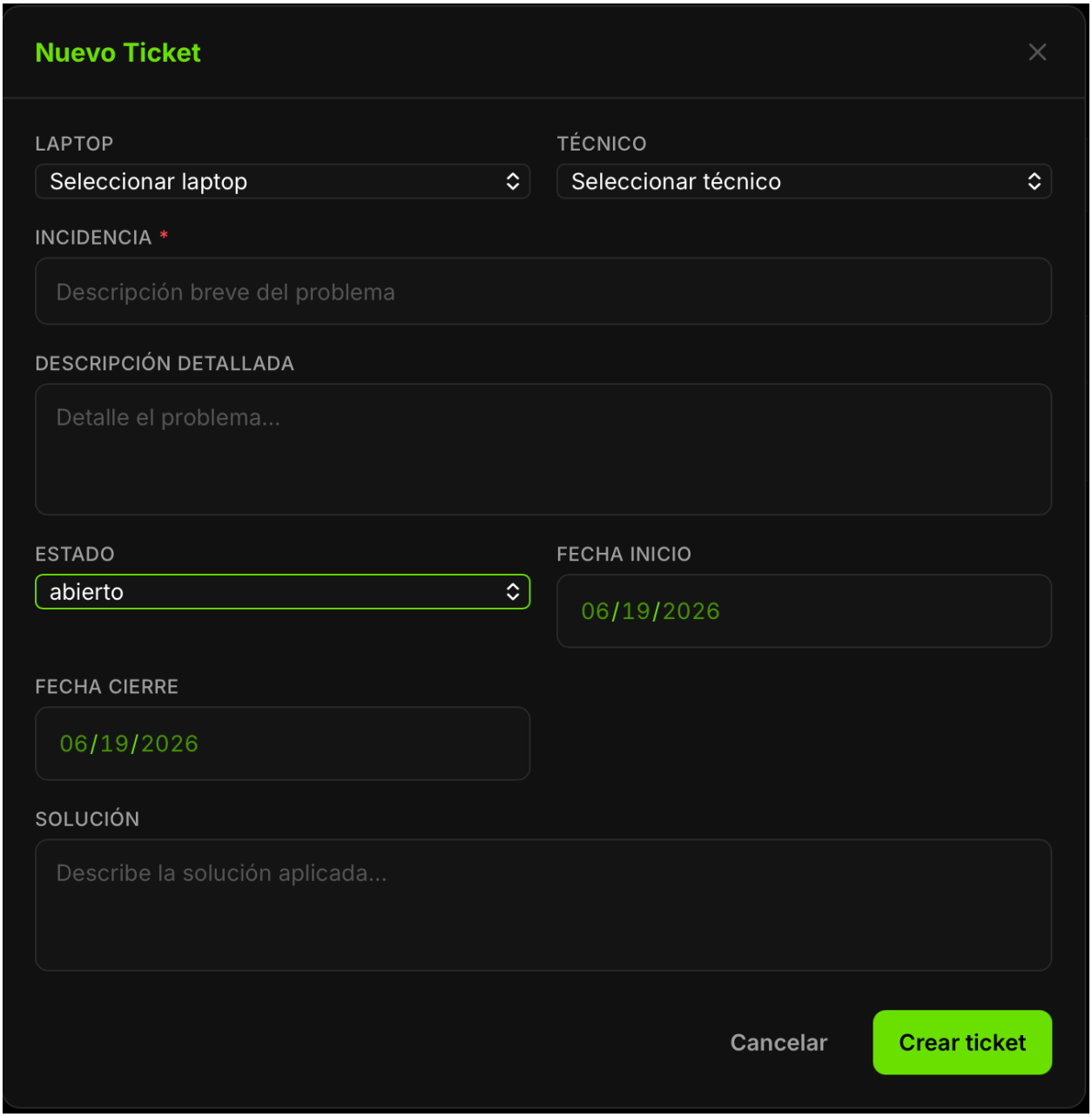
Nota. Pantalla de administración de credenciales con el rol asignado y el estado de cada cuenta. Acceso restringido al rol Administrador. Elaboración propia.

Figura A8: Formulario de registro de nueva laptop



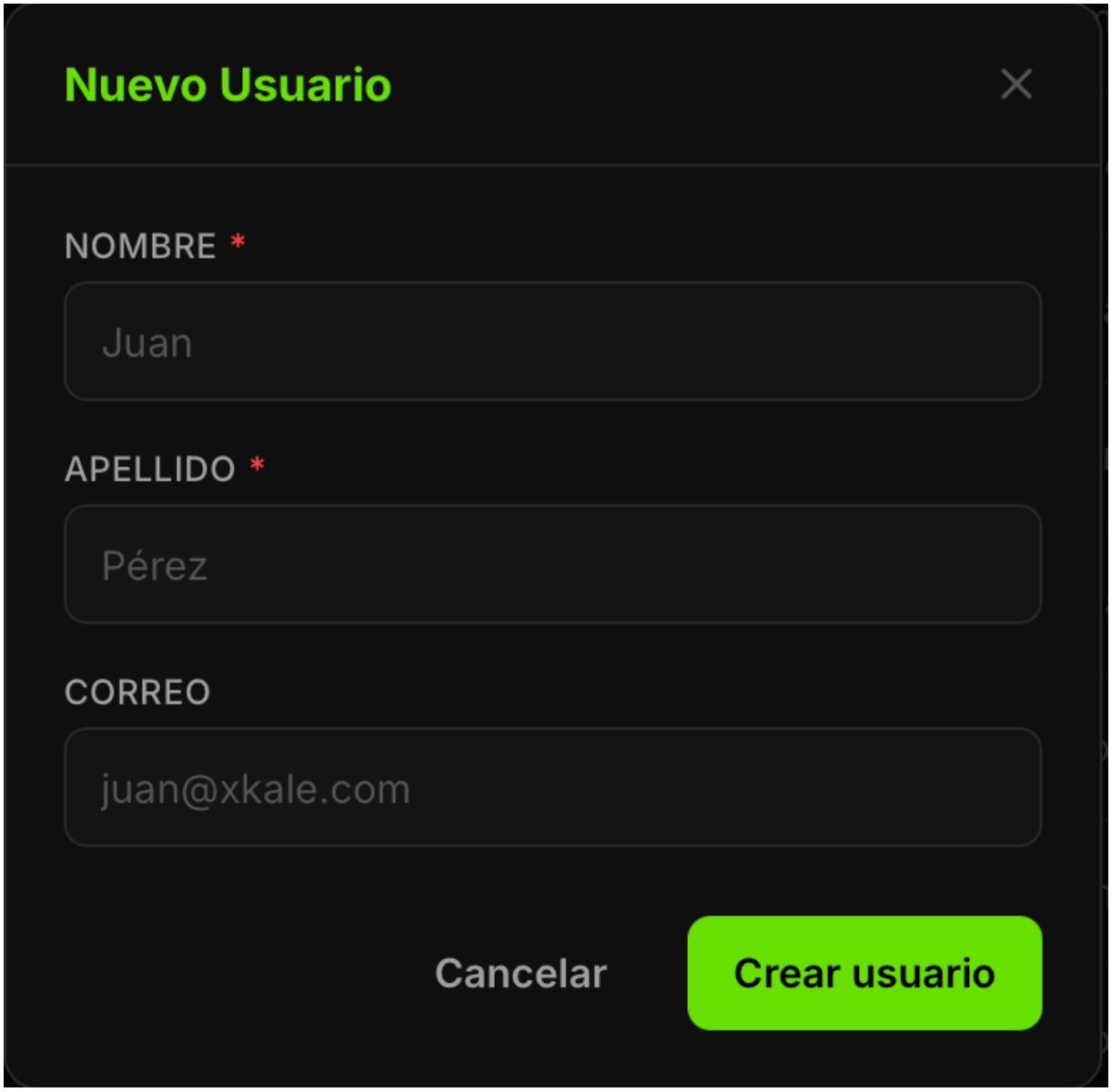
Nota. Modal de creación de un nuevo equipo en el inventario, con los campos de identificación, especificaciones técnicas y asignación a un colaborador. Elaboración propia.

Figura A9: Formulario de creación de ticket



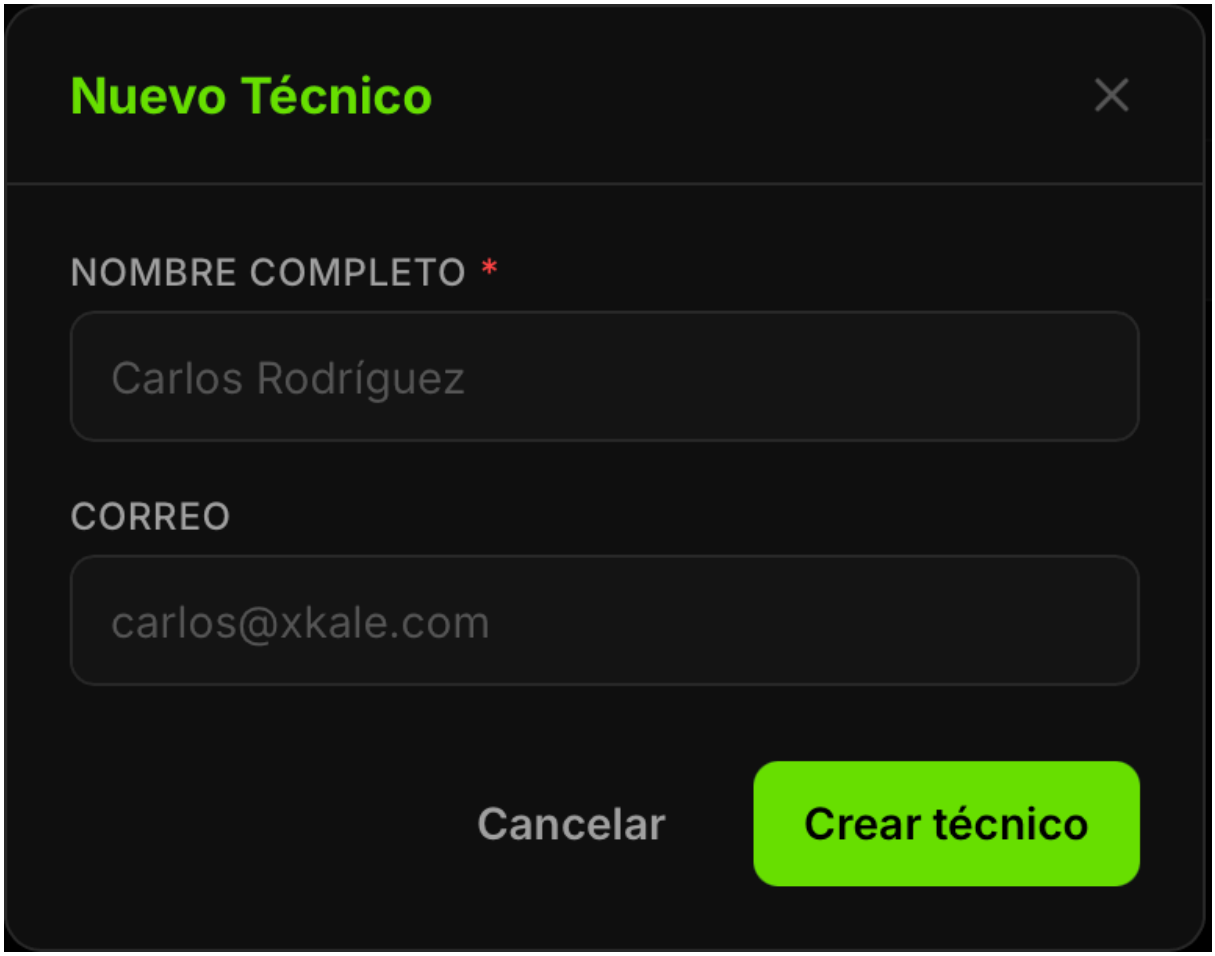
Nota. Modal de registro de una nueva incidencia técnica, donde se vincula el equipo afectado con el técnico responsable. Incluye campos para la descripción del problema, el estado, las fechas y la solución aplicada. Elaboración propia.

Figura A10: Formulario de registro de nuevo usuario



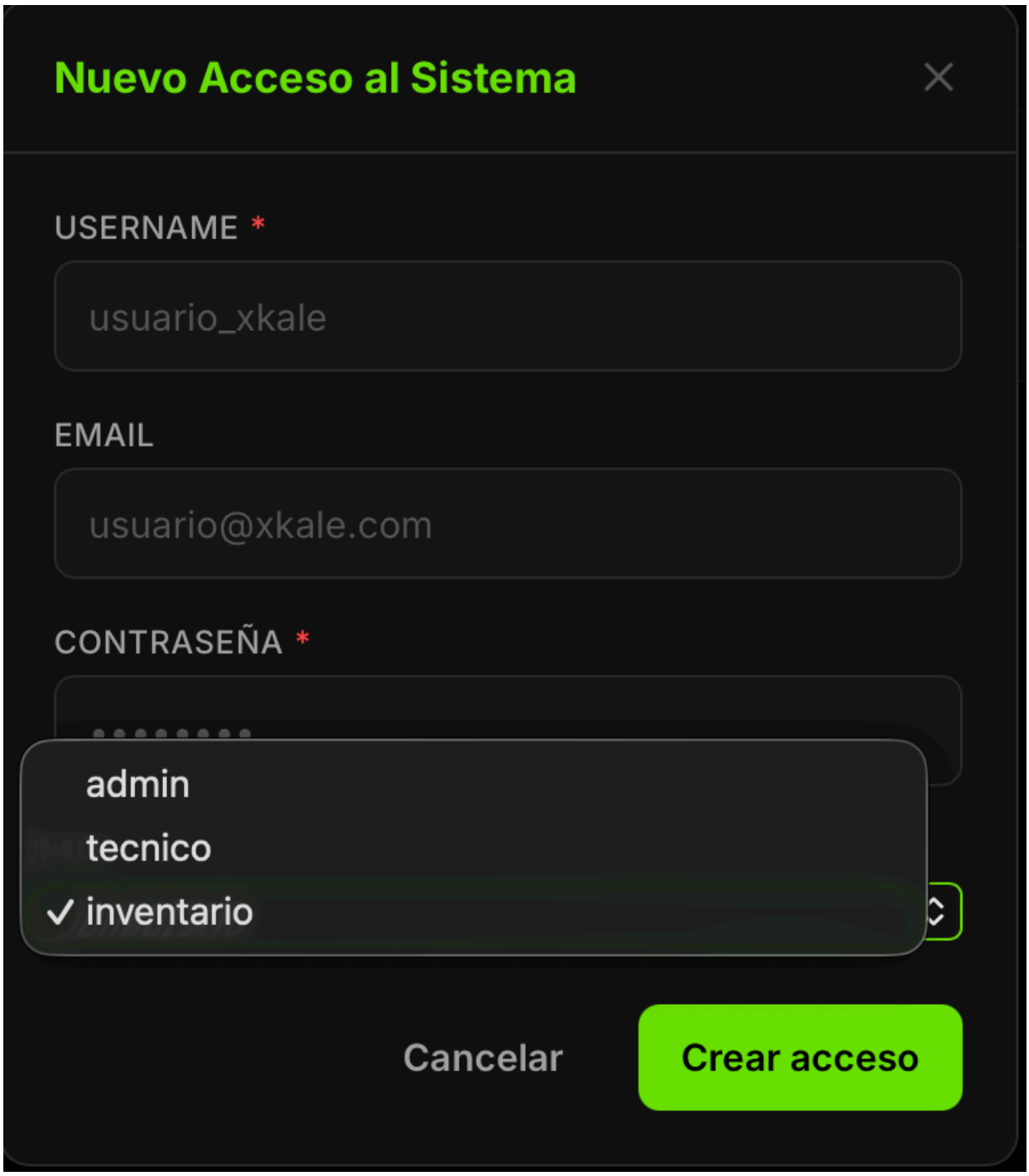
Nota. Modal de creación de un nuevo colaborador en el directorio de usuarios de la organización. Elaboración propia.

Figura A11: Formulario de registro de nuevo técnico



Nota. Modal de creación de un nuevo técnico responsable del soporte de equipos. Elaboración propia.

Figura A12: Formulario de creación de credenciales de acceso



Nota. Modal de generación de un nuevo acceso al sistema, incluyendo nombre de usuario, correo, contraseña y rol asignado (Administrador, Técnico o Inventario). Elaboración propia.

ANEXO B: Script de Definición de la Base de Datos

El siguiente fragmento corresponde al script de definición (DDL) de la base de datos PostgreSQL del sistema, generado mediante la herramienta de modelado entidad-relación integrada en pgAdmin 4. El script crea las cinco tablas principales del sistema con sus respectivas claves primarias, restricciones de integridad y relaciones foráneas.

```
-- This script was generated by the ERD tool in pgAdmin 4.
-- Please log an issue at https://github.com/pgadmin-org/pgadmin4/issues/new/choose if you find any bugs, including reproduction steps.
BEGIN;

CREATE TABLE IF NOT EXISTS public.auth_user
(
    id_auth serial NOT NULL,
    username text COLLATE pg_catalog."default" NOT NULL,
    email text COLLATE pg_catalog."default",
    hashed_password text COLLATE pg_catalog."default" NOT NULL,
    rol text COLLATE pg_catalog."default" NOT NULL DEFAULT 'inventario'::text,
    activo boolean NOT NULL DEFAULT true,
    created_at timestamp without time zone DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT auth_user_pkey PRIMARY KEY (id_auth),
    CONSTRAINT auth_user_username_key UNIQUE (username)
);

CREATE TABLE IF NOT EXISTS public.laptop
(
    id_laptop integer NOT NULL GENERATED ALWAYS AS IDENTITY ( INCREMENT 1 START 1 MINVALUE 1 MAXVALUE 2147483647 CACHE 1 ),
    usu_id_laptop integer NOT NULL,
    serial text COLLATE pg_catalog."default",
    marca text COLLATE pg_catalog."default",
    modelo text COLLATE pg_catalog."default",
```

```
    cpu text COLLATE pg_catalog."default",
    gpu text COLLATE pg_catalog."default",
    ram text COLLATE pg_catalog."default",
    disco text COLLATE pg_catalog."default",
    pantalla text COLLATE pg_catalog."default",
    no_factura text COLLATE pg_catalog."default",
    fecha_compra date,
    hostname text COLLATE pg_catalog."default",
    CONSTRAINT laptop_pkey PRIMARY KEY (id_laptop)
);

CREATE TABLE IF NOT EXISTS public.tecnico
(
    id_tecnico integer NOT NULL GENERATED ALWAYS AS IDENTITY ( INCREMENT 1
START 1 MINVALUE 1 MAXVALUE 2147483647 CACHE 1 ),
    tecnico_nombre text COLLATE pg_catalog."default",
    tecnico_correo text COLLATE pg_catalog."default",
    CONSTRAINT tecnico_pkey PRIMARY KEY (id_tecnico)
);

CREATE TABLE IF NOT EXISTS public.ticket
(
    id_ticket integer NOT NULL GENERATED ALWAYS AS IDENTITY ( INCREMENT 1
START 1 MINVALUE 1 MAXVALUE 2147483647 CACHE 1 ),
    tec_id integer NOT NULL,
    lap_id integer NOT NULL,
    incidencia text COLLATE pg_catalog."default",
    descripcion text COLLATE pg_catalog."default",
    estado text COLLATE pg_catalog."default",
    fecha_inicio date,
    fecha_cierre date,
```

```
    solucion text COLLATE pg_catalog."default",
    CONSTRAINT ticket_pkey PRIMARY KEY (id_ticket)
);

CREATE TABLE IF NOT EXISTS public.usuario
(
    id_usuario integer NOT NULL GENERATED ALWAYS AS IDENTITY ( INCREMENT 1
START 1 MINVALUE 1 MAXVALUE 2147483647 CACHE 1 ),
    nombre text COLLATE pg_catalog."default",
    apellido text COLLATE pg_catalog."default",
    correo text COLLATE pg_catalog."default",
    CONSTRAINT usuario_pkey PRIMARY KEY (id_usuario)
);

ALTER TABLE IF EXISTS public.laptop
    ADD CONSTRAINT fk_laptop_usuario FOREIGN KEY (usu_id_laptop)
    REFERENCES public.usuario (id_usuario) MATCH SIMPLE
    ON UPDATE RESTRICT
    ON DELETE RESTRICT;
CREATE INDEX IF NOT EXISTS idx_laptop_usuario
    ON public.laptop(usu_id_laptop);

ALTER TABLE IF EXISTS public.ticket
    ADD CONSTRAINT fk_ticket_laptop FOREIGN KEY (lap_id)
    REFERENCES public.laptop (id_laptop) MATCH SIMPLE
    ON UPDATE RESTRICT
    ON DELETE RESTRICT;
CREATE INDEX IF NOT EXISTS idx_ticket_laptop
    ON public.ticket(lap_id);

ALTER TABLE IF EXISTS public.ticket
```

```
    ADD CONSTRAINT fk_ticket_tecnico FOREIGN KEY (tec_id)
    REFERENCES public.tecnico (id_tecnico) MATCH SIMPLE
    ON UPDATE RESTRICT
    ON DELETE RESTRICT;
CREATE INDEX IF NOT EXISTS idx_ticket_tecnico
    ON public.ticket(tec_id);

END;
```

Nota. Script DDL generado mediante pgAdmin 4 para PostgreSQL 17. Elaboración propia.

ANEXO C: Catálogo de Endpoints de la API REST

El presente anexo documenta el conjunto completo de endpoints expuestos por la API REST del sistema, construida sobre FastAPI. Los endpoints se agrupan por módulo funcional siguiendo la organización en routers descrita en el Capítulo 4, y se describen mediante cuatro atributos: el método HTTP empleado (GET, POST, PUT o DELETE), la ruta relativa expuesta por el servidor, el rol mínimo requerido para invocar la operación y una descripción breve del propósito de cada endpoint.

Adicionalmente a la documentación presentada en este anexo, FastAPI genera de manera automática una página de documentación interactiva accesible en la ruta /docs del servidor, la cual permite explorar todos los endpoints, conocer la estructura detallada de los esquemas de entrada y salida, y ejecutar pruebas manuales directamente desde el navegador. Esta característica resulta particularmente útil para verificar el comportamiento de los endpoints durante el desarrollo y para entregar documentación técnica actualizada al equipo de Tecnologías de Información de la organización.

Los roles mencionados en la columna correspondiente se refieren al control de acceso basado en roles (RBAC) implementado mediante las dependencias require_admin, require_admin_or_tecnico y require_any_role descritas en la Sección 4.2.6 del Capítulo 4. La indicación "Todos los roles" significa que el endpoint requiere autenticación válida pero no impone restricciones adicionales por rol.

C.1 Módulo de Autenticación

Los endpoints del módulo de autenticación gestionan el ciclo de inicio de sesión, la administración de credenciales y la consulta del perfil del usuario autenticado. El endpoint POST /auth/login es el único endpoint público del sistema, dado que es el punto de entrada para obtener el token JWT requerido por todos los demás endpoints. La Tabla C1 detalla la composición de este módulo.

Tabla C1: Endpoints del módulo de autenticación

Método	Ruta	Rol	Descripción
POST	/auth/login	Público	Autenticar usuario con credenciales y obtener token JWT.
POST	/auth/register	Administrador	Registrar una nueva credencial de acceso en el sistema.
GET	/auth/users	Administrador	Listar todas las credenciales registradas en el sistema.
PUT	/auth/change-password	Administrador	Actualizar la contraseña de un usuario existente.
PUT	/auth/toggle/{username}	Administrador	Activar o desactivar la credencial indicada.
GET	/auth/me	Autenticado	Obtener la información del usuario actualmente autenticado.

Nota. Elaboración propia a partir del archivo routers/auth.py del proyecto.

C.2 Módulo de Usuarios

Los endpoints del módulo de usuarios permiten gestionar el directorio de colaboradores de la organización a quienes se asignan las laptops. Las operaciones de lectura están disponibles para los tres roles del sistema, mientras que las operaciones de creación, modificación y eliminación se restringen al rol de Administrador. La Tabla C2 presenta el detalle de cada endpoint.

Tabla C2: Endpoints del módulo de usuarios

Método	Ruta	Rol	Descripción
GET	/usuarios/	Todos los roles	Listar todos los colaboradores del directorio.
GET	/usuarios/{id}	Todos los roles	Obtener los datos de un colaborador específico por su identificador.
POST	/usuarios/	Administrador	Registrar un nuevo colaborador en el directorio.
PUT	/usuarios/{id}	Administrador	Actualizar los datos de un colaborador existente.
DELETE	/usuarios/{id}	Administrador	Eliminar un colaborador del directorio.

Nota. Elaboración propia a partir del archivo routers/usuarios.py del proyecto.

C.3 Módulo de Técnicos

Los endpoints del módulo de técnicos administran el registro del personal de soporte responsable de atender las incidencias reportadas. La consulta del listado y de registros individuales se encuentra disponible para todos los roles, mientras que las operaciones de modificación quedan reservadas al rol de Administrador. La Tabla C3 documenta este módulo.

Tabla C3: Endpoints del módulo de técnicos

Método	Ruta	Rol	Descripción
GET	/tecnicos/	Todos los roles	Listar todos los técnicos de soporte registrados.
GET	/tecnicos/{id}	Todos los roles	Obtener los datos de un técnico específico.
POST	/tecnicos/	Administrador	Registrar un nuevo técnico

			de soporte.
PUT	/tecnicos/{id}	Administrador	Actualizar los datos de un técnico existente.
DELETE	/tecnicos/{id}	Administrador	Eliminar un técnico del registro.

Nota. Elaboración propia a partir del archivo routers/tecnicos.py del proyecto.

C.4 Módulo de Laptops

Los endpoints del módulo de laptops constituyen el núcleo operativo del sistema, ya que gestionan el ciclo de vida completo del parque informático. Las operaciones de creación, lectura y actualización están autorizadas para todos los roles, lo cual habilita al personal del área de Inventario para mantener el registro actualizado. Sin embargo, la eliminación de equipos se restringe al rol de Administrador como medida de protección frente a borrados accidentales del histórico. La Tabla C4 detalla los endpoints del módulo.

Tabla C4: Endpoints del módulo de laptops

Método	Ruta	Rol	Descripción
GET	/laptops/	Todos los roles	Listar todas las laptops registradas en el inventario.
GET	/laptops/{id}	Todos los roles	Obtener los datos de una laptop específica.
POST	/laptops/	Todos los roles	Registrar una nueva laptop en el inventario.
PUT	/laptops/{id}	Todos los roles	Actualizar los datos de una laptop existente.
DELETE	/laptops/{id}	Administrador	Eliminar una laptop del inventario.

Nota. Elaboración propia a partir del archivo routers/laptops.py del proyecto.

C.5 Módulo de Tickets

Los endpoints del módulo de tickets gestionan las incidencias técnicas reportadas sobre los equipos del parque informático. La consulta de tickets está habilitada para los tres roles del sistema, mientras que las operaciones de creación, modificación y eliminación requieren el rol de Técnico o de Administrador, dado que estas acciones impactan directamente sobre la información histórica de servicio. La Tabla C5 presenta el catálogo completo de este módulo.

Tabla C5: Endpoints del módulo de tickets

Método	Ruta	Rol	Descripción
GET	/tickets/	Todos los roles	Listar todos los tickets de servicio técnico.
GET	/tickets/{id}	Todos los roles	Obtener los datos de un ticket específico.
POST	/tickets/	Administrador, Técnico	Registrar un nuevo ticket de incidencia.
PUT	/tickets/{id}	Administrador, Técnico	Actualizar los datos de un ticket existente.
DELETE	/tickets/{id}	Administrador, Técnico	Eliminar un ticket del registro.

Nota. Elaboración propia a partir del archivo routers/tickets.py del proyecto.

C.6 Módulo de Consultas Avanzadas

Los endpoints del módulo de consultas avanzadas extienden las capacidades CRUD básicas mediante operaciones que resuelven casos de uso específicos que requieren cruzar información entre varias entidades del sistema. Estas consultas aprovechan el mecanismo de carga ansiosa (joinedload) de SQLAlchemy para minimizar el número de consultas SQL ejecutadas. Todas las consultas avanzadas se encuentran disponibles para los tres roles autenticados, dado que su propósito es estrictamente de consulta y no modifican el estado del sistema. La Tabla C6 documenta los endpoints disponibles.

Tabla C6: Endpoints del módulo de consultas avanzadas

Método	Ruta	Rol	Descripción
GET	/consultas/laptops/hostname/{hostname}/detalle	Todos los roles	Obtener una laptop por su hostname con usuario asignado y tickets asociados.
GET	/consultas/tickets/hostname/{hostname}/detalle	Todos los roles	Obtener todos los tickets de una laptop identificada por su hostname.
GET	/consultas/usuarios/correo/{correo}/laptops	Todos los roles	Obtener un usuario con todas las laptops asignadas a su nombre.
GET	/consultas/usuarios/correo/{correo}	Todos los roles	Buscar un usuario por su correo electrónico.
GET	/consultas/laptops/marca/{marca}	Todos los roles	Listar todas las laptops de una marca específica con sus tickets.
GET	/consultas/tickets/tecnico/{correo}	Todos los roles	Obtener todos los tickets atendidos por un técnico identificado por su correo.

Nota. Elaboración propia a partir del archivo routers/consultas.py del proyecto.

C.7 Endpoint de Estado del Servicio

Finalmente, el sistema expone un endpoint público en la ruta raíz del servidor, el cual permite verificar de manera rápida el estado operativo del servicio sin requerir autenticación previa. Este endpoint resulta útil para herramientas de monitoreo externo y para confirmar que el servidor se encuentra disponible. La Tabla C7 describe este endpoint.

Tabla C7: Endpoint de verificación de estado

Método	Ruta	Rol	Descripción
GET	/	Público	Endpoint de verificación de estado del servicio (health check).

Nota. Elaboración propia.

C.8 Resumen del Catálogo

El sistema expone un total de 33 endpoints distribuidos en siete módulos funcionales, conforme al detalle de la Tabla C8. Esta organización modular facilita la mantenibilidad del código y permite que la documentación interactiva generada por FastAPI agrupe los endpoints relacionados, mejorando la experiencia del desarrollador o auditor que consulta la API.

Tabla C8: Resumen cuantitativo de endpoints por módulo

Módulo	**Endpoints**
Autenticación	6
Usuarios	5
Técnicos	5
Laptops	5
Tickets	5
Consultas avanzadas	6
Estado del servicio	1
TOTAL	**33**

Nota. Elaboración propia.

ANEXO D: Repositorio del Código Fuente

La totalidad del código fuente desarrollado durante el presente trabajo de titulación se encuentra disponible en un repositorio público de GitHub, lo cual facilita su revisión, su reproducción y su evolución futura por parte del personal del departamento de Tecnologías de Información de Xkale Cía. Ltda. La publicación del código fuente bajo un repositorio versionado constituye además una práctica recomendada en el desarrollo profesional moderno, dado que documenta el histórico completo de cambios y proporciona trazabilidad sobre las decisiones técnicas adoptadas durante la construcción del sistema.

D.1 Información General del Repositorio

Tabla D1: Información del repositorio del proyecto

Atributo	**Valor**
Plataforma	GitHub
Propietario	cbuitron03
Nombre del repositorio	xkale-inventory
URL completa	https://github.com/cbuitron03/xkale-inventory
Visibilidad	Público
Rama principal	master
Lenguajes principales	JavaScript (73.1%), Python (24.1%), CSS (2.5%), HTML (0.3%)
Sistema de control de versiones	Git

Nota. Elaboración propia.

D.2 Acceso al Repositorio

El acceso al repositorio se realiza mediante la URL pública indicada en la Tabla D1. Para obtener una copia local del código fuente, el lector puede emplear el sistema de control de versiones Git mediante el comando que se presenta a continuación, el cual descarga la totalidad del proyecto incluyendo su historial de cambios:

```
git clone https://github.com/cbuitron03/xkale-inventory.git
```

Alternativamente, GitHub ofrece la opción de descargar el contenido completo del repositorio como un archivo comprimido en formato ZIP a través del botón Code disponible en la página principal del proyecto, lo cual permite obtener el código fuente sin necesidad de tener instalado Git en el equipo del lector.

D.3 Guía de Navegación del Código

Para facilitar la exploración del repositorio, la Tabla D2 presenta los directorios y archivos más relevantes del proyecto, junto con una indicación del contenido y del módulo del sistema al que corresponden. Esta guía permite al lector localizar rápidamente las implementaciones discutidas en los capítulos previos sin necesidad de explorar toda la estructura de carpetas.

Tabla D2: Guía de navegación de los archivos principales del repositorio

Ruta	Descripción
backend/	Componente del lado del servidor, implementado con FastAPI.
backend/requirements.txt	Lista de dependencias de Python con versiones específicas.
backend/app/main.py	Punto de entrada de la aplicación FastAPI.
backend/app/database.py	Configuración de SQLAlchemy y sesión de base de datos.
backend/app/auth.py	Lógica de autenticación JWT, hashing con bcrypt y dependencias de roles.
backend/app/models/	Modelos SQLAlchemy de las cinco entidades del sistema.
backend/app/schemas/	Esquemas Pydantic para validación de datos de entrada y salida.
backend/app/routers/	Routers que agrupan los endpoints por módulo funcional.
backend/app/routers/consultas.py	Consultas avanzadas con JOIN entre múltiples

	entidades.
backend/create_admin.py	Script de arranque para crear el primer usuario administrador.
frontend/	Componente del lado del cliente, implementado con React.
frontend/package.json	Lista de dependencias de Node.js y scripts del proyecto.
frontend/tailwind.config.js	Configuración de Tailwind CSS con la paleta corporativa.
frontend/vite.config.js	Configuración del bundler Vite.
frontend/src/main.jsx	Punto de entrada de la aplicación React.
frontend/src/App.jsx	Definición de rutas y protección por rol.
frontend/src/api/	Cliente axios y servicios de comunicación con el backend.
frontend/src/components/ui/	Componentes reutilizables: Button, Input, Card, Modal, Table, Badge.
frontend/src/components/layout/	Componentes de estructura: Layout y Sidebar.
frontend/src/context/AuthContext.jsx	Contexto global de autenticación.
frontend/src/pages/	Páginas principales agrupadas por módulo funcional.
frontend/src/theme/index.js	Definición JavaScript de la paleta de colores.

Nota. Elaboración propia.

D.4 Licencia y Propiedad Intelectual

El código fuente publicado en el repositorio constituye propiedad intelectual del autor del presente trabajo de titulación. La empresa Xkale Cía. Ltda. dispone de los derechos de uso interno del sistema para los fines descritos en el Capítulo 1, dado que su construcción se realizó como parte del trabajo de integración curricular orientado a resolver una necesidad operativa específica de la organización. Cualquier uso del código por parte de terceros ajenos a la empresa o al autor requiere autorización expresa.