Centro Industrial y Desarrollo
Empresarial Soacha
Regional Cundinamarca

Manual Técnico

Flooky Pets

Juan Bocanegra

Juan Rodríguez

Kevin Moreno

CENTRO INDUSTRIAL Y DESARROLLO EMPRESARIAL – CIDE

ANÁLISIS Y DESARROLLO DE SOFTWARE - ADSO

SOACHA, CUNDINAMARCA

2025

Centro Industrial y Desarrollo
Empresarial Soacha
Regional Cundinamarca

Manual Técnico

1. Introducción

El presente Manual Técnico detalla la arquitectura, el funcionamiento y los procesos clave de Flooky Pets, una plataforma web diseñada para la gestión integral de la atención veterinaria. Desarrollado por los Aprendices SENA del Centro Industrial y Desarrollo Empresarial (CIDE), programa ANÁLISIS Y DESARROLLO DE SOFTWARE (ADSO), en Soacha, Cundinamarca, durante el año 2025, este sistema busca transformar la interacción entre los dueños de mascotas y las clínicas veterinarias, así como optimizar significativamente los flujos de trabajo internos de estas últimas.

Flooky Pets se concibe como una solución robusta y accesible, centrada en la eficiencia y la usabilidad. Para lograrlo, la plataforma se estructura en torno a diferentes interfaces principales, cada una adaptada a las necesidades de sus usuarios con roles definidos:

- Menú de Administrador: Este módulo centralizado, como se detalla en la Sección 10.1, está diseñado para la supervisión y gestión completa de las operaciones de la clínica. Permite a los administradores un control granular sobre usuarios (clientes, veterinarios y otros administradores), servicios ofrecidos, programación de citas y la gestión de historiales médicos. Componentes clave como el AdminDashboard, Gestión de Administradores, Gestión de Servicios y Gestión de Citas aseguran una administración eficiente y segura del sistema.

- Menú de Veterinario: Como se describe en la Sección 10.2, esta interfaz está dedicada a los profesionales de la salud animal. Permite a los veterinarios agendar y gestionar sus propias citas, lo que les da flexibilidad en la administración de su tiempo. Además, es fundamental para el manejo de la información de los pacientes, permitiéndoles agregar nuevas mascotas a clientes existentes y, de manera crítica, registrar y consultar entradas en el historial médico de cada animal. Esto asegura una atención informada y un seguimiento preciso del estado de salud de las mascotas.
- Menú de Usuario (Clientes/Dueños de Mascotas): Si bien este manual se enfoca en la parte técnica del lado del personal de la clínica, el sistema también provee una interfaz intuitiva para los dueños de mascotas. Este módulo permite a los usuarios interactuar con la plataforma para gestionar sus propias citas, acceder a los perfiles de sus mascotas y consultar su historial clínico de manera controlada y segura, facilitando así una participación activa en el cuidado de sus animales.

2. Objetivos del Proyecto

2.1 Objetivo general

Desarrollar una plataforma web para la gestión integral de la atención veterinaria, facilitando la interacción entre dueños de mascotas y la clínica, y optimizando los procesos internos de la misma.

2.2 Objetivos específicos

Desarrollar la funcionalidad de gestión de citas en línea (programación, modificación, cancelación): Este objetivo es crucial ya que permite a los dueños de mascotas programar, modificar o cancelar citas de manera eficiente y autónoma, reduciendo la carga de trabajo del personal de la clínica y mejorando la experiencia del usuario.

Desarrollar un módulo de historia clínica electrónica (HCE) para el registro y consulta de información médica: La implementación de un HCE es fundamental para llevar un registro detallado y organizado de la información médica de cada mascota. Esto facilita el seguimiento de tratamientos, diagnósticos y alergias, mejorando la calidad de la atención veterinaria y la toma de decisiones clínicas.

Desarrollar un panel de administración para la gestión del sistema y generación de informes: Un panel de administración robusto es esencial para el correcto funcionamiento del sistema. Permite

a los administradores gestionar usuarios, servicios, citas y generar informes que facilitan la toma de decisiones estratégicas y la optimización de los procesos internos de la clínica.

3. Alcance del proyecto

El proyecto Flooky Pets tiene como alcance principal el desarrollo e implementación de una plataforma web integral para la gestión de servicios veterinarios. La solución está específicamente diseñada para atender las necesidades de clínicas veterinarias y dueños de mascotas ubicados en la ciudad de Soacha, Cundinamarca.

La concepción y el alcance de Flooky Pets se fundamentan en una investigación del contexto actual del sector veterinario en Soacha, revelando los siguientes hallazgos críticos (según datos del DANE y la Cámara de Comercio de Soacha):

- **Alta Demanda Insatisfecha y Desequilibrio Geográfico:** Soacha, con aproximadamente 1.2 millones de habitantes y una población estimada de 180,000 mascotas, cuenta con un número limitado de clínicas veterinarias registradas (entre 50 y 80). Esto genera una saturación de servicios en las zonas céntricas y un marcado desabastecimiento en áreas periféricas como Altos de Cazucá, dificultando el acceso oportuno a la atención veterinaria.
- **Informalidad y Desorganización del Sector:** Se estima que el 20% de los servicios veterinarios operan de manera informal, sin registro en la Cámara de Comercio.

Esta situación obstaculiza la recopilación de datos reales, impide la trazabilidad efectiva de los historiales clínicos de las mascotas y afecta la estandarización de los servicios.

- **Deficiencia de Herramientas Digitales Integradas:** Un muestreo realizado a través de Google Maps indica que aproximadamente el 65% de las veterinarias en Soacha carecen de sistemas de gestión integrados. Esto se traduce en la ausencia de funcionalidades cruciales como historiales médicos digitales, agendamiento de citas en línea y perfiles de mascota detallados con su estado de salud actual.

En respuesta a estas problemáticas, Flooky Pets se posiciona para ofrecer una solución robusta y optimizada a todas aquellas veterinarias de Soacha que actualmente carecen de un sistema de información eficiente. Nuestro objetivo es facilitar y centralizar los procesos veterinarios mediante un conjunto de funcionalidades clave:

- **Gestión Integral de Citas:** Permitiendo la programación, visualización, modificación, reasignación y cancelación de citas en línea por parte de dueños de mascotas, así como la gestión y seguimiento por parte del personal de la clínica (administradores y veterinarios).
- **Administración de Usuarios y Roles:** Estableciendo un sistema de autenticación y autorización robusto que define roles específicos para administradores, veterinarios y dueños de mascotas, con funcionalidades y permisos diferenciados para cada tipo de usuario.
- **Gestión Detallada de Mascotas y Perfiles:** Ofreciendo la capacidad de registrar perfiles completos de mascotas y vincularlos a sus dueños, incluyendo datos generales y la gestión de su estado actual de salud.
- **Módulo de Historia Clínica Electrónica (HCE):** Proporcionando una herramienta centralizada para el registro sistemático y la consulta de todos los eventos médicos relevantes de cada mascota, desde diagnósticos y prescripciones hasta resultados de laboratorio y procedimientos, garantizando la trazabilidad.
- **Catálogo y Gestión de Servicios Veterinarios**: Permitir la definición, actualización y visualización de los diversos servicios que ofrece la clínica, con información detallada sobre cada uno.

- **Funcionalidades de Comunicación y Notificaciones:** Implementación de un sistema de alertas para recordatorios de citas, actualizaciones de estado y comunicaciones importantes entre la clínica y los dueños de mascotas.
- **Panel de Administración y Reportes:** Una interfaz centralizada para la supervisión global del sistema, incluyendo la gestión de todos los datos maestros (usuarios, servicios, etc.) y la generación de informes básicos que apoyen la toma de decisiones operativas y estratégicas.

Exclusiones del Alcance:

Es importante destacar que el alcance actual del proyecto no incluye:

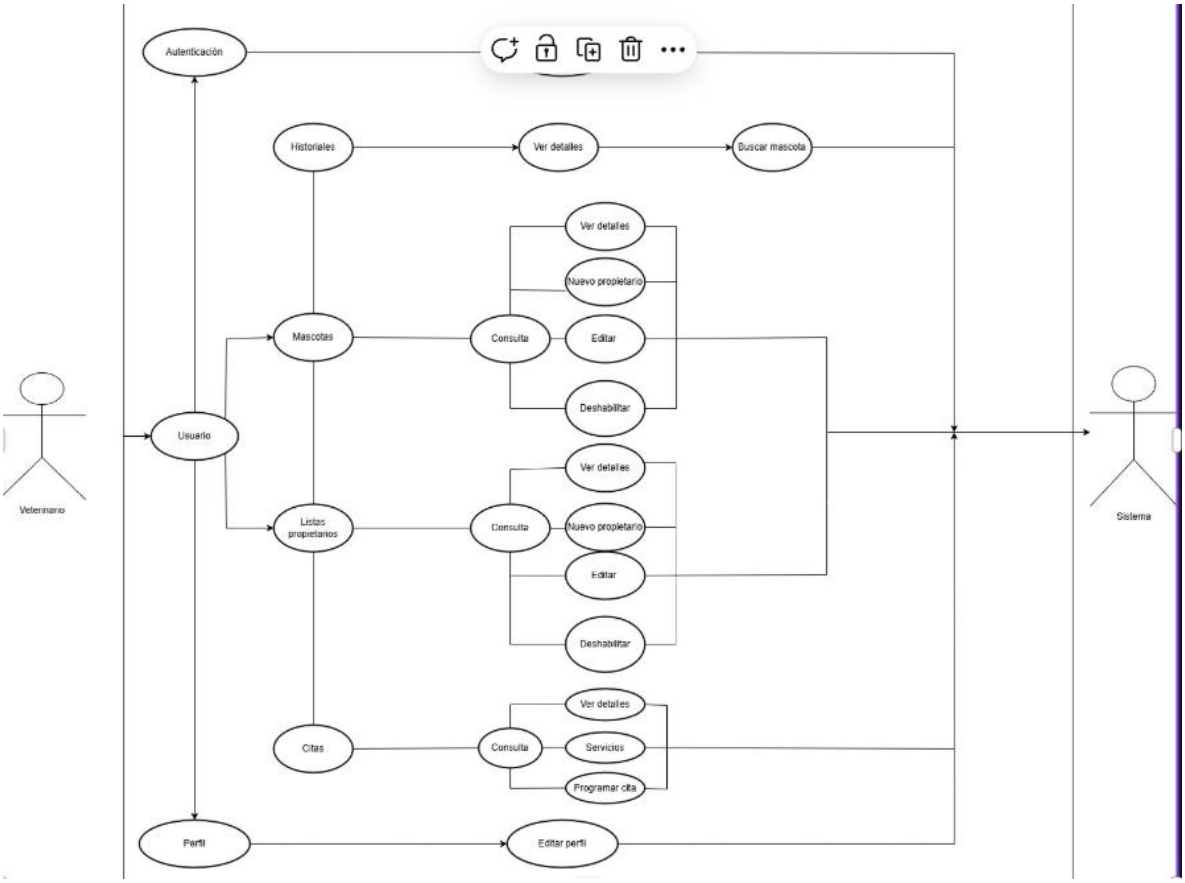
- Integración directa con sistemas de pago externos para transacciones financieras.
- Funcionalidades de inventario para productos o medicamentos de la clínica.
- Módulo de contabilidad o facturación avanzada.
- Aplicaciones móviles nativas (el enfoque es una plataforma web responsiva).
- Soporte para múltiples sucursales de clínicas veterinarias (el sistema está diseñado inicialmente para una única clínica).

El desarrollo de Flooky Pets busca, por tanto, no solo optimizar los procesos de las veterinarias en Soacha, sino también contribuir a la formalización del sector y a mejorar sustancialmente el acceso y la calidad de la atención veterinaria para la comunidad y sus mascotas en la región.

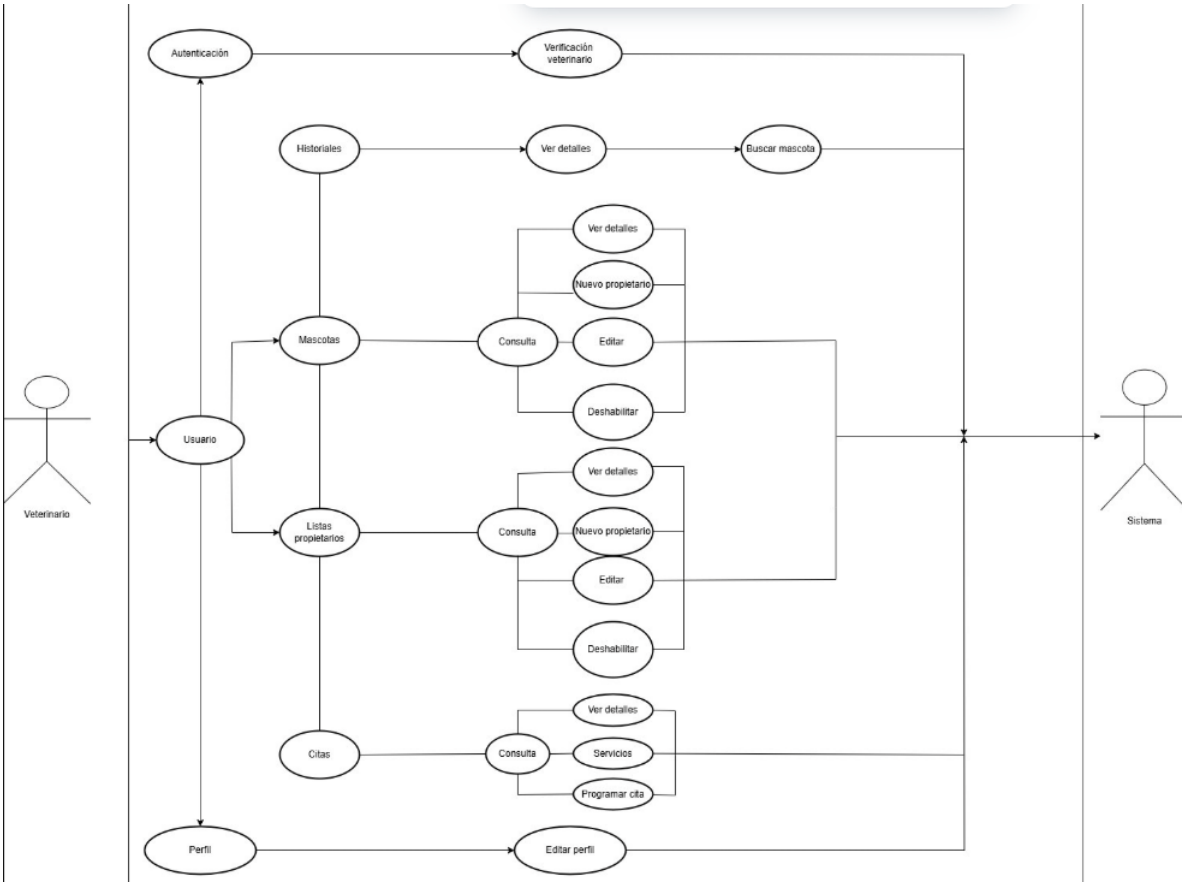
4. Personal Involucrado

4.1 Diagramas de Casos de Uso

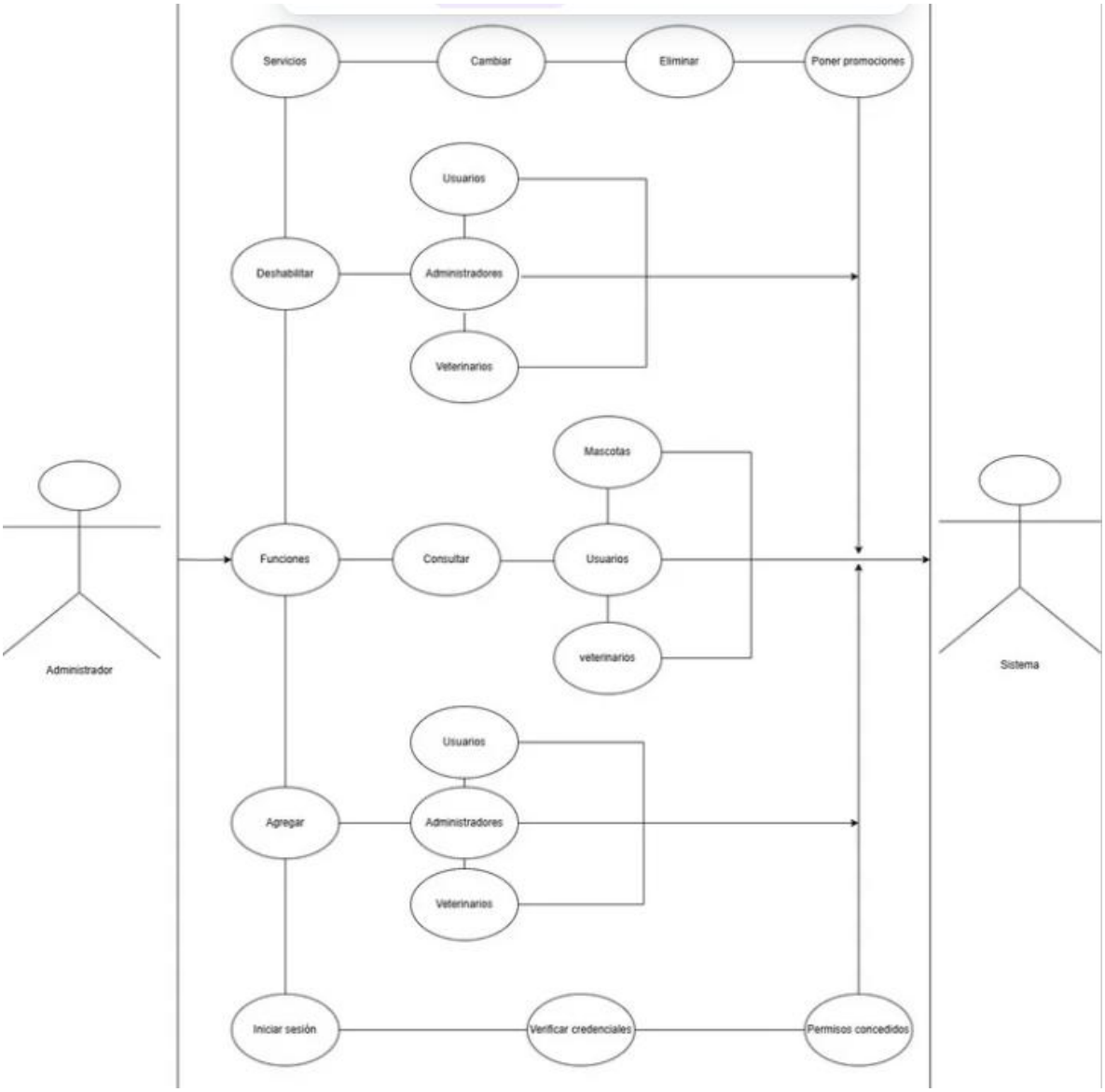
4.1.1 Caso de uso Veterinario



4.1.2 Caso de uso Usuario



4.1.2 Caso de uso Administrador



5. Características de los usuarios

En el desarrollo de Flooky Pets, se han identificado y caracterizado tres tipos principales de usuarios que interactuarán con la plataforma, cada uno con roles, responsabilidades y necesidades específicas. Comprender estas características es crucial para asegurar que el diseño de la interfaz y las funcionalidades del sistema se adapten eficazmente a cada perfil.

5.1 Administrador

Los usuarios con rol de Administrador son el personal encargado de la gestión global y la supervisión del sistema Flooky Pets.

- **Rol Principal:** Gestión y configuración del sistema, mantenimiento de datos maestros, supervisión de operaciones, y gestión de otros usuarios (clientes, veterinarios y otros administradores).
- **Funcionalidades Clave:** Acceso al AdminDashboard, gestión de cuentas de administrador, gestión de servicios, gestión integral de citas (vista global), gestión de usuarios (clientes y veterinarios), y potencial generación de informes.
- **Nivel de Interacción:** Interacción profunda y constante con todas las funcionalidades de backend y configuración. Requiere acceso a datos sensibles y críticos del sistema.
- **Habilidades/Expectativas:** Se espera que los administradores tengan una familiaridad media-alta con sistemas de gestión y herramientas informáticas. Necesitan una interfaz clara, organizada y eficiente que les permita realizar múltiples tareas rápidamente y con precisión, con un fuerte enfoque en la seguridad y la integridad de los datos.

5.2 Veterinario

Los usuarios con rol de Veterinario son los profesionales de la salud animal que utilizan la plataforma para gestionar su agenda y mantener los registros médicos de las mascotas.

- **Rol Principal:** Atención clínica, gestión de sus citas asignadas, creación y actualización de perfiles de mascotas de clientes, y mantenimiento exhaustivo del historial clínico electrónico.
- **Funcionalidades Clave:** Visualización y gestión de sus citas, búsqueda y selección de clientes/mascotas, adición y edición de perfiles de mascotas, registro de diagnósticos, tratamientos, vacunas, y consulta del historial médico detallado.
- **Nivel de Interacción:** Interacción frecuente y directa con las funcionalidades relacionadas con la atención de pacientes. Requieren acceso rápido y seguro a la información clínica y de contacto.
- **Habilidades/Expectativas:** Se espera que los veterinarios sean competentes en el uso de software profesional, aunque no necesariamente expertos en IT. Necesitan una interfaz intuitiva y ergonómica que agilice la entrada de datos durante las consultas y les permita acceder a la información crítica sin interrupciones, priorizando la precisión y la integridad de los datos clínicos.

5.3 Usuario (Dueño de Mascota / Cliente)

Los usuarios con rol de Usuario son los dueños de mascotas que interactúan con la plataforma para gestionar las citas de sus animales y acceder a su información.

- **Rol Principal:** Autogestión de citas, mantenimiento de perfiles de sus mascotas, y consulta del historial clínico de sus animales.

- **Funcionalidades Clave:** Registro y gestión de su cuenta personal, programación, modificación y cancelación de citas, creación y actualización de perfiles de sus mascotas, y visualización de historiales médicos y servicios.
- **Nivel de Interacción:** Interacción asincrónica y autogestionada, generalmente menos frecuente que los roles de administración/veterinario, pero crucial para la experiencia del cliente.

Habilidades/Expectativas: El rango de habilidades técnicas de este grupo es amplio. La interfaz debe ser altamente intuitiva, fácil de usar, visualmente atractiva y accesible desde diferentes dispositivos (responsiva). Necesitan un proceso de agendamiento simplificado, retroalimentación clara y notificaciones oportunas. La confianza en la privacidad y seguridad de sus datos y los de sus mascotas es fundamental.

6. Requerimientos

6.1 Requerimientos funcionales

- Registro de usuarios (dueños de mascotas y personal veterinario).
- Inicio de sesión seguro con recuperación de contraseña.
- Agenda de citas veterinarias con control de disponibilidad.
- Perfil para cada mascota con historial clínico, vacunas y tratamientos.
- Visualización de próximos controles, recordatorios y notificaciones automáticas.
- Módulo de facturación y gestión de pagos (si aplica).
- Área administrativa para la clínica veterinaria (gestión de personal, servicios, clientes).
- Buscador interno para mascotas, citas o servicios.

6.2 Requerimientos no funcionales

- Diseño responsivo adaptable a móviles, tablets y PCs.
- Alta disponibilidad y tolerancia a fallos.
- Interfaz amigable y accesible para usuarios no técnicos.

- Encriptación de datos sensibles (contraseñas, información médica).
- Soporte para múltiples sedes veterinarias.
- Soporte multilenguaje (español e inglés, por ejemplo).
- Documentación técnica y manual de usuario integrados.

---

7. Arquitectura del Proyecto

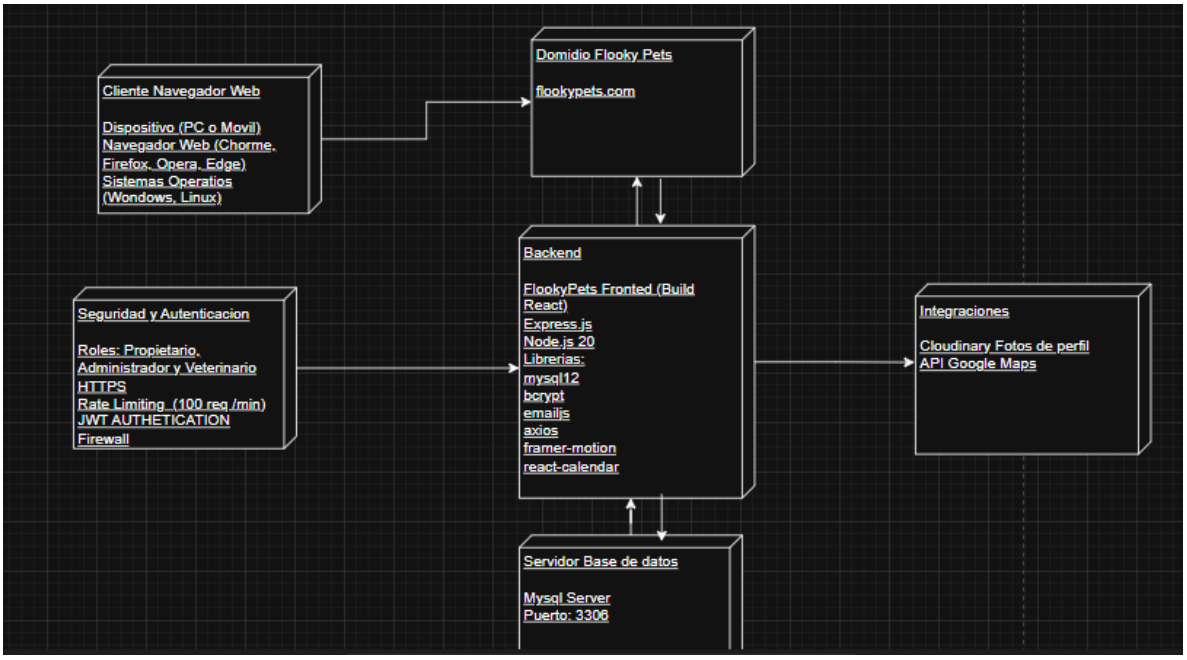
7.1 Definición metodología

SCRUM como marco de trabajo ágil, con ciclos de desarrollo semanales (sprints), reuniones de revisión y entregas funcionales continuas.

7.2 Ciclo de vida

1. Planificación: recopilación de requerimientos y establecimiento del MVP.
2. Diseño: estructura del sistema, prototipado de interfaces y arquitectura técnica.
3. Desarrollo: implementación modular de frontend y backend.
4. Pruebas: validación de funcionalidades, pruebas de carga, usabilidad y seguridad.
5. Despliegue: puesta en producción y monitoreo inicial.
6. Mantenimiento: corrección de errores y evolución continua del sistema.

7.3 Diagrama de despliegue



---

8. Fundamentos y Herramientas Utilizadas

- Lenguajes: JavaScript, HTML5, CSS3.
- Frameworks: React.js (frontend), Express.js (backend).
- Base de datos: MySQL.
- Diseño y UX: Figma.
- Control de versiones: Git + GitHub.
- Gestión de tareas: Trello o Jira.
- APIs externas: dogapi (imagenes), emails (correos)

9. Requisitos del Sistema

Requisitos del servidor

- Sistema operativo Linux (Ubuntu 20.04+).
- Node.js v18 o superior.
- MongoDB 5.0+.
- 2 GB RAM, 2 núcleos CPU, 10+ GB de almacenamiento SSD.

Requisitos del cliente

- Navegador actualizado (Chrome, Firefox, Safari, Edge).
- Conexión estable a internet.
- Resolución mínima recomendada: 1280x720 px.

10.Procesos del Software

10.1 Módulo Administrador

El **Módulo Administrador** es la interfaz central para la gestión y supervisión de todas las operaciones dentro del sistema de la veterinaria. Está diseñado para ofrecer a los administradores una serie de herramientas para controlar usuarios, servicios, citas y registros médicos. Este módulo asegura que solo los usuarios con rol de administrador puedan acceder a estas funcionalidades críticas.

10.1.1 Componente AdminDashboard

El componente AdminDashboard sirve como la vista principal del Módulo Administrador. Orquesta la navegación entre las diferentes subsecciones (Dashboard, Gestión de Usuarios, Servicios, Veterinarios, Administradores, Citas, Historiales Médicos y Configuración) y maneja la lógica inicial de autenticación y carga de estadísticas generales.

Ruta de Acceso: /admin

Manual Técnico

Propósito del Código: Este código define la estructura principal del dashboard de administración. Se encarga de:

- Gestionar el estado de la pestaña activa (activeTab).
- Verificar que el usuario autenticado tenga el rol de admin.
- Cargar estadísticas generales para el dashboard inicial.
- Renderizar el componente (AdminStats, AdminUsers, ServicesManagement, etc.) correspondiente a la pestaña activa.
- Manejar la funcionalidad de cierre de sesión.

Análisis del Código y Partes Importantes:

- **useState para el Manejo de Estado:**
  - activeTab: Controla qué sección del dashboard está visible en un momento
  - isLoading: Un booleano que indica si los datos iniciales del dashboard (estadísticas) están siendo cargados.
  - stats: Almacena los datos de las métricas generales obtenidas de la API para ser mostrados en el AdminStats (Dashboard).

- **useEffect para Efectos Secundarios:**
  - El primer useEffect se ejecuta cuando el componente se monta y cada vez que user, Maps o fetchAdminStats cambian. Su función principal es **validar la autenticación del usuario y su rol**. Si el user no está definido o su role no es admin, redirige al usuario a la página de /login. Después de la validación, llama a fetchAdminStats para obtener las métricas iniciales del dashboard.
  - Este useEffect asegura que solo los administradores autorizados puedan acceder al panel.
- **useCallback para Memoización de Funciones:**
  - fetchAdminStats: Esta función es responsable de realizar la llamada a la API (/admin/stats) para obtener las métricas generales del sistema. Utiliza

authFetch para incluir el token de autenticación. Gestiona el estado isLoading y error durante la petición.

- handleLogout: Permite al administrador cerrar su sesión. Elimina la información del usuario y el token del localStorage y luego redirige al usuario a la página de inicio de sesión (/login).

- **Función renderTabContent():**
  - Es una función clave que determina qué componente hijo se renderizará dentro del área de contenido principal del dashboard, basándose en el valor de activeTab. Esto permite una navegación dinámica y modular dentro del panel de administración.
  - Cada case en el switch corresponde a una pestaña diferente del sidebar y devuelve el componente de gestión respectivo (AdminStats, AdminUsers, ServicesManagement, etc.).
- **Manejo del Estado de Carga:**
  - El componente muestra un mensaje de "Cargando dashboard..." con un spinner (loading-spinner) mientras isLoading es true. Esto mejora la experiencia de usuario al indicar que la aplicación está trabajando.
- **Renderizado Condicional:**
  - La interfaz del dashboard (AdminNavbar, AdminSidebar, main) solo se renderiza una vez que isLoading es false, es decir, una vez que los datos iniciales han sido cargados.
- **Importaciones de Componentes:**
  - Importa diversos componentes (AdminNavbar, AdminSidebar, AdminStats, ServicesManagement, etc.) que representan las diferentes secciones y funcionalidades del módulo administrador. Esto promueve la modularidad y reutilización de código.

- **authFetch:**

- Se importa desde ./api y es crucial para todas las comunicaciones seguras con el backend, ya que automáticamente adjunta el token de autenticación a las solicitudes.

10.1.2 Gestión de Administradores

El componente AdminAdmins se encarga de la administración de las cuentas de usuario con rol de administrador. Permite listar, buscar, añadir, editar y deshabilitar perfiles de administradores, asegurando un control granular sobre quién tiene acceso privilegiado al sistema.

Ruta de Acceso: Dentro del AdminDashboard, seleccionando la pestaña "Administradores".

Propósito del Código: Este código implementa la lógica para:

- Cargar y mostrar una lista de todos los administradores.
- Filtrar administradores por términos de búsqueda.
- Abrir un formulario para crear o editar perfiles de administrador.
- Manejar la lógica de envío de formularios (creación y actualización).
- Procesar la eliminación de administradores.
- Mostrar notificaciones de éxito o error al usuario.

Análisis del Código y Partes Importantes:

- **useState para el Manejo de Estado:**
  - admins: Almacena la lista completa de administradores obtenida de la API.
  - filteredAdmins: Contiene la lista de administradores después de aplicar el filtro de búsqueda.
  - searchTerm: Guarda el texto ingresado por el usuario para la búsqueda.
  - isLoading: Indica si los datos de los administradores están siendo cargados.
  - error: Almacena cualquier mensaje de error que ocurra durante las operaciones.

  - isFormOpen: Controla la visibilidad del formulario modal para añadir/editar.
  - currentAdmin: Si se está editando, guarda el objeto del administrador que se está modificando.
  - formData: Almacena los datos del formulario (nombre, email, contraseña, etc.) tanto para creación como para edición.
  - isSubmitting: Indica si una operación de envío de formulario está en curso.
  - notification: Almacena los detalles de cualquier notificación que deba mostrarse al usuario (mensaje y tipo).
- **getAuthToken():**
  - Una función auxiliar para recuperar el token de autenticación del localStorage, que es necesario para las peticiones seguras a la API.
- **authFetch():**
  - Una función central para todas las comunicaciones con la API, asegurando que cada petición incluya el token de autenticación en los encabezados. Maneja la serialización del cuerpo de la petición a JSON y la verificación de respuestas HTTP exitosas.
- **fetchAdmins() (Función Asíncrona):**
  - **Propósito:** Es la función principal para obtener la lista de administradores del backend.
  - **Funcionamiento:** Realiza una llamada GET a la API (/admin/administrators) usando authFetch. Procesa los datos recibidos y los formatea para ser utilizados en la interfaz, actualizando los estados admins y filteredAdmins.
  - **Manejo de Errores/Carga:** Establece isLoading a true al inicio y a false al finalizar, y captura cualquier error que pueda ocurrir durante la petición.
- **useEffect para Carga Inicial y Filtrado:**
  - El primer useEffect llama a fetchAdmins() cuando el componente se monta, asegurando que la lista de administradores se cargue al inicio.

    - El segundo useEffect se encarga del filtrado en tiempo real. Se ejecuta cada vez que searchTerm o admins cambian, actualizando la lista filteredAdmins basándose en la cadena de búsqueda.

- **showNotification():**
    - Función auxiliar para mostrar mensajes informativos, de éxito o de error al usuario, con un temporizador para que desaparezcan automáticamente.
- **handleDelete(id):**
    - **Propósito:** Elimina un administrador del sistema.
    - **Funcionamiento:** Primero, realiza una validación para **evitar que un administrador se elimine a sí mismo**. Luego, solicita una confirmación al usuario y, si se confirma, realiza una petición DELETE a la API. Si tiene éxito, actualiza el estado admins para reflejar la eliminación.
    - **Seguridad:** Incluye una verificación crucial (id === user.id) para impedir la auto-eliminación, lo que es una buena práctica de seguridad.

- **handleEdit(admin):**
    - **Propósito:** Prepara el formulario para editar un administrador existente.
    - **Funcionamiento:** Establece currentAdmin con los datos del administrador a editar y precarga el formData con sus valores actuales, luego abre el formulario modal (setIsFormOpen(true)).
- **handleAddNew():**
    - **Propósito:** Prepara el formulario para crear un nuevo administrador.
    - **Funcionamiento:** Limpia currentAdmin y resetea formData a sus valores iniciales vacíos, luego abre el formulario modal.
- **handleInputChange(e):**

  - **Propósito:** Actualiza el estado formData a medida que el usuario escribe en los campos del formulario.

- **handleSubmit(e) (Función Asíncrona):**
  - **Propósito:** Procesa el envío del formulario, ya sea para crear un nuevo administrador o para actualizar uno existente.
  - **Validaciones:** Contiene lógica de validación para campos obligatorios (nombre, teléfono, email para nuevos) y para la consistencia de las contraseñas.
  - **Lógica de API:**
    - Si currentAdmin existe, realiza una petición PUT (actualización) a la API.
    - Si currentAdmin es nulo, realiza una petición POST (creación) a la API.
  - **Actualización de Estado:** Después de una operación exitosa, actualiza la lista admins para que la interfaz refleje los cambios y cierra el formulario.
  - **Manejo de Errores/Carga:** Utiliza isSubmitting para deshabilitar el formulario durante el envío y muestra notificaciones de éxito o error.

- **Renderizado Condicional de Interfaz:**
  - Se utiliza isLoading para mostrar un spinner de carga mientras se obtienen los datos.
  - Se muestra un error-message si hay un problema.
  - El formulario modal (modal-overlay) se renderiza solo cuando isFormOpen es true.
  - La tabla o cuadrícula de administradores (admins-grid) se muestra solo si filteredAdmins tiene elementos; de lo contrario, se muestra un mensaje de "No results".
- **Interfaz de Usuario (JSX):**

- Utiliza iconos (FaUserShield, FaPlus, FaSearch, FaTimes, FaSave, FaSpinner) para mejorar la usabilidad.
- El campo de búsqueda permite una experiencia de filtrado interactiva.
- Las tarjetas de administrador (admin-card) muestran información clave y botones de acción (Editar, Eliminar).
- El formulario modal proporciona campos para ingresar y editar los datos del administrador.

10.1.3 Gestión de Servicios

El componente AdminServices permite a los administradores gestionar el catálogo de servicios ofrecidos por la veterinaria. Desde aquí, se pueden visualizar, buscar, añadir, modificar y eliminar servicios, asegurando que la información de los servicios esté siempre actualizada para los clientes.

Ruta de Acceso: Dentro del AdminDashboard, seleccionando la pestaña "Servicios".

Propósito del Código: Este código implementa la lógica para:

- Cargar y mostrar una lista de servicios predefinidos (o simulados para demostración).
- Permitir la búsqueda y filtrado de servicios.
- Navegar a una vista para añadir un nuevo servicio.
- Navegar a una vista para editar un servicio existente.
- Manejar la eliminación de servicios.
- Formatear la visualización de precios y duraciones.
- Mostrar notificaciones al usuario.

Análisis del Código y Partes Importantes:

- **useState para el Manejo de Estado:**
  - services: Almacena la lista completa de servicios.

- filteredServices: Contiene la lista de servicios después de aplicar el filtro de búsqueda.
- searchTerm: Guarda el texto ingresado por el usuario para la búsqueda.
- isLoading: Indica si los datos de los servicios están siendo cargados.
- error: Almacena cualquier mensaje de error.
- notification: Almacena los detalles de la notificación (mensaje y tipo).

- **useEffect para Carga Inicial y Filtrado:**
  - El primer useEffect simula la carga de datos de servicios. Aunque en un entorno real se haría una llamada a una API (fetchServices simula esta llamada con un setTimeout), este useEffect se encarga de poblar services y filteredServices al inicio y establecer isLoading a false.
  - El segundo useEffect se encarga del filtrado en tiempo real. Se ejecuta cada vez que searchTerm o services cambian, actualizando la lista filteredServices basándose en el nombre, descripción o categoría del servicio.

- **showNotification():**
  - Una función auxiliar para mostrar mensajes temporales de éxito o error al usuario.
- **handleEdit(service):**
  - **Propósito:** Inicia el proceso de edición de un servicio.
  - **Funcionamiento:** Utiliza Maps de react-router-dom para redirigir al usuario a una ruta específica (/admin/services/edit/:id), pasando el ID del servicio a editar. Esto implica que la lógica de edición real se manejaría en un componente separado en esa ruta.
- **handleDelete(id):**
  - **Propósito:** Elimina un servicio de la lista.

- **Funcionamiento:** Solicita una confirmación al usuario. Si se confirma, actualiza el estado services eliminando el servicio con el id correspondiente y muestra una notificación de éxito. En una aplicación real, esta acción enviaría una petición DELETE a la API.

- **handleAddNew():**
  - **Propósito:** Inicia el proceso para agregar un nuevo servicio.
  - **Funcionamiento:** Redirige al usuario a la ruta '/admin/services/new', donde se esperaría un formulario para la creación de nuevos servicios.
- **formatPrice(price) y formatDuration(duration):**
  - Funciones utilitarias para dar formato legible a los datos numéricos de precio y duración, mostrando "Consultar" o "Variable" si el valor es cero, y formateando el precio como moneda colombiana (COP).
- **Manejo del Estado de Carga y Errores:**
  - El componente renderiza una animación de carga (loading-container) mientras isLoading es true.
  - Si se detecta un error, muestra un mensaje de error dedicado.
- **Animaciones con Framer Motion:**
  - Se utilizan componentes y utilidades de framer-motion (motion.div, AnimatePresence) para añadir animaciones fluidas a la carga de datos, la aparición de notificaciones y la interacción con las tarjetas de servicio (e.g., efectos whileHover). Esto mejora la experiencia visual del usuario.
- **Interfaz de Usuario (JSX):**
  - El encabezado (dashboard-header) incluye el título de la sección y la barra de búsqueda.

- La sección de búsqueda permite filtrar servicios por texto.
- El botón "Nuevo Servicio" (add-btn) redirige a la página de creación.
- Los servicios se muestran en una cuadrícula de tarjetas (services-grid), cada una mostrando nombre, categoría, descripción, precio y duración.
- Cada tarjeta incluye botones para "Editar" (edit-btn, FaEdit) y "Eliminar" (delete-btn, FaTrash) servicios.
- Si no hay servicios que coincidan con la búsqueda, se muestra un mensaje de "No se encontraron servicios".

10.1.4 Gestión de Citas

El componente AdminAppointments proporciona a los administradores una herramienta integral para la gestión de citas en la veterinaria. Permite una visión general de todas las citas, con capacidades de búsqueda, filtrado por estado y visualización detallada.

Propósito del Componente: Este componente implementa la lógica para:

- Cargar y mostrar citas desde el backend, con la capacidad de filtrar por estado.
- Permitir la búsqueda de citas por dueño, servicio o ubicación/detalle.
- Gestionar el formulario modal para añadir nuevas citas o editar existentes.
- Realizar operaciones CRUD (Leer) sobre las citas.
- Visualizar detalles completos de una cita en un modal separado.
- Cargar dinámicamente listas de servicios, clientes y veterinarios para los campos del formulario.
- Mostrar notificaciones y estados de carga/error para una mejor experiencia de usuario.

Análisis Técnico del Componente y Partes Importantes:

- **Gestión de Estado (useState):**
  - appointments: Almacena la lista completa de citas obtenidas del backend.

- filteredAppointments: Contiene la lista de citas después de aplicar el filtro de búsqueda local.
- searchTerm: El texto de búsqueda ingresado por el usuario.
- isLoading: Booleano que indica si los datos están siendo cargados.
- error: Mensaje de error, si lo hay.
- filter: La categoría de estado actual por la que se filtran las citas (all, pendiente, aceptada, etc.). Este filtro se envía al backend.
- notification: Objeto que contiene el mensaje y tipo de notificación a mostrar.
- isViewModalOpen: Controla la visibilidad del modal de "Ver Detalles".
- viewingAppointment: Almacena los datos de la cita que se está visualizando.
- formData: Objeto que contiene los datos del formulario de cita (fecha, estado, ubicación, IDs de servicio, cliente y veterinario). Es crucial notar que ubicacion en el frontend se mapea a ubicacion_servicio en el backend para el payload.
- services, clients, vets: Listas de datos para poblar los select de los formularios, obtenidas de la API.

- **Efectos Secundarios y Carga de Datos (useEffect y useCallback):**
  - **getMinDateTime():** Una función utilitaria que calcula la fecha y hora actual más un minuto. Esto se usa para establecer el atributo min del input de fecha y hora, previniendo que se seleccionen fechas en el pasado, lo cual es una validación crítica para las citas.
  - **showNotification():** Función memoizada (useCallback) para mostrar y ocultar notificaciones temporales al usuario. Mejora la retroalimentación visual.
  - **fetchDropdownData():** Función asíncrona (useCallback) que realiza llamadas simultáneas a la API (/servicios, /admin/usuarios, /usuarios/veterinarios) para obtener los datos necesarios para los

dropdowns (listas desplegables) del formulario de cita. Maneja los errores de carga.

- **fetchAppointments():** Función asíncrona (useCallback) para cargar las citas desde el backend. Es clave que esta función ahora construye la URL de la API dinámicamente (/admin/citas?status=${filter}) para enviar el filtro de estado directamente al servidor, permitiendo que el backend se encargue del filtrado eficiente. Formatea los datos recibidos, mapeando el id_cita a id y el campo ubicacion_servicio del backend a ubicacion en el frontend para la visualización. Gestiona los estados isLoading y error.
- **Primer useEffect para Carga de Datos:** Se ejecuta al montar el componente y cuando el usuario (user) cambia. Su propósito es inicializar la carga de citas (fetchAppointments) y los datos de los dropdowns (fetchDropdownData) si el usuario está autenticado. Si no, establece un mensaje de error de autorización.
- **Segundo useEffect para Filtrado Local:** Este useEffect se encarga de filtrar las appointments (que ya vienen pre-filtradas por estado desde el servidor) basándose en el searchTerm local. Esto permite una búsqueda de texto sobre los resultados ya obtenidos.

- **Manejo de Formularios y Operaciones CRUD (useCallback):**
  - **handleEdit(appointment):** Carga los datos de una cita existente en formData y abre el modal para edición. Es importante cómo mapea appointment.fecha a formData.fecha y appointment.ubicacion_servicio a formData.ubicacion.
  - **handleViewDetails(appointmentId):** Una función dedicada para obtener y mostrar detalles completos de una cita específica en un modal aparte. Realiza una llamada a la API (/citas/:id) para asegurar que se obtengan los datos más recientes y completos de esa cita.
  - **handleFormChange():** Una función genérica para actualizar el estado formData a medida que los campos del formulario son modificados.
  - **handleSubmit():** Maneja el envío del formulario de adición/edición. Incluye una validación crítica para asegurar que la fecha seleccionada no sea en el pasado, mejorando la lógica de negocio. Realiza una transformación de

datos crucial: formData.ubicacion se convierte en payload.ubicacion_servicio antes de enviar la petición a la API. También parsea los IDs a enteros y maneja el caso de id_veterinario nulo. Realiza peticiones PUT para editar y POST para crear, y luego refresca la lista de citas (fetchAppointments).

- **handleDelete(id):** Confirma la eliminación con el usuario y luego realiza una petición DELETE a la API para eliminar la cita. Refresca la lista si es exitoso.
- **handleChangeStatus(id, currentStatus, targetStatus):** Una funcionalidad clave que permite a los administradores cambiar el estado de una cita (pendiente, aceptada, rechazada, completa, cancelada). Solicita confirmación y envía una petición PUT con el nuevo estado. Los botones para estas acciones se muestran condicionalmente en la tabla.

- **Estilado y Componentes de Soporte:**
  - **getStatusBadge(status):** Una función utilitaria para asignar clases CSS (badge-success, badge-warning, etc.) a los *badges* de estado, lo que permite una representación visual clara de cada estado.
  - **Renderizado Condicional:** Muestra *spinners* de carga y mensajes de error detallados cuando los datos se están cargando o si hay problemas de conexión/datos. Los modales de añadir/editar y ver detalles se controlan con isModalOpen e isViewModalOpen respectivamente.
  - **Interfaz de Usuario (JSX):**
    - **Notificaciones:** Integración de AnimatePresence y motion.div para notificaciones animadas.
    - **Encabezado:** Título, barra de búsqueda con FaSearch, y botones de filtro por estado que se comunican con setFilter.
    - **Tabla de Citas:** Muestra ID, Fecha, Dueño, Detalle de Ubicación/Servicio (nombre nuevo para app.ubicacion) y Estado. Cada fila tiene botones de acción.
    - **Botones de Acción:**
      - handleViewDetails: Con FaEye para ver detalles.

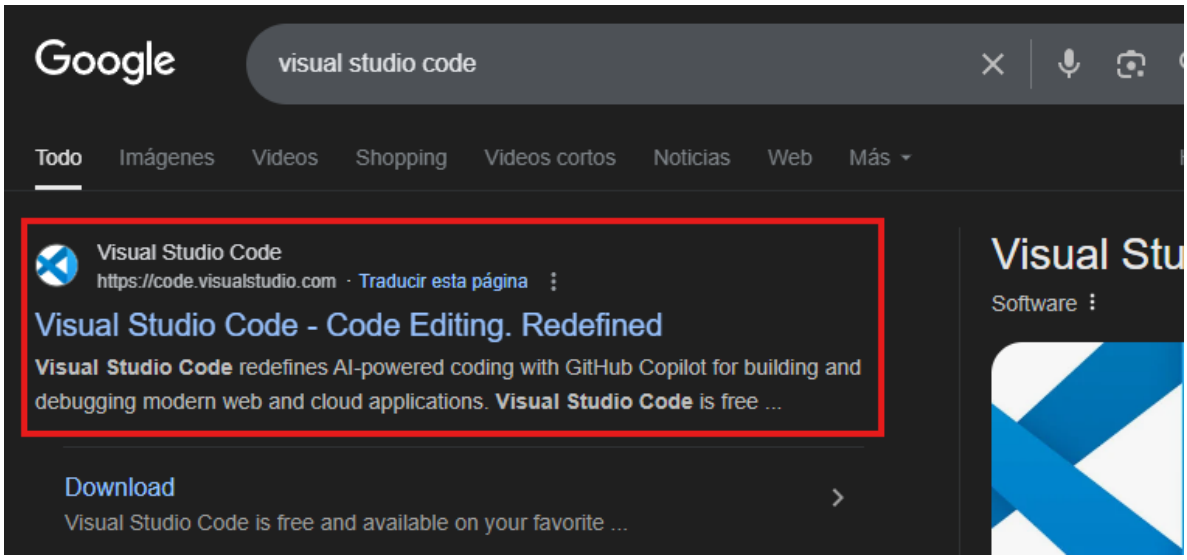
- handleEdit: Con FaEdit para editar.
- **Cambio de Estado:** Botones con FaRegCheckCircle (Aceptar), FaRegTimesCircle (Rechazar), FaCalendarCheck (Completar), FaTimesCircle (Cancelar) que aparecen condicionalmente según el estado actual de la cita, permitiendo transiciones lógicas.
- handleDelete: Con FaTrash para eliminar.

11: instalación de aplicación

Requisitos previos

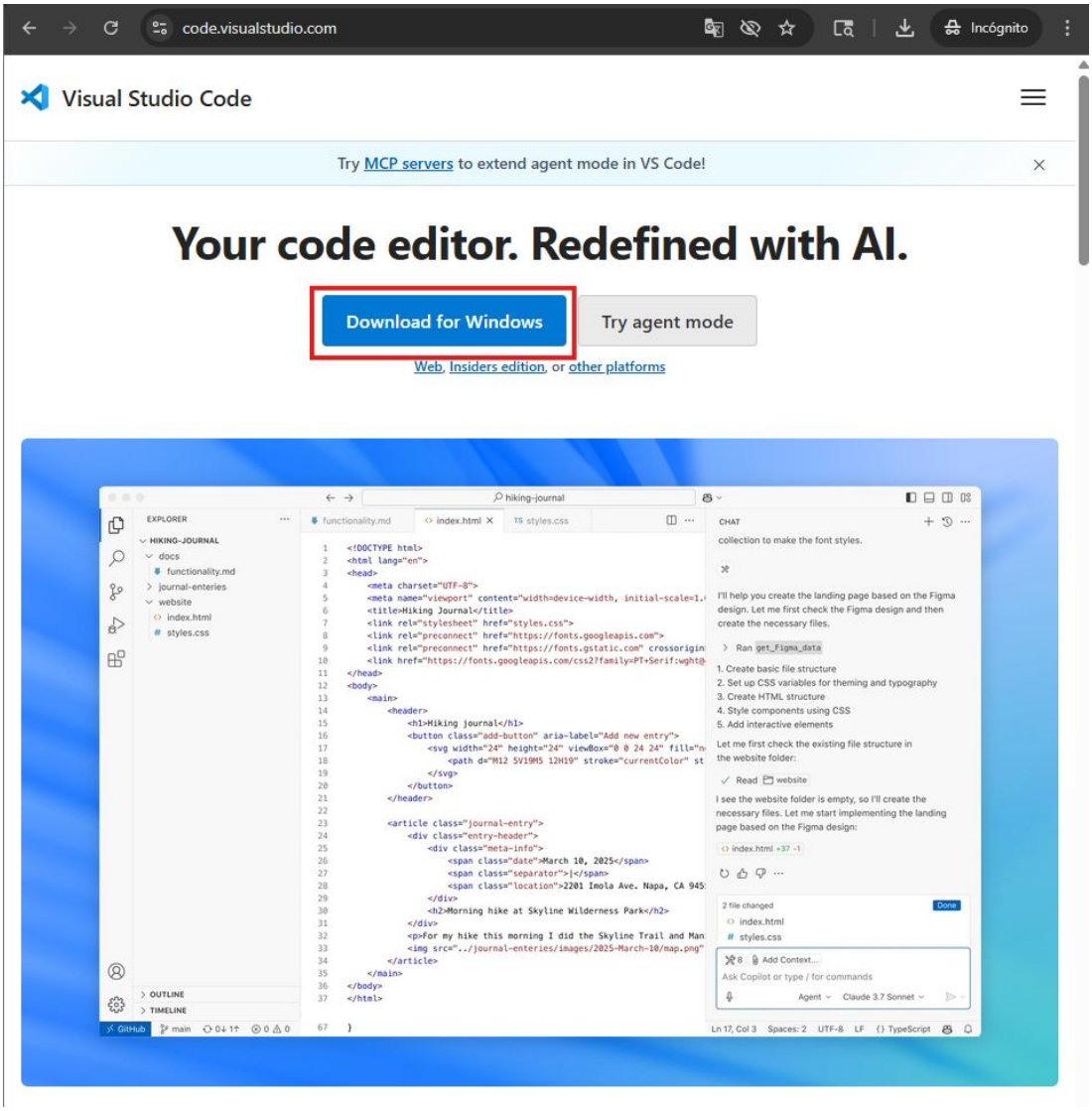
- Node.js v21 o superior
- MySQL
- Git

Guía de instalación (Visual Code)



Entran a su buscador de preferencia, Chrome es el ejemplo en este caso , ingresan al primer vinculo que les aparezca.

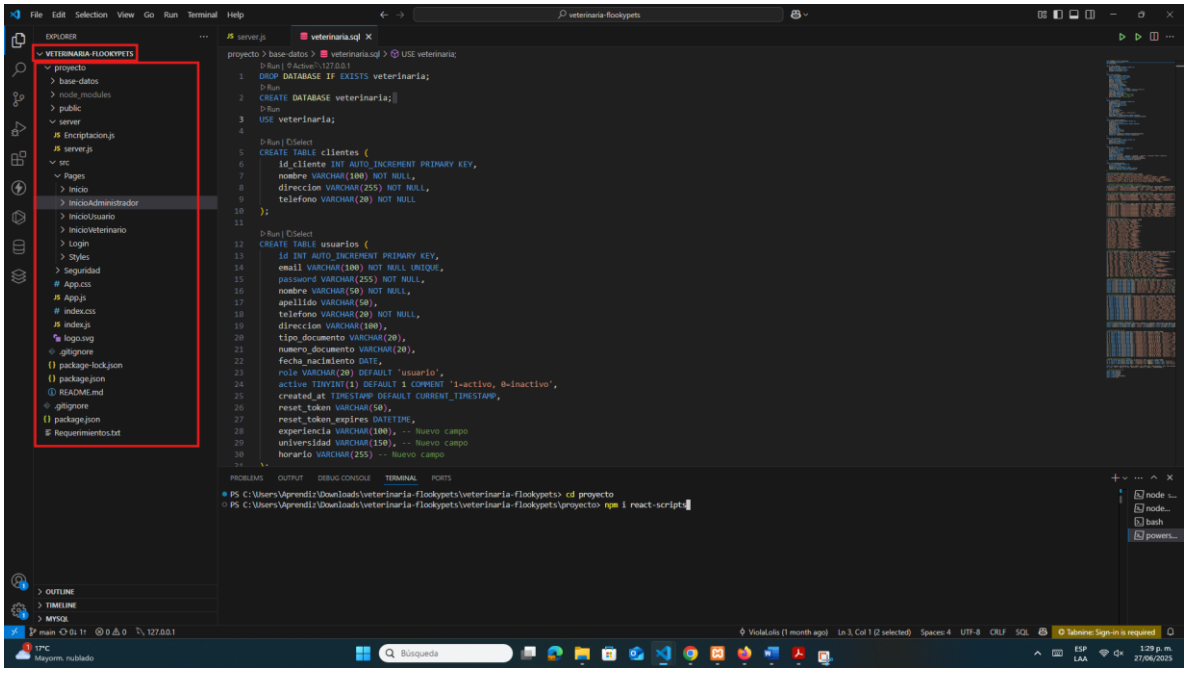
Manual Técnico



Le damos en la opción de “Download for Windows” para empezar la instalación.

Aquí les saldrá la versión , lo guardan y comienza la instalación.

Manual Técnico



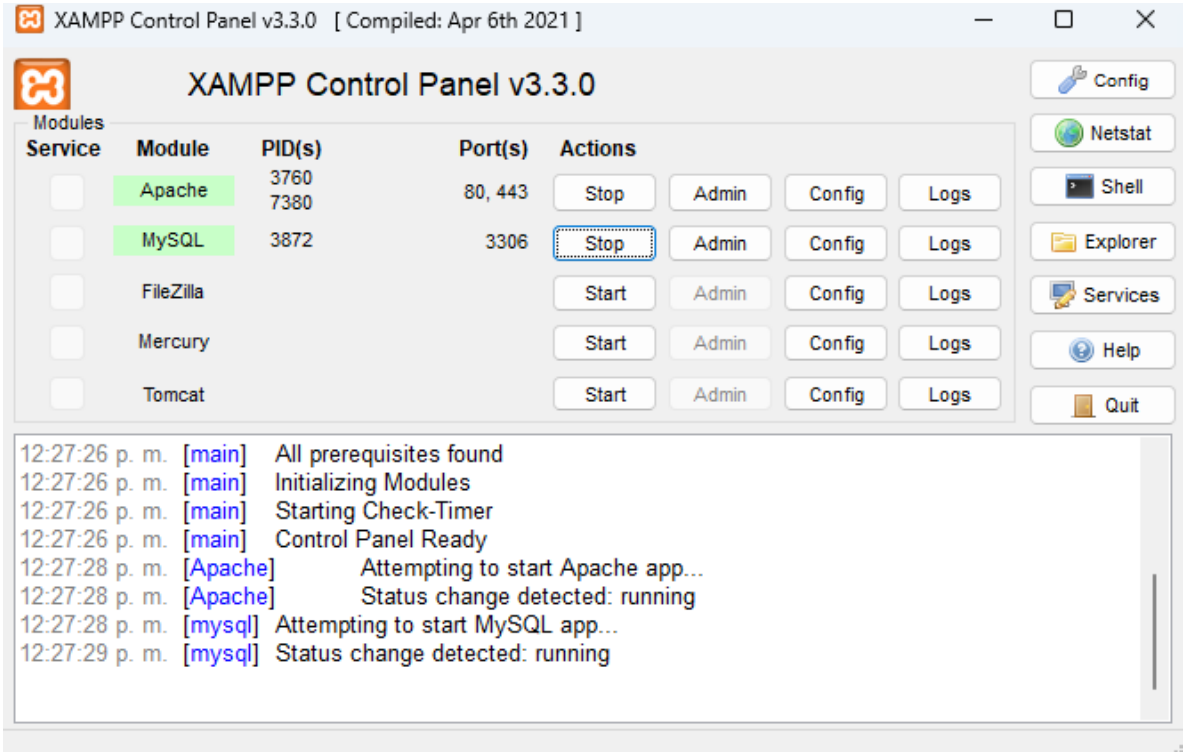
1) “VETERINARIA-FLOOKPETS” es el archivo de la carpeta donde esta metido todo nuestro código fuente

2) “Proyecto” es el archivo principal donde esta todo dividido por secciones para mayor control

```
PS C:\Users\Aprendiz\Downloads\veterinaria-flookypets\veterinaria-flookypets> cd proyecto
PS C:\Users\Aprendiz\Downloads\veterinaria-flookypets\veterinaria-flookypets\proyecto> cd server
```

Usamos el comando “cd” para entrar a nuestra carpeta “server” para hacer la conexión a la base de datos

Manual Técnico

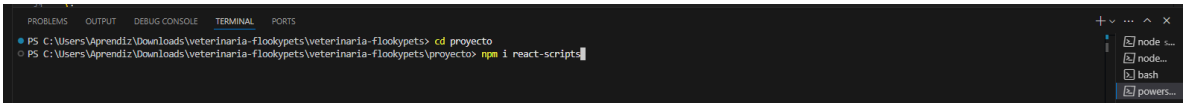


antes de conectar la base de datos asegúrate que tengas bien el XAMP con Apache y MySQL

```
PS C:\Users\Aprendiz\Downloads\veterinaria-flookypets\veterinaria-flookypets\proyecto\server> node server
Servidor corriendo en el puerto 5000
Conectado a la base de datos MySQL
```

Ya después de asegurarte pones el comando “node server” para hacer la conexión a la base de datos , ten en cuenta que debes tener el node en una versión superior a 21.

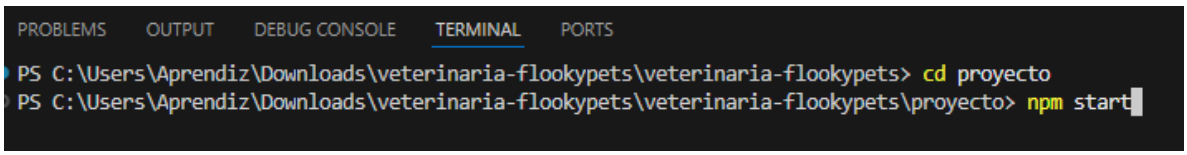
Creamos una terminal diferente a la de la conexión de la base de datos



El comando “Cd” lo usamos para entrar a la carpeta de “proyecto” donde iniciaremos toda la pagina web.

El comando “npm i react-scripts” lo usamos para descargar unos archivos necesarios para ejecutar, en caso de omitir algún paso causara error y no lo dejara visualizar.

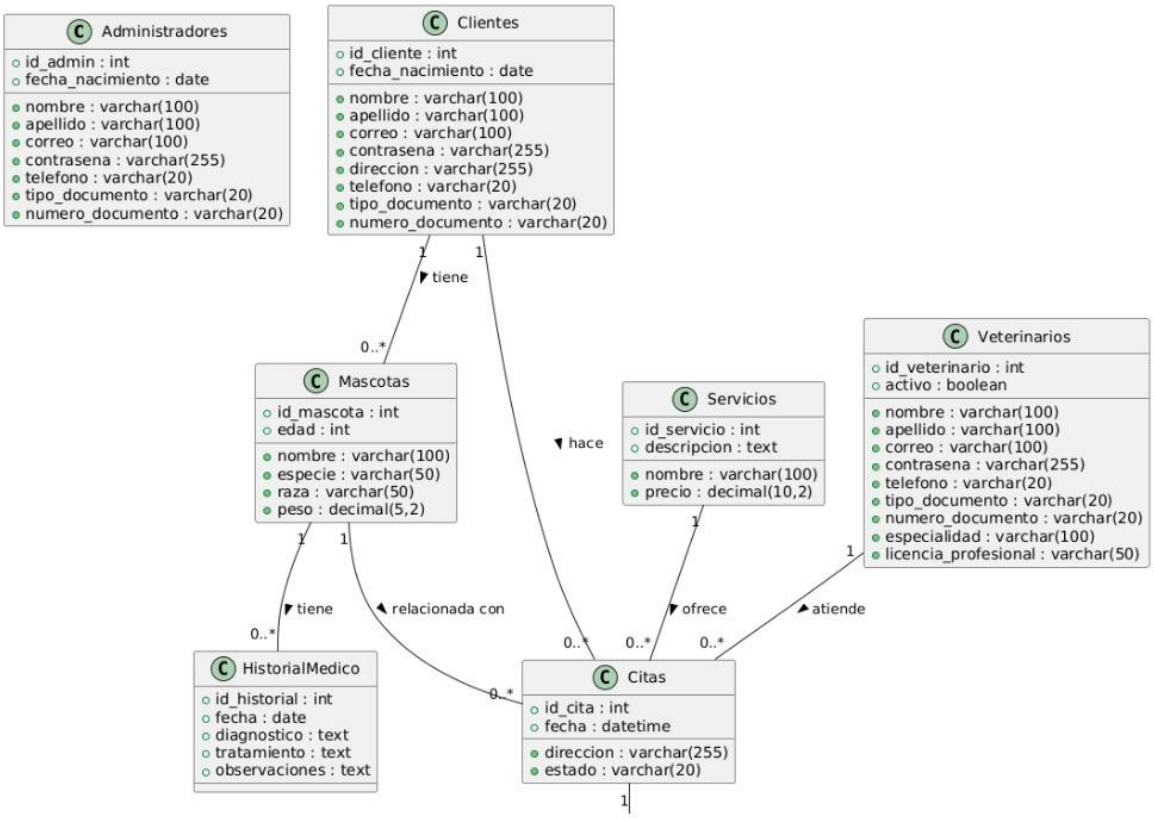
Ya descargando los Scripts necesarios, saldrá de esta forma donde ya tenemos los paquetes necesarios.



Para comenzar el despliegue , entramos a la carpeta de “proyecto” y “npm start” para iniciar a visualizar en el navegador.

12. Diagrama de Clases

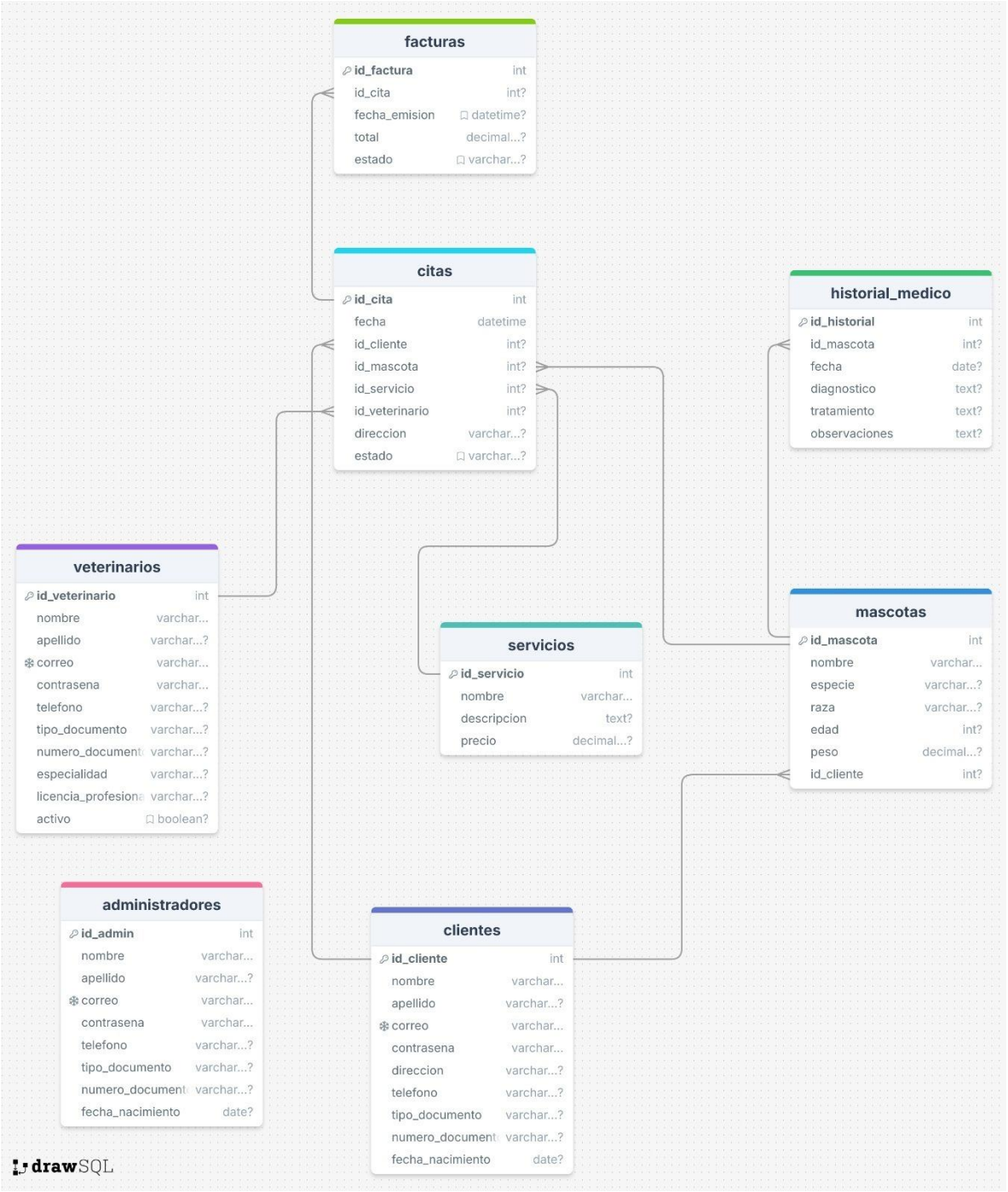
Manual Técnico



13: modelo entidad relación

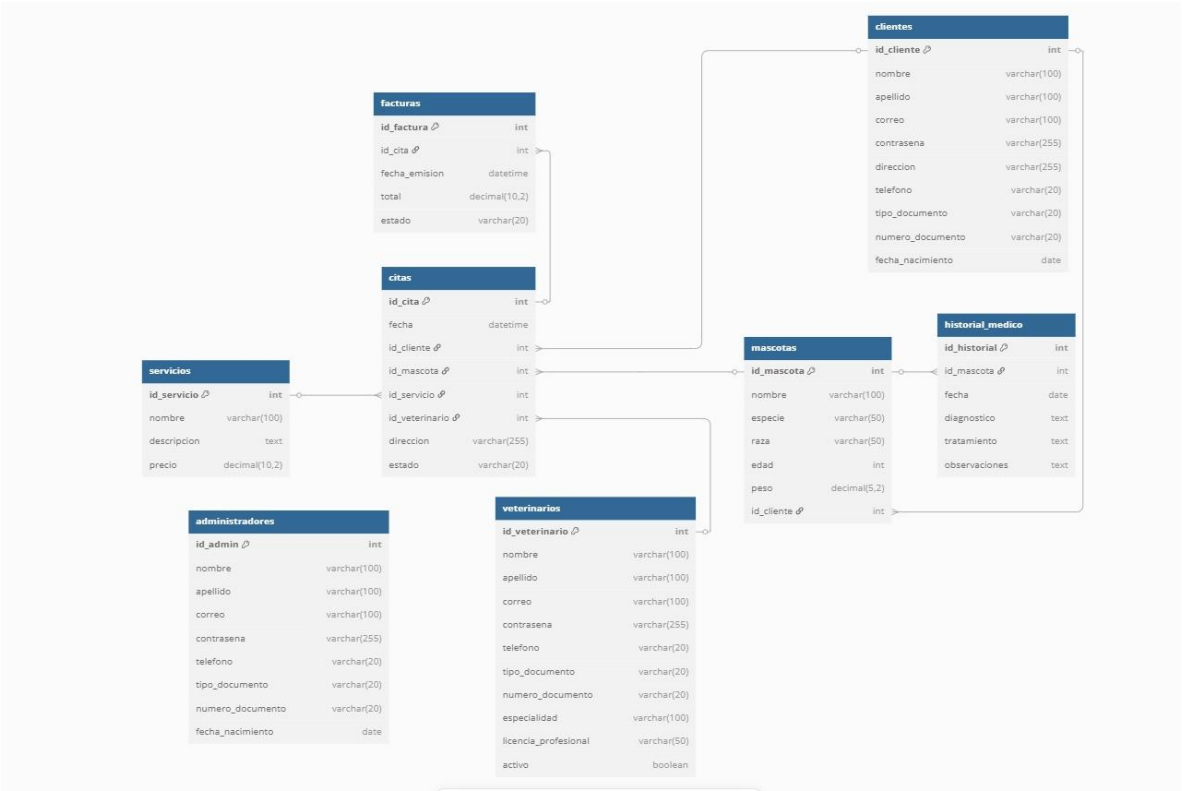
(DER)

Manual Técnico

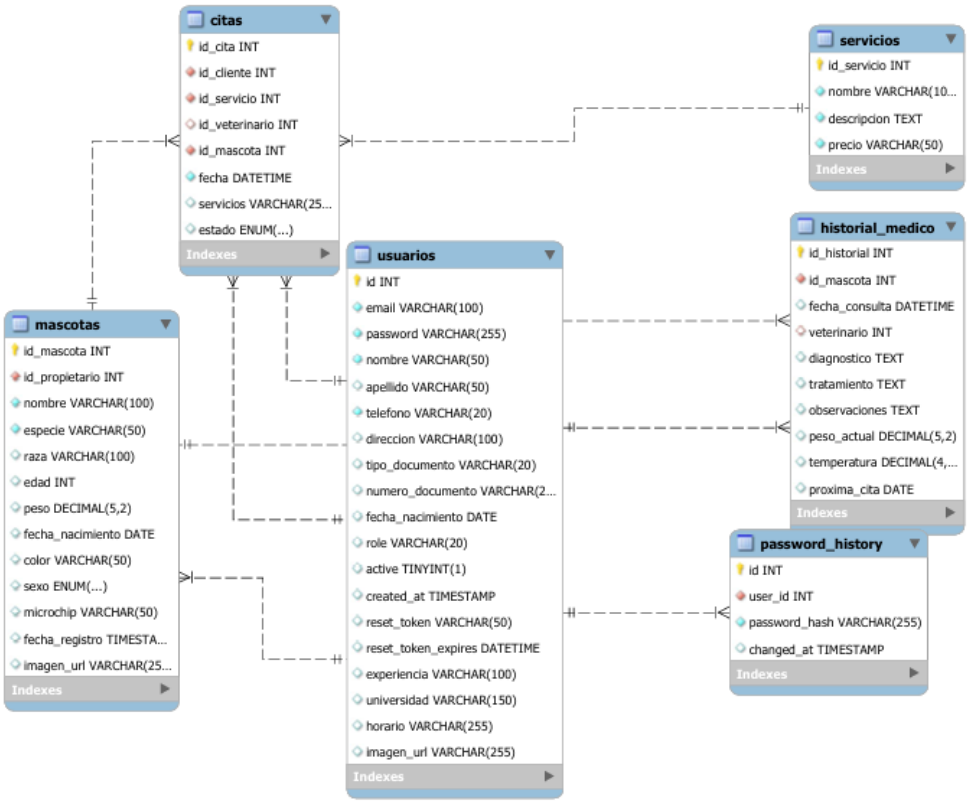


14. Diagrama entidad relación

(MER)



15. Diccionario de Datos



16. Funcionalidad (código)

Login

```
48  const handleLogin = async (e) => {
49    e.preventDefault();
50
```

Línea 48–49: Declara una función asincrónica llamada handleLogin, que maneja el envío del formulario. Se recibe el evento e, y e.preventDefault() evita que la página se recargue al hacer submit.


```
51    setTouched({
52      email: true,
53      password: true
54    });
```

Línea 51–54: Marca ambos campos (email y password) como "tocados" (touched: true), lo cual sirve para activar la validación visual en pantalla.

```
55
56    const hasErrors = Object.entries(errors)
57      .filter(([key]) => key !== 'form')
58      .some(([, error]) => error !== '');
```

Línea 56–58: Verifica si hay errores en los campos email o password. Excluye el error del formulario (form) y devuelve true si hay al menos un error activo.

```
60    if (!email || !password || hasErrors) {
61      setErrors(prev => ({
62        ...prev,
63        form: '⚠ Correo o contraseña son incorrectas ⚠'
64      }));
65      return;
66    }
```

Línea 60–66: Si el usuario no completó el email o la contraseña, o hay errores, se muestra un mensaje de error general (form) y se detiene la ejecución de la función con return.

```
68    try {
69      const response = await fetch('http://localhost:5000/login', {
70        method: 'POST',
71        headers: {
72          'Content-Type': 'application/json'
73        },
74        body: JSON.stringify({ email, password })
75      });
```

Línea 68–75: Inicia un bloque try/catch para manejar errores. Hace una petición POST al backend (/login) con el email y password en el cuerpo de la solicitud, codificados en JSON.

```
77      const data = await response.json();
```

Línea 77: Convierte la respuesta del servidor en un objeto JavaScript (se espera que sea un JSON con datos del usuario o un mensaje de error).

```
79      if (!response.ok) {
80        setErrors(prev => ({
81          ...prev,
82          form: data.message || '⚠ Error al iniciar sesión ⚠'
83        }));
```

Línea 79–83: Si el response.ok no es verdadero (es decir, hubo error HTTP como 400 o 401), se muestra un mensaje de error proporcionado por el backend o uno genérico.

```
88      setUser(data);
```

Línea 88: Llama a setUser (una función pasada por props) para guardar los datos del usuario en el estado global (probablemente el contexto o estado principal del app).

```
90      localStorage.setItem('token', data.token); // <-- ¡Guardando el token!
91      localStorage.setItem('user', JSON.stringify(data));
```

Línea 90–91: Guarda el token de autenticación y el objeto completo del usuario en el localStorage, para mantener la sesión activa entre recargas o visitas.

```
95      if (data.role === 'admin') {
96        navigate('/admin');
97      } else if (data.role === 'veterinario') {
98        navigate('/veterinario');
99      } else {
100       navigate('/usuario');
101     }
```

Línea 95–101: Según el rol recibido en la respuesta (admin, veterinario, usuario), redirige al usuario a su respectiva interfaz usando navigate.


```
103     } catch (error) {
104       console.error('Error al conectar con el servidor:', error);
105       setErrors(prev => ({
106         ...prev,
107         form: '⚠ Error al conectar con el servidor ⚠'
108       }));
109     }
```

Línea 103–109: Si ocurre un error inesperado (por ejemplo, si el servidor está apagado), se captura y se muestra un mensaje de error al usuario.

Olvide contraseña

Generación del código

```
52    const generarCodigo = () => {
53      const nuevoCodigo = Math.floor(100000 + Math.random() * 900000).toString();
54      setCodigoGenerado(nuevoCodigo);
55      return nuevoCodigo;
56    };
```

Crea un código de 6 dígitos aleatorio (123456, 938221, etc.).

Lo guarda en el estado codigoGenerado.

4. Envío del código al correo

```
58  const enviarCodigoPorCorreo = async (codigo, email) => {
59    const templateParams = {
60      to_email: email,
61      from_name: 'Flooky Pets',
62      reply_to: 'soporte@flookypets.com',
63      verification_code: codigo,
64      user_name: email.split('@')[0]
65    };
66
67    try {
68      await send(serviceId, templateId, templateParams, publicKey);
69      setCodigoEnviado(true);
70      setTiempoRestante(60);
71      setError('');
72      setSuccessMessage('Código de verificación enviado a tu correo');
73      return true;
74    } catch (err) {
75      console.error('Error al enviar correo:', err);
76      let errorMsg = 'Error al enviar el código. Intenta nuevamente.';
```

Usa **EmailJS** para enviar el código.

- Si todo sale bien, inicia el temporizador y cambia el estado a codigoEnviado: true.


```
99
100     const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
101     if (!emailRegex.test(email)) {
102       throw new Error('Correo electrónico inválido');
103     }
104
105     const response = await fetch('http://localhost:5000/forgot-password', {
106       method: 'POST',
107       headers: {
108         'Content-Type': 'application/json',
109       },
110       body: JSON.stringify({ email })
111     });
112
113     const data = await response.json();
114
115     if (!response.ok) {
116       throw new Error(data.message || 'Error al verificar el correo');
117     }
```

Explicación: Se define una expresión regular para verificar que el formato del correo sea válido. Esta expresión asegura que:

El correo no contenga espacios en blanco.

Incluya un símbolo @ y un punto (.) como separador entre el nombre del usuario y el dominio.

Si el correo no pasa la validación, se lanza un error con el mensaje "Correo electrónico inválido".

Envío de la solicitud al backend

```
js
CopiarEditar
const response = await fetch('http://localhost:5000/forgot-password', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({ email })
});
```

- **Explicación:** Se realiza una solicitud POST al endpoint /forgot-password de un servidor local (en el puerto 5000).
- Se especifica que los datos se enviarán en formato JSON.

- El cuerpo de la solicitud incluye el correo electrónico del usuario, convertido en cadena JSON.

Procesamiento de la respuesta del servidor

```
js
CopiarEditar
const data = await response.json();
if (!response.ok) {
  throw new Error(data.message || 'Error al verificar el correo');
}
```

Explicación: Se convierte la respuesta del servidor en un objeto JSON.

Luego se verifica si la solicitud fue exitosa (response.ok).

Si no lo fue, se lanza un error con el mensaje devuelto por el servidor (data.message), o con un mensaje genérico en caso de que no exista.

```
123       const codigo = data.resetToken || generarCodigo();
124       // 👇 Línea agregada para mostrar el código en consola (solo para pruebas)
125       console.log( `Codigo de verificación para ${email}: ${codigo}`);
126       console.log(`🔐 Código de verificación para ${email}: ${codigo}`);
127       // 👇 Línea agregada para mostrar el código en consola (solo para pruebas) HEAD
128       console.log( `Codigo de verificación para ${email}: ${codigo}`);
129
130       console.log(`🔐 Código de verificación para ${email}: ${codigo}`);
131
132
133       const emailEnviado = await enviarCodigoPorCorreo(codigo, email);
134
135       if (emailEnviado) {
136         setCodigoGenerado(codigo);
137       }
138
139     } catch (error) {
140       setError(error.message);
141     } finally {
142       setIsSubmitting(false);
143     }
```

Mensajes de depuración en consola

```
js
CopiarEditar
console.log(`Código de verificación para ${email}: ${codigo}`);
```

- **Explicación:** Estas líneas imprimen el código generado en la consola junto con el correo electrónico.
- **Nota:** Hay varias versiones de la misma línea, probablemente utilizadas durante pruebas locales o para realizar seguimiento en diferentes entornos de desarrollo.
- Este paso **no debería mantenerse en producción**, ya que revela información sensible (código de verificación) en la consola del servidor.

Envío del código por correo

```
js
CopiarEditar
const emailEnviado = await enviarCodigoPorCorreo(codigo, email);
```

- **Explicación:** Se llama a la función `enviarCodigoPorCorreo`, que probablemente se encarga de enviar el correo al usuario con el código generado.
- El resultado es almacenado en `emailEnviado`, que indica si el envío fue exitoso.

Almacenamiento del código si el envío fue exitoso

```
js
CopiarEditar
if (emailEnviado) {
  setCodigoGenerado(codigo);
}
```

- **Explicación:** Si el correo se envió correctamente, se actualiza el estado con el código generado utilizando `setCodigoGenerado(codigo)`.
- Esto permite que el componente o sistema recuerde el código para futuras verificaciones (por ejemplo, cuando el usuario lo ingrese para validar su identidad).

Manejo de errores y finalización del proceso

```
js
CopiarEditar
} catch (error) {
  setError(error.message);
} finally {
  setIsSubmitting(false);
}
```

- **Explicación:**
  - Si ocurre un error durante el proceso (por ejemplo, fallo en el envío del correo), se captura con `catch` y se muestra el mensaje de error usando `setError`.

- Finalmente, se indica que el proceso ha terminado (por ejemplo, desactivando un spinner de carga) con setIsSubmitting(false).

Registro

```
11  function Registro() {
53      // Limpiar el temporizador al desmontar el comp
54      useEffect(() => {
55          return () => {
56              if (buttonTimerRef.current) {
57                  clearTimeout(buttonTimerRef.current);
58              }
59          };
60      }, []);
61
62      useEffect(() => {
63          let timer;
64          if (codigoEnviado && tiempoRestante > 0 && !codigoVerificado) {
65              timer = setInterval(() => {
66                  setTiempoRestante(prevTime => prevTime - 1);
67              }, 1000);
68          } else if (!codigoEnviado || tiempoRestante === 0 || codigoVerificado) {
69              clearInterval(timer);
70          }
71          return () => clearInterval(timer);
72      }, [codigoEnviado, tiempoRestante, codigoVerificado]);
73
74      useEffect(() => {
75          if (registroExitoso) {
76              const timer = setTimeout(() => {
77                  navigate('/login');
78              }, 3000);
79              return () => clearTimeout(timer);
```

Buscar Aa ab .* No hay resultados

1. Limpieza del temporizador al desmontar el componente

```
js
CopiarEditar
useEffect(() => {
  return () => {
    if (buttonTimerRef.current) {
      clearTimeout(buttonTimerRef.current);
    }
  };
}, []);
```

- **Finalidad:** Este efecto se ejecuta una única vez al montar el componente y su función de retorno se ejecuta al desmontar.
- **Funcionalidad:** Si existe un temporizador activo (buttonTimerRef.current), lo limpia usando clearTimeout para evitar fugas de memoria o comportamiento inesperado si el componente se desmonta antes de que termine el temporizador.

---

2. Temporizador de cuenta regresiva del código de verificación

```
js
CopiarEditar
useEffect(() => {
  let timer;
  if (codigoEnviado && tiempoRestante > 0 && !codigoVerificado) {
    timer = setInterval(() => {
      setTiempoRestante(prevTime => prevTime - 1);
    }, 1000);
  } else if (!codigoEnviado || tiempoRestante === 0 || codigoVerificado) {
    clearInterval(timer);
  }

  return () => clearInterval(timer);
}, [codigoEnviado, tiempoRestante, codigoVerificado]);
```

- **Finalidad:** Controlar una cuenta regresiva de verificación que se ejecuta cada segundo.
- **Funcionamiento:**
  - Si el código fue enviado (codigoEnviado) y todavía hay tiempo (tiempoRestante > 0) y no se ha verificado el código (!codigoVerificado), se inicia un setInterval que reduce el tiempo restante cada segundo.
  - Si ya no es necesario continuar (porque no se envió código, el tiempo terminó o se verificó el código), se limpia el intervalo.
- **Retorno:** Siempre limpia el intervalo anterior antes de ejecutar uno nuevo, lo que evita múltiples temporizadores simultáneos.

---

3. Redirección automática tras registro exitoso

```
js
CopiarEditar
useEffect(() => {
  if (registroExitoso) {
    const timer = setTimeout(() => {
      navigate('/login');
```


```
  }, 3000);
  return () => clearTimeout(timer);
 }
}, [registroExitoso]);
```

- **Finalidad:** Redirigir al usuario a la pantalla de login después de un registro exitoso.
- **Funcionamiento:**
  - Cuando registroExitoso es verdadero, se inicia un setTimeout que espera 3 segundos antes de ejecutar navigate('/login').
  - El temporizador se limpia automáticamente si cambia la condición o se desmonta el componente antes de cumplirse el tiempo.

```
116
117     try {
118         const response = await send(serviceId, templateId, templateParams, publicKey);
119         setError('Se ha enviado un código de verificación a tu correo electrónico.');
120         return response;
121     } catch (error) {
122         console.error('Error al enviar el correo electrónico:', error);
123
124         let errorMessage = 'Error al enviar el código de verificación. Por favor, inténtelo de nuevo.';
125
126         if (error.text && error.text.includes("recipient's address is corrupted")) {
127             errorMessage = 'El formato del correo electrónico no es válido. Por favor, verifícalo.';
128         } else if (error.status === 400) {
129             errorMessage = 'Error de configuración: Verifica que el Service ID "Flooky Pets" y Template ID "template_z3izl33" sean correctos
130         } else if (error.status === 422) {
131             errorMessage = 'El servicio de correo no pudo procesar tu dirección de email. Verifica que sea correcta.';
132         } else if (error.status === 429) {
133             errorMessage = 'Has excedido el límite de envíos. Por favor, espera unos minutos.';
134         }
135
```

try {

const response = await send(serviceId, templateId, templateParams, publicKey);

setError('Se ha enviado un código de verificación a tu correo electrónico.');

return response;

}

Propósito: Intenta enviar el correo electrónico con los parámetros necesarios (serviceId, templateId, etc.).

send(...): Función de envío (posiblemente EmailJS) que retorna una promesa.

setError(...): Aunque su nombre sugiere lo contrario, aquí se usa para mostrar un mensaje de confirmación informando al usuario que el código fue enviado correctamente.

return response: Devuelve la respuesta en caso de éxito.

Manejo de errores

js

```
catch (error) {
  console.error('Error al enviar el correo electrónico:', error);
  let errorMessage = 'Error al enviar el código de verificación. Por favor, inténtelo de nuevo.';
```

Si ocurre un error durante el envío, se entra en este bloque.

Se registra el error completo en la consola para propósitos de depuración.

Se inicializa un mensaje de error genérico (errorMessage) como fallback.

Interpretación detallada del error

js

```
if (error.text && error.text.includes("recipient's address is corrupted")) {
  errorMessage = 'El formato del correo electrónico no es válido. Por favor, verifícalo.';
}
```

Si el error contiene texto y detecta que la dirección del destinatario es inválida, se muestra un mensaje específico de error al usuario.

js

```
else if (error.status === 400) {

  errorMessage = 'Error de configuración: Verifica que el Service ID "Flooky Pets" y Template ID "template_z3izl33" sean correctos.';

}
```

400 (Bad Request): Indica un posible error en los identificadores del servicio o plantilla usados para enviar el correo. El mensaje guía al desarrollador a revisar la configuración.

js

```
else if (error.status === 422) {

  errorMessage = 'El servicio de correo no pudo procesar tu dirección de email. Verifica que sea correcta.';

}
```

422 (Unprocessable Entity): El servidor no pudo procesar la dirección proporcionada, usualmente por errores de validación.

```
else if (error.status === 429) {

  errorMessage = 'Has excedido el límite de envíos. Por favor, espera unos minutos.';

}
```

429 (Too Many Requests): Indica que se alcanzó el límite de envíos permitido por el servicio (rate limit). Se sugiere al usuario esperar antes de intentarlo de nuevo.

```
145
146         switch (name) {
147             case 'nombre':
148             case 'apellido':
149                 if (!value.trim()) {
150                     error = 'Este campo es obligatorio';
151                 } else if (!/^[A-ZÁÉÍÓÚÑ\s]+$/.test(value)) {
152                     error = 'Solo se permiten letras y espacios';
153                 } else if (value.length > 50) {
154                     error = 'Máximo 50 caracteres';
155                 }
156                 break;
157 
158             case 'telefono':
159                 if (!value.trim()) {
160                     error = 'Este campo es obligatorio';
161                 } else if (!/^(3\d{9}|[1-9]\d{6,7})$/.test(value)) {
162                     error = 'Formato de teléfono colombiano inválido';
163                 }
164                 break;
165 
166             case 'direccion':
167                 if (!value.trim()) {
168                     error = 'Este campo es obligatorio';
169                 } else if (!/^(Calle|Cll|Cl|Carrera|Cra|Cr|Avenida|Av|Avda|Avd|Transversal|Trans|Circular|Cir)\s?\d+.*$/i.test(value)) {
170                     error = 'La dirección debe comenzar con el tipo de vía (Ej: Calle, Carrera, Av.) seguido de número';
```

case 'nombre':

case 'apellido':

Campos requeridos: Se verifica que el valor no esté vacío con value.trim().

Solo letras y espacios: La expresión regular ^[A-Za-zÁÉÍÓÚÑáéíóúüÜ\s]+$ permite letras con tildes, diéresis y espacios.

Límite de longitud: No se permiten valores con más de 50 caracteres.

Errores posibles:

'Este campo es obligatorio'

'Solo se permiten letras y espacios'

'Máximo 50 caracteres'

Caso 'telefono'

case 'telefono':

Campo requerido: Verifica que no esté vacío.

Formato colombiano: Se espera un número entre 7 y 10 dígitos, comenzando por un dígito distinto de cero:

/^\(?\d{1,4}?\)?[1-9]\d{6,7}$/

o como se muestra en tu código: ^(3\d{9}|[1-9]\d{6,7})$

Errores posibles:

'Este campo es obligatorio'

'Formato de teléfono colombiano inválido'

Caso 'direccion'

case 'direccion':

Campo obligatorio: Validado con trim().

Formato estructurado: Se valida que la dirección inicie con un tipo de vía común en Colombia, como:

Calle, Carrera, Avenida, Av., Transversal, Circular, etc.

Seguido de un número con este patrón:/^.*\s\d+.*$/

Expresión regular completa usada:

/^(Calle|Cll|Cl|Carrera|Cra|Cr|Avenida|Av|Avda|Avd|Transversal|Trans|Circular|Cir)\s?\d+.*$/i

Error mostrado:

'La dirección debe comenzar con el tipo de vía (Ej: Calle, Carrera, Av.) seguido de número'

Usuario

```
    try {
      // Usar authFetch para obtener los datos del usuario real
      const response = await authFetch(`/usuarios/${currentUser.id}`); // Asumiendo que esta ruta existe
      if (response.success) {
        setUserData(response.data);
      } else {
        setError(response.message || 'Error al cargar los datos del usuario.');
        showNotification(response.message || 'Error al cargar datos del perfil', 'error');
      }
    } catch (err) {
      setError('Error de conexión con el servidor. Intenta de nuevo más tarde.');
      console.error("Error al obtener datos del usuario:", err);
      showNotification('Error de conexión', 'error');
    } finally {
      setIsLoading(false);
    }
  }, [user, showNotification, navigate, setUser]);

  // FUNCIÓN ACTUALIZADA PARA OBTENER NOTIFICACIONES DESDE LA API
  const fetchNotifications = useCallback(async (currentUserId) => {       unknown, hace 10 horas • notificacione
    if (!currentUserId) {
      console.warn('No user ID provided for fetching notifications.');
      return;
    }
```

Línea	Código	Explicación técnica
1	try {	Se inicia un bloque try para manejar errores en la ejecución de la petición a la API.
2	// Usar authFetch para obtener los datos del usuario real	Comentario explicativo del propósito de la línea siguiente.
3	const response = await authFetch(/usuarios/${currentUser.id});	Llama a la API /usuarios/{id} para obtener los datos reales del usuario autenticado.
4	if (response.success) {	Evalúa si la respuesta fue exitosa (success === true).
5	setUserData(response.data);	Almacena los datos del usuario en el estado de la aplicación.

Manual Técnico

Línea	Código	Explicación técnica
6	} else {	Bloque alternativo si la respuesta no fue exitosa.
7	`setError(response.message	
8	`showNotification(response.message	
9	}	Cierre del bloque if...else.
10	} catch (err) {	Captura errores imprevistos (fallas de red, excepciones, etc.).
11	setError('Error de conexión con el servidor. Intenta de nuevo más tarde.');	Muestra un mensaje de error genérico si no se puede conectar al servidor.
12	console.error("Error al obtener datos del usuario:", err);	Imprime el error en la consola para depuración.
13	showNotification('Error de conexión', 'error');	Muestra una notificación visual al usuario indicando el fallo.
14	} finally {	Este bloque se ejecuta siempre, haya éxito o error.
15	setIsLoading(false);	Cambia el estado de carga a falso para indicar que terminó la operación.
16	}, [user, showNotification, navigate, setUser]);	Lista de dependencias del hook useEffect o useCallback, evita llamadas innecesarias.

Manual Técnico

```
      </button>
      {showNotificationsMenu && (
        <div className={styles.notificationsDropdown}>
          <h3>Notificaciones</h3>      unknown, hace 10 horas • notificaciones
          {notifications.length === 0 ? (
            <p>No tienes notificaciones nuevas.</p>
          ) : (
            <ul>
              {notifications
                .sort((a, b) => new Date(b.fecha_creacion) - new Date(a.fecha_creacion)) // Más recientes primero
                .map((notif) => (
                  <li key={notif.id_notificacion} className={notif.leida ? styles.read : styles.unread}>
                    <div className={styles.notificationContent}>
                      <p className={styles.notificationMessage}>{notif.mensaje}</p>
                      <span className={styles.notificationDate}>
                        {new Date(notif.fecha_creacion).toLocaleString()}
                      </span>
                    </div>
                    {!notif.leida && (
                      <button
                        className={styles.markAsReadButton}
                        onClick={() => markNotificationAsRead(notif.id_notificacion)}
                        title="Marcar como leída"
                      >
```

Línea	Código	Explicación técnica
1	{showNotificationsMenu && (	Si showNotificationsMenu es verdadero, se renderiza el bloque de notificaciones. Esto es una condición lógica corta (&&) en React.
2		Contenedor principal del menú de notificaciones, con estilo CSS aplicado desde el módulo styles.
3	Notificaciones	Título del menú desplegable.
4	{notifications.length === 0 ? (	Si no hay notificaciones (length === 0), se muestra un mensaje indicando que no hay notificaciones.
5	No tienes notificaciones nuevas.	Mensaje mostrado cuando no hay notificaciones.
6	) : (	Cierre del operador ternario: si sí hay notificaciones, se renderiza una lista.

Manual Técnico

Línea	Código	Explicación técnica
7		Inicio de la lista (ul) que contendrá todas las notificaciones.
8	{notifications	Accede al array notifications.
9	.sort((a, b) => new Date(b.fecha_creacion) - new Date(a.fecha_creacion))	Ordena las notificaciones por fecha de creación en orden descendente (más recientes primero).
10	.map((notif) => (	Itera sobre cada notificación para renderizarla individualmente.
11	<li key={notif.id_notificacion} className={notif.leida ? styles.read : styles.unread}>	Crea un elemento de lista (li) para cada notificación con clase condicional: read si está leída, unread si no.
12		Contenedor del contenido de la notificación (mensaje y fecha).
13	{notif.mensaje}	Muestra el mensaje de la notificación.
14		Muestra la fecha de la notificación en un span estilizado.
15	{new Date(notif.fecha_creacion).toLocaleString()}	Convierte la fecha a un formato legible para el usuario con toLocaleString().
16		Cierre del span de fecha.
17		Cierre del contenedor del contenido de la notificación.
18	{!notif.leida && (	Si la notificación **no está leída**, se muestra un botón para marcarla como leída.
19	<button	Comienza la definición del botón.

Manual Técnico

Línea	Código	Explicación técnica
20	className={styles.markAsReadButton}	Aplica estilo al botón usando clases del módulo styles.
21	onClick={() => markNotificationAsRead(notif.id_notificacion)}	Define el evento onClick que llama a markNotificationAsRead con el ID de la notificación.
22	title="Marcar como leída"	Muestra una sugerencia al pasar el cursor por el botón.

```
if (error && !user?.id) {
  return (
    <div className={styles.errorMessage}>
      <FontAwesomeIcon icon={faTimesCircle} className={styles.errorIcon} />
      <h2>¡Oh no! Ha ocurrido un problema.</h2>
      <p>{error}</p>
      <button onClick={() => navigate('/login')} className={styles.retryButton}>
        Ir a Iniciar Sesión
      </button>
    </div>      You, hace 3 días • Conexion Menu Usuarios e implementacion de Ima
  );
}
```

Aquí es donde detecta un error en el sistema al momento de ya estar dentro de la pagina web.

Administrador

Manual Técnico

```
// Redirige si el usuario no es admin o no está logueado      unknown, hace 6 horas • dasdas
useEffect(() => {
    if (!user || user.role !== 'admin') {
        navigate('/login', { replace: true });
    }
}, [user, navigate]);

// Maneja clics fuera del sidebar para cerrarlo
useEffect(() => {
    function handleClickOutside(event) {
        // Cierra el sidebar en móvil si se hace clic fuera, pero no en el botón de toggle
        if (window.innerWidth <= 768 && sidebarRef.current && !sidebarRef.current.contains(event.targ
            const menuToggleBtn = document.querySelector('.menu-toggle-btn');
            if (menuToggleBtn && !menuToggleBtn.contains(event.target)) {
                setIsSidebarOpen(false);
            }
        }
```

Impide el acceso a usuarios no autenticados o sin privilegios de administrador.

Redirecciona a /login utilizando el hook navigate de react-router-dom.

```
                <li>
                    <Link to="users" className={getNavLinkClass('/admin/users')} onClick={() => setIsSidebarOpen(false)}>
                        <FaUsers className="admin-nav-icon" /> Gestión de Clientes
                        {location.pathname.startsWith('/admin/users') && <span className="admin-active-indicator"></span>}
                    </Link>
                </li>      You, hace 15 horas • gdggd
                <li>
                    <Link to="appointments" className={getNavLinkClass('/admin/appointments')} onClick={() => setIsSidebarOpen(false)}
                        <FaCalendarAlt className="admin-nav-icon" /> Gestión de Citas
                        {location.pathname.startsWith('/admin/appointments') && <span className="admin-active-indicator"></span>}
                    </Link>
                </li>
                <li>
                    <Link to="medical-records" className={getNavLinkClass('/admin/medical-records')} onClick={() => setIsSidebarOpen(f
                        <FaNotesMedical className="admin-nav-icon" /> Historiales Médicos
                        {location.pathname.startsWith('/admin/medical-records') && <span className="admin-active-indicator"></span>}
                    </Link>
                </li>
                <li>
                    <Link to="settings" className={getNavLinkClass('/admin/settings')} onClick={() => setIsSidebarOpen(false)}>
                        <FaCog className="admin-nav-icon" /> Configuración
                        {location.pathname.startsWith('/admin/settings') && <span className="admin-active-indicator"></span>}
                    </Link>
                </li>
            </ul>
        </nav>
```

Cada elemento del menú (<li>) contiene lo siguiente:

- Un componente <Link> que redirige a la ruta correspondiente (/admin/users, /admin/appointments, etc.).
- Un ícono de FontAwesome (FaUsers, FaCalendarAlt, FaNotesMedical, FaCog) que visualmente representa la funcionalidad.
- Un texto descriptivo como "Gestión de Clientes" o "Historiales Médicos".

- Una clase dinámica calculada mediante getNavLinkClass() que activa estilos según si la ruta actual coincide con la ruta del link.
- Una lógica de comparación con location.pathname.startsWith() que permite marcar visualmente la sección activa mediante la clase admin-active-indicator.
- Un evento onClick={() => setIsSidebarOpen(false)} que cierra el sidebar cuando se hace clic en cualquier enlace, optimizando la experiencia en dispositivos móviles o pantallas pequeñas.

```
return (
    <div className="admin-dashboard-container">
        {/* Sidebar (barra lateral izquierda) */}
        <aside className={`admin-sidebar ${isSidebarOpen ? 'open' : ''}`} ref={sidebarRef}
            <div className="admin-sidebar-header">
                <div className="admin-logo-container">
                    <FaStethoscope className="admin-logo-icon" />
                    <h2>Flooky Pets</h2>
                </div>
                <p className="admin-clinic-name">Panel de Administración</p>
                <button className="close-sidebar-btn" onClick={() => setIsSidebarOpen(false
                    <FaTimes />
                </button>
            </div>

            <div className="admin-user-profile">
                <div className="admin-avatar">
                    {user?.profilePicture ? (
                        <img src={user.profilePicture} alt="Avatar de usuario" className="
                    ) : (
                        <FaUserCircle />
                    )}
                </div>
                <div className="admin-user-info">
                    <h3>{user?.nombre}</h3>
                    <p>{user?.role === 'admin' ? 'Administrador' : user?.role}</p>
                    <Link to="profile" className="admin-profile-button" onClick={() => set
                        Ver Perfil
```

Este código crea la interfaz principal para administradores, con:

1. **Barra lateral** desplegable que incluye:
    - Logo y nombre de la plataforma
    - Botón para abrir/cerrar el menú
2. **Perfil de usuario** con:

- Foto (o ícono predeterminado si no hay imagen)
- Nombre y tipo de rol (ej. "Administrador")
- Enlace para editar el perfil

```
<li>
    <Link to="dashboard" className={getNavLinkClass('/admin/dashboard')} onClick={()
        <FaTachometerAlt className="admin-nav-icon" /> Dashboard
        {location.pathname.startsWith('/admin/dashboard') && <span className="admin-a
    </Link>
</li>
<li>
    <Link to="services" className={getNavLinkClass('/admin/services')} onClick={() =>
        <FaBriefcaseMedical className="admin-nav-icon" /> Gestión de Servicios
        {location.pathname.startsWith('/admin/services') && <span className="admin-ac
    </Link>
</li>
<li>
    <Link to="veterinarians" className={getNavLinkClass('/admin/veterinarians')} onCl
        <FaStethoscope className="admin-nav-icon" /> Gestión de Veterinarios
        {location.pathname.startsWith('/admin/veterinarians') && <span className="adm
    </Link>
</li>
<li>
    <Link to="administrators" className={getNavLinkClass('/admin/administrators')} on
        <FaUserShield className="admin-nav-icon" /> Gestión de Administradores
        {location.pathname.startsWith('/admin/administrators') && <span className="ad
    </Link>
</li>
<li>
    <Link to="users" className={getNavLinkClass('/admin/users')} onClick={() => setIs
        <FaUsers className="admin-nav-icon" /> Gestión de Clientes
        {location.pathname.startsWith('/admin/users') && <span className="admin-activ
    </Link>
</li>
<li>
```

"El menú lateral proporciona acceso rápido a todas las áreas de administración:

1. **Dashboard** - Vista general del sistema
2. **Servicios** - Gestionar servicios ofrecidos
3. **Veterinarios** - Administrar el equipo veterinario
4. **Administradores** - Gestionar permisos y cuentas
5. **Clientes** - Administrar usuarios registrados

Veterinaria

```
// Asegúrate de que esta URL base coincida con la de tu backend
const API_BASE_URL = process.env.REACT_APP_API_BASE_URL || 'http://localhost:5000';

const showNotification = useCallback((message, type = 'info', duration = 3000) => {
    if (notificationTimeout) {
        clearTimeout(notificationTimeout);
    }

    setNotification({ message, type });

    const timeout = setTimeout(() => {
        setNotification(null);
    }, duration);
    setNotificationTimeout(timeout);
}, [notificationTimeout]);

useEffect(() => {
    return () => {
        if (notificationTimeout) {
            clearTimeout(notificationTimeout);
        }
    };
}, [notificationTimeout]);

// Función para obtener notificaciones del veterinario desde el backend
const fetchVetNotifications = useCallback(async (currentVetId) => {
    if (!currentVetId) {
        console.log("No veterinarian ID available to fetch notifications.");
        setVetNotifications([]);
        return;
```

Componentes Clave

1. Configuración de URL Base

javascript

const API_BASE_URL = process.env.REACT_APP_API_BASE_URL || 'http://localhost:5000';

- Define la URL base para las llamadas al API
- Usa una variable de entorno (REACT_APP_API_BASE_URL) con fallback a localhost

- **Importante**: Debe coincidir con la configuración del backend

2. Sistema de Notificaciones (Frontend)

javascript

```
const showNotification = useCallback((message, type = 'info', duration = 3000) => {

 // Lógica de manejo de notificaciones

}, [notificationTimeout]);
```

- **Parámetros**:
  - message: Contenido de la notificación
  - type: Tipo de notificación (por defecto 'info')
  - duration: Tiempo de visualización (3000ms por defecto)
- **Funcionalidad**:
  - Cancela notificaciones previas pendientes
  - Muestra la nueva notificación
  - Oculta automáticamente después del tiempo especificado
  - Usa useCallback para optimización de rendimiento

3. Limpieza de Efectos

javascript

```
useEffect(() => {

 return () => {

  if (notificationTimeout) {

   clearTimeout(notificationTimeout);

  }

 };

}, [notificationTimeout]);
```

- Limpia los timeouts al desmontar el componente
- Previene memory leaks

4. Obtención de Notificaciones para Veterinarios

javascript

```
const fetchVetNotifications = useCallback(async (currentVetId) => {

  // Lógica de obtención

}, [/* dependencias */]);
```

- **Parámetro**: currentVetId - ID del veterinario actual
- **Funcionalidad**:
  - Valida la existencia del ID
  - Si no hay ID, limpia las notificaciones existentes
  - (El código completo mostraría la llamada API al backend)

Diagrama de Flujo

1. showNotification es llamado → Cancela notificación existente → Muestra nueva → Programa ocultación
2. Al desmontar componente → Limpieza de timeout
3. fetchVetNotifications → Verifica ID → Recupera datos del backend

Manual Técnico

```
const MainVeterinario = ({ user, setUser }) => {
    useEffect(() => {
        const handleClickOutside = (event) => {
        };

        document.addEventListener('mousedown', handleClickOutside);
        return () => {
            document.removeEventListener('mousedown', handleClickOutside);
        };
    }, []);

    const navigateTo = useCallback((path) => {
        navigate(`/veterinario/${path}`);
    }, [navigate]);

    const handleLogout = () => {
        localStorage.removeItem('token');
        localStorage.removeItem('user');
        setUser(null);
        showNotification('¡Hasta pronto! Cerrando tu sesión...', 'success');
        navigate('/login');
    };

    const isDashboard = location.pathname === '/veterinario' ||
        location.pathname === '/veterinario/' ||
        location.pathname === '/veterinario/navegacion';

    return (
        <motion.div
            className={styles.vetMainContainer}
            variants={containerVariants}
            initial="hidden"
            animate="visible"
            exit="exit"
```

Estructura Principal

1. Manejo de Eventos Externos

javascript

useEffect(() => {

const handleClickOutside = (event) => {

// Lógica para detectar clicks fuera del componente

};


```
 document.addEventListener('mousedown', handleClickOutside);
 return () => {
  document.removeEventListener('mousedown', handleClickOutside);
 };
}, []);
```

- **Propósito: Detectar clicks fuera del área del componente**
- **Implementación:**
  - **Añade listener al montar el componente**
  - **Limpia listener al desmontar (patrón de limpieza)**
  - ***Nota*: Falta implementación completa de la lógica de detección**

2. Navegación Programática

```
javascript
const navigateTo = useCallback((path) => {
 navigate('/veterinario/$(path)');
}, [navigate]);
```

- **Función: Navegación dinámica dentro del área veterinaria**
- **Características:**
  - **Memoizada con useCallback para optimización**
  - **Usa template literals (corregir $() a ${})**
  - **Depende del hook navigate (react-router-dom)**

3. Manejo de Cierre de Sesión

```
javascript
const handleLogout = () => {
 localStorage.removeItem('token');
```


```
  localStorage.removeItem('user');
  setUser(null);
  showNotification('¡Hasta pronto! Cerrando tu sesión...', 'success');
  navigate('/login');
};
```

- Flujo:
    1. Elimina credenciales de localStorage
    2. Limpia estado de usuario
    3. Muestra notificación de éxito
    4. Redirige a login
- Seguridad:
    - Limpieza completa de datos sensibles
    - Feedback visual al usuario

4. Detección de Ruta Actual

javascript

```
const isDashboard = location.pathname === '/veterinario' ||
                    location.pathname === '/veterinario/' ||
                    location.pathname === '/veterinario/navegacion';
```

- Uso: Determinar si está en la vista dashboard
- Lógica: Compara pathname contra rutas base

5. Animaciones (Framer Motion)

javascript

```
<motion.div
  className={styles.vetMainContainer}
```

variants={containerVariants}

initial="hidden"

animate="visible"

exit="exit"

>

- Biblioteca: Framer Motion para animaciones

```
                <div className={styles.vetQuickActions}>
                    <h4>Acciones rápidas</h4>
                    <motion.button
                        whileHover={{ x: 5, backgroundColor: "#00bcd4", color: "white" }}
                        whileTap={{ scale: 0.95 }}
                        onClick={() => navigateTo('citas/agendar')}
                        className={styles.vetQuickButton}
                    >
                        <FontAwesomeIcon icon={faPlus} />
                        <span>Nueva Cita</span>
                    </motion.button>
                    <motion.button
                        whileHover={{ x: 5, backgroundColor: "#4CAF50", color: "white" }}
                        whileTap={{ scale: 0.95 }}
                        onClick={() => navigateTo('mascotas/registrar')}
                        className={styles.vetQuickButton}
                    >
                        <FontAwesomeIcon icon={faPaw} />
                        <span>Registrar Mascota</span>
                    </motion.button>
                    <motion.button
                        whileHover={{ x: 5, backgroundColor: "#FF9800", color: "white" }}
                        whileTap={{ scale: 0.95 }}
                        onClick={() => navigateTo('historiales/registrar')}
                        className={styles.vetQuickButton}
                    >
                        <FontAwesomeIcon icon={faNotesMedical} />
                        <span>Registrar Historial</span>
                    </motion.button>
                </div>

                <div className={styles.vetSidebarFooter}>
                    <motion.button
```

crea tres botones de acceso rápido para veterinarios en un panel de administración. Cada botón:

1. **Tiene un ícono y texto descriptivo**
    - **+ Nueva Cita (icono de suma)**
    - **Registrar Mascota (icono de huella)**
    - **Registrar Historial (icono de notas médicas)**
2. **Reacciona visualmente al interactuar**
    - **Se mueve ligeramente a la derecha (x: 5) y cambia de color cuando el mouse pasa encima**
    - **Se aplasta un poco (scale: 0.95) al hacer clic**
3. **Al hacer clic**
    - **Navega a una sección diferente del sistema:**
        - **Agendar citas (citas/agendar)**
        - **Registrar mascotas nuevas (mascotas/registrar)**
        - **Añadir historiales médicos (historiales/registrar)**
4. **Está agrupado bajo el título *"Acciones rápidas"* en un contenedor con estilos propios (vetQuickActions).**

17. Glosario

- **API**: Interfaz que permite que el frontend y backend se comuniquen.
- **JWT**: JSON Web Token, sistema de autenticación basado en tokens.
- **CRUD**: Operaciones básicas de una base de datos: Crear, Leer, Actualizar, Eliminar.
- **DER/MER**: Diagramas usados para modelar las relaciones entre datos.
- **MongoDB**: Base de datos NoSQL usada para almacenar información del sistema.

18. Referencias

MySQL Documentation
Oracle Corporation (2024). *MySQL 8.0 Reference Manual*. Disponible en: https://dev.mysql.com/doc/

Node.js Documentation
Node.js Foundation (2024). *Node.js v18+ Documentation*. Disponible en: https://nodejs.org/en/docs/

React Official Docs
Meta (2024). *React – A JavaScript library for building user interfaces*. Disponible en: https://react.dev/

Express.js Documentation
OpenJS Foundation (2024). *Express – Node.js web application framework*. Disponible en: https://expressjs.com/

PlantUML Documentation
PlantUML Team (2024). *Diagrams generation from plain text*. Disponible en: https://plantuml.com/

Figma – Collaborative UI Design Tool
Figma Inc. (2024). *Design, prototype, and collaborate all in the browser*. Disponible en: https://www.figma.com/

Scrum Guide
Schwaber, K., & Sutherland, J. (2020). *The Scrum Guide – The Definitive Guide to Scrum*. Disponible en: https://scrumguides.org/

Lucidchart – Diagramming Tool
Lucid Software Inc. (2024). *Visual collaboration suite for diagramming*. Disponible en: https://www.lucidchart.com/pages/

Jest Testing Framework
Meta (2024). *Jest: Delightful JavaScript Testing*. Disponible en: https://jestjs.io/