Ingeniería del Software II

Trabajo Práctico Integrador final
“Proyecto de Desarrollo de Software”

Docentes a cargo:
Victor Valotto, Cielo Godoy

Alumnas:
Inderkumer, Priscila
Murzi, Ana Sol

04 - NOV - 2025

Contexto del proyecto

Descripción breve de la necesidad que el sistema aborda

En la ciudad de Oro Verde la pérdida de mascotas domésticas representa una problemática frecuente. Muchos dueños no cuentan con herramientas eficientes para difundir la pérdida, recibir alertas de avistamientos o localizar a sus animales.
Actualmente, la comunicación depende de redes sociales o carteles físicos, lo que limita la rapidez y la efectividad de la búsqueda.
Proponemos entonces un “Sistema de Recuperación de Mascotas Perdidas” en Oro Verde. Este sistema surge como una solución tecnológica que permite:

- Registrar mascotas y generar reportes de pérdida o hallazgo
- Geolocalizar los reportes sobre un mapa para facilitar las búsquedas
- Enviar notificaciones automáticas a vecinos, veterinarias y autoridades municipales cercanas
- Centralizar la información en una plataforma accesible y confiable

De esta manera, el sistema reduce los tiempos de respuesta ante una pérdida, mejora la comunicación entre la comunidad y aumenta la probabilidad de recuperar a las mascotas.

Identificación de los usuarios principales

Dueños de mascotas

- Registrar sus animales y generar reportes de pérdida
- Consultar el estado de alertas y recibir notificaciones

Autoridades municipales

- Administrar reportes y validar coincidencias entre pérdidas y hallazgos
- Gestionar estadísticas sobre casos y zonas críticas

Vecinos de Oro Verde

- Recibir alertas sobre mascotas perdidas en su zona
- Reportar avistamientos

Objetivos del software

- Facilitar la comunicación inmediata entre usuarios ante una pérdida
- Utilizar la geolocalización para optimizar la búsqueda de mascotas
- Centralizar y mantener actualizada la información de mascotas registradas
- Integrar notificaciones automáticas y visualización en mapas
- Proveer un sistema escalable y de fácil acceso desde la web o dispositivos móviles

Ingeniería de Software I – Análisis y diseño

Principales requerimientos funcionales y no funcionales

Requerimientos funcionales

- El sistema permitirá registrar usuarios (dueños, vecinos, Municipalidad de Oro Verde)
- El sistema permitirá registrar y actualizar datos de mascotas
- El usuario podrá reportar una mascota perdida o un avistamiento
- El sistema permitirá consultar reportes activos y sus ubicaciones
- Se generarán notificaciones automáticas a usuarios cercanos cuando una mascota sea reportada como perdida
- El sistema integrará un servicio externo de geolocalización
- Los administradores podrán visualizar estadísticas y coincidencias entre reportes

Requerimientos no funcionales

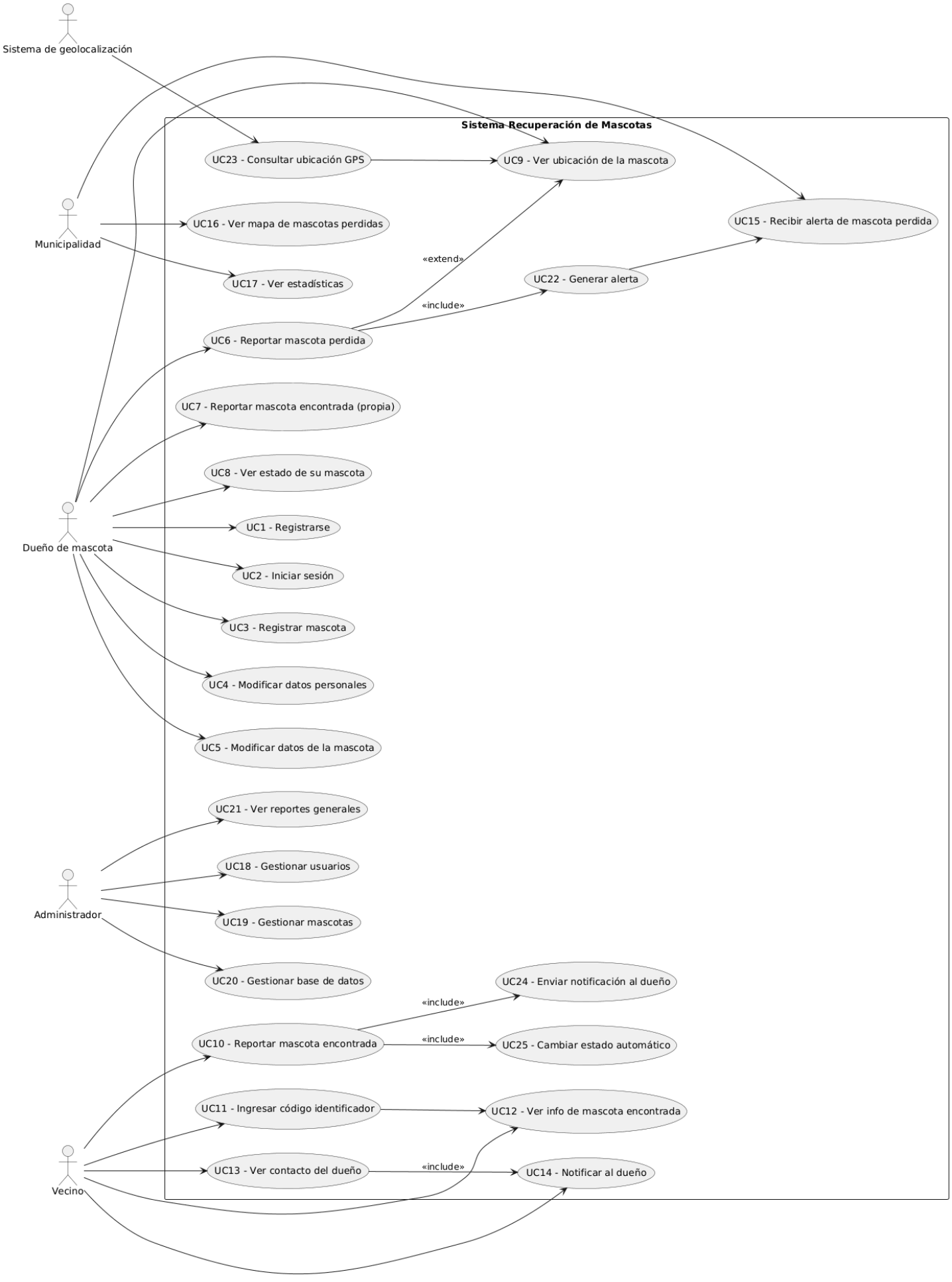
- Usabilidad
- Disponibilidad
- Seguridad
- Escalabilidad
- Mantenibilidad
- Interoperabilidad
- Performance
- Recuperación
- Confiabilidad
- Administrabilidad

Diagrama de Casos de Uso y descripción resumida de uno o dos casos clave

El diagrama de casos de uso fue elaborado utilizando la plataforma PlantUML, una herramienta que permite generar diagramas a partir de texto. El mismo se encuentra disponible en el siguiente link:
Diagrama de UC

Facultad de Ingeniería
UNER

TP FINAL - Ingeniería de Software II



Los casos de uso seleccionados para esta sección fueron:

- **UC22-** Generar alerta de mascota perdida

Descripción	El sistema genera automáticamente una alerta cuando se reporta una mascota como perdida
Actores	Dueño de mascota
Escenario típico	
Actor	**Software**
1. El dueño reporta su mascota como perdida (UC6)	2. El sistema crea una alerta con los datos de la mascota.
	3. El sistema notifica a la municipalidad y a vecinos registrados.
	4. El caso de uso termina.
Excepciones	**Software**
Fallo en la generación de la alerta	El sistema informa al dueño y lo reintenta
Alternativas	**Software**
UC relacionados	UC6, UC15, UC24.
Pre-condiciones	La mascota debe estar registrada y ser marcada como perdida
Post-condiciones	La alerta queda registrada y distribuida.

- **UC23 -** Consultar ubicación a sistema de geolocalización

Descripción	El sistema intenta obtener la ubicación de la mascota
Actores	Sistema de geolocalización
Escenario típico	
Actor	**Software**
1. El dueño reporta su mascota como perdida (UC6)	2. El sistema consulta ubicación en tiempo real
	3. Se muestra en el mapa del sistema (UC9)
	4. El caso de uso termina

Excepciones	**Software**
Alternativas	**Software**
Dispositivo no disponible o sin señal	Se muestra mensaje de "sin resultados"

UC relacionados	UC9, UC6.
Pre-condiciones	La mascota debe tener geolocalización habilitada.
Post-condiciones	Se muestra la ubicación de la mascota.

Decisión de calidad: descripción de los atributos de calidad priorizados

Se priorizaron los siguientes atributos de calidad:

Recuperación

Desde el punto de vista técnico, la pérdida de datos de registros de mascotas comprometería la integridad del sistema completo, por lo que se requieren procesos de recuperación robustos y automatizados (respaldo, redundancia y reinicio seguro).

Confiabilidad

Es crítica porque el sistema gestiona estados que deben mantenerse consistentes en tiempo real.

Las inconsistencias de datos no solo afectarían la experiencia del usuario, sino que podrían generar reportes duplicados, errores de coincidencia y pérdida de confianza en la plataforma. La integridad referencial entre las entidades *Mascota*, *Estado* y *Usuario* es fundamental para garantizar el correcto funcionamiento del sistema.

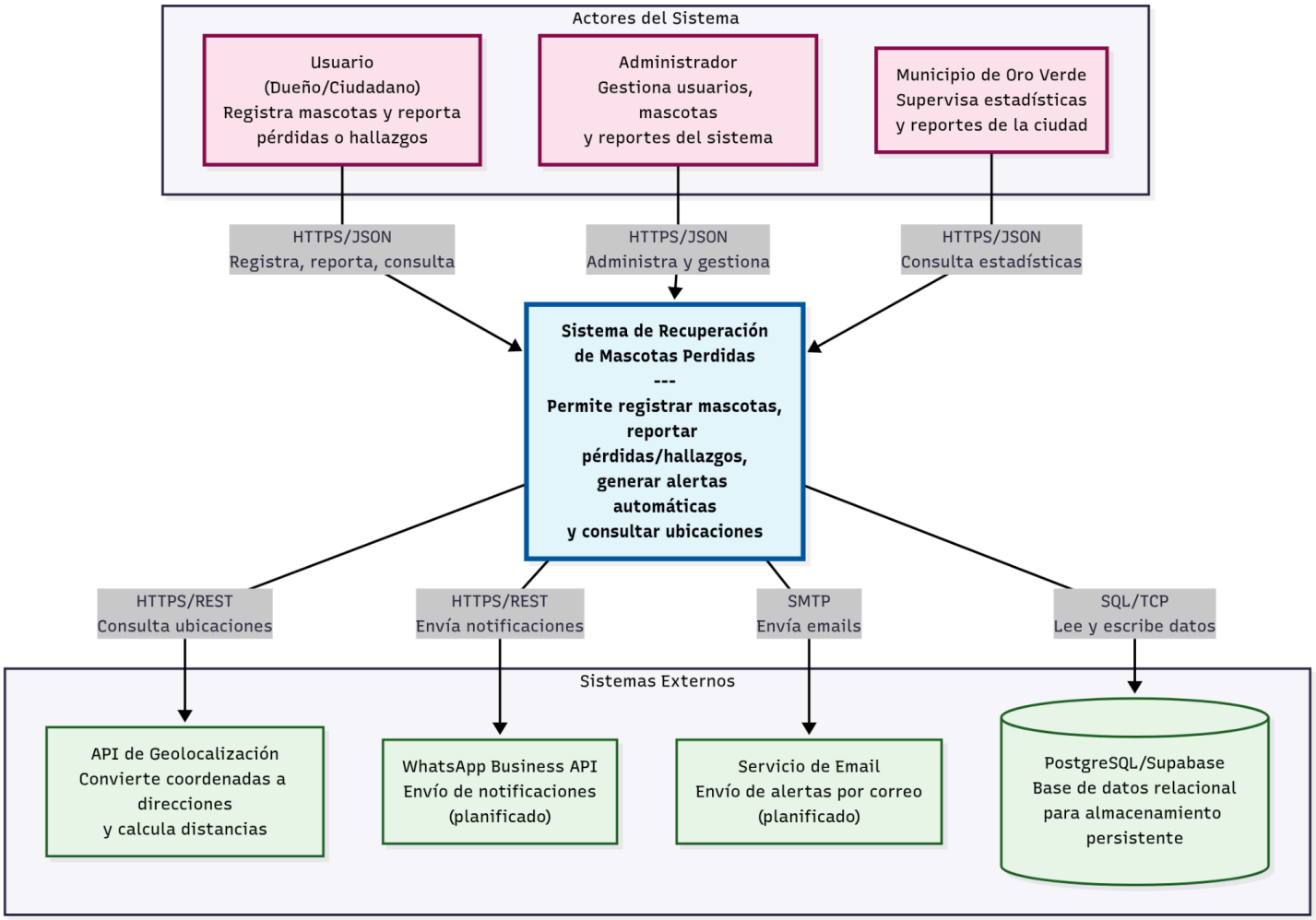
Usabilidad

Desde el punto de vista técnico, una interfaz compleja o sobrecargada reduciría la efectividad del sistema y podría provocar errores de entrada o abandono del proceso de reporte, afectando directamente su propósito principal

Ingeniería de Software II – Arquitectura y despliegue

Diagrama C4

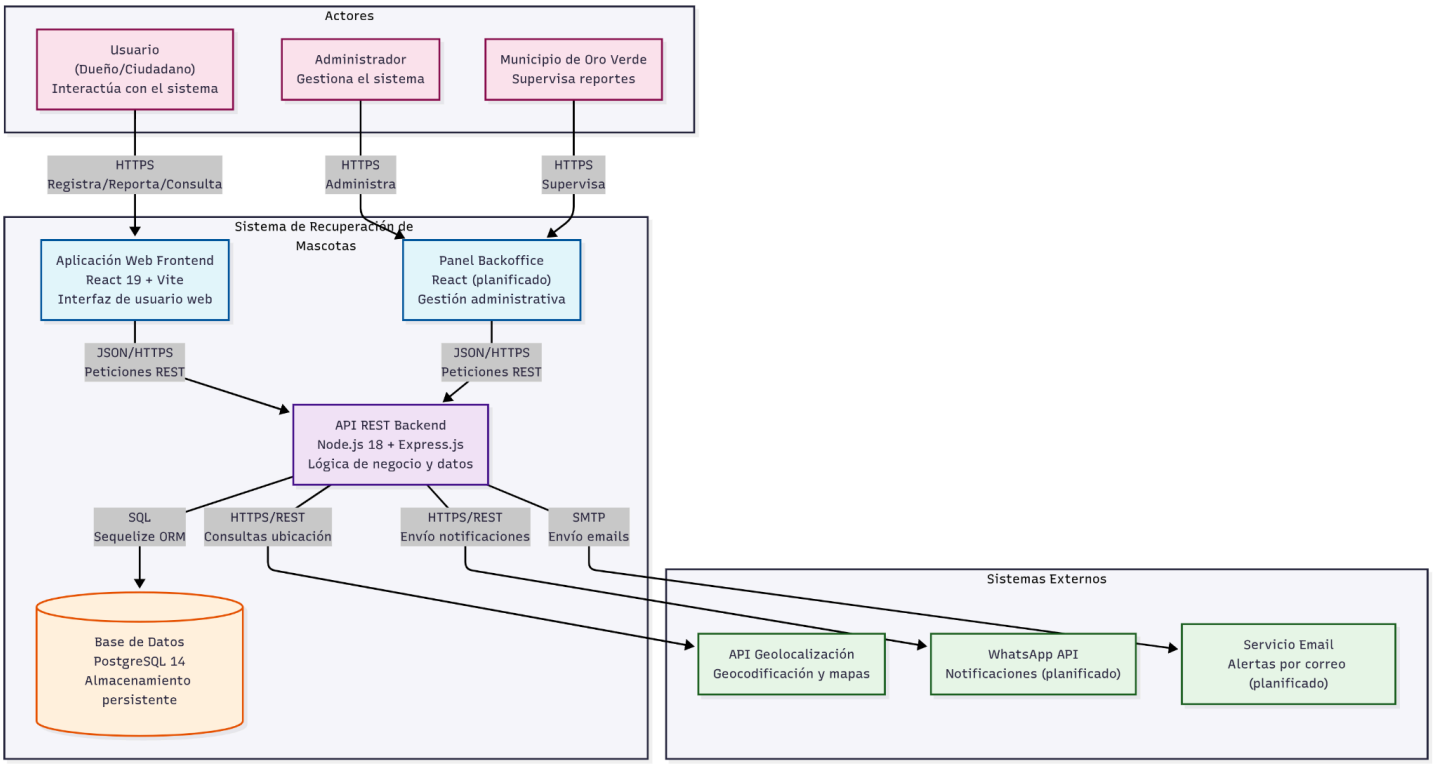
Diagrama de contexto - link



Facultad de UNER Ingeniería

TP FINAL - Ingeniería de Software II

Diagrama de contenedores - link



Facultad de UNER Ingeniería

TP FINAL - Ingeniería de Software II

Diagrama de componentes - link

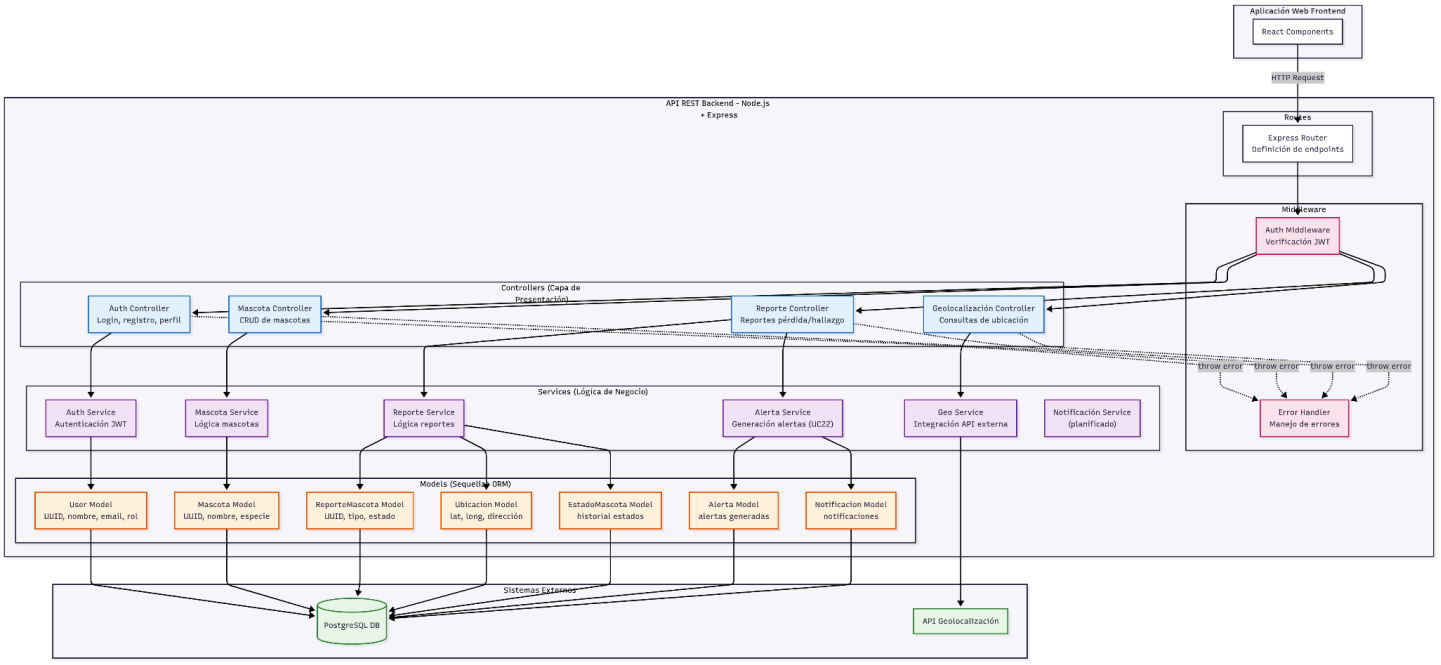


Diagrama de código - link

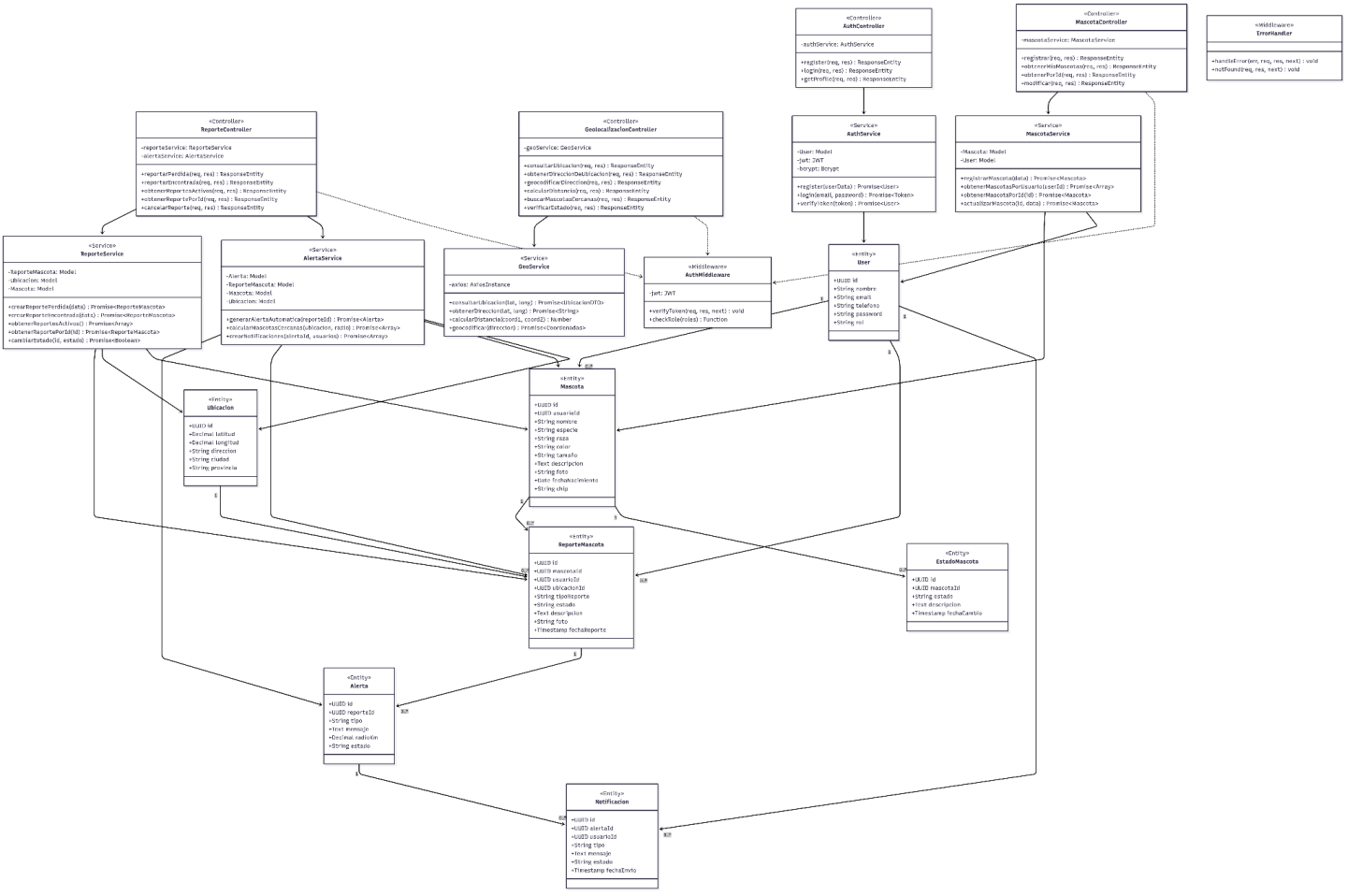
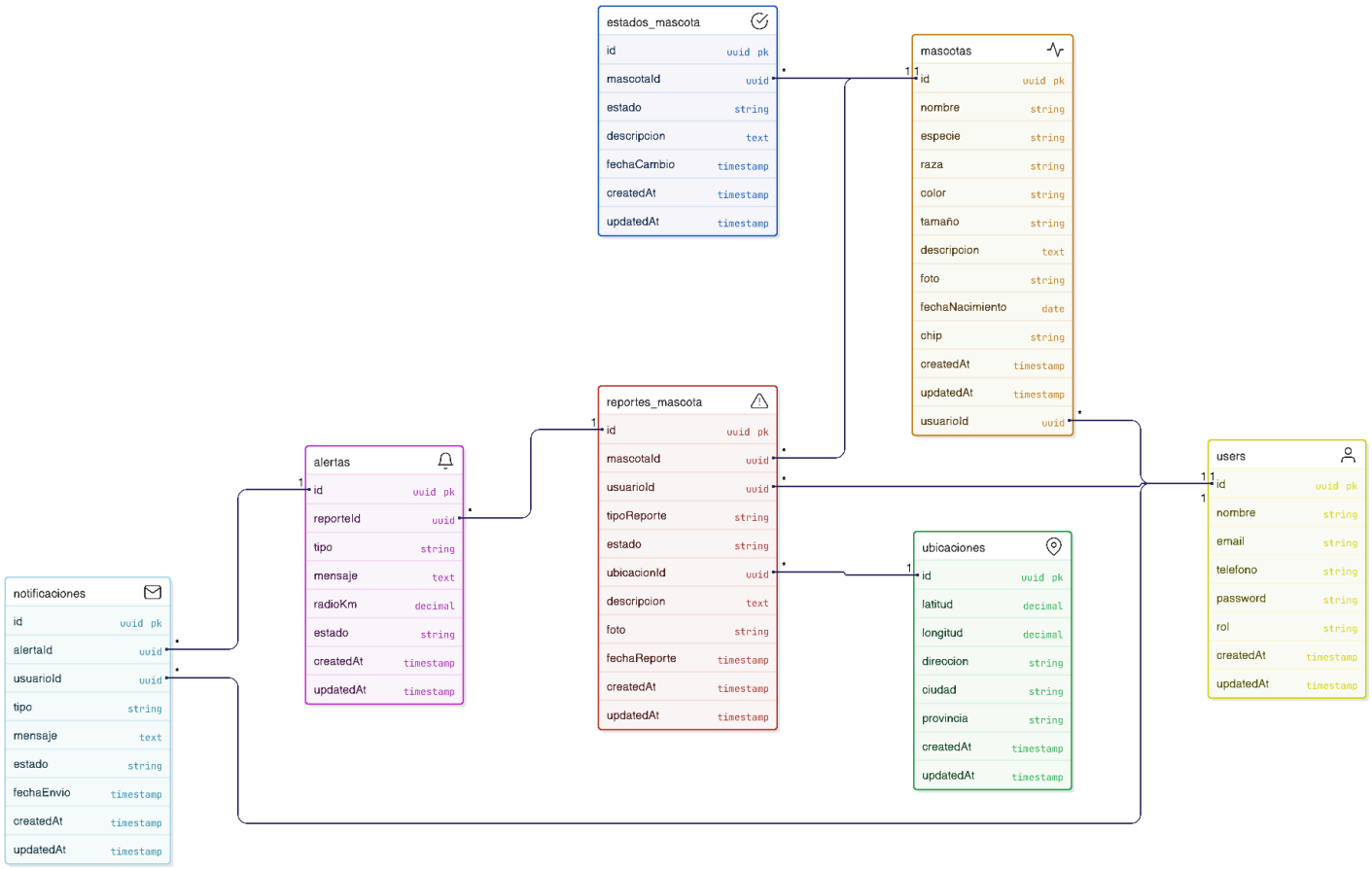


Diagrama de Clases UML representativo del diseño interno - link



Implementación

El sistema implementa una Arquitectura en Capas con separación Client-Server, basada en el modelo C4 documentado en los diagramas de diseño.

Clases principales implementadas de la capa de modelos:

Clase User

- Ubicación: server/src/models/User.js
- Correspondencia con el diseño: Implementa la entidad "User" del diagrama de código C4 y el diagrama de entidades.
- Responsabilidad: Gestionar la información de usuarios del sistema (dueños, vecinos, municipalidad).
- Características de Seguridad:
  - Hash de contraseñas con bcrypt
  - Método de comparación de contraseñas:
    User.prototype.compararPassword = async function(passwordIngresado) {
      return await bcrypt.compare(passwordIngresado, this.password);
    };
  - Ocultamiento de contraseña en respuestas JSON
- Validaciones implementadas:

- Email único y formato válido
- Contraseña mínimo 6 caracteres
- Campos obligatorios: nombre, apellido, email, password, celular
- Enumeración de roles: 'dueno', 'vecino', 'municipalidad'

Clase Mascota

- Ubicación: server/src/models/Mascota.js
- Correspondencia con el diseño: Implementa la entidad "Mascota" del diagrama de entidades y diagrama de código.
- Responsabilidad: Almacenar información de mascotas registradas en el sistema.
- Relaciones: Mascota N:1 User (FK: usuarioId)

Clase ReporteMascota

- Ubicación: server/src/models/ReporteMascota.js
- Correspondencia con el diseño: Implementa la entidad "ReporteMascota" del diagrama de entidades.
- Responsabilidad: Gestionar reportes de mascotas perdidas, encontradas o vistas.
- Relaciones:
  - ReporteMascota N:1 Mascota (FK: mascotaId)
  - ReporteMascota N:1 User (FK: usuarioId)
  - ReporteMascota N:1 Ubicacion (FK: ubicacionId)
- Estados del ciclo de vida:
  - activo: Reporte en búsqueda activa
  - resuelto: Mascota encontrada/recuperada
  - cancelado: Reporte cancelado por el usuario

Clases principales implementadas de la capa de controladores:

AuthController

- Ubicación: server/src/controllers/auth.controller.js
- Responsabilidad: Gestionar autenticación y autorización de usuarios.
- Métodos principales:
  - registrarUsuario() Implementa el registro de nuevos usuarios con:
    - Validación de campos obligatorios
    - Verificación de email único
    - Generación automática de token JWT
    - Hash de contraseña (delegado al modelo User)
  - iniciarSesion() Implementa autenticación con:
    - Verificación de credenciales
    - Comparación segura de contraseñas
    - Verificación de estado activo
    - Generación de token JWT
  - Función auxiliar generarToken()
- Correspondencia con el diseño: Implementa el flujo de autenticación del diagrama de contexto, genera tokens JWT como se especifica en el diagrama de componentes y utiliza HTTPS/JSON como protocolo de comunicación (diagrama de contexto).

ReporteController

- Ubicación: server/src/controllers/reporte.controller.js
- Responsabilidad: Gestionar reportes de mascotas perdidas/encontradas y generar alertas automáticas.
- Método principal: reportarMascotaPerdida()
- Correspondencia con el diseño: Implementa el flujo completo del diagrama de componentes, genera alertas automáticamente como se especifica en el diseño, crea notificaciones para vecinos (integración con sistemas externos planificada) y actualiza el estado de la mascota según el diagrama de estados.

Clase implementada de la capa de servicios:

GeolocalizacionService

- Ubicación: server/src/services/geolocalizacion.service.js
- Correspondencia con el diseño: Implementa el componente "Geo Service" del diagrama de componentes C4 y la integración con "API de Geolocalización" del diagrama de contexto. También utiliza el protocolo HTTP/REST como se especifica en el diagrama de contenedores
- Responsabilidad: Integración con sistemas externos de geolocalización (UC23).

Stack Tecnológico Implementado

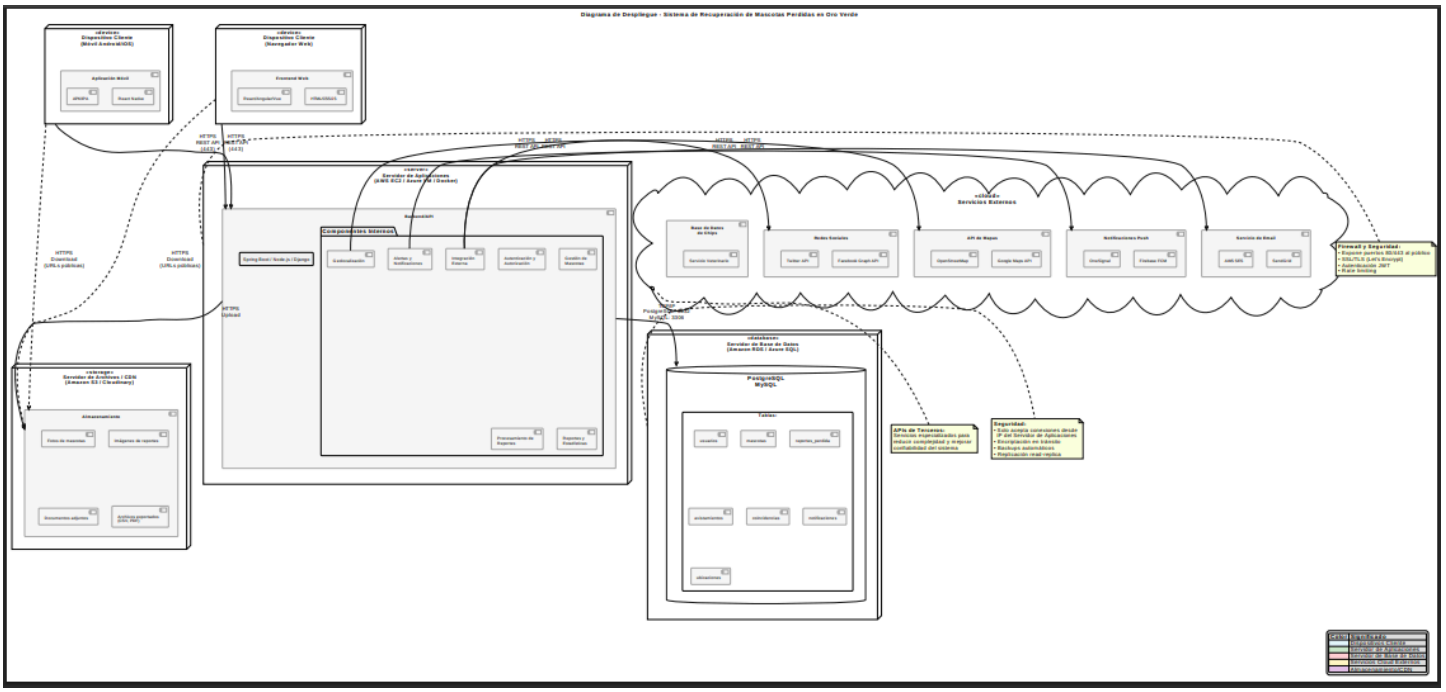
Backend:

- Node.js v19+ - Runtime de JavaScript
- Express.js 4.18.2 - Framework web
- Sequelize 6.35.0 - ORM para PostgreSQL
- PostgreSQL 14 - Base de datos relacional
- JWT (jsonwebtoken 9.0.2) - Autenticación stateless
- bcryptjs 2.4.3 - Hash de contraseñas
- Axios 1.13.1 - Cliente HTTP para APIs externas
- Jest 29.7.0 - Framework de testing
- Supertest 7.1.4 - Testing de APIs HTTP

Frontend:

- React 19.1.1 - Librería UI
- Vite 5.4.2 - Build tool
- React Router DOM 7.9.5 - Enrutamiento
- Axios 1.13.1 - Cliente HTTP

Diagrama de despliegue completo - [link]



Testing

Planilla de casos de prueba con ejemplos de flujos principales y alternativos

Tests UC22

	Caso de prueba 1: Generar alerta de mascota perdida (UC22)	**Caso de prueba 2: Rechazar reporte si faltan campos obligatorios (UC22)**
Id	CP-22-01	CP-22-02
Caso de Uso Relacionado	UC22 - Generar alerta de mascota perdida	UC22 - Generar alerta de mascota perdida
Caso de Prueba	Generación automática de alerta al reportar mascota perdida	Validación de campos obligatorios al reportar pérdida
Descripción	Verifica que el sistema genere una alerta y notifique a las entidades correspondientes cuando se reporta una mascota perdida	Verifica que el sistema rechace reportes de pérdida cuando faltan datos obligatorios y no genere alertas en estos casos
Fecha	03 - 11 - 2025	03 - 11 - 2025
Área Funcional/ Subproceso	Módulo de alertas y notificaciones	Módulo de validación y alertas

Funcionalidad/ Característica	Generación de alerta de mascota perdida	Validación de datos de entrada para reportes
Datos/ Acciones de Entrada	ID mascota, estado = “perdida”, botón “Reportar pérdida”	Reporte incompleto: {mascotaId: UUID, latitud: -31.7833} Faltante: longitud (campo obligatorio) Endpoint: POST /api/reportes/perdida
Resultado Esperado	Se crea una alerta en el sistema, se almacena en la base de datos y se envía una notificación a vecinos registrados y a la municipalidad	Status HTTP: 400 (Bad Request) Mensaje de error indicando campo faltante. No se crea el reporte en la BD ni se genera la alerta.
Requerimientos de Ambiente de Pruebas	Node.js 18.20.8, servidor backend activo, base de datos PostgreSQL (Supabase), Jest 29.7.0 + Supertest 7.1.4, Variables de entorno configuradas (.env.test), Sequelize ORM configurado	Node.js 18.20.8, servidor backend activo, base de datos PostgreSQL (Supbase), Jest 29.7.0 + Supertest 7.1.4, Variables de entorno configuradas (.env.test), Sequelize ORM configurado
Precondiciones	La mascota debe estar registrada y marcada como perdida y el usuario debe estar autenticado	El usuario debe estar autenticado, la mascota registrada y el contador de alertas conocido antes de la petición
Dependencias con otros casos de prueba	Depende de autenticación (JWT)	Depende de autenticación
Resultado obtenido	PASS	PASS
Estado	Aprobado	Aprobado
Última Fecha de Estado	04 - 11 - 2025	04 - 11 - 2025
Observaciones	Validar que las notificaciones se registren correctamente en log del sistema	Test automatizado en mismo archivo /server/tests/uc22-generar-alerta.test.js Validación implementada en middleware/controller Previene generación de alertas con datos incompletos y mejora la robustez del sistema
Origen Presunto	Fase de integración entre backend y servicio de notificaciones	Fase de validación de entrada de datos

Tests UC23

	Caso de prueba 1: Consultar ubicación a sistema de geolocalización (UC23)	**Caso de prueba 2: Manejo de error si la API de geolocalización falla (UC23)**
Id	CP-23-01	CP-23-02
Caso de Uso Relacionado	UC23 - Consultar ubicación a sistema de geolocalización	UC23 - Consultar ubicación a sistema de geolocalización
Caso de Prueba	Consulta de ubicación a sistema de geolocalización	Manejo de errores cuando el servicio de geolocalización no está disponible
Descripción	Verificar que el sistema obtenga correctamente la ubicación de una mascota en tiempo real desde el servicio de geolocalización	Verificar que el sistema maneje correctamente los errores cuando la API externa de geolocalización falla o no está disponible
Fecha	03 - 11 - 2025	03 - 11 - 2025
Área Funcional/ Subproceso	Módulo de geolocalización	Módulo de geolocalización - Manejo de excepciones
Funcionalidad/ Característica	Consulta de ubicación en mapa	Resiliencia ante fallos de servicios externos
Datos/ Acciones de Entrada	Coordenadas: latitud y longitud captadas del dispositivo GPS	Latitud: -31.783, longitud: -60.516, API externa mockeada para retornar error, Error simulado: "Servicio no disponible"
Resultado Esperado	El sistema muestra en pantalla la dirección correspondiente a las coordenadas y la posición de la mascota en el mapa	La función debe lanzar una excepción. Mensaje de error: "Servicio no disponible". No debe retornar datos parciales o corruptos. El error debe ser capturado y propagado correctamente
Requerimientos de Ambiente de Pruebas	API o microservicio de geolocalización disponible, conexión a internet estable	Node.js 18.20.8, Jest 29.7.0, Axios mockeado con mockRejectedValue(), Async/await error handling
Precondiciones	La mascota debe tener la geolocalización habilitada	API externa configurada para fallar (mock) y sistema preparado para capturar excepciones asíncronas
Dependencias con otros casos de prueba	Independiente	Independiente, prueba caso de error. Complementa CP-23-01 (caso exitoso)
Resultado obtenido	PASS	PASS
Estado	Pendiente/ En ejecución/ Aprobado	Aprobado

Última Fecha de Estado	04 - 11 - 2025	04 - 11 - 2025
Observaciones	Validar la exactitud de la dirección devuelta frente a las coordenadas simuladas	Garantiza que errores de API externa no crashean la aplicación. Importante para UX: permite mostrar mensaje de error al usuario. Permite implementar estrategias de retry o fallback en el futuro
Origen Presunto	API externa real no especificada (genérica, puede ser Google Maps, OpenStreetMap, etc.)	Fase de integración con API externa - Manejo de excepciones

Pruebas automatizadas y resultados

Las pruebas automatizadas se realizaron con Jest en Node.js

Para UC22

```
const request = require('supertest');
const app = require('../src/index');
const { sequelize, User, Mascota, EstadoMascota, ReporteMascota, Ubicacion, Alerta, Notificacion } =
require('../src/models');

describe('UC22 - Generar alerta de mascota perdida', () => {
  let authToken;
  let duenoId;
  let mascotaId;
  const timestamp = Date.now();
  const duenoEmail = `test.dueno.${timestamp}@test.com`;
  const vecinoEmail = `test.vecino.${timestamp}@test.com`;

  // Setup: Crear usuario dueño y mascota antes de los tests
  beforeAll(async () => {
    // Sincronizar base de datos (sin force: true para no eliminar tablas en uso)
    await sequelize.sync();

    // Crear usuario dueño
    const dueno = await User.create({
      nombre: 'Test',
      apellido: 'Dueño',
      email: duenoEmail,
      password: 'password123',
      celular: '3415551234',
      rol: 'dueno',
      permitirVisualizacionDatos: true,
      activo: true
    });
    duenoId = dueno.id;

    // Crear vecino para notificaciones
```

```
  await User.create({
    nombre: 'Test',
    apellido: 'Vecino',
    email: vecinoEmail,
    password: 'password123',
    celular: '3415555678',
    rol: 'vecino',
    activo: true
  });

  // Login para obtener token
  const loginResponse = await request(app)
    .post('/api/auth/login')
    .send({
      email: duenoEmail,
      password: 'password123'
    });

  if (loginResponse.status !== 200 || !loginResponse.body.data?.token) {
    console.error('Login failed:', loginResponse.body);
    throw new Error(`Login failed with status ${loginResponse.status}`);
  }

  authToken = loginResponse.body.data.token;

  // Crear mascota
  const mascota = await Mascota.create({
    nombre: 'Max',
    raza: 'Labrador',
    especie: 'Perro',
    fotoUrl: 'https://example.com/max.jpg',
    tamano: 'grande',
    colores: 'Amarillo',
    chip: `TEST${timestamp}`,
    observaciones: 'Muy amigable',
    usuarioId: duenoId
  });
  mascotaId = mascota.id;

  // Crear estado inicial
  await EstadoMascota.create({
    mascotaId: mascota.id,
    estado: 'activa',
    razonCambio: 'Registro inicial'
  });
}, 15000); // Timeout de 15 segundos para el setup

// Cleanup: Limpiar base de datos después de los tests
afterAll(async () => {
  await sequelize.close();
});

// TEST 1: Reportar mascota perdida y generar alerta exitosamente
test('Debe reportar mascota perdida y generar alerta automáticamente', async () => {
```

```
const reporteData = {
  mascotaId: mascotaId,
  latitud: -31.7833,
  longitud: -60.5167,
  descripcionLugar: 'Plaza San Martín, Oro Verde',
  descripcion: 'Se perdió esta mañana cerca de la plaza'
};

const response = await request(app)
  .post('/api/reportes/perdida')
  .set('Authorization', `Bearer ${authToken}`)
  .send(reporteData);

// Verificar respuesta HTTP
expect(response.status).toBe(201);
expect(response.body.success).toBe(true);
expect(response.body.message).toContain('alerta generada');

// Verificar que se creó el reporte
expect(response.body.data.reporte).toBeDefined();
expect(response.body.data.reporte.tipoReporte).toBe('perdida');
expect(response.body.data.reporte.estadoReporte).toBe('activo');

// Verificar que se creó la alerta
expect(response.body.data.alerta).toBeDefined();
expect(response.body.data.alerta.mensaje).toContain('MASCOTA PERDIDA');
expect(response.body.data.alerta.mensaje).toContain('Max');
expect(response.body.data.alerta.mensaje).toContain('Labrador');

// Verificar que se generaron notificaciones
expect(response.body.data.alerta.notificacionesEnviadas).toBeGreaterThan(0);

// Verificar en base de datos que se creó el reporte
const reporteDB = await ReporteMascota.findOne({
  where: { mascotaId: mascotaId, tipoReporte: 'perdida' }
});
expect(reporteDB).not.toBeNull();
expect(reporteDB.estadoReporte).toBe('activo');

// Verificar que se actualizó el estado de la mascota a 'perdida'
const estadoActual = await EstadoMascota.findOne({
  where: { mascotaId: mascotaId },
  order: [['createdAt', 'DESC']]
});
expect(estadoActual).not.toBeNull();
expect(estadoActual.estado).toBe('perdida');
expect(estadoActual.razonCambio).toBe('Reportado como perdido');

// Verificar que se creó la alerta en base de datos
const alertaDB = await Alerta.findOne({
  where: { reporteId: reporteDB.id }
});
expect(alertaDB).not.toBeNull();
expect(alertaDB.tipoAlerta).toBe('mascota_perdida');
```

```
    expect(alertaDB.enviado).toBe(true);
    expect(alertaDB.descripcionMensaje).toContain('Max');

    // Verificar que se crearon notificaciones para vecinos
    const notificaciones = await Notificacion.findAll({
      where: { alertaId: alertaDB.id }
    });
    expect(notificaciones.length).toBeGreaterThan(0);
    expect(notificaciones[0].estado).toBe('pendiente');
    expect(notificaciones[0].canal).toBe('email');
  });

  // TEST 2: Rechazar reporte si faltan campos obligatorios
  test('Debe rechazar reporte de mascota perdida si faltan campos obligatorios', async () => {
    // Contar alertas antes del test
    const alertasAntes = await Alerta.count();

    const reporteIncompleto = {
      mascotaId: mascotaId,
      latitud: -31.7833
      // Falta longitud y descripcion
    };

    const response = await request(app)
      .post('/api/reportes/perdida')
      .set('Authorization', `Bearer ${authToken}`)
      .send(reporteIncompleto);

    // Verificar respuesta HTTP
    expect(response.status).toBe(400);
    expect(response.body.success).toBe(false);
    expect(response.body.message).toContain('campos obligatorios');

    // Verificar que NO se creó el reporte en base de datos
    const reportesCount = await ReporteMascota.count({
      where: { mascotaId: mascotaId, descripcion: null }
    });
    expect(reportesCount).toBe(0);

    // Verificar que NO se crearon alertas adicionales
    const alertasDespues = await Alerta.count();
    expect(alertasDespues).toBe(alertasAntes); // No debe haber alertas nuevas
  });
});
```

Para UC23

```
const axios = require('axios');
const geoService = require('../src/services/geolocalizacion.service');

// Mock de axios
```

```
jest.mock('axios');

describe('UC23 - Consultar ubicación a sistema de geolocalización', () => {
  // Test 1: Caso exitoso
  test(' debería devolver una dirección válida cuando la API responde correctamente', async () => {
    // Simulamos la respuesta de la API externa
    const mockResponse = {
      data: {
        direccion: 'Av. Los Eucaliptos 123, Oro Verde, Entre Ríos',
        latitud: -31.783,
        longitud: -60.516,
      },
    };

    // Mockeamos la respuesta de axios.post
    axios.post.mockResolvedValue(mockResponse);

    // Ejecutamos la función
    const resultado = await geoService.consultarUbicacion(-31.783, -60.516);

    // Verificamos el resultado
    expect(resultado.data.direccion).toBe('Av. Los Eucaliptos 123, Oro Verde, Entre Ríos');
    expect(resultado.data.latitud).toBeCloseTo(-31.783, 3);
    expect(resultado.data.longitud).toBeCloseTo(-60.516, 3);
  });

  // Test 2: Caso de error o fallo de API
  test(' debería manejar correctamente un error si la API de geolocalización falla', async () => {
    // Simulamos un fallo de la API
    axios.post.mockRejectedValue(new Error('Servicio no disponible'));

    // Verificamos que la función lance una excepción
    await expect(geoService.consultarUbicacion(-31.783, -60.516))
      .rejects
      .toThrow('Servicio no disponible');
  });
});
```

Conclusiones

Principales decisiones

Durante el desarrollo del sistema se tomaron decisiones clave que afectaron la arquitectura y la implementación final.

A nivel de arquitectura, se consolidaron los nombres y responsabilidades de los componentes para mantener la coherencia entre las vistas C4, los diagramas UML y los módulos del código.

Decisiones de Diseño Implementadas:

- Uso de UUIDs en lugar de IDs incrementales. Razón: Mayor seguridad, imposibilidad de enumerar registros Implementación: type: DataTypes.UUID, defaultValue: DataTypes.UUIDV4
- Hooks de Sequelize para hash de contraseñas. Centralizar lógica de seguridad en el modelo. Implementación: beforeCreate y beforeUpdate hooks
- Validaciones a nivel de modelo. Garantizar integridad de datos independientemente de la capa de presentación. Implementación: Validadores de Sequelize (validate: {...})

Dificultades encontradas y resolución

Una de las principales dificultades fue mantener la consistencia de nombres entre los diferentes artefactos (modelos, controladores, servicios y diagramas). En etapas iniciales, las entidades y funciones no seguían un mismo patrón, lo que generaba confusión entre los componentes del backend y la documentación. Para resolverlo se definió una nomenclatura estandarizada basada en la funcionalidad principal de cada módulo.
Otros errores encontrados fueron:
Al iniciar el proyecto, el archivo package.json contenía la configuración "type": "module", lo que indicaba el uso de ES Modules. Sin embargo, todo el código del backend estaba escrito usando la sintaxis CommonJS (require() y module.exports). Esto causó un error crítico al intentar ejecutar el servidor:
ReferenceError: require is not defined in ES module scope
Solución implementada:
Se eliminó la línea "type": "module" del package.json para mantener el proyecto en CommonJS y se realizaron los ajustes necesarios para mantener la consistencia.

Aprendizajes obtenidos y posibles mejoras futuras

El proyecto permitió afianzar el conocimiento en diseño de arquitecturas modulares, la aplicación del modelo C4 y la correspondencia entre componentes y clases UML.
Se destacó la importancia de mantener la coherencia entre la documentación y el código fuente, especialmente en proyectos colaborativos.
Entre las posibles mejoras futuras se incluye integración con Amazon S3 o Cloudinary para almacenamiento de imágenes (actualmente se almacenan como URLs/strings en BD, sin gestión real de archivos). También se podría incorporar un chatbot con NLP (procesamiento del lenguaje natural) que funcione como asistente virtual que ayuda a usuarios 24/7 y de esta forma reducir la carga administrativa y mejorar la accesibilidad.

Link al repo

https://github.com/AnaaSol/Sistema-de-recuperacion-de-mascotas-perdidas-en-Oro-Verde