1. Introducción

El proyecto **AnimalesMasquefa** surge como una respuesta tecnológica a la necesidad de digitalizar y modernizar la gestión de adopciones de animales en el municipio de Masquefa. La iniciativa busca facilitar tanto a los ciudadanos como a los gestores del refugio un acceso más directo, sencillo y eficiente a toda la información relacionada con los animales disponibles para adopción.

El sistema propuesto consiste en una aplicación web full-stack, desarrollada como Trabajo de Fin de Grado del Ciclo Superior de Desarrollo de Aplicaciones Web. Esta solución combina un frontend moderno basado en React con un backend robusto desarrollado con Django y Django REST Framework. El proyecto incorpora funcionalidades como:

- Listado y ficha individual de animales.
- Gestión de usuarios y autenticación.
- Solicitud de adopciones mediante formularios.
- Gestión de noticias, contacto y contenidos informativos.
- Panel de administración exclusivo para el personal del refugio.
- Notificaciones automáticas vía email.
- Gestión y almacenamiento de imágenes en la nube con Cloudinary.

Además de desarrollar una plataforma funcional, el proyecto también busca fomentar la adopción responsable, visibilizar animales en espera de hogar y ofrecer una herramienta sostenible, segura y escalable en el tiempo.

Esta documentación recoge el análisis, diseño, desarrollo, pruebas y conclusiones del proyecto, así como anexos técnicos y visuales que ayudan a comprender su funcionamiento y estructura.

2. Objetivos del proyecto

2.1. Objetivo general

Desarrollar una aplicación web full-stack que permita gestionar de forma integral la adopción de animales en el municipio de Masquefa. El sistema debe ofrecer funcionalidades de acceso público y privado, estar basado en tecnologías modernas, y cumplir con buenas prácticas de seguridad, usabilidad y rendimiento.

2.2. Objetivos específicos

- **Crear una API REST robusta y segura** que permita operaciones CRUD sobre entidades clave como animales, usuarios, noticias y solicitudes de adopción.
- **Desarrollar una interfaz frontend desacoplada**, construida con React, que consuma dicha API y ofrezca una experiencia de usuario moderna, responsive y accesible.
- **Diseñar y desplegar un sistema de autenticación mediante tokens JWT**, que permita la identificación segura de usuarios y la aplicación de permisos según su rol.
- **Implementar una zona administrativa** mediante el panel de Django y/o interfaces específicas, desde donde se puedan gestionar todos los elementos del sistema.
- **Permitir el envío de formularios de contacto y adopción**, con validación y respuesta automática mediante email templado.
- **Integrar almacenamiento en la nube (Cloudinary)** para manejar imágenes de animales, evitando carga en el servidor local y optimizando tiempos de carga.
- **Configurar el despliegue completo del sistema**, diferenciando entorno de desarrollo y producción, incluyendo gestión de variables de entorno.
- **Implementar mecanismos de auditoría y registros automáticos**, mediante señales (`signals.py`) y eventos que garanticen trazabilidad y control.
- **Aplicar medidas de throttling, validación y permisos** para proteger la API contra abusos o accesos no autorizados.
- **Documentar la API mediante Swagger (OpenAPI)** para facilitar su uso, mantenimiento y futuras ampliaciones.
- **Diseñar el sistema con criterios de escalabilidad horizontal**, permitiendo su despliegue en plataformas como Heroku (backend) y Netlify (frontend).

3. Análisis previo

3.1. Identificación de actores del sistema

Actor	Descripción
Usuario anónimo	Persona que accede al sitio web sin autenticación. Puede visualizar animales y noticias.
Usuario registrado	Usuario autenticado. Puede enviar formularios de adopción y contacto, y gestionar su perfil.
Administrador	Personal autorizado del refugio. Tiene acceso al panel de administración para gestionar animales, usuarios, noticias y solicitudes.

3.2. Requisitos funcionales

1. **Visualización de animales en adopción**
   - Listado general de animales.
   - Ficha detallada por animal.
2. **Gestión de adopciones**
   - Envío de solicitud de adopción.
   - Visualización y gestión de adopciones por parte del admin.
   - Notificaciones automáticas de aceptación/rechazo.
3. **Autenticación y gestión de usuarios**
   - Registro, inicio y cierre de sesión.
   - Recuperación de contraseña.

- Edición de perfil.

4. **Gestión de contenido (admin)**
   - Publicación de noticias.
   - Gestión de imágenes y datos de animales.
   - Envío de emails personalizados.
5. **Formulario de contacto**
   - Envío de mensaje desde frontend.
   - Recepción automática por email en backend.

3.3. Requisitos no funcionales

- **Rendimiento**: el sistema debe responder a peticiones REST en menos de 500ms bajo condiciones normales.
- **Escalabilidad**: la arquitectura debe permitir desplegar frontend y backend de forma independiente en servidores cloud.
- **Seguridad**:
  - Comunicación HTTPS.
  - Autenticación con JWT.
  - Protecciones CSRF y CORS configuradas.
  - Validaciones en frontend y backend.
- **Usabilidad**: diseño responsive con compatibilidad móvil.
- **Mantenibilidad**: uso de estándares de código, separación de capas (MVC, servicios, componentes).
- **Internacionalización**: preparado para mensajes en español, y posibilidad de traducción futura.

---

4. Diseño del sistema

El sistema ha sido diseñado aplicando principios de arquitectura desacoplada y escalable, dividiendo claramente las responsabilidades entre cliente (frontend), servidor (backend), y servicios externos (almacenamiento y notificaciones).

La implementación sigue un enfoque **modular y mantenible**, con foco en la **reutilización de componentes**, la **seguridad de las operaciones** y la **eficiencia del flujo de datos**.

4.1. Arquitectura general

El proyecto utiliza una arquitectura **cliente-servidor desacoplada**, basada en los siguientes componentes principales:

[Usuario] ↓ [Frontend en React] ↓ (HTTP/HTTPS - JSON) [API REST en Django] ↓ [PostgreSQL + Cloudinary + Servicios Email]

5. Modelado de Base de Datos

El modelado de datos constituye el esqueleto estructural de toda la aplicación. En el caso de **AnimalesMasquefa**, se ha optado por un enfoque orientado a objetos, utilizando el ORM de Django para definir modelos relacionales que se traducen directamente a esquemas de base de datos en PostgreSQL.

El diseño se fundamenta en los principios de normalización (hasta 3FN), extensibilidad y coherencia semántica. Además, se ha prestado especial atención a la trazabilidad de

los datos y la auditabilidad de las operaciones críticas, utilizando `signals`, timestamps automáticos y campos controlados por roles.

5.1. Entidades principales y atributos

A continuación se detallan las entidades claves y sus campos asociados:

 `Animal`

Representa a cada animal del refugio. Incluye información visual, sanitaria y de adopción.

Campo	Tipo	Descripción
`id`	UUID / AutoField	Identificador único.
`nombre`	CharField	Nombre del animal.
`especie`	CharField	Gato, perro, etc.
`raza`	CharField	Raza declarada.
`edad`	IntegerField	Edad aproximada.
`descripcion`	TextField	Texto descriptivo.
`imagen`	CloudinaryField	URL de la imagen en Cloudinary.
`estado`	CharField (choices)	Disponible, Adoptado, En revisión.
`fecha_creacion`	DateTimeField	Timestamp automático de alta.

 `Adopcion`

Gestiona el proceso de adopción y su historial.

Campo	Tipo	Descripción
`id`	UUID / AutoField	Identificador único.
`usuario`	ForeignKey a `CustomUser`	Usuario solicitante.
`animal`	ForeignKey a `Animal`	Animal solicitado.
`estado`	CharField (choices)	Pendiente, Aceptada, Rechazada.
`mensaje`	TextField	Comentarios adicionales del usuario.
`comentario_admin`	TextField (opcional)	Observaciones internas del personal.
`fecha_creacion`	DateTimeField	Alta automática.
`fecha_respuesta`	DateTimeField (null)	Fecha de resolución.

 `CustomUser`

Extiende el modelo `AbstractUser` de Django para personalización completa.

Campo	Tipo	Descripción

email	EmailField	Usado como identificador de inicio de sesión.
nombre_completo	CharField	Nombre real del usuario.
rol	CharField	Admin o Usuario.
foto_perfil	CloudinaryField	Imagen de perfil en Cloudinary.
is_active	BooleanField	Controla acceso al sistema.
fecha_registro	DateTimeField	Timestamp de creación.

 Noticia

Módulo para la publicación de novedades y campañas.

Campo	Tipo	Descripción
titulo	CharField	Título de la noticia.
contenido	TextField	Texto enriquecido (HTML).
imagen	CloudinaryField	Imagen asociada.
autor	ForeignKey a CustomUser	Solo rol admin.
fecha_publicacion	DateTimeField	Fecha automática.

 Contacto

Registra los mensajes enviados desde el formulario web.

Campo	Tipo	Descripción
nombre	CharField	Nombre del remitente.
email	EmailField	Correo del remitente.
mensaje	TextField	Texto del mensaje.
fecha_envio	DateTimeField	Timestamp automático.

5.2. Relaciones entre modelos

```
[CustomUser] 1---* [Adopcion] *---1 [Animal]
     |
     +---* [Noticia]
     |
     +---* [Contacto]
```

5.3. Validaciones y lógica de integridad

- Validación en `serializer` para evitar múltiples adopciones simultáneas de un mismo animal.
- Hook `post_save` para cambiar el estado del animal una vez la adopción es aceptada.
- Control de duplicidad de correos mediante `unique=True` en el modelo de usuario.
- Lógica en `signals.py` para log de auditoría (`audit.py`) de eventos clave.

5.4. Auditoría y trazabilidad

Se implementó una clase personalizada en `audit.py` conectada mediante `signals` para registrar eventos como:

- Creación de usuarios.
- Creación/modificación de animales.
- Cambios en el estado de una adopción.
- Publicación de noticias.

5.5. Migraciones y control de versiones

Todas las entidades se migraron usando el sistema de migraciones de Django. Se han incluido archivos de migración progresiva (de `0001_initial.py` a `0007_alter_customuser_foto_perfil.py`) lo que permite recrear la base de datos en cualquier momento en entorno nuevo.

5.6. Alternativas evaluadas

Alternativa	Motivo de descarte
Firebase + Firestore	Dificultad para consultas relacionales y dependencias en tiempo real innecesarias.
MongoDB	No se justifica modelo NoSQL, ya que las relaciones entre entidades son críticas.
MySQL	Compatibilidad menor con ciertas librerías y preferencia del equipo por PostgreSQL.

6. Casos de uso

En esta sección se detallan los distintos **casos de uso** identificados durante la fase de análisis funcional del sistema **AnimalesMasquefa**. Cada caso de uso describe una interacción específica entre los actores del sistema y la plataforma, abarcando tanto la experiencia del usuario final como la operativa administrativa. Se ha seguido un enfoque centrado en el usuario, priorizando la usabilidad, seguridad y claridad de los flujos.

6.1. Metodología

Se ha adoptado el enfoque de **modelado UML simplificado** para describir los casos de uso, complementado con diagramas textuales, flujos alternativos y escenarios de error. La documentación de cada caso de uso incluye:

- **Nombre del caso de uso**
- **Actor/es involucrado/s**
- **Precondiciones**
- **Flujo principal**
- **Flujos alternativos**
- **Postcondiciones**
- **Errores esperados / excepciones**

6.2. Casos de uso públicos (usuarios no autenticados)

 CU01 – Visualizar animales en adopción

- **Actor:** Usuario anónimo
- **Precondiciones:** Ninguna
- **Flujo principal:**
  1. El usuario accede a la página de inicio.
  2. El sistema carga el listado de animales disponibles (consumo vía API REST).
  3. El usuario puede aplicar filtros por especie, estado, edad.
  4. Al hacer clic en un animal, se muestra la ficha detallada.
- **Flujos alternativos:**
  - Filtro por especie sin resultados → mensaje: "No hay animales disponibles con estos criterios".
- **Postcondiciones:** El usuario ha podido visualizar información útil para valorar una adopción.
- **Errores esperados:** Fallo de conexión con backend → mensaje: "Error al cargar animales".

 CU02 – Enviar formulario de contacto

- **Actor:** Usuario anónimo
- **Precondiciones:** El usuario debe completar todos los campos requeridos.
- **Flujo principal:**
  1. El usuario accede a la sección "Contacto".
  2. Rellena el formulario con su nombre, email y mensaje.
  3. Envía el formulario.
  4. El backend recibe y valida los datos.
  5. Se envía un correo automático al administrador.
  6. Se muestra un mensaje de éxito al usuario.
- **Flujos alternativos:**
  - Email inválido → se muestra error de validación.
- **Postcondiciones:** El mensaje ha sido almacenado y reenviado correctamente.
- **Errores esperados:** Timeout SMTP → reintento automático con backoff exponencial.

6.3. Casos de uso autenticados (usuario registrado)

 CU03 – Registrarse en la plataforma

- **Actor:** Usuario anónimo
- **Precondiciones:** No estar autenticado
- **Flujo principal:**
  1. El usuario accede a la vista de registro.

2. Completa email, nombre, contraseña.
3. El backend verifica que el email no esté en uso.
4. Se crea un nuevo usuario con rol "usuario" y se envía correo de bienvenida.
5. Se redirige al login.

- **Flujos alternativos:**
  - Email ya registrado → mensaje de error específico.
- **Postcondiciones:** Usuario creado y listo para autenticarse.
- **Errores esperados:** Validación incorrecta de contraseña → feedback dinámico.

 CU04 – Solicitar adopción de animal

- **Actor:** Usuario registrado
- **Precondiciones:** Sesión iniciada
- **Flujo principal:**
  1. El usuario visualiza un animal disponible.
  2. Pulsa en "Solicitar adopción".
  3. Rellena mensaje opcional.
  4. Se crea una instancia de `Adopcion` con estado "pendiente".
  5. El backend dispara un email al administrador.
  6. Se informa al usuario del envío exitoso.
- **Flujos alternativos:**
  - Ya existe una solicitud para ese animal → mensaje: "Ya has solicitado esta adopción".
- **Postcondiciones:** Solicitud registrada y en espera de revisión.
- **Errores esperados:** Error de API → reintento / feedback.

6.4. Casos de uso administrativos (admin)

 CU05 – Aprobar/rechazar solicitud de adopción

- **Actor:** Administrador
- **Precondiciones:** Sesión admin activa
- **Flujo principal:**
  1. El admin accede al panel de "Adopciones pendientes".
  2. Selecciona una solicitud.
  3. Visualiza detalle completo.
  4. Decide aceptar o rechazar.
  5. Se actualiza el estado en la base de datos.
  6. Se dispara el envío de correo automático con plantilla (`adopcion_aceptada.html` o `adopcion_rechazada.html`).
  7. Si es aceptada, el animal se marca como "Adoptado" automáticamente.
- **Flujos alternativos:**

    - La solicitud ya fue gestionada → mensaje de conflicto.
- **Postcondiciones:** Solicitud procesada, usuario informado.
- **Errores esperados:** Error de red o fallo de plantilla → fallback a texto plano.

 CU06 – Publicar noticia

- **Actor:** Administrador
- **Precondiciones:** Sesión admin activa
- **Flujo principal:**
    1. El admin accede a la sección "Noticias".
    2. Pulsa en "Nueva noticia".
    3. Introduce título, contenido e imagen.
    4. Guarda y publica.
    5. El sistema registra la noticia y la expone al frontend vía API.
    6. Se envía email informativo (si está habilitado).
- **Flujos alternativos:**
    - Imagen inválida → rechazo inmediato y validación frontend.
- **Postcondiciones:** Noticia visible públicamente.
- **Errores esperados:** Fallo de subida a Cloudinary → reintento / log.

Este desglose permite entender en detalle las interacciones clave entre usuarios y sistema, lo que facilita la validación de requisitos, el desarrollo dirigido por pruebas (TDD) y la planificación de futuras mejoras o integraciones externas.

7. Tecnologías utilizadas

La selección tecnológica del proyecto **AnimalesMasquefa** ha sido cuidadosamente planificada para garantizar un equilibrio entre robustez, mantenibilidad, rendimiento y facilidad de desarrollo. El stack elegido responde a una arquitectura moderna desacoplada, lo cual permite una escalabilidad horizontal y favorece el mantenimiento independiente del frontend y backend.

7.1. Visión general del stack

Capa	Tecnología	Rol principal
Frontend	React.js	Construcción de la interfaz de usuario (SPA)
Backend	Django + Django REST Framework	Lógica de negocio, API REST, autenticación
Base de datos	PostgreSQL	Almacenamiento estructurado relacional
Cloud Media	Cloudinary	Gestión y optimización de imágenes en la nube

Despliegue	Render (backend) / Netlify (frontend)	Infraestructura en la nube para entornos productivos
Email	SMTP (Gmail API / SendGrid)	Notificaciones automáticas
Documentación API	Swagger (drf-yasg)	Exploración y documentación automática de endpoints
Control de versiones	Git + GitHub	Seguimiento de cambios y trabajo colaborativo
Testing	Django TestCase	Pruebas automáticas del backend

7.2. Frontend – React

Se ha utilizado **React** por su capacidad para construir aplicaciones de una sola página (SPA), su ecosistema maduro y la posibilidad de dividir la interfaz en componentes reutilizables.

- Proyecto estructurado por **componentes funcionales** y **hooks** (`useState` , `useEffect` , `useContext`).
- Navegación mediante `react-router-dom` .
- Control de sesión vía `localStorage` y contexto global (`AuthContext`).
- Estilos propios + CSS modularizado por componente.
- Gestión de errores y loaders visuales con animaciones Lottie (`GlobalLoader.js`).

Ventajas clave:

- Reducción de recargas completas.
- Experiencia fluida y rápida.
- Componentización facilita el mantenimiento futuro.

7.3. Backend – Django y Django REST Framework

Django ha sido elegido por su rapidez de desarrollo, seguridad incorporada y comunidad activa. La API REST se expone mediante Django REST Framework (DRF), lo que permite desacoplar completamente la lógica de negocio del cliente.

- Uso de modelos personalizados (`CustomUser` , `Animal` , `Adopcion`).
- Serialización avanzada con `ModelSerializer` , validaciones personalizadas.
- Autenticación con JWT (`SimpleJWT`).
- Control de permisos con `IsAdminUser` , `IsAuthenticated` , y permisos personalizados (`permissions.py`).
- Throttling y rate-limiting mediante `throttles.py` .
- Emails automáticos generados desde `email.py` (HTML + texto plano).
- Uso extensivo de señales (`signals.py`) y auditoría (`audit.py`).

Módulos clave utilizados:

- `django-cors-headers` para permitir solicitudes entre dominios.
- `drf-yasg` para la documentación Swagger.
- `django-environ` para configuración de variables de entorno.

7.4. Base de datos – PostgreSQL

El sistema utiliza **PostgreSQL**, una base de datos relacional robusta y ampliamente adoptada en producción.

- Integración nativa con Django.
- Soporte completo para tipos avanzados, transacciones y constraints.
- Buen rendimiento con múltiples relaciones y filtros complejos.

Justificación:

- Mayor compatibilidad con plataformas como Render.
- Recomendación oficial de Django para entornos de producción.

7.5. Cloudinary – Almacenamiento multimedia

Cloudinary gestiona todas las imágenes del sistema, tanto de animales como de perfiles y noticias.

- Integración con `cloudinary_storage`.
- Subida automática desde el backend mediante comandos de administración (`migrar_archivos_a_cloudinary.py`).
- URLs optimizadas para dispositivos móviles.
- Conversión automática de formatos (WebP, AVIF) y lazy loading.

Ventajas:

- Reducción de carga del servidor.
- CDN global integrada.
- Panel de control visual para gestión manual.

7.6. Despliegue: Render y Netlify

- **Render** gestiona el backend.
  - Configuración del servidor WSGI con Gunicorn definida en `render.yaml`.
  - Variables de entorno y base de datos PostgreSQL como servicio configurado directamente en el panel de Render.
  - Despliegue automático conectado con GitHub.
- **Netlify** aloja el frontend.
  - Configuración de build directamente desde el panel de control.
  - Redirecciones controladas mediante el archivo `_redirects`.
  - Integración con repositorio GitHub para CI/CD.

Configuraciones clave:

- `.env.example` para variables compartidas entre entornos.
- Separación clara entre `DEBUG=True` y `DEBUG=False`.

7.7. Testing

- **Backend:**
  - Uso de `TestCase` para validación de modelos, vistas y flujos funcionales.

Objetivo: Minimizar regresiones y facilitar mantenimiento futuro.

7.8. Git y GitHub

Todo el código fuente se encuentra bajo control de versiones con Git, hospedado en GitHub.

Esta elección tecnológica busca asegurar que el sistema sea escalable, seguro, accesible y fácil de mantener a largo plazo, con posibilidad de ampliación modular en futuras fases.

8. Implementación

La implementación de **AnimalesMasquefa** fue cuidadosamente estructurada para garantizar la claridad, modularidad y eficiencia del desarrollo. Este proceso abarca desde la planificación inicial hasta la puesta en producción, y se apoya en prácticas reales observadas en el repositorio, sin asumir herramientas ni estructuras inexistentes.

8.1. Metodología de desarrollo

El desarrollo se organizó de forma iterativa, aplicando principios de desarrollo ágil y versiones controladas mediante Git. Las decisiones y ciclos fueron registrados en `README.md` y la estructura del repositorio muestra una clara separación de responsabilidades:

- Desacoplamiento de frontend y backend.
- Documentación clara en archivos como `.env.example`, `netlify.toml` y `requirements.txt`.
- Scripts de administración para poblado de datos (`seed_real_data.py`) y migración a Cloudinary (`migrar_archivos_a_cloudinary.py`).

8.2. Organización del código fuente

Backend (Django)

Ubicado bajo `animalesmasquefa/`, el backend contiene todo lo necesario para gestionar la lógica del sistema, la API REST, la autenticación, y la interacción con servicios externos como Cloudinary y el envío de correos.

Estructura principal:

```
animalesmasquefa/
├── animalesmasquefa/      # Configuración del proyecto Django
│   ├── settings.py        # Configuraciones generales y de producción
│   ├── urls.py            # Rutas principales del backend
├── appmustafa/            # Aplicación principal con la lógica de negocio
│   ├── models.py          # Modelos: Animal, Adopcion, CustomUser, Comentario,
Noticia
│   ├── serializers.py     # Serializadores DRF
│   ├── views.py           # Vistas y endpoints
│   ├── permissions.py     # Reglas de acceso por roles
│   ├── signals.py         # Señales para auditoría automática
│   ├── audit.py           # Registro interno de eventos sensibles
```

```
│   ├── email.py           # Utilidades para enviar correos templados
│   ├── throttles.py       # Limitadores de peticiones DRF
│   ├── urls.py            # Enrutamiento modular por app
│   ├── tests.py           # Casos de prueba automatizados con Django TestCase
│   ├── management/        # Comandos custom para gestión de datos
│   └── templates/email/   # Plantillas HTML para emails
```

Decisiones clave:

- Todos los modelos principales están en una sola app para facilitar relaciones entre entidades.
- La autenticación se gestiona mediante tokens JWT.
- Se utilizan `signals` para generar auditorías cada vez que se crea, modifica o elimina contenido sensible.
- El uso de comandos propios (`management/commands`) permite una automatización parcial del entorno y datos.

Frontend (React)

El código fuente del frontend se encuentra en `frontend/src/`, construido con React.js utilizando JSX, CSS modular y organización basada en componentes.

Estructura principal:

```
frontend/
├── public/                # Archivos estáticos y metadatos
│   └── _redirects         # Redirección para SPA en Netlify
├── src/
│   ├── pages/             # Vistas principales (routing)
│   ├── components/        # Componentes genéricos reutilizables
│   ├── contexts/          # Gestión de estado global (AuthContext)
│   ├── services/          # Comunicación con la API mediante fetch/axios
│   ├── App.js             # Componente raíz
│   └── index.js           # Entrada principal de la app
```

Observaciones clave:

- Se utiliza `AuthContext.js` para mantener el estado de usuario global.
- El archivo `_redirects` asegura funcionamiento de rutas en producción.
- Las conexiones a la API están organizadas por entidad: `animalService.js`, `usuarioService.js`, etc.

8.3. Ciclo de desarrollo real

1. **Análisis**
    - Identificación de usuarios, casos de uso y reglas de negocio.
    - Diseño del modelo de datos inicial y endpoints clave.
2. **Desarrollo paralelo**
    - Backend: se desarrollan modelos, serializadores, vistas, permisos.
    - Frontend: se construyen páginas principales y se integran formularios con validación.
3. **Integración**

- Comunicación entre React y Django REST Framework.
- Pruebas manuales con solicitudes reales desde frontend.

4. **Automatización parcial**
   - Comandos de Django (`migrar_archivos_a_cloudinary.py`, `seed_real_data.py`).
   - Señales conectadas a eventos críticos (ej: creación de usuario, envío de emails).
5. **Pruebas básicas**
   - Tests en `tests.py` validados con `TestCase`.
   - Validaciones funcionales desde el cliente web.
6. **Despliegue**
   - Backend en Render.
   - Frontend en Netlify.
   - Variables separadas por entorno usando `.env.example`.

8.4. Seguridad aplicada

- Todos los endpoints protegidos con JWT y permisos DRF.
- Validación tanto en backend como frontend.
- Throttling de peticiones automatizado.
- Sanitización de entradas del usuario.
- Protección de archivos y credenciales sensibles con `.gitignore`.

8.5. Configuración de entorno local

Backend:

```
cd animalesmasquefa
python manage.py runserver
```

Frontend:

```
cd frontend
npm install
npm run dev
```

Variables de entorno compartidas se encuentran documentadas en:

- `frontend/.env.example`
- `animalesmasquefa/animalesmasquefa/.env.example`

Esta fase de implementación refleja un desarrollo funcional y adaptado a un entorno real, documentado con claridad y con división efectiva de responsabilidades. El sistema está preparado para futuras mejoras y nuevas funcionalidades, manteniendo una base estable y escalable.

9. Backend (Django)

El backend del proyecto **AnimalesMasquefa** está construido con el framework Django, complementado con Django REST Framework (DRF) para exponer una API RESTful robusta, segura y modular. Esta sección profundiza en su estructura, los principales módulos desarrollados, y las funcionalidades clave implementadas.

9.1. Estructura del backend

La lógica del backend se encuentra centralizada en la app llamada `appmustafa`, y está configurada dentro del directorio raíz `animalesmasquefa/`.

```
animalesmasquefa/
├── animalesmasquefa/       # Configuración general del proyecto
│   ├── settings.py         # Configuración de Django, DRF, JWT, Cloudinary
│   ├── urls.py             # Rutas base, Swagger y administración
│   └── wsgi.py/asgi.py     # Entradas WSGI/ASGI para servidores
├── appmustafa/             # App principal con la lógica de negocio
│   ├── models.py           # Modelado de datos relacional
│   ├── serializers.py      # Serialización y validación DRF
│   ├── views.py            # Lógica de control para los endpoints
│   ├── permissions.py      # Permisos personalizados por rol
│   ├── signals.py          # Auditoría de acciones sensibles
│   ├── audit.py            # Registro estructurado de cambios
│   ├── throttles.py        # Limitación de peticiones
│   ├── email.py            # Envío de emails con plantillas HTML
│   ├── tests.py            # Pruebas unitarias con Django TestCase
│   ├── management/         # Comandos custom CLI
│   ├── templates/email/    # Plantillas para emails automáticos
│   └── urls.py             # Enrutado REST modular por recurso
```

9.2. Modelos principales

Los modelos están definidos en `models.py` y representan la estructura relacional de la base de datos. Los más relevantes son:

- **CustomUser**: modelo de usuario extendido con rol, foto de perfil y campos adicionales.
- **Animal**: representa a cada animal disponible para adopción, con imagen, edad, especie y descripción.
- **Adopcion**: formulario completado por el usuario final solicitando adoptar un animal.
- **Comentario**: asociado a noticias, permite una comunicación tipo blog.
- **Noticia**: publicaciones informativas visibles desde el frontend.

Las relaciones están normalizadas y utilizan claves foráneas para garantizar integridad referencial. Se han aplicado restricciones `on_delete=models.CASCADE` cuando corresponde.

9.3. Serializadores y validación

Los serializadores en `serializers.py` permiten transformar modelos a JSON y viceversa, incorporando validaciones personalizadas. Ejemplos:

- Validación de duplicidad en nombres de animales.

- Campos de solo lectura para fechas de creación.
- Uso de `depth=1` para relaciones anidadas (ej: usuario que solicita una adopción).

9.4. Vistas y endpoints

Las vistas están organizadas en `views.py` mediante clases heredadas de `APIView` o `GenericViewSet`. Se utilizan decoradores como `@action` para definir métodos personalizados (ej. aceptar/rechazar una adopción).

Ejemplos de endpoints definidos:

- `/api/animales/`
- `/api/adopciones/`
- `/api/usuarios/`
- `/api/noticias/`

La documentación de estos endpoints se genera automáticamente con Swagger (`drf-yasg`).

9.5. Seguridad: autenticación y permisos

- **JWT**: la autenticación se gestiona mediante JSON Web Tokens con `SimpleJWT`.
- **Permisos personalizados**:
  - `IsAdminUser`: acceso exclusivo para el panel administrativo.
  - `IsAuthenticated`: para rutas protegidas.
  - Custom: validación dinámica por rol o relación con la entidad (usuario que envió la adopción).

También se usa `django-cors-headers` para permitir solicitudes entre el frontend y la API.

9.6. Emails automáticos

Mediante el módulo `email.py` y las plantillas HTML en `templates/email/`, se generan correos automáticos en los siguientes eventos:

- Nueva solicitud de adopción.
- Aprobación o rechazo de adopciones.
- Nuevas noticias o publicaciones.
- Confirmación de contacto.

Los emails se configuran vía SMTP y se adaptan para dispositivos móviles.

9.7. Auditoría y señales

El sistema implementa una auditoría interna mediante señales de Django (`signals.py`), que registran eventos relevantes en `audit.py`.

Ejemplos registrados:

- Alta/baja de animales.
- Modificación de datos sensibles.
- Gestión de adopciones.

Esto permite trazabilidad y transparencia en operaciones clave.

9.8. Pruebas automáticas

En `tests.py` se han definido casos de prueba que validan el correcto funcionamiento del backend:

- Creación de animales y adopciones.
- Autenticación JWT válida y caducada.
- Permisos por rol y estado.
- Endpoints públicos vs privados.

Se ejecutan mediante `python manage.py test`.

El backend constituye el núcleo lógico y funcional del sistema, asegurando seguridad, coherencia de datos, trazabilidad y una API limpia y mantenible para el frontend. Está diseñado para ser extensible y escalable, permitiendo la incorporación de nuevos recursos y reglas de negocio en el futuro.

10. Frontend (React)

La interfaz de usuario de **AnimalesMasquefa** ha sido desarrollada como una **Single Page Application (SPA)** en **React.js**, con el objetivo de ofrecer una experiencia de usuario moderna, rápida y fluida. Esta sección describe en profundidad la estructura, patrones, lógica de interacción y diseño implementado en el cliente.

10.1. Estructura del frontend

El frontend se encuentra en el directorio `/frontend` y su código principal en `/frontend/src`. Su estructura se organiza por componentes y páginas, con separación clara entre la lógica, la vista y los servicios de comunicación con la API.

```
frontend/
├── public/                  # Archivos estáticos, favicon, HTML base
│   └── _redirects           # Reglas de rutas para Netlify
├── src/
│   ├── AdopcionEnviadaComponents/  # Componentes para feedback de adopción
│   ├── AdoptarComponents/          # Página de solicitud de adopción
│   ├── AnimalesComponents/         # Listado y visualización de animales
│   ├── Authentication/             # Registro, login y recuperación de contraseña
│   ├── ContactoComponents/         # Formulario de contacto
│   ├── DetalleAnimalComponents/    # Vista detallada de cada animal
│   ├── DetalleNoticiasComponents/  # Vista detallada de noticias
│   ├── HomeComponents/             # Home, navbar, footer, noticias destacadas
│   ├── NoticiasComponents/         # Listado de noticias
│   ├── PerfilComponents/           # Gestión del perfil de usuario
│   ├── PoliticaYTerminosComponents/ # Páginas legales
│   ├── SobreNosotros/              # Página informativa
│   ├── LoadingComponents/          # Animaciones de carga con Lottie
│   ├── contexts/                   # Contextos globales: Auth, Loading
│   ├── services/                   # Comunicación API: adopciones, usuarios,
noticias, etc.
│   ├── pages/                      # Rutas principales conectadas con React Router
│   ├── App.js                      # Componente raíz
│   └── index.js                    # Entrada principal
```

10.2. Componentes reutilizables y diseño modular

El sistema está construido mediante **componentes funcionales**. Cada página importa sus secciones desde subdirectorios, promoviendo la **modularidad** y el **reuso**. Ejemplos:

- `Navbar.js` y `Footer.js` se utilizan en todas las vistas principales.
- `DetalleAnimal.js` recibe props del router y muestra un animal concreto.
- `HomeCats.js`, `HomeForm.js`, `HomeNoticias.js` son secciones de la página principal.

Ventajas de este enfoque:

- Aislamiento de responsabilidades.
- Escalabilidad: fácil añadir nuevas vistas.
- Mejor mantenibilidad y comprensión del código.

10.3. Gestión de rutas

La navegación entre páginas se gestiona con **React Router**. El sistema permite rutas públicas y privadas:

- **Públicas:** Inicio, Noticias, DetalleAnimal, Registro, Login, Contacto.
- **Protegidas:** Perfil, EnviarAdopción (requieren sesión).

La protección se gestiona mediante el componente `privateRoute.js`, que comprueba si existe un token JWT válido en `localStorage`.

10.4. Estado global y contexto de autenticación

Se ha implementado un contexto de autenticación (`AuthContext.js`) que mantiene:

- Usuario actual (decodificando el JWT).
- Funciones de login, logout y verificación de sesión.

También se incluye un `LoadingContext.js` para controlar cargas globales con animaciones personalizadas de Lottie (`GlobalLoader.js`).

10.5. Comunicación con la API REST

El frontend interactúa con el backend mediante funciones asíncronas definidas en la carpeta `services/`. Cada archivo representa un dominio:

```
services/
├── adopcionService.js
├── animalService.js
├── comentarioService.js
├── loginService.js
├── noticiaService.js
├── profileService.js
├── usuarioService.js
└── privateRoute.js
```

Estas funciones usan `fetch` o `axios`, gestionan errores, y añaden encabezados JWT si el usuario está autenticado. Se centraliza el manejo de errores y la navegación

condicional en base a permisos.

10.6. Estilos y diseño visual

- Se emplea **CSS modularizado por componente**, lo que evita colisiones y mejora la organización.
- Animaciones ligeras y mensajes interactivos mejoran la experiencia del usuario.
- Uso de **Lottie JSON** para loaders, feedback visual de éxito/error, y páginas 404 personalizadas.

Responsive Design: todas las páginas están adaptadas a móvil y escritorio, usando media queries manuales.

10.7. Accesibilidad y usabilidad

- Navegación por teclado disponible en menús.
- Contrastado visual correcto para usuarios con visión reducida.
- Etiquetado de formularios con `htmlFor`, y mensajes de error accesibles.
- Comprobación de validaciones antes de enviar formularios al backend.

El frontend de **AnimalesMasquefa** está diseñado para ser intuitivo, accesible y veloz. Su arquitectura modular permite su evolución futura, integración de nuevas vistas o incluso migración a otras plataformas como React Native si se quisiera extender el alcance a dispositivos móviles.

11. API REST

El sistema **AnimalesMasquefa** expone una API RESTful completa, desarrollada con **Django REST Framework (DRF)**. Esta API permite al frontend interactuar con todos los recursos del sistema de manera desacoplada, cumpliendo con los principios REST (Stateless, recursos identificables por URI, uso de métodos HTTP estandarizados).

11.1. Estructura general

Todos los endpoints están organizados mediante **ViewSets** y **Routers** en el archivo `urls.py` dentro de la app `appmustafa`. Las rutas siguen el prefijo `/api/` para claridad y segmentación.

```
/api/
├── animales/               # CRUD de animales
├── adopciones/            # Gestión de solicitudes de adopción
├── usuarios/              # Registro, perfil y eliminación de cuenta
├── login/                 # Inicio de sesión (JWT)
├── noticias/              # CRUD de noticias informativas
├── comentarios/           # Comentarios asociados a noticias
├── contacto/              # Envío de formularios de contacto
├── auth/                  # Refresh y verificación de JWT
```

11.2. Ejemplo de endpoints

 Animales

- `GET /api/animales/` → Lista todos los animales disponibles (filtro por especie, estado).
- `GET /api/animales/{id}/` → Devuelve los detalles de un animal.
- `POST /api/animales/` → Solo admins. Crea un nuevo animal.
- `PUT/PATCH /api/animales/{id}/` → Modifica información.
- `DELETE /api/animales/{id}/` → Elimina el animal (con control de permisos).

Adopciones

- `POST /api/adopciones/` → Solicita una adopción (requiere autenticación).
- `GET /api/adopciones/usuario/` → Ver solicitudes del usuario actual.
- `PUT /api/adopciones/{id}/aceptar/` → Admin: acepta la solicitud.
- `PUT /api/adopciones/{id}/rechazar/` → Admin: rechaza la solicitud.

Usuarios y autenticación

- `POST /api/usuarios/` → Registro de nuevos usuarios.
- `POST /api/login/` → Obtiene token JWT.
- `POST /api/token/refresh/` → Refresca el token JWT.
- `GET /api/usuarios/perfil/` → Retorna los datos del usuario autenticado.
- `PUT /api/usuarios/perfil/` → Permite editar el perfil.
- `DELETE /api/usuarios/eliminar/` → Elimina su cuenta (autenticado).

11.3. Permisos, roles y seguridad

- **IsAuthenticated**: necesario para cualquier acción que involucre usuarios, adopciones o envío de formularios.
- **IsAdminUser**: controla acceso a endpoints sensibles (CRUD de animales, revisión de adopciones, publicación de noticias).
- **Permisos personalizados**:
  - Validan si un usuario puede acceder/modificar solo sus propios datos.
  - Previenen adopciones duplicadas del mismo animal.

Los tokens JWT se generan con `SimpleJWT` y se validan automáticamente en cada petición protegida mediante middleware.

11.4. Validaciones y respuestas

Los serializers integran validaciones personalizadas:

- Emails únicos.
- Campos obligatorios con errores específicos.
- Formatos de imagen (cuando se usan uploads a Cloudinary).
- Protección contra nombres de animales duplicados.

Ejemplo de error estándar:

```
{
  "email": ["Este correo ya está registrado."],
  "password": ["La contraseña debe tener al menos 8 caracteres."]
}
```

11.5. Paginación, filtros y rendimiento

La API implementa **paginación por defecto** con `PageNumberPagination`. Se retorna un máximo de 10 ítems por página.

Filtros soportados:

- En `/api/animales/`:
  - `?especie=Perro`
  - `?estado=Disponible`

La configuración de CORS (`django-cors-headers`) permite comunicación segura entre React y Django.

11.6. Documentación automática con Swagger

Usando `drf-yasg`, el backend expone una documentación interactiva en:

- `/swagger/` → UI con navegación interactiva de endpoints.
- `/redoc/` → Alternativa visual con vista jerárquica.

Esto permite:

- Probar endpoints desde el navegador.
- Compartir documentación fácilmente con nuevos desarrolladores.
- Validar tipos de entrada/salida sin leer el código fuente.

La API REST es el puente fundamental entre el frontend y el backend, permitiendo desacoplamiento completo, escalabilidad y facilidad de mantenimiento. Está diseñada para ser clara, segura, extensible y correctamente documentada, cumpliendo con estándares modernos de desarrollo web.

12. Autenticación y autorización

El sistema de **autenticación y autorización** de **AnimalesMasquefa** está diseñado para garantizar la seguridad, integridad y control de acceso en función del rol del usuario. Esta sección detalla cómo se implementa este sistema, incluyendo flujos de login, gestión de tokens, permisos condicionales y segregación de funcionalidades por perfil.

12.1. Sistema de autenticación con JWT

La autenticación se gestiona mediante **JSON Web Tokens (JWT)** usando la biblioteca `djangorestframework-simplejwt`. Esta elección permite un sistema sin sesiones (stateless), ideal para aplicaciones con frontend desacoplado como React.

Endpoints relevantes:

- `POST /api/login/` → recibe `email` y `password`, y retorna `access` y `refresh` tokens.
- `POST /api/token/refresh/` → dado un token de refresco válido, entrega un nuevo access token.

Configuración clave (`settings.py`):

```
SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=60),
```

```
    'REFRESH_TOKEN_LIFETIME': timedelta(days=1),
    'AUTH_HEADER_TYPES': ('Bearer',),
    'AUTH_TOKEN_CLASSES': ('rest_framework_simplejwt.tokens.AccessToken',),
}
```

12.2. Modelo de usuario personalizado (CustomUser)

Se ha extendido el modelo de usuario nativo de Django para adaptarlo a las necesidades del proyecto. El modelo `CustomUser` incluye:

- Campo `rol`: distingue entre usuarios normales y administradores (`choices`).
- Foto de perfil (`foto_perfil`) almacenada en Cloudinary.
- `is_active` y `is_staff` para validación y control de accesos.

Autenticación personalizada: Se utiliza un `AuthenticationBackend` modificado para validar por `email` en lugar de `username`.

12.3. Gestión de permisos (authorization)

El control de permisos se implementa mediante:

- **Decoradores estándar de DRF**: `@permission_classes([IsAuthenticated])`, `@permission_classes([IsAdminUser])`.
- **Clases personalizadas (`permissions.py`)**:
  - `EsPropietarioOAdmin`: permite acceder/modificar un objeto solo si el usuario es propietario o admin.
  - `SoloLecturaOAdmin`: restringe escritura a admins.

Estas clases se aplican directamente en las vistas (ViewSets o APIViews).

12.4. Protección de vistas y rutas

Backend:

- Las vistas públicas (GET animales, noticias, contacto) no requieren autenticación.
- Las vistas protegidas (crear adopción, ver perfil, eliminar cuenta) requieren token JWT.
- Se valida el rol y la identidad del usuario antes de acceder a cualquier objeto.

Frontend:

- Se utiliza `privateRoute.js` para proteger rutas de React.
- Los tokens se almacenan temporalmente en `localStorage`, y se validan en cada solicitud a la API.

12.5. Expiración y renovación de sesión

- Los **access tokens** expiran a los 60 minutos.
- Los **refresh tokens** permiten renovar sesión por 24 horas sin volver a iniciar sesión.
- El frontend detecta expiraciones e intenta renovar automáticamente mediante `refreshToken()`.

12.6. Flujo de login y logout

1. El usuario se autentica en `/api/login/`.
2. El backend valida email y password, y emite tokens JWT.
3. El frontend guarda el token, lo adjunta en cada petición (`Authorization: Bearer <token>`).
4. Si el token caduca, intenta refrescarlo.
5. Logout simplemente borra los tokens del navegador.

12.7. Eliminación segura de cuentas

El endpoint `DELETE /api/usuarios/eliminar/` permite al usuario eliminar su cuenta.

- Se valida que sea el propietario de la cuenta.
- Se invalidan sesiones y registros asociados.
- Se envía confirmación por email si corresponde.

12.8. Seguridad adicional

- CORS configurado para aceptar solo el dominio del frontend.
- Headers JWT verificados en cada petición.
- Throttling aplicado a autenticación y formularios.
- Protecciones CSRF no necesarias por usar JWT (stateless).
- `.gitignore` protege credenciales y variables sensibles.

Este sistema ofrece un balance entre **seguridad robusta** y **fluidez de usuario**, permitiendo segmentación de roles, control granular de accesos y compatibilidad con un frontend moderno desacoplado.

13. Almacenamiento multimedia (Cloudinary)

El sistema de almacenamiento multimedia de **AnimalesMasquefa** está basado en **Cloudinary**, un servicio en la nube optimizado para alojar, transformar y servir imágenes. Esta integración permite eliminar la dependencia del almacenamiento local, mejorar los tiempos de carga y facilitar el escalado del proyecto.

13.1. Motivación del uso de Cloudinary

El uso de almacenamiento local en aplicaciones modernas supone múltiples desventajas: dificultad para el escalado horizontal, gestión manual de backups, y riesgos de pérdida de datos. Cloudinary resuelve estos problemas proporcionando:

- URLs públicas optimizadas para web.
- Almacenamiento persistente en la nube.
- Transformaciones en tiempo real.
- Alta disponibilidad y redundancia.

Además, la gestión de imágenes en un refugio es un aspecto central de la aplicación, ya que las fotografías de los animales son el principal punto de atracción para los adoptantes.

13.2. Configuración en el backend (Django)

La configuración de Cloudinary se encuentra en `settings.py`, y está externalizada mediante variables de entorno para proteger credenciales sensibles:

```
CLOUDINARY_STORAGE = {
    'CLOUD_NAME': os.getenv('CLOUDINARY_CLOUD_NAME'),
    'API_KEY': os.getenv('CLOUDINARY_API_KEY'),
    'API_SECRET': os.getenv('CLOUDINARY_API_SECRET'),
}
DEFAULT_FILE_STORAGE = 'cloudinary_storage.storage.MediaCloudinaryStorage'
```

Estas variables están documentadas en `.env.example`.

Dependencias incluidas en `requirements.txt`:

- `cloudinary`
- `django-cloudinary-storage`

13.3. Uso en los modelos

Se han modificado los modelos `Animal` y `CustomUser` para almacenar imágenes en Cloudinary en lugar del sistema de ficheros local:

```
imagen = models.ImageField(upload_to='animales/', blank=True, null=True)
foto_perfil = models.ImageField(upload_to='usuarios/', blank=True, null=True)
```

Al estar configurado `DEFAULT_FILE_STORAGE`, Django automáticamente redirige estos campos a Cloudinary sin necesidad de lógica adicional.

13.4. Migración de archivos existentes

Se ha desarrollado un comando personalizado en `management/commands/migrar_archivos_a_cloudinary.py` que permite migrar imágenes antiguas almacenadas localmente a Cloudinary.

Este script automatiza:

- Revisión de rutas locales.
- Subida a Cloudinary con la librería oficial.
- Actualización del campo `imagen` del modelo con la nueva URL.

Este mecanismo evita la pérdida de contenido tras un despliegue en producción.

13.5. Visualización desde el frontend

El frontend simplemente consume las URLs generadas por Cloudinary y las renderiza mediante etiquetas `<img>` estándar.

Ejemplo:

```
<img src={animal.imagen} alt={animal.nombre} />
```

No se requiere lógica extra en el cliente para mostrar o transformar imágenes. Esto reduce complejidad, y Cloudinary se encarga de servir versiones optimizadas automáticamente.

13.6. Ventajas adicionales

- **Optimización automática**: Cloudinary adapta las imágenes a tamaños, calidad y formato ideales para cada dispositivo.
- **CDN integrada**: las imágenes se sirven desde una red global de distribución de contenido.
- **Escalabilidad**: nuevas imágenes no saturan el servidor ni afectan el rendimiento del backend.
- **Seguridad**: las imágenes privadas pueden restringirse mediante URLs firmadas (funcionalidad no usada actualmente pero extensible).

13.7. Mantenimiento

- El sistema no requiere limpieza manual: Cloudinary gestiona duplicados y expiraciones si se configura.
- El panel de Cloudinary permite navegar, buscar, y auditar el historial de subidas.

La integración con Cloudinary refuerza la fiabilidad, rendimiento y profesionalismo del sistema **AnimalesMasquefa**, asegurando que los recursos visuales estén siempre disponibles y optimizados, tanto en desarrollo como en producción.

14. Despliegue

El despliegue del proyecto **AnimalesMasquefa** se ha realizado mediante servicios cloud especializados para frontend y backend. Esta estrategia facilita la escalabilidad, separación de responsabilidades y optimización de recursos. A continuación se describen ambos entornos de despliegue utilizados: **Render** (backend) y **Netlify** (frontend), junto con la configuración de variables y medidas específicas de producción.

14.1. Despliegue del backend en Render

El backend basado en **Django + Django REST Framework** se aloja en **Render**, una plataforma moderna para aplicaciones web.

Estructura básica del despliegue:

- Se utiliza un **repositorio Git** conectado a Render.
- Render detecta los cambios en el repositorio y realiza deploy automático.
- Se define un archivo `Procfile` en la raíz de `animalesmasquefa/` para especificar el comando de arranque:

```
web: gunicorn animalesmasquefa.wsgi
```

Configuraciones clave:

- Archivo `requirements.txt` contiene todas las dependencias necesarias.
- `settings.py` detecta si se ejecuta en producción mediante variables de entorno:
  - `DEBUG=False`
  - `ALLOWED_HOSTS` configurado con dominio Render
  - CORS restringido para aceptar sólo peticiones del dominio Netlify

Variables de entorno definidas en Render (copiadas desde `.env.example`):

- Claves secretas (`SECRET_KEY` , `CLOUDINARY_*` , `EMAIL_*`)
- Configuración de base de datos y almacenamiento

Almacenamiento y archivos estáticos:

- Archivos estáticos gestionados con `whitenoise` .
- Imágenes servidas desde Cloudinary, por lo tanto no se requieren rutas estáticas para `MEDIA_ROOT` .

14.2. Despliegue del frontend en Netlify

La interfaz cliente en **React** se despliega en **Netlify**, una plataforma ideal para SPAs modernas.

Configuración base:

- Repositorio Git conectado directamente a Netlify.
- Script de build automático:

```
"scripts": {
  "build": "react-scripts build"
}
```

- Archivo `netlify.toml` en raíz del frontend configura el comportamiento de redirección:

```
[[redirects]]
  from = "/*"
  to = "/index.html"
  status = 200
```

Esto garantiza que la navegación en rutas internas del frontend funcione correctamente en producción (SPA routing).

Variables de entorno en Netlify:

- `REACT_APP_API_URL=https://<tu-backend-en-render>.onrender.com/api/`
- Otras variables documentadas en `frontend/.env.example`

Nota: Netlify reemplaza automáticamente las variables `REACT_APP_*` al momento del build. No deben cambiarse en tiempo de ejecución.

14.3. Seguridad y dominios cruzados (CORS)

- Se configura `django-cors-headers` para aceptar solo peticiones desde el dominio de Netlify.
- Se desactiva `DEBUG` en producción.
- Se usa HTTPS obligatorio.
- Tokens JWT se almacenan en el frontend temporalmente y se envían mediante `Authorization: Bearer <token>` .

14.4. Automatización y mantenimiento

- **Deploy automático**: tanto en Netlify como en Render, cada `push` a rama `main` dispara un nuevo despliegue.
- **Rollback**: ambas plataformas permiten revertir fácilmente a versiones anteriores si un deploy falla.
- **Logs**: disponibles desde el panel de cada plataforma.

14.5. Consideraciones de producción

- **Entornos separados**: se mantiene separación clara entre desarrollo local y despliegue.
- **Protección de secretos**: nunca se suben archivos `.env`, y se utilizan valores cifrados en ambas plataformas.
- **Gestión de errores**: se incluye una página `404` personalizada en el frontend, y manejo de excepciones en backend con logs controlados.

El sistema desplegado es **robusto, escalable y fácilmente mantenible**. El uso de Render y Netlify simplifica la operación en producción sin necesidad de infraestructura propia, permitiendo al equipo centrarse en nuevas funcionalidades y evolución del sistema.

15. Variables de entorno y configuración

La configuración del proyecto **AnimalesMasquefa** está cuidadosamente externalizada mediante archivos `.env.example`, tanto en el backend como en el frontend, garantizando una separación clara entre código y credenciales sensibles. Esto permite una configuración robusta y segura, adaptable a múltiples entornos (desarrollo, staging, producción).

15.1. Uso de variables en el backend (Django)

Ubicación: `animalesmasquefa/animalesmasquefa/.env.example`

Estas variables son cargadas automáticamente mediante `os.getenv()` en el archivo `settings.py`, y se usan para:

Configuración general:

- `SECRET_KEY` → clave criptográfica base de Django.
- `DEBUG` → activación del modo desarrollo.
- `ALLOWED_HOSTS` → dominios permitidos para evitar ataques host header.

Configuración de base de datos:

- `DB_ENGINE`, `DB_NAME`, `DB_USER`, `DB_PASSWORD`, `DB_HOST`, `DB_PORT`

Cloudinary:

- `CLOUDINARY_CLOUD_NAME`
- `CLOUDINARY_API_KEY`
- `CLOUDINARY_API_SECRET`

Envío de correos:

- `EMAIL_HOST`, `EMAIL_PORT`, `EMAIL_HOST_USER`, `EMAIL_HOST_PASSWORD`, `EMAIL_USE_TLS`

Estas claves se acceden desde `settings.py` con fallbacks seguros:

```
SECRET_KEY = os.getenv('SECRET_KEY', 'default-secret')
DEBUG = os.getenv('DEBUG', 'False') == 'True'
```

Seguridad avanzada:

- `CORS_ALLOWED_ORIGINS` → solo el dominio de Netlify.
- JWT settings también dependen de variables opcionales como tiempo de expiración.

15.2. Variables en el frontend (React)

Ubicación: `frontend/.env.example`

Las variables de entorno de React deben comenzar obligatoriamente por `REACT_APP_`.

Variables activas:

- `REACT_APP_API_URL` → URL base del backend (ej: `https://api.animalesmasquefa.com/api/`)
- `REACT_APP_GOOGLE_ANALYTICS_ID` (opcional)

Estas variables se usan en los servicios:

```
const API_URL = process.env.REACT_APP_API_URL;
```

Y son accesibles únicamente en **tiempo de compilación**, por lo que cualquier cambio requiere recompilar el frontend.

15.3. Buenas prácticas aplicadas

- **No se sube ningún archivo `.env` real** al repositorio gracias al `.gitignore`.
- **Se proveen plantillas `.env.example`** completas, para facilitar la configuración del entorno a otros desarrolladores.
- Las claves están documentadas en README.md y se explican en esta misma documentación.
- Se valida la existencia de variables críticas al iniciar el servidor, evitando errores silenciosos.

15.4. Ejemplo de variables en producción (Render + Netlify)

Render (backend):

```
SECRET_KEY=**************************
DEBUG=False
ALLOWED_HOSTS=.onrender.com
CLOUDINARY_CLOUD_NAME=masquefa
CLOUDINARY_API_KEY=****
EMAIL_HOST=smtp.gmail.com
EMAIL_PORT=587
EMAIL_HOST_USER=admin@masquefa.com
EMAIL_HOST_PASSWORD=****
EMAIL_USE_TLS=True
```

Netlify (frontend):

```
REACT_APP_API_URL=https://masquefa-backend.onrender.com/api/
```

La externalización de la configuración mediante variables de entorno garantiza un sistema profesional, seguro y escalable, listo para operar en distintos entornos sin tocar el código base.

16. Pruebas y validaciones

La fase de pruebas y validación en **AnimalesMasquefa** ha sido esencial para asegurar la estabilidad, fiabilidad y calidad del sistema. A continuación, se detallan las estrategias empleadas, los tipos de pruebas implementadas y los resultados obtenidos en base a la estructura real del proyecto.

16.1. Tests automatizados (backend)

El proyecto incluye un conjunto de **pruebas unitarias y de integración** implementadas en el archivo `appmustafa/tests.py`, utilizando `TestCase` de Django.

Principales pruebas implementadas:

- **Autenticación y login**
  - Verifica el inicio de sesión con JWT mediante `/api/login/`.
  - Comprueba la expiración y renovación de tokens.
- **Usuarios**
  - Registro correcto.
  - Acceso a perfil con y sin token.
  - Eliminación segura de cuenta (con protección por rol y autenticación).
- **Animales**
  - Creación, modificación y eliminación por parte del admin.
  - Listado público y fichas individuales.
- **Adopciones**
  - Envío correcto de solicitudes.
  - Validación de campos requeridos.
  - Gestión desde el panel de administración.

Estas pruebas se ejecutan con:

```
python manage.py test appmustafa
```

El entorno de test utiliza una base de datos temporal en memoria.

16.2. Pruebas funcionales (frontend y backend integrados)

Dado que el frontend está desacoplado, se han realizado **pruebas funcionales manuales** sobre la aplicación desplegada (Netlify + Render) simulando diferentes perfiles y flujos.

Casos validados:

- Registro, login, logout.

- Navegación entre rutas.
- Visualización de animales y noticias.
- Envío de formulario de adopción y contacto.
- Acceso a perfil de usuario.
- Cambios de contraseña y eliminación de cuenta.
- Control de accesos según rol (admin vs usuario común).

Estas pruebas fueron realizadas desde navegadores modernos (Chrome, Firefox, móvil) para verificar:

- Comportamiento responsive.
- Correcto manejo de errores.
- Redirecciones esperadas.
- Validación de formularios desde frontend.

16.3. Validaciones integradas

La validación de datos se realiza tanto en frontend como en backend:

Frontend:

- Validación de formularios (`required` , `pattern` , `minLength` , etc.).
- Control visual de errores.

Backend:

- Validación declarativa en modelos y serializadores DRF.
- Protección contra inyecciones y entradas maliciosas.

Ejemplo en `serializers.py` :

```
if len(data['mensaje']) < 10:
    raise serializers.ValidationError("El mensaje es demasiado corto.")
```

16.4. Throttling y seguridad contra abusos

En `throttles.py` se han definido políticas específicas para limitar el número de peticiones a endpoints sensibles:

- Límite de peticiones por IP a `/api/contacto/` y `/api/adopciones/` .
- Protección contra bots y automatizaciones abusivas.

Estas medidas refuerzan la validación activa del sistema ante intentos de abuso o spam.

16.5. Pruebas de despliegue

Tras cada despliegue en Netlify (frontend) o Render (backend), se realiza un chequeo de funcionalidad completo:

- Verificación del login.
- Acceso a todos los endpoints públicos.
- Validación de tokens JWT en producción.
- Carga y visualización de imágenes desde Cloudinary.
- Envío real de emails de contacto y adopción.

16.6. Resultados

El sistema ha demostrado un comportamiento **estable, coherente y seguro** en todos los flujos principales.

- Todos los tests automatizados pasan correctamente (`OK` en consola).
- Las pruebas funcionales reflejan un flujo de usuario fluido y sin errores críticos.
- No se han detectado fugas de datos ni accesos indebidos en producción.

Esta fase de validación ha permitido consolidar la aplicación como un sistema confiable y apto para el entorno real de gestión de adopciones. Su arquitectura modular y buenas prácticas garantizan mantenibilidad y calidad a largo plazo.

17. Manual de usuario

La aplicación **AnimalesMasquefa** está diseñada para ser utilizada por dos perfiles claramente diferenciados: el **usuario administrador**, encargado de la gestión completa del sistema, y el **usuario final**, que accede a las funcionalidades públicas y privadas relacionadas con la adopción y el contacto. A continuación, se detalla el funcionamiento para cada tipo de usuario, acompañado de ejemplos reales del flujo de navegación.

17.1. Usuario administrador

Acceso al panel de administración

- URL: `/admin/`
- Se accede con las credenciales creadas previamente en la consola o mediante migraciones.
- El panel utiliza Django Admin, mejorado visualmente con Jet.

Funcionalidades disponibles:

1. **Gestión de animales**
    - Crear, editar o eliminar registros de animales.
    - Subida de imágenes directamente integradas con Cloudinary.
    - Campos editables: nombre, raza, edad, descripción, estado de salud, etc.
2. **Gestión de usuarios**
    - Visualización de cuentas registradas.
    - Búsqueda por email o username.
    - Posibilidad de eliminar cuentas manualmente si es necesario.
3. **Gestión de noticias**
    - Publicación de entradas informativas visibles en el frontend.
    - Edición de contenido en formato enriquecido (uso de campos de texto HTML).
4. **Gestión de solicitudes de adopción**
    - Ver todas las solicitudes recibidas desde el frontend.
    - Aceptar o rechazar solicitudes, lo cual dispara correos automáticos personalizados.

5. **Auditoría automática**
    - Las acciones críticas como eliminación o modificación quedan registradas mediante `signals.py` y `audit.py`.
6. **Notificaciones por email**
    - Sistema templado de notificaciones: nuevos animales, nuevas noticias, solicitudes recibidas, etc.
    - Plantillas ubicadas en `templates/email/`.

Consejos para administradores:

- Evitar eliminar contenido sin respaldo.
- Usar las herramientas de búsqueda del admin para filtrar por nombre, especie o estado.
- Actualizar regularmente el contenido visible para mantener la plataforma dinámica.

17.2. Usuario final

El usuario final interactúa con la plataforma a través de una interfaz clara, responsiva y orientada a la experiencia de adopción.

1. Registro y autenticación

- Acceso a `/registro` para crear cuenta.
- Autenticación mediante `/login`.
- Recuperación de contraseña vía `/forgot-password` y `/reset-password`.

2. Visualización de animales

- Página `/animales` muestra el listado completo con filtros visuales.
- Acceso a `/animal/:id` para ver ficha detallada con foto, descripción, estado y botón de adopción.

3. Envío de solicitud de adopción

- Solo usuarios registrados pueden acceder al formulario.
- Ruta: `/adoptar` o desde botón en ficha del animal.
- Validación de campos obligatorios: motivo, situación personal, experiencia previa.
- Mensaje de confirmación visual y por email.

4. Gestión de su perfil

- Página `/perfil` permite:
    - Ver datos personales.
    - Cambiar contraseña.
    - Eliminar cuenta de forma definitiva.

5. Contacto

- Página `/contacto` accesible desde el menú.
- Formulario con nombre, email y mensaje.
- Envío mediante backend y confirmación visual + correo de respuesta automática.

6. Acceso a noticias y contenidos

- Página `/noticias` con artículos publicados por los administradores.

- Detalles en `/noticia/:id`.
- Sección de inicio `/` incluye últimas noticias, CTA para adoptar y testimonios.

7. Página de error personalizada

- Ruta `/404` muestra animación de gato y mensaje de error.
- Redirección automática si se intenta acceder a rutas no válidas.

17.3. Consideraciones de usabilidad

- Todos los formularios muestran validaciones en tiempo real.
- Navegación adaptada a dispositivos móviles.
- Transiciones suaves entre páginas.
- Feedback visual tras cada acción importante (adopción, contacto, login, etc.).

El sistema ha sido construido pensando tanto en la eficiencia del personal del refugio como en la empatía hacia los ciudadanos interesados en adoptar. La interfaz y funcionalidades ofrecen una experiencia clara, confiable y alineada con el objetivo principal: **fomentar la adopción responsable**.

18. Conclusión y mejoras futuras

El desarrollo de **AnimalesMasquefa** ha representado una experiencia integral en la creación de un sistema web completo, orientado tanto a necesidades reales como a buenas prácticas de desarrollo profesional. Este proyecto no solo cubre el objetivo inicial de facilitar la adopción de animales en Masquefa, sino que establece las bases para un sistema sostenible, seguro y escalable.

18.1. Conclusiones del proyecto

- **Arquitectura sólida y desacoplada**: la elección de Django + DRF para el backend y React para el frontend ha permitido un desarrollo modular y bien organizado, facilitando el mantenimiento y la futura evolución.
- **Experiencia de usuario cuidada**: se ha priorizado la claridad en los flujos, la validación inmediata, el diseño responsive y la empatía con el usuario final, favoreciendo un entorno cómodo y confiable.
- **Seguridad integrada desde el inicio**: autenticación JWT, políticas de permisos, throttling, validaciones, control de CORS y protección de datos sensibles mediante variables de entorno garantizan un sistema preparado para producción.
- **Uso de servicios externos clave**: la integración con Cloudinary, Render, Netlify y envío de correos SMTP ha permitido crear un ecosistema profesional sin necesidad de infraestructura propia.
- **Automatización parcial**: la inclusión de comandos personalizados (`seed_real_data.py`, `migrar_archivos_a_cloudinary.py`) y señales (`signals.py`) aporta trazabilidad y eficiencia operativa.
- **Validación rigurosa**: se han realizado pruebas unitarias y funcionales que aseguran la correcta ejecución de los flujos clave tanto en desarrollo como en producción.

18.2. Mejoras futuras

A pesar de su robustez, el sistema actual admite varias posibles ampliaciones o refactorizaciones que lo harían aún más completo:

Funcionales:

- **Sistema de comentarios moderados**: ampliar los comentarios existentes con posibilidad de respuesta, moderación por admin y valoraciones.
- **Panel frontend para administradores**: crear una interfaz administrativa React separada del Django Admin, más amigable para usuarios no técnicos.
- **Seguimiento de estado de adopciones**: permitir al usuario ver en tiempo real si su solicitud ha sido aceptada, pendiente o rechazada.
- **Historial de adopciones exitosas**: añadir una sección donde se muestren casos reales de adopciones, con testimonios y fotos.
- **Chat de consulta rápida**: integrar un sistema de mensajería o chatbot para resolver dudas frecuentes desde el frontend.

Técnicas:

- **Mejorar cobertura de tests**: ampliar las pruebas unitarias en backend y considerar herramientas como Cypress o Playwright para tests E2E en frontend.
- **Soporte multilingüe**: traducir la interfaz y contenidos al inglés y catalán.
- **Sistema de logs externos**: integrar herramientas como Sentry o LogRocket para monitorizar errores en producción.
- **Backups automáticos**: definir estrategia de respaldo de la base de datos en servicios cloud.
- **Versionado de API**: implementar `/api/v1/` y preparar la arquitectura para futuras versiones sin romper compatibilidad.

18.3. Reflexión personal

Este TFG ha representado un verdadero reto de análisis, diseño e implementación, acercando la práctica académica a un entorno real de desarrollo profesional. La aplicación no solo funciona como un portal web funcional, sino también como una herramienta con impacto social directo. La elección tecnológica, las decisiones arquitectónicas y la constante documentación son prueba del compromiso con la calidad del software.

La experiencia adquirida abarca desde diseño de APIs hasta despliegue continuo, pasando por integración con servicios externos, control de versiones, auditoría y pruebas automatizadas. En conjunto, **AnimalesMasquefa** no solo cumple con los objetivos académicos del TFG, sino que demuestra la capacidad de aplicar ingeniería de software con responsabilidad, visión a futuro y sensibilidad social.

19. Anexos

En esta sección se recopila contenido técnico y visual relevante para ampliar la comprensión del proyecto **AnimalesMasquefa**. Estos anexos incluyen fragmentos representativos de código, capturas de pantalla reales de la aplicación y enlaces a fuentes utilizadas en el desarrollo.

19.1. Código relevante

Modelo `Animal` (appmustafa/models.py)

```python
class Animal(models.Model):
    nombre = models.CharField(max_length=100)
    descripcion = models.TextField()
    edad = models.PositiveIntegerField()
    imagen = CloudinaryField('image', blank=True, null=True)
    disponible = models.BooleanField(default=True)
    especie = models.CharField(max_length=50)
    raza = models.CharField(max_length=100, blank=True)
    fecha_rescate = models.DateField()
    ...
```

Serializer de adopciones (`serializers.py`)

```python
class AdopcionSerializer(serializers.ModelSerializer):
    class Meta:
        model = Adopcion
        fields = '__all__'

    def validate_motivo(self, value):
        if len(value) < 10:
            raise serializers.ValidationError("Debes escribir un motivo más
detallado.")
        return value
```

Vista protegida con permisos personalizados (`views.py`)

```python
class AnimalViewSet(viewsets.ModelViewSet):
    queryset = Animal.objects.all()
    serializer_class = AnimalSerializer
    permission_classes = [IsAdminOrReadOnly]
```

Comando personalizado (`migrar_archivos_a_cloudinary.py`)

```python
class Command(BaseCommand):
    def handle(self, *args, **kwargs):
        for animal in Animal.objects.all():
            if animal.imagen:
                resultado = cloudinary.uploader.upload(animal.imagen.path)
                animal.imagen = resultado['public_id']
                animal.save()
```

Contexto de autenticación React (`AuthContext.js`)

```javascript
export const AuthContext = createContext();

export const AuthProvider = ({ children }) => {
  const [user, setUser] = useState(null);

  useEffect(() => {
    const token = localStorage.getItem("token");
    if (token) {
```

```
      const decoded = jwtDecode(token);
      setUser(decoded);
    }
  }, []);

  return (
    <AuthContext.Provider value={{ user, setUser }}>
      {children}
    </AuthContext.Provider>
  );
};
```

19.3. Referencias y fuentes utilizadas

- Django Official Documentation
- Django REST Framework
- React Documentation
- Cloudinary Python Docs
- Render Deployment Guide
- Netlify React Hosting
- JWT.io
- PostgreSQL Docs

Estos anexos refuerzan la transparencia, trazabilidad y comprensión del proyecto, facilitando su evaluación técnica y su posible reutilización o ampliación futura.