ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA INFORMÁTICA

Desarrollo de una aplicación web para la gestión de la Sociedad Protectora de Animales y Plantas de Málaga

Implementación de las funcionalidades del módulo de gestión de animales

Development of a web application for the "Sociedad Protectora de Animales y Plantas de Málaga"

Implementation of the functionalities of the animal management module

Realizado por
Rafael Moret Galán

Otros integrantes del grupo
Cristina Espejo Roque

Tutorizado por
Juan Antonio Falgueras Cano

Departamento
Departamento de Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, SEPTIEMBRE 2024

Fecha defensa: septiembre de 2024

UNIVERSIDAD
DE MÁLAGA

E.T.S. INGENIERÍA
INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Resumen

Este proyecto tiene como objetivo **desarrollar una aplicación web** que solucione de manera simple y óptima el trabajo de los voluntarios de la Protectora de Animales y Plantas de Málaga, una organización sin ánimo de lucro dedicada a la recogida, acogida y cuidado de animales. La herramienta propuesta facilitará tanto la **gestión de los animales** como la **administración del equipo de voluntarios**, centralizando las tareas clave en un solo sistema. De esta manera, se busca mejorar la eficiencia en las operaciones diarias y asegurar que los recursos de la protectora se utilicen de manera efectiva.

Este desarrollo se realizará en varias etapas, comenzando con un **análisis de la situación y las necesidades actuales** de la entidad, hasta llegar a la **implementación y construcción de un sistema** que garantice un rendimiento óptimo y una experiencia de usuario especialmente cuidada.

Esta aplicación incluirá una variedad de **casos de uso** para cubrir todas las necesidades de la protectora. Permitirá centralizar en una única plataforma la gestión del voluntariado, que incluye los perfiles de las personas implicadas en las adopciones y usuarios de la aplicación, la administración de cuadrantes horarios y la generación de informes de turnos. Asimismo, se ocupará de todo lo relacionado con el módulo de los animales, abarcando desde adopciones y acogidas hasta seguimientos, citas médicas, localización e información detallada de cada animal, entre otros aspectos.

Palabras clave: Desarrollo web, protectora de animales, voluntariado, plataforma de gestión

Abstract

This project aims **to develop a web solution** that simplifies and optimizes the work of the volunteers at the "Protectora de Animales y Plantas de Málaga", a non-profit organization dedicated to the rescue, shelter, and care of animals. The proposed tool will facilitate both the **management of animals and the administration of the volunteer team** by centralizing key tasks into a single system. The goal is to improve efficiency in daily operations and ensure that the shelter's resources are used effectively.

The development will be carried out in several stages, starting with an **analysis of the current situation and needs of the organization**, leading to the **implementation and construction of a system** that guarantees optimal performance and a carefully designed user experience.

This application will include a variety of **use cases** to cover all the needs of the shelter. It will allow for the centralization of volunteer management on a single platform, including the profiles of individuals involved in adoptions and users of the application, the administration of schedules, and the generation of shift reports. Additionally, it will handle everything related to the animal module, covering adoptions and fostering, follow-ups, medical appointments, location, and detailed information on each animal, among other aspects.

Keywords: Web development, animal shelter, volunteer work, management platform

Índice

Resumen 3

Abstract 4

Índice 5

Introducción 7
- 1.1 Motivación 7
- 1.2 Objetivos 8
- 1.3 Estructura de la memoria 10
- 1.4 Metodología 11
- 1.5 Estado del arte 12

Tecnologías 15
- 2.1 Stack tecnológico principal 15
- 2.2 Librerías destacadas 17

Definición de requisitos 21
- 3.1 Requisitos funcionales 21
- 3.2 Requisitos no funcionales 22
- 3.3 Casos de uso implementados 24

Diseño de la aplicación 35
- 4.1 Arquitectura 35
- 4.2 Modelo de datos 40

Desarrollo del software 43
- 5.1 Backend 43
- 5.2 Frontend 53
- 5.3 Módulos 65

Conclusiones 101
- 6.1 Líneas futuras 102

Referencias 105

Manual de Instalación 107

Guía de usuario 109
- Inicio de sesión 109
- Edición de perfil 110
- Ver el listado de animales 111
- Visualización de características de un animal 112
- Visualización y edición del orden de patio 115
- Listado de adopciones 116
- Creación del PDF del listado de animales por patio 118
- Visualización de las personas registradas en la aplicación 119
- Apuntarse a un turno 121
- Listado de informes 122
- Guía para coordinadores 126

Estructura de ficheros 133

B.1 Backend **133**
B.2 Frontend **134**

Índice de figuras **137**

1

Introducción

La redacción de esta memoria hace un repaso por todo el proceso de desarrollo del proyecto que hemos creado en conjunto los dos integrantes de este Trabajo de Fin de Grado. Pese a que cada uno de los integrantes ha trabajo en una parte diferente del proyecto, tanto en la implementación como en este propio documento hay múltiples partes comunes que acompañan a aquellas que muestran el trabajo individual de cada integrante.

Con el objetivo de especificar qué partes ha redactado cada integrante y cuáles forman parte del trabajo conjunto, insertamos aquí una pequeña descripción de los capítulos que muestra esta información.

Los capítulos *1 - Introducción*, *2 - Tecnologías*, *4 - Diseño de la aplicación* y los apéndices A, B y C forman parte de la redacción conjunta.

Por otro lado, el capítulo *5 - Desarrollo del software*, contiene los subcapítulos *5.1 - Backend* y *5.2 - Frontend*, que detallan las implementaciones comunes que se han dado durante el desarrollo para cada parte de la aplicación, formando así parte de la redacción común.
A su vez, del capítulo *3 - Definición de requisitos*, tan solo el subcapítulo *3.2 - Requisitos no funcionales* es común a ambos, ya que el resto del capítulo es completamente individual.

Por último, el capítulo *6 - Conclusiones* forma parte de la redacción individual, ya que cuenta el punto de vista de cada integrante sobre el trabajo en este proyecto.

1.1 Motivación

La Protectora de Málaga, formalmente conocida como la Sociedad Protectora de Animales y Plantas de Málaga, es una sociedad encargada de la defensa y protección de los animales que actúa a nivel local en nuestra ciudad. La Protectora ofrece servicios de rescate, cuidado y

refugio para los gatos y perros más necesitados, con el fin de permitir que encuentren un nuevo hogar gestionando y facilitando sus procesos de adopción.

Las labores necesarias para poder ofrecer estos servicios son llevadas a cabo por el cuerpo de voluntarios de la Protectora. Estos voluntarios realizan turnos semanalmente para proporcionar a los animales cuidados básicos como alimentación, limpieza, socialización y medicación, además de facilitar sus adopciones llevando a cabo la comunicación con los adoptantes.

En todo este proceso, los voluntarios cuentan con la ayuda de una pequeña web estática que sirve a modo de índice para acceder a un conjunto de ficheros Excel y Docs que usan a modo de base de datos. Esto les permite gestionar de forma muy limitada los datos de animales y voluntarios, la rotación de turnos y las adopciones. Entre algunos de los problemas con los que cuenta el sistema actual están: la duplicidad de los datos, la poca automatización de procesos repetitivos y tediosos, la falta de accesibilidad y la carencia de perfiles y roles de usuarios.

Al haber participado activamente como voluntarios en esta entidad, pudimos identificar claramente las deficiencias que ralentizaban y complicaban nuestras tareas. Esto nos motivó a colaborar en la creación de un software que centralizara todas las necesidades informáticas mediante la realización de este TFG. Como voluntarios, al desarrollar los casos de uso que consideramos necesarios y mantener una relación directa con los miembros administrativos y coordinadores, encontraríamos más sencillo y gratificante aprovechar esta oportunidad para contribuir a mejorar la eficiencia de la protectora en este ámbito.

1.2 Objetivos

El objetivo de este proyecto se centra en mantener en un mismo entorno, la posibilidad de gestionar tanto el módulo de gestión de animales como el módulo de la administración del voluntariado. Para cada uno de ellos, establecemos un conjunto de necesidades acordadas con la Protectora.

Para el módulo de animales, se requieren los siguientes casos de uso:

- **Almacenamiento, consulta y modificación de los datos de animales.** Esto incluye mostrar una fotografía de cada animal, sus datos de personalidad, características físicas y demográficas, así como la fecha de entrada al refugio.

- **Visualización y modificación de los distintos espacios del recinto.** La protectora está organizada en varias áreas o patios. Por ello, es esencial contar con una herramienta que permita ver de manera rápida y clara los animales que se encuentran en cada patio, siendo

posible modificar su asignación a estos espacios según sea necesario. Además, por motivos de organización e higiene, es importante establecer un orden específico en el recorrido de los patios durante los turnos de voluntariado. Este orden debe ser visible y editable, permitiendo también la adición o eliminación de patios según las necesidades del recinto.

- **Generación de un documento PDF con el listado de animales.** Dado que las necesidades de los voluntarios varían ampliamente debido a la diversidad demográfica, muchos requieren un soporte físico que muestre de manera clara la distribución de los patios, su orden y los animales que los integran en cada uno de ellos. Por ello, es necesario contar con una sección que permita visualizar esta información en un formato PDF, para que luego pueda descargarse e imprimirse.

- **Gestión de citas médicas.** Todos los animales de la protectora deben acudir a la clínica, y los voluntarios son responsables de llevarlos. Para simplificar esta tarea, cada animal deberá tener un módulo de citas médicas donde se pueda visualizar rápidamente la próxima cita médica más cercana, así como un historial de citas pasadas y futuras.

- **Gestión de tratamientos.** Así como para las citas médicas, los voluntarios deben poder añadir, eliminar y acceder a un listado con todos los tratamientos que tiene cada animal. Este debe incluir el nombre de la medicación, la frecuencia con la que se le administra, la fecha en la que finaliza el tratamiento y otras observaciones médicas si así se requiere.

- **Administración de adopciones.** La aplicación debe contar con un listado de todos los animales que se encuentran tanto en acogida, que es una adopción temporal, como en adopción definitiva. Este listado debe contener información sobre la persona que los adoptó y detalles relevantes del proceso de adopción. Además, dado que los voluntarios deben realizar un seguimiento de la adaptación del animal en su nuevo hogar, el sistema debe registrar las notas fechadas que los voluntarios agreguen durante este seguimiento.

Para el módulo de gestión de miembros, se han detallado los siguientes objetivos:

- **Distinción de roles de usuario.** Debido a que hay una jerarquía establecida en el equipo de voluntarios, dependiendo de su puesto y rol deberá tener permisos a unos apartados u otros. Se debe hacer distinción entre voluntarios veteranos y novatos, así como miembros coordinadores que tengan acceso único a ciertas opciones, como la creación de nuevos usuarios.

- **Registro y acceso de usuarios.** Dado que esta es una aplicación de administración interna de la Protectora, la creación de usuarios debe realizarse de manera interna por los coordinadores, quienes son los encargados de agregar nuevos voluntarios a la aplicación. Solo los coordinadores tendrán la capacidad de crear nuevos usuarios, asignándoles una

contraseña aleatoria que será enviada al momento del primer registro. Una vez hecho registrado, el usuario podrá acceder a través de un login con correo y contraseña.

- **Generación y visualización de informes de turno.** Durante un turno, los voluntarios deben ir documentando todo lo que ocurre en él. Esto incluye medicación dada a los animales, comportamientos concretos de estos, limpieza de los espacios, visitas realizadas de personas externas, prueba de compatibilidades y voluntarios que realizaron el turno, entre otra información relevante a este. Con ello, se podrá visualizar y editar informes de turnos pasados, ya que es muy importante para el equipo de personal saber qué ha ocurrido en turnos anteriores. Para esto será necesario añadir un historial de informes, datado con fecha y voluntarios pertenecientes.

- **Generación de un cuadrante horario semanal para los turnos de voluntariado.** Dado que se deben hacer turnos matutinos y vespertinos, el grupo de voluntarios debe poder distribuirse entre la semana para garantizar la cobertura de la mayoría de los turnos. Un turno está compuesto por un día de la semana y una franja horaria (mañana/tarde). La aplicación debe incluir una sección donde los voluntarios puedan inscribirse en el día y la franja horaria de su elección, con la posibilidad de visualizar en tiempo real qué voluntarios se han registrado en cada franja. Además, cada turno debe incluir al menos un voluntario veterano, por lo que es importante identificar los turnos que no están completamente cubiertos. Adicionalmente se debe incluir un botón de borrado de todos los turnos, accesible únicamente a los coordinadores, para reiniciar el cuadrante cada semana.

1.3 Estructura de la memoria

La memoria documentará todo el proceso que hemos seguido durante el desarrollo de este proyecto. Esta seguirá los siguientes apartados:

1. **Introducción.** Se dará el contexto y motivación del proyecto, así como la descripción de cómo está desarrollada la memoria.

2. **Metodología.** Se describirá la metodología seguida en el desarrollo del proyecto. Debido a ser un TFG en grupo y con retroalimentación de la entidad externa, se deberá definir cómo ha sido la organización seguida en este contexto detalladamente.

3. **Estado del arte.** Se realizará un estudio sobre los distintos softwares con funcionalidades similares que hay en el mercado, comparándolos con nuestra aplicación y obteniendo las conclusiones sobre este estudio.

4. **Tecnologías y librerías.** Se detallarán cuáles han sido las tecnologías usadas para el desarrollo, así como una explicación de su elección. Así mismo, se deberán enumerar las librerías relevantes usadas para el proyecto.

5. **Definición de requisitos.** De forma cooperativa con la Protectora, establecimos un conjunto de requisitos tanto funcionales como no funcionales. Estos se definen en la memoria en forma de listado, de forma que se recoja todo lo necesario para satisfacer las necesidades requeridas.

6. **Diseño de la aplicación.** Una vez se definen los requisitos, se definirá la arquitectura que debe tener el software, junto con los diagramas de datos y flujo necesarios para guiar su implementación. Estos elementos asegurarán que todas las funcionalidades requeridas se integren de manera coherente, facilitando tanto el desarrollo como el mantenimiento del sistema a largo plazo.

7. **Desarrollo del software.** Se definirá cómo ha sido la iteración del desarrollo del proyecto, detallando cómo se ha realizado programáticamente cada requisito y caso de uso definido.

8. **Conclusiones.** Se detallarán las conclusiones que han ido surgiendo durante el desarrollo del proyecto, así como un balance de las dificultades de este y mejoras que se podrían añadir a la aplicación.

9. **Apéndice A: Manual de instalación:** Manual para poder ejecutar la aplicación en un entorno desde cero.

10. **Apéndice B: Manual de usuario.** Este apéndice está diseñado para servir como una guía para el usuario final de la aplicación, considerando la diversidad demográfica y los diferentes niveles de conocimiento tecnológico. Su propósito es resolver cualquier duda que pueda surgir durante el uso de la aplicación.

11. **Apéndice C: Estructura de ficheros**

1.4 Metodología

Para el desarrollo del software, hemos empleado la metodología ágil Scrum. Este enfoque es un marco de trabajo ágil que permite a los equipos abordar problemas complejos y adaptativos, mientras entregan productos de manera eficiente y creativa, maximizando su valor. Scrum facilita la colaboración entre los equipos y promueve un trabajo de alto impacto. Además, esta metodología ofrece un conjunto de valores, roles y pautas que ayudan a los equipos a enfocarse en la iteración y la mejora continua en proyectos complejos.

Al tener un cliente concreto para el que estamos desarrollando este software, hemos elegido esta metodología adecuada por los siguientes motivos:

1. **Interacción continua con los coordinadores.** Al ser una metodología en la que sus bases se asientan en ciclos de trabajo cortos, llamados sprints, permite la entrega frecuente de resultados de la aplicación. Dado que solicitamos feedback de forma periódica, Scrum encaja en este marco de trabajo al contar con los stakeholders, en nuestro caso, los coordinadores y voluntarios de la protectora.

2. **Adaptabilidad y flexibilidad.** Al permitir Scrum adaptarse rápidamente a los cambios y nuevas necesidades, si durante el desarrollo los stakeholders sugieren algún cambio o ajuste añadiendo nuevos requisitos, estos se pueden adaptar mejor en el siguiente sprint. A diferencia de la metodología Waterfall, que es más rígida frente a los cambios y con una retroalimentación tardía, SCRUM se ajusta mejor a nuestras necesidades. [1]

3. **Entrega incremental.** Al realizarse el desarrollo de manera incremental, al final de cada sprint se entrega una parte funcional de la aplicación. Esto permite a los coordinadores ver avances tangibles y probar funcionalidades específicas, asegurando que el desarrollo se alinee constantemente con sus expectativas y necesidades.

4. **Desarrollo por prioridades.** SCRUM prioriza las funcionalidades más importantes, entregando primero las de mayor valor para el cliente. Al trabajar de la mano con los coordinadores, se asegura que las partes críticas de la aplicación, tanto para la gestión de animales como de voluntarios, se desarrollaran primero.

1.5 Estado del arte

Las aplicaciones de gestión para protectoras de animales han evolucionado significativamente en los últimos años, integrando diversas tecnologías para facilitar el trabajo de organizaciones que se dedican al rescate, cuidado y adopción de animales. Estas aplicaciones suelen ofrecer funcionalidades comunes, diseñadas para optimizar operaciones diarias de gestión en este ámbito: gestión del listado completo de animales que constituye la protectora, gestión de adopciones, organización del equipo de voluntariado, manejo de citas médicas e incluso algunas aportan la gestión financiera y de donaciones.

Entre algunas aplicaciones que recogen estas características, las más relevantes que hemos encontrado son:

- **Shelterluv.** Es una plataforma integral que ofrece soluciones para la gestión de adopciones, donaciones, seguimiento de animales y gestión de voluntarios. Entre

algunas de las funcionalidades más disruptivas, se encuentran la generación de to-do list de forma automática, cuenta con plantillas médicas diseñadas por veterinarios de distintas protectoras e historial de transacciones de donaciones. Si comparamos este software con el desarrollado:
Ventajas:
  - Cuenta con un historial financiero
  - Plantillas y detalle médico más detallado
  - Cuenta con un predictor de donaciones

Desventajas:
  - Se encuentra solo en inglés, lo cual no podría haber sido utilizado en nuestro caso debido a la variedad demográfica de los voluntarios
  - No cuenta con un gestor de turnos
  - No cuenta con una generación ni listado de informes
  - Es de pago, lo cual era un requisito imprescindible para esta protectora el ser gratuito

- **Petfinder Pro.** Es un software utilizado principalmente para la gestión de adopciones y la promoción de animales en adopción a través de una red de difusión por internet.
  Ventajas:
  - Incluye herramientas de análisis de datos y tendencias
  - Buscador de animales en adopción abierta al público para fomentar las adopciones

  Desventajas:
  - No incluye la mayoría de funcionalidades que requiere el equipo de voluntarios, ya que está más enfocada en promover las adopciones de forma externa

- **Animal Shelter Manager (ASM)**: Es un software gratuito y opensource que abarca desde la gestión de animales, adopciones, voluntarios y donaciones, incluyendo además estadísticas y reportes.
  Ventajas
  - No necesita instalarse ya que es accesible desde cualquier navegador
  - Tiene gestor de medicamentos distribuidos en un calendario
  - Contiene un registro de transacciones para gestionar las finanzas
  - Adaptable a web y móvil

  Desventajas:
  - No tiene distinción de roles de voluntarios
  - No contiene gestión de turnos ni informes
  - No almacena un registro de personas adoptantes

En conclusión, al tratarse de una aplicación diseñada específicamente para una causa y entidad concreta, no se pretende venderla de forma genérica en el mercado. Aunque otras aplicaciones puedan ofrecer funcionalidades adicionales, ninguna cumple todos los requisitos de la Protectora ni se ajusta tan fielmente a sus necesidades como una desarrollada exclusivamente para ella.

2

Tecnologías

2.1 Stack tecnológico principal

Para el desarrollo en backend, hemos optado por las siguientes tecnologías:

2.1.1 Python

Python es un lenguaje de programación interpretado de alto nivel orientado a objetos, diseñado con una filosofía que prioriza la facilidad de interpretación y la legibilidad. Este es el lenguaje que se ha usado para desarrollar toda la lógica de backend, debido a su amplia biblioteca de frameworks, su fácil integración y su capacidad multiplataforma. [2]

2.1.2 FastAPI

FastAPI es un framework para construir APIs de forma fácil, rápida e intuitiva utilizando *Python*. Ha sido el framework principal para construir las APIs debido a su alto rendimiento, la simplicidad que proporciona durante el desarrollo y el soporte que ofrece para la programación asíncrona. [3]

2.1.3 MongoDB

MongoDB es un sistema de gestión de BBDD no relacionales y de código abierto. Este se basa en utilizar documentos flexibles en lugar de tablas para procesar y almacenar varias formas de datos. Proporciona un modelo de almacenamiento elástico para almacenar y consultar datos multivariados. [4] Este modelo fue utilizado en ciertos módulos de la aplicación, ya que, debido a la naturaleza de los datos almacenados, una base de datos no relacional se adaptaba mejor.

2.1.4 SQLite

SQLite es un sistema de gestión de BBDD relacional y compatible con las características ACID. Implementa un motor de base de datos SQL transaccional, autónomo y sin configuración [5]. Al ser un sistema ligero y por su capacidad de portabilidad, para este proyecto fue suficiente, teniendo siempre la posibilidad de migrarlo de forma sencilla a otros sistemas de bases de datos más robustos como PostgreSQL o MySQL.

2.1.5 Docker

Docker es una tecnología que permite crear, probar e implementar aplicaciones de manera rápida y eficiente. Empaqueta el software en contenedores, los cuales contienen lo necesario para que la aplicación se ejecute correctamente, como bibliotecas, código y tiempo de ejecución, asegurando que la aplicación funcione de manera consistente en distintos entornos. Esta tecnología ha facilitado la replicación del entorno de desarrollo, garantizando la portabilidad de la aplicación y simplificando el proceso de despliegue.

Para el desarrollo del Frontend, las tecnologías empleadas han sido las siguientes:

2.1.6 TypeScript

TypeScript es un lenguaje de programación construido sobre JavaScript que ofrece tipado estático, lo que ayuda a detectar errores durante el desarrollo y mejora la estabilidad del código. La elección de este lenguaje para desarrollar el frontend del proyecto fue por su buena adaptabilidad para trabajar con frameworks como *Vue.js*, ya que proporciona herramientas para un desarrollo más robusto, seguro y escalable. Además, su compatibilidad total con JavaScript permite integrar fácilmente bibliotecas existentes, facilitando la transición y aprovechando las ventajas de ambos lenguajes. [6]

2.1.7 Vite

Vite es una herramienta de compilación y tooling, proporcionando una experiencia de desarrollo rápida para proyectos web. Es independiente al framework utilizado, teniendo plantillas para poder iniciar proyectos en *Vue.js.* [7]

2.1.8 Vue 3

Vue es un framework de JavaScript para crear interfaces de usuario. Está construido sobre los estándares de HTML, CSS y JS, proporcionando un modelo de programación declarativo y basado en componentes que ayudan a desarrollar interfaces de usuario de manera más eficiente. Entre sus usos, mejora el HTML estático sin un paso de compilación, incrusta

componentes web en cualquier página, renderiza del lado del servidor y está orientado a aplicaciones de escritorio, móvil o WebGL. [8] Su curva de aprendizaje es más suave y su configuración inicial más simple que *React*, al tiempo que ofrece un conjunto de herramientas más integrado. Comparado con *Angular*, Vue es más ligero y menos complejo, permitiendo una integración progresiva y modular sin requerir una estructura rígida.

2.1.9 Tailwind CSS

Tailwind es un framework CSS que da prioridad a la utilidad sobre el propio estilo que, a diferencia de otros frameworks como *Boostrap* o *Bulma*, no provee componentes predefinidos. En su lugar, proporciona un conjunto de clases para estructura y estilado, permitiendo crear rápidamente diseños personalizados con facilidad. [9]

Como tecnologías extras que nos ha permitido el desarrollo y organización del proyecto, se encuentran:

2.1.10 Visual Studio Code

Visual Studio Code ha sido el IDE utilizado para el desarrollo del proyecto, un editor ligero y altamente configurable y personalizable. Nos ha ofrecido características como la integración con el control de versiones, el soporte de todos los lenguajes usados, la capacidad de detección de errores y su alta biblioteca de extensiones.

2.1.11 Git

Git es un sistema de control de versiones distribuido que permite gestionar y coordinar cambios de código de forma eficiente y facilitando el desarrollo en equipo. Ha sido imprescindible para estructurar el proyecto en ramas, coordinar los módulos de desarrollo y fusionas las contribuciones de ambos desarrolladores sin conflictos, manteniendo siempre el historial de versiones y el código actualizado.

2.2 Librerías destacadas

2.2.1 Pydantic

Pydantic es la librería de *Python* más usada para validación de datos. Esta permite que la validación de esquemas y serialización se controlen por anotaciones de tipos, siendo fácil de aprender, con menos líneas de código y permitiendo integrarse con el IDE utilizado.

2.2.2 Beanie

Es una mapeador de documentos de Mongo a objetos Python de forma asíncrona, basando los modelos de datos en *Pydantic.* Usando esta librería, cada colección de la BD tiene su correspondiente documento que se utiliza para interaccionar con esa colección. Además de recuperar datos, permite realizar todo el CRUD de documentos a cada colección. [10]

2.2.3 SQLModel

SQLModel es una librería para interactuar con bases de datos SQL con objetos *Python*, diseñada para simplificar la interacción de BBDD en aplicaciones de FastAPI. Al combinarse con *Pydantic*, busca simplificar el código al minimizar duplicidades y mejorar la experiencia del desarrollador. [11]

2.2.4 Vue Router

Es una librería de *Vue.js* para la gestión de rutas, la cual permite definir y manejar rutas facilitando la creación de aplicaciones de una sola página. Ofrece funciones como poder configurar rutas dinámicas, anidar y usar guardias de navegación, entre otras. Destaca entre otras librerías y frameworks por su simplicidad y flexibilidad, integrándose con *Vue.js* de manera natural. [12]

2.2.5 Pinia

Para gestionar los datos de forma cohesionada y coordinada, se introduce el patrón Redux: un patrón de arquitectura de datos que busca gestionar los estados y datos de la aplicación para que sea predecible en un futuro y pueda mantener de forma coherente el flujo de datos. Sus principios se definen como almacenamiento único, inmutabilidad y funciones puras. Uno de los elementos que conforman este patrón es el Store, que es el punto de comunicación entre las acciones solicitadas y la respuesta lógica, encargándose posteriormente de actualizar los estados de toda la aplicación en función de estas. [13] *Pinia* es una librería de *Vue.js* que sirve para utilizar este patrón de forma simplificada y eficiente.

2.2.6 DaisyUI

DaisyUI es una librería de componentes que funciona como un plugin de *TailwindCSS*, la cual ofrece una gran cantidad de componentes pre construidos que son fácilmente integrables con proyectos que usen este framework.

2.2.7 Zod

Zod es una librería de validación de esquemas en *TypeScript*, la cual permite definir, analizar y validar datos de forma sencilla. Ha sido usada en el proyecto para garantizar que los datos recibidos y manipulados por la aplicación cumplen las especificaciones requeridas, asegurando así la integridad de datos a lo largo del desarrollo.

3

Definición de requisitos

En este apartado, abordaremos tanto los requisitos funcionales como los no funcionales por cada módulo de la aplicación, siguiendo la división en módulos establecida para este TFG grupal. En esta memoria, nos enfocaremos específicamente en los apartados relacionados con el módulo de animales.

3.1 Requisitos funcionales

En este apartado se abordarán los requisitos funcionales que abarca este módulo.

- **RF1. CRUD de animales:** Un usuario de la aplicación podrá crear, eliminar, modificar y consultar los datos de un animal concreto, incluyendo sus características de personalidad, físicas, demográficas, de conducta y fotografías.

- **RF2. Visualización del listado de animales:** Se podrá visualizar un listado de todos los animales en la protectora, organizados por patios, mostrando el nombre, foto y sexo de cada uno de ellos.

- **RF3. Filtrado del listado de animales:** En la vista de animales, se podrán filtrar los animales a visualizar por distintos valores: características físicas, demográficas, de ubicación, así como por el nombre de cada uno de ellos.

- **RF4. CRD de patios de la protectora:** Será posible crear, consultar y eliminar los patios de la protectora, los cuales se mostrarán en un formato de listado.

- **RF5. Modificación y visualización del orden de los patios:** El usuario podrá ver y modificar el orden en que se visitan los patios durante los turnos.

- **RF6. Generación de un PDF con el listado de animales:** Se podrá generar automáticamente un documento PDF que funcione como un listado para facilitar la identificación de los animales, mostrando la imagen, nombre y tratamientos de cada uno según su ubicación.

- **RF7. CRD de tratamientos de cada animal:** Se podrán crear, eliminar y visualizar en forma de listado los tratamientos que tiene cada animal. Esta información debe recoger la información básica que se necesita saber para su administración.

- **RF8. CRD de citas médicas de cada animal:** Se podrán crear, eliminar y visualizar en forma de histórico ordenado cronológicamente las citas médicas que tiene cada animal, diferenciando entre pasadas y próximas

- **RF9. Visualización de la próxima cita médica de un animal:** El usuario podrá consultar la fecha, hora, y detalles de la próxima cita médica programada para cada animal.

- **RF10. CRD adopciones:** El usuario podrá crear, eliminar y consultar los datos de la adopción de cada animal adoptado, incluyendo la información del adoptante y la información del animal.

- **RF11. Visualización del listado de animales adoptados:** Se podrá visualizar en forma de listado todas las adopciones que se han llevado a cabo en la protectora, diferenciando si son temporales o permanentes.

- **RF12. CRD de seguimientos:** Para cada adopción, se podrán crear, consultar y eliminar registros fechados sobre la evolución de la adopción, lo que se conoce entre los voluntarios como seguimiento.

3.2 Requisitos no funcionales

- **RNF1. Rendimiento:** La aplicación debe ser capaz de procesar y responder a las solicitudes de los usuarios de forma rápida, garantizando una experiencia fluida y sin mucha latencia.

- **RNF2. Seguridad:** Debe garantizar la protección de los datos de los usuarios y personas registradas en la aplicación. Esto incluye la autenticación de usuarios, autorización de roles y encriptación de credenciales de inicio de sesión.

- **RNF3. Usabilidad:** La interfaz de usuario debe ser intuitiva y fácil de navegar para personas con distintos niveles de habilidad tecnológica. Debe incluir una guía clara, un diseño accesible y minimizar la cantidad de clics necesarios para completar tareas comunes.

- **RNF4. Mantenibilidad:** El código fuente de la aplicación debe seguir buenas prácticas de desarrollo para facilitar su actualización, corrección de errores y mejoras futuras sin afectar el funcionamiento actual.

- **RNF5. Experiencia de usuario óptima para dispositivos móviles:** La aplicación debe estar diseñada con el punto de mira puesto en los dispositivos móviles, los cuales serán su principal cliente, garantizando una experiencia simple y familiar al usuario.

3.3 Casos de uso implementados

Los casos de uso estarán intervenidos por los distintos actuadores de la aplicación. A continuación, se describen los posibles actuadores:

- **Usuarios**: Cuando hablamos de usuarios, intervienen todo tipo de actuadores de manera genérica: tanto coordinadores como voluntarios de cualquier tipo, pudiendo cualquiera de ellos cumplir con la funcionalidad descrita.

- **Coordinadores**: Son los administradores, usuarios con permisos que solo ellos pueden hacer funcionalidades concretas de la aplicación. Estos permisos lo dan los implementadores o personas de mantenimiento de la aplicación de forma interna.

- **Voluntarios veteranos**: Son voluntarios de la protectora que tienen un tiempo concreto de veteranía. Al pasar a ser veteranos, tendrán acceso a funcionalidades concretas o intervendrán en condiciones de visualización en la aplicación.

- **Voluntarios novatos:** Son voluntarios de la protectora con poco tiempo de voluntariado. Sus funciones son las más restringidas

Estos son los casos de uso que implementan los requisitos funcionales descritos:

Titulo	Caso de uso RF1. CRUD de animales
Descripción	Se describe cómo un usuario puede crear, editar, visualizar y eliminar animales
Precondición	El usuario ha iniciado sesión en la aplicación
Postcondición	1. El usuario ha consultado un animal en la aplicación 2. El coordinador ha creado, eliminado o editado los datos de un animal en la aplicación
Actores	Coordinador, voluntario
Escenario principal	1. El usuario inicia la aplicación e inicia sesión 2. El usuario va al apartado de animales 3. El usuario puede consultar los datos de los datos de un animal concreto de la aplicación haciendo clic en él 4. El coordinador puede crear a un animal nuevo

	a. El coordinador da al botón de crear b. El coordinador introduce los datos necesarios como nombre, características personales y físicas y fotografía del nuevo animal c. El coordinador da a guardar d. Se comprueban los datos y se registra un nuevo animal e. Se notifica que se ha creado correctamente 5. El coordinador puede editar los datos de un animal a. El coordinador clica en el animal que quiere consultar b. El coordinador clica en el botón de editar c. El coordinador introduce los nuevos datos en la vista que se abre d. El coordinador da a guardar los datos e. Se comprueban los datos y se registran los cambios f. Se notifica que se ha editado correctamente 6. El coordinador puede eliminar a un animal a. El usuario da clic al botón de eliminar b. El usuario acepta la notificación de borrado c. Se notifica que se ha eliminado correctamente
Escenario alternativo	4.d.b Los datos ingresados no son válidos, aparecerá un mensaje de error por pantalla 5.e.b Los datos ingresados no son válidos, aparecerá un mensaje de error por pantalla 6.b.b Ha ocurrido un error en el sistema y no se ha podido eliminar, aparecerá un mensaje de error por pantalla

Tabla 1. Caso de uso RF1

Titulo	Caso de uso RF2. Visualización del listado de animales
Descripción	Se describe cómo un usuario puede visualizar los animales de la protectora organizados por su ubicación en patios
Precondición	El usuario ha iniciado sesión en la aplicación

Postcondición	El usuario ha consultado el listado de animales completo que hay en la protectora
Actores	Usuario
Escenario principal	1. El usuario ingresa en la aplicación 2. El usuario entra en el apartado de animales 3. El usuario puede visualizar los animales que hay en la protectora, organizados por patios
Escenario alternativo	3.b No hay registro de animales en la protectora, se notifica al usuario de que no existen animales 3.c Ha ocurrido un error en el sistema al cargar los animales de la base de datos y se notifica al usuario de ello

Tabla 2. Caso de uso RF2

Titulo	Caso de uso RF3. Filtrado del listado de animales
Descripción	Se describe cómo un usuario puede filtrar los animales del listado según distintas características
Precondición	El usuario ha iniciado sesión en la aplicación
Postcondición	El usuario ha filtrado los animales deseados de la lista
Actores	Usuario
Escenario principal	1. El usuario ingresa en la aplicación 2. El usuario entra en el apartado de animales 3. El usuario da clic al botón de filtros 4. El usuario introduce los filtros deseados, introduciendo texto o interactuando con los componentes de selección 5. Se actualiza la vista del listado de manera dinámica, visualizando los animales filtrados 6. El usuario cierra la pestaña de filtros
Escenario alternativo	3.b No hay registro de animales en la protectora, se notifica al usuario de que no existen animales

	3.c Ha ocurrido un error en el sistema al cargar los animales de la base de datos y se notifica al usuario de ello

Tabla 3. Caso de uso RF3

Titulo	Caso de uso RF4. CRD de patios de la protectora
Descripción	Se describe cómo un usuario puede consultar, eliminar y crear patios de la protectora
Precondición	El usuario ha iniciado sesión en la aplicación
Postcondición	El usuario ha añadido, eliminado o consultado un patio de la protectora
Actores	Coordinador, voluntario
Escenario principal	1. El usuario ingresa en la aplicación 2. El usuario entra en el apartado de animales 3. El usuario da al botón de patios 4. El usuario visualiza el listado de patios que hay actualmente en la protectora 5. El coordinador puede crear un patio nuevo a. Da clic al botón de añadir patio b. Introduce el nombre del patio nuevo en el desplegable que aparece c. Da clic a crear nuevo patio d. El sistema valida los datos y se añade 6. El coordinador puede eliminar un patio a. Da clic en la cruz del patio que quiere eliminar b. Confirma el mensaje de confirmación de borrado c. El sistema valida los datos y se elimina
Escenario alternativo	4.b No hay patios en la protectora, se notifica que no hay patios añadidos. 5.d.b Ha ocurrido un error en el sistema al validar los datos, se notifica por pantalla que no se ha podido crear 6.c.b Ha ocurrido un error en el sistema y no ha sido posible eliminar, se notifica al usuario.

Tabla 4. Caso de uso RF4

Titulo	Caso de uso RF5. Modificación y visualización del orden de los patios
Descripción	Se describe cómo un usuario puede consultar y modificar el orden de patios que hay actualmente en la protectora
Precondición	El usuario ha iniciado sesión en la aplicación
Postcondición	El usuario ha consultado o modificado el orden de patios
Actores	Usuario
Escenario principal	1. El usuario ingresa en la aplicación 2. El usuario entra en el apartado de animales 3. El usuario da al botón de patios 4. El usuario visualiza el listado de patios que hay actualmente en la protectora, dispuestos en orden 5. El usuario puede modificar el orden de patios a. Da clic al botón de editar b. El usuario da a la flecha de subir o bajar del patio que quiere alterar el orden c. El usuario de clic en guardar d. Se valida el nuevo orden y se añade a la BD
Escenario alternativo	4.b No hay patios en la protectora, se notifica que no hay patios añadidos. 5.d.b Ha ocurrido un error en el sistema al validar los datos, se notifica por pantalla que no se ha podido alterar el orden

Tabla 5. Caso de uso RF5

Titulo	Caso de uso RF6. Generación de un PDF con el listado de animales
Descripción	Se describe cómo se genera un documento PDF con el listado de todos los animales de la protectora, organizado por patio, e incluyendo fotografías y datos básicos de cada uno
Precondición	El usuario ha iniciado sesión en la aplicación

Postcondición	El usuario obtiene en su dispositivo un archivo PDF con el listado de animales
Actores	Usuario
Escenario principal	1. El usuario ingresa en la aplicación 2. El usuario entra en el apartado de animales 3. El usuario clica en el botón de generar PDF 4. El usuario selecciona el lugar donde guardar el nuevo documento dentro de su dispositivo 5. Da clic en guardar 6. Se descarga el documento en la ubicación indicada 7. El usuario abre el documento y visualiza el listado
Escenario alternativo	6.b Ha ocurrido un error en el sistema, se notifica por pantalla que no se ha podido descargar el documento

Tabla 6. Caso de uso RF6

Titulo	Caso de uso RF7. CRD de tratamientos de cada animal
Descripción	Se describe cómo un usuario puede crear, consultar o eliminar un tratamiento de la lista de tratamientos de cada animal
Precondición	El usuario ha iniciado sesión en la aplicación
Postcondición	El usuario ha añadido, consultado o editado un tratamiento de la lista de tratamientos de un animal específico
Actores	Usuario
Escenario principal	1. El usuario inicia la aplicación e inicia sesión 2. El usuario va al apartado de animales 3. El usuario da clic al animal que quiere consultar sus tratamientos 4. El usuario da clic en la nueva pantalla al botón de tratamientos 5. El usuario puede consultar el listado de tratamientos de ese animal 6. El usuario puede añadir un nuevo tratamiento

	a. El usuario da al botón de crear b. El usuario introduce los datos necesarios, haciendo uso de los inputs de texto para rellenar la información necesaria de ese tratamiento c. El usuario da a guardar d. Se comprueban los datos y se registra un nuevo tratamiento a la lista e. Se notifica que se ha creado correctamente 7. El usuario puede eliminar un tratamiento de la lista a. El usuario da clic al botón de eliminar del tratamiento específico b. El usuario acepta la notificación de borrado c. Se notifica que se ha eliminado correctamente
Escenario alternativo	6.d.b Los datos ingresados no son válidos, aparecerá un mensaje de error por pantalla 7.c.b Ha ocurrido un error en el sistema y no se ha podido eliminar, aparecerá un mensaje de error por pantalla

Tabla 7. Caso de uso RF7

Titulo	Caso de uso RF8. CRD de citas médicas de cada animal
Descripción	Se describe cómo un usuario puede crear, consultar o eliminar una cita médica de un animal
Precondición	1. El usuario ha iniciado sesión en la aplicación 2. La cita médica que se quiere borrar no es pasada
Postcondición	El usuario ha añadido, consultado o editado una cita médica de un animal
Actores	Usuario
Escenario principal	1. El usuario inicia la aplicación e inicia sesión 2. El usuario va al apartado de animales 3. El usuario da clic al animal que quiere consultar sus citas médicas 4. El usuario da clic en la nueva pantalla al botón de citas médicas

	5. El usuario da clic en la nueva pantalla a las citas médicas
	6. El usuario puede consultar el histórico de citas médicas, tanto pasadas como futuras
	7. El usuario puede añadir una nueva cita médica
	    a. El usuario da al botón de crear
	    b. El usuario introduce los datos necesarios, haciendo uso de los inputs de texto para rellenar la información necesaria de esa cita médica, así como añadir la fecha deseada
	    c. El usuario da a guardar
	    d. Se comprueban los datos y se registra una nueva cita médica
	    e. Se notifica que se ha creado correctamente
	8. El usuario puede eliminar una cita médica futura
	    a. El usuario da clic al botón de eliminar de la cita médica específica
	    b. El usuario acepta la notificación de borrado
	    c. Se notifica que se ha eliminado correctamente

Escenario alternativo	7.d.b Los datos ingresados no son válidos, aparecerá un mensaje de error por pantalla 8.c.b Ha ocurrido un error en el sistema y no se ha podido eliminar, aparecerá un mensaje de error por pantalla

Tabla 8. Caso de uso RF8

Titulo	Caso de uso RF9. Visualización de la próxima cita médica de un animal
Descripción	Se describe cómo un usuario puede visualizar la próxima cita médica más cercana de cada animal
Precondición	El usuario ha iniciado sesión en la aplicación
Postcondición	El usuario ha consultado la próxima cita médica
Actores	Usuario
Escenario principal	1. El usuario inicia la aplicación e inicia sesión 2. El usuario va al apartado de animales

	3. El usuario da clic al animal que quiere consultar su cita médica 4. El usuario da clic en la nueva pantalla al botón de citas médicas 5. El usuario puede consultar su próxima cita médica, tanto su fecha como su motivo de consulta
Escenario alternativo	5.b No hay próximas citas médicas, se notifica por pantalla su ausencia.

Tabla 9. Caso de uso RF9

Titulo	Caso de uso RF10. CRD de adopciones
Descripción	Se describe cómo un usuario puede crear, consultar o eliminar una adopción
Precondición	El usuario ha iniciado sesión en la aplicación
Postcondición	El usuario ha añadido, consultado o eliminado una adopción
Actores	Coordinador, voluntario
Escenario principal	1. El usuario inicia la aplicación e inicia sesión 2. El usuario va al apartado de animales 3. El usuario da clic al botón de adopciones 4. El coordinador puede eliminar una adopción a. El coordinador da clic al botón de eliminar de la adopción específica b. El coordinador acepta la notificación de borrado c. Se notifica que se ha eliminado correctamente 5. El usuario puede añadir una nueva adopción a. El usuario da al botón de crear b. El usuario introduce los datos necesarios, haciendo uso de los inputs de texto para rellenar la información necesaria, así como los componentes desplegables para elegir la persona adoptante y animal c. El usuario da a guardar d. Se comprueban los datos y se registra una nueva adopción e. Se notifica que se ha creado correctamente

	6. El usuario da clic a la adopción concreta que quiere visualizar 7. El usuario visualiza los datos de esa adopción
Escenario alternativo	6.d.b Los datos ingresados no son válidos, aparecerá un mensaje de error por pantalla 7.c.b Ha ocurrido un error en el sistema y no se ha podido eliminar, aparecerá un mensaje de error por pantalla

Tabla 10. Caso de uso RF10

Titulo	Caso de uso RF11. Visualización del listado de animales adoptados
Descripción	Se describe cómo un usuario puede consultar el listado de todas las adopciones realizadas, tanto permanentes como acogidas
Precondición	El usuario ha iniciado sesión en la aplicación
Postcondición	El usuario ha visualizado las adopciones
Actores	Usuario
Escenario principal	1. El usuario inicia la aplicación e inicia sesión 2. El usuario va al apartado de animales 3. El usuario da clic al botón de adopciones 4. El usuario puede visualizar el listado de adopciones con los datos básicos: tipo, persona y animal
Escenario alternativo	4.b No hay adopciones registradas en la base de datos, se notifica por pantalla que no hay adopciones

Tabla 11. Caso de uso RF11

Titulo	Caso de uso RF12. CRD de seguimientos
Descripción	Se describe cómo un usuario puede crear, consultar o eliminar un registro de seguimiento

<table>
<tr><td>Precondición</td><td>El usuario ha iniciado sesión en la aplicación</td></tr>
<tr><td>Postcondición</td><td>El usuario ha añadido, consultado o eliminado un registro de seguimiento</td></tr>
<tr><td>Actores</td><td>Usuario</td></tr>
<tr><td>Escenario principal</td><td>1. El usuario inicia la aplicación e inicia sesión
2. El usuario va al apartado de animales
3. El usuario da clic al botón de adopciones
4. El usuario da clic a la adopción concreta que quiere visualizar para consultar un registro
5. El usuario visualiza los registros de esa adopción
6. El usuario puede añadir un nuevo registro de seguimiento
a. El usuario da al botón de crear registro
b. El usuario introduce los datos necesarios, tanto la fecha de registro como la anotación a añadir
c. El usuario da a guardar
d. Se comprueban los datos y se registra una nueva adopción
e. Se notifica que se ha creado correctamente
7. El usuario puede eliminar un registro
a. El usuario da clic al botón de eliminar del registro específico
b. El usuario acepta la notificación de borrado
c. Se notifica que se ha eliminado correctamente</td></tr>
<tr><td>Escenario alternativo</td><td>6.d.b Los datos ingresados no son válidos, aparecerá un mensaje de error por pantalla
7.c.b Ha ocurrido un error en el sistema y no se ha podido eliminar, aparecerá un mensaje de error por pantalla</td></tr>
</table>

Tabla 12. Caso de uso RF12

4

Diseño de la aplicación

A continuación, vamos a documentar todo el proceso de diseño de la aplicación, considerando desde la arquitectura de los diferentes servicios que interactúan entre ellos, el modelo de datos que sustenta la aplicación o las decisiones tomadas sobre la interfaz de usuario que faciliten su comprensión.

4.1 Arquitectura

La idea inicial de este proyecto es crear un software usable por todo el equipo de miembros de la Protectora, tanto durante el turno como fuera de él. Por esta razón, una de las motivaciones principales es que sea fácilmente accesible en cualquier momento. Esto nos brinda un marco perfecto para el desarrollo de una aplicación web enfocada principalmente en dispositivos móviles, ya que estos nos aportan la mayor facilidad de adaptación posible.

Debido a las funcionalidades que esta ofrece, la aplicación debe ser altamente interactiva, lo que nos lleva a decidirnos por crear una aplicación de tipo **Single Page Application (SPA)**.

Para brindar de persistencia y algunas funcionalidades adicionales a la aplicación necesitaremos un servidor que responda las peticiones de esta, con lo cual vamos a emplear una arquitectura de tipo Cliente – Servidor.

4.1.1 Cliente – SPA

Una aplicación SPA es aquella que se caracteriza por ir modificando la interfaz dinámicamente conforme el usuario interactúa con ella. Este método de renderizado es conocido como Client Side Rendering, ya que es el lado del cliente el encargado de renderizar el HTML que se le mostrará al usuario.

En contraposición a este, tendríamos el Server Side Rendering (SSR), donde en lugar de modificar la interfaz, las acciones del usuario solicitan nuevas páginas al servidor, quien es el encargado de renderizarlas y mandarlas al usuario. Este modelo es característico de tecnologías con una larga trayectoria, como Spring Boot (Java), Symfony (PHP) o ASP.NET (C#), que antes se consideraban la forma tradicional de crear una aplicación web. También se utiliza en alternativas más modernas, generalmente implementadas en Javascript (o Typescript), como Next.js, Nuxt o Astro.

Las SPA son particularmente ideales en entornos donde el usuario realiza constantemente acciones sobre ellas y se quiere priorizar una experiencia de usuario fluida. Esto ocurre ya que al modificar el contenido que se ha descargado una única vez al inicio de la aplicación, en lugar de recargarlo constantemente, conseguimos una reactividad más cercana a la que ofrecería una aplicación nativa. Es por esto que para nuestra aplicación la decisión lógica es optar por un cliente SPA. [14]

4.1.2 Servidor – API REST

Una vez que tenemos claro qué tipo de arquitectura vamos a utilizar en el cliente, debemos decidir qué vamos a necesitar del lado del servidor. En este caso, al tener una SPA como cliente vamos a necesitar un servidor que sirva para dar soporte a todas las peticiones que el cliente va a solicitar mientras el usuario navegue por la interfaz.

Aunque utilicemos una arquitectura Client Side Rendering, donde no se están enviando constantemente nuevas páginas HTML, no quiere decir que el cliente no realice nuevas peticiones al servidor. La diferencia radica en que, en lugar de solicitar la página completa, el cliente solo necesita que el servidor le mande los datos necesarios para mostrar la parte de la interfaz que quiere actualizar, generalmente aquella con la que está interactuando el usuario.

Estas nuevas peticiones deben hacerse de forma asíncrona a la ejecución del cliente, de forma que este no deje de prestar servicio mientras se esperan los nuevos datos. Esta forma de comunicación se denomina **AJAX** (Asynchronous JavaScript + XML)[1].

Existen muchas alternativas diferentes que podríamos implementar del lado del servidor para responder las peticiones del cliente, algunas de las que podríamos considerar son:

- GraphQL

[1] Aunque se especifica expresamente el formato XML en el título, en la realidad este término es usado para cubrir todas aquellas peticiones asíncronas en JavaScript que se hacen para obtener nuevos datos, independientemente del formato que se utilice para transferirlas (XML, JSON, YAML, ...)

- WebSockets
- gRPC

Basándonos en los requisitos de nuestra aplicación, por los cuales no necesitamos ningún caso de uso especialmente complejo, lo ideal es utilizar una solución simple y sobre todo altamente compatible e integrable con ambos extremos de la comunicación, por lo que decidimos optar por la alternativa estándar del sector, con una curva de aprendizaje inferior al resto y perfectamente funcional para nuestro caso, una **API REST**.

Las APIs REST se comunican utilizando el protocolo HTTP y generalmente enviando archivos JSON para la transferencia de información, lo que las hace altamente compatibles con casi cualquier sistema. Funcionan disponibilizando una serie de recursos y acciones que realizar sobre ellos (CRUD), lo que es perfecto para el uso que queremos hacer del servidor (principalmente de acceso a datos).

4.1.2.1 Servidor – WebSockets

Hemos comentado anteriormente que no necesitamos ningún caso de uso especial para nuestro servidor y hemos descartado los WebSockets como forma de comunicación de nuestro servidor, pero esto es así de forma general y para la mayor parte de la aplicación, salvo por un módulo concreto: el cuadrante de turnos.

En el cuadrante de turnos uno de los requisitos fundamentales es que las actualizaciones se vean en tiempo real, lo que haría que la implementación usando una comunicación por API REST fuera más tediosa y no en estricto tiempo real (probablemente necesitaríamos alguna forma de "polling", que fuera comprobando constantemente los datos).

Es por esta razón por lo que, aunque de forma general el envío de datos se hará mediante la API, solo en este módulo vemos idóneo implementar una comunicación por websockets que garantice y simplifique la instantaneidad.

Los websockets funcionan estableciendo un canal de comunicación bidireccional persistente entre cliente y servidor, de forma que queda abierto hasta que uno de los dos cierra la comunicación, por lo que ambos pueden mandar y recibir información en cualquier momento. Es por esto que esta solución es mucho más conveniente para obtener las actualizaciones en tiempo real que el protocolo HTTP, donde la comunicación es en un solo sentido (el cliente solicita, el servidor responde) y cada comunicación se termina después de recibir la respuesta.

4.1.3 Autorización

Una vez que tenemos claro qué tipo de comunicación va a existir entre nuestro cliente y nuestro servidor, el siguiente paso lógico es pensar cómo vamos a garantizar que estas comunicaciones se produzcan de manera segura.

En nuestra aplicación no todos los usuarios podrán acceder a determinados recursos, por lo que es fundamental implementar una forma en la que el servidor se capaz de garantizar que la persona que está intentando acceder a cierto recurso pueda hacerlo. Para esto distinguimos varios tipos de usuario desde el punto de vista de la seguridad:

- Usuario no registrado
- Usuario registrado (voluntarios)
- Administrador (coordinadores)

Al tratarse de una aplicación cerrada, el acceso a cualquier recurso debe garantizar al menos que el usuario que intenta acceder a él sea un usuario registrado, además de poder distinguir para otros recursos más sensibles si el usuario tiene rol de administrador o no.

Para garantizar el primer punto, lo ideal es que la aplicación solicite las credenciales al usuario antes de darle paso a utilizar sus funcionalidades. En nuestro caso, estas credenciales serán email y contraseña, ya que son la forma más efectiva y simple de identificar a los usuarios de la Protectora. Qué se hace posteriormente con estas credenciales para autorizar las peticiones de los usuarios depende del método de autorización que escojamos, el cual en nuestro caso se trata de OAuth2.

OAuth2 es el estándar de la industria para autorizar usuarios a acceder a determinados recursos. Este introduce la figura del servidor de autorización, quien es el encargado de validar las credenciales del usuario y responder con un token de autorización al cliente que sirva para identificarle frente al servidor de recursos.

OAuth2 contempla diferentes flujos de autorización, que varían según dónde se validan las credenciales del usuario, cómo se devuelve el token y a dónde se redirige tras el inicio de sesión, entre otros factores. En nuestro caso, vamos a utilizar el flujo denominado **Resource Owner Password Credentials Grant**, ya que nuestro servidor de autorización y de recursos será el mismo. Los pasos que sigue el flujo son:

1. El usuario envía sus credenciales (usuario y contraseña) al servidor mediante un endpoint específico para ello.
2. El servidor valida las credenciales y genera un token que identifica al usuario y envía al cliente como respuesta.
3. El cliente almacena el token y lo inserta en las cabeceras de las posteriores peticiones que hará al servidor

En nuestro caso, el token que obtiene el cliente contiene la información de quién es el usuario que ha iniciado sesión y el rol que este cumple. De esta forma, cuando el servidor reciba el token buscará el usuario en su registro, si existe significa que ha iniciado sesión, y si no, que no se trata de un usuario registrado.

Dentro de este flujo tenemos que insertar algunos factores más para garantizar la seguridad y evitar que algunos usuarios intenten hacerse pasar por otros.

Para conseguir esto, lo que haremos será utilizar una clave privada que solo el servidor conoce, de esta forma, cuando el servidor genere un token con los datos del usuario para el que ha validado sus credenciales, incluirá en este una firma generada en base a los datos del token y la clave secreta del servidor.

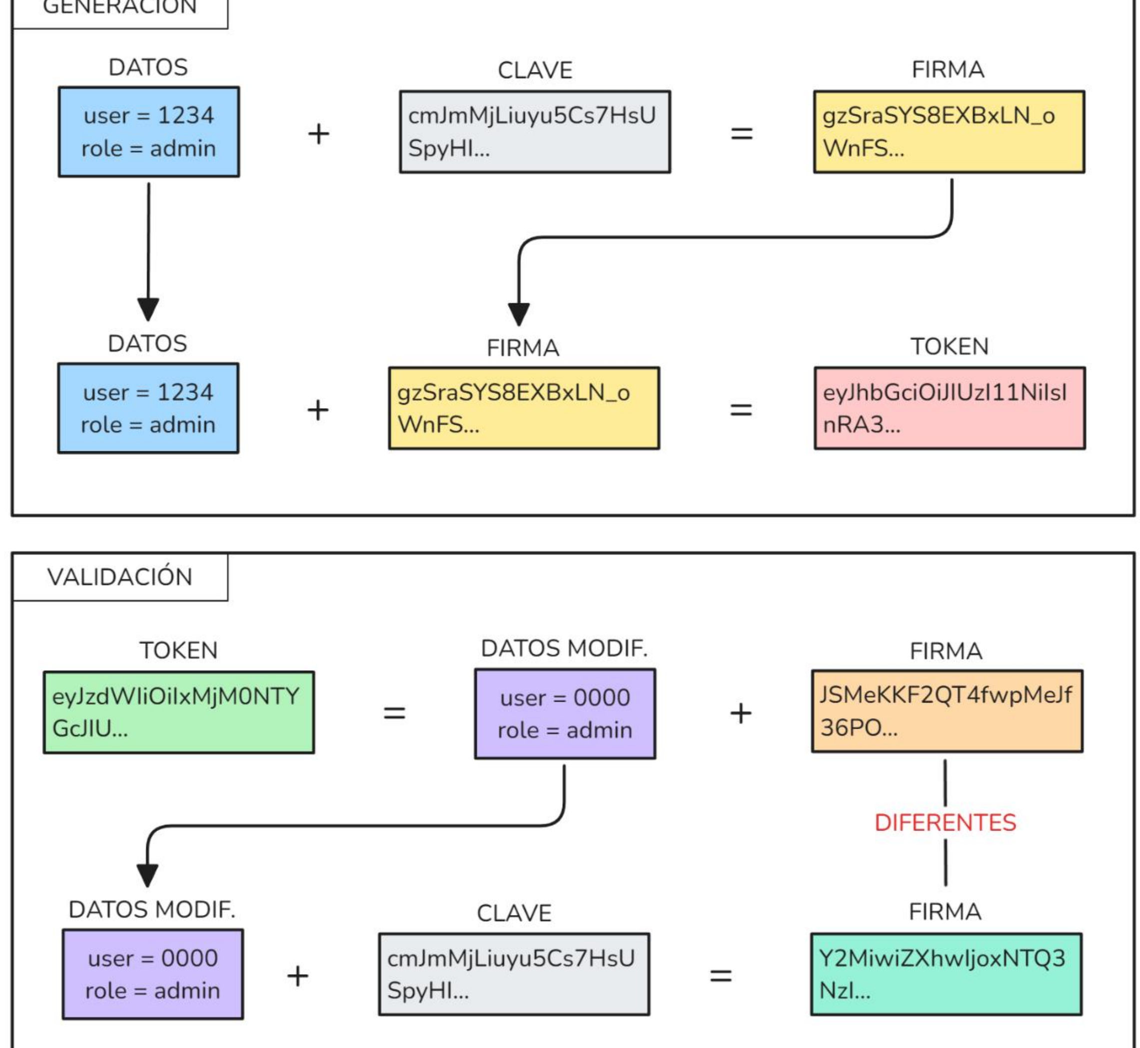


Ilustración 1. Diagrama de generación y validación del token

Así, cuando el servidor reciba un token de autorización en una petición, antes de comprobar el usuario enviado deberá confirmar que los datos que se incluyen en el token firmados con su clave dan como resultado la firma que incluye el token, garantizando así que nadie ha modificado dichos datos y él ha sido quien generó el token.

4.2 Modelo de datos

Ahora que ya conocemos la arquitectura de los sistemas de nuestra aplicación, debemos hablar de la estructura de los datos que harán de esta una aplicación funcional.

Dentro de la aplicación debemos destacar dos apartados de datos fundamentales, los relacionales y los no relaciones. La decisión de utilizar ambos tipos se basa en las características y uso de estos.

En general la aplicación está fuertemente basada en las relaciones que existen entre Usuarios, Personas, Animales, Adopciones, etc., lo que hace indiscutible la necesidad de una base de datos relacional que nos permita modelar todas estas relaciones y hacer consultas conjuntas para mostrar todos los datos necesarios.

Por otro lado, también contamos con datos de otra índole, donde su función no es tanto la de modelar relaciones complejas donde debamos mantener una consistencia estricta o una estructura firme, si no:

- Almacenar una alta cantidad de datos históricos de los Informes de los turnos, así como también de los registros del orden de Patios.
- Ofrecer una estructura de datos común, optimizada para recibir muchas operaciones de escritura y lectura al mismo tiempo en el Cuadrante semanal de turnos.

Estos modelos, poco complejos en cuanto a relaciones, no se beneficiarían de ser descritos en un formato relacional. En cambio, sí sería más apropiado utilizar un formato de documento, como un objeto JSON simple con poca anidación, ya que así ganamos eficiencia en su acceso y modificación.

4.2.1 Datos relacionales

- **Animales**. Entidad principal de la aplicación, recoge todos los detalles de un animal que forma parte de la Protectora.
- **Personas**. Entidad que modela los detalles de identificación de una persona (nombre, teléfono, ...). Sirve tanto para personas adoptantes como para usuarios de la aplicación.
- **Usuarios**. Miembro del equipo de la Protectora. Cada usuario es fundamentalmente una persona con datos adicionales para usar la aplicación.
- **Patios**. Localizaciones dentro de la Protectora donde residen los animales.
- **Adopciones**. Evento que ocurre cuando un animal se marcha de la protectora con un adoptante, de forma temporal o indefinida.
- **Seguimientos**. Comentarios fechados que los miembros de la Protectora hacen sobre una adopción para ilustrar su progreso de adaptación.
- **Citas médicas**. Citas de veterinaria que los animales tienen dentro de la Protectora.
- **Tratamientos**. Información sobre los tratamientos médicos de los animales.

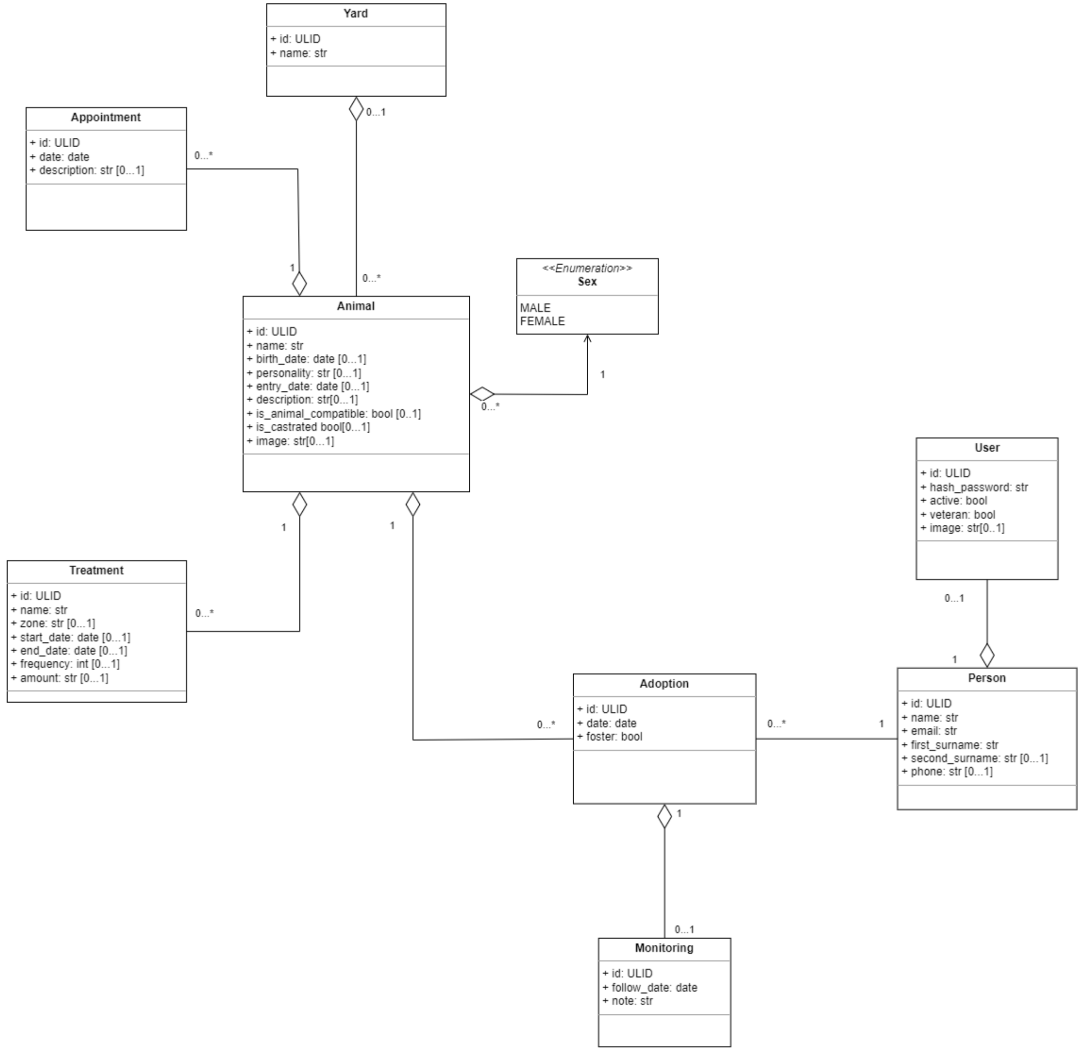
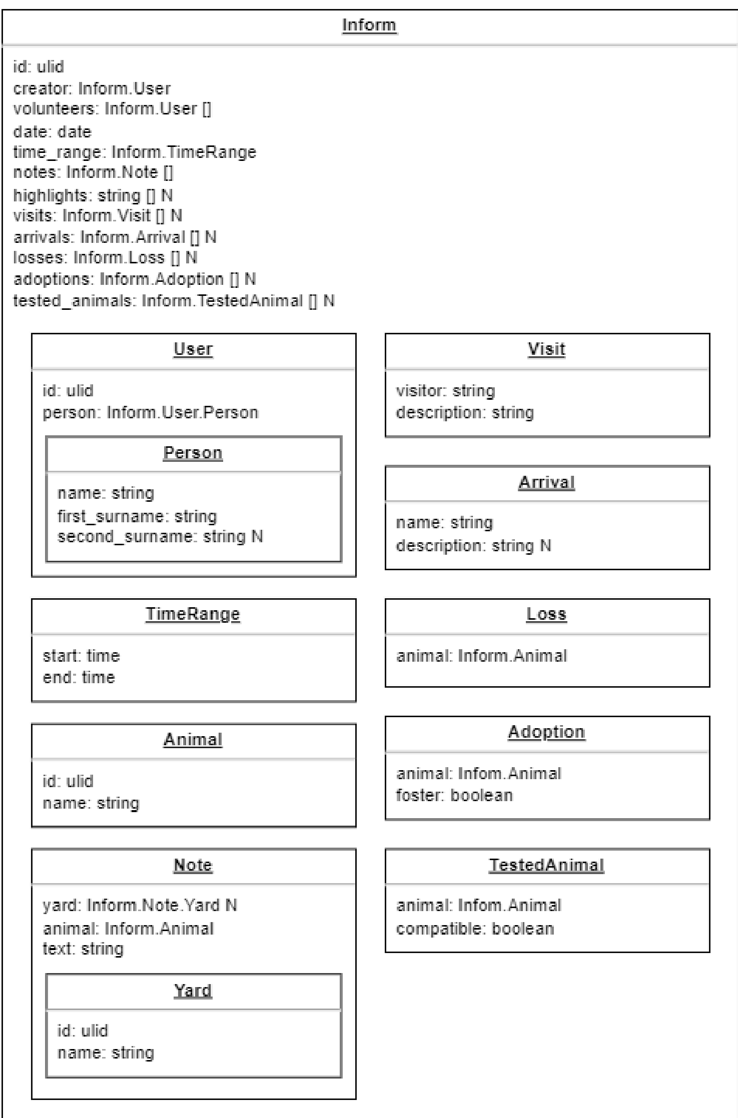
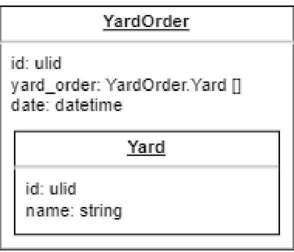


Ilustración 2. Diagrama de clases de los datos relacionales

4.2.2 Datos no relacionales





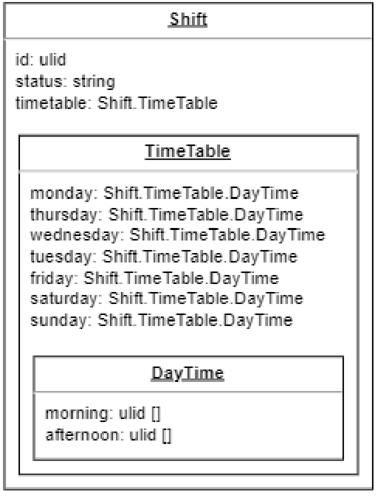


Ilustración 3. Diagrama de las entidades no relacionales

- **Informe**. Documento de registro de acciones que ocurren durante el turno de voluntariado. Desde comentarios sobre el estado de los animales hasta nuevas incorporaciones.
- **Orden de patios**. Estructura que indica en qué orden se deben ir recorriendo los patios durante el turno. Es cambiante en el tiempo y condiciona los informes.
- **Turno**. Estructura única que sirve como cuadrante horario para organizar los turnos de los voluntarios durante la semana.

5

Desarrollo del software

Comenzaremos hablando de cómo se estructura nuestra aplicación a nivel de código para ir poco a poco entrando en los detalles más internos de cómo está implementado todo nuestro software.

Tenemos dos partes completamente independientes y separadas, que son la parte backend (donde se incluye todo lo relacionado al servidor) y la parte frontend (donde ocurre lo mismo para el cliente).

5.1 Backend

5.1.1 Configuración

Empezaremos detallando cuál es la configuración del servidor y los archivos que la llevan a cabo.

Por una parte, tenemos la estructura del proyecto de backend, la cual consiste en un paquete de Python siguiendo el diseño "flat", lo que implica que el directorio del paquete se encuentra directamente dentro del directorio superior, en contraposición al diseño "src", donde este se encuentra dentro de una carpeta con dicho nombre.

Dentro del directorio superior tendremos varios archivos de configuración, los cuales son:

- **.env**: archivo que contiene todas las variables de entorno del proyecto. En nuestro caso solo necesitamos tres: JWT_SECRET_KEY, JWT_ALGORITHM y RESEND_API_KEY.
- **pyproject.toml**: archivo principal de configuración del proyecto. En este archivo se especifican los datos del proyecto como el autor, la versión, la descripción, pero sobre todo lo más importante, las versiones de las diferentes dependencias del proyecto.
- **poetry.lock**: este archivo es un registro de las dependencias gestionadas por Poetry (el gestor de dependencias que usamos en el proyecto). Cumple la función de asegurar que todos los participantes del proyecto usen exactamente las mismas dependencias, garantizando la reproducibilidad del proyecto.

Una vez tenemos claro los archivos del exterior, debemos comentar los archivos de configuración de dentro del paquete. En este caso, lo primero a destacar que nos encontramos al entrar en el directorio *proteapp* es el punto de entrada del proyecto, el archivo *__main__.py.*

Este archivo es el encargado de levantar el servidor de Uvicorn que ejecuta nuestra API. Además, según si el parámetro *mode* del script recibe el valor "dev", también levantará un contenedor de Docker que ejecutará una instancia de la base de datos NoSQL MongoDB. De esta forma, cuando el proyecto esté en modo desarrollo podremos utilizar nuestra propia base de datos local y cuando lo despleguemos en modo producción solo debemos indicarle a qué base de datos debe conectarse.

Una vez se levanta la API, el siguiente script de configuración en ejecutarse es el archivo *main.py* dentro de nuestro directorio *api.* Este fichero es el encargado de ejecutar la configuración que repercute directamente en la API. Como tal, se encarga de iniciar las conexiones a las bases de datos (NoSQL y SQL), introducir algunos datos de prueba dentro de estas (solo necesario mientras se ejecuta el desarrollo de la aplicación) e inicializar el cuadrante de turnos vacío en caso de no tener guardados datos previos.

Además, también es el punto desde el que se configuran los middlewares de la API, en nuestro caso solo necesitamos un middleware para gestionar el CORS, y cada uno de los módulos independientes de la API se unen bajo la misma aplicación de FastAPI.

5.1.2 Imágenes

A lo largo del proyecto, tanto en la sección de Animales como en la de Usuarios, vamos a necesitar poder almacenar las imágenes de ambos. Para esto, fuera del paquete de Python del proyecto se crea una carpeta llamada *images*, que contendrá todas las imágenes subidas por los usuarios, identificadas con un identificador único (ULID).

Para esto, es necesario que los endpoints que recogen los datos de las imágenes que se quieren subir se encarguen de asignar este nombre y almacenar los ficheros en la carpeta

correspondiente. Por ello, tenemos una función llamada *save_image* que se encargará de todo esto y veremos con más detalle en el apartado de dependencias.

Por otro lado, para hacer que estas imágenes estén disponibles para mostrar dentro de nuestra aplicación debemos tener la forma de disponibilizarlas. En este caso, la solución es crear un endpoint independiente del resto bajo la ruta "/images/{image}" que se encargue de acceder al directorio de imágenes, buscar la imagen solicitada y devolverla si existe, dando error en caso contrario.

5.1.3 Plantillas

El uso de plantillas HTML se debe a que desde la parte backend de la aplicación, en dos secciones concretas es necesario crear un formato preestablecido para mostrar una serie de datos a los usuarios.

En este caso, hablamos del cuadrante de animales que se genera en PDF para los voluntarios y del email de bienvenida a los usuarios registrados, dos casos de uso muy diferentes pero que necesitan lo mismo, mostrar un contenido cuyos datos varían, pero la estructura es la misma.

En el caso del PDF usamos HTML porque es un formato sencillo de formatear y estilizar que se puede convertir a PDF de manera simple. Mientras que para el email el HTML es ideal para poder mostrar un contenido algo más vistoso que no sea un correo simplemente de puro texto. Con estas premisas que justifican su utilidad, empleamos Jinja2, una librería de Python, para crear unas plantillas HTML que que luego enviamos completas al usuario.

Estas plantillas las almacenamos dentro del paquete de Python, en la carpeta *templates*.

5.1.4 Dependencias

Cuando hablamos de dependencias en este apartado, nos referimos a aquellas dentro del código que son necesarias para la ejecución de algún endpoint, no a las librerías o paquetes descargados que forman parte del proyecto.

Las dependencias son empleadas en el mecanismo de inyección de dependencias propio de FastAPI (ej. get_register_email_sender, get_logged_user_http, ...) para evitar tener que instanciar los objetos dentro de la implementación de los endpoint, de forma que desacoplamos la instanciación de la dependencia de su uso. Estas están declaradas dentro del archivo *deps.py* del directorio *api.*

A continuación, vamos a listar las dependencias que hemos necesitado y cuáles son sus funciones.

Sesión de base de datos relacional

En nuestro caso estamos usando SQLModel (*wrapper* de SQLAlchemy) para la gestión de la base de datos SQL. Este framework sigue el patrón de necesitar un motor único que se conecta a la base de datos e ir creando diferentes sesiones sobre este para cada operación que queramos hacer. Es por esto, que necesitamos una dependencia que cree y cierre la sesión para cada endpoint que haga uso de la base de datos SQL.

Esta dependencia, gracias al mecanismo de inyección de dependencias de FastAPI es tan sencilla de implementar como una función que cree la sesión y la devuelva.

```
def get_sql_session():
    with Session(sql_engine) as session:
        yield session
```

Ilustración 4. Dependencia de la sesión SQL

En ella, la clase *Session* se crea en la sentencia *with*, que automáticamente la elimina cuando su bloque termina. Por eso utilizamos un *yield* en lugar de un *return*, porque la ejecución de la función se pausa en ese momento para continuar en el sitio donde haya sido llamada. Cuando esta ejecución termine, se reanudará la dependencia y eliminará la sesión automáticamente.

Para poder usar esta dependencia en nuestros endpoints, debemos inyectarla, de forma que podamos acceder a ella sin ejecutar la función directamente. Para esto FastAPI proporciona la clase *Depends*, que se encarga de hacer esto de forma automática.

```
@router.get("/search")
def get_animals(session: Session = Depends(get_sql_session)):
    animals = session.exec(select(Animal)).all()
    return animals
```

Ilustración 5. Ejemplo de uso de la dependencia de la sesión SQL

Usuario con la sesión iniciada

Al igual que necesitamos la sesión de la base de datos en los endpoints, también necesitamos comprobar en la mayoría que el usuario ha iniciado sesión correctamente (e incluso en la sección del perfil de usuario necesitaremos acceder a este usuario directamente).

Para esto, lo mejor que podemos hacer es crear una dependencia que se encargue de obtener el token de autenticación de la petición, decodificarlo (comprobando que sea correcto) y asegurarse de que el usuario que envía en él sea un usuario registrado.

Para facilitar esto, FastAPI nos ofrece a su vez dos dependencias que vamos a utilizar: *OAuth2PasswordBearer* y *SecurityScopes*. De la segunda hablaremos en el siguiente apartado, mientras que la primera nos servirá para recuperar el token de autenticación que el cliente enviará en la cabecera de las peticiones que realice.

Una vez tenemos el token disponible, debemos decodificarlo llamando a la función *decode_token*. Esta se encarga de comprobar la firma del token (asegurando que no ha sido modificado) y recuperar los datos enviados en él, como el identificador del usuario que ha iniciado sesión.

```python
def decode_token(token: str, security_scopes: SecurityScopes) -> TokenData:
    try:
        payload = jwt.decode(
            token, os.environ["JWT_SECRET_KEY"], algorithms=[os.environ["JWT_ALGORITHM"]]
        )
        user_id: str | None = payload.get("sub")
        scopes: list[str] = payload.get("scopes", [])

        if not user_id:
            raise TokenDecodificationError("Could not validate credentials")

        token_data = TokenData(user_id=user_id, scopes=scopes)

        for scope in security_scopes.scopes:
            if scope not in token_data.scopes:
                raise TokenDecodificationError("Not enough permissions")

    except (InvalidTokenError, ValidationError):
        raise TokenDecodificationError("Could not validate credentials")

    return token_data
```

Ilustración 6. Método de decodificación y validación del token

En el momento en el que tenemos el identificador del usuario obtenido del token, lo que debemos hacer es buscar a este en la base de datos, devolviéndolo una vez encontrado o dando un error si no se encuentra.

De esta forma, implementar una comprobación de autenticación en los endpoints es tan sencillo como usar esta dependencia en su declaración, pudiendo utilizar el usuario que da como respuesta en el cuerpo del endpoint, o no, solamente usándolo para comprobar el inicio de sesión.

```python
@router.post("/image")
def post_profile_image(
    ...,
    my_user: User = Depends(get_logged_user_http),
):
    ...
```

Ilustración 7. Ejemplo de uso de la dependencia de autenticación

Una vez claro cómo funciona todo el proceso, debemos explicar que en el caso de los websockets la autenticación es diferente. El estándar OAuth2, concretamente con el flujo que estamos usando nosotros, especifica que el token ha de pasarse de cliente a servidor como una cabecera HTTP llamada *Authorization*.

Esto, en el caso de los websockets no es posible, ya que no están pensados para poder llevar cabeceras adicionales, por lo que debemos utilizar otra forma para pasar el token del cliente al servidor.

- La primera opción era sobreescribir la cabecera *Sec-Websocket-Protocol* con el *Bearer* token, ya que esta sí que es enviada por defecto. El problema de esta opción es que estaríamos usando una cabecera reservada con un valor fuera del estándar, por lo que no nos parecía una buena práctica.
- La segunda opción era pasar el token como un parámetro en la URL, lo que permitiría que el servidor pudiera recibirlo sin complicaciones. Esta fue la opción empleada.

Como a partir de este momento existe una diferencia en la obtención del token de la petición del cliente, era necesario que implementáramos dos dependencias diferentes para obtener el usuario con la sesión iniciada, una para peticiones HTTP y otra para websockets: *get_logged_user_http* y *get_logged_user_ws*.

Usuario administrador

Además de comprobar que el usuario haya iniciado sesión, para ciertas secciones y funcionalidades de la aplicación también tenemos que comprobar que el usuario posea el rol de administrador, ya que este es un rol con privilegios que permite acciones adicionales.

Para hacer esta comprobación vamos a utilizar los *scopes* del estándar OAuth2, que no son más que permisos que indican qué tipo de acciones puede llevar a cabo el usuario. Los *scopes* van generalmente dentro de los datos del token, junto al identificador del usuario. Para utilizar estos *scopes* vamos a emplear las clases *Security* y *SecurityScopes* que nos facilita FastAPI.

- **Security**: tiene un comportamiento similar a la clase *Depends* para ejecutar la inyección de dependencias, explicada anteriormente, salvo que esta añade la

funcionalidad de poder requerir ciertos permisos a través de los *scopes.* En el momento de su uso, idéntico al de *Depends*, solo debemos añadir un nuevo parámetro *scopes* en formato lista de strings que indica qué permisos necesita esa dependencia para funcionar.

- **SecurityScopes**: se utiliza al momento de comprobar que el usuario tiene los permisos (*scopes*) requeridos. Cuando se usa como parámetro de una función que actúa como dependencia, automáticamente obtiene los valores de todos los *scopes* que se hayan requerido hasta el momento en el flujo de ejecución del endpoint. De esta forma, al momento de comprobar los permisos, tenemos una lista simple con todos ellos, independientemente del número de dependencias ejecutadas y los *scopes* adicionales que estas requieran.

Una vez que conocemos estas dos clases, es sencillo observar que para poder comprobar que el usuario es administrador, lo que vamos a hacer es:

1. Incluir el rol en el token que el backend general para el usuario (*regular* o *admin*).

2. Crear una nueva dependencia basada en la anterior que exigía que el usuario hubiera iniciado sesión. Esta nueva dependencia además usará la clase *Security*, en lugar de *Depends*, para incluir el rol de *admin* como *scope*.

```
def admin_required(user: Annotated[User, Security(get_logged_user_http, scopes=["admin"])]):
  pass
```

Ilustración 8. Dependencia para comprobar el rol de administrador

3. Cuando se compruebe la validez del token, recopilaremos los *scopes* que se han solicitado con la clase *SecurityScopes* y comprobaremos que el token incluya todos los necesarios, como se contempla en la ilustración 6.

De esta forma, para requerir que un usuario sea administrador para poder hacer la llamada a un endpoint, nos basta con utilizar la nueva dependencia en lugar de la que exclusivamente requería iniciar sesión.

```python
@router.post(
    "/shift/clean",
    status_code=status.HTTP_204_NO_CONTENT,
    dependencies=[Depends(admin_required)],
)
async def clean_shift():
    ...
```

Ilustración 9. Ejemplo de uso de la dependencia para comprobar el rol de administrador

5.1.5 Modelos

La declaración de los modelos de datos depende de qué tipo de modelo sea (SQL o NoSQL), aunque para hacer este proceso lo más uniforme posible hemos implementado nuestras propias clases base que permitan declarar los modelos de una forma muy similar, con todo lo necesario para funcionar, sin tener que especificar la misma configuración en todos los modelos.

Los comportamientos comunes que deben tener todas nuestras clases e instancias almacenadas en bases de datos son:

- Deben contener un identificador único en formato ULID (las clases en Python usarán la clase ULID, mientras que la representación en la base de datos dependerá de cada una).
- Deben crear el valor de este identificador al momento de instanciar una nueva clase, de forma que no exista instancia sin identificar.
- Cuando se serialicen para enviar los datos al cliente, deben convertir los nombres identificativos de los campos de la convención *snake_case* a *camelCase*, para seguir los estándares de cada lenguaje.

Con estas restricciones, ya tenemos claro que funcionalidades comunes debemos añadir a los modelos que creemos, independientemente de la base de datos en la que vayan a ser almacenados.

Las dos librerías que usamos para transformar los datos de las bases de datos a instancias en Python, ya sea SQLModel como **ORM** (Object Relational Mapper) o Beanie como **ODM** (Object Document Mapper), están basadas en Pydantic, la cual es una librería ideada para la validación y modelado de datos. Esto implica que los objetos que ambas manejan son a su vez modelos de Pydantic, con toda la funcionalidad extra que esto aplica.

Por una parte, tenemos la clase *SQLModel* de SQLModel, que es la base de aquellas clases que son tablas en nuestra base de datos relacional, y por otro lado tenemos la clase *Document* que es el equivalente de Beanie para aquellas clases que sean documentos de la base de datos no relacional.

A estas clases le vamos a sumar algunas más que nos ayuden a conseguir cumplir las restricciones mencionadas anteriormente.

Por una parte, tenemos la clase *BaseSchema*, esta es la clase más básica de todas, se basa en Pydantic para convertir los nombres de los campos al serializarlos de una convención a otra. Todas las clases de nuestra aplicación que vayan a ser utilizadas como esquemas o modelos deben heredar de esta clase.

Seguido a esta tenemos la clase *ULIDSchema*, que solo declara el campo id de tipo ULID sobre la clase *BaseSchema*, antes mencionada. Esta clase nos servirá para unificar a todas aquellas clases que contengan un id entre sus atributos, con lo que conseguimos que sea fácilmente modificable si fuera necesario. No todas las clases que empleemos necesitarán un identificador, por lo que algunas no heredarán de esta clase si no es así.

```
class BaseSchema(BaseModel):
    model_config = ConfigDict(alias_generator=to_camel, populate_by_name=True)


class ULIDSchema(BaseSchema):
    id: ULID
```

Ilustración 10. Declaración de las clases base

A partir de este momento, ya tenemos las dos primeras restricciones cubiertas simplemente heredando los modelos de las clases que acabamos de crear, según corresponda, aunque aún no hemos indicado en ningún momento que se almacenen en alguna base de datos.
A continuación, debemos diferenciar el modelo base de SQL y NoSQL, ya que su implementación será diferente.

Por una parte, tenemos el modelo SQL llamado *SQLULIDSchema*. Este modelo, utiliza la clase *Field* de SQLModel para especificar que debe generar el identificador al momento de crear la instancia, así como que este actuará de clave primaria. Además, debido a que SQLAlchemy (librería que envuelve SQLModel) no tiene compatibilidad con el tipo ULID, debemos crear una clase que especifique cómo esta debe tratar el nuevo tipo cuando quiera almacenarlo en la base de datos, básicamente convirtiéndolo en una cadena de texto.

```python
class SQLAlchemyULIDType(TypeDecorator[ULID]):
    impl = String

    cache_ok = True

    def process_bind_param(self, value: ULID | None, dialect: Dialect) -> str | None:
        return value if value is None else str(value)

    def process_result_value(self, value: str | None, dialect: Dialect) -> ULID | None:
        return value if value is None else ULID.from_str(value)


class SQLULIDSchema(ULIDSchema, SQLModel):
    id: ULID = Field(default_factory=ULID, primary_key=True, sa_type=SQLAlchemyULIDType)
```

Ilustración 11. Clase base para los modelos SQL

Por otro lado, tenemos el modelo NoSQL, llamado *NoSQLULIDSchema*. Este modelo, utiliza directamente la clase *Field* de Pydantic para especificar lo mismo que en el modelo anterior, que el identificador se genera de manera automática.

Además, también debemos especificar cómo se guarda el dato en la base de datos (*encoder*), ya que MongoDB tampoco tiene por defecto compatibilidad con el tipo ULID. Dentro de los *encoders* incluimos también uno para el tipo time que usamos en los modelos de los informes, ya que por defecto no sigue el formato ISO.

Por último, en este caso, también es necesario especificar cómo debe tratar el campo en la serialización (convirtiéndolo a cadena), ya que en el modelo SQL esto lo hacía SQLModel por defecto.

```python
class NoSQLULIDSchema(ULIDSchema, Document):
    id: ULID = Field(default_factory=ULID)

    @field_serializer("id")
    def serialize_ulid(self, id: ULID):
        return str(id)

    class Settings:
        validate_on_save = True
        bson_encoders = {ULID: str, time: time.isoformat}
```

Ilustración 12. Clase base para los modelos NoSQL

Con todas estas clases base especificadas, ahora la declaración de los modelos es prácticamente transparente a la base de datos donde se guarden, solo teniendo que heredar de una base u otra según lo que necesitemos.

5.2 Frontend

5.2.1 Configuración

Comencemos explicando cómo está estructurado el proyecto de frontend de la aplicación. Por una parte, en la raíz del proyecto tenemos múltiples archivos de configuración que se encargan de ajustar cómo funcionan las diversas tecnologías que hemos usado, vamos a obviar aquellos menos relevantes o que están configurados por defecto para explicar los más importantes como:

- **tailwind.config.js**: configura todo lo relacionado con TailwindCSS. En este archivo es donde se especifica el paquete de componentes visuales (DaisyUI) que vamos a incluir, así como el paquete de iconos que vamos a utilizar en la aplicación (Mingcute).
- **package.json**: especifica todas las versiones de los paquetes que estamos usando para el proyecto. En nuestro caso hemos utilizado npm como gestor de paquetes, de forma que es con esta herramienta con la que gestionamos las dependencias que luego se escriben en este archivo.
- **tsconfig.json, tsconfig.node.json, tsconfig.app.json**: son tres archivos encargados de la configuración de TypeScript, uno genérico y los otros dos dependientes del entorno de ejecución de la aplicación. En este archivo se especifica la versión de la especificación de ECMAScript que vamos a utilizar, en nuestro caso ES2020.

5.2.2 Navegación (router)

Pese a ser nuestra aplicación una SPA (una única página), las buenas prácticas y accesibilidad nos indican que es ideal que el usuario pueda ver reflejado en la URL las diferentes secciones de la aplicación a la que está accediendo, es por esto que para conseguir este comportamiento vamos a utilizar un router.

Un router no es más que un software (en nuestro caso una librería de Vue) que se encarga de enlazar diferentes URLs del navegador con componentes que debe renderizar cuando el usuario cargue dichas URLs.

De esta forma conseguimos por ejemplo que cuando el usuario clique un botón podamos transportarlo a otra zona de la aplicación diferente, así como mantener un historial de navegación sobre el que se puede avanzar o retroceder (funcionalidad que nos viene bien en caso de que se produzca algún error, ya que podremos mover al usuario a la última vista correcta).

En nuestro caso, el patrón que hemos seguido para diseñar las rutas es el siguiente:

1. Diferenciar a qué módulo de la aplicación se va a acceder (inform, people, shift, ...), incluyendo los submódulos en caso de haberlos (volunteers dentro de people).
2. En caso de querer acceder al listado o página principal, usar directamente el nombre del módulo (/animals, /inform, ...)
3. En caso de que se quiera crear un nuevo elemento dentro del módulo, utilizar el nombre del módulo seguido del verbo create (/animals/create, /people/create, ...)
4. Si queremos obtener los detalles de un elemento en concreto, usar el nombre del módulo seguido del identificador del elemento (/animals/1, /inform/4, ...)
5. Si por el contrario se quiere editar un elemento en concreto, debemos usar el nombre seguido del identificador y el verbo edit (/animals/1/edit, /people/8/edit, ...)

Como indicación adicional, destacar que la ruta de inicio de la aplicación (/) nos redirige automáticamente a la pantalla de login de esta (/login) o al inicio de animales (/animals) según si tenemos la sesión iniciada o no.

Además, si no tenemos la sesión iniciada e intentamos entrar en una página diferente del login mediante la URL, la aplicación nos redirigirá automáticamente de vuelta a este. De la misma forma, si intentamos acceder a una de las rutas restringidas para administradores cuando nuestro usuario no lo es, la aplicación nos llevará de vuelta al inicio (/animals).

5.2.3 Componentes globales

Dentro del desarrollo de la aplicación vamos a encontrar componentes específicos para una vista o sección concretas que están localizados en el módulo en el que se usan, pero también tendremos otros componentes pensados para ser reutilizados en partes diferentes de la aplicación, tanto por reusabilidad y facilidad de modificación como por intentar brindar un estilo común a todas las vistas de la aplicación.

Estos componentes reutilizables los hemos denominado componentes globales, y se pueden localizar en el directorio component*s* dentro de la carpeta principal (*src*) del proyecto. En él encontraremos principalmente inputs de diferentes tipos (*TextInput*, *DateInput*, *HourInput*, ...) pero también otros componentes que nos ayudarán a mostrar partes de la aplicación similares de una forma común (*ItemList*, *ItemSelector*, ...). Vamos a mencionar los más importantes y a explicar en qué consiste cada uno de ellos.

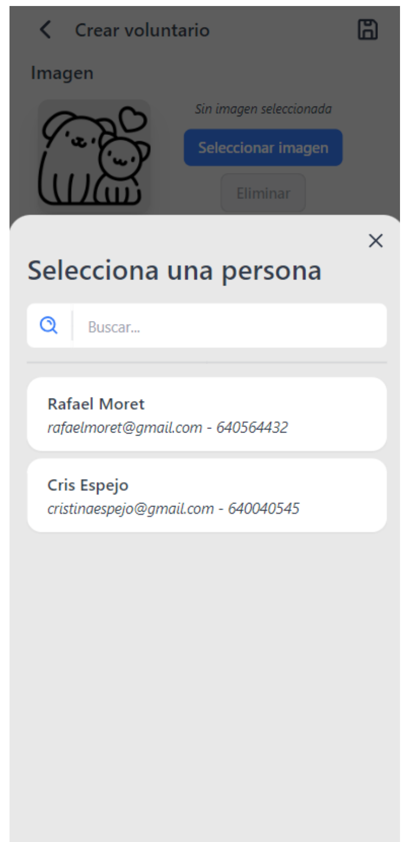
ItemSelector

Es un componente que permite la renderización de una lista de elementos con opciones de búsqueda y selección. De forma predeterminada, muestra los elementos de la lista utilizando un "slot" que permite personalizar la forma en que se representa cada elemento. El

desarrollador puede definir cómo se visualizan los elementos seleccionados y no seleccionados, y proporcionar un diseño personalizado (como tarjetas u otras vistas).

Este componente también incluye una funcionalidad de búsqueda opcional que, al activarse, muestra un campo de búsqueda en la parte superior. A medida que el usuario escribe en este campo, los resultados de la lista se filtran en tiempo real según una función de búsqueda configurable, que determina qué propiedad de cada elemento se usa para el filtrado.

Para gestionar la selección, el componente permite seleccionar o deseleccionar elementos de la lista. Si un elemento está seleccionado, se elimina de la lista de seleccionados al hacer clic nuevamente, y si no lo está, se añade, siempre que no se haya alcanzado el límite máximo de selección configurado.



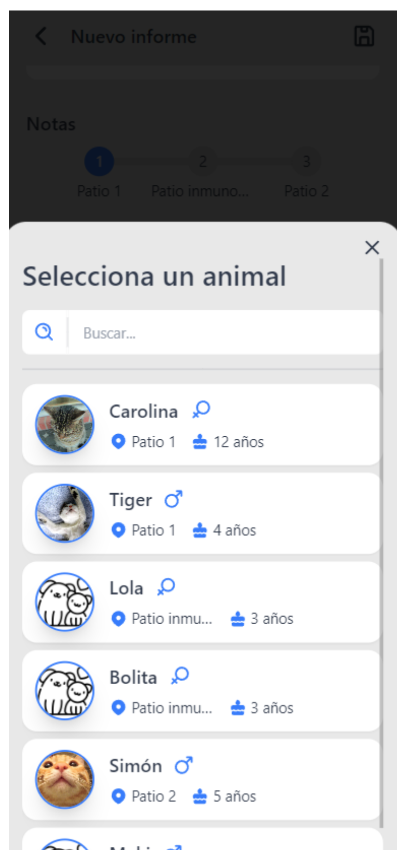


Ilustración 13. Ejemplos de ItemSelector

Define como parámetros (*props*) la lista completa de elementos (*items*), la lista de elementos seleccionados (*modelValue*), la opción de incluir un campo de búsqueda (*includeSearch*), una función personalizada de búsqueda (*searchFn*) y el número máximo de elementos seleccionables (*maxSelections*). Además, emite eventos para actualizar la lista de elementos seleccionados en función de las acciones del usuario.

BottomDrawer

Es un componente que permite mostrar un panel deslizable desde la parte inferior de la pantalla, con opciones de tamaño y un botón de cierre configurable. Este componente siempre se muestra por encima de todos los demás contenidos.

El componente permite ajustar su tamaño mediante la propiedad size, que acepta valores distintos tamaños prefijados de como big, small, medium o tiny, definiendo así la altura que ocupa en la pantalla. Por defecto, el tamaño es medium. Además, cuenta con la opción includeClose que, si está activada, muestra un botón de cierre en la parte superior del panel.

El panel se controla mediante una propiedad isOpen que determina su visibilidad. Cuando el panel se cierra, ya sea mediante el botón de cierre o haciendo clic fuera de él, se activa una animación de deslizamiento hacia abajo (slide-out), y se emite un evento de cierre para notificar al componente padre de que el panel ha sido cerrado. Esta animación inversa se asegura de que la transición sea fluida y agradable al usuario.

El componente define como parámetros (props) si incluye el botón de cierre, el tamaño del panel y emite eventos como close cuando el panel se cierra. La apertura y cierre del panel están gestionados por las propiedades reactivas isOpen e isClosing, que permiten controlar su estado y la animación de cierre. Las imágenes mostradas en el anterior componente son un ejemplo de este componente.

ItemList

Se trata de un componente que permite la renderización de una lista de elementos. Por defecto mostrará un componente predefinido simple para cada elemento en la lista, que simplemente listará todas las propiedades del elemento. Esto es configurable para poder utilizar un componente personalizable (por ejemplo, una tarjeta específica para ese tipo de elemento).
Además, el componente incluye un menú secundario como confirmación de borrado, ya que cada elemento de la lista incluye un botón para eliminar ese elemento. Para ello, define como parámetros (props) la lista que se quiere mostrar, el texto configurable del menú de borrado y diversos valores para configurar cómo se muestran los datos en el componente predefinido.

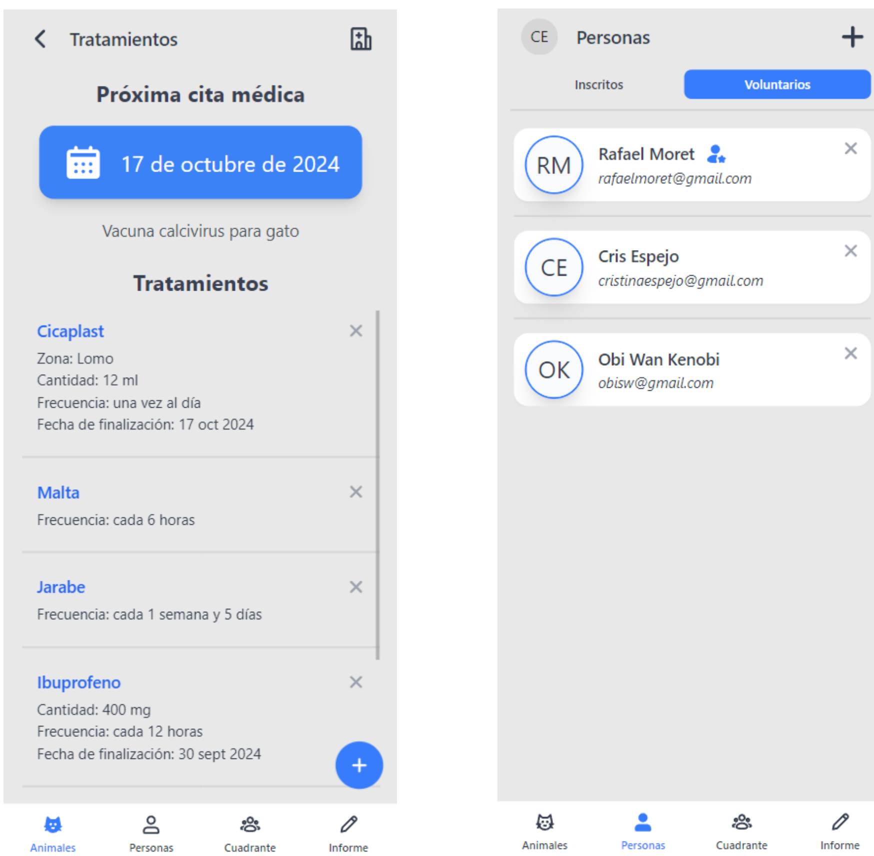


Ilustración 14. Ejemplos de ItemList

TextInput

Es un componente de entrada de texto que proporciona una interfaz personalizable con soporte para íconos y diferentes tipos de entradas. Este componente permite a los usuarios ingresar texto en un campo de entrada que puede ser configurado con diferentes atributos, como el nombre del campo, el tipo de entrada (por ejemplo, texto, contraseña, correo electrónico), y un marcador de posición (placeholder) que orienta al usuario sobre el contenido esperado.

El componente admite la inclusión de un ícono a través de un "slot" llamado icon. Si se proporciona este ícono, se muestra dentro de un contenedor a la izquierda del campo de entrada. Esta opción es útil para añadir íconos contextuales que indiquen el propósito del campo, como una lupa para búsquedas o un candado para contraseñas.
El componente acepta parámetros (props) como nombre, para asignar un nombre e ID al campo de entrada, placeholder, para mostrar un texto sugerido cuando el campo está vacío, y type ,para especificar el tipo de datos permitidos. El modelo de datos (model) enlaza el valor introducido por el usuario, permitiendo que el componente sea reactivo y que los cambios se sincronicen con otros elementos de la interfaz de usuario o la lógica de la aplicación.

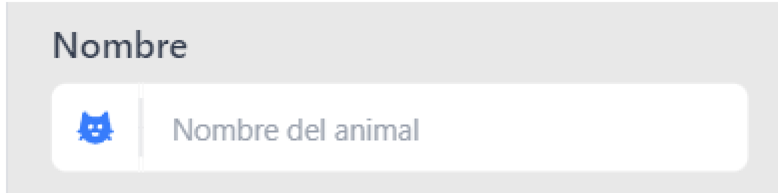


Ilustración 15. Ejemplo de TextInput

DateInput

Es un componente de selección de fechas. El componente utiliza @vuepic/vue-datepicker, una biblioteca de componentes de vue, para ofrecer una funcionalidad de calendario avanzada.
Permite incluir un ícono de calendario al lado del campo de entrada, que facilita la interacción del usuario al indicar visualmente la función del campo.
El formato de la fecha se puede ajustar con las propiedades dateFormat, que además permite elegir entre un formato de fecha corto o largo.

Además, el campo de entrada puede permitir a los usuarios borrar rápidamente la selección, y se puede configurar un texto de marcador de posición (placeholder) para mostrar cuando el campo está vacío.

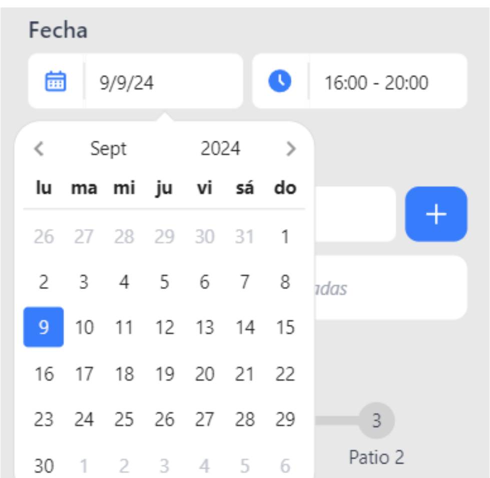


Ilustración 16. Ejemplo de DateInput

TimeInput

Es un componente de entrada para gestionar y seleccionar unidades de tiempo, como segundos, minutos, horas y días. Este componente está diseñado para permitir a los usuarios

introducir un valor numérico para una unidad de tiempo específica y cambiar entre diferentes unidades mediante un menú desplegable.
El componente acepta varias propiedades configurables a través de props. Estas incluyen modelValue, que representa el valor inicial de tiempo a mostrar, y name, que se utiliza para asignar nombres únicos a los campos de entrada. También se pueden especificar timeSize para ajustar el tamaño del campo de entrada (pequeño o grande) y units para seleccionar qué unidades de tiempo están disponibles en el menú desplegable.

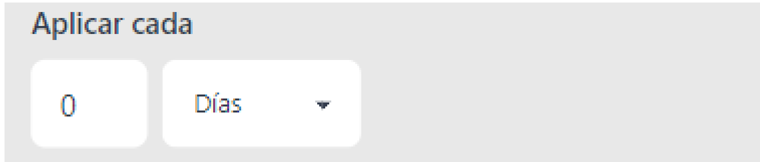


Ilustración 17. Ejemplo de TimeInput

HoursInput

Se trata de un componente de entrada pensado para recoger tramos horarios. De esta forma podemos indicar la hora de inicio y final de algún evento.
Incluye por defecto la comprobación de que la primera hora sea inferior a la segunda, ya que está pensado para tramos horarios dentro de un mismo día.

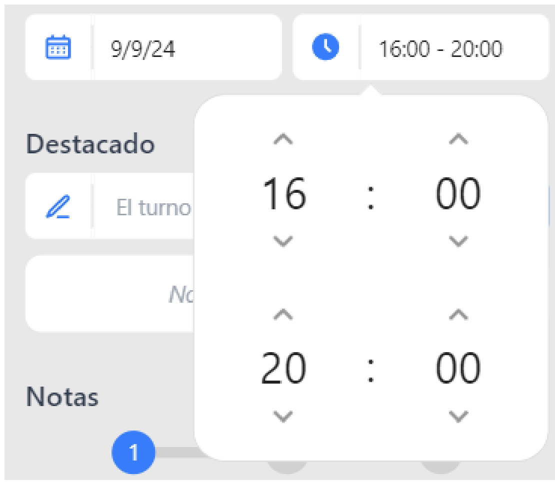


Ilustración 18. Ejemplo de HoursInput

DropdownSelector

Es un selector que despliega un menú flotante para que el usuario clique en el elemento deseado. Además, incluye la posibilidad de mostrar una barra de búsqueda (así como usar una función de búsqueda en ella) como en el ItemSelector.

La representación del botón clicable para abrir el menú es configurable, ya que el componente solo estila la parte del menú desplegable.

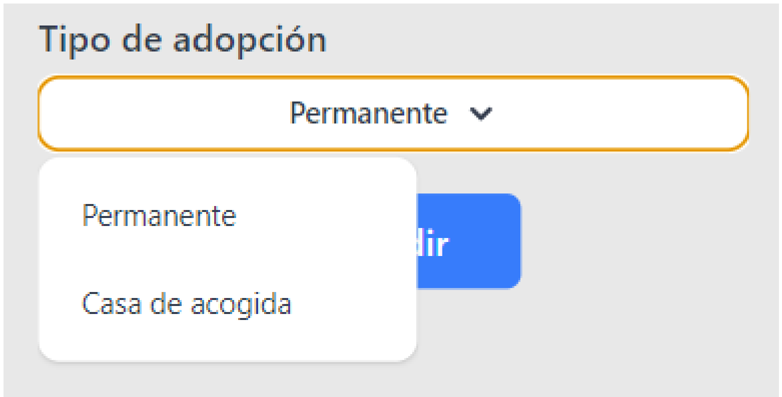


Ilustración 19. Ejemplo de DropdownSelector

ToastNotifications

Se trata de un componente que hace de panel de notificaciones, por lo que debe colocarse en aquellas vistas que vayan a lanzar notificaciones al usuario. Puede colocarse en cualquier lugar de la vista ya que internamente el gestiona la posición de las notificaciones para que aparezcan siempre en la parte superior de la pantalla.

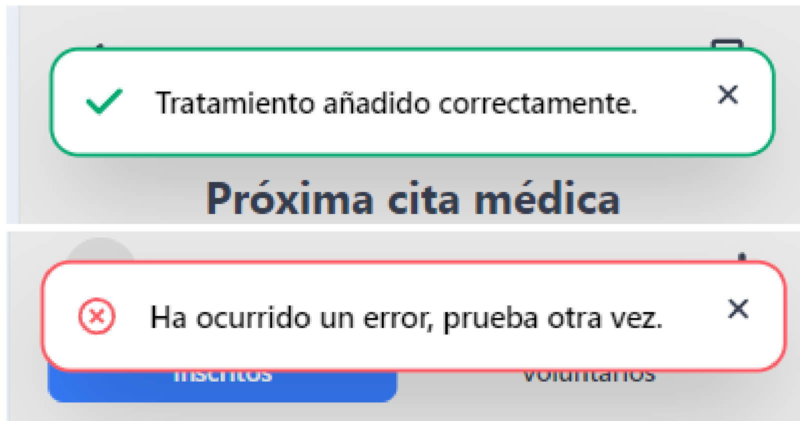


Ilustración 20. Ejemplo de notificaciones

El componente por sí solo no genera ninguna notificación, ya que solo sirve para establecer el panel donde se mostrarán. Para poder lanzar notificaciones es necesario utilizar el *composable* que funciona en conjunto con el componente: *useToastNotifications.*

```
const notificationsRef = ref<InstanceType<typeof ToastNotifications> | null>(null)
const { showErrorNotification, showSuccessNotification } = useToastNotifications(notificationsRef)
```

Ilustración 21. Ejemplo de uso del *composable useToastNotifications*

Este *composable* recibe como parámetro una referencia al componente *ToastNotifications* que habremos insertado en la vista. De esta forma sincronizamos las funciones del *composable* con el panel insertado.

El *composable* dispone de tres funciones para mostrar tres tipos de notificaciones predefinidas diferentes: *showErrorNotification*, *showSuccessNotification* y *showInfoNotification*; así como de una función adicional *addNotification* que permite configurar completamente la notificación (icono y estilos).

ImagePicker

Este componente cumple la función de subida y previsualización de imágenes personalizadas.

Para realizar esta función primero utilizamos la dependencia *useFileDialog* de *vue-use* para abrir una ventana de selección de archivo local al clicar en el botón principal.
Una vez seleccionado el archivo (en nuestro caso está configurado para solo permitir imágenes), se le crea una URL local a través de la dependencia *useObjectUrl*, también de *vue-use*, lo que nos permite tener una referencia a la imagen válida dentro de nuestra ejecución, sin tener que tratar de momento con los datos binarios de la imagen.

El componente usa el patrón de *modelValue* para gestionar esta referencia, de forma que cuando es obtenida se emite un evento de modificación de la prop *modelValue* para notificar al componente padre de la actualización de esta.

La previsualización de la imagen se hace mostrando un elemento imagen con la URL de la prop *modelValue*, o en caso de que esta no tenga valor, mostrando una imagen por defecto a modo de placeholder. De esta manera, también podemos mostrar una imagen precargada inicial pasando la URL de esta como prop *modelValue* al componente.

Ilustración 22. Ejemplo de ImagePicker

Usando la referencia en lugar de directamente el binario, cuando posteriormente queramos disponer de este, solo tendremos que hacer un *fetch* de la referencia para obtener los datos requeridos.

Además, el componente también incorpora un botón de eliminación que permite al usuario eliminar la imagen seleccionada (fijando así la referencia que estuviera creada en ese momento a indefinido).

Iconos Mingcute

Además, aunque no es un componente como tal, merece la pena destacar cómo hemos integrado los iconos de la aplicación en el código. Para esto hemos utilizado la herramienta de Iconify, con una de sus librerías para Javascript que da acceso a todo su catálogo de iconos (en realidad es una colección de muchos catálogos de iconos más reconocidos) de forma compatible con TailwindCSS.

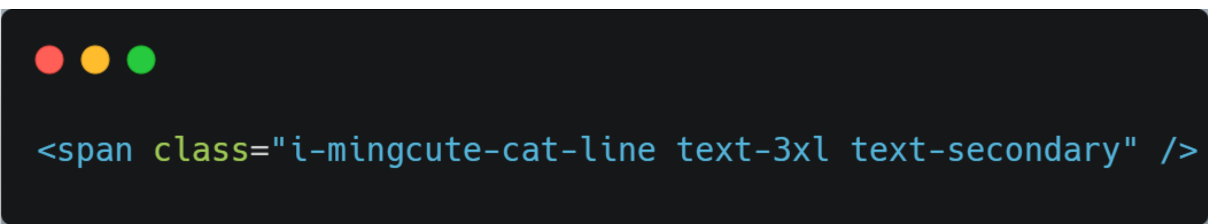


Ilustración 23. Ejemplo de uso de un icono

De esta forma, para nosotros es tan sencillo como insertar un elemento *span* con la clase de CSS adecuada para conseguir el icono que queremos emplear. Además, las propiedades de tamaño o color son perfectamente configurables a través de TailwindCSS, lo que lo hace perfectamente integrable con el resto de la aplicación.

5.2.4 Stores

Dentro de los elementos que forman un módulo en nuestra aplicación, tenemos los stores. Los stores de Vue (concretamente de Pinia), son estructuras de datos utilizadas para la gestión de estados dentro de la aplicación. Son una alternativa al típico flujo de transferencia de datos dentro de una aplicación frontend, el cual consiste en que los componentes superiores (padres), pasan props (datos) a los componentes inferiores (hijos), mientras que estos últimos notifican a sus padres de las actualizaciones lanzando eventos que van subiendo en el árbol de componentes.

Los stores, como alternativa son una solución mucho menos jerárquica que permite acceder a datos centralizados desde cualquier parte de la aplicación, sin tener que caer en esa cadena

de props y eventos. De esta forma, es ideal para gestionar variables que son utilizadas por varios componentes en varios lugares de la aplicación.

Estos estados centralizados pueden ser modificados (*actions*) y recuperados (*getters*) mediante funciones dentro del store. Con esto que acabamos de mencionar, podemos comentar que una buena práctica común en el desarrollo frontend es encapsular todas las peticiones externas (*fetch*) dentro de los stores. De esta forma, podemos utilizar los stores para gestionar los datos que recibe la aplicación del exterior (del backend, por ejemplo) y disponibilizarlos a lo largo de toda la aplicación.

Un ejemplo de esto sería el store de personas, como podemos ver en la captura inferior (se muestra solo una parte del store).

```
export const usePersonStore = defineStore('PersonStore', () => {
  const personList = ref<Person[]>([])
  const personDetails = ref<Person>()
  const isLoading = ref(false)

  async function fetchPerson(id: string) {
    isLoading.value = true

    await authFetch(`${import.meta.env.VITE_BACKEND_URL}/${crudPersonApi}/${id}`)
      .then((res) => res.json())
      .then(PersonAdapter)
      .then((person) => (personDetails.value = person))

    isLoading.value = false
  }

  async function fetchPeople() {
    isLoading.value = true
    await authFetch(`${import.meta.env.VITE_BACKEND_URL}/${listPeopleApi}`)
      .then((res) => res.json())
      .then((json) => json.map(PersonAdapter))
      .then((person) => (personList.value = person))

    isLoading.value = false
  }
}
```

Ilustración 24. Ejemplo de store

En este fragmento, los estados están declarados al inicio y las *actions* son las funciones que modifican dichos estados. Además, la petición se hace dentro del *action*, por lo que, por ejemplo, a la hora de consumir los datos de la lista de personas, nos basta con utilizar la variable *personList* y llamar al método *fetchPeople* cada vez que queramos refrescar su valor con los datos del backend.

De esta manera simplificaremos el uso de estos datos externos en las vistas y será más fácil acceder a ellos desde cualquier punto de la aplicación.

5.2.4.1 Adaptadores

Como hemos descrito, los stores los vamos a utilizar para realizar las peticiones al backend dentro de ellos, por lo que es una buena práctica comprobar siempre que los datos que se obtienen del exterior tienen la forma correcta. En este punto es donde introducimos el concepto de adaptador.

Los adaptadores son funciones que se encargan de comprobar que los datos que reciben como parámetros tienen la forma que se espera de ellos. Para esto, primero se define un esquema utilizando la librería Zod. Estos esquemas serán el contrato de datos que el adaptador se encargará de comprobar.

```
export const PersonSchema = z.object({
  id: IdSchema,
  name: z.string(),
  firstSurname: z.string(),
  phone: z.string(),
  secondSurname: z.string().nullish(),
  email: z.string()
})
```

Ilustración 25. Ejemplo de esquema

Los esquemas estarán siempre dentro de los archivos denominados *declarations.ts* en el directorio de cada módulo. Una vez tenemos el esquema pasamos a utilizarlo para crear el adaptador.

```
export const PersonAdapter = (input: any) => PersonSchema.parse(input)
```

Ilustración 26. Ejemplo de adaptador

De esta forma, cuando se utilice PersonAdapter para comprobar algún objeto, si este no cumple la estructura deseada el adaptador lanzará un error, mientras que si lo hace devolverá un objeto del tipo del esquema con los datos correspondientes.

Cada entidad de los módulos principales tiene sus adaptadores (declarados en los archivos denominados *adapters.ts*) y estos deben ser usados en el store del módulo para comprobar que los datos del backend son correctos.

5.3 Módulos

5.3.1 Animales

El propósito fundamental de este módulo es proporcionar una representación integral de toda la información relativa a los animales gestionados por la protectora. Esto incluye tanto un listado completo de todos los animales organizados por su ubicación dentro de las instalaciones, como una vista detallada de cada animal individualmente, mostrando todas sus características y atributos de manera detallada.

Actualmente, esta información se encuentra almacenada en documentos de Excel, que actúan como la principal "base de datos" del sistema. Estos documentos permiten la gestión y el filtrado de los datos a través de las herramientas y filtros integrados en el software de hojas de cálculo, aunque esta solución presenta limitaciones en términos de escalabilidad y accesibilidad comparado con sistemas de bases de datos más robustos y especializados. El objetivo del módulo es ofrecer una solución más estructurada y eficiente para la visualización y el manejo de la información de los animales, facilitando así el trabajo del personal de la protectora y mejorando la gestión de los recursos.

Interfaz de usuario - Listado de animales

Esta vista es una de las más cruciales de la aplicación, ya que permite visualizar de manera rápida y eficiente todos los animales presentes en cada patio de la protectora. Tener esta información centralizada y accesible facilita enormemente el trabajo de los voluntarios y coordinadores, ya que pueden localizar a un animal del patio rápidamente por su fotografía, o consultar datos específicos encontrándolo de manera rápida.

Otro requisito importante es poder buscar rápidamente con filtros. Estos filtros deben abarcar desde filtrar por nombre, hasta por características más concretas: edad, sexo, castración y si son compatibles con otros animales o no. Esto es crucial ya que, si vienen visitas a la protectora buscando animales para adoptar con algún requisito, permite filtrar rápidamente para buscar los que más se adapten, entre otros casos.

Por eso, al diseñar la interfaz que mejor se adaptara a estos requisitos, optamos por mantener una lista doble: una lista vertical para los patios, con desplazamiento hacia abajo, y una lista horizontal dentro de cada patio para los animales. Esta disposición permite visualizar rápidamente tanto la distribución de los patios como los animales presentes en cada uno, facilitando la navegación y el acceso a la información de manera eficiente.

El diseño del filtrado incluye una barra de búsqueda para localizar animales por nombre, mientras que los filtros más específicos se presentan en un menú desplegable. Cada filtro es interactivo y fácil de usar, permitiendo opciones de selección múltiple o rangos, como en el caso de la edad. Los resultados deben actualizarse en tiempo real al aplicar los filtros, mostrando únicamente los animales que coincidan con los criterios seleccionados.

Ilustración 27. Vista principal del listado de animales

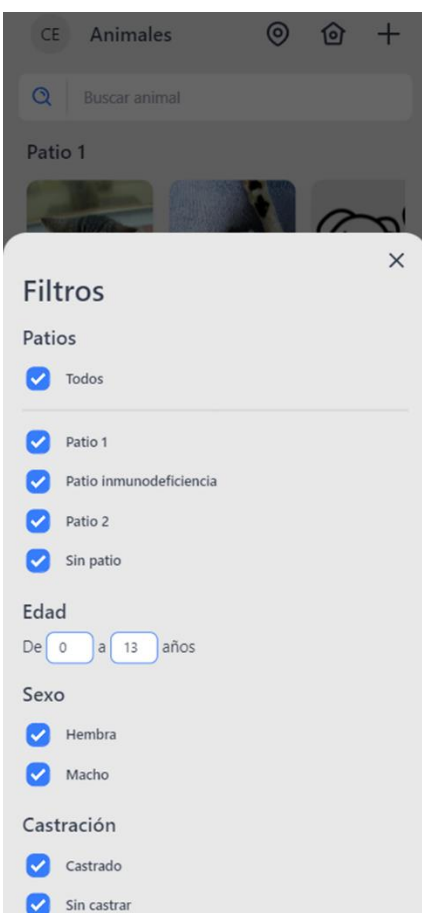


Ilustración 28. Menú adicional con el filtrado de animales

Para desarrollar técnicamente esto en Vue, la idea principal es usar un bucle para recorrer un conjunto de datos **animalsByYard,** la cual contiene listas de animales agrupadas por patio.

Este conjunto de datos se define con una propiedad computada que agrupa los animales filtrados por el nombre del patio al que pertenecen. Para ello, recorre la lista de patios (yardList) y crea un objeto donde cada clave es el nombre de un patio y cada valor es un arreglo de animales que pertenecen a ese patio. Además, filtra los animales que no tienen un patio asignado y los agrupa bajo la clave "Sin patio", combinando ambos conjuntos en un solo objeto para facilitar la visualización de los animales organizados por su ubicación en la interfaz de usuario.

```
const animalsByYard = computed(() => {
  const groupedAnimals = yardList.value.reduce(
    (acc, yard) => ({
      ...acc,
      [yard.name]: filteredAnimals.value.filter((animal) => animal.yard?.name === yard.name)
    }),
    {} as Record<string, AnimalInfo[]>
  )

  const nonYardAnimals = filteredAnimals.value.filter((animal) => animal.yard == undefined)
  return { ...groupedAnimals, 'Sin patio': nonYardAnimals }
})
```

Ilustración 29. Ordenación de los animales por patio

Por cada patio, se genera un contenedor para incluir el título del nombre del patio y un carrusel que presenta las tarjetas verticales para cada animal dentro de ese patio. Estas tarjetas se han realizado en un componente personalizado llamado **VerticalAnimalCard**, que muestran la información de cada animal (imagen, sexo y nombre) permitiendo además que, al hacer click en ellas, permiten navegar a una página de detalles específicos para ese animal.

```
<div class="flex flex-col gap-1 px-0">
  <div
    v-for="(animals, yard, yardNumber) in animalsByYard"
    :key="yard"
    class="flex flex-col px-6"
  >
    <h2 class="text-xl font-semibold">{{ yard }}</h2>
    <div class="carousel w-full gap-4 py-4">
      <VerticalAnimalCard
        v-for="animal in animals"
        :key="animal.id"
        :animal="animal"
        class="carousel-item rounded-lg shadow-lg"
        @click="navigateAnimal(animal.id)"
      />
    </div>
    <div v-if="yardNumber != yardList.length" class="divider m-1" />
  </div>
</div>
```

Ilustración 30. Distribución de los animales en la vista

Para el filtrado, se ha utilizado el componente **TextInput** descrito anteriormente para tener la funcionalidad de filtrado por nombre. Adicionalmente, para el filtrado por características, se ha desarrollado el siguiente componente:

- **AnimalFiltersMenuButton:** Este componente implementa un botón interactivo con una acción secundaria y un menú desplegable para aplicar filtros a una lista de animales. Integra varios componentes personalizados, como *ButtonWithSubaction, BottomDrawer* y *AnimalFiltersMenu*, para proporcionar una experiencia de usuario fluida e intuitiva al trabajar con filtros en la interfaz de la aplicación.

La propiedad *modelValue* que recibe este componente representa los filtros actualmente aplicados a los animales y está definida con el tipo *AnimalFilters*, asegurando que solo se transmitan los datos de filtro correctos. El componente principal que facilita el filtrado es *AnimalFilterMenu*, un menú de filtros altamente configurable que permite definir los criterios de filtrado.

Interfaz de usuario - Vista del animal

Esta vista muestra la información detallada de cada animal, dando prioridad a visualizar una fotografía de él ya que es importante para su identificación. La interfaz proporciona datos como su edad, características de comportamiento, estado de salud (como si está castrada) y su ubicación dentro de la protectora. También incluye una breve descripción de su personalidad y hábitos. La vista está optimizada para que los voluntarios puedan acceder rápidamente a la información esencial de cada animal y facilitar su cuidado o manejo dentro del refugio.

Ilustración 31. Vista detallada de un animal

Para el desarrollo de esta vista, se han creado dos componentes principales:

- **AnimalImage:** Este componente proporciona una solución elegante para la visualización de imágenes en una aplicación, manejando de manera eficaz los errores de carga y casos en los que no se proporciona una imagen.

- **AnimalDetails:** Este componente muestra detalles sobre un animal en forma de íconos y etiquetas informativas. Para ello, utiliza propiedades para determinar y mostrar información específica sobre el animal, como su estado de castración, compatibilidad con otros animales, ubicación y fecha de nacimiento. Además, dependiendo de cada característica booleana como si está castrado, si es compatible o su sexo, elige de manera dinámica un icono acorde con su valor.

Interfaz de usuario - Editor de animales

Esta vista está diseñada para que los voluntarios de la protectora puedan editar los datos de un animal dentro de la aplicación interna. Aquí se permite modificar la información básica del animal, como su nombre, imagen, sexo, fechas clave (nacimiento y entrada a la protectora), su ubicación (patio), así como su personalidad y detalles adicionales, como si está castrado o es compatible con otros animales. Esta funcionalidad es crucial para mantener actualizados los perfiles de los animales, garantizando que los voluntarios tengan acceso a la información más reciente y precisa. Esta vista se utilizará tanto para la edición de datos de animales como para la creación de un nuevo animal, ya que se adapta a ambas funcionalidades.

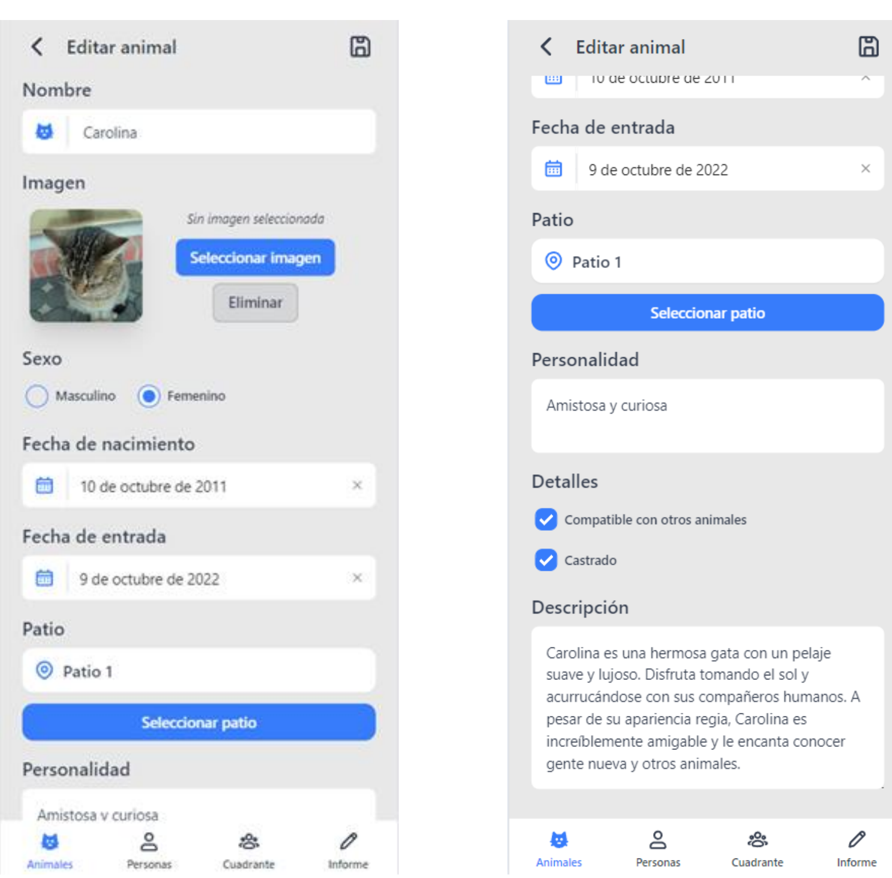


Ilustración 32. Vista del editor de animales

Para desarrollar esta interfaz, se ha optado por diseñarla de manera simple y clara, facilitando la modificación de datos con campos etiquetados, controles intuitivos, menús desplegables y selectores.

La mayor parte de estos componentes forma parte de los componentes globales que hemos comentado en uno de los apartados anteriores, como *TextInput*, *DateInput* o *ImagePicker*. Sin embargo, merece la pena destacar el componente:

- **YardSelector:** La funcionalidad de este componente radica en facilitar la selección de un patio de animales. Para esto, vamos a utilizar el componente previamente explicado *ItemSelector* con la lista de patios disponibles, ya que nos brinda un buscador por texto y una interfaz para efectuar la selección que podemos reutilizar perfectamente.

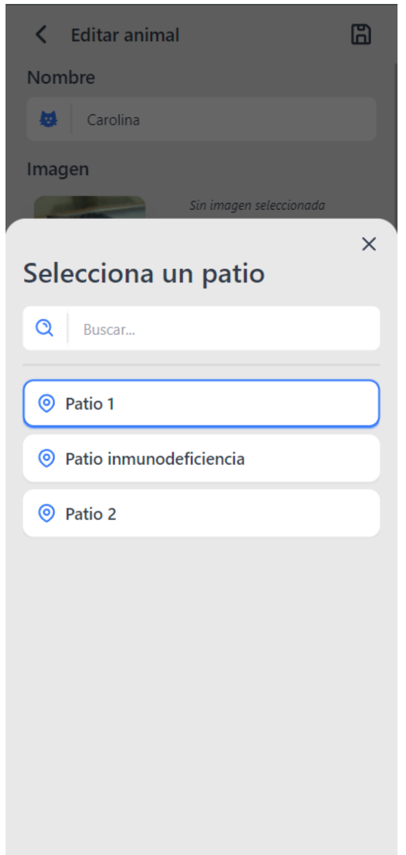


Ilustración 33. Menú adicional de selección de patio

Este componente, no se mostrará hasta que el usuario no clique en el botón principal "Seleccionar patio", el cual abrirá un menú adicional de tipo *BottomDrawer* con la lista de animales y el buscador de *ItemSelector*. En el momento que la selección se efectúa,

podemos cerrar el menú y el componente mostrará el resultado con una *YardCard* como las del selector, para dar retroalimentación de la decisión al usuario.

Además de estos componentes propios también se han utilizado para algunos campos elementos simples de HTML como inputs de tipo radio o textareas para los campos de texto de mayor tamaño.

```
<label class="label cursor-pointer gap-2">
  <input
    type="radio"
    name="radio-sex"
    class="radio-secondary radio"
    :checked="editingAnimal.sex === 'male'"
    @change="handleSexInput('male')"
  />
  <span class="label-text font-semibold">Masculino</span>
</label>
```

```
<h2 class="text-xl font-semibold">Personalidad</h2>
<textarea
  v-model="editingAnimal.personality"
  class="rounded-lg p-4"
  rows="2"
  placeholder="Una pequeña frase que resuma su comportamiento"
/>
```

Ilustración 34. Ejemplos de uso de elementos simples en el editor de animales

Store

Para realizar las solicitudes al servidor para obtener los datos necesarios y actualizar los estados de la aplicación con estos datos, definimos un store de animales llamado **AnimalStore**.

De este store, para esta vista se necesitará lo siguiente:

- **fetchAnimals():** es una función asíncrona que solicita una lista de animales desde la API REST utilizando *authFetch*, una función de utilidad para peticiones autenticadas. Posteriormente, procesa la respuesta en formato JSON, transforma los datos recibidos mediante el adaptador *AnimalInfoAdapter* para adaptarlos al formato requerido y actualiza el estado reactivo *animalList* con la lista de animales resultante.

- **fetchAnimal(id):** función asíncrona que obtiene los detalles de un animal específico desde el backend de la aplicación, utilizando un identificador único. El objetivo es

solicitar los datos de un solo animal y actualizar la variable reactiva del store *animalDetails*. Para esto, se utiliza el adaptador *AnimalAdapter* para comprobar el contrato de los datos.

- **createAnimal(animal):** esta función se encarga de hacer la llamada para la creación de un nuevo animal. Para esto, se comprueba el contrato de los datos con el adaptador EditAnimalAdapter y se envía una solicitud POST a la API para crear el registro del animal, excluyendo el campo de la imagen del cuerpo de la solicitud. Si el animal tiene una imagen asociada, la función obtiene el binario de la imagen a través de su URL y la envía a la API en una solicitud nueva POST para asociarla con el registro del animal creado. Finalmente, actualiza la lista de animales llamando a la función *fetchAnimals* para reflejar los cambios en la interfaz de usuario.

- **updateAnimal(id, animal):** esta función se comporta de manera similar a la función *createAnimal*. La diferencia principal radica en que en lugar de hacer una petición POST a la API, realiza una petición PUT de actualización. La imagen sigue gestionándose de manera independiente de forma idéntica a la creación del animal, salvo porque en caso de que los datos actualizados no incluyan imagen (se han dejado expresamente vacíos), lo que se realiza es una petición de DELETE indicando que hay que eliminar la imagen del animal. Por último, se llama a *fetchAnimals* para actualizar la lista de animales en la aplicación, reflejando el nuevo registro y su imagen en la interfaz de usuario.

- **isLoading:** es una variable reactiva que indica si la aplicación está en proceso de cargar datos. Es utilizada para controlar la visualización de indicadores de carga en la interfaz, asegurando que el usuario sepa que una operación asíncrona, como una solicitud al servidor, está en curso. Cuando se establece en true, muestra que la carga está en progreso, y cuando se establece en false, indica que la carga ha finalizado.

Modelo de datos

Para poder mostrar toda esta información, necesitamos una estructura que almacene a los animales y todas sus características. Adicionalmente establece relaciones de uno a muchos con otras entidades como tratamientos, citas y adopciones, incluyendo una única relación de muchos a muchos con patios.

```python
class Animal(SQLULIDSchema, table=True):
    name: str
    sex: Sex
    personality: str | None = Field(default=None)
    description: str | None = Field(default=None)
    birth_date: date | None = Field(default=None)
    entry_date: date | None = Field(default=None)
    is_animal_compatible: bool | None = Field(default=None)
    is_castrated: bool | None = Field(default=None)
    image: str | None = Field(default=None)

    yard_id: ULID | None = Field(default=None, foreign_key="yard.id", sa_type=SQLAlchemyULIDType)

    treatments: list["Treatment"] = Relationship(back_populates="animal")
    appointments: list["Appointment"] = Relationship(back_populates="animal")
    yard: Optional["Yard"] = Relationship(back_populates="animals")
    adopters: list["Adoption"] = Relationship(back_populates="animal")
```

Ilustración 35. Modelo de datos de un animal

Los únicos campos obligatorios son nombre y sexo, dejando el resto a opcionales ya que pueden no ser conocidos en el momento de la creación, por lo que necesitamos la mayor flexibilidad posible.

Endpoints

GET /animal/search

Se encarga de retornar todos los animales de la base de datos.

POST /animal/

Se encarga de añadir un animal a la base de datos.

POST /animal/{id}/image

Permite subir una imagen para un animal específico identificado por su id. Este guarda la imagen en el sistema de archivos y obtiene su ruta y, si el animal ya tiene una imagen asociada, elimina la imagen antigua antes de actualizar la ruta de la nueva imagen en el registro del animal.

DELETE /animal/{id}/image

Permite borrar una imagen de un animal concreto de la base de datos.

GET /animal/{id}

Retorna la información de un animal concreto de la base de datos.

PUT /animal/{id}

Permite la edición de datos de un animal existente en la base de datos.

DELETE /animal/{id}

Permite el borrado de un animal concreto de la base de datos.

Esquemas

Este módulo tiene varios esquemas para controlar cuál es el contrato de salida y entrada de datos de los endpoints.

```
class ListedAnimal(ULIDSchema):
    class Treatment(ULIDSchema):
        name: str

    class Yard(ULIDSchema):
        name: str

    name: str
    sex: Sex
    image: str | None
    yard: Yard | None
    birth_date: date | None
    is_animal_compatible: bool | None
    is_castrated: bool | None
    treatments: list[Treatment]


class CompleteAnimal(ListedAnimal):
    class Treatment(ULIDSchema):
        name: str
        zone: str | None
        frequency: int | None
        end_date: datetime | None
        amount: str | None

    class Appointment(ULIDSchema):
        date: datetime
        description: str
        is_past: bool

    personality: str | None
    description: str | None
    birth_date: date | None
    entry_date: date | None
    is_animal_compatible: bool | None
    is_castrated: bool | None
    treatments: list[Treatment]
    appointments: list[Appointment]
```

Ilustración 36. Esquemas de salida del módulo de animales

Por una parte, tenemos *ListedAnima* y *CompleteAnimal*, los cuales son los contratos utilizados como resultado en los endpoints que deben devolver un animal al finalizar su ejecución (en nuestro caso, todos los endpoints menos el DELETE del animal).

El primero es la versión reducida, usada sobre todo en la lista de animales, donde no necesitamos tanto detalle. El segundo sin embargo incluye todos los campos de un animal.

Por otro lado, tenemos el esquema de entrada, que recoge todos los datos que se pueden dar de un animal a la hora de añadirlo a nuestra base de datos o editarlo de la misma.

```
class EditableAnimal(BaseSchema):
    name: str
    sex: Sex
    personality: str | None = None
    description: str | None = None
    birth_date: date | datetime | None = None
    entry_date: date | datetime | None = None
    is_animal_compatible: bool | None = None
    is_castrated: bool | None = None
    yard: ULID | None = None
```

Ilustración 37. Esquema de entrada del módulo de anmales

5.3.2 Animales - Tratamientos

Este módulo, que forma parte en sí del módulo de animales, se enfoca en gestionar los tratamientos asociados a cada animal en particular. Es fundamental para los turnos, ya que permite visualizar los tratamientos que recibe cada animal y sus características, como la frecuencia de administración, el lugar de aplicación y la fecha de finalización. Además, debe permitir añadir nuevos tratamientos y eliminar los que ya están registrados.

Interfaz de usuario

La vista, además de la próxima cita médica de la cual se hablará en un futuro apartado, incluye una lista de los tratamientos que el animal está recibiendo. Cada ítem en la lista proporciona detalles relevantes sobre el tratamiento, como su nombre, zona de aplicación, cantidad a aplicar, frecuencia y fecha de finalización. Esta visualización listada se realiza mediante el componente explicado anteriormente *ItemList*.

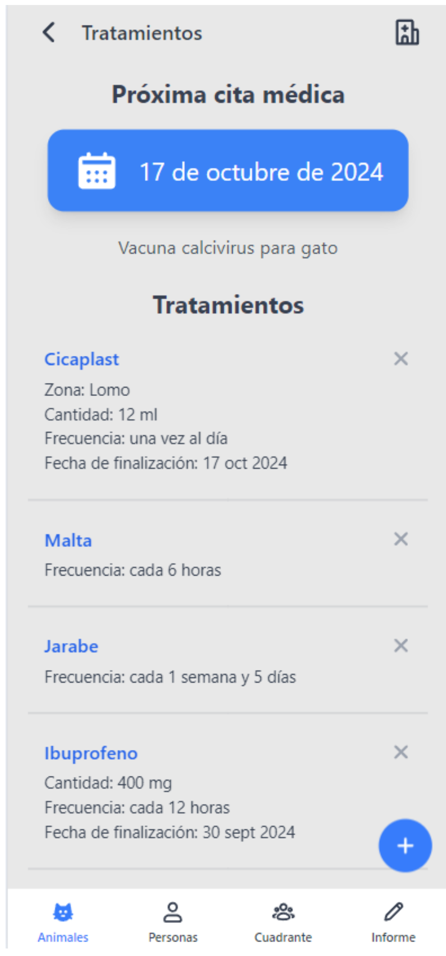


Ilustración 38. Vista de la lista de tratamientos de un animal

Para que la visualización de datos sea más amigable, esta vista se apoya de funciones como *formatFrequency*, que convierte la frecuencia de aplicación en una cadena legible, y humanizeDuration, que traduce la duración guardada en milisegundos en términos comprensibles.

```
function formatFrequency(minutes: number): string {
  if (minutes === 1440) {
    return 'una vez al día'
  }

  const milliseconds = minutes * 60 * 1000
  return `cada ${humanizeDuration(milliseconds, { language: 'es', units: ['w', 'd', 'h', 'm'], round: true,
conjunction: ' y ' })}`
}
```

Ilustración 39. Implementación del formato de la frecuencia

Para añadir un nuevo tratamiento, hacemos uso del componente **BottomDrawer** para obtener un menú desplegable con los inputs necesarios para añadir un nuevo tratamiento. Estos inputs son mayoritariamente **TextInput** para los datos de texto, **TimeInput** para la frecuencia de aplicación y **DateInput** para el selector de fechas.

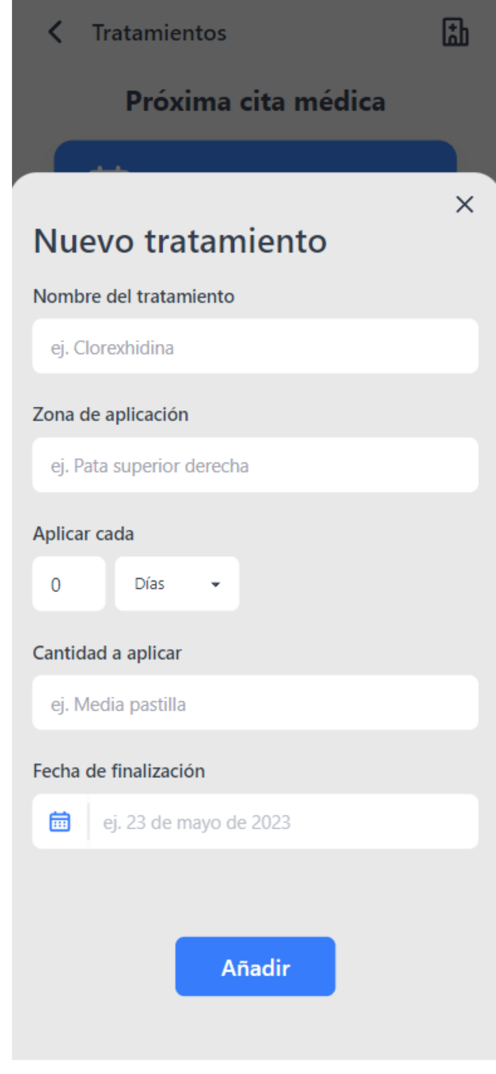


Ilustración 40. Menú de creación de nuevos tratamientos

Para eliminar un tratamiento, basta con darle al botón con icono de cruz del tratamiento que queramos eliminar de la lista de tratamientos, aceptar la confirmación de borrado que sale en un nuevo menú desplegable y observar cómo la vista se actualiza automáticamente.

Store

Las funciones necesarias serán las siguientes:

- **createTreatment(treatment):** esta función se encarga de añadir un nuevo tratamiento para un animal en la base de datos. Como parámetro, se reciben los datos creados de la forma definida en el archivo de declarations, concretamente con el tipo *EditTreatment*. Para ello, verifica si el tratamiento a crear está asociado con el animal cargado en la vista, comprobando la coincidencia del ID del animal del tratamiento con el animal cargado. Posteriormente, realiza una solicitud POST al servidor utilizando *authFetch* para enviar los detalles del tratamiento en formato JSON.

- **deleteTreatment(id):** esta función se encarga de eliminar un tratamiento específico de un animal en la base de datos. Como parámetro, recibe el ID del tratamiento a eliminar. Antes de borrar, se verifica que el tratamiento a eliminar esté asociado con el animal actualmente cargado en la aplicación. Si el ID del tratamiento no se encuentra en la lista de tratamientos del animal cargado, se lanza un error para evitar la eliminación de un tratamiento que no corresponde al animal en cuestión. Finalmente realiza una solicitud DELETE al servidor utilizando la URL correspondiente del endpoint para eliminar el tratamiento.

Modelo de datos

El modelo de datos representa un tratamiento específico asociado a un animal en la base de datos. Este modelo incluye campos para almacenar detalles clave del tratamiento. Además, se incluye el campo animal_id que sirve como la clave foránea que enlaza el tratamiento con un animal específico en la base de datos. Este campo establece una relación bidireccional con el modelo Animal, permitiendo la navegación desde un tratamiento a su animal asociado y viceversa.

```
class Treatment(SQLULIDSchema, table=True):
    name: str
    zone: str | None = Field(default=None)
    frequency: int | None = Field(default=None)
    end_date: datetime | None = Field(default=None)
    amount: str | None = Field(default=None)

    animal_id: ULID = Field(default=None, foreign_key="animal.id", sa_type=SQLAlchemyULIDType)

    animal: "Animal" = Relationship(back_populates="treatments")
```

Ilustración 41. Modelo de datos de un tratamiento

Endpoints

POST /treatment
Permite añadir un nuevo tratamiento a la base de datos

DELETE /treatment/{id}
Permite el borrado de un tratamiento concreto de la base de datos.

Esquemas

El esquema donde se detallan todos los tipos de datos de tratamiento es **CompleteTreatment**, el cual especifica todos los campos que contiene un tratamiento, así como el nombre del animal asociado a este.

```
class CompleteTreatment(ULIDSchema):
    class Animal(ULIDSchema):
        name: str

    name: str
    zone: str | None
    frequency: int | None
    end_date: datetime | None
    amount: str | None
    animal: Animal
```

Ilustración 42. Esquema de salida del módulo de tratamientos

Adicionalmente, se cuenta con el esquema **EditableTreatment** utilizado para la edición de tratamientos, el cual detalla los campos que se pueden incluir a la hora de editar o crear un animal.

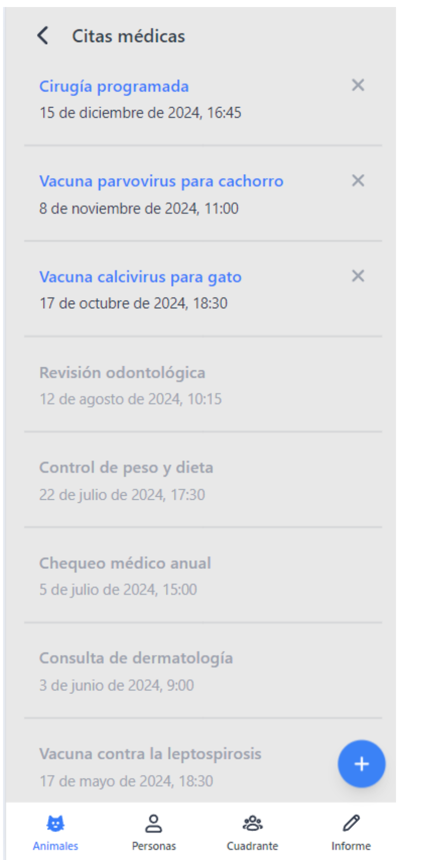
5.3.3 Animales - Citas médicas

Este módulo está dedicado a la gestión de citas médicas asociadas de un animal específico. Permite visualizar tanto el historial de citas pasadas como las futuras programadas para el animal. Además, permite añadir nuevas citas y eliminar las existentes.

Interfaz de usuario

El histórico de citas es mostrado en un formato de lista, indicando el motivo de la cita y fecha y hora de la consulta. Para diferenciar citas pasadas de futuras, las programadas están destacadas frente a las pasadas, que se muestran con un color menos intenso. El borrado de

las citas solo está permitido en las programadas, ya que el animal debe contar con el historial completo de citas a las que ha asistido sin modificaciones.
Para añadir una nueva cita médica, se cuenta con el menú desplegable de creación que hemos visto ya anteriormente, pudiendo introducir el motivo y fecha de programación de la futura cita médica.



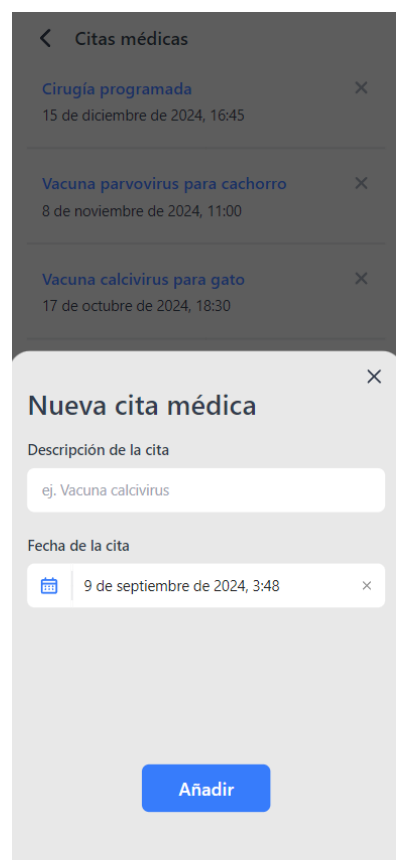


Ilustración 43. VIsta del histórico y creador de citas médicas

Para implementar esto, se ha contado con el componente de listado **ItemList,** el cual lista los datos de las citas del animal identificado. Esta lista viene previamente ordenada por su fecha en orden descendente.

Para cambiar el estilo de forma dinámica dependiendo de si la cita es pasada o futura, se configura en Vue en la propia etiqueta HTML, introduciendo ambas posibilidades:

```
<div class="flex flex-col">
  <p
    :class="[
      'mb-1 text-lg font-semibold',
      item.isPast ? 'text-gray-400' : 'text-secondary'
    ]"
  >
    {{ item.description }}
  </p>
  <p :class="['text-base', item.isPast ? 'text-gray-400' : 'text-gray-700']">
    {{ format(item.date, { date: 'long', time: 'short' }) }}
  </p>
</div>
```

Para añadir la cita médica, se introducen los componentes **TextInput** y **EditInput** que facilitan introducir el campo de texto con el motivo de la cita y la selección de fechas que se programa. De igual forma que para tratamientos, para el borrado, basta con darle a la cruz de la cita médica deseada y aceptar la notificación de borrado que sale en el desplegable.

Store

Las funciones necesarias serán las siguientes:

- **createAppointment(appointment):** Esta función se encarga de añadir una nueva cita médica para un animal en la base de datos. Como parámetro, se reciben los datos creados de la forma definida en el archivo de *declarations.ts*, concretamente con el tipo *EditAppointment*. Para ello, verifica si la cita médica a crear está asociada con el animal cargado en la vista, comprobando la coincidencia del ID del animal de la cita con el animal cargado. Posteriormente, realiza una solicitud POST al servidor utilizando *authFetch* para enviar los detalles en formato JSON.

- **deleteAppointment(id):** Esta función se encarga de eliminar una cita médica específica de un animal en la base de datos. Como parámetro, recibe el ID la cita a eliminar. Antes de borrar, se verifica que la cita a eliminar esté asociada con el animal actualmente cargado en la aplicación. Si el ID no se encuentra en la lista de citas médicas del animal cargado, se lanza un error. Finalmente realiza una solicitud DELETE al servidor utilizando la URL correspondiente del endpoint para eliminarla.

Modelo de datos

El modelo de datos representa una cita médica específica asociado a un animal en la base de datos. Este modelo incluye campos para almacenar tanto la fecha de la cita como su

descripción. Adicionalmente cuenta con una propiedad computada que no se almacena directamente en la base de datos, sino que se calcula dinámicamente en función del valor de date. Esta propiedad devolverá True si la fecha de la cita es anterior a la fecha y hora actuales, indicando que la cita es en el pasado.
Además, se incluye el campo animal_id que sirve como la clave foránea que enlaza el tratamiento con un animal específico en la base de datos. Este campo establece una relación bidireccional con el modelo Animal, permitiendo la navegación desde una cita médica a su animal asociado y viceversa.

```
class Appointment(SQLULIDSchema, table=True):
    date: datetime
    description: str

    @computed_field
    @property
    def is_past(self) -> bool:
        return self.date < datetime.now()

    animal_id: ULID = Field(default=None, foreign_key="animal.id", sa_type=SQLAlchemyULIDType)

    animal: "Animal" = Relationship(back_populates="appointments")
```

Ilustración 44. Modelo de datos de una cita médica

Endpoints

POST /appointment
Permite añadir una uneva cita médica a la base de datos

DELETE /appointment/{id}
Permite el borrado de una cita médica concreta de la base de datos.

Esquemas

El esquema que define todos los tipos de datos de una cita médica es **CompleteAppointment**. Este esquema detalla todos los campos de una cita, incluyendo la propiedad calculada que indica si la cita es pasada o no, así como el nombre del animal asociado a ella.

```
class CompleteAppointment(ULIDSchema):
    class Animal(ULIDSchema):
        name: str

    date: datetime
    description: str
    is_past: bool
    animal: Animal
```

Ilustración 45. Esquema de salida del módulo de citas médicas

Adicionalmente se define el esquema **EditableAppointment,** el cual solo incluye los datos de la fecha, descripción e id del animal asociado, ya que con esta información nos basta para crear una cita médica.

5.3.4 Animales - Cuadrante de animales

El cuadrante de animales no es un submódulo en sí mismo, si no un documento PDF que debe contener los datos más importantes de cada animal que puedan ser interesantes conocer durante el turno, esto es: su nombre, imagen (para identificarlos), su sexo y sus tratamientos.

La razón de ser de este documento es que actualmente algunos voluntarios crean y actualizan este cuadrante manualmente para luego poder imprimirlo y llevarlo al turno con sus propias anotaciones. Lo ideal sería que desde el módulo de animales estos voluntarios puedan generar y descargar automáticamente este cuadrante.

Además, con este documento conseguimos dar una vista de los patios y los tratamientos que se necesitan en cada uno de ellos, lo cual es importante para simplificar las tareas durante el turno.

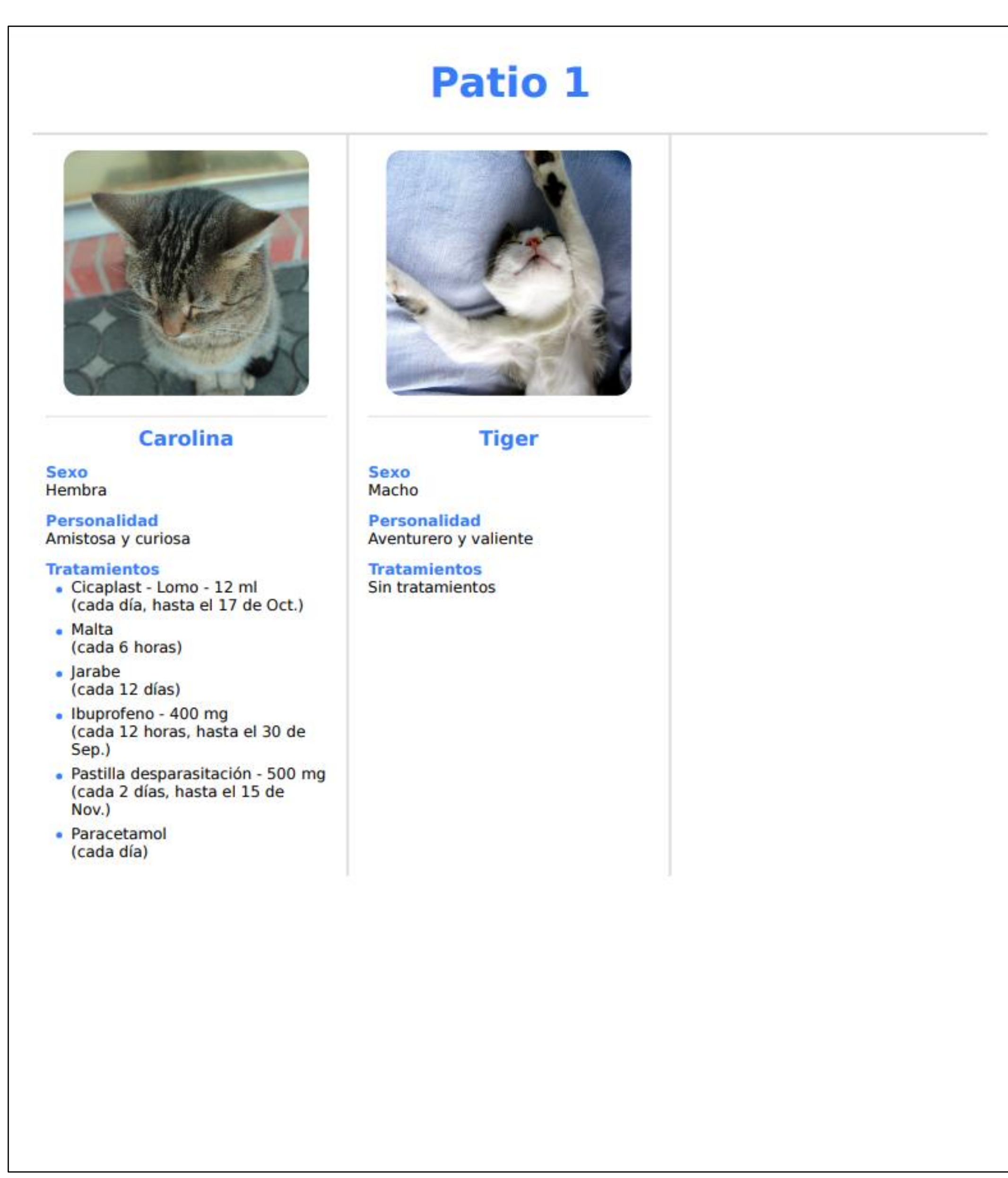


Ilustración 46. Ejemplo de la vista de un patio en el cuadrante de animales

Para esto contamos con dos partes, primero el módulo *animal_report.py*, el cual contiene la función *create_report* que se encarga de la generación del documento. Esta función realiza las siguientes tareas:

1. Carga la plantilla HTML lista para renderizar los datos de los animales de un patio (uno solo) en formato rejilla.

2. Procesa los datos de los animales para su correcta visualización. Por ejemplo, cambia los valores de sexo del animal (male y female) por sus traducciones al castellano (Macho y Hembra) o modifica el dato de la frecuencia de los tratamientos para mostrar una representación más sencilla (cada X horas, cada día, ...).

3. Agrupa los animales por patio.

4. Procesa cada patio renderizando el HTML de sus páginas con los datos completos, para posteriormente convertir estas páginas a PDF y añadirlas a una variable común.

5. Combina todas las páginas de la variable común en un mismo PDF y guárdalo en la ruta que inicialmente se le pasó por parámetro a la función.

Para la plantilla, lo que hemos hecho es crear un fichero HTML en formato Jinja2, que recibe como parámetros el nombre del patio que se quiere mostrar y todos los animales que pertenecen a este. Para estilarlo hemos utilizado CSS simple (sin Tailwind, ya que este HTML se renderiza en el servidor) para que esté acorde al estilo general de la aplicación.

Por otro lado, tenemos el endpoint "*/report*" dentro del módulo de animales. Este se encarga de obtener todos los animales de la base de datos, ejecutar la función descrita anteriormente, mandar el PDF como respuesta al cliente y eliminar el documento generado una vez el cliente ha recibido su PDF (no necesitamos almacenarlo ya que en cada petición el cuadrante debe estar lo más actualizado posible).

```
...

tempdir = tempfile.TemporaryDirectory(dir=".")
report_path = f"{tempdir.name}/report.pdf"
create_report(animals, report_path)

return FileResponse(
    report_path,
    media_type="application/pdf",
    filename="animal_report.pdf",
    background=BackgroundTask(tempdir.cleanup),
)
```

Ilustración 47. Implementación del guardado y envío del cuadrante en el endpoint "/report"

Para conseguir esta última parte, utilizamos un directorio temporal donde se almacena el documento que después es retornado en el endpoint como un FileResponse. Esta respuesta incluye una tarea en segundo plano para eliminar el directorio temporal.

Las tareas en segundo plano de FastAPI se ejecutan justo después de mandar la respuesta al cliente, de forma que este no tiene que esperar a su ejecución para recibir los datos.

5.3.5 Adopciones

El módulo de adopciones tiene como propósito principal registrar todas las adopciones realizadas en la protectora, vinculando a cada animal adoptado con una persona adoptante registrada en el sistema. Este módulo debe incluir tanto las adopciones temporales, también llamadas acogidas, como las adopciones permanentes.
Otro requisito importante es mantener un seguimiento actualizado de cada adopción, ya que una de las tareas esenciales del voluntariado es asegurar un contacto continuo con el adoptante para evaluar cómo se adapta el animal a su nuevo hogar. Para ello, se debe llevar un registro fechado de todas las observaciones que los voluntarios realicen para cada adopción que se registre.

Interfaz de usuario - Listado de adopciones

Esta vista permite mostrar en forma de listado todas las adopciones registradas en la protectora, de forma que se pueda ver de manera rápida el nombre y fotografía del animal, nombre de la persona adoptante y la fecha en la que se realizó la adopción, ya que esos serían los datos básicos que identifican a una adopción concreta, junto con la identificación del tipo de adopción que se realizó.

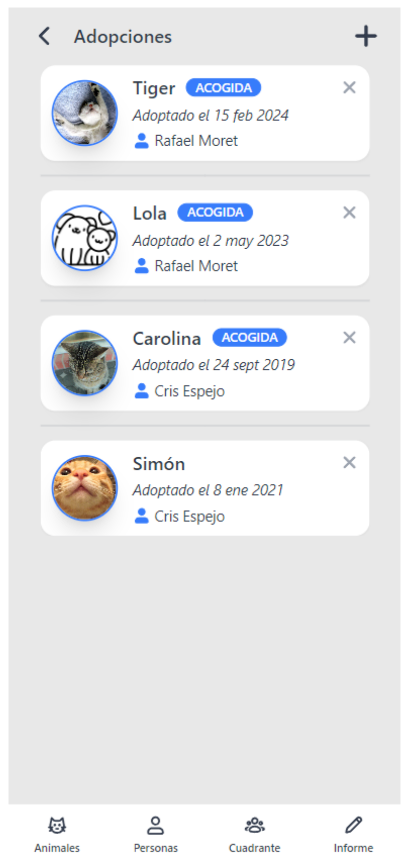


Ilustración 48. Vista del listado de adopciones

El componente principal donde se desarrolla la interfaz gestiona principalmente la obtención y consumo de datos necesarios para listar los campos de todas las adopciones y la lógica para agregar nuevas. La visualización de la lista de adopciones se realiza a través del componente **AdoptionList**, el cual utiliza el componente común **ItemList** para mostrar cada adopción obtenida. Este último recorre todas las adopciones y utiliza el componente **AdoptionCard** para la visualización individual de cada una, pasando como propiedades los datos de la adopción en el formato definido por **AdoptionInfo**, lo que garantiza que se muestren los detalles necesarios en cada tarjeta.

```
<ItemList
  :items="props.adoptionList"
  delete-title="¿Estás seguro de que quieres borrar esta adopción?"
  class="mx-5"
  @delete="(adoption) => handleDeleteAdoption(adoption.id)"
>
  <template #empty>
    <div class="mt-6 flex flex-col items-center">
      <span class="i-mingcute-heart-crack-fill text-6xl" />
      <p class="text-gray-500">No hay adopciones</p>
    </div>
  </template>
  <template #item="{ item, openConfirm }">
    <AdoptionCard :adoption="item" @click="navigateAdoption(item.id)">
      <template #action>
        <button class="my-1 flex flex-col" @click.stop="openConfirm(item)">
          <span class="i-mingcute-close-fill text-xl text-gray-400" />
        </button>
      </template>
    </AdoptionCard>
  </template>
  <template #delete="{ item }">
    <p>
      <span class="font-semibold">{{ item.animal.name }}</span>
      <span>
        {{ `, el día ${format(item.registerDate, { date: 'medium' })} ` }}
      </span>
      <span>por {{ item.person.name }} {{ item.person.firstSurname }}</span>
    </p>
  </template>
</ItemList>
```

Ilustración 49. Implementación del listado de adopciones utilizando el componente ItemList

Para la interfaz de creación de adopciones, se utilizan los componentes ya mencionados **AnimalSelector** y **PersonSelector**, de manera que sea fácil la selección y búsqueda tanto del animal adoptado y la persona adoptante. La selección de la fecha de registro se maneja con el componente **DateInput**, que proporciona un campo de entrada de fecha intuitivo para el

usuario, mientras que el componente **DropdownSelector** permite elegir el tipo de adopción, ya sea permanente o temporal (casa de acogida).

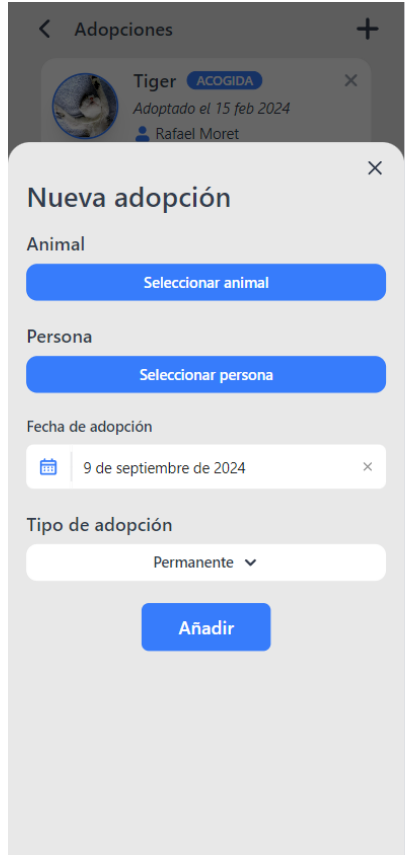


Ilustración 50. VIsta de creación de adopciones

Además, se utiliza el componente **BottomDrawer** como contenedor para mostrar el formulario de manera modal, ofreciendo una experiencia de usuario fluida y focalizada. Este diseño permite gestionar todos los datos necesarios de la adopción en un solo lugar, asegurando que se ingresen correctamente antes de enviarlos al backend para su registro en el sistema.

Interfaz de usuario - Adopción

Al seleccionar una adopción de la lista, se debe mostrar información detallada de la misma, incluyendo características específicas del animal adoptado, como su edad, una descripción básica de su comportamiento, su estado de castración y su compatibilidad. También se deben presentar los datos principales de contacto del adoptante para facilitar una comunicación rápida. Además, es importante disponer de un listado de registros de seguimiento de la adopción para visualizar la evolución del animal con su nuevo adoptante.

Ilustración 51. Vista concreta de una adopción

Para la visualización de datos concretos del animal y la persona, se gestiona sobre la misma vista **AdoptionView** donde también se gestionan la obtención y guardado de datos. Esto se desarrolla mediante etiquetas y componentes básicos de html. Además, dependiendo de cada característica booleana como si está castrado o si es compatible, se elige de manera dinámica un icono acorde con su valor.

Para listar todos los registros de seguimiento, utilizamos el componente de listado **ItemList**, el cual recorre los datos de seguimiento de cada animal y muestra su fecha y descripción.

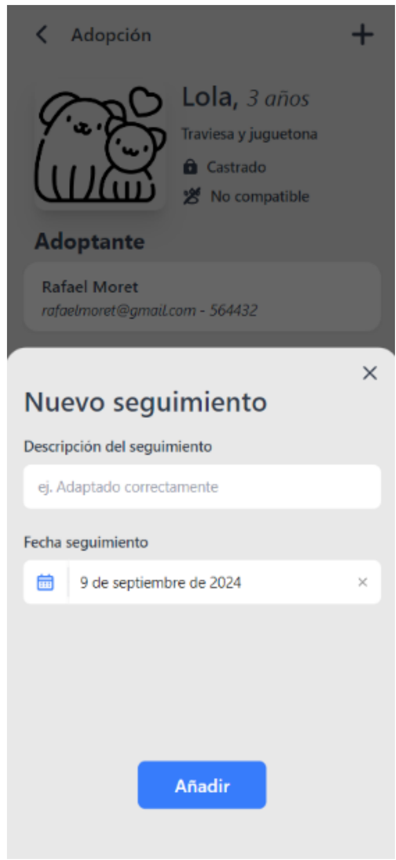


Ilustración 52. Vista del creador de nuevos seguimientos

Para crear un nuevo seguimiento, hacemos uso nuevamente de **BottomDrawer** para una visualización desplegable donde podemos introducir los datos de un registro unevo, haciendo uso de **TextInput** para la descripción del registro y **DateInput** para la fecha registrada.

El borrado de cada uno de los registros se hace clicando en el botón de la cruz del elemento de *ItemList*, aceptando la confirmación igual que en el resto de las interfaces creadas con dicho componente.

Store

Las funciones necesarias para desarrollar estos requisitos son:

- **fetchAdoptions():** se encarga de recuperar y procesar la lista de adopciones desde la API de la aplicación. Para ello, realiza una petición a la API utilizando *authFetch*, obteniendo los datos en formato JSON. Estos datos son transformados mediante el adaptador *AdoptionInfoAdapter* para adaptarlos al formato necesario y, posteriormente, se actualiza el estado reactivo *adoptionList* con la lista de adopciones obtenida. Después de la actualización, la función filtra la lista en función del parámetro opcional *foster*, mostrando solo las adopciones que coincidan con este criterio, si está definido.

- **fetchAdoption(id)**: se encarga de obtener los detalles de una adopción específica a partir de su identificador único. Utilizando *authFetch*, realiza una solicitud GET a la API para recuperar los datos de la adopción en formato JSON. Estos datos se procesan mediante el adaptador AdoptionAdapter para ajustarlos al formato requerido. Una vez transformada la información, actualiza el estado reactivo *adoptionDetails* con los datos de la adopción obtenida.

- **createAdoption(adoption):** se encarga de añadir una nueva adopción a la base de datos. Toma como parámetro un objeto *adoption* del tipo *EditAdoption*, que contiene los detalles necesarios para crear la adopción. La función realiza una solicitud POST a la API utilizando *authFetch*, enviando los datos en formato JSON con los encabezados adecuados. Tras la creación de manera satisfactoria de la adopción, la función *fetchAdoptions* es llamada para actualizar la lista de adopciones en la aplicación, asegurando que la nueva adopción aparezca en la interfaz de usuario.

- **deleteAdoption(id):** esta función se encarga de eliminar una adopción específica en la base de datos. Como parámetro, recibe el ID la adopción. Antes de borrar, verifica si la adopción a eliminar existe en la lista actual de adopciones *adoptionList* para lanzar un error si no lo está. Si la adopción está presente, realiza una solicitud DELETE a la API utilizando *authFetch* para eliminar el registro correspondiente en el servidor. Finalmente, actualiza el estado reactivo *adoptionList* al filtrar y descartar la adopción eliminada, asegurando que la lista refleje correctamente los cambios realizados.

- **createMonitoring(monitoring):** se encarga de añadir un nuevo registro de seguimiento para una adopción específica. Primero, verifica que los detalles de la adopción actual estén cargados y que el ID de la adopción del seguimiento coincida con el de la adopción actualmente cargada. Posteriormente, realiza una solicitud POST a la API utilizando *authFetch*, enviando los datos del seguimiento en formato JSON y con los encabezados adecuados. Tras la creación correcta del seguimiento, la función *fetchAdoption* se llama para actualizar la información de la adopción en la aplicación, lo que permite que el nuevo seguimiento se refleje en la interfaz de usuario.

- **deleteMonitoring(id):** esta función está diseñada para eliminar un registro de seguimiento específico. Inicialmente, verifica que los detalles de la adopción actual estén disponibles y que el seguimiento que se intenta eliminar pertenezca a la adopción actualmente cargada. Luego, realiza una solicitud DELETE a la API usando *authFetch*, especificando el ID del seguimiento que se desea eliminar. Si la solicitud es exitosa, la función actualiza la lista de seguimientos en el estado reactivo *adoptionDetails* para reflejar la eliminación, eliminando el seguimiento correspondiente de la lista.

Modelo de datos

Para el modelo de datos de una adopción, se necesita incluir los campos de la fecha de registro, si es una adopción temporal, y una fecha opcional para la revocación.

Una adopción está relacionada con personas y animales mediante identificadores únicos que actúan como claves foráneas. Además, establece relaciones bidireccionales con las entidades Persona y Animal, incluyendo además una relación con el componente Monitoring para permitir un seguimiento detallado de cada adopción.

```python
class Adoption(SQLULIDSchema, table=True):
    register_date: date
    foster: bool
    revocation_date: date | None = Field(default=None)

    person_id: ULID = Field(
        default=None,
        foreign_key="person.id",
        primary_key=False,
        sa_type=SQLAlchemyULIDType,
    )
    animal_id: ULID = Field(
        default=None,
        foreign_key="animal.id",
        primary_key=False,
        sa_type=SQLAlchemyULIDType,
    )

    person: "Person" = Relationship(back_populates="adoptions")
    animal: "Animal" = Relationship(back_populates="adopters")
    monitorings: list["Monitoring"] = Relationship(back_populates="adoption")
```

Ilustración 53. Modelo de datos de una adopción

Endpoints

GET /adoption/search
Permite obtener el listado de todas las adopciones de la base de datos

GET /adoption/{id}
Retorna los datos de una adopción concreta

DELETE /adoption/{id}
Permite el borrado de una adopción concreta

POST /adoption
Permite crear una adopción nueva en la base de datos

POST /monitoring
Permite crear un seguimiento nuevo en la base de datos

DELETE /monitoring/{id}
Permite borrar de la base de datos un registro de seguimiento específico

Esquemas

El esquema que define todos los tipos de datos de una adopción es **CompleteAdoption,** el cual detalla la información clave del proceso de adopción, como la fecha de registro, si se trata de un hogar temporal o la fecha de revocación si aplica. Adicionalmente, contiene las tres subclases principales que conlleva una adopción: los datos del animal adoptado, la información de la persona adoptante y los registros de seguimiento asociados.

```python
class CompleteAdoption(ULIDSchema):
    class Animal(ULIDSchema):
        name: str
        image: str | None = None
        description: str | None = None
        personality: str | None = None
        birth_date: date | None = None
        is_animal_compatible: bool | None = None
        is_castrated: bool | None = None

    class Person(ULIDSchema):
        name: str
        first_surname: str
        phone: str
        second_surname: str | None = None
        email: str | None = None

    class Monitoring(ULIDSchema):
        follow_date: date | datetime
        note: str

    register_date: date | datetime
    foster: bool
    revocation_date: date | None = None
    animal: Animal
    person: Person
    monitorings: list[Monitoring]
```

Ilustración 54. Esquema de salida principal del módulo de adopciones

Junto a este esquema, se definen dos variantes adicionales:

- **ListedAdoption:** Proporciona únicamente la información necesaria para visualizar un listado de seguimientos, omitiendo detalles específicos como los registros completos y la información más detallada del animal o de la persona.

- **EditableAdoption**: Incluye únicamente los identificadores únicos de la persona y el animal, el tipo de adopción y la fecha de registro, ya que estos son los únicos datos necesarios para crear una nueva adopción.

5.3.6 Patios

Este módulo se centra en la visualización de todos los espacios donde están distribuidos los animales, llamados patios. Es necesario mostrar un listado de todos ellos mostrándose en el orden en los que se visitan durante un turno. Esto permite gestionar de manera eficiente las tareas de cuidado y mantenimiento en la protectora, asegurando un recorrido ordenado que minimiza el riesgo de contagio de enfermedades entre los diferentes patios. Este orden debe poder ser editable por el usuario.

Adicionalmente, la posibilidad de crear y eliminar patios ya existentes debe estar asegurada.

Interfaz de usuario

La visualización de los patios cuenta con una vista simple donde se visualiza el listado de patios en orden, visualizando los botones de eliminar cada uno de ellos, crear un nuevo patio con un desplegable donde se pueda introducir el nombre del patio a introducir y la posibilidad de ordenación.

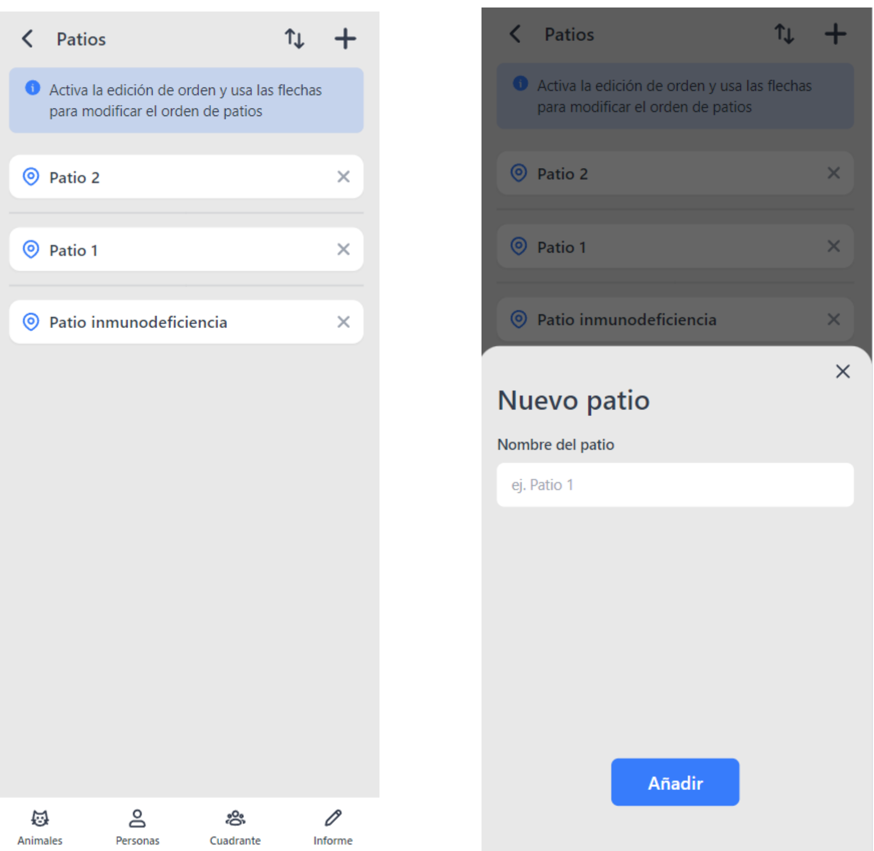


Ilustración 55. Vista del listado de patios y su menú de creación

La ordenación se gestiona mediante un sistema de flechas que permite cambiar la posición de cada patio en la lista, restringiendo los movimientos opuestos en los extremos y permitiendo subir o bajar la posición de los patios intermedios.

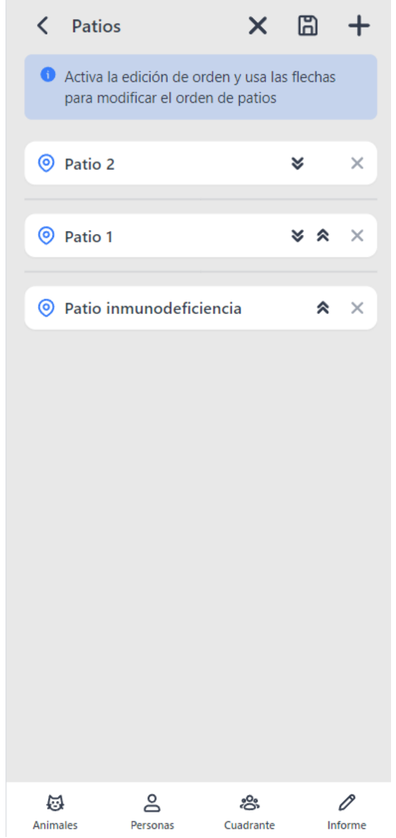


Ilustración 56. Vista del listado de patios con el modo de reordenación activado

Para el desarrollo de estas interfaces, se ha creado un componente llamado **YardList** que gestiona y visualiza una lista de patios en la aplicación. Este componente utiliza otros como **ItemList**, que presenta la información en forma de listado, y **YardCard**, que muestra una tarjeta con los detalles del patio y acciones específicas.

Este último permite reorganizar los patios en la lista mediante un sistema de flechas, visibles únicamente en el modo de edición. Estas flechas están configuradas para garantizar que los patios solo puedan moverse dentro de los límites de la lista, sin superar los extremos. Adicionalmente permite mostrar si la opción de eliminación de patio está activa o no y gestiona que aparezca un mensaje de "No hay patios" si no se encuentra ninguno en la base de datos.

Para la creación de patios, se ha vuelto a hacer uso del componente **BottomDrawer** para el modal necesario para introducir la nueva información y un **TextInput**, ya que introducir el nombre del nuevo patio es lo único necesario. A la hora de crear, se actualizará automáticamente el orden de patios introduciendo este nuevo en último lugar, haciendo una

petición a la API tanto de guardar este patio en la base de datos como añadir un nuevo orden de patios.

Store

En este módulo, es fundamental almacenar tanto los patios registrados en la base de datos como el orden en que aparecen. Este orden debe guardarse con un identificador único, ya que está vinculado al informe que utilizan los voluntarios durante sus turnos. Es importante mantener un historial de los cambios realizados en el orden de los patios para reflejar adecuadamente cualquier alteración en el recorrido a lo largo del tiempo.
Para ello, se han definido las siguientes funciones:

- **fetchYards():** se encarga de obtener una lista de patios desde la API. Para ello, utiliza *authFetch* para enviar una solicitud GET a la API, especificando la URL construida con el identificador del endpoint de patios. Una vez que recibe la respuesta, la convierte a formato JSON y la procesa utilizando el adaptador *YardInfoAdapter* para transformar los datos según el formato requerido. El resultado, que es una lista de objetos *YardInfo*, se asigna al estado *yardList* .

- **fetchYard(id):** se encarga de obtener los detalles de un patio específico desde la API. Para ello se realiza una petición GET a la API usando *authFetch*, con la URL construida a partir del endpoint para patios y el identificador del patio requerido. Una vez recibida la respuesta, la convierte a formato JSON y la procesa mediante el adaptador *YardAdapter*, que transforma los datos en el formato adecuado. Los detalles del patio adaptados se asignan al estado *yardDetails*.

- **createYard(yard):** se encarga de añadir un nuevo patio a la base de datos. Inicia enviando una solicitud POST a la API mediante *authFetch*, utilizando la URL correspondiente y el cuerpo de la solicitud en formato JSON, que contiene los datos del patio a crear. Si la respuesta del servidor indica un error se lanza una excepción con un mensaje de error que incluye el detalle del error obtenido de la respuesta. Tras completar la solicitud, se actualiza la lista de patios llamando a la función *fetchYards* para reflejar los cambios en el estado de la aplicación.

- **deleteYard(id):** se encarga de eliminar un patio específico de la base de datos. Primero, verifica si el patio que se intenta eliminar está en la lista de patios cargados en la aplicación (*yardList*). Si el patio no está en la lista o no corresponde con el patio que se está visualizando actualmente, se lanza una excepción con un mensaje de error. Si el patio es válido, se realiza una solicitud DELETE a la API mediante *authFetch*, utilizando la URL correspondiente para el patio específico que se desea eliminar. Una vez completada la solicitud, se actualiza la lista de patios en la aplicación, eliminando

el patio que ha sido borrado, para asegurar que la lista refleje correctamente los datos actuales.

- **createOrder(order):** La función se encarga de crear una nueva orden de patio en la base de datos. Utiliza la función *authFetch* para enviar una solicitud POST al servidor, especificando la URL para la API de órdenes de patio. La solicitud incluye los datos de la nueva orden (*yardOrder*), que se envían en formato JSON. Si la respuesta del servidor indica un error, se lanza una excepción con un mensaje de error que indica que se produjo un problema inesperado al intentar crear el nuevo orden.

- **fetchCurrentOrder():** se encarga de recuperar la información de la orden de patio actual desde la API. Para ello, utiliza *authFetch* para enviar una solicitud GET a la URL correspondiente para obtener la orden actual. En caso satisfacotrio, se procesa la respuesta JSON, que se adapta mediante el *YardOrderAdapter*, y se actualiza el estado reactivo *currentOrder* con la información de la orden. Si ocurre algún error durante el proceso, se asegura que *currentOrder* se establezca como una lista vacía.

Modelo de datos

Para el modelo de datos, debemos distinguir entre el modelo que únicamente guarda los patios existentes en la protectora, y otro que guarda el orden. Para el primero, se ha definido un modelo de datos simple con la siguiente estructura:

```
class Yard(SQLULIDSchema, table=True):
    name: str

    animals: list["Animal"] = Relationship(back_populates="yard")
```

Ilustración 57. Modelo de datos de un patio

Este guarda su nombre y establece una relación con los animales que contiene ese patio.
Para el orden de patios, se establece el siguiente modelo:

```
class YardOrder(NoSQLULIDSchema):
    class Yard(ULIDSchema):
        name: str

    yard_order: list[Yard]
    date: datetime
```

Ilustración 58. Modelo de datos de un orden de patios

El modelo **YardOrder** está diseñado para gestionar el orden de los patios, manteniendo un registro detallado de cómo están organizados y cuándo se realizó esta organización. Incluye una lista de patios (yard_*order*) y una marca de tiempo (*date*) que indica cuándo se registró ese orden. La subclase *Yard* define la estructura de un patio, que incluye su nombre.

Endpoints

GET /yard/search
Permite el listado de todos los patios guardados en la base de datos

POST /yard
Permite crear y guardar un nuevo patio en la base de datos

GET /yard/{id}
Permite obtener los datos de un patio concreto

PUT /yard/{id}
Permite editar los datos de un patio concreto

DELETE /yard/{id}
Permite eliminar un patio de la base de datos

GET /yard_order/search
Permite listar todos los órdenes de patios que hay guardados en la base de datos.

POST /yard_order/
Permite crear y guardar un nuevo orden de patios

GET /yard_order/current
Permite obtener el orden de patios que está activo al momento de hacer la petición

Esquemas

Los esquemas de un patio se definen de la siguiente forma:

```
class EditableYard(BaseSchema):
    name: str


class ListedYard(ULIDSchema):
    name: str


class CompleteYard(ListedYard):
    class Animal(ULIDSchema):
        name: str

    animals: list[Animal]
```

Ilustración 59. Esquemas referentes a un patio

Esto permite extraer la información concreta según si se requiere toda la información de un patio, incluyendo el listado de animales que lo contienen, o solo su nombre a la hora de listar o crearlo.

Para el orden de patios: se establecen los dos siguientes:

```
class ListedYardOrder(ULIDSchema):
    date: datetime
    yard_order: list[ListedYard]


class EditableYardOrder(BaseSchema):
    date: datetime
    yard_order: list[ULID]
```

Ilustración 60. Esquemas referentes a un orden de patios

ListedYardOrder almacena detalles completos de cada patio junto con la fecha del registro, facilitando un seguimiento detallado. Por otro lado, **EditableYardOrder** solo guarda identificadores únicos de los patios y la fecha, proporcionando una forma más simplificada obteniendo solo los datos necesarios para crear un nuevo orden.

6

Conclusiones

La creación de la aplicación para la Protectora de Málaga ha sido una experiencia verdaderamente desafiante, a la par que enriquecedora. Desde el principio tuvimos claro que nuestro objetivo principal no era solo conseguir una aplicación funcional que simplificara gran parte de las tareas que nosotros, como voluntarios de esta, hemos tenido que realizar múltiples veces, sino que además pudiera ser lo suficientemente sencilla de usar como para que perfiles más o menos acostumbrados a lidiar con tecnología pudieran aprovechar sus ventajas.

Es por esta razón que, uno de los aspectos más gratificantes ha sido el proceso de diseño de la interfaz de usuario. El requisito de simplicidad y claridad de la interfaz nos ha exigido un esfuerzo extra para garantizar que esta fuera lo más accesible que podíamos conseguir, intentando mantener una curva de aprendizaje mínima. Esto no solo ha hecho que estemos muy orgullosos de tener una aplicación vistosa a la par que útil, si no que me ha ayudado personalmente a mejorar mi capacidad para diseñar interfaces con el objetivo puesto en el usuario.

Además, el proceso de desarrollo me ha permitido también mejorar mis habilidades de trabajo conjunto, ya que he tenido que trabajar codo a codo con mi compañera para tomar decisiones que no solo dependían de mi punto de vista, sino de escuchar diferentes planteamientos para optar de manera sincera por el que más beneficiara a la experiencia de usuario final. Por otro lado, ambos hemos tenido que poner especial atención en priorizar tareas de manera eficiente, ya que, con un proyecto tan amplio e interesante, era fácil comenzar a añadir funcionalidades o desarrollos adicionales que complicaban las limitaciones de tiempo a las que estamos obligados a cumplir.

En resumen, el proyecto ha concluido con una aplicación funcional de la que ambos estamos muy orgullosos y esperamos que pueda resultar en una mejora de la calidad en la gestión de la Protectora, pero además con un gran aprendizaje a nivel profesional, tanto en lo técnico como en lo personal, ya que me ha ayudado a crecer en materias que quería y en otras que desconocía.

6.1 Líneas futuras

A pesar de haber logrado desarrollar un software capaz de suplir los problemas previos que tenía la protectora y de haber obtenido una solución funcional y satisfactoria, durante el tiempo de desarrollo hemos detectado que existe un margen de mejora donde se pueden aplicar optimizaciones y funcionalidades adicionales. Estas son algunas de las funcionalidades que pueden ser desarrolladas, brindando beneficios tanto a usuarios como administradores de la protectora:

1. **Optimización a interfaz en vista por ordenador:** Pese a que lo primordial era adaptarlo a una vista móvil, algunos usuarios utilizan la aplicación desde ordenador, mayoritariamente los coordinadores. Por ello, sería conveniente mejorar la experiencia de uso en este tipo de dispositivos, asegurando diseño adaptativo que aproveche mejor el espacio disponible en pantallas grandes.

2. **Aplicación instalable:** Aunque actualmente la aplicación se ejecuta en el navegador, sería beneficioso desarrollar una versión instalable para dispositivos móviles y ordenadores. Esto permitiría a los usuarios acceder a la aplicación de manera más rápida y sin depender de una conexión constante a Internet, además de ofrecer funcionalidades adicionales como notificaciones en tiempo real y un rendimiento más optimizado.

3. **Posibilidad de cambio de contraseña:** Actualmente, el cambio de contraseña no está disponible para el usuario, si no que tiene que contactar con el administrador. Hacerlo de manera autónoma permitiría agilizar el proceso y evitar latencia en la posibilidad de usar la aplicación.

4. **Compresión de las imágenes subidas:** Para mejorar el rendimiento y reducir el tiempo de carga, sería recomendable implementar un sistema de compresión automática de las imágenes subidas a la aplicación. Esto no solo optimizaría el uso del almacenamiento y del ancho de banda, sino que también reduciría significativamente el espacio necesario para guardar archivos en el servidor, garantizando así una experiencia de usuario más ágil y eficiente, especialmente en conexiones de internet más lentas.

5. **Autenticación en dos pasos:** Para aumentar la seguridad de la aplicación, se podría implementar un sistema de autenticación en dos pasos. Este método requeriría que los usuarios, además de ingresar su contraseña, verifiquen su identidad a través de un segundo factor, como un código enviado a su teléfono móvil o correo electrónico.

6. **Posibilidad de asignar seguimientos a voluntarios:** Actualmente, las notas de seguimiento de las adopciones pueden ser realizadas por cualquier voluntario. Sería útil implementar un sistema de asignación que permita designar a un voluntario específico para cada adopción, asegurando un seguimiento personalizado. Además, se podría añadir una funcionalidad que permita a cada voluntario visualizar rápidamente todos los seguimientos que tiene asignados, facilitando así la organización de sus tareas y mejorando la eficiencia en la gestión del proceso de adopción.

7. **Despliegue:** Este proyecto acaba de completar la fase final de desarrollo, por lo que actualmente toda la aplicación se ejecuta en un entorno local. Para su funcionamiento en un entorno de producción, el proyecto debería desplegarse en un servidor adecuado. Esto incluye configurar el servidor para alojar tanto el frontend como el backend, asegurarse de que la base de datos esté correctamente integrada y realizar las pruebas necesarias para garantizar que el despliegue se realice sin inconvenientes.

Referencias

[1] Julia Martins, Scrum: conceptos clave y cómo se aplica en la gestión de proyectos, https://asana.com/es/resources/what-is-scrum

[2] Python, https://www.python.org/

[3] FastAPI, Documentation, https://fastapi.tiangolo.com/

[4] ¿Qué es MongoDB? https://www.ibm.com/es-es/topics/mongodb

[5] About SQLite, https://www.sqlite.org/about.html

[6] José Luis Chacón, TypeScript: qué es, diferencias con JavaScript y por qué aprenderlo, https://profile.es/blog/que-es-typescript-vs-javascript/

[7] ¿Qué es Vite?, https://www.paradigmadigital.com/dev/como-crear-aplicacion-react-vite

[8] ¿Qué es Vue?, https://vue3-spanish.netlify.app/guide/introduction.html

[9] Qué es Tailwind CSS, https://openwebinars.net/blog/que-es-tailwind-css-y-por-que-deberias-usarlo/

[10] Beanie overview, https://beanie-odm.dev/

[11] SQLModel, https://sqlmodel.tiangolo.com/

[12] Vue.js guide, https://router.vuejs.org/guide/

[13] Franco Brutti, Redux: el patrón de arquitectura de datos, https://thepower.education/blog/redux-el-patron-de-arquitectura-de-datos

[14] Understanding client-side JavaScript frameworks, https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks

Apéndice A

Manual de Instalación

Este manual te guiará en el proceso de instalación de la aplicación ProteApp. La plataforma se compone de dos partes principales: el frontend (la interfaz de usuario) y el backend (el servidor).

Requerimientos:

Para proceder a ejecutar la aplicación, se necesita tener los siguientes componentes:

- **Node.js**: Necesario para ejecutar el código del servidor y las herramientas de desarrollo basadas en JavaScript.

- **Docker**: Utilizado para crear y gestionar contenedores que facilitan el despliegue y la configuración del entorno de la aplicación.

- **Git**: Clonado del repositorio con el código fuente.

- **Weasyprint**: Necesario para convertir documentos HTML y CSS en archivos PDF.

Instalación:

1. Previamente, clonar el repositorio donde está el código fuente: **https://github.com/r-moret/proteapp.git**

2. Instalar dependencias en backend y ejecutar:

    a. Navegar al directorio de backend: cd backend
    b. Ejecuta el comando para instalar las dependencias: poetry install
    c. Encender el daemon de docker
    d. Ejecutar con el comando: poetry run start

3. Instalar dependencias en frontend y ejecutar:
    a. Navega al directorio de frontend: cd frontend
    b. Ejecuta el comando para instalar sus dependencias: npm install
    c. Ejecuta la aplicación con el comando: npm run dev

La aplicación estará disponible en http://localhost:5174/

Apéndice B

Guía de usuario

¡Bienvenido al Manual de Usuario de ProteApp!

Este manual ha sido creado para mejorar tu experiencia con aplicación. Si eres voluntario de la Protectora de Plantas y Animales de Málaga, aquí encontrarás toda la información necesaria para aprovechar al máximo las funcionalidades que ofrece ProteApp.

Inicio de sesión

Si te acabas de dar de alta como voluntario en la Protectora, deberías haber recibido un correo al email que diste a los coordinadores. En el email, debería aparecer la contraseña que se te ha asignado para poder ingresar a la aplicación.

Ilustración 61. GDU - Visualización del mail de registro

Esta contraseña y email deberás introducirlo en el login que aparece cuando inicias la aplicación. Recuerda mantener tus credenciales seguras y no compartirlas con nadie.

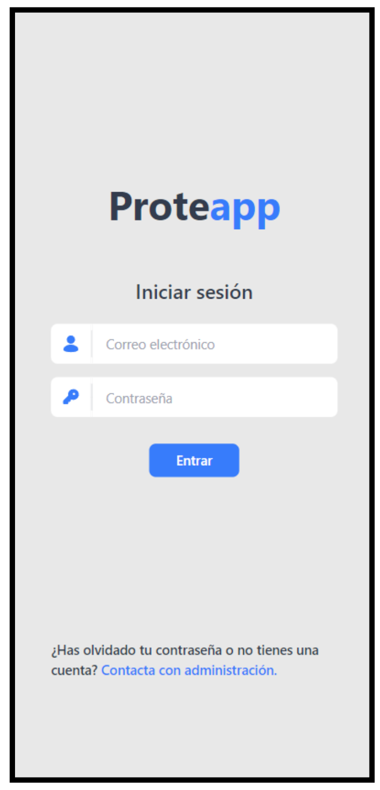


Ilustración 62. GDU - Visualización del login

Una vez las introduzcas correctamente, tendrás acceso a todas las funcionalidades de la aplicación.

Edición de perfil

Los coordinadores habrán creado un perfil básico inicialmente con los datos que les distes cuando te diste de alta como voluntario, pero estos datos los puedes modificar para tener tu perfil adaptado a tu gusto, o si tus datos personales han cambiado. Para ello, vamos a ir al apartado de editor del perfil haciendo clic en tu imagen de usuario.

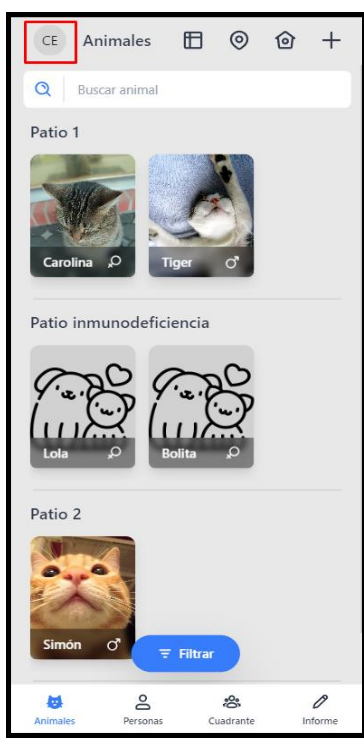


Ilustración 63. GDU - Paso 1 edición perfil

En esta pantalla vas a poder cambiar todos los datos que se muestran, como tu nombre y apellidos, email y teléfono. Además, será posible editar tu foto de perfil dándole a "Seleccionar imagen" y eligiendo una de tu dispositivo. Una vez tengas todos tus datos cambiados, deberás guardarlos dirigiéndote al icono de guardado.

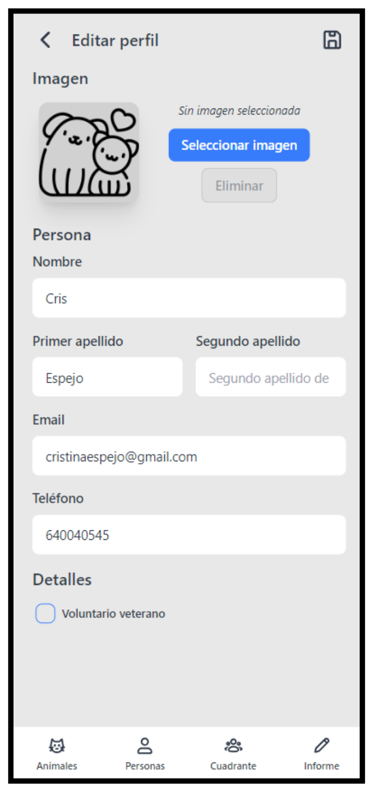


Ilustración 64. GDU - Edición de perfil

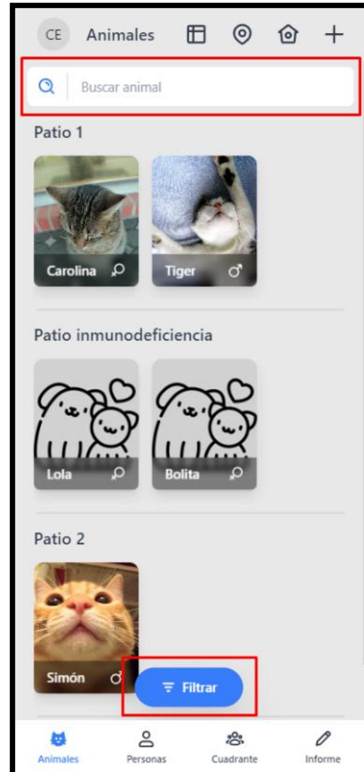
Ver el listado de animales

En la página principal de animales, podrás ver un listado completo de todos los animales de la protectora. Desplazándote verticalmente, encontrarás una lista de patios disponibles, con todos los animales ubicados en cada uno de ellos. Para visualizar los animales dentro de un patio, utiliza el desplazamiento horizontal para navegar entre las imágenes de cada uno. Si deseas obtener más información sobre un animal específico, haz clic en la tarjeta correspondiente para consultar sus detalles.

Filtrado

Si desea buscar un animal concreto o visualizar solo un conjunto de animales con características concretas, podrá hacerlo de dos maneras distintas:

1. Escribiendo el nombre del animal deseado en la barra de búsqueda
2. Seleccionando el icono de filtros en la parte inferior



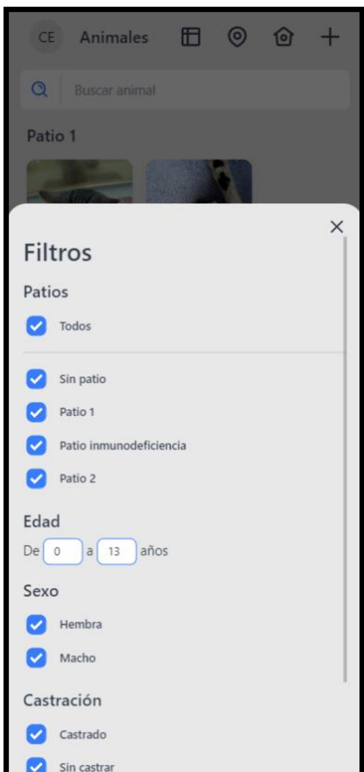


Ilustración 65. GDU - Filtrado de animales

En la pantalla de filtros, podrás ir seleccionando las características de los animales que quieras visualizar. Una vez tengas tu selección hecha, simplemente cierra la pestaña y los cambios se habrán actualizado automáticamente con el filtro deseado.

Visualización de características de un animal

Una vez hayas hecho clic en el animal deseado, te llevará a una página en la que podrás visualizar más detalladamente todas sus características, incluyendo edad, estado de castración, compatibilidad, fecha de nacimiento, su historia… etc. Si en el icono de arriba a la derecha, que visualiza un icono médico, aparece destacado con un asterisco, significa que ese animal tiene tratamientos asignados o futuras citas médicas. Para ver sus detalles clínicos, haga clic en ese icono.

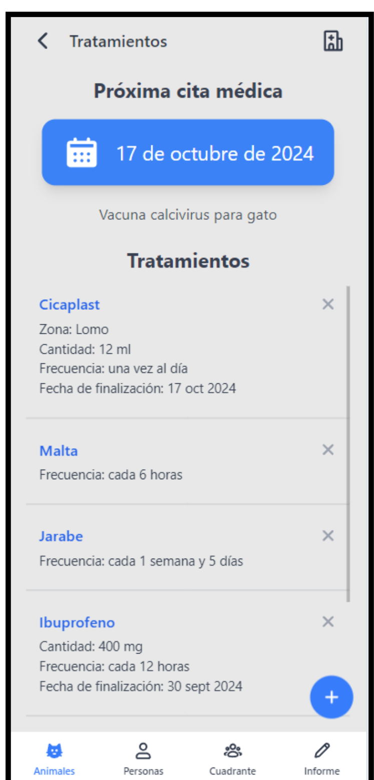


Ilustración 66. GDU - Visualización de citas médicas

En esta pantalla, podrá consultar la próxima cita médica del animal y ver el listado de tratamientos que debe recibir, junto con la información detallada de cada uno. Para agregar un nuevo tratamiento para el animal, haga clic en el icono flotante de añadir que aparece en la parte inferior y complete los datos del nuevo tratamiento. Cuando haya ingresado toda la información, haga clic en "Añadir" para guardarla y verá que aparece en el listado.

Si por el contrario necesita eliminar alguno del listado, haga clic en el icono de la cruz del tratamiento a eliminar y acepte la notificación si está seguro de su borrado.

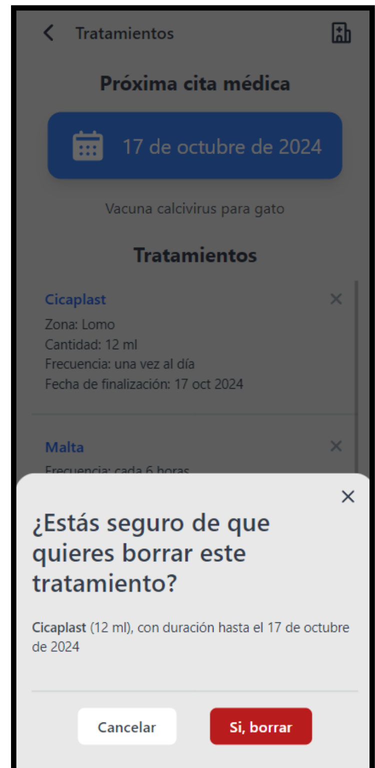


Ilustración 67. GDU - Creación y borrado de tratamientos

Si quiere visualizar el historial de citas médicas, así como futuras citas programadas, dele al icono del hospital dentro de esta pantalla de tratamientos.

Podrá contemplar, de forma más resaltada, las futuras citas médicas programadas para ese animal, las cuales puede eliminar haciendo clic en la cruz correspondiente si ya no se va a realizar dicha cita. Debajo de estas, puede encontrar el historial de citas médicas pasadas que ha realizado este animal.
Si desea programar una nueva cita médica, siga el mismo proceso que utilizó para añadir los tratamientos, haciendo clic en el icono de añadir y escribiendo su información.

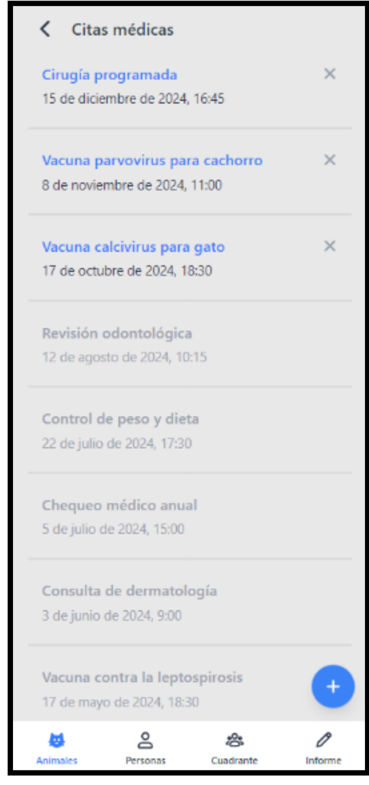


Ilustración 68. GDU - Visualización de citas médicas

Visualización y edición del orden de patio

Para visualizar el orden de patios actual que se debe seguir durante el turno, haga clic en el icono de ubicación dentro de la página principal del listado de animales. Esto mostrará una lista ordenada de los patios que hay en la protectora y su orden.

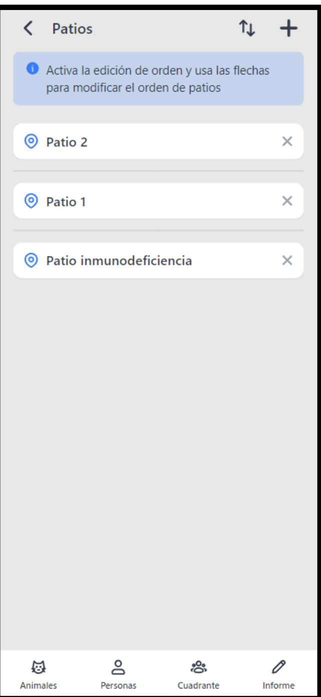


Ilustración 69. GDU - Visualización de patios

El orden de patios es posible alterarlo haciendo clic en las flechas que aparecen arriba a la derecha. Vaya alterando el orden deseado con las flechas que aparecen en cada uno de ellos y, cuando esté listo, haga clic en el icono de guardado.

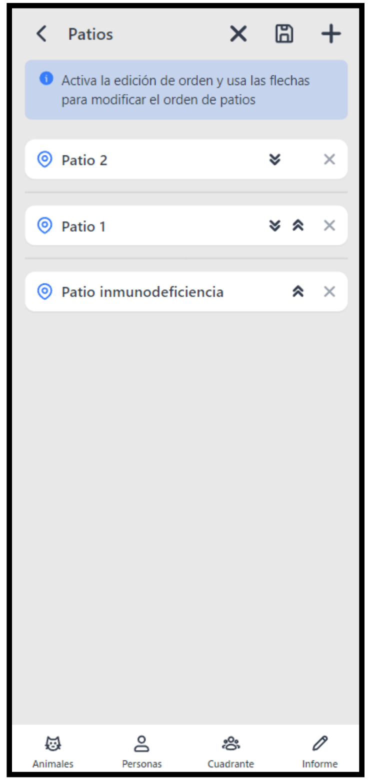
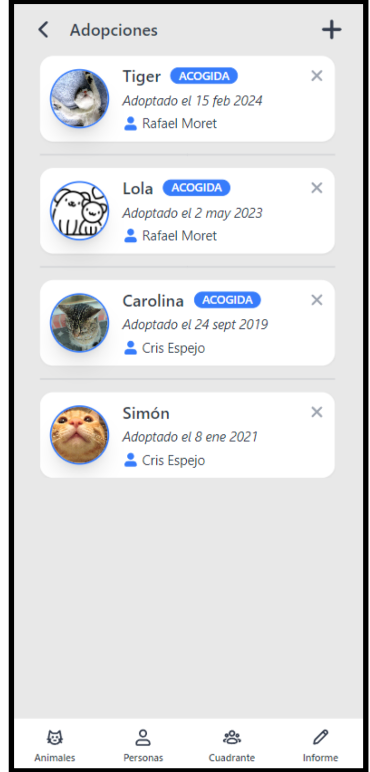


Ilustración 70. GDU - Visualización de orden de patios

Listado de adopciones

Para ver el listado de todas las adopciones registradas, diríjase a la página principal de adopciones y haga clic en el ícono de la casa. Esto abrirá una nueva pantalla que muestra todas las adopciones registradas, tanto temporales como permanentes. Esta vista nos dará información sobre el tipo de adopción que es, el animal adoptado y la persona adoptante.

Para crear una adopción, siga los mismos pasos utilizados en otras vistas: haga clic en el ícono "+". El proceso de creación de una adopción es sencillo: solo necesita elegir un animal y una persona de las opciones disponibles, seleccionar la fecha en la que se realizó la adopción y si es una acogida temporal o una adopción permanente



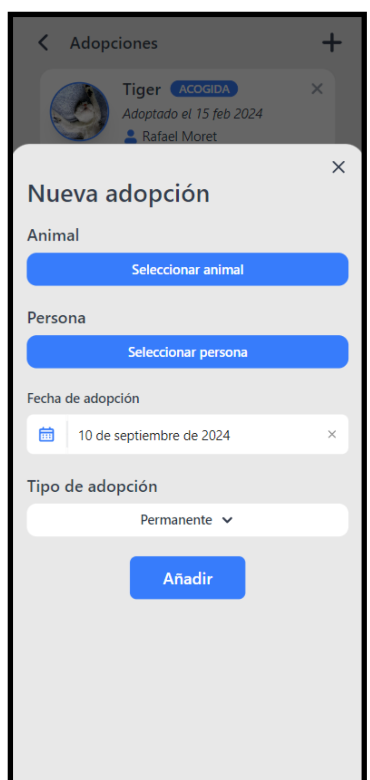
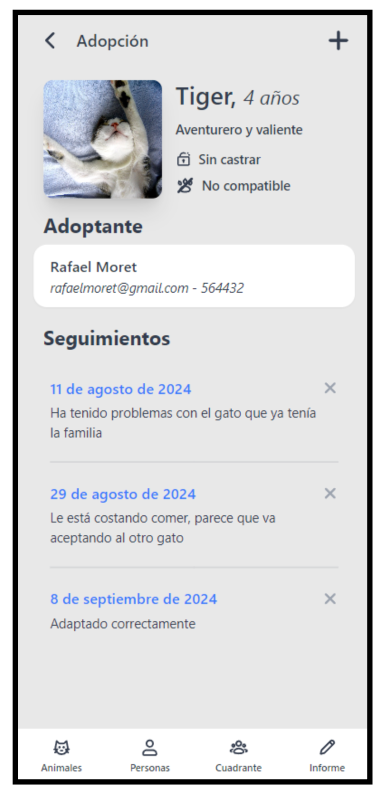


Ilustración 71. GDU - Listado y creación de adopciones

Para ver más detalles del animal adoptado o de la persona adoptante, así como realizar el seguimiento de esta, haga clic en la tarjeta de la adopción a ver.
Se podrá añadir notas de seguimiento con la opción del "+" arriba a la derecha, seleccionando la fecha de la anotación y el comentario.



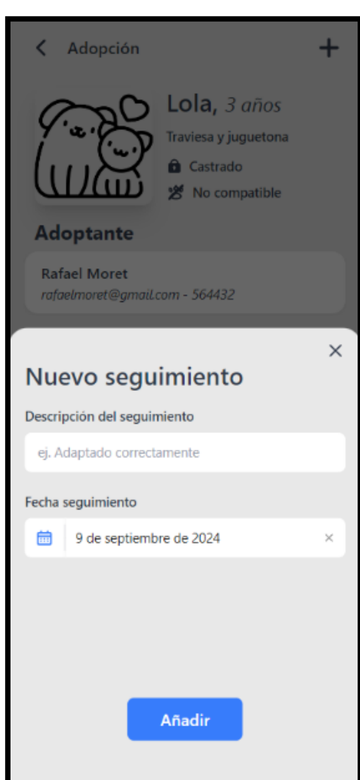


Ilustración 72. GDU - Visualización de adopción y seguimiento

Creación del PDF del listado de animales por patio

Para obtener un pdf con todos los animales listados por patio, así como los tratamientos de cada uno de ellos, en la pantalla principal pulse en el icono de cuadrante. Automáticamente se descargará en su dispositivo un archivo .pdf que puede visualizar con un lector adaptado a ese tipo de archivo.

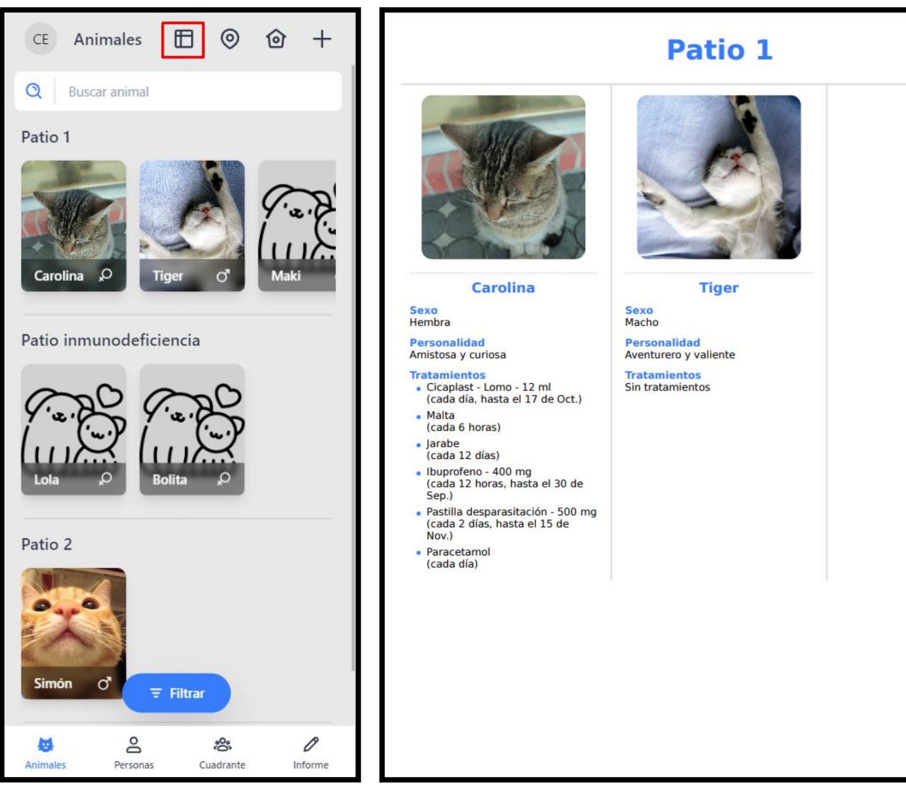
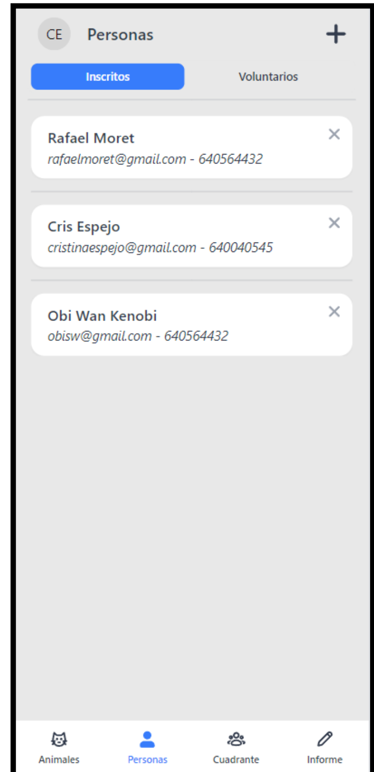


Ilustración 73. GDU - Visualización de creación de PDF

Visualización de las personas registradas en la aplicación

El listado de toda la información personal y de contacto se encuentra en el apartado de **Personas**. En la pestaña de inscritos, encontrarás la información tanto de las personas pertenecientes al grupo de voluntariado, como a personas que se hayan registrado como adoptantes. Como en el resto de los módulos, se puede crear personas mediante el mismo procedimiento. Es posible introducir la información de una persona haciendo clic en el botón "+" y en la nueva pantalla que aparece, deberá introducir los datos.



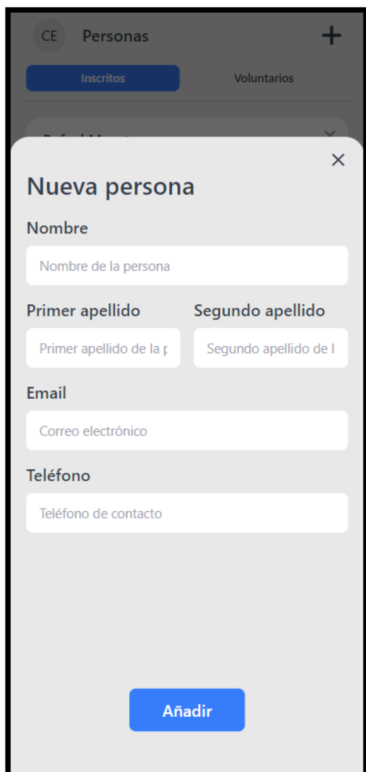


Ilustración 74. GDU - Visualización del listado y creación de personas

Para ver la información referida a los usuarios que manejan la aplicación, concretamente el grupo de voluntarios haga clic en la pestaña de **Voluntarios** que se indica. Podrá ver el nombre y apellidos de las personas registradas en la aplicación, indicando cuáles son veteranos con un icono distintivo.

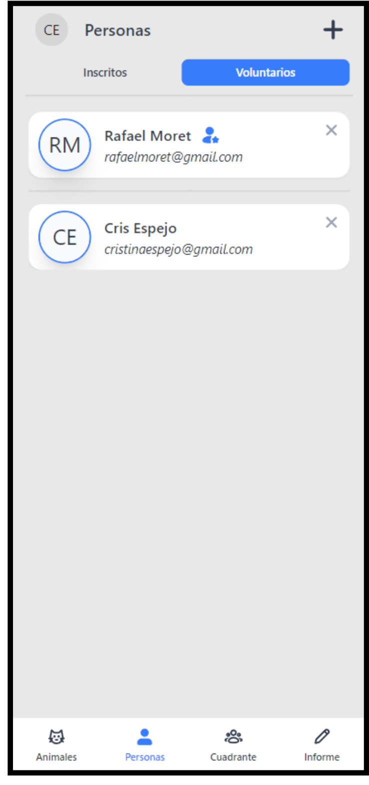


Ilustración 75. GDU - Visualización del listado de voluntarios

Apuntarse a un turno

Para apuntarse a un turno, seleccione la franja horaria y el día de la semana en el cuadrante cuando este se abra. En la pantalla, podrá ver a las personas conectadas en ese momento y los días y franjas horarias a las que se están apuntando. Mientras el turno no esté completo, podrá unirse haciendo clic sobre él. Si desea cambiar su elección, haga clic nuevamente en el turno al que se apuntó y seleccione el nuevo turno deseado.
Cuando el cuadrante se cierre, no podrá apuntarse y su turno quedará registrado.

Ilustración 76. GDU - Visualización del cuadrante semanal

Listado de informes

Puede visualizar todos los informes que se han ido realizando a lo largo del tiempo, haga clic en la pestaña de **informe**. Si quiere ver un informe de una fecha concreta, puede filtrarlo en la barra de búsqueda por fecha seleccionando la deseada.

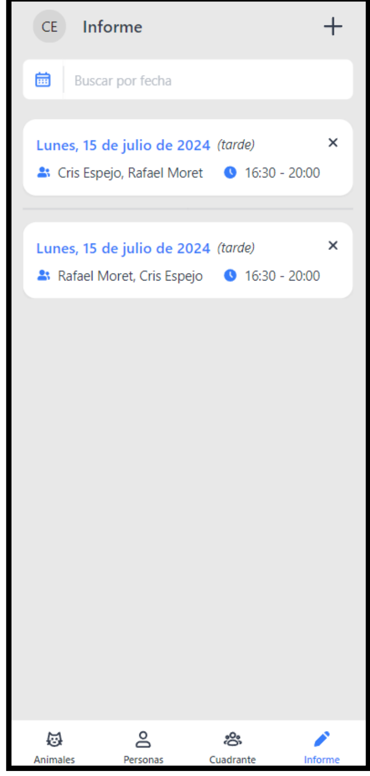


Ilustración 77. GDU - Visualización del listado de informes

Para ver los datos de un informe concreto para visualizar qué ocurrió durante ese turno, haga clic sobre el informe deseado.

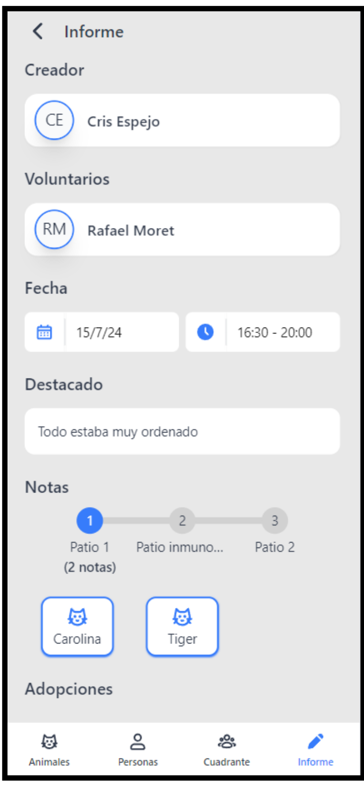


Ilustración 78. GDU - Visualización de un informe de turno

Aquí podrá visualizar la información sobre quién creó el informe, los voluntarios que asistieron al turno, la fecha y franja horaria en la que se realizó y las anotaciones que hicieron durante el turno.

Es posible visualizar notas destacadas sobre eventos importantes ocurridos durante el turno, así como notas personalizadas para cada animal en el patio. Para acceder a ellas, diríjase al apartado de notas y navegue por los patios visibles. Al seleccionar un patio, verá resaltados los animales con anotaciones. Al hacer clic en uno de ellos, podrá leer las notas registradas durante el turno.

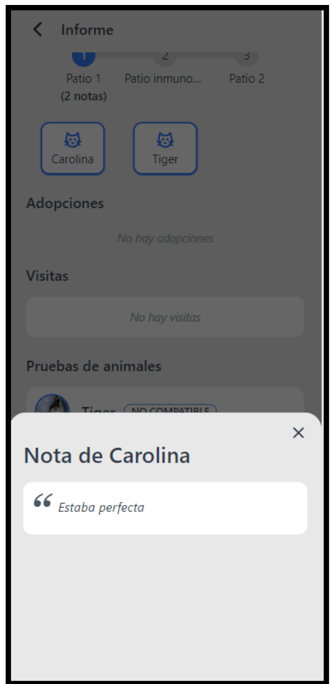


Ilustración 79. GDU - Visualización de nota de informe

El resto de apartados muestra si hubo visitas de gente externa preguntando por algún animal concreto, si se hizo alguna prueba de animal y ocurrió algo destacable durante ella, así como entradas y pérdidas de animales.

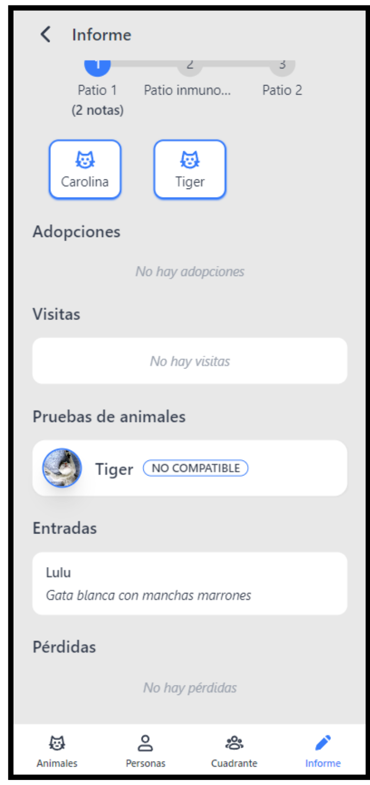


Ilustración 80. GDU - Visualización parte 2 del informe

Para crear un nuevo informe, una vez haya realizado un turno, haga clic en el icono "+" en la vista de listado de informes. Le aparecerá una nueva pantalla donde podrá ir rellenando los datos del turno que realizó.

Para ello, utilice los desplegables que ofrecen opciones disponibles para añadir, como en la selección de voluntarios, donde podrá elegir a las personas que participaron en el turno en ese momento.

Del mismo modo, puede añadir notas destacadas escribiendo en la sección de "Destacados" y presionando el botón "+" para agregarlas. Para incluir notas específicas a animales, siga el mismo procedimiento que para visualizarlas; ahora aparecerá un espacio donde podrá escribir las notas necesarias para cada animal.

Para los demás campos, utilice los formularios y desplegables proporcionados para facilitar la introducción de la información requerida.

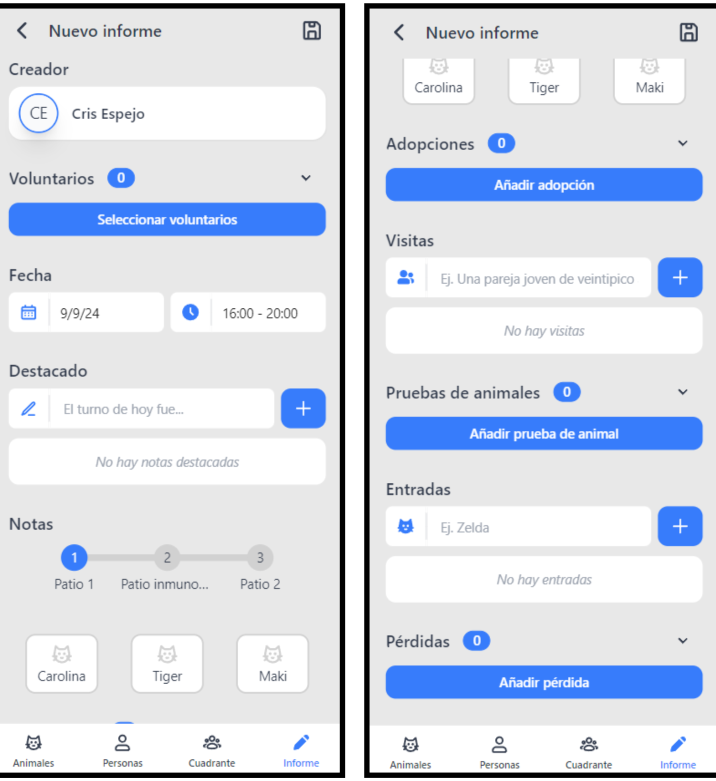


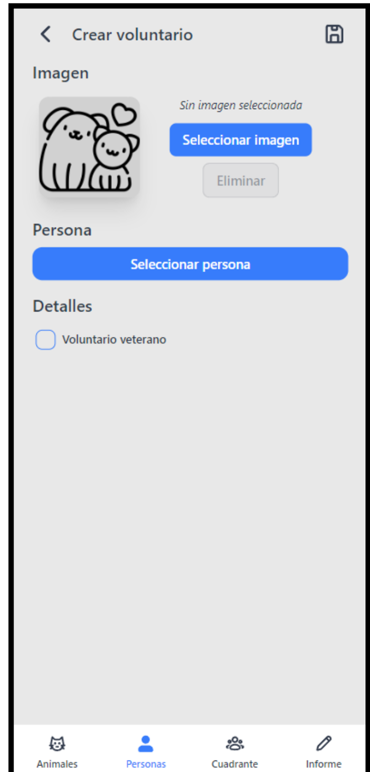
Ilustración 81. GDU - Visualización de creación de informe

Guía para coordinadores

Este apartado va dirigido a los coordinadores de la aplicación, ya que tendrán funcionalidades concretas que solo ellos podrán realizar, debido a que su rol incluye la administración y gestión de permisos especiales.

Eliminar, crear y editar voluntarios

El coordinador es el único usuario capaz de modificar, crear y eliminar datos de voluntarios en la aplicación.
Para crear un nuevo usuario, primero debe crear el perfil de la persona asociada a él y rellenar los datos en la manera que se indicó en el apartado de creación de personas del manual. Luego, en la pestaña de voluntarios, dará clic al botón "+" de arriba a la derecha para añadir un nuevo voluntario.



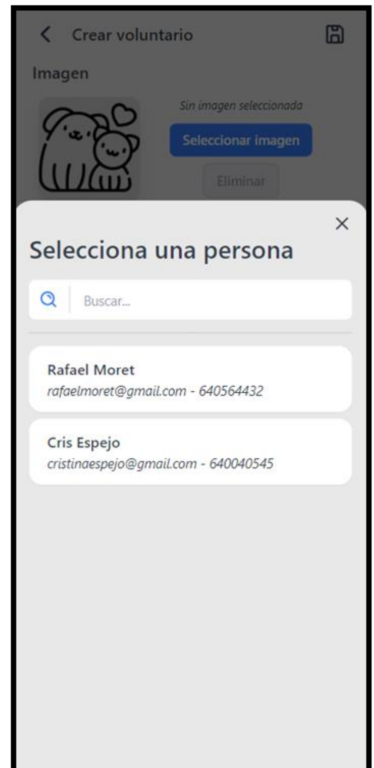


Ilustración 82. GDU - Visdualización creación de voluntario

Aquí podrá selecciónarle una imagen inicial haciendo clic en "seleccionar imagen" y eligiendo una de su dispositivo. Luego, seleccione la persona asociada al usuario a añadir en el desplegable que se abre en **Seleccionar persona.** Una vez ingresados todos los cambios, podrá guardar el unevo usuario haciendo clic al botón de guardar arriba a la derecha, enviándose directamente una contraseña aleatoria al email del usuario para que pueda ingresar en la aplicación.

Adicionalmente, podrá editar los datos de un voluntario concreto haciendo clic en él en la lista de voluntarios, pudiéndose editar tantos sus datos de usuario (como su imagen o el nivel de veteranía que tiene) como sus datos personales.

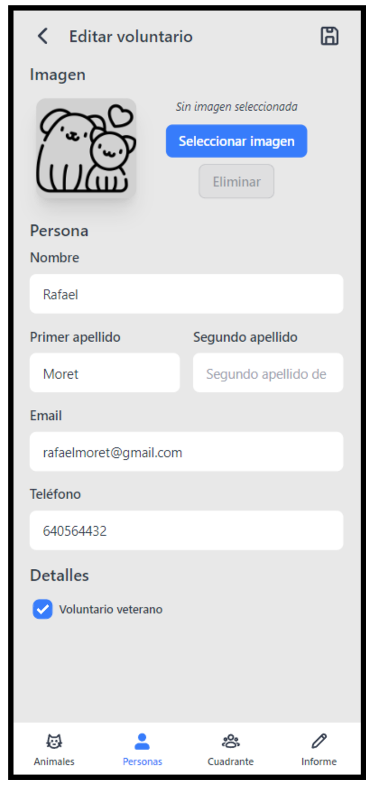


Ilustración 83. GDU - Visualización editor de voluntario

Tendrá además la posibilidad de eliminar voluntarios de la protectora haciendo clic en la cruz que aparece en su tarjeta en el listado.

Editar y eliminar personas

En la pantalla de inscritos, del mismo modo que para voluntarios, podrá eliminar personas haciendo clic en la cruz. Para poder editar, seguirá también el mismo procedimiento que para voluntarios, donde al hacer clic en su tarjeta en el listado le aparecerá una pestaña con los datos a editar.

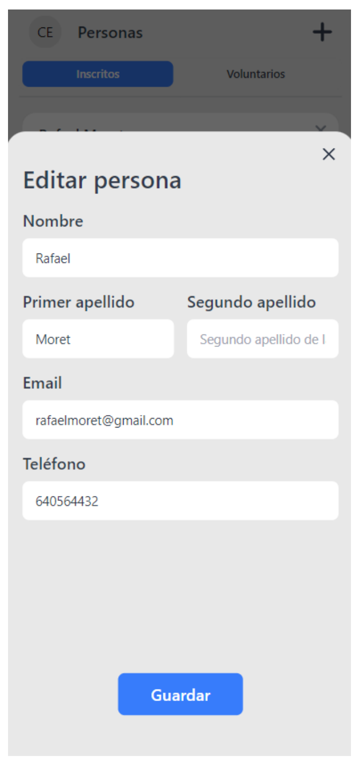


Ilustración 84. GDU - Visualización de editor de persona

Editar estado y limpiar cuadrante

Podrá cambiar el estado del cuadrante a "cerrado" para impedir que alguien se apunte durante ese periodo. Para hacerlo, solo debe hacer clic en el botón de pausado en la parte superior de la pantalla.
Del mismo modo, cuando desee reiniciar el cuadrante para la nueva semana de turnos, puede hacerlo haciendo clic al botón contiguo.

Ilustración 85. GDU - Visualización del cambio de estados del cuadrante

Creación y edición de animales

En el listado de animales, tendrá la posibilidad de añadir un nuevo animal al registro haciendo clic al botón superior de "+". Tras abrirse una nueva pantalla, podrá rellenar los datos del animal, como su sexo, nombre, ubicación en la que se va a encontrar, rasgos de personalidad y añadir una imagen, entre otras. Una vez registrados todos los datos, haga clic en el icono de guardado que aparece. Verá que el nuevo animal se ha registrado en el listado.

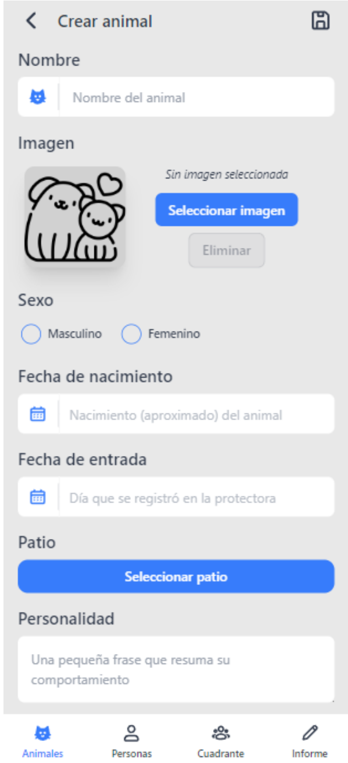


Ilustración 86. GDU - Visualización creación de animal

Para editar la información de un animal, haga clic en el animal que desee modificar del listado y luego en el botón del lápiz que aparece en la parte superior, Tras ello, se abrirá una nueva pantalla como la de creación de animales donde podrá editar los datos que desee.

Ilustración 87. GDU - Visualización icono de edición

Creación y eliminación de patios

Análogo al resto de módulos, puede eliminar y crear patios haciendo clic en los iconos indicados. Para la creación, simplemente introduzca el nombre de patio a crear, donde se añadirá en último lugar en el orden de patios.

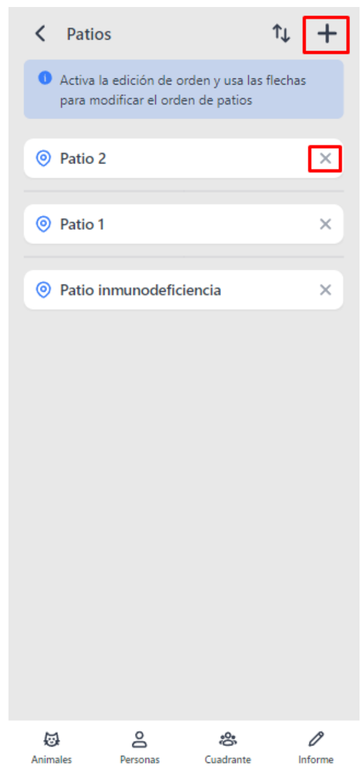


Ilustración 88. GDU - Visualización creación y eliminación de patios

Eliminar una adopción

Del mismo modo que para el resto de los módulos, podrá eliminar una adopción haciendo clic en el icono de borrado que se encuentra en la tarjeta de la adopción a borrar de la lista.

Apéndice C

Estructura de ficheros

B.1 Backend

- *backend*
    - *README.md*
    - *poetry.lock*
    - *pyproject.toml*
    - *.env*
    - *images*
        - *...*
    - *proteapp*
        - *animal_report.py*
        - *email.py*
        - *websockets.py*
        - *exceptions.py*
        - *templates.py*
            - *register.html*
            - *...*
        - *models*
            - *base.py*
            - *time.py*
            - *nosql*
                - *inform.py*
                - *shift.py*
                - *yard_order.py*
            - *sql*
                - *animals.py*
                - *adoptions.py*
                - *...*
        - *api*
            - *main.py*
            - *deps.py*
            - *animals*

        - *adapters.py*
        - *routes.py*
        - *schemas.py*
    - *auth*
        - *...*
    - *informs*
        - *...*
    - *...*

B.2 Frontend

- *frontend*
    - *README.md*
    - *env.d.ts*
    - *index.html*
    - *package-lock.json*
    - *package.json*
    - *postcss.config.js*
    - *tailwind.config.js*
    - *tsconfig.app.json*
    - *tsconfig.json*
    - *tsconfig.node.json*
    - *vite.config.ts*
    - *public*
        - *favicon.ico*
        - *images*
            - *...*
    - *src*
        - *app.vue*
        - *main.ts*
        - *types.ts*
        - *utils.ts*
        - *assets*
            - *...*
        - *components*
            - *BottomDrawer.vue*
            - *DateInput.vue*
            - *...*
        - *composable*
            - *useAuthFetch.vue*
            - *useToastNotifications.vue*

            - *...*
    - *config*
        - *NavigationBar.ts*
        - *RestrictedRoutes.ts*
    - *modules*
        - *Animal*
            - *adapters.ts*
            - *animal.config.ts*
            - *api.ts*
            - *declarations.ts*
            - *components*
                - *AnimalCard.vue*
                - *AnimalDetails.vue*
                - *...*
            - *composble*
                - *useAnimalFilters.vue*
            - *views*
                - *AnimalEditorView.vue*
                - *AnimalListView.vue*
        - *Inform*
            - *adapters.ts*
            - *api.ts*
            - *...*
        - *Person*
            - *...*
    - *router*
        - *index.ts*
    - *skeleton*
        - *AppHeader.vue*
        - *NavigationBar.vue*
    - *store*
        - *AnimalStore.ts*
        - *AuthStore.ts*
        - *...*
    - *views*
        - *HomeView*
        - *LoginView.vue*
        - *...*

Índice de figuras

Ilustración 1. Diagrama de generación y validación del token ... 39
Ilustración 2. Diagrama de clases de los datos relacionales ... 41
Ilustración 3. Diagrama de las entidades no relacionales ... 42
Ilustración 4. Dependencia de la sesión SQL ... 46
Ilustración 5. Ejemplo de uso de la dependencia de la sesión SQL ... 46
Ilustración 6. Método de decodificación y validación del token ... 47
Ilustración 7. Ejemplo de uso de la dependencia de autenticación ... 48
Ilustración 8. Dependencia para comprobar el rol de administrador ... 49
Ilustración 9. Ejemplo de uso de la dependencia para comprobar el rol de administrador ... 50
Ilustración 10. Declaración de las clases base ... 51
Ilustración 11. Clase base para los modelos SQL ... 52
Ilustración 12. Clase base para los modelos NoSQL ... 52
Ilustración 13. Ejemplos de ItemSelector ... 55
Ilustración 14. Ejemplos de ItemList ... 57
Ilustración 15. Ejemplo de TextInput ... 58
Ilustración 16. Ejemplo de DateInput ... 58
Ilustración 17. Ejemplo de TimeInput ... 59
Ilustración 18. Ejemplo de HoursInput ... 59
Ilustración 19. Ejemplo de DropdownSelector ... 60
Ilustración 20. Ejemplo de notificaciones ... 60
Ilustración 21. Ejemplo de uso del *composable useToastNotifications* ... 61
Ilustración 22. Ejemplo de ImagePicker ... 61
Ilustración 23. Ejemplo de uso de un icono ... 62
Ilustración 24. Ejemplo de store ... 63
Ilustración 25. Ejemplo de esquema ... 64
Ilustración 26. Ejemplo de adaptador ... 64
Ilustración 27. Vista principal del listado de animales ... 66
Ilustración 28. Menú adicional con el filtrado de animales ... 66
Ilustración 29. Ordenación de los animales por patio ... 67
Ilustración 30. Distribución de los animales en la vista ... 67
Ilustración 31. Vista detallada de un animal ... 68
Ilustración 32. Vista del editor de animales ... 69
Ilustración 33. Menú adicional de selección de patio ... 70
Ilustración 34. Ejemplos de uso de elementos simples en el editor de animales ... 71
Ilustración 35. Modelo de datos de un animal ... 73
Ilustración 36. Esquemas de salida del módulo de animales ... 74
Ilustración 37. Esquema de entrada del módulo de anmales ... 75
Ilustración 38. Vista de la lista de tratamientos de un animal ... 76
Ilustración 39. Implementación del formato de la frecuencia ... 76
Ilustración 40. Menú de creación de nuevos tratamientos ... 77
Ilustración 41. Modelo de datos de un tratamiento ... 78
Ilustración 42. Esquema de salida del módulo de tratamientos ... 79

Ilustración 43. VIsta del histórico y creador de citas médicas 80
Ilustración 44. Modelo de datos de una cita médica 82
Ilustración 45. Esquema de salida del módulo de citas médicas 83
Ilustración 46. Ejemplo de la vista de un patio en el cuadrante de animales 84
Ilustración 47. Implementación del guardado y envío del cuadrante en el endpoint "/report"
.... 85
Ilustración 48. Vista del listado de adopciones 86
Ilustración 49. Implementación del listado de adopciones utilizando el componente ItemList
.... 87
Ilustración 50. VIsta de creación de adopciones 88
Ilustración 51. Vista concreta de una adopción 89
Ilustración 52. Vista del creador de nuevos seguimientos 90
Ilustración 53. Modelo de datos de una adopción 92
Ilustración 54. Esquema de salida principal del módulo de adopciones 94
Ilustración 55. Vista del listado de patios y su menú de creación 95
Ilustración 56. Vista del listado de patios con el modo de reordenación activado 96
Ilustración 57. Modelo de datos de un patio 98
Ilustración 58. Modelo de datos de un orden de patios 98
Ilustración 59. Esquemas referentes a un patio 100
Ilustración 60. Esquemas referentes a un orden de patios 100
Ilustración 61. GDU - Visualización del mail de registro 109
Ilustración 62. GDU - Visualización del login 110
Ilustración 63. GDU - Paso 1 edición perfil 110
Ilustración 64. GDU - Edición de perfil 111
Ilustración 65. GDU - Filtrado de animales 112
Ilustración 66. GDU - Visualización de citas médicas 113
Ilustración 67. GDU - Creación y borrado de tratamientos 114
Ilustración 68. GDU - Visualización de citas médicas 115
Ilustración 69. GDU - Visualización de patios 115
Ilustración 70. GDU - Visualización de orden de patios 116
Ilustración 71. GDU - Listado y creación de adopciones 117
Ilustración 72. GDU - Visualización de adopción y seguimiento 118
Ilustración 73. GDU - Visualización de creación de PDF 119
Ilustración 74. GDU - Visualización del listado y creación de personas 120
Ilustración 75. GDU - Visualización del listado de voluntarios 121
Ilustración 76. GDU - Visualización del cuadrante semanal 122
Ilustración 77. GDU - Visualización del listado de informes 123
Ilustración 78. GDU - Visualización de un informe de turno 123
Ilustración 79. GDU - Visualización de nota de informe 124
Ilustración 80. GDU - Visualización parte 2 del informe 125
Ilustración 81. GDU - Visualización de creación de informe 126
Ilustración 82. GDU - Visdualización creación de voluntario 127
Ilustración 83. GDU - Visualización editor de voluntario 128
Ilustración 84. GDU - Visualización de editor de persona 129
Ilustración 85. GDU - Visualización del cambio de estados del cuadrante 130
Ilustración 86. GDU - Visualización creación de animal 131
Ilustración 87. GDU - Visualización icono de edición 131

Ilustración 88. GDU - Visualización creación y eliminación de patios 132

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga