UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

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 miembros

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

Implementation of the functionalities of the member management module

Realizado por
Cristina Espejo Roque

Otros integrantes del grupo
Rafael Moret Galán

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

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

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 las 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, weasegurando 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 voluntariado.

3.1 Requisitos funcionales

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

- **RF1. Registro de voluntarios:** Un coordinador podrá registrar a nuevos usuarios desde dentro de la aplicación con un email indicado, estableciéndose una contraseña aleatoria que luego ellos podrán cambiar desde la aplicación.

- **RF2. Inicio de sesión:** Los voluntarios podrán ingresar en la aplicación con email y contraseña.

- **RF3. CRUD de voluntarios:** Será posible acceder al listado completo de usuarios, así como modificar su propio perfil. Adicionalmente, los administradores podrán crear y eliminar usuarios.

- **RF4. Distinción de roles:** Los usuarios podrán tener dos tipos de roles distintos en la aplicación, siendo estos voluntario y coordinador. Estos roles sirven además para restringir algunas funcionalidades de la aplicación únicamente al rol de coordinador.

- **RF5. CRUD de inscritos:** Se podrá consultar el listado de los datos de personas tanto pertenecientes al grupo de voluntariado (usuarios) como a los datos de adoptantes que han figurado en la protectora. Además, estos datos podrán ser modificados y eliminados únicamente por el coordinador.

- **RF6. Modificación de tu perfil:** El usuario podrá modificar los datos de su perfil, como el nombre, apellidos, email y teléfono, así como cambiar su imagen de perfil.

- **RF7. CRD Informe de turnos:** Se podrán crear informes de turno con los datos necesarios para detallar lo ocurrido durante un turno, así como los voluntarios que pertenecieron, la fecha y franja horaria. Además, una vez creados, se podrán eliminar.

- **RF8. Acceso a historial de informes de turnos:** Se podrán consultar todos los informes de turnos creados anteriormente en orden cronológico.

- **RF9. Visualización y modificación de datos del cuadrante:** Se podrán visualizar los datos que hay en el cuadrante de turnos en ese momento, así como apuntarse a un día y franja deseado. La modificación solo se podrá realizar si el cuadrante está abierto.

- **RF10. Actualización en tiempo real de los cambios en el cuadrante:** Se podrá ver en tiempo real las personas que hay actualmente apuntadas en el cuadrante sin tener que recargar la página.

- **RF11. Borrado directo de todos los datos del cuadrante:** El coordinador podrá reiniciar el cuadrante para volver a dejarlo limpio cuando reinicie la semana.

- **RF12. Visualización y modificación de estado del cuadrante:** Se podrá ver el estado del cuadrante (abierto o cerrado) en todo momento para ver si es posible apuntarse o no. Solo el coordinador podrá cambiar este estado

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:** código fuente de la aplicación debe estar bien documentado y 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 registrados**: 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. Registro de usuario
Descripción	Se describe cómo un coordinador puede registrar a un nuevo usuario
Precondición	El usuario no está registrado en la aplicación
Postcondición	El usuario está registrado con email y contraseña
Actores	Coordinador
Escenario principal	1. El coordinador ingresa en la aplicación 2. El coordinador entra en el apartado de personas 3. El coordinador crea una nueva persona con los datos personales del nuevo usuario: nombre, apellidos, email y teléfono 4. Se registra la nueva persona con los datos introducidos comprobando si los datos son válidos 5. El coordinador se dirige dentro de la misma página a la pestaña de Voluntarios

	6. El coordinador le da a registrar nuevo usuario 7. El coordinador selecciona a la persona a la que se le va a asignar el nuevo usuario, elige opcionalmente una imagen y añade si es veterano o no 8. Se registra el nuevo usuario añadido comprobando si los datos son válidos, asignándole una contraseña aleatoria 9. Al nuevo usuario registrado le llega al email registrado la contraseña generada para que pueda ingresar en la aplicación
Escenario alternativo	3.b Los datos registrados no son válidos, cerrará la pestaña de creación y aparecerá un mensaje de error por pantalla 7.b Los datos registrados no son válidos, cerrará la pestaña de creación y aparecerá un mensaje de error por pantalla

Tabla 1. Caso de uso RF1

Titulo	Caso de uso RF2. Inicio de sesión
Descripción	Se describe cómo un usuario registrado puede ingresar en la aplicación con correo y contraseña
Precondición	El usuario está registrado en la aplicación y no ha iniciado sesión
Postcondición	El usuario ha iniciado sesión y está dentro de la aplicación
Actores	Usuarios
Escenario principal	1. El usuario inicia la aplicación 2. El usuario introduce su email y contraseña en el apartado del login 3. El usuario da a iniciar sesión 4. Se verifican los datos ingresados 5. El usuario entra en la aplicación tras haber iniciado la sesión
Escenario alternativo	4.b Los datos ingresados no son válidos, aparecerá un mensaje de error por pantalla

Tabla 2. Caso de uso RF2

Titulo	Caso de uso RF3. CRUD de voluntarios
Descripción	Se describe cómo un usuario puede consultar, modificar y eliminar voluntarios
Precondición	El usuario ha iniciado sesión en la aplicación
Postcondición	El usuario ha consultado, eliminado, editado a un voluntario en la aplicación
Actores	Coordinador, voluntario
Escenario principal	1. El usuario inicia la aplicación e inicia sesión 2. El usuario va al apartado de personas 3. El usuario va a la pestaña de voluntarios 4. El usuario puede consultar los datos de los voluntarios que hay inscritos en la aplicación 5. El coordinador puede editar los datos de un voluntario a. El coordinador clica en el voluntario que quiere consultar b. El coordinador introduce los nuevos datos en la pestaña que se abre c. El coordinador da a guardar los datos d. Se comprueban los datos y se registran los cambios e. Se notifica que se ha editado correctamente 6. El coordinador puede eliminar a un voluntario a. El coordinador da clic al botón de eliminar b. El coordinador acepta la notificación de borrado c. Se notifica que se ha eliminado correctamente 7. El coordinador puede crear a un nuevo voluntario a. El coordinador da al botón de crear b. El coordinador introduce los datos necesarios como persona sobre la que se crea el voluntario, su imagen y su veteranía c. El coordinador da a guardar d. Se comprueban los datos y se registra un nuevo voluntario e. Se envía un email al correo del usuario registrado indicando su nueva contraseña f. Se notifica que se ha creado correctamente

Escenario alternativo	5.d.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 7.d.b Ha ocurrido un error en el sistema y no se ha podido crear, aparecerá un mensaje de error por pantalla

Tabla 3. Caso de uso de RF3

Titulo	Caso de uso RF5. CRUD de personas
Descripción	Se describe cómo un usuario puede crear, editar, visualizar y eliminar personas
Precondición	El usuario ha iniciado sesión en la aplicación
Postcondición	El usuario ha creado, eliminado, editado o consultado a una persona en la aplicación
Actores	Coordinador, voluntario
Escenario principal	1. El usuario inicia la aplicación e inicia sesión 2. El usuario va al apartado de personas 3. El usuario puede consultar los datos de las personas que hay inscritas en la aplicación 4. El usuario puede crear a una nueva persona a. El usuario da al botón de crear b. El usuario introduce los datos necesarios como nombre, apellidos, email y teléfono c. El usuario da a guardar d. Se comprueban los datos y se registra una nueva persona e. Se notifica que se ha creado correctamente 5. El coordinador puede editar los datos de una persona a. El coordinador clica en la persona que quiere consultar b. El coordinador introduce los nuevos datos en la pestaña que se abre c. El coordinador da a guardar los datos d. Se comprueban los datos y se registran los cambios e. Se notifica que se ha editado correctamente

	6. El coordinador puede eliminar a una persona a. El coordinador da clic al botón de eliminar b. El coordinador 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.d.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 4. Caso de uso RF5

Titulo	Caso de uso RF6. Modificación de tu perfil
Descripción	Se describe cómo un usuario puede editar los datos de su propio perfil
Precondición	El usuario ha iniciado sesión en la aplicación
Postcondición	El usuario ha editado los datos de su perfil
Actores	Usuarios
Escenario principal	1. El usuario entra en la aplicación 2. El usuario da clic a la imagen de su perfil arriba a la izquierda 3. El usuario entra en la pantalla de edición de perfil 4. El usuario modifica sus datos a. El usuario introduce datos de texto nuevos referentes al nombre, apellidos, teléfono o correo b. El usuario da a elegir una nueva imagen i. Selecciona una imagen de sus archivos ii. Acepta la imagen elegida 5. El usuario da a guardar la información 6. La información se valida en sistema 7. Los cambios se registran en la base de datos y se notifica de que los cambios han sido registrados adecuadamente
Escenario alternativo	4.b.ii.b Ha habido algún error en el sistema de selección de fotografía, salga un error por pantalla

	6.b Los datos ingresados no son válidos, aparecerá un mensaje de error por pantalla

Tabla 5. Caso de uso RF6

Titulo	Caso de uso RF7. CRD de informes de turno
Descripción	Se describe cómo un usuario puede crear, eliminar o consultar un informe de turno
Precondición	El usuario ha iniciado sesión en la aplicación
Postcondición	El usuario ha creado, eliminado o consultado un informe
Actores	Usuarios
Escenario principal	1. El usuario inicia la aplicación e inicia sesión 2. El usuario va al apartado de informes de turno 3. El usuario puede consultar los datos de un informe de turno concreto haciendo clic en él 4. El usuario puede crear un nuevo informe de turno a. El usuario da al botón de crear b. El usuario introduce los datos necesarios, haciendo uso de los desplegables de selección, los inputs de texto y los componentes de selección de fechas c. El usuario da a guardar d. Se comprueban los datos y se registra un nuevo informe de turno e. Se notifica que se ha creado correctamente 5. El usuario puede eliminar a un informe de turno 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.d.b Los datos ingresados no son válidos, aparecerá un mensaje de error por pantalla

Tabla 6. Caso de uso RF7

Titulo	Caso de uso RF8. Acceso al historial de informes de turno
Descripción	Se describe cómo un usuario puede consultar el histórico de turnos en orden cronológico
Precondición	El usuario ha iniciado sesión en la aplicación
Postcondición	El usuario ha consultado el histórico de informes
Actores	Usuarios
Escenario principal	1. El usuario inicia la aplicación e inicia sesión 2. El usuario va al apartado de informes de turno 3. El usuario puede visualizar el histórico de informes en orden cronológico, pudiéndose ver los creadores y la fecha de creación
Escenario alternativo	3.b Ha ocurrido un error en el sistema y se notifica que no se puede acceder al listado de informes 3.c No hay ningún informe creado y se notifica de que no hay informes

Tabla 7. Caso de uso RF8

Titulo	Caso de uso RF9. Visualización y modificación de datos del cuadrante de turnos
Descripción	Se describe cómo un usuario puede consultar y modificar los datos del cuadrante de turnos
Precondición	1. El usuario ha iniciado sesión en la aplicación 2. Para una nueva inscripción, el cuadrante está en estado abierto y no tiene voluntarios veteranos o el número de voluntarios totales es inferior a dos
Postcondición	El usuario ha consultado y/o se ha apuntado en el cuadrante de turnos
Actores	Usuarios
Escenario principal	1. El usuario inicia la aplicación e inicia sesión 2. El usuario va al apartado de cuadrante semanal

	3. El usuario puede visualizar quiénes son los usuarios apuntados en cada día y franja horaria. 4. El usuario hace clic en el día y franja horaria al que quiere apuntarse 5. Los datos del usuario quedan registrados en ese turno
Escenario alternativo	4.b Si el turno aún no tiene ningún voluntario de tipo veterano, el usuario podrá apuntarse. 4.b.1 Si el usuario es veterano, ese turno se cierra y no pueden apuntarse más usuarios 4.b.2 Si el usuario es novato, el turno seguirá abierto para que pueda apuntarse otro usuario

Tabla 8. Caso de uso RF9

Titulo	Caso de uso RF10. Actualización en tiempo real de los cambios en el cuadrante
Descripción	Se describe cómo la modificación de los datos del turno por parte de un usuario es inmediatamente mostrada al resto de usuarios
Precondición	1. El usuario ha iniciado sesión en la aplicación 2. Se le permite hacer modificaciones en el turno
Postcondición	El resto de los usuarios puede ver la modificación que ha hecho el usuario al momento
Actores	Usuarios
Escenario principal	1. El usuario inicia la aplicación e inicia sesión 2. El usuario va al apartado de cuadrante semanal 3. El usuario hace una modificación en los datos del turno 4. Esta modificación es representada en la vista del resto de usuarios instantáneamente
Escenario alternativo	3.b Se ha producido un error en el sistema, se notifica del error al usuario

Tabla 9. Caso de uso RF10

Titulo	Caso de uso RF11. Borrado directo de todos los datos del cuadrante de turno
Descripción	Se describe cómo un coordinador puede limpiar el cuadrante de turnos
Precondición	1. El coordinador ha iniciado sesión en la aplicación 2. El cuadrante de turnos tiene usuarios apuntados en cada turno
Postcondición	El cuadrante queda sin ningún usuario apuntado en sus turnos
Actores	Coordinador
Escenario principal	1. El coordinador inicia la aplicación e inicia sesión 2. El coordinador va al apartado de cuadrante semanal 3. El coordinador da clic al botón de limpiar cuadrante 4. El cuadrante se reinicia y queda vacío 5. Se es notificado por pantalla de que todo ha ido correctamente
Escenario alternativo	4.b Ha ocurrido un error en el sistema y no se ha podido reiniciar, dando un aviso por pantalla de que ha ocurrido algo inesperado

Tabla 10. Caso de uso RF11

Titulo	Caso de uso RF12. Visualización y modificación del estado del cuadrante de turno
Descripción	Se describe cómo un coordinador puede cambiar el estado del cuadrante a abierto o cerrado
Precondición	1. El coordinador ha iniciado sesión en la aplicación
Postcondición	El estado del turno queda actualizado
Actores	Coordinador
Escenario principal	1. El coordinador inicia la aplicación e inicia sesión 2. El coordinador va al apartado de cuadrante semanal 3. El coordinador visualiza el estado actual del cuadrante 4. El coordinador da clic al botón de cambiar estado

	5. El sistema registra el cambio y se visualiza el cambio de estado
Escenario alternativo	5.b Ha ocurrido un error en el sistema y no se ha podido modificar, dando un aviso por pantalla de que ha ocurrido algo inesperado

Tabla 11. 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.

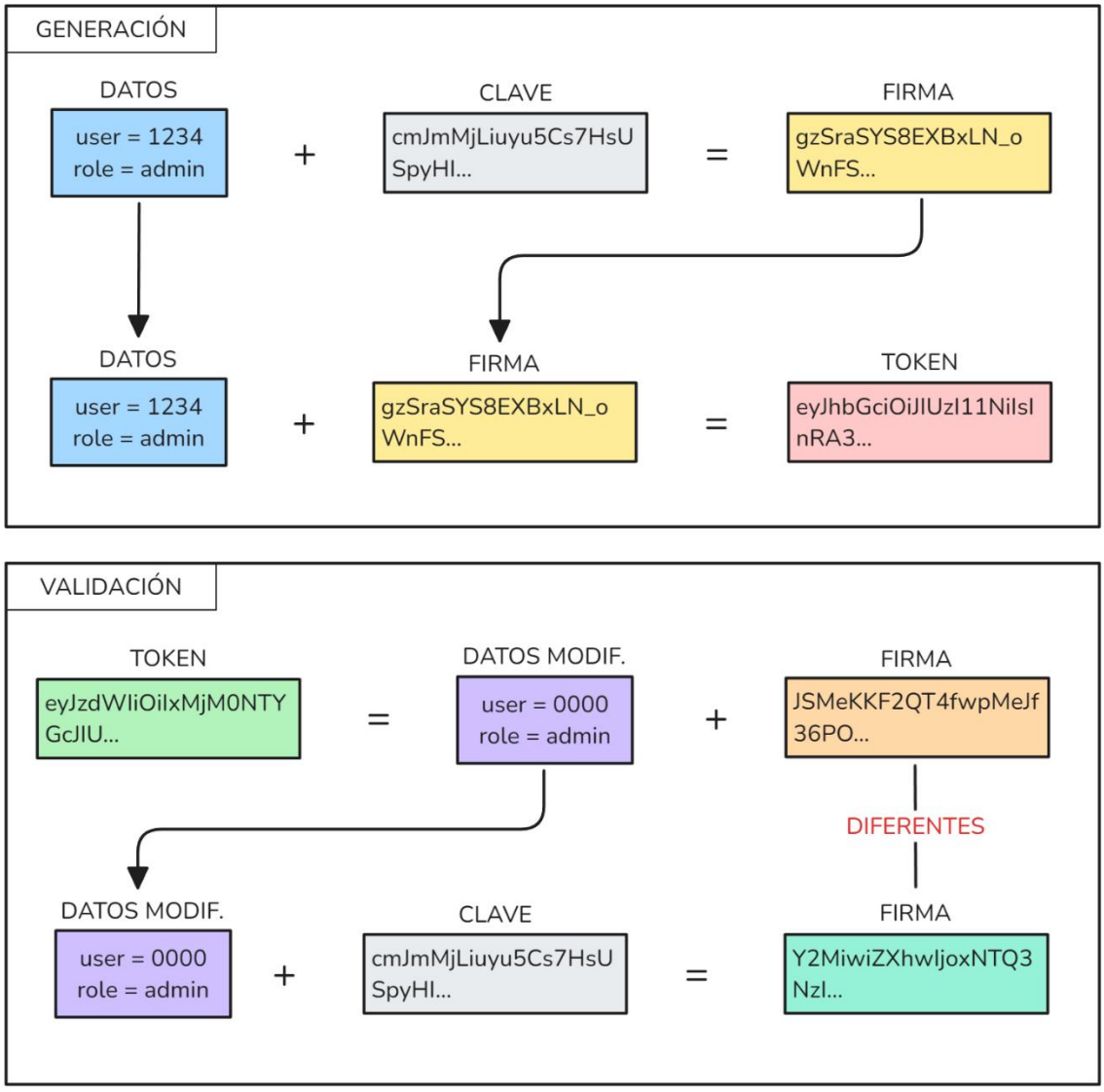


Ilustración 1. Diagrama de generación y validación del token

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.

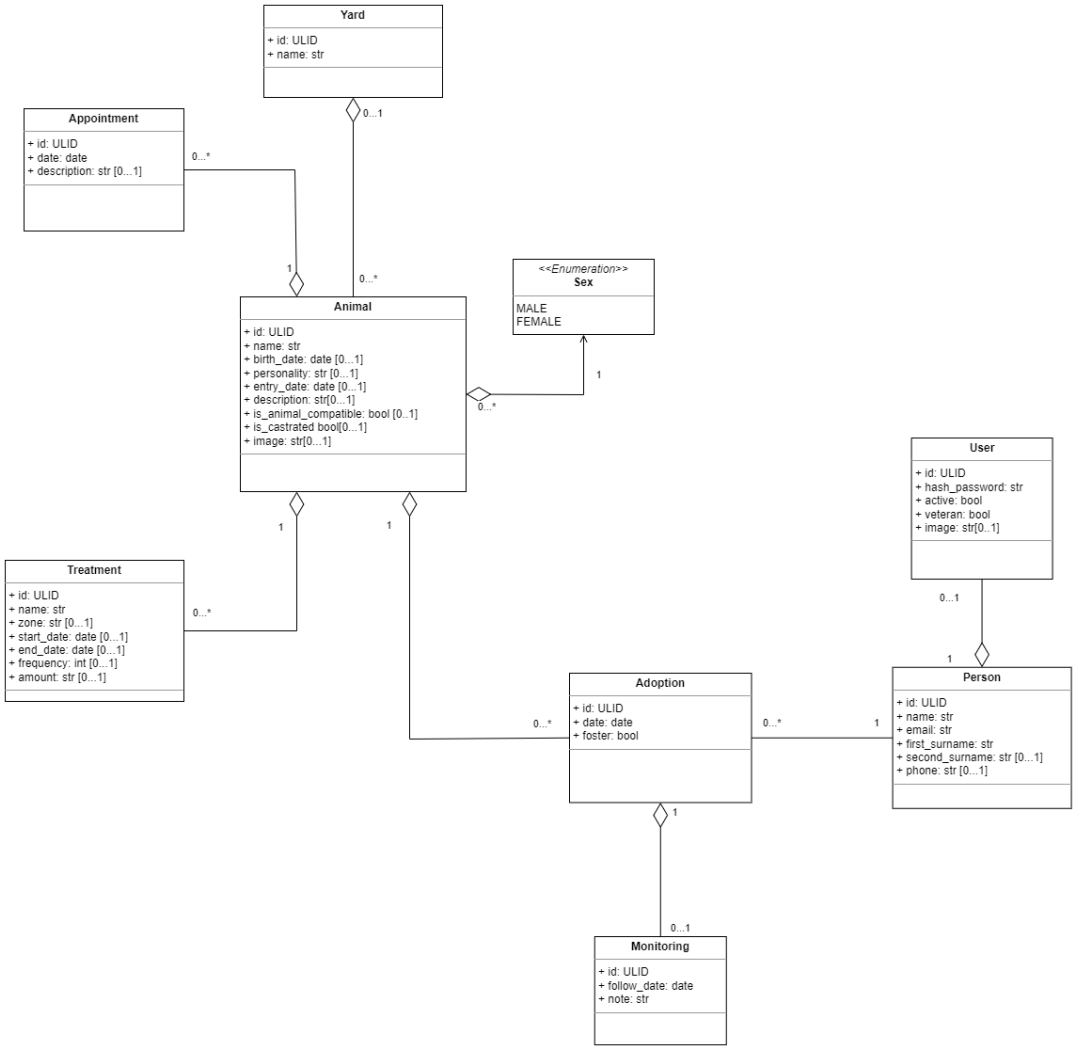


Ilustración 2. Diagrama de clases de los datos relacionales

4.2.2 Datos no relacionales

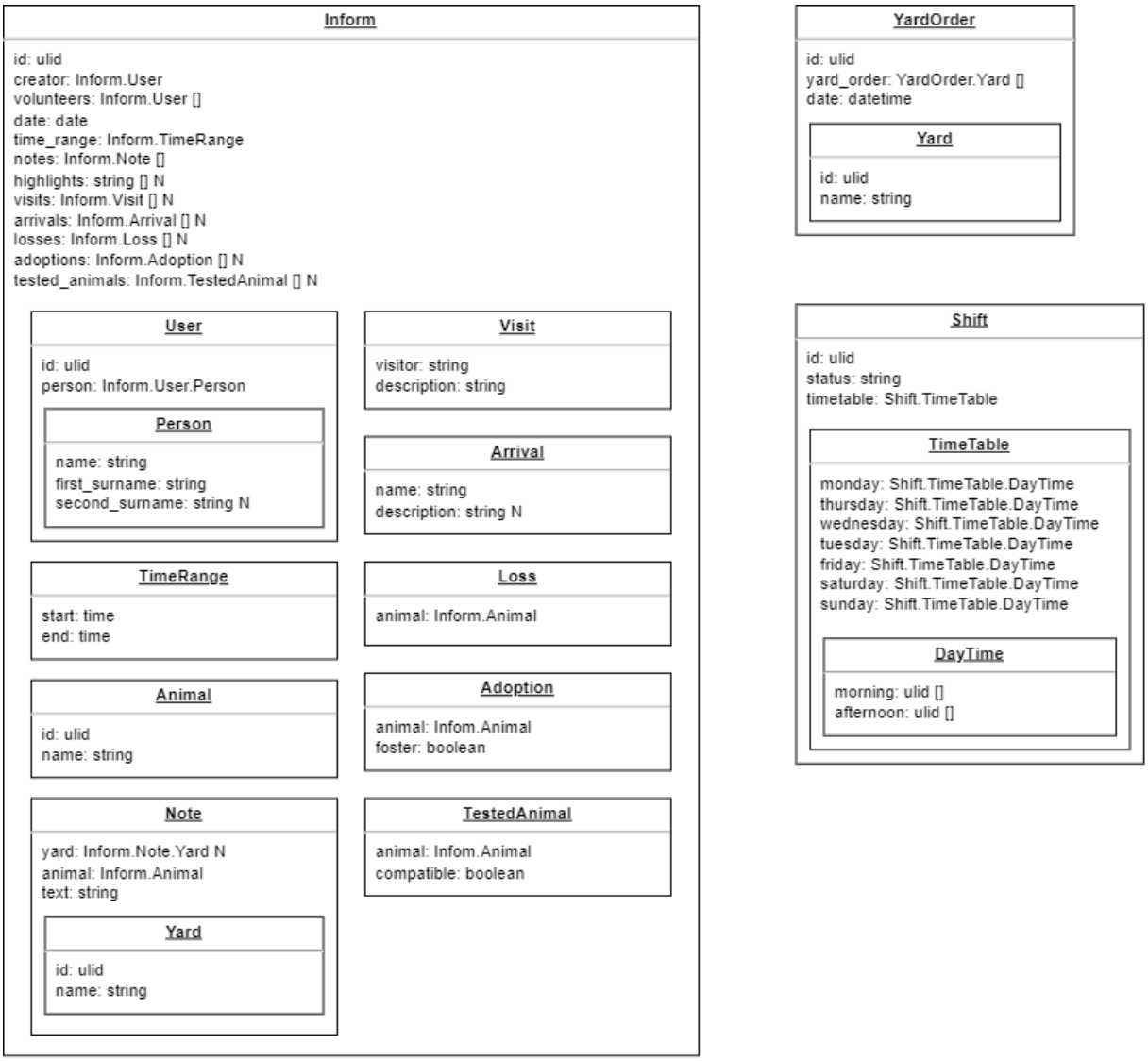


Ilustración 3. Diagrama de las entidades no relacionales

- **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 necesitaremos 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 hacerlas disponibles. 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.

```python
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.

```python
@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.

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 sobrescribir 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*).

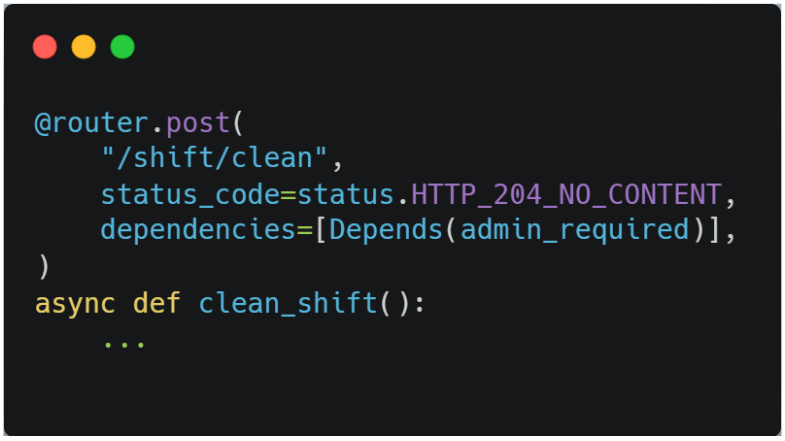
2. Crear una nueva dependencia basada en la anterior que exigía que el usuario hubiera iniciado sesión. Esta nueva dependencia además usará la clase *Security*, en lugar de *Depends*, para incluir el rol de *admin* como *scope*.

```
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.

```python
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.

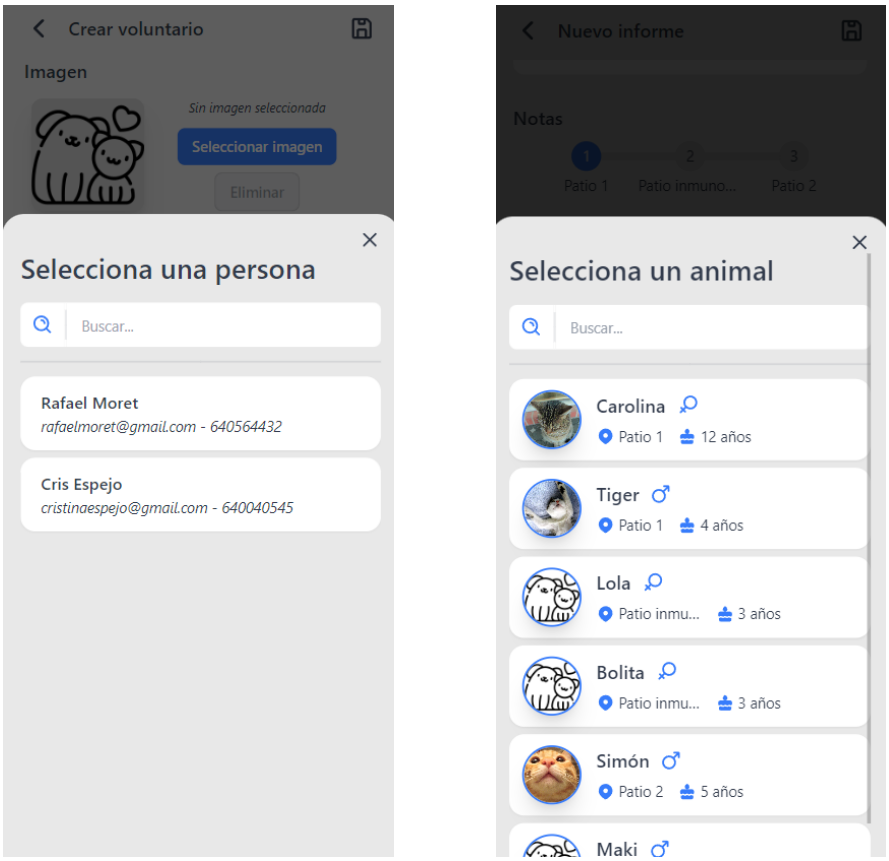


Ilustración 13. Ejemplos de ItemSelector

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.

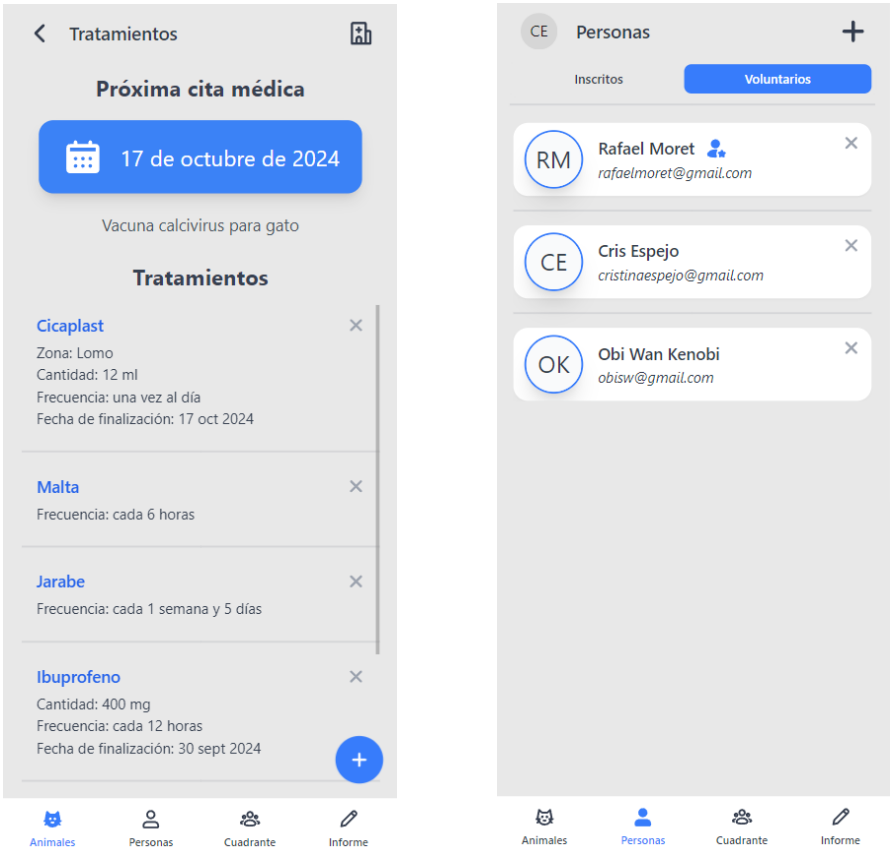


Ilustración 14. Ejemplos de ItemList

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.

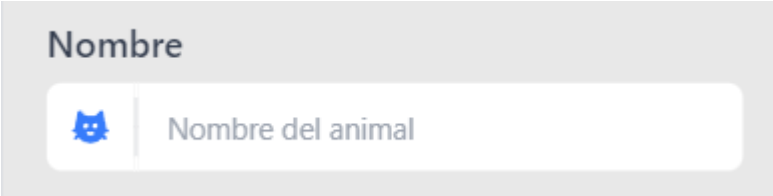


Ilustración 15. Ejemplo de TextInput

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.

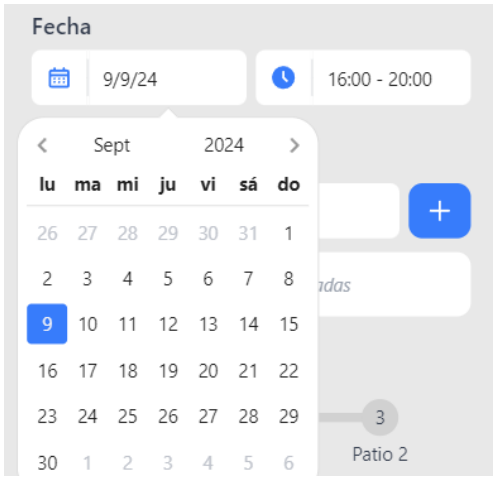


Ilustración 16. Ejemplo de DateInput

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.

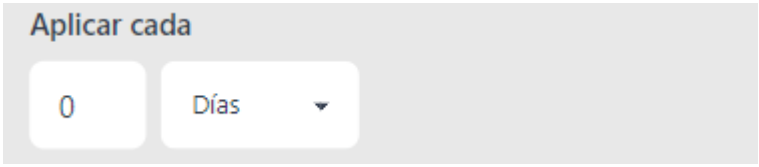


Ilustración 17. Ejemplo de TimeInput

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.

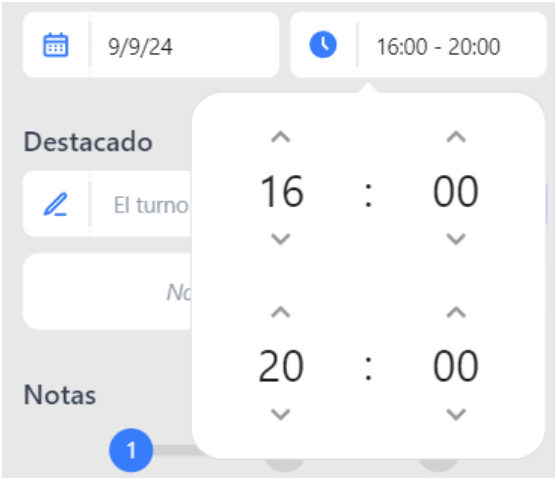


Ilustración 18. Ejemplo de HoursInput

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.

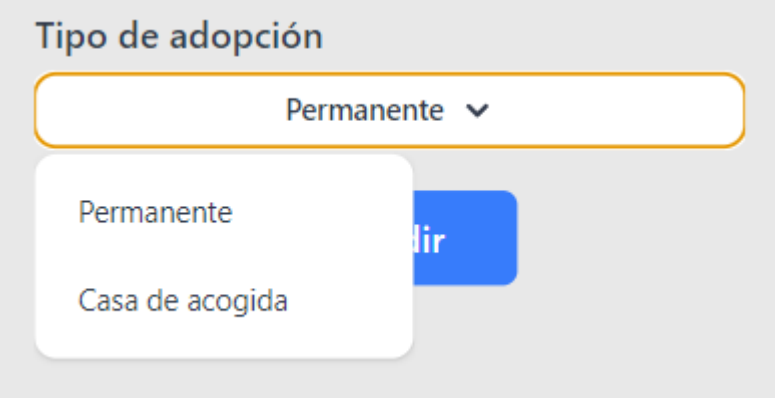


Ilustración 19. Ejemplo de DropdownSelector

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.

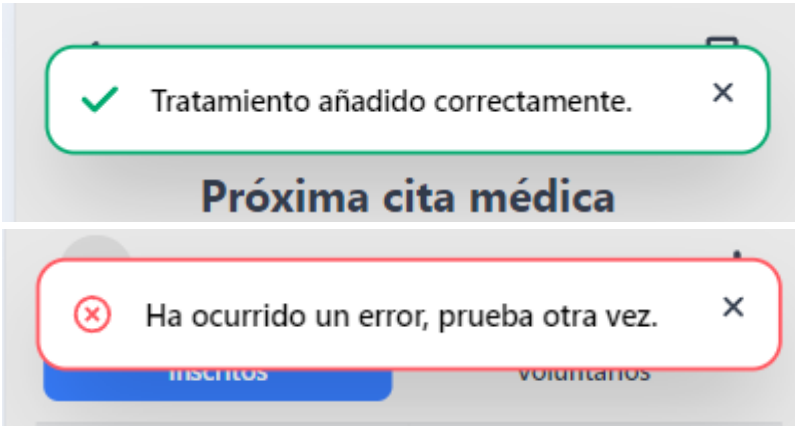


Ilustración 20. Ejemplo de notificaciones

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.

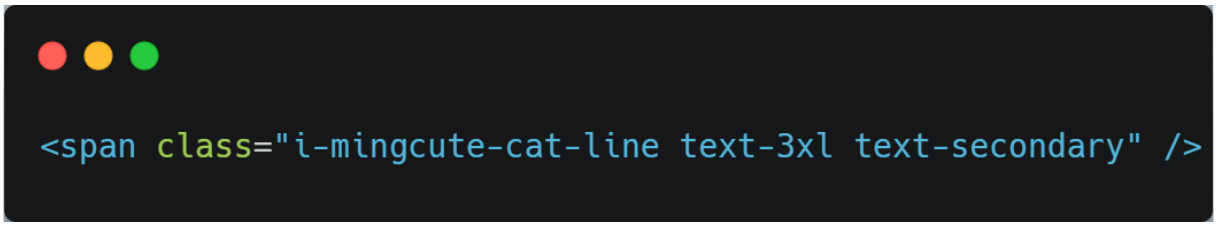


Ilustración 23. Ejemplo de uso de un icono

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 Cuadrante de turnos

La idea del módulo del cuadrante de turnos es presentar un punto donde los voluntarios puedan anotar qué días y horas son los que quieren realizar sus turnos semanales.

Actualmente, esta funcionalidad la realizan inscribiendo su nombre en un documento de Google Docs compartido, donde pueden ver en tiempo real en qué horas se están apuntando sus compañeros, lo que les ayuda a configurar la distribución de los turnos.
La distribución de los turnos es importante ya que sigue un par de normas a la hora de permitir o no que una persona se apunte a esa franja horaria:

- El turno tiene al menos un voluntario veterano en él
- El turno no alcanza el número máximo de voluntarios (dos).

Además de estas normas, es importante permitir que hasta que un voluntario veterano no se apunte en el turno, todos los voluntarios novatos que quieran puedan apuntarse en él, aunque supere el máximo, y será luego a posteriori de forma manual cuando se decida si es necesario desapuntar a alguno de ellos.

Es por esta razón, que es muy importante que nuestro módulo también contenga la funcionalidad de ver en tiempo real las actualizaciones.

Interfaz de usuario

Para la interfaz de usuario, era importante conseguir mostrar el estado actual del turno a la vez que fuera sencillo permitir al usuario apuntarse en nuevas franjas horarias.

Este fue el motivo para decidirnos por una rejilla que mostrara días en un eje y tramos horarios en el otro. Cada celda sería ciclable para permitir apuntarse en una nueva franja, a la vez que dentro te mostraría qué otros voluntarios estaban ya apuntados. Además, podríamos incluir algún comentario adicional para indicar estados especiales, como que el turno está completo o que necesita un voluntario veterano.

Ilustración 27. Vista del cuadrante de turnos

Cada uno de los botones rectangulares que permiten apuntarse a un turno es un componente reutilizable llamado *ShiftSigner*. Este componente recibe cuál es la franja horaria que representa (mañana o tarde), los usuarios que están apuntados en ella (mostrando su imagen en la parte inferior), el máximo de usuarios permitidos por turno y si el botón debe ser remarcado o no (lo que indica que nuestro usuario ha seleccionado ese turno).

Con toda esta información es con la que somos capaces de mostrar los diferentes estilos que le dan al usuario la suficiente retroalimentación para comprender en qué turnos puede apuntarse, en cuáles no, en cuáles está ya apuntado y cuáles son los turnos donde están apuntados sus compañeros.

Además, para transmitir al usuario el hecho de que lo que está viendo es en tiempo real, incluimos un contador que indica cuántos usuarios están ahora mismo conectados viendo y editando el turno. Así como un indicador del estado actual del mismo, el cual puede ser abierto o cerrado, permitiendo o no que se envíen nuevas modificaciones de este.

Por último, en barra superior tenemos un par de botones que servirán a los administradores para limpiar completamente el turno (poniendo todas las franjas vacías) y para cambiar el estado de este.

Store

En el store gestionamos todas las conexiones con los distintos websockets y las acciones adicionales que permite el backend. Es por esto por lo que tenemos las siguientes acciones y variables reactivas en él:

- **status**: se trata de la variable que contiene el estado del turno. Está ligada a la conexión del websockets, de forma que está indefinida mientras la conexión no exista y se actualiza automáticamente conforme un nuevo mensaje llega por la conexión (se actualiza el valor del turno). Cada vez que un nuevo mensaje llega, se comprueba su estructura para asegurar que cumple el contrato del *ShiftStatusAdapter*. Es utilizada en la vista para mostrar cuál es el estado, por lo que cuando es modificada la vista se altera correspondientemente.

- **shift**: igual que con la variable anterior, se trata de una variable reactiva ligada a la conexión con el (otro) websockets. Esta contiene los datos de todo el turno (días, horas y usuarios), por lo que es la variable que se actualiza cuando algún otro usuario realiza una modificación sobre el turno. Los datos que llegan a través de la conexión son comprobados en forma mediante el *ShiftAdapter*, que asegura que sigan el contrato correcto.

- **connections**: variable reactiva igual que las dos anteriores, contiene el número de usuarios conectados en este momento al websocket.

- **isConnecting**: es una variable booleana que indica el momento en el que los websockets no están completamente inicializados y se está produciendo la conexión. Sirve para mostrar retroalimentación al usuario indicando que se está conectando.

- **startConnection()**: inicializa las conexiones a los websockets que dispone el backend. Más adelante veremos por qué existen varios websockets, pero de momento nos basta con conocer que en esta operación se inician ambas conexiones a la vez, como si fuera una. En el momento en el que se producen ambas conexiones los valores de *status*, *shift* y *connections* son inicializados al valor que corresponda.
- **finishConnection()**: idéntico a la operación anterior salvo que se encarga de cerrar las conexiones en lugar de abrirlas. Tras cerrar la conexión, el resto de clientes recibirá el dato del turno con el número de conexiones abiertas actualizado.

- **sendShiftAction(type, user, shift)**: es la operación utilizada para mandar una modificación del turno al servidor. En ella debemos especificar como parámetros que tipo de operación queremos hacer y sobre qué turno y usuario. Estos datos son validados mediante el ShiftActionAdapter y posteriormente enviados al backend. En el

momento en el que la actualización se produzca, la variable *shift* es actualizada automáticamente en todos los clientes mostrando el nuevo valor.

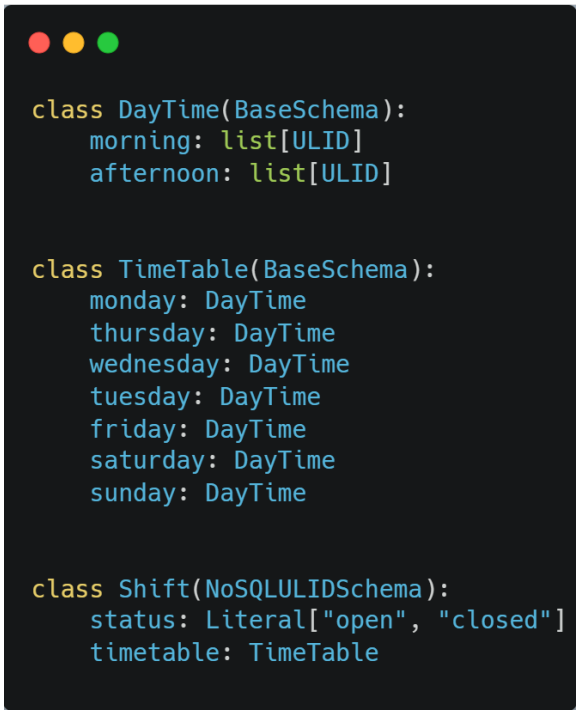
- **toggleStatus()**: invierte el estado actual del turno, de forma que lo cierra si estaba abierto y viceversa. Cuando esta modificación se produce la variable *status* es actualizada en todos los clientes automáticamente.

- **cleanShift()**: reinicia el turno a sus valores por defecto, dejándolo completamente vacío. Para esto utiliza el endpoint correspondiente del servidor. Tras la actualización, la variable *shift* es modificada por parte del websocket.

Modelo de datos

Para poder mostrar toda la información del cuadrante de turnos, necesitamos una estructura que almacene a los usuarios y los turnos que han escogido, de forma que todos puedan acceder a ella para modificarla.

Dado que la estructura de los turnos es constante (lunes, martes, mañana, tarde, ...), lo mejor es almacenar a los usuarios dentro de cada uno de ellos, en lugar de anotar para cada usuario cuál es su turno seleccionado, de esta forma conseguimos reducir el tamaño de la información que necesitamos.



```python
class DayTime(BaseSchema):
    morning: list[ULID]
    afternoon: list[ULID]


class TimeTable(BaseSchema):
    monday: DayTime
    thursday: DayTime
    wednesday: DayTime
    tuesday: DayTime
    friday: DayTime
    saturday: DayTime
    sunday: DayTime


class Shift(NoSQLULIDSchema):
    status: Literal["open", "closed"]
    timetable: TimeTable
```

Ilustración 28. Modelo de datos de cuadrante de turnos

Esta clase *Shift*, será la que conecte con los datos de la base de datos y modifique el turno cada vez que algún usuario haga una actualización.

Websockets

La dependencia principal de este módulo se encuentra en el archivo websockets.py y consiste en una clase llamada WSConnectionManager, la cual es un manager encargado de gestionar las conexiones sobre un websocket concreto, registrando nuevas conexiones en una lista de conexiones activas y permitiendo el envío individual o en difusión de mensajes a través de ellas.

```python
class WSConnectionManager:
    def __init__(self):
        self.active_connections: list[WebSocket] = []

    async def connect(self, websocket: WebSocket):
        await websocket.accept()
        self.active_connections.append(websocket)

    def disconnect(self, websocket: WebSocket):
        self.active_connections.remove(websocket)

    async def send(self, message: dict[str, Any], websocket: WebSocket):
        await websocket.send_json(message)

    async def broadcast(self, message: dict[str, Any]):
        for connection in self.active_connections:
            await connection.send_json(message)
```

Ilustración 29. Declaración de websockets

La forma en la que declaramos un websocket en nuestra API es a través de los routers de FastAPI, al igual que como se hace con los endpoints HTTP.

La ruta, al iniciarse, deberá así indicar la conexión al manager y utilizarlo para enviar mensajes. A su vez, en el caso de que se quieran recibir mensajes de la parte del cliente, el websocket debe esperar asíncronamente su recepción, ya sea de forma indefinida o para un número de mensajes establecido.

```python
status_ws_manager = WSConnectionManager()

@router.websocket("/shift/status", dependencies=[...])
async def status_ws(websocket: WebSocket):
    await status_ws_manager.connect(websocket)

    ...

    try:
        await status_ws_manager.send(..., websocket)

        while True:
            await websocket.receive_json()

    except WebSocketDisconnect:
        status_ws_manager.disconnect(websocket)
```

Ilustración 30. Implementación del websocket

Una vez terminada la conexión por cualquiera de las dos partes, se debe notificar al manager para que actualice su lista de conexiones activas.

Endpoints

WS /shift/status
Es la conexión por websocket que inician los clientes para obtener el estado del turno en tiempo real. Una vez se registra la conexión, se obtiene el dato de la base de datos y se manda devuelta al cliente que inició la conexión. Es una conexión solo de lectura, los clientes no mandan nada a través de ella, solo reciben.

WS /shift
Es la conexión por websocket que inician los clientes para mandar y recibir actualizaciones del turno. En el momento en que esta se registra, se manda a todas las conexiones (broadcast) el dato del turno actual y el número de conexiones registradas (para aumentar el contador).

A partir de ese momento la conexión queda esperando nuevos mensajes, de forma que cuando el cliente mande una nueva actualización (agregar usuario o eliminar usuario), se procede de la siguiente forma:

- Se recibe la acción
- Se obtienen y comprueban los datos del turno actual
- Se comprueba que el turno no esté cerrado, en caso afirmativo se termina la acción
- Se actualiza el turno en la base de datos en base a la acción del usuario
- Se envía a todas las conexiones (broadcast) los nuevos datos del turno para que actualicen su vista

Si en algún momento ocurre un error de conexión esta se cierra, se actualiza el número de conexiones actuales y se envía a todas las conexiones estos datos para mostrar la desconexión.

POST /shift/status
Se encarga de actualizar el estado del cuadrante de turnos entre "abierto" y "cerrado". En el momento en el que se produce la actualización se utiliza el manager del websocket del estado del turno para notificar a todas las conexiones el nuevo estado.

POST /shift/clean
Se encarga de restablecer el cuadrante, dejándolo completamente vacío. En cuanto se actualizan los datos se envía a todas las conexiones del manager del websocket de los datos del turno el nuevo estado de este, para que se actualicen al momento.

Esquemas

Los datos de entrada y salida que utilizan los endpoints de este módulo, así como los websockets, son los siguientes:

```python
class ShiftStatus(BaseSchema):
    status: Literal["open", "closed"]


class ShiftData(BaseSchema):
    connected: int
    timetable: TimeTable


class ShiftAction(BaseSchema):
    class ActionType(StrEnum):
        ADD_USER = "add_user"
        REMOVE_USER = "remove_user"

    class Selection(BaseSchema):
        day: Literal["monday", "thursday", "wednesday", "tuesday", "friday", "saturday", "sunday"]
        time: Literal["morning", "afternoon"]

    type: ActionType
    user: ULID
    shift: Selection
```

Ilustración 31. Esquema del cuadrante de turnos

Donde tenemos un simple diccionario con una propiedad *status* para actualizar el estado del turno (*ShiftStatus*), así como los datos del número de usuarios conectados más la lista de usuarios por franja horaria (ShiftData), que es lo que luego es enviado a todas las conexiones cada vez que se produce una actualización de los datos del turno.

Además, también tenemos el contrato de los datos que envía el cliente cuando quiere realizar una acción de actualización (*ShiftAction*), con los datos del usuario que actualiza, el tipo de actualización y la franja horaria donde se produce la actualización.

5.3.2 Informes

El informe de turno es un documento que sirve como registro de todo lo que ocurre durante un turno de voluntariado. Cada turno siempre debe llevar asociado un turno, que será creado por uno de los voluntarios que participaron en él.

Actualmente los informes son creados en base a una plantilla para un documento de Google Docs, con lo que suelen seguir estructuras muy diferentes entre ellos. La idea principal es unificar esta recogida de datos, estandarizarla y hacerla de fácil acceso durante el transcurso del turno para los voluntarios.

Interfaz de usuario – Histórico de informes

En cuanto a la interfaz, debíamos conseguir tres funcionalidades principales: mostrar el histórico de informes ya creados, poder consultar los datos de informes pasados y poder crear nuevos informes. Esto se tradujo en tres vistas correspondientes diferenciadas, por lo que vamos a detallar cada una de ellas.

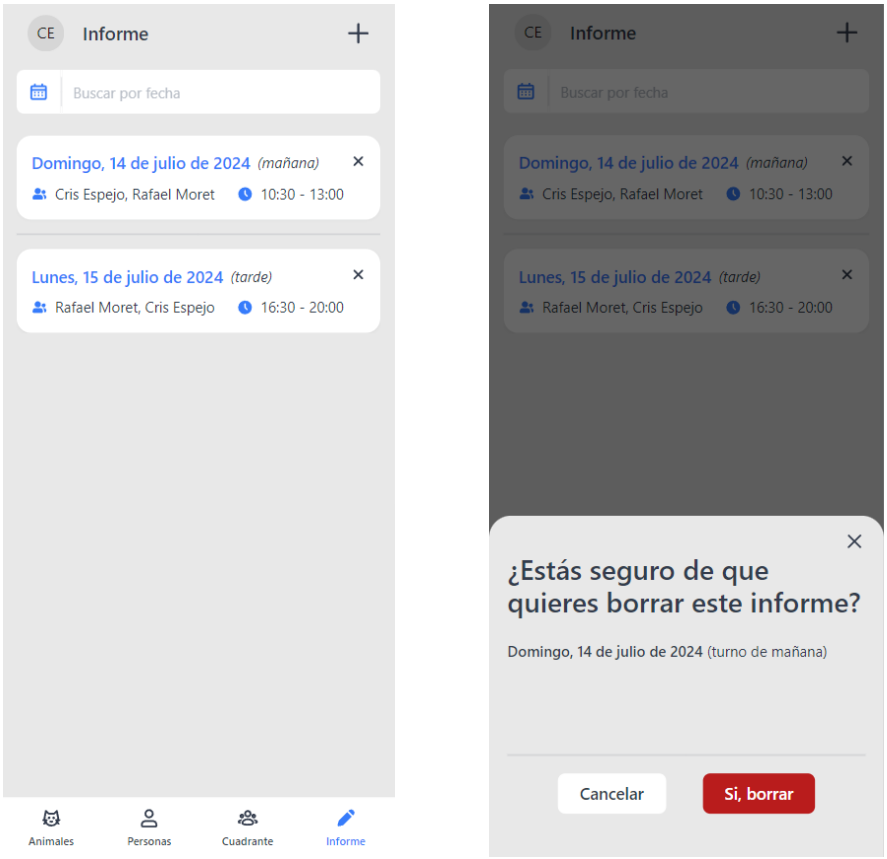


Ilustración 32. Visualización de listado y borrado de informes

Primero tenemos la vista donde listar los informes pasados. Esta vista es una vista simple que se basa en el uso del componente *ItemList,* que ya explicamos anteriormente. En ella utilizamos una tarjeta personalizada para mostrar la información más importante a la hora de distinguir diferentes informes entre ellos, que son: la fecha, los voluntarios que participaron y la hora a la que se realizó el turno.

Además, incluimos un pequeño botón en cada tarjeta que permite eliminar los informes y un buscador por fecha en la parte superior para facilitar la búsqueda de informes concretos. Este último buscador es parte del conjunto de componentes globales que hemos desarrollado, concretamente el componente *DateInput.*

Por último, en la barra superior encontramos el botón para acceder al editor de nuevos informes y si clickamos en alguna de las tarjetas de un informe podremos entrar a inspeccionar los datos concretos de un informe.

Interfaz de usuario – Editor de informes

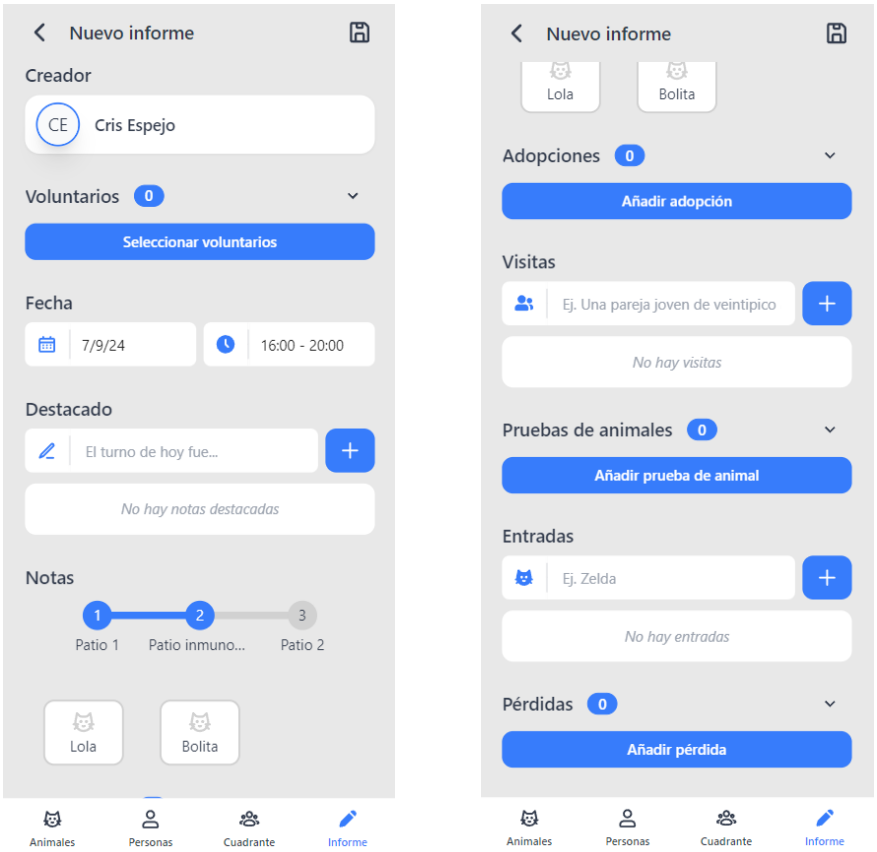


Ilustración 33. Visualización del editor de informes

Por la parte del editor de nuevos informes sí que tenemos una vista más compleja que combina muchos tipos diferentes de componentes para conseguir una experiencia de edición lo más sencilla posible, por lo que vamos a ir detallando el proceso completo de edición paso a paso.

Para comenzar, es importante destacar que es la vista la encargada de guardar e inicializar el informe, mientras que su componente principal, *InformEditor*, es el que gestiona las modificaciones de cada campo del informe. Además, debido a la diferencia que existe entre los datos del informe que se muestran y los que realmente se guardan en la base de datos, también es este componente el encargado de transformar la versión editable del informe, en una versión enriquecida que tiene datos adicionales para conseguir una representación más atractiva.
Por ejemplo, a nivel histórico la imagen de los voluntarios no es un dato que nos interese, por lo que no está incluido en los datos del informe que se almacenan, pero a la hora de representar y escoger a los usuarios sí que queremos mostrar su imagen.

El primer campo que observamos en el editor es el voluntario creador del informe. Este campo no es editable, ya que se obtiene el dato del usuario que tenga la sesión iniciada en ese momento.
Justo a continuación encontramos la lista del resto de voluntarios que participaron en el informe, donde podremos seleccionar varios de ellos en una nueva ventana desplegable que aparece cuando pulsamos "Seleccionar voluntarios" y se mostrarán los escogidos en modo lista.

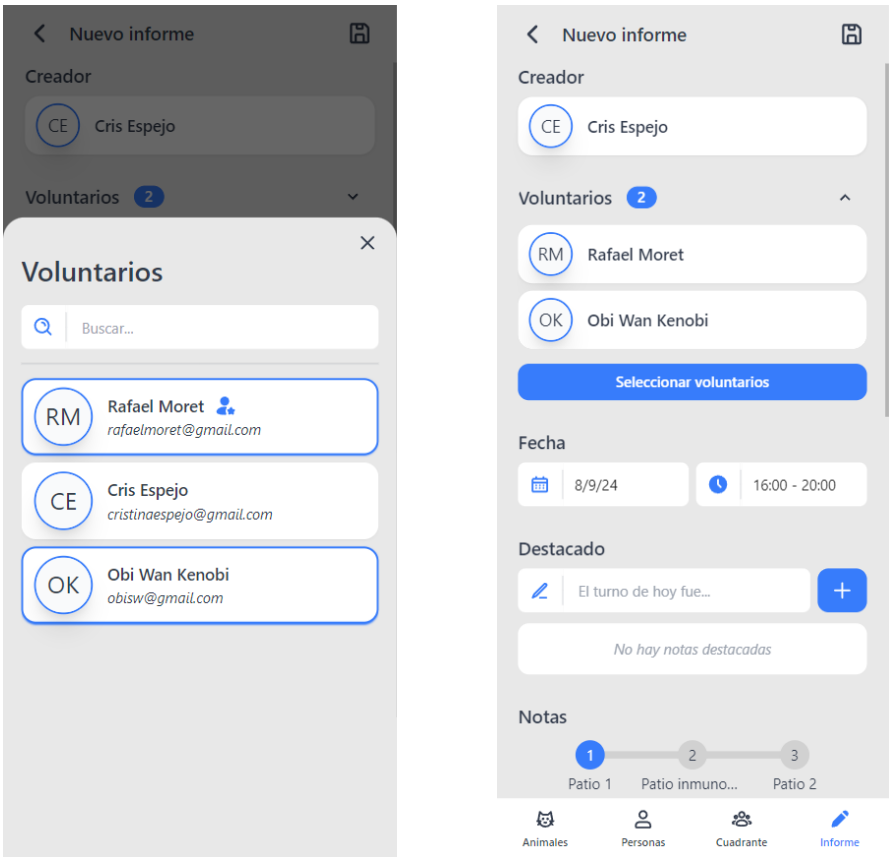


Ilustración 34. Visualización de creación de informe

Este componente selector que nos sirve para escoger voluntarios forma parte de los componentes globales que hemos desarrollado, ya que nos servirá para seleccionar diferentes tipos de elementos en múltiples contextos. Su nombre es *ItemSelector* y como entradas principales solo requiere la lista de opciones, las opciones seleccionadas y algunas configuraciones adicionales como si incluir el buscador por texto o el número máximo de elementos a seleccionar. Además, deberemos pasarle un componente que sea la representación que queremos utilizar para los elementos seleccionables (en este caso, una tarjeta para mostrar los datos de usuario).

```
<ItemSelector
  :items="userList"
  :model-value="enrichedInform.volunteers"
  :include-search="true"
  :search-fn="
    (user) =>
      `${user.person.name} ${user.person.firstSurname} ${user.person.secondSurname ?? ''}`
  "
  @update:model-value="
    (selectedVolunteers: UserInfo[]) =>
      handleFieldUpdate('volunteers', selectedVolunteers)
  "
>
  <template #item="{ item, isSelected, toggleSelect }">
    <UserCard
      size="regular"
      :user="item"
      :class="[
        'border-2',
        isSelected ? 'border-secondary shadow-secondary' : 'border-transparent'
      ]"
      @click="toggleSelect(item)"
    />
  </template>
</ItemSelector>
```

Ilustración 35. Implementación del listado de informes

A continuación, podremos indicar la fecha y horas del turno realizado, los cuales serán escogidos a través de dos componentes globales desarrollados (*DateInput* y *HourInput*) para ser reutilizables, que mostrarán al ser clicados un desplegable con información para escoger el día o rango horario deseado.

Después de la fecha, tenemos la sección de notas destacadas, un componente denominado *TextListInput* que sirve para añadir y mostrar múltiples textos en formato lista. Además, también permite la eliminación de cada uno de estos textos de forma individual. Esto nos servirá para anotar cualquier comentario general que sea importante remarcar al principio del informe, ya que tras esto podremos añadir otros comentarios mucho más específicos que se asocien a cada animal.

Para hacer esto usaremos la sección bajo el nombre de "Notas", que utiliza el componente *AnimalNotesEditor*, donde se nos muestran todos los patios de la Protectora (ordenados según el orden de patios) para que vayamos uno por uno seleccionando los animales sobre los que queramos dejar un comentario.

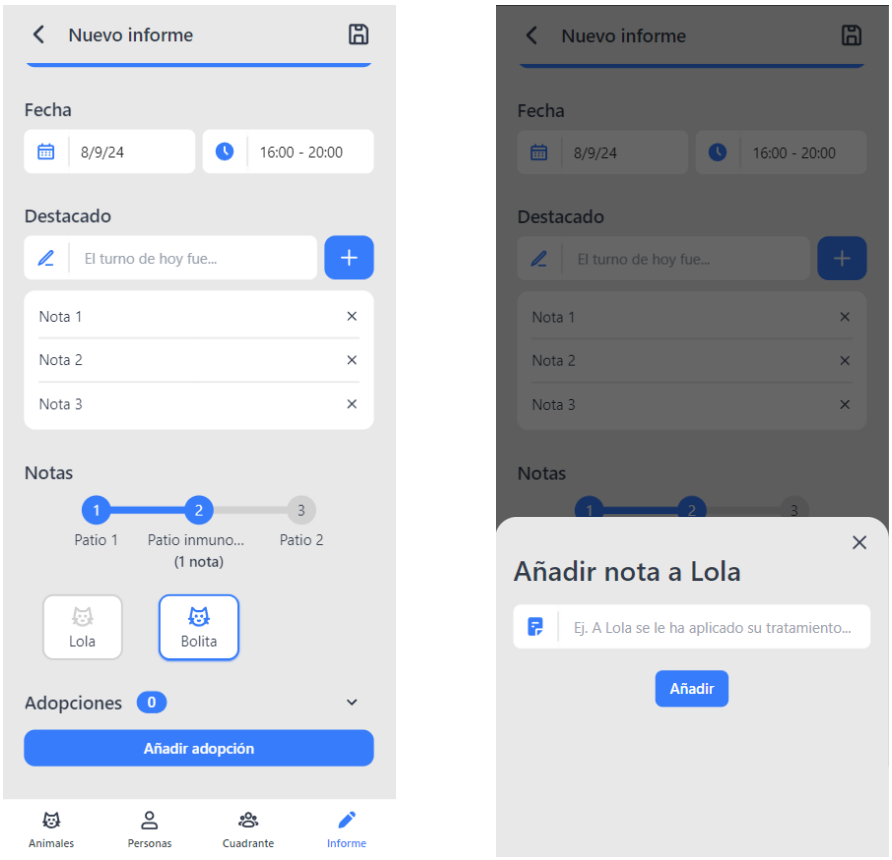


Ilustración 36. Visualización de añadir notas en el informe

Una vez que seleccionemos el animal sobre el que dejar un comentario se nos abrirá un nuevo menú para poder escribir lo que queramos, de forma que cuando terminemos el botón del animal quedará marcado en azul para indicar que ya contiene una nota.

La idea de este componente es facilitar el seguimiento del turno, de forma que se pueda ver el orden de patios actual (actualizado en base al dato almacenado en la base de datos) que se debe seguir, a la vez que ir añadiendo comentarios sobre los animales que pertenecen a cada patio.

A partir de este momento tenemos dos tipos de secciones basándonos en la funcionalidad de sus componentes. Por un lado, tenemos las secciones de selección de animal: Adopciones, Pruebas de animales y Pérdidas; y por otro lado tenemos las secciones de campo de texto doble: Visitas y Entradas.

El primer tipo consiste en aquellas secciones donde se selecciona un animal para marcar un evento concreto sobre él (que ha sido adoptado, que ha sido probado su comportamiento con otros animales o que ha fallecido). Para ello se utiliza el mismo componente *AnimalSelector* en los tres, con la diferencia de que los dos primeros añaden además un campo de selección de opciones para indicar un dato extra (el tipo de adopción o si la prueba del animal ha sido positiva o negativa).

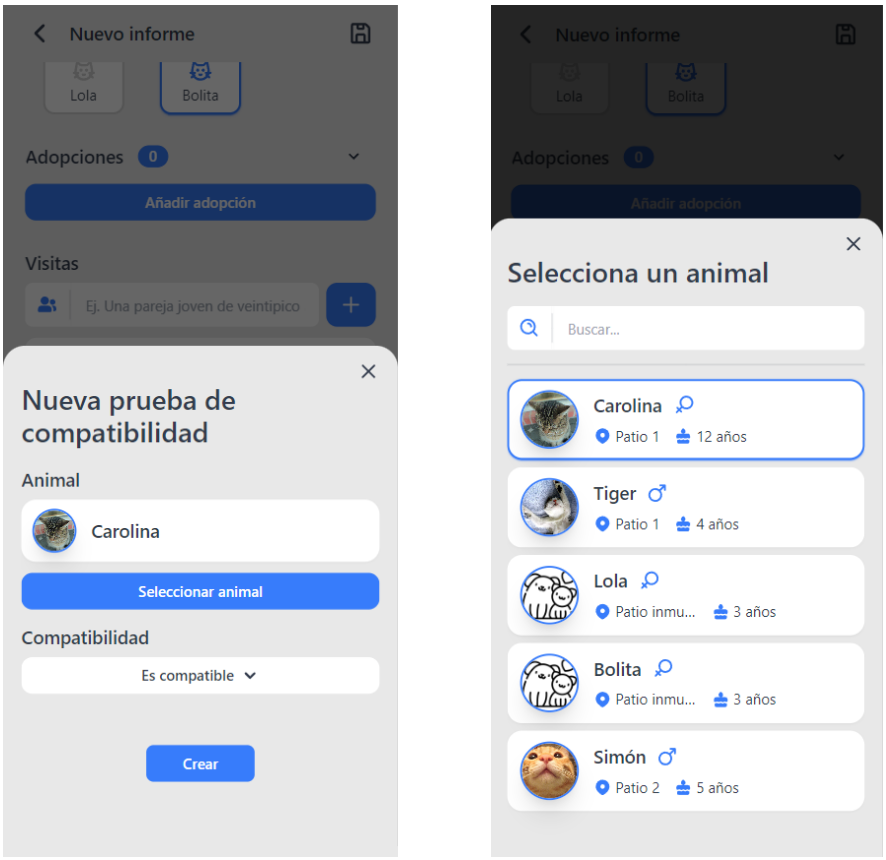


Ilustración 37. Visualización de prueba de compatibilidad en informe

Este *AnimalSelector* no es más que un botón para abrir un menú adicional con el componente *ItemSelector* para seleccionar un animal, similar al que hemos tratado antes en la sección de voluntarios, y un espacio para mostrar el animal seleccionado (si es que se ha seleccionado alguno).
Por otro lado, el componente para seleccionar las opciones es un *DropdownSelector*, que al momento de clicar abre un menú desplegable con las opciones que se pueden seleccionar.

En el segundo grupo de secciones tenemos las de campo de texto doble, que consisten en un editor de notas similar al que hemos visto para las notas destacadas, salvo porque en este al momento de añadir el primer texto se abre un segundo menú que permite añadir información en un campo de texto adicional (por ejemplo, la descripción de la visita o la descripción del nuevo animal que acaba de entrar).

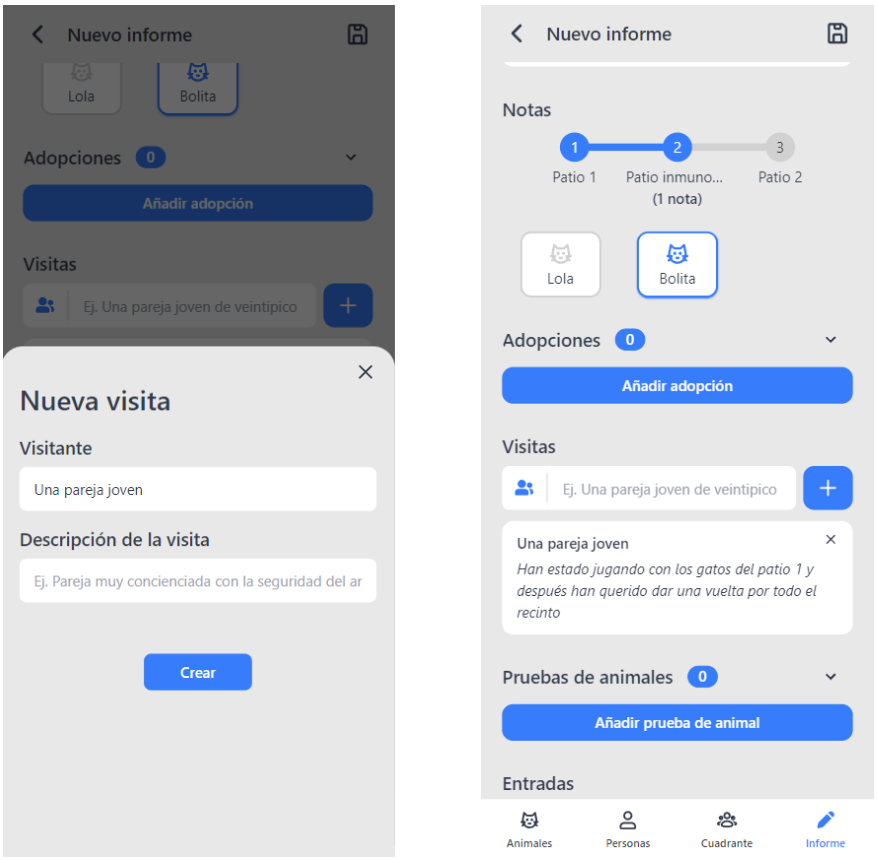


Ilustración 38. Visualización de añadir visita al informe

De esta forma se puede ampliar la información de eventos que pueden requerir algo más que simplemente un solo texto. Este componente lo hemos denominado *MultipleTextListInput*, y también está pensado para poder ser reutilizado en otros lugares de la aplicación.

Interfaz de usuario – Consulta de informes

Para poder consultar los datos de un informe concreto queríamos que la interfaz fuese lo más parecida al editor de nuevos informes, de forma que sea fácilmente reconocible cada uno de los campos.

Por esta razón, la idea en esta vista es la de mantener la misma estructura, pero utilizando versiones no editables de los componentes, de forma que fue necesario modificar la mayoría de los componentes de la vista utilizados para recopilar datos para incluir una nueva propiedad booleana que active o desactive la edición en el componente.

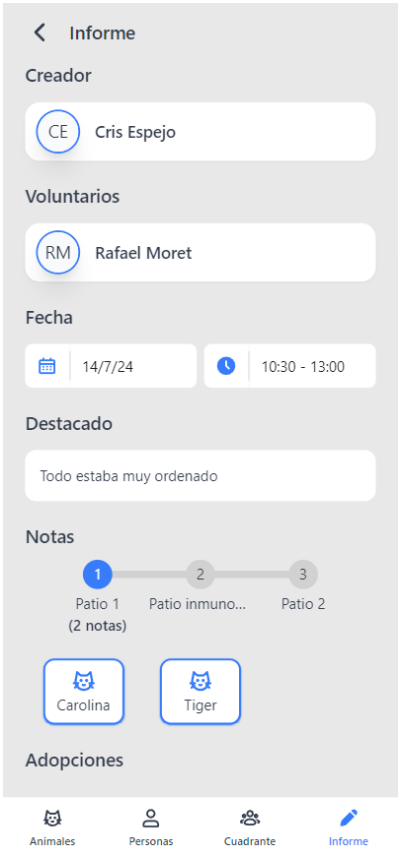


Ilustración 39. Visualización del visor de informe

Store

Pese a la complejidad de las vistas que hemos detallado (especialmente la del editor), realmente las operaciones que se efectúan sobre el backend por parte de este módulo son muy sencillas, de ahí que el store que maneja estos datos y operaciones también lo sea. Tenemos:

- ***fetchInforms()***: es la operación encargada de recuperar todos los informes del backend y asignarlos a la variable reactiva *informList*. En el proceso de recuperación, usaremos la función auxiliar *authFetch* (así como en el resto de las operaciones) para realizar una llamada autenticada al backend y aplicaremos el adaptador *InformInfoAdapter* para asegurarnos de los datos llegan en el formato adecuado.

- ***fetchInform(id)***: funciona de manera similar a la operación anterior, salvo que se encarga de recuperar los datos de un solo informe que especificaremos a través de su parámetro *id*. Estos datos deben cumplir el contrato especificado en el adaptador *InformAdapter* (basado en *InformInfoAdapter* pero extendido con datos extra) y será almacenados en la variable *informDetails*.

- ***createInform(inform)***: dados los datos de un nuevo informe como parámetro, hace la llamada al backend que se encarga de crearlo. Antes de esto comprueba que los datos introducidos cumplan el contrato necesario, *EditInformAdapter*. Esta operación se utiliza al hacer clic en el botón de guardar dentro del editor de nuevos informes.

- ***deleteInform(id)***: se encarga de hacer la llamada para que el backend elimine de la base de datos un informe. Este se utiliza en el botón de eliminar informe que aparece en las tarjetas de la vista del histórico de informes.

Modelo de datos

Para el modelo de datos, era importante que los datos que se almacenen en la base de datos fueran independientes del resto del estado de la base de datos SQL, de forma que, si en algún momento se pierden o editan datos, el histórico de los informes permanece estático. Es por esta razón por la que no solo se almacenan los identificadores de otras entidades, como Usuario o Animal, sino que además se guardan otros detalles que puedan servir para mostrar la información mínima de la entidad en caso de que esta sea eliminada e irrecuperable de la base de datos SQL.

```python
class Inform(NoSQLULIDSchema):
    class User(ULIDSchema):
        class Person(BaseSchema):
            name: str
            first_surname: str
            second_surname: Optional[str] = None

        person: Person

    class Animal(ULIDSchema):
        name: str

    class Note(BaseSchema):
        class Yard(ULIDSchema):
            name: str

        yard: Optional[Yard] = None
        animal: "Inform.Animal"
        text: str = Field(min_length=1)

    class Visit(BaseSchema):
        visitor: str = Field(min_length=1)
        description: str = Field(min_length=1)

    class Arrival(BaseSchema):
        name: str = Field(min_length=1)
        description: Optional[str] = Field(default=None, min_length=1)

    class Loss(BaseSchema):
        animal: "Inform.Animal"

    class Adoption(BaseSchema):
        animal: "Inform.Animal"
        foster: bool

    class TestedAnimal(BaseSchema):
        animal: "Inform.Animal"
        compatible: bool

    class TimeRange(UTCSchema, BaseSchema):
        start: time
        end: time

    creator: User
    volunteers: list[User]
    date: date
    time_range: TimeRange
    notes: list[Note]
    highlights: Optional[list[str]] = None
    visits: Optional[list[Visit]] = None
    arrivals: Optional[list[Arrival]] = None
    losses: Optional[list[Loss]] = None
    adoptions: Optional[list[Adoption]] = None
    tested_animals: Optional[list[TestedAnimal]] = None
```

Ilustración 40. Modelo de datos de informes

Endpoints

GET /inform/search
Se encarga de listar el histórico de informes.

POST /inform
Crea e inserta un nuevo informe en la base de datos.

GET /inform/{id}
Se encarga de devolver los datos completos de un informe concreto.

PUT /inform/{id}
Se encarga de la actualización de los informes. Primero comprueba que el identificador que se le pasa corresponde a un informe previamente guardado y después actualiza los datos de este.

DELETE /inform/{id}
Elimina el informe correspondiente al identificador que se le pasa después de comprobar que existe.

Esquemas

Para este módulo tenemos tres esquemas que se utilizan en los distintos endpoints anteriores.

ListedInform, utilizado en el momento de listar el histórico de informes, ya que es el menos detallado.

```
class ListedInform(ULIDSchema):
    class User(ULIDSchema):
        class Person(BaseSchema):
            name: str
            first_surname: str
            second_surname: str | None

        person: Person

    class TimeRange(UTCSchema, BaseSchema):
        start: time
        end: time

    creator: User
    volunteers: list[User]
    date: date
    time_range: TimeRange
```

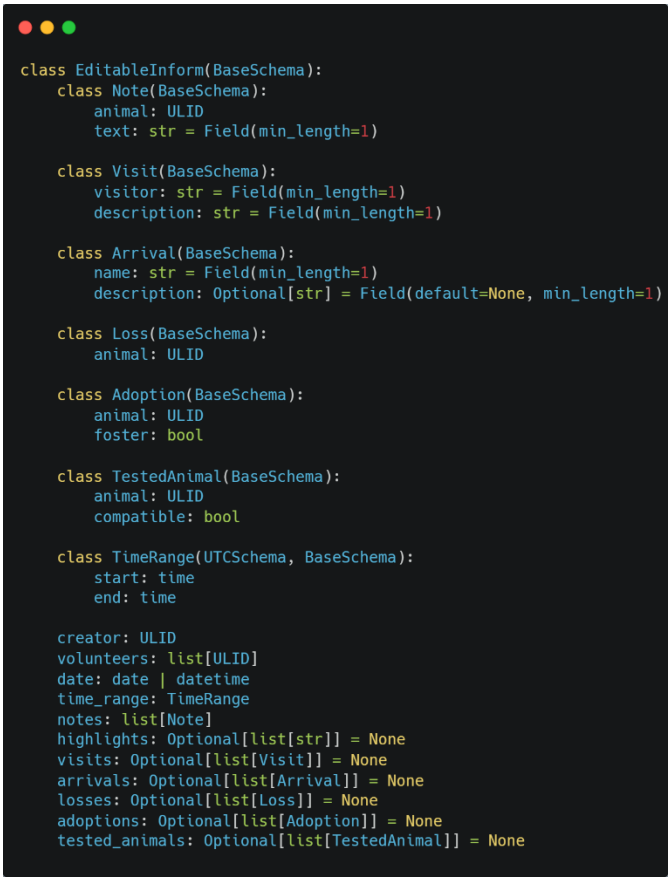
Ilustración 41. Esquema ListedInform

CompleteInform, que consiste en una versión extendida del anterior. Se utiliza como respuesta a todos los endpoints que realizan una operación sobre un informe concreto (menos el borrado).

```python
class CompleteInform(ListedInform):
    class Animal(ULIDSchema):
        name: str

    class Note(BaseSchema):
        class Yard(ULIDSchema):
            name: str

        yard: Optional[Yard] = None
        animal: "CompleteInform.Animal"
        text: str = Field(min_length=1)

    class Visit(BaseSchema):
        visitor: str = Field(min_length=1)
        description: str = Field(min_length=1)

    class Arrival(BaseSchema):
        name: str = Field(min_length=1)
        description: Optional[str] = Field(default=None, min_length=1)

    class Loss(BaseSchema):
        animal: "CompleteInform.Animal"

    class Adoption(BaseSchema):
        animal: "CompleteInform.Animal"
        foster: bool

    class TestedAnimal(BaseSchema):
        animal: "CompleteInform.Animal"
        compatible: bool

    notes: list[Note]
    highlights: Optional[list[str]] = None
    visits: Optional[list[Visit]] = None
    arrivals: Optional[list[Arrival]] = None
    losses: Optional[list[Loss]] = None
    adoptions: Optional[list[Adoption]] = None
    tested_animals: Optional[list[TestedAnimal]] = None
```

Ilustración 42. Esquema CompleteInform

Por último, tenemos el esquema editable (*EditableInform*), que se utiliza como contrato para recopilar los datos que necesitamos para crear o actualizar un informe.



```python
class EditableInform(BaseSchema):
    class Note(BaseSchema):
        animal: ULID
        text: str = Field(min_length=1)

    class Visit(BaseSchema):
        visitor: str = Field(min_length=1)
        description: str = Field(min_length=1)

    class Arrival(BaseSchema):
        name: str = Field(min_length=1)
        description: Optional[str] = Field(default=None, min_length=1)

    class Loss(BaseSchema):
        animal: ULID

    class Adoption(BaseSchema):
        animal: ULID
        foster: bool

    class TestedAnimal(BaseSchema):
        animal: ULID
        compatible: bool

    class TimeRange(UTCSchema, BaseSchema):
        start: time
        end: time

    creator: ULID
    volunteers: list[ULID]
    date: date | datetime
    time_range: TimeRange
    notes: list[Note]
    highlights: Optional[list[str]] = None
    visits: Optional[list[Visit]] = None
    arrivals: Optional[list[Arrival]] = None
    losses: Optional[list[Loss]] = None
    adoptions: Optional[list[Adoption]] = None
    tested_animals: Optional[list[TestedAnimal]] = None
```

Ilustración 43. Esquema EditableInform

5.3.3 Personas

El propósito de este módulo es registrar los perfiles de personas que se registran en la aplicación. La interfaz de persona registra los datos personales asociados a ellas, como su nombre, apellidos, mail y teléfono. Mantener estos datos es esencial para facilitar la comunicación, gestionar el involucramiento en la protectora y realizar consultas posteriores, asegurando así una administración eficiente de las relaciones y actividades dentro de la aplicación.

Interfaz de usuario

Es necesario tener un listado completo de todas las personas pertenecientes a la aplicación, mostrando todos sus datos. En esencia, necesitamos datos básicos de identificación (nombre y apellidos) y datos de contacto, dotados con un mail y un teléfono. La interfaz cuenta con un creador de personas y un editor, teniendo estos la misma interfaz gráfica.

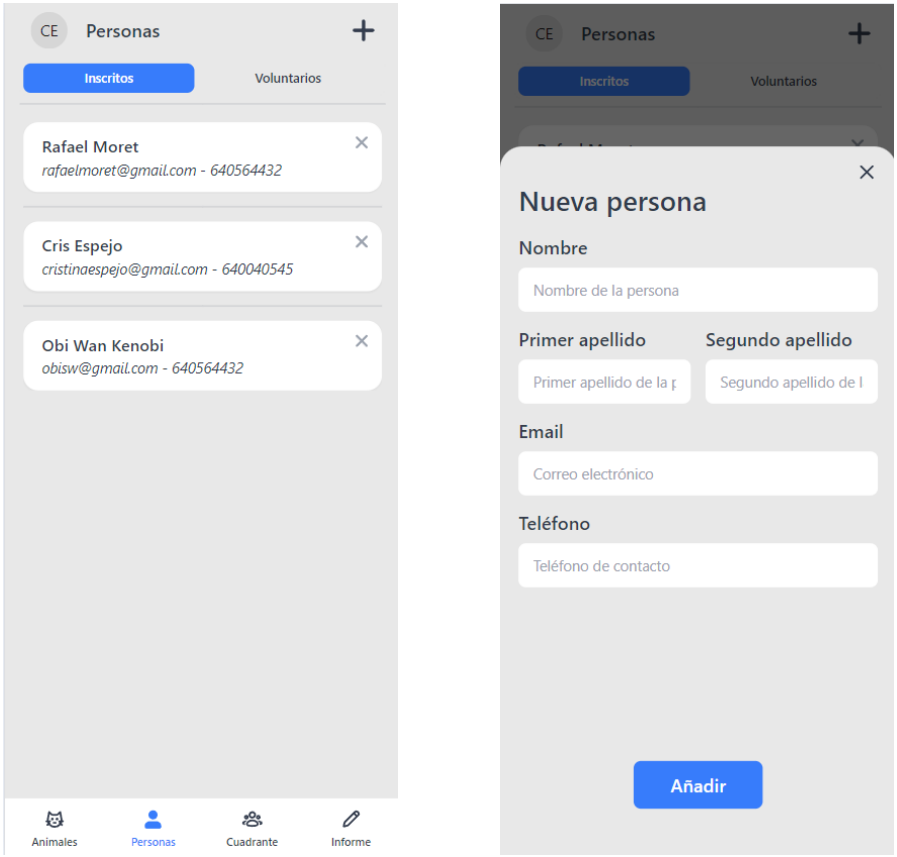


Ilustración 44. Visualización de listado y creación de personas

Este desarrollo del listado se permite utilizando el componente PersonList mencionados anteriormente, un subtipo de **ItemList** específico para mostrar datos de la persona en formato de cards con el componente **PersonCard.** Este, además de mostrar los datos, gestiona la eliminación de la persona concreta seleccionada para su eliminación.

La vista principal, además de gestionar la entrada y salida de datos con el uso de su store, permite la creación y edición de personas.
La creación se realiza mediante el modal creado con **BottomDrawer** que permite introducir en una nueva interfaz los datos de creación de personas, todos ellos utilizando el componente **PersonEditor** el cual gestiona toda la vista de introducción de datos con **TextInput**.
La edición de estas se realiza haciendo clic sobre la persona concreta a editar, desplegando de la misma forma el modal de creación de personas.

Store

Para la gestión de datos de personas, se utilizan las siguientes funciones:

- **fetchPeople ():** se encarga de obtener el listado de personas de la base de datos 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 personas. Una vez que recibe la respuesta, la convierte a formato JSON y la procesa utilizando el adaptador *PersonAdapter* para transformar los datos según el formato requerido. El resultado, que es una lista de objetos, se asigna al estado *personList*.

- **fetchPerson (id: string):** se encarga de obtener los detalles de una persona específica desde la API. Para ello se realiza una petición GET a la API usando *authFetch*, con la URL construida a partir del endpoint para personas y el identificador de la persona requerida. Una vez recibida la respuesta, la convierte a formato JSON y la procesa mediante el adaptador *PersonAdapter*, que transforma los datos en el formato adecuado. Los detalles de la persona adaptados se asignan al estado *personDetails*.

- **deletePerson (personId: string):** se encarga de eliminar una persona de la lista de personas cargadas en la aplicación. Primero, verifica si la lista de personas (*personList*) está disponible y contiene el identificador de la persona que se desea eliminar. Si la persona está en la lista, se realiza una solicitud DELETE al servidor utilizando *authFetch*, apuntando a la URL específica para la eliminación de esa persona (crudPersonApi/{personId}). En caso contrario, lanza un error. Después de la eliminación, se actualiza *personList* para que excluya la persona eliminada, filtrando la lista para que ya no contenga el *personId* especificado.

- **createPerson (person: EditPerson):** se encarga de añadir una nueva persona a la base de datos de la aplicación. Primero, realiza una solicitud POST al servidor utilizando *authFetch*, enviando los datos de la nueva persona en formato JSON. La solicitud se dirige a la URL específica para la creación de personas (crudPersonApi). Si la respuesta del servidor indica algún error, se lanza una excepción con un mensaje de error que

incluye la respuesta del backend. Una vez que la solicitud se completa sin errores, la función llama a *fetchPeople* para actualizar la lista de personas en la aplicación, asegurando que la nueva persona se refleje en la interfaz.

- **updatePerson (personId: string, person: EditPerson):** se encarga de actualizar la información de una persona existente en la base de datos. Para ello, realiza una solicitud PUT al servidor, enviando los datos actualizados de la persona en formato JSON. La solicitud se dirige a la URL específica para la actualización de una persona (crudPersonApi/${personId}). Una vez completada la actualización sin errores, la función llama a *fetchPeople* para refrescar y actualizar la lista de personas en la aplicación, asegurando que los cambios se reflejen en la interfaz de usuario. En caso de error, se lanza una excepción.

Modelo de datos

El modelo de datos de persona está diseñado para encapsular la información básica de una persona dentro del sistema y facilitar su integración con otras entidades. Además de almacenar los datos de estas, se definen dos relaciones clave:

1. **Relación con Usuario**: Esta relación opcional permite vincular a la persona con una cuenta de usuario, facilitando la gestión de autenticación y permisos. Al ser usuario un tipo de persona perteneciente a la aplicación, los datos personales del voluntario y los datos de su cuenta se guardan en modelos de datos distintos pero relacionados entre sí.

2. **Relación con Adopción**: La relación con este modelo permite rastrear todas las adopciones realizadas por la persona. Esto proporciona una forma eficiente de acceder a un historial de adopciones y gestionar información relacionada con los procesos de la adopción.

```python
class Person(SQLULIDSchema, table=True):
    name: str
    first_surname: str
    email: str
    phone: str
    second_surname: str | None = Field(default=None)

    user: Optional["User"] = Relationship(back_populates="person")
    adoptions: list["Adoption"] = Relationship(back_populates="person")
```

Ilustración 45. Modelo de datos de Persona

Endpoints

GET /person/search

Permite obtener el listado de personas almacenadas en la base de datos.

POST /person

Permite crear y guardar los datos de una persona.

GET /person/{id}
Permite obtener los datos de una persona concreta.

PUT /person/{id}

Permite actualizar los datos de una persona concreta.

DELETE /person/{id}

Permite el borrado de una persona concreta.

Esquemas

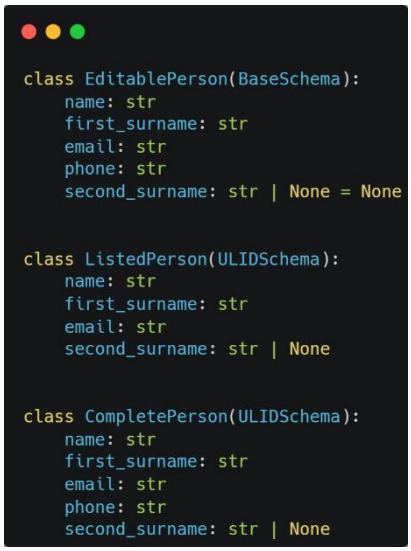
Estos esquemas definen diferentes vistas de la información de una persona en el sistema:

- **EditablePerson**: Diseñado para gestionar la edición de los datos de una persona. Incluye campos esenciales para actualizar la información de contacto y nombres, con un enfoque en permitir modificaciones.

- **ListedPerson**: Proporciona una vista simplificada de la persona, adecuada para mostrar en listas o resúmenes. Omite algunos detalles, como el número de teléfono, enfocándose en la información básica.

- **CompletePerson**: Ofrece una vista detallada y completa de la persona, incluyendo todos los campos relevantes para una presentación exhaustiva de la información.

Cada esquema se adapta a diferentes necesidades de visualización y edición dentro del sistema.



```python
class EditablePerson(BaseSchema):
    name: str
    first_surname: str
    email: str
    phone: str
    second_surname: str | None = None


class ListedPerson(ULIDSchema):
    name: str
    first_surname: str
    email: str
    second_surname: str | None


class CompletePerson(ULIDSchema):
    name: str
    first_surname: str
    email: str
    phone: str
    second_surname: str | None
```

Ilustración 46. Esquemas de Persona

5.3.4 Usuarios

El módulo de Usuarios es el encargado de gestionar los miembros voluntarios de la Protectora que tienen acceso a la aplicación.

Interfaz de usuario - Lista de usuarios

Dentro del menú de Personas de la barra de navegación, en la segunda pestaña que encontramos, llamada Voluntarios, tendremos la vista con la que poder consultar la lista completa de usuarios (voluntarios) registrados en la aplicación.

Para construir esta lista hemos utilizado de nuevo el componente *ItemList*, combinado con la tarjeta de usuario que permite mostrar su imagen, nombre, veteranía y correo electrónico.

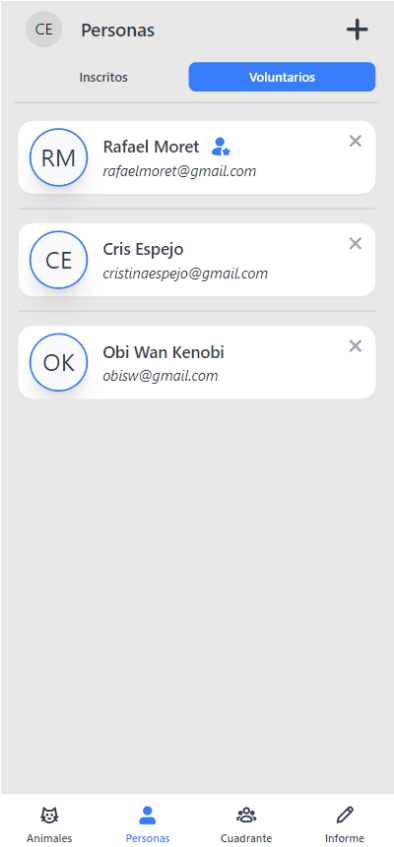


Ilustración 47. Visualización del listado de personas

En esta vista tenemos diferentes interacciones que nos permiten: editar al usuario, clicando en la tarjeta de aquel que queramos modificar; eliminar un usuario, utilizando el botón con forma de cruz en la tarjeta; y crear un nuevo usuario, usando el botón con el símbolo de más en la barra superior.

Interfaz de usuario - Editor de usuarios

Al entrar a modificar los datos de un usuario nos encontraremos con la siguiente vista. En ella, pese a que parece estar ocurriendo una sola edición, la del usuario, realmente están ocurriendo dos diferentes: la edición del usuario y la edición de la persona del usuario.

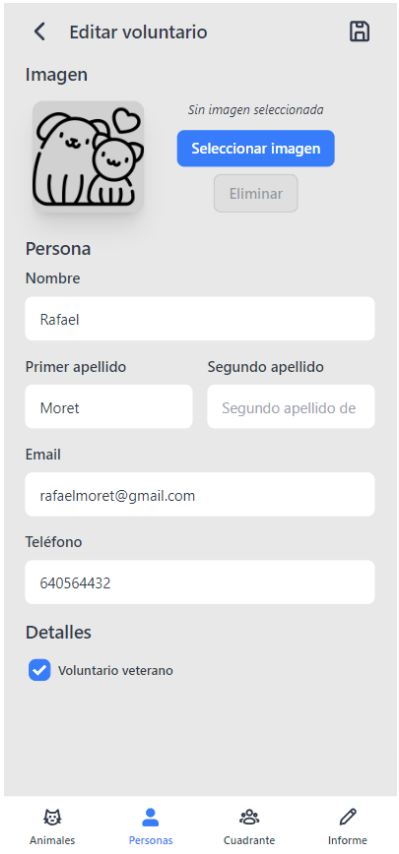


Ilustración 48. Visualización de edición de voluntarios

De esta forma, podemos editar los datos del usuario como la imagen o la veteranía, pero también podemos indagar a modificar los datos de la persona que identifica al usuario, como su nombre, email o teléfono.
Es importante destacar esto ya que, de lo contrario, si solo editáramos el usuario podríamos haber asignado una nueva persona al usuario, pero no modificar sus datos, caso de uso que para nosotros no tiene mucho sentido.

Al momento de pulsar el botón de guardado, ambas modificaciones son enviadas al servidor para ser persistidas.

Interfaz de usuario - Creación de nuevos usuarios

En este caso, tenemos prácticamente la misma vista anterior con los datos recién inicializados, pero con la diferencia de que ahora no podemos crear una persona de cero de forma independiente y autoasignarla al usuario, debemos asignarle directamente una persona ya existente, lo que significa que previamente deberemos haber creado dicha persona.

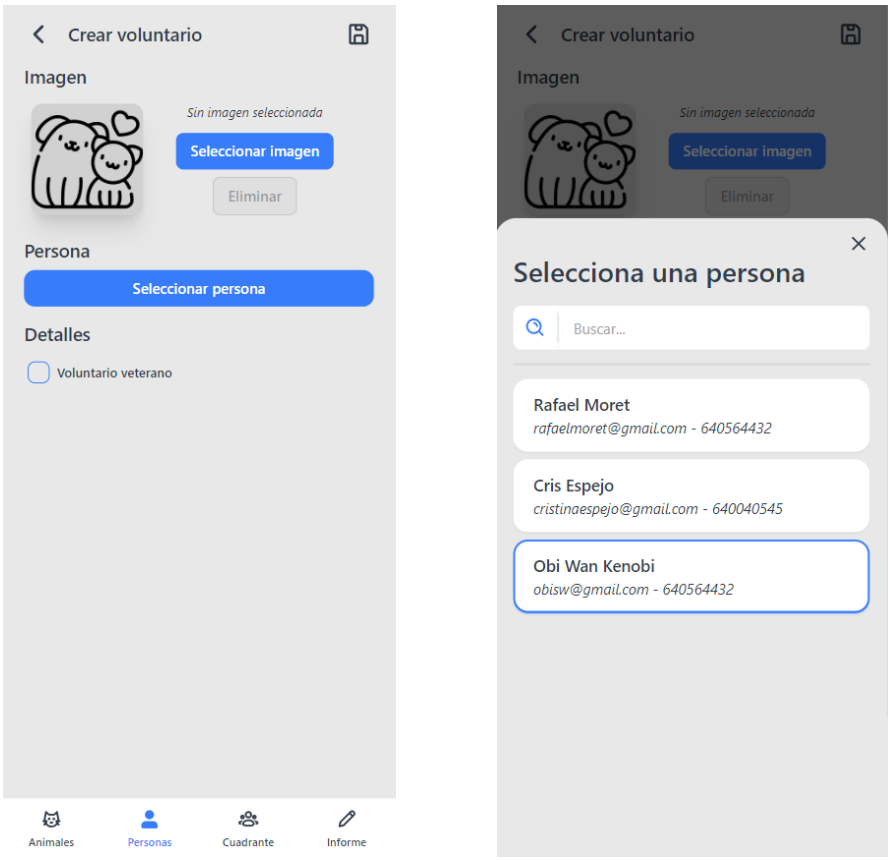


Ilustración 49. Visualización de creación de voluntarios

Esto es así debido a que tenemos dos casos de uso para la creación de un usuario.
Por un lado, la persona que quiere ser voluntaria por primera vez, sin haber tenido contacto con la Protectora anteriormente, en cuyo caso no tendremos dato alguno de ella y por tanto deberemos tomarlos para crear su persona.

Mientras que por otro lado, también tenemos a la persona que ya ha adoptado previamente a un animal en la Protectora, por lo tanto está registrada, y ahora quiere pasar a formar parte del equipo de voluntarios. En este caso los datos de la persona ya están tomados y ella está registrada, con lo que solo debemos asignar esta persona al nuevo usuario.

Como la idea es simplificar la interfaz para que sea lo más sencilla posible, decidimos optar porque el creador de usuario solo permita asignar personas (en lugar de crearlas) para así evitar tener que indicar en la interfaz las dos opciones.

Store

Para gestionar todas estas interacciones, el store del módulo tiene las siguientes acciones:

- **fetchUsers()**: es la función encarga de obtener los datos de todos los usuarios registras. Una vez adquiridos, se comprueba que tengan la forma correcta con respecto al contrato de *UserInfoAdapter* y se asignan a la variable reactiva *userList*.

- **fetchUser(id)**: realiza la misma funcionalidad que la función anterior, pero para un solo usuario concreto. Los datos que obtiene son más extensos debido a que siguen el contrato de *UserAdapter*. El resultado de los datos es luego asignado a la variable reactiva *userDetails*.

- **deleteUser(id)**: se encarga de hacer la llamada al backend para eliminar a un usuario concreto.

- **createUser(user)**: dado que los endpoints para actualizar la imagen del usuario y el resto de los datos son diferentes, en esta acción, después de comprobar que los datos cumplen el contrato correcto (EditableUser), primero se realiza la llamada al backend para crear al usuario (sin imagen) y luego se realiza una segunda llamanda que asigna la nueva imagen al usuario. Para este segundo paso, lo necesario es obtener la imagen (blob) para la que el componente de *ImagePicker* ha creado una URL, asignarla en los datos del body en formato *FormData* y enviarla en la petición.

- **updateUser (id, user)**: esta acción funciona de manera muy similar a la anterior, salvo que en caso de que los datos del usuario no incluyan una imagen (se ha dejado vacía), se envía una petición de borrado al backend para la imagen del usuario, en lugar de la de actualización.

Modelo de datos

El modelo de datos debe cubrir todos los datos necesarios para el uso de un usuario dentro de la aplicación. Esto incluye datos como el hash de la contraseña (no la contraseña en plano, lo cual sería un punto muy débil en nuestra seguridad), la veteranía del usuario o el rol que este cumple.

Además, es importante destacar la relación con la entidad *Person*, ya que un usuario solo puede ser creado si pertenece a una persona concreta. Es de esta relación de donde sacamos datos como el nombre de la persona del usuario, su correo electrónico, etc.

```python
class Role(StrEnum):
    admin = "admin"
    regular = "regular"


class User(SQLULIDSchema, table=True):
    hashed_password: str | None = Field(default=None, nullable=False)
    active: bool
    veteran: bool
    image: str | None = None
    role: Role = Role.regular

    person_id: ULID = Field(
        default=None,
        foreign_key="person.id",
        unique=True,
        sa_type=SQLAlchemyULIDType,
    )

    person: "Person" = Relationship(back_populates="user")
```

Ilustración 50. Modelo de datos de usuario

Envío de emails

Para el envío de emails lo primero que debemos destacar es que vamos a utilizar como servidor de correo el servicio llamado Resend. Este servicio ofrece una capa gratuita muy generosa y solo necesita configurar un dominio propio para poder realizar envíos de correos. Para la configuración del dominio solo hemos tenido que introducir algunos registros de correo en los registros DNS del dominio.

El dominio que hemos adquirido para esto es **proteapp.es**, por lo que nuestros correos tendrán como remitente <algo>@proteapp.es.

Una vez tenemos esto claro, pasamos a la implementación. En primer lugar, tenemos un módulo llamado *email.py* que contiene la clase *EmailSender*, la cual se encarga de enviar mensajes de correo electrónico a través de la API de Resend, utilizando la plantilla HTML que se le indique como contenido. Para esto, primero carga la plantilla y con cada llamada se encarga de renderizarla y mandarla con los parámetros requeridos.

```python
class EmailSender:
    def __init__(self, template: str, sender: str) -> None:
        env = Environment(loader=FileSystemLoader(searchpath="proteapp/templates"))

        self.template = env.get_template(template)
        self.sender = sender

    def send(self, to: str, subject: str, template_kwargs: dict[str, str]):
        email_content = self.template.render(**template_kwargs)

        params: Emails.SendParams = {
            "from": self.sender,
            "to": to,
            "subject": subject,
            "html": email_content,
        }

        Emails.send(params)

email_sender = EmailSender(
    template="register.html",
    sender="Proteapp <registro@proteapp.es>",
)

email_sender.send(
    to=user.person.email,
    subject="¡Bienvenido a tu nueva cuenta de Proteapp!",
    template_kwargs={"name": user.person.name},
)
```

Ilustración 51. Implementación EmailSender

En el momento de usar esta funcionalidad no lo haremos instanciando directamente el EmailSender, sino utilizando una dependencia inyectada que configure previamente el uso de la plantilla para el email *register.html* y el remitente como "Proteapp <registro@proteapp.es>".

Endpoints

GET /user/search

Lista a todos los usuarios de la aplicación.

POST /user

Crea e inserta un nuevo usuario en la base de datos. Al momento de hacerlo le asigna una contraseña aleatora y envía un correo electrónico al email del usuario registrado dando la bienvenida e indicando la contraseña que se la ha asignado. Un ejemplo de este correo sería:

Ilustración 52. Visualización email de registro de usuario

GET /user/{id}
Recupera los datos visibles de un usuario concreto.

PUT /user/{id}
Actualiza los datos de un usuario, sin considerar la imagen.

DELETE /user/{id}
Elimina todos los datos de un usuario.

POST /user/{id}/image
Actualiza la imagen de un usuario. Si este tenía otra imagen anteriormente es borrada.

DELETE /user/{id}/image
Elimina la imagen de un usuario.

Esquemas

Los esquemas de salida que utilizamos para dar respuesta en los endpoints que devuelven uno o varios usuarios son *ListedUser* y *CompleteUser*. El primero es utilizado en el endpoint donde se devuelve la lista de usuarios, mientras que el otro se utiliza como respuesta en el resto de endpoints (excepto en el de eliminar un usuario, que no devuelve contenido).

La única diferencia entre ellos es que *CompleteUser* contiene todos los datos de la persona del usuario y *ListedUser* recorta algunos campos.

Ilustración 53. Esquema de usuario

Por otro lado, el esquema editable para modificar y crear nuevos usuarios consiste en los campos active, veteran y el ULID de persona, heredando de BaseSchema. La imagen no es

parte de este, ya que para modificarla lo hacemos de forma independiente, sin modificar el resto de los valores del usuario. Por otro lado, como explicábamos anteriormente, a la hora de crear un usuario debemos especificar el identificar de la persona a la que queremos crearle el usuario, no es posible especificar los datos de la persona en conjunto para que esta sea creada al momento.

5.3.5 Autenticación

El módulo de autenticación es el que engloba toda la parte de inicio de sesión e identificación del usuario. Pese a que ya hemos explicado gran parte de este módulo a lo largo del documento, como en el punto *5.1.4 Dependencias - Usuario con la sesión iniciada*, vamos a invertir un momento en explicar el resto de las partes del mismo, como las vistas o el endpoint que contiene.

Interfaz de usuario - Inicio de sesión

Lo primero que nos encontramos en relación con este módulo es la vista de inicio de sesión. En ella, el usuario introducirá sus datos, email y contraseña, si son correctos pasará a la lista de animales mientras que si no lo son la vista dará un error y reiniciará los datos del formulario.

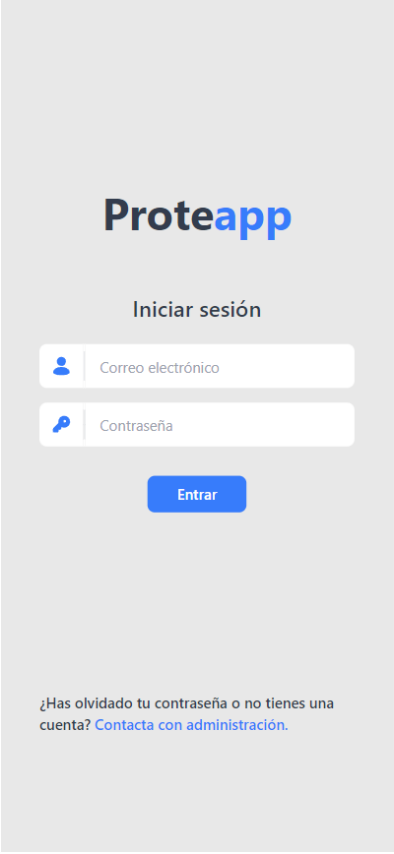


Ilustración 54. Visualización del login

Interfaz de usuario - Editar mi perfil

También, como parte de este módulo referente a las acciones del usuario autenticado, tenemos la edición del perfil del usuario que ha iniciado sesión. A esta vista se accede clicando en el avatar del usuario de la barra superior y no es más que una vista idéntica a la que hemos visto en el punto *5.3.4 Interfaz de usuario - Editor de usuarios*.

La única diferencia con esta es que aquí solo podremos modificar la imagen y los datos de la persona, no la veteranía, ya que esta modificación se reserva a los administradores, por lo que es más sencillo relegarla a la edición de usuarios (que solo es accesible por estos).

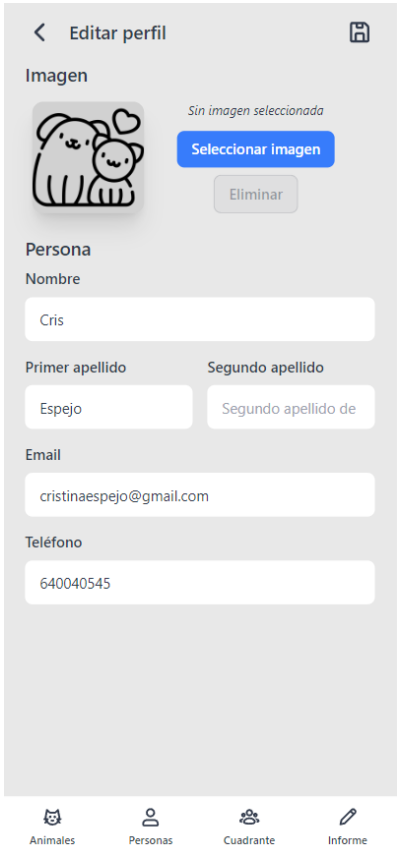


Ilustración 55. Visualización del editor de perfil

Store

Para este módulo tenemos dos stores diferentes. Una incluye toda la parte de autenticación, recuperación y envío del token, así como gestión del usuario logeado. El otro store se dedica en exclusiva a la actualización del perfil del usuario.

En la primera tenemos las siguientes acciones:

- **login (username, password)**: se encarga de hacer la llamada de obtención del token al backend. Si esta es correcta y el token es devuelto este se almacena en la variable reactiva *token* y se ejecuta la siguiente acción *fetchLoggedUser*.

- **fetchLoggedUser ()**: dado el token obtenido del backend, esta acción se encarga de decodificarlo, obtener el identificador del usuario y hacerle una petición al backend para obtener los datos de este usuario. Luego estos datos son almacenados en la variable reactiva *loggedUser* y se fija también la variable *isAuthenticated* a verdadero. Por último, se comprueba el rol de usuario y se fija la variable reactiva *isAdmin*, la cual es usada a lo largo de la aplicación para modificar las vistas conforme al permiso, de acuerdo a si contiene el rol de administrador o no.

En la segunda store, tenemos:

- **updateProfile (user)**: se encarga de realizar las llamadas al backend para modificar el perfil del usuario. En este caso, solo actualiza los datos de la imagen ya que los datos de la persona se hacen en otra acción y el dato de veteranía del usuario no puede ser actualizado desde el perfil.
  Una vez que la actualización termina se produce un refresco del usuario que ha iniciado sesión, ejecutando la acción que hemos visto hace un momento, *fetchLoggedUser*, para mostrar la imagen actualizada de este.

- **updateProfilePerson (person)**: idéntico al caso anterior, solo que actualiza los datos de la persona del usuario, en lugar de la imagen.

Endpoints

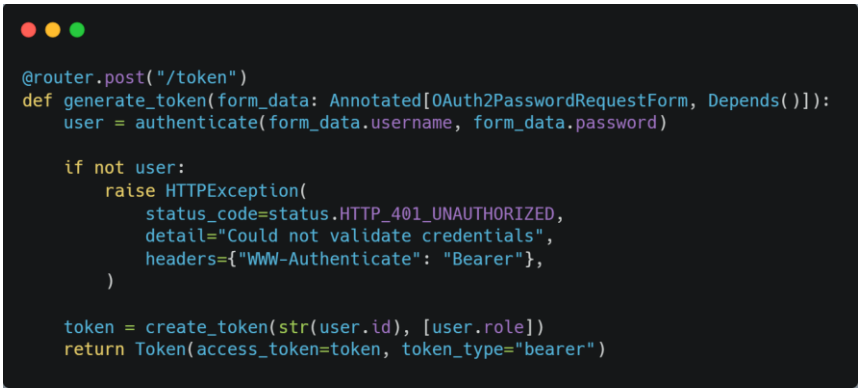
PUT /profile/person
Actualiza los datos identificativos de la persona asociada al usuario que ha inciado sesión.

POST /profile/image
Actualiza la imagen del usuario que ha iniciado sesión. La nueva imagen es almacenada en la carpeta *images* y si había una anterior es eliminada en el momento en el que la nueva es guardada.

DELETE /profile/image
Elimina la imagen del usuario que ha iniciado sesión.

POST /auth/token
Recibe los datos de autenticación en formato FormData (como requiere el estándar OAuth2), comprueba que las credenciales sean correctas y en caso de serlo genera un token con los datos del usuario (identificador y roles) que le es mandado de vuelta a este para identificarlo

en posteriores peticiones. Si ocurre un error se manda un código 401 de respuesta como no autorizado.



```python
@router.post("/token")
def generate_token(form_data: Annotated[OAuth2PasswordRequestForm, Depends()]):
    user = authenticate(form_data.username, form_data.password)

    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Could not validate credentials",
            headers={"WWW-Authenticate": "Bearer"},
        )

    token = create_token(str(user.id), [user.role])
    return Token(access_token=token, token_type="bearer")
```

Ilustración 56. Implementación del método POST del token de autenticación

Esquemas

Los únicos nuevos esquemas de este módulo, ya que la parte de editar perfil utiliza los esquemas de usuario, son los datos que contiene el token y los que se envían de vuelta al usuario.

```python
class Token(BaseModel):
    access_token: str
    token_type: str


class TokenData(BaseModel):
    user_id: str | None = None
    scopes: list[str] = []
```

Ilustración 57. Esquemas de Usuario

6

Conclusiones

El desarrollo de esta aplicación ha sido una experiencia muy gratificante, especialmente porque hemos estado comprometidos con la causa al formar parte activa de la protectora. Desde dentro, hemos comprendido la relevancia de los datos y registros que se mantienen y también las limitaciones técnicas con las que contábamos. Trabajar con los recursos disponibles anteriormente resultaba ineficiente, ya que perdíamos mucho tiempo gestionando turnos y realizando seguimientos, lo que impactaba negativamente en la efectividad de nuestras tareas diarias.

Nos hemos esforzado por adaptarnos no solo a nuestras ideas iniciales, sino a las verdaderas necesidades y requerimientos de los usuarios que utilizarán la aplicación. Esto nos llevó a ajustar nuestras propias ideas y expectativas, centrándonos más en crear una herramienta intuitiva, práctica y accesible para quienes estarán en contacto directo con ella. En lugar de seguir únicamente nuestra visión original, priorizamos un diseño más adaptativo, asegurándonos de que la aplicación fuera fácil de usar y verdaderamente útil en el contexto específico de la protectora.

Este enfoque nos permitió desarrollar una solución que no solo satisface nuestras expectativas técnicas, sino que también aborda las demandas reales del día a día de los usuarios, mejorando su experiencia y aumentando la eficiencia en sus labores. La comunicación constante con los miembros de la protectora, proporcionándonos su feedback y readaptando nuestro desarrollo a ello, nos ha hecho aprender cómo se trabaja en un entorno en el que el enfoque está siempre en la mejora continua, la colaboración y la flexibilidad.

En lo referente al desarrollo y la tecnología utilizada, ha representado un desafío significativo aprender un lenguaje y un entorno completamente nuevos, como Vue.js y TypeScript, especialmente considerando que no había trabajado de manera exhaustiva en el desarrollo frontend. Este proceso me permitió salir de mi zona de confort, ampliar mis competencias técnicas y adquirir una comprensión más profunda sobre la construcción de interfaces de usuario dinámicas y eficientes.
La elección de tecnologías para este proyecto se basó en un análisis detallado de las necesidades específicas y las posibles exigencias futuras de la aplicación. Optamos por un stack tecnológico que ofreciera flexibilidad, eficiencia y escalabilidad. Incluir tecnologías como WebSockets, FastAPI, Docker o MongoDB ha permitido ofrecer un software robusto y adaptable, capaz de manejar comunicaciones en tiempo real, optimizar el rendimiento del backend, asegurar la portabilidad y la consistencia en diferentes entornos de desarrollo y producción, y gestionar eficazmente datos semiestructurados.

Finalmente, hemos logrado cumplir con todos los requisitos iniciales, con la excepción de la implementación de una PWA. Inicialmente, consideramos que una PWA sería la mejor opción para adaptarse tanto a la web como a dispositivos móviles. Sin embargo, dada la naturaleza de los usuarios en la protectora, decidimos priorizar el desarrollo de una versión completamente funcional para la web móvil, sin necesidad de instalación ni pasos adicionales, para asegurar la máxima accesibilidad para todos. En el futuro, planeamos incorporar gradualmente las funcionalidades de una PWA, como la capacidad de instalación y el uso offline, para enriquecer la experiencia del usuario.

En conclusión, el desarrollo de esta aplicación ha tenido un impacto transformador en la protectora, mejorando significativamente la gestión de datos y la practicidad en la visualización de la información, lo que ha incrementado la eficiencia operativa. Al optar por un diseño accesible y funcional para la web móvil, hemos garantizado que todos los usuarios puedan utilizar la herramienta sin complicaciones adicionales. El seguimiento continuo y el desarrollo futuro se centrarán en mantener y mejorar la aplicación mediante actualizaciones regulares, corrección de errores y ajustes según las necesidades emergentes de la protectora, asegurando así que la herramienta siga siendo efectiva y adaptada a los requerimientos cambiantes de la organización.

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 58. 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.

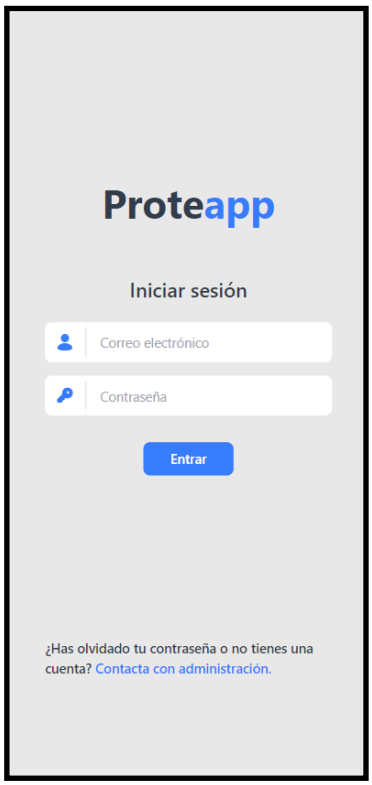


Ilustración 59. GDU - Visualización del login

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.

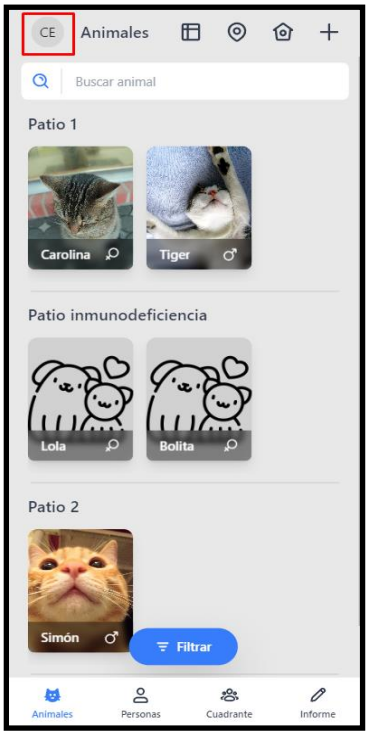


Ilustración 60. GDU - Paso 1 edición perfil

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.

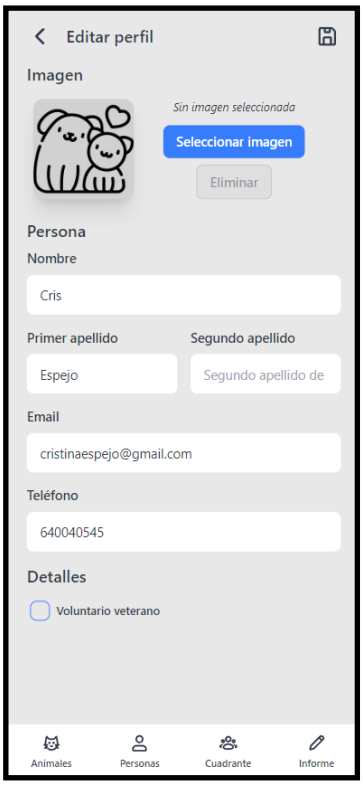


Ilustración 61. GDU - Edición de perfil

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

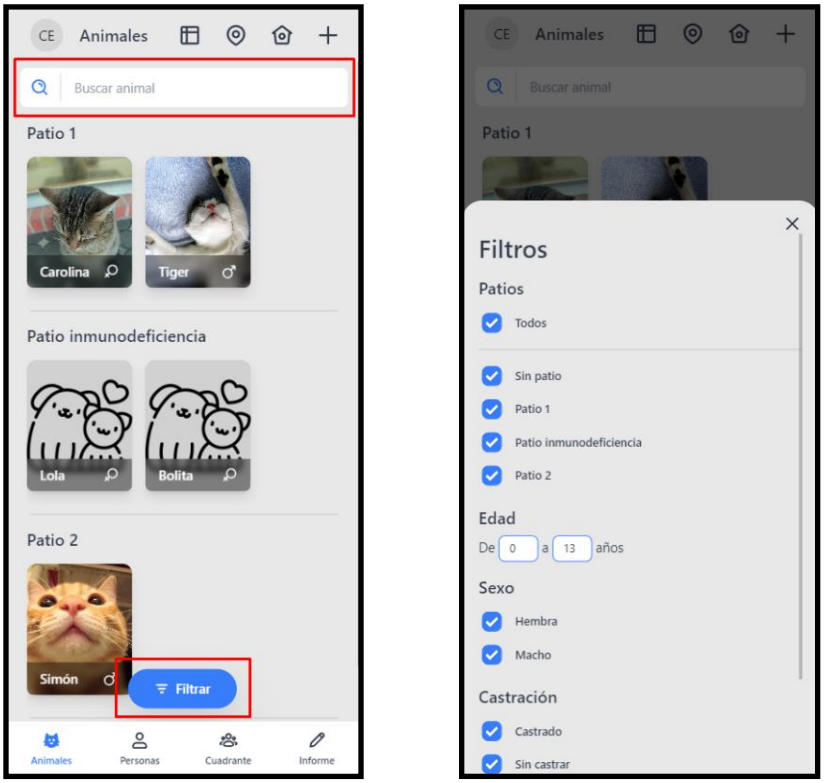


Ilustración 62. GDU - Filtrado de animales

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.

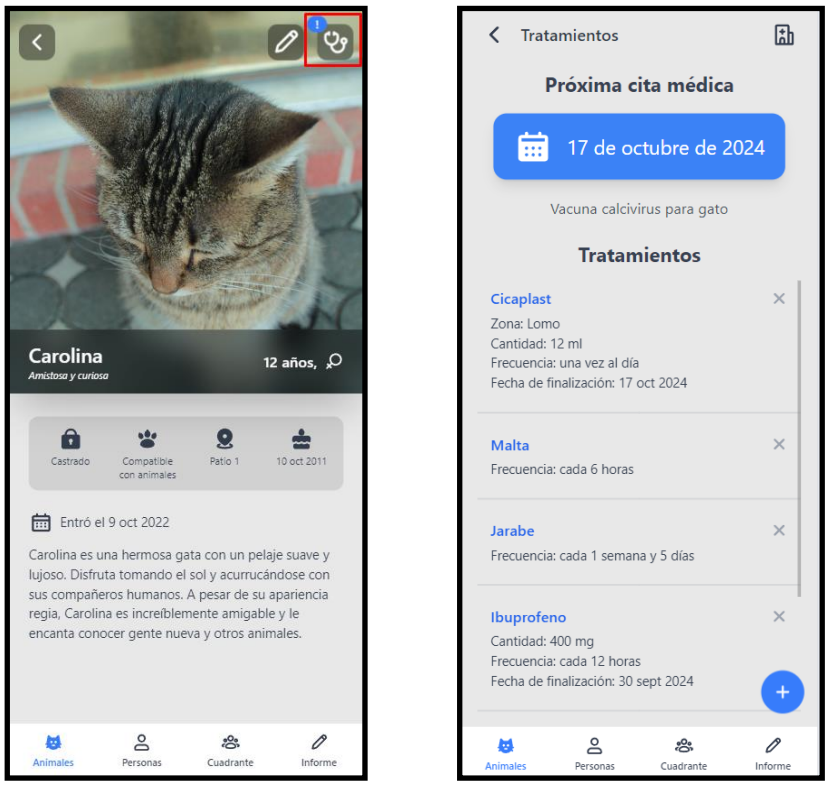


Ilustración 63. GDU - Visualización de citas médicas

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.

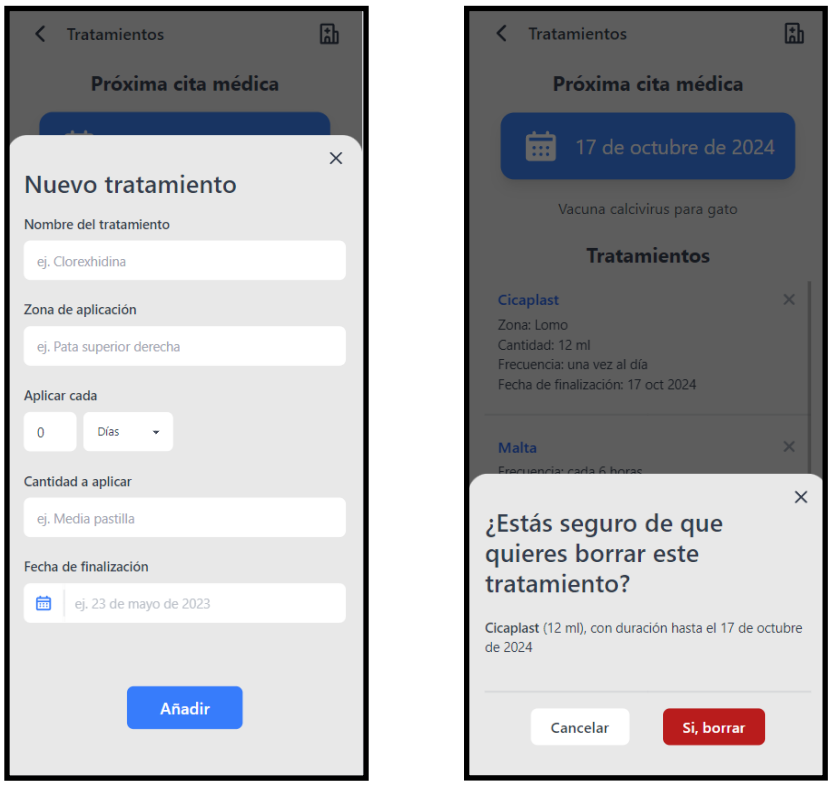


Ilustración 64. GDU - Creación y borrado de tratamientos

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.

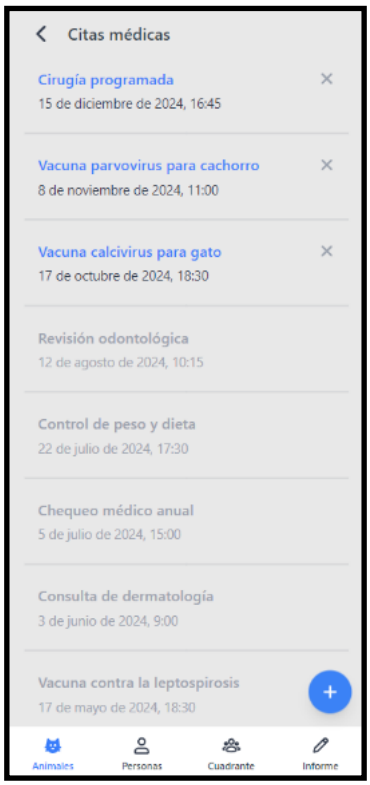


Ilustración 65. GDU - Visualización de citas médicas

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.

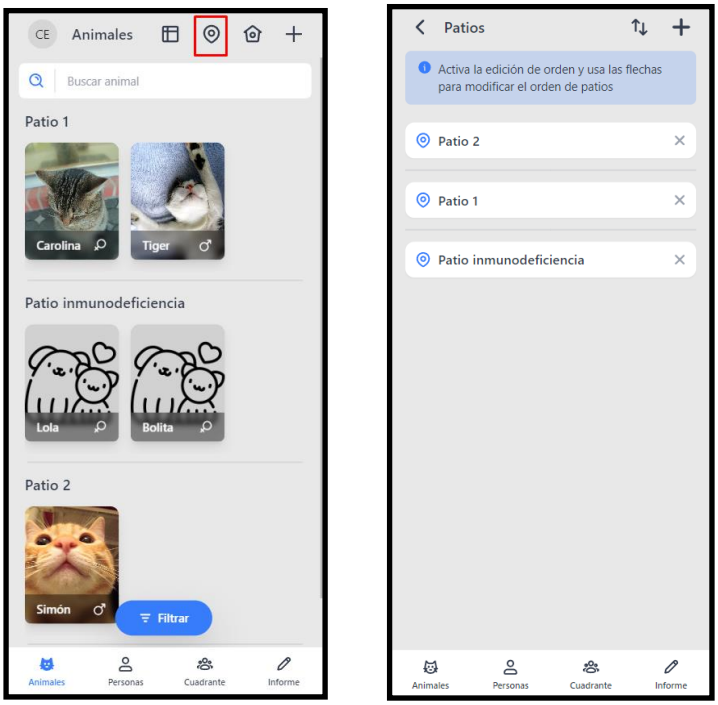


Ilustración 66. GDU - Visualización de patios

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.

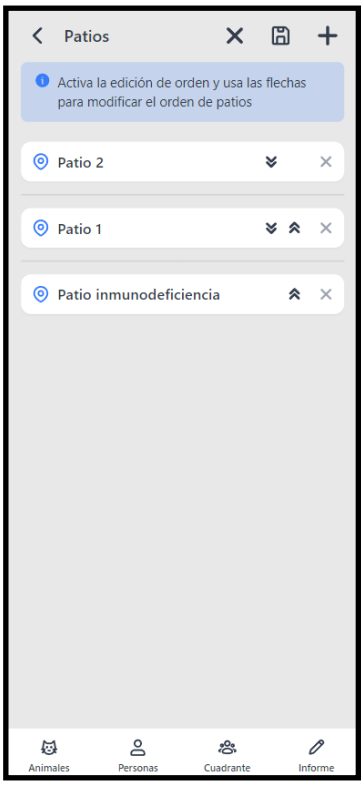


Ilustración 67. GDU - Visualización de orden de patios

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

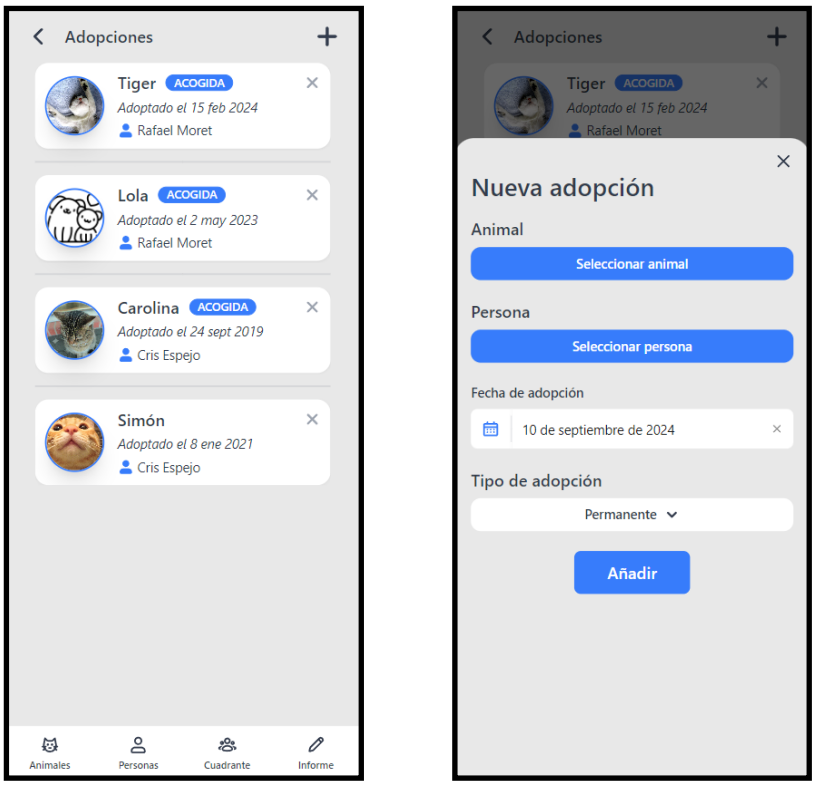


Ilustración 68. GDU - Listado y creación de adopciones

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.

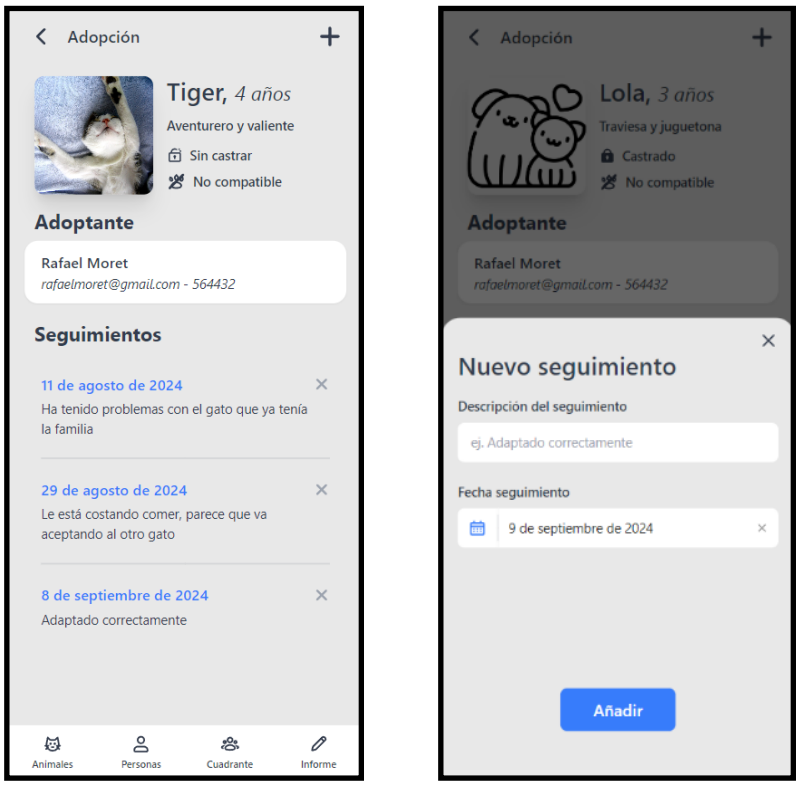


Ilustración 69. GDU - Visualización de adopción y seguimiento

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.

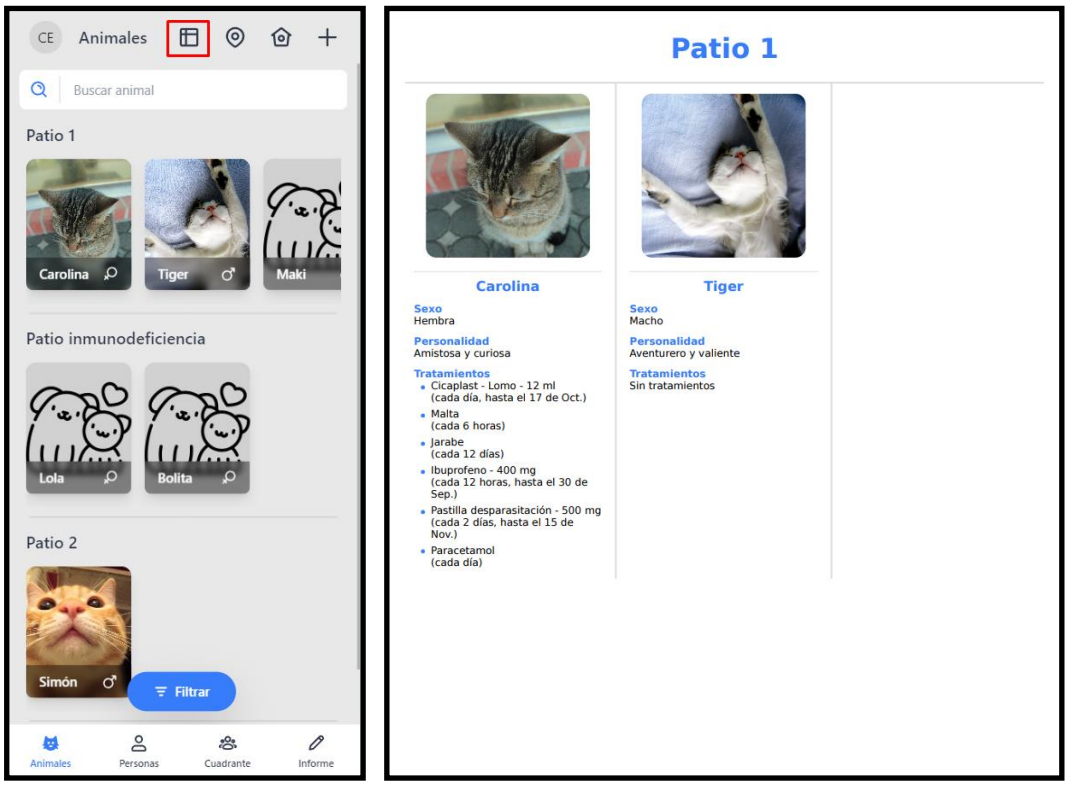


Ilustración 70. GDU - Visualización de creación de PDF

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.

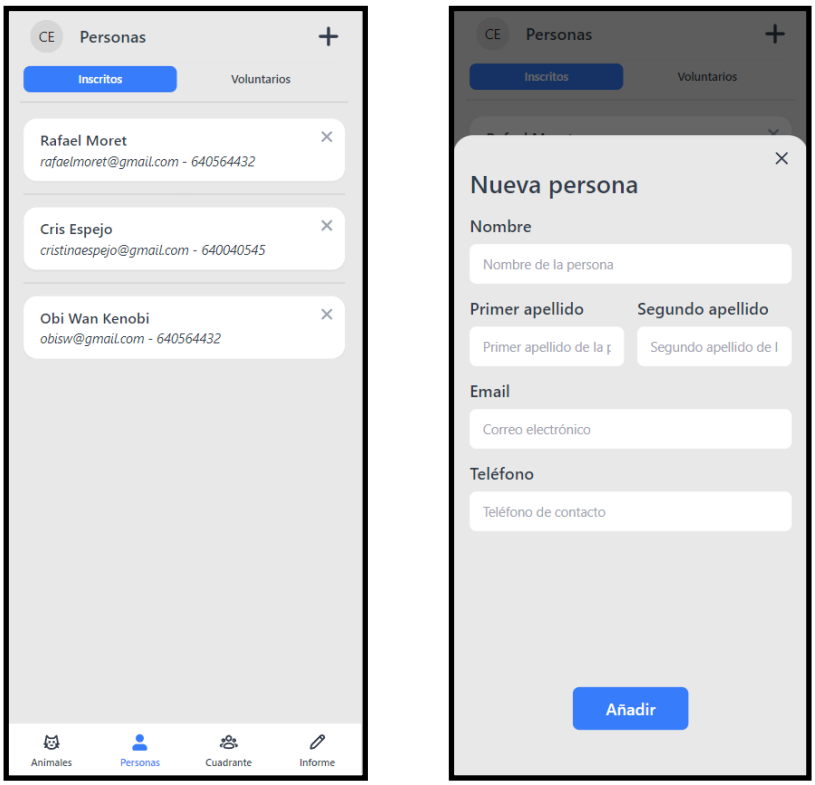


Ilustración 71. GDU - Visualización del listado y creación de personas

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.

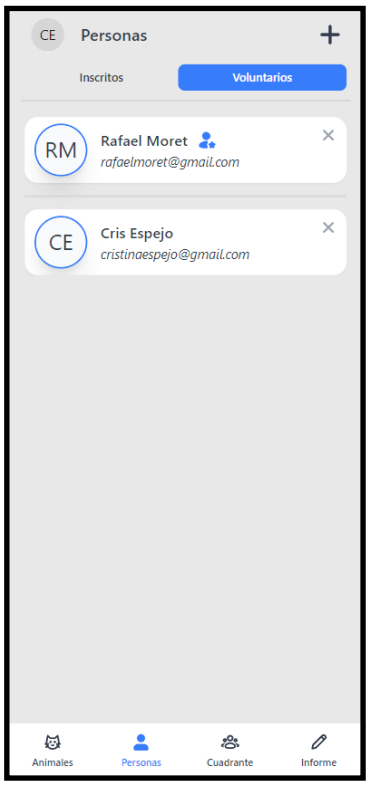


Ilustración 72. GDU - Visualización del listado de voluntarios

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 73. 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.

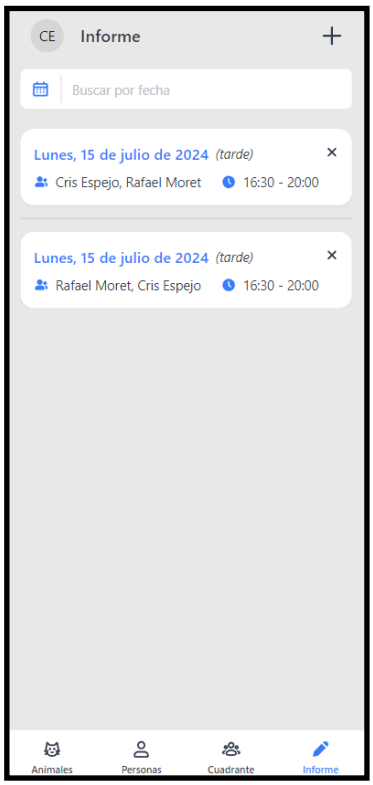


Ilustración 74. GDU - Visualización del listado de informes

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

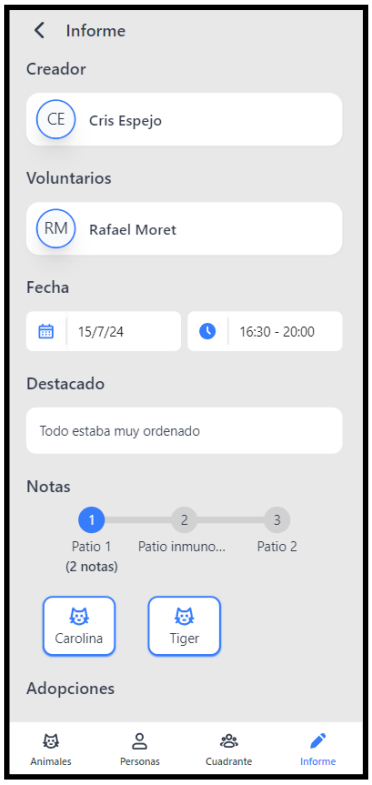


Ilustración 75. GDU - Visualización de un informe de turno

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.

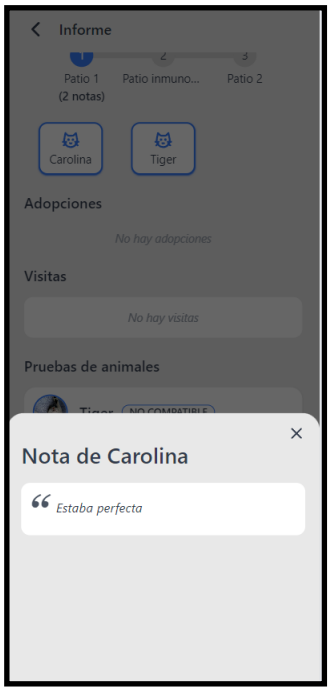


Ilustración 76. GDU - Visualización de nota de informe

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.

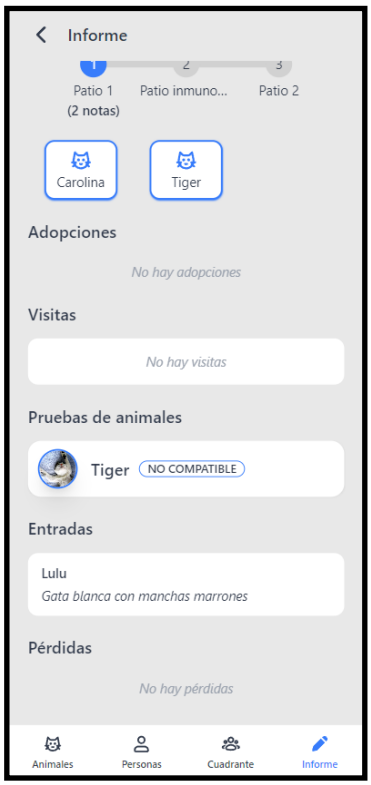


Ilustración 77. GDU - Visualización parte 2 del informe

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.

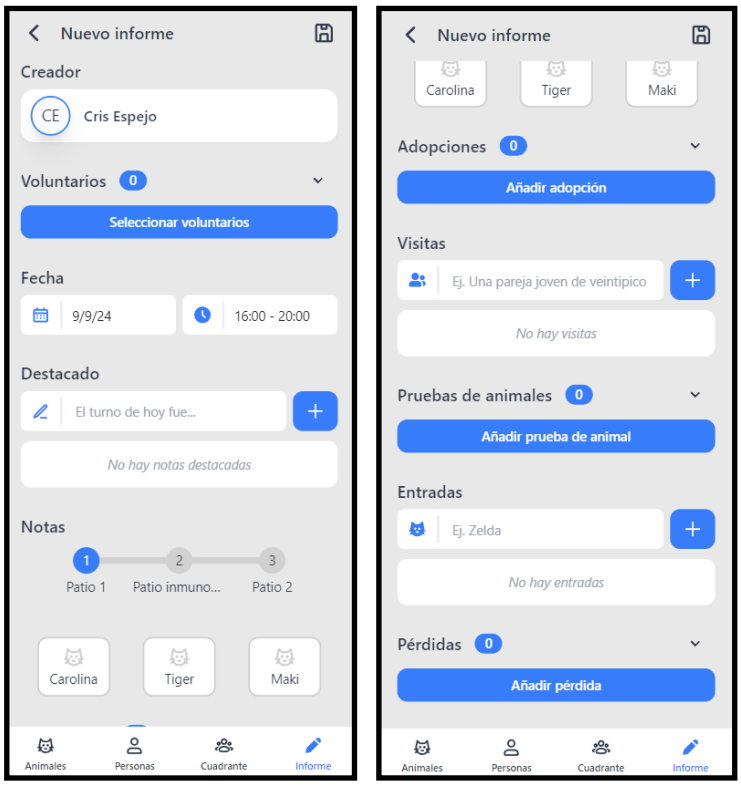


Ilustración 78. GDU - Visualización de creación de informe

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.

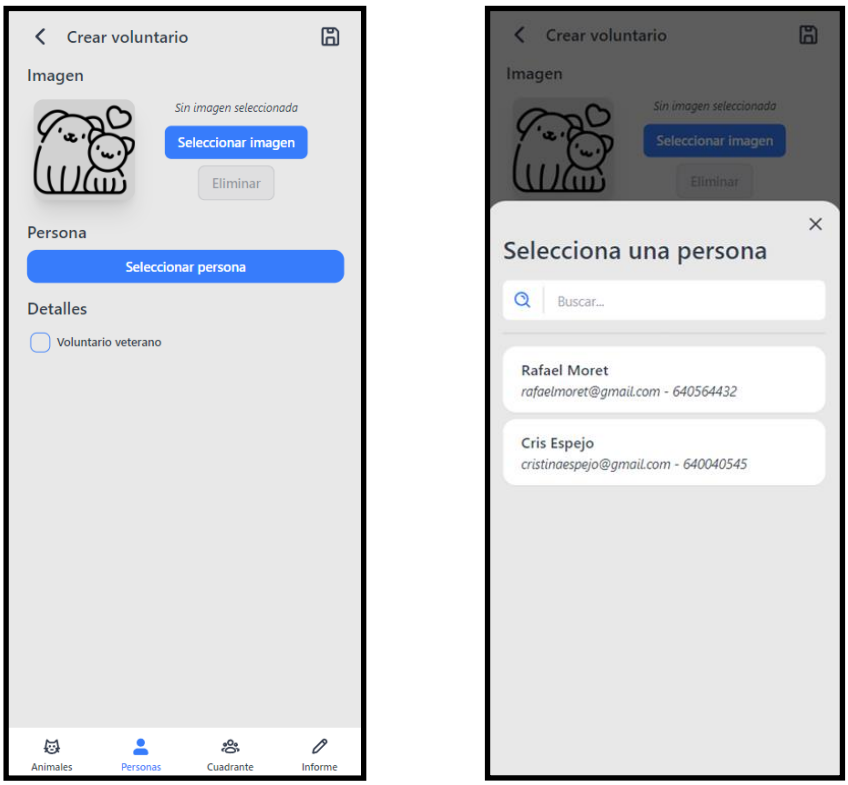


Ilustración 79. GDU - Visdualización creación de voluntario

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.

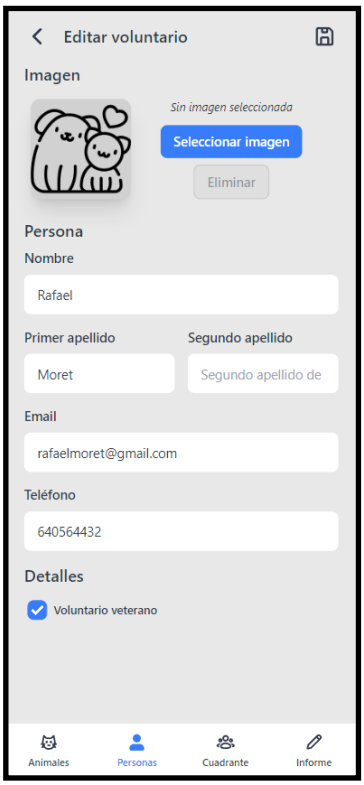


Ilustración 80. GDU - Visualización editor de voluntario

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.

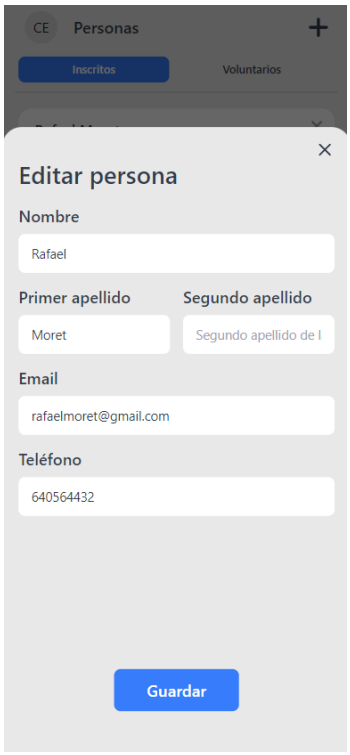


Ilustración 81. GDU - Visualización de editor de persona

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 82. 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.

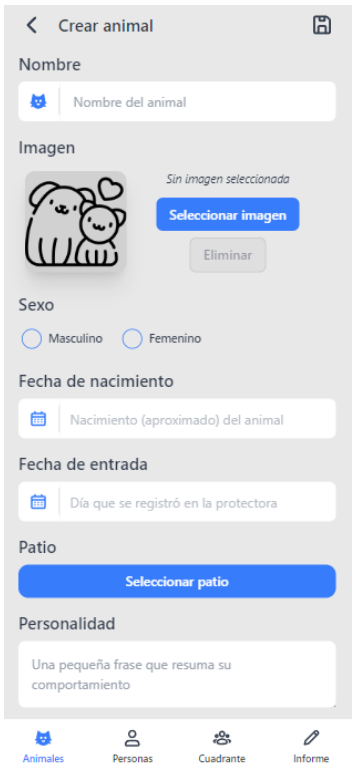


Ilustración 83. GDU - Visualización creación de animal

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 84. 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.

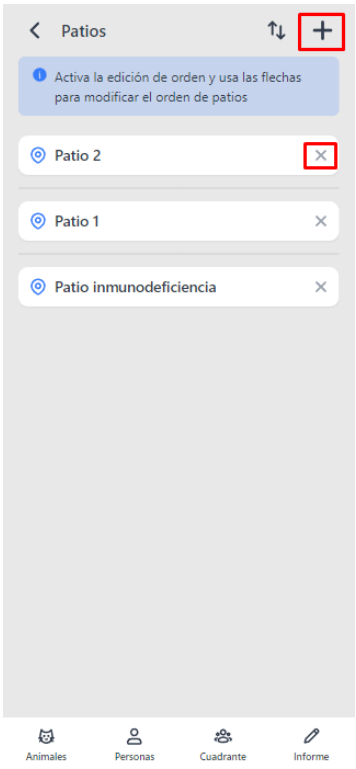


Ilustración 85. GDU - Visualización creación y eliminación de patios

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

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

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