instituto
superior de
engenharia
de lisboa

Vetly
An All-In-One Multiplatform Solution to Clinic and Pet Management

Daniel Carvalho

Supervisor: Diego Passos

Final report written for Project and Seminar
BSc in Computer Science and Computer Engineering

July 2025

Instituto Superior de Engenharia de Lisboa

Vetly

49515 Daniel Alexandre Marto Queijo de Carvalho

Supervisor: Diego Gimenez Passos

Final report written for Project and Seminar
BSc in Computer Science and Computer Engineering

July 2025

Resumo

Vetly pretende oferecer uma solução multiplataforma que reúna funcionalidades valiosas que atualmente encontram-se dispersas por diferentes serviços, tanto para clínicas como para colaboradores e utilizadores.

Assente num backend robusto desenvolvido em *Kotlin* [1] e *Spring Boot* [2], garante escalabilidade, utilizando uma base de dados *PostgreSQL* [3] e o *Firebase* [4] para armazenamento de ficheiros, resultando em consistência e disponibilidade de dados. Com o *Expo* [5] a suportar uma base de código partilhada, o processo de lançamento multiplataforma é simplificado, proporcionando ampla acessibilidade, baixos custos de manutenção e uma ótima experiência de utilização, já que os módulos podem ser modificados e adaptados a diferentes plataformas, sempre que necessário.

As funcionalidades principais incluem a gestão de inventário e de suprimentos, permitindo às clínicas monitorizar os recursos disponíveis; o registo de animais de estimação e consultas, oferecendo aos proprietários uma forma de manter um histórico organizado de consultas e tratamentos; e guias elaborados por veterinários certificados, disponibilizando dicas fiáveis de cuidados para várias espécies e condições.

A pensar no futuro, o *Vetly* foi concebido com flexibilidade e crescimento em mente. O seu design modular permite a adição de novas funcionalidades adaptadas às necessidades específicas dos utilizadores, como agregação de múltiplos animais por família, tratamentos especializados ou fluxos de trabalho próprios de cada clínica. Ao centrar-se num processo de desenvolvimento escalável e orientado pelo feedback dos utilizadores, o *Vetly* procura não só apoiar as práticas veterinárias atuais, mas também evoluir em conjunto com estas — aproximando o cuidado profissional do acompanhamento em casa e fomentando relações mais fortes entre as clínicas e as comunidades que servem.

Abstract

Vetly aims to deliver a multi-platform solution that unites valuable functionalities currently scattered across different services for clinics, employees, and users alike.

Built on a robust backend using *Kotlin* [1] and *Spring Boot* [2], guaranteeing scalability, and with a *PostgreSQL* [3] database and *Firebase* [4] for file storage, resulting in data consistency and availability. With Expo [5] powering a shared codebase, deployment on Android, iOS and web is streamlined, as well as offering wide accessibility, low maintenance cost and a great user experience, as modules can be modified and suited for different platforms if needed.

Core features would include inventory and supply management, allowing clinics to keep track of available resources; pet and checkup tracking, giving owners a way of maintaining an organized record of medical appointments and treatments; and curated guides created by certified veterinarians, offering reliable care tips for various species and conditions.

Looking ahead, *Vetly* is built with flexibility and growth in mind. Its modular design allows for the addition of new features tailored to specific user needs, such as multi-pet households, specialized treatments, or clinic-specific workflows. By focusing on a scalable, user-informed development process, *Vetly* strives not only to support current veterinary practices but to evolve with them - bridging gaps between professional care and at-home follow-up, and fostering stronger relationships between clinics and the communities they serve.

List of Accronyms

API Application Programming Interface

SQL Structured Query Language

NoSQL Not Only SQL

JPA Jakarta Persistence API

JSON JavaScript Object Notation

DB Database

CRUD Create, Read, Update, Delete

HTTP HyperText Transfer Protocol

HTTPS HyperText Transfer Protocol Secure

JPQL Jakarta Persistence Query Language

UI User Interface

UX User Experience

Acknowledgements

I would like to first thank my parents and grandmother. They were my driving force and support through tough decisions, and I could always count on them.

Secondly, my mother-in-law, whose knowledge and feedback throughout the development of this project were essential in ensuring the inclusion of necessary and relevant features.

I also thank my professor, whose guidance and insights during the course were invaluable in shaping the direction of this project and encouraging a practical and thoughtful approach.

And finally, my girlfriend and friends, who were patient with my brainstorming, questioning, and troubleshooting throughout the process.

Contents

1 Introduction **1**

1.1 Background 1

1.2 Proposed Solution – Vetly 1

1.3 Motivation 2

1.4 Requirements 2

1.4.1 Functional Requirements 2

1.4.2 Non-functional Requirements 3

1.5 Report Organization 3

2 Architecture **5**

2.1 Overview 5

2.2 Main Components 5

2.3 Technologies Used 6

2.3.1 Backend Stack 6

2.3.2 Frontend Stack 6

3 Backend **9**

3.1 Structure 9

3.2 Data Model 11

3.3 Database Access 17

3.4 Repository 18

3.5 Services 19

3.6 Controller 20

3.7 Authentication 21

3.8 Global Exception Handling 23

3.9 Testing 24

3.10 Deployment 24

4 Frontend **25**

4.1 Structure 25

4.2 User Interface 26

4.3 Routes 29
4.4 Services 30
4.5 Components 31
4.5.1 Hooks 31
4.5.2 Platform Specific 31

5 Conclusion and Future Work **33**
5.1 Conclusion 33
5.2 Future Work 33

References **35**

List of Figures

2.1 Overall architecture 6

3.1 Backend structure 10
3.2 Relational model overview 11
3.3 User relational model 11
3.4 User requests relational model 12
3.5 Animal relational model 13
3.6 Clinic relational model 14
3.7 Checkup relational model 15
3.8 Medical supply relational model 16

4.1 Mobile interface (on the public about page) 26
4.2 Web interface (on the public about page) 26
4.3 Mobile interface (on the private pet route) 27
4.4 Web interface (on the private pet route) 27
4.5 Mobile drawer 28

Code Listing

3.1 `application-prod.properties` file 17

3.2 User repository interface 18

3.3 Checkup repository interface 18

3.4 User service class 19

3.5 User controller interface 20

3.6 User controller class 20

3.7 Production-facing Firebase token verifier (implementing the corresponding interface) 22

3.8 Global exception handler 23

Chapter 1

Introduction

Over the years, the veterinary field has undergone significant transformation, driven in part by the increasing availability and demand for specialized software solutions. Systems such as *Otto*, *Vet Manager*, *ezyVet*, and *Vetspire* have emerged as responses to the inefficiencies of traditional clinic management, offering digital platforms to streamline operations, improve data accessibility, and enhance client relationships.

1.1 Background

As previously mentioned, there are currently implementations marketed towards these issues, like *Otto* [6], *Vet Manager* [7], *ezyVet* [8], and *Vetspire* [9]. Each of these platforms attempts to fill this niche within the veterinary management software market — for instance, offering appointment scheduling, medical recordkeeping, client communication, or billing. However, they often prioritize traditional pet clinics and overlook alternative veterinary contexts, such as mobile vets, animal shelters, or mixed-practice professionals who care for both domestic and farm animals.

Despite their advancements, they still struggle with fragmented functionalities, complex data management, and weak engagement with pet owners.

1.2 Proposed Solution – Vetly

Vetly, our proposed solution, aims to address these shortcomings by offering a unified, intuitive platform that prioritizes ease of use, seamless feature integration, and stronger connections between clinics and the communities they serve.

By drawing inspiration from the strengths of current systems while introducing unique features tailored to both professionals and pet owners, *Vetly* seeks to redefine how veterinary care is managed and experienced in the modern world.

Core features would include clinic resource management, pet medical histories, prescription details, emergency services, as well as veterinarian-curated guides for everyone.

1.3 Motivation

The main driving force for this project was identifying significant issues with legacy software systems in the veterinary sector, particularly those developed in the late 1990s that have become outdated and unmaintained. While the conventional solution would be to engage specialized software providers, most of these services are positioned behind substantial paywalls and do not offer comprehensive enough functionality to justify their high costs.

Through market analysis, it became apparent that there was a notable gap in solutions that effectively connect pet owners and veterinarians while providing a cost-effective alternative to expensive commercial platforms. This identified need, combined with the academic requirements of developing a practical application using JPA (Jakarta Persistence API) technology, presented an opportunity to create a modern, accessible solution that addresses real-world challenges in veterinary practice management.

The convergence of these factors - the need for modernized veterinary software, the lack of affordable comprehensive solutions, and the educational objective of exploring JPA capabilities - provided the foundation and motivation for undertaking this development project.

1.4 Requirements

1.4.1 Functional Requirements

A substantial section of the project requires basic CRUD (Create, Read, Update, Delete) operations for all entities. The following functional requirements define the essential capabilities of the system:

- User registration and authentication (sign in / sign up)
- Role-based access control (*e.g.*, veterinarian *vs* regular user)
- Animal profile management (create, read, update, delete)
- Checkup creation and management
- Prescription and medical record management
- Viewing and managing user-specific or vet-specific data
- Guide and documentation creation for treatment or care instructions
- Request system (*e.g.*, requesting appointments, treatments, etc.)
- Search functionality for animals, users, or medical data

1.4.2 Non-functional Requirements

A non-functional requirement is a feature or mechanism that is not essential to the project. It can and should be implemented in the future; however, due to the scope of the project, labor requirements, and/or time constraints to completely develop and test, it is considered a low-priority item.

- Use of an API (Application Programming Interface) for rough specie determination
- Use of an API for pharmaceutical search
- Notification system

1.5 Report Organization

The progress report for this project is divided into four main chapters:

Chapter 2: Architecture Presents the overall structure of the solution, including initial ideas for the data model, the chosen technologies, and the rationale behind architectural decisions.

Chapter 3: Backend Describes the server-side implementation, covering API design, database structure, authentication mechanisms, and integrations with external services (if any).

Chapter 4: Frontend Details the client-side development, including user interface design, user experience considerations, and the interaction flow between views and the backend.

Chapter 5: Conclusion Provides a summary of the main features implemented in the project, reflects on the results and challenges encountered, and outlines potential improvements or future extensions to enhance the system.

Chapter 2

Architecture

2.1 Overview

The system adopts a traditional client-server architecture, where a central backend exposes a RESTful API, and multiple frontends (web, mobile) consume this API. The backend also communicates with two databases to persist application data in different capabilities.

The containerized architecture provided by Docker [10] enables horizontal scalability through the deployment of multiple identical application instances. This approach allows the system to handle increased throughput by distributing load across multiple containers, which can be dynamically scaled up or down based on demand. The stateless nature of containerized applications eases this scaling process, as new instances can be rapidly created without complex configuration dependencies.

Note: This report is based on the 1.0 release.

2.2 Main Components

The system is composed of four main components, each fulfilling a key role in the overall architecture:

Backend Implemented in Kotlin [1] using Spring Boot [2], the backend exposes a RESTful API that serves as the core of the system. It handles authentication, authorization, request validation, business logic, and communication with the database.

Frontend Consists of web and mobile clients developed using the Expo [5] framework and TypeScript [11]. These clients provide the user interface, handle session and state management, and communicate with the backend through HTTP (Hypertext Transfer Protocol) requests.

Database A PostgreSQL [3] database persists all structured data, including user profiles, requests, roles, and associated metadata. The schema was designed for clarity, normalization, and future scalability.

External Services Firebase [4] is integrated to manage authentication and cloud file storage. Additional APIs or third-party services can be added as needed to extend platform capabilities.

A visual representation of the overall structure can be seen in Figure 2.1, where each main component is connected in a different way based on their usage.

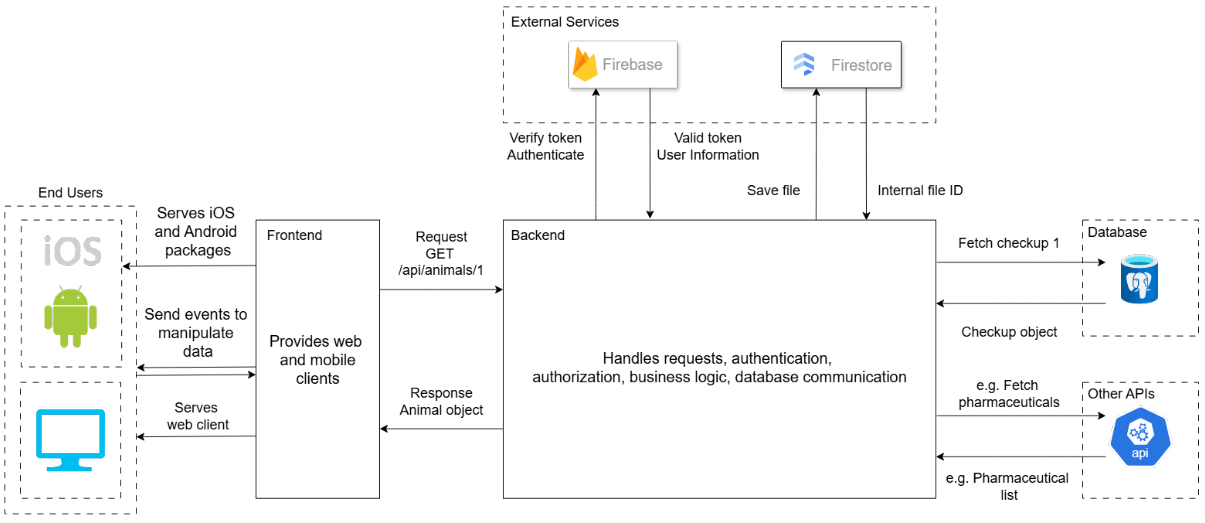


Figure 2.1: Overall architecture

2.3 Technologies Used

2.3.1 Backend Stack

The backend is developed in Kotlin [1] using the Spring Boot [2] framework, which simplifies the creation of RESTful APIs. It handles core responsibilities such as authentication, authorization, and database interaction. For data persistence, it integrates with a PostgreSQL [3] database, ensuring structured storage and relational consistency.

Firebase [4] is used for both authentication and cloud storage. Authentication via Firebase [4] provides a secure and scalable login mechanism, while Firebase Cloud Storage supports the handling of user-uploaded files. This stack ensures a consistent, high-availability, backend that can scale easily when deployed in Dockerized environments.

2.3.2 Frontend Stack

The frontend layer includes both web and mobile clients, providing users with responsive and platform-appropriate interfaces. These clients interact with the backend via HTTP requests to consume the RESTful API.

Both web and mobile applications are developed using modern Typescript [11] technologies, making the most out of the Expo [5] framework, allowing dynamic and reactive user experience while maintaining a consistent codebase across platforms. Both clients share common logic for authentication and state management and are optimized for seamless interaction with the backend services.

Chapter 3

Backend

3.1 Structure

The architecture is composed of five core layers, each with clearly defined responsibilities:

Interceptors handle route-level authentication and authorization, ensuring that only authorized users can access protected endpoints.

Argument Resolvers act as a bridge between raw HTTP request data and controller method parameters, preparing and injecting necessary arguments automatically.

Controllers serve as the entry point for the application's API. They orchestrate the flow of data by delegating tasks to the underlying service layer.

Services contain the application's business logic. These components are stateless and designed to be reusable across different parts of the application.

Repositories interface with the database using Spring Data JPA [12]. They manage data persistence, retrieval, and querying, abstracting away boilerplate SQL (Structured Query Language) operations.

As seen in Figure 3.1, the top three layers — interceptors, argument resolvers, and controllers — form the web-facing tier and are responsible for handling request-specific logic. In contrast, services and repositories are part of the core application logic, remaining decoupled from web concerns to ensure better testability and modularity.

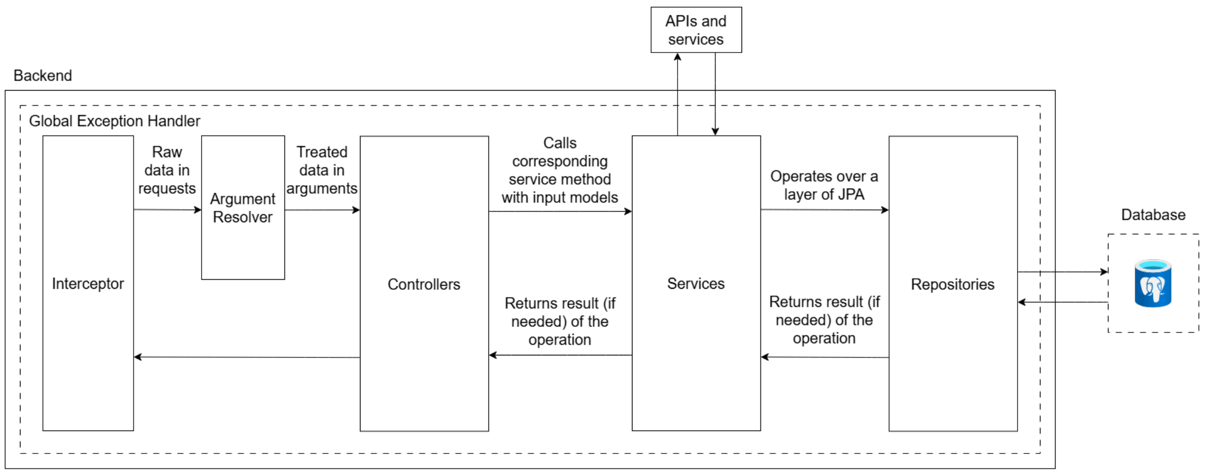

APIs and services
Backend
Global Exception Handler
Raw data in requests
Treated data in arguments
Calls corresponding service method with input models
Operates over a layer of JPA
Database
Interceptor
Argument Resolver
Controllers
Services
Repositories
Returns result (if needed) of the operation
Returns result (if needed) of the operation

Figure 3.1: Backend structure

3.2 Data Model

Prior to defining the complete data model, auditability was considered a key requirement. To support this, a base table was introduced containing the `created_at` and `updated_at` timestamp columns. Any table requiring temporal tracking can inherit from this base, ensuring consistency and avoiding schema duplication.

An overview of the main data model can be seen in Figure 3.2.

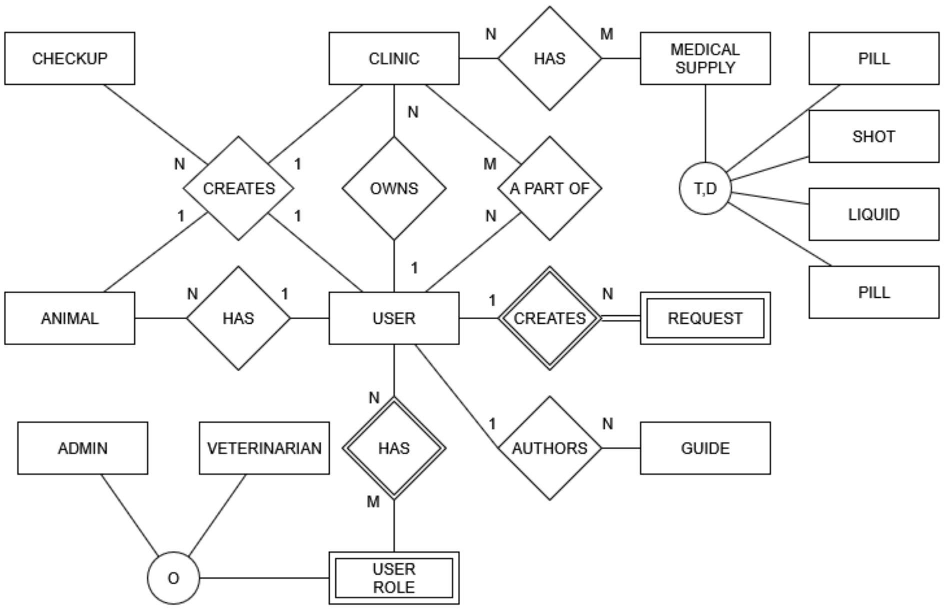


Figure 3.2: Relational model overview

Given the system's core requirements and the need for a reliable user management structure, the foundational entity developed was the `user`, as shown in Figure 3.3

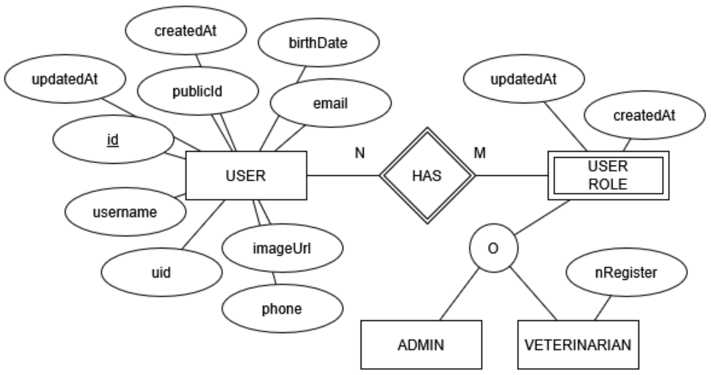


Figure 3.3: User relational model

A noteworthy aspect of the `user` entity is its use of three distinct identifiers, each serving a unique purpose:

- **ID** – The internal primary key used for relational mappings and database operations such as JOINs.

- **UID** – The unique Firebase identifier associated with the user, used for authentication and external identity resolution.
- **Public ID** – A UUID exposed for safe external access, enabling public or limited-scope profile fetching without revealing internal identifiers.

Due to time constraints and the need for scalable role management, a normalized many-to-many (N:M) relationship was implemented between users and roles. Each role can optionally be extended through a dedicated table containing additional attributes. While this approach introduces some complexity—such as additional joins, it ensures type safety and allows for role-specific validation rules to be enforced in the service layer.

In parallel, a `roles` array column was included in the `users` table. Although this introduces a degree of redundancy, it facilitates simplified access and verification in certain application workflows. While not ideal in strict normalization terms, this compromise offers practical advantages given the project's constraints.

An alternative design considered was the use of a single `roles` table (containing metadata such as name and description), accompanied by a role-specific data table keyed by a shared identifier. However, this approach would complicate data validation, as it would require runtime parsing based on role type, reducing type safety and increasing the potential for runtime errors.

To manage elevation requests (*e.g.*, when a user requests a higher-privilege role), a `request` entity was introduced. Each request captures the intended action (*e.g.*, CRUD operations), the target entity, the request's current status (*e.g.*, pending, approved, rejected), and a justification for the decision. Supporting fields such as `files` (for uploaded documents) and `extra_data` (a JSON object for dynamic, form-related data) provide the necessary flexibility for more complex workflows.

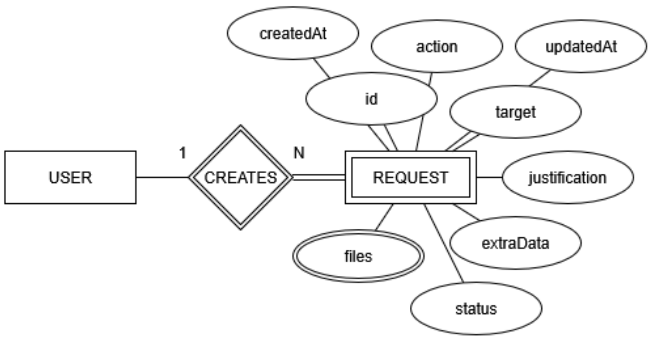


Figure 3.4: User requests relational model

Approval of requests is delegated to users with the appropriate roles, as determined and validated in the service layer.

The `animal` entity was introduced to represent individual animals in the system, capturing essential identification and medical tracking attributes.

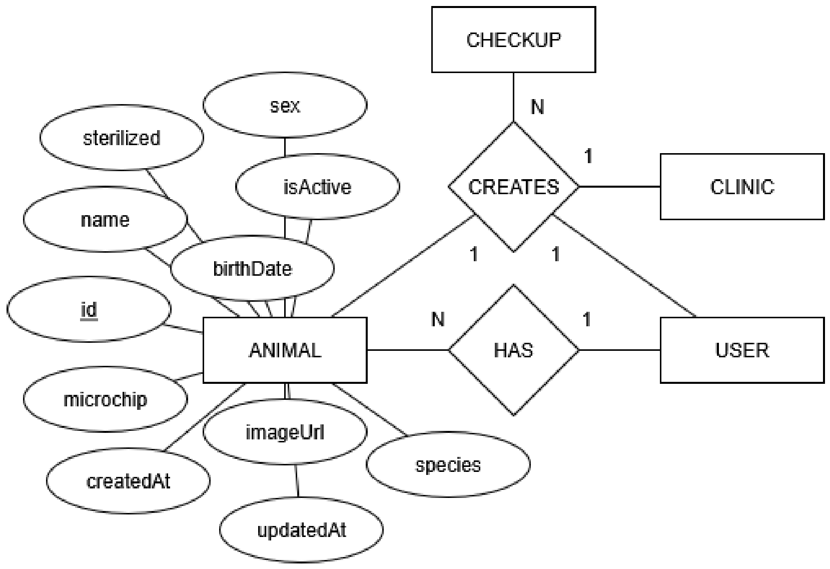


Figure 3.5: Animal relational model

Each animal may be optionally identified by a `microchip`, which is unique but not required. Due to international differences in chip formatting, only a maximum length is enforced. Fields such as `name`, `species`, and `birth_date` are nullable to accommodate edge cases like newborns or undocumented animals.

The `sex` field uses a custom enum type `vetly.sex`, with a default value of `UNKNOWN`. This allows unambiguous handling of cases where sex is not specified or relevant. A boolean `sterilized` flag indicates whether the animal has been neutered or spayed.

Instead of deleting animal records, a boolean `is_active` field supports soft-deletion. This choice ensures data consistency in historical queries and enables reversible deactivation. The `image_url` field stores a reference to a profile image; while simple, this could be enhanced in future iterations via a dedicated asset management table.

Animals are referenced by checkups in conjunction with users and clinics to identify the animal in question for a given checkup, as well as ownership via the `owner_id` field. The absence of cascading delete ensures that animal records persist even if a user is removed from the system.

Another greatly important entity to implement is `clinic`, whose responsibility is to represent as accurately as possible a clinic and its information.

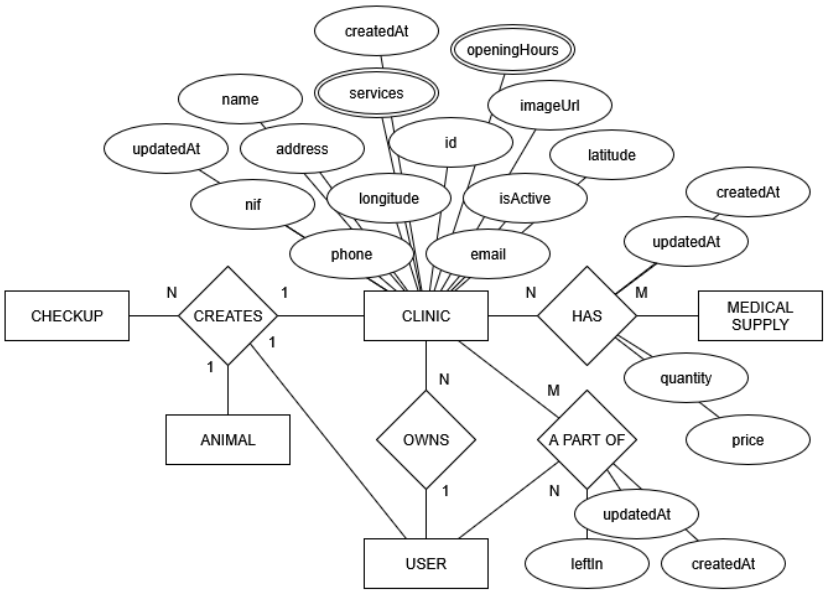


Figure 3.6: Clinic relational model

Most of the attributed fields are relatively basic, as shown in Figure 3.6, however others had to have some thought. For example, `nif` is a variable length string and doesn't have any constraint necessarily. In this scenario, it is enough, but it could be enhanced by taking other use cases into consideration. Other pieces of information such as `services` and `opening_hours` were set as multi-value due to allocating lists and/or lists of objects, respectively. Just like with the `animal` table, an `is_active` field is used to soft delete data, as to keep database consistency and search.

The `clinic` entity is referenced by several others to support core application features. It is associated with the `checkup` entity to indicate where a medical appointment takes place, in conjunction with the associated `animal` and attending `user` (specifically someone with a veterinarian `role`). It also relates to the `user` entity to distinguish between clinic owners and employees. Furthermore, the `medical_supply` entity links to `clinic` to enable inventory tracking and supply availability per location.

The checkup entity models a veterinary consultation or examination. It captures essential data for both administrative tracking and clinical history, making it a key part of the system's relational design.

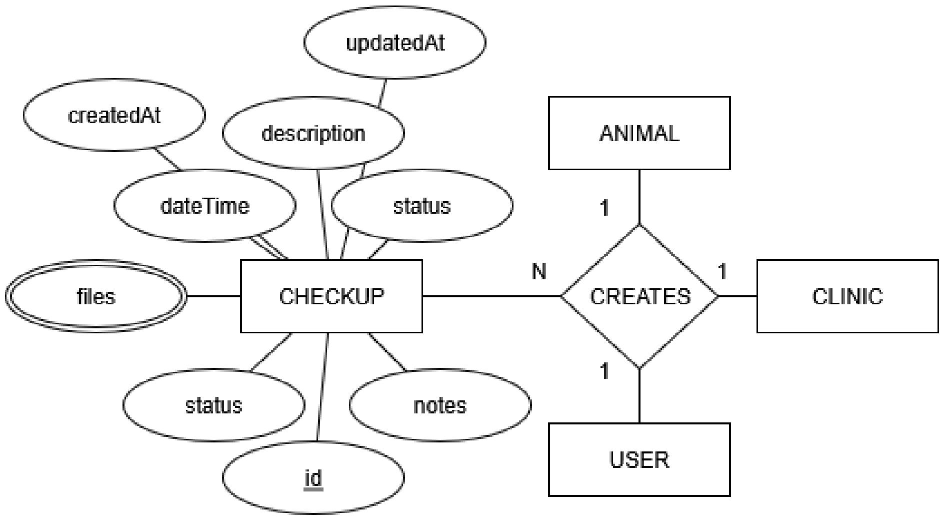


Figure 3.7: Checkup relational model

As shown in Figure 3.7, a checkup is associated with a single animal, a clinic, and a user acting as veterinarian. These relationships are defined through foreign keys with ON DELETE CASCADE semantics to maintain referential integrity.

Attributes such as title and description are constrained in length to promote concise input and maintain UI clarity. The status field uses the vetly.checkupStatus enum, allowing lifecycle state management (*e.g.*, SCHEDULED, COMPLETED, CANCELLED). The notes field stores optional internal commentary or procedural outcomes.

Previously, a separate files table was present to support metadata storage for each file. The complexity cost of this proved to be more than its worth, so it was replaced with a simpler list approach. It can be implemented in the future if the need for extra information proves to be worth the extra overhead.

The `medical_supply` entity models equipment, pharmaceuticals, or other consumable materials used by clinics, forming the foundation of the system's inventory management capabilities.

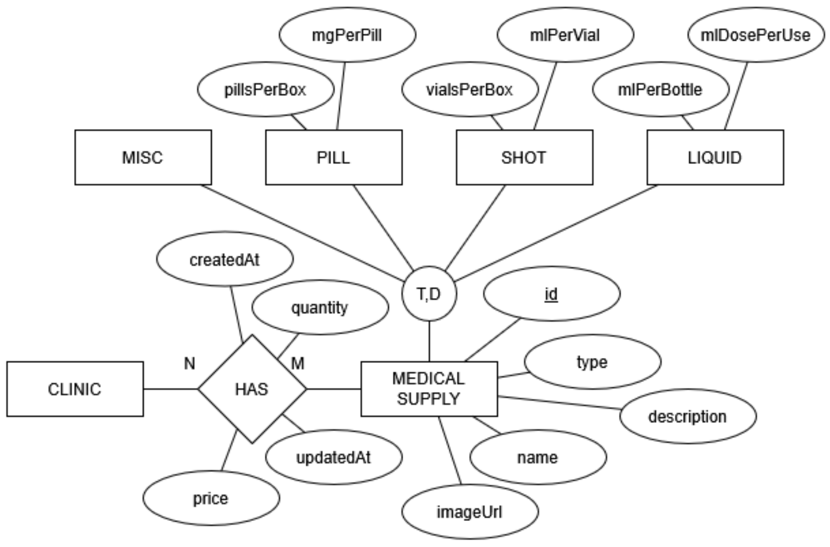


Figure 3.8: Medical supply relational model

Due to the variety of categories an item can have, the most basic were created, while a `MISC` type serves the rest. This can be expanded and can be implemented in a more dynamic form using, for example, JSON-representing columns, however, due to time constraints and importance of data consistency, this approach was used.

As illustrated in Figure 3.8, each supply record includes core fields such as `name` and `description`, as well as an optional `imageUrl` for visual identification. A unique UUID is assigned to every supply item at creation, providing a secure, opaque identifier for client-server communication.

The `type` field represents the supply category (*e.g.*, `EQUIPMENT`, `PHARMACEUTICAL`), enabling classification and easier filtering of inventory. To handle uncategorized items, a `MISC` type is available. This simple approach was chosen for its clarity and consistency enforcement, despite introducing some redundancy. A more flexible alternative, such as using a JSON column for dynamic attributes, was considered but deprioritized due to time constraints and the increased complexity it would add to validation and querying.

The `medical_supply` entity establishes a many-to-many relationship with `clinic` through the `medical_supplies_clinics` linking table. This relationship allows multiple clinics to share the same supply definitions while customizing quantities and prices per clinic. The linking table includes additional attributes such as price and quantity, defined when added.

3.3 Database Access

Spring [13] provides built-in support for managing configuration through environment-specific property files. This feature was leveraged to simplify the database connection process by defining key properties within the corresponding configuration files.

Spring [13] applies a hierarchical configuration strategy. The base file, `application.properties`, is always loaded regardless of the active profile (*e.g.*, `dev`, `test`, `prod`), while profile-specific files such as `application-prod.properties`, as shown in Figure 3.1, or `application-dev.properties` override or extend the base configuration as needed. This design encourages a clean separation of concerns and allows for consistent database access across environments.

```
server.port=8080
spring.datasource.url=jdbc:postgresql://${POSTGRES_HOST}:${
    ↪ POSTGRES_PORT}/${POSTGRES_DB}
spring.datasource.username=${POSTGRES_USER}
spring.datasource.password=${POSTGRES_PASSWORD}
spring.jpa.open-in-view=false
```

Listing 3.1: `application-prod.properties` file

3.4 Repository

For data persistence, the project utilizes Java Persistence API in combination with PostgreSQL [3] as the underlying relational database. To streamline repository management and reduce boilerplate code, Spring Data JPA [12] was integrated.

Spring Data JPA [12] builds upon the standard JPA specification and provides a repository abstraction layer that enables automatic query generation based on method naming conventions. Custom queries, where needed, can still be defined using JPQL (Jakarta Persistence Query Language) or native SQL. This abstraction enables rapid development, consistent structure, and easier maintenance of the data access layer, while still offering the flexibility required for more complex operations.

Entities are mapped to the database using JPA annotations such as `@Entity` [14] and `@Table` [15]. Repositories are defined through interfaces annotated with `@Repository` [16], which may extend `JpaRepository` [17], as shown in 3.2. Where necessary, queries are customized using the `@Query` [18] annotation or named queries.

```
@Repository
interface UserRepository : JpaRepository<User, Long> {
    // Custom query methods
}
```

Listing 3.2: User repository interface

If paging is necessary from this entity, the interface would also extend `JpaSpecificationExecutor` [19], as shown in 3.3.

```
@Repository
interface CheckupRepository : JpaRepository<Checkup, Long>,
    ↪ JpaSpecificationExecutor<Checkup> {
    // Custom query methods
}
```

Listing 3.3: Checkup repository interface

3.5 Services

Each controller is backed by a corresponding `@Service` [20] annotated class. These service classes encapsulate the business logic of the application, ensuring separation of concerns between web-facing controllers and the internal application processes, as shown in Listing 3.4.

By adhering to the principle of single responsibility, services remain modular and easier to test. Through Spring's [13] dependency injection mechanism, services are injected with one or more `@Repository` [16] components, which handle data access logic. This enables a clean layering where services act as intermediaries between controllers and the persistence layer.

Testing is also simplified, as services can be tested in isolation by mocking repository dependencies. This modularity and dependency injection architecture contribute to maintainable, readable, and scalable code.

```
@Service
class UserService(
    private val userRepository: UserRepository
    private val userRoleRepository: UserRoleRepository,
    private val roleRepository: RoleRepository,
) {
    // User-related services
}
```

Listing 3.4: User service class

3.6 Controller

Controllers are responsible for handling incoming HTTP requests, acting as the entry point for application logic. In this architecture, each controller is paired with a dedicated interface, shown in Listing 3.5. This interface defines the method signatures and is annotated using Spring Doc [21] to automatically generate comprehensive API documentation.

```
@Tag(name = "User")
interface UserApi {
    @Operation( ... )
    @ApiResponses( ... )
    @GetMapping(GET)
    fun getUserProfile(
        @PathVariable userId: UUID,
    ): ResponseEntity<UserInformation>
}
```

Listing 3.5: User controller interface

The actual controller class implements the interface and is annotated with security and routing-related annotations (*e.g.*, `@ProtectedRoute`, `@RequestMapping`), as shown in Listing 3.6. This separation of concerns helps maintain clarity: documentation and method declarations are isolated from request-handling logic and authentication policies.

```
@RestController
class UserController(
    private val userService: UserService,
) : UserApi {
    // Implementation of the UserApi interface
}
```

Listing 3.6: User controller class

This structure supports:

- Clear, centralized API documentation through Spring Doc annotations at the interface level.
- Explicit and testable controller implementations that handle security and routing.
- Easier API maintenance and readability, since method contracts are defined in one place and behavior in another.

Additionally, this design pattern enables alternative controller implementations or testing stubs, while maintaining consistent API contracts.

3.7 Authentication

Due to the limitations of cookie-only authentication in certain environments (*e.g.*, mobile apps or APIs without browser context), the system supports two complementary authentication methods:

- **Authorization Bearer Token** – This method uses Firebase-issued ID tokens, typically retrieved during sign-in on mobile or single-page apps. The token is a short-lived JWT containing standard Firebase claims, including the user's UID, email, and optionally custom claims such as roles. The token is validated server-side with Firebase's `verifyIdToken` method. Since ID tokens are designed to be presented in the HTTP Authorization header, they work well for stateless REST APIs or clients where cookies are unavailable.

- **Session Cookie** – This method issues a long-lived JWT (the session cookie itself), which is stored as an HTTP-only cookie. The cookie is created via Firebase Admin SDK and validated server-side with `verifySessionCookie`. It contains similar claims to the ID token (*e.g.*, UID, email, custom claims) but with a longer expiration (*e.g.*, days vs. hours). This approach supports traditional web applications where maintaining a session through browser cookies is preferable, while still offering JWT-based stateless validation on the server.

Both tokens encapsulate user identity information in a signed JWT, but differ primarily in their intended use: ID tokens (bearer tokens) are optimized for mobile or API contexts, while session cookies are designed for browser-based apps needing persistent, secure sessions.

This dual-mode authentication strategy ensures flexibility across platforms while leveraging Firebase's [4] built-in user identity verification.

Once a user is authenticated, a custom `HandlerInterceptor` is used to enforce authorization. This interceptor is executed before controller logic and performs the following steps:

1. Checks whether the targeted handler method is annotated with either `@AuthenticatedRoute` or `@ProtectedRoute`.

2. If one of the annotations is present, it attempts to extract the authenticated user from the request using the appropriate Firebase [4] verification method.

3. The authenticated user is then attached to the request for downstream access (*e.g.*, in controllers or services).

4. If the method is annotated with `@ProtectedRoute`, the user's roles are checked against the required role.

5. If the role check fails, a `ForbiddenException` is thrown; if no valid token is found, an `UnauthorizedAccessException` is thrown.

```kotlin
@Component
@Profile("prod")
class ProdFirebaseTokenVerifier : FirebaseTokenVerifier {
    override fun verify(token: String): FirebaseToken? {
        return try {
            FirebaseAuth.getInstance().verifySessionCookie(token, true)
        } catch (e: FirebaseAuthException) {
            return try {
                FirebaseAuth.getInstance().verifyIdToken(token)
            } catch (e: FirebaseAuthException) {
                null
            }
        }
    }
}
```

Listing 3.7: Production-facing Firebase token verifier (implementing the corresponding interface)

A notable aspect to consider is the use of the `@Profile` [22] annotation, which specifies the active Spring profile under which the component should be loaded. In the case of Listing 3.7, the `prod` profile is targeted, meaning this implementation is only active in a production environment. Typically, during development, a different profile is used. Since the server operates over HTTP, authentication is commonly handled through an Authorization header containing a Bearer token, which is then verified accordingly. In case of authentication failure via the Session Cookie, it falls back to the Authorization Bearer token.

This setup offers fine-grained, annotation-driven access control with minimal boilerplate, allowing route-level protection without embedding logic into the controller itself. Roles are enforced in a centralized manner, improving maintainability and consistency across the application.

3.8 Global Exception Handling

Spring [13] provides the `@ControllerAdvice` [23] annotation, which allows for centralized exception handling across all controllers. By using this mechanism, exceptions thrown during request processing can be intercepted and handled uniformly, enabling consistent error responses and reducing boilerplate code in individual controllers. This approach also supports pre-processing logic for specific exception types, improving the clarity and maintainability of error-handling logic across the application.

Most of the exceptions thrown in the application are custom-defined to better reflect the context of the error. These exceptions encapsulate relevant details such as the HTTP status code, an error message, and the specific type of error being reported, shown in Listing 3.8, where it is treating the VetException family of exceptions.

```
@ControllerAdvice
class GlobalExceptionHandler {
    private val logger = LoggerFactory.getLogger(this::class.java)

    @ExceptionHandler(VetException::class)
    fun handleVetExceptions(ex: VetException): ResponseEntity<ApiError> {
        // ...
    }
}
```

Listing 3.8: Global exception handler

3.9 Testing

To centralize and reuse common test logic and data, base classes were created for both unit and integration testing. The unit test base class provides mock data and configurations, while the integration test base sets up mocked repositories and relevant application contexts.

Unit tests were written for controllers, interceptors, and JSON (JavaScript Object Notation) serializers using `JUnit` [24] and `MockK` [25], ensuring isolated validation of component behavior through mocking. For integration testing, the `MockMvc` [26] framework was used to simulate HTTP requests, and the application was configured to run with an in-memory `H2` [27] database managed by `Hibernate` [28]. This allowed for realistic end-to-end testing of the service layer and repository interactions, while maintaining speed and isolation from the production environment.

3.10 Deployment

The deployment process was streamlined using Docker [10]. A Dockerfile was created to containerize the application, ensuring that it runs consistently across different environments. This container encapsulates all dependencies and runtime configurations required to run the application. Additionally, docker-compose was used to manage multi-container setups, such as combining the application with a database service, which facilitates both local development and production deployment. This approach simplifies environment management, enhances portability, and reduces the risk of configuration-related issues during deployment.

Chapter 4

Frontend

4.1 Structure

The frontend is developed using Expo [5], a framework for building cross-platform applications with React Native [29]. One of the key architectural tools used is Expo Router [30], which brings file-based routing to React Native, similar to frameworks like Next.js [31].

The project follows a modular folder structure:

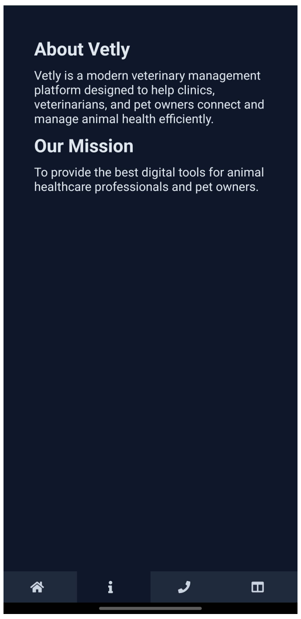
- `app/` – Contains the main route definitions. Each file or folder within `app/` maps to a screen or nested navigator.
- `components/` – Houses reusable UI (User Interface) components, such as buttons, headers, and modals.
- `hooks/` – Includes shared logic like authentication or data fetching (*e.g.*, `useAuth`, `useResource`).
- `api/` – Contains all service logic, including API calls made via Axios [32].
- `assets/` – Stores static files like icons, fonts, and images.
- `libs/` - Contains library logic and configuration like Firebase's.
- `handlers/` - Defines global request and/or error handlers for user-friendly viewing.

This structure provides a scalable and maintainable foundation for a cross-platform application, allowing clean separation of UI, logic, and navigation.

Note: Testing on iOS devices requires access to Apple hardware. Since the development team did not have an Apple device available during the testing phase, correct behavior on iOS cannot be fully guaranteed.

4.2 User Interface

The multi-platform nature of the project meant that users would have potentially different interface layouts to adapt to their device of choice, but with some similarities between them in one way or another.

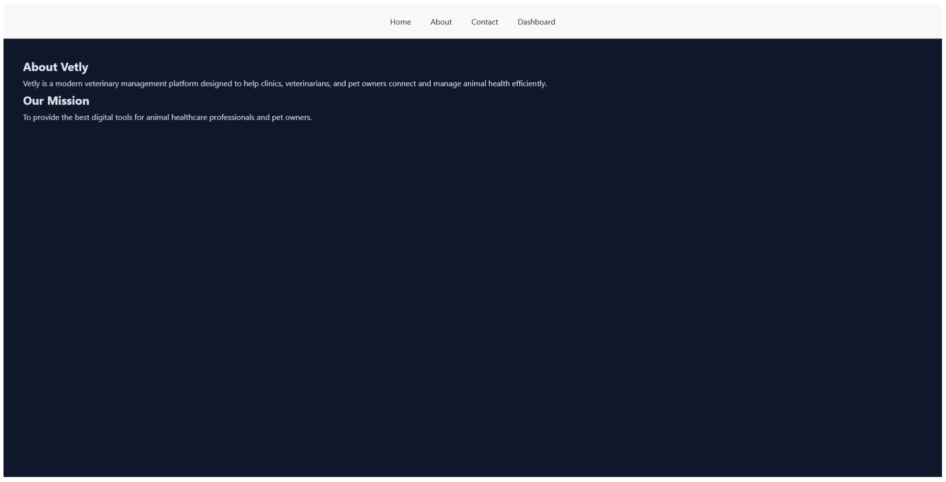

About Vetly

Vetly is a modern veterinary management platform designed to help clinics, veterinarians, and pet owners connect and manage animal health efficiently.

Our Mission

To provide the best digital tools for animal healthcare professionals and pet owners.

Figure 4.1: Mobile interface (on the public about page)



About Vetly

Vetly is a modern veterinary management platform designed to help clinics, veterinarians, and pet owners connect and manage animal health efficiently.

Our Mission

To provide the best digital tools for animal healthcare professionals and pet owners.

Figure 4.2: Web interface (on the public about page)

For example, the mobile version uses Expo Router's Tabs navigator [33], as shown in Figure 4.1, allowing quick switching between main public routes, while a common, top navigation bar was used for desktop, shown in Figure 4.2. Both allow the same routes with a different interface.

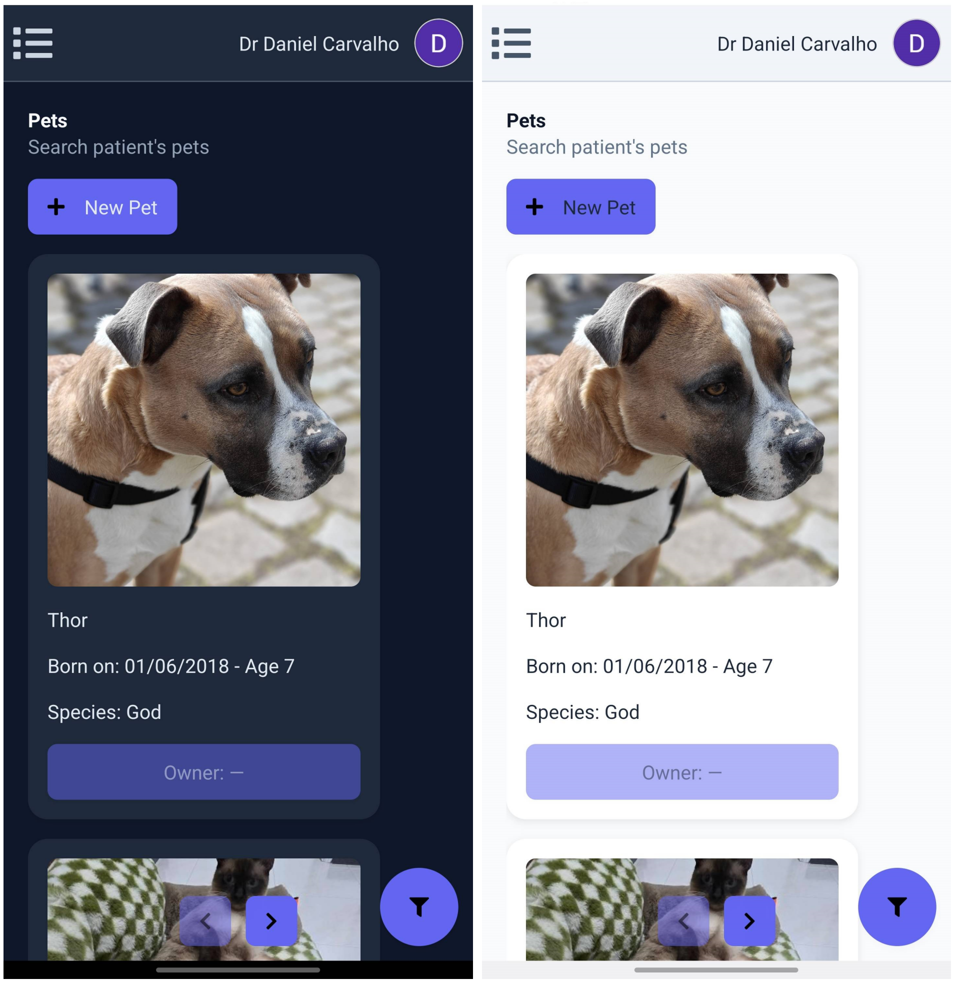
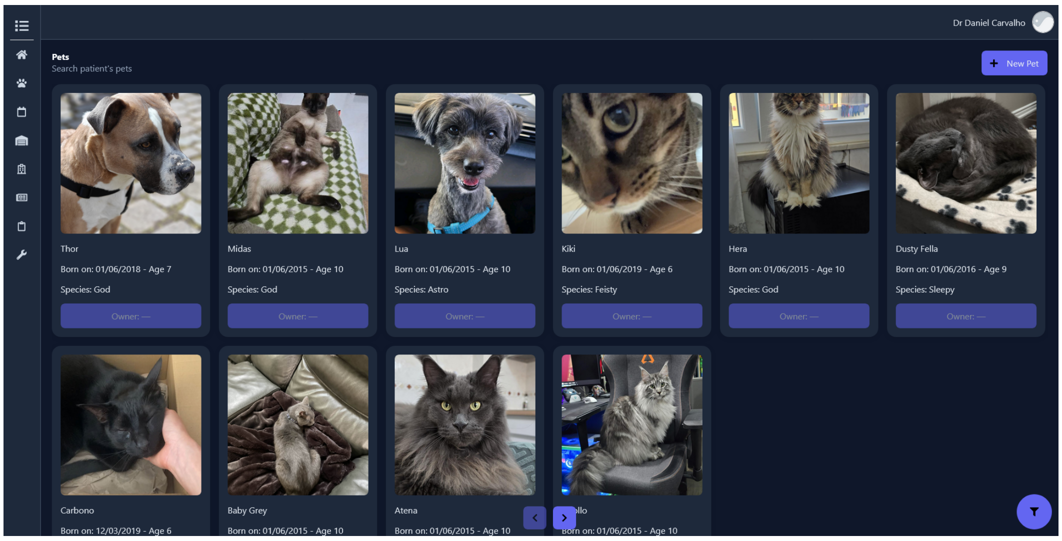


Figure 4.3: Mobile interface (on the private pet route)



Dr Daniel Carvalho
Pets
Search patient's pets
New Pet
Thor
Born on: 01/06/2018 - Age 7
Species: God
Owner: —
Midas
Born on: 01/06/2015 - Age 10
Species: God
Owner: —
Lua
Born on: 01/06/2015 - Age 10
Species: Astro
Owner: —
Kiki
Born on: 01/06/2019 - Age 6
Species: Feisty
Owner: —
Hera
Born on: 01/06/2015 - Age 10
Species: God
Owner: —
Dusty Fella
Born on: 01/06/2016 - Age 9
Species: Sleepy
Owner: —
Carbono
Born on: 12/03/2019 - Age 6
Baby Grey
Born on: 01/06/2015 - Age 10
Atena
Born on: 01/06/2015 - Age 10
llo
Born on: 01/06/2015 - Age 10

Figure 4.4: Web interface (on the private pet route)

Every main route is essentially composed of a single `BaseComponent`, responsible for checking the authentication status of the user as well as route accessibility and, if valid, rendering the correct children and a `PageHeader` component that renders operation buttons (like deleting a resource) with a brief title and description.

In an authenticated context, a drawer navigator from Expo Router [34] is used for both platforms due to its unified behavior on both mobile and web. The drawer is fully hidden on mobile, shown in Figure 4.4, while on web the route icons are still shown to make the navigating flow smoother, seen in Figure 4.3 and 4.5. This economizes screen space, which is critical for smaller devices, but also takes advantage of big displays.

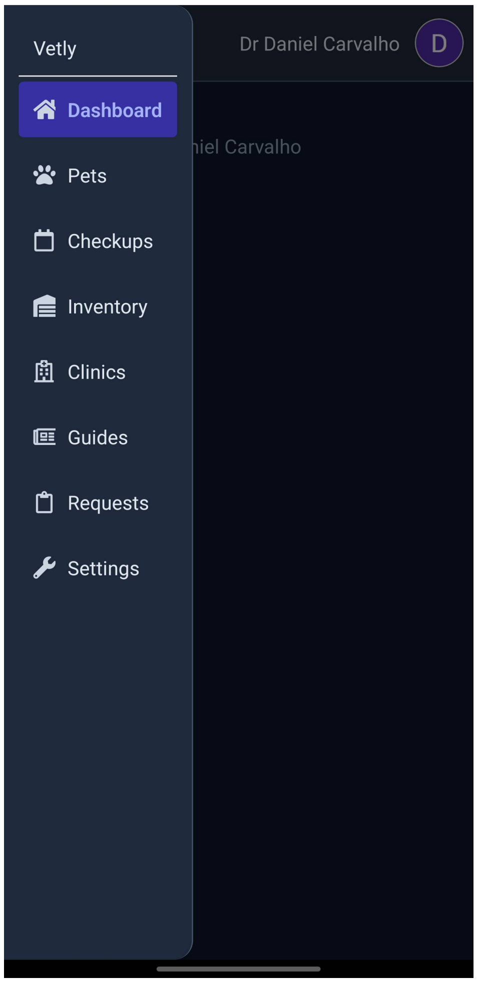


Figure 4.5: Mobile drawer

Unfortunately, due to Expo SDKs [5] compatibility and instability, features like protected routes [35] had to be skipped and custom solutions built from scratch. This added an extra layer of complexity, needing to verify the users' access based on a static list of permissions and their roles.

4.3 Routes

Routing is implemented using Expo Router [30], which uses a file-based approach inside the `app/` directory. Each file corresponds to a screen, and folders can be used to define nested navigation stacks.

For example:

- `app/(public)/index.tsx` – Represents the root screen (home page).
- `app/(public)/login.tsx` – Represents the login screen.
- `app/(private)/` – A layout group used to organize routes that require authentication.

Expo Router [30] also supports layout routes, which allow persistent UI elements (like tab bars or headers) to be shared across grouped screens. Route grouping is used to separate authenticated and unauthenticated areas of the app.

Authentication is enforced using layout middleware or conditionally rendered routes. For example, a guard can be implemented using a custom hook such as `useAuth` to redirect unauthenticated users to the login screen.

4.4 Services

As in the backend, each controller has a corresponding frontend service that communicates with it via predefined paths. The project uses the Axios [32] library, which reduces boilerplate HTTP code and enables global request and response interceptors. These interceptors play a crucial role in the authentication flow, allowing centralized handling of cookies, tokens, or both, depending on the platform and environment.

To single out responsibilities per main module, each was split into 3 types of files:

- `*.api.ts` - Stores the actual methods that call the API using the types in the respective `.input.ts` and `.output.ts`.
- `*.input.ts` - Stores the input types of a given path (*e.g.* `AnimalCreate` used in the request to create an animal).
- `*.output.ts` - Stores the output types of a given path (*e.g.* `AnimalInformation` when fetching an animals' details).

This approach was applied to every main entity previously created, allowing a maintainable and scalable structure.

4.5 Components

Components are organized into reusable building blocks to promote consistency and reduce duplication across the app. The `components/` directory contains elements such as buttons, text inputs, cards, and modals, which are used across multiple screens.

Each component is designed to be reusable, controlled via props, and platform-agnostic when possible. Styling [36] is handled using `StyleSheet` from React Native [29], enhanced by utility helpers and shared theme constants to ensure visual consistency.

The app avoids tight coupling by keeping components stateless and isolated from navigation logic or API calls. Where necessary, responsive behavior (*e.g.*, different paddings or input modes) is handled using React Native's [29] `Platform` [37] module or media queries via `react-native-responsive-dimensions` [38].

Layout components (*e.g.*, containers with consistent padding or headers) are also reused to maintain a coherent layout across screens.

4.5.1 Hooks

In line with common React [39] development patterns, a custom `useAuth` hook was implemented to encapsulate authentication logic, including `signIn`, `signOut`, and session verification. This centralization facilitates consistent access to the authentication state across the application.

However, cross-platform constraints introduced complications. Some authentication libraries are not universally supported, requiring platform-specific handling within the hook logic.

A reusable `useResource` hook was developed to handle asynchronous resource loading. This ensures safe data fetching by accounting for potential null returns and reducing repetitive code in components.

To maintain consistency across the applications' styling, a simple `useThemedStyles` hook is used, providing the colour scheme and layout of various containers and nodes.

4.5.2 Platform Specific

Authentication

Implementing authentication required careful consideration of platform-specific constraints. On mobile platforms, cookies are not reliably supported due to limitations in native HTTP clients and sand-boxed environments. As a result, mobile authentication is handled via bearer tokens, which offer a lightweight and maintainable authorization mechanism.

Conversely, web clients operating over HTTPS (Hypertext Transfer Protocol Secure) can safely use cookies for session management, benefiting from secure, HTTP-only flags and automatic transmission. However, during development (`dev` mode), tokens are used instead, since HTTPS certificates are often not configured in local development environments.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The final state of the software meets the defined functional requirements and has been tested within the available timeframe.

However, time constraints were the primary limiting factor for implementing several planned optional features. These features varied in complexity, with some requiring significantly more development and testing effort than others.

Certain technical decisions, such as the choice of frameworks and programming languages, were made either based on personal preference or to streamline the development process. While these decisions supported faster progress, they may not represent the optimal choices for future iterations or scaling.

5.2 Future Work

Future work should focus on completing the planned optional features, improving test coverage, and revisiting architectural choices to better align with long-term maintainability, scalability, and performance goals.

References

[1] JetBrains. *Kotlin Language Documentation.* [Online; accessed 27-May-2025]. 2024. URL: `https://kotlinlang.org/docs/home.html`.

[2] Spring Team. *Spring Boot Documentation.* [Online; accessed 27-May-2025]. VMware. 2024. URL: `https://docs.spring.io/spring-boot/docs/current/reference/html/`.

[3] PostgreSQL Global Development Group. *PostgreSQL 16 Documentation.* [Online; accessed 27-May-2025]. 2024. URL: `https://www.postgresql.org/docs/`.

[4] Google Inc. *Firebase Documentation.* [Online; accessed 27-May-2025]. 2024. URL: `https://firebase.google.com/docs`.

[5] Expo Contributors. *Expo Documentation.* [Online; accessed 27-May-2025]. 2024. URL: `https://docs.expo.dev/`.

[6] Otto Veterinary Software. *Otto Flow. Veterinary workflow software for modern practices.* URL: `https://www.otto.vet/` (visited on 03/05/2025).

[7] Vetmanager LLC. *Vet Manager. Veterinary practice management software for clinics and hospitals.* URL: `https://www.vetmanager.com/` (visited on 03/05/2025).

[8] ezyVet Limited. *ezyVet. Cloud-based veterinary practice management software.* URL: `https://www.ezyvet.com/` (visited on 03/05/2025).

[9] Vetspire Inc. *Vetspire. Veterinary EMR and AI-powered practice management platform.* URL: `https://www.vetspire.com/` (visited on 03/05/2025).

[10] Docker Inc. *Docker Documentation.* [Online; accessed 27-May-2025]. 2024. URL: `https://docs.docker.com/`.

[11] Microsoft. *Typescript Language Documentation.* [Online; accessed 27-May-2025]. 2024. URL: `https://www.typescriptlang.org/docs/`.

[12] Spring Team. *Spring Data JPA Documentation.* [Online; accessed 27-May-2025]. VMware. 2024. URL: `https://docs.spring.io/spring-data/jpa/reference/jpa.html`.

[13] Spring Team. *Spring Documentation.* [Online; accessed 27-May-2025]. VMware. 2024. URL: `https://docs.spring.io/spring-framework/reference/index.html`.

[14] Jakarta Persistence API. *@Entity annotation.* `https://jakarta.ee/specifications/persistence/`. Accessed: 2025-06-02. 2024.

[15] Jakarta Persistence API. *@Table annotation.* `https://jakarta.ee/specifications/persistence/`. Accessed: 2025-06-02. 2024.

[16] Spring Framework. *@Repository annotation.* `https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Repository.html`. Accessed: 2025-06-02. 2024.

[17] Spring Data JPA. *JpaRepository interface.* https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html. Accessed: 2025-06-02. 2024.

[18] Spring Data JPA. *@Query annotation.* https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.at-query. Accessed: 2025-06-02. 2024.

[19] Spring Data JPA. *JpaSpecificationExecutor interface.* https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaSpecificationExecutor.html. Accessed: 2025-06-02. 2024.

[20] Spring Framework. *@Service annotation.* https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Service.html. Accessed: 2025-06-02. 2024.

[21] SpringDoc Contributors. *SpringDoc OpenAPI Documentation.* [Online; accessed 27-May-2025]. 2024. URL: https://springdoc.org/.

[22] Spring Framework. *@Profile annotation.* https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Profile.html. Accessed: 2025-06-02. 2024.

[23] Spring Framework. *@ControllerAdvice annotation.* https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ControllerAdvice.html. Accessed: 2025-06-02. 2024.

[24] JUnit Team. *JUnit 5 User Guide.* [Online; accessed 27-May-2025]. 2024. URL: https://junit.org/junit5/docs/current/user-guide/.

[25] MockK Contributors. *MockK Documentation.* [Online; accessed 27-May-2025]. 2024. URL: https://mockk.io/.

[26] Spring Team. *MockMvc - Spring Framework.* [Online; accessed 27-May-2025]. 2024. URL: https://docs.spring.io/spring-framework/reference/testing/spring-mvc-test-framework.html.

[27] Thomas Mueller. *H2 Database Engine Documentation.* [Online; accessed 27-May-2025]. 2024. URL: https://www.h2database.com/html/main.html.

[28] Hibernate Team. *Hibernate ORM Documentation.* [Online; accessed 27-May-2025]. 2024. URL: https://hibernate.org/orm/documentation/.

[29] Meta Platforms Inc. *React Native Documentation.* [Online; accessed 27-May-2025]. 2024. URL: https://reactnative.dev/docs/getting-started.

[30] Expo Documentation Team. *Expo Router. Declarative routing for React Native apps.* URL: https://docs.expo.dev/router/introduction/ (visited on 06/02/2025).

[31] Vercel. *Next.js Documentation.* https://nextjs.org/docs. Accessed: 2025-06-02. 2024.

[32] Axios. *Axios Documentation.* https://axios-http.com/docs/intro. Accessed: 2025-06-02. 2024.

[33] Expo Documentation Team. *Tabs Navigator. Expo Router guide to creating tab-based navigation.* URL: https://docs.expo.dev/router/tabs/ (visited on 06/02/2025).

[34] Expo Documentation Team. *Drawer Navigator. Expo Router guide to creating drawer-based navigation.* URL: https://docs.expo.dev/router/drawer/ (visited on 06/02/2025).

[35] Expo Documentation Team. *Protected Routes. Expo Router guide to making screens inaccessible to client-side navigation.* URL: `https://docs.expo.dev/router/advanced/protected/` (visited on 06/02/2025).

[36] Meta Platforms, Inc. *React Native Documentation.* Accessed: 2025-06-02. 2025. URL: `https://reactnative.dev/docs/stylesheet`.

[37] React Native. *Platform module.* `https://reactnative.dev/docs/platform`. Accessed: 2025-06-02. 2024.

[38] Nishan Jalal. *react-native-responsive-dimensions.* `https://github.com/nishan7/react-native-responsive-dimensions`. Accessed: 2025-06-02. 2024.

[39] Meta. *React – A JavaScript library for building user interfaces.* `https://react.dev`. Accessed: 2025-06-02. 2024.