Software Requirements Specification

for

VetCare - Online Vet Clinic Management System

Version 1.0 approved

Prepared by Group-P09-03

Developers:
Aphisith Siphaxay [s3987059]
Ashmit Sachan [s3873827]
Henry Van Toledo [s3849054]
Kai Hei Kong [s3971187]
Kaiyang Zheng [s3992987]
Preeti Goel [s3879991]

20/08/2024

Table of Contents

Revision History

Name	Date	Reason For Changes	Version
Milestone 1	20/08/2024	Initial Software Requirements Specification	V1.0.0
Milestone 2	22/09/2024	Sprint 1 Application Implementation	V2.0.0
Milestone 3	13/10/2024	Sprint 2 Application Implementation and additional stories/users	V3.0.0

1. Introduction

1.1 Purpose

This document specifies the software requirements for the VetCare Online Vet Clinic Management System, version 1.0. The VetCare system is designed to facilitate pet owners in managing their pets' health by providing features such as appointment scheduling, access to medical records, prescription management, and educational resources and real time appointment notifications.

The scope of this SRS covers the entire VetCare system, including all major features and functionalities that will be delivered in version 1.0. This includes the full integration with veterinary clinics and stores, user interface design, back-end services, and the database schema. The document outlines both functional and non-functional requirements, ensuring that the system meets the needs of end-users and stakeholders.

This SRS does not cover future enhancements or versions beyond 1.0, nor does it address any hardware or infrastructure specifics beyond those necessary to support the software application.

1.2 Document Conventions

This document follows standard SRS conventions and adheres to the following guidelines:

Headings and Subheadings: Key sections are highlighted using distinct headings and subheadings to ensure the document is easy to navigate.

Numbering: All requirements are numbered sequentially and organized according to their priority. All requirements have their own priority.

Font and Style: The document uses a consistent font (e.g., Times New Roman, 12 pt) for the main text, with bold used for section headings and italics used for emphasis or when referring to other documents, external systems, or components.

Requirements Inheritance: It is assumed that priorities for higher-level requirements are inherited by detailed sub-requirements unless otherwise specified.

Terminology: Technical terms and acronyms are defined in the glossary (Appendix A). First use of an acronym is followed by its full form in parentheses.

References: All external documents, standards, and resources are referenced in the "References" section and are italicized when mentioned in the text.

1.3 Intended Audience and Reading Suggestions

This document is intended for the following audiences:

- Developers
- Product Owner
- Product User
- Stakeholders
- Veterinarians and Clinic Administrators

Reading Suggestions:

Readers are advised to begin with the "*Introduction*" to gain an understanding of the document's purpose, scope, and intended use. The "Overall Description" provides a comprehensive overview of the system, which is useful for all audiences. Depending on their specific role:

Product Owner should proceed to read through the whole document. Emphasis on Product Perspective is suggested

Developers should proceed to "*External Interface Requirements*" , "*Non Functional Requirements*", "*Other Requirements*", "*System Architecture*" and "*User Interface Design*" to dive into the technical details.

Project Managers should focus on "*Nonfunctional Requirements*" and "*Other Requirements*" to understand the project's constraints.

Testers should prioritize "*External Interface Requirements*" and "*Nonfunctional Requirements*" and "*Other Requirements*" for detailed specifications that will guide their testing strategies.

Stakeholders should review "*Introduction*", "*Product Scope*" and "*Other Requirements*"

Veterinarians and Clinic Administrators should focus on "User Interfaces," "Product Functions," and "User Documentation" to better understand how the system will support clinic workflows and improve operational efficiency. Additionally, "Product Scope" will provide a broader view of how the system enhances service delivery and client interactions.

1.4 Product Scope

The "VetCare" project is an all-inclusive web-based veterinary clinic administration system created to greatly improve the efficiency and accessibility of pet care services. Its main goal is to make managing several facets of pet healthcare, like scheduling appointments, accessing medical records, managing prescriptions, and providing instructional materials, easier. The application integrates these services into an easy-to-use platform to make interactions between pet owners and veterinarian clinics simple. This guarantees prompt medical attention and adherence to treatment guidelines, which not only improves

convenience but also leads to improved health outcomes. Additionally, by utilising digital solutions, we are able to increase the outreach of our platform, enhancing user experience within this industry. VetCare's goal is to encourage sustainable practices, through reduction of excessive paperwork and inefficient processes. Additionally, we aim to better veterinary care by increasing accessibility and making processes far more manageable and intuitive.

1.5 References

Ahirav, D (2024), Real-Time Communication with WebSockets: A Complete Guide, DEV, accessed 24 August 2024.
https://dev.to/dipakahirav/real-time-communication-with-websockets-a-complete-guide-32g4

Animal Welfare Standards and Guidelines 2023, State and Territory Legislation, Animal Welfare Standards, accessed 25 August 2024.
https://animalwelfarestandards.net.au/welfare-standards-and-guidelines/state-and-territory-legislation/

Australian Cyber Security Centre (ACSC) (2023), Essential Eight Assessment Guidance Package, Australian Signals Directorate, accessed 24 August 2024.
https://www.cyber.gov.au/about-us/news/essential-eight-assessment-guidance-package.

Australian Direct Marketing Association (ADMA) (n.d.) 3 things you need to know about the Spam Act, accessed August 25, 2024.
https://www.adma.com.au/resources/3-things-you-need-know-about-spam-act

Barot, S (2023), Spring vs Spring Boot: Technical Comparison of Both the Frameworks, AGLOW ID, accessed 24 August 2024.
https://aglowiditsolutions.com/blog/spring-vs-spring-boot/

DNV (n.d.) ISO/IEC 27001 - Information Security Management System (ISMS), DNV, accessed 24 August, 2024.
https://www.dnv.com/services/iso-iec-27001-information-security-management-system-3327

Doglio F (2023), Monolith vs Microservice Architecture: A Comparison. Camunda, accessed 24 August 2024.
https://camunda.com/blog/2023/08/monolith-vs-microservice-architecture-comparison/

International Organization for Standardization (n.d.), ISO/IEC 27001 - Information security management, accessed August 25, 2024. https://www.iso.org/standard/27001

Lumigo (n.d), Containerized Applications: Benefits, Challenges & Best Practices, accessed August 24, 2024.
https://lumigo.io/container-monitoring/containerized-applications-benefits-challenges-best-practices/

NNG(Nielsen Norman Group) (2024) 10 Usability Heuristics for User Interface Design, NNG website, accessed 24 August 2024.
https://www.nngroup.com/articles/ten-usability-heuristics/

Office of Local Government, NSW (n.d.) Microchipping and registration, NSW Government, accessed August 25, 2024.
https://olg.komosionstaging.com/public/dogs-cats/nsw-pet-registry/microchipping-and-registration/

Office of the Australian Information Commissioner (n.d.), About the Notifiable Data Breaches scheme, accessed August 25, 2024.
https://www.oaic.gov.au/privacy/notifiable-data-breaches/about-the-notifiable-data-breaches-scheme

Office of the Australian Information Commissioner (n.d.), Australian Privacy Principles, accessed August 25, 2024. https://www.oaic.gov.au/privacy/australian-privacy-principles

Victorian Government (2018) Veterinary Practice Regulations 2018 (Statutory rule No. 11/2018), accessed 25 August 2024.
https://www.legislation.vic.gov.au/in-force/statutory-rules/veterinary-practice-regulations-2018/001

Wikipedia contributors (n.d.), Terms of service, Wikipedia, The Free Encyclopedia, accessed August 25, 2024. https://en.wikipedia.org/wiki/Terms_of_service

World Intellectual Property Organization (n.d.), Trademarks, WIPO, accessed August 25, 2024. https://www.wipo.int/trademarks/en/

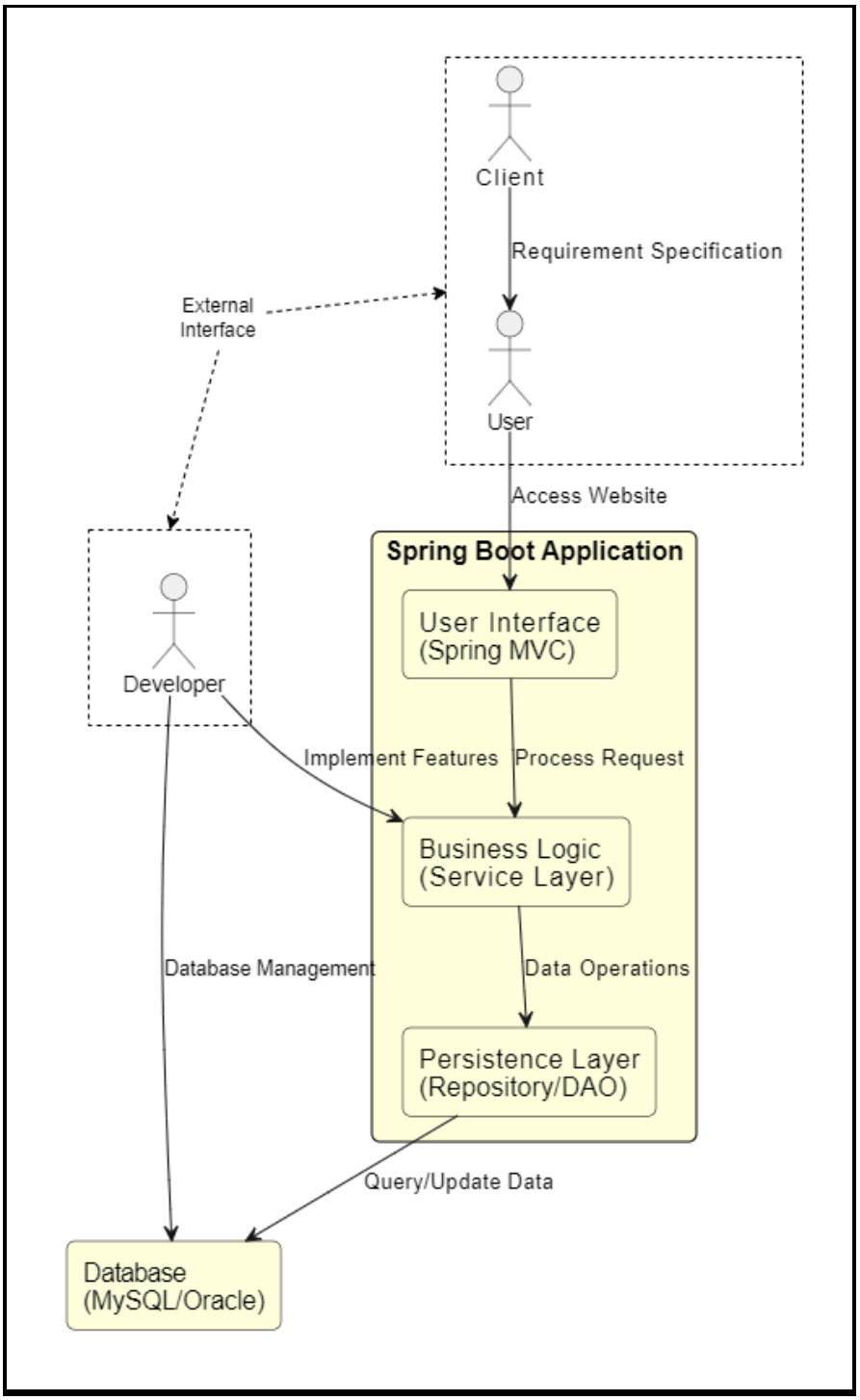
World Intellectual Property Organization (n.d.), Copyright, WIPO accessed August 25, 2024. https://www.wipo.int/copyright/en/

WP Crux (2024) Why MySQL is better than other databases in 2024? accessed August 24, 2024, from https://wpcrux.com/blog/why-mysql-is-better-than-other-databases

2. Overall Description

2.1 Product Perspective

Figure 1 : Product Perspective Diagram



The VetCare Online Vet Clinic Management System is a newly developed, standalone web application. It is not part of an existing product family nor is it a replacement for any previous system. VetCare was created from the ground up to meet the specific needs of modern veterinary practices and pet owners, offering a comprehensive solution for managing pet healthcare.

VetCare is self-contained, managing all operations internally, without reliance on external systems. However, the system is designed with potential future integrations in mind, such as connecting with e-commerce platforms or external veterinary management systems.

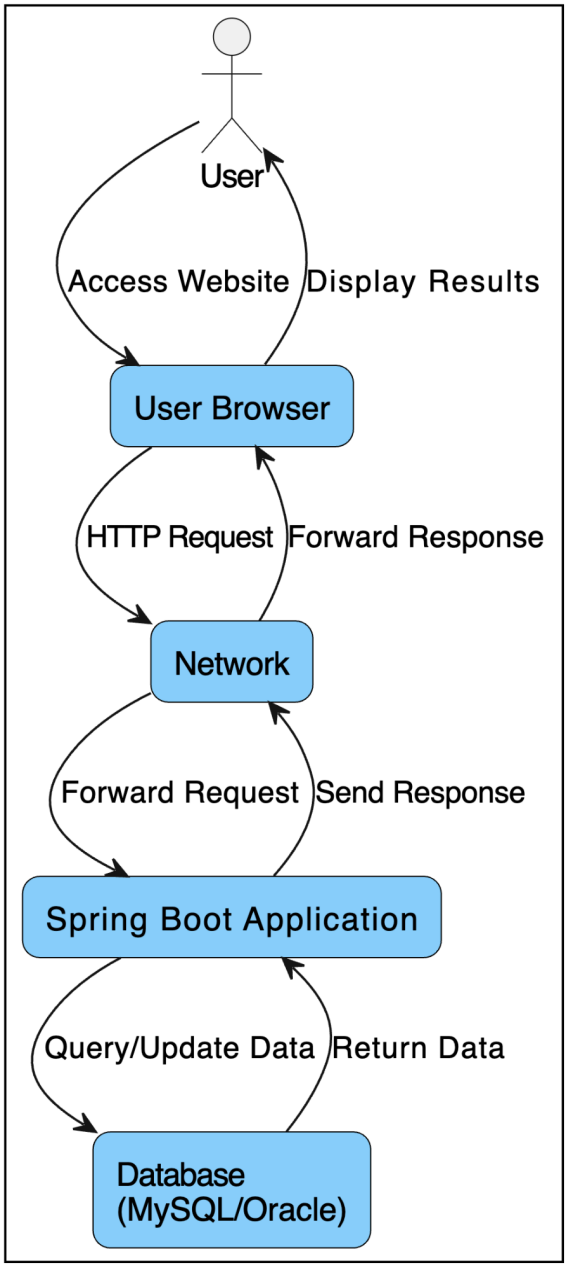
The application is structured into three key layers:

1. **User Interface (UI) Layer:** This layer provides a responsive and intuitive interface for users, including pet owners, veterinarians, and clinic administrators. It facilitates easy interaction with the system's features.
2. **Business Logic Layer:** This is the core of the system, where all the business rules and processes are implemented. It manages functionalities such as appointment scheduling, medical record management, prescription handling, and notification delivery.
3. **Persistence Layer:** This layer handles data storage and retrieval using a MySQL database. It ensures secure and efficient management of all user data, including medical histories, appointment details, and other critical information.

This SRS document covers the requirements for the entire VetCare system as implemented in version 1.0, focusing on the internal functionalities provided by these three layers. Future enhancements and integrations will be considered in subsequent versions of the SRS.

2.2 Product Functions

Figure 2: Top level data flow diagram



The VetCare system is designed to perform a range of functions that facilitate the management of pet healthcare. The major functions include:

Medical Records Management: Securely store and manage comprehensive pet health records, including vaccination histories, treatment plans, and more.

Appointment Scheduling: Allow users to book, reschedule, or cancel veterinary appointments. Automated reminders are sent to ensure that users remember their upcoming appointments.

Prescription Management: Enable pet owners to request additional medication for their pets, with VetCare coordinating the processing and delivery of these prescriptions.

Educational Resources: Provide users with access to a library of articles, videos, and guides on pet care and wellness, helping them stay informed about best practices in veterinary medicine.

Notifications and Reminders: Automatically send notifications to users about upcoming appointments, medication schedules, and other important events related to their pets' healthcare.

These functions are designed to work seamlessly together, ensuring that users can efficiently manage their pets' healthcare needs within a single, integrated system.

2.3 User Classes and Characteristics

In the context of the VetCare Online Vet Clinic Management System, the following user classes have been identified, focusing on the primary users for whom the system is being developed:

1. **Pet Owners**
    - **Usage Frequency:** Pet owners are expected to be the most frequent users of the system, regularly accessing it to manage their pets' healthcare needs.
    - **Primary Functions:** Pet owners will use the system to schedule appointments, view and manage their pets' medical records, request prescription refills, and access educational resources. They will also receive notifications and reminders for appointments and medication schedules.
    - **Technical Proficiency:** The system is designed to be user-friendly, with an intuitive interface that caters to users with basic to intermediate technical skills.
    - **Security and Access Level:** Pet owners have restricted access, limited to their own accounts and their pets' information. They cannot access or modify data related to other users or system settings.
    - **Education and Experience:** The system should be accessible to a wide range of users, with varying levels of education and experience, ensuring ease of use for all.

2. Veterinarians

- **Usage Frequency:** Veterinarians are expected to use the system frequently, especially during clinic hours, to manage patient appointments, update medical records, prescribe treatments, and monitor ongoing care plans.

- **Primary Functions:** Veterinarians will utilize the system to manage patient records, diagnose health conditions, prescribe medications, and monitor treatment plans. The system will also help streamline communication between vets and pet owners by providing tools for appointment management and follow-up care. They will be able to generate reports, access medical history, and handle prescription renewals.
- **Technical Proficiency:** Veterinarians are expected to have moderate technical proficiency, as they will regularly interact with both basic system features (appointment scheduling, record management) and advanced functionalities (medical data analysis, report generation). The system will need to be robust yet simple to accommodate various levels of tech-savviness.
- **Security and Access Level:** Veterinarians will have elevated access rights compared to pet owners. They will have full access to medical records, be able to modify patient health information, and manage appointments. However, they will not have access to system configuration settings or data related to other veterinarians' activities unless assigned administrative roles.
- **Education and Experience:** Veterinarians are highly educated professionals, with a strong background in veterinary medicine. While their primary focus is on patient care, the system should be designed to minimize administrative overhead, allowing them to efficiently use the platform for patient management without requiring extensive technical training.

Clinic Administrators

- **Usage Frequency:** Clinic administrators will use the system regularly to manage operational aspects, such as scheduling, billing, and inventory.
- **Primary Functions:** Administrators will focus on scheduling appointments, managing client accounts, overseeing billing processes, and ensuring the clinic runs smoothly by maintaining an up-to-date system.
- **Technical Proficiency:** Administrators are expected to have intermediate to advanced technical proficiency, particularly in managing clinic operations and ensuring that both vets and clients can easily use the system.
- **Security and Access Level:** Administrators will have administrative access, allowing them to modify clinic-wide settings, manage vet and staff accounts, and oversee data access and system security.
- **Education and Experience:** They are typically experienced in business or healthcare administration, and the system should support seamless management of both the clinical and administrative sides of the practice.

3. Developers (Project Team)

- **Usage Frequency:** As the team responsible for building and maintaining the system, developers interact with the system regularly throughout its development lifecycle.
- **Primary Functions:** Developers are responsible for implementing, testing, and refining the system's features, including the user interface, business logic, and data management.

- **Technical Proficiency:** Developers possess a strong technical background, with skills in programming, database management, and web development.
- **Security and Access Level:** Developers have full access to the system's codebase, databases, and configuration settings, enabling them to make necessary changes and improvements.
- **Education and Experience:** Developers are students gaining practical experience in software development, applying their theoretical knowledge to create a functional system.

Importance of User Classes

Primary Users: Primary Users: The **Pet Owners** and **Veterinarians** are the primary focus of the VetCare system, as they will be the most frequent users and are the intended beneficiaries of the system's features.

Support Users: Clinic Administrators and **Developers** play critical supporting roles. Administrators ensure the smooth operation of the system in a clinic setting, while developers are essential during the project development phase, ensuring the system is built and functions as intended.

2.4 Operating Environment

The VetCare Online Vet Clinic Management System operates in a standard web-based environment, and its deployment involves the following key components:

1. **Hardware Platform:**
   - **Server Requirements:** The application is deployed on a server, which can be cloud-based or on-premise. Minimum specifications include a quad-core CPU, 8 GB RAM, and 100 GB SSD storage.
   - **Continuity:** An estimate of how much compute resources are required can be predicted once the app is live and an estimate of the number of users per hour is known according to which changes will be made to the run time environment and available resources compute resources.
   - **Client Devices:** The system is accessible on any device with a web browser that supports JavaScript and is able to render HTML, including desktops, laptops, tablets, and smartphones.
2. **Operating System:**
   - **Server Operating System:** Compatible with Linux distributions (e.g., Ubuntu 20.04 LTS).
   - **Client Operating Systems:** Users can access the system on any device that allows users to make HTTP requests, execute JavaScript and display HTML.
3. **Web Server:**
   - The system is hosted on an HTTP server, which could be Apache Tomcat, Jetty, or any compatible server that supports Java-based web applications.
4. **Software Components:**
   - **User Interface:** Built using Spring MVC, the UI is rendered as HTML/CSS in the browser.

- **Business Logic:** Handled by the service layer within the Spring Boot framework.
- **Persistence Layer:** Manages data through a MySQL or Oracle database.
- **Data Exchange:** JSON is used for communication between the front end and back end.

5. **Development and Testing Tools:**
   - **Code Repository and CI/CD:** Managed via GitHub and GitHub Actions.
   - **Build Tool:** Maven is used for project management and builds.
   - **Unit Testing:** JUnit5 is used for testing.
6. **Network Environment:**
   - Users connect to the VetCare system through a web browser over the internet, with the HTTP server processing requests and interacting with the database.

This operating environment ensures that the VetCare system runs smoothly across various devices and platforms, providing a reliable and consistent user experience.

2.5 Design and Implementation Constraints

1. **Technology Stack:**

   Mandatory Technologies: The project requires the use of Java 17 or later, Spring Boot for the web framework, and MySQL or Oracle as the database management system. These technologies have been chosen based on project requirements and must be adhered to throughout the development process.

   Build and Dependency Management: Maven is mandated as the build tool for managing dependencies and project builds, restricting the use of other tools like Gradle.

2. **Monolithic Architecture Constraints:**

   Single Deployment Unit: The system is built as a monolithic application, where all components are packaged and deployed together. This complicates scalability and makes it difficult to update or modify specific parts of the application independently.

   Tight Coupling: Components within the monolithic architecture are tightly coupled, meaning changes in one part of the system can affect others, increasing the complexity of updates and testing.

   Scalability Challenges: The entire application must be scaled as a single unit, potentially leading to inefficient resource use when only certain parts of the application require scaling.

   Deployment Complexities: Any change requires redeployment of the entire application, leading to longer downtime and more complex deployment processes as the system grows.

   Maintenance Challenges: Maintaining a monolithic architecture can become increasingly difficult over time. It becomes harder for any single developer to understand the entire program, leading to reliance on a few key individuals with deep system knowledge.

Big Ball of Mud: Over time, the monolithic architecture can lead to a "Big Ball of Mud," where the system becomes entangled with dependencies, making it difficult to manage, test, and extend.

Testing and Deployment Delays: The tightly coupled nature of the monolithic architecture means that testing and deployment can take longer, slowing down release cycles.

3. **MVC Structure Constraints:**

   Complexity in Large Applications: While MVC (Model-View-Controller) architecture is effective for separating concerns, it can lead to increased complexity as the application grows. Maintaining clear boundaries between models, views, and controllers can become challenging.

   Dependency Management: Controllers in MVC often handle multiple dependencies and business logic, which can make the code harder to maintain and test.

   Testing Challenges: The MVC structure requires thorough testing of each layer independently. Due to interdependencies, achieving high test coverage and isolating tests can be challenging.

4. **Testing and Scalability Constraints:**

   Limited User Base for Testing: As a university project, the system will not undergo mass testing by real users, limiting the ability to accurately gauge the system's performance, scalability, and user experience under load.

   Scalability Uncertainty: Due to the lack of mass user testing, the system's scalability is unverified. There is no practical data on how well it scales with a significant increase in users or data transactions.

5. **Hardware Limitations:**

   Server Specifications: The application must run on servers with a minimum specification of 8 GB RAM and a quad-core CPU, which may limit scalability under high user loads.

   Client Devices: The application should be optimized for devices with lower-end specifications (e.g., 4 GB RAM), affecting the complexity of the user interface and client-side processing.

6. **Interface Constraints:**

   Database Connectivity: The system is designed to connect only to MySQL or Oracle databases, limiting flexibility to integrate with other database systems.

   HTTP Server: The application must be hosted on an HTTP server compatible with Java-based web applications (e.g., Apache Tomcat, Jetty).

7. **Security Considerations:**

   Data Privacy: The system must ensure secure handling of user data, particularly sensitive information related to pets' medical records, requiring strong encryption protocols for data at rest and in transit.

   Authentication and Authorization: Role-based access control (RBAC) must be implemented to ensure users only access information and features relevant to their role.

8. **Design Conventions and Standards:**

   Coding Standards: The project adheres to standard Java coding conventions, using JUnit5 for testing and GitHub for version control. All code must be documented, and pull requests reviewed before merging.

   Responsive Design: The user interface must be responsive, functioning smoothly on various devices and screen sizes, limiting the complexity of the UI design.

9. **Project Scope and Time Constraints:**

   Limited Development Time: Development time is constrained by the academic calendar, requiring prioritization of core functionalities such as appointment scheduling, medical record management, and prescription handling.

   Team Size and Skills: The project team consists of students with varying levels of experience, limiting the use of complex or less familiar technologies.

10. **Regulatory Compliance (If Applicable):**

    Data Protection Regulations: While not strictly enforced at the university level, the project should consider basic data protection principles (e.g., GDPR) for potential real-world applications.

11. **Post-Delivery Maintenance Responsibility:**

    Maintenance Responsibility: As a university project, the development team is responsible for initial development, testing, and deployment. In a real-world scenario, another organization or team would assume long-term maintenance, requiring comprehensive documentation and relevant technical skills for ongoing support.

2.6 User Documentation

The following user documentation components will be delivered along with the VetCare platform:

- **Video Tutorials:** We will provide video tutorials that demonstrate how to use the various features of the application. These tutorials will offer users a visual and step-by-step guide to understanding the platform's functionalities.

- **Interactive User Onboarding Guide:** An interactive User Onboarding Guide will be included to assist new users in familiarizing themselves with the platform. This guide will automatically appear when users first access the site or when new features are introduced. It offers step-by-step instructions and highlights key features of the platform, guiding users with "Next" and "Previous" buttons. This interactive element ensures that users can quickly and effectively learn how to navigate and use the platform, thereby enhancing their overall experience.
- **Frequently Asked Questions (FAQ):** An FAQ section will be included as part of the user documentation. This section will address common questions and concerns that users might have while using the platform. The FAQ will cover a range of topics, from basic usage to more advanced features, and will provide quick, concise answers. The FAQ is intended to help users find solutions to common issues without needing to consult the full user manual or contact support, thereby improving their overall experience with the platform.
- **Contact Web Administrator:** Users will have access to a dedicated "Contact Web Administrator" section within the platform. This feature will provide users with a direct way to reach out to the web admin team for assistance with technical issues, account problems, or other concerns that cannot be resolved through the user manual or FAQ. The contact section will include a form for submitting inquiries, as well as information on how to reach the support team via email or phone. This ensures that users have access to personalized support when needed, further enhancing their experience with the VetCare platform.
    - - will have email, first name, last name, contact and what the problem is.

2.7 Assumptions and Dependencies

The development of the VetCare Online Vet Clinic Management System is based on several assumptions and dependencies that, if incorrect or altered, could impact the requirements and overall success of the project. These include:

Assumptions:

- **Consistent Development Environment:** It is assumed that all development team members will have access to a consistent development environment, including the necessary software, hardware, and network resources. Any variation in these environments could lead to integration issues or unexpected bugs.
- **Availability of Required Tools and Technologies:** The project assumes that all required tools and technologies (e.g., Java 17, Spring Boot, MySQL, Docker) will be available and functional throughout the development process. Any disruption in access to these tools, such as licensing issues or software deprecations, could delay the project.
- **Stable Network Connectivity:** It is assumed that the development, testing, and deployment environments will have stable and reliable network connectivity. This is critical for accessing cloud-based resources, collaborating via GitHub, and deploying the application.
- **Third-Party Services and APIs:** The project assumes that any third-party services or APIs (e.g., SMTP servers for email notifications) used in the system will remain stable and available throughout the project lifecycle. Changes or discontinuation of these services could require significant rework.

- **User Familiarity with Web Applications:** It is assumed that the end-users (pet owners, veterinarians, and clinic administrators) have basic familiarity with using web applications. Extensive user training is not anticipated as part of this project.

Dependencies:

- **Database Management System:** The project depends on the availability and reliability of MySQL or Oracle as the database management system. Any issues with these databases, such as version incompatibilities or performance problems, could impact the system's functionality.
- **Docker and Containerization:** The project relies on Docker for containerizing the monolithic application to ensure environment consistency across the development, testing, and production stages. Any issues with Docker or related tools could affect deployment and testing processes.
- **External Libraries and Frameworks:** The project depends on external libraries and frameworks (e.g., Spring Boot, JUnit) for critical functionalities. If these components receive significant updates or become deprecated, the project may need to adapt, potentially affecting timelines and stability.
- **GitHub for Version Control and CI/CD:** The project assumes that GitHub will be used consistently for version control and continuous integration/continuous deployment (CI/CD) throughout the development process. Any interruptions in GitHub's service or issues with integrating CI/CD pipelines could impact the project's progress.
- **Project Team Availability:** The project assumes that all team members will be available and able to contribute according to the project plan. Any unforeseen circumstances (e.g., illness, conflicting academic responsibilities) that reduce team availability could delay the project.

3. External Interface Requirements

3.1 User Interfaces

The VetCare platform has been designed for all users that require an all in one application to manage their pet's health. It includes features such as a display of all previous medical records, prescriptions, appointments as well as account details. Our platform has an emphasis on intuitive design, allowing for ease of use amongst our users, through our standardised navigation systems across our pages. The key features of our platform are as follows:

- **Dashboard Interface:** The homepage includes a user friendly dashboard that utilises simplicity for ease of navigation for the user. The use of tiles helps us achieve this, whereby the user is prompted with pages they can click on depending on their needs.
- **Navigation Bar and Footer**: We have implemented a consistent navigation and footer across all our pages serving as a default way to navigate through the entire site. This will reduce the chance of users getting lost on the platform as well as help standardise the application, enhancing user experience. Furthermore, for mobile phone screen size

constraints, we have developed a condensed, drop-down menu for our navigation bar that overcomes the issue of size constraints.

- **Page Interfaces**: The platform will have a variety of different interfaces that the user will encounter across the website, we have standardised the tables across both desktops and mobiles to help ensure consistency for the user. This is in an attempt to simplify any searches for information that the user may require.
- **Calendars**: We have implemented a calendar UI design that has a focus on ease of use for our users. To achieve this, we mimicked calendars that users would already be familiar with in real world scenarios, making booking appointments simple and familiar.
- **Buttons:** Each of our button components follow the same structure and design, this is to aid the user in drawing their attention to focal points to help them navigate through the site.
- User Onboarding using Tooltips: The VetCare platform includes an interactive **User Onboarding Guide** that uses tooltips to assist new users in familiarizing themselves with the platform. This guide appears when users first access the site or when new features are introduced. It provides step-by-step instructions and highlights key features of the platform, with "Next" and "Previous" buttons to guide users through each step. This interactive element ensures that users can quickly learn how to navigate and use the platform effectively, enhancing their overall experience.

3.2 Hardware Interfaces

The VetCare application is built on a monolithic architecture, interfacing with various hardware components through standard protocols. This section outlines the logical and physical characteristics of the hardware interfaces that the application interacts with.

Supported Device Types

- **User Devices**:
    - **Desktops and Laptops**: The application supports modern web browsers such as Google Chrome, Mozilla Firefox, Safari, and Microsoft Edge running on desktop and laptop computers. These devices connect to the application through HTTP/HTTPS protocols and render HTML, CSS, and JavaScript content provided by the server.
    - **Mobile Devices**: The system is accessible on mobile devices running Android and iOS. The user interface adapts responsively to different screen sizes and orientations, ensuring a consistent experience across all devices.
- **Servers**:
    - **HTTP Server**: The application leverages an HTTP server like Jetty or Apache to handle incoming requests from user devices. This server processes HTTP/HTTPS requests and serves the required web pages.
    - **Application Server**: The business logic of the VetCare application is managed by an application server running Java 17, utilizing the Spring Boot framework. This server manages core functionalities, including business operations and database interactions.

    - **Database Server**: The application's persistence layer connects to a relational database server (e.g., Oracle, MySQL) to store and manage application data. The database server is a critical component, ensuring reliable data storage and retrieval.

Data and Control Interactions

- **User Interface to HTTP Server**: User interactions are handled by the HTTP server, which processes user requests and serves dynamic content. Communication between the client's web browser and the server is managed through HTTP/HTTPS protocols.
- **Business Logic to Persistence Layer**: The business logic interacts with the database via the Repository/DAO pattern, performing CRUD operations on the database. These interactions are managed through JDBC or another ORM tool within the Spring Boot application.
- **Peripheral Devices**: The system may interface with peripherals like printers and scanners through standard operating system interfaces. This enables functionalities such as printing documents or uploading scanned files.

Communication Protocols

- **HTTP/HTTPS**: The primary protocol for communication between client devices and the HTTP server, with HTTPS ensuring secure, encrypted data transmission.
- **TCP/IP**: Used for communication between the application server and the database server, ensuring reliable data transfer.
- **JSON**: Used for data interchange between the front-end and back-end components, particularly for transferring structured data in a lightweight format.

Hardware Requirements

- **Client Devices**:
    - **Desktops/Laptops**: Requires a modern web browser, 4GB RAM, Intel i3 processor or equivalent, 250GB HDD/SSD.
    - **Mobile Devices**: Requires Android or iOS with 2GB RAM, 16GB storage.
- **Servers**:
    - **HTTP Server**: Requires 8GB RAM, Quad-Core processor, 100GB SSD storage.
    - **Application Server**: Requires Java 17, 8GB RAM, Quad-Core processor, 100GB SSD.
    - **Database Server**: Requires 16GB RAM, Quad-Core processor, 500GB SSD for Oracle, MySQL, or equivalent RDBMS.

3.3 Software Interfaces

Overview

The VetCare application is containerized using Docker, which allows all components—including the application server, database, and other services—to run in isolated and consistent environments. This section describes the connections between the VetCare application and other specific software components, detailing the data exchanges, communication protocols, and integration points.

Connections to Specific Software Components

- **Operating System**:
  - The application runs within Docker containers, which are OS-agnostic, but typically run on a Linux-based host in production environments. The containers ensure that the application and all its dependencies are consistent across different environments.
- **Database**:
  - **MySQL/Oracle**: The persistence layer of the VetCare application interacts with a MySQL or Oracle database, which is also containerized using Docker. The application uses JDBC (Java Database Connectivity) through Spring Data JPA to perform CRUD operations. Data items include user profiles, medical records, appointments, and transactional data.
- **Web Framework**:
  - **Spring Boot (Java 17)**: The core application is implemented using Spring Boot, which is responsible for handling HTTP requests, executing business logic, and interacting with the database. The application's web layer is managed by Spring MVC, rendering HTML/CSS for the user interface.
- **JSON Communication**:
  - **Front-End to Back-End**: Data interchange between the user interface (front-end) and the application logic (back-end) is facilitated using JSON. This format is used for transmitting data such as form inputs, API responses, and other dynamic content.
- **CI/CD Pipeline**:
  - **GitHub Actions**: Continuous Integration and Continuous Deployment (CI/CD) are managed via GitHub Actions. The pipeline automates testing (using JUnit 5), building (using Maven), and deploying the application into Docker containers.
- **Code Repository**:
  - **GitHub**: The source code is managed using GitHub, with version control and collaboration features enabling smooth development workflows.

Services and Communication

- **HTTP/HTTPS**:
  - Communication between client devices (e.g., web browsers) and the server is handled via HTTP/HTTPS, ensuring secure data transmission.
- **API Documentation**:
  - API endpoints are documented using Swagger or Spring REST Docs, providing clear guidelines for how different services interact within the application.

Data Sharing Mechanism

- **Session Management**:
  - **Spring Security**: User session data is managed through Spring Security, ensuring that user authentication and session states are maintained securely across different components of the application.
- **Containerized Environment**:
  - All components (application server, database, etc.) run in Docker containers, ensuring isolated and consistent environments. Data sharing and communication

between containers (e.g., the application server and the database) occur over Docker's internal networking, typically using standard network protocols like TCP/IP.

External APIs and Integration Points

- **Payment Gateways (if applicable)**:
    - The application can integrate with external payment gateways (e.g., Stripe, PayPal) through their APIs, facilitating online transactions securely.

3.4 Communications Interfaces

Overview

The VetCare system utilizes a variety of communication methods to ensure users receive timely updates and alerts. These methods include email and web browser notifications. The communication functions are designed to keep users informed about appointments, medical records, prescription renewals, and other relevant updates.

Communication Methods

Email:

- **Purpose:** Will be utilised as the standard form of contact with the user for all functions such as booking confirmations and prescriptions.
- **Formatting:** Emails will use an HTML structure, ensuring consistency amongst users and their emailing platforms.
- **Protocol/Security:** We will use SMTP for sending our emails for reliability, as well as TLS to ensure a secure communication line with all users.

Web Browser Notifications:

- **Purpose:** Delivers in-app alerts to our users to notify them of changes and updates on the platform, including confirmations of purchases and bookings.
- **Formatting:** Notifications will be succinct and direct to the user, using JavaScript to notify them of any reminders/notifications. This enables real time communication with the user.
- **Protocol/Security:** All web browser notifications will bruising HTTPS to maintain data security for our users.

Communication Standards and Protocols

- **HTTP/HTTPS**: All web-based communications use HTTP/HTTPS protocols to ensure secure data transmission between the server and client.
- **SMTP**: Emails are sent using SMTP, with TLS encryption to protect the contents during transit.
- **Web Push Protocol**: Used for sending browser notifications, ensuring compatibility across different web browsers and secure delivery.

- **Firebase Cloud Messaging (FCM)/Apple Push Notification service (APNs)**: These protocols are used for sending mobile push notifications securely to Android and iOS devices, respectively.

Security and Encryption

- **TLS Encryption**: Ensures that all emails are encrypted during transmission to prevent unauthorized access.
- **HTTPS**: Used for all web-based communications to protect data integrity and confidentiality.
- **Data Transfer Rates**: While specific rates depend on the network, the system is optimized to ensure timely delivery of all communications without noticeable delays to the end user.
- **Synchronization Mechanisms**: The system uses periodic checks and acknowledgments to ensure that notifications and alerts are synchronized across devices and communication channels.

4. Nonfunctional Requirements

4.1 Performance Requirements

Performance requirements are crucial to ensure that the VetCare platform provides a seamless and efficient experience for both pet owners and veterinary clinic staff. These requirements must be met under various operating conditions to maintain the integrity and usability of the system.

1. **Response Time**:
    - **Normal Load**: The application should respond to user inputs and requests within **2 seconds** under normal load conditions.
    - **Peak Load**: During peak load times, response times should not exceed **5 seconds**. This ensures a fluid user experience and minimizes frustration during busy periods.
2. **System Availability**:
    - The VetCare application should maintain **99.9% uptime**, allowing for scheduled maintenance windows, which should be communicated to users in advance. High availability is critical to ensure users can access the service when needed, especially during emergencies.
3. **Concurrent Users**:
    - The system must support up to **1,000 concurrent users** without degradation in performance. This requirement is based on anticipated peak usage scenarios, ensuring that the system remains responsive during high-traffic periods.
4. **Data Processing**:
    - The application should process data transactions, such as appointment bookings or medical record updates, within **3 seconds 95%** of the time. This speed is essential to ensure that records are up-to-date and accurate, reflecting changes in real-time, which is crucial for proper healthcare management.
5. **Scalability**:

- VetCare must be designed to scale seamlessly with increases in user base and data volume. The system should support a **50% increase in concurrent users** with proportional increases in infrastructure, without significant loss in performance. This ensures that the platform can accommodate future growth.

4.2 Safety Requirements

Safety is paramount due to the sensitive nature of the data and the critical services provided by the VetCare application. These safety requirements are designed to minimize the risks associated with both human error and system failures, ensuring a secure and reliable user experience.

- **Data Backup and Recovery**:
  - The application must implement robust data backup procedures to prevent data loss in the event of a system failure. Regular backups should be scheduled, with backups stored in a secure, off-site location. Data recovery processes must be in place, tested regularly, and capable of restoring data to its most recent state without data corruption or loss.
- **Error Handling and Logging**:
  - The system must include comprehensive error handling mechanisms to prevent system crashes, data corruption, or unintended operations. All critical errors should be logged with detailed information to facilitate rapid troubleshooting. User activities and system interactions must also be logged to monitor for potential security breaches or operational failures.
- **Regulatory Compliance**:
  - The application must comply with all relevant veterinary medical privacy laws and regulations applicable in the jurisdictions where the system is deployed. For example, in the U.S., the system should comply with the Health Insurance Portability and Accountability Act (HIPAA), which mandates the protection and confidential handling of medical information.
- **Safety Certifications**:
  - The system should aim to obtain and maintain certifications such as ISO 27001, which provides a framework for establishing, implementing, maintaining, and continually improving an information security management system (ISMS). This certification ensures that the system adheres to industry-recognized best practices for security management and comprehensive security controls as outlined in ISO 27001.
- **Preventive Measures**:
  - The application must implement preventive measures to protect against data breaches, unauthorized access, and other forms of security threats. This includes regular security audits, vulnerability assessments, and adherence to secure coding practices.

4.3 Security Requirements

Security in the VetCare application is critical to protect both the privacy of users and the integrity of the medical data it handles. The following security requirements are established to ensure comprehensive protection in compliance with Australian regulations.

1. **Authentication and Authorization**:
    - **Multi-Factor Authentication (MFA)**: Implement MFA to provide an additional layer of security, ensuring that only authorized users can access the system. This may include a combination of passwords, security tokens, or biometric verification.
    - **Role-Based Access Control (RBAC)**: Define user roles and permissions carefully to ensure that users can only access the information and functionalities relevant to their role (e.g., administrators, veterinarians, pet owners).
2. **Data Encryption**:
    - **Data at Rest**: Encrypt all sensitive data stored within the system, including the database, backup files, and any other data storage systems, using strong encryption standards such as AES-256.
    - **Data in Transit**: Protect data during transmission using TLS (Transport Layer Security) to prevent unauthorized access and ensure data integrity as it moves between the user's device and the server.
3. **Compliance and Privacy Policies**:
    - **Australian Privacy Act 1988**: Ensure compliance with the Australian Privacy Act 1988, which governs the collection, use, storage, and disclosure of personal information. This includes adhering to the Australian Privacy Principles (APPs) that outline how personal information must be managed.
    - **Data Breach Notification**: Implement processes to comply with the Notifiable Data Breaches (NDB) scheme under the Privacy Act, which requires organizations to notify affected individuals and the Office of the Australian Information Commissioner (OAIC) of eligible data breaches.
    - **User Data Transparency**: Provide mechanisms for users to view the data stored about them, including the ability to request corrections or deletion in accordance with the Privacy Act.
4. **Security Certifications**:
    - **ISO/IEC 27001 Certification**: Pursue ISO/IEC 27001 certification to demonstrate the application's commitment to comprehensive information security management, which is recognized internationally and can bolster trust with Australian users.
    - **Australian Signals Directorate (ASD) Essential Eight**: Implement the Essential Eight mitigation strategies as recommended by the ASD to improve security posture and protect against cybersecurity threats.
5. **Security Monitoring and Incident Response**:
    - **Continuous Monitoring**: Implement continuous security monitoring to detect and respond to potential security threats in real-time, including unauthorized access attempts and suspicious activities.
    - **Incident Response Plan**: Develop and maintain a robust incident response plan that aligns with Australian regulatory requirements, outlining procedures for responding to security breaches, including containment, mitigation, and notification protocols.

4.4 Software Quality Attributes

The VetCare application's development will focus on several key quality attributes to ensure that it meets the expectations of both users and developers. These attributes must be specific, quantifiable, and verifiable to contribute effectively to the overall quality and reliability of the application.

1. **Reliability**:
    - **Uptime**: The application must maintain *99.9% uptime*, with downtime limited to scheduled maintenance windows. Any unexpected downtime must be resolved within *1 hour* to minimize disruption to users, ensuring continuous availability of critical services.
2. **Usability**:
    - **Ease of Use**: The user interface should be designed to be intuitive and user-friendly, requiring no more than *30 minutes of training* for new users. Usability should be assessed through regular user satisfaction surveys, with a target satisfaction score of *90%* or higher. User feedback will be actively used to make iterative improvements.
3. **Performance**:
    - **Responsiveness**: The application must handle user interactions efficiently, ensuring that response times do not exceed *2 seconds* under normal load conditions. During peak load times, response times should not exceed *5 seconds* to maintain a seamless user experience.
4. **Maintainability**:
    - **Modularity**: The system should be built using a modular architecture, allowing for straightforward updates and maintenance. The code should be well-documented and adhere to clean coding practices, making it easy for developers to manage dependencies and implement updates with minimal impact on the overall system.
5. **Portability**:
    - **Cross-Platform Compatibility**: The application must be easily deployable across multiple environments, including major operating systems (Windows, macOS, Linux) and web browsers (Chrome, Firefox, Safari, Edge). The platform should also support mobile and desktop use with minimal configuration changes required for each environment.
6. **Testability**:
    - **Automated Testing**: The application must support comprehensive automated testing, including unit tests, integration tests, and regression tests. These tests should cover *100% of critical functionalities* and aim for at least *80% overall code coverage*. Detailed logging must be implemented to facilitate debugging and validation of test results.

4.5 Business Rules

Business rules dictate the operational guidelines and principles that govern the functions of the VetCare system. These rules are not only essential for maintaining the order and integrity of the system but also imply certain functional requirements necessary for enforcing these rules.

- **Role-Based Access Control (RBAC):** Access to different parts of the system should be governed by user roles. For example:
    - Veterinary Staff can access and update medical records, manage appointments, and process prescriptions.

    - Pet Owners are restricted to viewing and managing their pets' appointments, medical records, and prescriptions.
    - Administrators have overarching access to the entire system for management and maintenance purposes.
- **Data Privacy:** Any access to pet medical records must comply with relevant privacy laws and regulations. Only authorized users (vet staff and pet owners) can view specific medical records, and the sharing of information must be explicitly authorized by the pet owner.
- **Appointment Booking Rules:** Appointments can only be booked if available slots are confirmed. Automatic checks should prevent double bookings and provide alternatives when slots are not available.
- **Prescription Access:** Only licensed veterinary professionals can create or modify prescription details. Pet owners can view and request refills but cannot alter the prescription data.
- **Audit Trails:** All significant actions within the system, such as access to sensitive information or changes to medical records, must be logged to ensure traceability and accountability.

5. Other Requirements

Other requirements include:-

Animal Welfare Regulations

- **State and Territory Legislation**:
    - **Overview**: Animal welfare in Australia is governed by laws specific to each state and territory. These laws ensure that animals are treated humanely and that veterinary practices comply with welfare standards.
    - **Requirements**:
        - **Veterinary Practices**: Ensure that all veterinary services offered through your platform comply with state and territory regulations. This includes ensuring that the veterinarians are licensed and adhere to professional standards.
        - **Advertising and Information**: Ensure that any information related to pet care, treatment options, or breed-specific issues provided through the platform is accurate and not misleading. Misrepresentation can lead to legal liabilities.

Professional Regulations

- **Veterinary Surgeons Acts (State-Specific)**:
    - **Overview**: Each state in Australia has its own Veterinary Surgeons Act that regulates the practice of veterinary medicine. These acts establish requirements for the licensing of veterinarians and the standards they must adhere to.
    - **Requirements**:

        - **Verification of Credentials**: Your platform must verify the credentials of veterinarians listed on the platform to ensure they are properly licensed to practice in their respective states.
        - **Professional Conduct**: Ensure that the services provided by veterinarians through your platform comply with the professional conduct standards as defined by state legislation.

Animal Registration and Microchipping

- **State and Local Government Regulations**:
    - **Overview**: Regulations around animal registration and microchipping vary by state and local government. In many areas, it is mandatory for pets to be registered and microchipped.
    - **Requirements**:
        - **Integration with Local Regulations**: Your application should provide users with information about the registration and microchipping requirements in their area. It may also be beneficial to offer features that help users manage these aspects (e.g., reminders for registration renewals).

Advertising and Marketing Regulations

- **Spam Act 2003**:
    - **Overview**: The Spam Act regulates the sending of commercial electronic messages, including emails and SMS.
    - **Requirements**:
        - **Consent**: Obtain explicit consent from users before sending marketing communications.
        - **Unsubscribe Mechanism**: Include a clear and functional unsubscribe mechanism in all marketing communications.
        - **Identification**: Ensure that all communications clearly identify your business as the sender.

Intellectual Property

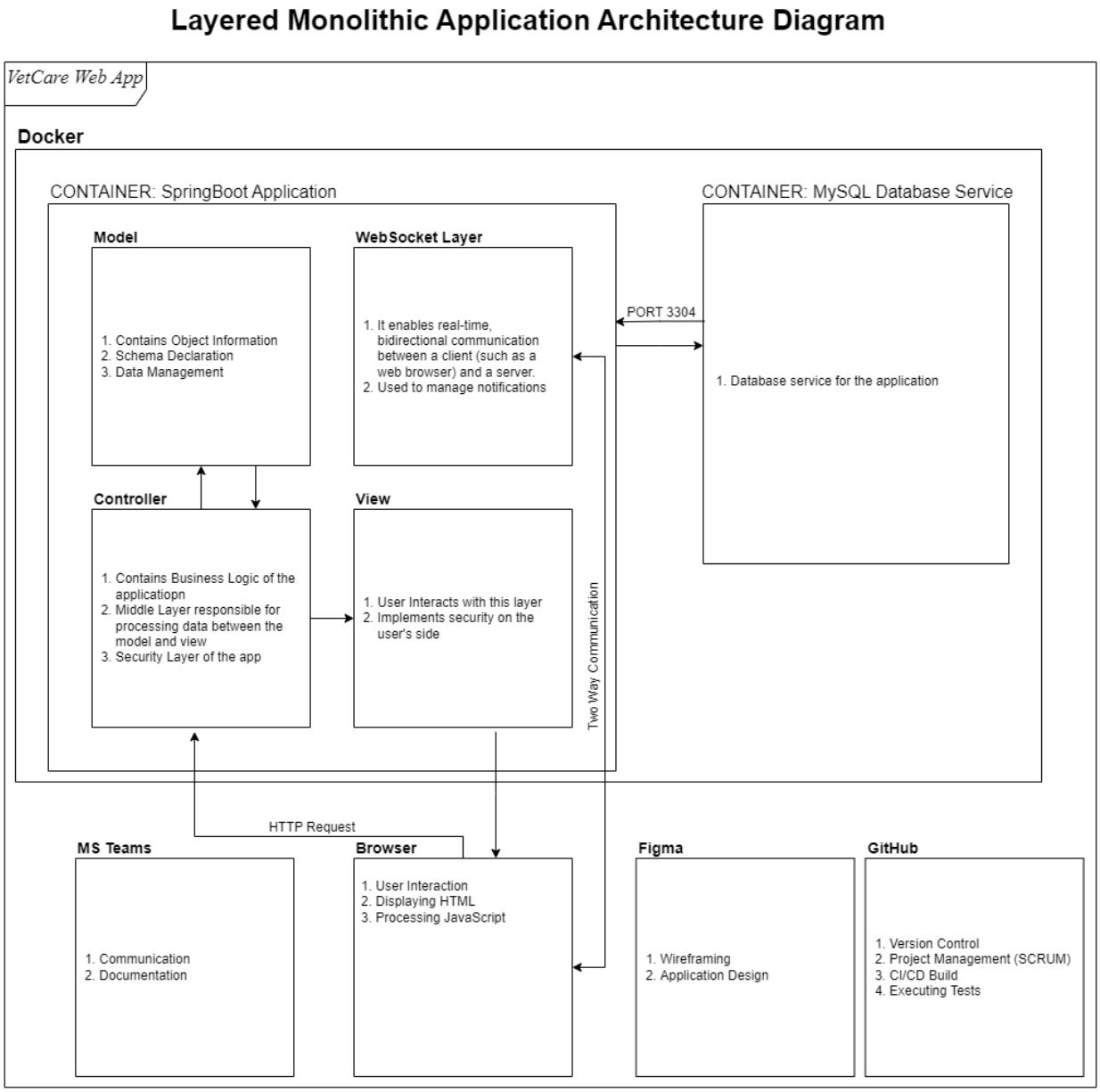
- **Trademark and Copyright Laws**:
    - **Overview**: If your platform uses any proprietary content, brand names, or logos, ensure that these are properly protected under trademark or copyright laws.
    - **Requirements**:
        - **Trademark Registration**: Consider registering your platform's name, logo, and other branding elements as trademarks to protect your intellectual property.
        - **Copyright Compliance**: Ensure that any content provided on your platform (e.g., educational materials, images) does not infringe on third-party copyrights.
- Any publicly available data on the application is the property of the application owner and is not meant to be scraped/used in any way/form on any public or private platform without the permission of the application owner.

Contractual Obligations

- **Terms of Service and User Agreements**:
    - **Overview**: Clearly define the terms of service and user agreements that govern the use of your platform.
    - **Requirements**:
        - **Liability Limitations**: Include clauses that limit your platform's liability in cases where services provided by third-party veterinarians do not meet user expectations.
        - **Dispute Resolution**: Establish a dispute resolution process to handle conflicts between users and service providers.

6. System Architecture

Figure 3: Architecture Diagram



The VetCare application follows a monolithic architecture, encapsulating all components within a single, cohesive Docker container environment. This approach simplifies deployment and scaling while ensuring that all parts of the application can efficiently interact within the same runtime environment.

Fundamental Decisions and Solution Strategies:

Technology Stack:

- **Spring Boot (Spring MVC Framework)**: The application is built using Spring Boot, leveraging the Spring MVC framework for handling web requests and responses. This framework was chosen for its robustness, ease of use, and extensive support for web-based applications.
- **Docker**: The entire application is containerized using Docker, enabling consistent deployment across various environments and simplifying the management of dependencies.
- **MySQL Database**: MySQL serves as the relational database, selected for its reliability, scalability, and wide adoption, making it a solid choice for handling structured data.

Top-Level Decomposition:

The system is decomposed into several key components:

- **User Interface (UI)**: Delivered through a web browser, it interacts with the application via HTTP requests.
- **Spring MVC Application**: Comprising the HTTP server, controller, model, view, service layer (business logic), and persistence layer (repository/DAO), this component handles all business operations and user interactions.
- **Notification Service**: Implements real-time notifications using WebSocket connections, ensuring that users receive updates promptly within the browser.
- **Database (MySQL)**: Manages all data storage and retrieval operations, interfaced by the persistence layer of the application.

Approaches to Achieve Top-Quality Goals:

- **Performance and Scalability**: The monolithic architecture, when combined with Docker, allows the application to be easily scaled by replicating the container. The use of MySQL ensures that data operations are efficient, even under high load.
- **Reliability and Availability**: The architecture is designed to achieve high reliability, with a robust error-handling mechanism within the service layer. Docker's containerization provides consistent environments, reducing the risk of deployment issues.
- **Real-Time Communication**: Real-time notifications are handled via WebSocket connections, allowing the application to push updates to users instantly without requiring them to refresh their browsers.

Relevant Organizational Decisions:

- **Centralized Development**: The monolithic approach was chosen to simplify development and maintenance, allowing a centralized team to work on the entire application without the complexities of microservices.
- **Deployment Strategy**: By using Docker, the application can be deployed consistently across different environments (development, testing, production) with minimal configuration changes, ensuring that the application behaves the same way regardless of the deployment environment.

6.1 Architecture Overview

The VetCare application is structured as a monolithic system encapsulated within a Docker container. The system is built using a hierarchical approach where the main components (white boxes) contain subcomponents (black boxes), providing a clear abstraction of the source code and its organization.

1. VetCare Application (White Box)

- The overall VetCare application is contained within a Docker environment, ensuring consistency across development, testing, and production environments.
- 1.1 Docker Container (White Box)
    - The Docker container serves as the environment for the entire application, encapsulating all components and ensuring that dependencies and configurations are managed consistently.
    - 1.1.1 Spring Boot Application (White Box)
        - The Spring Boot Application is the core of the VetCare system, implementing the MVC (Model-View-Controller) pattern to manage user interactions and business logic.
        - 1.1.1.1 HTTP Server (Black Box)
            - Manages incoming HTTP requests from the user's browser and routes them to the appropriate controllers.
        - 1.1.1.2 Controller Layer (Black Box)
            - Handles the incoming requests by interacting with the service layer (business logic) and preparing the data needed for the view layer.
        - 1.1.1.3 Model Layer (Black Box)
            - Represents the data structures used within the application. It interacts with the persistence layer to fetch and update data stored in the database.
        - 1.1.1.4 View Layer (Black Box)
            - Responsible for rendering the user interface by using HTML and CSS. The view layer presents the data processed by the model layer to the user.
        - 1.1.1.5 Business Logic (Service Layer) (Black Box)
            - Implements the core business rules and logic of the application. It processes data received from the controller layer and interacts with the persistence layer for data operations.
        - 1.1.1.6 Persistence Layer (Repository/DAO) (Black Box)
            - Manages database interactions by interfacing with the MySQL database. It performs CRUD (Create, Read, Update, Delete) operations on the data.
        - 1.1.1.7 Notification Service (Black Box)
            - Responsible for managing real-time notifications. It pushes updates to the user interface via WebSocket connections.
        - 1.1.1.8 WebSocket Connection (Black Box)

        - Manages real-time, bi-directional communication between the server and the user's browser, ensuring that users receive instant updates.
    - 1.1.2 Database (MySQL) (White Box)
        - The MySQL database serves as the primary data store for the application, handling all persistent data.
        - 1.1.2.1 Database Tables (Black Box)
            - Represents the structured data stored in various tables, such as Users, Pets, Appointments, and Medical Records. Each table corresponds to a specific model in the application.
        - 1.1.2.2 SQL Queries and Procedures (Black Box)
            - Encapsulates the SQL queries and stored procedures that the persistence layer uses to interact with the database, ensuring efficient data retrieval and manipulation.

2. User Interface (White Box)

- The user interface is the front-end component that users interact with, accessed through a web browser.
- 2.1 Browser (White Box)
    - The browser acts as the client interface for accessing the VetCare application over the network.
    - 2.1.1 HTML/CSS (Black Box)
        - Defines the structure and styling of the web pages that are rendered in the user's browser, ensuring a consistent and user-friendly interface.
    - 2.1.2 JavaScript (Black Box)
        - Manages client-side logic, enabling dynamic interactions within the web pages, such as form validations, AJAX requests, and real-time updates.

3. Network Connection (White Box)

- The network connection facilitates communication between the user's browser and the Docker container hosting the VetCare application.
- 3.1 HTTP Requests and Responses (Black Box)
    - Represents the communication protocol for transmitting data between the client and server, handling standard web requests and responses.
- 3.2 WebSocket Connection (Black Box)
    - Manages real-time updates from the server to the client, allowing for instant notifications and updates in the browser without needing to refresh the page.

6.2 Architectural Decisions

In developing the VetCare application, several key architectural decisions were made that are important due to their impact on the system's functionality, cost, scalability, and risk management. Below are some of these decisions, along with the rationales behind them:

1. Monolithic Architecture Decision

Decision: The decision was made to use a monolithic architecture instead of a microservices architecture.

Rationale:

Simplicity: A monolithic architecture is simpler to design, develop, and manage for a small to medium-sized application like VetCare. It allows all components to be tightly integrated within a single codebase, reducing the overhead of managing multiple services.

Cost-Effectiveness: Developing and deploying a monolithic application is typically less expensive than a microservices architecture, which requires more complex infrastructure, orchestration, and monitoring tools.

Performance: A monolithic architecture can offer better performance in this context, as it avoids the latency and overhead associated with inter-service communication in microservices.

2. Use of Docker for Containerization

Decision: The entire VetCare application is containerized using Docker.

Rationale:

Consistency Across Environments: Docker ensures that the application behaves the same across development, testing, and production environments by encapsulating all dependencies and configurations within the container.

Scalability: While VetCare uses a monolithic architecture, Docker allows the application to be easily scaled horizontally by deploying additional container instances as needed.

Simplified Deployment: Docker simplifies the deployment process, allowing for continuous integration and continuous deployment (CI/CD) pipelines to be easily implemented, reducing the risk of deployment failures.

3. Spring Boot and Spring MVC Framework

Decision: The application is built using Spring Boot with Spring MVC for web framework functionality.

Rationale:

Mature Ecosystem: Spring Boot is a mature and widely-used framework that provides comprehensive support for building enterprise-grade applications. It comes with built-in tools and libraries for handling web requests, business logic, security, and data persistence.

Rapid Development: Spring Boot's convention-over-configuration approach accelerates development by minimizing boilerplate code and simplifying configuration.

Community Support: The large Spring community ensures that there is extensive documentation, libraries, and third-party integrations available, reducing development time and risks associated with less established frameworks.

4. MySQL as the Primary Database

Decision: MySQL was selected as the primary relational database for VetCare.

Rationale:

Reliability and Performance: MySQL is a proven, reliable, and performant relational database system that can handle the transactional workload of VetCare, ensuring data integrity and fast query performance.

Cost-Effectiveness: MySQL is open-source and free to use, making it a cost-effective choice for the project while still offering enterprise-level features.

Scalability: MySQL can scale both vertically (on a single server) and horizontally (using replication), providing flexibility as the application's data and user base grow.

5. Real-Time Notifications via WebSockets

Decision: Real-time notifications are implemented using WebSockets.

Rationale:

User Experience: WebSockets allow for instant communication between the server and the client, providing users with real-time updates without the need for page refreshes. This enhances the user experience by ensuring timely updates, especially for critical features like appointment reminders and prescription alerts.

Performance: WebSockets are more efficient than HTTP polling or long-polling, as they maintain a single open connection between the client and server, reducing overhead and improving responsiveness.

6. CI/CD Pipeline with GitHub Actions

Decision: The CI/CD pipeline is managed using GitHub Actions.

Rationale:

Integration with Codebase: GitHub Actions is tightly integrated with the GitHub repository, allowing seamless automation of build, test, and deployment processes whenever changes are pushed to the codebase.

Flexibility and Customization: GitHub Actions provides a flexible framework that can be easily customized to fit the specific needs of the VetCare application's development and deployment workflow.

Cost Considerations: As part of the GitHub ecosystem, GitHub Actions is cost-effective, especially for smaller teams, as it avoids the need for additional CI/CD tools or platforms.

7. RSS Feeds Integration for Article Updates

Decision: RSS feeds are used to fetch articles and display them on the website.

Rationale:

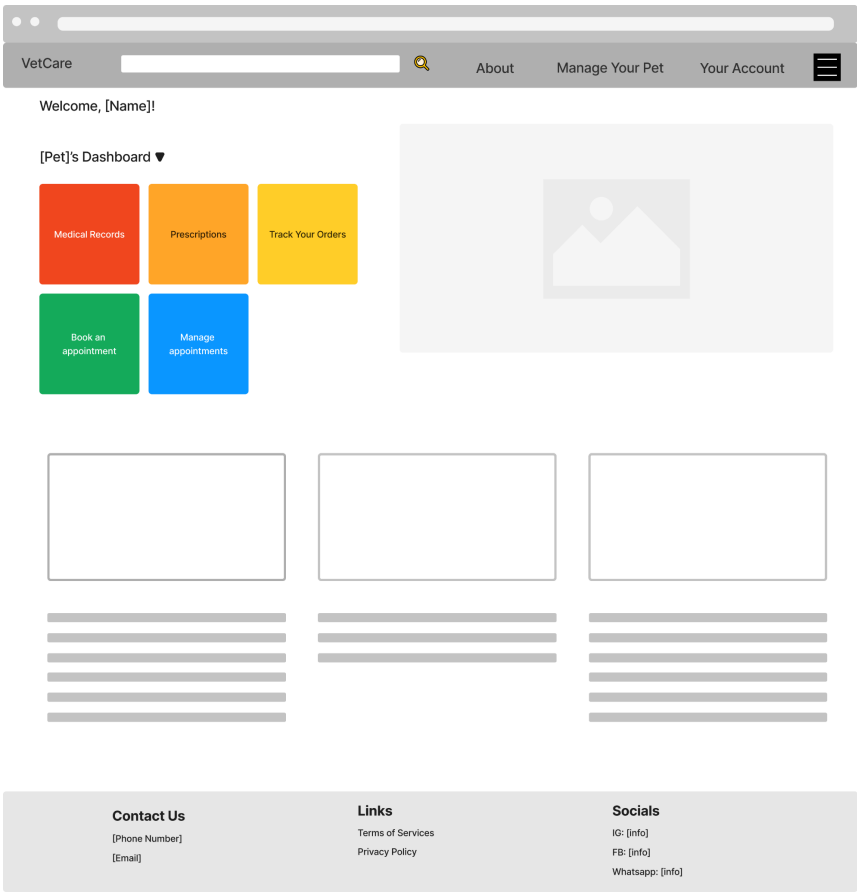
Content Collection: Using RSS feeds allows the VetCare platform to collect relevant articles from trusted external sources, providing users with up-to-date information on pet care, health tips, and veterinary news without manually having to create our own content.

Automation: The RSS feeds integration automatically fetches new articles as they are published, ensuring that the content on the platform remains new and relevant.

Improved User Experience: Having a consolidated view of various pet-related topics on the platform reduces the users' need to visit multiple external websites for the latest articles which makes it more convenient for users.

7. User Interface Design

Figure 4: Dashboard Wireframe



The homepage dashboard for the VetCare website has a comprehensive and user friendly interface that is simple and minimalistic, adhering to the 8th Nielsen's heuristic. At the top, there is a header with relevant navigation links, a search bar and a hamburger dropdown, which helps to maintain the visibility of system status, and also allows easy navigation across the application. This header will also be consistent throughout the rest of the application.

Panning down, there is a welcome text that will display a welcome message to the logged in user, and below that, and dropdown bar that allows the user to switch between their pets if they have multiple pets, to allow for flexibility and efficiency when managing pets.

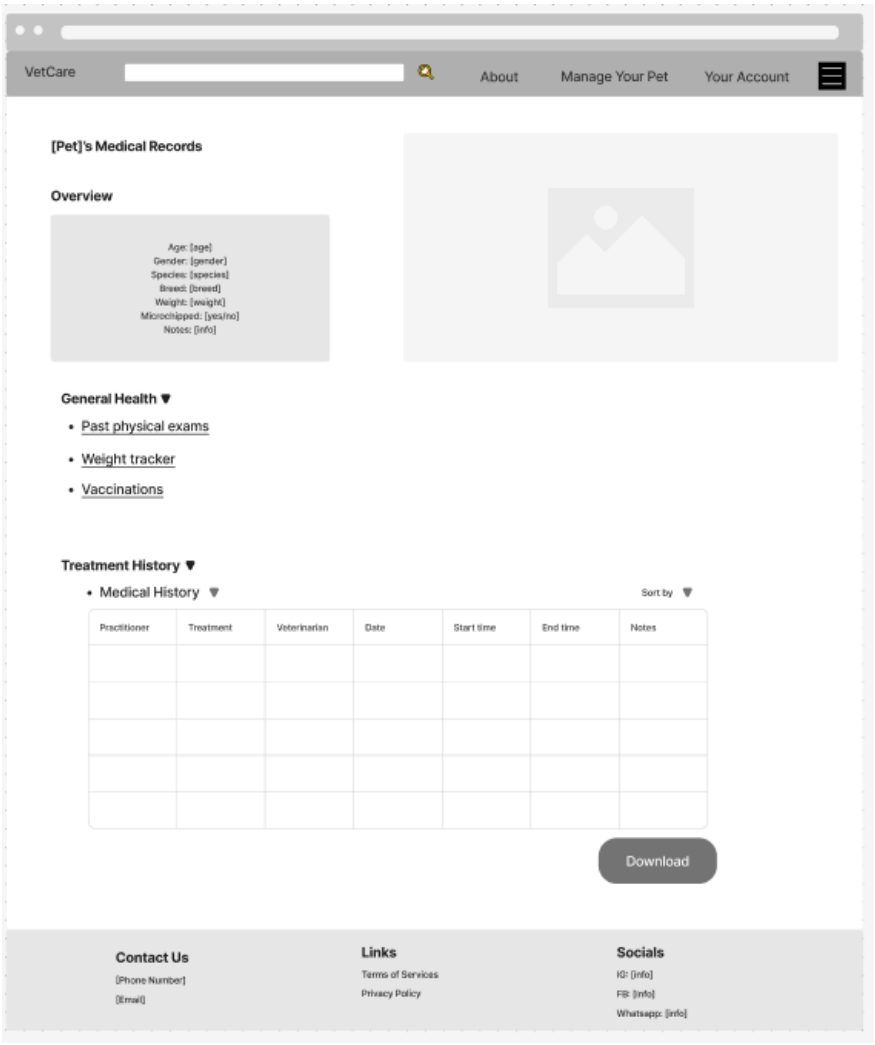
The dashboard itself is structured in blocks, with each block representing a key function of the application, such as accessing medical records, managing prescriptions, booking and managing

appointments, and tracking your order. These blocks utilise vibrant and contrasting colours, which enhance the aesthetic appeal, but also allows the user to clearly identify the difference between each feature and choose the one they would like to use with ease.

To the right of the dashboard, an image placeholder is used to allow users to manually upload pictures of their pets, to add customisation to the page to make the page more engaging to the user.

Scrolling down, a section of the homepage is reserved for personalised education articles. This area is reserved for educational resources so users can easily discover and access information relevant to them, which enhances the website's overall utility. A footer is also included at the bottom to provide users with relevant links and contact details should the user want extra information.

Figure 5: Medical Records Wireframe



The medical records page is designed to be clear and concise for the comfort of the user. It features a general overview that summarises the basic details in relation to the pet's health, such as age, gender, weight etc. This is to ensure the basic information about the pet is correct if a practitioner requires this information, and also allows the user to identify errors or incorrect information

regarding their pet in order to resolve it in a timely manner. An image placeholder is present on the right for the same purpose as the one on the dashboard, and will also utilise the same image as the dashboard.

The first section of the page, the “General Health” section, enables the user easy access to general pet records, such as past physical exam results, weight tracking access and vaccination history, if the user requires access to this information. The second section of this page features a sortable table detailing any past treatments they may have done, which will primarily consist of appointments and check-ups, but may also include any information about previous scans and tests. The table can be filtered as needed, and the results can be exported into a pdf or excel document using the “Download” button at the bottom right hand corner underneath the table. This structure is highly intuitive and is an efficient and accurate way of allowing the user to obtain relevant information.

Figure 6: Prescription Management Wireframe

The prescription management page is designed to be an organised and succinct way of displaying past and present prescription details. It is structured similarly to the medical records page for the benefit of the user, and this follows Nielsen's 4th heuristic of consistency and standard.

The first section has an overview of key information about the pet, for both the convenience of the user and any practitioners requiring access to this information. An image placeholder is present on the right for the same purpose as the one on the dashboard, and will also utilise the same image as the dashboard.

The middle section has a list of the pet's current prescriptions in a table format, with important information pertaining to the dosage and description of each prescription, alongside information such as the name of the practice that filled the prescription, the date of this filling, and the recommended date of refill. If the user has yet to receive a prescription, relevant details regarding order tracking can also be seen via a full page popup that can be clicked in the last column of the table. There is also a button that redirects directly to the refill page for the user's convenience, as well as buttons to add, edit or delete prescriptions. This allows more control to the user.
The last section is akin to the medical history table of the medical record access page. It lists every past prescription into a sortable table, which has all the applicable information required, such as dosage numbers and a comprehensive description. The table can be filtered as needed, and the results can be exported into a pdf or excel document using the "Download" button found in the bottom right hand corner underneath the table. This structure is consistent with different parts of this application, and therefore is an intuitive way of displaying all the relevant information for use.

Figure 7: Profile Management Wireframe

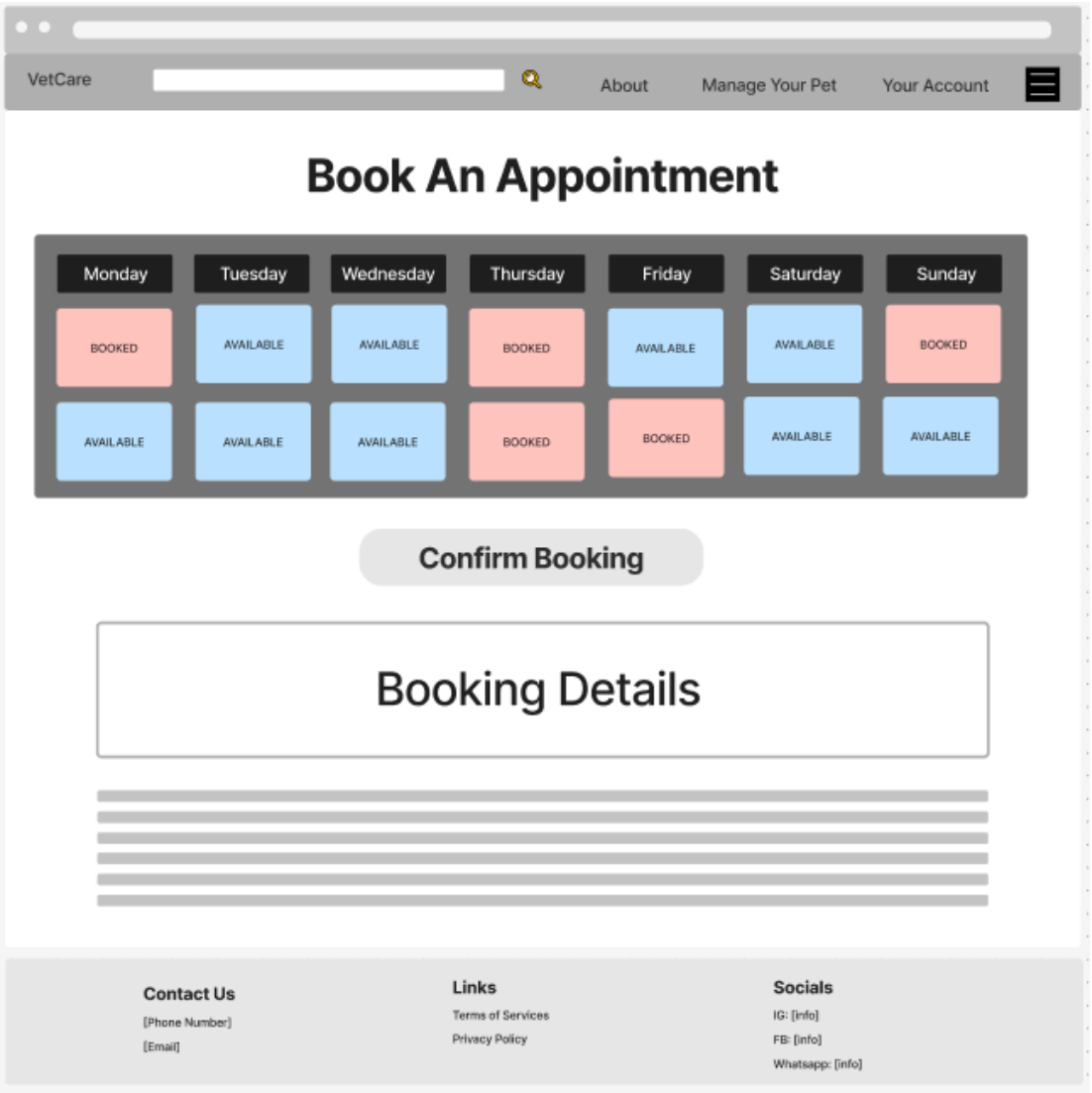
The design of the Account Details page is intuitive and user friendly. It has a focus on Nielsen's heuristic of visibility, in that there is a clear distinction between the settings tab in black on the left, as well as the profile details on the right.

The layout has several menu options on the left that are hyperlinked to indicate that it is a menu option. We then have all the edit settings sections on the left (gray boxes) that will interchange between one another depending on the setting option selected. The forms/check boxes follow a simple structure that is used consistently between our site as well as other sites in the industry. This is in an attempt to ensure consistency and standards heuristic is being adhered to.

Moreover, the large buttons used are clearly labeled and colour, not only drawing the attention to the user, but also adhering to the recall principle, making the process simple for the user to follow.

Moreover, this minimalist design, helps reduce clutter on the site, tying into the aesthetic and design heuristic.

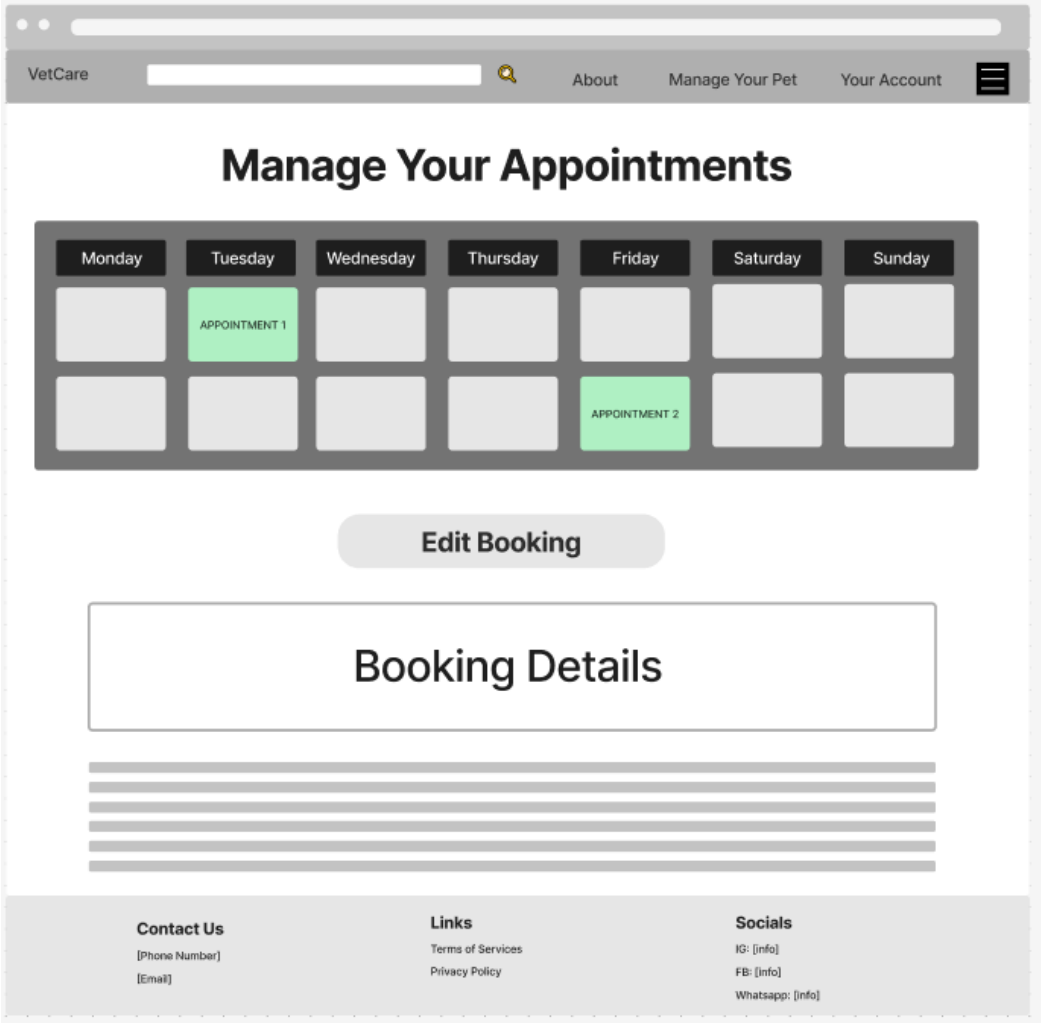
Figure 8: Appointment Booking Wireframe



The Book an Appointment page is clear and adheres to the heuristic of matching system to real world objects, as seen through the calendar format. Each day is either ‘AVAILABLE’ or ‘BOOKED’, making identification a simple process, reducing the efforts required from the user. This further ties into the recognition rather than recall principle.

Moreover, our simplistic design, combined with the ability to confirm/edit appointments, allows users to easily navigate through the site. This adheres to the error prevention heuristic, as it minimises the chance that the user makes a mistake or gets lost.

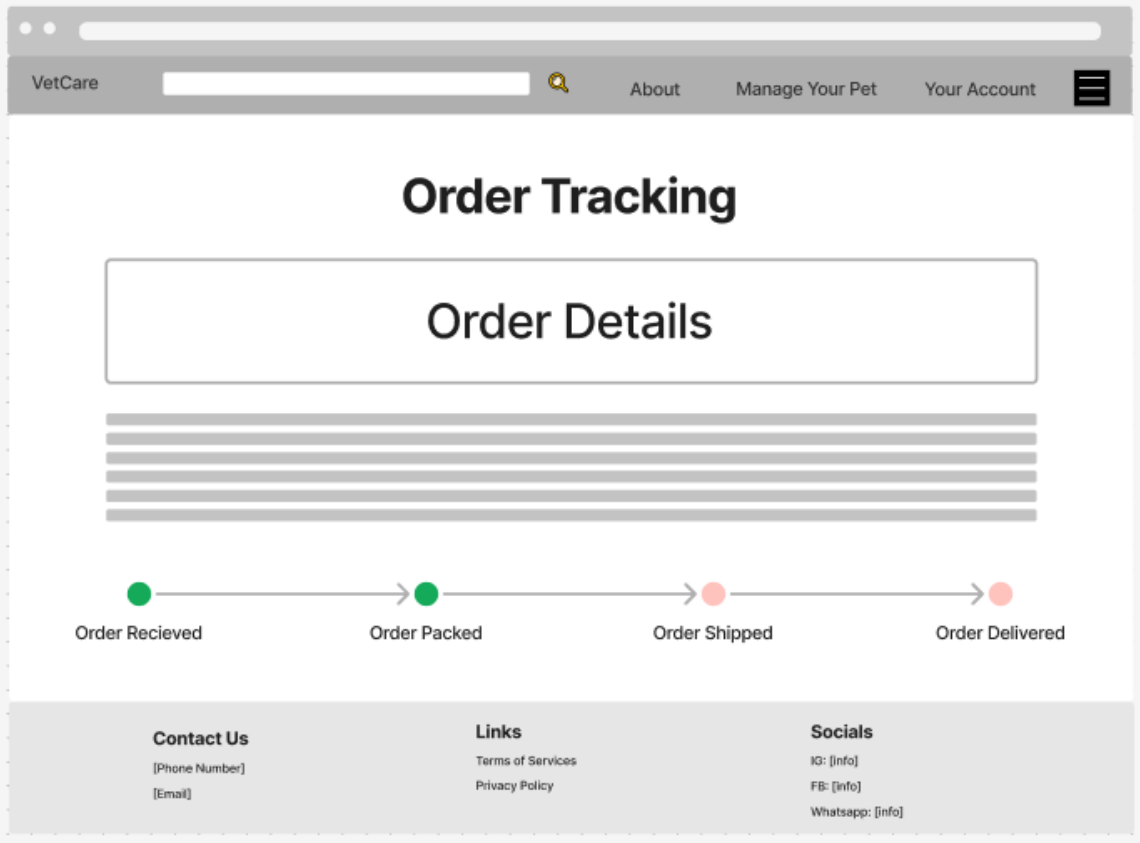
Figure 9: Manage Appointments Wireframe



This UI is designed to allow users to easily manage their appointments, clearly highlighting their existing bookings in a green box. The calendar format is consistent with the booking interface, further tying into the recognition rather then recall principle, making the booking system a familiar process for the user.

The clear labels of booked appointments such as “APPOINTMENT 1”, enable quick recognition of when a user has an upcoming appointment with a veterinarian. The option to ‘Edit Booking’, also provides the user with freedom and flexibility to modify existing appointments. The clean, uncluttered design reduces user confusion and supports the aesthetic and minimalist design heuristic.

Figure 10: Order Tracking Wireframe



The Order Tracking UI is designed to be both intuitive and informative, adhering to *Nielsen's heuristic of visibility* of system status by clearly showing the order's progress through different stages: Order Received, Order Packed, Order Shipped, and Order Delivered.

The order tracking UI is designed to be a intuitive, informative and simplistic as possible, to minimise any potential errors that the user may encounter. This is achieved through providing different stages ranging from 'Order Recieved' through to 'Order Delivered', meaning the user can identify exactly where their product is.

The step by step process also adheres to the match the system and real world heuristic, as it mimics a real world order process. The interface was created to be as straight forward as possible, communication essential information whilst also reducing clutter. This was in an attempt to adhere to the aesthetic and minimalist design heuristics.

Figure 11: Articles Page Wireframe

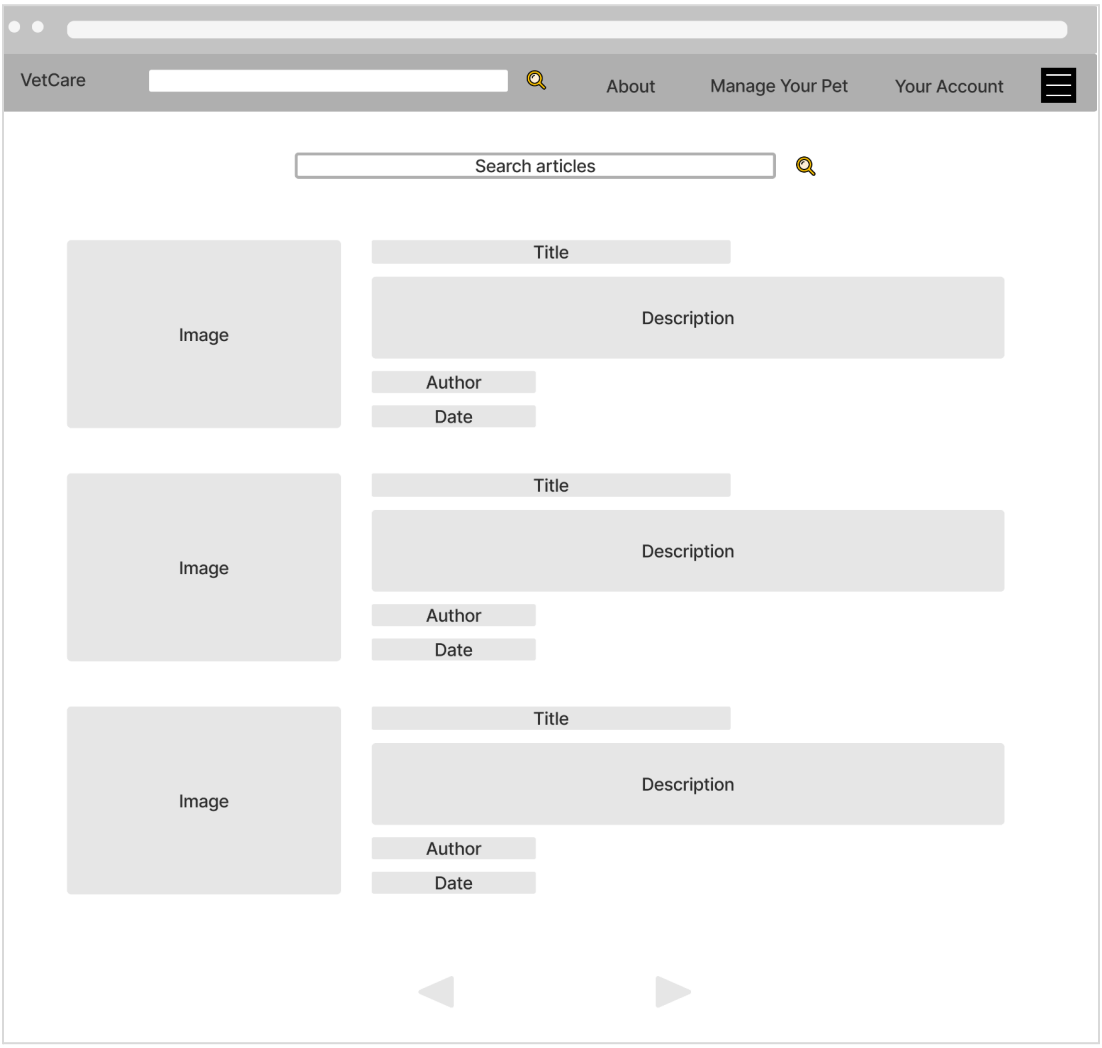


Figure 12: Mobile View Dashboard Wireframe

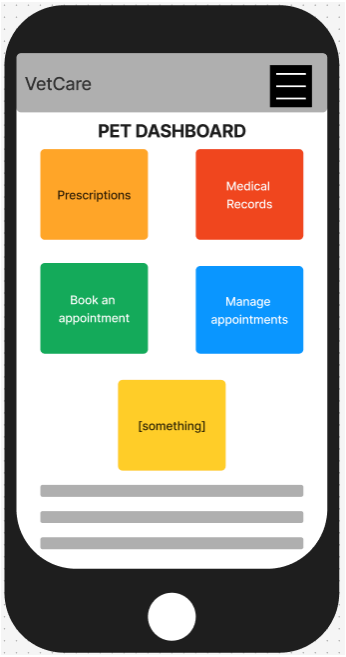
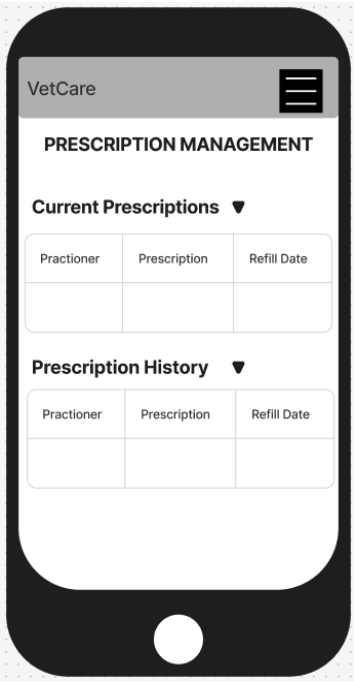


Figure 13: Mobile View Prescription Management Wireframe



Our mobile design for the VetCare web application closely follows *Nielsen's heuristics* to create an intuitive and user-friendly interface. Due to the size constraints of a mobile view, we had to prioritise visibility and clarity, due to the lack of space to provide information. We achieved this through the user of expandable menus, such as the burger menu in the top right corner, which allows the user to view different pages if necessary. However, when it is not in use, it is reduced in size and out of the way, which reduces clutter as well as cognitive load of our user.

Moreover, our design incorporates collapsible tables for our Prescription Management screen. This allows us to further reduce clutter, but also control the information that they wish to view. This prevents information overload and allows for simplistic navigation.

Lastly, we emphasized consistency and standards throughout the design. We utilised consistent colour patterns across the site, as well as simple iconography, which adheres to the recognition rather then recall heuristic, as the user will be familiar with the environment.

Figure 14: Mobile View Articles Page Wireframe

User Interface Implemented In Sprint 1:

Figure 15: Navigation Bar Logged Out View

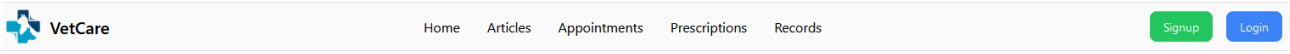
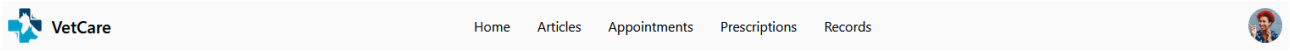


Figure 16: Navigation Bar Logged In View



The navigation layout that we decided to implement was designed to try and provide the user with an intuitive and simplistic style that avoids confusion. In figure 13, we can see the view that a logged out user would see, which has 2 clear ‘signup’ and ‘login’ buttons that the user can select depending on their requirements. Furthermore, having each of the pages clearly indicated in the middle of the navigation bar, helps the user navigate through the site to their desired location. This adds to the user experience due to its clear and simple nature.

Upon the user logging into their account, they are prompted with a different navigation bar as seen in figure 14, this removes the ‘login’ and ‘signup’ button and replaces it with a profile icon that clearly displays the users profile. This also has a drop down menu that allows the user to either logout or view their profile icons. We has a focus on trying to reduce the number of items within the navbar to reduce clutter. This was in an attempt to really stress the key parts of our site, as well as key functionalities such as logging out, such that the user doesn’t get disoriented or confused with any processes.

Figure 17: Signup Form

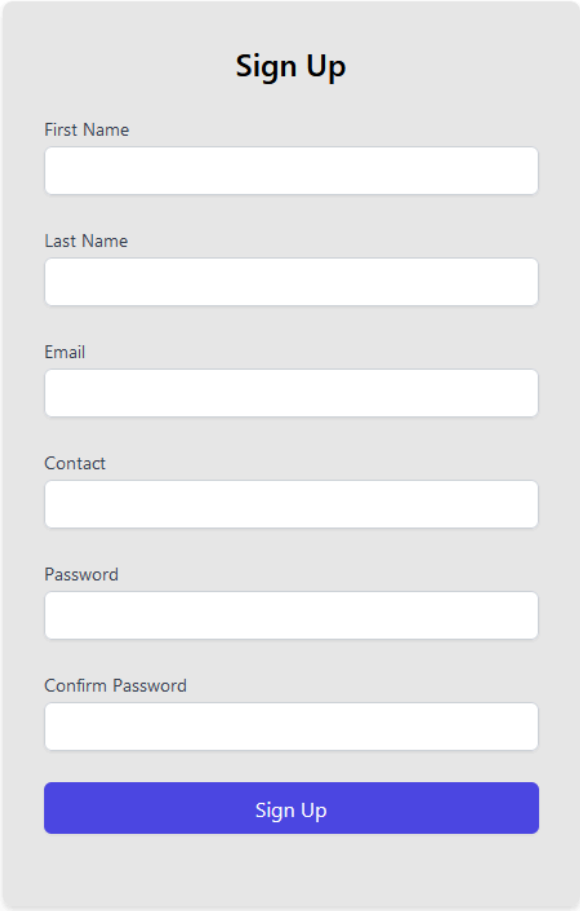
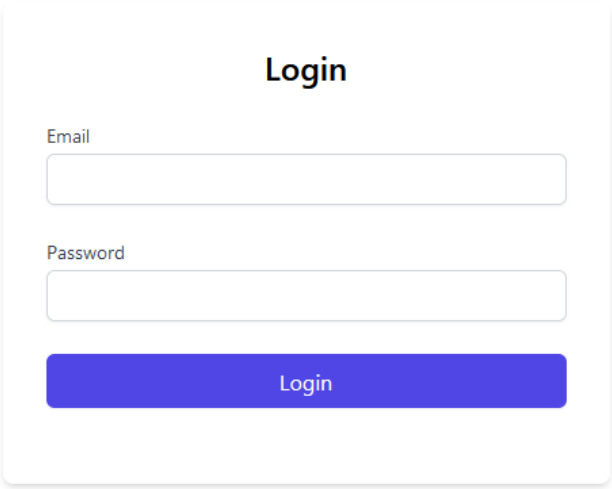
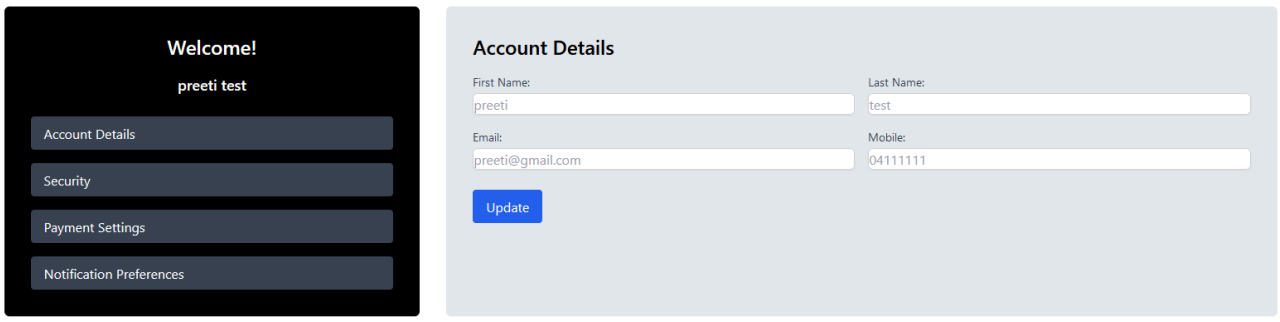


Figure 16: Login Form



The signup and login forms were created to be very similar, in an attempt to appeal to Nielsens design heuristics, ensuring that systems are replicated from what the users are used to. Moreover, through repetition, we created very similar forms, with similar colours, which helps create a unified process that users can follow through the site. This can be seen within both figure 15 and 16. Furthermore, each of the form inputs are clearly labeled to help the user identify what to enter in the fields. We also embedded various error codes in case of duplicate emails, non-matching passwords in attempt to help guide the user in case they were doing something incorrectly.

Figure 18: User Profile Page

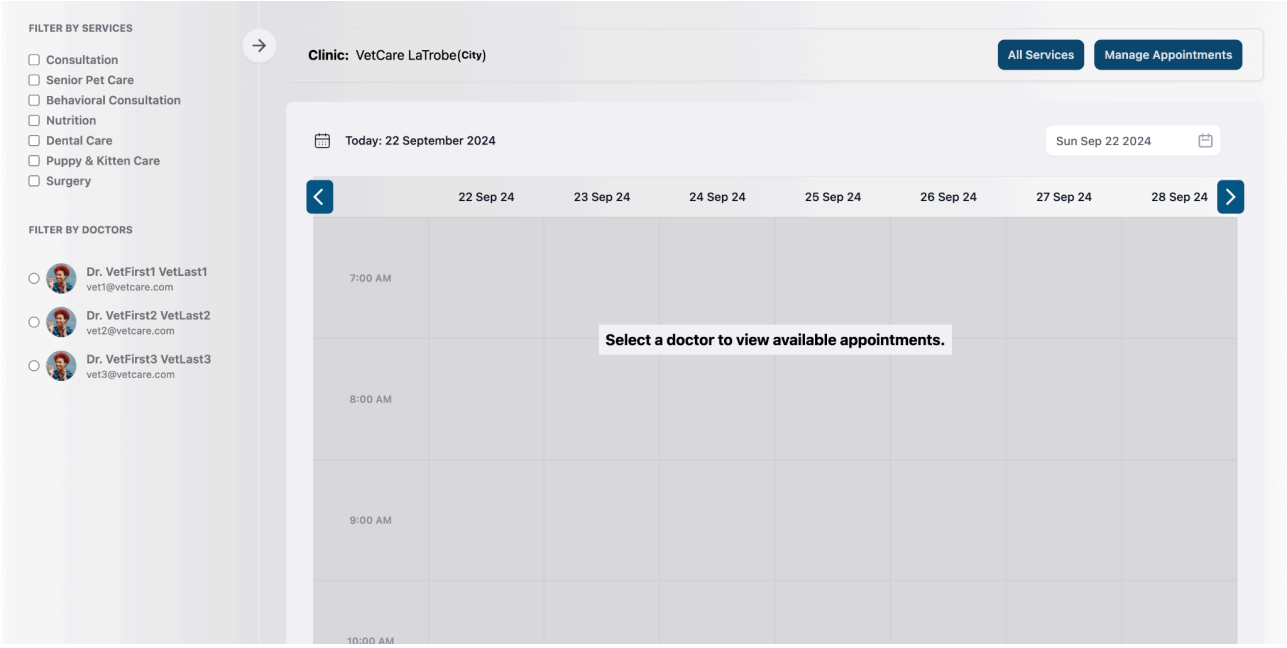


The user profile page follows a very clear structure, dividing the page into two clear parts. The left side, within figure 17, is black and contains the user details, as well as a series of buttons that the user can click on to edit details. The right side of the page contains the selected buttons display, whether it is regarding account details, security, notifications, payment settings, it will display that relevant section. We implemented this well defined structure such that the user has freedom and control to view and edit what they please.

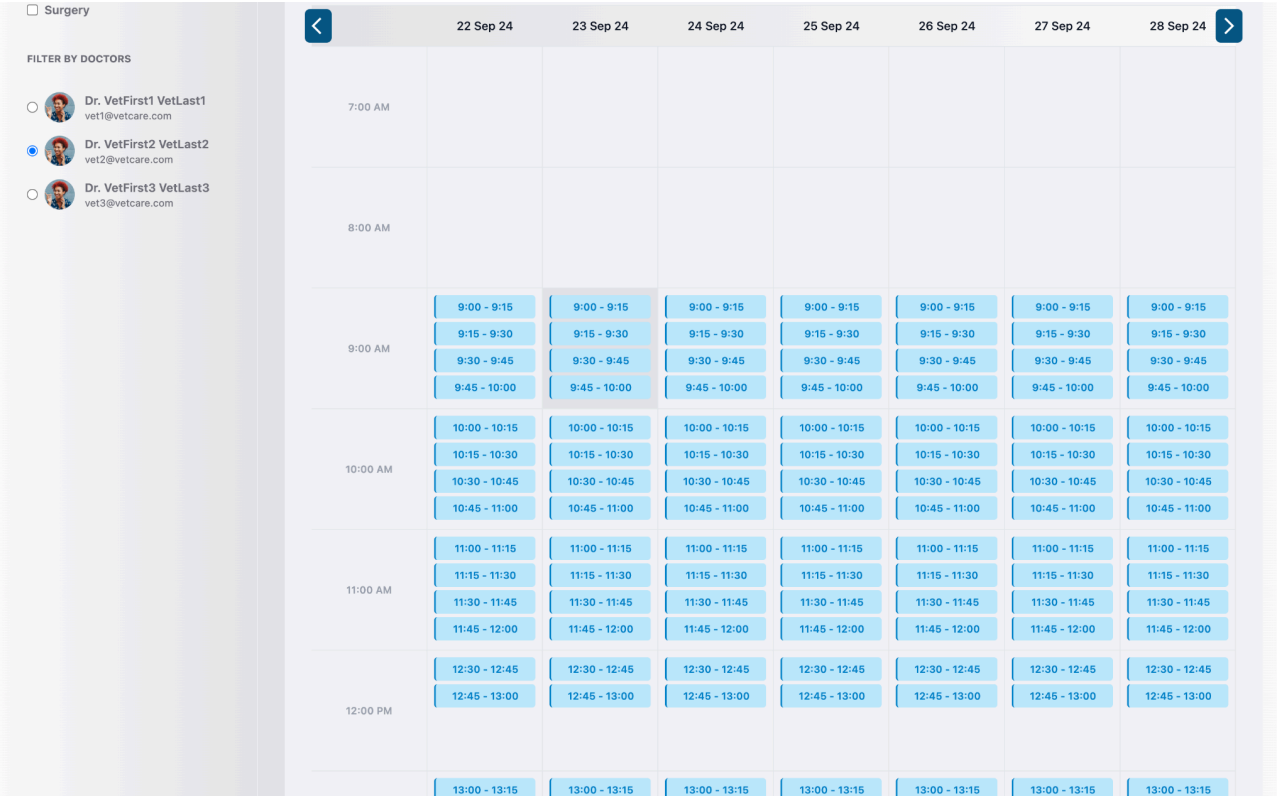
We also implemented the input fields to contain the preexisting content as seen in figure 17, we can see the existing user name, last name etc… which helps the user to identify what they are editing. Furthermore, we used buttons that are not only distinct, but also the same as what was used in the login and signup forms, to further help guide the user around the site.

Appointment Booking Page has 3 states, which are as follows:

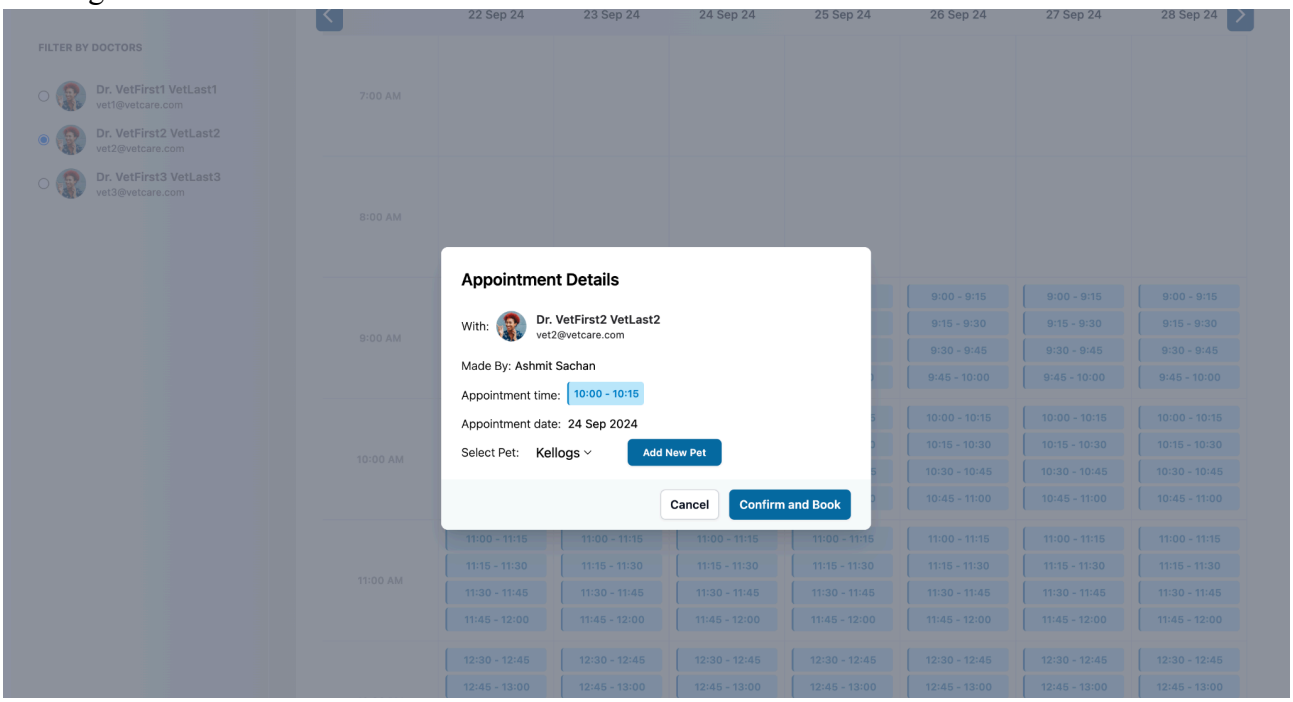
1. Initial stage: Juist displays options for filters and an option to select or change dates and user's preferred veterinarian



2. Veterinarian selected stage: Once the veterinarian has been selected by the user, available slots are shown.

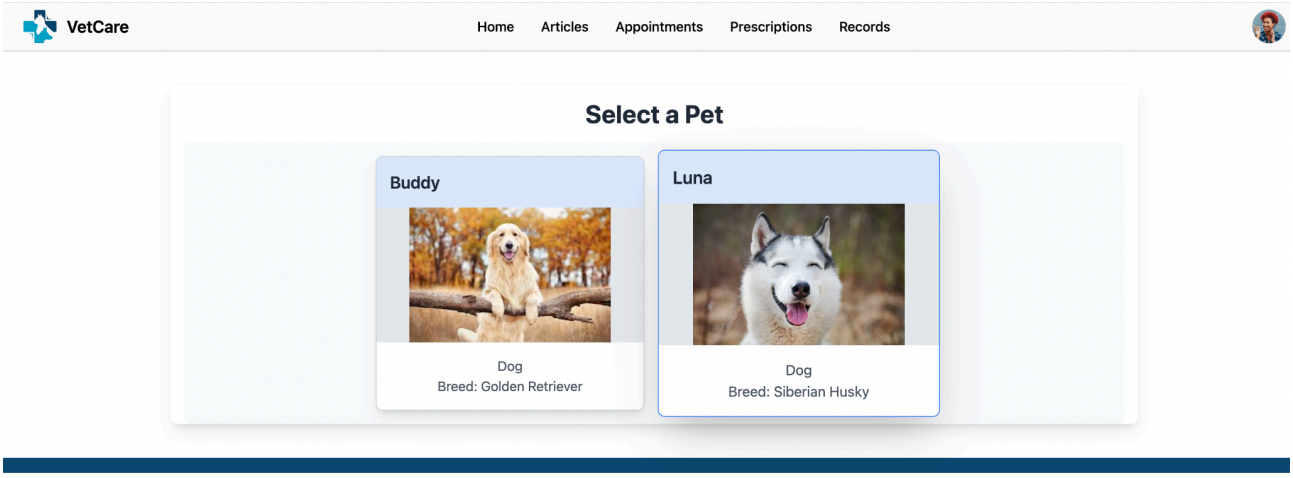


3. Booking confirmation stage: Upon selecting the slot the user can select the pet and confirm booking.



Medical record Access:

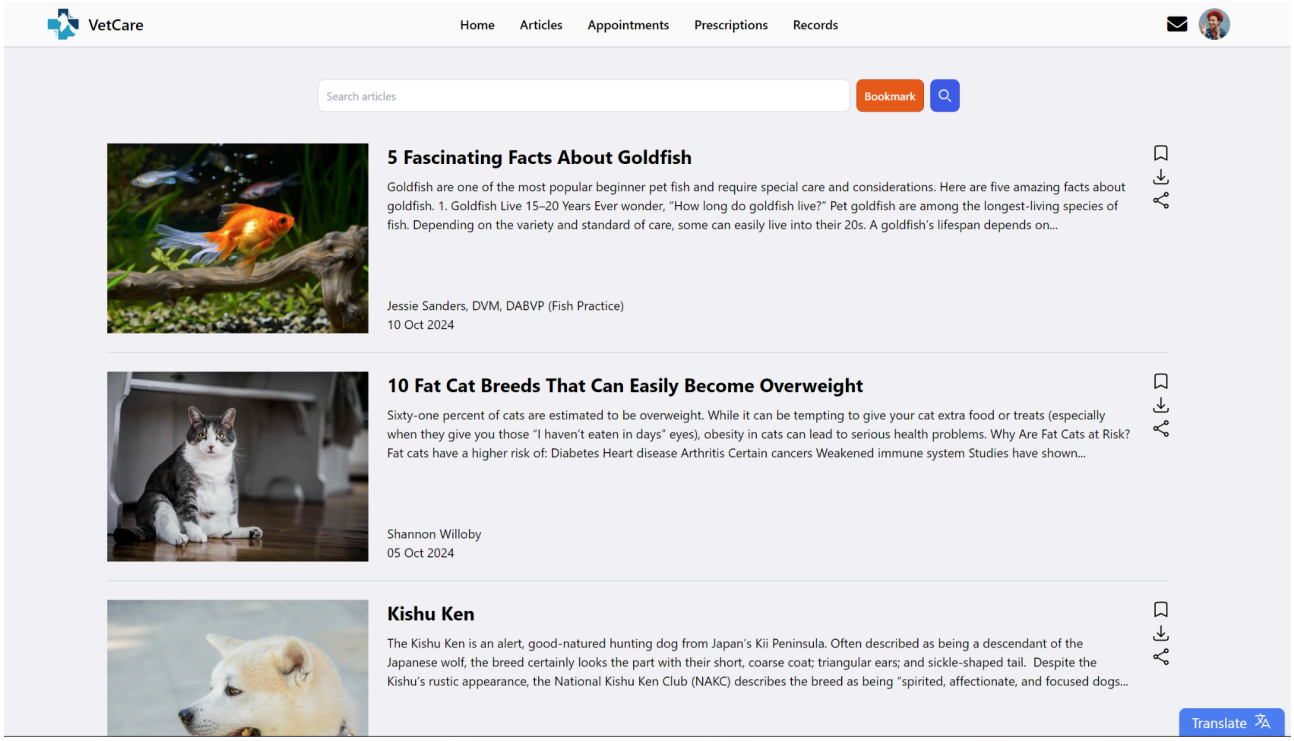
1. First page: the first page shows all pets the owner has added and allows user to select pet



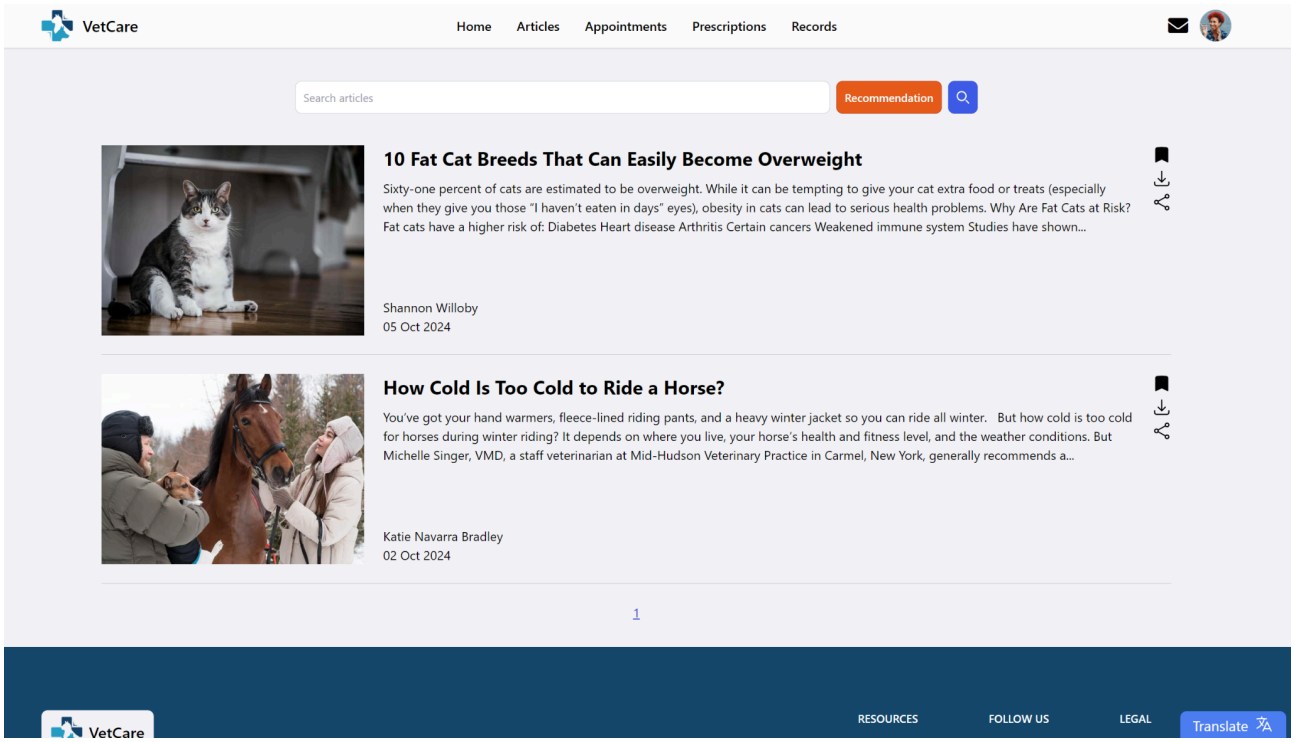
2. Records Page: History of the pet is shown in this page with an option to download the records and share them

Educational Resources:

1. Feed page: The newest articles fetched from the rss feed are shown, with the option to bookmark, download, change page, translate and view each article in a new page. There is also a button to toggle the bookmark view.

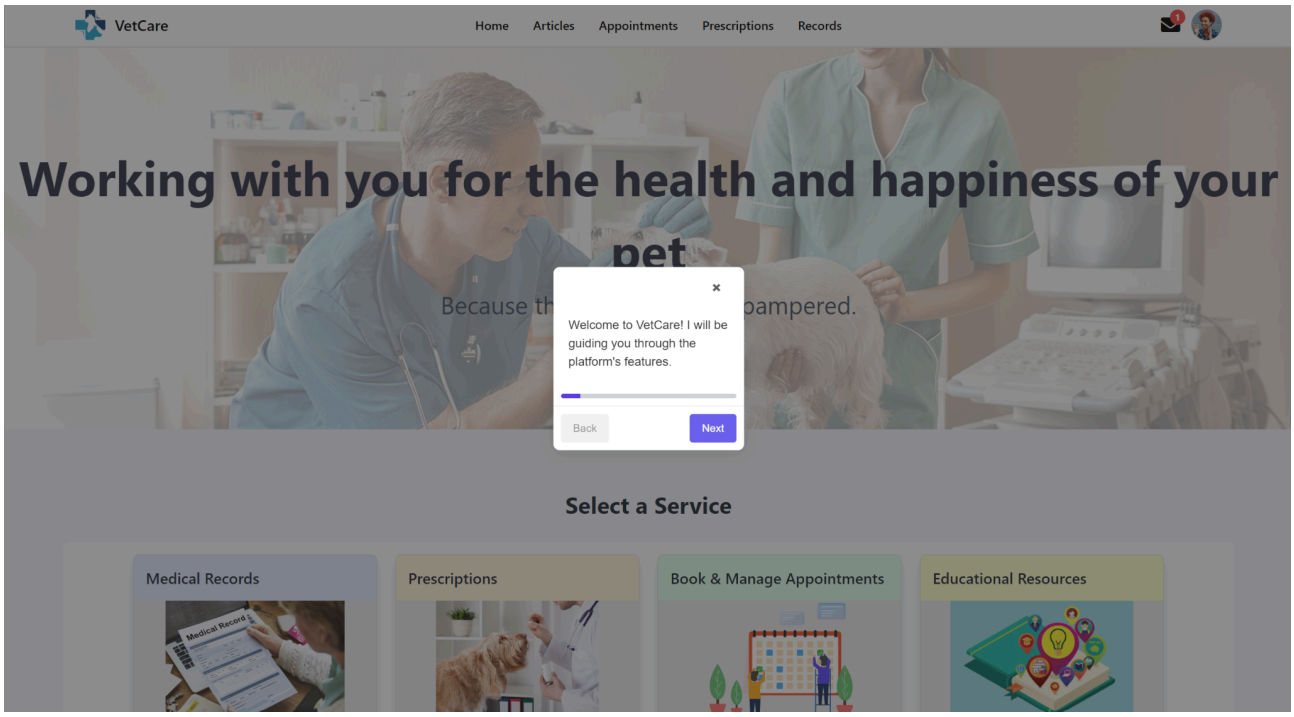


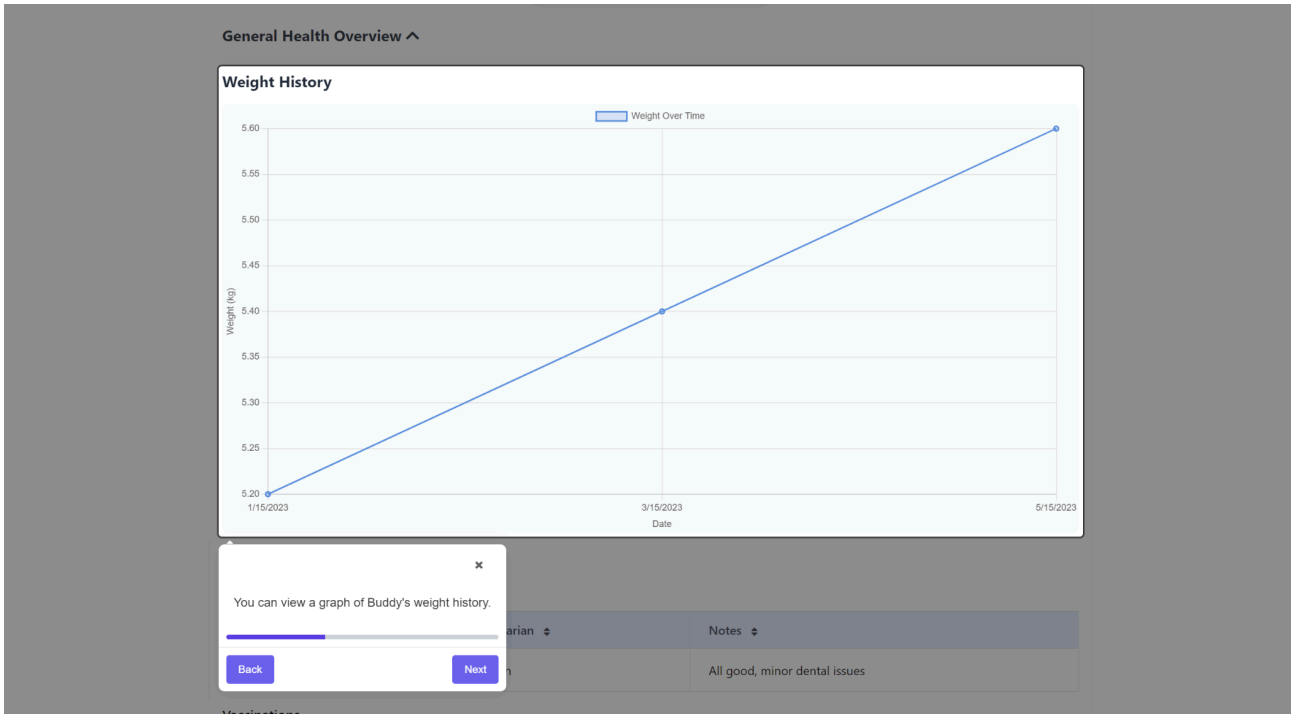
2. Bookmark page: The bookmarked articles of the logged in user are shown, with the same options.


VetCare
Home
Articles
Appointments
Prescriptions
Records
Search articles
Recommendation
10 Fat Cat Breeds That Can Easily Become Overweight
Sixty-one percent of cats are estimated to be overweight. While it can be tempting to give your cat extra food or treats (especially when they give you those "I haven't eaten in days" eyes), obesity in cats can lead to serious health problems. Why Are Fat Cats at Risk? Fat cats have a higher risk of: Diabetes Heart disease Arthritis Certain cancers Weakened immune system Studies have shown...
Shannon Willoby
05 Oct 2024
How Cold Is Too Cold to Ride a Horse?
You've got your hand warmers, fleece-lined riding pants, and a heavy winter jacket so you can ride all winter. But how cold is too cold for horses during winter riding? It depends on where you live, your horse's health and fitness level, and the weather conditions. But Michelle Singer, VMD, a staff veterinarian at Mid-Hudson Veterinary Practice in Carmel, New York, generally recommends a...
Katie Navarra Bradley
02 Oct 2024
1
VetCare
RESOURCES
FOLLOW US
LEGAL
Translate

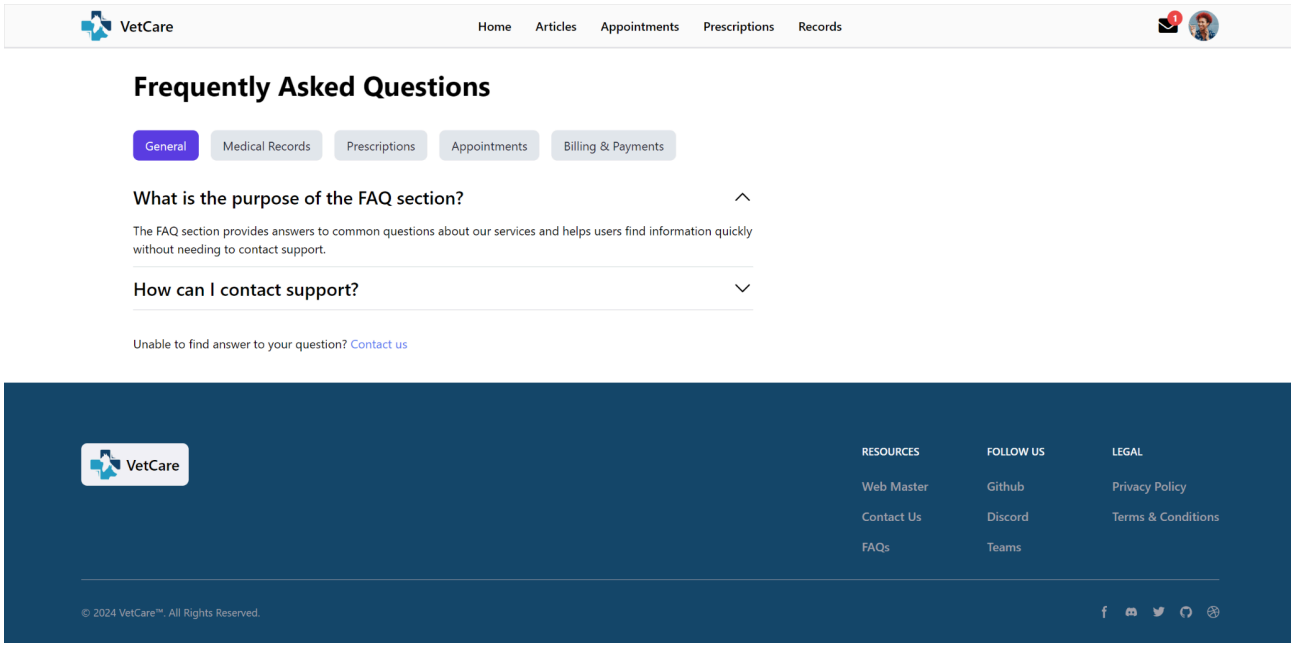
User Interface Implemented In Sprint 2:

User onboarding guide: New users will be greeted with the onboarding guide. Users are able to skip or use the guide at their own pace. Existing users are able to restart the onboarding guide in the settings.

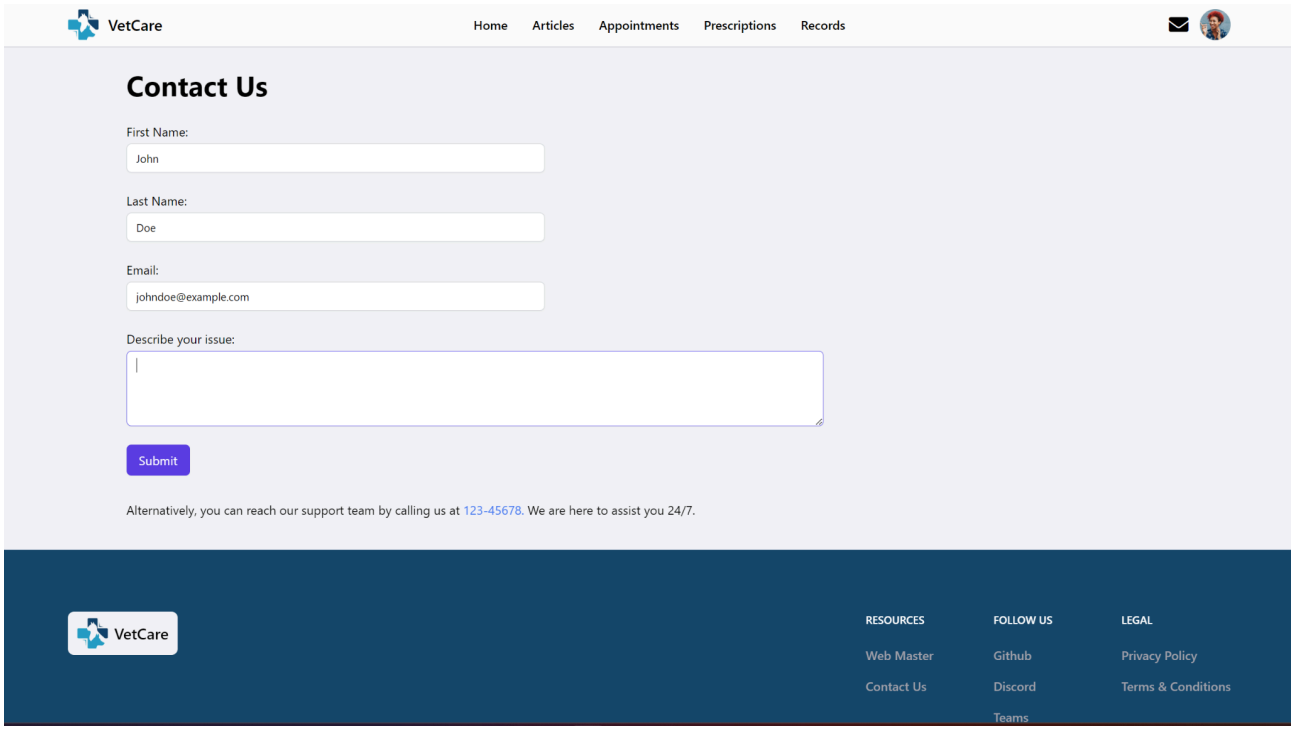




FAQs page: Users are able to filter the questions by category. The default category is “general”. Users are able to click on the questions to expand or collapse the answers.

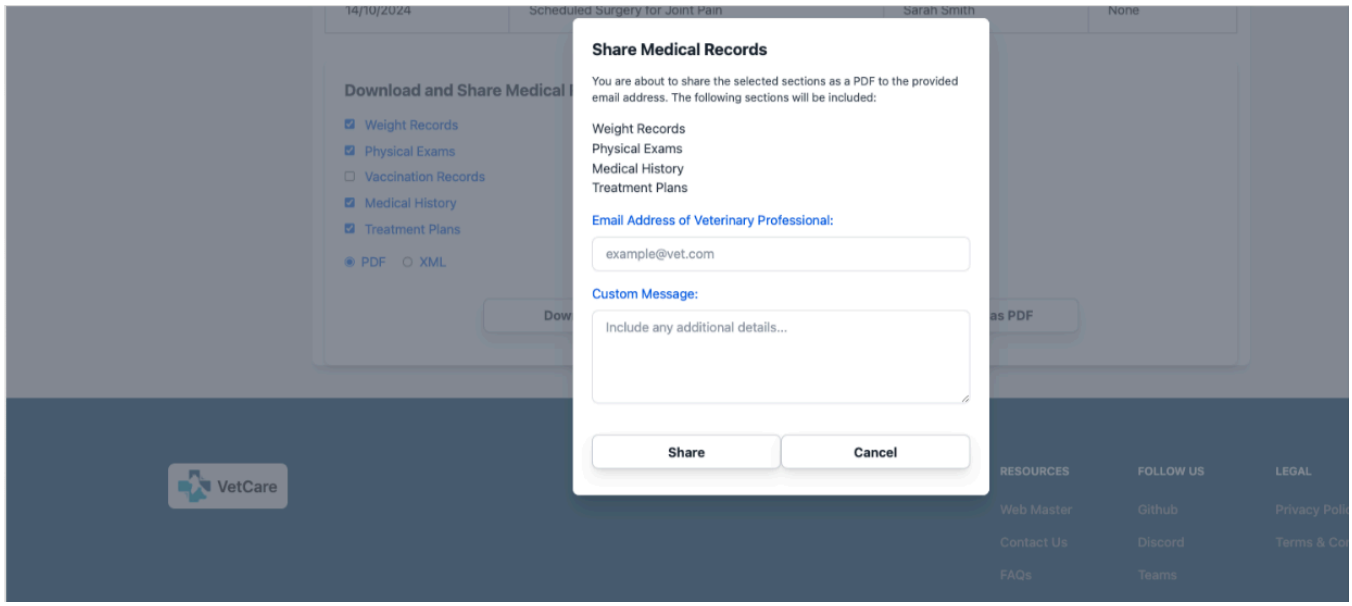


Contact Us page: The details for logged in users will be automatically filled.



Medical Records PDF, Distribution - Preeti

The user can download the PDF according to what records they'd like to download in either PDF or XML and then share accordingly with the veterinarian.



The following is a screenshot of how the pdf document looks like.

Medical Records for: Buddy

Records as of: Sun Oct 13 16:05:09 AEDT 2024

Weight Records

Date	Weight (kg)
2023-01-15	5.2
2023-03-15	5.4
2023-05-15	5.6
2023-07-15	5.8

Full Medical History

Date	Treatment	Practitioner	Veterinarian	Notes
2023-08-10	Vomiting	John Doe	John Doe	Prescribed antiemetic medication
2023-05-12	Sprained leg	Karl Malus	Sarah Smith	Prescribed rest and anti-inflammatory medication
2023-01-25	Fever	John Doe	John Doe	Fever observed, prescribed antipyretics

Vaccination Records

Vaccine Name	Vaccination Date	Administered By	Next Due Date
Rabies	1970-01-01	John Doe	2024-10-14
Parvovirus	1970-01-01	John Doe	2024-10-14

Treatment Plans

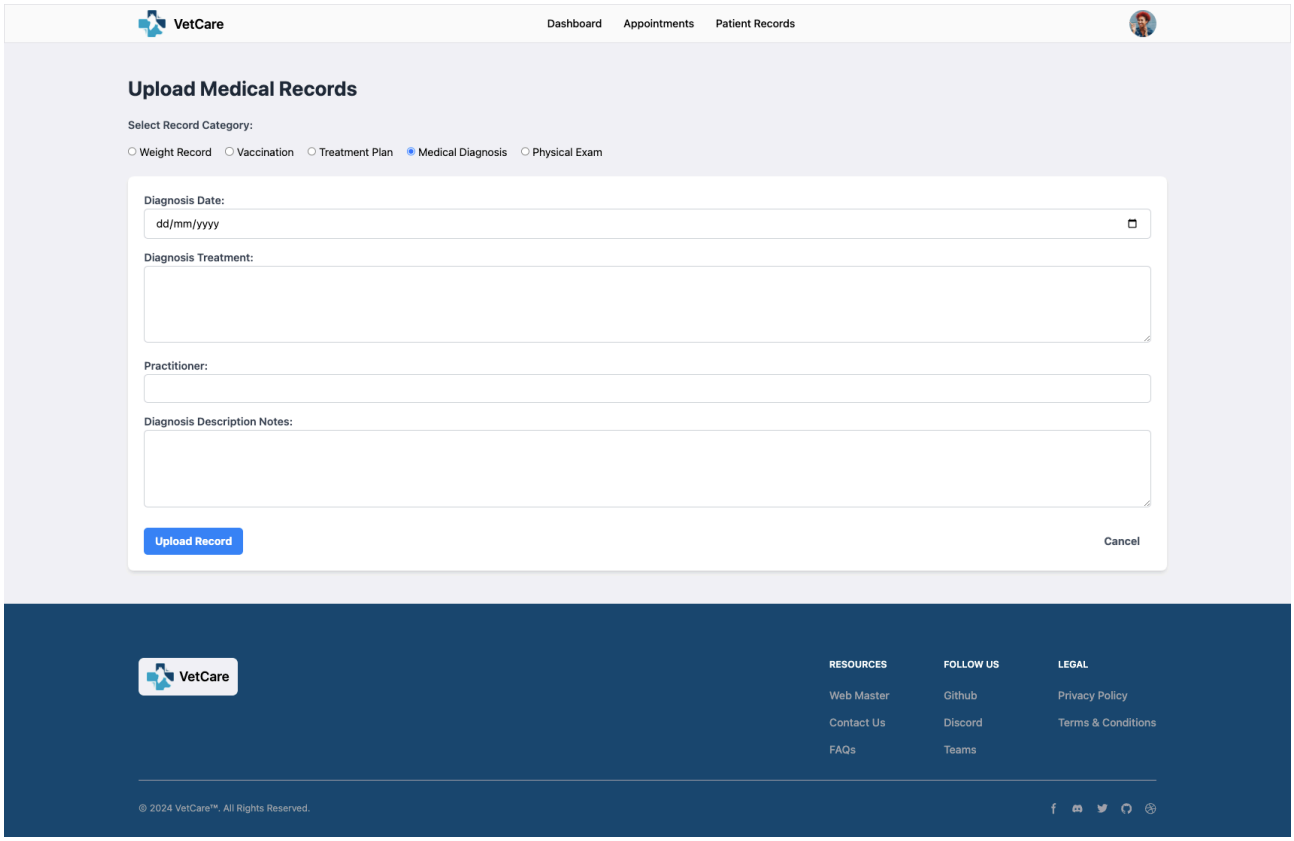
Plan Date	Treatment Description	Practitioner	Notes
2024-10-14	Arthritis Management	John Doe	None
2024-10-14	Scheduled Surgery for Joint Pain	Sarah Smith	None

Medical Record Management - Vet Side - Preeti

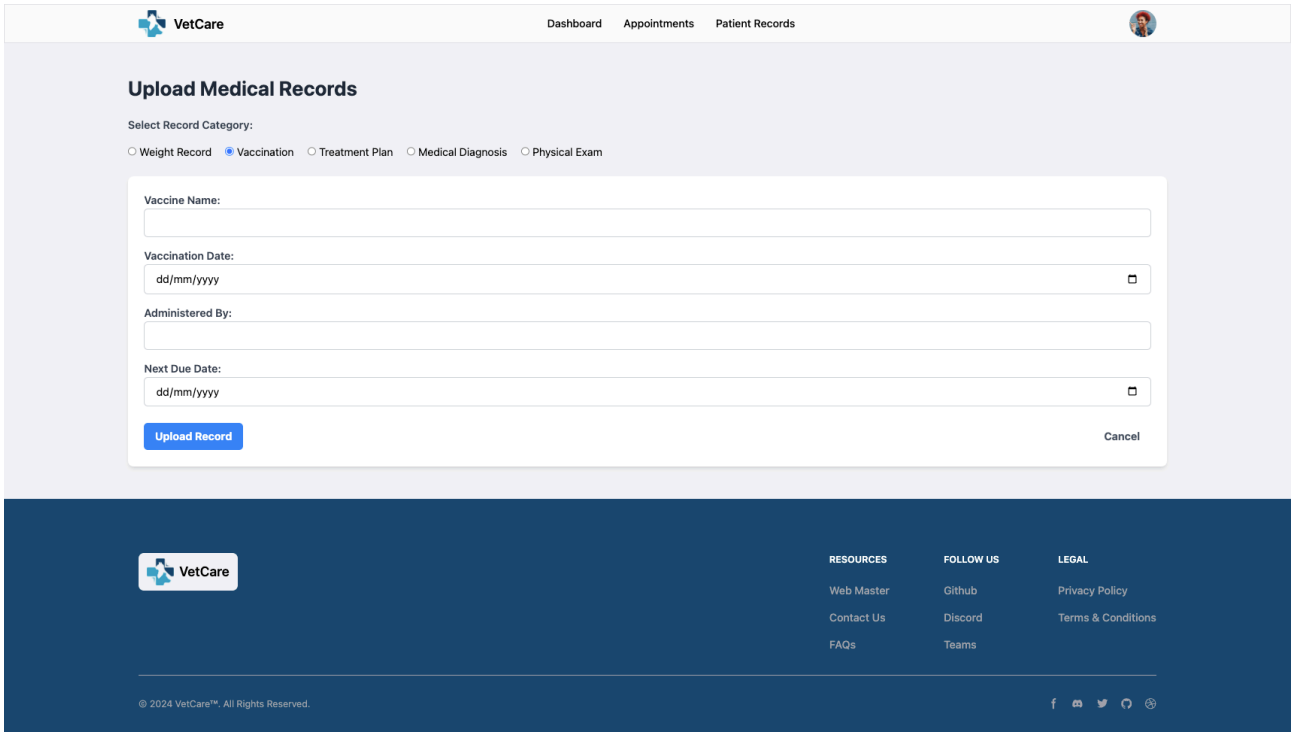
The vet can login and go to corresponding appointment and then click on upload medical records for that appointment which is associated with a pet.

Then the vet can see the respective forms with their categories and upload data accordingly. Upon successful upload, it will be reflected in the user side as well accordingly.

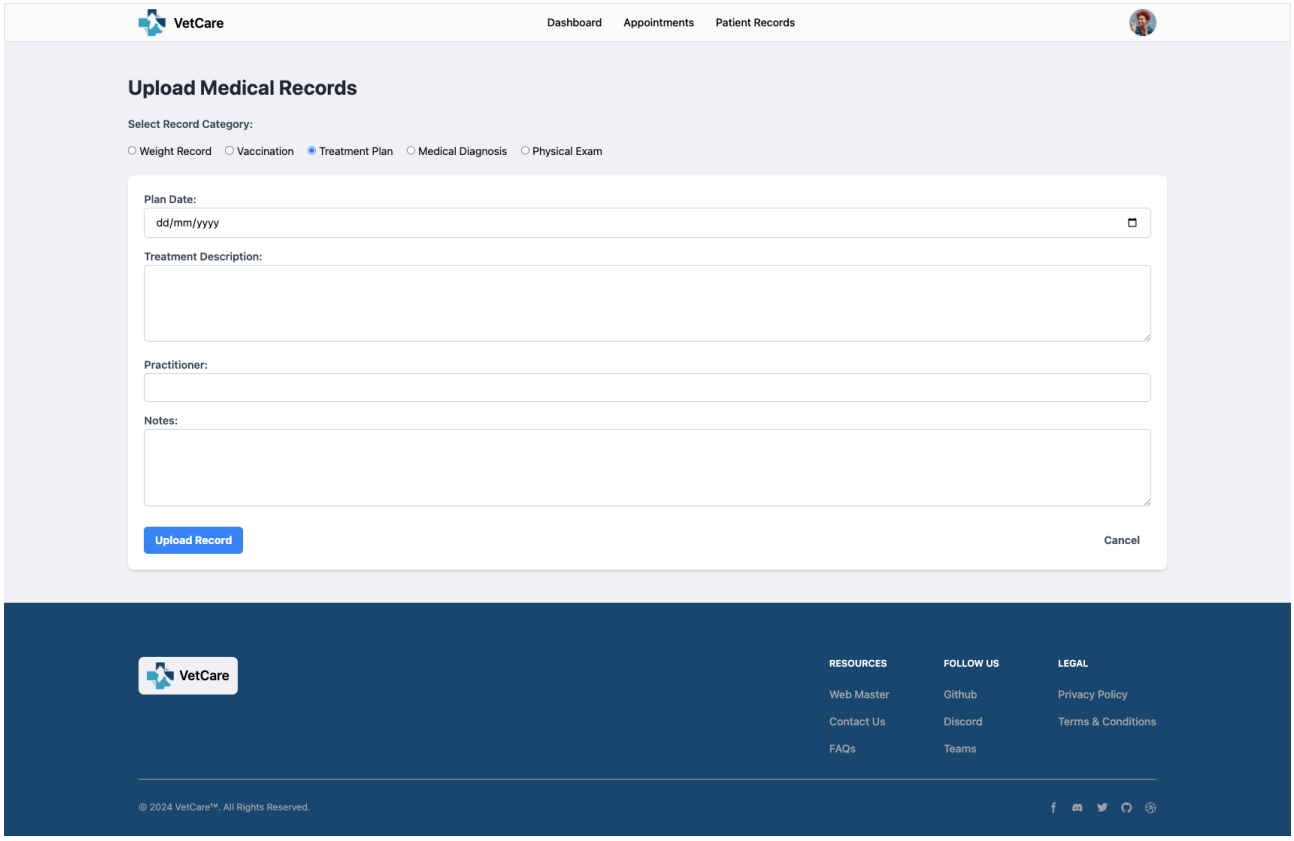
Form for **Medical Diagnosis** Record Upload:



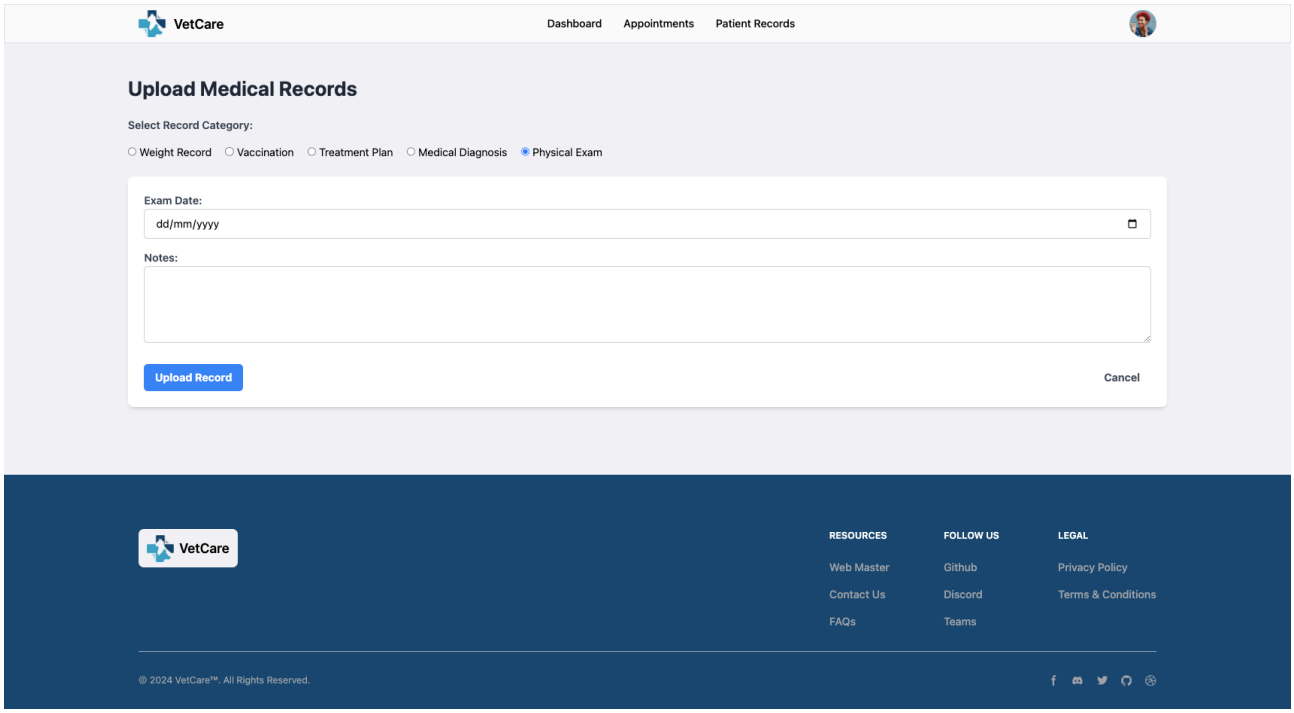
Form for **Vaccination Record** Upload:



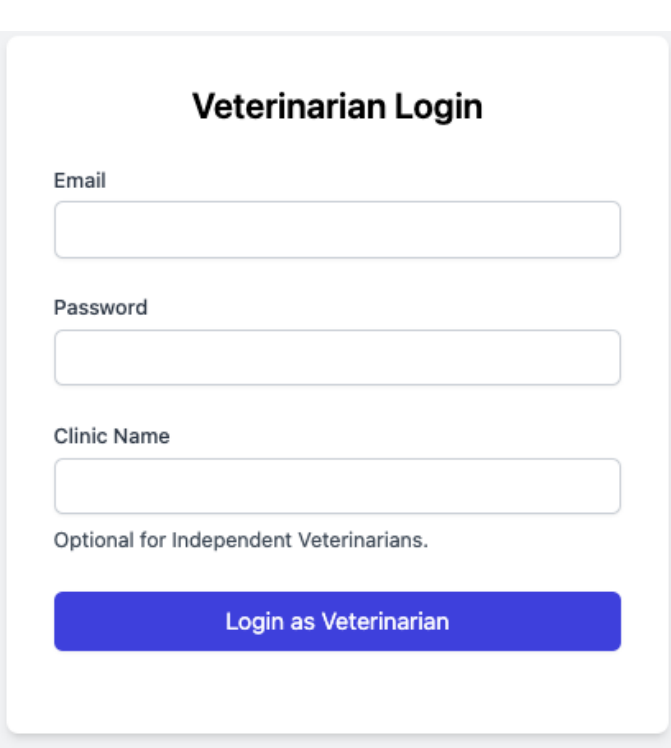
Form for **Treatment Plan Records** Upload:

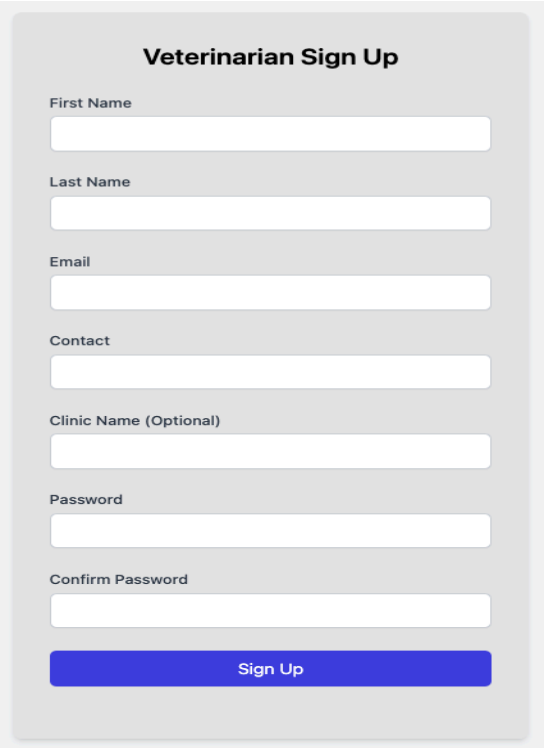


Form for **Physical Exam Record** Upload:



Vet Login and Signup - Preeti

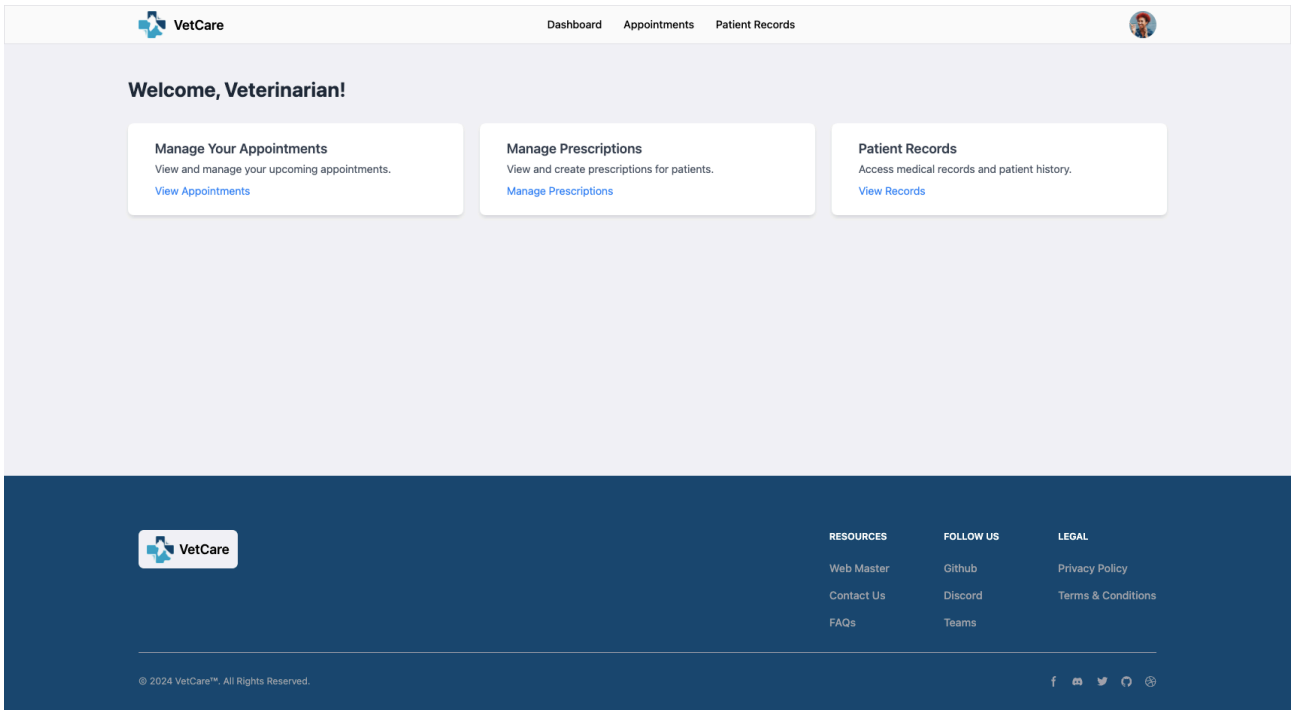




The veterinarian login and signup is similar to the signup and login for user, the difference mostly lying in the fact that a Clinic Name is required if veterinarian wants to be linked to a particular clinic already linked with us. Otherwise, without it the veterinarian is treated as an independent user and not linked to any particular clinic.

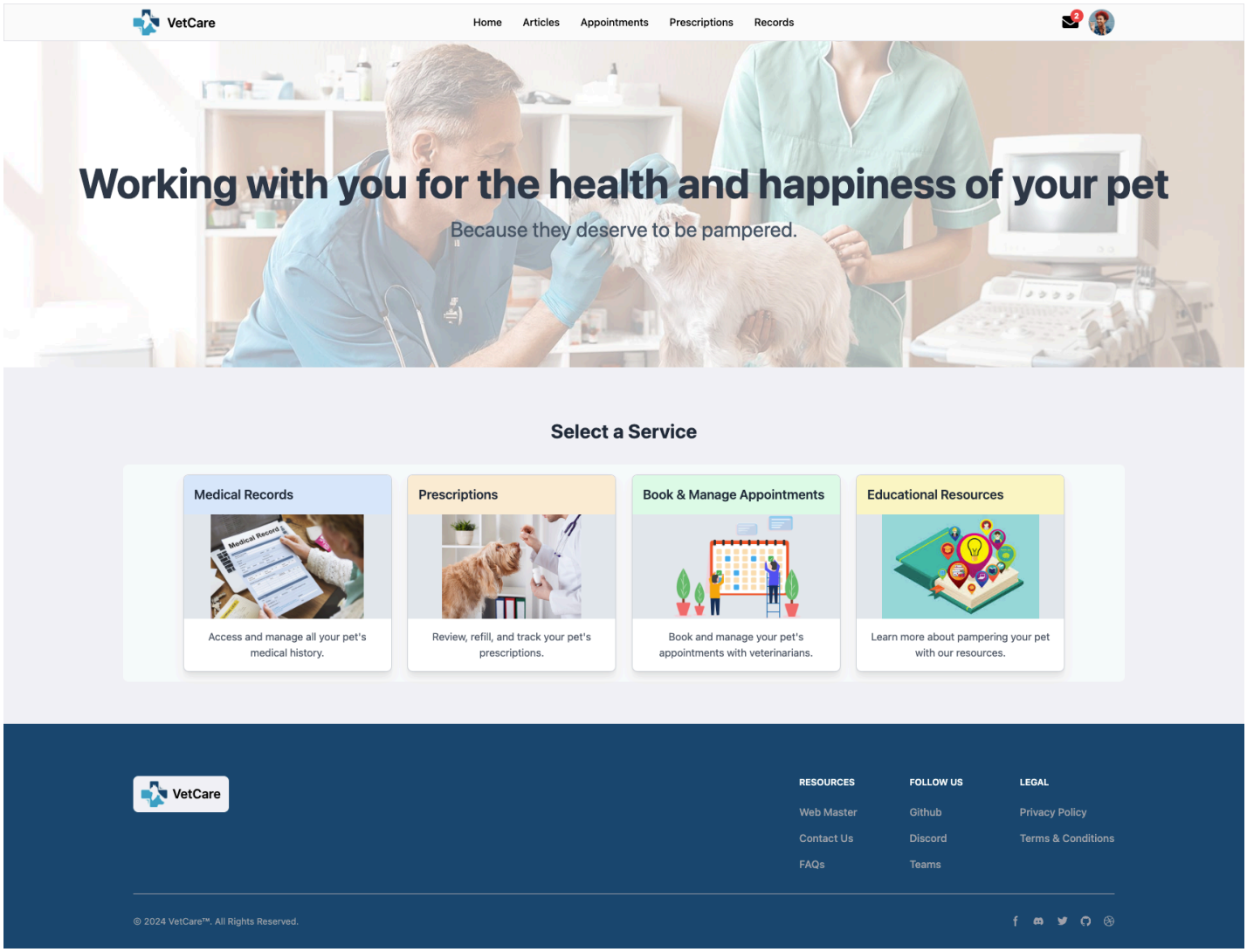
Vet Dashboard - Preeti

The following is the Vet Dashboard that shows up once the vet signs in successfully. It has Appointments, Patient Records and Manage Prescriptions.



User Home Page UI Design - Preeti

For this sprint, the home page UI was redesigned to look liek the following with selectable services cards.

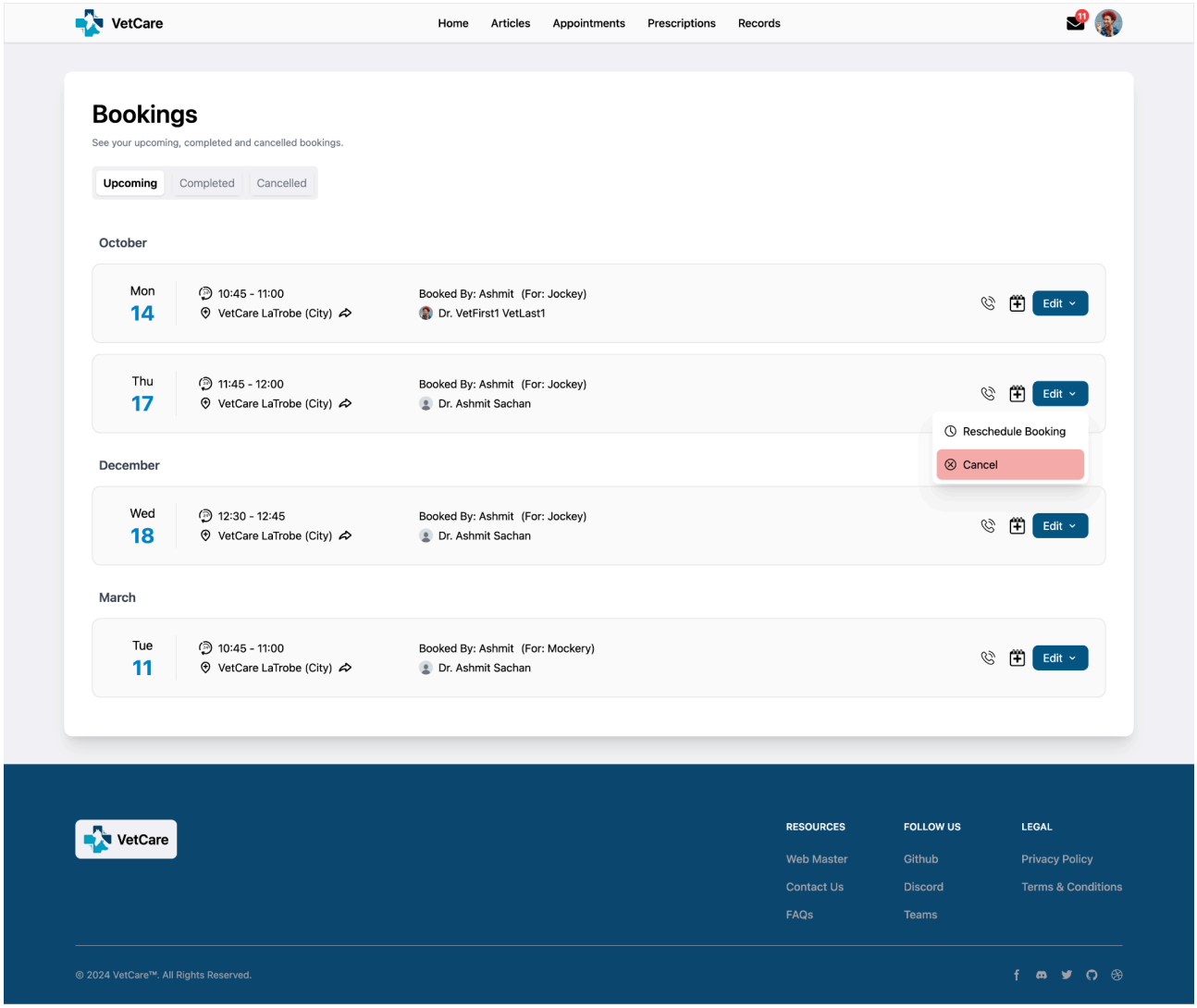


User Appointment Management Section - Ashmit

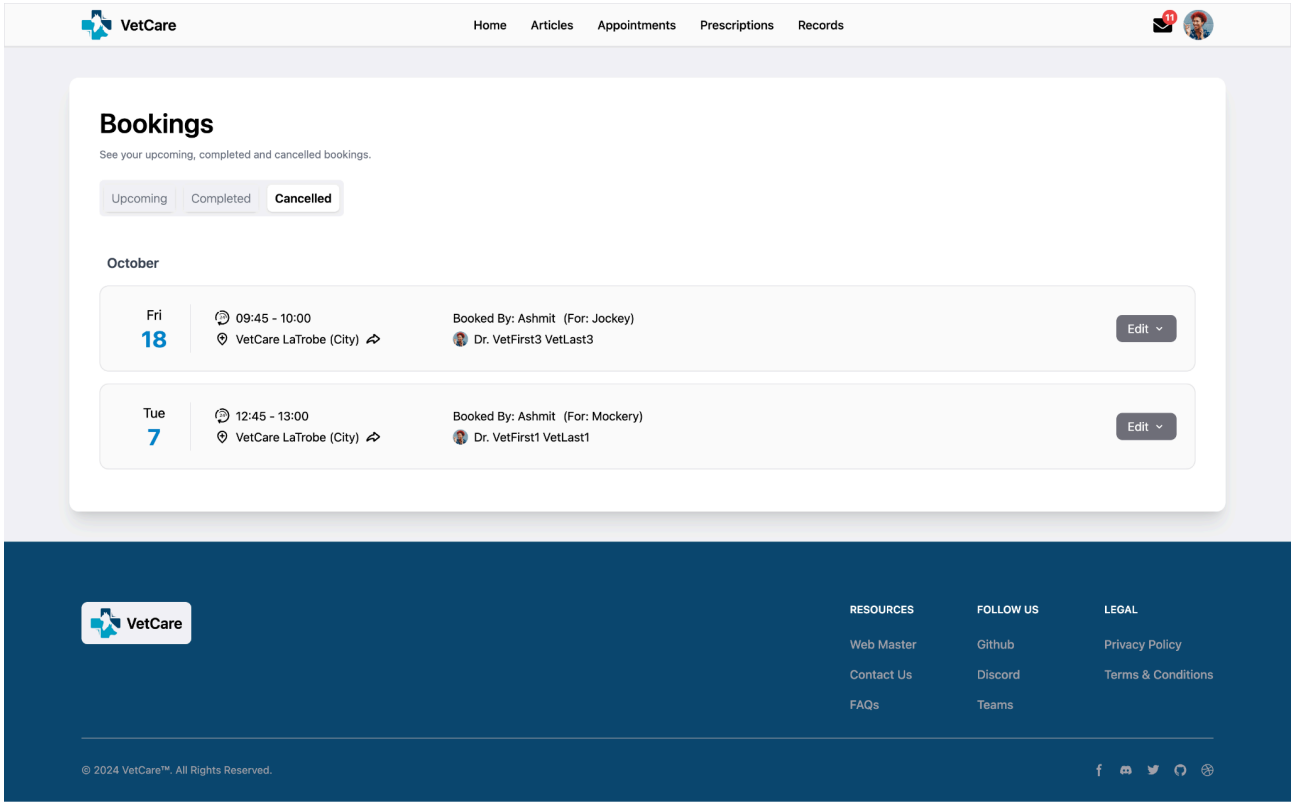
For this sprint, I designed The UI and backend functionality to edit existing appointments and cancel then.

1. This view also provides options to view all upcoming, completed and cancel appointments.
2. The user has options to view the location of the clinic on google maps.
3. The user has options to add the appointment details to their respective calendars, Google calendar or Apple calendar respectively just by one click of a button.
4. The user has options to cancel an appointment.

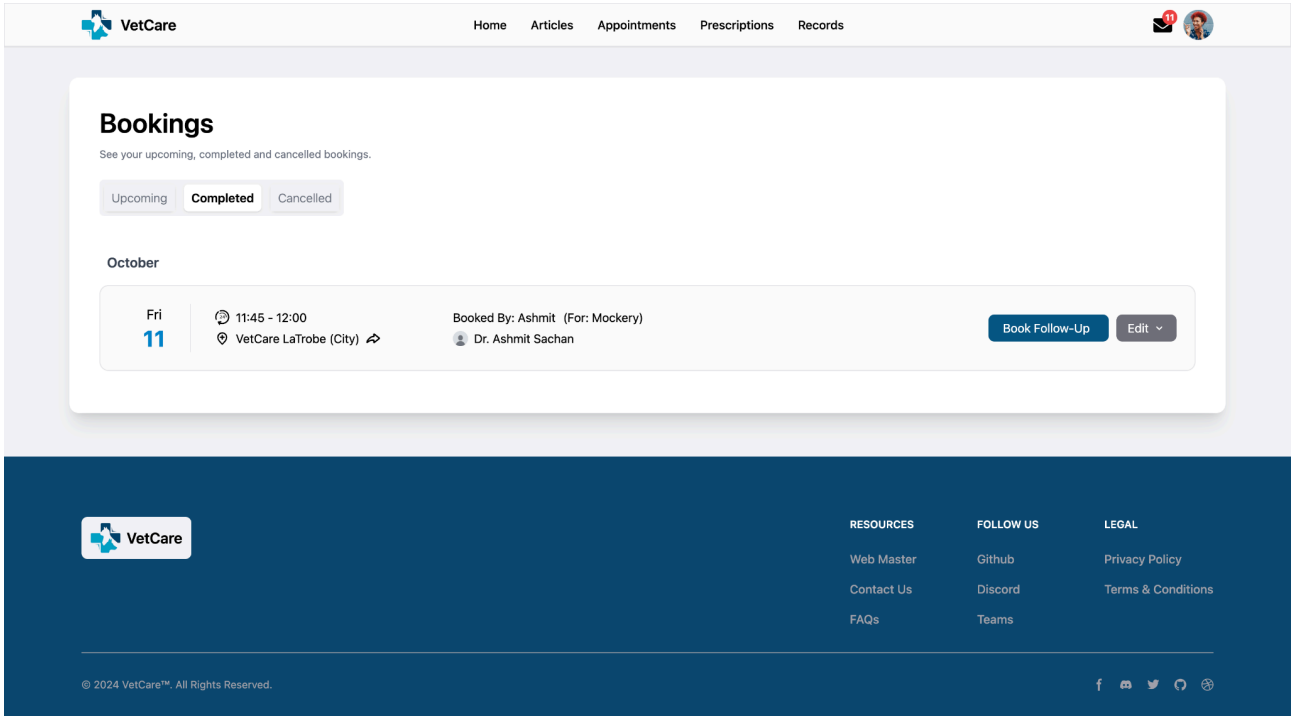
Upcoming appointments:



Canceled Appointments:



Completed appointments:



CI/CD Automated Build

8. Testing in Sprint 1

User Service Tests

Test 1:

Test Name: testSaveUser_WithEncryption

Purpose: The purpose of this test was to check that the passwords were correctly being encrypted when being stored into the database.

Approach: The test works by having a normal password being created and passed to saveUser, of which the test then checks that the password is no longer saved in its original form, meaning it has been successfully encrypted.

Expected Outcome: The test is successful if the outcome is encrypted, meaning it's different to the input password.

Test 2:

Test Name: testEmailExists

Purpose: This test is a check to see if a given email already exists within the database.

Approach: An email is saved into the database, the test then calls emailExists() to check whether the email does or does not exist within the database.

Expected Outcome: The test should return true when the email already exists in the database and false if the email is unique

Test 3:

Test Name: testValidateUserCredentials_Success

Purpose: This test is used to authenticate user information and whether or not the format is correct

Approach: The test works by creating, then saving a new user into the database, then returning the user object to see if the new user matches the expected results.

Expected Outcome: The test expects the credentials to be validated successfully when the user object is successfully returned.

Test 4:

Test Name: testValidateUserCredentials_IncorrectPassword

Purpose: This test checks that the validateUserCredentialswill fail when an incorrect password is used.

Approach: A user is created and saved with a known password. The test then attempts to validate this user with an incorrect password, which will then produce an error due to it being incorrect.

Expected Outcome: The test expects an Exception with the message "Invalid credentials" when the password does not match.

Test 5:

Test Name: testValidateUserCredentials_EmailNotFound

Purpose: This test is to validate that unique email addresses are not found within the database.

Approach: The test uses a unique email address that is tested within the database to prove that it is unique and not found

Expected Outcome: The test expects an Exception with the message "Email not found," which confirms that the email is in fact unique.

User Controller Tests

Test 6:

Test Name: signup_Success

Purpose: This test verifies that when a user signs up, that it successfully populates the database

Approach: A JSON payload containing valid user details is sent to the signup controller, that then checks that a 200 status is returned, which confirms a successful user signup attempt.

Expected Outcome: The test expects a success message 200 status to be returned, confirmed a successful user signup

Test 7:

Test Name: signup_EmailAlreadyExists

Purpose: This test is to check that the signup process successfully checks if an email is already registered.

Approach: The test pre-saves an email, which is then used to test against, if the status 400 is returned, the test is a success as it highlights that the email does in fact already exist in the database and thereby is not unique.

Expected Outcome: The test expects the response to include the message "Email already exists," as well as a status 400 response, confirming the email exists.

Test 8:

Test Name: login_Success

Purpose: This test validates that the loginUser endpoint in the UserController correctly authenticates a user with valid details.

Approach: A user is created, which is then used to test to see if it can successfully login, if status is 200, it confirms that the details were successfully signed in with.

Expected Outcome: The test expects successful authentication, with the correct user details being returned in the response.

Test 9:

Test Name: login_IncorrectPassword

Purpose: This test checks that the loginUser endpoint fails when an incorrect password is provided when attempting to login.

Approach: A user is saved with a known password, and then a using these details, a login is made with a different password. If a status 400 is returned, it indicates that the password does not match and therefore the user is not logged in.

Expected Outcome: The test expects a failure message and a 400 status, confirming that it was an incorrect password.

Test 10:

Test Name: login_EmailNotFound

Purpose: This test ensures that the loginUser endpoint correctly handles login attempts with an email that hasn’t been attached to any account, making it non-existent in the database.

Approach: The test sends a login request with a non-existent email and checks that the response indicates failure due to the email not being found.

Expected Outcome: The test expects a failure response when trying to login with the incorrect email, confirming that it wasn't found in the database.

Article Repository Tests

Setup: Two custom test articles with unique titles, descriptions, and authors are added to an already existing articles database.

Test 11:

Test Name: testSearchArticlesByKeyword_TitleMatch

Purpose: This test ensures that the search functionality for articles by keywords works correctly by matching the keyword with the article's title

Approach: Two separate searches are performed with the input "Curabitur" and "abitur".

Expected Outcome: The test expects both search results to return exactly one article with the title "Curabitur non justo".

Test 12:

Test Name: testSearchArticlesByKeyword_DescriptionMatch

Purpose: This test ensures that the search functionality for articles by keywords works correctly by matching the keyword with the article's description.

Approach: Two separate searches are performed with the input "dolor amet" and "doLoR a".

Expected Outcome: The test expects both search results to return exactly one article with the description "Dolor amet".

Test 13:

Test Name: testSearchArticlesByKeyword_AuthorMatch

Purpose: This test ensures that the search functionality for articles by keywords works correctly by matching the keyword with the article's author.

Approach: Two separate searches are performed with the input "Nullam" and "am volut".

Expected Outcome: The test expects both search results to return exactly one article with the author "Nullam Volutpat".

Test 14:

Test Name: testSearchArticlesByKeyword_NoMatch

Purpose: This test ensures that the search functionality correctly handles cases where no articles match the provided keyword.

Approach: A search is performed with an input “asdfjkl”.

Expected Outcome: The test expects that no articles are found and therefore the number of search results is zero.

Test 15:

Test Name: testSearchArticlesByKeyword_LargeDataset

Purpose: This test ensures that the search functionality remains accurate and efficient when dealing with large datasets.

Approach: The test generates 1000 random articles and saves them to the database. A timer is started before the search query is executed to measure how long the search takes.
A search is performed using the keyword "Lorem Ipsum". The timer is then stopped when the search query is completed.

Expected Outcome: The search query should complete within one second and return exactly one article.

Prescription Tests

Located in branch “broken-prescription”

Test 16:

Test Name: getCurrentPrescriptions_Success()

Purpose: This test ensures that the system can retrieve the current prescriptions for a specific pet.

Approach: The pet information is fetched from the database using petId, and then checks if the system returns the correct prescription details.

Expected Outcome: The test should return a list of current prescriptions for the test with the correct details, such as medication name and dosage details.

Test 17:

Test Name: getPrescriptionHistory_Success()

Purpose: This test ensures that the system can retrieve the prescription history for a specific pet.

Approach: The pet information is fetched from the database using petId, and then checks if the system returns the correct details of the selected pet's prescription history.

Expected Outcome: The test should return a list of past prescriptions for the test with the correct details, such as medication name, duration and dosage details.

Test 18:

Test Name: addPrescription_Success()

Purpose: This test ensures that new prescriptions can be added successfully to the system via the user's manual input.

Approach: Sends a request to the system with the details of the prescription, with those details being: pet ID, medication, dosage, frequency, duration, issue date and refill date.

Expected Outcome: The test should add these details successfully to the system and return a "Prescription added successfully" message.

Test 19:

Test Name: editPrescription_Success()

Purpose: This test ensures that existing prescriptions can be modified successfully in the system via the user's manual input.

Approach: Sends a request to the system with the details of the prescription, with those details being: pet ID, medication, dosage, frequency, duration, issue date and refill date.

Expected Outcome: The test should update these details successfully to the system and return a "Prescription updated successfully" message.

Test 20:

Test Name: deletePrescription_Success()

Purpose: This test ensures that existing prescriptions can be deleted successfully in the system via the user's request.

Approach: Sends a deletion request to the system containing the pet ID and the prescription information.

Expected Outcome: The test should delete the prescription from the system successfully and return a “Prescription deleted successfully” message.

Test 21:

Test Name: exportToPDF_TriggersPrint()

Purpose: This test ensures that the print button will trigger the browser’s native print window in order to turn the prescription information into a PDF.

Approach: When the button is pressed, the system will send an instruction to the browser to open its native print window.

Expected Outcome: The test should trigger the browser’s print function and return a “Print function triggered” message.

ServiceControllerTests

This test file checks the behavior of the Service Controller for fetching services using the Spring `MockMvc` framework.

Test 22: `getAllServices_Success`

- **Purpose**: This test ensures that the API endpoint for fetching all services is working as expected.
- **Approach**: It sends a POST request to `/api/service/all` and checks that the response is OK (HTTP status 200) and that the response contains a non-empty array of services.
- **Expected Outcome**: The test expects the API to return a status of 200 (OK) and a non-empty array of services, indicating that services were fetched successfully.

This test validates the system's ability to retrieve a list of all available services, ensuring that the backend service and the endpoint are functioning as intended.

Test 23: `getServiceById_Success`

- **Purpose**: This test checks if a specific service can be fetched by its ID.
- **Approach**: It sends a POST request to `/api/service/1` to fetch the service with ID 1 and expects a successful response.
- **Expected Outcome**: The test expects a 200 (OK) status, confirming that the service with ID 1 is successfully retrieved.

This test ensures that individual services can be queried by ID, which is important for viewing service details in various scenarios, such as a customer checking service offerings.

VeterinarianControllerTests

This test file verifies the Veterinarian Controller's endpoints, particularly fetching veterinarians based on different criteria.

Test 24: `getAllVeterinarians_Success`

- **Purpose**: This test ensures that the endpoint for retrieving all veterinarians is functioning correctly.
- **Approach**: It sends a POST request to `/api/veterinarian/all` and verifies that the response status is 200 and contains a non-empty list of veterinarians.
- **Expected Outcome**: The test expects an OK status and a non-empty list of veterinarians.

This test guarantees that the application can return a list of all veterinarians, which is essential for functionalities like displaying available veterinarians to users.

Test 25: `getVeterinariansByClinic_Success`

- **Purpose**: This test verifies that the system can retrieve veterinarians associated with a particular clinic ID.
- **Approach**: A POST request is sent to `/api/veterinarian/clinic/1` to retrieve veterinarians from clinic ID 1. The test expects a successful response.
- **Expected Outcome**: A status of 200 (OK) confirms that the veterinarians from the specified clinic are successfully fetched.

This test ensures that users can filter veterinarians based on clinic, which is important for users who are looking for clinic-specific veterinarians.

Test 26: `getVeterinariansByService_Success`

- **Purpose**: This test checks if veterinarians can be filtered by a specific service.
- **Approach**: The test sends a POST request to `/api/veterinarian/service/1` to fetch veterinarians who provide a specific service with ID 1.
- **Expected Outcome**: The test expects an OK status, confirming that veterinarians offering the specific service are successfully retrieved.

This test helps in identifying veterinarians based on the services they provide, which is crucial for enabling service-specific searches for customers.

VeterinarianAvailabilityControllerTests

This test file is focused on the availability of veterinarians and ensuring that the system correctly retrieves their availability data.

Test 27: `getAvailabilityByVeterinarianId_Success`

- **Purpose**: The purpose of this test is to verify that the system can retrieve the availability of a specific veterinarian based on their ID.
- **Approach**: A POST request is sent to `/api/veterinarian-availability/1` to fetch the availability of the veterinarian with ID 1. The response is expected to contain availability information.
- **Expected Outcome**: The test expects a 200 status and a non-empty availability array for the veterinarian.

This test ensures that the system provides users with the availability of veterinarians, which is critical for appointment scheduling and resource planning.

MedicalRecordsControllerTests

This test class validates the behavior of the Medical Records Controller by testing various endpoints related to retrieving and downloading medical records for pets.

Test 28: `getUserPets_Success`

- **Purpose**: To ensure the API can successfully fetch all pets associated with a specific user.
- **Approach**: It sends a `GET` request to the `/api/medical-records/user-pets` endpoint with a `userId` parameter and checks that the response contains a non-empty list of pets.
- **Expected Outcome**: The test expects an HTTP status of 200 (OK) and that the response JSON contains a non-empty array, confirming pets are associated with the user.

This test validates that the system can correctly retrieve the pets for a user, which is essential for displaying the user's pets in a dashboard or similar feature.

Test 29: `downloadMedicalRecords_PDF_Success`

- **Purpose**: To verify that medical records for a pet can be downloaded in PDF format.
- **Approach**: It sends a `GET` request to `/api/medical-records/download` with the `selectedPetId`, `format` (set to "pdf"), and specific medical record sections. The test checks that the response is a PDF file.
- **Expected Outcome**: The test expects a 200 (OK) status and verifies that the response header indicates a PDF content type (`application/pdf`).

This test ensures the system can generate and serve medical records as a PDF file, an important feature for users who need downloadable medical documents.

Test 30: `downloadMedicalRecords_XML_Success`

- **Purpose**: To ensure medical records can be downloaded in XML format.
- **Approach**: It sends a `GET` request to `/api/medical-records/download` with parameters for the `selectedPetId`, `format` (set to "xml"), and specific sections of the medical records. The test checks that the response is an XML file.
- **Expected Outcome**: The test expects a 200 (OK) status and verifies that the response header indicates an XML content type (`application/xml`).

This test ensures that medical records can be generated in XML format, allowing users or systems to access structured data for further processing or storage.

Test 31: `getUserPets_NoPets_SeedsData`

- **Purpose**: To verify that the system seeds pet data for a user if no pets exist.
- **Approach**: It sends a `GET` request to `/api/medical-records/user-pets` with a `userId` that does not have any pets. The test checks if the system returns a non-empty list, implying that pets have been seeded.
- **Expected Outcome**: The test expects a 200 (OK) status and a non-empty JSON array, showing that pets have been seeded when none existed initially.

This test ensures that the system can handle cases where users have no pets and automatically seed data, providing a complete user experience even for new users.

FileGenerationServiceTests

This test class focuses on verifying the functionality of the `FileGenerationService`, particularly for generating XML files with pet medical records.

Test 31: `testGenerateXML`

- **Purpose**: To test that XML can be successfully generated with all relevant sections of a pet's medical records.
- **Approach**: The test calls the `generateXML` method, passing a pet object and lists of medical history, physical exams, vaccinations, treatment plans, and weight records. It then checks if the generated XML contains the expected sections.
- **Expected Outcome**: The test expects the generated XML stream to be non-null, contain data, and include tags for each section (e.g., `<MedicalHistory>`, `<PhysicalExams>`).

This test verifies that the system can properly serialize a pet's medical records into an XML format, ensuring that all required sections are included.

Test 32: `testGenerateXML_EmptySections`

- **Purpose**: To test XML generation when no sections are requested.
- **Approach**: The test calls `generateXML` with an empty list of sections. It then verifies that the generated XML does not contain any medical record sections.
- **Expected Outcome**: The test expects the XML to contain the pet information but exclude sections such as `<MedicalHistory>`, `<Vaccinations>`, and others.

This test ensures that the XML generation behaves correctly when specific sections are omitted, providing flexibility in the types of data included in the output.

9. Summary of Updated User Stories for Sprint 2 (i.e Milestone 3)

#480 [User Guide] As a new user, I want an interactive onboarding guide to appear when I first access the platform so that I can learn how to use the platform effectively.

Acceptance Criteria:

- The onboarding guide should automatically be shown for new users.
- There should be an option to dismiss or skip the guide.
- There should be next and previous buttons to navigate through the steps.
- Each step should be clearly explained and visually highlighted on the platform.

Definition of done:

- The onboarding guide automatically appears for new users.
- Users are provided with buttons to skip the guide and navigate the steps.
- Instructions and steps are clear and highlighted.

#482 [User Guide] As an existing user, I want to be able to revisit the onboarding guide so that I can review the platform's features.

Acceptance Criteria:

- There should be a link in the settings for the user to restart the onboarding guide.
- The onboarding guide should be shown once the user presses the link.

Definition of Done:

- After the user presses the link in the settings, they are redirected to the home page and the guide is restarted successfully.

#494 [FAQ section] As a user, I want to access the FAQ section on the website, so that I can easily find answers to common questions and resolve my issues without needing to contact customer support.

Acceptance Criteria:

- The FAQ section must be accessible from the site's footer.
- Users should be able to filter the questions by categories.
- Each question must expand to reveal answers and collapse to hide answers.
- There should be a link for users to contact support if their question is not listed in the FAQ.

Definition of Done:

- The users are able to access the faq page from the link in the footer.
- Users are able to filter questions by categories and the questions for that category are correctly shown.
- The UI expands and collapses correctly during user interaction.
- The link to the contact page is clearly displayed.

#530 [Contact Support] As a user, I want to have a contact support page where I can describe the issue I am facing so that I can reach out to the support team with my questions or problems.

Acceptance Criteria:

- There should be a form for users to fill in their first name, last name, email and the issue they are facing.
- Logged in user only have to fill in the issue they are facing.
- The email and phone of the support team should be displayed on the page.
- The system should notify the user upon a successful submission.
- The details should be sent to the support team email.

Definition of Done:

- Users are able to fill in their details in the form.
- The details for logged in users are automatically filled in.
- The email and phone number of the support team is clearly displayed on the contact page.
- Users are able to see the notification when submitting the form.

#535 [Prescription] As a vet, I want to manage prescriptions and view the prescription history for all pets under my care so that I can provide accurate and up-to-date medication plans.

#505 [CI/CD Pipeline with AWS ECR] As a DevOps engineer, implement a pipeline for automated **build, test, and deployment** using **Docker** and **AWS ECR**

Key Acceptance Criteria:

- Pipeline triggers on **push/pull requests**.
- Builds app with **Maven** (tests skipped), creates and pushes Docker images to **AWS ECR**.
- Integrates **MySQL** and health checks via **docker-compose**.

Definition of Done:

- Pipeline runs successfully, Docker image is pushed to **AWS ECR**, and logs are verified.

#510 [Veterinarian Signup and Login Functionality] As a veterinarian, sign up and log in through a web interface to manage profile and services.

Key Acceptance Criteria:

- Veterinarians can sign up and log in with **unique email**, optional **clinic association**, and **password encryption**.
- Independent veterinarians can skip clinic association.
- Proper error handling for **duplicate emails** and **invalid credentials**.

Definition of Done:

- Veterinarians can sign up and log in successfully.
- **Email uniqueness** and **input validation** are enforced.
- **Password** is securely hashed and stored.
- UI is **responsive** and integrated with backend.

#515 [Veterinarian Dashboard] After signing up or logging in, veterinarians can access a personal dashboard to manage appointments, prescriptions, and patient records.

Key Acceptance Criteria:

- Veterinarians can sign up with **email, name, contact**, and optionally a **clinic** (defaults to "Independent").
- Successful login redirects to the **veterinarian dashboard**.
- The dashboard provides options for **appointments, prescriptions**, and **patient records**.
- **Error handling** for duplicate emails, invalid credentials, and missing clinics.

Definition of Done:

- Fully functional and **validated** signup and login forms.
- Veterinarians are redirected to their **dashboard** after login.
- **Dashboard UI** includes navigation for managing appointments, prescriptions, and records.
- **API** endpoints are tested for valid/invalid data, with edge cases covered.
- **Unit and integration tests** are implemented for signup, login, and dashboard functionality.

#564 [Veterinarian Uploads Medical Records After Each Appointment] After signing up or logging in, veterinarians can access a personal dashboard to manage appointments, prescriptions, and patient records.

Key Acceptance Criteria:

- Veterinarians can sign up with **email, name, contact**, and optionally a **clinic** (defaults to "Independent").
- Successful login redirects to the **veterinarian dashboard**.
- The dashboard provides options for **appointments, prescriptions**, and **patient records**.
- **Error handling** for duplicate emails, invalid credentials, and missing clinics.

Definition of Done:

- Fully functional and **validated** signup and login forms.

- Veterinarians are redirected to their **dashboard** after login.
- **Dashboard UI** includes navigation for managing appointments, prescriptions, and records.
- **API** endpoints are tested for valid/invalid data, with edge cases covered.
- **Unit and integration tests** are implemented for signup, login, and dashboard functionality.

#499 [Design and Implement the Home Page UI] Users need a visually appealing, user-friendly home page to easily navigate and access key features like Medical Records, Prescriptions, Appointments, and Educational Resources.

Key Acceptance Criteria:

- **Hero section** includes a prominent background image with a headline introducing the service.
- Clickable **service cards** (Medical Records, Prescriptions, Appointments, Educational Resources) with icons, titles, and descriptions redirect users to relevant sections.
- Design is fully **responsive** for desktop and mobile.
- **Tailwind CSS** is used for layout and styling.
- Hero section includes a **faded overlay** for readability.
- Images are **optimized** for performance.
- **Hover effects** on service cards for better interactivity.

Definition of Done:

- Hero section and service cards are fully implemented and styled.
- Layout is **responsive** and tested across small, medium, and large screen sizes.
- All **images are optimized** and the page performance is verified.
- Hover effects and interactive elements are functional.
- **Tailwind CSS** is used throughout for consistent design.

10. Summary of Updated Tests and Code for Sprint 2 (i.e Milestone 3)

ArticleController Tests

Test 1: testAddArticle_success()

- **Purpose:** To test the integration of the article service from the web layer when an article is added
- **Approach:** The test calls POST handler using a MockMvc to add an article to the database
- **Expected Outcome:** An article is successfully added to the database

Test 2: testRemoveArticle_success()

- **Purpose:** To test the integration of the article service from the web layer when an article is removed.
- **Approach:** The test saves an article directly to the repository, then calls the delete handler using MockMvc to remove the article by its ID.
- **Expected Outcome:** The specified article is successfully removed from the database, and a subsequent search returns no result for that article.

Test 3: testAddBookmark_success()

- **Purpose:** To test the integration of the bookmark service from the web layer when a bookmark is added.
- **Approach:** The test calls a POST handler using MockMvc to add a bookmark for a given user and article. It then checks if the bookmark and the associated article have been added to their respective repositories.
- **Expected Outcome:** The bookmark is successfully added to the database, and the article associated with the bookmark exists in the repository.

Test 4: testAddBookmark_duplicate()

- **Purpose:** To test the integration of the bookmark service from the web layer when a duplicate bookmark is added.
- **Approach:** The test adds a bookmark for two different users with the same article using MockMvc and verifies that both bookmarks are linked to the same article in the repository.
- **Expected Outcome:** Each user can add a bookmark for the same article, resulting in two distinct bookmarks, but only one instance of the article exists in the database.

Test 5: testRemoveBookmark_success()

- **Purpose:** To test the integration of the bookmark service from the web layer when a bookmark is removed.
- **Approach:** The test first saves a bookmark directly into the repository for a test user and a saved article. It then calls a POST handler using MockMvc to remove the bookmark using its ID and verifies that the bookmark no longer exists in the repository.
- **Expected Outcome:** The specified bookmark is successfully removed from the database, and a subsequent search confirms its absence.

Test 6: testSearchArticle_success()

- **Purpose:** To test the integration of the article service when searching for an article based on a keyword.
- **Approach:** The test saves an article into the repository and then uses MockMvc to perform a GET request with a search keyword. It verifies that the response contains the link to the article.
- **Expected Outcome:** The search operation successfully retrieves the article containing the specified keyword, and the response body includes the article's link.

Test 7: testSearchArticle_notFound()

- **Purpose:** To test the integration of the article service when a search yields no results.
- **Approach:** The test ensures that no articles exist in the repository and then performs a GET request using MockMvc with a search keyword. It verifies that the response indicates no results found.
- **Expected Outcome:** The search operation returns a message indicating that no articles were found.

Test 8: testDownloadArticle_success()

- **Purpose:** To test the functionality of downloading an article as a ZIP file using the web layer.
- **Approach:** The test uses MockMvc to perform a GET request to the downloadArticle endpoint, passing a URL as a parameter. It verifies that the response is returned with a content type indicating a ZIP file.
- **Expected Outcome:** The response contains a ZIP file, demonstrating that the article download function works correctly.

ArticleService Tests

Test 9: testGetRssArticles_success()
- **Purpose:** To test the functionality of fetching articles from an RSS feed using the article service.
- **Approach:** The test calls the getRssArticles method with a valid RSS feed URL and checks if articles are returned.
- **Expected Outcome:** A list of articles is returned, with the size greater than zero.

Test 10: testGetRssArticles_malformedUrl()
- **Purpose:** To verify the behavior when the RSS feed URL is malformed.
- **Approach:** The test passes a clearly malformed URL to getRssArticles and checks the returned list.
- **Expected Outcome:** The service should return an empty list, as the URL cannot be parsed.

Test 11: testGetRssArticles_iOError()
- **Purpose:** To test the service's response when the RSS feed URL is unreachable (e.g., network error).
- **Approach:** The test uses a URL that simulates being down or unreachable and checks the result.
- **Expected Outcome:** The service returns an empty list when the URL is unreachable.

Test 12: testGetRssArticles_rssFeedError()
- **Purpose:** To verify the service's behavior when a valid URL points to a non-RSS feed.
- **Approach:** A valid but non-RSS URL is passed to getRssArticles to check the result.
- **Expected Outcome:** The service returns an empty list, indicating that the content is not a valid RSS feed.

Test 13: testGetArticles_success()
- **Purpose:** To test fetching a page of articles from the database.
- **Approach:** The test calls getArticles with a page number and checks the result.
- **Expected Outcome:** The method returns a non-empty page of articles from the database.

Test 14: testGetSearchResult_success()
- **Purpose:** To verify the functionality of searching for articles by keyword.
- **Approach:** The test saves articles and then searches using a keyword to find matching articles.
- **Expected Outcome:** The method returns a page with the expected number of articles matching the search keyword.

Test 15: testAddToZip_success()
- **Purpose:** To test adding file content to a ZIP archive.
- **Approach:** The test calls addToZip with a file path and content, then checks the resulting ZIP file for the expected entry.
- **Expected Outcome:** The ZIP file contains an entry with the specified path, and the content matches the original bytes.

Test 16: testDownloadFile_success()

- **Purpose:** To test downloading a file from a URL.
- **Approach:** The test calls downloadFile with a valid file URL and verifies that content is returned.
- **Expected Outcome:** The downloaded file content is not empty, indicating successful retrieval.

Test 17: testDownloadFile_malformedUrl()
- **Purpose:** To test the response when attempting to download from a malformed URL.
- **Approach:** The test calls downloadFile with an invalid URL and checks the result.
- **Expected Outcome:** The method returns an empty byte array, as the URL is invalid.

Test 18: testDownloadFile_iOError()
- **Purpose:** To test downloading from a URL that simulates an unreachable server.
- **Approach:** The test calls downloadFile with a URL that cannot be reached and checks the result.
- **Expected Outcome:** The method returns an empty byte array due to the inability to reach the server.

Test 19: testWriteZipToStream_success()
- **Purpose:** To test the process of writing data to a ZIP output stream from a URL.
- **Approach:** The test calls writeZipToStream with a URL and verifies that the resulting ZIP contains the expected entry.
- **Expected Outcome:** The ZIP stream contains an entry named article.html, indicating that the data from the URL was successfully written to the ZIP file.

BookmarkService Tests

Test 20: testFindByUser_success()
- **Purpose:** To test retrieving bookmarks for a user who has saved at least one bookmark.
- **Approach:** The test uses findByUser with testUser1 (who has a bookmark) and verifies the number of bookmarks returned.
- **Expected Outcome:** One bookmark is returned in the result.

Test 21: testFindByUser_notFound()
- **Purpose:** To test retrieving bookmarks for a user who has not saved any bookmarks.
- **Approach:** The test uses findByUser with testUser2 (who has no bookmarks) and verifies the number of bookmarks returned.
- **Expected Outcome:** Zero bookmarks are returned.

Test 22: testFindByUser_negativePage()
- **Purpose:** To test the behavior when a negative page number is requested.
- **Approach:** The test calls findByUser with a negative page number for testUser1 and checks the result.
- **Expected Outcome:** Zero bookmarks are returned, indicating that negative page numbers are not valid.

Test 23: testAddBookmark_success()
- **Purpose:** To test adding a bookmark for an article that is not yet in the database.
- **Approach:** The test calls addBookmark for testUser1 with article2, which is not saved in the database. It checks that both the article and the bookmark are stored.

- **Expected Outcome:** Both the new article and the bookmark are successfully added to the database.

Test 24: testAddBookmark_existingArticle()
- **Purpose:** To test adding a bookmark for an article that already exists in the database.
- **Approach:** The test calls addBookmark for testUser2 with article1 (which is already in the database). It verifies that only the bookmark is added, not a duplicate of the article.
- **Expected Outcome:** The article count remains unchanged, while a new bookmark is added.

Test 25: testRemoveBookmark_success()
- **Purpose:** To test removing a bookmark for an article that exists for a user.
- **Approach:** The test calls removeBookmark for testUser1 with article1 and checks if the method returns true, indicating successful removal.
- **Expected Outcome:** The method returns true, confirming that the bookmark has been successfully removed.

Test 26: testRemoveBookmark_notPresent()
- **Purpose:** To test removing a bookmark that does not exist for a user.
- **Approach:** The test calls removeBookmark for testUser2 with article1 (for which there is no saved bookmark) and checks if the method returns false.
- **Expected Outcome:** The method returns false, indicating that there was no bookmark to remove for the specified user and article.

FAQ Controller Tests

Test 27: testGetFaqByCategorySuccess
- **Purpose:** The purpose of this test is to verify that the system correctly retrieves a list of FAQs from the database based on a given category.
- **Approach:** The test sets up a mock list of FAQs containing two entries, each belonging to the "general" category. It then simulates a request to retrieve FAQs by the general category. The test mocks the behavior of the repository to return the pre-defined list of FAQs when queried for the "general" category.
- **Expected Outcome:**The system successfully returns the list of FAQs with the correct size , the correct category, and the list is non-empty.

Test 28: testGetFaqByCategoryEmpty
- **Purpose:** The purpose of this test is to verify the system's behavior when no FAQs are available for a given category.
- **Approach:** The test sets up an empty list of FAQs to simulate a scenario where no FAQs exist for the "general" category. It mocks the repository behavior to return this empty list when queried for the "general" category.
- **Expected Outcome:**The system successfully returns an empty list of FAQs, with the isEmpty attribute set to true.

Test 29: testGetFaqByCategoryFail
- **Purpose:** The purpose of this test is to verify that the system handles exceptions properly when there is an error while retrieving FAQs from the database, such as a database error.

- **Approach:** The test sets up a scenario where calling faqRepository.findByCategory for the "general" category throws a RuntimeException with a message indicating a "Database error." It then performs a request to the /faq/general endpoint and checks if the system correctly handles the exception.
- **Expected Outcome:**The system successfully handles the exception by adding an error attribute to the model. The error is displayed on the "index" view, and the page is rendered with the correct response.

HomeTemplateIntegrationTest - Preeti

Test: testHomePageRendersSuccessfully()

- Purpose: To verify that the home page (/home) renders successfully and contains all the essential elements for users to manage pet healthcare.
- Approach:
    - Perform an HTTP GET request to the /home endpoint.
    - Check that the response returns a 200 OK status and the page contains key content related to veterinary services.
- Verification Steps:
    - The response status is 200 OK, indicating that the page loads successfully.
    - The hero section contains the text: "Working with you for the health and happiness of your pet."
    - The presence of the following cards is confirmed:
        - "Medical Records" card
        - "Prescriptions" card
        - "Book & Manage Appointments" card
        - "Educational Resources" card
- Expected Outcome:
    - The home page loads successfully with status code 200 OK.
    - All the specified text and cards are present, confirming the page contains the necessary elements to help users manage their pets' healthcare.

VeterinarianControllerTests.java - Preeti

Test 1: signup_Success()

- Purpose: To test the successful signup of a veterinarian, ensuring correct input validation and interaction with the service layer.
- Approach: The test calls the signup endpoint with valid veterinarian details. It checks that the signup is successful, and the veterinarian is added to the system.
- Expected Outcome: The veterinarian is successfully signed up, and the response contains the message "Veterinarian signed up successfully!"

Test 2: signup_EmailAlreadyExists()

- Purpose: To test the scenario where the signup fails due to an email that already exists in the system.
- Approach: The test calls the signup endpoint with a duplicate email. It verifies that proper validation occurs and an error message is returned.
- Expected Outcome: The signup fails, and the response contains the message "Email already exists," with a status of Bad Request (400).

Test 3: updatePassword_Success()

- Purpose: To test successful password updates for a veterinarian.
- Approach: The test calls the updatePassword endpoint with valid veterinarian ID and new password. It verifies that the password is updated successfully in the system.
- Expected Outcome: The password is updated, and the response contains the message "Password updated successfully!"

Test 4: getAppointmentsByVeterinarian_Success()

- Purpose: To test retrieving appointments for a specific veterinarian, ensuring the request completes successfully.
- Approach: The test calls the getAppointmentsByVeterinarian endpoint with a valid veterinarian ID and checks that the appointments are returned successfully.
- Expected Outcome: The veterinarian's appointments are retrieved, and the response status is 200 OK.

VeterinarianServiceTests.java - Preeti

Test 1: validateVeterinarianCredentials_Success()

- **Purpose:** To verify that a veterinarian's credentials are validated correctly, ensuring the clinic and password match.
- **Approach:** The test calls validateVeterinarianCredentials with valid email, password, and clinic name. It checks that the credentials are verified and the veterinarian is returned.
- **Expected Outcome:** The credentials are validated successfully, and the veterinarian object is returned.

Test 2: validateVeterinarianCredentials_InvalidPassword()

- **Purpose:** To test the scenario where the password provided does not match the stored password.
- **Approach:** The test calls validateVeterinarianCredentials with the correct email and clinic, but an incorrect password. It verifies that an appropriate exception is thrown for invalid credentials.
- **Expected Outcome:** An exception is thrown with the message "Invalid credentials: Password mismatch."

Test 3: validateVeterinarianCredentials_VeterinarianNotFound()

- **Purpose:** To test the behavior when a veterinarian is not found for the provided email and clinic combination.
- **Approach:** The test calls validateVeterinarianCredentials with an email that does not correspond to any veterinarian in the clinic and verifies that an exception is thrown.

- **Expected Outcome:** An exception is thrown with the message "Invalid credentials: Veterinarian not found."

Test 4: validateVeterinarianCredentials_ClinicNotFound()

- **Purpose:** To test the behavior when the provided clinic is not found.
- **Approach:** The test calls validateVeterinarianCredentials with a non-existent clinic name and checks that an exception is thrown due to invalid clinic information.
- **Expected Outcome:** An exception is thrown with the message "Invalid credentials: Clinic not found."

Test 5: saveVeterinarian_Success()

- **Purpose:** To test the saving of a veterinarian, ensuring that the password is encrypted correctly.
- **Approach:** The test calls saveVeterinarian with a veterinarian's details and checks that the password is encrypted and the veterinarian is saved successfully in the database.
- **Expected Outcome:** The veterinarian is saved with an encrypted password, and the save operation is successful.

Test 6: updateVeterinarian_VeterinarianNotFound()

- **Purpose:** To test the behavior when attempting to update a veterinarian that does not exist in the database.
- **Approach:** The test calls updateVeterinarian for a veterinarian ID that does not exist and verifies that an appropriate exception is thrown.
- **Expected Outcome:** An exception is thrown with the message "Veterinarian not found."

Test 7: updatePassword_VeterinarianNotFound()

- **Purpose:** To ensure proper error handling when trying to update a password for a veterinarian that does not exist.
- **Approach:** The test calls updatePassword with a nonexistent veterinarian ID and checks that an exception is thrown.
- **Expected Outcome:** An exception is thrown with the message "Veterinarian not found."

MedicalRecordsControllerTests - Preeti

Test 1: getUserPets_Success()

- Purpose: To verify that fetching all pets for a specific user who has pets returns the correct number of pets.
- Approach: This test retrieves pets for testUser1 (who has saved pets in the system) by calling getUserPets and checking the number of pets returned.
- Expected Outcome: The test returns the correct number of pets (greater than zero) for the user.

Test 2: getUserPets_NoPets_SeedsData()

- Purpose: To test the system behavior when a user has no pets, triggering the system to seed data automatically.
- Approach: The test calls getUserPets for testUser2 (who has no pets). It checks that the system seeds data and returns at least one pet.
- Expected Outcome: The system successfully seeds data and returns one or more pets for the user.

Test 3: getMedicalRecords_Success()

- Purpose: To test retrieving medical records for a specific pet that exists in the system.
- Approach: The test fetches medical records for testPet by calling getMedicalRecords and checks that all expected medical records (medical history, vaccinations, weight records, etc.) are returned.
- Expected Outcome: The correct medical records are returned, including medical history, physical exams, vaccinations, and more, for the selected pet.

Test 4: downloadMedicalRecords_PDF_Success()

- Purpose: To test downloading medical records for a pet as a PDF.
- Approach: The test calls downloadMedicalRecords with the format set to "pdf" and checks that the response is a valid PDF file with the expected content.
- Expected Outcome: A valid PDF file containing the pet's medical records is returned, with a status of 200 OK.

Test 5: downloadMedicalRecords_XML_Success()

- Purpose: To test downloading medical records for a pet as an XML file.
- Approach: The test calls downloadMedicalRecords with the format set to "xml" and checks that the response is a valid XML file with the expected content.
- Expected Outcome: A valid XML file containing the pet's medical records is returned, with a status of 200 OK.

Test 6: uploadWeightRecord_Success()

- Purpose: To test uploading a weight record for a pet's appointment.
- Approach: The test calls uploadWeightRecord for a valid appointment and veterinarian and verifies that the weight record is successfully saved.
- Expected Outcome: The weight record is uploaded successfully, and a confirmation message is returned.

Test 7: uploadVaccinationRecord_Success()

- Purpose: To test uploading a vaccination record for a pet's appointment.
- Approach: The test calls uploadVaccinationRecord with details of the vaccination (e.g., vaccine name, date administered) and checks that the record is successfully saved.
- Expected Outcome: The vaccination record is uploaded successfully, and a confirmation message is returned.

Test 8: uploadTreatmentPlan_Success()

- Purpose: To test uploading a treatment plan for a pet's appointment.

- Approach: The test calls uploadTreatmentPlan with the treatment details and verifies that the plan is successfully stored.
- Expected Outcome: The treatment plan is uploaded successfully, and a confirmation message is returned.

Test 9: uploadMedicalHistory_Success()

- Purpose: To test uploading a medical history record for a pet's appointment.
- Approach: The test calls uploadMedicalHistory with details such as treatment and notes and checks that the medical history record is saved successfully.
- Expected Outcome: The medical history record is uploaded successfully, and a confirmation message is returned.

Test 10: uploadPhysicalExam_Success()

- Purpose: To test uploading a physical exam record for a pet's appointment.
- Approach: The test calls uploadPhysicalExam with the exam details (e.g., exam date, notes) and checks that the record is saved successfully.
- Expected Outcome: The physical exam record is uploaded successfully, and a confirmation message is returned.

MedicalHistoryRepositoryTests.java - Preeti

Test 1: testFindByPetIdOrderByEventDateDesc()

- Purpose: To verify that the medical history records for a pet are retrieved and ordered correctly by event date in descending order.
- Approach:
    - Medical history records are created for testPet with different event dates.
    - The test calls findByPetIdOrderByEventDateDesc for testPet and verifies the number of records and their order.
- Expected Outcome:
    - Three records are returned.
    - The records are ordered by event date, with the most recent first.

Test 2: testFindByPetIdOrderByEventDateDesc_NoRecords()

- Purpose: To test the behavior when no medical history records exist for a pet.
- Approach: The test calls findByPetIdOrderByEventDateDesc with a non-existent pet ID (999L) and checks the result.
- Expected Outcome: An empty list is returned, indicating that no medical records are found for the pet.

Test 3: testFindByPetIdOrderByEventDateDesc_SameDateRecords()

- Purpose: To verify the behavior when multiple medical history records share the same event date.
- Approach:

    - Two medical history records are created for testPet with the same event date but different treatments.
    - The test calls findByPetIdOrderByEventDateDesc and checks the order of the records.
- Expected Outcome:
    - Five records are returned in total (including the previously created ones).
    - The records with the same event date are returned in the correct order, and the treatments are as expected.

Test 4: testFindByPetIdOrderByEventDateDesc_NullPetId()

- Purpose: To test the behavior when a null pet ID is provided.
- Approach:
    - The test calls findByPetIdOrderByEventDateDesc with a null pet ID and checks the result.
- Expected Outcome:
    - An empty list is returned, indicating no records are found when the pet ID is null.

PhysicalExamRepositoryTests.java - Preeti

Test 1: `testFindByPetId()`

- Purpose: To verify that the correct physical exam records are retrieved for a specific pet by its ID.
- Approach: The test calls the `findByPetId` method for `testPet` (who has multiple physical exam records). It checks the number of records returned and verifies that the exam dates and notes match the expected values.
- Expected Outcome: Three physical exam records are returned, and their dates and notes are correct, as follows:
    - Exam 1: Date = 2023-01-01, Notes = "Healthy"
    - Exam 2: Date = 2023-02-01, Notes = "Minor issues"
    - Exam 3: Date = 2023-03-01, Notes = "Requires follow-up"

Test 2: `testFindByPetId_NoRecords()`

- Purpose: To test the behavior when attempting to retrieve physical exam records for a pet that doesn't exist.
- Approach: The test calls the `findByPetId` method with a non-existent pet ID (`999L`). It checks that the result is an empty list.
- Expected Outcome: No physical exam records are returned, and the list is empty.

TreatmentPlanRepositoryTests.java - Preeti

Test 1: `testFindByPetId()`

- Purpose: To verify that the correct treatment plan records are retrieved for a specific pet by its ID.
- Approach: The test calls the `findByPetId` method for `testPet`, which has several treatment plan records. It checks the number of records returned and verifies that the plan dates, descriptions, practitioners, and notes match the expected values.
- Expected Outcome: Three treatment plan records are returned, and their details are as follows:
    - Plan 1: Date = 2023-01-01, Description = "Plan for arthritis", Practitioner = "John Doe", Notes = "Follow-up in 1 month"
    - Plan 2: Date = 2023-02-01, Description = "Plan for dental care", Practitioner = "John Doe", Notes = "Routine cleaning"
    - Plan 3: Date = 2023-03-01, Description = "Plan for injury recovery", Practitioner = "John Doe", Notes = "Rest and medication"

Test 2: `testFindByPetId_NoRecords()`

- Purpose: To test the behavior when attempting to retrieve treatment plan records for a pet that doesn't exist.
- Approach: The test calls the `findByPetId` method with a non-existent pet ID (`999L`). It checks that the result is an empty list.
- Expected Outcome: No treatment plan records are returned, and the list is empty.

VaccinationRepositoryTests.java - Preeti

Test 1: `testFindByPetId()`

- Purpose: To verify that the correct vaccination records are retrieved for a specific pet by its ID.
- Approach: The test calls the `findByPetId` method for `testPet`, which has several vaccination records. It checks the number of records returned and verifies that the vaccine names, administrators, vaccination dates, and next due dates match the expected values.
- Expected Outcome: Three vaccination records are returned, and their details are as follows:
    - Vaccination 1: Vaccine = "Rabies", Administered By = "John Doe", Date = 2023-01-01, Next Due Date = 2024-01-01
    - Vaccination 2: Vaccine = "Distemper", Administered By = "John Doe", Date = 2023-02-01, Next Due Date = 2024-02-01
    - Vaccination 3: Vaccine = "Parvovirus", Administered By = "John Doe", Date = 2023-03-01, Next Due Date = 2024-03-01

Test 2: `testFindByPetId_NoRecords()`

- Purpose: To test the behavior when attempting to retrieve vaccination records for a pet that doesn't exist.
- Approach: The test calls the `findByPetId` method with a non-existent pet ID (`999L`). It checks that the result is an empty list.
- Expected Outcome: No vaccination records are returned, and the list is empty.

WeightRecordRepositoryTests.java - Preeti

Test 1: testFindByPetId()

- Purpose: To verify that the correct weight records are retrieved for a specific pet by its ID.
- Approach: The test calls the findByPetId method for testPet, which has several weight records. It checks the number of records returned and verifies that the weight values and record dates match the expected values.
- Expected Outcome: Three weight records are returned, and their details are as follows:
    - Record 1: Weight = 12.5, Date = 2023-01-01
    - Record 2: Weight = 13.0, Date = 2023-02-01
    - Record 3: Weight = 13.5, Date = 2023-03-01

Test 2: testFindByPetId_NoRecords()

- Purpose: To test the behavior when attempting to retrieve weight records for a pet that doesn't exist.
- Approach: The test calls the findByPetId method with a non-existent pet ID (999L). It checks that the result is an empty list.
- Expected Outcome: No weight records are returned, and the list is empty.

FileGenerationServiceTests.java - Preeti

Test 1: testGeneratePDF()

- **Purpose:** To test generating a PDF that includes all sections (medical history, physical exams, vaccinations, treatment plans, weight records).
- **Approach:** The test calls the generatePDF method with a list of sections and verifies that the generated PDF is not empty by checking the size of the content.
- **Expected Outcome:** A PDF is generated successfully, and the byte array representing the PDF content is not empty.

Test 2: testGeneratePDF_EmptyData()

- **Purpose:** To test generating a PDF when no data is available for any of the sections.
- **Approach:** The test calls the generatePDF method with empty lists for all sections and verifies that a PDF is still generated, even without data.
- **Expected Outcome:** A non-empty PDF is generated, confirming that the system can handle the case where no data exists for the pet.

Test 3: testGeneratePDF_WeightRecordsOnly()

- **Purpose:** To test generating a PDF that includes only the weight records section.
- **Approach:** The test calls the generatePDF method with only the "weightRecords" section specified. It verifies that the generated PDF contains data and is not empty.
- **Expected Outcome:** A PDF is generated containing only the weight records, and the byte array representing the PDF content is not empty.

MedicalHistoryServiceTests.java - Preeti

Test 1: `testGetMedicalHistoryByPetId()`

- Purpose: To verify that the correct medical history records are fetched for a pet by its ID.
- Approach: The test calls the `getMedicalHistoryByPetId` method with the ID of `testPet`, which has two medical history records. It checks the number of records returned and verifies that the treatments match the expected values.
- Expected Outcome: Two medical history records are returned, and their treatments are as follows:
    - Record 1: Treatment = "Checkup"
    - Record 2: Treatment = "Vaccination"

Test 2: `testGetMedicalHistoryByPetId_Empty()`

- Purpose: To test the behavior when attempting to retrieve medical history records for a pet that has no records.
- Approach: The test calls the `getMedicalHistoryByPetId` method with the ID of a newly created pet with no medical history. It checks that the result is an empty list.
- Expected Outcome: No medical history records are returned, and the list is empty.

Test 3: `testSaveMedicalHistory_NewRecord()`

- Purpose: To verify that a new medical history record can be saved and retrieved correctly from the database.
- Approach: The test creates a new medical history record, saves it using the `save` method, and then fetches the record from the database to verify the data.
- Expected Outcome: The new medical history record is saved successfully, and the fetched record has the correct treatment and practitioner:
    - Treatment = "Surgery"
    - Practitioner = "Dr. Brown"

Test 4: `testSaveMedicalHistory_ExistingRecord()`

- Purpose: To verify that an existing medical history record can be updated and saved correctly in the database.
- Approach: The test updates the treatment field of an existing medical history record, saves the changes, and then fetches the updated record from the database to verify that the update was successful.
- Expected Outcome: The existing medical history record is updated successfully, and the fetched record has the updated treatment:
    - Treatment = "Updated Checkup"

PetServiceTests.java - Preeti

Test 1: `testGetPetById_PetExists()`

- Purpose: To verify that a pet can be retrieved by its ID when the pet exists in the database.
- Approach: The test calls the `getPetById` method with the ID of `testPet` and verifies that the correct pet is returned.
- Expected Outcome: The pet is successfully fetched, and its name matches the expected value: "TestPet."

Test 2: `testGetPetById_PetNotFound()`

- Purpose: To test the behavior when attempting to retrieve a pet by an ID that doesn't exist in the database.
- Approach: The test calls the `getPetById` method with a non-existent pet ID (`999L`) and verifies that the service returns null.
- Expected Outcome: No pet is found, and the service returns null.

Test 3: `testGetPetsByUserId()`

- Purpose: To verify that all pets associated with a specific user ID are retrieved correctly.
- Approach: The test calls the `getPetsByUserId` method with the ID of `testUser` and verifies the list of pets returned.
- Expected Outcome: One pet is returned, and its name matches the expected value: "TestPet."

Test 4: `testSavePet_NewPet()`

- Purpose: To verify that a new pet can be saved to the database successfully.
- Approach: The test creates a new pet, saves it using the `save` method, and then fetches the saved pet from the database to verify that it was saved correctly.
- Expected Outcome: The new pet is saved successfully, and its name matches the expected value: "Charlie."

Test 5: `testDeletePet()`

- Purpose: To verify that a pet can be deleted from the database using its ID.
- Approach: The test deletes `testPet` using the `delete` method and then verifies that the pet is no longer present in the database.
- Expected Outcome: The pet is deleted successfully, and it can no longer be found in the database.

Test 6: `testGetPetsByVeterinarianId()`

- Purpose: To test the behavior of fetching pets associated with a specific veterinarian ID when no pets are linked to that veterinarian.
- Approach: The test calls the `getPetsByVeterinarianId` method with a veterinarian ID (`1L`) and verifies that no pets are returned.
- Expected Outcome: The returned list is empty, confirming that no pets are linked to this veterinarian ID.

PhysicalExamServiceTests.java - Preeti

Test 1: testGetPhysicalExamsByPetId()

- Purpose: To verify that the correct physical exam records are retrieved for a specific pet by its ID.
- Approach: The test mocks the findByPetId method in the repository to return two physical exams for testPet. It then calls the getPhysicalExamsByPetId method and verifies the number of exams returned and their details.
- Expected Outcome: Two physical exam records are returned with the following details:
    - Exam 1: Notes = "Annual Checkup"
    - Exam 2: Notes = "Vaccination"

Test 2: testGetPhysicalExamsByPetId_EmptyList()

- Purpose: To test the behavior when no physical exam records exist for a pet.
- Approach: The test mocks the findByPetId method in the repository to return an empty list. It then calls the getPhysicalExamsByPetId method and verifies that the result is an empty list.
- Expected Outcome: No physical exam records are returned, and the list is empty.

Test 3: testSaveNewPhysicalExam()

- Purpose: To verify that a new physical exam record can be saved to the database successfully.
- Approach: The test creates a new physical exam, mocks the save method in the repository, and then calls the save method in the service. It verifies that the repository's save method is called once.
- Expected Outcome: The new physical exam is saved successfully, and the repository's save method is called.

Test 4: testUpdatePhysicalExam()

- Purpose: To verify that an existing physical exam record can be updated and saved to the database successfully.
- Approach: The test updates the notes of an existing physical exam, mocks the save method in the repository, and then calls the save method in the service. It verifies that the repository's save method is called once.
- Expected Outcome: The existing physical exam is updated and saved successfully, and the repository's save method is called.

TreatmentPlanServiceTests.java - Preeti

Test 1: testGetTreatmentPlansByPetId()

- **Purpose:** To verify that the correct treatment plan records are retrieved for a specific pet by its ID.
- **Approach:** The test mocks the findByPetId method in the repository to return two treatment plans for testPet. It then calls the getTreatmentPlansByPetId method and verifies the number of plans returned and their descriptions.
- **Expected Outcome:** Two treatment plan records are returned with the following details:
    - Plan 1: Description = "Vaccination Plan"
    - Plan 2: Description = "Dental Care Plan"

Test 2: testGetTreatmentPlansByPetId_EmptyList()

- **Purpose:** To test the behavior when no treatment plan records exist for a pet.
- **Approach:** The test mocks the findByPetId method in the repository to return an empty list. It then calls the getTreatmentPlansByPetId method and verifies that the result is an empty list.
- **Expected Outcome:** No treatment plan records are returned, and the list is empty.

Test 3: testSaveNewTreatmentPlan()

- **Purpose:** To verify that a new treatment plan record can be saved to the database successfully.
- **Approach:** The test creates a new treatment plan, mocks the save method in the repository, and then calls the save method in the service. It verifies that the repository's save method is called once.
- **Expected Outcome:** The new treatment plan is saved successfully, and the repository's save method is called.

Test 4: testUpdateTreatmentPlan()

- **Purpose:** To verify that an existing treatment plan record can be updated and saved to the database successfully.
- **Approach:** The test updates the description of an existing treatment plan, mocks the save method in the repository, and then calls the save method in the service. It verifies that the repository's save method is called once.
- **Expected Outcome:** The existing treatment plan is updated and saved successfully, and the repository's save method is called.

VaccinationServiceTests.java - Preeti

Test 1: testGetVaccinationsByPetId()

- Purpose: To verify that the correct vaccination records are retrieved for a specific pet by its ID.
- Approach: The test calls the getVaccinationsByPetId method with the ID of testPet and verifies the number of vaccinations returned and their details.
- Expected Outcome: Two vaccination records are returned with the following details:
    - Vaccination 1: Vaccine Name = "Rabies", Administered By = "Doctor Strange"
    - Vaccination 2: Vaccine Name = "Distemper", Administered By = "Doctor Strange"

Test 2: testGetVaccinationsByPetId_Empty()

- Purpose: To test the behavior when no vaccination records exist for a pet.
- Approach: The test creates a new pet with no vaccinations, calls the getVaccinationsByPetId method, and verifies that the result is an empty list.
- Expected Outcome: No vaccination records are returned, and the list is empty.

Test 3: testSaveNewVaccination()

- Purpose: To verify that a new vaccination record can be saved to the database successfully.
- Approach: The test creates a new vaccination record, saves it using the save method, and then fetches the saved record from the database to verify the data.
- Expected Outcome: The new vaccination is saved successfully, and the fetched record has the following details:
    - Vaccine Name = "Parvovirus"
    - Administered By = "Doctor Strange"

Test 4: testUpdateVaccination()

- Purpose: To verify that an existing vaccination record can be updated and saved to the database successfully.
- Approach: The test updates the vaccine name of an existing vaccination, saves the changes using the save method, and then fetches the updated record from the database to verify that the update was successful.
- Expected Outcome: The existing vaccination is updated successfully, and the fetched record has the updated vaccine name:
    - Vaccine Name = "Updated Rabies”

WeightRecordServiceTests.java - Preeti

Test 1: testGetWeightRecordsByPetId()

- Purpose: To verify that the correct weight record entries are retrieved for a specific pet by its ID.
- Approach: The test mocks the findByPetId method in the repository to return two weight records for testPet. It then calls the getWeightRecordsByPetId method and verifies the number of records returned and their details.
- Expected Outcome: Two weight records are returned with the following details:
    - Record 1: Weight = 10.5
    - Record 2: Weight = 12.0

Test 2: testGetWeightRecordsByPetId_Empty()

- Purpose: To test the behavior when no weight record entries exist for a pet.
- Approach: The test mocks the findByPetId method in the repository to return an empty list. It then calls the getWeightRecordsByPetId method and verifies that the result is an empty list.
- Expected Outcome: No weight records are returned, and the list is empty.

Test 3: testSaveNewWeightRecord()

- Purpose: To verify that a new weight record entry can be saved to the database successfully.
- Approach: The test creates a new weight record, calls the save method in the service, and verifies that the repository's save method is called with the new record.
- Expected Outcome: The new weight record is saved successfully, and the repository's save method is called.

Test 4: testUpdateWeightRecord()

- Purpose: To verify that an existing weight record entry can be updated and saved to the database successfully.
- Approach: The test modifies an existing weight record, calls the save method in the service, and verifies that the repository's save method is called with the updated record.
- Expected Outcome: The existing weight record is updated successfully, and the repository's save method is called.

Prescription Design Choices

The code for the UI originally for the user side of prescriptions in milestone 2 was moved to the veterinarian side, as having add, edit and delete buttons for a veterinarian is more sensible than allowing the user to add, edit and delete their own prescriptions. The modified prescriptions will appear on the user side.

The user side UI for prescriptions is a modified version of the veterinarian view, with the add, edit and delete buttons removed, and replaced with a new refill button, which will write refills to a table and redisplay it on the pet prescriptions page under the prescription and prescription history tables. The refills are cancellable. A download button is also present and will download the entire view of the page using the browser's in-built function, minus the buttons.

PrescriptionControllerTests.java

1. **testGetVetPets_success**

Purpose: Ensure that the system returns a non-empty list of pets for a valid veterinarian ID.

Approach:

- Call the /api/prescriptions/vet-pets endpoint with a valid vetId.
- Verify that the response status is 200 OK.
- Check that the JSON response contains a non-empty list.

Expected Outcome: The endpoint returns a list of pets associated with the veterinarian, and the list is not empty.

2. **testGetVetPets_vetDoesNotExists**

Purpose: Ensure that the system returns an empty list when a veterinarian ID does not exist.

Approach:

- Call the /api/prescriptions/vet-pets endpoint with an invalid vetId.
- Verify that the response status is 200 OK.
- Check that the response is an empty JSON array.

Expected Outcome: The system returns an empty list, indicating no pets are associated with an invalid veterinarian ID.

3. **testAddPrescription_success**

Purpose: Ensure that a new prescription can be successfully added to the system.

Approach:

- Perform a POST request to /api/prescriptions/add with a valid prescription JSON.
- Verify that the response status is 200 OK.
- Check the prescription repository to ensure the count is increased by 1.

Expected Outcome: The prescription is added successfully, and the repository contains one new record.

4. **testUpdatePrescription_success**

Purpose: Ensure that an existing prescription can be updated successfully.

Approach:

- Save a prescription with incorrect data (e.g., incorrect practitioner name).

- Perform a PUT request to /api/prescriptions/{id} with the updated prescription JSON.
- Verify that the response status is 200 OK.
- Check the updated prescription to ensure the practitioner name has been updated.

Expected Outcome: The practitioner name in the prescription is updated successfully in the database.

5. **testUpdatePrescription_notFound**

Purpose: Ensure that updating a non-existing prescription returns a 404 Not Found error.

Approach:

- Perform a DELETE request to /api/prescriptions/1, where no prescription exists.
- Verify that the response status is 404 Not Found.

Expected Outcome: The system returns a 404 status when attempting to update a non-existing prescription.

6. **testDeletePrescription_success**

Purpose: Ensure that an existing prescription can be deleted successfully.

Approach:

- Add a prescription to the repository.
- Perform a DELETE request to /api/prescriptions/{id} with the prescription ID.
- Verify that the response status is 204 No Content.
- Check the repository to ensure the prescription has been removed.

Expected Outcome: The prescription is deleted successfully, and the repository is empty.

7. **testDeletePrescription_notFound**

Purpose: Ensure that deleting a non-existing prescription returns a 404 Not Found error.

Approach:

- Perform a DELETE request to /api/prescriptions/1, where no prescription exists.

- Verify that the response status is 404 Not Found.

Expected Outcome: The system returns a 404 status when attempting to delete a non-existing prescription.

8. testAddPrescription_invalidData

Purpose: Ensure that adding a prescription with invalid data returns a 400 Bad Request error.

Approach:

- Perform a POST request to /api/prescriptions/add with an invalid prescription JSON (missing or empty fields).
- Verify that the response status is 400 Bad Request.
- Ensure no prescription is added to the repository.

Expected Outcome: The system returns a 400 Bad Request error, and no prescription is added to the repository.

9. testGetAllPrescriptions_success

Purpose: Ensure that all prescriptions for a specific pet can be retrieved successfully.

Approach:

- Add a prescription to the repository.
- Perform a GET request to /api/prescriptions/all with the pet's ID as a request parameter.
- Verify that the response status is 200 OK.
- Check that the response contains a non-empty list of prescriptions.

Expected Outcome: The system returns all prescriptions for the specified pet.

10. testGetAllPrescriptions_empty

Purpose: Ensure that the system returns a 404 Not Found error when there are no prescriptions for a specific pet.

Approach:

- Perform a GET request to /api/prescriptions/all/1, where no prescriptions exist.

- Verify that the response status is 404 Not Found.

Expected Outcome: The system returns a 404 status indicating no prescriptions were found.

11. testGetPrescriptionById_success

Purpose: Ensure that a specific prescription can be retrieved by its ID.

Approach:

- Add a prescription to the repository.
- Perform a GET request to /api/prescriptions/{id} with the prescription ID.
- Verify that the response status is 200 OK.
- Check that the response contains the correct prescription details.

Expected Outcome: The prescription is retrieved successfully by its ID.

12. testGetPrescriptionById_notFound

Purpose: Ensure that retrieving a non-existing prescription by ID returns a 404 Not Found error.

Approach:

- Perform a GET request to /api/prescriptions/999, where no prescription exists.
- Verify that the response status is 404 Not Found.

Expected Outcome: The system returns a 404 status when attempting to retrieve a non-existing prescription.

13. testAddRefill_success

Purpose: Ensure that a refill can be successfully added to the system.

Approach:

- Perform a POST request to /api/prescriptions/refills/add with a valid refill JSON.
- Verify that the response status is 200 OK.
- Ensure the refill is saved in the repository.

Expected Outcome: The refill is added successfully, and the repository contains the new refill record.

14. **testAddRefill_invalidData**

Purpose: Ensure that adding a refill with invalid data returns a 400 Bad Request error.

Approach:

- Perform a POST request to /api/prescriptions/refills/add with an invalid refill JSON.
- Verify that the response status is 400 Bad Request.

Expected Outcome: The system returns a 400 Bad Request error, and no refill is added.

15. **testGetRefillsByUserId_success**

Purpose: Ensure that all refills for a specific user can be retrieved successfully.

Approach:

- Add a refill to the repository.
- Perform a GET request to /api/prescriptions/refills/user/{userId} with the user ID.
- Verify that the response status is 200 OK.
- Check that the response contains a non-empty list of refills.

Expected Outcome: The system returns all refills for the specified user.

16. **testGetRefillsByUserId_notFound**

Purpose: Ensure that retrieving refills for a non-existing user returns an empty list.

Approach:

- Perform a GET request to /api/prescriptions/refills/user/999, where no refills exist for the user.
- Verify that the response status is 200 OK.
- Ensure the response is an empty list.

Expected Outcome: The system returns an empty list for a non-existing user ID.

17. testDeleteRefill_success

Purpose: Ensure that an existing refill can be deleted successfully.

Approach:

- Add a refill to the repository.
- Perform a DELETE request to /api/prescriptions/refills/{id} with the refill ID.
- Verify that the response status is 204 No Content.
- Check the repository to ensure the refill has been removed.

Expected Outcome: The refill is deleted successfully, and the repository is empty.

18. testDeleteRefill_notFound

Purpose: Ensure that deleting a non-existing refill returns a 404 Not Found error.

Approach:

- Perform a DELETE request to /api/prescriptions/refills/999, where no refill exists.
- Verify that the response status is 404 Not Found.

Expected Outcome: The system returns a 404 status when attempting to delete a non-existing refill.

PrescriptionRepositoryTests.java

1. testFindByPet

Purpose: Verify that the findByPet method correctly retrieves prescriptions associated with a specific pet.

Approach:

- Call the findByPet method with the test pet.
- Check the size of the returned list and verify that it matches the expected number of prescriptions.
- Ensure that the prescriptions in the list match the expected values.

Expected Outcome: The method returns a list containing two prescriptions, with the correct names "Amoxicillin" and "Ibuprofen".

2. testAddPrescription

Purpose: Ensure that a new prescription can be successfully added to the repository.

Approach:

- Create a new prescription for the test pet.
- Save the prescription using the repository.
- Retrieve the saved prescription by its ID and verify its properties.

Expected Outcome: The prescription is successfully added, and its properties match those of the saved prescription.

3. testUpdatePrescription

Purpose: Verify that an existing prescription can be updated successfully.

Approach:

- Retrieve an existing prescription for the test pet.
- Update a field (dosage) of the prescription.
- Save the updated prescription and retrieve it again.

Expected Outcome: The prescription's dosage is updated successfully, and the new dosage matches the expected value.

4. testDeletePrescription

Purpose: Ensure that an existing prescription can be deleted from the repository.

Approach:

- Retrieve an existing prescription for the test pet.
- Delete the prescription using the repository.
- Attempt to retrieve the deleted prescription and verify its absence.

Expected Outcome: The prescription is successfully deleted, and it cannot be found in the repository afterwards.

5. testFindAllPrescriptions

Purpose: Verify that the repository can retrieve all prescriptions correctly.

Approach:

- Call the findAll method of the prescription repository.
- Check the size of the returned list against the expected number of prescriptions.

Expected Outcome: The method returns a list containing two prescriptions.

6. testAddPrescriptionWithNullFields

Purpose: Ensure that attempting to add a prescription with null required fields raises a DataIntegrityViolationException.

Approach:

- Create a prescription with a null value for the required prescription field.
- Attempt to save the prescription and assert that an exception is thrown.

Expected Outcome: A DataIntegrityViolationException is thrown when trying to save the prescription.

7. testUpdatePrescriptionWithNullFields

Purpose: Ensure that updating an existing prescription to have null required fields raises a DataIntegrityViolationException.

Approach:

- Retrieve an existing prescription for the test pet.
- Set the prescription field to null.
- Attempt to save the updated prescription and assert that an exception is thrown.

Expected Outcome: A DataIntegrityViolationException is thrown when trying to save the updated prescription.

8. testAddPrescriptionWithInvalidDosage

Purpose: Ensure that attempting to add a prescription with an invalid dosage raises a DataIntegrityViolationException.

Approach:

- Create a prescription with a null value for the dosage field.
- Attempt to save the prescription and assert that an exception is thrown.

Expected Outcome: A DataIntegrityViolationException is thrown when trying to save the prescription with an invalid dosage.

9. testDeleteNonExistentPrescription

Purpose: Verify that attempting to delete a non-existent prescription ID does not throw an exception and that the ID remains absent from the repository.

Approach:

- Attempt to find a prescription with a non-existent ID and assert that it is not present.
- Attempt to delete the non-existent prescription ID and verify that it does not throw an exception.
- Confirm that the prescription ID still does not exist after the deletion attempt.

Expected Outcome: No exceptions are thrown during the deletion attempt, and the non-existent prescription ID remains absent from the repository.

PrescriptionHistoryTests.java

1. testFindByPet

Purpose: To verify that the findByPet method of the PrescriptionHistoryRepository correctly retrieves prescription histories associated with a specific pet.

Approach:

- The test first retrieves all prescription histories for testPet.
- It checks that the size of the returned list is 2, confirming that both entries added during the setup are present.
- It also checks if at least one history entry contains a specified practitioner ("Dr. John Smith") and if at least one entry has a specified prescription ("Antibiotic").

Expected Outcome: The test should pass if the size of the retrieved list is 2 and contains the specified practitioner and prescription, indicating that the repository method is functioning correctly.

2. testPrescriptionDetails

Purpose: To validate the specific details of the first prescription history entry retrieved for a given pet.

Approach:

- The test retrieves all prescription histories for testPet and checks the details of the first entry in the list.
- It asserts that the practitioner's name, prescription name, and additional information match the expected values set during the setup.

Expected Outcome: The assertions should pass, confirming that the PrescriptionHistory objects returned by the repository contain the correct details.

3. testFindByPet_NoHistory

Purpose: To test the behavior of the `findByPet` method when there is no prescription history associated with a pet.

Approach:

- The test creates a new user and a new pet that do not have any prescription histories.
- It then attempts to retrieve prescription histories for this new pet.

Expected Outcome: The test should pass if the retrieved list is empty (size is 0), indicating that the repository correctly handles cases where no histories exist for the provided pet.

RefillRepositoryTests.java

1. testAddRefill

Purpose: To verify that a refill can be successfully added to the repository.

Approach:

- Create a new Refill object with valid details and save it to the repository.
- Retrieve the saved refill by its ID and check that it exists and that the first name matches the expected value.

Expected Outcome: The refill should be present in the repository, and the first name should match "John".

2. testFindByUserId

Purpose: To test the findByUserId method and ensure it correctly retrieves refills associated with a specific user ID.

Approach:

- Create and save a Refill object, then retrieve refills by the user ID associated with the pet.
- Assert that the size of the returned list is 1 and that the first name matches the expected value.

Expected Outcome: The test should pass if one refill is found with the first name "John".

3. testUpdateRefill

Purpose: To verify that an existing refill can be updated successfully in the repository.

Approach:

- Create and save a Refill object, then update its last name and save it again.
- Retrieve the refill by its ID and check that the last name matches the updated value.

Expected Outcome: The updated refill should have the last name "Smith".

4. testDeleteRefill

Purpose: To test that a refill can be deleted from the repository.

Approach:

- Create and save a Refill object, then delete it from the repository.
- Attempt to retrieve the deleted refill by its ID and assert that it no longer exists.

Expected Outcome: The test should pass if the refill cannot be found after deletion.

5. testFindAllRefills

Purpose: To verify that all refills can be retrieved from the repository.

Approach:

- Create and save two Refill objects, then retrieve all refills from the repository.
- Assert that the size of the retrieved list is 2.

Expected Outcome: The test should pass if exactly two refills are found.

6. testFindNonExistentRefill

Purpose: To test the behavior when attempting to find a refill that does not exist.

Approach:

- Attempt to retrieve a refill with an ID that is not present in the repository (e.g., ID 999).

Expected Outcome: The test should pass if the retrieved optional is empty, indicating the refill does not exist.

7. testDeleteNonExistentRefill

Purpose: To verify that attempting to delete a non-existent refill does not throw an exception.

Approach:

- Create a Refill object with a non-existent ID and attempt to delete it.

Expected Outcome: The test should pass if no exception is thrown during the delete operation.

8. testAddRefillWithNullFields

Purpose: To test that attempting to add a refill with required fields set to null (e.g., prescription) results in a DataIntegrityViolationException.

Approach:

- Create a Refill object with the prescription field set to null and attempt to save it.

Expected Outcome: The test should pass if a DataIntegrityViolationException is thrown, indicating that the save operation failed due to the null field.

List of New Code Files For Milestone 3

Controller

- **VeterinarianController.java**
- **NotificatonController.java**
- **FAQController.java**
- **ContactController.java**

Service

- **VeterinarianService.java**
- **NotificationService.java**

Repository

- **VeterinarianRepository.java**
- **RefillRepository.java**
- **NotificationRepository.java**

- **FAQRepository.java**

Model

- **Refill.java**
- **Notification.java**

Html Files:

- **_veterinarianNavigation.html**
- **veterinarian-upload-records.html**
- **veterinarian-signup.html**
- **veterinarian-login.html**
- **veterinarian-prescription.html**
- **veterinarian-dashboard.html**
- **veterinarian-appointments.html**
- **faq.html**
- **contact.html**

Js Files:

- **veterinarian-prescription.js**
- **veterinarian-appointments.js**
- **userGuide.js**

Tests:

Controller

- **PrescriptionControllerTests.java**
- **NotificationControllerTests.java**
- **VeterinarianControllerTests.java**
- **FAQControllerTests.java**

Repository

- **PrescriptionRepositoryTests.java**
- **RefillRepositoryTests.java**
- **MedicalHistoryRepositoryTests**
- **PhysicalExamRepositoryTests**
- **TreatmentPlanRepositoryTests**
- **VaccinationRepositoryTests**
- **WeightRecordRepositoryTests**

Service

- **PrescriptionHistoryTests.java**
- **NotificationServiceTests.java**
- **ArticleServiceTests.java**
- **MedicalHistoryServiceTests**
- **PhysicalExamServiceTests**
- **TreatmentPlanServiceTests**
- **VaccinationServiceTests**
- **VeterinarianServiceTests**
- **WeightRecordsServiceTests**

Evidence/documentation of test execution and results - from SureFire Reports

au.edu.rmit.sept.webapp.controller.ArticleControllerTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.controller.ArticleControllerTests
-------------------------------------------------------------------------------
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.256 s - in
au.edu.rmit.sept.webapp.controller.ArticleControllerTests
```

au.edu.rmit.sept.webapp.controller.FAQControllerTests.txt

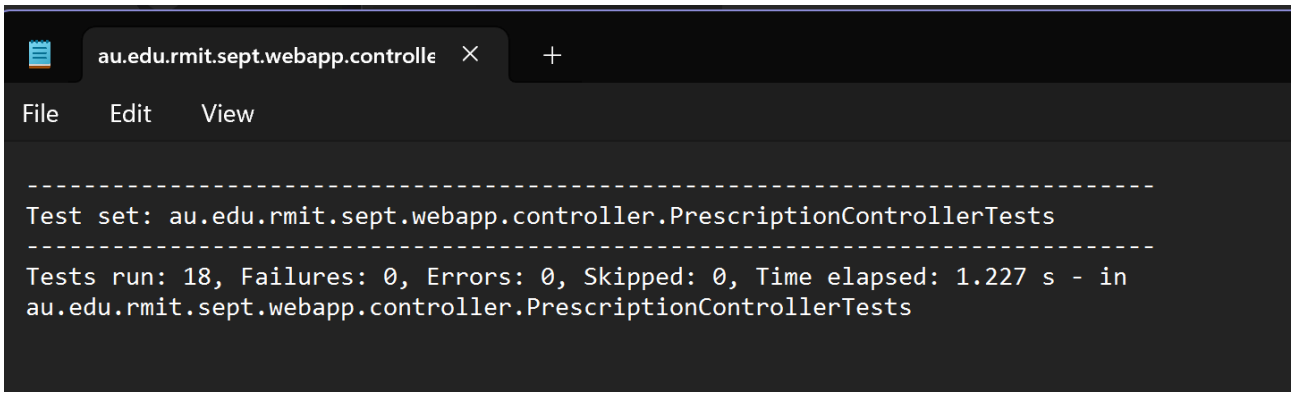
```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.controller.FAQControllerTests
-------------------------------------------------------------------------------
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.588 s - in
au.edu.rmit.sept.webapp.controller.FAQControllerTests
```

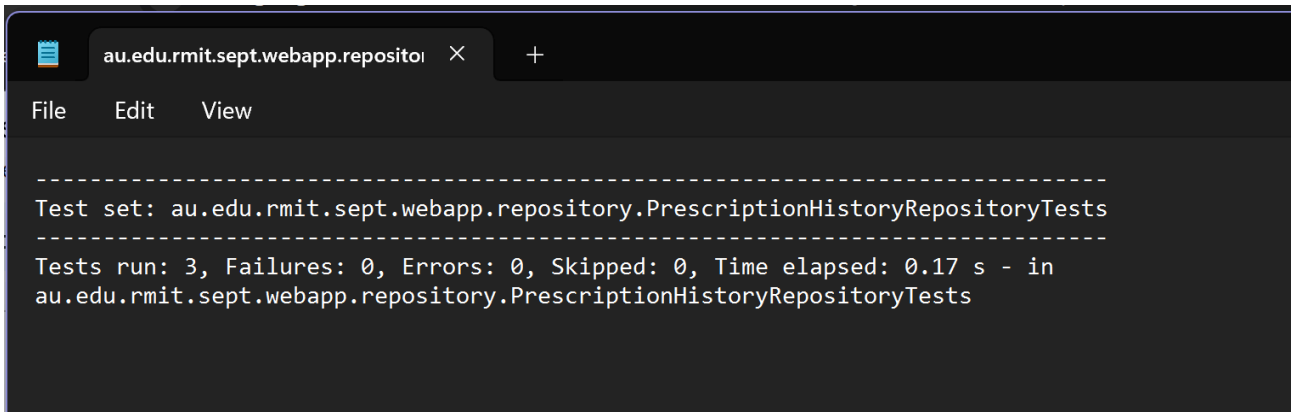
au.edu.rmit.sept.webapp.controller.MedicalRecordsControllerTests.txt

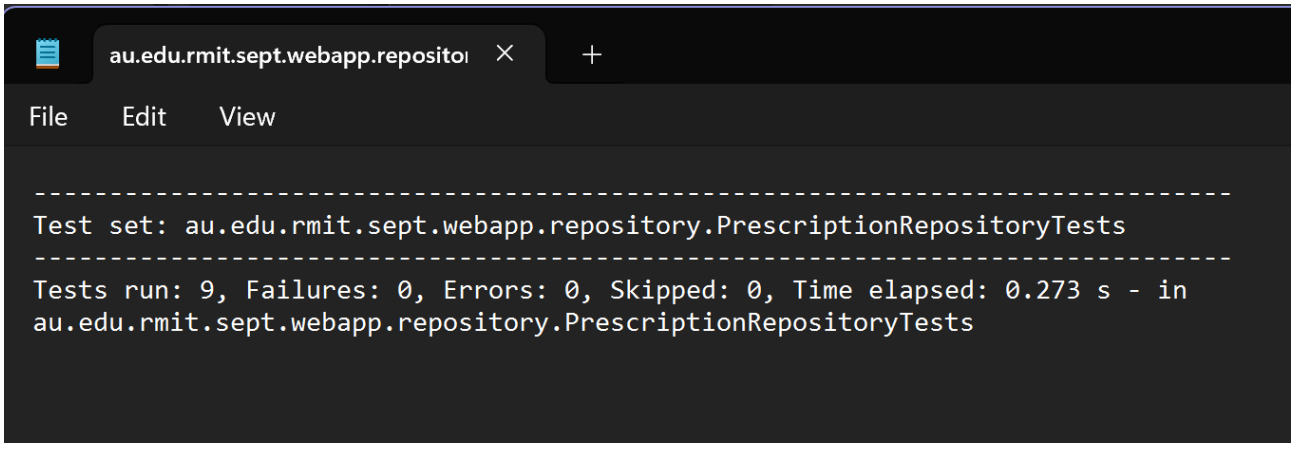
```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.controller.MedicalRecordsControllerTests
-------------------------------------------------------------------------------
Tests run: 10, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.47 s - in
au.edu.rmit.sept.webapp.controller.MedicalRecordsControllerTests
```

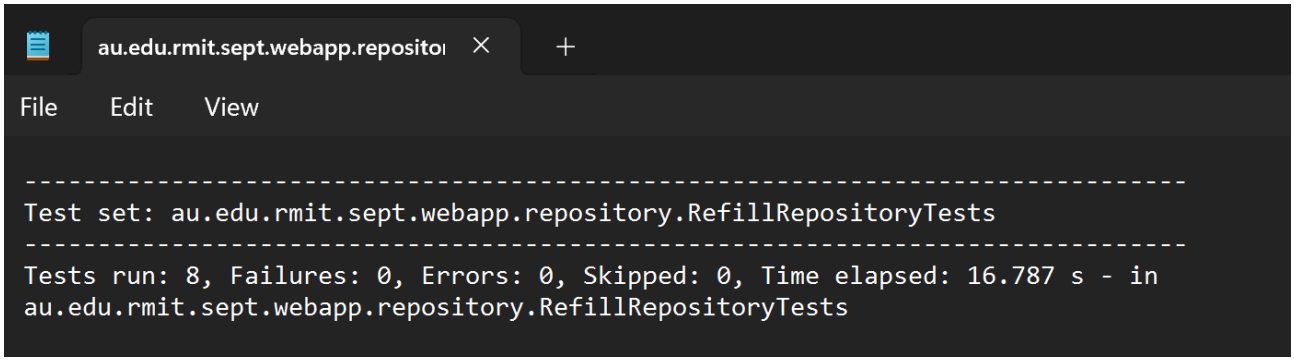
au.edu.rmit.sept.webapp.controller.NotificationControllerTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.controller.NotificationControllerTests
-------------------------------------------------------------------------------
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.271 s - in
au.edu.rmit.sept.webapp.controller.NotificationControllerTests
```


au.edu.rmit.sept.webapp.controlle
File
Edit
View
Test set: au.edu.rmit.sept.webapp.controller.PrescriptionControllerTests
Tests run: 18, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.227 s - in
au.edu.rmit.sept.webapp.controller.PrescriptionControllerTests


au.edu.rmit.sept.webapp.repositol
File
Edit
View
Test set: au.edu.rmit.sept.webapp.repository.PrescriptionHistoryRepositoryTests
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.17 s - in
au.edu.rmit.sept.webapp.repository.PrescriptionHistoryRepositoryTests


au.edu.rmit.sept.webapp.repositol
File
Edit
View
Test set: au.edu.rmit.sept.webapp.repository.PrescriptionRepositoryTests
Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.273 s - in
au.edu.rmit.sept.webapp.repository.PrescriptionRepositoryTests


au.edu.rmit.sept.webapp.repositol
File
Edit
View
Test set: au.edu.rmit.sept.webapp.repository.RefillRepositoryTests
Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 16.787 s - in
au.edu.rmit.sept.webapp.repository.RefillRepositoryTests

au.edu.rmit.sept.webapp.controller.ServiceControllerTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.controller.ServiceControllerTests
-------------------------------------------------------------------------------
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.026 s - in
au.edu.rmit.sept.webapp.controller.ServiceControllerTests
```

au.edu.rmit.sept.webapp.controller.UserControllerTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.controller.UserControllerTests
-------------------------------------------------------------------------------
Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.987 s - in
au.edu.rmit.sept.webapp.controller.UserControllerTests
```

au.edu.rmit.sept.webapp.controller.VeterinarianAvailabilityControllerTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.controller.VeterinarianAvailabilityControllerTests
-------------------------------------------------------------------------------
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.022 s - in
au.edu.rmit.sept.webapp.controller.VeterinarianAvailabilityControllerTests
```

au.edu.rmit.sept.webapp.controller.VeterinarianControllerTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.controller.VeterinarianControllerTests
-------------------------------------------------------------------------------
Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.071 s - in
au.edu.rmit.sept.webapp.controller.VeterinarianControllerTests
```

au.edu.rmit.sept.webapp.HomeTemplateIntegrationTest.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.HomeTemplateIntegrationTest
-------------------------------------------------------------------------------
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 3.056 s - in
au.edu.rmit.sept.webapp.HomeTemplateIntegrationTest
```

au.edu.rmit.sept.webapp.repository.ArticleRepositoryTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.repository.ArticleRepositoryTests
-------------------------------------------------------------------------------
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 3.707 s - in
au.edu.rmit.sept.webapp.repository.ArticleRepositoryTests
```

au.edu.rmit.sept.webapp.repository.MedicalHistoryRepositoryTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.repository.MedicalHistoryRepositoryTests
-------------------------------------------------------------------------------
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.262 s - in
au.edu.rmit.sept.webapp.repository.MedicalHistoryRepositoryTests
```

au.edu.rmit.sept.webapp.repository.PhysicalExamRepositoryTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.repository.PhysicalExamRepositoryTests
-------------------------------------------------------------------------------
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.103 s - in
au.edu.rmit.sept.webapp.repository.PhysicalExamRepositoryTests
```

au.edu.rmit.sept.webapp.repository.TreatmentPlanRepositoryTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.repository.TreatmentPlanRepositoryTests
-------------------------------------------------------------------------------
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.064 s - in
au.edu.rmit.sept.webapp.repository.TreatmentPlanRepositoryTests
```

au.edu.rmit.sept.webapp.repository.UserRepositoryTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.repository.UserRepositoryTests
-------------------------------------------------------------------------------
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 15.752 s - in
au.edu.rmit.sept.webapp.repository.UserRepositoryTests
```

au.edu.rmit.sept.webapp.repository.VaccinationRepositoryTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.repository.VaccinationRepositoryTests
-------------------------------------------------------------------------------
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.219 s - in
au.edu.rmit.sept.webapp.repository.VaccinationRepositoryTests
```

au.edu.rmit.sept.webapp.repository.WeightRecordRepositoryTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.repository.WeightRecordRepositoryTests
-------------------------------------------------------------------------------
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.07 s - in
au.edu.rmit.sept.webapp.repository.WeightRecordRepositoryTests
```

au.edu.rmit.sept.webapp.service.ArticleServiceTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.service.ArticleServiceTests
-------------------------------------------------------------------------------
Tests run: 11, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.158 s - in
au.edu.rmit.sept.webapp.service.ArticleServiceTests
```

au.edu.rmit.sept.webapp.service.BookmarkServiceTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.service.BookmarkServiceTests
-------------------------------------------------------------------------------
Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.567 s - in
au.edu.rmit.sept.webapp.service.BookmarkServiceTests
```

au.edu.rmit.sept.webapp.service.FileGenerationServiceTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.service.FileGenerationServiceTests
-------------------------------------------------------------------------------
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.522 s - in
au.edu.rmit.sept.webapp.service.FileGenerationServiceTests
```

au.edu.rmit.sept.webapp.service.MedicalHistoryServiceTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.service.MedicalHistoryServiceTests
-------------------------------------------------------------------------------
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.121 s - in
au.edu.rmit.sept.webapp.service.MedicalHistoryServiceTests
```

au.edu.rmit.sept.webapp.service.NotificationServiceTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.service.NotificationServiceTests
-------------------------------------------------------------------------------
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.919 s - in
au.edu.rmit.sept.webapp.service.NotificationServiceTests
```

au.edu.rmit.sept.webapp.service.PetServiceTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.service.PetServiceTests
-------------------------------------------------------------------------------
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.377 s - in
au.edu.rmit.sept.webapp.service.PetServiceTests
```

au.edu.rmit.sept.webapp.service.PhysicalExamServiceTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.service.PhysicalExamServiceTests
-------------------------------------------------------------------------------
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.261 s - in
au.edu.rmit.sept.webapp.service.PhysicalExamServiceTests
```

au.edu.rmit.sept.webapp.service.TreatmentPlanServiceTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.service.TreatmentPlanServiceTests
-------------------------------------------------------------------------------
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.969 s - in
au.edu.rmit.sept.webapp.service.TreatmentPlanServiceTests
```

au.edu.rmit.sept.webapp.service.UserServiceTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.service.UserServiceTests
-------------------------------------------------------------------------------
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.989 s - in
au.edu.rmit.sept.webapp.service.UserServiceTests
```

au.edu.rmit.sept.webapp.service.VaccinationServiceTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.service.VaccinationServiceTests
-------------------------------------------------------------------------------
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.132 s - in
au.edu.rmit.sept.webapp.service.VaccinationServiceTests
```

au.edu.rmit.sept.webapp.service.VeterinarianServiceTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.service.VeterinarianServiceTests
-------------------------------------------------------------------------------
Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 3.274 s - in
au.edu.rmit.sept.webapp.service.VeterinarianServiceTests
```

au.edu.rmit.sept.webapp.service.WeightRecordServiceTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.service.WeightRecordServiceTests
-------------------------------------------------------------------------------
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 3.989 s - in
au.edu.rmit.sept.webapp.service.WeightRecordServiceTests
```

au.edu.rmit.sept.webapp.WebappApplicationTests.txt

```
-------------------------------------------------------------------------------
Test set: au.edu.rmit.sept.webapp.WebappApplicationTests
-------------------------------------------------------------------------------
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.016 s - in
au.edu.rmit.sept.webapp.WebappApplicationTests
```

Appendix A: Glossary

AES-256	Advanced Encryption Standard 256-bit
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
APNs	Apple Push Notification Service
APPS	Applications (commonly used to refer to software applications)
ASD	Autism Spectrum Disorder
CI/CD	Continuous Integration/Continuous Delivery

CPU	Central Processing Unit
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
DAO	Data Access Object
FAQ	Frequently Asked Questions
GB	Gigabyte
GDPR	General Data Protection Regulation
GUI	Graphical User Interface
HDD	Hard Disk
HIPAA	Health Insurance Portability and Accountability Act
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IP	Internet Protocol
IEC 27001	International Electrotechnical Commission 27001
ISMS	Information Security Management System
ISO 27001	International Standard on requirements for information security management
ISO 27002	Information security, Cybersecurity and privacy protection – Information security controls
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
LTS	Long-Term Support
MFA	Multi-factor Authentication
MVC	Model View Controller
NDB	Notifiable Data Breaches
OAIC	Office of the Australian Information Commissioner
ORM	Object-Relational Mapping
OS	Operating System
RAM	Random Access Memory
RBAC	Role-based Access Control
RDBMS	Relational Database Management System
REST	Representational State Transfer
SMTP	Simple Mail Transfer Protocol
SOC 2 Type II	Service Organization Control Type 2
SMS	Short Message Service
SQL	Structured Query Language
SRS	Software Requirements Specification
SSD	Solid State Drive
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UI	User Interface

Appendix B: Analysis Models

Will be added upon in Milestone 2

Appendix C: To Be Determined List

Don’t have any to be determined references for tracking to closure yet