UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

MARCO ANTONIO ROZO

APLICATIVO MÓVEL PARA GESTÃO DO GADO EM PROPRIEDADES RURAIS

PATO BRANCO

2023

MARCO ANTONIO ROZO

APLICATIVO MÓVEL PARA GESTÃO DO GADO EM PROPRIEDADES RURAIS

Mobile Application For Cattle Management In Rural Properties

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas da Universidade Tecnológica Federal do Paraná.

Orientador: Vinicius Pegorini

PATO BRANCO

2023

MARCO ANTONIO ROZO

APLICATIVO MÓVEL PARA GESTÃO DO GADO EM PROPRIEDADES RURAIS

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 28/Junho/2023

Vinicius Pegorini
Mestrado
Universidade Tecnológica Federal do Paraná

Andreia Scariot Beulke
Mestrado
Universidade Tecnológica Federal do Paraná

Dalcimar Casanova
Doutorado
Universidade Tecnológica Federal do Paraná

PATO BRANCO

2023

RESUMO

Mesmo com o estado do Paraná sendo um dos maiores responsáveis pela produção nacional do leite, ainda é possível notar a falta do manejo do produtor quanto ao seu rebanho, principalmente quando se trata da alimentação, pois mesmo sendo de qualidade, pode não ser adequada às necessidades nutricionais dos animais. Essas atividades podem ser otimizadas por meio de um sistema computacional, que pode auxiliar no tratamento e armazenamento dos dados relacionados ao rebanho e alimentação, permitindo que o gerenciamento do rebanho seja realizado de maneira mais eficiente, pois a consulta dos dados de forma ordenada pode auxiliar na tomada de decisões. O sistema proposto e apresentado neste projeto, tem como objetivo auxiliar tanto os técnicos do Instituto de Desenvolvimento Rural (IDR) quanto os produtores na gestão dos animais, gestão nutricional, acompanhamento do gado nas propriedades rurais e gestão das informações das propriedades. As principais funcionalidades desenvolvidas foram o armazenamento e gestão de maneira *offline* dos dados coletados nos módulos de animais e seus submódulos correspondentes, sincronização dos dados entre aplicativo e sistema *Application Programming Interface* (Interface de Programação de Aplicação) (API).

Palavras-chave: balanceamento nutricional; produção leiteira; bovinocultura de leite; sistema offline; aplicativo móvel.

ABSTRACT

Even with the state of Paraná being one of the largest responsible for national production of milk, it is still possible to note the lack of management of the producer about his herd, especially when it comes to food, because even if it is of quality, it may not be adequate to the nutritional needs of the animals. These activities can be optimized by means of a computer system, which can assist in the treatment and storage of data related to the herd and feed, allowing the herd management to be performed be performed more efficiently, because the consultation of data in an orderly manner can the decision making process. The system proposed and presented in this project, aims to assist both the technicians of the IDR and producers in the management of animals, nutritional management, monitoring of cattle in rural properties and management information of the properties. The main functionalities developed were the offline storage and management of data collected in the animal modules and their modules and their corresponding sub-modules, synchronization of data between application and API system.

Keywords: nutritional balance; milk production; milk-cattle; offline system; mobile application.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

Siglas

API	*Application Programming Interface* (Interface de Programação de Aplicação)
APK	*Android Package Kit* (Kit de pacote Android)
CMS	Consumo de Matéria Seca
CNCPS	*Cornell Net Carbohydrate and Protein System*
CRUD	*Create, Read, Update, Delete*
HTTP	*Hypertext Transfer Protocol*
IDR	Instituto de Desenvolvimento Rural
JSON	*JavaScript Object Notation*
NRC	*National Research Council*
REST	*Representational State Transfer*

SUMÁRIO

1 INTRODUÇÃO

Segundo a Confederação da Agricultura e Pecuária do Brasil (CNA), o setor do agronegócio no ano de 2021 alcançou a marca de 27,4% de participação no Produto Interno Bruto (PIB) brasileiro, com crescimento de 8,36%. Maior participação desde 2004 (27,53%), mesmo ficando abaixo da estimativa, de 9,37% (CNA, 2022).

No ano de 2017, a produção brasileira de leite, ultrapassou 30 bilhões de litros, sendo o sul e sudeste as principais regiões produtoras, com 35,7% e 34,2% do total de litros produzidos, respectivamente (IBGE, 2018).

Segundo o Ministério da Agricultura, a Agricultura Familiar, empregando mais de 10 milhões de pessoas (MDA, 2019), é o principal setor responsável pela produção e cultivo dos alimentos que são consumidos pela população brasileira.

Dentre os alimentos, destaca-se a produção de leite, que tem importância econômica e social e está presente em número considerável de propriedades com mão de obra familiar.

Como descrito por Corrêa, Veloso e Barczsz (2010) e Souza, Amin e Gomes (2009), desde o início da década de 90, a atividade leiteira vem enfrentando várias transições, com o objetivo de adequar-se e manter-se no mercado global de produção, o qual vem se tornando cada vez mais inovador e competitivo, almejando uma produção em larga escala, a industrialização de produtos diferenciados, mas prezando sempre por produtos de qualidade e que agreguem valor comercial.

Em função dos avanços tecnológicos, sabe-se que o setor leiteiro vem vivenciando um processo de modernização intensa, para adequação às novas alternativas apresentadas nos sistemas de produção, visando o melhoramento da propriedade, otimização do tempo, da produção e da qualidade dos produtos, contribuindo, consequentemente para a melhoria da vida no campo (BOTEGA *et al.*, 2007).

Na busca por competitividade, algumas empresas procuram investir na melhoria da qualidade do leite, na consolidação da imagem dos lácteos junto ao mercado e na abertura de mercados à exportação e na inovação e aperfeiçoamento tecnológico (VILELA *et al.*, 2016).

Ainda segundo a Vilela *et al.* (2016), o baixo grau de instrução destes produtores é uma grande barreira para a introdução dos conceitos como registro das receitas e despesas, ou mesmo de controle zootécnico, assim dificultando, inclusive, a utilização de ferramentas simples de coleta de informações.

O gerenciamento do empreendimento, em seus aspectos econômico-financeiros e técnicos, como sanidade, manejo de alimentação influenciam nos resultados da produção de leite (MARTINS *et al.*, 2007). Visando o melhoramento e resolução desse problema nas propriedades rurais do Paraná, o IDR busca, por meio de recolhimento de dados em visitas as propriedades de pequenos agricultores, fazer o gerenciamento e acompanhamento dos animais. Coletando dados de: produção de leite, gestação e nascimento, peso, quantidade, produção e tipos de alimentos fornecidos, entre outras informações, e armazenando-os em planilhas.

Essa atividade pode ter o recolhimento, armazenamento e administração dos dados otimizadas se utilizado um sistema de informação. Assim os técnicos e proprietários serão capazes de realizar o gerenciamento do rebanho de forma mais eficaz, auxiliando na tomada de decisões.

Pensando na otimização do tempo e da carga de trabalho este trabalho propõe o desenvolvimento de um aplicativo para dispositivos móveis para auxiliar na gestão dos animais, gestão nutricional e acompanhamento do gado leiteiro nas propriedades rurais, para auxiliar tanto os técnicos do IDR quanto os produtores.

O aplicativo móvel desenvolvido neste trabalho tem como objetivo funcionar em conjunto com um sistema web e uma API *Representational State Transfer* (REST) que fazem parte do mesmo projeto. Embora esses outros sistemas estejam sendo desenvolvidos em paralelo com o trabalho atual, eles estão fora do escopo abordado nesta produção. No entanto, é importante mencionar a existência desses sistemas complementares, pois o aplicativo móvel dependerá da interação com a API REST para acessar os dados necessários e compartilhar os dados com o sistema web.

A API REST está sendo desenvolvida para fornecer *endpoints* que permitirão ao aplicativo móvel acessar e manipular os dados necessários para seu funcionamento. Essa API seguirá os princípios do estilo arquitetural REST (RIBEIRO; FRANCISCO, 2016), permitindo uma comunicação eficiente e padronizada entre o aplicativo móvel e o sistema web. O sistema web, por sua vez, está sendo projetado para oferecer uma interface de usuário amigável e funcionalidades adicionais que não serão abordadas neste trabalho acadêmico.

Embora esses sistemas estejam fora do escopo do trabalho desenvolvido, sua existência e desenvolvimento paralelo são fundamentais para o sucesso e funcionamento adequado do aplicativo móvel. A integração entre o aplicativo, a API REST e o sistema web garantirá uma experiência de usuário completa e coesa, permitindo que os usuários acessem e gerenciem os recursos fornecidos por meio da aplicação móvel de maneira eficiente e intuitiva.

1.1 Objetivos

1.1.1 Objetivo geral

Desenvolver um aplicativo para dispositivos móveis a fim de realizar o controle da gestão e acompanhamento do gado leiteiro nas propriedades rurais.

1.1.2 Objetivos específicos

- Permitir a coleta de dados do gado da propriedade.
- Permitir a manutenção dos dados do gado da propriedade.

- Permitir o registro de identificações de doenças e pragas nas plantações da propriedade.
- Permitir a gestão dos dados de forma *offline*.
- Permitir a sincronização dos dados entre o sistema (API) e o aplicativo.

1.2 Justificativa

O Paraná, em 2020, foi responsável por, aproximadamente, 4,6 bilhões de litros, equivalentes a 13% da produção nacional, o que lhe concede o *ranking* de terceiro maior produtor de leite do Brasil, segundo dados do CNA (2021). Em produtividade, o Paraná também fica em terceiro lugar, com média de 3.490 litros/vaca/ano, ficando atrás de Santa Catarina, ordenhando 3.716 litros/vaca/ano e Rio Grande do Sul, com média de 3.695 litros/vaca/ano.

Mesmo com bons indicadores de produtividade do estado do Paraná, e região sul em geral, que podem ser explicados pelas características climáticas próprias da região, e permitem a criação de rebanhos altamente especializados na produção leiteira, ainda nota-se a falta do manejo do ambiente em que este rebanho está introduzido.

O acompanhamento zootécnico, a manutenção dos rebanhos e pastagens e, principalmente, melhoria no manejo nutricional dos animais são procedimentos importantes, porém, muitas vezes, esquecidos ou ignorados pelos proprietários, por serem consideradas atividades rotineiras.

Como citado anteriormente, o IDR vem fazendo um trabalho de acompanhamento e gestão dos dados de produção leiteira das propriedades e seus animais coletando os dados em planilhas. Contudo, mesmo com os esforços dos técnicos, a análise das informações coletadas se torna muito lenta, consequentemente, demorando na emissão dos gráficos e relatórios. Existe ainda a eventualidade de ocorrer erros no lançamento dos dados, mesmo com os cuidados dos técnicos, seja pela falta de validação na inserção ou complexidade do ambiente de trabalho (planilhas), o que afetaria as próximas etapas do serviço de análise. Além do retrabalho de lançar esses dados para outras plataformas se necessário.

Diante disso, vê-se a necessidade da elaboração de um sistema de aplicativo móvel para administração destes dados, possibilitando a inserção das informações sobre o rebanho e a propriedade, realizando o balanceamento nutricional, que pode ser individual, com base nos alimentos que constituirão a dieta dos animais, facilitando assim a organização e armazenamento das informações, permitindo o acompanhamento com base nos dados coletados, inclusive em diferentes plataformas.

1.3 Estrutura do trabalho

Este texto está organizado em capítulos, este é o primeiro e apresenta a introdução, as considerações iniciais, os objetivos e a justificativa da execução deste trabalho. O Capítulo 2 apresenta o referencial teórico. O Capítulo 3 apresenta os materiais e o método utilizado no desenvolvimento. No Capítulo 4 é apresentado o resultado do aplicativo elaborado, juntamente com o escopo pensado, modelagem e o elaboração deste trabalho. Por fim, está a conclusão do trabalho e as referências utilizadas para a desenvolvimento.

2 REFERENCIAL TEÓRICO

Na produção leiteira, visando a obtenção de maior qualidade e produtividade do leite, com um custo menor, a nutrição do animal é um dos principais e mais importantes fatores a serem considerados (TOMICH *et al.*, 2015).

Principalmente na bovinocultura de leite, o Consumo de Matéria Seca (CMS) afeta diretamente fatores produtivos, em que é imprescindível atingir níveis notáveis de reprodução e produção (ZANIN; HENRIQUE; FLUCK, 2017). O CMS pode refletir entre 60 a 90% das variações na performance dos bovinos e apenas de 10 a 40% do restante são referentes a variações na qualidade substancial dos alimentos (ZANIN; HENRIQUE; FLUCK, 2017 apud MERTENS, 1987).

Assim, existem modelos que podem predizer o CMS e tornar a produção mais sustentável. Estes modelos são normas e padrões definidos por cada país, conforme suas características (TOMICH *et al.*, 2015). Para bovinos de leite, alguns modelos utilizados são *Cornell Net Carbohydrate and Protein System* (CNCPS) e *National Research Council* (NRC). Ambos são modelos norte-americanos, que empregam informações de regiões de clima temperado, alimentos característicos do sistema de produção norte-americano e animais da raça holandesa. Ao mesmo tempo que o CNCPS calcula as exigências a partir da massa do animal e da produção de leite, o NRC utiliza também dados sobre a composição do leite e a semana de lactação (ZANIN; HENRIQUE; FLUCK, 2017).

Para poder atender as premissas nutricionais do animal é necessário uma oferta apropriada de nutrientes em sua dieta (BETT, 2022 apud ARRIGONI *et al.*, 2013), assim favorecendo o aspecto do potencial genético em todas as fases da vida deste animal.

A fim de realizar o balanceamento nutricional para esses animais são necessárias algumas etapas, definidas por Salman, Osmari e Santos (2011):

1. Distinguir os animais em que se deseja balancear a ração;
2. Apurar os requisitos dos nutrientes dos animais de acordo com o primeiro item;
3. Levantar e estimar os alimentos disponíveis;
4. Associar a composição química e o valor energético dos alimentos a serem utilizados, considerando os nutrientes de relevância;
5. Proceder ao balanceamento da ração para a proteína bruta e energia;
6. Depois de concluído o cálculo da ração, examinar se todas as exigências foram atendidas.

Buscando suprir a exigência do animal, é necessário selecionar os alimentos de acordo com seu valor nutritivo para o mesmo. Assim, cada alimento deve conter sua composição química, que designará a sua quantidade no balanceamento Tomich *et al.* (2015).

Alguns dos principais métodos práticos para formular as rações, segundo Salman *et al.* (2020), são:

- Método algébrico, que possibilita a combinação de dois ou mais elementos e consiste em estruturar um sistema de equações simultâneas, sendo as incógnitas os ingredientes que serão manipulados na ração. Esta técnica torna-se progressivamente mais complexo à medida que se aumenta o número de nutrientes e ingredientes considerados.

- Método do Quadrado de Pearson. Considerado simples, simples e concede o cálculo das proporções de dois ingredientes de uma mistura, com o objetivo de satisfazer um nível de nutriente pretendido, via de regra a proteína. Nessa técnica, é possível ser utilizado dois alimentos ou grupos de alimentos misturados previamente.

3 MATERIAIS E MÉTODO

Este capítulo apresenta os materiais e o método utilizado para a realização deste trabalho. Os materiais estão relacionados às ferramentas e tecnologias utilizadas e o método apresenta a sequência das principais atividades realizadas.

3.1 Materiais

As tecnologias que foram utilizadas no desenvolvimento do trabalho estão relacionadas no Tabela 1.

Tabela 1 – Materiais utilizados no desenvolvimento do sistema

Ferramenta/Tecnologia	Versão	Disponível em	Finalidade
Java	17	https://www.java.com/pt-BR/	Linguagem de Desenvolvimento
Intellij Idea Ultimate	2022.2.3	https://www.jetbrains.com/idea	IDE de Desenvolvimento
Visual Studio Code	1.72	https://code.visualstudio.com	Editor de Código
Flutter	3.3.6	https://docs.flutter.dev	Framework Para Desenvolvimento Híbrido
Dart	2.18.3	https://dart.dev/	Linguagem de Desenvolvimento
PostgreSQL	14.5	https://www.postgresql.org/	Banco de Dados Relacional
Spring Framework	2.7.5	https://spring.io/projects/spring-boot	Framework para Desenvolvimento
Hive	2.2.3	https://pub.dev/packages/hive	Banco de Dados offline
Getx	4.6.5	https://pub.dev/packages/get	Gerenciamento de estado, controle de injeção de dependência e gerenciamento de rotas
LucidChart		https://www.lucidchart.com/pages/pt	Diagramação e Modelagem do Sistema
Figma		https://www.figma.com/	Design e Prototipação
Planilhas eletrônicas disponibilizadas pelo IDR			Estudo do projeto e levantamento dos requisitos

Fonte: Autoria própria (2023).

Nos próximos parágrafos, serão comentadas as ferramentas e os recursos tecnológicos que desempenharam um papel fundamental na concepção e implementação deste projeto, a utilização de cada uma dessas tecnologias contribuiu para alcançar os objetivos propostos.

O Spring Framework foi escolhido para desenvolver o *back-end* da aplicação pois permite maior simplicidade e produtividade no desenvolvimento de aplicações Java, pois já oferece grande parte dos recursos e configurações iniciais necessárias prontas para utilização, permitindo que os desenvolvedores concentrem-se na parte lógica da aplicação e também por contribuir com projetos mais ágeis e organizados. Além disso, a API a ser utilizada pelo projeto do sistema WEB do IDR será reaproveitada para este sistema *mobile*.

Foi optado por utilizar o Flutter juntamente com Dart para o desenvolvimento da aplicação móvel por se tratar de um *framework* de desenvolvimento híbrido/cruzado, código aberto e gratuito, com performance nativa, assim podendo ser utilizada a mesma base de códigos para interfaces iOS e Android, tornando o desenvolvimento mais ágil sem a necessidade de desenvolver diferentes versões das aplicações.

O PostgreSQL foi escolhido por ser um banco de dados relacional, apropriado para o formato dos dados que serão coletados, e ser de uso livre, ou seja, gratuito.

O Hive foi escolhido para como banco de dados local na aplicação para dispositivos móveis, por ser leve e consumir poucos recursos do dispositivo. Os resultados do seu *benckmark* pode ser visualizados na Figura 1. Os gráficos apresentados indicam o desempenho superior do Hive sobre a manipulação de leitura e gravação em 1000 iterações de dados sobre outros 2 pacotes com o mesmo objetivo, o SharedPreferences e o SQLite,

Figura 1 – *Benckmark* Hive

Fonte: Autoria própria (2023).

3.2 Método

Para o desenvolvimento deste projeto o método utilizado foi a abordagem da metodologia ágil (PONTES; ARTHAUD, 2019), no qual houve recolhimento de *feedbacks* durante o processo de desenvolvimento buscando realizar melhorias em cima dos mesmos.

De início, para o levantamento dos requisitos, foram ouvidos e lidos os documentos disponibilizados pelos técnicos do IDR Paraná, já que os mesmos atuam diretamente na área. Deste modo, entendendo os procedimentos rotineiros indispensáveis do sistema.

Os técnicos do IDR Paraná atualmente utilizam aplicações de criação de planilhas eletrônicas para realizar o gerenciamento dos dados das propriedades que os fazem o acompanhamento. Com base nestas planilhas é possível realizar uma análise e levantamento dos dados necessários para o sistema e também como funciona a interação dos mesmos.

Em seguida, foram definidos os requisitos funcionais e não funcionais do sistema, com base nos documentos obtidos e conversas com os técnicos do IDR, assim como o desenvolvimento dos casos de uso.

Na etapa seguinte foi realizada a prototipação do sistema, juntamente com suas interfaces com o objetivo de entender melhor o que seria desenvolvido, eliminando o que for desnecessário e permitir testar a experiência do usuário no sistema.

Em sequencia, foi iniciado o processo de desenvolvimento dos códigos-fonte da aplicação. Em paralelo a isso foram realizados os testes do sistema, buscando minimizar e corrigir os possíveis erros durante o tempo de desenvolvimento. Os testes e validação da interface foram feitos em conjunto com o professor orientador.

4 RESULTADOS

Nesta seção serão apresentados os resultados do desenvolvimento de um aplicativo para dispositivos móveis para controle da gestão e acompanhamento dos animais nas propriedades rurais com sincronização para controlar, de maneira *offline* os dados da aplicação.

Inicialmente é apresentado o escopo que relata as principais funcionalidades e atores envolvidos, seguido da modelagem do sistema que estabelece a definição dos requisitos funcionais e não-funcionais, diagrama dos casos de uso e de entidade e relacionamento do banco de dados.

4.1 Escopo do sistema

O aplicativo para dispositivos móveis desenvolvido neste trabalho será utilizado pelos técnicos do IDR. A principal finalidade é gerenciar os dados relacionados aos animais de propriedades rurais. O aplicativo irá consumir dados de uma API REST que está sendo desenvolvida em paralelo à este trabalho, por isso alguns dados que serão utilizados pelo aplicativo já deverão estar pré-cadastrados na API.

Alguns dados utilizados pelo aplicativo serão cadastrados exclusivamente na API, sendo eles as propriedades rurais e os técnicos, que serão os usuários do sistema. Uma importante regra de negócio que deverá ser observada é que cada propriedade rural estará vinculada a um ou mais técnicos e, esses técnicos só poderão acessar as propriedades com as quais possui vínculo.

O aplicativo permite ao usuário realizar o controle dos dados de rebanho de uma propriedade rural de forma prática e rápida e o principal, de maneira *offline*. Assim possibilitando o controle, acompanhamento e evolução dos animais dessa propriedade rural mesmo sem conexão à Internet.

O sistema inicia pela tela de autenticação, em que somente usuários pré cadastrados na API terão acesso às informações. Os técnicos poderão inserir, consultar, remover e editar os dados dos submódulos relacionados às propriedades que ele possui acesso.

Na área de inserção de informações, o sistema deve permitir o cadastro dos dados necessários do rebanho, como peso, data de nascimento, número de identificação, raça do animal, etc. O sistema também deverá permitir a avaliação e acompanhamento da evolução dos animais. Também, irá permitir realizar o cadastro de ocorrências identificadas com os animais da propriedade como inseminações, identificação de doenças, uso de medicamentos e identificação de mastite. Além dos módulos destinados aos animais, o sistema permite a identificação de pragas e doenças nas plantações da propriedade. Todos os registros estão sempre relacionados à uma propriedade rural. O sistema permite controlar os dados de várias propriedades rurais.

Além disso, o usuário pode cadastrar eventos que acontecem com o rebanho, tais como, o nascimento ou morte de animais e também a situação de compra e venda.

Todos os dados cadastrados são armazenados de modo *offline*, permitindo aos usuários do sistema o deslocamento a locais sem acesso à Internet.

Quando o dispositivo possui acesso à Internet, o sistema permite realizar a sincronização dos dados recolhidos de maneira *offline* para garantir a persistência destes dados no sistema *online* (API). A sincronização é importante pois vai permitir aos administradores emitirem os relatórios gerenciais baseados em todos os dados coletados nos sistemas.

4.2 Modelagem do sistema

Esta seção apresenta os requisitos funcionais e não funcionais, casos de uso e os diagramas usados para detalhar os processos e a estrutura do sistema a desenvolvido. Cada requisito funcional é apresentado seguido dos requisitos não funcionais relacionados.

Os quadros 1, 2, 3, 4 e 5 apresentam os requisitos funcionais e não funcionais respectivamente.

4.2.1 Requisitos funcionais e não funcionais

No Quadro 1 é apresentado o requisito funcional Autenticação do usuário, que é um requisito para acessar os demais módulos do sistema. Nos seu requisitos não funcionais é descrito os dados utilizados para realizar a autenticação (*login*).

Quadro 1 – Autenticação do usuário

F1 Autenticação do usuário	
Descrição: O sistema deve permitir a realização da autenticação do usuário previamente cadastrado;	
Requisitos Não-Funcionais	
Nome	**Restrição**
NF1.1 Controle de acesso ao sistema	O acesso ao sistema será realizado por meio de email e senha.
NF1.2 Controle de acesso aos módulos	O acesso aos módulos do sistema deve ser permitido somente caso o usuário já esteja autenticado (*logado*) no sistema.

Fonte: Autoria própria (2023).

Com o requisito funcional apresentado no Quadro 2 é descrito como serão abordados os cadastros de animais de uma determinada propriedade, com a descrição dos seus campos, demais funções e módulos do sistema que podem ser acessados a partir do registro de deter-

minado animal, juntamente com os quesitos de segurança e validações que são necessárias para acessar este recurso.

Quadro 2 – Manter animais

F2 Manter animais	
Descrição: Inserção e edição dos animais da propriedade, seja por nascimento, compra ou levantamento dos animais já existentes na propriedade. Os campos do cadastro serão: data de nascimento, peso nascido, peso atual, peso previsto, raça, identificação (número do brinco), nome, gênero, ecc (escore de condição corporal). Manutenção do que envolve o animal, incluindo: parto, inseminação, identificação de mastite, identificação de doenças, aplicação medicamentos, reprodução (prenhez), mortes, registro de venda, registro de compra.	
Requisitos Não-Funcionais	
Nome	**Restrição**
NF2.1 Controle de acesso aos registros de animais	O acesso aos registros de animais só poderá ser permitido após a autenticação do usuário no sistema e seleção de uma propriedade.
NF2.2 Controle de acesso aos módulos de manutenção.	Não permitir registrar manutenção em um animal com uma data de morte registrada (identificado como morto) ou uma venda registrada.
NF2.3 Controle de registro vinculado a propriedade	Para realizar alguma das operações CRUD referentes ao módulo de "animais" e seus módulos de manutenção é necessário o usuário ter uma propriedade previamente selecionada.
NF2.4 Controle de mastite	O sistema deve realizar o controle do histórico de mastites diagnosticadas nos animais, assim como possibilitar o registro de novos diagnósticos, inserir a data do diagnóstico, tipo de mastite identificada e o resultado do teste CMT.
NF2.5 Controle de inseminações nos animais	O sistema deve realizar o controle do histórico de inseminações realizadas nos animais, assim como possibilitar o registro de novas inseminações feitas, permitindo inserir a identificação do touro e a data da inseminação.
NF2.6 Controle de doenças dos animais	O sistema deve realizar o controle do histórico de doenças e diagnósticos registrados nos animais, assim como possibilitar o registro de novos diagnósticos, permitindo inserir a descrição do diagnóstico feito e a data do diagnóstico.
NF2.7 Controle de medicamentos dos animais	O sistema deve realizar o controle das aplicações de medicamentos nos animais, assim como possibilitar o registro de novas aplicações, permitindo inserir um vínculo de produto (remédio previamente cadastrado) a ser utilizado na aplicação, a forma da aplicação, a dose e o princípio ativo.
NF2.8 Controle de prenhez dos animais	O sistema deve realizar o controle do histórico de prenhez diagnosticadas nos animais, assim como possibilitar o registro de novos diagnósticos identificados, permitindo inserir as datas de diagnóstico e data do último diagnóstico de prenhez identificado.
NF2.9 Controle de venda dos animais	O sistema deve permitir inserir um registro de venda, com os campos a data de venda, motivo da venda, peso do animal na venda, valor recebido, destino do animal.
NF2.10 Controle de compra dos animais	O sistema deve permitir inserir um registro de compra, com os campos de data da compra, data de nascimento, peso do animal na compra, valor pago.

Fonte: Autoria própria (2023).

A sincronização dos dados entre o aplicativo para dispositivos móveis e a API tem como base o requisito representado no Quadro 3. Inicialmente será realizada uma carga inicial dos dados que estão na API de acordo com os dados do usuário autenticado. Após o uso do aplicativo pelo usuário os dados podem ter sido modificados ou dados podem ter sido inseridos. Então, para realizar a sincronização com a API deve-se enviar os dados armazenados de maneira *offline* no aplicativo. Após a sincronização dos dados do aplicativo é realizada novamente a carga dos dados da API, permitindo o usuário trabalhar com os dados sempre atualizados no aplicativo.

Quadro 3 – Sincronização

F3 Sincronização de dados.	
Descrição: O sistema deve realizar a sincronização dos dados. Possibilitando buscar todos os dados necessários para o funcionamento do sistema *mobile*, e também realizar o envio dos dados registrados de *offline* e buscando novos dados necessários que foram inseridos por outra plataforma.	
Requisitos Não-Funcionais	
Nome	**Restrição**
NF3.1 Controle dos dados já sincronizados	O sistema deve realizar o controle dos dados já sincronizados, com o objetivo de não perder ou duplicar dados, tanto *offline* quanto *online*.
NF3.2 Controle de acesso	É necessário o usuário estar autenticado no sistema para conseguir realizar a sincronização dos dados.

Fonte: Autoria própria (2023).

No requisito sobre Controle de pragas vegetais são explicados os métodos para controle de identificação e registro das pragas, como pode ser visto no Quadro 4.

Quadro 4 – Controle de pragas

F4 Controle das pragas vegetais	
Descrição: Inserção e edição das doenças vegetais identificadas na propriedade. Permitindo inserir a data da identificação, tipo da infestação, da cultura e da praga . Também realizar o controle do histórico de pragas identificadas na propriedade.	
Requisitos Não-Funcionais	
Nome	**Restrição**
NF4.1 Controle de acesso	Para realizar alguma das operações CRUD referentes ao módulo de "pragas vegetais" é necessário o usuário estar autenticado no sistema.
NF4.2 Controle de registro vinculado a propriedade	Para realizar alguma das operações CRUD referentes ao módulo de "pragas vegetais" é necessário o usuário ter uma propriedade previamente selecionada.

Fonte: Autoria própria (2023).

O requisito funcional Controle de doenças vegetais da propriedade retrata a maneira de salvar e controlar os dados de doenças identificadas na propriedade, conforme pode ser observado no Quadro 5.

Quadro 5 – Controle de doenças

F5 Controle das doenças vegetais	
Descrição: Inserção e edição das doenças vegetais identificadas na propriedade. Permitindo inserir a data da identificação, tipo da infestação, da cultura e da doença. Também realizar controle do histórico de pragas identificadas na propriedade.	
Requisitos Não-Funcionais	
Nome	**Restrição**
NF5.1 Controle de acesso	Para realizar alguma das operações CRUD referentes ao módulo de "doenças vegetais" é necessário o usuário estar autenticado no sistema.
NF5.2 Controle de registro vinculado a propriedade	Para realizar alguma das operações CRUD referentes ao módulo de "doenças vegetais" é necessário o usuário ter uma propriedade previamente selecionada.

Fonte: Autoria própria (2023).

4.2.2 Casos de uso

No Tabela 2 são listados os casos de uso, os atores e os requisitos funcionais que são referenciados em cada um dos casos. O ator do sistema *mobile* se limita para apenas o técnico rural (responsável pela visita a propriedade).

Tabela 2 – Casos de uso

ID	**Nome do caso de uso**	**Atores**	**Referências Cruzadas**
UC1	Adicionar um novo registro (CRUD).	Técnico.	F1, F2, F4, F5
UC2	Editar registro (CRUD).	Técnico.	F1, F2, F4, F5
UC3	Apagar registro (CRUD).	Técnico.	F1, F2, F4, F5
UC4	Sincronizar dados com o serviço *online*.	Técnico.	F1, F3
UC5	Enviar dados obtidos de maneira *offline*.	Técnico.	F1, F3

Fonte: Autoria própria (2023).

A Figura 2 apresenta o diagrama de casos de uso do sistema, exibindo o ator e as conexões com casos de uso.

Figura 2 – Diagrama de casos de uso

Fonte: Autoria própria (2023).

4.2.3 Casos de uso expandidos

Nos quadros 6, 7, 8, 9 e 10 são descritos os casos de uso do software de maneira expandida. Com seus autores, pré e pós-condições, seus eventos e exceções.

No Quadro 6 é descrito o caso de uso sobre o cadastro de um novo registro em algum dos módulos da aplicação. Para a inserção de um novo registro é indispensável o usuário estar autenticado na aplicação e ter uma propriedade pré-selecionada. Cada um dos registro possuem campos com preenchimento obrigatório e não obrigatório, a validação de cada um deles é feito dentro do próprio módulo.

Quadro 6 – Adicionar um novo registro

Caso de Uso: UC1 - Adicionar um novo registro
Atores: Técnico.
Precondições: Estar autenticado. Ter uma propriedade previamente selecionada. Sem impedimentos de acordo com as regras do sistema (EX: registrar uma mastite para um animal com registro de venda).
Pós-condições: Novo registro adicionado.
Sequência típica de eventos (Fluxo Principal): Esse caso de uso inicia quando: 1. [IN] Ator seleciona a página relacionada ao cadastro do novo registro. 2. [IN] Ator insere os dados do novo registro 3. [IN] Ator clica em salvar o novo registro 4. [OUT] Sistema guarda os dados no banco e confirma a inserção.
Tratamento de Exceções e Variantes: Exceção: 1. Dados não preenchidos. a. [OUT] O sistema verifica quais os campos não foram preenchidos. b. [OUT] O sistema sinaliza os campos não preenchidos. c. [IN] Ator preenche os campos. d. [IN] Ator clica em salvar o registro.

Fonte: Autoria própria (2023).

O Quadro 7 descreve o caso de uso relacionado à edição de um registro já salvo *offline* na aplicação.

Quadro 7 – Editar registro

Caso de Uso: UC2 - Editar registro
Atores: Técnico.
Precondições: Estar autenticado.
Pós-condições: Sucesso ao editar registro.
Sequência típica de eventos (Fluxo Principal): Esse caso de uso inicia quando: 1. [IN] Ator seleciona o registro que pretende editar. 2. [OUT] Sistema abre a tela de edição com os dados do registro. 3. [IN] Ator altera os dados do registro. 4. [IN] Ator clica em salvar o registro. 5. [OUT] Sistema guarda os dados no banco e confirma a edição.
Tratamento de Exceções e Variantes: Exceção: 1. Dados não preenchidos. a. [OUT] O sistema verifica quais os campos não foram preenchidos. b. [OUT] O sistema sinaliza os campos não preenchidos. c. [IN] Ator preenche os campos. d. [IN] Ator clica em salvar o registro.

Fonte: Autoria própria (2023).

No Quadro 8 é retratado o caso de uso de remoção de um registro salvo *offline* na aplicação.

Quadro 8 – Apagar registro

Caso de Uso: UC3 - Apagar registro.
Atores: Técnico.
Precondições: Estar autenticado.
Pós-condições: Sucesso ao apagar registro.
Sequência típica de eventos (Fluxo Principal): Esse caso de uso inicia quando: 1. [IN] Ator seleciona o registro que pretende apagar. 2. [OUT] Sistema pede a confirmação da exclusão do registro. 3. [IN] Ator confirma a exclusão. 4. [OUT] Sistema apaga registro do banco e confirma a exclusão.
Tratamento de Exceções e Variantes: Exceção: 1. Registro vinculado a outros registros. a. [OUT] O sistema identifica que o registro possui outros registros vinculados e não permite a exclusão.

Fonte: Autoria própria (2023).

No Quadro 9 é apresentado o caso de uso da sincronização de dados da aplicação, descrevendo os eventos interpretados da Figura 3 sobre a busca dos dados.

Quadro 9 – Sincronizar dados com o serviço *online*

Caso de Uso: UC4 - Sincronizar dados com o serviço online.
Atores: Técnico, sistema.
Precondições: Estar autenticado. Ter acesso a internet.
Pós-condições: Sucesso ao sincronizar dados. Exceção ao sincronizar dados.
Sequência típica de eventos (Fluxo Principal): Esse caso de uso inicia quando: 1. [IN] Ator seleciona a opção de sincronização. 2. [OUT] Sistema identifica se possui internet. 3. [OUT] Sistema inicia o fluxo da sincronização pelo grupo de dados inicial a ser sincronizado. 4. [OUT] Sistema realiza a conexão e busca os dados online. 5. [OUT] Sistema identifica os dados buscados e os insere no banco offline. 6. [OUT] Sistema confirma a sincronização. 7. [OUT] Sistema parte para próximo grupo de dados inicial a ser sincronizado (volta ao passo 4). 8. [OUT] Sistema verifica que não há próximo grupo de dados e finaliza a operação
Tratamento de Exceções e Variantes: Exceção: 1. Sem acesso a internet. a. [OUT] Sistema identifica que não possui acesso a internet b. [OUT] Retorna aviso para o usuário. 2. Perda de conexão. a. [OUT] Sistema identifica perda de conexão b. [OUT] Retorna aviso para o usuário. 3. Exceção ao exportar dados. a. [OUT] Identifica alguma exceção no *backend* ou *mobile*. b. [OUT] Retorna aviso para o usuário. c. [OUT] Sistema passa para próximo grupo de dados a ser sincronizado.

Fonte: Autoria própria (2023).

O Quadro 10 expõe o caso de uso de envio dos dados que foram obtidos e salvos *offline* na aplicação, descrevendo os dados interpretados da Figura 3.

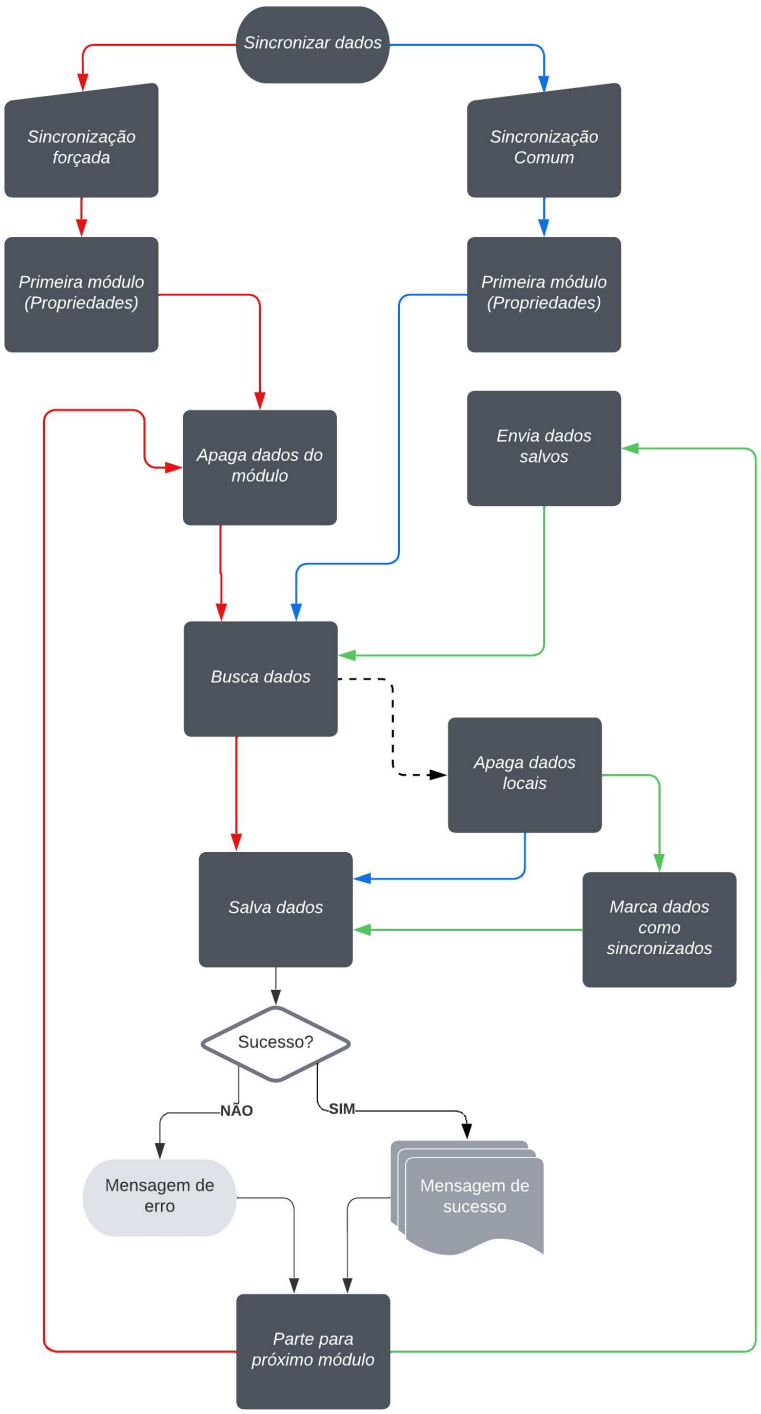
Quadro 10 – Enviar dados *offline* obtidos

Caso de Uso: UC5 - Enviar dados offline obtidos.
Atores: Técnico.
Precondições: Estar autenticado. Ter acesso a internet.
Pós-condições: Sucesso ao enviar dados.
Sequência típica de eventos (Fluxo Principal): Esse caso de uso inicia quando: 1. [IN] Ator seleciona a opção de envio de dados. 2. [OUT] Sistema identifica se possui internet. 3. [OUT] Sistema inicia o fluxo da sincronização pelo grupo de dados inicial a ser sincronizado. 4. [OUT] Sistema realiza a conexão e envia os dados online. 5. [OUT] Sistema confirma o envio do grupo de dados. 6. [OUT] Sistema apaga os dados *offline* do grupo de dados. 7. [OUT] Sistema realiza a busca dos dados do grupo de dados online atualizados. 8. [OUT] Sistema identifica os dados buscados e os insere no banco offline. 9. [OUT] Sistema confirma a sincronização. 10. [OUT] Sistema parte para próximo grupo de dados inicial a ser sincronizado (volta ao passo 4). 11. [OUT] Sistema verifica que não há próximo grupo de dados e finaliza a operação
Tratamento de Exceções e Variantes: Exceção: 1. Sem acesso a internet. a. [OUT] Sistema identifica que não possui acesso a internet b. [OUT] Retorna aviso para o usuário. 2. Perda de conexão. a. [OUT] Sistema identifica perda de conexão b. [OUT] Retorna aviso para o usuário. 3. Exceção ao exportar dados. a. [OUT] Identifica alguma exceção no *backend* ou *mobile*. b. [OUT] Retorna aviso para o usuário. c. [OUT] Sistema passa para próximo grupo de dados a ser sincronizado.

Fonte: Autoria própria (2023).

A Figura 3 exibe o fluxograma da sincronização dos dados. Nessa figura ser observado que o aplicativo possui duas maneira de sincronização, a sincronização forçada e a sincronização comum. A sincronização forçada tem como objetivo fazer o carregamento inicial de dados, buscando os dados *online* (API) e adicionando os dados na aplicação para dispositivos móveis. A sincronização comum tem o propósito de enviar os dados já coletados e mantidos *offline* na aplicação, após a confirmação do envio os dados são novamente carregados da API para a aplicação, para manter os dados sempre atualizados para o usuário do aplicativo.

Figura 3 – Fluxograma sincronização dos dados



Fonte: Autoria própria (2023).

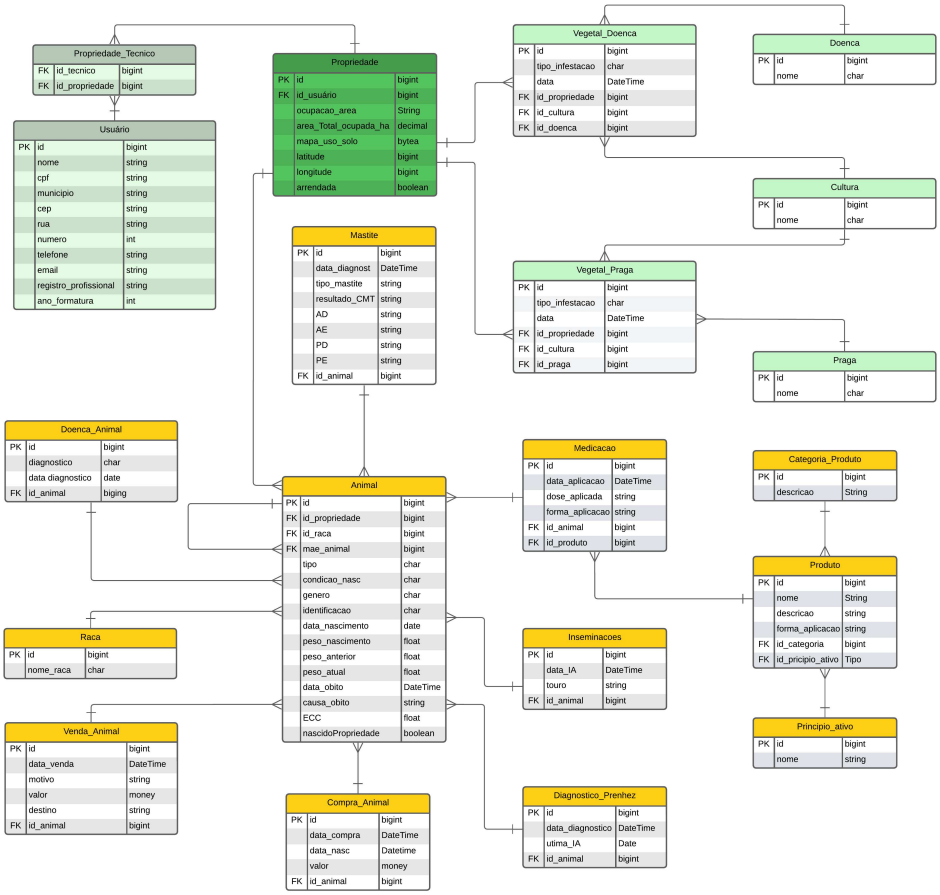
4.2.4 Diagrama de Entidade e Relacionamento

A Figura 4 representa o diagrama de entidades e relacionamentos das entidades utilizadas no desenvolvimento deste trabalho. A aplicação desenvolvida neste trabalho faz parte de um projeto mais amplo na área de balanceamento de nutrição do gado leiteiro e com outras funcionalidades, que possui um sistema web que utiliza-se de outras entidades para o seu desenvolvimento, as quais não estão apresentadas no diagrama da Figura 4. As principais entidades do diagrama apresentado são:

- Usuário: Responsável por armazenar os dados dos usuário, suas permissões e os dados para realização da autenticação no sistema.
- Propriedade: Encarregado de recolher os dados das propriedades como localização, ocupação e usuário que possui acesso a estes dados. A partir dela podem ser acessados outros módulos dentro do sistema.
- Animal: Responsável por manter o registro de animais de um determinada propriedade, aonde a partir desta entidade são registrados as manutenções e registros de outros módulos/entidades dentro do sistema.

Além das entidades citadas anteriormente a entidade mastite é responsável por armazenar a identificação de mastites nos animais, a entidade doença_animal tem como objetivo manter as doenças constatadas nos animais, a entidade medicação tem como propósito guardar o uso de medicamento pelos animais, a entidade inseminação mantém as inseminações feitas nos animais, a entidade venda_animal armazena as vendas de animais da propriedade, a entidade compra_animal tem como objetivo guardar as compras de animais, a entidade diagnóstico_prenhez é responsável por manter os diagnósticos de prenhez identificados nos animais. As entidades raça, produto, categoria_produto e principio_ativo não possuem cadastro ou alteração dentro da aplicação, estas entidades armazenam dados previamente cadastrados, que são apenas consumidos em alguns módulos dentro da aplicação.

Figura 4 – Diagrama de Entidade e Relacionamento.



Fonte: Autoria própria (2023).

4.3 Apresentação do sistema

O sistema para dispositivos móveis que será apresentado nesta seção foi desenvolvido utilizando o *framework* Flutter. O aplicativo ainda não está listado na PlayStore (Android) e AppStore (iOS), sendo possível instalar apenas em modo de *debug*, desenvolvimento ou *release* a partir da execução de um *Android Package Kit* (Kit de pacote Android) (APK) gerado. Após instalar o aplicativo no emulador ou dispositivo móvel o ícone, que pode ser visualizado na Figura 7, é disponibilizado na tela do dispositivo. Ao clicar no ícone a tela *splashscreen*, que pode ser visualizada na Figura 6, é exibida ao usuário enquanto a tela principal do aplicativo é carregada. A tela inicial e o ícone foram gerados a partir da logomarca do IDR, disponível para visualização na Figura 5.

Figura 5 – Logomarca IDR

Fonte: Autoria própria (2023).

Figura 6 – *Splashscreen*

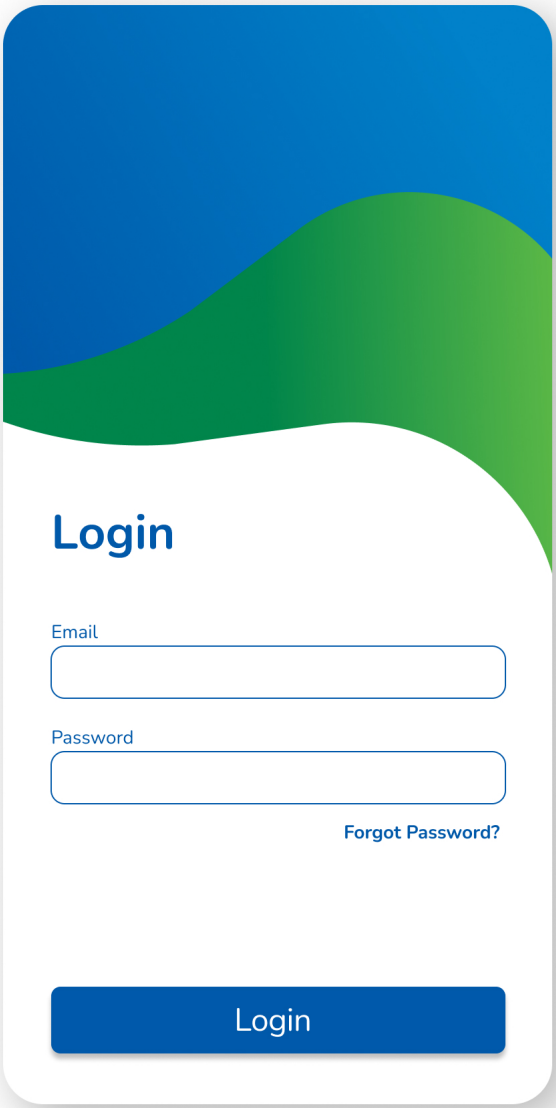
Fonte: Autoria própria (2023).

Figura 7 – Ícone do Aplicativo

Fonte: Autoria própria (2023).

A prototipação das telas do sistema foram desenvolvidas no Figma, buscando seguir o padrão de cores da logo marca do IDR, disponível para visualização na Figura 8.

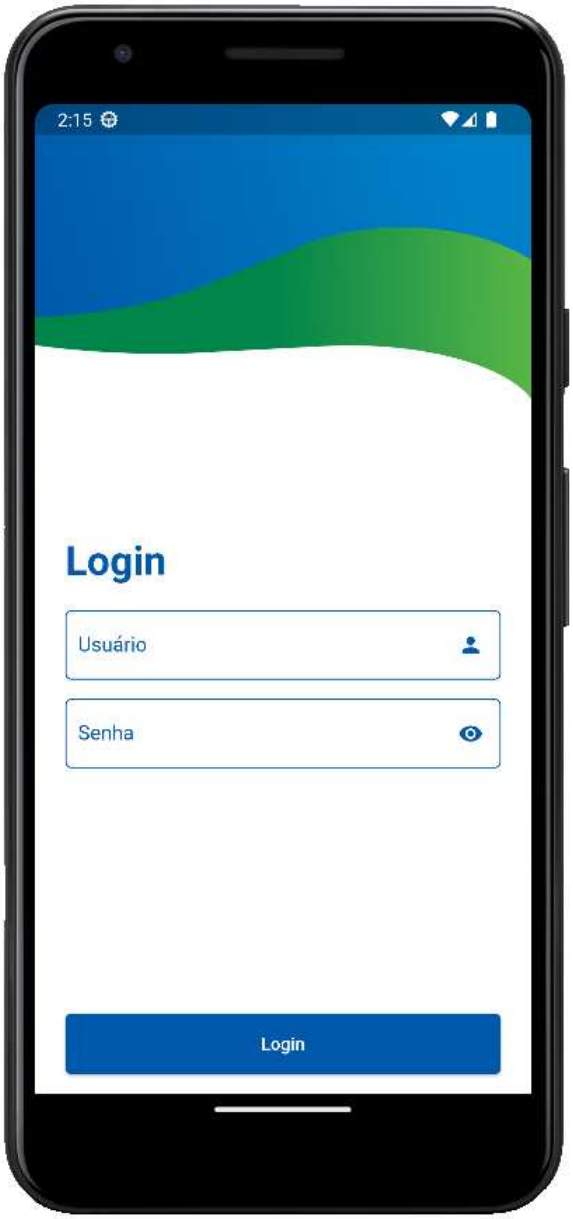
Figura 8 – Protótipo tela de login do sistema.



Fonte: Autoria própria (2023).

Seguindo a prototipação feita no Figma, foi codificado/desenvolvida a tela de autenticação do aplicativo, na qual o usuário deverá inserir o e-mail e a senha, como pode ser observado na Figura 9, para poder se autenticar no sistema, assim podendo acessar os outros dos módulos.

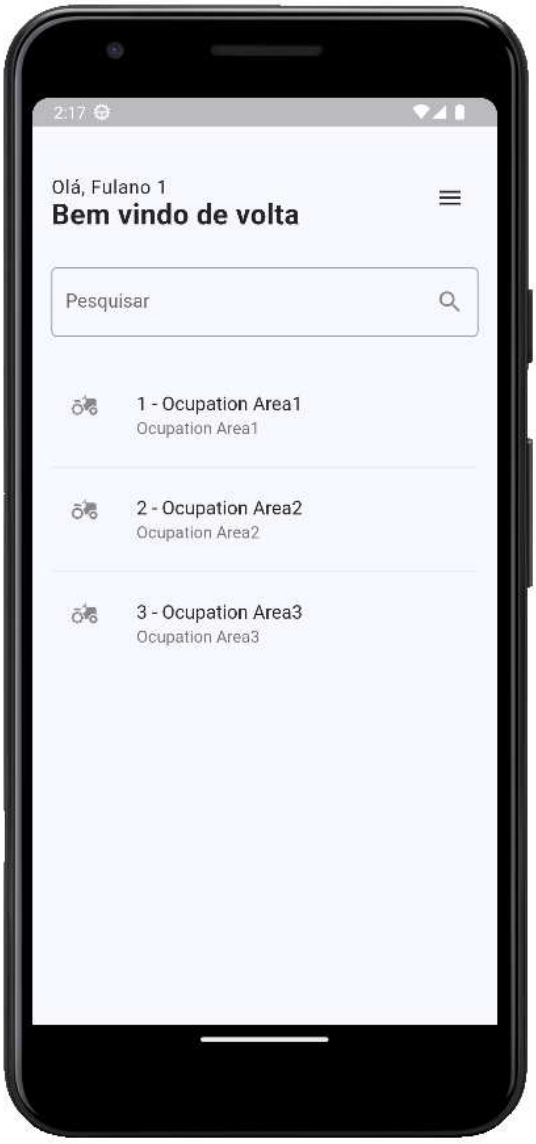
Figura 9 – Tela de login do sistema.



Fonte: Autoria própria (2023).

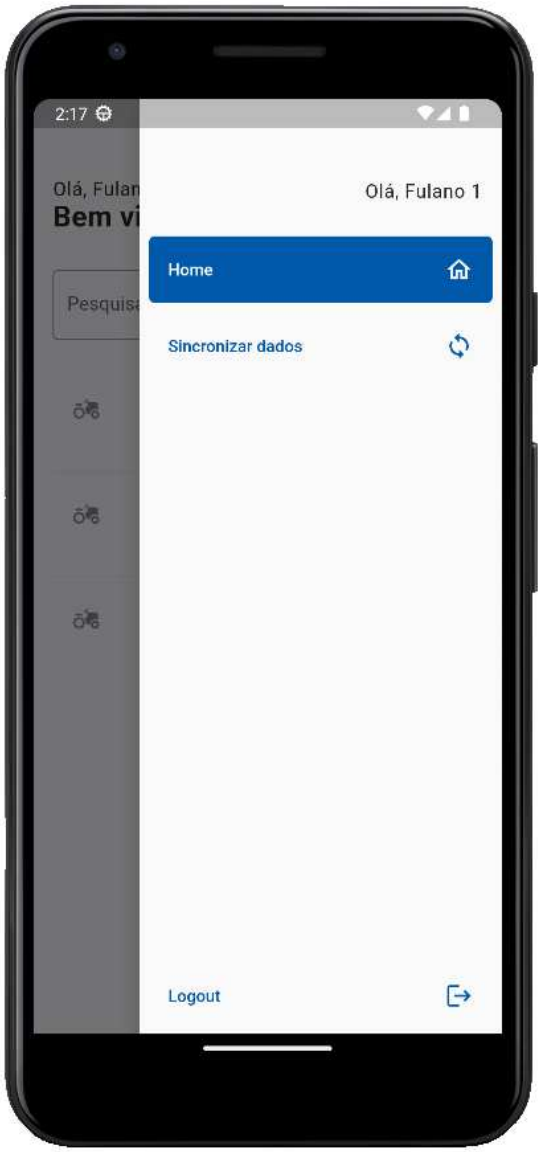
Após a autenticação, é apresentada a tela *home*, como pode ser observada na Figura 10, na qual é exibido o nome do usuário, uma listagem das propriedades disponíveis para acesso do usuário juntamente com um campo de pesquisa, para caso o usuário deseje buscar uma propriedade específica. Além disso, é apresentada uma opção para exibir o menu lateral, que pode ser visualizada na Figura 11, que possui os acessos disponíveis para este nível na aplicação, sendo eles: página *home*, página de sincronização dos dados e *logout* (saída) do sistema.

Figura 10 – Tela *home*



Fonte: Autoria própria (2023).

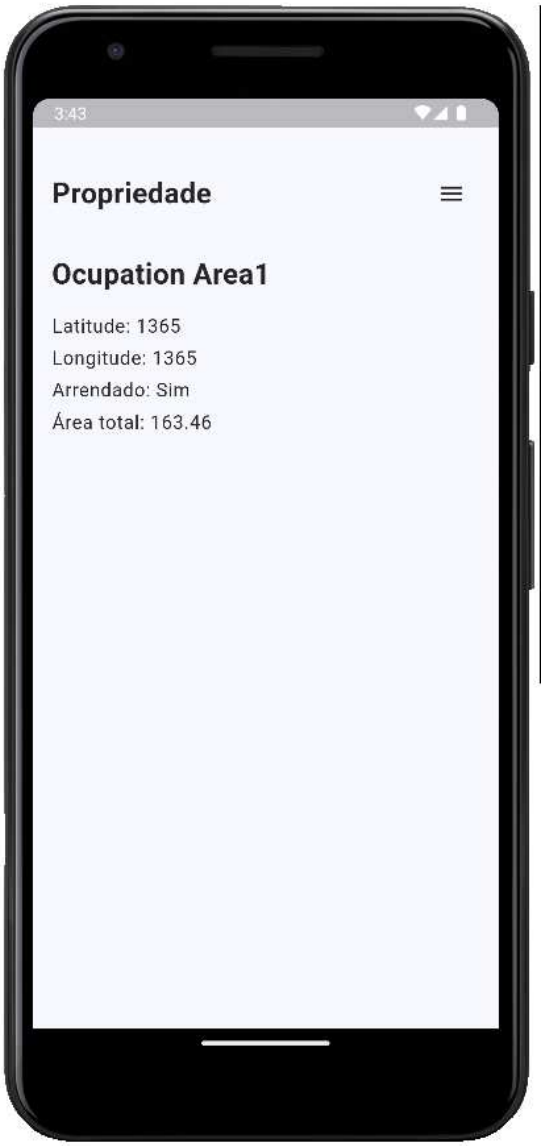
Figura 11 – Menu lateral 1



Fonte: Autoria própria (2023).

Para iniciar a alteração de dados de uma propriedade, na tela *home* o usuário deve selecionar uma das propriedades listadas, assim permitindo o acesso para outro nível da aplicação. Na tela seguinte são exibidas algumas informações do cadastro da propriedade selecionada, como pode ser observado na Figura 12. Ao clicar no menu lateral dessa tela será possível a interação com outros módulos do sistema, como apresentado na Figura 13, nesse menu os dados estarão relacionados a propriedade previamente selecionada.

Figura 12 – Informações da propriedade



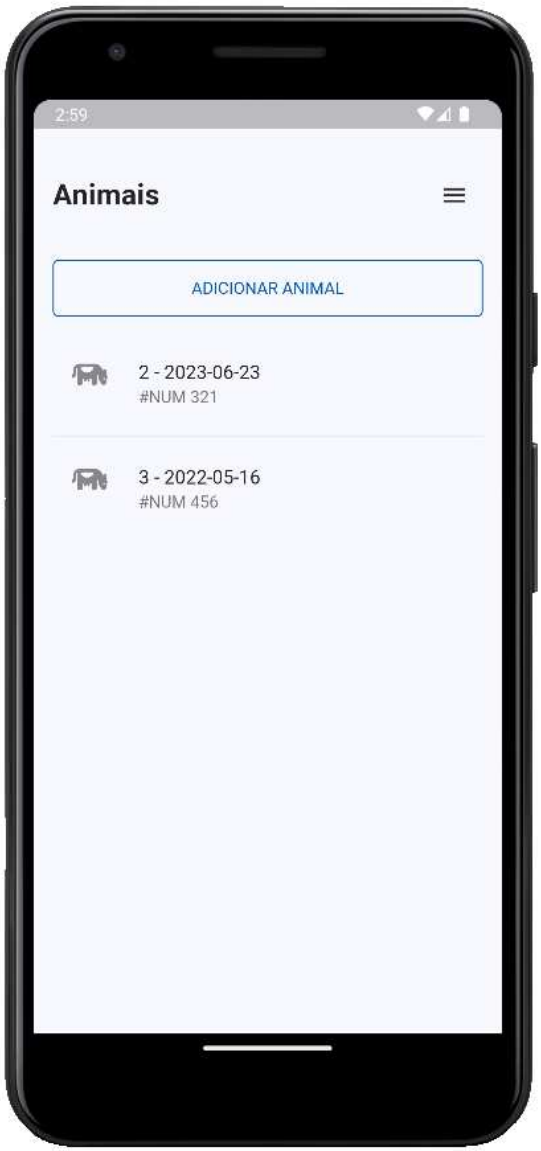
Fonte: Autoria própria (2023).

Figura 13 – Menu lateral 2

Fonte: Autoria própria (2023).

Quando selecionado o menu de animais o usuário é direcionado para a página na qual são buscados os animais referentes a propriedade previamente selecionada. Os animais encontrados são apresentados em uma listagem, juntamente com o acesso ao menu lateral e botão para cadastrar um novo animal, como pode ser observado na Figura 14. As opções de "remover"e "editar"de um animal, apresentadas na Figura 15, podem ser acessadas deslizando um item da lista da esquerda para direita, pois nesta lista foi implementado um componente "*slide*".

Figura 14 – Tela de animais



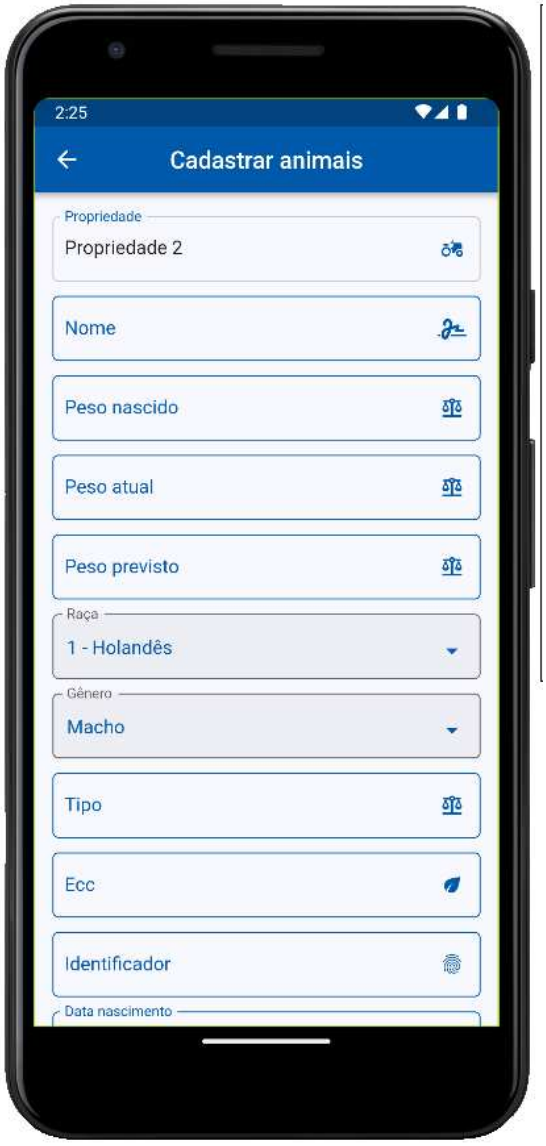
Fonte: Autoria própria (2023).

Figura 15 – Componente *slide*

Fonte: Autoria própria (2023).

Ao clicar no botão "Adicionar Animal", apresentado na Figura 14, o usuário é direcionado para o formulário de cadastro da Figura 16, que é composto por campos do tipo *input* de inserção de conteúdo (caixa de texto), *dropdowns* para seleção de algumas opções (caixa de seleção), que pode ser visualizado um exemplo na Figura 18. São apresentados também *input* de calendário para seleção de data como pode ser visto na Figura 19. Na Figura 17 são exibidas as mensagens de validação dos campos que são obrigatórios do formulário de cadastro.

Figura 16 – Formulário de animais

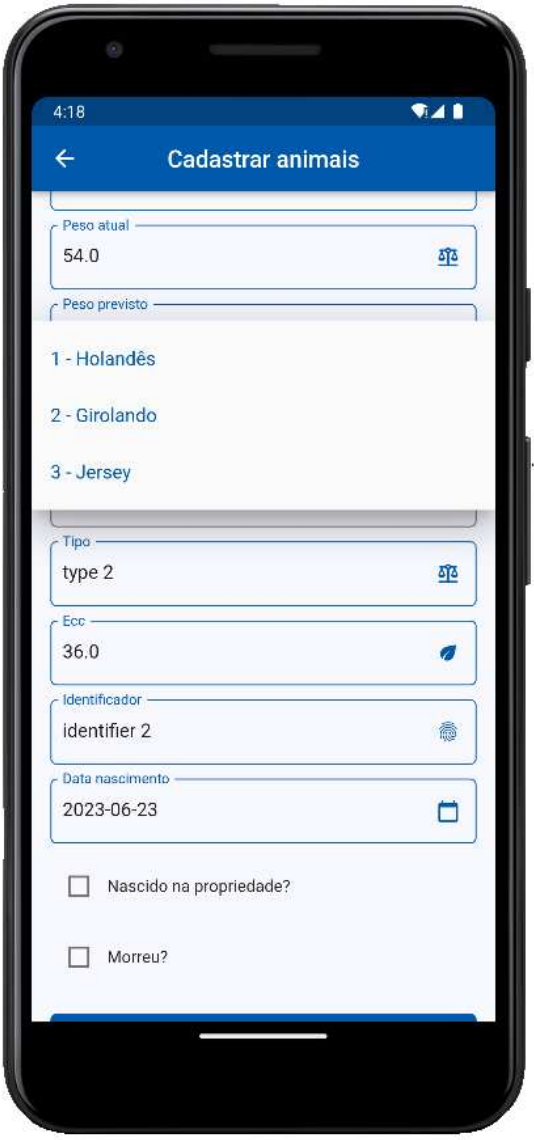


Fonte: Autoria própria (2023).

Figura 17 – Validações de inputs

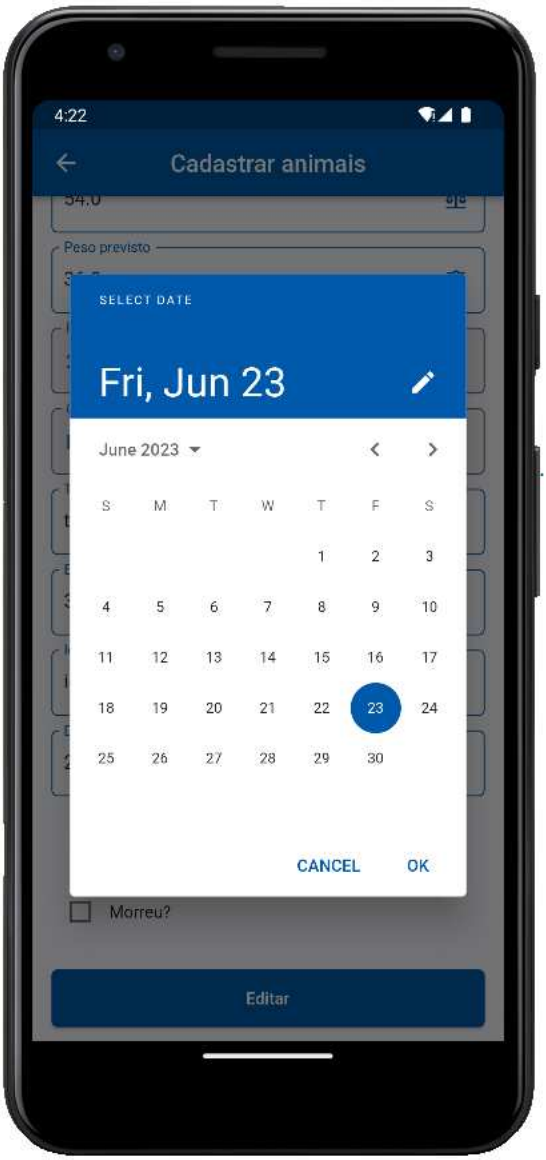
Fonte: Autoria própria (2023).

Figura 18 – *Dropdown* raças animais



Fonte: Autoria própria (2023).

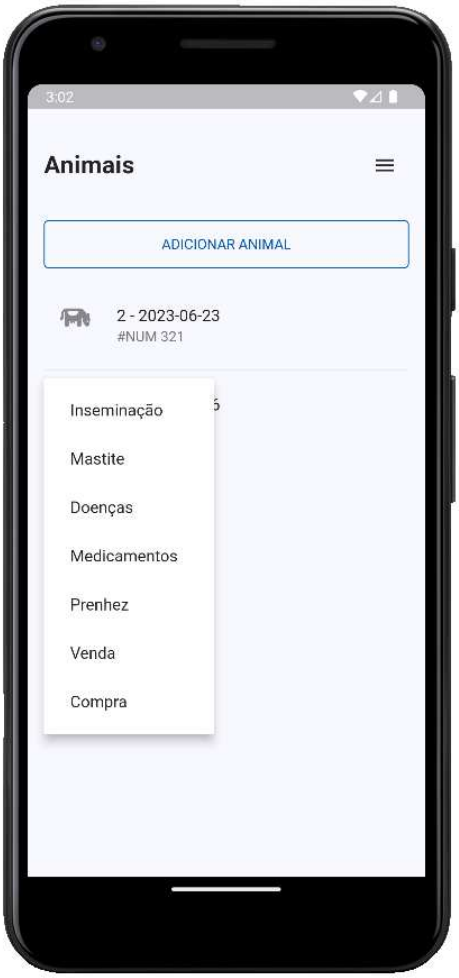
Figura 19 – Calendário exemplo



Fonte: Autoria própria (2023).

Ao clicar em um item (animal) da lista da Figura 14 é exibido um menu com as opções de acesso aos submódulos disponíveis, como pode ser verificado na Figura 20. Ao acessar algum dos submódulos apresentados no menu, são listados as informações referentes a este submódulo e ao animal que foi selecionado na lista.

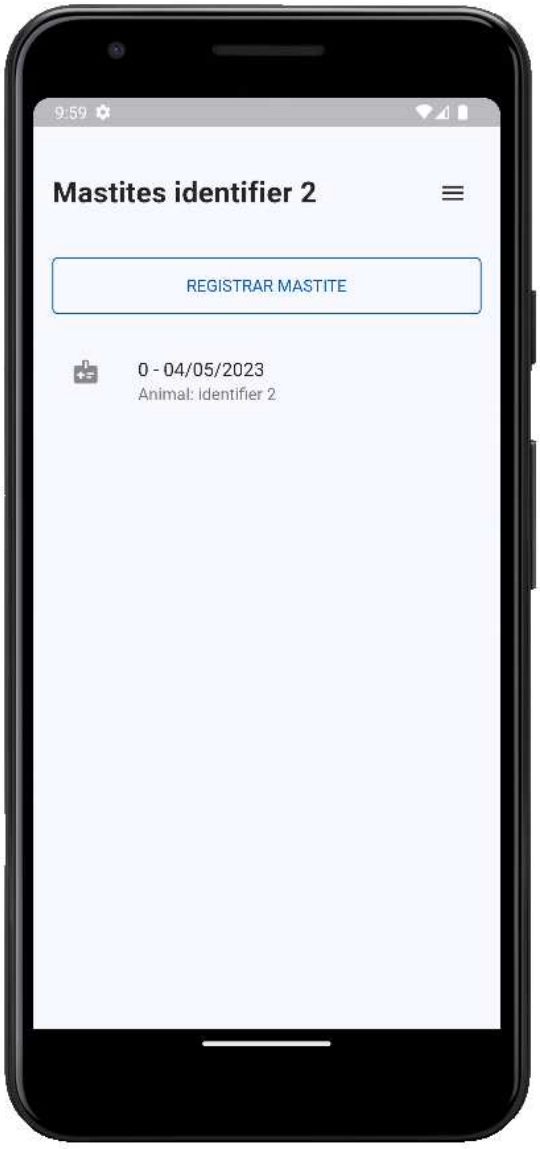
Figura 20 – Submódulos de animais.



Fonte: Autoria própria (2023).

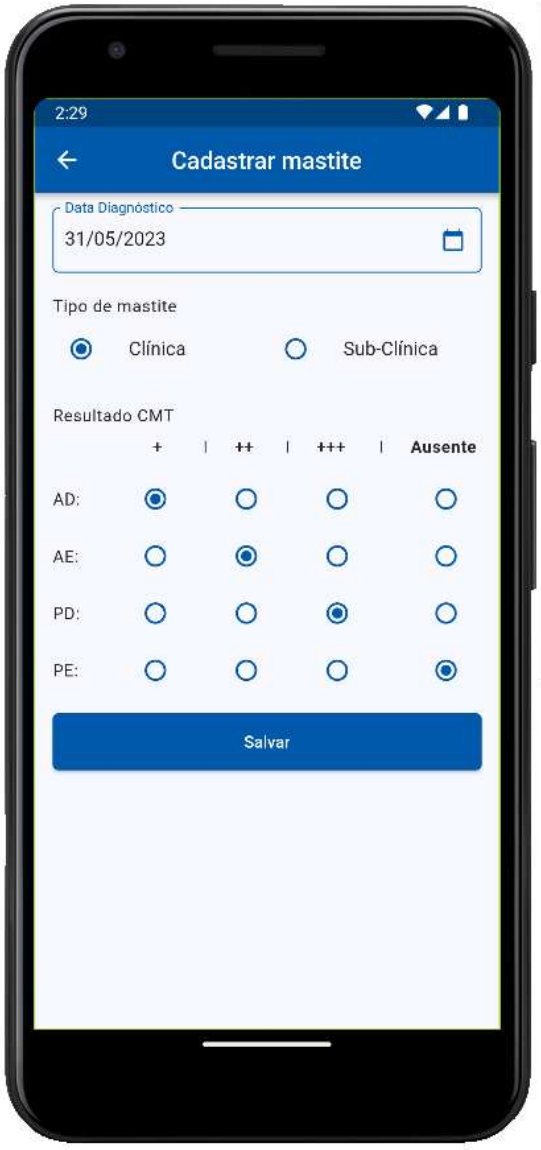
Ao acessar um dos submódulos apresentados na Figura 20 são listadas as informações referentes a este submódulo e ao animal que foi selecionado. Na Figura 21 são exibidas as mastites registradas deste animal previamente selecionado. Já na Figura 22 é mostrado o formulário para registro de uma nova identificação de mastite no animal, em que constam exemplos de botões do tipo *radio* criados, simulando o preenchimento de uma ficha técnica, utilizada pelos técnicos do IDR.

Figura 21 – Tela mastites



Fonte: Autoria própria (2023).

Figura 22 – Formulário de mastite



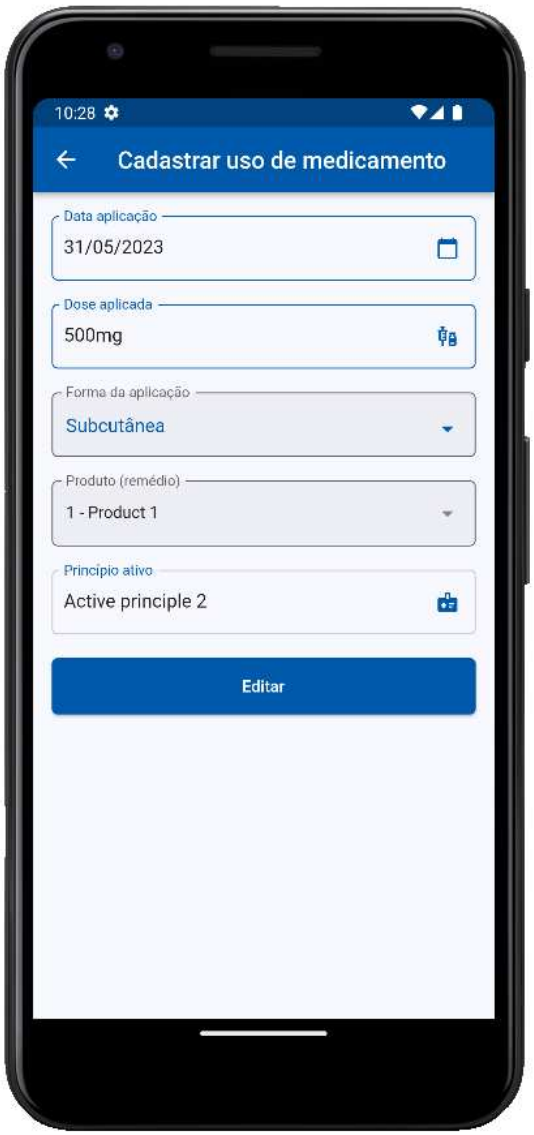
Fonte: Autoria própria (2023).

As Figuras 23 e 24 representam a tela de listagem dos medicamentos usados em determinado animal e a tela de registro de uso de um medicamento, respectivamente. Como nas tela anteriores o uso de medicamento está relacionado ao animal selecionado anteriormente.

Figura 23 – Tela medicamentos

Fonte: Autoria própria (2023).

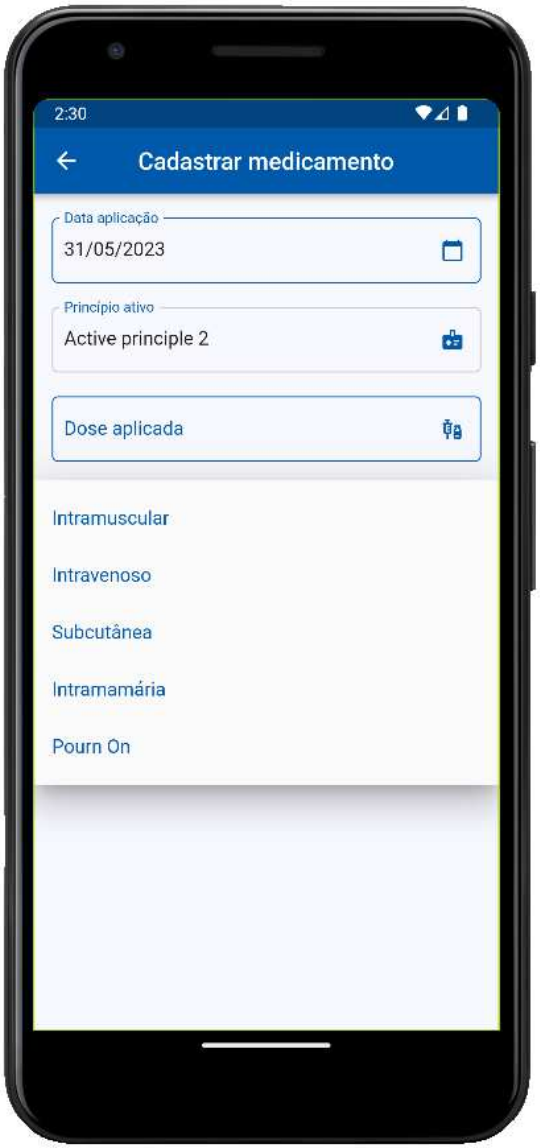
Figura 24 – Formulário uso de medicação



Fonte: Autoria própria (2023).

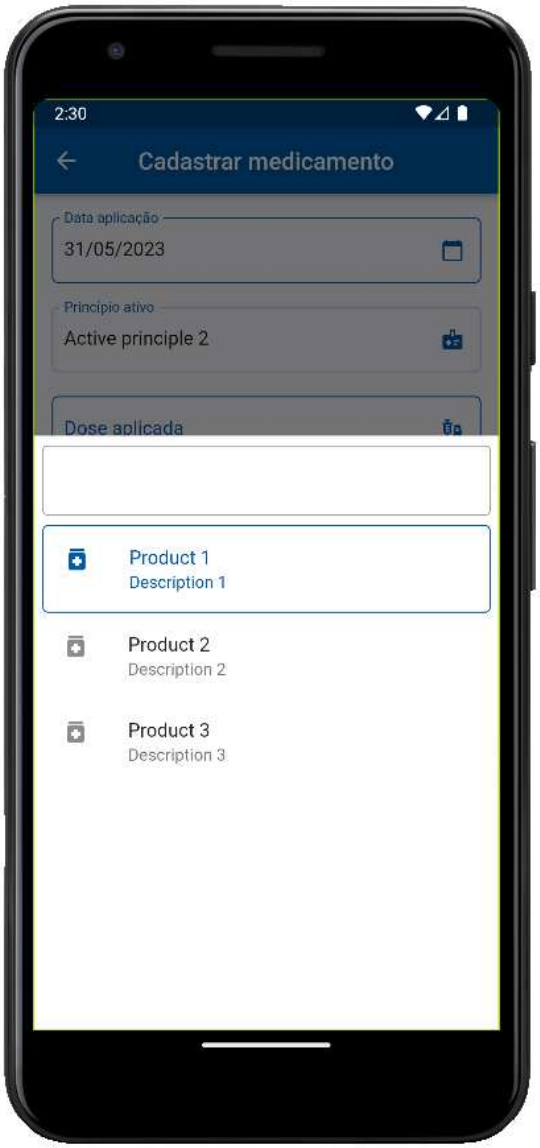
Nas Figura 25 e Figura 26 são apresentados os campos do tipo *dropdown* presentes no formulário de uso de um medicamento. A Figura 26 exibe um *dropdown* dos produtos (remédios) previamente cadastrados no sistema, com filtro de pesquisa, possibilitando ao usuário encontrar o produto de maneira mais eficiente.

Figura 25 – *Dropdown* formas de aplicação



Fonte: Autoria própria (2023).

Figura 26 – *Dropdown* com filtro

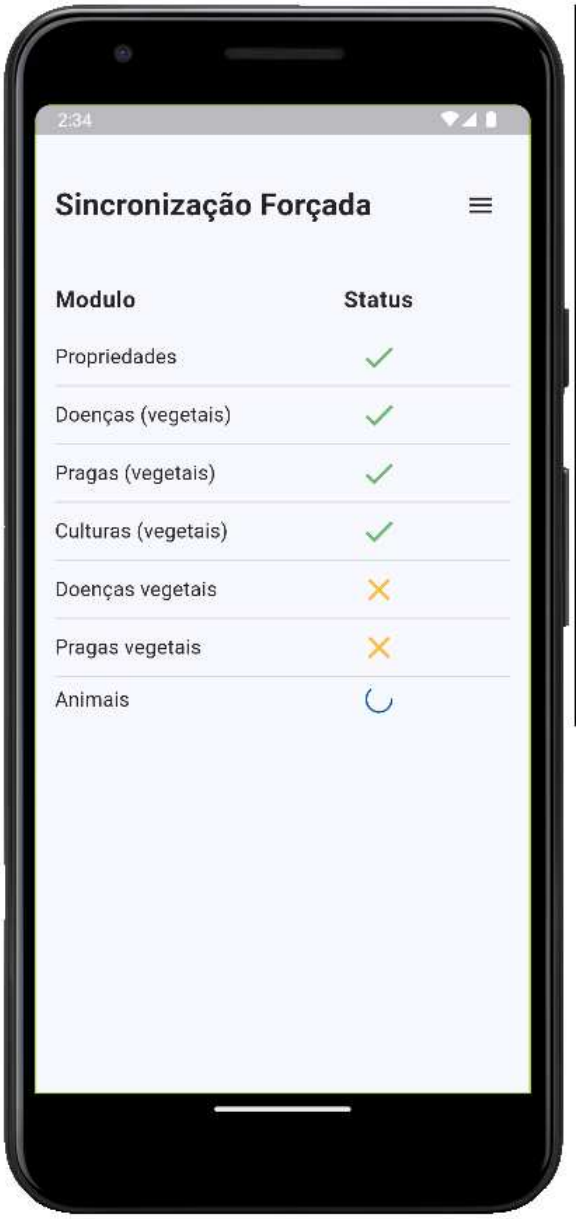


Fonte: Autoria própria (2023).

Quando um técnico finalizar o preenchimento dos dados após uma visita à uma propriedade rural é possível realizar a sincronização dos dados coletados de maneira *offline* com a API. Para realizar a sincronização dos dados, seguindo a ideia apresentada no Fluxograma do Figura 3, o usuário acessa o módulo de sincronização, apresentado no menu da Figura 11. Ao abrir a página a sincronização é iniciada, listando os módulos e a situação de cada um, como pode ser visto na Figura 27. Todos os dados alterados ou inseridos no aplicativo serão sincronizados.

A Figura 27 representa a tela de sincronização iniciada. Quando o módulo esta sendo sincronizado, a coluna de situação exibe uma animação, como pode ser notado no item "Animais", já quando o módulo é sincronizado com sucesso, é apresentado um ícone de *"check"*, observado no item "Propriedades"e quando ocorre alguma exceção durante a sincronização é mostrado um ícone de "erro".

Figura 27 – Tela de sincronização.



Fonte: Autoria própria (2023).

Quando a sincronização é finalizada é exibida ao usuário uma mensagem por meio de uma tela de informação, que pode ser vista na Figura 28. Também é possível visualizar o erro ocorrido no momento da sincronização de algum módulo, pressionando o ícone de "erro", na lista de módulos sincronizados, o sistema exibe uma tela com a mensagem de aviso, apresentado na Figura 29, que contém as informações da exceção encontrada.

Figura 28 – *Dialog* de informação

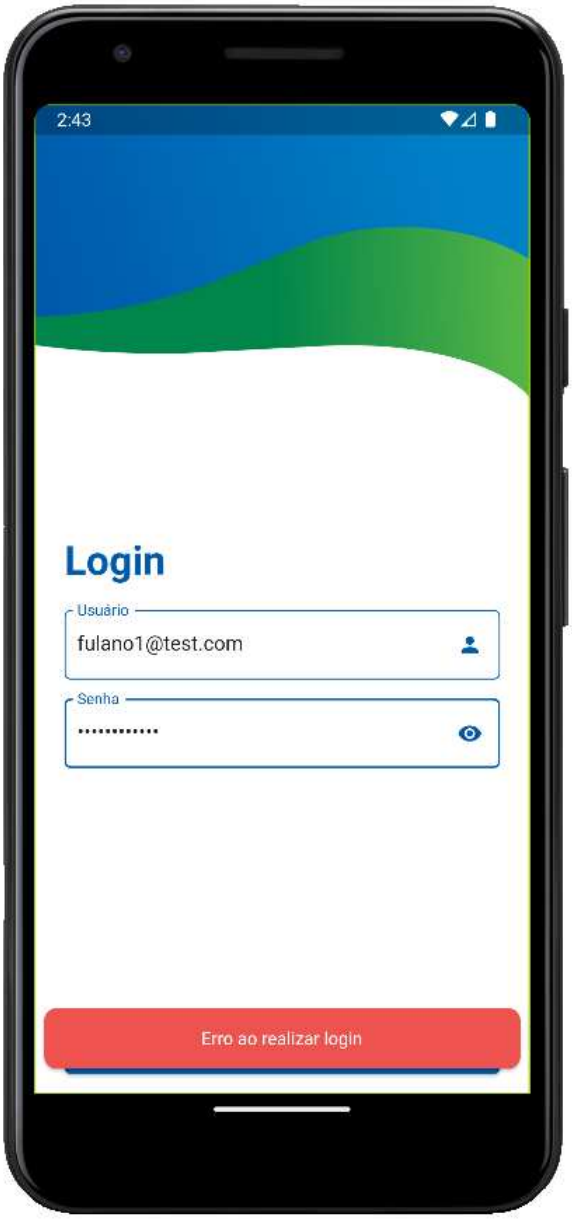
Fonte: Autoria própria (2023).

Figura 29 – *Dialog* de aviso

Fonte: Autoria própria (2023).

Para notificação do usuário durante o uso da aplicação, além da reutilização dos componentes de *dialog* apresentados nas Figuras 28 e 29, o sistema possui um componente de *snackbar*, exibido na Figura 30, que pode ser utilizado para notificar um erro, aviso ou informação, podendo ser personalizado de acordo com a necessidade do desenvolvedor.

Figura 30 – Exemplo *snackbar*



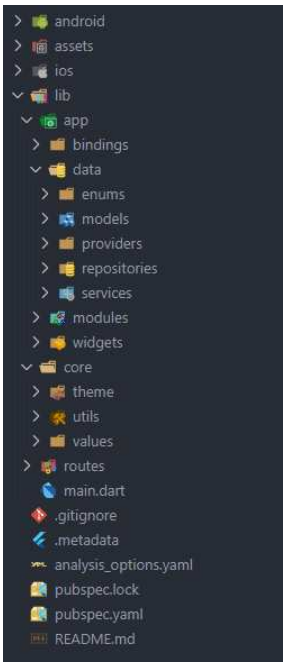
Fonte: Autoria própria (2023).

4.4 Implementação do sistema

O aplicativo foi implementando em Flutter, visando economia de tempo e esforço, por se tratar de um *framework* multiplataforma. O *back-end* do sistema foi desenvolvido na linguagem Java e Spring Framework e não faz parte do escopo deste trabalho. A comunicação entre as duas plataformas é realizada via REST por meio de requisições *Hypertext Transfer Protocol* (HTTP) trocando dados no formato *JavaScript Object Notation* (JSON). Além do Flutter, foi utilizando o pacote *GetX* para gerenciamento de estado, injeção de dependência e gerenciamento de rotas, além de alguns outros recursos disponibilizados pelo pacote e do pacote *Hive* para armazenamento dos dados *offline*.

A Figura 31 apresenta a estrutura da aplicação móvel. O desenvolvimento da aplicação é feito dentro da pasta *lib*.

Figura 31 – Estruturação do projeto



Fonte: Autoria própria (2023).

A organização de pastas e arquivos da estrutura:

- *app/bindings*: Diretório que contém a classe responsável por realizar o gerenciamento das dependências da aplicação;
- *app/data*: Diretório que contém os arquivos relacionado aos dados da aplicação;
- *app/data/enum*: Diretório responsável por conter os *enums* do sistema. Os *enums* representam um conjunto fixo de valores constantes, permitindo uma forma organizada e legível de representar opções ou categorias específicas;

- app/data/models: Diretório responsável por armazenar as entidades/modelos responsáveis por abstrair nossos objetos;
- app/data/providers: Diretório responsável por grupar os provedores da aplicação. Neste caso, os provedores de dados são a API e o banco de dados local;
- app/data/services: Diretório responsável por conter os módulos de serviço. Os *services*, dentro da aplicação, são classes que irão mediar a comunicação entre o *controller* e a camada de dados;
- app/data/repositories: Diretório que contém os repositórios dos nosso módulos, responsáveis por acessar e gerenciar os dados dos provedores;
- app/modules: Diretório que contém o desenvolvimento dos módulos do sistema, com as suas páginas e controladores;
- app/widgets: Diretório que contém os `widgtes` (componentes) que podem ser reutilizados por vários módulos;
- core: Diretório que possui arquivos de compartilhados por toda aplicação como configuração de tema e estilos, constantes, funções úteis;
- routes: Diretório que possui a configuração de rotas da aplicação.

As entidades do aplicativo utilizado para persistência dos dados foram desenvolvidas de maneira padronizada, contendo suas propriedades, métodos e construtores. Normalmente possuem métodos como `copyWith` para criar uma cópia modificada do objeto, métodos `toMap` e `toJson` para converter o objeto em um mapa ou JSON, `factory method fromMap` e `fromJson` para criar uma instância do entidade a partir de um mapa ou JSON e também implementa os métodos `toString` e `hashCode` para facilitar a representação em formato de *string* e o cálculo do `hashcode`, respectivamente. Como a entidade de Raça, apresentada na Listagem 1.

As anotações `@HiveType` e `@HiveField` são responsáveis por gerar o *adapter* responsável por salvar este objeto de maneira *offline* com o pacote Hive. Todos as entidades e propriedades destas entidades que necessitam ser salvas *offline* precisam ser anotadas.

Listagem 1 – Entidade Raça

```dart
@HiveType(typeId: 12)
class BreedModel {
  @HiveField(1)
  String? internalId;
  @HiveField(2)
  int? id;
  @HiveField(3)
  String? breedName;
  BreedModel({
    this.internalId,
    this.id,
    this.breedName,
  });
  BreedModel copyWith({
    String? internalId,
    int? id,
    String? breedName,
  }) {
    return BreedModel(
      internalId: internalId ?? this.internalId,
      id: id ?? this.id,
      breedName: breedName ?? this.breedName,
    );
  }
  Map<String, dynamic> toMap() {
    return <String, dynamic>{
      'internalId': internalId,
      'id': id,
      'breedName': breedName
    };
  }
  factory BreedModel.fromMap(Map<String, dynamic> map) {
    return BreedModel(
      internalId: map['internalId'],
      id: map['id']?.toInt(),
      breedName: map['breedName'],
    );
  }
  String toJson() => json.encode(toMap());
  factory BreedModel.fromJson(String source) =>
    BreedModel.fromMap(json.decode(source));
  @override
  String toString() => '$id - $breedName';
  @override
  int get hashCode => internalId.hashCode ^ id.hashCode ^ breedName.hashCode;
}
```

Fonte: Autoria própria (2023).

Seguindo o fluxo de uso do aplicativo a autenticação é realizada a partir da tela de *login*, vista na Figura 9, em que o usuário informa os dados no formulário, estes são validados e enviados para o método `signIn`, localizado no arquivo controlador do módulo de autenticação, disponível para visualização na Listagem 2.

O método `signIn` faz o tratamento dos dados do formulário e envia para o *service* de autenticação, que pode ser visto na Listagem 3, que então encaminha para o *repository*, responsável por gerenciar os dados, assim realizando requisição HTTP para a rota de autenticação da API, exibida na Listagem 4.

Listagem 2 – Classe controladora com método de autenticação

```dart
class LoginController extends GetxController {
  final LoginService _loginService;
  //... trecho código omitido
  //... método signIn
  Future<void> signIn({
    required String username,
    required String password,
  }) async {
    isLoading.value = true;
    var login = {"username": username, "password": password,};
    try {
      await _loginService.login(login).then((value) {
        if (value != null) {
          auth!.changeApiToken(value.token);
          auth!.changeIsLogged(true);
          auth!.changeDisplayName(value.displayName);
          reauth();
        } else {
          throw Exception(e);
        }
      });
    } catch (e) {
      //Exibe snackbar
      Snack.show(
        content: 'Erro ao realizar login',
        snackType: SnackType.error,
        behavior: SnackBarBehavior.floating,
      );
    }

    isLoading.value = false;
  }
}
```

Fonte: Autoria própria (2023).

Listagem 3 – Classe *service* da autenticação

```
class LoginServiceImpl implements LoginService {
  LoginRepository _loginRepository;

  LoginServiceImpl({
    required LoginRepository loginRepository,
  }) : _loginRepository = loginRepository;

  @override
  Future<LoginModel?> login(dynamic json) => _loginRepository.login(json);
}
```

Fonte: Autoria própria (2023).

Listagem 4 – Repositório com método de autenticação

```
class LoginRepositoryImpl implements LoginRepository {
  final RestClient _restClient;

  LoginRepositoryImpl({
    required RestClient restClient,
  }) : _restClient = restClient;

  @override
  Future<LoginModel?> login(dynamic json) async {
    final result = await _restClient.post(
      'login',
      json,
      decoder: (data) {
        final resultData = data;
        if (resultData != null) {
          return LoginModel.fromMap(resultData);
        } else {
          return null;
        }
      },
    );

    if (result.hasError) {
      print('Error [${result.statusText}]');
      throw Exception('Error _ ${result.body}');
    }

    return result.body;
  }
}
```

Fonte: Autoria própria (2023).

Caso o retorno da rota de autenticação da API seja um sucesso, os dados de autenticação são salvos no dispositivo e o usuário redirecionado para a página *home* pela função `reauth` apresentado na Listagem 5.

Listagem 5 – Função `reauth`

```
1   reauth() async {
2     await Future.delayed(Duration.zero, () {
3       if (auth!.isLogged()) {
4         Get.offNamed(Routes.HOME);
5       }
6     });
7   }
```

Fonte: Autoria própria (2023).

Visando a separação de responsabilidades, flexibilidade e manutenção é aplicado o conceito de injeção de dependência (GONÇALVES; MENDES, 2022) em algumas classes da aplicação. Neste trabalho é encarregado ao pacote GetX realizar o controle da injeção de dependência. O pacote GetX no Flutter permite registrar e fornecer instâncias de dependências de forma centralizada, facilitando o gerenciamento e acesso a essas dependências em todo o aplicativo. O exemplo das dependências injetadas nas classes das Listagens 2, 3 e 4, como pode ser visto na Listagem 6.

Listagem 6 – Classe gerenciadora das dependências

```
1   class ApplicationBindings implements Bindings {
2     @override
3     void dependencies() {
4       Get.lazyPut(
5         () => RestClient(),
6         fenix: true,
7       );
8       Get.lazyPut(
9         () => AuthService(),
10        fenix: true,
11      );
12      Get.lazyPut<LoginRepository>(
13        () => LoginRepositoryImpl(restClient: Get.find()),
14        fenix: true,
15      );
16      Get.lazyPut<LoginService>(
17        () => LoginServiceImpl(loginRepository: Get.find()),
18        fenix: true,
19      );
20      ...
21    }
22  }
```

Fonte: Autoria própria (2023).

Continuando o fluxo de utilização do sistema, após autenticado, o usuário irá acessar telas com listas de dados, por exemplo, as propriedades e os animais. A listagem dos animais exibida na Figura 14 é construída a partir da busca dos dados realizada pela função `loadAnimals`, que está contida no controlador apresentado na Listagem 7. Após a busca dos dados os mesmos são atribuídos para uma lista tipada de acordo com a entidade `AnimalModel`.

Listagem 7 – Controlador Animal

```
1  class AnimalController extends GetxController {
2    final AnimalService _animalService;
3    final PropertyService _propertyService;
4    final AuthService _authService;
5    AnimalController({
6      required AnimalService animalService,
7      required PropertyService propertyService,
8      required AuthService authService,
9    })   : _animalService = animalService,
10           _propertyService = propertyService,
11           _authService = authService;
12   AuthService? auth;
13   final animalsFinal = <AnimalModel>[].obs;
14   final property = PropertyModel().obs;
15   @override
16   void onInit() async {
17     super.onInit();
18     property.value = _authService.property();
19   }
20   @override
21   void onReady() async {
22     super.onReady();
23     loadAnimals();
24   }
25   void loadAnimals() async {
26     try {
27       final animalsData = await _animalService.getAllAnimals(
28                           property.value.id);
29       animalsFinal.assignAll(animalsData);
30     } catch (e) {
31       //Exibe snackbar
32       Snack.show(
33         content: 'Ocorreu um erro ao buscar animais',
34         snackType: SnackType.error,
35         behavior: SnackBarBehavior.floating,
36       );
37     }
38   }
39 }
```

Fonte: Autoria própria (2023).

Como o gerenciador de estado do *Getx* trabalha com programação reativa, é necessário adicionar `.obs` ao final das variáveis que podem sofrer alteração, para assim reconstruir o componente que utiliza da variável em tela quando ela ter seu valor alterado, como pode ser observado nas linhas 13 e 14 da Listagem 7. Para isso, é necessário colocar esta variável dentro de um *widget* `Obx()` .

O *widget* responsável por construir em tela a listagem dos animais é apresentado na Listagem 8. Este componente, além de montar a lista, faz com que em cada um dos itens seja possível diferente interações, como as apresentadas nas Figuras 15 e 20.

Listagem 8 – Classe gerenciadora das dependências

```
1  Expanded(
2    child: Obx(() {
3      return ListView.separated(
4        separatorBuilder: (context, index) => Divider(),
5        itemCount: controller.animalsFinal.length,
6        shrinkWrap: true,
7        scrollDirection: Axis.vertical,
8        itemBuilder: (context, index) {
9          var animal = controller.animalsFinal[index];
10         return PopupMenuButton<AnimalMenuType>(
11           initialValue: selectedMenu,
12           onSelected: (AnimalMenuType item) =>
13               onSelected(item, animal),
14           itemBuilder: (BuildContext context) =>
15               _itemsPopMenu(),
16           child: CustomSlidable(
17             identity: index,
18             content:
19                 '${animal.name ?? animal.identifier ?? ''}',
20             title:
21                 '${animal.id ?? index + 1} - ${animal.bornDate}',
22             icon: FontAwesomeIcons.cow,
23             onPressedEditCallBack: (BuildContext context) {
24               controller.goToForm(animal, index);
25             },
26             onPressedRemoveCallBack: (BuildContext context) {
27               controller.removeAnimal(animal);
28             },
29           ),
30         );
31       },
32     );
33   }),
34 ),
```

Fonte: Autoria própria (2023).

A função `itemsPopMenu` retorna, quando o usuário clicar em um dos itens da lista, um `PopMenu`, como pode ser observado na Figura 20. Esse objeto que é construído com o código apresentado na Listagem 9. Já a função `onSelected`, apresentada na linha 13 da Listagem 10, valida o menu que foi clicado, para direcionar o usuário para a tela solicitada.

Listagem 9 – Construção do *PopMenu*

```
1   [ PopupMenuItem<AnimalMenuType>(
2       value: AnimalMenuType.insemination,
3       child: Row(
4         children: const [
5           Padding(
6             padding: EdgeInsets.only(left: 10),
7             child: Text(
8               'Inseminação',
9             ),
10          )
11        ],
12      ),
13    ),
14    PopupMenuItem<AnimalMenuType>(
15      value: AnimalMenuType.mastitis,
16      child: Row(
17        children: const [
18          Padding(
19            padding: EdgeInsets.only(left: 10),
20            child: Text(
21              'Mastite',
22            ),
23          )
24        ],
25      ),
26    ), ...
27  ]
```

Fonte: Autoria própria (2023).

Além das telas de lista de dados a aplicação conta com diversos formulários de cadastro, nos quais são exibidos para o usuário diversos campos de entradas de dados (*inputs*). Na Listagem 11 é dado um exemplo breve da construção de um componente de *input*. Com seus métodos e exemplos de estilização codificados seriam necessários cerca de 80 linhas para construção de um único campo de inserção de texto.

Utilizando o conceito de componentização, que é processo de dividir um sistema em partes independentes e reutilizáveis de código, os campos de texto utilizados na aplicação desenvolvida permitem que, além da padronização do estilo em toda a aplicação, sejam necessários cerca de 10 linhas de código para a construção do campo de inserção, além da validação, que pode ser única para cada campo. O código que exemplifica a criação de um componente de entrada de dados pode ser observado na Listagem 12.

Listagem 10 – Validação do menu clicado pelo usuário

```
onSelected(item, animal) {
    {
      switch (item) {
        case AnimalMenuType.insemination:
          controller.goToNextPage(animal, Routes.INSEMINATION);
          break;
        case AnimalMenuType.mastitis:
          controller.goToNextPage(animal, Routes.MASTITIS);
          break;
        case AnimalMenuType.disease:
          controller.goToNextPage(animal, Routes.DISEASE_ANIMAL);
          break;
      }
    }
}
```

Fonte: Autoria própria (2023).

Listagem 11 – Exemplo de construção de um componente *input*

```
TextFormField(
    onChanged: (value) {},
    onSaved: (newValue) {},
    validator: (value) {},
    maxLines: 1,
    controller: inputController,
    style: TextStyle(...),
    focusNode: inputFocus,
    keyboardType: keyboardType,
    inputFormatters: inputFormatters ?? [],
    obscureText: isObscureText,
    cursorColor: UIColors.primaryColor,
    decoration: InputDecoration(
        isDense: true,
        contentPadding: ...,
        labelText: '',
        labelStyle: TextStyle(...),
        suffixIcon: Widget(...),
        errorBorder: OutlineInputBorder(...),
        focusedErrorBorder: OutlineInputBorder(...),
        enabledBorder: OutlineInputBorder(...),
        disabledBorder: OutlineInputBorder(...),
        focusedBorder: OutlineInputBorder(...),
        border: OutlineInputBorder(...),
        enabled: true,
        fillColor: UIColors.dangerColor,
        ),
  );
```

Fonte: Autoria própria (2023).

Listagem 12 – Exemplo da utilização do componente de um *input*

```
CustomInputField(
    inputController: controller.nameController,
    labelText: 'Nome',
    suffixIcon: FontAwesomeIcons.signature,
    onChanged: (_) => controller.animal.update((val) => val!.name= _),
    validator: (value) { //Exemplo de validação
    if (value == null || value.toString().trim().isEmpty) {
      return "Campo não pode ser vazio";
    }
    return null;
    },
),
```

Fonte: Autoria própria (2023).

Na Listagem 13 é apresentado um um exemplo da construção de um componente *drop-down*. E na Listagem 14 exibido um exemplo de como utilizar este componente, neste caso, com a lista de raças carregadas no controlador do formulário de cadastro de animais.

Listagem 13 – Exemplo de construção de um componente *dropdown*

```
DropdownButtonFormField<T>(
  decoration: InputDecoration(
    filled: true, labelText: label,
    constraints: BoxConstraints(maxHeight: 55),
    border: OutlineInputBorder(
      borderRadius: BorderRadius.circular(5),
      borderSide: BorderSide( color: UIColors.primaryColor,
        width: 2, strokeAlign: 1,  style: BorderStyle.solid,
      ),
    ),
  ),
  value: selectedValue,
  icon: Icon( Icons.arrow_drop_down, color: UIColors.primaryColor, ),
  iconSize: 24.0, elevation: 16,
  style: TextStyle( color: UIColors.primaryColor, fontSize: 16.0, ),
  isExpanded: true, alignment: Alignment.centerRight,
  onChanged: (newValue) => {},
  items: items.map<DropdownMenuItem<T>>((T value) {
    return DropdownMenuItem<T>(
      value: value,
      child: Text(
        _defaultItemText(value),
        style: TextStyle( color: UIColors.primaryColor, fontSize: 16.0, ),
      ),
    );}).toList(),
);
```

Fonte: Autoria própria (2023).

Listagem 14 – Exemplo da utilização do componente *dropdown*

```
CustomDropdownButton<BreedModel>(
  items: controller.breedsFinal,
  label: 'Raça',
  selectedValue: controller.breedSelected.value,
  onChanged: (BreedModel value) {
    controller.animal.update((val) =>
        val!.breed = value.id.toString());
  },
),
```

Fonte: Autoria própria (2023).

A Figura 16 representa a tela de cadastro de um animal, sendo composta por 11 campos, sendo eles *inputs* de texto de diferentes tipo, *dropdowns* e, 2 *checkbox* que, quando selecionados, exibem mais 2 campos para preenchimento. Utilizando-se do conceito de componentização, explicado anteriormente, é possível reduzir o número de linhas de código e padronizar o formulário de maneira prática e eficiente, uma vez que caso seja necessário mudanças de código nos componentes listados, será necessário apenas alterar a classe do componente.

Para realizar o cadastro de um animal, o formulário possui um controlador próprio, apresentado na Listagem 15, permitindo manipular apenas os dados necessários no cadastro ou edição de um animal. Neste controlador são definidas as propriedades utilizadas nos campos, como apresentadas na Listagem 16. Essa listagem de código também contém o carregamento dos dados que compõem a lista de opções dos *dropdowns*, o objeto a ser manipulado e as funções envio dos dados recolhidos, seja envio para edição ou cadastro de um novo animal.

Listagem 15 – Controlador do formulário de cadastro/edição de animal

```
class AnimalFormController extends GetxController {
  final AnimalService? _animalService;
  final BreedService? _breedService;
  AnimalFormController({ required AnimalService animalService,
    required BreedService breedService, required Uuid uuid,
  }) : _animalService = animalService, _breedService = breedService;
    @override
    void onInit() async {
        super.onInit();
        loadBreeds();
        ...
        if (data[0]['animal'] != null) {
            animal.value = data[0]['animal'];
            setFormValues(data[0]['animal']);
            buttonText.value = "Editar";
        }
    } //...
}
```

Fonte: Autoria própria (2023).

Listagem 16 – Definição das propriedades e variáveis utilizadas dentro do formulário

```
final animal = AnimalModel().obs;

RxString selectedProperty = ''.obs;
RxString buttonText = 'Salvar'.obs;

int? idxAnimal = null;

RxBool isBornInProperty = false.obs;
RxBool isDead = false.obs;

final formKey = GlobalKey<FormState>();
final bornDateController = TextEditingController();
final nameController = TextEditingController();
final bornWeightController = TextEditingController();
final typeController = TextEditingController();
final breedController = TextEditingController();
final currentWeightController = TextEditingController();
final eccController = TextEditingController();
final identifierController = TextEditingController();
final previousWeightController = TextEditingController();
final propertyController = TextEditingController();
final animalMotherIdentifierController = TextEditingController();
final deadDateController = TextEditingController();
final breedsFinal = <BreedModel>[].obs;
final breedSelected = BreedModel().obs;

RxString genderTypeSelected = ''.obs;

List<String> genderTypeList = [];

RxBool isLoading = false.obs;
```

Fonte: Autoria própria (2023).

Ao iniciar o controlador é verificado se é uma edição ou cadastro de um animal, com base no parâmetro passado da tela anterior. Caso seja passado um valor identificador de animal, os campos do formulário são preenchidos e o texto do botão alterado para "editar", como apresentado na Listagem 17. Também são carregados do banco local as raças, como pode ser visto no Listagem 18, para então serem adicionadas ao *dropdown* do formulário com um valor já pré selecionado.

A função `onFormSubmit`, apresentada na Listagem 17, está presente no controlador de animais e valida se é a edição ou cadastro de um animal, enviando para os dados para a função determinada aguardando como retorno um valor *booleano*, em caso de sucesso retorna o valor `true` e em caso de falha retorna o valor `false`, notificando o usuário por meio de um `snackbar` e retornando para a página anterior, recarregando a lista de animais para trazer os dados atualizados.

Listagem 17 – Envio dos dados do cadastro de animal

```
  onFormSubmit() async {
    var isSaved = idxAnimal != null //Verifica se vai salvar ou editar o animal
        ? await _animalService!.editAnimal(animal.value)
        : await _animalService!.saveAnimal(animal.value);
    //Exibe snackbar de acordo com o status da operação anterior
    Snack.show(
      content: isSaved ?
        'Sucesso ao salvar animal' : 'Ocorreu um erro ao salvar animal',
      snackType: isSaved ? SnackType.success : SnackType.error,
      behavior: SnackBarBehavior.floating,
    );
    Get.back(result: isSaved); //retorna para lista de animais
  }
```

Fonte: Autoria própria (2023).

Listagem 18 – Carregamento das raças do banco de dados

```
  void loadBreeds() async {
    isLoading.value = true;
    await Future.delayed(
      const Duration(seconds: 2),
    );
    try {
      final breedsData = await _breedService!.getAllBreeds();
      breedsFinal.assignAll(breedsData);
      setBreedSelected(breedsFinal.value);
      isLoading.value = false;
    } catch (e) {
      isLoading.value = false;
      Snack.show(
        content: 'Ocorreu um erro ao buscar raças :(',
        snackType: SnackType.error,
        behavior: SnackBarBehavior.floating,
      );
    }
  }
  void setBreedSelected(List<BreedModel> list) {
    if (animal.value.breed != null) {
      BreedModel? b = list.firstWhereOrNull(
          (element) => element.id == int.parse(animal.value.breed!));
      b != null ? breedSelected.value = b : breedSelected.value = list[0];
    } else {
      breedSelected.value = breedsFinal.value[0];
      animal.update((val) => val!.breed = breedSelected.value.id.toString());
    }
  }
}
```

Fonte: Autoria própria (2023).

A função `saveAnimal` do *service* de animais, apresentada na Listagem 19 gera um `id` único baseado data e hora atual para ser utilizado como identificação de forma interna na aplicação móvel, pois, se trata de um dado registrado de maneira *offline*. Esse registro ainda não possui o `id` auto incrementável gerado automaticamente pelo banco de de dados da API. Após a geração do `id`, o *service* encaminha os dados a serem salvos para o *repository*. Já a função `editAnimal` apenas encaminha o objeto para a função de edição no *repository*, pois não é necessário fazer nenhuma validação a mais nos dados.

Na Listagem 20, o método `saveAnimalInDb` recebe o animal por parâmetro e adiciona na lista para salvar. Caso obtenha sucesso na busca e ao salvar os dados, retorna o valor `true`, caso contrário retorna o valor `false`. Já o método de `editAnimalInBd` recebe o objeto a ser editado, tenta realizar a busca dos animais já cadastrados e caso encontre o registro salva o novo animal com os dados recebidos.

Listagem 19 – Função `saveAnimal` no *service*

```
1  class AnimalServiceImpl implements AnimalService {
2    AnimalRepository _animalRepository;
3    Uuid _uuid;
4
5    AnimalServiceImpl({
6      required AnimalRepository animalRepository,
7      required Uuid uuid,
8    })   : _animalRepository = animalRepository,
9           _uuid = uuid;
10
11   @override
12   Future<bool> saveAnimal(AnimalModel animal) {
13     animal.internalId ??= _uuid.v1();
14
15     return _animalRepository.saveAnimalInDb(animal);
16   }
17
18   @override
19   Future<bool> editAnimal(AnimalModel animal) =>
20       _animalRepository.editAnimalInDb(animal);
21
22   @override
23   Future<bool> deleteAnimal(AnimalModel animal) =>
24       _animalRepository.deleteAnimal(animal);
25
26   ...
27 }
```

Fonte: Autoria própria (2023).

Listagem 20 – Funções de edição e salvar no *repository*

```
class AnimalRepositoryImpl implements AnimalRepository {
  final RestClient _restClient;
  final AuthService auth;
  late Box _box;
  AnimalRepositoryImpl({required RestClient restClient,
    required AuthService authService,
  })  : _restClient = restClient, auth = authService;
  @override
  Future<bool> saveAnimalInDb(AnimalModel animal) async {
    _box = await DatabaseInit().getInstance();
    var status = true;
    try {
      var animals = _box.get(ANIMALS) ?? [];
      animals.add(animal);
      _box.put(ANIMALS, animals);
    } catch (e) {
      print(e);
      status = false;
    }
    return status;
  }
  @override
  Future<bool> editAnimalInDb(AnimalModel animal) async {
    _box = await DatabaseInit().getInstance();
    var status = true;
    try {
      var animals = _box.get(ANIMALS) ?? [];
      List<AnimalModel> animalsList =
          animals != null ? List<AnimalModel>.from(animals as List) : [];
      List<AnimalModel> list = [];
      list.add(animal);
      AnimalModel? am = findAnimal(animalsList, animal);
      int? pos = null;
      if (am != null) {pos = animalsList.indexOf(am);}
      if (pos != null) {animalsList.replaceRange(pos, pos + 1, list);}
      _box.put(ANIMALS, animalsList);
    } catch (e) {
      status = false;
    }
    return status;
  }
  AnimalModel? findAnimal(List<AnimalModel> list, AnimalModel animal) {
    AnimalModel? am = list
    .firstWhereOrNull((element) => element.internalId == animal.internalId);
    return am;
  }
  ...
}
```

Fonte: Autoria própria (2023).

A função `removeAnimal`, apresentada na Listagem 21, recebe o animal que o usuário deseja remover ao clicar no botão "Remover". A tela com o botão que executa essa ação é exibida na Figura 15.

Listagem 21 – Função remover animal do controlador

```
removeAnimal(AnimalModel animal) async {
  var isRemoved = await _animalService.deleteAnimal(animal);
  Snack.show(
    content: isRemoved
        ? 'Sucesso ao remover animal'
        : 'Ocorreu um erro ao remover animal',
    snackType: isRemoved ? SnackType.success : SnackType.error,
    behavior: SnackBarBehavior.floating,
  );
  loadAnimals();
}
```

Fonte: Autoria própria (2023).

Como pode ser visto na Listagem 19, a função `deleteAnimal` apenas encaminha o objeto para o `repository`. No `repository`, a função responsável por apagar o animal da base de dados é chamada, exibida na Listagem 22, realizando a busca dos animais, removendo o objeto da lista e a salva novamente.

Listagem 22 – Função apagar animal do `repository`

```
@override
Future<bool> deleteAnimal(AnimalModel animal) async {
  _box = await DatabaseInit().getInstance();

  var status = true;
  try {
    var animals = _box.get(ANIMALS) ?? [];
    List<AnimalModel> animalsList =
        animals != null ? List<AnimalModel>.from(animals as List) : [];

    animalsList.remove(animal);

    _box.put(ANIMALS, animalsList);
  } catch (e) {
    print(e);
    status = false;
  }

  return status;
}
```

Fonte: Autoria própria (2023).

No cadastro de uma mastite, o formulário possui um controlador próprio, assim como os demais formulários e páginas. Neste controlador são definidas as propriedades utilizadas nos campos e definidos os grupos dos botões do tipo *radio*, como pode ser visualizado na Listagem 23. No método `onInit` do *controller* é verificado se é uma edição ou cadastro de uma mastite, com base no parâmetro passado pela tela anterior. Caso seja passado um valor por parâmetro os campos do formulário serão preenchidos pelo método `setFormValues`, que pode ser visto na Listagem 24.

Listagem 23 – *Controller* formulário mastite

```
1  class MastitisFormController extends GetxController {
2  final MastitisService? _mastitisService;
3
4  MastitisFormController({
5    required MastitisService mastitisService,
6  }) : _mastitisService = mastitisService;
7
8  final mastitis = MastitisModel().obs;
9  RxString typeMastitis = AnimalMastitisType.clinic.name.toString().obs;
10 RxString resultCmtADGroup = 'absent'.obs;
11 RxString resultCmtAEGroup = 'absent'.obs;
12 RxString resultCmtPDGroup = 'absent'.obs;
13 RxString resultCmtPEGroup = 'absent'.obs;
14
15 final dateController = TextEditingController();
16
17 @override
18 void onInit() async {
19   super.onInit();
20   if (data[0]['mastitis'] != null) {
21     setFormValues(data[0]['mastitis']);
22     mastitis.value = data[0]['mastitis'];
23   }
24 }
25 }
```

Fonte: Autoria própria (2023).

Listagem 24 – Adicionando valores no formulário

```
void setFormValues(MastitisModel values) {
    dateController.text = values.dateDiagnostic.toString();

    resultCmtADGroup.value = values.ad.toString();
    resultCmtAEGroup.value = values.ae.toString();
    resultCmtPDGroup.value = values.pd.toString();
    resultCmtPEGroup.value = values.pe.toString();
    typeMastitis.value = values.type.toString();
  }
```

Fonte: Autoria própria (2023).

O método `onFormSubmit`, apresentado na Listagem 25 verifica se é uma edição ou um cadastro de um novo registro e envia os dados recolhidos no formulário para os métodos correspondentes no *service* do módulo de mastite.

Listagem 25 – Método `onFormSubmit` formulário mastite

```
onFormSubmit() async {
    var isSaved = idxMastitis != null
        ? await _mastitisService!.editMastitis(mastitis.value)
        : await _mastitisService!.saveMastitis(mastitis.value);

    Snack.show(
      content: isSaved
          ? 'Sucesso ao salvar mastitiso'
          : 'Ocorreu um erro ao salvar mastitiso',
      snackType: isSaved ? SnackType.success : SnackType.error,
      behavior: SnackBarBehavior.floating,
    );

    Get.back(result: isSaved);
}
```

Fonte: Autoria própria (2023).

O módulo de sincronização segue a lógica estabelecida no fluxograma da Figura 3 e o código-fonte dessa funcionalidade pode ser visualizado na Listagem 27. No inicio do arquivo são injetados as dependências necessárias, visualizadas na Listagem 26. No método `onInit` é chamado o primeiro módulo a ser sincronizado. No método chamado é criado um objeto com o mesmo tipo de dados que a lista que será construída em tela, adicionado os valores das suas propriedades e em seguida adicionado na lista para sincronização.

Listagem 26 – Injeção dos módulos utilizados na sincronização

```
1   SyncForcedController({
2     required AuthService authService,
3     required PropertyService propertyService,
4     required BreedService breedService,
5     required AnimalService animalService,
6     required PlagueService plagueService,
7     required DiseaseService diseaseService,
8     required CultureService cultureService,
9     required VegetableDiseaseService vegetableDiseaseService,
10    required VegetablePlagueService vegetablePlagueService,
11    required SaleService saleService,
12    required PurchaseService purchaseService,
13    required PregnancyDiagnosisService pregnancyDiagnosisService,
14    required ProductService productService,
15    required MedicationService medicationService,
16    required MastitisService mastitisService,
17    required InseminationService inseminationService,
18    required DiseaseAnimalService diseaseAnimalService,
19  }) : _authService = authService,
20       _propertyService = propertyService,
21       _breedService = breedService,
22       _diseaseService = diseaseService,
23       _plagueService = plagueService,
24       _cultureService = cultureService,
25       _vegetableDiseaseService = vegetableDiseaseService,
26       _vegetablePlagueService = vegetablePlagueService,
27       _purchaseService = purchaseService,
28       _saleService = saleService,
29       _pregnancyDiagnosisService = pregnancyDiagnosisService,
30       _medicationService = medicationService,
31       _productService = productService,
32       _mastitisService = mastitisService,
33       _inseminationService = inseminationService,
34       _diseaseAnimalService = diseaseAnimalService,
35       _animalService = animalService;
```

Fonte: Autoria própria (2023).

Listagem 27 – Controlador do módulo de sincronização

```
class SyncForcedController extends GetxController {
    ... //Insejção dos módulos necessários para buscar os seus dados
    RxList<SyncModel> syncFinishedList = <SyncModel>[].obs;
    RxBool hasError = false.obs;
    @override
    void onInit() {
        super.onInit();
        loadProperties();
    }
    void loadProperties() async {
        SyncModel sync = SyncModel();
        sync.name = 'Propriedades';
        sync.status = -1;
        syncFinishedList.add(sync);
        try {
          await _propertyService.deleteAll();
          final propertiesData = await _propertyService
            .getAllPropertiesOnline();
          //Timeout para exibir o loading na tela
          await Future.delayed(
            const Duration(seconds: 1),
          );
          sync.status = 1;
        } catch (e) {
          sync.status = 0;
          sync.errorMessage = e.toString();
          hasError.value = true;
        }

        syncFinishedList.removeAt(syncFinishedList.length - 1);
        syncFinishedList.add(sync);
        loadDiseases();
    }

    //restante dos métodos
    ...
}
```

Fonte: Autoria própria (2023).

Por se tratar de uma variável reativa e o componente que engloba essa variável ser um `Obx()`, apresentado na Listagem 28, cada vez que essa lista ou os itens dessa lista tiverem alterações a tela será atualizada.

Listagem 28 – Visualização da lista de módulos sincronizados

```
Obx(() {
    return ListView.separated(
    separatorBuilder: (context, index) => Divider(thickness: 1.2),
    itemCount: controller.syncFinishedList.length,
    shrinkWrap: true,
    scrollDirection: Axis.vertical,
    itemBuilder: (context, index) {
      var synchronized = controller.syncFinishedList[index];
      return Row(
        crossAxisAlignment: CrossAxisAlignment.center,
        mainAxisAlignment: MainAxisAlignment.spaceBetween,
        children: [
            Text(
                synchronized.name.toString(),
                style: UIConfig.subtitleStyle,
            ),
            synchronized.status == -1
              ? SizedBox(
                  height: 20,
                  width: 20,
                  child: CircularProgressIndicator(
                    color: UIColors.primaryColor,
                    strokeWidth: 1.5,
                  ),
                )
              : synchronized.status == 1
                ? Icon(
                    Icons.check,
                    size: 30,
                    color: UIColors.successColor,
                  )
                : SizedBox(
                    height: 30.0,
                    width: 30.0,
                    child: IconButton(
                      padding: const EdgeInsets.all(0.0),
                      color: UIColors.warningColor,
                      icon: const Icon(
                        Icons.clear,
                        size: 30.0,
                      ),
                      onPressed: () => showExceptionDialog(),
                    ),
                  ),
        ],
      );
    },);
}),
```

Fonte: Autoria própria (2023).

Por se tratar da sincronização de maneira forçada, ou seja, só buscar os dados da API, o primeiro passo é remover os dados que estiverem armazenados localmente. Em seguida é chamado o método do *service* responsável que encaminha para o método do *repository* que realiza a requisição para o servidor, conforme pode ser visualizado na Listagem 30. Com o retorno da requisição, a função no próprio *service* do módulo é responsável por salvar os dados obtidos da API, como visto na Listagem 29. O objeto adicionado na lista anteriormente tem sua situação alterada de acordo com o sucesso ou exceção ocorrido na função, além de que em caso de exceção a mensagem de erro é adicionada ao objeto da lista, permitindo que o usuário visualize o erro, com mostrado na Figura 27 e Figura 29.

Listagem 29 – Método com requisição para o servidor

```
1     @override
2     Future<List<PropertyModel>> getAllPropertiesOnline() async {
3         List<PropertyModel> properties =
4             await _propertyRepository.getAllProperties();
5         saveProperties(properties);
6         return properties;
7     }
8
9     @override
10    Future<bool> saveProperties(List<PropertyModel> properties) {
11        for (var e in properties) {
12          {
13            e.internalId ??= _uuid.v1();
14          }
15        }
16
17        return _propertyRepository.savePropertiesInDb(properties);
18    }
```

Fonte: Autoria própria (2023).

Após a finalização do fluxo, interpretado na Figura 3, é partido para o próximo módulo a ser sincronizado, e assim por diante. Quando o ultimo módulo é finalizado, é exibido para o usuário uma notificação, apresentada na Figura 28.

Listagem 30 – Método com requisição para o servidor

```
@override
Future<List<PropertyModel>> getAllProperties() async {
  final result = await _restClient.get(
    'properties',
    headers: HeadersAPI(token: auth.apiToken()).getHeaders(),
    decoder: (data) {
      final resultData = data;
      if (resultData != null) {
        try {
          return resultData
              .map<PropertyModel>((p) => PropertyModel.fromMap(p))
              .toList();
        } catch (e) {
          throw Exception('Error _ $e');
        }
      } else {
        return <PropertyModel>[];
      }
    },
  );

  if (result.hasError) {
    print('Error [${result.statusText}]');
    throw Exception('Error _ ${result.body}');
  }

  return result.body ?? <PropertyModel>[];
}
```

Fonte: Autoria própria (2023).

Todas as entidades que permitem a edição e inserção de novos dados possuem a propriedade `isEdited`. Está propriedade salva um valor do tipo *boolean*, que tem como objetivo identificar quando uma entidade é editada ou uma nova é incluída, assim possibilitando identificar quais dados precisam ser atualizados de maneira *online*.

O módulo `sync_default` é responsável pela sincronização comum, representada no Figura 3, enviando os dados que foram editados ou incluídos de forma *offline* no aplicativo.

No módulo de sincronização comum cada entidade possui um método para realizar a sua sincronização. Na Listagem 31 é dado como exemplo a sincronização animal, neste método, é realizado todo fluxo interpretado Figura 3 juntamente com a interação com a tela do sistema.

Listagem 31 – Método de sincronização de animais

```
syncAnimals() async {
    SyncModel sync = SyncModel();
    sync.name = 'Animais';
    sync.statusSend = -1;
    sync.statusGet = -2;

    syncFinishedList.add(sync);

    try {
      final animalData = await _animalService.getAllAnimalsIfIsEdited();

      await _animalService.sendAnimals(animalData).then((value) async {
        if (value) {
          sync.statusGet = -1;
          sync.statusSend = 1;
          syncFinishedList.removeAt(syncFinishedList.length - 1);
          syncFinishedList.add(sync);

          await _animalService.deleteAll();
          final animalsData = await _animalService.getAllAnimalsOnline();
        } else {
          sync.statusSend = 0;
        }
      });

      sync.statusGet = 1;
    } catch (e) {
      sync.statusGet = 0;
      sync.statusSend = 0;
      sync.errorMessage = e.toString();
      hasError.value = true;
    }

    syncFinishedList.removeAt(syncFinishedList.length - 1);
    syncFinishedList.add(sync);
    syncInseminations();
  }
```

Fonte: Autoria própria (2023).

Na função apresentada na Listagem 32, são buscados todos os animais que possuem o atributo `isEdited` salva como `true`, ou seja, precisa enviado apra o sistema *online*.

Listagem 32 – Método para buscar animais alterados ou salvos

```
1  Future<List> getAllInseminationsIfIsEdited() async {
2    List<InseminationModel> inseminations = await getAllInseminations(null);
3
4    List inseminationsList = [];
5    for (var e in inseminations) {
6      if (e.isEdited != null) {
7        inseminationsList.add(e.toMap());
8      }
9    }
10   return inseminationsList;
11 }
```

Fonte: Autoria própria (2023).

Os animais encontrados no método Listagem 32 então são direcionados para o método `sendAnimals`, demonstrado na Listagem 33, aonde é verificado se a lista é vazia. Caso a lista seja vazia o sistema trata como sucesso por não possuir dados para serem enviados para a API, pois não houve dado da aplicação incluído ou editado. Caso a lista não seja vazia ela é dirigida para a função `postAnimals`.

Listagem 33 – Método para buscar animais alterados ou inseridos

```
1  Future<bool> sendAnimals(List animals) async {
2    if (animals.isEmpty) {
3      return Future.delayed(
4        const Duration(microseconds: 1),
5        () => true,
6      );
7    }
8
9    return await _animalRepository.postAnimals(animals);
10 }
```

Fonte: Autoria própria (2023).

A função `postAnimals`, exibida na Listagem 34, tem como objetivo realizar uma requisição HTTP enviado a lista de animais editados e/ou registrados *offline* no aplicativo para a API.

Listagem 34 – Método para enviar os dados para api

```
1  Future<bool> postAnimals(List animals) async {
2    final result = await _restClient.post(
3      'animals/sendAnimals',
4      jsonEncode(animals),
5      headers: HeadersAPI(token: auth.apiToken()).getHeaders(),
6      decoder: (data) {
7        return data;
8      },
9    );
10
11   if (result.status.code != HttpStatus.created &&
12       result.status.code != HttpStatus.ok) {
13     print('Error [${result.statusText}]');
14     throw Exception('Error _ ${result.body}');
15   }
16
17   return true;
18 }
```

Fonte: Autoria própria (2023).

Mesmo que o desenvolvimento da API API desenvolvida com Spring Framework e Java não tenha feito parte do escopo deste trablho, foram necessários o desenvolvimento de algumas funcionalidades na API. O método responsável por salvar a lista de animais na API é o `saveListAnimals`, demonstrado na Listagem 35. Neste método é recebido uma lista já tipada. Caso não ocorrer nenhuma exceção os dados desta lista são salvos no banco de dados do sistema, pra posteriormente estar disponível tanto para os usuários da aplicação móvel, quando os usuários de outros sistemas que consumirem dados da API.

Listagem 35 – Método da API salvar a lista de Animais

```
1  public boolean saveListAnimals(List<Animal> animals) {
2      boolean status = true;
3      try {
4          animalRepository.saveAll(animals);
5      } catch (Exception e){
6          status = false;
7          log.error(e.getMessage());
8      }
9
10     return status;
11 }
```

Fonte: Autoria própria (2023).

Quando na aplicação é realizada a inserção de um novo animal este não irá possuir o atributo id, por motivo de que esta propriedade é de responsabilidade do banco de dados online gerenciar. Desta maneira o *Create, Read, Update, Delete* (CRUD) dos submódulos de animais, como inseminações, mastites, identificação de doenças, são salvos referenciando a propriedade `animalIdentifier` do animal. Como consequência disto, quando for realizada a sincronização de algum destes submódulos é necessária que a API faça a identificação do animal com base no atributo `animalIdentifier`, para só então salvar a entidade na base de dados. Isso pode pode ser visto no exemplo do submódulo de inseminação, apresentado na Listagem 36.

Listagem 36 – Método da API salvar a lista de Inseminações

```
1  public boolean saveListInseminations(List<Insemination> inseminations) {
2      boolean status = true;
3      try {
4          for (Insemination insemination : inseminations) {
5
6              //Identifica o animal de acordo com o identifier
7              String animalIdentifier = insemination.getAnimal()
8                  .getIdentifier();
9              Animal animal = animalService.findByIdentifier(animalIdentifier);
10
11             insemination.setAnimal(animal);
12
13             inseminationRepository.save(insemination);
14         }
15     } catch (Exception e){
16         status = false;
17         log.error(e.getMessage());
18     }
19
20     return status;
21 }
```

Fonte: Autoria própria (2023).

Todos os módulos e submódulos da aplicação seguem esta mesma regra e fluxo para sincronização comum representadas nas funções das Listagens 31, 32, 33, 34, 35, 36, mas cada um com seus atributos individuais.

5 CONCLUSÃO

O objetivo deste trabalho foi o desenvolvimento de um aplicativo para dispositivos móveis com a capacidade de armazenar dados localmente sem a necessidade de conexão com à Internet para realizar o gerenciamento, manutenção e acompanhamento dos animais nas propriedades rurais com sincronização dos dados para uma API REST. O aplicativo é alimentado com dados dos animais, manutenções destes animais como identificação e registros de doenças, uso de medicamentos, inseminações, etc. Facilitando o acesso, armazenamento e consulta de dados utilizando dispositivos móveis, e permitindo posterior análise e tomada de decisão baseando-se nos dados coletados.

A utilização do Flutter em conjunto com o GetX oferece uma abordagem positiva no desenvolvimento de aplicativos móveis. O Flutter mostra-se um *framework* eficiente para a construção de interfaces de usuário, além de se tratar de um *framework* multiplataforma, o que aumenta a produtividade, pois permite escrever um único código-base para criar aplicativos para iOS e Android. Isso significa que os desenvolvedores não precisam escrever código separado para cada plataforma, economizando esforço e tempo. O GetX, por sua vez, facilita a gerência de estado, injeção de dependências e gerenciamento de rotas, permitindo um desenvolvimento ágil e organizado. Já o Hive oferece uma solução eficiente de armazenamento de dados local em dispositivos móveis, possibilitando uma manipulação simples e eficiênte dos dados do aplicativo. Com essa combinação de tecnologias, o desenvolvedor pode se concentrar no desenvolvimento das regras de negócio, utilizando-se ao máximo do reaproveitamento de código e simplificando a integração entre os módulos do sistema.

Um dos desafios ao usar o Hive como solução para o armazenamento de dados é lidar com o fato de não ser um banco de dados SQL tradicional. Ao contrário dos bancos de dados relacionais com os quais muitos desenvolvedores estão familiarizados, o Hive é uma solução de banco de dados NoSQL baseada em chave-valor. Isso requer uma abordagem diferente para modelar e manipular dados, o que acaba gerando mais trabalho para o desenvolvedor. É necessário entender as particularidades do Hive, como a necessidade de serialização e desserialização de objetos e as limitações de consultas e junções complexas. Porém, com um bom entendimento dessas funcionalidades e da capacidade do Hive, é possível superar esses desafios e aproveitar os benefícios que essa solução oferece, como desempenho otimizado e facilidade de integração com o Flutter.

Conforme os requisitos funcionais e não-funcionais levantados junto ao IDR, também analisando as planilhas utilizadas nas manutenções dos animais e com base nas conversas tidas com os representantes da instituição, pode-se concluir que o sistema busca ao máximo atender aos objetivos de gerenciar os animais e realizar as manutenções da propriedade, abrangendo aspectos como inseminações, reprodução, pesos, identificações de doenças, usos de medicamentos, identificação de mastite. Além disso, o sistema oferece a capacidade de armazenar os dados de maneira *offline*, permitindo que sejam realizadas as manutenções dos

animais mesmo sem conexão com a internet. Posteriormente, esses dados podem ser sincronizados com o restante do sistema, garantindo a integridade das informações.

No futuro, o sistema tem o potencial de receber melhorias e novas integrações de acordo com as necessidades da instituição. Essas melhorias podem incluir a adição de integrações ou implementações de outras ferramenta e sistemas que podem vir a ser utilizados pela instituição, como um módulo financeiro e estoque, a fim de otimizar e facilitar ainda mais o gerenciamento e manutenções das propriedades. Além disso, uma melhoria que pode ser considerada é a capacidade de gerar relatórios diretamente na aplicação, permitindo que os usuários tenham acesso rápido e fácil às informações essenciais. Ainda, uma versão específica para o dono da propriedade pode ser desenvolvida, proporcionando uma visão mais ampla e abrangente das atividades e resultados relacionados as suas terras. Essas possíveis melhorias garantiriam um sistema ainda mais completo e adaptado às necessidades específicas da instituição e de seus usuários.

REFERÊNCIAS

ARRIGONI, M. D. B. *et al.* NÍveis elevados de concentrado na dieta de bovinos em confinamento. **Veterinária e Zootecnia**, dez. 2013.

BETT, V. **Nutrição de Precisão para Vacas Leiteiras: Desenvolvimento do software e aprimoramento e manutenção das planilhas de balanceamento de rações**. 2022.

BOTEGA, J. V. L. *et al.* DiagnÓstico da automaÇÃo na produÇÃo leiteira. **Ciência e Agrotecnologia**, abr. 2007.

CNA. **Pesquisa Pecuária Municipal 2020**. 2021. Disponível em: https://cnabrasil.org.br/storage/arquivos/Comunicado-Tecnico-CNA-ed-30_2021.pdf. Acesso em: 24 out. 2022.

CNA. **PIB do agronegócio cresceu abaixo das projeções.** 2022. Disponível em: https://www.cepea.esalq.usp.br/upload/kceditor/files/Cepea_CNA_PIB_JAn_Dez_2021_Mar%C3%A7o2022.pdf. Acesso em: 24 out. 2022.

CORRêA, C. C.; VELOSO, A. F.; BARCZSZ, S. S. Sober - sociedade brasileira de economia admintração e sociologia rural. *In*: **DIFICULDADES ENFRENTADAS PELOS PRODUTORES DE LEITE: UM ESTUDO DE CASO REALIZADO EM UM MUNICÍPIO DE MATO GROSSO DO SUL**. [*S.l.*: *s.n.*], 2010.

GONÇALVES, B. P.; MENDES, O. L. Princípio da inversão de dependência na qualidade de software: Aplicação da injeção de dependência no desenvolvimento de software. **Revista Interface Tecnológica**, 2022. Disponível em: https://revista.fatectq.edu.br/interfacetecnologica/article/download/1362/746.

IBGE. **Produção da pecuária municipal 2017**. 2018. Disponível em: https://biblioteca.ibge.gov.br/visualizacao/periodicos/84/ppm_2017_v45_br_informativo.pdf. Acesso em: 24 out. 2022.

MARTINS, P. R. G. *et al.* Produção e qualidade do leite em sistemas de produção da região leiteira de pelotas, RS, brasil. **Ciência Rural**, FapUNIFESP (SciELO), v. 37, n. 1, p. 212–217, fev. 2007. Disponível em: https://doi.org/10.1590/s0103-84782007000100034.

MDA. **O que é a agricultura familiar**. 2019. Disponível em: https://www.gov.br/agricultura/pt-br/assuntos/agricultura-familiar/agricultura-familiar-1. Acesso em: 24 out. 2022.

MERTENS, D. R. Predicting intake and digestibility using mathematical models of ruminal function. **Journal of Animal Science**, Oxford University Press (OUP), v. 64, n. 5, p. 1548–1558, maio 1987. Disponível em: https://doi.org/10.2527/jas1987.6451548x.

PONTES, T. B.; ARTHAUD, D. D. B. METODOLOGIAS áGEIS PARA o DESENVOLVIMENTO DE SOFTWARES. **Ciência e Sustentabilidade**, REVISTA CIENCIA E SUSTENTABILIDADE, v. 4, n. 2, p. 173–213, mar. 2019. Disponível em: https://doi.org/10.33809/2447-4606.422018173-213.

RIBEIRO, M.; FRANCISCO, R. Web services rest conceitos, análise e implementação. jun. 2016. Disponível em: https://asetore.ifba.edu.br/etc/article/view/25.

SALMAN, A. K.; OSMARI, E.; SANTOS, M. G. R. D. S. **Manual de Formulação de Ração para Vacas Leiteiras**. 2011.

SALMAN, A. K. D. *et al.* **Manual de Formulação de Ração para Vacas Leiteiras - 2° Edição**. 2020.

SOUZA, M. P. de; AMIN, M. M.; GOMES, S. T. Agronegócio do leite: características da cadeia produtiva do estado de rondônia. **Revista de Administração e Negócios da Amazônia**, ago. 2009.

TOMICH, T. R. *et al.* **Nutrição de precisão na pecuária leiteira**. 2015. Disponível em: https://ainfo.cnptia.embrapa.br/digital/bitstream/item/139557/1/Cnpgl-2015-CadTecVetZoot-Nutricao.pdf. Acesso em: 24 out. 2022.

VILELA, D. *et al.* **Pecuária de leite no Brasil : cenários e avanços tecnológicos**. 2016. Disponível em: https://ainfo.cnptia.embrapa.br/digital/bitstream/item/164236/1/Pecuaria-de-leite-no-Brasil.pdf. Acesso em: 25 out. 2022.

ZANIN, E.; HENRIQUE, D. S.; FLUCK, A. C. Avaliação de equações para estimar o consumo de vacas leiteiras. **Revista Brasileira de Saúde e Produção Animal**, FapUNIFESP (SciELO), v. 18, n. 1, p. 76–88, mar. 2017. Disponível em: https://doi.org/10.1590/s1519-99402017000100008.