PONTIFICIA UNIVERSIDADE CATOLICA DE GOIAS

ESCOLA POLITÉCNICA E DE ARTES

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Lucas Barbosa Antonelli

SISTEMA DE GESTÃO DO AVIÁRIO CAMPO ALEGRE

GOIÂNIA

2025

LUCAS BARBOSA ANTONELLI

SISTEMA DE GESTÃO DO AVIÁRIO CAMPO ALEGRE

Trabalho de Conclusão de Curso apresentado à Escola Politécnica e Artes, da Pontifícia Universidade Católica de Goiás, como parte dos resultados requisitos para a obtenção do título de Bacharel em Ciência da Computação.

Orientador:

Prof. Me. Fabricio Schlag

Banca examinadora:

Prof. Me. Fernando Gonçalves Abadia

Prof. Me. Rafael Leal Martins

GOIÂNIA

2025

LUCAS BARBOSA ANTONELLI

SISTEMA DE GESTÃO DO AVIÁRIO CAMPO ALEGRE

Trabalho de Conclusão de Curso aprovado em sua forma final pela Escola Politécnica e Artes, da Pontifícia Universidade Católica de Goiás, para obtenção do título de Bacharel em Ciência da Computação, em 01/12/2025.

_____

Orientador: Prof. Me. Fabricio Schlag

_____

Prof. Me. Fernando Gonçalves Abadia

_____

Prof. Me. Rafael Leal Martins

GOIÂNIA

2025

RESUMO

Este trabalho tem como objetivo desenvolver um sistema web para gestão de um aviário de pequeno porte, o Campo Alegre. O projeto propõe a criação de uma aplicação monolítica desenvolvida em Laravel e PostgreSQL, estruturada sob a arquitetura MVC (*Model-View-Controlle*r), de forma a garantir modularidade, escalabilidade e facilidade de manutenção. O sistema permite o gerenciamento de lotes, rações, refeições, vacinas, vendas e gastos, além de possibilitar a emissão de relatórios de desempenho. A fundamentação teórica contempla conceitos de arquiteturas de software, padrões de projeto e metodologias de desenvolvimento web. Por meio dos diagramas de caso de uso, BPMN e de sequência, são descritas as principais interações entre os componentes do sistema. O sistema visa automatizar processos e aprimorar a gestão de pequenos empreendimentos rurais, promovendo eficiência nas atividades diárias.

Palavras-chave: Sistema Web; Arquitetura MVC; Monólito; Gestão de Aviário; Laravel.

ABSTRACT

This study aims to develop a web-based system for managing a small-scale poultry farm, Campo Alegre. The project proposes the creation of a monolithic application built with Laravel and PostgreSQL, structured under the Model-View-Controller (MVC) architecture to ensure modularity, scalability, and ease of maintenance. The system enables the management of batches, feed, meals, vaccines, sales, and expenses, as well as the generation of performance reports. The theoretical foundation covers concepts related to software architectures, design patterns, and web development methodologies. Through use cases, BPMN, and sequence diagrams, the main interactions among the system's components are described. The system seeks to automate processes and improve the management of small rural enterprises, promoting greater efficiency in daily operational activities.

Keywords: Web System; MVC Architecture; Monolith; Poultry Farm Management; Laravel.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

O avanço das Tecnologias da Informação e Comunicação (TIC) têm impactado diretamente a forma como a sociedade organiza seus processos, desde atividades empresariais até o cotidiano de pequenos empreendedores rurais. Essas tecnologias proporcionam meios para a automatização e otimização de tarefas, permitindo maior precisão, eficiência e controle sobre informações essenciais ao funcionamento de qualquer empreendimento. Segundo Silva e Teixeira (2020), as tecnologias têm alterado de maneira significativa o modo como a sociedade se organiza, promovendo mudanças estruturais e ampliando as possibilidades de gestão e inovação.

No contexto rural, a adoção de soluções tecnológicas ainda é limitada, principalmente entre pequenos produtores que enfrentam desafios como falta de acesso a ferramentas digitais, altos custos de implementação e escassez de conhecimento técnico. Diante disso, a informatização de processos agrícolas e zootécnicos representa uma oportunidade de modernização acessível e eficiente. A gestão de um aviário, por exemplo, demanda controle rigoroso sobre dados como alimentação, vacinação, lotes e custos operacionais, tornando-se um cenário propício para a aplicação de um sistema computacional.

O presente trabalho propõe o desenvolvimento de um sistema web monolítico voltado à gestão de um aviário de pequeno porte, denominado Aviário Campo Alegre. O objetivo é criar uma aplicação capaz de centralizar e automatizar o gerenciamento de informações vitais para o funcionamento diário da propriedade, proporcionando ao produtor uma visão clara e integrada das operações. Para isso, será utilizada a arquitetura MVC (*Model-View-Controller*), amplamente empregada no desenvolvimento de sistemas web, a fim de garantir modularidade, escalabilidade e facilidade de manutenção.

A escolha pelo modelo monolítico se justifica pela natureza do projeto, que visa atender a um único cliente com necessidades específicas, sem demandar complexas integrações entre múltiplos serviços. Além disso, a implementação em um servidor doméstico reduz custos de hospedagem e possibilita ao produtor o controle direto sobre sua infraestrutura tecnológica.

Assim, este projeto busca responder à seguinte questão de pesquisa: é possível desenvolver uma aplicação web que facilite e inove o cotidiano de um pequeno empresário rural, aprimorando a gestão de seu aviário de forma acessível e eficiente? A partir dessa questão, estabelecem-se como objetivos o desenvolvimento, a validação e a implementação de um sistema funcional que atenda às necessidades levantadas durante a análise de requisitos junto ao cliente final.

O trabalho está estruturado em capítulos que abordam desde os conceitos teóricos e tecnológicos que fundamentam o projeto até a metodologia de desenvolvimento, modelagem do sistema, testes e resultados esperados. Espera-se que o produto contribua significativamente para a informatização de empreendimentos rurais e sirva de base para futuras soluções tecnológicas aplicadas ao agronegócio.

2 OBJETIVOS

2.1 OBJETIVO GERAL

Desenvolvimento e validação de um Sistema Web de controle e gerenciamento avícola que centralize, organize e automatize o registro de dados vitais de produção, visando a melhoria da eficiência operacional, a otimização da tomada de decisão e a rastreabilidade das informações do aviário.

2.2 OBJETIVOS ESPECÍFICOS

- Especificação e modelagem dos requisitos funcionais e não-funcionais do sistema, contemplando os módulos de controle de alimentação, mortalidade e produção.
- Implementar a arquitetura da aplicação web (monolítica), utilizando as stacks PHP/Laravel e as tecnologias web padrão no frontend, garantindo a persistência dos dados no banco de dados PostgreSQL.
- Elaborar a documentação técnica e o manual do usuário do sistema, detalhando a estrutura, implantação (deploy) e a utilização da ferramenta.
- Realizar testes de funcionalidade e usabilidade (Validação) da aplicação em um ambiente simulado, demonstrando a eficácia e a precisão no gerenciamento das informações avícolas.

2.3 RESULTADOS ESPERADOS

Os resultados deste trabalho poderão auxiliar nas seguintes considerações:

1. Validação de um produto de software, que sirva como ferramenta prática para a informatização da gestão em pequenos aviários.
2. A demonstração da viabilidade técnica de sistemas monolíticos, acessíveis e de baixo custo, para o agronegócio de pequena escala.
3. A contribuição para a literatura técnica sobre o uso do framework Laravel e do SGBD PostgreSQL na criação de soluções específicas para a área.

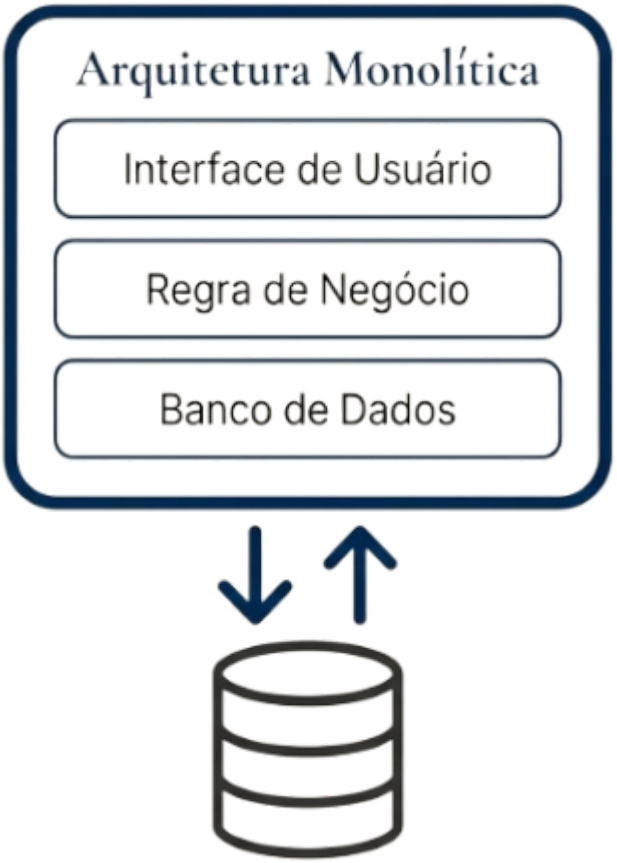
3 REFERENCIAL TEÓRICO

Neste capítulo são apresentados com maior profundidade, os conceitos utilizados durante o desenvolvimento deste trabalho, assim como padrões de projeto, arquiteturas de software e técnicas que foram utilizadas para alcançar o objetivo proposto.

3.1 SISTEMAS MONOLÍTICOS

A arquitetura monolítica é definida por uma aplicação de software onde todas as suas funcionalidades, incluindo *backend*, *frontend* e lógica de negócio, são agrupadas em uma única unidade de deployment e residem em um único repositório de código, conforme ilustrada na Figura 1. Este modelo, no qual múltiplos serviços estão estritamente agrupados, permite que a comunicação entre seus componentes seja direta e rápida. Conforme Dragoi et al. (2019, p. 1), as aplicações monolíticas podem se comunicar com sistemas externos através de métodos variados, como *web services*, páginas HTML ou APIs REST. A simplicidade inicial de desenvolvimento é uma vantagem inerente, porém, para aplicações de grande porte, o alto acoplamento e a complexidade do código podem dificultar significativamente a manutenção e a escalabilidade (DRAGOI ET AL., 2019).

Figura 1 – Representação da Arquitetura Monolítica. Fonte: Autoria Própria.



Contudo, ao se tratar de uma pequena aplicação que busca solucionar um problema recorrente e que demanda um rápido desenvolvimento, apesar das desvantagens apresentadas anteriormente, sistemas monolíticos se tornam uma ótima opção para este cenário.

3.2 ARQUITETURA MVC (*MODEL-VIEW-CONTROLLER*)

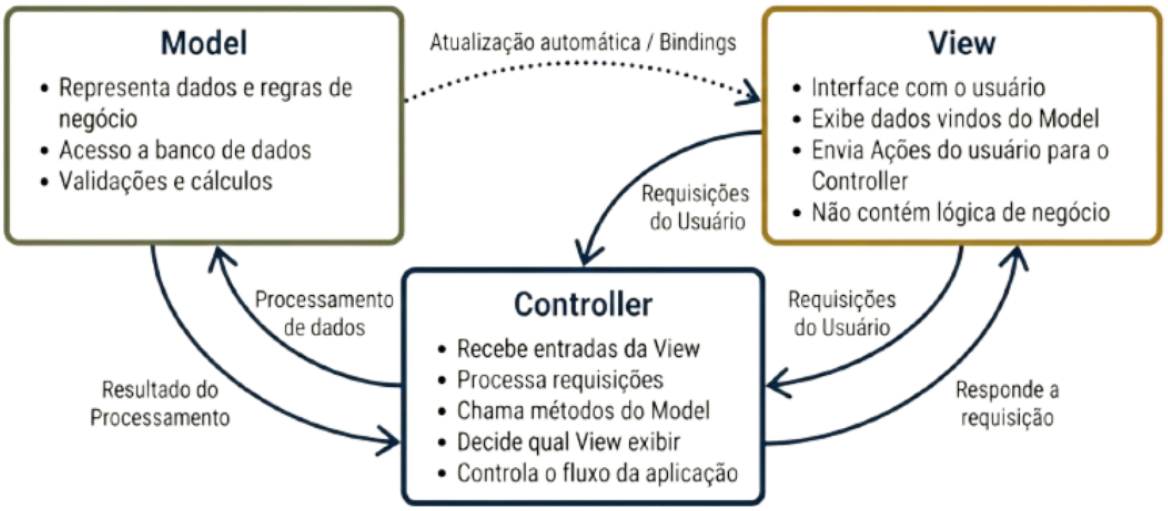
Model view controller (MVC) é um padrão arquitetural de software usado em aplicações web. Ele fornece três camadas principais: modelo (*model*), visão (*view*) e controlador (*controller*). O padrão MVC é amplamente difundido no mercado de trabalho e presente em diferentes aplicações. A arquitetura fornece três tipos de classes. Os *Models*, responsáveis por implementar a lógica de dados, são utilizados para manipular os dados do banco de dados atrelado a aplicação. As *Views*, usadas para implementar a interface da aplicação, responsável pela interação do usuário

com a aplicação. Os *Controllers*, utilizadas para realizar a comunicação entre *Views* e *Controllers*, isso é, responder às requisições do usuário. Essas classes trabalham em conjunto com os *Models* para selecionar as *views* apropriadas para cada requisição feita pelo usuário (MAJEED; RAUF, 2018).

A arquitetura deve e pode ser utilizada para separar os princípios da aplicação, isso é, ela facilita o desacoplamento da lógica empregada em cada classe do sistema. Separando, portanto, lógicas para renderização de interface para o usuário de lógicas complexas que serão responsáveis por cálculos dentro da aplicação, ou até mesmo operações e consultas realizadas no banco de dados. (MAJEED; RAUF, 2018).

A seguir, na Figura 2, evidencia as características de cada camada e a forma como se relacionam.

Figura 2 – Descrição MVC. Fonte: Autoria Própria.



3.3 PADRÕES DE PROJETO

Padrões de projeto são soluções desenvolvidas durante anos de experiências para problemas recorrentes na implementação de softwares. Cada padrão descreve uma solução para um determinado problema, de forma que não seja necessário a implementação desta solução mais de uma vez no software. Existem diferentes tipos de padrões de projeto, e cada um visa solucionar um problema diferente, seja

encontrar um determinado objeto ou especificar a implementação de interfaces (ASGHAR; ALAM; JAVED, 2019).

A aplicação de um determinado padrão de projeto para um problema depende inteiramente do conhecimento e experiência do desenvolvedor que o está utilizando. Em geral, os padrões de projeto buscam a modularização, consistência e reusabilidade do código. Existem três tipos de padrões de projeto: Estruturais, Criacionais e de Comportamento. O padrão estrutural está relacionado à composição de objetos, o padrão criacional está relacionado à criação dos objetos ou classes, e o padrão comportamental está relacionado a como as classes interagem entre si (ASGHAR; ALAM; JAVED, 2019).

Nesse contexto, destaca-se a importância de princípios arquiteturais associados aos padrões de projeto, como o *Single Responsibility Principle (SRP)*. De acordo com Martin (2018), o SRP afirma que cada classe ou módulo deve possuir apenas um motivo para mudar, ou seja, uma única responsabilidade claramente definida. Esse princípio reduz o acoplamento interno, evita a concentração excessiva de lógica em um único componente e facilita a manutenção e evolução do software. Em sistemas web modernos, especialmente aqueles estruturados em camadas, o SRP é fundamental para garantir que o código permaneça legível e sustentável ao longo do tempo.

Outro conceito essencial para a organização de sistemas robustos é o *Separation of Concerns* (SoC). Segundo Pressman e Maxim (2016), o SoC defende a divisão do sistema em partes distintas, de modo que cada parte seja responsável por uma preocupação específica. Essa separação contribui para maior clareza arquitetural, reduz o impacto de mudanças e permite que funcionalidades sejam desenvolvidas e testadas de forma independente. A aplicação desse princípio é particularmente evidente em frameworks como Laravel, nos quais a distinção entre camadas — rotas, controladores, serviços, modelos e views — sustenta a manutenibilidade do sistema.

A utilização de Services como padrão arquitetural reforça diretamente tanto o SRP quanto o SoC. Em arquiteturas baseadas em serviços internos (*Service Layer*), como descrevem Fowler (2003) e Evans (2004), classes de serviço encapsulam

regras de negócio e operações complexas, retirando essa responsabilidade dos controladores e modelos. Essa camada atua como intermediária entre a interface do usuário e o domínio, centralizando lógicas que não pertencem diretamente às entidades persistidas nem aos componentes de apresentação. Com isso, obtém-se um código mais modular, testável e alinhado às boas práticas de design orientado a domínio.

A adoção combinada de padrões de projeto, princípios SOLID — em especial o SRP — e a separação de responsabilidades por meio de Services contribui, portanto, para a criação de softwares mais coesos, reutilizáveis e de fácil evolução. Esses fundamentos oferecem suporte teórico e prático para o desenvolvimento de aplicações monolíticas bem estruturadas, garantindo que mesmo sistemas centralizados mantenham clareza arquitetural e flexibilidade para futuras expansões.

4 ASPECTOS METODOLÓGICOS

4.1 AMBIENTE DE DESENVOLVIMENTO

O Visual Studio Code, juntamente com Laravel 12 e o PostgreSQL foram utilizados para o desenvolvimento da aplicação, juntamente com linguagens como PHP, HTML, SCSS, Javascript e a biblioteca JQuery.

4.2 ESCOPO DO PROJETO

O projeto tem como objetivo o desenvolvimento de uma solução web escalável para o monitoramento e controle de um aviário para um pequeno produtor rural. Teremos como lista de Requisitos Funcionais os seguintes:

- RF01 – Autenticar Login
- RF02 – Gerenciar Lote
- RF03 – Gerenciar Refeições
- RF04 – Gerenciar Rações
- RF05 – Gerenciar Vendas
- RF06 – Gerenciar Vacinas
- RF07 – Emitir Relatórios
- RF08– Gerenciar Gastos

Note que os requisitos funcionais que envolvem o gerenciamento de algum modelo da aplicação, podem envolver alguns, ou todos, os elementos do famoso tipo de projeto CRUD (*Create*, *Read*, *Update* e *Delete*), que envolvem a criação, leitura, edição e possibilidade de deletar determinado elemento do projeto. Como Requisitos Não funcionais, foram levantados:

- RNF01 – Persistência de Dados
- RNF02 – Privacidade e Segurança
- RNF03 – Interface Web

Os Casos de uso da aplicação seguem os requisitos funcionais como base, sendo listados em:

- UC01 – Autenticar Login
- UC02 – Gerenciar Lote

- UC03 – Gerenciar Refeições
- UC04 – Gerenciar Rações
- UC05 – Gerenciar Vendas
- UC06 – Gerenciar Vacinas
- UC07 – Emitir Relatórios
- UC08 – Gerenciar Gastos

4.3 DEFINIÇÃO DOS CASOS DE USO

Nesta seção são apresentadas as definições dos casos de uso do sistema, com o propósito de descrever as principais funcionalidades e interações entre os atores e a aplicação. Cada caso de uso foi identificado a partir do processo de elicitação e análise de requisitos realizados junto ao cliente final, garantindo que as necessidades do usuário fossem devidamente contempladas.

Além da descrição textual, são incluídas as representações visuais em forma de diagramas de caso de uso, que ilustram de maneira clara e objetiva as relações entre os atores e as operações do sistema. Essas representações permitem uma visão geral do comportamento funcional da aplicação, servindo como base para o entendimento e o desenvolvimento das demais etapas do projeto.

4.3.1 UC01 - Autenticar login

4.3.1.1 Cenário

O usuário acaba de acessar o sistema.

4.3.1.2 Descrição

A aplicação cliente permite que o usuário insira seu e-mail e senha. Assim, fazendo uma requisição HTTP para o Servidor, em sequência o Servidor verifica com o banco de dados, e se esse administrador existir, criará a sessão de login para aquele usuário e prosseguirá para a tela inicial.

4.3.1.3 Atores

Usuário, Aplicação Cliente, Banco de Dados, Servidor.

4.3.1.4 Fluxo Principal

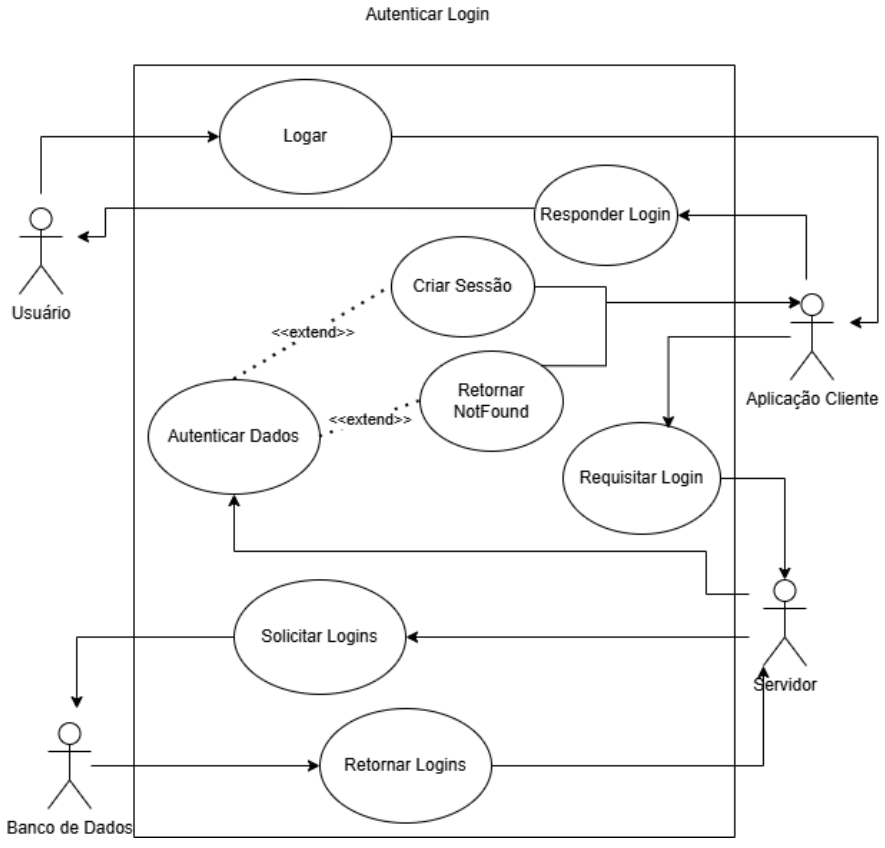
O Usuário insere corretamente suas informações de administrador e o sistema verifica com o Banco de Dados. Em sequência, é retornado para a tela inicial e aquele usuário tem sua sessão criada.

4.3.1.5 Fluxo Alternativo

O Usuário insere informações incorretas e o sistema não encontra essas informações no Banco de Dados. Em sequência, é retornado um erro de *NotFound* para a aplicação cliente.

4.3.1.6 Representação

Figura 3 – Diagrama UC01 – Autenticar Login. Fonte: Autoria Própria.



4.3.2 UC02 – Gerenciar lote

4.3.2.1 Cenário

Este caso de uso ocorre após UC01 - Autenticar Login.

4.3.2.2 Descrição

O Usuário irá enviar um conjunto de informações de acordo com a forma de gerência (Cadastro, Listagem ou Atualização) que deseja. A aplicação cliente então fará uma requisição HTTP correspondente com a ação desejada para o Servidor.

4.3.2.3 Ator

Usuário, Aplicação Cliente, Banco de Dados, Servidor.

4.3.2.4 Fluxo Principal

O Usuário insere corretamente as informações necessárias para a forma de gerência desejada e a aplicação cliente valida os tipos de informações, enviando assim o token CSRF juntamente com essas informações por uma requisição HTTP. A API então processa determinada requisição se comunicando com o Banco de Dados, retornando os dados obtidos ou uma confirmação de operação, dependendo da requisição.

4.3.2.5 Fluxo Alternativo

- **Usuário não está logado:**

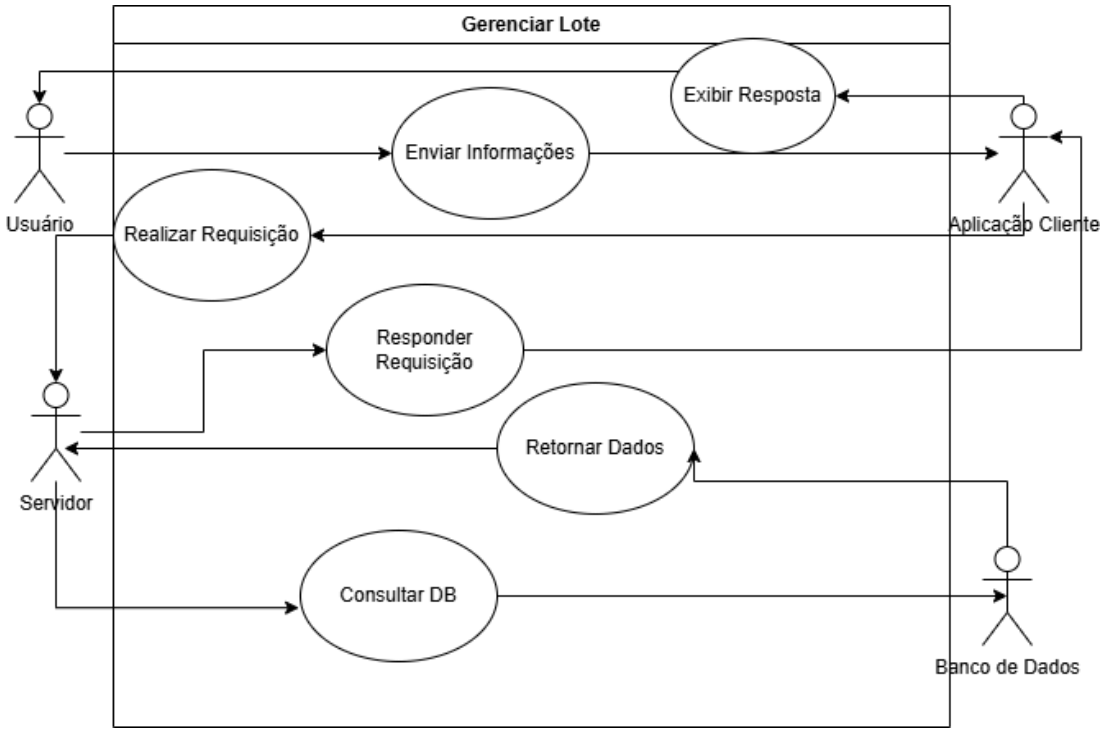
A aplicação cliente não reconhece o login do usuário e solicita que ele realize o login através do UC01 – Autenticar Login.

- **Usuário manda informações erradas:**

A aplicação cliente valida as informações e retorna um erro para o usuário.

4.3.2.6 Representação

Figura 4 – Diagrama UC02 – Gerenciar Lote. Fonte: Autoria Própria.



4.3.3 UC03 – Gerenciar refeições

4.3.3.1 Cenário

Este caso de uso ocorre após UC01 – Autenticar Login.

4.3.3.2 Descrição

O Usuário irá enviar um conjunto de informações de acordo com a forma de gerência (Cadastro de refeições para um lote ou Listagem de refeições) que deseja. A aplicação cliente então fará uma requisição HTTP correspondente com a ação desejada para o Servidor.

4.3.3.3 Ator

Usuário, Aplicação Cliente, Banco de Dados, Servidor.

4.3.3.4 Fluxo Principal

O Usuário insere corretamente as informações necessárias para a forma de gerência desejada e a aplicação cliente valida os tipos de informações, enviando

assim o token CSRF juntamente com essas informações por uma requisição HTTP. O Servidor então processa determinada requisição se comunicando com o Banco de Dados, retornando os dados obtidos ou uma confirmação de operação, dependendo da requisição.

4.3.3.5 Fluxo Alternativo

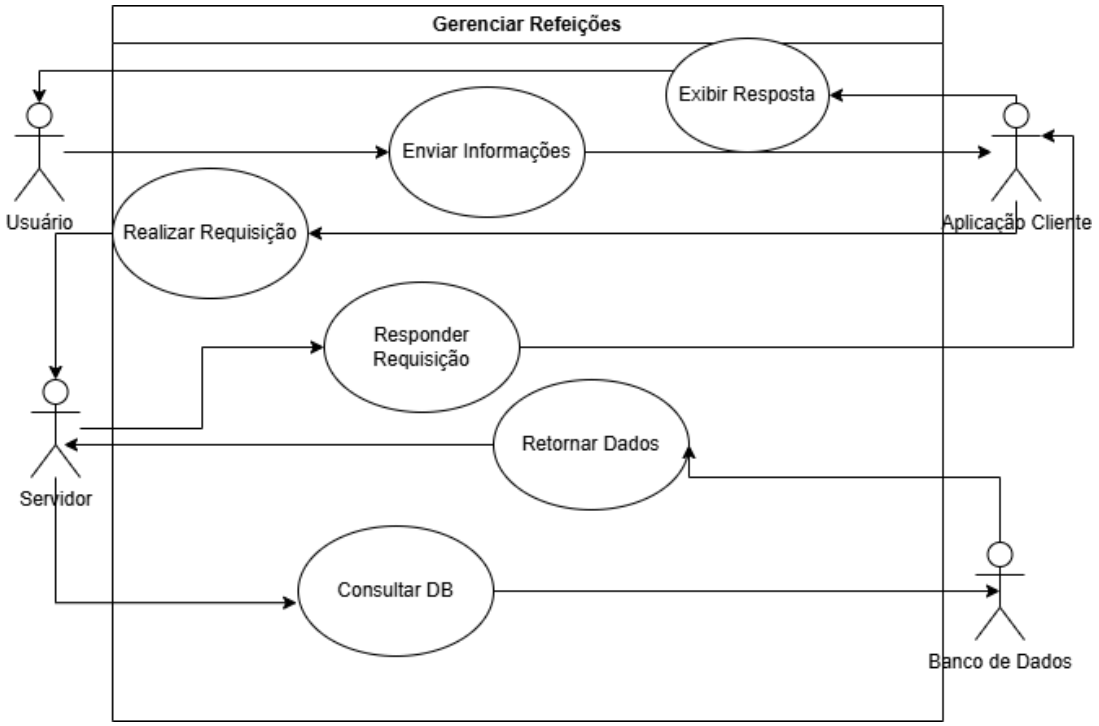
- **Usuário não está logado:**

A aplicação cliente não reconhece o login do usuário e solicita que ele realize o login através do UC01 - Autenticar Login.

- **Usuário manda informações erradas:**
  A aplicação cliente valida as informações e retorna um erro para o Usuário.

4.3.3.6 Representação

Figura 5 – Diagrama UC03 – Gerenciar Refeições. Fonte: Autoria Própria.



4.3.4 UC04 – Gerenciar Rações

4.3.4.1 Cenário

Este caso de uso ocorre após UC01 – Autenticar Login.

4.3.4.2 Descrição

O Usuário irá enviar um conjunto de informações de acordo com a forma de gerência (Cadastro, Listagem ou Atualização) que deseja. A aplicação cliente então fará uma requisição HTTP correspondente com a ação desejada para o Servidor.

4.3.4.3 Ator

Usuário, Aplicação Cliente, Banco de Dados, Servidor.

4.3.4.4 Fluxo Principal

O Usuário insere corretamente as informações necessárias para a forma de gerência desejada e a aplicação cliente valida os tipos de informações, enviando assim o token CSRF juntamente com essas informações por uma requisição HTTP. O Servidor então processa determinada requisição se comunicando com o Banco de Dados, retornando os dados obtidos ou uma confirmação de operação, dependendo da requisição.

4.3.4.5 Fluxo Alternativo

- **Usuário não está logado:**

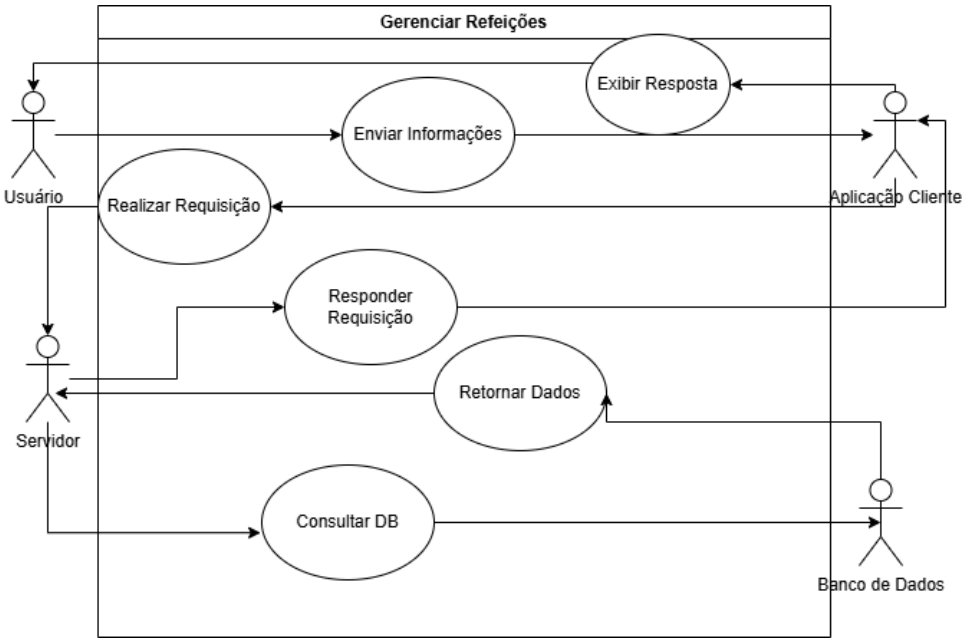
A aplicação cliente não reconhece o login do usuário e solicita que ele realize o login através do UC01 – Autenticar Login.

- **Usuário manda informações erradas:**

A aplicação cliente valida as informações e retorna um erro para o usuário.

4.3.4.6 Representação

Figura 6 – Diagrama UC04 – Gerenciar Rações. Fonte: Autoria Própria.



4.3.5 UC05 – Gerenciar Vendas

4.3.5.1 Cenário

Este caso de uso ocorre após UC01 – Autenticar Login.

4.3.5.2 Descrição

O Usuário irá enviar um conjunto de informações de acordo com a forma de gerência (Cadastrar uma venda para um lote, ou listar as vendas de um lote) que deseja. A aplicação cliente então fará uma requisição HTTP correspondente com a ação desejada para o Servidor.

4.3.5.3 Ator

Usuário, Aplicação Cliente, Banco de Dados, Servidor.

4.3.5.4 Fluxo Principal

O Usuário insere corretamente as informações necessárias para a forma de gerência desejada e a aplicação cliente valida os tipos de informações, enviando assim o token CSRF juntamente com essas informações por uma requisição HTTP.

O Servidor então processa determinada requisição se comunicando com o Banco de Dados, retornando os dados obtidos ou uma confirmação de operação, dependendo da requisição.

4.3.5.5 Fluxo Alternativo

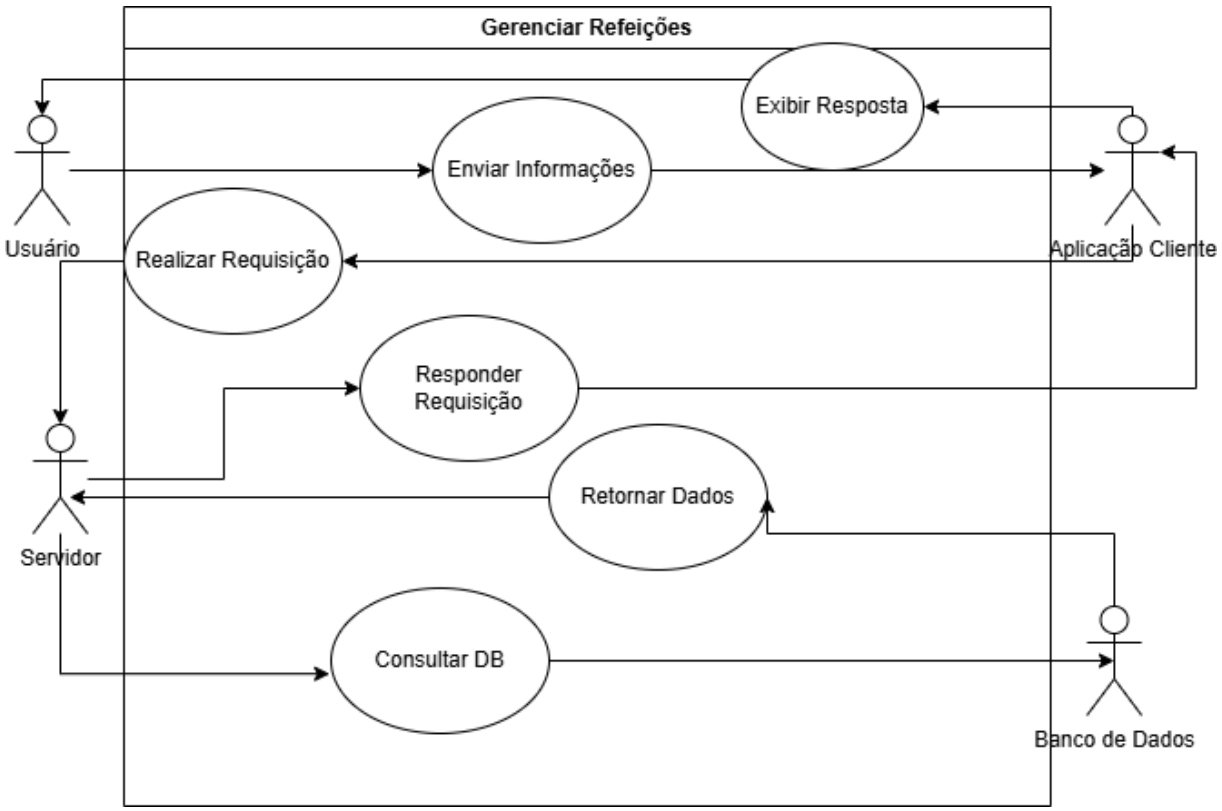
- **Usuário não está logado:**

A aplicação cliente não reconhece o login do usuário e solicita que ele realize o login através do UC01 – Autenticar Login.

- **Usuário manda informações erradas**
  A aplicação cliente valida as informações e retorna um erro para o usuário.

4.3.5.6 Representação

Figura 7 – Diagrama UC05 – Gerenciar Vendas. Fonte: Autoria Própria.



4.3.6 UC06 – Gerenciar Vacinas

4.3.6.1 Cenário

Este caso de uso ocorre após UC01 – Autenticar Login.

4.3.6.2 Descrição

O Usuário irá enviar um conjunto de informações de acordo com a forma de gerência (Cadastrar uma vacina ou listar vacinas) que deseja. A aplicação cliente então fará uma requisição HTTP correspondente com a ação desejada para o Servidor.

4.3.6.3 Ator

Usuário, Aplicação Cliente, Banco de Dados, Servidor.

4.3.6.4 Fluxo Principal

O Usuário insere corretamente as informações necessárias para a forma de gerência desejada e a aplicação cliente valida os tipos de informações, enviando assim o token CSRF juntamente com essas informações por uma requisição HTTP. O Servidor então processa determinada requisição se comunicando com o Banco de Dados, retornando os dados obtidos ou uma confirmação de operação, dependendo da requisição.

4.3.6.5 Fluxo Alternativo

- **Usuário não está logado**

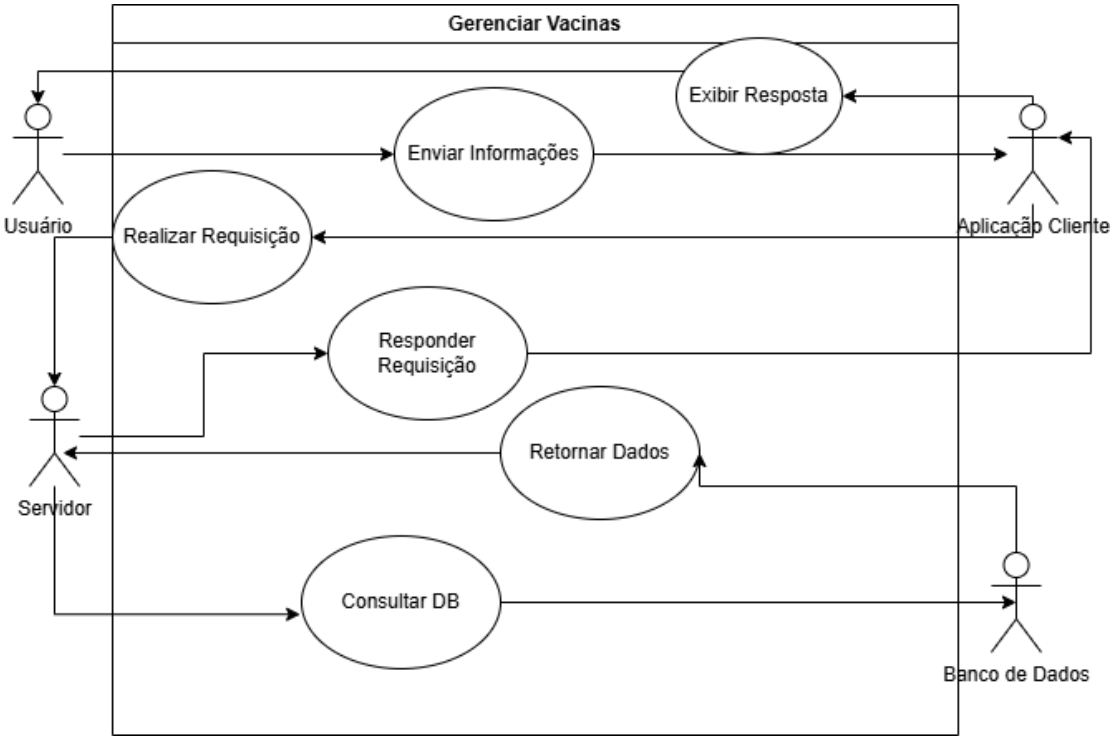
A aplicação cliente não reconhece o login do usuário e solicita que ele realize o login através do UC01 – Autenticar Login.

- **Usuário manda informações erradas:**

A aplicação cliente valida as informações e retorna um erro para o usuário.

4.3.6.6 Representação

Figura 8 – Diagrama UC06 – Gerenciar Vacinas. Fonte: Autoria Própria.



4.3.7 UC07 – Emitir Relatório

4.3.7.1 Cenário

Este caso de uso ocorre após UC01 – Autenticar Login.

4.3.7.2 Descrição

O usuário irá enviar o id do lote que deseja gerar o relatório e receberá um relatório.

4.3.7.3 Ator

Usuário, Aplicação Cliente, Banco de Dados, Servidor.

4.3.7.4 Fluxo Principal

O Usuário insere corretamente as informações necessárias para a forma de gerência desejada e a aplicação cliente valida os tipos de informações, enviando assim o token CSRF juntamente com essas informações por uma requisição HTTP. O Servidor então processa determinada requisição se comunicando com o Banco de

Dados, retornando os dados obtidos ou uma confirmação de operação, dependendo da requisição.

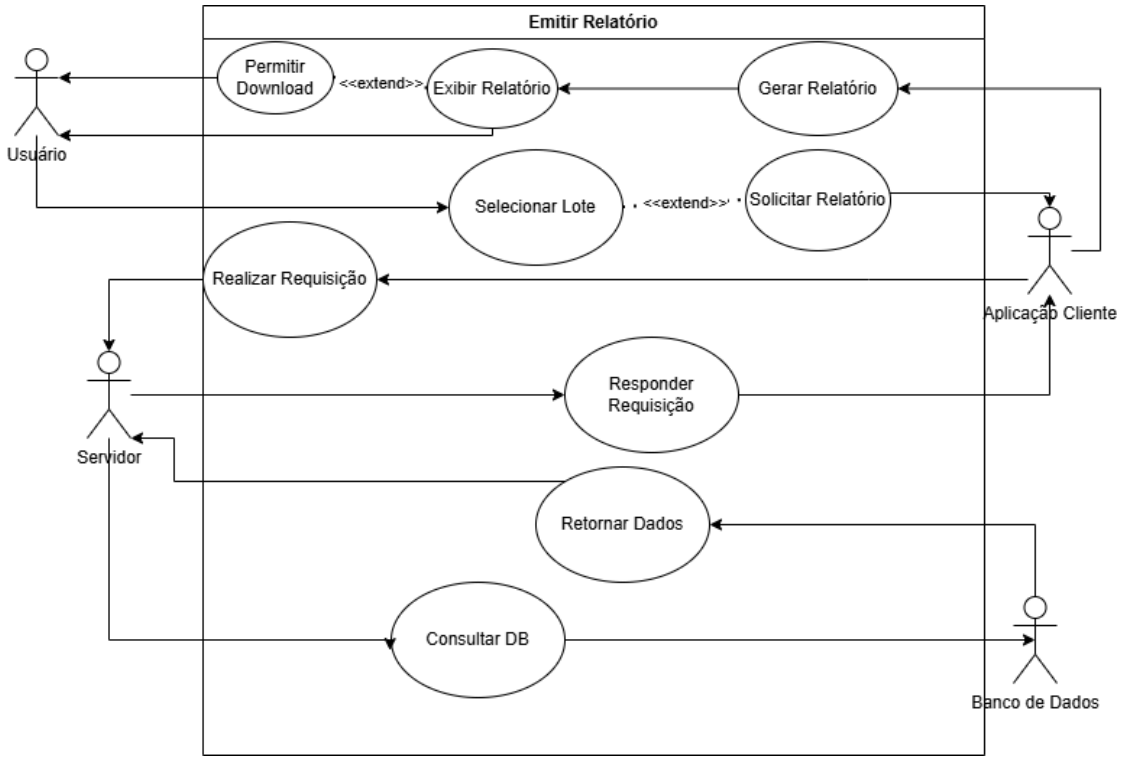
4.3.7.5 Fluxo Alternativo

- **Usuário não está logado:**

A aplicação cliente não reconhece o login do usuário e solicita que ele realize o login através do UC01 – Autenticar Login.

4.3.7.6 Representação

Figura 9 – Diagrama UC07 – Emitir Relatório. Fonte: Autoria Própria.



4.3.8 UC08 – Gerenciar Gastos

4.3.8.1 Cenário

Este caso de uso ocorre após UC01 – Autenticar Login.

4.3.8.2 Descrição

O Usuário irá enviar um conjunto de informações de acordo com a forma de gerência (Cadastro ou Listagem) que deseja. A aplicação cliente então fará uma requisição HTTP para a API correspondente com a ação desejada.

4.3.8.3 Ator

Usuário, Aplicação Cliente, Banco de Dados, Servidor.

4.3.8.4 Fluxo Principal

O Usuário insere corretamente as informações necessárias para a forma de gerência desejada e a aplicação cliente valida os tipos de informações, enviando assim o token CSRF juntamente com essas informações por uma requisição HTTP. O Servidor então processa determinada requisição se comunicando com o Banco de Dados, retornando os dados obtidos ou uma confirmação de operação, dependendo da requisição.

4.3.8.5 Fluxo Alternativo

- **Usuário não está logado:**

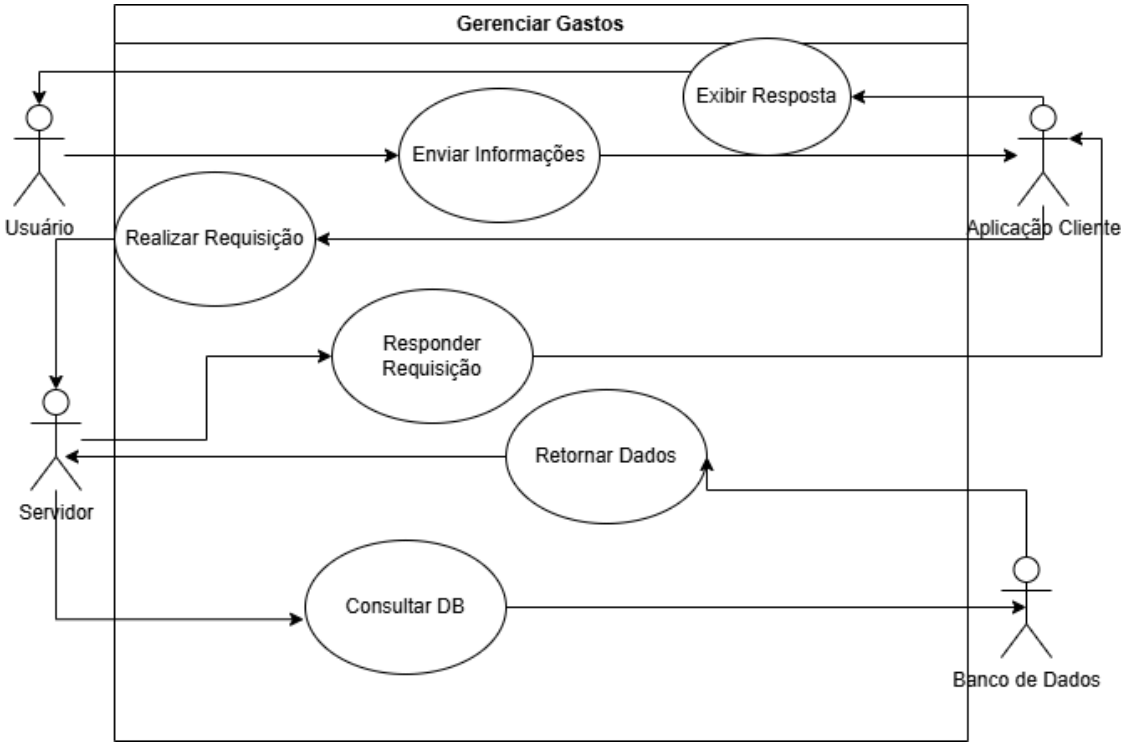
A aplicação cliente não reconhece o login do usuário e solicita que ele realize o login através do UC01 – Autenticar Login.

- **Usuário manda informações erradas:**

A aplicação cliente valida as informações e retorna um erro para o usuário.

4.3.8.6 Representação

Figura 10 – Diagrama UC08 – Gerenciar Gastos. Fonte: Autoria Própria.



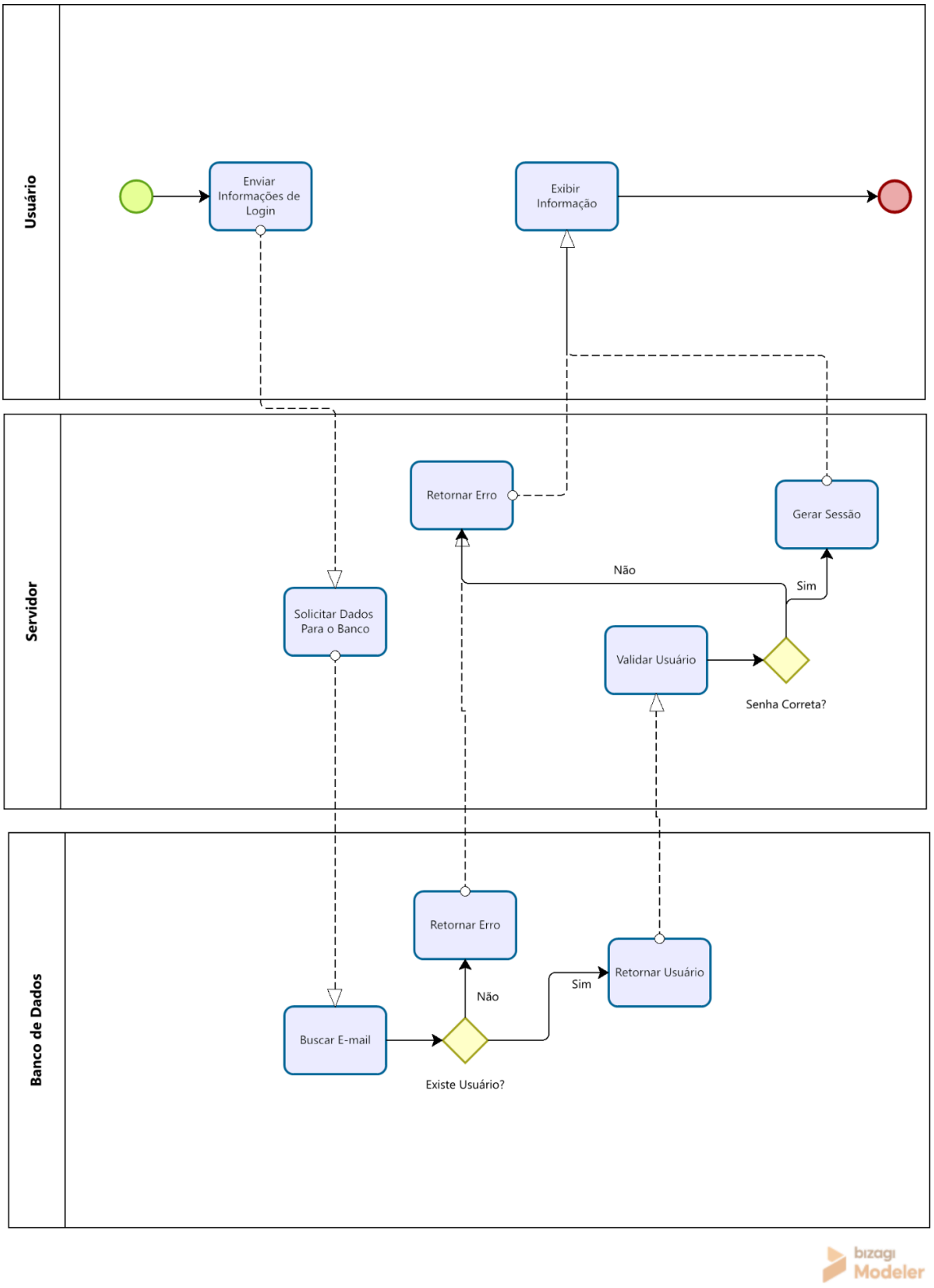
4.4 Interações do Sistema

Com o objetivo de representar de forma clara e estruturada as interações e fluxos de funcionamento do sistema, foram elaborados diagramas BPMN e de sequência. Essas representações visuais descrevem o comportamento dinâmico da aplicação, evidenciando a troca de mensagens entre os componentes e o fluxo dos processos internos. Os diagramas foram desenvolvidos a partir das informações obtidas durante a elicitação e análise dos requisitos junto ao cliente final, garantindo uma visão precisa e integrada do funcionamento do sistema de forma generalizada.

4.4.1 Diagrama BPMN01 – Autenticar Login

O diagrama abaixo apresenta o fluxo completo do processo de autenticação de um usuário em uma aplicação, abrangendo três raias principais: Usuário, Servidor e *DbContext* (responsável pela interação com o banco de dados). Ele ilustra, de forma sequencial e organizada, como as informações de login são tratadas desde o envio pelo usuário até a resposta final da aplicação.

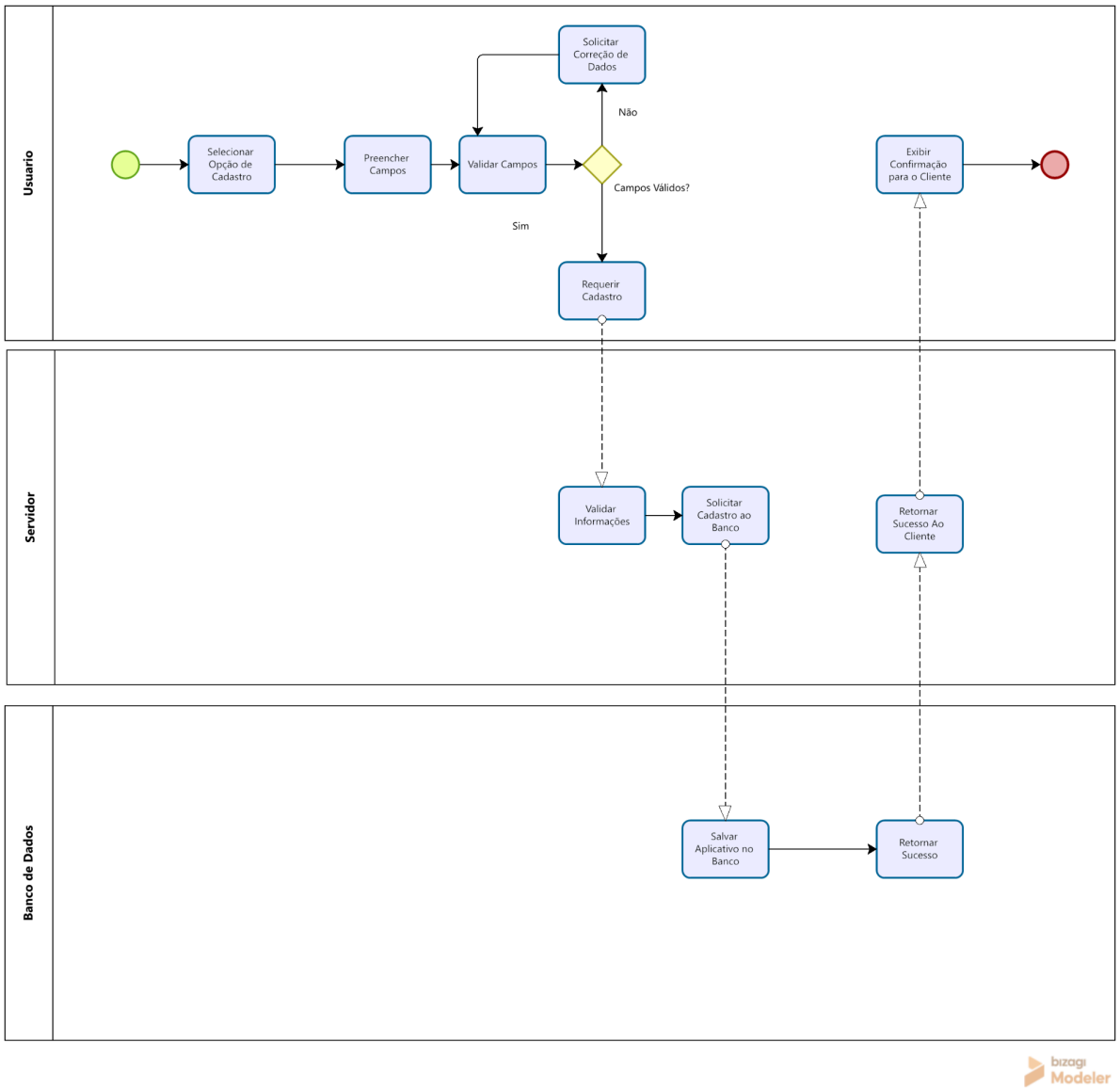
Figura 11 – Diagrama BPMN01 – Autenticar Login. Fonte: (Bizagi, Diagrama BPMN01 – Autenticar Login, 2025)



4.4.2 Diagrama BPMN02 – Gerenciar Modelo

O diagrama abaixo apresenta o fluxo completo do processo de cadastro de um novo tipo de modelo, estruturado em três raias principais: Usuário, Servidor e *DbContext* (camada responsável pela persistência dos dados). O modelo BPMN descreve como os dados são coletados, validados e finalmente armazenados no banco de dados, garantindo um processo seguro e consistente.

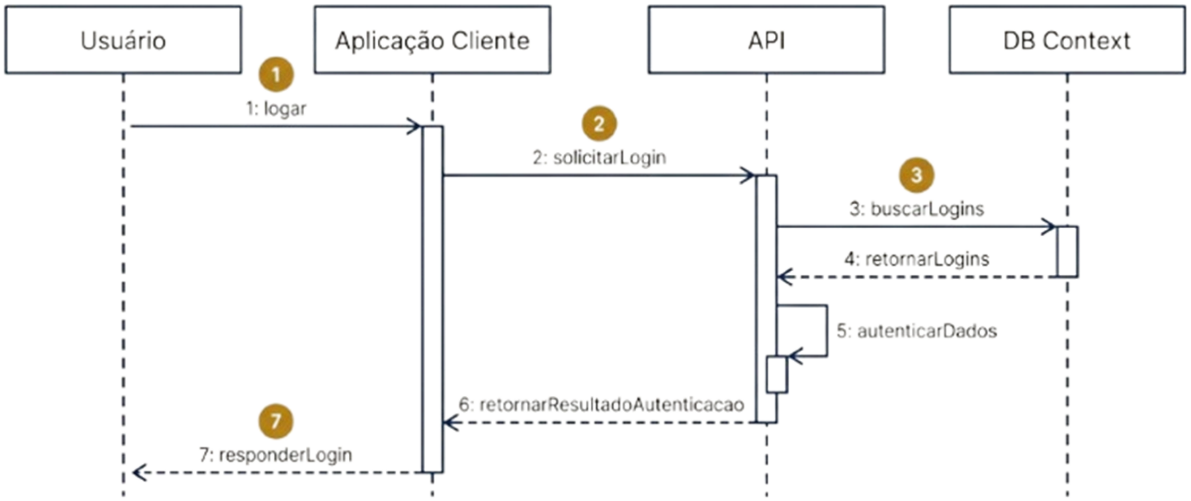
Figura 12 – Diagrama BPMN02 – Gerenciar Modelo. Fonte: (Bizagi, Diagrama BPMN02 – Gerenciar Modelo, 2025)



4.4.3 Diagrama de Sequência – Autenticar Login

O diagrama de sequência abaixo apresenta, de forma detalhada, a interação entre os principais componentes envolvidos no processo de autenticação de um usuário: Usuário, Aplicação Cliente, API e DbContext. Ele descreve a ordem cronológica das mensagens trocadas entre as camadas, permitindo visualizar com precisão como o sistema trata uma requisição de login.

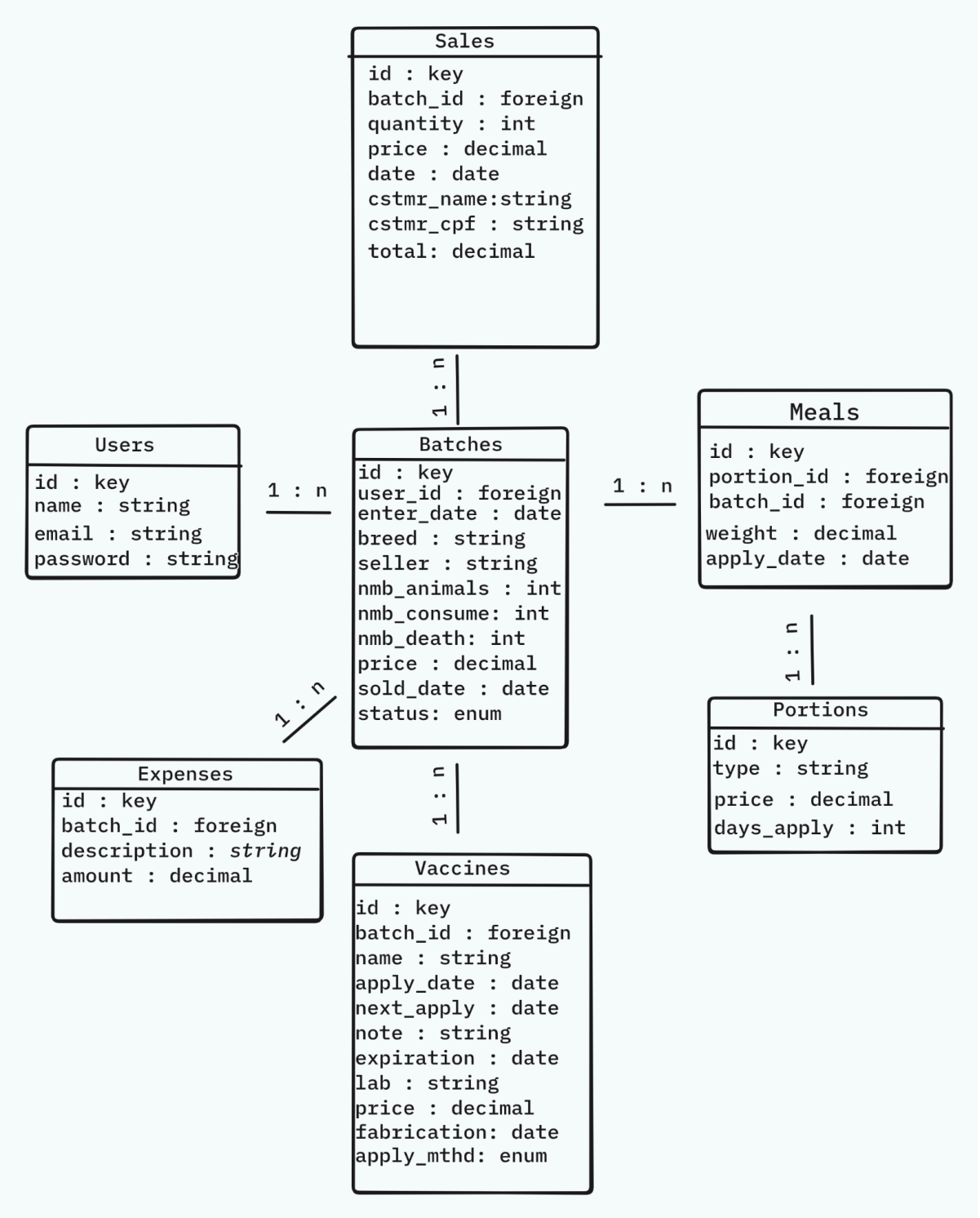
Figura 13 – Diagrama Sequência de Login. Fonte: Autoria Própria.



4.5 MODELAGEM LÓGICA

Note que, as nomenclaturas dos atributos de cada tabela seguem os padrões de nomenclatura do Laravel e siglas como “qnt” representam *quantity*.

Figura 14 – Estrutura Banco de Dados. Fonte: (TLDRAW, Estrutura Banco de Dados, 2025)

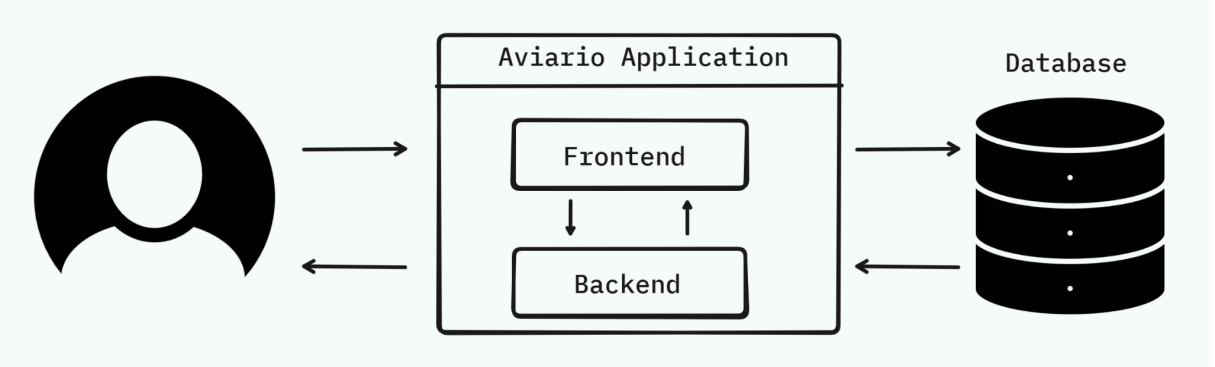


4.6 ARQUITETURA E MODELO DO SOFTWARE

O software seguirá uma arquitetura MVC (*Model View Controller*), isto é, organizado em *Models*, as entidades do software que representam cada tabela do banco de dados, *Views*, telas de visualização do software, o que realmente é mostrado em tela, e *Controllers,* responsáveis por fazer a comunicação entre as *Views* e os *Models.* A arquitetura MVC é um padrão de projeto amplamente adotado no desenvolvimento de aplicações, principalmente na engenharia de software e no desenvolvimento web. Ela separa a aplicação em três componentes principais: o *Model*, responsável pela lógica de dados e regras de negócio; a *View*, encarregada da interface com o usuário e da apresentação dos dados; e o *Controller*, que atua como intermediário entre *Model* e *View*, processando as entradas dos usuários e atualizando os dados e a interface conforme necessário. Essa separação promove maior modularidade, possibilidade de manutenção e reutilização de código (GAMMA et al., 1995; FOWLER, 2003).

Será implementado um software monolítico, isto é, a aplicação *frontend* e *backend* serão implementadas em conjunto no mesmo projeto, funcionando juntas de acordo com o modelo apresentado abaixo:

Figura 15 – Estrutura Aplicação. Fonte: (TLDRAW, Estrutura Aplicação, 2025).



Note que para a implementação deste software, estamos considerando que ele será hospedado em um servidor doméstico, evitando portanto, custos com hospedagens de terceiros como *Amazon AWS* ou *Google Cloud*. O servidor doméstico no qual será hospedado, terá como S.O *ubuntu server* e contará com tecnologias como *Docker*, para o uso de *containers.*

4.7 DEFINIÇÃO DO LAYOUT

Buscando uma padronização do layout da aplicação, foi definido um layout básico que será seguido por todas as telas da aplicação. Visando a reutilização de código, e aplicando práticas para criação de componentes intuitivos e adequados em relação às boas práticas de UI/UX *Design*.

5 RESULTADOS

O desenvolvimento da aplicação monolítica em Laravel para gestão do aviário Campo Alegre resultou em um sistema web capaz de centralizar, automatizar e organizar os principais processos relacionados ao manejo de frangos de corte. A solução implementada contemplou funcionalidades voltadas ao controle de lotes, monitoramento de parâmetros zootécnicos, registro de insumos, geração de relatórios e acompanhamento do desempenho produtivo, proporcionando uma visão integrada e em tempo real da operação.

Entre os resultados mais relevantes, destaca-se a implementação de um módulo completo para gestão de lotes, permitindo o registro da data de alojamento, linhagem, fornecedor, quantidade de aves e mortalidade diária. Esse módulo possibilitou a automatização de cálculos essenciais, como taxa de mortalidade acumulada e idade média do lote, reduzindo erros antes frequentes nos registros manuais e agilizando a tomada de decisão por parte do produtor.

Outro resultado significativo foi o desenvolvimento do módulo de controle de insumos, que viabilizou o acompanhamento das entradas e saídas de ração, medicamentos e materiais diversos. A integração desse módulo com os registros de lotes permitiu rastrear o uso de insumos por período e por ciclo produtivo, facilitando o planejamento de compras e a identificação de desvios ou desperdícios.

A aplicação também incorporou funcionalidades específicas para monitoramento ambiental, permitindo registrar e visualizar dados referentes à temperatura, umidade e ventilação ao longo do ciclo. Essas informações, antes anotadas manualmente, passaram a ser centralizadas no sistema e exibidas em gráficos, possibilitando ao gestor correlacionar variações ambientais com o desempenho zootécnico e agir preventivamente em situações críticas.

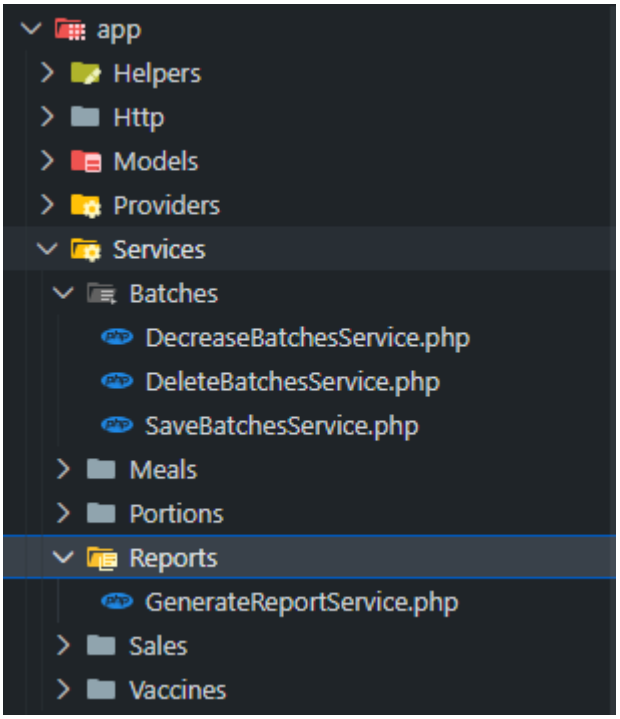
5.1 PADRÕES DE PROJETO E ORGANIZAÇÃO DO CÓDIGO APLICADOS

Durante o desenvolvimento, constatou-se que a utilização de padrões de projeto baseados na camada *Service* contribuiu diretamente para a clareza, coesão e testabilidade do sistema. As regras de negócio mais complexas — como cálculos zootécnicos, validação de parâmetros críticos, lógica de distribuição de insumos e geração de indicadores — foram extraídas dos controladores e encapsuladas em classes de serviço. Essa abordagem resultou em:

- Redução significativa da duplicação de lógica;
- Maior facilidade na manutenção e evolução das funcionalidades;
- Isolamento adequado entre regras de negócio e responsabilidades de controle;
- Aumento da cobertura e simplicidade nos testes unitários.

A utilização da camada de Services permitiu estruturar a aplicação conforme princípios como *Single Responsibility* e *Separation of Concerns*, favorecendo um código mais limpo, sustentável e alinhado às práticas recomendadas para sistemas monolíticos de médio porte. Como apresentado na imagem abaixo, cada serviço possui uma responsabilidade única e é invocado exclusivamente para executar sua função específica. As classes foram organizadas em diretórios conforme sua função primária, de modo que os serviços relacionados aos *Batches* (lotes) estão agrupados na pasta correspondente, e assim sucessivamente.

Figura 16 – Organização e Separação da Camada de Serviços. Fonte: Autoria Própria.



A chamada dos *services* ocorre somente após a conclusão de todas as validações necessárias, tendo como responsabilidade exclusiva executar sua função específica. Conforme ilustrado na imagem, o serviço encarregado de salvar um registro de lote no banco de dados — seja para atualização ou criação — é acionado apenas após essas etapas prévias, garantindo que o fluxo de processamento permaneça organizado e consistente.

Figura 17 – Exemplo de chamada de um Service. Fonte: Autoria Própria.

```php
public function store(Request $request)
{
    $batch = new Batch();

    $validator = $this->validation($request, $batch);

    if ($validator->fails()) {
        return redirect()->back()->withErrors($validator->errors())->withInput();
    }

    $service = new SaveBatchesService($batch);
    return $service->handle($request);
}
```

5.2 COMPONENTIZAÇÃO DO LARAVEL APLICADA

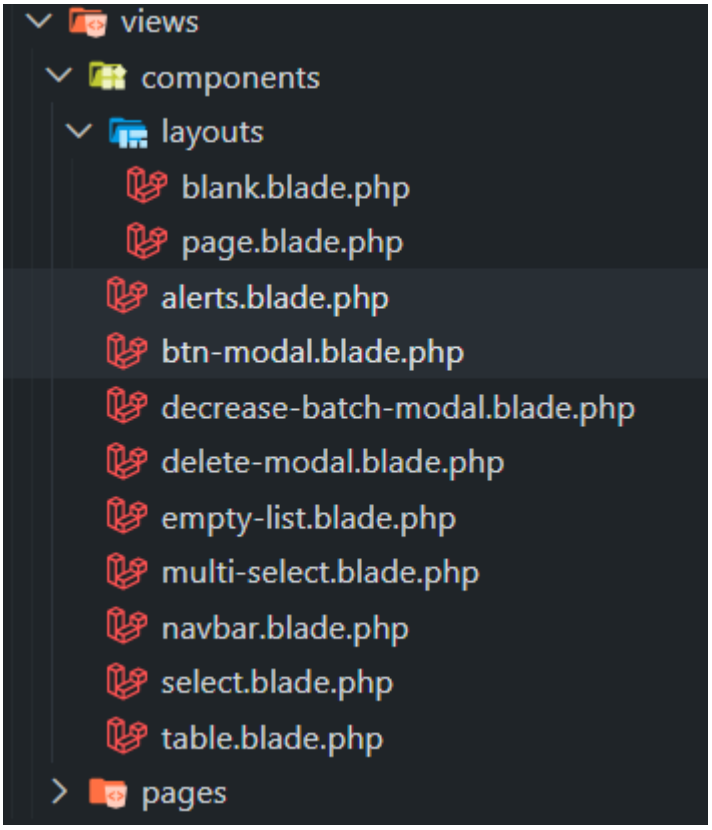
No front-end, a adoção da componentização nativa do Laravel — por meio de *Blade Components* e estruturas reutilizáveis — trouxe ganhos expressivos em modularidade e padronização da interface. Elementos como formulários de cadastro, cartões de informação, tabelas de dados e componentes de entrada de métricas foram abstraídos em componentes independentes, reduzindo a repetição de código e garantindo consistência visual entre as interfaces.

Essa estratégia proporcionou:

- Facilidade na manutenção e evolução da interface, permitindo ajustes centralizados;
- Maior legibilidade dos templates Blade;
- Separação clara entre a camada de apresentação e a lógica do sistema;
- Agilidade na criação de novas telas a partir da reutilização de componentes pré-existentes.

A modularização do front-end também contribuiu para a redução de falhas visuais e tornou o processo de desenvolvimento mais eficiente, especialmente na introdução de novos fluxos ou na expansão de funcionalidades já existentes. Na Figura 18, são apresentados alguns dos componentes utilizados na aplicação. Elementos utilizados com frequência foram transformados em componentes reutilizáveis, como *layouts*, alertas, campos de entrada, seletores múltiplos e *modals*.

Figura 18 – Componentes da Aplicação. Fonte: Autoria Própria.



5.3 APRESENTAÇÃO DA APLICAÇÃO DESENVOLVIDA

Esta seção apresenta a versão final da aplicação resultante do trabalho desenvolvido. São exibidas as principais telas, fluxos e componentes que compõem o sistema, destacando como as funcionalidades foram implementadas e como se integram para atender aos requisitos levantados. As imagens a seguir ilustram a interface, a estrutura visual e o funcionamento geral do software, permitindo uma compreensão concreta do produto final gerado ao longo do projeto.

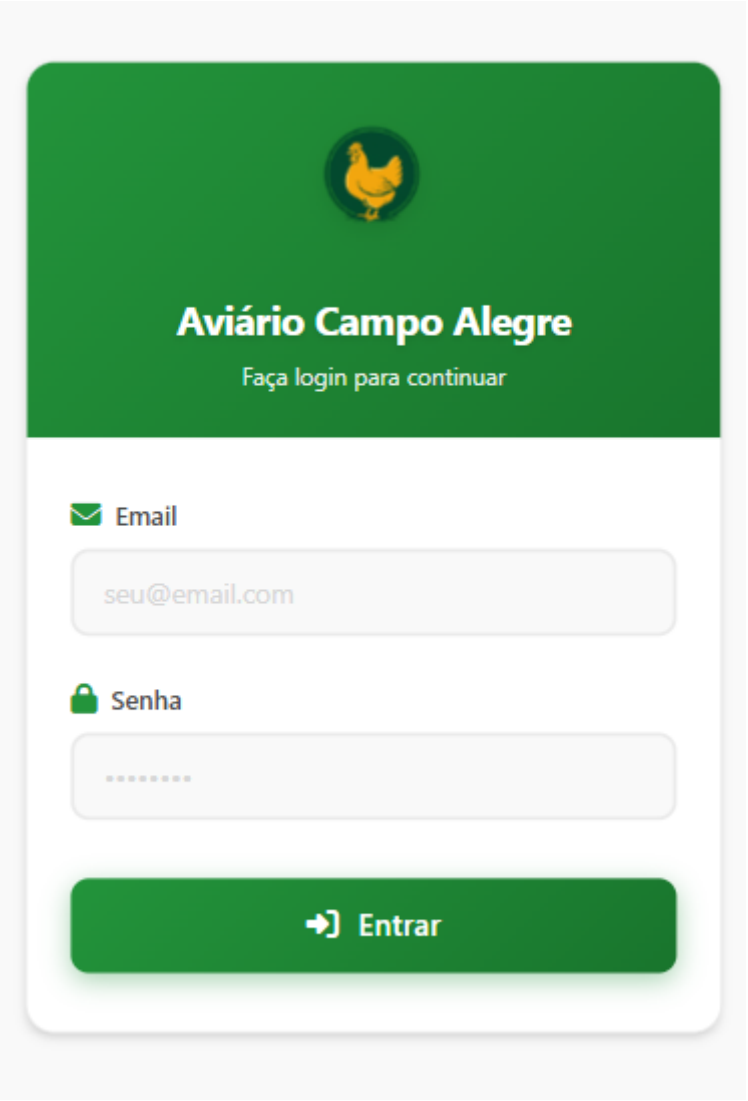
5.3.1 Interface de *Login*

A interface de login é o ponto de entrada da aplicação e tem como objetivo garantir que apenas usuários autorizados tenham acesso ao sistema. Ela apresenta um layout simples e direto, contendo os campos de e-mail e senha, acompanhados de ícones que facilitam sua identificação, além do botão de acesso posicionado em destaque.

Do ponto de vista de segurança, o sistema valida as credenciais informadas comparando-as com os registros previamente armazenados no banco de dados. Apenas usuários cadastrados conseguem prosseguir para as funcionalidades internas; caso contrário, a aplicação retorna uma mensagem de erro. Além disso, a senha é tratada de forma segura no processo de autenticação, reduzindo riscos de acesso indevido.

Essa abordagem garante um fluxo de entrada eficiente, visualmente limpo e alinhado com as boas práticas de segurança e controle de acesso e pode ser visualizada na Figura 19, que corresponde a tela de entrada para o sistema.

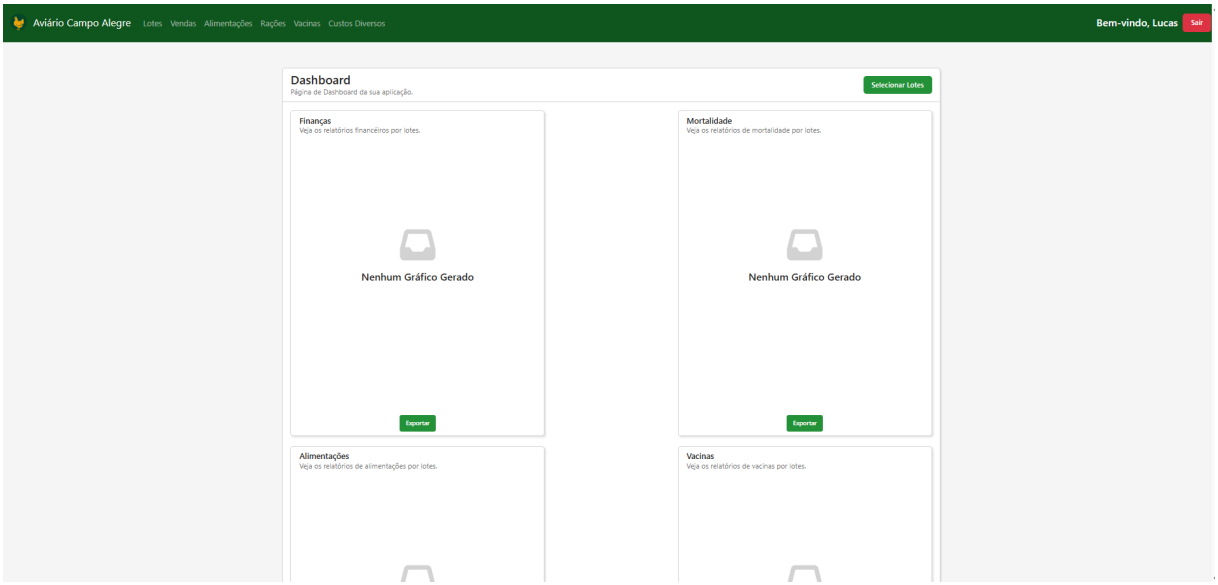
Figura 19 – Interface de Login. Fonte: Autoria Própria.



5.3.2 Dashboard

A Dashboard funciona como a página inicial da aplicação após o usuário autenticado realizar o login. Ela reúne, em um único local, os principais indicadores e relatórios do sistema, permitindo uma visualização geral das informações relacionadas aos lotes cadastrados.

Figura 20 – Dashboard. Fonte: Autoria Própria.

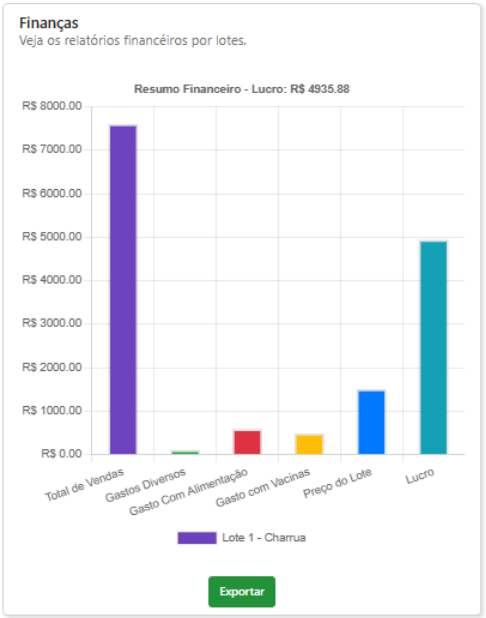


A interface é organizada em cartões independentes, cada um destinado a um tipo de relatório, como finanças, mortalidade, alimentação e vacinas. Quando não há dados selecionados ou registrados, os cartões exibem uma mensagem indicando que nenhum gráfico foi gerado. O botão “Selecionar Lotes”, localizado na parte superior da tela, permite que o usuário escolha quais lotes deseja visualizar. Essa seleção é feita por meio de um seletor exibido em uma modal, conforme ilustrado na imagem abaixo.

Figura 21 – Modal de Seleção de Lotes da Dashboard. Fonte: Autoria Própria.

Após selecionar e confirmar o lote desejado, os cartões anteriormente vazios são preenchidos automaticamente com seus respectivos relatórios, exibindo gráficos e informações baseadas nos dados reais do lote selecionado, como demonstrado nas próximas ilustrações.

Figura 22 – Gráficos de Lucro e Taxa de Mortalidade. Fonte: Autoria Própria.



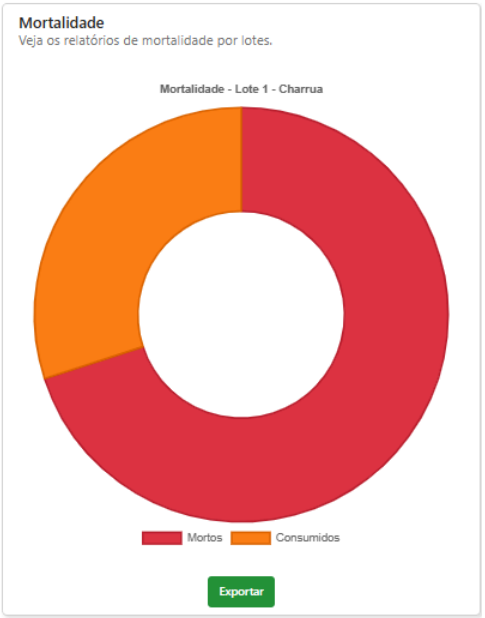
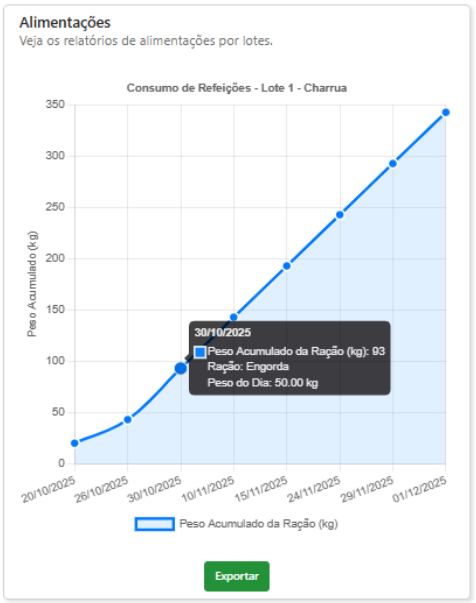
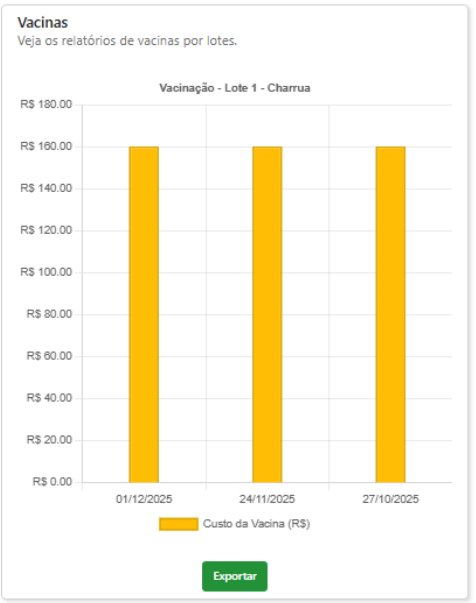


Figura 23 – Gráficos de Alimentação e Vacinas. Fonte: Autoria Própria.





5.3.3 Telas *Index*

As telas Index são responsáveis por exibir, em formato de tabela, todos os registros pertencentes a uma determinada entidade do sistema. Nelas, o usuário tem acesso rápido e organizado às informações principais, além das ações de gerenciamento, como editar, excluir ou criar um novo registro — ações que direcionam para a página de formulário correspondente.

Como mostrado nas imagens abaixo, a tela Index dos Lotes apresenta, além das funcionalidades padrão, opções específicas relacionadas ao controle operacional do aviário. Entre elas, destaca-se a possibilidade de registrar mortalidade, permitindo ao usuário informar ocorrências de consumo ou morte natural. Essa funcionalidade adicional, também representada nas imagens seguintes, garante maior detalhamento no acompanhamento dos lotes e contribui para o controle preciso dos dados de produção.

Essa estrutura torna as telas *Index* um elemento central do fluxo de gestão da aplicação, oferecendo praticidade e permitindo um gerenciamento completo e eficiente das entidades cadastradas.

Figura 24 – Exemplo de Tela Index. Fonte: Autoria Própria.

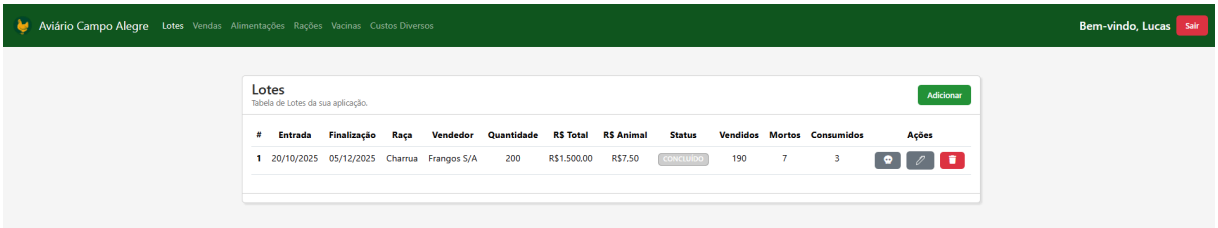
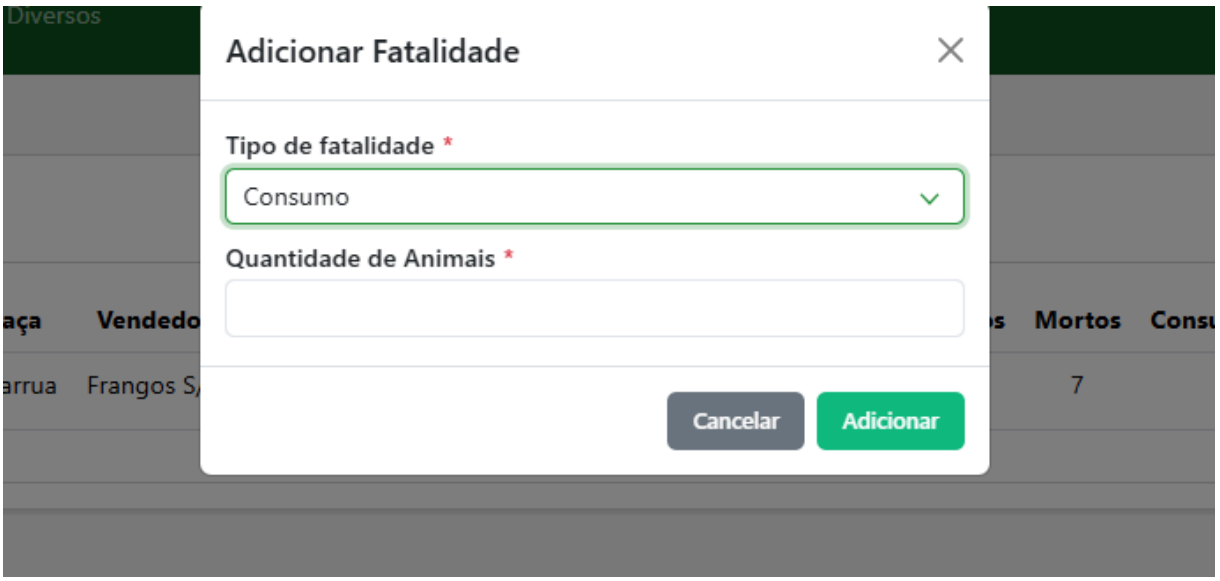


Figura 25 – Modal de Mortalidade dos Lotes. Fonte: Autoria Própria.



5.3.4 Telas de Formulário

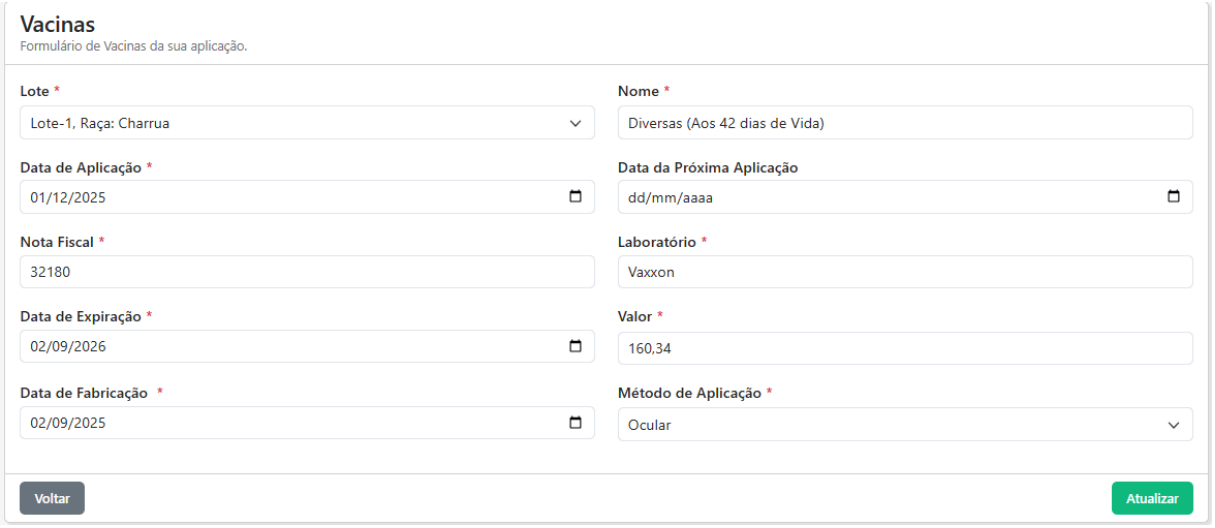
As telas de formulário são utilizadas para criar ou editar registros de cada entidade do sistema. Diferentemente das telas Index, que exibem listagens completas, os formulários apresentam um conjunto organizado de campos que permitem ao usuário inserir ou atualizar informações de maneira estruturada e consistente. Cada formulário foi projetado para ser simples, objetivo e alinhado às regras de validação necessárias para garantir a integridade dos dados.

Como mostrado na figura abaixo, o formulário de vacinas exemplifica esse padrão. Nele, o usuário pode selecionar o lote ao qual a vacinação pertence, informar detalhes como data, nome da vacina, quantidade aplicada e demais observações relevantes. Todos os campos são validados antes do envio, garantindo

que apenas informações completas e corretamente formatadas sejam registradas no banco de dados.

Essa abordagem padronizada nos formulários contribui para uma experiência mais clara e reduz a chance de erros, além de assegurar que todas as operações de criação e edição sejam realizadas de forma segura e consistente em toda a aplicação.

Figura 26 – Exemplo Tela de Formulário. Fonte: Autoria Própria.



5.4 IMPLANTAÇÃO DO SOFTWARE

A implantação da aplicação foi realizada em um Ubuntu Server, hospedado em um servidor doméstico acessível dentro da mesma rede local. Esse ambiente foi escolhido por oferecer baixo custo, controle total sobre a infraestrutura e facilidade de manutenção. Caso haja necessidade de acesso externo, o servidor pode ser alcançado de forma segura por meio de uma conexão Virtual Private Network (VPN) garantindo proteção dos dados e evitando exposição direta à internet.

Para padronizar o ambiente de execução e facilitar a administração dos serviços, a aplicação foi implantada utilizando Docker, permitindo que todos os componentes — aplicação, servidor web e banco de dados — fossem executados em contêineres independentes. Essa abordagem garante isolamento, evita conflitos de dependências e possibilita a replicação do ambiente com facilidade.

O Nginx foi utilizado dentro de um contêiner para atuar como servidor web, responsável por entregar os arquivos da aplicação e gerenciar as requisições de forma eficiente. Sua alta performance e baixo consumo de recursos o tornam uma opção ideal para ambientes de produção leves.

O banco de dados utilizado é o PostgreSQL, também executado em um container Docker. Essa escolha permite um gerenciamento mais organizado do armazenamento e facilita a realização de backups, atualizações de versão e migração de dados, mantendo a integridade das informações. A comunicação entre os contêineres ocorre por meio de uma rede interna do Docker, garantindo segurança e reduzindo a superfície de exposição.

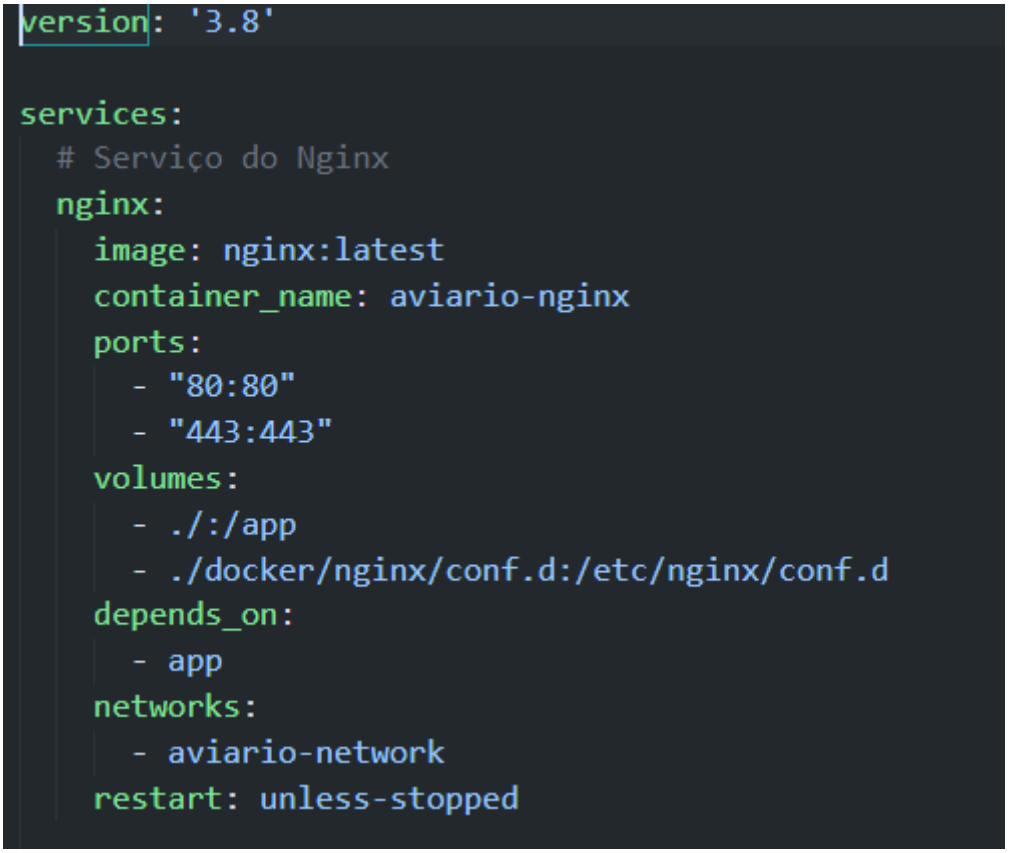
Essa estratégia de implantação com Docker, Nginx e PostgreSQL proporciona um ambiente estável, seguro e facilmente escalável, permitindo que o sistema funcione de forma consistente tanto em um servidor doméstico quanto em cenários mais robustos caso seja necessário evoluir a infraestrutura futuramente. Abaixo seguem imagens dos arquivos dockerfile e docker-compose.

Figura 27 – Dockerfile. Fonte: Autoria Própria.

Dockerfile

```
# Instalar dependências do sistema
RUN apt-get update && apt-get install -y \
    git \
    curl \
    zip \
    unzip \
    supervisor \
    nodejs \
    npm \
    libpq-dev \
    && rm -rf /var/lib/apt/lists/*

# Instalar extensões PHP necessárias
RUN docker-php-ext-install pdo pdo_pgsql

# Instalar Composer
COPY --from=composer:latest /usr/bin/composer /usr/bin/composer

# Definir diretório de trabalho
WORKDIR /app

# Copiar arquivos da aplicação
COPY . .

# Instalar dependências do Composer
RUN composer install --no-dev --optimize-autoloader

# Instalar dependências npm
RUN npm install && npm run build

# Definir permissões
RUN chown -R www-data:www-data /app/storage /app/bootstrap/cache

# Expor porta (PHP-FPM usa porta 9000)
EXPOSE 9000

# Comando padrão
CMD ["php-fpm"]
```

Figura 28 – Configuração do Nginx - Docker Compose. Fonte: Autoria Própria.



```yaml
version: '3.8'

services:
  # Serviço do Nginx
  nginx:
    image: nginx:latest
    container_name: aviario-nginx
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./:/app
      - ./docker/nginx/conf.d:/etc/nginx/conf.d
    depends_on:
      - app
    networks:
      - aviario-network
    restart: unless-stopped
```

Figura 29 – Configuração Aplicação - Docker Compose. Fonte: Autoria Própria.

```
# Serviço da aplicação PHP-FPM
app:
  build:
    context: .
    dockerfile: Dockerfile
  container_name: aviario-app
  working_dir: /app
  volumes:
    - ./:/app
    - ./docker/php/conf.d/php.ini:/usr/local/etc/php/conf.d/php.ini
  environment:
    - DB_CONNECTION=pgsql
    - DB_HOST=db
    - DB_PORT=5432
    - DB_DATABASE=aviario
    - DB_USERNAME=aviario_user
    - DB_PASSWORD=aviario_password
    - APP_ENV=production
    - APP_DEBUG=false
    - APP_KEY=
  depends_on:
    - db
  networks:
    - aviario-network
  restart: unless-stopped
```

Figura 30 – Configuração PostgreSQL, Rede, Volume - Docker Compose.Fonte: Autoria Própria.

```yaml
  # Serviço do banco de dados PostgreSQL
  db:
    image: postgres:15-alpine
    container_name: aviario-db
    environment:
      POSTGRES_DB: aviario
      POSTGRES_USER: aviario_user
      POSTGRES_PASSWORD: aviario_password
    volumes:
      - db_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"
    networks:
      - aviario-network
    restart: unless-stopped
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U aviario_user"]
      interval: 10s
      timeout: 5s
      retries: 5

volumes:
  db_data:

networks:
  aviario-network:
    driver: bridge
```

5.5 CONSIDERAÇÕES SOBRE A IMPLEMENTAÇÃO

Ao final da implementação, verificou-se que o modelo monolítico adotado mostrou-se adequado ao contexto da granja, dado o escopo centralizado e a necessidade de baixo custo operacional. O sistema atendeu aos requisitos funcionais definidos e demonstrou ser uma ferramenta eficaz para apoiar a gestão do aviário, promovendo maior controle sobre o ciclo produtivo, melhor organização das rotinas e maior confiabilidade das informações. O uso de boas práticas arquiteturais, como *Services* e componentização, contribuiu diretamente para a robustez, escalabilidade interna e longevidade da solução.

Além disso, a implantação em um servidor doméstico baseado em Ubuntu *Server* demonstrou-se uma estratégia viável e econômica. A utilização de contêineres *Docker*, juntamente com Nginx e PostgreSQL, permitiu criar um ambiente de produção estável, padronizado e facilmente replicável, sem a necessidade de infraestrutura corporativa dedicada. O acesso local simplificou o uso diário, enquanto a possibilidade de conexão remota via VPN garantiu segurança e flexibilidade quando o acesso externo se fez necessário. Essa abordagem consolidou a solução como prática, sustentável e alinhada às necessidades reais da granja.

6 CONCLUSÃO

Uma preocupação constante durante o desenvolvimento deste trabalho foi a de conseguir uma aplicação escalável e intuitiva para o usuário. Visto que será utilizado diariamente e pode acumular uma grande quantidade de dados. Por este motivo, boas práticas no desenvolvimento das consultas de banco de dados foram aplicadas, visando torná-las o mais otimizadas possível e evitar problemas clássicos do Laravel como o *lazy loading.* Evitando assim consultas repetitivas e maçantes na aplicação.

Ainda existem muitos fatores que podem somar à construção desta aplicação. A criação de uma página para cadastro de diferentes usuários, assim como a criação de assinaturas de planos para o uso da aplicação. Toda a base de dados e consultas são extremamente adaptáveis para este cenário.

Também para trabalhos futuros, o desenvolvimento de novas opções de relatórios e a possibilidade de exportação em formato pdf seriam um grande adicional à plataforma do sistema. A hospedagem do software em uma plataforma online como a AWS seria essencial para melhorar a disponibilidade da aplicação ao usuário.

Com isso, entregando um software escalável e funcional para a gestão de um aviário de pequeno porte, esse trabalho mostra que é possível desenvolver uma aplicação web que facilite e inove o cotidiano de um pequeno empresário rural, aprimorando a gestão de seu aviário de forma acessível e eficiente.

7 REFERÊNCIAS

ASGHAR, Muhammad Zeeshan; ALAM, Khubaib Amjad; JAVED, Shahzeb. **Software Design Patterns Recommendation**: A Systematic Literature Review. In: INTERNATIONAL CONFERENCE ON FRONTIERS OF INFORMATION TECHNOLOGY, 2019. Anais… p. 167–172. DOI: 10.1109/FIT47737.2019.00040. Disponível em: https://ieeexplore.ieee.org/document/8972440. Acesso em: 9 nov. 2025.

CARVALHO, I. M. de S.; CAMPOS, M. A. S. **As metodologias por projeto nas salas de aulas do CEFET-MG, desde o ponto de vista dos docentes do ensino médio**. Revista Ibero-Americana de Humanidades, Ciências e Educação, [S. l.], v. 7, n. 11, p. 46–67, 2021. DOI: 10.51891/rease.v7i11.2993. Disponível em: https://www.periodicorease.pro.br/rease/article/view/2993. Acesso em: 5 mai. 2025.

DRAGOI, Ioana et al. **A Comparative Review of Microservices and Monolithic Architectures**. arXiv preprint, arXiv:1909.09436, 2019. Disponível em: https://arxiv.org/abs/1909.09436. Acesso em: 9 nov. 2025.

EVANS, Eric. **Domain-Driven Design**: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley, 2004.

FOWLER, Martin. **Patterns of Enterprise Application Architecture**. Boston: Addison-Wesley, 2003.

GAMMA, Erich et al. **Design Patterns**: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley, 1995.

GIL, Antônio Carlos. **Como elaborar projetos de pesquisa**. 6. ed. São Paulo: Atlas, 2017.

GRAÇA DA COSTA, Mário; SANTOS E CAMPOS, Maria Aparecida. **Os reflexos das novas tecnologias de informação e comunicação na gestão escolar democrática, participativa e inclusiva e o seu contributo na melhoria de um ensino de qualidade**. RECIMA21 - Revista Científica Multidisciplinar, [S. l.], v. 4, n. 6, p. e463371, 2023. DOI: 10.47820/recima21.v4i6.3371. Disponível em: https://recima21.com.br/index.php/recima21/article/view/3371. Acesso em: 5 jun. 2025.

LUCAS, E. G.; SOUZA, L. S. de; CRUZ, K. R. da. **Educação de Jovens e Adultos**: o uso das tecnologias da informação e comunicação. Rebena - Revista Brasileira de Ensino e Aprendizagem, [S. l.], v. 5, p. 196–206, 2023. Disponível em: https://rebena.emnuvens.com.br/revista/article/view/83. Acesso em: 15 abr. 2025.

MAJEED, Abdul; RAUF, Ibtisam. **MVC Architecture**: A Detailed Insight to the Modern Web Applications Development. Peer Rev J Solar & Photoenergy Syst., v. 1, n. 1, p. 1–7, 2018. Disponível em: https://crimsonpublishers.com/prsp/pdf/PRSP.000505.pdf. Acesso em: 9 nov. 2025.

MARTIN, Robert C. **Clean Architecture**: A Craftsman's Guide to Software Structure and Design. Boston: Prentice Hall, 2018.

PRESSMAN, Roger S.; MAXIM, Bruce R. **Software Engineering**: A Practitioner's Approach. 8. ed. New York: McGraw-Hill, 2016.

SILVA, C. C. S. C. da; TEIXEIRA, C. M. de S. **O uso das tecnologias na educação: os desafios frente à pandemia da COVID-19** / The use of technologies in education: the challenges facing the COVID-19 pandemic. Brazilian Journal of Development, [S. l.], v. 6, n. 9, p. 70070–70079, 2020. DOI: 10.34117/bjdv6n9-452. Disponível em: https://ojs.brazilianjournals.com.br/ojs/index.php/BRJD/article/view/16897. Acesso em: 5 jun. 2025.

TEIXEIRA, Cenidalva; CARVALHO, S. M. **A gamificação como prática de ensino na disciplina Automação de Unidades de Informação**. Revista Querubim (Online), v. 16, p. 20–25, 2020.

TLDRAW. **Estrutura Banco de Dados**. 2025. Disponível em: https://www.tldraw.com/f/JMjvtxqNGl3HJeZ2XcgHw?d=v-791.-188.2935.1329.LpYo-basw0V7Ob_G7iQbD. Acesso em: 2 jun. 2025.

TLDRAW. **Estrutura da Aplicação**. 2025. Disponível em: https://www.tldraw.com/f/JMjvtxqNGl3HJeZ2XcgHw?d=v-791.-188.2935.1329.LpYo-basw0V7Ob_G7iQbD. Acesso em: 2 jun. 2025.

VERASZTO, E. V. **Projeto Teckids**: educação tecnológica no ensino fundamental. Disponível em: http://repositorio.unicamp.br/Acervo/Detalhe/315223. Acesso em: 5 jun. 2025.

WAZLAWICK, Raul Sidnei. **Metodologia de Pesquisa para Ciência da Computação**. 2. ed. Rio de Janeiro: Elsevier, 2014.