IFTO – INSTITUTO FEDERAL DO TOCANTINS

CAMPUS PARAÍSO DO TOCANTINS

Sistema de Informação

PABLO MOREIRA SANTOS

CRUD COM LARAVEL

PARAÍSO DO TOCANTINS – TO

2025

Sumário

RELATÓRIO TÉCNICO: SISTEMA DE CONTROLE DE ESTOQUE

1. Introdução e Arquitetura

Este documento detalha o processo de desenvolvimento de uma aplicação de controle de estoque. O sistema foi construído baseando-se na arquitetura MVC (Model-View-Controller), padrão nativo do *framework Laravel*. Esta escolha arquitetural visou garantir a separação de responsabilidades entre a lógica de negócios, a interface do usuário e a manipulação de dados, facilitando a manutenção e a escalabilidade do código.

2. Especificação de Requisitos do Sistema

Para garantir que a solução atendesse às necessidades de gestão e usabilidade, o projeto foi guiado pelos seguintes requisitos funcionais e não funcionais:

2.1. Requisitos Funcionais (RF)

- **RF001 - Autenticação de Usuário (Login):** O sistema restringe o acesso à área administrativa, exigindo que o usuário informe credenciais (e-mail e senha) previamente cadastradas.
- **RF002 - Cadastro de Produtos:** Permite o registro de novos produtos no estoque informando: Nome, Quantidade e Preço Unitário.
- **RF003 - Listagem Geral:** Exibe uma tabela contendo todos os produtos cadastrados, mostrando suas respectivas quantidades e preços.
- **RF004 - Edição de Registros:** Permite a alteração dos dados (Nome, Quantidade e Preço) de qualquer produto já cadastrado.
- **RF005 - Exclusão de Produtos:** Permite a remoção definitiva de um produto do banco de dados.
- **RF006 - Confirmação de Exclusão:** Exige uma confirmação explícita (via popup) antes de efetivar a exclusão de um registro, garantindo a segurança dos dados.
- **RF007 - Pesquisa e Filtragem:** O sistema possui uma barra de busca para filtrar produtos pelo nome em tempo real.
- **RF008 - Dashboard de Indicadores:** A tela inicial exibe o total de itens físicos em estoque e o valor patrimonial total acumulado.
- **RF009 - Tratamento Monetário:** O sistema interpreta valores no padrão brasileiro (vírgula) e os converte para o padrão do banco de dados (ponto).
- **RF010 - Formatação de Visualização:** Apresenta valores monetários formatados em Reais (R$).
- **RF011 - Validação de Dados:** Impede o cadastro de formulários com campos obrigatórios vazios.
- **RF012 - Logout Seguro:** Funcionalidade para encerrar a sessão do usuário.
- **RF013 - Navegação Cancelável:** Botões de "Cancelar" nos formulários para retornar sem salvar.

2.2. Requisitos Não Funcionais (RNF)

- **RNF001 - Interface Responsiva e Modo Escuro:** Interface construída com Bootstrap 5, utilizando nativamente o tema escuro (*Dark Mode*).
- **RNF002 - Arquitetura MVC:** Estrutura de código seguindo o padrão Model-View-Controller.
- **RNF003 - Persistência de Dados:** Utilização de banco de dados SQLite gerenciado via Migrations.
- **RNF004 - Segurança CSRF:** Proteção contra ataques *Cross-Site Request Forgery* em todos os formulários.

3. Processo de Desenvolvimento (CRUD)

A implementação das operações fundamentais (Create, Read, Update, Delete) seguiu um fluxo lógico dividido em três camadas:

3.1. Modelagem de Dados e Persistência

Iniciamos o desenvolvimento pelas *Migrations* para definir a estrutura do banco de dados relacional.

- Definições de Tabela: Na tabela *products*, optamos por uma tipagem forte — utilizando *integer* para quantidades e decimal(8,2) para valores monetários. Essa decisão foi crucial para garantir a integridade dos cálculos financeiros e de inventário.
- Model: No Model *Product*, configuramos a propriedade *$fillable*. Isso habilitou a atribuição em massa (*mass assignment*), agilizando o processo de persistência dos dados e reduzindo a verbosidade do código.

3.2. Controladores e Regra de Negócios

No *ProductController*, concentramos a orquestração do fluxo de dados:

- Captura: Utilizamos injeção de dependência e o objeto *Request* para capturar as entradas provenientes dos formulários.
- Tratamento: O foco foi implementar a lógica de sanitização antes do salvamento (nas operações de *Create* e *Update*).
- Leitura: Para a recuperação de dados, utilizamos o método *all()* do *Eloquent ORM*, otimizando a consulta ao banco.

3.3. Interface e Rotas

As rotas foram definidas seguindo estritamente o padrão *RESTful*. Na camada de visão (*View*), utilizamos o motor de *templates Blade*, que se mostrou fundamental por dois motivos:

1. Facilidade na criação de estruturas de repetição (@foreach);
2. Solução para a limitação dos formulários HTML nativos, permitindo a implementação dos verbos HTTP PUT e DELETE através da diretiva *@method*.

4. Desafios Técnicos e Soluções Implementadas

Durante o ciclo de desenvolvimento, a equipe enfrentou dois obstáculos técnicos principais que exigiram refatoração e pesquisa.

4.1. Conflito de Sintaxe no Motor de Renderização

Houve uma dificuldade inicial na distinção entre a sintaxe nativa do PHP (<?php ... ?>) e a sintaxe do *Blade* ({{ ... }}). A mistura de ambas gerou erros de renderização na tabela de produtos.

- Solução: Padronizamos todo o código do *front-end* para utilizar exclusivamente a sintaxe do *Blade*. Além de corrigir o erro visual, essa medida aumentou a segurança da aplicação, visto que o *Blade* realiza o escape automático de variáveis, prevenindo ataques do tipo XSS (*Cross-Site Scripting*).

4.2. Internacionalização e Formatação Numérica

Identificamos erros de tipagem (*TypeError*) ao persistir valores monetários. O conflito ocorreu devido à divergência entre o padrão do banco de dados/PHP (ponto decimal) e o padrão de entrada brasileiro (vírgula).

- Solução: Implementamos uma camada de higienização de dados no *Controller*. Antes da persistência, utilizamos *str_replace* para converter vírgulas em pontos. Inversamente, para a exibição na *View*, aplicamos a função *number_format*, garantindo que o dado seja salvo corretamente no *backend*, mas apresentado de forma amigável (pt-BR) no *frontend*.

5. Ferramentas e Tecnologias

Para otimizar o ciclo de desenvolvimento e garantir a qualidade do código, utilizamos o ecossistema *Laravel*:

- *Laravel Artisan* CLI: A interface de linha de comando foi essencial para o *scaffolding*. O uso do comando *php artisan make:model Product -mcr* permitiu a criação simultânea de *Migration, Controller e Model*, reduzindo drasticamente o tempo de escrita de código repetitivo (*boilerplate*).
- *Eloquent ORM*: A ferramenta de mapeamento objeto-relacional facilitou a interação com o banco de dados através da orientação a objetos, abstraindo a complexidade de consultas SQL manuais e prevenindo injeções de SQL.
- SQLite: A escolha por este SGBD *serverless* simplificou a configuração do ambiente, eliminando a necessidade de serviços de banco de dados pesados e permitindo uma prototipagem ágil.