

CodeIgniter na Prática: Uso de um CRUD de Plantas Ornamentais

Alcides Antonio Lorenski Neto¹

Luan Felipe Tenroller²

Luiz Gustavo da Silva Przygoda³

Nathaly Camargo do Nascimento⁴

Roberson Junior Alves⁵

Resumo

O artigo descreve o desenvolvimento de um sistema web para catálogo de plantas ornamentais, com objetivo de oferecer uma ferramenta intuitiva e funcional para cadastro, consulta e gerenciamento de informações sobre espécies vegetais. O objetivo principal é oferecer uma solução digital funcional e visualmente atrativa que facilite o acesso a dados relevantes sobre plantas ornamentais. Utilizando PHP, CodeIgniter, XAMPP e DBeaver, o sistema segue o padrão MVC e implementa operações CRUD, além de recursos como filtros por tipo de planta, cadastro e login de usuário e favoritos. A interface visual prioriza usabilidade, simplicidade e estética, voltada a entusiastas de jardinagem e tecnologia. A metodologia baseia-se na reutilização de componentes por meio de frameworks, garantindo organização e agilidade no processo de desenvolvimento.

Palavras-chave: PHP. CRUD. Catálogo de Plantas. CodeIgniter. Padrão MVC.

¹ Graduando do Curso de Ciência da Computação - Unoesc Campus São Miguel do Oeste.
alcidesantonio08@hotmail.com

² Graduando do Curso de Ciência da Computação - Unoesc Campus São Miguel do Oeste.
luanf.tenroller@gmail.com

³ Graduando do Curso de Ciência da Computação - Unoesc Campus São Miguel do Oeste.
luimprzygoda24@gmail.com

⁴ Graduanda do Curso de Ciência da Computação - Unoesc Campus São Miguel do Oeste.
nathaly20camargo@gmail.com

⁵ Docente do curso de Ciência da Computação - Unoesc Campus São Miguel do Oeste.
roberson.alves@unoesc.edu.br

1. INTRODUÇÃO

Integrado por uma combinação de funcionalidades interativas e design intuitivo, o sistema de Catálogo de Plantas Ornamentais é desenvolvido com base em tecnologias web como PHP e o framework CodeIgniter, oferecendo uma experiência fluida e organizada para os usuários. O sistema é projetado para permitir o cadastro, visualização e gerenciamento de informações detalhadas sobre plantas ornamentais e seus mais variados tipos, promovendo o acesso rápido e eficiente a dados relevantes sobre cada espécie.

A estrutura do sistema apresenta e dispõe de operações de CRUD, sendo essa palavra uma sigla das ações de: inserção, obtenção, atualização e exclusão para a manipulação de dados. Dessa maneira, possibilita ao usuário adicionar plantas com informações como nome popular, nome científico, cuidados necessários e imagens ilustrativas (Bauer, 2018, p. 20).

O catálogo facilita a busca do usuário ao oferecer filtros por tipo de planta, além de permitir a criação de listas personalizadas de plantas favoritas para consultas rápidas. A interface visual, inspirada em aplicativos modernos de jardinagem, destaca-se pela estética e exibição de imagens, priorizando uma navegação simples e agradável para amantes de plantas, entusiastas da jardinagem e iniciantes na programação.

Nos tópicos a seguir, exploramos os principais aspectos desse catálogo, abordando sua arquitetura em PHP/CodeIgniter com o adjunto do XAMPP, implantação ORM com o uso de *Models*, funcionalidades específicas, organização visual e a forma como promove uma interação eficiente entre usuário e conteúdo.

2. DESENVOLVIMENTO

O PHP é uma linguagem de programação voltada para o desenvolvimento web, amplamente utilizada por sua flexibilidade e integração facilitada com bancos de dados como o MySQL. Para criar um ambiente de desenvolvimento local, foi utilizado o XAMPP, um pacote gratuito que reúne servidores MySQL e PHP, simplificando a instalação desses serviços. Para a modelagem do banco de dados, foi utilizado o SGBD DBeaver. Dentro desse ecossistema, foi utilizado o CodeIgniter, um framework leve que adota o padrão MVC (*Model-View-Controller*), promovendo organização e agilidade no desenvolvimento (Rehman, 2025).

Frameworks são técnicas de reutilização no desenvolvimento de sistemas que oferecem componentes reutilizáveis e facilmente integráveis a novos projetos. Eles servem

como base para o desenvolvimento de aplicações maiores e mais específicas, pois reúnem classes, funções, técnicas e metodologias que facilitam e agilizam esse processo (Pandolfi, 2013).

Com essas métricas em mãos, foi possível realizar a construção do projeto de maneira eficaz e produtiva.

2.1 MODELAGEM DE SOFTWARE DO PROJETO

A modelagem de software é o processo de representar como um sistema vai funcionar, usando diagramas e descrições para planejar, entender e documentar sua estrutura e comportamento. Ela ajuda a organizar o desenvolvimento, facilitar a comunicação entre a equipe e evitar erros futuros. Utilizamos a linguagem padrão UML (*Unified Modeling Language*) para representar visualmente os componentes, estruturas e comportamentos de um sistema de software.

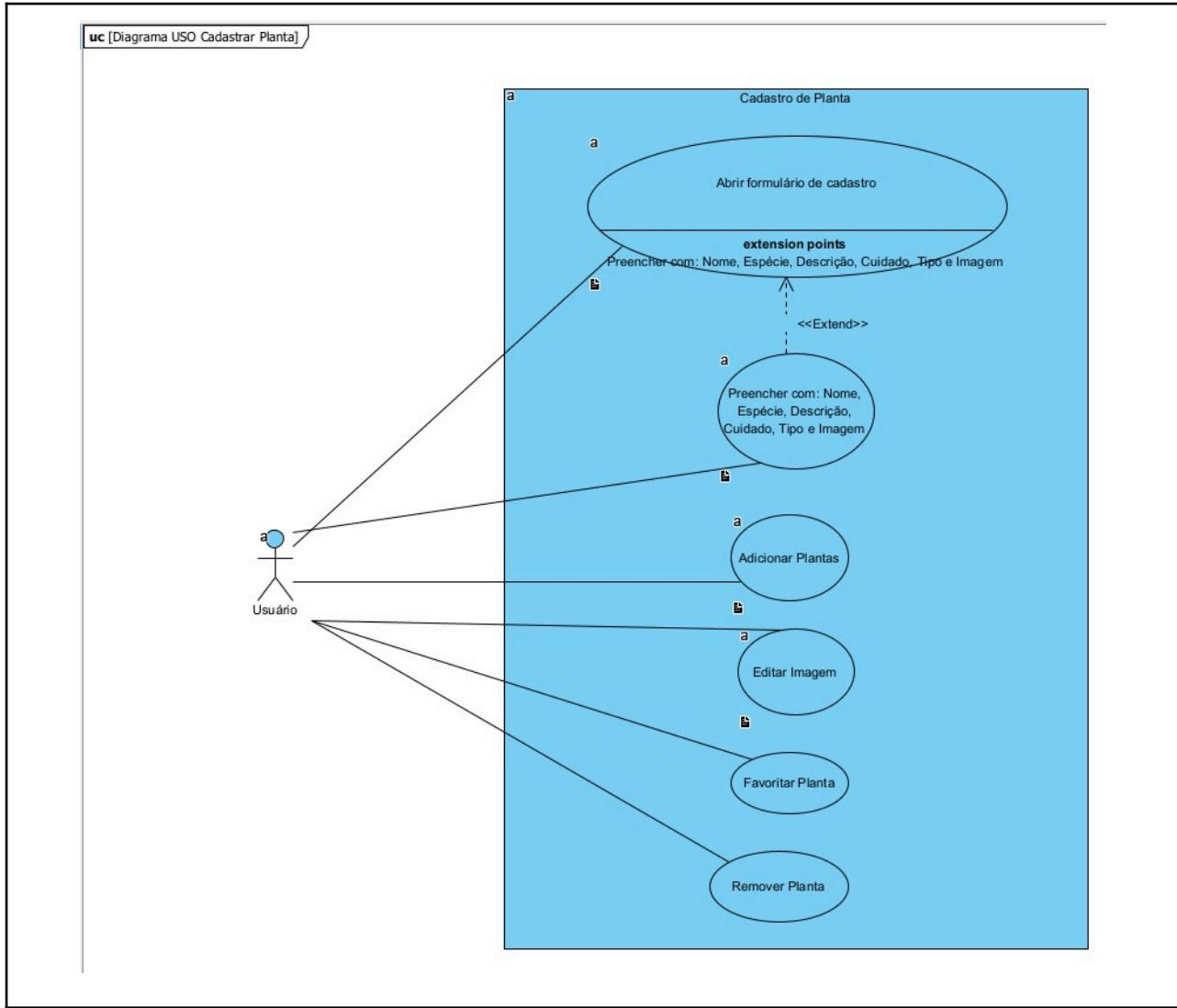
2.1.1 Diagrama de Casos de Uso

O diagrama de casos de uso é utilizado para representar as ações realizadas pelos atores (como o usuário), destacando como eles interagem com as funcionalidades do sistema.

É possível visualizar, no Diagrama 1 a seguir, o exemplo moldado para representar a interação do ator ao cadastrar uma planta.

Essas interações estão representadas graficamente, facilitando a compreensão do fluxo de uso e da lógica por trás das ações executadas pelo usuário dentro do sistema. O diagrama de casos de uso mostra as funcionalidades relacionadas ao cadastro de plantas, nas quais o ator principal, o usuário, pode realizar ações como abrir o formulário de cadastro, preencher os dados da planta (nome, espécie, descrição, cuidados, tipo e imagem), adicionar novas plantas, editar a imagem, favoritar uma planta e remover uma planta. Utilizamos a relação <<extend>> para indicar que o preenchimento dos dados é uma extensão do caso de uso “Abrir formulário de cadastro”, representando que essa ação depende do formulário estar aberto para ser realizada.

Diagrama 1 - Diagrama de caso de uso para Cadastrar Planta



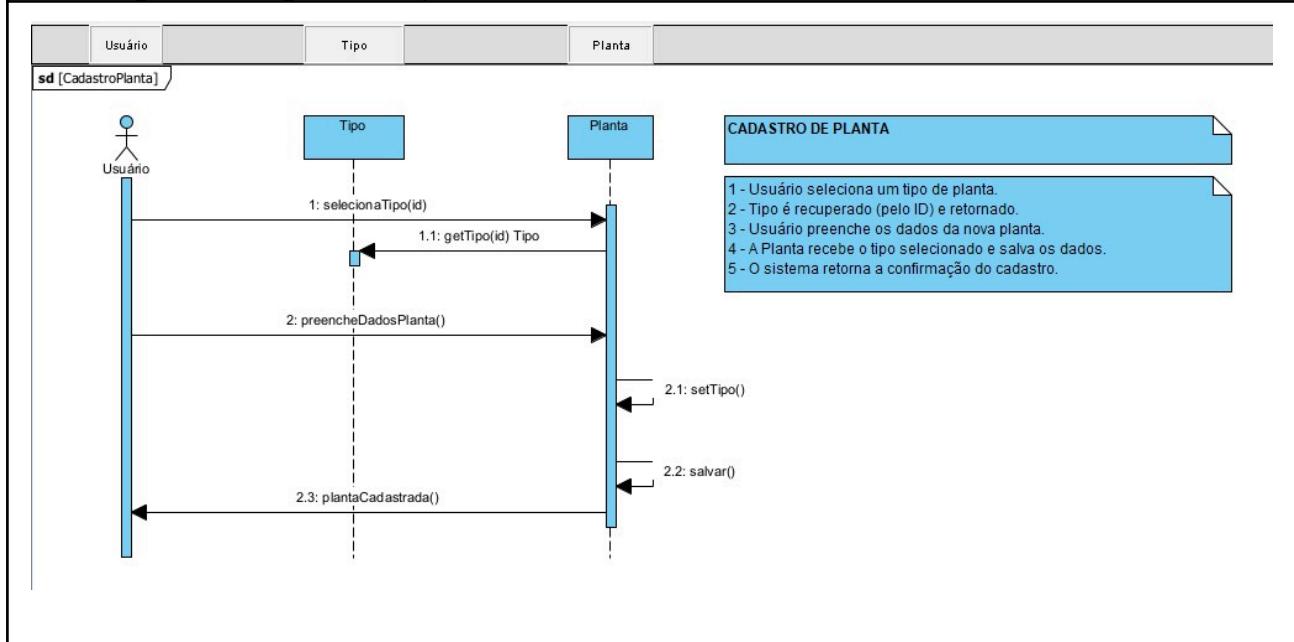
Fonte: Os autores (2025).

2.1.2 Diagrama de Sequência

O diagrama de sequência é um tipo de diagrama que representa a ordem das interações entre os objetos de um sistema ao longo do tempo. Ele mostra como os métodos são chamados, quem envia e quem recebe as mensagens, e em que sequência essas ações ocorrem durante um determinado processo ou funcionalidade. Esse tipo de diagrama é fundamental para compreender o comportamento dinâmico do sistema e garantir que as funcionalidades sigam a lógica esperada.

Criamos dois diagramas de sequência para o estudo, com o objetivo de representar o fluxo de interação entre os objetos do sistema em diferentes funcionalidades. O primeiro, chamado de Diagrama 2, pode ser visualizado logo abaixo, cujo fluxo se baseia no cadastro da planta.

Diagrama 2 - Diagrama de Sequências Cadastro de Planta

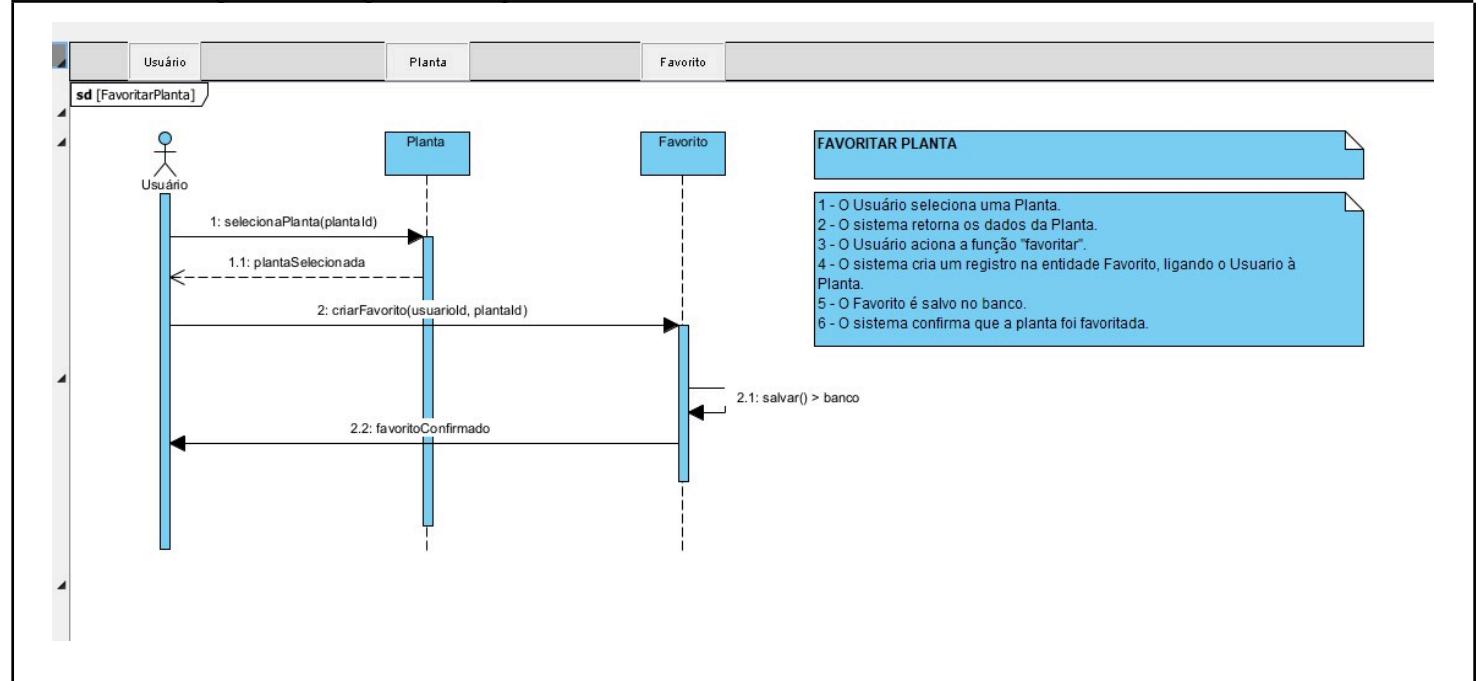


Fonte: Os autores (2025).

O processo de cadastro de uma planta envolve o usuário escolhendo o tipo da planta, que é buscado pelo ID, preenchendo os dados, e o sistema associando o tipo, salvando as informações e retornando uma confirmação do cadastro.

O segundo, chamado de Diagrama 3, apresenta o caminho da ação de favoritar a planta.

Diagrama 3 - Diagrama de Sequências Favorita Planta



Fonte: Os autores (2025).

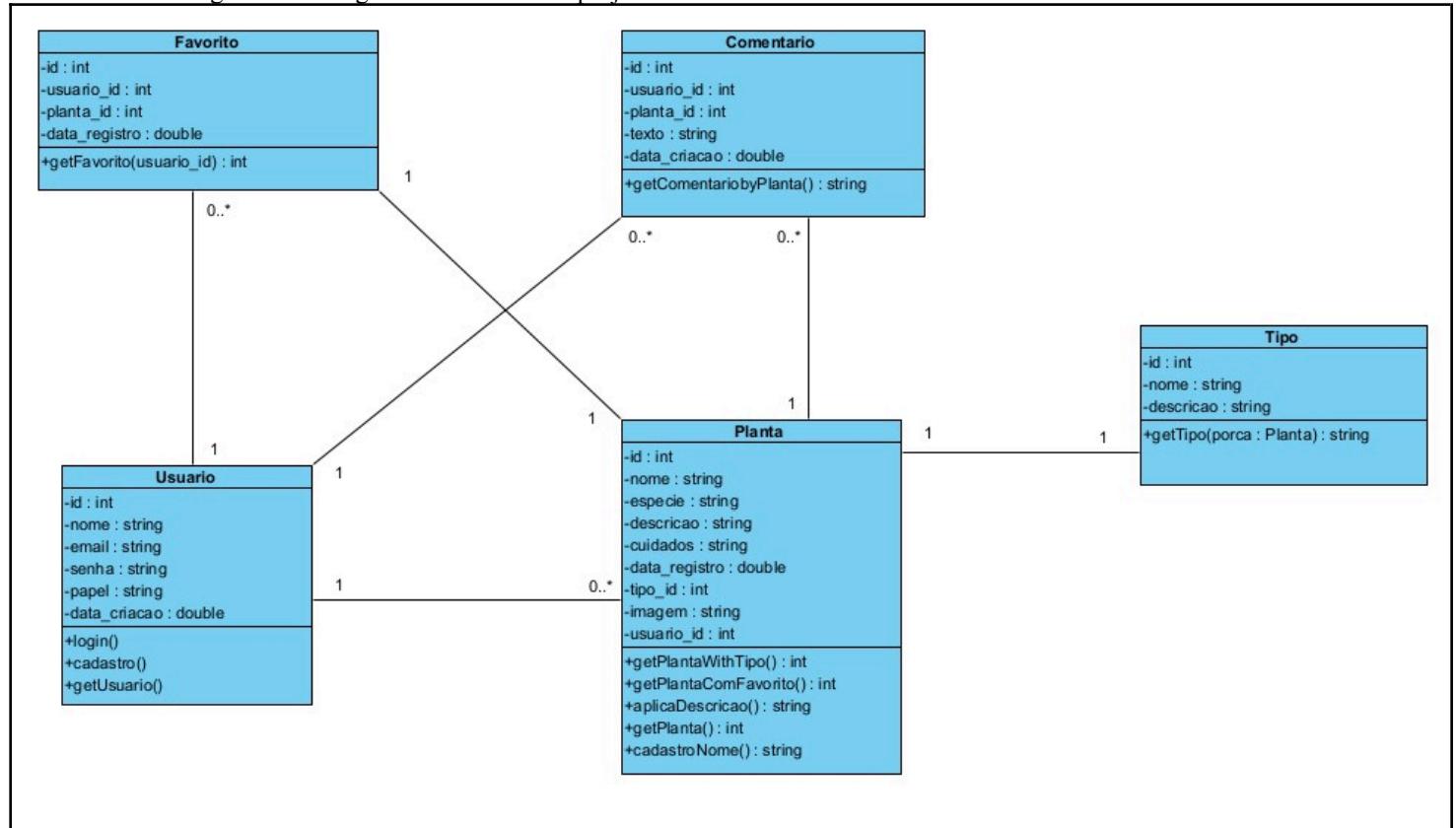
O processo de favoritar uma planta consiste no usuário selecionar a planta, o sistema exibir seus dados, o usuário clicar em “favoritar”, o sistema criar o vínculo na entidade Favorito, salvar essa informação e retornar a confirmação ao usuário.

2.1.3 Diagrama de Classes

O diagrama de classes representa a estrutura estática do sistema, mostrando as classes, seus atributos, métodos e os relacionamentos entre elas. Ele é fundamental para entender como os dados e comportamentos estão organizados e conectados no software.

No Diagrama 4, é possível analisar o diagrama de classes gerado com base na nossa aplicação PHP.

Diagrama 4 - Diagrama de Classes do projeto



Fonte: Os autores (2025).

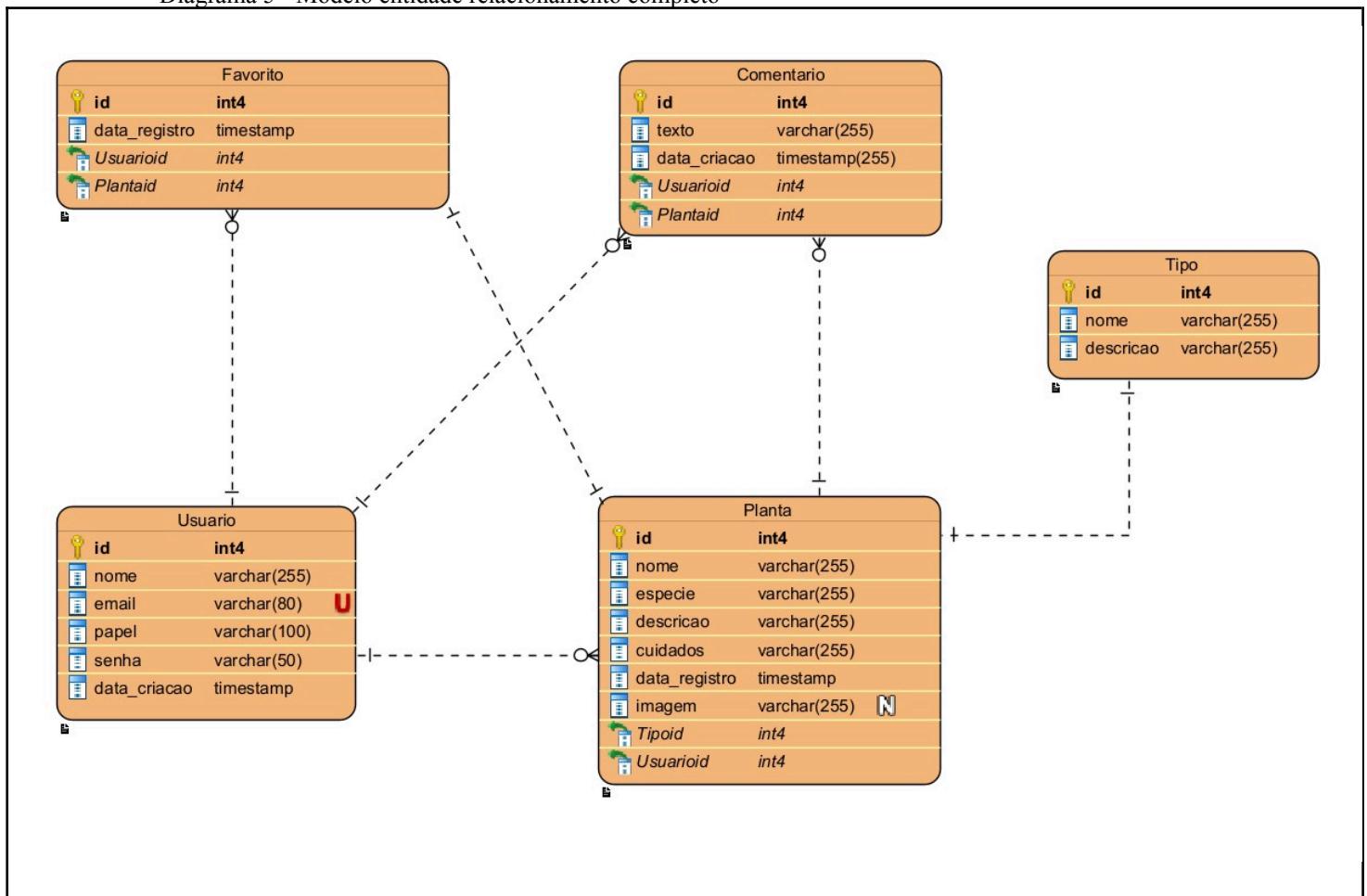
O diagrama apresenta a estrutura básica das classes da aplicação, destacando as principais entidades, seus atributos e métodos, além dos relacionamentos entre elas. Mostra como os usuários interagem com itens (como plantas), podendo criar, comentar e favoritar, enquanto os itens estão classificados por tipos e associados aos usuários, definindo as regras e conexões essenciais para o funcionamento do sistema.

2.1.4 Modelo Entidade-Relacionamento

O modelo entidade-relacionamento é uma representação visual que descreve os dados de um sistema, destacando entidades (como "Usuário" ou "Planta"), seus atributos (ex: nome, email) e os relacionamentos entre elas (ex: cadastro de plantas). Ele é usado para planejar a estrutura de um banco de dados de forma clara e lógica.

É possível visualizar, no Diagrama 5, o resultado obtido após a construção do modelo do projeto e as suas principais entidades: Usuário, Planta, Tipo, Favorito e Comentário, com seus respectivos atributos e relacionamentos.

Diagrama 5 - Modelo entidade relacionamento completo



Fonte: Os autores (2025).

2.2 VALIDAÇÃO E TESTES

2.2.1 Teste Unitário

O teste unitário é uma técnica de verificação que consiste em testar individualmente pequenas partes do código, como funções ou métodos, para garantir que cada unidade funcione corretamente isoladamente.

É possível conferir, no Teste 1, um exemplo de teste unitário realizado no modelo *UsuarioModel*.

Teste 1 - Teste Unitário no *UsuarioModel*

```

EXPLORADOR
...  ➔ UsuarioModelTest.php ×
tests > Models > UsuarioModelTest.php > PHP Intelephense > UsarioModelTest > testRegrasDeValidacao
1  <?php
2
3  namespace Tests\Models;
4
5  use CodeIgniter\Test\CIUnitTestCase;
6  use App\Models\UsuarioModel;
7
8  class UsuarioModelTest extends CIUnitTestCase
9  {
10     public function testInstanciaModel()
11     {
12         $model = new UsuarioModel();
13         $this->assertInstanceOf(UsuarioModel::class, $model);
14     }
15
16     public function testCamposPermitidos()
17     {
18         $model = new UsuarioModel();
19         $esperado = ['nome', 'email', 'senha', 'papel', 'data_criacao'];
20         $this->assertEquals($esperado, $model->allowedFields());
21     }
22
PROBLEMAS  SAÍDA  CONSOLE DE DEPURAÇÃO  TERMINAL  PORTAS
Runtime: PHP 8.2.12
Configuration: C:\xampp\htdocs\prog3-crud\crud-plantas\phpunit.xml.dist
...
Time: 00:00.148, Memory: 12.00 MB
There was 1 PHPUnit test runner warning:
1) No code coverage driver available
OK, but there were issues!
Tests: 3, Assertions: 4, PHPUnit Warnings: 1.
PS C:\xampp\htdocs\prog3-crud\crud-plantas>

```

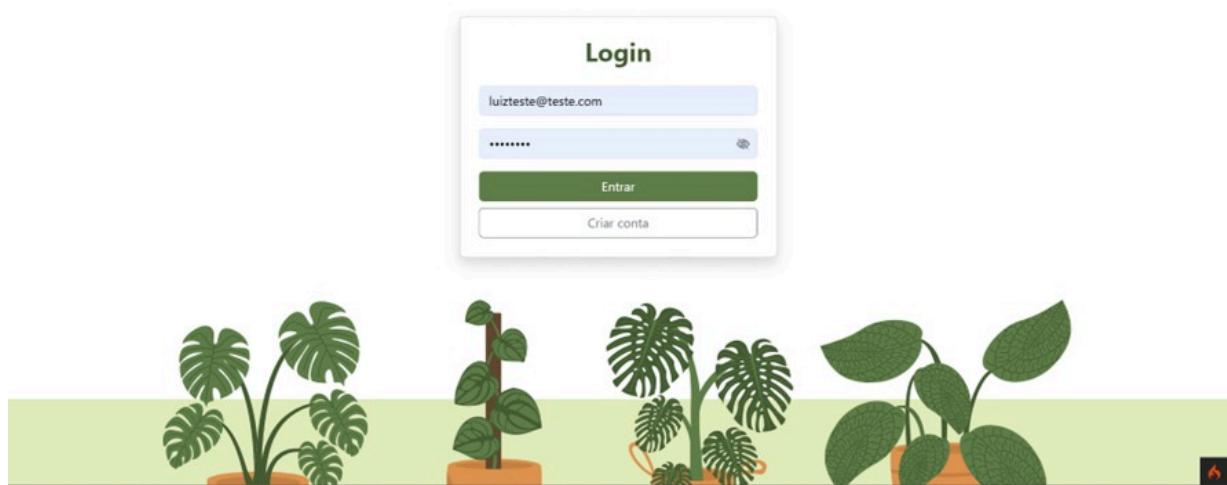
Fonte: Os autores (2025).

O teste verifica se o modelo *UsuarioModel* está funcionando corretamente. Ele contém dois métodos principais: o primeiro, *testInstanciaModel()*, garante que o modelo é aplicado corretamente. Já o segundo, *testCamposPermitidos()*, confere se os campos permitidos para preenchimento no modelo são exatamente os esperados isoladamente: nome, email, senha, papel e data_criacao. Dessa forma, ajudam a validar a estrutura e a segurança do modelo no sistema.

2.2.2 Teste de Funcionalidade

O teste de funcionalidade verifica se as funcionalidades do sistema funcionam corretamente, testando o software do ponto de vista do usuário. Ele garante que cada função cumpra o que foi especificado e que o sistema entregue os resultados esperados nas situações. Foram realizados testes funcionais com o Katalon Recorder, por meio da interface do usuário, identificados como Teste 2 ao Teste 6, para verificar se as principais funcionalidades do sistema estão funcionando corretamente.

Teste 2 - Teste De Funcionalidade - Tela de Login



Fonte: Os autores (2025).

Teste 3 - Teste De Funcionalidade - Tela Optativa



Fonte: Os autores (2025).

Teste 4 - Teste De Funcionalidade - Listagem de Plantas

Samambaia de Java - Microsorum pteropus

Bromélia - bromélias coloris

Girassol - Girassol

Girassol laranja - Girassol laranja

jiboia - jiboia

Rosa - Rosa vermelha

Fonte: Os autores (2025).

Teste 5 - Teste De Funcionalidade - Primeira tela do Katalon

Command	Target	Value
open	http://localhost/prog3-crud/crud-plantas/public/login	
click	xpath=//button[@type='submit']	
open	http://localhost/prog3-crud/crud-plantas/public/index.php	
click	link=Plantas	
open	http://localhost/prog3-crud/crud-plantas/public/plantas	
click	name=search	
type	name=search	girassol
click	xpath=//button[@type='submit']	
click	xpath=(//*[normalize-space(text()) and normalize-space(.)='Girassol'])[1]/following::a[1]	
click	link=Editar	
click	id=nome	
type	id=nome	Girassol laranja
click	id=especie	
type	id=especie	Girassol laranja
click	xpath=//button[@type='submit']	
open	http://localhost/prog3-crud/crud-plantas/public/index.php/plantas	
click	xpath=//li[4]/div[2]/button/i	

Fonte: Os autores (2025).

Teste 6 - Teste De Funcionalidade - Segunda tela do Katalon

Command	Target	Value
open	http://localhost/prog3-crud/crud-plantas/public/index.php/plantes	
click	xpath=//li[4]/div[2]/button/i	
click	link=Favoritos	
open	http://localhost/prog3-crud/crud-plantas/public/favoritos	
click	xpath=(//*[normalize-space(text()) and normalize-space(.)='Adicionado em: 24/06/2025 22:55'])[1]/following::a[1]	
open	http://localhost/prog3-crud/crud-plantas/public/plantas/view/5	
click	link=Voltar	
open	http://localhost/prog3-crud/crud-plantas/public/plantas	
click	link=Home	
open	http://localhost/prog3-crud/crud-plantas/public/	
click	link=Tipos	
click	name=search	
type	name=search	Cactos
click	xpath=//button[@type='submit']	
click	xpath=//a[@id='perfilicon']/i	
click	link=Sair	

Fonte: Os autores (2025).

O teste de funcionalidade automatizado simula um fluxo completo de uso do sistema, iniciando com o acesso à tela de login e navegação para a página de listagem de plantas. Em seguida, realiza uma busca pela planta "Girassol", acessa seus detalhes, e edita suas informações, alterando o nome e a espécie para "Girassol laranja". Após a edição, a planta é marcada como favorita e o sistema navega até a página de favoritos para confirmar sua adição.

O teste também valida a navegação entre as páginas principais do sistema, como Home e Tipos, incluindo uma busca pelo tipo "Cactos". Por fim, o fluxo é encerrado com o acesso ao menu do perfil e a execução do logout. O teste cobre ações importantes do sistema como busca, edição, relacionamento entre entidades (Planta e Favorito), navegação e controle de sessão do usuário, representando um cenário realista e bem estruturado de uso.

2.2.3 Teste de Carga

O teste de carga verifica como o sistema se comporta sob alta demanda, testando seu desempenho quando muitos usuários ou processos acessam simultaneamente para garantir estabilidade e tempo de resposta adequado. Foi utilizado o Apache JMeter para tais objetivos,

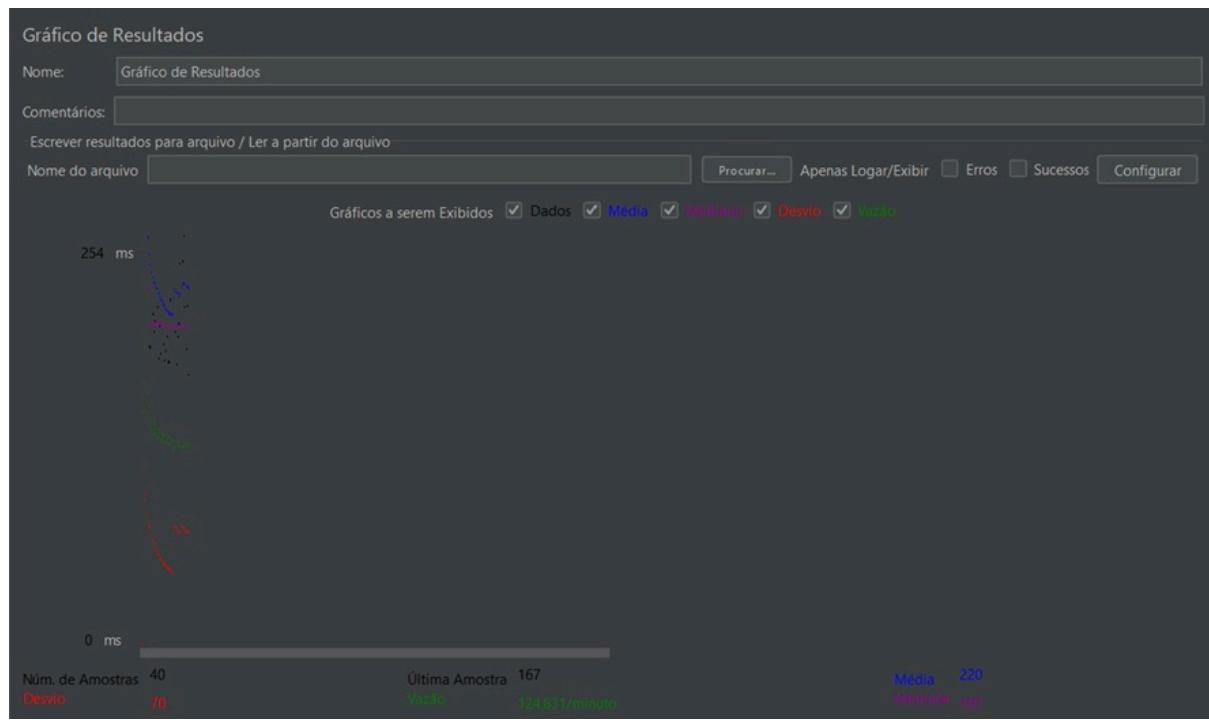
sendo possível visualizar os resultados em Teste 7 e Teste 8, onde foram avaliados o desempenho e a estabilidade do sistema sob carga.

Teste 7 - Teste De Carga - Relatório de Sumário

Relatório de Sumário										
Nome: Relatório de Sumário Comentários: Escrever resultados para arquivo / Ler a partir do arquivo Nome do arquivo <input type="text"/> <input type="button" value="Procurar..."/> Apenas Logar/Exibir <input type="checkbox"/> Erros <input type="checkbox"/> Sucessos <input type="button" value="Configurar"/>										
Rótulo	# Amostras	Média	Mín.	Máx.	Desvio Padrão	% de Erro	Vazão	KB/s	Sent KB/sec	Média de Byte..
Requisição H...	40	220	167	495	70,03	0,00%	2,1/sec	65,84	0,30	32407,5
TOTAL	40	220	167	495	70,03	0,00%	2,1/sec	65,84	0,30	32407,5

Fonte: Os autores (2025).

Teste 8 - Teste De Carga - Gráfico de Resultados



Fonte: Os autores (2025).

O teste de carga simula 20 usuários acessando a página de login da aplicação local, com *ramp up* de 20 segundos e duas repetições por usuário. A requisição é um HTTP GET para o endereço `http://localhost/prog3-crud/crud-plantas/public/login`. Foram adicionados três elementos para análise dos resultados: Relatório de Sumário, Árvore de Resultados e Gráfico de Resultados. O objetivo é avaliar o desempenho da aplicação sob carga para garantir que ela suporte um número elevado de usuários ou requisições simultâneas.

2.3 USO DE PADRÕES DE PROJETO

O projeto utiliza dois padrões de projeto fundamentais: MVC (*Model-View-Controller*) e *Filter*. O padrão MVC organiza a aplicação em três camadas: Model, responsável pela lógica de negócio e acesso ao banco de dados; View, que cuida da interface e apresentação dos dados ao usuário; e Controller, que atua como intermediário, recebendo requisições, processando regras e retornando as respostas. Essa separação melhora a organização do código e facilita a manutenção.

2.3.1 MVC

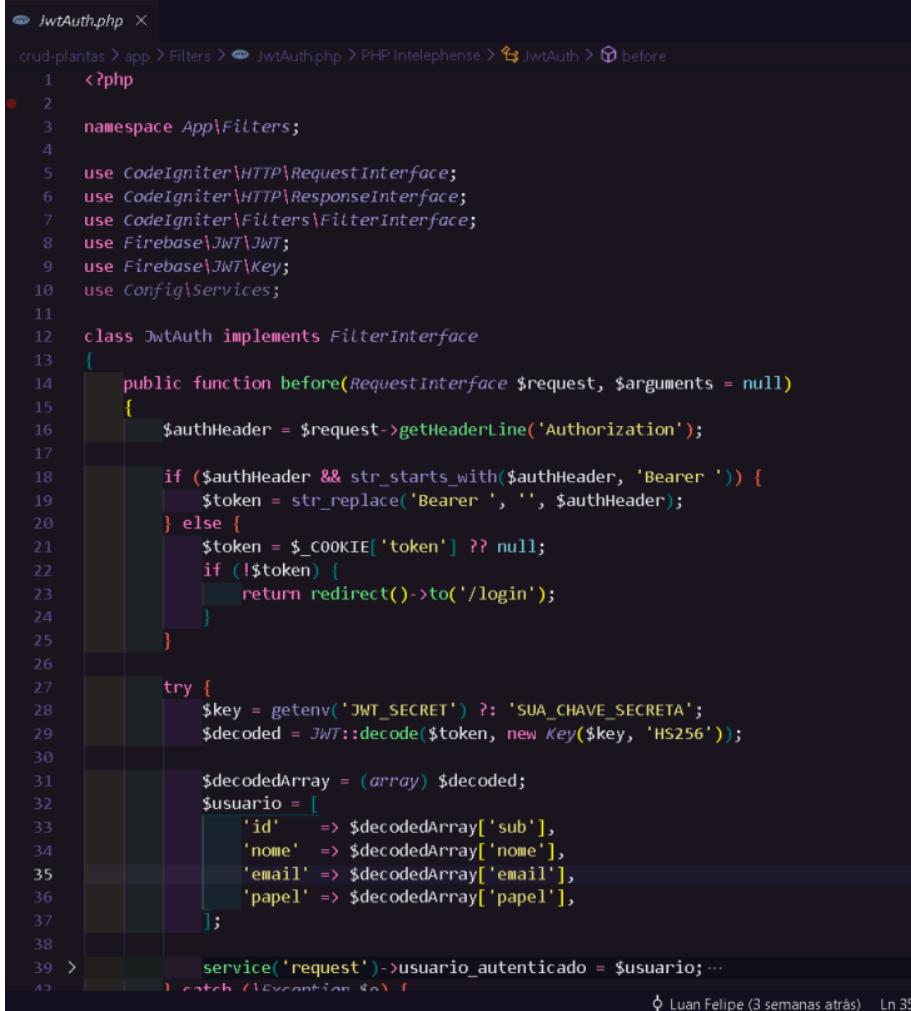
No projeto, o padrão MVC é aplicado de forma clara: modelos como `PlantaModel.php` e `ComentarioModel.php` lidam com o banco de dados, as *views* estão em `app/Views/`, exibindo dados ao usuário, e os controllers como `Plantas.php` controlam o fluxo entre modelo e visualização. Essa estrutura torna o sistema mais testável, reutilizável e fácil de entender.

2.3.2 Filter

O padrão *Filter* permite aplicar regras centralizadas antes ou depois das requisições, sendo útil para autenticação, autorização e controle de acesso. No CodeIgniter 4, os filtros ficam em `app/Filters/` e são configurados em `app/Config/Filters.php`. Neste projeto, eles protegem rotas sensíveis, como criação e edição de dados, verificando, por exemplo, se um token JWT é válido antes de permitir o acesso.

Apresenta-se, no Padrão 1, a utilização do padrão em nosso projeto, cujo objetivo é autenticação e autorização de usuários.

Padrão 1 - Padrão de Projeto Filter em app/Filters/



```

JwtAuth.php ×
crud-plantas > app > Filters > JwtAuth.php > PHP Intelephense > JwtAuth > before
1  <?php
2
3  namespace App\Filters;
4
5  use CodeIgniter\HTTP\RequestInterface;
6  use CodeIgniter\HTTP\ResponseInterface;
7  use CodeIgniter\Filters\FilterInterface;
8  use Firebase\JWT\JWT;
9  use Firebase\JWT\Key;
10 use Config\Services;
11
12 class JwtAuth implements FilterInterface
13 {
14     public function before(RequestInterface $request, $arguments = null)
15     {
16         $authHeader = $request->getHeaderLine('Authorization');
17
18         if ($authHeader && str_starts_with($authHeader, 'Bearer ')) {
19             $token = str_replace('Bearer ', '', $authHeader);
20         } else {
21             $token = $_COOKIE['token'] ?? null;
22             if (!$token) {
23                 return redirect()->to('/login');
24             }
25         }
26
27         try {
28             $key = getenv('JWT_SECRET') ?: 'SUA_CHAVE_SECRETA';
29             $decoded = JWT::decode($token, new Key($key, 'HS256'));
30
31             $decodedArray = (array) $decoded;
32             $usuario = [
33                 'id'      => $decodedArray['sub'],
34                 'nome'    => $decodedArray['nome'],
35                 'email'   => $decodedArray['email'],
36                 'papel'   => $decodedArray['papel'],
37             ];
38
39         >         service('request')->usuario_autenticado = $usuario;
40         >     } catch (\Exception $e) {
41
42
43
44
45
46
47
48
49
49 >     }
49 > }
49 > Luan Felipe (3 semanas atrás) Ln 35

```

Fonte: Os autores (2025).

Com o uso de *Filters*, o sistema ganha em segurança e modularidade. Ele evita duplicação de lógica nos controllers e permite que mudanças na autenticação sejam feitas diretamente nos filtros, sem impactar outras partes do sistema. Juntos, os padrões MVC e Filter tornam a aplicação mais organizada, segura e de fácil manutenção.

2.4 JUSTIFICATIVA DO FRAMEWORK ESCOLHIDO

De acordo com Pandolfi (2013), o CodeIgniter é um framework voltado ao desenvolvimento de aplicações PHP e utiliza o padrão MVC (*Model-View-Controller*) para organizar o código de forma mais eficiente. Ele fornece diversas bibliotecas para tarefas comuns, simplificando a criação de interfaces e a lógica de acesso aos recursos, o que contribui para uma produção mais rápida e com menos código. Sua estrutura facilita a

comunicação entre as camadas do sistema por meio de arrays ou objetos, promovendo uma integração simples e eficiente entre os componentes.

O padrão MVC é um paradigma de arquitetura de desenvolvimento web que implica que a lógica de negócio de qualquer aplicativo deve ser separada da apresentação. Desta forma, o padrão separa as três camadas do projeto de desenvolvimento em:

Modelo/Model – lida com o banco de dados, executa cálculos e muito mais. Em resumo, é onde está localizada a lógica de negócio e a Implementação ORM.

Visão/View – forma a camada de apresentação do aplicativo, na qual os dados dos modelos são incorporados.

Controlador/Controller – encaminha as solicitações do usuário para o modelo apropriado.

Nesse sentido, optou-se pela construção do projeto utilizando o framework descrito devido à sua versatilidade e baixa curva de aprendizado, o que permitiu que o projeto e seu desenvolvimento ocorram de forma mais ágil, possibilitando a compreensão e a adaptação rápida dos desenvolvedores ao ambiente de desenvolvimento. A escolha pelo framework também se justifica pela sua documentação clara e comunidade ativa, fatores que contribuem para a resolução de dúvidas e para a manutenção do sistema no longo prazo.

Além disso, segundo Pandolfi (2013), o uso dos controladores no *CodeIgniter* auxilia no processamento e na requisição das páginas web, atuando como intermediários entre os modelos e as visualizações. Eles são responsáveis por coordenar a lógica da aplicação, recebendo as requisições do usuário, interagindo com os dados necessários e retornando as respostas apropriadas por meio das interfaces, o que foi de fundamental importância para com o projeto, sendo possível compreender as etapas de maneira altamente satisfatória.

2.5 DIRETÓRIO DE PASTAS E ARQUIVOS ABSTRAÍDOS DO CODEIGNITER

Após a configuração e a instalação do *CodeIgniter*, o framework organiza automaticamente a estrutura do projeto em diretórios específicos, cada um com uma função definida. Os principais diretórios são: *app* (contém os arquivos da aplicação como *controllers*, *models* e *views*, *system* (armazena os arquivos do núcleo do framework) e *public* (fica os arquivos acessíveis pelo navegador, como imagens, folhas de estilo e scripts). Dentro da pasta *app*, destacam-se os diretórios *controllers*, responsáveis pela lógica de controle; *models*, que manipulam os dados e se comunicam com o banco e servem de base para a implementação ORM; e *views*, responsáveis pela apresentação da interface ao usuário.

Abaixo, na Lista 1, é possível visualizar a organização das pastas e diretórios dentro do editor de código utilizado.

Lista 1 - Diretório de Pastas e Arquivos do Codeigniter



Fonte: Os autores (2025).

2.6 MODELAGEM DO BANCO DE DADOS

A conexão com o banco de dados é a primeira necessidade presente em qualquer sistema web que necessite armazenar dados de forma persistente. Para isso, foi utilizado o SGBD MySQL devido à sua integração nativa com a linguagem, ampla documentação, facilidade de uso e bom desempenho, além de se adaptar bem a projetos de pequeno e médio porte, oferecendo recursos eficientes para armazenamento e manipulação de dados (Bauer, 2018).

O CodeIgniter, por sua vez, possui bibliotecas que facilitam a conexão e o uso do MySQL, tornando o desenvolvimento mais ágil e padronizado. Com a conexão ao banco de dados estabelecida, é necessário criar os dicionários de metadados, cada entidade presente no

sistema necessita de um, pois as tabelas e relações presentes no banco de dados são criadas através das informações presentes no dicionário de metadados (Bauer, 2018).

2.6.1 Criação das tabelas no DBeaver

Em primeiro lugar, considerando o fato de que em uma aplicação é necessário a manipulação dos dados de uma tabela no banco de dados, foram criadas as tabelas para armazenar as informações das plantas e dos tipos. Essas tabelas desenvolvidas utilizando o MySQL, permitiram a organização e o relacionamento dos dados de forma eficiente, possibilitando consultas otimizadas e facilitando o desenvolvimento de funcionalidades.

Abaixo, na Imagem 1, é possível visualizar alguns exemplos de tabelas criadas durante o andamento do projeto.

Imagen 1 - Exemplo de criação das tabelas no DBeaver

```
CREATE TABLE tipos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    descricao TEXT
);

CREATE TABLE plantas (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    especie VARCHAR(100),
    descricao TEXT,
    cuidados TEXT,
    data_registro TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    tipo_id INT,
    FOREIGN KEY (tipo_id) REFERENCES tipos(id)
);
```

Fonte: Os autores (2025).

2.6.2 Uso das Migrations

De acordo com Eduardo (2016), as *migrations* são uma maneira de controlar as versões da estrutura do banco de dados. Elas permitem criar, alterar ou remover tabelas de forma controlada, acompanhando as mudanças do sistema. Assim, ao invés de fazer alterações diretamente no banco de dados manualmente, as *migrations* garantem que essas mudanças ocorram de maneira automatizada, sincronizada com o código da aplicação.

Existem dois métodos utilizados nas *migrations*: o *up()* e o *down()*. O método *up()* é responsável por aplicar alterações no banco de dados, como a criação de tabelas ou adição de colunas, servindo como atualizações na estrutura da aplicação. Já o método *down()* realiza o

processo inverso, desfazendo ou removendo as alterações feitas, como a exclusão de colunas ou tabelas adicionadas anteriormente (Eduardo, 2016).

No projeto, o método *up()* das *migrations* foi usado para criar e definir a estrutura das tabelas no banco de dados. Essa etapa serve apenas para estruturar a tabela e não insere dados nela. Já o método *down()* desfaz as alterações feitas pelo *up()*, removendo a tabela criada para possibilitar a reversão da estrutura do banco. Assim, as *migrations* garantem que a criação, alteração e remoção das tabelas ocorram de forma organizada, controlada e sincronizada com o código da aplicação.

A seguir, na Imagem 2, é possível visualizar o código-fonte das *migrations* utilizadas no projeto.

Imagen 2 - *Migration* das tabelas

```
crud-plantas> app > Database > Migrations > 2025-06-03-172902_CreateTables.php > PHP Intelephense > CreateTables > down
1  <?php
2
3  namespace App\Database\Migrations;
4
5  use CodeIgniter\Database\Migration;
6
7  class CreateTables extends Migration
8  {
9      public function up()
10     {
11         // Tipos
12         $this->forge->addField([
13             'id'          => ['type' => 'INT', 'auto_increment' => true],
14             'nome'        => ['type' => 'VARCHAR', 'constraint' => 100, 'null' => false],
15             'descricao'   => ['type' => 'TEXT', 'null' => true],
16         ]);
17         $this->forge->addKey('id', true);
18         $this->forge->createTable('tipo', true);
19
20         // Usuarios
21         $this->forge->addField([
22             'id'          => ['type' => 'INT', 'auto_increment' => true],
23             'nome'        => ['type' => 'VARCHAR', 'constraint' => 100, 'null' => false],
24             'email'       => ['type' => 'VARCHAR', 'constraint' => 100, 'null' => false, 'unique' => true],
25             'senha'       => ['type' => 'VARCHAR', 'constraint' => 255, 'null' => false],
26             'papel'       => ['type' => 'ENUM', 'constraint' => ['comum', 'admin'], 'null' => false],
27             'data_criacao' => ['type' => 'DATETIME', 'null' => false],
28         ]);
29         $this->forge->addKey('id', true);
30         $this->forge->createTable('usuario', true);
31     }
}
```

Fonte: Os autores (2025).

2.7 ELABORAÇÃO DO PROJETO

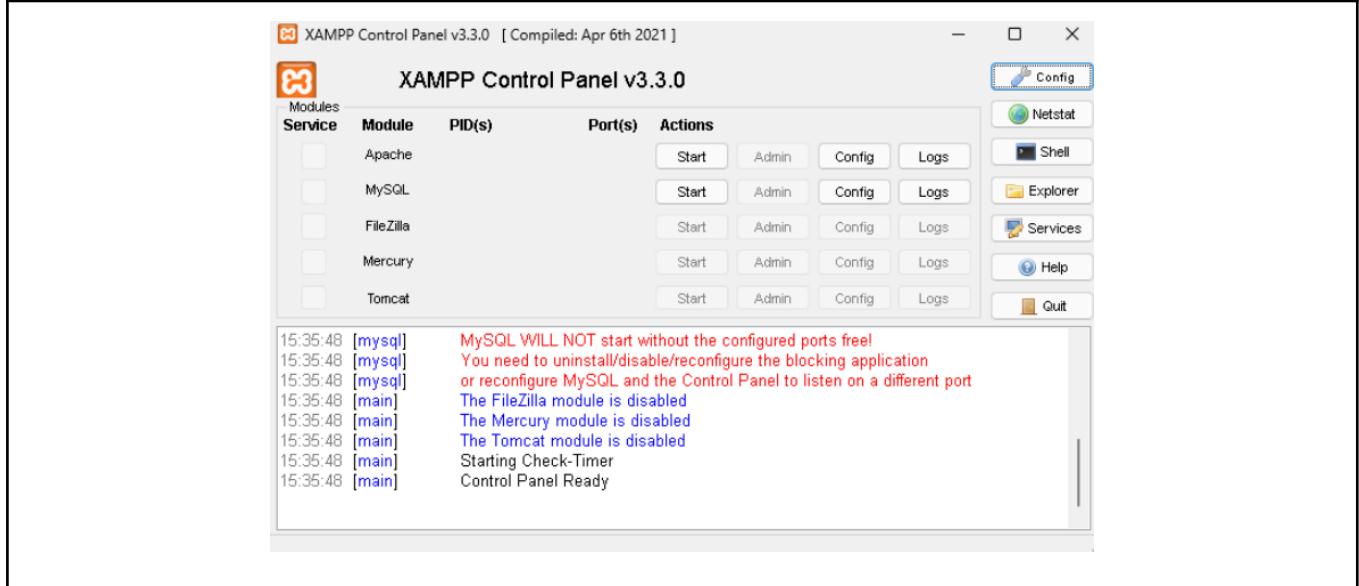
Para o desenvolvimento do sistema, utilizou-se CodeIgniter em conjunto com o Composer para o gerenciamento de dependências e a linguagem PHP como base da aplicação, seguindo o padrão MVC (*Model-View-Controller*). A seguir, serão apresentados os principais elementos que compõem o CRUD das plantas ornamentais, com destaque para a implementação dos *controllers*, *views* e *models*, que desempenham papéis essenciais no projeto.

2.7.1 Uso do XAMPP

Conforme Costa e Borges (2023), “o XAMPP é muito útil para desenvolvedores web, pois permite que eles configurem um ambiente de desenvolvimento local que simula um servidor web real, facilitando o desenvolvimento e teste de sites e aplicativos antes de implantá-los em servidores de produção”. Nesse sentido, foi utilizada tal ferramenta para configurar um ambiente local que simula um servidor real, possibilitando o desenvolvimento, testes e validações do sistema de forma prática e segura.

O XAMPP foi essencial no projeto, oferecendo um ambiente local com MySQL e suporte ao CodeIgniter, o que facilitou o desenvolvimento, os testes e a análise da integração do sistema. Na Imagem 3, é possível visualizar a interface do software antes de ser iniciada, com seus serviços de Apache e MySQL desligados.

Imagen 3 - Uso do XAMPP



Fonte: Os autores (2025).

2.7.2 Instalação do Composer

Uma das primeiras etapas do projeto foi a instalação do Composer como gerenciador de dependências, ferramenta essencial para automatizar a instalação, atualização e remoção de bibliotecas externas. O Composer permite que o desenvolvedor defina, de forma clara, quais pacotes são necessários para o funcionamento do sistema, assumindo a responsabilidade de gerenciá-los ao longo do ciclo de vida do projeto. Dessa forma, garante-se maior organização, controle e eficiência no uso de recursos externos. (Andrade, 2020).

Para isso, utilizamos o comando *composer install* para realizar a instalação do mesmo. Dessa maneira, foi possível prosseguir com o framework e o corpo do projeto pré-moldado.

2.7.3 Arquivo *README.md* do CodeIgniter

Ao iniciar o uso do CodeIgniter, um dos primeiros arquivos de referência disponíveis é o README. Esse documento, localizado na raiz do framework, fornece informações iniciais importantes sobre a estrutura do CodeIgniter, requisitos do sistema, instruções de instalação e orientações básicas para configuração, servindo como guia introdutório do framework.

Na Imagem 4, é possível observar o uso do arquivo no projeto e como suas instruções influenciam no escopo do trabalho como um todo.

Imagen 4 - *README.md* do Projeto

```
crud-plantas > README.md > ...
1 | "# CodeIgniter 4 Application Starter
2 |
3 | ## What is CodeIgniter?
4 |
5 | CodeIgniter is a PHP full-stack web framework that is light, fast, flexible and secure.
6 | More information can be found at the [official site](https://codeigniter.com).
7 |
8 | This repository holds a composer-installable app starter.
9 | It has been built from the
10 | [development repository](https://github.com/codeigniter4/CodeIgniter4).
11 |
12 | More information about the plans for version 4 can be found in [CodeIgniter 4](https://forum.codeigniter.com/forumdisplay.php?fid=28) on the forums.
13 |
14 | You can read the [user guide](https://codeigniter.com/user_guide/)
15 | corresponding to the latest version of the framework.
16 |
17 | ## Installation & updates
18 |
19 | `composer create-project codeigniter4/appstarter` then `composer update` whenever
20 | there is a new release of the framework.
21 |
22 | When updating, check the release notes to see if there are any changes you might need to apply
23 | to your `app` folder. The affected files can be copied or merged from
24 | `vendor/codeigniter4/framework/app`.
25 |
26 | ## Setup
27 |
28 | Copy `env` to `.env` and tailor for your app, specifically the baseURL
29 | and any database settings.
```

Fonte: Os autores (2025).

2.7.4 MVC (*Model-View-Controller*)

De maneira a fornecer uma divisão de funcionalidades, o modelo MVC é uma aplicação já usada amplamente no desenvolvimento de sistemas, por permitir uma separação clara entre a lógica de negócio (*Model*), a interface com o usuário (*View*) e o controle das requisições (*Controller*). Essa abordagem facilita a organização do código, promove a reutilização de componentes e torna a manutenção do sistema mais eficiente. No contexto

deste trabalho, o padrão MVC foi aplicado por meio do framework CodeIgniter, que estrutura o projeto de forma a seguir essa divisão de maneira padronizada e intuitiva (Andrade, 2011).

De acordo com Andrade (2011), nesse tipo de arquitetura, o *Model* representa os dados da aplicação e as regras de negócio que governam o acesso e a modificação desses dados, mantendo o estado da aplicação, servindo como base para a lógica do sistema e sendo responsável por interagir diretamente com o banco de dados. Dentro da camada *Model*, é comum utilizar o padrão ORM (*Object-Relational Mapping*), que permite interagir com o banco de dados por meio de objetos em vez de comandos SQL diretos, tornando o acesso aos dados mais intuitivo, seguro e orientado a objetos.

O ORM atua justamente nessa transição, permitindo que operações como inserção, leitura, atualização e exclusão sejam executadas diretamente por métodos das classes, sem a necessidade de escrever comandos SQL. Dessa forma, o desenvolvimento torna-se mais produtivo, padronizado e alinhado à lógica do sistema, além de facilitar a manutenção e a escalabilidade do código.

Na Imagem 5, é possível visualizar um exemplo da utilização de um *Model* em *PlantaModel*.

Imagen 5 - Exemplo do *Model* *PlantaModel*

```
onid-plantas > app > Models > PlantaModel.php > PHP Intelephense > PlantaModel > getPlantasComFavorito
1 <?php
2
3 namespace App\Models;
4
5 use CodeIgniter\Model;
6
7 class PlantaModel extends Model
8 {
9     protected $table      = 'planta';
10    protected $primaryKey = 'id';
11
12    protected $allowedFields = ['nome', 'especie', 'descricao', 'cuidados', 'data_registro', 'tipo_id', 'imagem', 'usuario_id'];
13
14    protected $useTimestamps = true;
15    protected $createdField = 'data_registro';
16    protected $updatedField = '';
17
18    protected $validationRules = [
19        'nome'          => 'required|min_length[3]|max_length[100]',
20        'especie'        => 'permit_empty|min_length[3]|max_length[100]',
21        'descricao'      => 'permit_empty|string',
22        'cuidados'       => 'permit_empty|string',
23        'usuario_id'    => 'required|integer',
24        'imagem'         => 'permit_empty|string',
25    ];
26
27    public function getPlantaWithTipo($id)
28    {
29        return $this->select('planta.*', 'tipo.nome as tipo_nome')
30            ->join('tipo', 'tipo.id = planta.tipo_id', 'left')
31            ->where('planta.id', $id)
32            ->first();
33    }
34
35    public function getPlantasComFavorito($usuario_id)
36    {
37        $builder = $this->db->table('planta p')
38            ->select(['p.id', 'p.nome', 'p.especie', 'p.descricao', 'p.cuidados', 'p.data_registro', 'p.tipo_id', 'p.usuario_id as usuario_cadastro_id', 'IF(f.usuario_id IS NOT NULL, 1, 0) as favorito', 'p.imagem'])
39            ->join('favorito f', 'p.id = f.planta_id AND f.usuario_id = ' . (int)$usuario_id, 'left')
40            ->orderBy('p.nome', 'ASC');
41
42        $query = $builder->get();
43
44        $result = $query->getResultArray();
45
46    }
47
48 >
```

Fonte: Os autores (2025).

Neste exemplo indicado, o *Model* mapeia a tabela *plantas* no banco de dados, especificando seus campos permitidos e gerenciando automaticamente as datas de criação/atualização. Ele também valida os dados (*name*, *especie*, etc...) e inclui um método (*getPlantasWithTipo()*) para consultar plantas e seus respectivos tipos.

As *Views*, no contexto do padrão MVC, são responsáveis por exibir ao usuário as informações processadas pela aplicação. Segundo Andrade (2011), um componente de visualização renderiza o conteúdo de uma parte específica do *Model*, acessando os dados por meio do *Controller* e definindo como essas informações devem ser apresentadas. Em outras palavras, a *View* representa a interface visível do sistema, podendo ser uma página web completa, como foi no caso do CRUD moldado, ou apenas um trecho dela, sempre com o objetivo de tornar a interação com o usuário clara e funcional.

A Imagem 6 apresenta um exemplo prático de *View* utilizada no repositório das Plantas Ornamentais, ilustrando como os dados processados pelo sistema são organizados e exibidos ao usuário final.

Imagen 6 - Exemplo de *View* da Tela Inicial

```
crud-plantas > app > Views > home > index.php > ...
1  <?php
2  helper('url');
3  $isHomePage = true;
4  $exibirHeader = false;
5  include APPPATH . 'Views/templates/header.php';
6  ?>
7
8  <div class="text-center" style="margin-top: 100px;">
9      <h2 class="fw-bold" style="color: #455c34; font-size: 3.2rem;">
10         Do que sua planta precisa?
11     </h2>
12     <p class="lead text-secondary">Use os botões abaixo para acessar os módulos:</p>
13
14     <div class="d-flex justify-content-center flex-wrap gap-3 mb-4">
15         <a href="= base_url('plantas') ?" class="btn px-4 py-2" style="background-color: #5f7e49; color: white;">
16             <i class="bi bi-flower1"></i> Plantas
17         </a>
18         <a href="= base_url('tipos') ?" class="btn btn-primary px-4 py-2">
19             <i class="bi bi-tags"></i> Tipos
20         </a>
21         <a href="= base_url('favoritos') ?" class="btn btn-warning px-4 py-2">
22             <i class="bi bi-star-fill"></i> Favoritos
23         </a>
24     </div>
25 </div>
26
27 <?php include APPPATH . 'Views/templates/footer.php'; ?>
```

Fonte: Os autores (2025).

A *View* exemplificada atua como o painel principal ou página inicial do Sistema de Cadastro de Plantas Ornamentais. Ela dá as boas-vindas ao usuário e fornece links de

navegação rápidos para os diferentes módulos da aplicação, como gerenciar plantas, gerenciar tipos e ver favoritos.

Partindo agora para o *Controller*, ele é reconhecido como o elo central entre os componentes da aplicação, sendo responsável por intermediar a comunicação entre o *Model* e a *View*. O *Controller* interpreta as requisições do usuário, como cliques e seleções, e define quais ações devem ser executadas. Ele processa essas interações, aciona os métodos adequados do *Model* e, com base no resultado obtido, direciona a *View* que será exibida como resposta. Dessa forma, o *Controller* garante que a lógica da aplicação seja executada corretamente, mantendo a fluidez entre entrada, processamento e saída de dados (Andrade, 2011).

É apresentado, na Imagem 7, um exemplo de *Controller* construído para o Cadastro de Plantas cuja função é gerenciar as operações fundamentais de criar, ler, atualizar e deletar as plantas do sistema. Ele lida com o salvamento de novas plantas (*store()*), a preparação para a edição de plantas existentes (*edit(\$id)*) e a atualização dessas plantas, incluindo o gerenciamento de upload e exclusão de imagens (*update(\$id)*). Para realizar essas operações, o *Controller* acessa e interage com os *Models* *PlantaModel* e *TipoModel*.

Imagen 7 - Exemplo do *Controller* *Plantas.php*

```

crud-plantas > app > Controllers > Plantas.php > PHP Intelephense > index
1  <?php
2
3  namespace App\Controllers;
4
5  use App\Models\PlantaModel;
6  use App\Models\TipoModel;
7  use CodeIgniter\Controller;
8  use Firebase\JWT\JWT;
9  use Firebase\JWT\Key;
10
11 class Plantas extends Controller
12 {
13     public function index()
14     {
15         $usuario_id = 0;
16         $key = getenv('JWT_SECRET') ?: 'SUA_CHAVE_SECRETA';
17
18         $jwt = $this->request->getCookie('token');
19
20         if ($jwt) {
21             try {
22                 $decoded = \Firebase\JWT\JWT::decode($jwt, new \Firebase\JWT\Key($key, 'HS256'));
23                 $usuario_id = isset($decoded->sub) ? (int)$decoded->sub : 0;
24             } catch (\Exception $e) {
25             }
26         }
27
28         $model = new \App\Models\PlantaModel();
29         $plantas = $model->getPlantasComFavorito($usuario_id);
30
31         foreach ($plantas as &$planta) {
32             $planta['favorito'] = (bool)$planta['favorito'];
33         }
34
35         $data['plantas'] = $plantas;
36         $data['usuario_logado_id'] = $usuario_id;

```

Fonte: Os autores (2025).

Em geral, o padrão MVC proporciona uma estrutura organizada que separa claramente as responsabilidades dentro de uma aplicação, facilitando o desenvolvimento, a manutenção e a escalabilidade do sistema. Ao distribuir as tarefas entre *Model*, *View* e *Controller*, o desenvolvimento tornou-se mais eficiente e o código mais fácil de entender e modificar.

2.7.5 Arquivo .env

No projeto em CodeIgniter, o arquivo `.env` (*environment*) foi utilizado com o objetivo de definir as variáveis de ambiente que controlam o comportamento da aplicação em diferentes contextos, como desenvolvimento, teste ou produção. Esse arquivo teve importância fundamental no projeto, pois permitiu manter as configurações sensíveis fora do código-fonte, onde são facilmente adaptáveis para diferentes ambientes, apenas trocando os valores conforme necessário. É apresentada, na Imagem 8, a interface do arquivo `.env` no projeto, além de suas especificações e padrões aplicados.

Imagen 8 - Arquivo `.env`

```

16
17 CI_ENVIRONMENT = development
18
19 #-----
20 # APP
21 #-----
22
23 # app.baseURL = ''
24 # If you have trouble with '.', you could also use '_'.
25 app.baseURL = 'http://localhost/prog3-crud/crud-plantas/public/'
26 # app.forceGlobalSecureRequests = false
27 # app.CSPEnabled = false
28 |
29 #-----
30 # DATABASE
31 #-----
32
33 database.default.hostname = 127.0.0.1
34 database.default.database = plantas_db
35 database.default.username = root
36 database.default.password = mysql
37 database.default.DBDriver = MySQLi
38 # database.default.DBPrefix =
39 database.default.port = 3306
40
41 # If you use MySQLi as tests, first update the values of Config\Database::$tests.
42 # database.tests.hostname = localhost
43 # database.tests.database = ci4_test
44 # database.tests.username = root
45 # database.tests.password = root
46 # database.tests.DBDriver = MySQLi
47 # database.tests.DBPrefix =
48 # database.tests.charset = utf8mb4

```

Fonte: Os autores (2025).

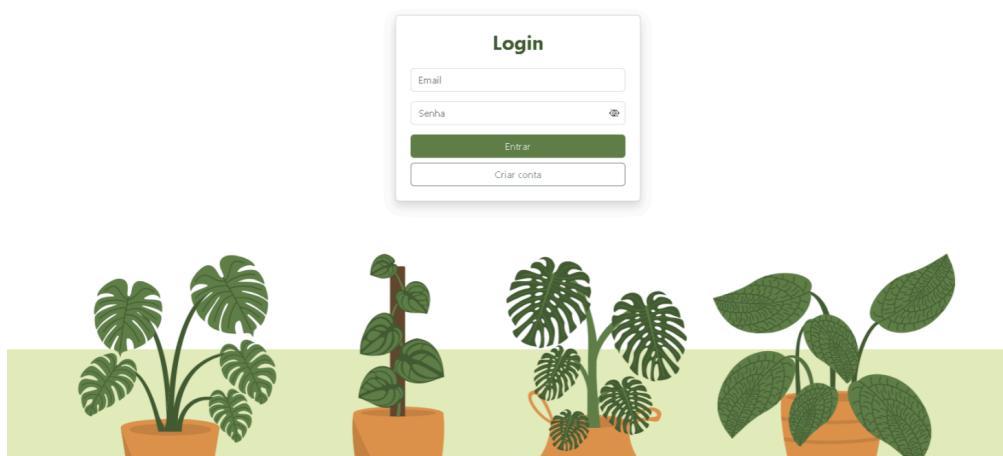
Na aplicação, o ambiente é definido por meio da linha `CI_ENVIRONMENT = development`, o que influencia diretamente comportamentos como a exibição de mensagens de erro e o nível de detalhamento do debug. Além disso, há a variável `app.baseURL`, que facilita os redirecionamentos e a geração de links na aplicação. Também estão presentes as credenciais `database.default.hostname`, `database.default.database`, `database.default.username` e `database.default.password`, responsáveis por armazenar os parâmetros de conexão com o banco de dados, como o host, nome do banco, usuário, senha e porta, garantindo uma configuração centralizada e segura.

Dessa maneira, é possível rodar o projeto em qualquer máquina, bastando ajustar as variáveis do arquivo `.env` de acordo com o ambiente local. Isso garante uma configuração prática, padronizada e reutilizável, facilitando a instalação e execução do sistema por outros desenvolvedores ou interessados no projeto.

2.7.6 Interfaces do Projeto

As interfaces desenvolvidas no projeto foram concluídas com foco na usabilidade, permitindo uma navegação simples e funcional. Foram implementadas telas de login, cadastro de usuário, tipos de plantas, visualização das plantas, entre outras, integrando de forma eficiente a parte visual com a lógica do sistema. É possível visualizar abaixo algumas interfaces e como cada uma foi estruturada para atender aos objetivos do projeto e suas denominações. Por exemplo, a Interface 1 apresenta a tela de login do usuário.

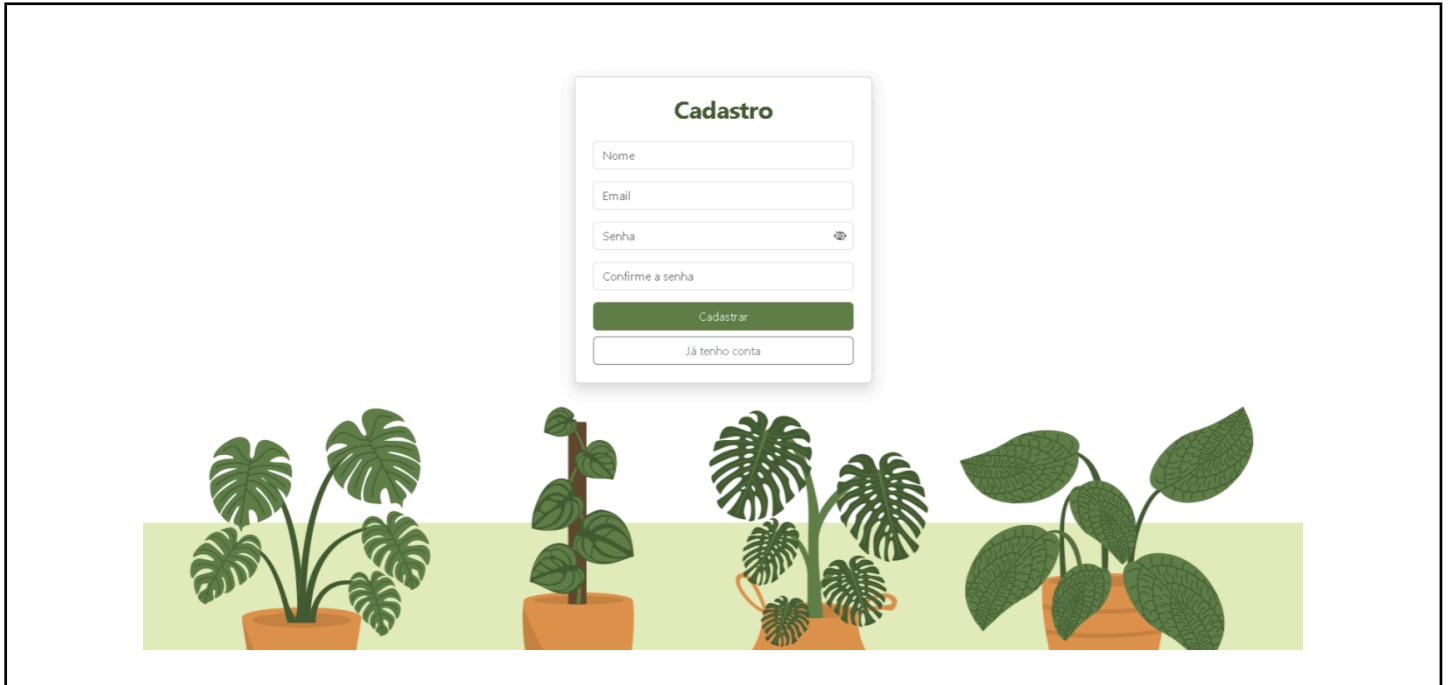
Interface 1 - Tela de Login do Usuário



Fonte: Os autores (2025).

A Interface 2 foi criada para atender o cadastro do usuário.

Interface 2 - Tela de Cadastro do Usuário



Fonte: Os autores (2025).

Na Interface 3, é possível visualizar a tela de listagem de plantas do usuário.

Interface 3 - Listagem de Plantas

The screenshot shows a list of plants under the heading 'Listagem de Plantas'. The first item in the list is 'teste - teste', with options to 'Ver', 'Editar', 'Excluir', and a star icon. Above the list is a navigation bar with links for Home, Plantas (selected), Tipos, and Favoritos. There is also a search bar with 'Pesquisar...' and a 'Buscar' button. The green decorative footer features the same four potted plants as the previous interface.

Fonte: Os autores (2025).

Na Interface 4, o sistema apresenta uma tela de tipos de plantas.

Interface 4 - Tipos de Plantas

Tipos de Plantas

- Arbustos ornamentais**
Plantas lenhosas de pequeno a médio porte, ideais para cercas-vivas e divisórias verdes.
- Árvores ornamentais**
Espécies arbóreas cultivadas por sua beleza, flores, folhas ou porte.
- Bromélias**
Plantas tropicais com folhas dispostas em roseta e flores coloridas.
- Cactos**
Plantas espinhosas e resistentes à seca, muito utilizadas em jardins de pedra e interiores.
- Cercas-vivas**
Plantas utilizadas para formar barreiras naturais, tanto ornamentais quanto funcionais.
- Cicadáceas**
Plantas pré-históricas de aparência semelhante a palmeiras, como a cica.
- Coníferas ornamentais**
Plantas como pinheiros e tuias, usadas em jardins e decorações natalinas.
- Ervas aromáticas**
Plantas que possuem aroma agradável e são usadas em culinária e jardinagem ornamental.
- Gramas ornamentais**
Tipos de gramineas usadas para forração com apelo visual.
- Herbáceas**
Plantas de caule mole e pequeno porte, geralmente usadas em forração.
- Orquídeas**
Plantas floríferas muito valorizadas pela beleza e variedade de suas flores.
- Palmeiras**
Plantas de médio a grande porte, usadas em paisagismo e decoração de ambientes abertos e internos.

Fonte: Os autores (2025).

E para finalizar a amostragem, a Interface 5 dispõe de uma tela de favoritos.

Interface 5 - Tela de Favoritos

Favoritos

teste	Adicionado em: 20/06/2025 19:42	Ver	Remover
-------	---------------------------------	------------	----------------

Fonte: Os autores (2025).

2.7.7 Verificação por Token do JWT

No projeto, foi implementado um sistema de autenticação baseado em JWT (*JSON Web Token*) para garantir segurança. Quando o usuário realiza o login com e-mail e senha válidos, o sistema gera um token contendo informações como o ID do usuário, nome, papel (perfil), horário de emissão e tempo de expiração. Esse token é criado usando uma chave secreta definida no `.env` (`JWT_SECRET`) e assinado com o algoritmo HS256. Após a geração, o token é enviado ao cliente e serve como credencial de acesso às rotas protegidas.

É possível visualizar o comportamento descrito acima na Imagem 9, onde, após a validação das credenciais do usuário, é gerado um token JWT contendo as informações essenciais para autenticação. Esse token é então enviado ao cliente e pode ser utilizado em requisições futuras para acesso a rotas protegidas, permitindo a verificação da identidade do usuário de forma segura e eficiente.

Imagen 9 - Verificação por Token do JWT no projeto

```
crud-plantas > app > Controllers > AuthController.php > PHP Intelephense > AuthController > login
1  <?php
2
3  namespace App\Controllers;
4
5  use App\Models\UsuarioModel;
6  use CodeIgniter\Controller;
7  use Firebase\JWT\JWT;
8
9  class AuthController extends Controller
10 {
11     public function loginForm()
12     {
13         return view('login/index');
14     }
15
16     public function cadastroForm()
17     {
18         return view('register/index');
19     }
20
21     public function login()
22     {
23         $data = $this->request->getPost();
24         if (empty($data)) {
25             $data = $this->request->getJSON(true);
26         }
27
28         $email = $data['email'] ?? null;
29         $senha = $data['senha'] ?? null;
30
31         if (!$email || !$senha) {
32             return $this->response
33                 ->setStatusCode(400)
34                 ->setJSON(['error' => 'Email e senha são obrigatórios']);
35         }
36
37         $usuarioModel = new UsuarioModel();
38         $usuario = $usuarioModel->where('email', $email)->first();
39
40         if (!$usuario || !password_verify($senha, $usuario['senha'])) {
41             return $this->response
42                 ->setStatusCode(401)
43                 ->setJSON(['error' => 'Credenciais inválidas']);
44         }
45
46         $key = getenv('JWT_SECRET') ?: 'SUA_CHAVE_SECRETA';
47         $payload = [
48             'sub' => $usuario['id'],
49             'name' => $usuario['nome'],
50             'email' => $usuario['email'],
51             'papel' => $usuario['papel'],
52             'iat' => time(),
53             'exp' => time() + 3600 // 1 hora
54         ];
55
56         $token = JWT::encode($payload, $key, 'HS256');
```

2.8 REFLEXÃO SOBRE DESAFIOS

Durante o desenvolvimento deste projeto de CRUD em PHP com CodeIgniter, um dos maiores desafios foi o início do processo. No começo, o domínio do PHP ainda era básico, e havia dificuldade para entender o funcionamento do framework, a organização dos arquivos e a configuração correta do XAMPP. Foi necessário um tempo considerável para fazer as *migrations* funcionarem sem erros, devido a problemas com tabelas duplicadas e comandos incorretos. No decorrer do processo, juntamente com seu avanço, o entendimento sobre a estrutura do CodeIgniter e o gerenciamento do banco de dados foi aprimorado, facilitando o progresso do projeto.

Outra etapa que exigiu bastante trabalho foi a configuração do JWT. Por ser uma ferramenta nova, houveram dificuldades para entender a proteção das rotas, a aplicação dos filtros e a validação dos tokens. Foram necessárias várias pesquisas e testes até que tudo funcionasse corretamente. Apesar das dificuldades enfrentadas, esse processo possibilitou um aprendizado significativo, servindo de base não apenas sobre PHP, mas também sobre estrutura de projeto, autenticação e segurança no backend.

3. CONCLUSÃO

Neste estudo, foi desenvolvido um sistema de gerenciamento de Catálogo de Plantas Ornamentais utilizando o framework CodeIgniter, aplicando os princípios da arquitetura MVC para estruturar a aplicação de forma organizada e sistemática. Foram implementadas funcionalidades essenciais como o cadastro de usuários, sistema de login e o envio de imagens vinculadas às plantas, proporcionando uma experiência mais interativa e dinâmica para o usuário. A camada *Model* utilizou conceitos de ORM para facilitar o acesso e manipulação dos dados do banco, enquanto as *Migrations* foram responsáveis por estruturar e versionar o banco de dados de forma segura e escalável. O projeto agregou em si também o arquivo de *enviroment*, fundamental para centralizar configurações sensíveis como credenciais de banco de dados e URL base da aplicação. O sistema também contou com a separação clara entre as camadas de controle e apresentação, garantindo manutenibilidade e clareza no código. Com isso, o projeto não apenas reforçou os conceitos técnicos de desenvolvimento web, como também demonstrou na prática a integração entre segurança, persistência de dados e usabilidade em uma aplicação completa.

REFERÊNCIAS

ANDRADE, Ana Paula de. O que é o Composer?. **TreinaWeb.** [s. I.], 2020. Disponível em: <https://www.treinaweb.com.br/blog/o-que-e-o-composer>. Acesso em: 21 jun. 2025.

ANDRADE, Fernando Francisco de. **Geração de interfaces de usuário para operações CRUD com base em metadados**, 2011. Trabalho de Diplomação (Curso Superior de Tecnologia em Desenvolvimento de Sistemas de Informação) - Universidade Tecnológica Federal do Paraná - UTFPR, Medianeira, 2011. Disponível em: https://riut.utfpr.edu.br/jspui/bitstream/1/13440/2/MD_COADS_2011_1_02.pdf. Acesso em: 22 jun. 2025.

BAUER, Jacob Edinei. **Geração de interfaces de usuário para operações CRUD com base em metadados**. Universidade do Extremo Sul Catarinense - UNESC, Curso de Ciências da Computação, Criciúma, 2018. Disponível em: <http://repositorio.unesc.net/bitstream/1/8143/1/EDINEI%20JACOB%20BAUER.pdf>. Acesso em: 22 jun. 2025.

COSTA, João Pedro Gonçalves; BORGES, João Henrique Gião. **Projeto de sistema web para gerenciamento de trabalhos de conclusão de curso**. Recima21, v. 4, p.4, 2023. Disponível em: <https://recima21.com.br/index.php/recima21/article/view/4594/3229>. Acesso em: 21 jun. 2025.

EDUARDO, Carlos. Migrations: o porque e como usar. **Medium.** [s. I.], 10 fev. 2016. Disponível em: <https://juniorb2s.medium.com/migrations-o-porque-e-como-usar-12d98c6d9269>. Acesso em: 21 jun. 2025.

PANDOLFI, Cláudio Rosse. **Suporte de interesses transversais para framework Codeigniter**. Centro Universitário Eurípides de Marília - UNIVEM, Curso de Ciências da Computação, p. Marília, 2013. Disponível em: <https://aberto.univem.edu.br/bitstream/handle/11077/977/03-Conte%c3%baddo.pdf?sequence=3&isAllowed=y>. Acesso em: 28 jun. 2025.