Plano de Teste
Gerência de Configuração e Mudanças

Projeto	FarmacoCheck
Versão do Produto	1.0

Histórico das alterações

Versão (XX.YY)	Data (DD/MMM)	Autor	Descrição	Aprovado por
00.00	08/12/24	Josimara; Silvio	Pesquisa e planejamento	
01.00	13/12/24	Josimara; Silvio	Plano base de testes para a primeira versão do FarmacoCheck.	
01.01	02/01/24	Daniel	Revisão do documento	

1. Introdução

Este plano de teste tem como principal objetivo servir de guia para avaliação de qualidade geral do sistema FarmacoCheck em todas as etapas de desenvolvimento, assegurando que o *software* atenda aos requisitos funcionais, de segurança e desempenho definidos, além de estar alinhado com as necessidades dos profissionais de saúde. O plano busca identificar e corrigir falhas que possam comprometer conceitos como: usabilidade, integridade dos dados e confiabilidade das informações sobre medicamentos e suas interações, buscando uma experiência consistente para os administradores (responsáveis pelo cadastro e gerenciamento de dados) e usuários finais (médicos e outros profissionais de saúde).

Por meio deste plano, será possível monitorar a evolução dos testes, realizar ajustes baseados no *feedback* contínuo e mitigar os riscos de falhas no ambiente de produção. Além disso, o plano contribui para a melhoria contínua do sistema, fornecendo uma base para priorizar correções e otimizações. O projeto FarmacoCheck, desenvolvido utilizando o *framework Laravel* em *PHP*, será validado quanto à sua capacidade de oferecer uma plataforma eficiente e confiável para consultas rápidas sobre interações medicamentosas, com foco na segurança do paciente e no suporte à tomada de decisões clínicas.

2. Requisitos a Testar

Os principais requisitos a serem avaliados para a primeira versão em produção da aplicação, de acordo com sua prioridade, incluem:

- **Autenticação e registro**: O sistema deve permitir o cadastro, *login* de todos os tipos de usuários descritos no documento de requisitos.
- **Inclusão e edição de Medicamentos e Interações**: O sistema deve permitir que Admins e SuperAdmins manipulem informações sobre medicamentos e interações.
- **Usabilidade da aplicação**: O sistema deve ser acessível por meio de um navegador em diferentes dispositivos.
- **Consultas**: Usuários podem fazer consultas sobre interações medicamentosas inserindo os nomes de medicamentos.
- **Integração com banco de dados:** Validação do armazenamento correto dos medicamentos, interações e usuários no banco de dados.

3. Tipos de teste

Buscando garantir a qualidade, os seguintes tipos de testes serão realizados em cada iteração do projeto:

- **Testes de Unidade:** Focar na validação de cada função e método de forma isolada.
- **Testes de Integração**: Garantir que os módulos funcionem corretamente quando integrados, especialmente cadastros, criação de interações medicamentosas e administração de usuários.
- **Testes de Sistema:** Validar o comportamento do sistema completo, garantindo que ele funcione conforme as expectativas dos usuários finais.
- **Testes de Regressão**: Garantir que novas funcionalidades não introduzem falhas em partes já testadas.
- **Testes de carga/estresse:** Avaliar a robustez e a capacidade do sistema sob condições extremas de carga, identificando os limites de desempenho e garantindo a estabilidade.

3.1. Métodos e Classes

No contexto do *Laravel*, que segue a arquitetura MVC, é essencial que as funcionalidades de cada camada do sistema sejam validadas de forma isolada e integrada. Para isso, os métodos das classes devem ser testados individualmente e em conjunto, verificando se os resultados obtidos estão de acordo com os requisitos esperados. O uso de testes automatizados é fortemente recomendado, com integração em *pipelines* de CI/CD para tornar mais robusto o processo de validação.

- **Verificação de retorno esperado:** Os métodos de cada camada devem ser testados para garantir que retornem os resultados corretos de acordo com as entradas fornecidas. Isso inclui testes com valores normais, valores limite e entradas inválidas, assegurando a robustez do sistema.
- **Testes de exceção**: Verificar se os métodos estão lidando corretamente com exceções, como entradas inválidas ou erros que possam surgir em operações de banco de dados, falhas de validação ou interações com serviços externos.
- **Testes de integração:** No *Laravel*, os testes de integração devem validar a interação entre os componentes do sistema, como *controllers*, *models*, e *services*.
- **Automatização com ferramentas:** Utilizar ferramentas nativas do *Laravel*, como *Pest* ou *PHPUnit*, para criar testes unitários, de integração e de funcionalidade.
- **Testes específicos do Laravel:**

- Testes de rotas: Verificar se as rotas estão mapeadas corretamente e retornam as respostas esperadas.
- Testes de *middleware*: Validar se os *middlewares* aplicados estão funcionando corretamente, como autenticação e autorização.
- Testes de banco de dados: Testar operações realizadas pelos *Eloquent Models*, como inserções, atualizações, exclusões e consultas, utilizando bancos de dados em memória ou *mocks*.

3.2. Persistência de dados

O teste de persistência de dados é utilizado para validar a integridade e consistência das informações armazenadas no banco de dados. Para isso, é necessário considerar os seguintes aspectos:

- **Teste de integridade de dados**: Verificar se as operações de inserção, atualização e exclusão realizadas pelos modelos *Eloquent* gravam e mantém os dados corretamente no banco de dados, sem inconsistências ou perdas.
- **Recuperação após falhas**: Simular falhas inesperadas, como interrupções durante a gravação de dados, para garantir que o banco de dados permaneça em um estado consistente. Verificar se os mecanismos de *rollback* de transações são aplicados corretamente quando necessário.
- **Transações**: Testar a conformidade com as propriedades ACID ao utilizar transações no *Laravel*, assegurando que falhas em uma operação revertam todas as alterações realizadas durante a transação.
- **Automatização dos testes:** Utilizar ferramentas como *PHPUnit* ou *Pest* para automatizar os testes relacionados à persistência de dados. Exemplos de cenários que podem ser testados incluem:

3.3. Integração dos Componentes

Os testes de integração verificam se as interações entre os componentes do sistema — como controladores, modelos, e rotas — funcionam de maneira correta. Esses testes são importantes para validar o fluxo completo de execução e validar que as partes do sistema se comportem como esperado, considerando os seguintes pontos:

- **Fluxo de ações no sistema:** Testar sequências completas de ações, como chamadas de rotas que passam por controladores, serviços e persistência no banco de dados.

Validar se os fluxos executam corretamente desde a interação com o frontend até a camada de persistência.

- **Interdependência de componentes:** Avaliar se os controladores, modelos, e serviços estão interagindo corretamente. Por exemplo, um controlador deve chamar os métodos de serviço adequados e receber os dados esperados para exibição.
- **Verificação de dados entre camadas**: Garantir que as informações fluam corretamente entre as camadas do padrão MVC. Verificar se os dados enviados pelo usuário chegam ao controlador, são processados pelos serviços e modelos, e retornados corretamente para exibição.
- **Simulação de cenários reais:** Criar cenários representando o uso comum da aplicação, como cadastro de usuários, autenticação, consultas e operações *CRUD*. Testar fluxos completos para validar a integração entre as diferentes partes do sistema.
- **Automatização dos testes:** Utilizar ferramentas como PHPUnit ou *Pest* para criar testes automatizados que validem a integração entre rotas, controladores e modelos. Esses frameworks de teste permitem simular requisições *HTTP* e verificar o comportamento da aplicação em cenários reais, garantindo que a interação entre os diferentes componentes do sistema ocorra de maneira correta e a aplicação funcione conforme o esperado.

3.4. Testes funcionais

Os testes funcionais validam se o sistema atende aos requisitos especificados, garantindo que as funcionalidades operem conforme o esperado.

- **Validação de requisitos**: Verificar operações como cadastro, login e manipulação de dados, isoladamente e em fluxos combinados.
- **Fluxos completos e casos limite**: Testar cenários reais e entradas inválidas, garantindo validações eficazes e mensagens claras.
- **Interação com o usuário:** Validar que botões, links e formulários executam ações corretas e a navegação é consistente.
- **Automatização**: Utilizar ferramentas como *Selenium* (buscar também por Laravel Dusk) para simular interações e agilizar verificações.
- **Regressão funcional:** executar testes novamente após mudanças para garantir que funcionalidades existentes não sejam afetadas.
- **Casos críticos:** Priorizar testes em áreas essenciais, como autenticação e controle de acesso.

3.5. Testes de Carga/Estresse

Os testes de carga e estresse avaliam o comportamento do sistema sob condições extremas, identificando limites, gargalos e falhas.

- **Carga extrema:** Simular usuários simultâneos acima do esperado para avaliar limites e pontos de falha.
- **Volume de dados:** Testar processamento e armazenamento com grandes volumes de dados, como uploads ou consultas complexas.
- **Resiliência a falhas:** Verificar a recuperação do sistema diante de falhas em componentes críticos, como banco de dados ou rede.
- **Desempenho:** Monitorar tempo de resposta sob alta carga, garantindo que a experiência do usuário não seja severamente impactada.
- **Picos de tráfego:** Simular picos repentinos para avaliar a adaptabilidade do sistema.
- **Métricas:** Analisar logs, uso de *CPU*, memória e rede para identificar gargalos.
- **Automatização:** Utilizar ferramentas como *JMeter* ou K6 para simular cenários de carga e gerar relatórios de desempenho.
- **Ambientes realistas:** Realizar testes em ambientes similares ao de produção para maior confiabilidade nos resultados.

4. Recursos

Para garantir a eficiência dos testes da aplicação FarmacoCheck, serão utilizadas ferramentas de automação e análise de qualidade, como *Pest* ou *PHPUnit*, e *SonarQube*, complementares em ambientes de desenvolvimento ágil.

- *Pest/PHPUnit*

  Ferramenta para testes automatizados em *PHP*, útil para testes unitários, de integração e funcionais.

  - Simples configuração e fácil aprendizado.
  - Suporte a fixtures para preparar ambientes de teste reutilizáveis.
  - Permite parametrização de testes e execução seletiva com filtros.
  - Integração com ferramentas de *mocking* e *frameworks* de comportamento (BDD).
  - Ecossistema de plugins para relatórios de cobertura e integração com CI/CD.

- *SonarQube*

Plataforma para análise estática de código, garantindo qualidade e segurança no desenvolvimento.

- Identifica *bugs*, vulnerabilidades e violações de padrões.
- Gera relatórios de cobertura de código e métricas de qualidade (complexidade, duplicação).
- Integração com *pipelines* CI/CD, como *GitActions* e *Jenkins*, para análises contínuas.
- Regras personalizáveis para adaptação às necessidades do projeto.
- Interface *Web* para monitorar métricas e acompanhar melhorias.

5. Cronograma

	Duração	**Data de início**	**Data de término**
Planejar testes	1 semana	8/12/24	16/12/24
Projetar testes	1 semana	16/12/24	23/12/24
Implementar testes	Contínuo	26/12/24	13/01/25
Executar testes	Contínuo	26/12/24	13/01/25
Avaliar testes	Contínuo	16/12/24	13/01/25

6. Referências