INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO PIAUÍ
CAMPUS TERESINA CENTRAL
TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

GIOVANNA LIMA SANTANA

TESTES NO *FRAMEWORK LARAVEL*: estudo de caso com métricas de qualidade e confiabilidade

TERESINA, PIAUÍ
2026

GIOVANNA LIMA SANTANA

TESTES NO *FRAMEWORK LARAVEL*: estudo de caso com métricas de qualidade e confiabilidade

Trabalho de Conclusão de Curso (monografia) apresentado como exigência parcial para obtenção do diploma do Curso de Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Piauí, Campus Teresina Central.

Orientador: Prof. M^{e}. Ely da Silva Miranda

TERESINA, PIAUÍ
2026

Dados Internacionais de Catalogação na Publicação (CIP) de acordo com ISBD

Santana, Giovanna Lima

S232t Teste no framework : estudo de caso com métricas de qualidade e confiabilidade / Giovanna Lima Santana. - 2026.

86 f.: il. color.

Trabalho de Conclusão de Curso (Tecnologia em Análise e Desenvolvimento de Sistemas) - Instituto Federal de Educação, Ciência e Tecnologia do Piauí, Campus Teresina Central, 2026.

Orientador : Prof Me. Ely da Silva Miranda .

1. Laravel. 2. Testes automatizados. 3. Métricas de software . 4. Testes de qualidade. I. Título.

CDD - 004.21

Elaborado por Maria Rosismar Farias CRB CRB 3/631

GIOVANNA LIMA SANTANA

TESTES NO *FRAMEWORK LARAVEL*: estudo de caso com métricas de qualidade e confiabilidade

Trabalho de Conclusão de Curso (monografia) apresentado como exigência parcial para obtenção do diploma do Curso de Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Piauí, Campus Teresina Central.

Aprovada em 23/06/2026.

BANCA EXAMINADORA:

Prof. M^{e}. Ely da Silva Miranda (Orientador)
Instituto Federal do Piauí (IFPI)

Prof. M^{e}. Fernando Castelo Branco Gonçalves Santana
Instituto Federal do Piauí (IFPI)

Prof. Dr. Otílio Paulo da Silva Neto
Instituto Federal do Piauí (IFPI)

TERESINA, PIAUÍ
2026

Dedico este trabalho a Giovanna de anos atrás que jamais imaginaria que teria chegado tão longe, olha só onde a gente está agora.

AGRADECIMENTOS

Agradeço a Deus, à minha família, aos amigos e a todos que estiveram ao meu lado durante essa jornada, tornando-a mais leve e possível, especialmente nos momentos de incerteza e desânimo. Um agradecimento especial aos meus amigos Alita e Lucas, e aos colegas de curso que se tornaram grandes amigos — Amanda, João Gabriel e Itallo — por toda a companhia, pelos conselhos, pelas conversas e pelo apoio constante. Vocês tornaram esse período muito mais acolhedor e significativo.

Sou profundamente grata ao meu orientador, Ely, pela paciência, disponibilidade e orientação ao longo do curso e, em especial, neste trabalho.

A experiência de realizar este curso foi marcante não apenas pelo aprendizado teórico e pela formação profissional, mas também pelo crescimento pessoal que ele proporcionou. Os professores, com seus ensinamentos e conselhos, e os servidores da instituição, sempre solícitos, contribuíram de maneira essencial para esse processo, assim como cada colega que compartilhou comigo os desafios e conquistas ao longo do caminho. Cada palavra de incentivo, cada troca de conhecimento e cada gesto de apoio foram fundamentais para a concretização deste TCC.

Se deve viver apesar de.
Clarice Lispector

RESUMO

Este trabalho tem como objetivo analisar a qualidade dos testes automatizados no *framework* Laravel, enfrentando a dificuldade recorrente de mensurar a efetividade e eficiência desses testes em projetos PHP. Para isso, foi utilizado um sistema *open-source* de página de status chamado *Cachet*, servindo como estudo de caso para a coleta e análise de métricas como tempo médio de execução, taxa de sucesso e falha, uso de recursos e *throughput*. Os resultados incluem a avaliação da eficácia dos testes implementados, a proposição de uma abordagem sistematizada para mensuração da qualidade em projetos utilizando o *framework* Laravel. Visando maior confiabilidade, manutenibilidade e escalabilidade desses projetos, como também a possibilidade de contribuição para um sistema real e relevante no mercado.

Palavras-chaves: Laravel; testes automatizados; métricas de software; teste de qualidade; métricas de qualidade; PHP.

ABSTRACT

This study aims to analyze the quality of automated tests within the Laravel framework, addressing the recurring difficulty of measuring the effectiveness and efficiency of these tests in PHP projects. To achieve this, an open-source status page system called Cachet was used as a case study to collect and analyze metrics such as average execution time, success and failure rates, resource utilization, and throughput. The results include an evaluation of the efficacy of the implemented tests and the proposal of a systematized approach for quality measurement in projects utilizing the Laravel framework. This approach aims to ensure greater reliability, maintainability, and scalability in such projects, while also contributing to a real and market-relevant system.

Keywords: Laravel; Automated Testing; software metrics; test quality; quality metrics; PHP.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

ABNT	Associação Brasileira de Normas Técnicas
API	Application Programming Interface
BDD	Behavior Driven Development
CPU	Central Processing Unit
HTTP	Hypertext Transfer Protocol
IFPI	Instituto Federal de Educação, Ciência e Tecnologia do Piauí
ISO	International Organization for Standardization
IEC	International Electrotechnical Commission
JSON	JavaScript Object Notation
MVC	Model-View-Controller
NBR	Norma Brasileira
RAM	Random Access Memory
SQL	Structured Query Language
SQuaRE	System and Software Quality Requirements and Evaluation
SWEBOK	Software Engineering Body of Knowledge
TDD	Test-Driven Development

LISTA DE SÍMBOLOS

%	Porcentagem

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento de *software* tem se tornado cada vez mais desafiador diante da crescente demanda por aplicações *web* robustas, confiáveis e entregues em prazos cada vez mais curtos. *Frameworks* são ferramentas que buscam otimizar esse processo, já que fornecem uma arquitetura, bibliotecas, classes, componentes e estruturas de auxílio prontas. Utilizar essas ferramentas em um projeto de *software* promove padronização e aumento da produtividade (PRESSMAN, 2016).

Nessa tangente, a adoção do uso de *frameworks* tem se tornado comum no processo de desenvolvimento de sistemas. Dentre os diversos *frameworks* disponíveis pra desenvolvimento *web*, em especial para a linguagem PHP, o *Laravel* se destaca por sua arquitetura moderna, comunidade ativa e ampla adesão. Além disso, sua modelagem completa com diversas ferramentas integradas, faz dele uma escolha frequente para quem quer rapidez e qualidade de projeto.

Concomitantemente, a preocupação com a garantia de qualidade de *software* cresce, especialmente em sistemas críticos onde a confiabilidade é um ponto crucial (SOMMERVILLE, 2011). Sob esse prisma, urge a necessidade de aplicar conceitos de engenharia de *software*, uma vez que a qualidade é o seu princípio mais almejado. Nesse contexto, os testes de *software* se tornam grandes aliados na confirmação da qualidade de um sistema.

Ferramentas como *PHPUnit* e *Laravel Dusk*, nativamente integradas ao *framework Laravel*, oferecem meios eficazes de implementar testes ao longo do ciclo de vida do *software* (LARAVEL, 2024). A adoção de testes automatizados traz diversos benefícios para o desenvolvimento de *software*, como a redução do retrabalho e maior segurança em alterações futuras (PRESSMAN, 2016). O *Laravel*, por oferecer ferramentas que apoiam essas práticas modernas, se encaixa bem nesse novo cenário.

Entretanto, o real benefício dessas ferramentas só acontece quando elas são aplicadas de forma adequada dentro do contexto do desenvolvimento do projeto. Por isso, além de conhecer os recursos que o *framework* oferece, é importante entender como usá-los na prática (PRESSMAN, 2016). Compreender a aplicabilidade das ferramentas de teste no *Laravel* permite não apenas melhorar a qualidade do *software*, mas também otimizar o processo de desenvolvimento e aumentar a confiança nas entregas.

Ao aplicar os testes de maneira consistente, é possível detectar falhas mais cedo no ciclo de vida do sistema, o que contribui para produtos mais estáveis e sustentáveis. Apesar das vantagens oferecidas pelo Laravel em relação aos testes automatizados, observa-se que muitos desenvolvedores não exploram de forma plena os recursos disponíveis, especialmente por falta de conhecimento (LARAVEL..., 2023; MCCONNELL, 2004). Assim, este trabalho teve como propósito analisar, na prática,

como os testes podem ser aplicados de forma eficaz no *Laravel*, contribuindo com a comunidade e incentivando a difusão de boas práticas no uso do *framework*.

Para atingir este objetivo,foi utilizado como estudo de caso o sistema *open source Cachet*. A partir da investigação das ferramentas nativas e complementares disponíveis no ecossistema Laravel, busca-se identificar vantagens, limitações e possibilidades de aplicação prática em um sistema real. Para isso, foram analisados os testes já implementados no *Cachet*, bem como propostas de novos casos de teste com base em funcionalidades críticas da aplicação.

1.1 JUSTIFICATIVA

Em projetos de *software*, tem-se observado uma crescente popularização do uso de *frameworks* com o objetivo de otimizar o processo de desenvolvimento (SOMMERVILLE, 2011). Como consequência, a busca por construir *software* de qualidade tornou-se ainda mais necessária, impulsionando a consolidação de práticas que possibilitam identificar, rastrear e corrigir defeitos com maior agilidade e menor custo (ISO/IEC JTC 1/SC 7, 2014). Nesse cenário, os testes de *software* assumem um papel fundamental na verificação e validação do comportamento esperado das aplicações, contribuindo diretamente para o aumento da confiabilidade, manutenibilidade e robustez dos sistemas (PRESSMAN, 2016).

Dessa forma, torna-se relevante um estudo aprofundado sobre métodos de teste no contexto de *frameworks* modernos, a fim de demonstrar, de forma objetiva, como os testes automatizados podem contribuir para a melhoria da qualidade e confiabilidade das aplicações. Considerando a ampla adoção do *Laravel* no desenvolvimento *web* da atualidade, sua escolha como base para este estudo configura-se como uma oportunidade estratégica para consolidar boas práticas de teste, analisando as ferramentas nativas da plataforma e avaliando sua eficácia em um cenário voltado à qualidade de *software*. Com o crescimento da demanda por entregas rápidas e com alto nível de qualidade, o domínio de práticas de testes tornou-se uma competência estratégica no mercado.

Profissionais que as aplicam tendem a desenvolver sistemas mais seguros, confiáveis e sustentáveis, reduzindo falhas em produção, custos com correções e manutenção, além de promover maior confiança por parte dos clientes. Ademais, o uso de testes automatizados contribui para a manutenção de uma documentação viva e proporciona maior segurança durante evoluções e refatorações do código. Diante disso, discutir e incentivar o uso adequado dessas práticas representa uma contribuição significativa para a melhoria contínua da qualidade no desenvolvimento de *software* e para a formação de profissionais mais preparados para os desafios do setor.

1.2 OBJETIVOS

1.2.1 Objetivo geral

Realizar um estudo sobre métodos de testes incluídos nativamente no *framework Laravel*, por meio da criação e aplicação de testes em um sistema relevante feito com essa ferramenta, afim de analisar por meio de métricas estabelecidas a eficiência dos testes na garantia de qualidade de *software*.

1.2.2 Objetivos específicos

- Mapear vantagens e desvantagens das ferramentas de teste automatizado nativas e complementares disponíveis no *Laravel*, com foco na sua aplicabilidade prática;
- Colaborar com a evolução contínua do *framework* Laravel ao identificar lacunas ou limitações nas ferramentas de teste oferecidas pelo *framework* Laravel em um sistema real;
- Contribuir com a análise e aprimoramento de um sistema real e amplamente utilizado no ecossistema PHP.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo será introduzida a base conceitual necessária para compreensão dos temas principais deste trabalho. A primeira seção trata do conceito de "Engenharia de *Software*", onde é abordada a caracterização da área, definição de *software* e detalhamento sobre 'Ciclo de Vida do *Software*'. Na sequência, a segunda seção discute sobre "Qualidade de *Software*", definindo o conceito de qualidade de *software* e apresentando modelos de qualidade.

Em seguida, a terceira seção, dedicada a "Testes de *Software*", explica a necessidade e detalha as seis principais classificações clássicas de testes: Teste Unitário, Testes de Integração, Teste de Sistema, Teste de Regressão, Teste de Performance e Teste Automatizado. Esta seção também divide os testes a partir de sua estratégia: Desenvolvimento Orientado a Testes (TDD) e Desenvolvimento Orientado ao Comportamento (BDD), ademais falará sobre métricas de qualidade. Seguidamente, a quarta seção ficará responsável por apresentar o *framework* de estudo Laravel, comentando brevemente sobre sua arquitetura e ferramentas de teste disponíveis. Dando continuidade, a quinta seção irá discorrer sobre o sistema que servirá como base para estudo de caso e aplicação de testes de *software*, o *Cachet*, um sistema de página de status desenvolvido com *Laravel* e que vem sendo amplamente utilizado por empresas.

Ademais, a sexta seção apresenta uma análise comparativa com trabalhos correlatos, com o objetivo de posicionar este estudo em relação ao que já foi desenvolvido na área. Essa comparação é importante para evidenciar como este trabalho se diferencia ou se alinha com outras abordagens que também exploram testes em projetos Laravel ou em contextos similares. Essa etapa também ajuda a demonstrar que a proposta deste trabalho não está isolada, mas dialoga com iniciativas anteriores, complementando e fortalecendo a discussão sobre qualidade e testes no uso de *frameworks* modernos. Por fim, a sétima e última seção traz as ferramentas e tecnologias a serem utilizadas na execução deste trabalho.

2.1 ENGENHARIA DE SOFTWARE

O *software* pode ser entendido como um produto completo que abrange código, documentação e suporte, um conjunto de elementos que juntos atendem a necessidade do cliente (PRESSMAN, 2016). A aplicação de métodos e princípios bem definidos para modelar, desenvolver, testar e manter *softwares* caracteriza o que hoje se conhece como Engenharia de Software. Essa definição foi formalizada pela primeira vez em 1968, durante a conferência promovida pelo Comitê Consultivo para Ciência e Tecnologia (COTAN), da OTAN, com o objetivo de enfrentar a chamada “crise do

software".

Segundo Mosharraf (2018), um dos conceitos mais essenciais da Engenharia de *Software* é o Ciclo de Vida do Software, que compreende quatro fases principais: análise, planejamento do projeto, implementação e teste. Cada uma dessas fases é fundamental para garantir a qualidade, a funcionalidade e a manutenção eficiente do sistema ao longo do tempo. Essas etapas estruturam o processo de desenvolvimento de *software*, a seguir serão detalhadas as principais atividades envolvidas em cada uma delas.

O processo de análise começa pela compreensão do problema, das necessidades do cliente e dos objetivos que ele deseja alcançar. A partir desse entendimento, a equipe inicia o levantamento de requisitos, definindo o que o sistema deve fazer, sem especificar como será feito — os chamados requisitos funcionais (MOSHARRAF, 2018). Na mesma linha, a análise também contempla os requisitos não funcionais, como desempenho, segurança, escalabilidade e manutenabilidade do sistema (SWEBOK, 2024).

Essa etapa é extremamente importante, pois as decisões tomadas nesse momento influenciam diretamente todas as fases seguintes. A falha na definição dos requisitos é uma das principais causas de fracasso no desenvolvimento de projetos de *software* (SOMMERVILLE, 2011). Por esse motivo, a equipe realiza um estudo aprofundado da problemática, utilizando ferramentas como entrevistas com *stakeholders*, modelagens UML e questionários.

Após a análise, inicia-se o planejamento do projeto, que organiza e estrutura o desenvolvimento do *software*. Dessa forma, nessa etapa, define-se a arquitetura, o design do *software*, as tecnologias a serem utilizadas, além de prazos, custos, riscos, cronogramas e o modelo de desenvolvimento adotado (SOMMERVILLE, 2019). Segundo Pressman (2016), um planejamento eficaz é essencial para garantir a viabilidade do projeto, permitindo antecipar e mitigar problemas antes que eles ocorram. Além disso, é comum a elaboração do plano de testes, que servirá de base para a fase de verificação.

Em seguida, o projeto entra na fase de implementação, em que o sistema é efetivamente construído. É nesse momento que os requisitos e o planejamento se transformam em código-fonte, utilizando linguagens de programação, *frameworks* e ferramentas de apoio, de forma a seguir fielmente as definições estabelecidas nas etapas anteriores (SWEBOK, 2014). Por fim, com o sistema implementado, inicia-se a fase de testes. O objetivo principal dessa etapa é garantir que o produto final seja entregue livre de erros. De acordo com Beizer (1995), testes eficazes não apenas identificam falhas, mas também comprovam que o *software* atende aos requisitos estabelecidos.

Além disso, é importante destacar que os testes podem ser realizados durante o próprio processo de desenvolvimento, conforme estiver definido no plano de testes. A

adoção de uma abordagem orientada a testes contribui significativamente para detectar e corrigir problemas de forma antecipada, aumentando a confiabilidade do sistema (SWEBOK, 2024).

Esse ciclo tem como objetivo garantir um desenvolvimento viável, confiável e que resulte em um produto de *software* com qualidade. Nesse contexto, torna-se essencial compreender o que se entende por qualidade de *software*, bem como os atributos e práticas que permitem alcançá-la ao longo do processo de desenvolvimento.

2.2 QUALIDADE DE SOFTWARE

Devido à grande ascensão da tecnologia e ao fato de a qualidade sempre ter sido um fator essencial para o sucesso de um projeto, torna-se necessário compreender o que é qualidade e como ela se aplica ao desenvolvimento de *software*. Conforme definido pela Associação Brasileira de Normas Técnicas (2015a), na norma técnica NBR ISO 9000, a qualidade é o grau que um sistema atende os requisitos estabelecidos, e não apenas isso como também agrega valor ao cliente e às partes interessadas. Segundo a norma ISO/IEC (2011) 25010, que define o conjunto de características da qualidade de *software*, classifica os atributos de qualidade em:

- Funcionalidade: Refere-se à capacidade do *software* de realizar as funções para as quais foi projetado, atendendo aos requisitos definidos;
- Desempenho: Mede o nível de eficiência com que o sistema executa suas funções, levando em consideração parâmetros como uso de memória, tempo de resposta e rendimento;
- Compatibilidade: Envolve a interoperabilidade, ou seja, a capacidade do sistema de trocar e utilizar informações de outros produtos, e a coexistência, que se refere à habilidade de um *software* operar em conjunto com outros recursos e produtos no mesmo ambiente;
- Usabilidade: Relaciona-se à facilidade de uso do sistema, garantindo a adequação da interface, acessibilidade, proteção contra erros e uma experiência intuitiva e amigável para o usuário;
- Confiabilidade: Refere-se à capacidade do sistema de operar de maneira estável e sem falhas, em condições específicas, assegurando alta disponibilidade, recuperabilidade e tolerância a falhas internas ou externas;
- Segurança: Representa a capacidade do sistema de proteger dados e informações contra vulnerabilidades, garantindo a confidencialidade, autenticidade e integridade das informações;

- Manutenabilidade: Refere-se à capacidade do sistema de ser não apenas corrigido, mas também melhorado e adaptado a mudanças, sejam elas externas ou internas, como novos requisitos ou necessidades de atualização;

- Portabilidade: Refere-se à capacidade do *software* de ser transferido para diferentes ambientes, garantindo sua adaptabilidade, facilidade de instalação e a possibilidade de ser substituído por versões ou plataformas diferentes.

A falta de qualidade em um *software* pode resultar em altos custos de produção, falhas no sistema e insatisfação por parte dos usuários (Consortium for Information & Software Quality (CISQ), 2020). Esse custo foi quantificado em uma pesquisa que revelou que falhas em sistemas de baixa qualidade resultaram em uma perda estimada de 1,8 trilhão de dólares (GORDIEIEV *et al.*, 2023).

2.2.1 Modelos de qualidade de software

Dada a crescente relevância da qualidade de *software*, diversos modelos e padrões têm sido propostos ao longo dos anos para garantir que essa qualidade seja alcançada de forma eficaz. Para uma melhor compreensão do tema, é fundamental definir os conceitos de "modelo"e "padrão".

Segundo a Associação Brasileira de Normas Técnicas (2015b), o conceito de padrão é um documento elaborado e aprovado por consenso, que estabelece regras ou diretrizes que têm como objetivo estabelecer requisitos de qualidade, segurança e padronizar terminologias e características. Por outro lado, um modelo pode ser definido como uma representação estruturada de um produto, utilizada como base para análise, validação e tomada de decisões (ULRICH; EPPINGER, 2016).

Ao abordar os padrões de qualidade de *software*, é importante destacar que existem diversas normas relevantes. Entre elas, está a NBR ISO 9000 (Associação Brasileira de Normas Técnicas, 2015a), já mencionada anteriormente, que tem como principal objetivo padronizar a terminologia relacionada aos sistemas de gestão da qualidade. Além disso, essa norma estabelece sete princípios fundamentais para uma gestão eficaz da qualidade:

- Foco no cliente: A organização deve atender aos requisitos dos clientes e, sempre que possível, superar suas expectativas. Esse princípio tem como objetivo aumentar a confiança e o valor percebido pelo cliente em relação aos serviços oferecidos;

- Liderança: Os líderes devem promover um ambiente de direção e propósito bem definidos, alinhando estratégias e processos para aumentar a eficiência da equipe e garantir o alcance dos objetivos estabelecidos;

- Engajamento de pessoas: O respeito, reconhecimento, empoderamento e o desenvolvimento contínuo da equipe resultam em maior comprometimento das pessoas, o que contribui diretamente para a qualidade do produto final e da organização como um todo;
- Abordagem de processo: Consiste em alinhar os processos internos de forma integrada, promovendo uma compreensão clara de como eles se inter-relacionam. Esse alinhamento deve ocorrer tanto dentro da equipe quanto na interação com o cliente, fortalecendo a confiança mútua;
- Melhoria: A busca constante por melhorias é essencial para manter um bom desempenho, adaptar-se a mudanças e fomentar a inovação;
- Tomada de decisão com base em evidência: As decisões devem ser fundamentadas em dados concretos. A análise, compreensão e avaliação cuidadosa das informações permitem decisões mais seguras e eficazes;
- Gestão de Relacionamento: Estabelecer relações sólidas com partes interessadas relevantes, como parceiros, distribuidores e fornecedores, pode otimizar o desenvolvimento dos projetos e melhorar o desempenho da organização como um todo.

Ademais, foi desenvolvida uma série de normas que estabelece um modelo para a avaliação da qualidade de *software*: a ISO/IEC 25000 - *System and Software Quality Requirements and Evaluation (SQuaRE)* (ISO/IEC JTC 1/SC 7, 2014). Essas normas fornecem uma base padronizada para definir, medir e validar requisitos de qualidade em produtos e processos de software ao longo de seu ciclo de vida. Esse conjunto é composto por diversas normas complementares, entre as quais encontra-se a ISO 25010 - Modelos de Qualidade de Sistema e *Software*, que já foi abordada anteriormente.

Além da ISO/IEC 25010, é relevante destacar brevemente outras normas relevantes que compõe o conjunto *SQuaRE*. A primeira menção é sobre a ISO/IEC 25030 - que especifica os requisitos de qualidade, em conformidade com os princípios abordados na NBR ISO 9000, também já citada anteriormente. Já a ISO/IEC 25040 define um modelo de referência para avaliação de qualidade enquanto a ISO/IEC 25041 trata das diretrizes para essa avaliação, voltadas para desenvolvedores e avaliadores (ISO/IEC JTC 1/SC 7, 2014).

A avaliação da qualidade de um *software* exige o uso de métricas específicas que permitam mensurar atributos conforme mencionados pela NBR ISO 9000. No entanto, para que essas métricas forneçam dados confiáveis, é indispensável a realização de testes sistemáticos que identifiquem defeitos, verifiquem funcionalidades e assegurem que os requisitos de qualidade estejam sendo atendidos. Dessa forma,

os testes de *software* tornam-se uma etapa essencial no processo de validação da qualidade e serão abordados na próxima seção.

2.3 TESTE DE SOFTWARE

Você deve desejar encontrar erros no seu código, pode parecer estranho, mas é melhor que você os encontre, e não outra pessoa (MCCONNELL, 2004). Segundo Hetzel (1987, p. 4), o "teste é um processo de aquisição de confiança no fato de que um programa ou sistema faz o que se espera dele", essa definição reforça mais uma vez que a qualidade de um produto se baseia na conformidade com os seus requisitos. Antes de adentrar sobre o teste em si, é importante entender a diferença entre "erro", "defeito"e "falha".

Segundo o SWEBOK (2024), o erro refere-se a uma ação humana incorreta que pode resultar em inconsistências no sistema, como, por exemplo, a implementação de uma lógica errônea ou digitação incorreta de um caractere durante o desenvolvimento. Já o defeito é a manifestação desse erro no artefato de *software*, evidenciando quando uma funcionalidade não se comporta conforme o esperado. No entanto, nem todo erro resulta em um defeito observável.

A falha, por sua vez, ocorre quando há um desvio no comportamento do sistema, ou uma resposta inesperada durante a execução. A falha é consequência de um ou mais defeitos, os quais, por tabela, são causados por erros. Entretanto, é importante pontuar mais uma vez que nem todo defeito leva a uma falha vísivel, já que ela só será acionada em cenários específicos. Portanto, é possível concluir que a falha é a forma mais visível da baixa qualidade do *software*, já que ela se torna evidente na interação do usuário com o sistema.

Testar um *software* é uma atividade essencial que contribui com a otimização dos processos de desenvolvimento, ele permite economia de tempo e de recursos ao identificar falhas nas etapas iniciais do ciclo de vida do *software*, o que reduz o custo de correção nas fases seguintes. Além disso, os testes fornecem uma base concreta para avaliação da qualidade do sistema, garantindo se os requisitos estabelecidos estão sendo atendidos (ISO/IEC/IEEE, 2022).

O processo de teste também desempenha um papel importante ao auxiliar na correção das falhas encontradas e ao revelar padrões recorrentes de erros que podem virar defeitos. Essa identificação contínua de problemas permite ajustes no processo de desenvolvimento, além de contribuir com a melhora da eficiência do time, que foca em desenvolver a solução e não dedica muito tempo para corrigir erros (ISO/IEC/IEEE, 2022). O ideal é que os testes sejam realizados de forma contínua ao longo de todo o processo de desenvolvimento, permitindo a identificação e correção de erros o mais cedo possível.

Segundo Pressman (2016), quanto mais cedo um erro é detectado, menor é

o custo para corrigi-lo e menor o impacto sobre as demais etapas do projeto. Além disso, testar antes promove uma reflexão sobre a conformidade do sistema com os requisitos estabelecidos, contribuindo não apenas para validar a implementação, como também para verificar se os próprios requisitos estão corretamente definidos (SWEBOK, 2014). Dessa forma, a antecipação dos testes no ciclo de desenvolvimento não apenas reduz retrabalho, mas também favorece a identificação precoce de inconsistências, contribuindo para a construção de um *software* mais alinhado aos requisitos do usuário final.

Testar é encontrar erros no código. Esse é um dos motivos para, apesar de ser uma etapa essencial no desenvolvimento de software com qualidade, ela seja uma fase que sofre muita resistência por parte dos programadores. A pressuposição de que existem erros é algo que não agrada o desenvolvedor (MCCONNELL, 2004). Ademais, o tempo e esforço para criar e manter testes, especialmente em projetos com prazos apertados, muitas vezes são vistos como um carga extra em vez de um investimento na qualidade do produto.

Além disso, testes não garantem um aumento na qualidade do *software*. O que realmente melhora são as técnicas de desenvolvimento aplicações e refatorações pontuais após observações dos testes. Ademais, testes não são capazes de comprovar a ausência completa de erros no sistema. Eles servem, sobretudo, como ferramentas de apoio para detectar falhas, reduzir riscos e aumentar a confiabilidade do software ao longo de seu desenvolvimento e manutenção.

Apesar dessas limitações, os testes continuam sendo uma ferramenta indispensável no desenvolvimento de *software*. Eles permitem avaliar o comportamento do sistema diante de diferentes cenários, validar se os requisitos estão sendo atendidos e identificar falhas que poderiam comprometer a experiência do usuário ou a confiabilidade e segurança da aplicação. Ainda que não assegurem a ausência total de erros, os testes aumentam a confiança no produto final e fornecem dados objetivos para a tomada de decisões técnicas, como a necessidade de correções, refatorações ou reavaliação de requisitos.

Como reforçado pelo SWEBOK (2014), os testes são parte fundamental das práticas de verificação e validação, promovendo a qualidade ao longo de todo o ciclo de vida do *software*. Os testes podem ser classificados em quatro tipos: quanto ao objetivo que se busca, a técnica utilizada, quanto a intenção e quanto ao escopo dos testes (PRESSMAN, 2016). Também é importante ressaltar que esses testes podem ser integrados para uma melhor avaliação da qualidade do *software*.

2.3.1 Classificação por objetivo

Essa classificação se refere ao que será testado no sistema.

- Teste Funcional: O teste funcional tem como objetivo verificar se o sistema executa corretamente as funcionalidades especificadas nos requisitos. Nessa abordagem, o foco está no comportamento do *software* frente às entradas fornecidas, avaliando se as saídas produzidas estão de acordo com o esperado;
- Teste Não Funcional: O teste não funcional avalia aspectos de qualidade do sistema que não estão diretamente relacionados às suas funcionalidades, mas sim ao seu desempenho sob determinadas condições. Entre os atributos verificados estão: desempenho, escalabilidade, usabilidade, segurança, confiabilidade, entre outros.

2.3.2 Classificação por técnica

Essa classificação se refere a como os testes serão realizados, especialmente ao nível de conhecimento do testador sobre a estrutura do código.

- Caixa Preta: O teste de caixa preta (*black-box testing*) é uma técnica em que o testador não possui acesso ao código-fonte nem conhecimento sobre a estrutura interna do sistema. O foco está em verificar se o sistema responde corretamente às entradas fornecidas, com base nos requisitos funcionais;
- Caixa Branca: O teste de caixa branca (*white-box testing*) consiste em analisar o funcionamento interno do sistema. O testador tem total acesso ao código-fonte e utiliza isso para verificar fluxos de controle, condições lógicas, estruturas de decisão, entre outros;
- Caixa Cinza: O teste de caixa cinza (*gray-box testing*) combina características das abordagens de caixa preta e caixa branca. O testador possui conhecimento parcial da estrutura interna do sistema, o que possibilita a criação de casos de teste mais direcionados e eficazes.

2.3.3 Classificação por intenção

Essa classificação se refere ao propósito dos testes criados.

- Teste Limpo: O teste limpo, também conhecido como *clean test*, é aquele em que as entradas fornecidas ao sistema são válidas e esperadas. O principal objetivo é verificar se o sistema se comporta corretamente diante de condições normais de uso, conforme definido pelos requisitos;
- Teste Sujo: O teste sujo, ou *dirty test*, consiste em submeter o sistema a entradas inválidas e inesperadas, com o intuito de expor falhas e erros, e avaliar a qualidade da aplicação. Essa abordagem visa identificar comportamentos indesejados ou

vulnerabilidades que possam comprometer a confiabilidade ou a segurança do sistema.

2.3.4 Classificação por escopo

Essa classificação considera a etapa que o desenvolvimento está e o escopo que o tipo de teste abrange.

- Teste Unitário: O teste unitário é responsável por verificar, de forma isolada, as menores unidades testáveis de um sistema, como funções, métodos ou classes. Seu principal objetivo é garantir que cada unidade de código se comporte corretamente conforme o esperado;
- Teste de Integração: O teste de integração avalia a interação entre diferentes módulos, componentes ou serviços de um sistema. O foco está em verificar se os elementos funcionam corretamente quando combinados e trocam informações de forma adequada;
- Teste de Sistema: Também conhecido como "teste *end-to-end*", este teste verifica o sistema em sua totalidade, simulando o ambiente de produção e os cenários de uso, é conduzido a partir da perspectiva do usuário. O objetivo é validar se o *software* atende aos requisitos funcionais e não funcionais definidos na especificação;
- Teste de Regressão: O teste de regressão é executado após modificações no sistema, como correções de *bugs*, melhorias ou novas funcionalidades, com o intuito de garantir que as alterações não tenham introduzido novos erros ou comprometido funcionalidades existentes;
- Teste de Performance: Esse tipo de teste é voltado para a avaliação de requisitos não funcionais, especialmente relacionados ao desempenho do sistema sob diferentes condições de uso. Inclui testes de carga e escalabilidade, permitindo analisar o tempo de resposta, consumo de recursos (CPU, memória) e a estabilidade do sistema quando submetido a altas demandas;
- Teste Automatizado: Os testes automatizados são aqueles implementados com o auxílio de ferramentas que permitem a execução automática dos casos de teste. Eles são úteis para garantir repetibilidade, agilidade e precisão na verificação contínua do sistema, especialmente em cenários de integração contínua e entrega contínua (CI/CD).

Além dessas classificações tradicionais dos testes de *software*, é possível também a divisão segundo estratégias de desenvolvimento. Essas estratégias não apenas

definem quando e como os testes são escritos, mas também refletem filosofias de desenvolvimento que impactam diretamente na estrutura, qualidade e manutenção do *software* ao longo do tempo. Dentre os principais exemplos, tem-se: Desenvolvimento Orientado a Testes e Desenvolvimento Orientado ao Comportamento.

2.3.5 Test-driven development (TDD)

O Desenvolvimento Orientado a Testes (TDD — *Test-Driven Development*) é uma prática de desenvolvimento que propõe a criação de testes automatizados antes da implementação efetiva do código. Essa abordagem se baseia em ciclos curtos e repetitivos de desenvolvimento, com o objetivo de validar o comportamento do sistema desde as fases iniciais da codificação (BECK, 2010). O ciclo clássico do TDD é conhecido como *Red-Green-Refactor*, esse ciclo pode ser descrito em três etapas principais, conforme detalhado a seguir.

Iniciando, o desenvolvedor escreve um teste para uma funcionalidade ainda não implementada, configurando a etapa "vermelha", pois o teste deve falhar. Logo em seguida, ele implementa o código mínimo necessário para que o teste passe, a etapa "verde". Por fim, realiza melhorias no código garantindo que os testes continuem a passar, o que representa a etapa de "refatoração"e o ciclo é finalizado.

Essa metodologia é eficaz para evitar regressões, garantir a cobertura do código e promover um design mais limpo e sem dependências. Segundo Pressman (2016), incorporar os testes no processo de criação da lógica da aplicação, permite que falhas sejam detectadas de forma precoce, ainda no ambiente de desenvolvimento, o que acontece com o uso do TDD. Isso reduz drasticamente a possibilidade de que erros críticos sejam entregues ao cliente final, preservando a confiabilidade do sistema e a reputação do time de desenvolvimento.

Além disso, por fazer o desenvolvedor pensar previamente sobre os requisitos e comportamentos esperados de cada funcionalidade, o TDD também funciona como uma ferramenta de refinamento de requisitos, contribuindo para um entendimento mais claro das demandas do sistema, algo que Beizer (1995) sempre encorajou. Essa prática favorece a construção de soluções focadas em atender exatamente ao que é solicitado pelo teste, evitando aumento na complexidade ou funcionalidades desnecessárias. Outro benefício importante do TDD é a melhoria da manutenibilidade e evolutividade do sistema.

Como os testes formam uma espécie de documentação verificável e atualizável do comportamento da aplicação, futuras modificações podem ser feitas com mais segurança, já que os testes ajudam a garantir que novas alterações não quebrem funcionalidades existentes (BECK, 2010). Além disso, o TDD estimula a criação de componentes com baixa dependência e alta coesão, características essenciais para sistemas que precisam evoluir continuamente. No contexto de *frameworks* como o

Laravel, o TDD pode ser aplicado de maneira integrada com ferramentas como o *PHPUnit*, permitindo a criação de testes automatizados de forma eficiente e alinhada com as boas práticas do ecossistema.

2.3.6 Behavior driven development (BDD)

O Desenvolvimento Orientado ao Comportamento (BDD - *Behavior Driven Development*) surgiu como uma evolução do TDD, porém não tem o objetivo de substitui-lo. Assim como o TDD, o BDD tem como base a criação de testes antes do código. Concebido inicialmente por North (2006), ele afirma que "teste é uma frase que descreve o próximo comportamento no qual você está interessado".

Nessa técnica o foco não está apenas na validação do código, mas sim na descrição clara dos comportamentos esperados da aplicação a partir da perspectiva do usuário final. Esse método promove uma comunicação eficaz entre todas as partes envolvidas no projeto com o objetivo de minimizar ambiguidades e criar um entendimento comum sobre o que o sistema deve fazer. O BDD, portanto, atua não apenas como uma técnica de teste, mas como uma prática de especificação colaborativa.

Um dos princípios centrais do BDD é a chamada Regra dos Três Amigos, definida por Dinwiddie (2011) e incorporada por Dan North, que recomenda que cada funcionalidade seja discutida e refinada por três perspectivas diferentes: a do desenvolvedor, que pensa em como implementar, do testador, que pensa em como validar, e do representante do negócio, que pensa no valor e na lógica da funcionalidade. Essa visão tripla serve como base para a criação de cenários de testes claros e objetivos, geralmente escritos no formato Gherkin, usando as estruturas Dado (*Given*), Quando (*When*) e Então (*Then*).

Um exemplo prático desse formato seria: "Dado"um contexto "Quando"ele for acionado "Então"deve retornar ou fazer algo. A linguagem natural dos cenários garante que todos os envolvidos compreendam os critérios de aceitação antes da implementação, o que reduz falhas de entendimento e retrabalho. Com isso, a documentação dos requisitos deixa de ser estática e se transforma em algo vivo, verificável e integrado ao código, mais uma vez reforçando a necessidade da compreensão profunda dos requisitos.

No contexto do desenvolvimento com *Laravel*, o BDD permite conectar os cenários escritos em Gherkin a funções de teste em PHP. Esse processo possibilita testar comportamentos reais do sistema, como fluxos de usuário, regras de negócio e interações entre componentes, com validação automatizada. Ao priorizar a clareza dos requisitos e o entendimento compartilhado entre os membros da equipe, o BDD facilita a construção de sistemas mais confiáveis, focados em entregar valor ao usuário final. Além disso, por funcionar como uma documentação executável, o BDD também contri-

bui para a manutenibilidade do sistema e para a detecção precoce de inconsistências no comportamento da aplicação.

2.3.7 Métricas de qualidade

A avaliação da qualidade de testes de *software* baseia-se na aplicação de métricas que quantificam atributos como desempenho, confiabilidade, eficiência e cobertura. Essas métricas permitem analisar tanto a eficácia dos testes quanto a qualidade do próprio sistema testado (PRESSMAN, 2016). No contexto deste trabalho, serão consideradas as seguintes métricas:

- Tempo médio de execução: mede o tempo médio necessário para executar um conjunto de testes. É uma métrica associada à eficiência dos testes e à capacidade de integração em ciclos de desenvolvimento ágeis e automação contínua;
- Taxa de sucesso: indica a porcentagem de testes que foram concluídos com êxito, ou seja, cujos resultados coincidem com os comportamentos esperados. Essa métrica reflete a conformidade do sistema com os requisitos funcionais;
- Taxa de falhas: representa a proporção de testes que falham durante a execução. Uma taxa elevada pode sinalizar problemas de estabilidade no software, regressões ou falhas na construção dos testes;
- *Throughput:* refere-se ao número de testes, requisições ou operações realizadas por unidade de tempo. É uma métrica fundamental em testes de carga e desempenho, pois indica a capacidade do sistema de lidar com volume (BEIZER, 1995);
- Uso de recursos: mede o consumo de recursos computacionais durante a execução dos testes, como CPU, memória RAM, disco e rede. É particularmente relevante em testes de estresse e desempenho;
- Cobertura de testes: avalia o percentual de código-fonte que é exercitado pelos testes. Embora uma alta cobertura não garanta qualidade, valores baixos indicam testes insuficientes;
- Tempo de resposta: especialmente em testes de performance, essa métrica mede quanto tempo o sistema leva para responder a uma requisição. Ela é decisiva para a experiência do usuário final e a escalabilidade da aplicação (PRESSMAN, 2016).

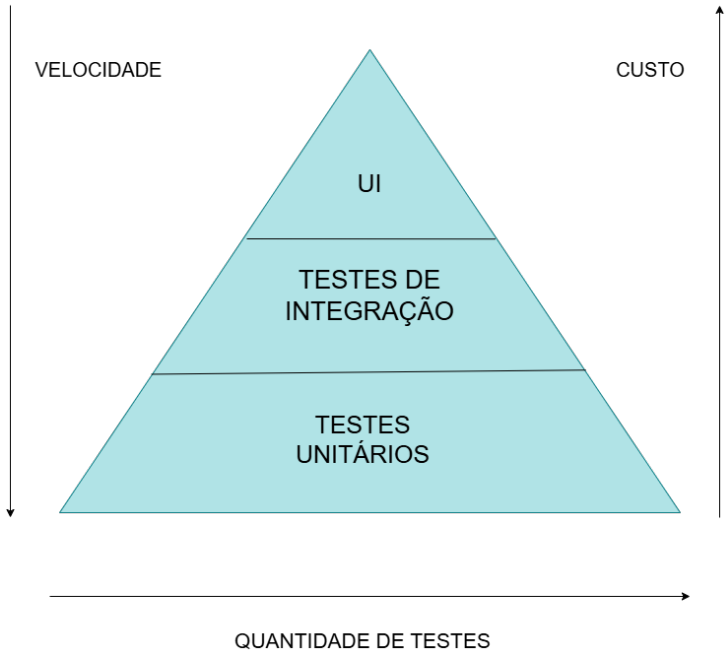
Essas métricas, quando utilizadas em conjunto, proporcionam uma visão abrangente sobre a robustez e a eficiência dos testes aplicados. Além disso, permitem

comparações entre diferentes abordagens de testes, ferramentas e *frameworks* de desenvolvimento, como é o caso do *Laravel* no presente estudo.

2.4 PIRÂMIDE DE TESTES

A Pirâmide de Testes é um modelo conceitual amplamente utilizado para representar a distribuição ideal de testes automatizados em um sistema de *software*. O modelo foi popularizado por Fowler (2018), essa abordagem propõe a organização dos testes em diferentes níveis de abstração, priorizando testes mais rápidos, simples e de menor custo de manutenção na base da estrutura. Na base da pirâmide encontram-se os testes unitários, responsáveis por validar o comportamento isolado de métodos, funções e regras de negócio específicas. A figura 1 representa visualmente a pirâmide.

Figura 1 – Pirâmide de Testes



Fonte: Adaptado de (FOWLER, 2018)

Os testes unitários possuem baixo custo de execução, fornecem retorno rápido durante o desenvolvimento e facilitam a identificação de falhas pontuais no sistema. Por esse motivo, é esperado que representem a maior parte do conjunto de testes. No nível intermediário estão os testes de integração, utilizados para verificar a comunicação entre diferentes componentes da aplicação, como acesso ao banco de dados, serviços internos e integração entre módulos. Embora mais lentos e complexos que os testes unitários, esses testes são importantes para validar o comportamento conjunto das partes do sistema.

No topo da pirâmide estão os testes funcionais ou *end-to-end*, que simulam cenários reais de utilização da aplicação a partir da perspectiva do usuário final.

Esses testes validam fluxos completos do sistema, incluindo interface, navegação e interação entre múltiplas camadas da aplicação. Apesar de essenciais para garantir o funcionamento geral do *software*, possuem maior custo de execução e manutenção, devendo ser utilizados de forma mais restrita.

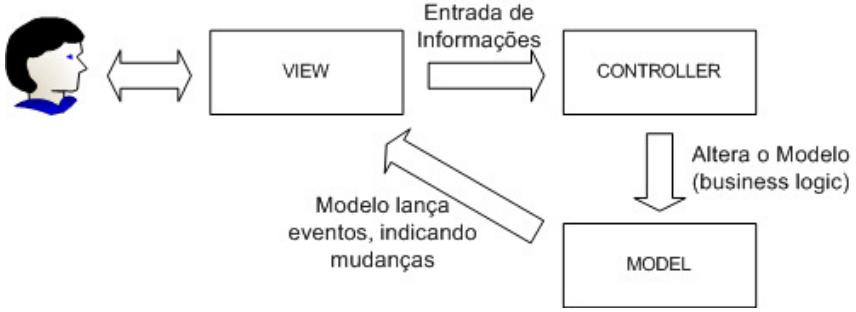
A principal proposta da Pirâmide de Testes consiste em evitar a dependência excessiva de testes funcionais, priorizando uma base sólida de testes unitários e complementando-a com testes de integração e sistema. Segundo (FOWLER, 2018), essa distribuição contribui para maior estabilidade, rapidez de execução e sustentabilidade da suíte de testes ao longo da evolução do *software*.

2.5 LARAVEL

O *Laravel* é um *framework* PHP de código aberto, criado por Taylor Otwell em 2011, que se propõe a ser uma ferramenta completa para aumentar a produtividade dos desenvolvedores. Ele oferece soluções elegantes e eficientes para aplicações *web* modernas, com uma sintaxe simples, porém poderosa. Trata-se de um *framework full-stack*, ou seja, fornece recursos tanto para o *front-end*, responsável pela interface visual do sistema, quanto para o *back-end*, responsável pela lógica da aplicação (OTWELL, 2025).

Sua estrutura baseia-se no padrão arquitetural MVC (*Model-View-Controller*), que organiza o código em três camadas: o *Model*, que representa os dados e a lógica de negócio; a *View*, responsável pela interação com o usuário; e o *Controller*, que atua como intermediário entre as requisições feitas pelo usuário e a resposta gerada pelo sistema, conforme representado na Figura 2. Essa separação de responsabilidades favorece a organização e a manutenção do código (LARAVEL, 2024). A estrutura de diretórios do Laravel foi pensada para escalar com o projeto, possuindo pastas bem definidas para modelos, controladores, rotas, visualizações e outros componentes.

Figura 2 – Fluxo de eventos e informações em uma arquitetura MVC



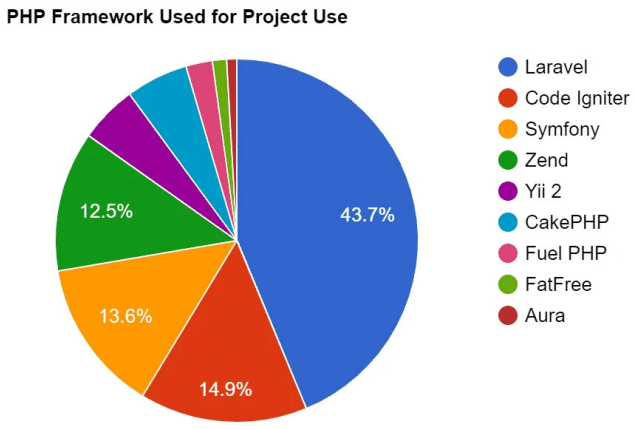
Fonte: <http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/arqu/mvc/mvc.htm>

Cada elemento da aplicação possui seu local apropriado, o que facilita tanto o desenvolvimento quanto a compreensão do sistema. Complementando essa estrutura,

o *Laravel* oferece o *Artisan CLI*, uma interface de linha de comando que automatiza tarefas como criação de classes, execução de migrações e testes, otimizando significativamente o fluxo de desenvolvimento (LARAVEL, 2024). Entre os conceitos fundamentais do framework estão as rotas, que mapeiam *URLs* para métodos de controladores ou funções anônimas; os controladores, que tratam as requisições e fornecem respostas; os modelos, que interagem com o banco de dados via *ORM Eloquent*; e as *views*, construídas com o construtor de *templates Blade* (LARAVEL, 2024).

O *Laravel* também se destaca pelo suporte avançado a banco de dados e testes automatizados. O *Eloquent* permite criar modelos, migrações, relacionamentos e executar consultas com fluidez, além de facilitar a criação de dados por meio de *factories* e *seeds*, que automatizam a geração de registros para desenvolvimento e testes (LARAVEL, 2024). Por fim, o *Laravel* se destaca por sua documentação clara e acessível, além de contar com uma comunidade ampla e ativa, fator decisivo para sua popularidade entre os desenvolvedores PHP, como mostra a Figura 3 (ExcellentWebWorld, 2025).

Figura 3 – Gráfico de Pizza do Uso de *Frameworks* PHP em Projetos Web.



Fonte: (ExcellentWebWorld, 2025).

2.5.1 Testes no Laravel

A estrutura de testes é baseada no *PHPUnit*, uma biblioteca de testes automatizados, já integrada ao *framework*, com diretórios organizados para testes unitários e de funcionalidade. O *Laravel* também fornece o *Laravel Dusk*, uma ferramenta para testes *end-to-end* que simula interações reais do usuário no navegador, como cliques, preenchimento de formulários e validações em *JavaScript*. Essa combinação entre *PHPUnit* e *Dusk* garante cobertura tanto da lógica interna quanto da interface visual da aplicação (LARAVEL, 2024). Essa abordagem aproxima-se da Pirâmide de Testes proposta por Fowler (2018), pois envolve a combinação de testes unitários, integração e sistema para validar diferentes níveis da aplicação.

2.6 CACHET

O *Cachet* é um sistema de código aberto desenvolvido para facilitar a comunicação do *status* de serviços *online*. Sua principal função é permitir que empresas, organizações ou equipes técnicas informem seus usuários sobre a disponibilidade de serviços, falhas, interrupções e manutenções programadas. Ele é utilizado como uma "*status page*", ou página de *status*, que pode ser acessada publicamente para verificar a saúde e o funcionamento dos sistemas de uma empresa em tempo real (CachetHQ-Documentação oficial do Cachet, em desenvolvimento ativo, 2025).

O *Cachet* é desenvolvido utilizando o *framework Laravel*, o que o torna um sistema compatível com as práticas modernas de desenvolvimento e manutenção de *software* orientadas à qualidade, sendo possível testar desde funcionalidades isoladas até fluxos completos de uso. Outro aspecto importante é que o *Cachet* encontra-se atualmente em processo de atualização, com uma nova versão principal, o *Cachet* v3, em andamento. O sistema *Cachet* adota o *PestPHP* como principal *framework* de testes, devido à sua sintaxe mais expressiva e legível. Essa nova versão traz uma reestruturação significativa no código-fonte, com melhorias na organização do projeto, na experiência de uso e no suporte a novas tecnologias.

A escolha do *Cachet* como estudo de caso neste trabalho se justifica pelo amplo uso de *Laravel*, a existência de funcionalidades concretas e testáveis, a natureza open source e a relevância prática da ferramenta. Além disso, o fato de o sistema estar passando por atualizações constantes reforça sua atualidade e utilidade como objeto de estudo, além de abrir a possibilidade para que os testes planejados neste trabalho possam contribuir com sugestões reais de melhoria ou validação.

Trata-se de um sistema suficientemente complexo para a aplicação de diferentes técnicas de testes, mas ainda assim acessível e bem documentado, o que facilita sua análise no contexto acadêmico. Portanto, o *Cachet* representa uma base sólida e realista para explorar a aplicação de testes de *software* em sistemas Laravel, permitindo que este trabalho una teoria e prática na avaliação da qualidade de sistemas *web*.

2.7 TRABALHOS RELACIONADOS

Um dos estudos analisados, realizado por Carosia, Castro & Calazans (2024), adotou uma abordagem metodológica qualitativa, fundamentada em pesquisa bibliográfica e análise heurística. Ele teve como objetivo identificar pontos fortes e pontos de melhoria na experiência do usuário ao utilizar *sites* de *e-commerce*. A metodologia utilizada se baseou em uma revisão de literatura sobre Engenharia de *Software* e testes de usabilidade, seguida da aplicação das heurísticas de Jakob Nielsen para avaliar a experiência do usuário nos *sites* da *Amazon* e Magazine Luiza.

Os resultados mostraram que tanto a *Amazon* quanto o Magazine Luiza apre-

sentaram pontos fortes em aspectos como prevenção de erros, consistência e usabilidade, embora ambos possuam cenários de melhora. A avaliação heurística evidenciou que, apesar de bem empregada, a experiência do usuário ainda pode ser aprimorada para proporcionar uma experiência do usuário mais intuitiva e satisfatória. As considerações finais destacaram que não houve um vencedor em termos de usabilidade, porém é fundamental continuar investindo na otimização da experiência de compra *online*, garantindo maior conveniência, eficiência e satisfação aos usuários.

Em relação ao presente trabalho, o estudo de Carosia, Castro & Calazans (2024) apresenta similaridades ao adotar uma abordagem qualitativa e fundamentada em análise de critérios de qualidade, como usabilidade e prevenção de erros. Ambos os trabalhos compartilham o objetivo de identificar falhas e oportunidades de melhoria em sistemas utilizados por usuários finais. No entanto, enquanto o estudo citado foca na experiência do usuário por meio de uma avaliação heurística aplicada a interfaces de *e-commerce*, este trabalho adota uma perspectiva mais técnica e automatizada, voltada à análise estrutural e funcional de um sistema real no contexto do *framework Laravel*. Este trabalho avança ao propor um conjunto de testes capazes de garantir não apenas uma boa experiência do usuário, mas também a confiabilidade, manutenção e desempenho do sistema em sua totalidade oferecendo uma contribuição complementar e alinhada com práticas modernas de desenvolvimento.

Outra revisão de literatura qualitativa, desenvolvida por Bettoni, Florian & Farina (2024), teve como objetivo evidenciar a importância dos testes de *software* na garantia da qualidade em sistemas empresariais. Testes funcionais, de integração e de desempenho foram analisados comparativamente, demonstrando como cada um contribui de forma específica para a redução de falhas e melhoria contínua dos produtos desenvolvidos. O procedimento adotado combinou pesquisa bibliográfica com testes realizados em uma empresa de desenvolvimento de *software*, buscando verificar, de forma prática, o impacto na qualidade final do produto. Os testes não apenas detectaram falhas, como também serviram como ferramenta de apoio na tomada de decisões e no aprimoramento dos processos internos da empresa.

Graças a esses testes, observou-se uma significativa diminuição de erros e retrabalho, indicando que a validação sistemática ao longo do desenvolvimento pode prevenir problemas antes da entrega ao cliente. Os resultados mostraram que cerca de metade dos cenários testados apresentaram comportamentos inesperados, o que reforça a importância de uma cultura de testes robusta e bem estruturada. A identificação precoce dessas falhas permitiu que ajustes fossem feitos antes da entrega final, aumentando a qualidade percebida e a confiança dos clientes nos sistemas fornecidos. Conclui-se, portanto, que os testes de *software* são fundamentais para assegurar não apenas a funcionalidade adequada dos sistemas, mas também sua confiabilidade e fidelidade às expectativas dos usuários finais.

O estudo de Bettoni, Florian & Farina (2024) e este trabalho compartilham o objetivo de destacar a relevância dos testes de *software* para a qualidade de sistemas. Ambos analisam diferentes tipos de testes e seus impactos na redução de falhas. No entanto, enquanto o estudo citado foca no ambiente corporativo e nos benefícios organizacionais dos testes, este estudo se volta à aplicação técnica e prática de testes no *Laravel*, com base no sistema escolhido. A principal vantagem do trabalho analisado está na comprovação de ganhos operacionais reais, enquanto este estudo contribui ao explorar ferramentas específicas do *framework* e propor melhorias alinhadas às práticas modernas de desenvolvimento.

Outro trabalho relevante, desenvolvido por Pelizza, Bertolini & Silveira (2018), apresentou uma análise detalhada das técnicas e tipos de testes de *software* presentes no *framework Laravel*. Destacou-se a importância do teste contínuo para garantir a qualidade e a confiabilidade de aplicações, mesmo quando se utilizam *frameworks* que facilitam o desenvolvimento. Foram abordadas as diferentes categorias de testes, como unitários, de sistema e de aceitação, além dos recursos específicos que o *Laravel* oferece para automação e configuração de testes, facilitando sua implementação de forma eficiente. O estudo enfatizou a necessidade de uma abordagem organizada e integrada ao processo de desenvolvimento, ressaltando que o uso de testes contribui para a redução de custos e para a detecção precoce de falhas.

Além disso, avaliou as dificuldades enfrentadas no teste de aplicações *web* complexas, como a questão de diferentes fluxos de navegação, reforçando o papel dos testes na identificação de falhas durante o desenvolvimento. O trabalho também mostrou que o *Laravel* fornece ferramentas que favorecem a criação de testes robustos, contribuindo para a manutenção e evolução do *software*. Em seguida, o trabalho propôs esquemas de teste baseados nas funcionalidades do *Laravel*, buscando maximizar a efetividade na identificação de erros e na validação do comportamento do sistema. Por fim, ressaltou a importância de integrar o teste como uma etapa fundamental no ciclo de vida do desenvolvimento de aplicações web com *Laravel*.

O trabalho de Pelizza, Bertolini & Silveira (2018) possui forte relação com este trabalho, ao abordar diretamente os recursos de teste disponíveis no *Laravel* e sua aplicação prática. Tanto o trabalho de Pelizza, Bertolini & Silveira (2018) quanto este trabalho reforçam a importância dos testes no desenvolvimento *web*, especialmente no contexto do *Laravel*. No entanto, enquanto o estudo citado apresenta uma visão mais ampla e teórica das ferramentas e práticas de teste, este trabalho busca aplicar essas abordagens em um sistema real, avaliando na prática seus benefícios, limitações e possibilidades de aprimoramento. A principal contribuição deste estudo está na proposta e análise concreta de testes específicos, aprofundando o uso das ferramentas em um caso real para fomentar boas práticas e reflexões técnicas mais aplicadas.

2.8 FERRAMENTAS E TECNOLOGIAS

Esta seção tem como objetivo listar e especificar brevemente as tecnologias e ferramentas envolvidas no processo de desenvolvimento e aplicação dos testes.

2.8.1 Composer

O *Composer* é o gerenciador de dependências oficial do PHP. Ele permite que desenvolvedores declarem as bibliotecas e ferramentas que o projeto necessita, instalando-as automaticamente e mantendo suas versões atualizadas (Composer Project, 2025). No contexto do *Laravel*, o *Composer* é essencial para a instalação do próprio *framework*, bem como de pacotes complementares, como o *PHPUnit* e outras bibliotecas de testes ou ferramentas auxiliares.

2.8.2 Visual Studio Code

O *Visual Studio Code*, ou *VS Code*, é um editor de código aberto desenvolvido pela *Microsoft*, disponível para *Windows*, *Linux* e *MAC* que possui funcionalidades como: edição de código com suporte a várias linguagens de programação, controle de versão e terminal embutido (Microsoft, 2025). Um ponto forte é a customização, atalhos e ampla variedade de extensões disponíveis, tanto para personalização quanto para aumento de produtividade. No projeto, o *VS Code* será utilizado como ambiente de desenvolvimento, auxiliando na escrita, organização e execução dos testes no sistema analisado.

2.8.3 Github

O *GitHub* é uma plataforma de hospedagem de código baseada no sistema de versionamento *Git*. Ele permite o controle de alterações no código-fonte, o trabalho colaborativo entre desenvolvedores e a integração com ferramentas de *CI/CD* (GitHub Inc., 2025). Para este trabalho, o *GitHub* é utilizado para armazenar e versionar o código-fonte do sistema *Cachet*, permitindo o acesso ao repositório oficial e facilitando a análise do projeto.

2.8.4 Git

O *Git* é um sistema de controle de versão distribuído, amplamente utilizado no desenvolvimento de *software* (Git Project, 2025). Ele permite acompanhar o histórico de mudanças no código, criar ramificações para novas funcionalidades, e colaborar de forma segura em projetos com múltiplos desenvolvedores. No contexto deste trabalho, o *Git* é utilizado em conjunto com o *GitHub* para acompanhar e registrar modificações no código durante a análise e planejamento dos testes.

2.8.5 PHPUnit

O *PHPUnit* é um *framework* de testes para a linguagem PHP, amplamente utilizado no ecossistema *Laravel*. Ele permite a criação de testes automatizados que verificam o comportamento de funções, métodos e fluxos do sistema, contribuindo para a detecção precoce de erros e a melhoria da qualidade do código (Sebastian Bergmann and contributors, 2025). O *Laravel* possui integração nativa com o *PHPUnit*, o que facilita a configuração e execução dos testes em projetos desenvolvidos com o *framework*.

2.8.6 PestPHP

O *PestPHP* é um *framework* de testes moderno para aplicações *PHP* que se destaca por sua sintaxe concisa e expressiva, baseada em princípios de desenvolvimento orientado a comportamento (Nuno Maduro et al., 2024). Totalmente compatível com o *PHPUnit*, ele oferece uma camada simplificada para a escrita de testes unitários, funcionais e de integração, promovendo maior legibilidade e organização do código de teste. No contexto deste trabalho, o *Pest* será utilizado como ferramenta principal para a criação de testes automatizados, especialmente na verificação da lógica interna da aplicação e no comportamento de suas classes e serviços.

2.8.7 Laravel Dusk

O *Laravel Dusk* é uma ferramenta oficial para testes automatizados de interface (*end-to-end*) no Laravel, baseada no uso de navegadores reais via o *ChromeDriver* (Laravel: Documentação oficial, 2024a). Ela permite simular interações reais do usuário com a aplicação, como cliques, preenchimento de formulários e navegação por páginas, sendo útil na validação de fluxos completos e regressões visuais. No contexto deste trabalho, o *Dusk* será utilizado na etapa de testes funcionais automatizados com foco na experiência do usuário.

2.8.8 Laravel Telescope

O *Laravel Telescope* é uma ferramenta de monitoramento e depuração mantida oficialmente pelo time do Laravel (Laravel: Documentação oficial, 2024b). Ele fornece uma interface gráfica para inspecionar requisições *HTTP*, comandos *Artisan*, consultas ao banco de dados, filas, exceções, entre outros eventos relevantes. Essa visibilidade detalhada é fundamental durante o desenvolvimento e será especialmente útil na coleta de métricas de desempenho e comportamento da aplicação no decorrer dos testes.

2.8.9 Laravel Debugbar

O Laravel Debugbar é uma ferramenta de depuração amplamente utilizada na comunidade Laravel, baseada no *PHP Debugbar* (HEUVEL, 2024). Ela adiciona uma barra interativa à interface da aplicação, exibindo informações como *queries SQL* executadas, tempo de resposta, rotas utilizadas, variáveis carregadas e eventos do *framework*. Apesar de não ser oficial, sua integração simples com o *Laravel* a torna uma aliada valiosa durante a análise de comportamento em tempo real.

2.8.10 Xdebug

O *Xdebug* é uma extensão para o PHP que oferece recursos avançados de depuração e análise de desempenho (RETHANS, 2024). Ele permite a inspeção passo a passo do código, a geração de relatórios de cobertura de testes e o perfil e execução. No contexto deste trabalho, o *Xdebug* será empregado para mensurar a cobertura de código dos testes automatizados, identificando pontos críticos não testados e apoiando a melhoria contínua da base de testes.

2.8.11 ApacheBench

O *ApacheBench* é uma ferramenta de linha de comando desenvolvida pela *Apache Software Foundation* para realizar testes de desempenho em aplicações *web* (Apache Software Foundation, 2024). Ela permite simular múltiplas requisições simultâneas a um determinado *endpoint HTTP*, gerando métricas como tempo médio de resposta, taxa de requisições por segundo (*throughput*) e percentual de falhas. Neste trabalho, o *ApacheBench* será utilizado para simular cenários de carga na aplicação *Cachet*, com o objetivo de medir o desempenho e a estabilidade do sistema sob diferentes volumes de acesso.

3 METODOLOGIA

Neste capítulo, serão descritas as técnicas, ferramentas e tecnologias adotadas para o desenvolvimento dos testes, bem como a apresentação do sistema que servirá como estudo de caso na aplicação e avaliação dos testes propostos. A escolha dessas abordagens tem como objetivo a garantia de confiabilidade dos resultados. Antes de iniciar a aplicação dos testes no sistema escolhido, é fundamental definir uma metodologia clara e organizada para garantir que o planejamento, desenvolvimento e avaliação dos testes sejam realizados de forma eficiente.

Para este trabalho, optou-se por uma abordagem estruturada, que envolve o estudo do sistema *Cachet*, o planejamento detalhado dos testes e a utilização das ferramentas nativas do Laravel, como o *PHPUnit*, reconhecidas por sua robustez e ampla aceitação na comunidade de desenvolvimento. Essa metodologia visa assegurar a confiabilidade dos resultados e a qualidade do processo de teste durante o desenvolvimento do estudo.

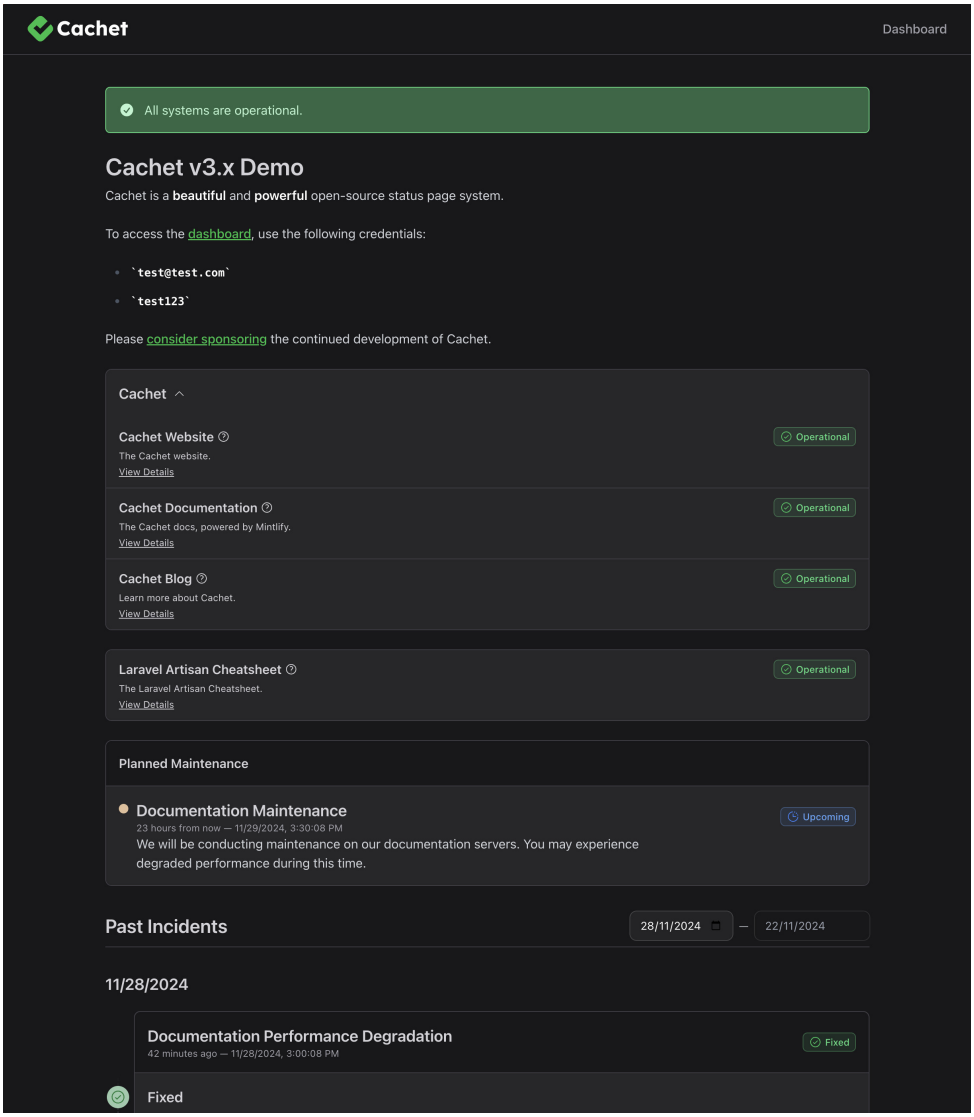
3.1 ESTUDO DE CASO

O sistema utilizado como base para este trabalho é o *Cachet*, um sistema *open source* utilizado para criar páginas de status de serviços, permitindo a comunicação transparente com usuários sobre incidentes, manutenções e disponibilidade de sistemas. Sua arquitetura estruturada e uso real no mercado tornam o *Cachet* uma base prática ideal para estudo e aplicação de testes. O sistema *Cachet* segue o padrão arquitetural MVC, no qual as requisições *HTTP* são inicialmente tratadas pelas rotas definidas no *framework Laravel*, direcionadas aos controladores responsáveis pela lógica de negócio, que por sua vez interagem com os modelos e o banco de dados por meio do *ORM Eloquent*.

Desenvolvido com *Laravel* 11, esse padrão promove separação de responsabilidades, facilitando a testabilidade e manutenção do sistema. Ele também conta com funcionalidades específicas e necessárias para um sistema de página de *status*, e que são relevantes para análise de testes, conforme descritas e ilustradas pelas figuras a seguir, obtidas a partir de sua versão *demo*, e *dashboard* disponíveis no *site* oficial.

- **Página Pública de Status:** Representada pela Figura 4, é a interface *web* acessível aos usuários finais, na qual são exibidos os incidentes registrados, manutenções agendadas e o estado atual dos serviços. Essa página serve como um canal de transparência, permitindo que os clientes acompanhem o histórico de incidentes e a saúde geral do sistema em tempo real;

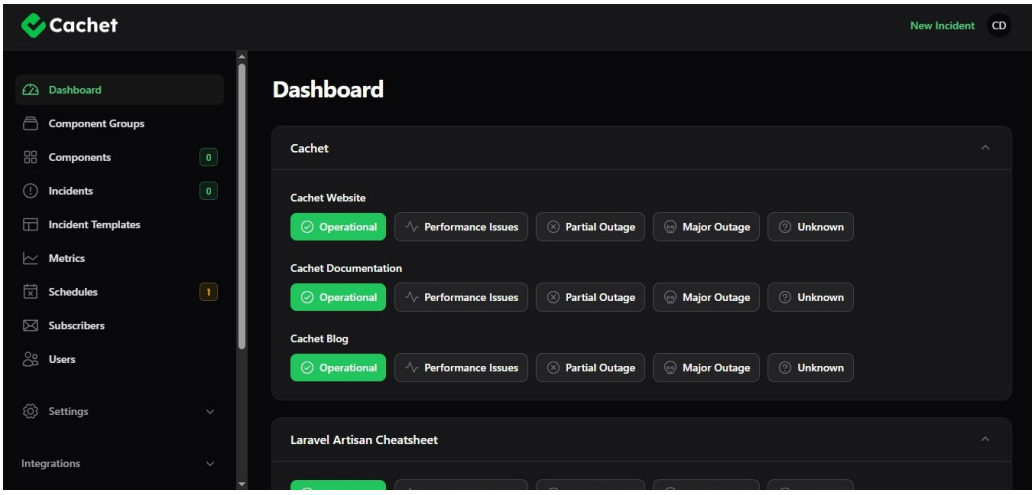
Figura 4 – Captura de tela do Cachet v3.x, página de status



Fonte: Cachet v3.x Demo

- **Tela inicial (*Dashboard*):** É a primeira interface exibida ao acessar o sistema de gerenciamento do *Cachet*. Nela, é possível obter uma visualização simplificada, rápida e objetiva dos serviços monitorados, oferecendo um panorama geral do estado atual dos componentes acompanhados pela aplicação, conforme observado na Figura 5;

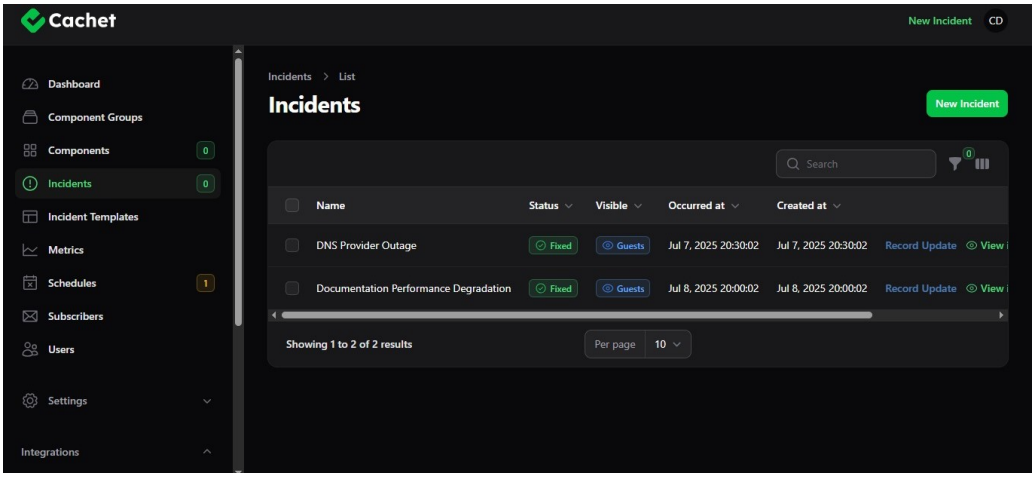
Figura 5 – Tela inicial de gerenciamento, Dashboard



Fonte: Cachet Demo Dashboard

- **Tela de Incidentes:** Faz parte do conjunto de telas do *Dashboard*. Nessa interface observada na Figura 6, são listados todos os incidentes cadastrados, além de permitir o registro de novos. Também é possível visualizar de forma resumida o status, a visibilidade e informações relevantes para auditoria, como a data da ocorrência e o momento do cadastro no sistema. A tela possibilita ainda a criação de relatórios de incidentes, contendo título, descrição, impacto e atualizações, podendo estes ser classificados como planejados ou inesperados;

Figura 6 – Tela de Incidentes

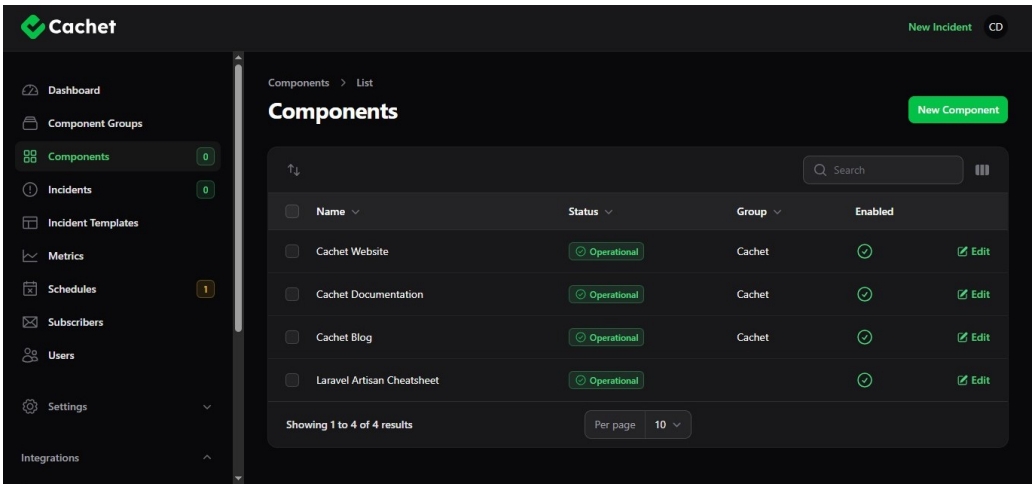


Fonte: Cachet Demo Dashboard

- **Tela de Componentes:** Também integrante do *Dashboard* de gerenciamento, esta tela é destinada ao cadastro e gerenciamento dos componentes, ou seja, das partes que compõem o sistema. Nela, é possível registrar novos serviços e monitorar sistemas específicos, vinculando-os a um grupo identificado como "*Group*", como pode ser visto na Figura 7. A interface permite visualizar e alterar o

status de operacionalidade de cada componente, que pode ser classificado como: operacional, falha parcial, falha total, manutenção programada, entre outros;

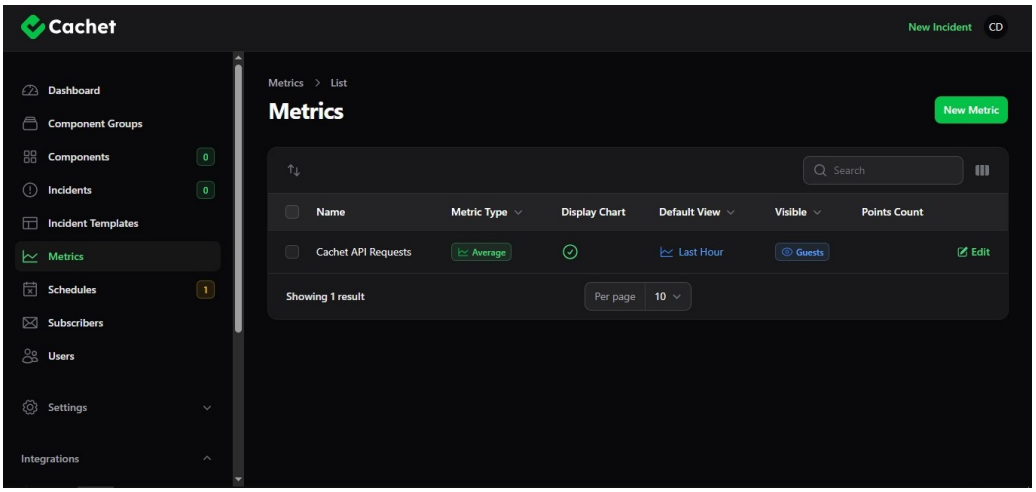
Figura 7 – Tela de Componentes



Fonte: Cachet Demo Dashboard

- **Tela de Métricas:** Ainda dentro do *Dashboard*, essa tela é responsável por gerenciar todos os aspectos mensuráveis do sistema, como o número de requisições e a quantidade de componentes operacionais. É possível definir o tipo de métrica, por exemplo, média ou soma, configurar sua forma de medição e decidir se ela será visível para os clientes, conforme representa a Figura 8. Além disso, a tela exibe indicadores de desempenho em formato gráfico, permitindo o acompanhamento visual de métricas como latência e tempo de resposta;

Figura 8 – Tela de Métricas

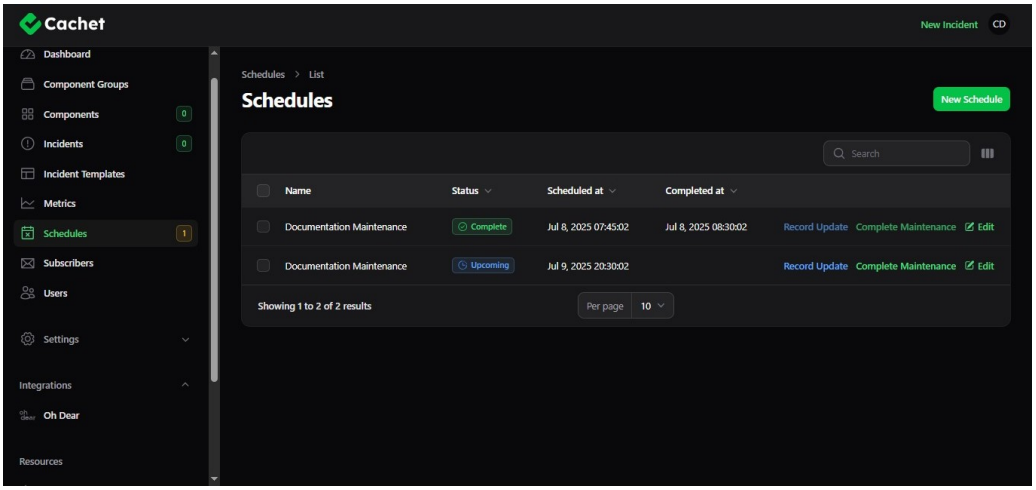


Fonte: Cachet Demo Dashboard

- **Tela de Cronogramas:** Representada pela Figura 9, essa tela do *Dashboard* permite o gerenciamento completo de manutenções agendadas no sistema. Nela,

é possível cadastrar novas manutenções, informando dados como data de início e término, além do impacto previsto para os usuários, por exemplo, possíveis períodos de inatividade. Também é possível acompanhar e atualizar o status da manutenção em tempo real, como "agendada", "em andamento"ou "concluída". Essa funcionalidade contribui para uma comunicação clara e proativa com os clientes, promovendo transparência e confiança ao sinalizar previamente eventuais interrupções planejadas nos serviços;

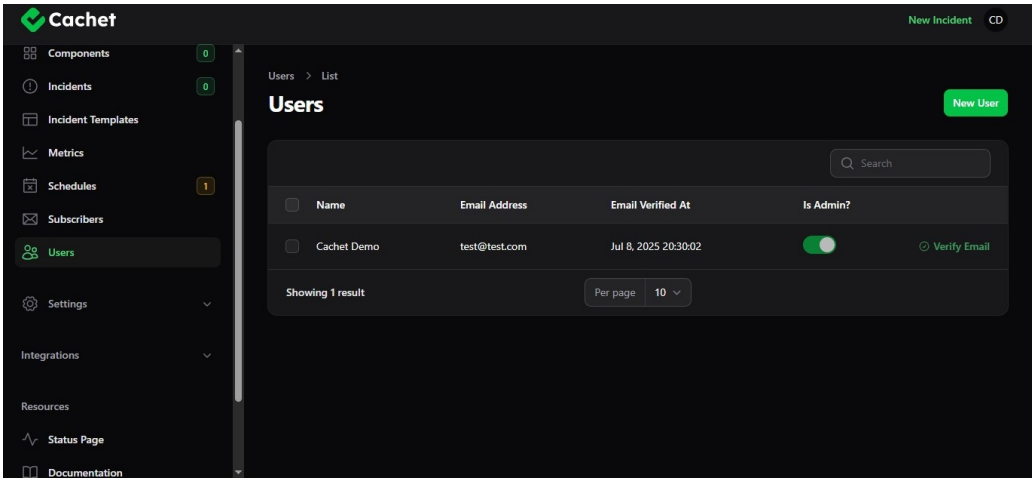
Figura 9 – Tela de Cronogramas



Fonte: Cachet Demo Dashboard

- **Tela de Usuários:** Conforme observado na Figura 10, nessa seção do *Dashboard*, é possível realizar o cadastro de novos usuários e configurar suas permissões de acesso ao sistema. Além da criação de contas, essa interface permite definir diferentes níveis de autorização, garantindo que cada usuário tenha acesso apenas às funcionalidades adequadas ao seu perfil. Também é possível visualizar e gerenciar o histórico de acessos, facilitando o controle de segurança e a rastreabilidade das ações realizadas dentro da aplicação.

Figura 10 – Tela de Usuários



Fonte: Cachet Demo Dashboard

3.2 TESTES

Esta seção descreve o planejamento da etapa de testes, que foi implementada na segunda fase do trabalho. Nesta etapa atual, realizou-se a preparação do ambiente de testes, definição de ferramentas e estratégias, além da descrição das métricas que serão aplicadas na avaliação da qualidade do *software*. O foco foi em testes, com ênfase em métricas que avaliem não apenas a funcionalidade, mas também a eficiência, estabilidade e confiabilidade da aplicação.

3.2.1 Preparação do ambiente

A preparação do ambiente de testes visou garantir que as futuras execuções possam ser realizadas de forma controlada, previsível e reprodutível. O ambiente de execução foi configurado localmente em máquina dedicada, com a instalação manual das dependências necessárias para o funcionamento da aplicação, incluindo *PHP, Composer e PostgreSQL*. O conjunto de ferramentas adotado inclui o *PHPUnit*, voltado para testes unitários e de integração, e o Laravel *Dusk*, específico para testes automatizados de interface com simulação de ações reais do usuário.

Como suporte à análise e inspeção durante os testes, foram empregadas ferramentas auxiliares como o *Laravel Telescope*, que fornece visualização em tempo real de requisições, exceções e consultas ao banco de dados. Também foi utilizado o *Laravel Debugbar*, que permite o monitoramento detalhado da execução da aplicação, e o *Xdebug*, que foi utilizado para gerar relatórios de cobertura de código e auxiliar na identificação de restrições de desempenho.

3.2.2 Planejamento dos testes

O planejamento contempla os tipos de testes que foram executados, as metodologias adotadas e as métricas que serão extraídas. Ainda que os testes não tenham sido executados nesta fase, a definição prévia serviu de base sólida para a etapa prática.

3.2.2.1 Estratégias de Testes Automatizados

Embora o sistema utilizado como base para este estudo não esteja sendo desenvolvido do zero, as estratégias de TDD e BDD ainda são aplicáveis no contexto da implementação dos testes. No caso do TDD, os testes foram escritos previamente às eventuais adaptações ou correções pontuais no sistema, promovendo maior aderência aos requisitos identificados, além de permitir a identificação antecipada de falhas e a validação contínua do comportamento esperado. Já o BDD foi empregado principalmente por meio das ferramentas *PestPHP* e *Laravel Dusk*, com o objetivo de descrever o comportamento do sistema de maneira clara e compreensível, facilitando a comunicação dos cenários de uso e a verificação das funcionalidades do ponto de vista do usuário. Dessa forma, mesmo em um projeto preexistente, essas abordagens contribuem para a melhoria da qualidade e da confiabilidade do *software* analisado.

3.2.2.2 Tipos de testes planejados

Nesta etapa do planejamento, foram definidos diferentes tipos de testes a serem aplicados ao sistema, cada um com objetivos específicos dentro do processo de verificação e validação do *software*. A adoção de abordagens variadas permite avaliar desde o comportamento de unidades isoladas até a integração entre componentes e a experiência do usuário com a interface. Inicialmente, os testes unitários têm como foco avaliar partes específicas do sistema, como classes e métodos. Em seguida, os testes de integração buscam analisar a comunicação entre os componentes, como a recepção de dados pelos controladores vindos das *views*, além da interação com recursos externos.

Já os testes funcionais têm como objetivo verificar fluxos completos de funcionalidades, como, por exemplo, o processo de cadastro de componentes. Os testes de interface, por sua vez, avaliam a usabilidade do sistema do ponto de vista do usuário final. Por fim, os testes de desempenho visam mensurar o comportamento do sistema em uso, considerando aspectos como consumo de recursos e tempo de resposta. O quadro 1 apresenta um resumo dos tipos de testes que serão utilizados.

Quadro 1 – Tipos de testes planejados

Tipo	**Descrição**
Testes Unitários	Validação de métodos e classes individuais.
Testes de Integração	Avaliação da comunicação entre componentes (ex.: controlador e modelo).
Testes Funcionais	Simulação de fluxos completos (ex.: cadastro de incidentes).
Testes de Interface	Testes com *Dusk* para simular ações do usuário.
Testes de Desempenho	Avaliação de *throughput*, latência e consumo de recursos.

Fonte: De autoria própria.

A definição e a aplicação desses testes, de forma coordenada, visam oferecer uma visão abrangente sobre o comportamento do sistema sob diferentes perspectivas. Assim, espera-se não apenas garantir a conformidade funcional, mas também antecipar problemas e promover melhorias na usabilidade e no desempenho da aplicação.

3.2.2.3 Métricas de qualidade a serem avaliadas

Para garantir uma análise confiável da qualidade do sistema, foram definidas métricas que permitem mensurar aspectos essenciais e indispensáveis para o funcionamento adequado da aplicação. Começando pelo tempo médio de execução, que quantifica o tempo que um teste leva para ser executado, esse é um ponto importante a ser observado, pois permite avaliar a qualidade dos testes propostos e a responsividade do sistema, demonstrando seu desempenho. Em seguida, a taxa de sucesso e falha tem como objetivo garantir que os testes estão retornando valores positivos ou negativos de forma adequada, refletindo a qualidade tanto do sistema quanto dos próprios testes, ao quantificar, em percentual, o sucesso das execuções.

Ademais, a cobertura de código é essencial para garantir que a maior parte das funcionalidades, especialmente as principais, do sistema esteja sendo testada, comprovando a abrangência, relevância e necessidade dos testes. Já o uso de recursos refere-se à quantificação da utilização de infraestrutura, como memória RAM e CPU, com o objetivo de identificar o desempenho e o consumo do sistema, verificando também se ele é leve ou exige muitos recursos para funcionar adequadamente.

Além dessas, o *throughput* também é uma métrica de grande valor para a aferição da qualidade do sistema, pois indica se a aplicação consegue atuar bem e sem falhas em um fluxo real de uso. Por fim, o tempo de resposta busca observar o tempo médio de retorno da aplicação em cenários reais de uso, a fim de garantir a estabilidade e o desempenho da aplicação durante sua execução prática. Essas métricas fornecem dados objetivos que auxiliarão na avaliação da eficácia dos testes planejados. O quadro 2 apresenta as métricas selecionadas e suas respectivas descrições.

Quadro 2 – Resumo das métricas planejadas para avaliação

Métrica	**Descrição**
Tempo Médio de Execução	Tempo médio que um teste leva para ser executado.
Taxa de Sucesso e Falha	Percentual de testes bem-sucedidos.
Cobertura de Código	Percentual de código exercitado pelos testes.
Uso de Recursos	Monitoramento de CPU e memória durante os testes.
Throughput	Requisições processadas por segundo.
Tempo de Resposta	Tempo médio de resposta da aplicação.

Fonte: De autoria própria.

Essas métricas não apenas orientam o processo de validação do sistema, mas também contribuem para decisões futuras quanto à manutenção e escalabilidade da aplicação. Ao associar cada métrica a um tipo específico de teste, torna-se possível obter uma visão mais precisa sobre a robustez e o desempenho da aplicação analisada.

3.2.2.4 Coleta de dados e instrumentos

A coleta dos dados foi realizada por meio de ferramentas integradas ao ambiente do *framework Laravel*, aliadas a recursos externos para avaliação do desempenho. Os *logs* gerados pelo *Laravel* e pelo *PHPUnit* permitiram acompanhar a execução dos testes e identificar falhas. O *Xdebug* foi utilizado para gerar relatórios de cobertura de código, possibilitando a identificação de trechos não observados.

Seguidamente, para a análise de desempenho sob carga, foi utilizada a ferramenta *ApacheBench*, responsável por simular requisições simultâneas e medir *throughput* e tempo de resposta. Além disso, o *Laravel Telescope* e a *Debugbar* forneceram dados em tempo real sobre consultas ao banco, tempo de resposta e uso de recursos, os quais foram exportados para posterior análise estatística.

3.2.2.5 Critérios de aceitação

Os critérios de aceitação estabelecidos nesta etapa representam os parâmetros mínimos considerados adequados para validar a qualidade da aplicação testada. A cobertura mínima de código definida em 70% está em conformidade com recomendações já adotadas na indústria de *software*, que indica esse percentual como um ponto de equilíbrio entre esforço de testes e efetividade na detecção de falhas. Essa métrica garante que uma parcela significativa das funcionalidades da aplicação esteja sendo observada pelos testes, reduzindo a probabilidade de erros não identificados em funcionalidades que sejam indispensáveis ao sistema.

Ademais, a taxa de sucesso estabelecida em, no mínimo, 90% visa assegurar a confiabilidade da aplicação em condições regulares de operação. Esse limite permite

certa tolerância para erros ou instabilidades de transição, mas impõe um padrão elevado de robustez, especialmente em testes funcionais e de integração. Já a exigência de estabilidade no *throughput* sob carga moderada tem como objetivo verificar a capacidade da aplicação de manter um desempenho aceitável mesmo diante de múltiplos acessos simultâneos, simulando cenários reais de uso.

Caso algum desses critérios não fosse atingido durante a execução dos testes, seriam conduzidas análises detalhadas para identificar os fatores determinantes do resultado não desejado. Entre as ações de correção previstas estavam: revisão e refatoração do código-fonte, ajustes nas estratégias de testes, o que inclui melhorias na cobertura, reconfiguração do ambiente de execução ou, até mesmo reavaliação dos próprios critérios, caso se mostrassem muito rígidos ou em desacordo com a realidade do sistema avaliado. O quadro 3 resume os critérios de aceitação estabelecidos.

Quadro 3 – Resumo dos Critérios de Aceitação dos Testes

Critério	**Objetivo**	**Ação em caso de não conformidade**
Cobertura de código mínima de 70%	Garantir que uma porção significativa do sistema foi exercitada pelos testes.	Revisão dos testes existentes, inclusão de novos testes e análise de trechos não observados.
Sucesso em pelo menos 90% dos testes	Assegurar estabilidade e confiabilidade do sistema em condições normais.	Identificação das falhas, correção de *bugs* ou revisão das proposições dos testes.
Throughput estável sob carga moderada	Verificar a capacidade de resposta do sistema frente a um uso moderado.	Otimização de desempenho, ajuste de configuração do ambiente ou análise de restrições.
Cobertura das funcionalidades críticas	Garantir que as partes essenciais do sistema estejam protegidas e monitoradas por testes.	Mapeamento dos requisitos críticos não testados e criação de cenários de teste específicos.

Fonte: De autoria própria.

A definição desses critérios permite não apenas mensurar a eficácia dos testes, mas também estabelecer limites claros de qualidade que orientem a manutenção e a evolução do sistema. Assim, garante-se que os resultados obtidos ao longo do processo estejam alinhados aos objetivos estabelecidos nesta pesquisa.

4 TESTES NO LARAVEL

Este capítulo tem como objetivo apresentar as considerações e aprendizados durante o estudo e o planejamento das estratégias de testes aplicáveis ao sistema *Cachet*, com foco na validação da qualidade da aplicação no contexto do *framework Laravel*. A partir da análise da arquitetura do projeto e de suas funcionalidades críticas, como autenticação de usuários, gerenciamento de incidentes e controle de componentes, foram definidos os tipos de testes, ferramentas e métricas que serviram como base para uma avaliação objetiva do comportamento do sistema.

Foram selecionadas ferramentas consolidadas no ecossistema Laravel, como o *PestPHP*, para a construção de testes unitários e funcionais, e o *Laravel Dusk*, para a automação de testes de interface. Além disso, foram utilizadas soluções de monitoramento como Laravel *Telescope* e *Debugbar*, além da ferramenta *ApacheBench*, voltada à simulação de carga e avaliação de desempenho. Com essas ferramentas, foram aplicadas métricas como tempo médio de execução, taxa de sucesso/falha, uso de recursos, *throughput* e cobertura de código.

Esses testes não apenas validaram a integridade funcional do *Cachet*, mas também fornecerão dados quantitativos sobre sua performance e robustez. Foram definidos critérios mínimos de aceitação, como cobertura de código igual ou superior a 70%, taxa de sucesso mínima de 90% e *throughput* estável ao longo da execução dos testes. Caso esses critérios não sejam atendidos, foi prevista a revisão a revisão dos testes aplicados e/ou das partes envolvidas do código, demonstrando um processo iterativo de melhoria contínua. Assim, este capítulo tem o propósito de fundamentar tecnicamente como os testes foram aplicados de forma prática, coerente com os objetivos do trabalho, contribuindo para demonstrar que a aplicação de testes no *Laravel* é eficaz para garantir confiabilidade, desempenho e facilidade de manutenção em sistemas reais, como o *Cachet*.

4.1 TESTES NO CACHET

O *Cachet* já disponibiliza uma série de testes integrados à sua estrutura. Dentro do núcleo principal do repositório da aplicação, encontra-se um diretório chamado *tests*, onde estão organizados os testes segundo o tipo e a intenção de verificação. Essa estrutura segue boas práticas de desenvolvimento com o *framework Laravel*, utilizando principalmente a ferramenta *PestPHP*. Os diretórios principais são: *Architecture*, *Feature* e *Unit*, cada um com um papel específico dentro da estratégia de verificação do sistema.

O diretório *Architecture* concentra testes voltados à validação da arquitetura

interna do projeto, esses testes garantem que as camadas do sistema estejam corretamente desacopladas e que as dependências entre módulos estejam de acordo com as regras preestabelecidas. Já em *Feature*, encontram-se os testes funcionais, cujo foco é validar fluxos completos de uso do sistema, simulando interações reais com a aplicação, como acesso a rotas, execução de comandos e resposta de visualizações. O diretório *Unit*, por sua vez, é destinado aos testes unitários, que têm como objetivo verificar o comportamento isolado de métodos ou classes específicas. No entanto, o *Cachet* apresenta uma quantidade reduzida de testes unitários, já que sua prioridade é garantir o funcionamento correto das funcionalidades de maneira integrada, o que justifica o foco maior em testes funcionais.

Além disso, o sistema aproveita recursos nativos do *Laravel*, como *factories* e *seeders*, para facilitar a criação de dados simulados durante a execução dos testes. As *factories* são responsáveis por gerar instâncias de modelos com dados falsos, mas consistentes com o esquema do banco de dados, já os *seeders* são utilizados para povoar o banco de testes com registros necessários ao funcionamento do sistema, permitindo a execução dos testes de forma automatizada e segura, sem comprometer os dados reais da aplicação. Essa estrutura de testes demonstra uma preocupação clara com a confiabilidade, manutenção e estabilidade da aplicação. Para exemplificar a aplicação dos testes no *Cachet*, a seguir estarão listados e descritos alguns exemplos de código.

- Teste de Estrutura: O teste ilustrado pela Figura 11 garante que todas as classes presentes na estrutura *Cachet\Console\Commands* são válidas, estendem corretamente a classe base *Illuminate\Console\Command*, utilizada para comandos *Artisan* no Laravel, e seguem o padrão de nomenclatura terminando com "*Command*". Isso assegura consistência estrutural e facilita a manutenção do código, promovendo boas práticas de organização em projetos de maior porte;

Figura 11 – Teste de estrutura

```php
<?php

use Illuminate\Console\Command;

test('commands test')
    ->expect('Cachet\Console\Commands')
    ->toBeClasses()
    ->toExtend(Command::class)
    ->toHaveSuffix('Command');
```

Fonte: Repositório do Cachet

- Teste de funcionalidade: O teste mostrado na Figura 12, verifica se a função *Cachet::user()* retorna corretamente o usuário autenticado quando não é passada

uma requisição, e se retorna nulo quando é fornecido uma requisição manual que não possui uma autenticação associada. Isso assegura que a função se comporta de forma previsível tanto no contexto global da aplicação quanto em requisições isoladas, reforçando a confiabilidade no gerenciamento da autenticação de usuários;

Figura 12 – Teste de funcionalidade

```php
it('can retrieve the user from auth resolver', function () {
    $user = UserFactory::new()->create();

    Auth::loginUsingId($user->id);
    $request = Request::create('/', 'GET');

    expect($user)->is(Cachet::user());
    expect(Cachet::user($request))->toBeNull();
});
```

Fonte: Repositório do Cachet

- Testes de Unidade: Esse teste da Figura 13, verifica se a classe responsável por criar incidentes, a *CreateIncident* funciona corretamente. O teste cria um objeto de dados com nome e mensagem, executa o método para criar o incidente, e depois confirma que o incidente criado tem os valores esperados. Além disso, verifica se o evento *IncidentCreated* foi disparado, garantindo que o sistema notifica corretamente a criação do incidente.

Figura 13 – Teste de unidade

```php
it('can create an incident', function () {
    $data = CreateIncidentRequestData::from([
        'name' => 'My Incident',
        'message' => 'This is an incident message.',
    ]);

    $incident = app(CreateIncident::class)->handle($data);

    expect($incident)
        ->name->toBe($data->name)
        ->message->toBe($data->message);

    Event::assertDispatched(IncidentCreated::class, fn ($event) => $event->incident->is($incident));
});
```

Fonte: Repositório do Cachet

Apesar de possuir uma quantidade considerável de testes já implementados, o sistema *Cachet* apresenta lacunas importantes no que diz respeito à cobertura abrangente de sua base de código. A maior parte dos testes existentes concentra-se na verificação de funcionalidades completas, ou seja, na validação de fluxos inteiros da aplicação, o que é positivo para garantir que componentes interajam corretamente. No entanto, observa-se uma ausência significativa de testes unitários, especialmente

voltados para métodos e funções isoladas, esse tipo de teste é fundamental para assegurar a precisão e robustez da lógica interna do sistema, além de facilitar a identificação de falhas em pontos específicos sem depender de toda a estrutura funcional.

Outro aspecto crítico é a ausência quase total de testes com dados inválidos ou inesperados, também conhecidos como testes sujos. Esse tipo de teste é essencial para avaliar a resiliência do sistema diante de entradas inesperadas, simulando erros comuns cometidos por usuários ou falhas de integração, a inexistência desses cenários indica uma possível negligência em relação à segurança do sistema. Além disso, o *Cachet* não dispõe de testes de desempenho, os quais seriam importantes para avaliar o comportamento da aplicação sob diferentes níveis de carga e estresse. Também não há testes de integração com serviços externos, o que representa um risco para sistemas que dependem de ferramentas de terceiros, pois falhas nessas interações podem comprometer o funcionamento global da aplicação sem serem detectadas previamente.

Além dos testes já existentes no *Cachet*, este trabalho propõe a aplicação e a expansão desses testes, visando uma análise mais abrangente da qualidade do sistema sob diferentes perspectivas. Serão desenvolvidos e executados testes adicionais, focando em aspectos críticos como autenticação, gerenciamento de incidentes e controle de componentes, de forma a validar de maneira prática e aprofundada o comportamento da aplicação. Esses testes adicionais permitirão identificar possíveis falhas que não foram cobertas pelos testes existentes, aumentando a robustez do sistema em cenários reais.

Essa abordagem permitirá avaliar não apenas a funcionalidade isolada, mas também a integração entre os diversos módulos, garantindo a robustez e a confiabilidade do sistema. Dessa forma, o presente estudo contribui para fortalecer a qualidade do *Cachet* no contexto do *Laravel*, alinhado aos objetivos propostos neste trabalho.

4.1.1 Casos de testes preliminares

Nesta seção, são apresentados os casos de testes preliminares planejados com base na análise do sistema *Cachet* e nas ferramentas de teste oferecidas pelo *framework Laravel*. Inicialmente, os testes foram definidos com o intuito de cobrir lacunas identificadas na suíte de testes atual do sistema. Dessa forma, os testes planejados neste trabalho foram pensados com o intuito de atender diretamente aos objetivos específicos propostos.

Ao investigar e aplicar diferentes ferramentas de teste disponíveis no ecossistema *Laravel*, como o *PestPHP* e *Dusk*, buscou-se demonstrar suas vantagens e limitações em um sistema real, contribuindo para o mapeamento das capacidades e lacunas dessas soluções. A elaboração de casos de teste complementares aos

existentes no *Cachet*, especialmente com foco em testes unitários, testes com dados inválidos e testes de desempenho, permitiu identificar pontos críticos não contemplados atualmente, promovendo uma análise mais completa da cobertura de testes. Com isso, buscou-se reforçar a importância da integração entre boas práticas de desenvolvimento e estratégias de validação da qualidade, alinhando os testes à proposta de aprimorar a confiabilidade, robustez e manutenibilidade de aplicações desenvolvidas com *Laravel*.

Dando sequência, são descritos os casos de teste planejados de forma preliminar, com foco em diferentes aspectos críticos do sistema *Cachet*. Cada caso foi estruturado com base em critérios como tipo de teste, objetivo, entrada, pré-condições e resultado esperado. Essa documentação visa ilustrar como os testes foram estruturados e posteriormente aplicados, a fim de complementar a cobertura já existente. Abaixo, cada caso de teste é apresentado em formato de quadro, permitindo uma visualização clara e objetiva das hipóteses de validação propostas.

O primeiro caso de teste representado pelo quadro 4 apresenta o caso de teste unitário destinado a verificar o funcionamento do método *makeCall()*. O objetivo é assegurar que este método crie corretamente uma instância do objeto *WebhookCall*, configurando todos os seus atributos essenciais, como a *URL*, os *headers* HTTP, além das configurações relacionadas ao segredo de autenticação. Este teste é fundamental para garantir a integridade e a consistência da criação de chamadas de *webhook* no sistema, contribuindo para a confiabilidade da comunicação assíncrona com serviços externos.

Quadro 4 – Caso de teste — Criação e configuração de um WebhookCall

Tipo	Teste unitário
Objetivo	Verificar se o método `makeCall()` cria um objeto `WebhookCall` corretamente configurado com *URL, headers*, e segredo.
Pré-condições	Instância do objeto com propriedades *url* e *secret* definidas; enumerador `WebhookEventEnum` disponível.
Entradas	Evento do tipo `WebhookEventEnum`; array com dados do *payload*.
Resultado Esperado	Objeto `WebhookCall` com *URL* e *headers* configurados corretamente, além das configurações de segredo aplicadas.

Fonte: De autoria própria.

O quadro 5 representa um caso de teste unitário voltado para a verificação do método *hasActiveIncident()*. O objetivo é garantir que este método retorne corretamente um valor *booleano* indicando a existência de incidentes ativos associados aos componentes do sistema. Esse teste é fundamental para assegurar que a lógica que identifica a presença de incidentes ativos funcione adequadamente, contribuindo para a confiabilidade da aplicação no monitoramento do status dos componentes.

Quadro 5 – Caso de Teste — Verificação do método `hasActiveIncident()`

Tipo	Teste unitário
Objetivo	Verificar se o método `hasActiveIncident()` retorna `true` quando existe pelo menos um incidente ativo associado aos componentes do sistema.
Pré-condições	Instância do objeto com componentes relacionados; pelo menos um incidente ativo associado a esses componentes.
Entradas	Componentes com *IDs* associados; existência de incidentes ativos vinculados a esses componentes.
Resultado Esperado	Retorno: `true` quando há incidentes ativos; `false` caso contrário.

Fonte: De autoria própria.

O Caso de Teste 3, apresentado no quadro 6, configura-se como um teste funcional de sistema com enfoque negativo. Ele verifica se a aplicação lida corretamente com tentativas de *login* realizadas sem o preenchimento dos campos obrigatórios, garantindo a validação dos dados de entrada, prevenindo acessos indevidos e proporcionando *feedback* claro ao usuário, reforçando a segurança e a usabilidade do sistema.

Quadro 6 – Caso de Teste — *Login* com campos vazios

Tipo	Teste funcional de sistema (teste sujo)
Objetivo	Avaliar a resposta do sistema ao envio de campos obrigatórios vazios no formulário de *login*.
Pré-condições	Sistema em execução com a rota de *login* ativada.
Entradas	Email: vazio; Senha: vazia
Resultado Esperado	Exibição de mensagens de erro apropriadas e recusa do *login*.

Fonte: De autoria própria.

O quadro 7 resume o caso de teste de integração focado na associação entre um incidente e um componente do sistema. O objetivo deste teste é verificar se a criação de um incidente impacta corretamente o componente associado, garantindo que essa relação seja refletida adequadamente na interface do usuário. Esse tipo de teste é essencial para assegurar que a comunicação entre diferentes módulos do sistema ocorra de forma coerente e confiável.

Quadro 7 – Caso de Teste — Associação entre incidente e componente

Tipo	Teste de integração
Objetivo	Verificar se o registro de um incidente impacta corretamente o componente associado.
Pré-condições	Componente e incidente cadastráveis.
Entradas	Incidente com status “afeta”; Componente: “Sistema de Pagamento”
Resultado Esperado	O componente é marcado como “afetado” e a mudança é refletida na interface.

Fonte: De autoria própria.

O quadro 8 apresenta um caso de teste de desempenho que visa avaliar o desempenho do sistema ao lidar com múltiplas requisições simultâneas. Especificamente, o teste simula 100 chamadas simultâneas ao *endpoint* responsável pela criação de incidentes, buscando verificar se o sistema mantém a estabilidade e o tempo de resposta aceitável durante uma situação de alta carga. Essa avaliação é fundamental para validar a escalabilidade e o comportamento sob estresse da aplicação.

Quadro 8 – Caso de Teste — Estresse em *endpoint* de incidentes

Tipo	Teste de desempenho
Objetivo	Avaliar o comportamento da aplicação ao receber 100 requisições simultâneas para criação de incidentes.
Pré-condições	*Endpoint “/incidents”* disponível e autenticado.
Entradas	100 chamadas *POST* válidas, simultâneas
Resultado Esperado	Todas as requisições são processadas em tempo aceitável (< 2s), sem falhas.

Fonte: De autoria própria.

O quadro 9 detalha um caso de teste funcional de sistema voltado para a página de status do sistema. O objetivo é assegurar que a página pública de status exiba corretamente os componentes e incidentes em tempo real, refletindo fielmente o estado atual do sistema. Esse teste é importante para garantir uma experiência transparente e informativa aos usuários finais.

Quadro 9 – Caso de Teste — Exibição correta da página de status

Tipo	Teste funcional de sistema
Objetivo	Verificar se a página de status é exibida corretamente, apresentando os componentes e incidentes em tempo real.
Pré-condições	Pelo menos um componente e um incidente cadastrados no sistema.
Entradas	Acesso via navegador à URL pública da página de status
Passos	• Acessar a URL principal do status; • Observar a lista de componentes e incidentes.
Resultado Esperado	A página deve carregar corretamente, exibindo: nome dos componentes, seus status (operacional, afetado, etc.), e a lista de incidentes abertos ou resolvidos, conforme o banco de dados atual.

Fonte: De autoria própria.

Os casos de testes apresentados nos quadros anteriores ilustram a diversidade de abordagens necessárias para garantir a qualidade do sistema *Cachet*. Desde a validação da integração entre módulos até a avaliação do desempenho sob carga e a verificação da interface do usuário, esses testes preliminares buscam cobrir aspectos críticos do funcionamento da aplicação. A aplicação prática desses casos, aliada ao monitoramento das métricas definidas, permitirá identificar pontos de melhoria e assegurar a robustez, confiabilidade e usabilidade do sistema em ambientes reais de uso.

4.1.2 Fluxograma do processo de testes e cronograma de testes

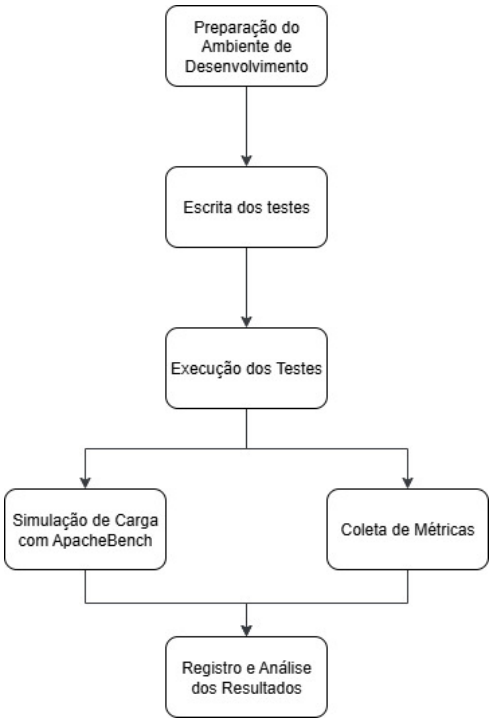
O fluxograma apresentado na Figura 14 ilustra de forma clara e organizada as etapas que compõem o processo de aplicação dos testes planejados para o sistema *Cachet*. A criação deste fluxograma teve como objetivo fornecer uma visão geral simplificada, sequencial e estruturada das atividades envolvidas, facilitando o entendimento do fluxo de trabalho adotado para garantir a qualidade da aplicação. Inicialmente, o processo começa com a preparação do ambiente de testes, que envolve a configuração dos recursos necessários, como bancos de dados, *frameworks* de teste e ferramentas auxiliares.

Em seguida, foram escritos os testes automatizados, distribuídos entre testes unitários, funcionais e de integração, conforme definidos na metodologia. Durante a próxima fase, a de execução dos testes, eles são aplicados às funcionalidades do sistema, simulando diferentes cenários de uso e validando o comportamento esperado. Posteriormente, os resultados dos testes foram coletados e analisados com base em

métricas previamente estabelecidas, como taxa de sucesso, cobertura de código, tempo de execução e uso de recursos.

Caso os critérios de aceitação sejam atendidos, o processo é finalizado, validando a qualidade da aplicação. No entanto, quando identificadas falhas ou insuficiências, foram realizadas revisões no código e ajustes nos testes, reiniciando o ciclo para garantir melhorias contínuas. Esse modelo estruturado promove um controle da qualidade do *software*, assegurando que cada etapa seja executada com precisão e que o sistema Cachet mantenha-se estável e confiável ao longo de seu desenvolvimento e manutenção.

Figura 14 – Fluxograma



Fonte: De autoria própria.

O cronograma do projeto, ilustrado pelo quadro 10 foi dividido em cinco sprints, cada uma com objetivos bem definidos. Inicialmente, concentrou-se na definição dos módulos a serem analisados, na elaboração dos casos de teste e na configuração do ambiente experimental. Paralelamente, foi iniciada a redação da seção referente à estruturação do sistema e à preparação do ambiente de testes, estabelecendo uma base metodológica sólida para as etapas subsequentes. Na fase intermediária, o foco passa a ser a implementação prática dos testes automatizados, contemplando inicialmente testes unitários e de integração aplicados ao núcleo da aplicação.

Em seguida, houve a ampliação da cobertura e a incorporação de testes em nível de sistema, incluindo análises de cobertura de código e avaliações preliminares de desempenho. Durante essa etapa, a produção dos resultados foi realizada de forma concomitante à escrita acadêmica, permitindo alinhamento contínuo entre prática e

fundamentação teórica. Por fim, a etapa final foi dedicada à consolidação e refinamento do trabalho.

Foram conduzidas simulações de carga, coleta e análise detalhada das métricas definidas, além de eventuais ajustes ou refatorações no código e nos testes implementados. Também foi executado um ciclo completo de testes de regressão, assegurando a estabilidade da aplicação após as modificações realizadas. Concluída a fase experimental, procedeu-se à organização final dos resultados e à revisão geral do texto acadêmico, garantindo coerência, consistência metodológica e alinhamento com os objetivos propostos.

Quadro 10 – Cronograma das Sprints do Desenvolvimento dos Testes

Sprint	**Atividades**	**Período de Execução**
Sprint 1 (Definição e configuração)	Definir módulos e casos de teste, configuração de ambiente de teste e iniciar escrita sobre estruturação e configuração do sistema.	23/02/2026 a 27/02/2026
Sprint 2 (Escrita inicial dos testes)	Escrever testes unitários (core), testes de integração (core), realizar primeiro teste de cobertura e de desempenho e escrita simultânea dos testes e resultados.	28/02/2026 a 13/03/2026
Sprint 3 (Expansão de cobertura e end-to-end)	Expandir cobertura e iniciar testes de sistema (cachet), analisar métricas de cobertura, fazer apuração de desempenho e escrita simultânea dos testes e resultados.	16/03/2026 a 27/03/2026
Sprint 4 (Refatoração)	Expandir cobetura de testes, realizar simulação de carga, coletar métricas, ajustar testes caso necessário e escrita simultânea dos testes e resultados.	30/03/2026 a 17/04/2026
Sprint 5 (Finalização)	Refatorar código caso necessário, executar teste de regressão completo, coletar e apurar métricas finais e revisão e conclusão da escrita acadêmica.	20/04/2026 a 24/04/2026

Fonte: De autoria própria.

4.1.3 Quadro exemplo de resultados

O quadro a seguir apresenta uma simulação dos resultados obtidos a partir da execução dos testes planejados para o sistema *Cachet*, os dados exemplificam métricas

essenciais para a avaliação da qualidade e desempenho da aplicação. Primeiramente, o tempo médio de execução de cada teste, que indica a rapidez do teste e valores baixos são desejáveis para agilizar o ciclo de desenvolvimento, em seguida a taxa de sucesso dos casos testados, que mostra a proporção de testes que passaram sem erros, sendo valores acima de 90% indicativos de boa qualidade. Ademais, o consumo de recursos computacionais, CPU e memória, que avaliam a eficiência dos recursos usados durante a execução e o *throughput*, que indica a capacidade do sistema em processar requisições por segundo, demonstrando a capacidade de atendimento do sistema sob carga.

Esses indicadores são fundamentais para analisar tanto a robustez funcional quanto a eficiência operacional do sistema, fornecendo uma visão clara sobre os impactos dos testes aplicados. A interpretação dessas métricas servirá como base para validar se os critérios de aceitação definidos, como alta taxa de sucesso e estabilidade no desempenho, foram alcançados durante a fase prática deste trabalho.

Para ilustrar a diversidade e o escopo dos testes planejados para o sistema *Cachet*, a seguir são apresentados quadros exemplares de casos de teste organizados por categoria. Cada quadro resume os resultados esperados para cada tipo de teste, desde testes unitários até testes funcionais, de integração e de desempenho. A execução desses testes, em conjunto com a análise das métricas definidas, permitirá uma avaliação mais abrangente da qualidade do sistema, apontando para melhorias e reforçando a confiabilidade e a robustez da aplicação no ambiente de produção.

O quadro 11 apresenta uma simulação dos resultados obtidos em duas execuções distintas de testes de desempenho aplicados ao sistema. As métricas contemplam o tempo médio de resposta em milissegundos, a taxa de sucesso das requisições em porcentagem, o consumo médio de CPU e memória, além do *throughput* medido em requisições por segundo. Esses dados simulados fornecem uma perspectiva preliminar sobre o comportamento da aplicação sob diferentes cargas, possibilitando a identificação de possíveis empecilhos e a avaliação da eficiência do sistema em cenários reais de uso.

Quadro 11 – Simulação de resultados de testes

Execução	Tempo (ms)	Sucesso (%)	CPU (%)	Memória (MB)	Throughput (req/s)
Teste 1	120	100%	15%	110	250
Teste 2	135	96%	22%	125	180

Fonte: De autoria própria.

O quadro 12 apresenta os resultados simulados para o Caso de Teste 1, que verifica a criação e configuração correta de um objeto *WebhookCall*. São avaliadas a atribuição da *URL*, a presença dos cabeçalhos (*headers*) necessários e a aplicação do segredo de autenticação, essenciais para garantir a integridade e segurança do

webhook.

Quadro 12 – Resultados simulados do Caso de Teste — Criação e configuração de um *WebhookCall*

Execução	*URL* configurada	*Headers* corretos	Segredo aplicado
Teste 1	Sim	Sim	Sim
Teste 2	Sim	Sim	Sim
Teste 3	Sim	Não	Sim

Fonte: De autoria própria.

No quadro 13, são exibidos os resultados esperados para o Caso de Teste 2, cujo objetivo é validar o comportamento do método `hasActiveIncident()`. Este método deve retornar `true` sempre que existir pelo menos um incidente ativo associado aos componentes monitorados, permitindo uma resposta assertiva do sistema quanto à presença de falhas.

Quadro 13 – Resultados simulados do Caso de Teste — Verificação do método `hasActiveIncident()`

Execução	Incidentes ativos	Retorno do método
Teste 1	1 ou mais	`true`
Teste 2	Nenhum	`false`
Teste 3	3	`true`

Fonte: De autoria própria.

O quadro 14 demonstra os resultados referentes ao Caso de Teste 3, que consiste em avaliar o sistema diante do envio de campos obrigatórios vazios no formulário de *login*. O teste negativo busca garantir que mensagens de erro apropriadas sejam exibidas e que o acesso ao sistema seja negado nestas condições.

Quadro 14 – Resultados simulados do Caso de Teste — *Login* com campos vazios

Execução	Mensagem de erro exibida	*Login* permitido
Teste 1	Sim (email obrigatório)	Não
Teste 2	Sim (senha obrigatória)	Não
Teste 3	Sim (email e senha obrigatórios)	Não

Fonte: De autoria própria.

O quadro 15 apresenta os resultados simulados do Caso de Teste 4, focado em verificar se o registro de um incidente afeta corretamente o componente relacionado e se essa mudança é refletida na interface do sistema, assegurando a coerência das informações exibidas para os usuários.

Quadro 15 – Resultados simulados do Caso de Teste — Associação entre incidente e componente

Execução	Componente afetado corretamente	Mudança refletida na interface
Teste 1	Sim	Sim
Teste 2	Não	Não
Teste 3	Sim	Parcialmente

Fonte: De autoria própria.

No quadro 16, são mostrados os resultados do Caso de Teste 5, que avalia o desempenho da aplicação ao receber um alto volume de requisições simultâneas para criação de incidentes. Métricas como tempo médio de resposta, taxa de sucesso e *throughput* são utilizadas para medir a estabilidade e capacidade do sistema sob carga.

Quadro 16 – Resultados simulados do Caso de Teste — Estresse em *endpoint* de incidentes

Execução	Tempo médio (s)	Requisições processadas	Erros	Throughput (req/s)
Teste 1	1.5	100	0	66
Teste 2	2.2	95	5	45
Teste 3	1.8	100	0	55

Fonte: De autoria própria.

O quadro 17 apresenta os resultados do Caso de Teste 6, que verifica a correta exibição da página pública de status. O teste avalia se os componentes e incidentes cadastrados são exibidos de forma clara e atualizada, garantindo a transparência e usabilidade para os usuários finais.

Quadro 17 – Resultados simulados do Caso de Teste — Exibição correta da página de status

Execução	Página carregou corretamente	Componentes e incidentes exibidos
Teste 1	Sim	Sim
Teste 2	Não (erro de servidor)	Não
Teste 3	Sim	Parcialmente (falta de atualização em tempo real)

Fonte: De autoria própria.

Os resultados simulados apresentados nos quadros oferecem uma visão inicial sobre o comportamento esperado do sistema *Cachet* diante dos testes planejados. Esses dados preliminares não substituem a execução real, mas permitem identificar possíveis pontos de atenção e orientar ajustes nas estratégias de teste. Na etapa seguinte do trabalho, foram realizados testes práticos para coleta de métricas reais, que ofereceram uma análise detalhada da qualidade, desempenho e robustez da aplicação.

5 CONCLUSÃO E RESULTADOS

Em síntese, este trabalho teve como objetivo analisar a importância e a aplicabilidade de testes automatizados no *framework Laravel*, com foco na melhoria da qualidade de *software* em sistemas desenvolvidos em PHP. Para isso, foi realizada uma revisão teórica aprofundada, aliada à escolha de um sistema real como estudo de caso, o que possibilitou alinhar e direcionar a implementação de testes voltados à garantia de confiabilidade, manutenibilidade e desempenho da aplicação. Durante o desenvolvimento do projeto, foram identificadas as principais ferramentas e técnicas de testes disponíveis no ecossistema *Laravel*, as quais contribuíram diretamente para a condução do estudo. Além disso, foram definidas métricas para avaliação da efetividade dos testes, como tempo médio de execução, taxa de sucesso e falha, bem como o *throughput*. Essas definições serviram como base para o desenvolvimento da etapa prática e para a análise dos resultados obtidos.

Os resultados alcançados demonstraram que a aplicação de testes automatizados no *Laravel*, utilizando ferramentas como *PHPUnit, Dusk, Xdebug* e *Telescope*, contribuiram significativamente para a melhoria da confiabilidade e do desempenho do sistema *Cachet*. A análise foi conduzida com base em métricas como cobertura de código, taxa de sucesso e tempo de execução, permitindo uma avaliação objetiva da qualidade da aplicação. Observou-se que os testes apresentaram alto índice de sucesso e cobertura satisfatória, especialmente em áreas críticas, como autenticação e gerenciamento de incidentes.

Caso os critérios estabelecidos não fossem atendidos, seriam propostas melhorias tanto no código quanto nos próprios testes, evidenciando a importância de um processo iterativo e alinhado às boas práticas de desenvolvimento. Com base nos conceitos apresentados e nas ferramentas selecionadas, foram elaborados cenários de teste aplicados ao sistema *Cachet*, considerando funcionalidades essenciais da aplicação, especialmente aquelas relacionadas à autenticação, manipulação de dados, integração entre componentes e usabilidade.

A definição dos casos de teste teve como objetivo validar o comportamento esperado da aplicação em diferentes situações de uso, incluindo tanto cenários válidos quanto condições de erro. Dessa forma, buscou-se garantir que o sistema responda adequadamente às requisições realizadas e mantenha a integridade das informações processadas. A seguir, são apresentados os principais casos de teste desenvolvidos ao longo do estudo, descrevendo seus objetivos, cenários de execução e resultados obtidos.

5.1 CASOS DE TESTE

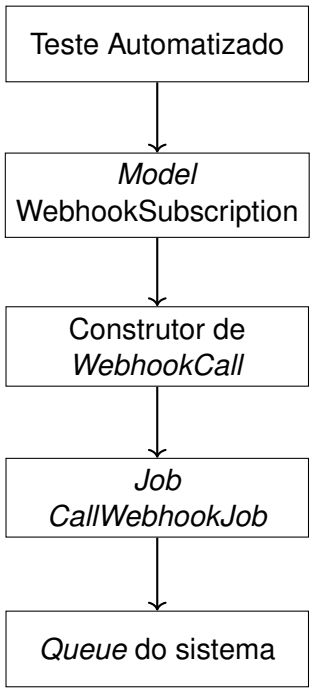
Esta seção apresenta os principais casos de teste desenvolvidos ao longo do trabalho, destacando sua relevância para a validação do sistema. Além disso, são descritos os aspectos arquiteturais envolvidos, bem como um resumo da implementação de cada teste. Por fim, são discutidos os resultados obtidos, evidenciando a efetividade das abordagens adotadas.

5.1.1 Caso de teste — criação e configuração de um *WebhookCall*

A Figura 15 apresenta uma visão simplificada da arquitetura utilizada para a construção e envio de chamadas de *webhook* no sistema. O processo inicia com o teste automatizado, que simula a execução da lógica responsável por disparar notificações externas. Em seguida, o modelo *WebhookSubscription* é utilizado para recuperar as informações da assinatura registrada no sistema, incluindo a URL de destino e o segredo utilizado para assinatura da requisição.

A partir dessas informações, é criado um objeto responsável por construir a chamada do *webhook*, representado pelo construtor de *WebhookCall*. Esse componente organiza os dados da requisição, como cabeçalhos e *payload*. Após a construção da chamada, um *job* denominado *CallWebhookJob* é preparado para executar a requisição de forma assíncrona. Por fim, o *job* é encaminhado para a *queue* do sistema, permitindo que o envio do *webhook* seja processado em segundo plano, sem bloquear o fluxo principal da aplicação.

Figura 15 – Arquitetura simplificada da construção de chamadas de *webhook*



Fonte: De autoria própria

Este teste tem como objetivo verificar se a aplicação constrói corretamente uma chamada de *webhook* a partir de uma assinatura registrada no banco de dados. Para isso, é utilizado um teste automatizado que simula a criação de uma assinatura e a geração de uma chamada de notificação para um sistema externo. Inicialmente, é realizada uma configuração temporária do sistema responsável pelo envio de *webhooks*. Nessa etapa são definidos parâmetros como a classe responsável pelo processamento do *job*, o nome da fila de execução, o método HTTP utilizado e outras propriedades relacionadas à comunicação com serviços externos.

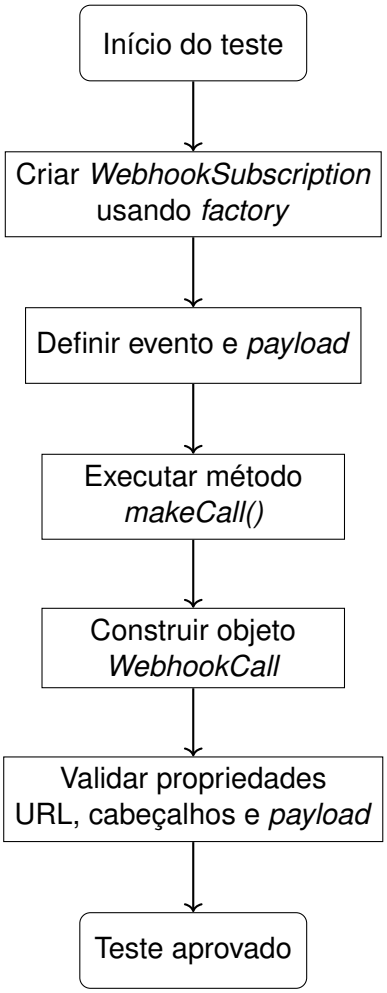
Durante o teste, a classe responsável pelo envio do *webhook* é substituída por uma implementação simulada denominada *FakeWebhookJob*, evitando que requisições HTTP reais sejam executadas. Em seguida, uma assinatura de *webhook* é criada utilizando uma *factory* do *framework Laravel*. Essa assinatura representa um usuário externo interessado em receber notificações quando determinados eventos ocorrerem no sistema. Entre os dados configurados estão a URL de destino da requisição e o segredo utilizado para assinatura criptográfica da mensagem.

Após a criação da assinatura, é definido um *payload* contendo dados de exemplo, bem como o evento associado à notificação. O sistema então executa o método responsável por construir a chamada de *webhook*, gerando um objeto do tipo *WebhookCall*. Esse objeto contém todas as informações necessárias para que o *webhook* seja executado posteriormente.

O teste realiza diversas verificações para garantir que a chamada foi configurada corretamente. São validados elementos como a URL do *webhook*, os cabeçalhos HTTP configurados, o conteúdo do *payload*, os metadados associados ao evento e as configurações de fila utilizadas para execução assíncrona. Dessa forma, o teste assegura que o sistema prepara corretamente as notificações externas que serão enviadas quando eventos relevantes ocorrerem na aplicação.

O fluxograma representado pela figura 16, resume de forma visual e objetiva o fluxo da construção do teste criado. Começando pelo ínicio do teste, logo em seguida a criação do *WebhookSubscription*, criação do evento e *payload*, a execução do método de chamada, a construção do objeto, por fim a validação das informações necessárias e a aprovação do teste.

Figura 16 – Fluxo de construção de uma chamada de *webhook*



Fonte: De autoria própria

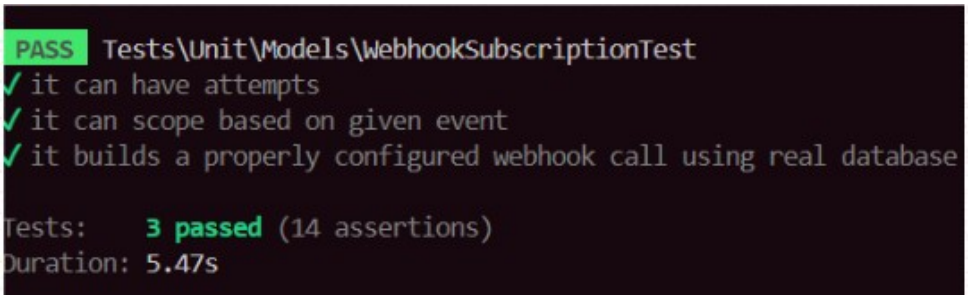
5.1.1.1 Resultados do Teste

Após a execução do teste, foi possível verificar que a aplicação constrói corretamente a chamada de *webhook* a partir das informações armazenadas na assinatura registrada no banco de dados. O objeto retornado pelo método responsável pela construção da chamada foi corretamente identificado como uma instância da classe *WebhookCall*, confirmando que o processo de criação da requisição foi realizado conforme o esperado. Durante a validação, foi confirmado que a URL utilizada para o envio da requisição corresponde exatamente ao endereço configurado na assinatura do *webhook*.

Além disso, os cabeçalhos HTTP foram corretamente definidos, incluindo o cabeçalho *User-Agent*, utilizado para identificar a aplicação responsável pela requisição. O conteúdo do *payload* também foi verificado, garantindo que os dados associados ao evento fossem corretamente estruturados antes do envio. O teste confirmou ainda que os metadados vinculados ao *job*, como o identificador da assinatura e o tipo de evento associado, foram corretamente atribuídos durante a construção da chamada. A figura

17 ilustra o *output* no editor de código do teste executado.

Figura 17 – Resultado da execução do teste de criação e chamada de webhook



Fonte: De autoria própria

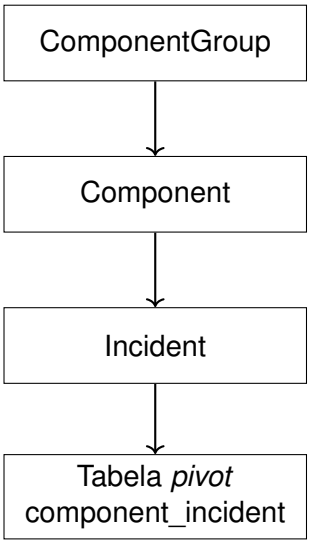
Outro aspecto validado foi a configuração do processamento assíncrono da requisição. O *job* responsável pelo envio do *webhook* foi corretamente configurado para utilização da *queue* definida nas configurações da aplicação, assegurando que o envio da notificação possa ser executado em segundo plano. Com base nessas verificações, conclui-se que o sistema prepara adequadamente as chamadas de *webhook*, garantindo que notificações externas possam ser enviadas com todos os parâmetros necessários quando eventos relevantes ocorrerem na aplicação.

5.1.2 Caso de teste — verificação do método *hasActiveIncident()*

A Figura 18 apresenta uma representação simplificada da relação entre os grupos de componentes, os componentes individuais e os incidentes registrados no sistema. Inicialmente, os componentes são organizados em entidades do tipo *ComponentGroup*, que permitem agrupar elementos da infraestrutura de acordo com sua função ou área de serviço. Cada *Component* representa um elemento específico que pode ser afetado por falhas ou interrupções.

Quando um problema ocorre, um registro do tipo *Incident* é criado para representar o evento ocorrido. A associação entre componentes e incidentes é armazenada em uma tabela intermediária, conhecida como tabela *pivot*, denominada `component_incident`. Essa estrutura permite registrar informações adicionais sobre a relação entre as entidades, como o estado do componente durante o incidente. Dessa forma, o sistema consegue representar de maneira flexível quais componentes foram afetados por determinados incidentes.

Figura 18 – Arquitetura simplificada da relação entre grupos, componentes e incidentes



Fonte: De autoria própria

O teste apresentado nesta seção tem como objetivo validar o correto funcionamento das associações entre incidentes e componentes dentro do sistema. Essas associações são fundamentais para representar a relação entre falhas registradas na plataforma e os elementos da infraestrutura que foram afetados.

Na primeira parte do teste, é verificada a associação entre um componente e um incidente utilizando a relação definida no modelo da aplicação. Para isso, são criadas instâncias das entidades *Component* e *Incident* por meio de *factories* do *framework Laravel*. Em seguida, o incidente é associado ao componente utilizando o método `attach`, que estabelece a relação no banco de dados juntamente com informações adicionais armazenadas na tabela intermediária.

Entre essas informações está o campo `component_status`, que indica o estado do componente no momento do incidente. Após a associação, o componente é recarregado juntamente com seus incidentes relacionados, garantindo que os dados atualizados sejam recuperados do banco de dados. O teste então verifica se exatamente um incidente está associado ao componente e se o valor armazenado no campo *pivot* corresponde ao estado esperado.

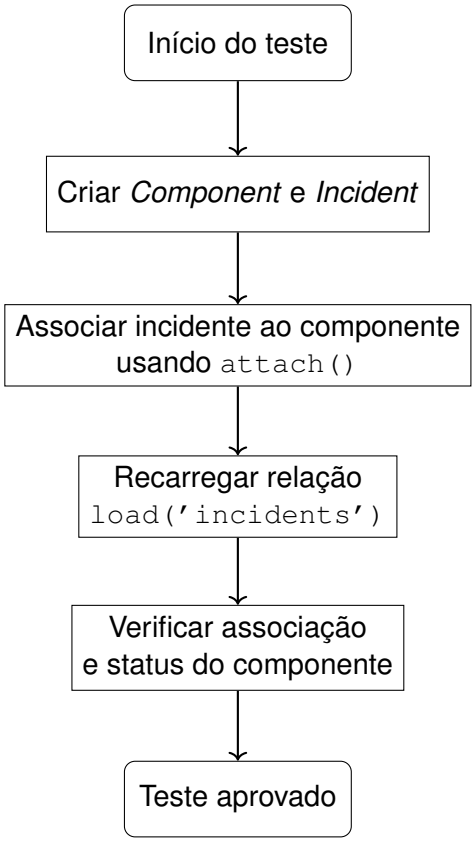
Na segunda parte do teste, é avaliada a lógica responsável por determinar se um grupo de componentes possui incidentes ativos. Inicialmente, um grupo de componentes é criado e um componente é associado a esse grupo. Em seguida, é verificado que o grupo não possui incidentes ativos, confirmando o comportamento esperado antes da criação de qualquer incidente.

Posteriormente, um incidente ativo é criado e associado ao componente previamente criado. Após recarregar os dados do grupo e de seus componentes, o método responsável por verificar a existência de incidentes ativos é executado novamente. O teste então confirma que o grupo passa a indicar corretamente a presença de incidentes

ativos.

Para que esses testes funcionassem corretamente, foram realizados ajustes nos modelos da aplicação, garantindo que as relações entre componentes, incidentes e grupos fossem corretamente definidas. Esses ajustes asseguraram que as consultas realizadas pelo sistema mostrem com melhor precisão o estado atual da infraestrutura monitorada. A figura 19 representa de forma visual e simplificada o fluxo do teste de associação entre um incidente e componente.

Figura 19 – Fluxo de associação entre incidente e componente



Fonte: De autoria própria

5.1.2.1 Resultados do Teste

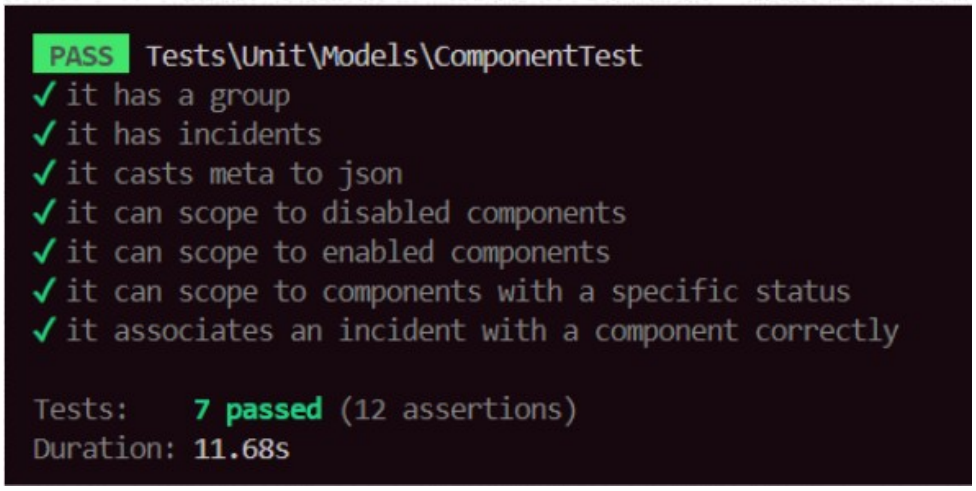
Após a execução dos testes automatizados, foi possível confirmar que as associações entre componentes e incidentes estão sendo corretamente persistidas e recuperadas pelo sistema. Na primeira parte do teste, foi verificado que o método responsável por associar um *Incident* a um *Component* por meio da relação definida no modelo funciona conforme esperado. A utilização do método `attach` registrou corretamente a relação na tabela intermediária *pivot* `component_incident`, incluindo o valor correspondente ao campo `component_status`. Além disso, após o recarregamento da relação utilizando o método `load`, foi possível confirmar que o componente possui exatamente um incidente associado e que o estado armazenado corresponde

ao valor definido durante a execução do teste.

Na segunda parte do teste, foi validado o funcionamento do método responsável por identificar a existência de incidentes ativos em um grupo de componentes. Inicialmente, quando nenhum incidente estava associado aos componentes do grupo, o método retornou o valor `false`, indicando corretamente a ausência de falhas registradas. Após a criação de um incidente ativo associado ao componente, o método foi executado novamente e passou a retornar `true`, confirmando que o sistema consegue identificar corretamente a presença de incidentes ativos.

Esses resultados demonstram que as relações definidas entre as entidades *ComponentGroup*, *Component* e *Incident* estão funcionando conforme o esperado, permitindo que o sistema represente de forma consistente o estado dos componentes monitorados e identifique corretamente situações de falha na infraestrutura, assim como representado na figura 20.

Figura 20 – Resultado da execução do teste de associação entre incidente e componente



Fonte: De autoria própria

5.1.3 Caso de teste — requisição inválida em um endpoint protegido

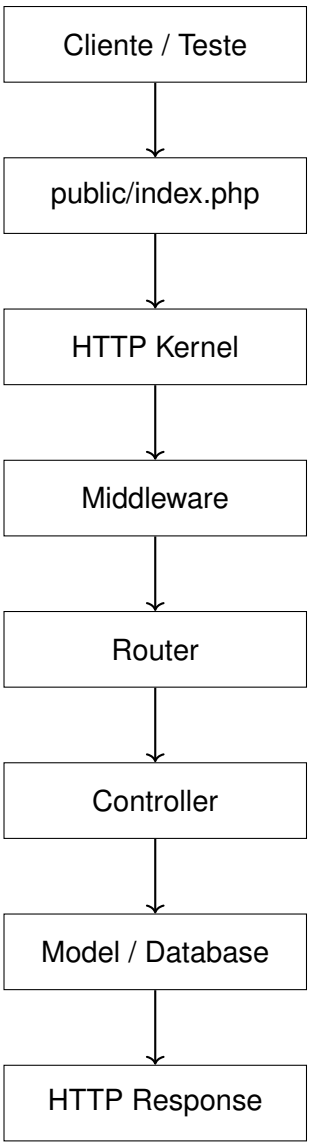
A Figura 21 apresenta uma visão simplificada da arquitetura de processamento de requisições no *framework Laravel*. Quando uma requisição é realizada, ela é inicialmente recebida pelo ponto de entrada da aplicação, localizado no arquivo `public/index.php`. Esse arquivo é responsável por inicializar o *framework* e encaminhar a requisição para o *HTTP Kernel*.

O *Kernel* gerencia o ciclo de vida da requisição dentro da aplicação, aplicando os *middlewares* configurados para realizar diferentes tipos de verificações, como autenticação, autorização e validação de dados. Após passar pelos *middlewares*, a requisição é direcionada para o sistema de roteamento do *framework*, que identifica qual controlador deve processar a solicitação. O controlador executa então a lógica de

negócio associada à operação solicitada, podendo interagir com modelos e bancos de dados para manipulação ou recuperação de informações. Após o processamento, uma resposta HTTP é gerada e retornada ao cliente.

No contexto do teste automatizado apresentado neste trabalho, o fluxo de execução é interrompido no nível do *middleware* de autenticação, impedindo que a requisição chegue ao controlador da aplicação. Esse comportamento reforça o mecanismo de segurança implementado no sistema, garantindo que apenas usuários autenticados possam acessar determinados recursos da API.

Figura 21 – Arquitetura simplificada do ciclo de requisição no Laravel

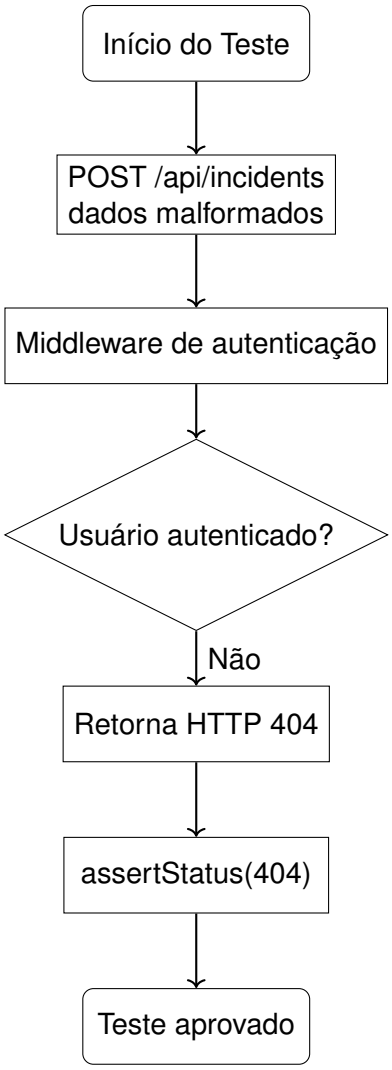


Fonte: De autoria própria

O caso de teste apresentado a seguir, é referente ao teste unitário do tipo teste sujo, onde o foco é verificar como o sistema reage a entradas inesperadas. O fluxograma apresentado na Figura 22 ilustra o fluxo lógico do teste automatizado responsável por verificar o comportamento do sistema ao receber uma requisição inválida

em um *endpoint* protegido. Inicialmente, o teste envia uma requisição do tipo *POST* para o *endpoint* `/api/incidents`, contendo dados propositalmente malformados.

Figura 22 – Fluxo do teste automatizado para requisição sem autenticação



Fonte: De autoria própria

Após o envio da requisição, o fluxo segue para o *middleware* de autenticação da aplicação, responsável por verificar se o usuário possui credenciais válidas para acessar o recurso solicitado. Como a requisição é realizada sem autenticação, o *middleware* interrompe o processamento antes que a requisição alcance o controlador responsável pela lógica de negócio.

Como resultado, o sistema retorna um código de resposta HTTP 404, indicando que o acesso ao recurso não foi permitido. Por fim, o teste automatizado valida se o código retornado corresponde ao comportamento esperado, utilizando uma asserção para confirmar que a resposta da API possui o status adequado. Esse processo garante que o sistema esteja protegido contra acessos não autenticados, mesmo quando são enviados dados inválidos na requisição.

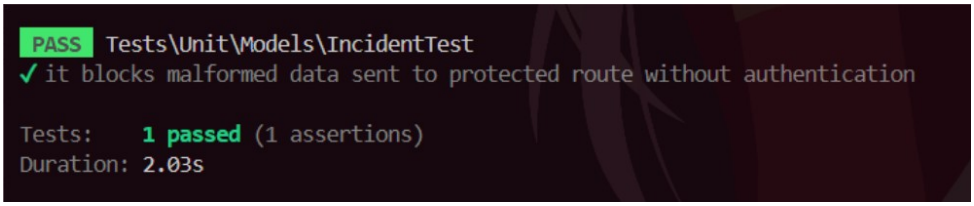
5.1.3.1 Resultados do Teste

Após a execução do teste automatizado, foi possível verificar que o sistema responde corretamente ao receber uma requisição inválida destinada a um *endpoint* protegido. A requisição enviada durante o teste contém dados malformados e não possui credenciais de autenticação, simulando uma tentativa de acesso inadequado à API. O resultado obtido confirmou que a aplicação bloqueia corretamente a requisição antes que qualquer processamento adicional seja realizado.

Nesse cenário, o sistema retornou o código de resposta HTTP correspondente ao comportamento esperado, impedindo que os dados inválidos fossem processados pelo sistema. Esse comportamento demonstra que os mecanismos de proteção da aplicação estão funcionando corretamente, garantindo que requisições não autenticadas ou contendo dados inconsistentes não consigam acessar rotas protegidas. Dessa forma, o sistema mantém a integridade das informações e reduz riscos relacionados ao envio de dados malformados ou potencialmente maliciosos.

Os resultados do teste ilustrados na figura 23 indicam, portanto, que a aplicação possui um controle adequado de acesso e validação de requisições, assegurando que apenas requisições válidas e devidamente autenticadas possam prosseguir para as etapas internas de processamento da API.

Figura 23 – Resultado da execução do teste de requisição a um *endpoint* protegido

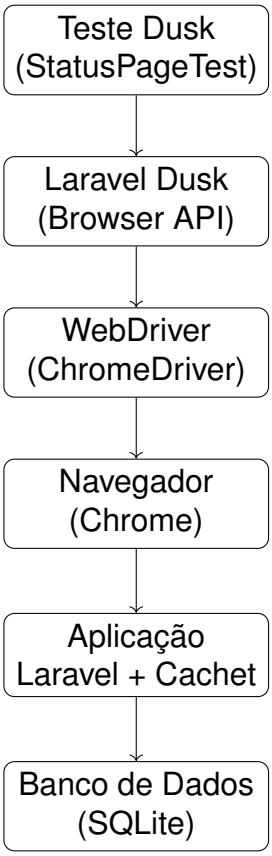


Fonte: De autoria própria

5.1.4 Caso de teste - exibição correta da página de *status*

A Figura 24 apresenta a arquitetura do teste automatizado utilizando o *Laravel Dusk*. Observa-se que o teste é estruturado em camadas, iniciando na classe de teste, responsável por definir o comportamento a ser validado. Em seguida, o *Laravel Dusk* atua como intermediador, fornecendo uma API para controle do navegador.

Figura 24 – Arquitetura em camadas do teste automatizado com Laravel Dusk

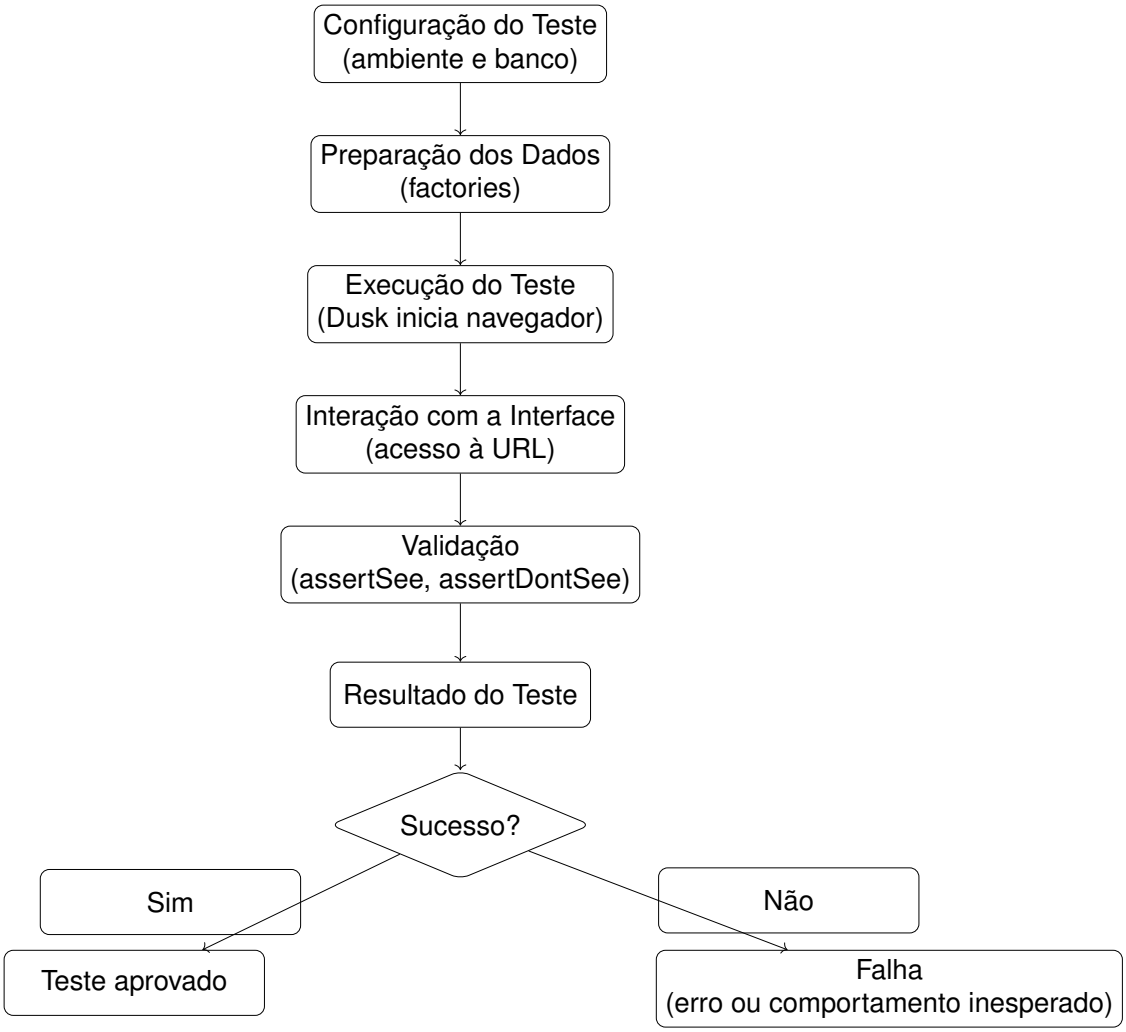


Fonte: De autoria própria

A execução do teste ocorre por meio do *WebDriver*, que estabelece a comunicação com o navegador, permitindo a simulação de ações do usuário. O navegador, por sua vez, interage diretamente com a aplicação desenvolvida em *Laravel*, responsável por processar as requisições e acessar o banco de dados. Essa organização em camadas facilita a compreensão do fluxo de execução e evidencia a separação de responsabilidades entre os componentes envolvidos.

Complementando essa visão, a Figura 25 apresenta o fluxo de execução do teste automatizado. Foi implementado um teste automatizado de interface utilizando o *Laravel Dusk*, com o objetivo de verificar o funcionamento da página pública de status do sistema. Inicialmente, é realizada a configuração do ambiente e a preparação dos dados necessários para o teste, utilizando factories para garantir consistência e isolamento.

Figura 25 – Fluxo de execução do teste automatizado da página de status



Fonte: De autoria própria.

O teste simulou o acesso de um usuário à aplicação e validou se as informações principais são exibidas corretamente. Para isso, foram criados, durante a execução, um componente e um incidente no banco de dados, garantindo que existiam dados para serem apresentados na interface. Em seguida, o teste é executado com a inicialização do navegador automatizado, que acessa a página de status do sistema. Durante essa etapa, são simuladas interações do usuário, como o acesso à URL principal da aplicação. Seguidamente, o navegador automatizado acessou a página inicial do sistema e verificou se os elementos esperados estão visíveis, como o nome do componente e o título do incidente.

Essas verificações são realizadas por meio de asserções, que confirmam se os dados aparecem corretamente na tela. Após o carregamento da página, são realizadas validações por meio de asserções, verificando se as informações esperadas estão presentes na interface. Por fim, o teste é concluído com a verificação do resultado, que pode indicar sucesso, quando o comportamento está correto, ou falha, caso haja divergências entre o esperado e o obtido. Dessa forma, a combinação entre a

arquitetura e o fluxo do teste permite uma visão completa do processo de validação, desde a execução técnica até a análise dos resultados.

5.1.4.1 Resultados do Teste

Durante a implementação, ocorreram erros relacionados ao ambiente de execução, principalmente falhas do tipo erro 500. Esses problemas estavam associados a inconsistências no banco de dados utilizado nos testes. Para resolver isso, foi utilizado o banco de dados *SQLite* no ambiente de testes, juntamente com a execução das migrations para garantir que todas as tabelas fossem criadas corretamente. Também foi necessário gerar a chave da aplicação, essencial para o funcionamento do *Laravel*.

Após essas correções, o teste passou a ser executado com sucesso, confirmando que a página de *status* é carregada corretamente e apresenta os dados esperados. Por fim, destaca-se que esse tipo de teste valida o comportamento da aplicação do ponto de vista do usuário, sendo importante para garantir a qualidade do sistema, mesmo não contribuindo diretamente para a cobertura de código.

5.2 ANÁLISE DA COBERTURA DE TESTES E DESEMPENHO DO SISTEMA

5.2.1 Estado inicial da cobertura

A primeira medição da cobertura de testes foi realizada com o objetivo de identificar o nível inicial de validação automatizada do sistema. O relatório gerado indicou uma cobertura de 40,63% das linhas de código, evidenciando que menos da metade do código executável estava sendo exercitada pelos testes. Observou-se uma distribuição desigual da cobertura entre os diferentes módulos do sistema.

Componentes relacionados à lógica de negócio, como *Actions*, *Jobs* e *Query-Builders*, apresentaram altos níveis de cobertura, enquanto elementos como *Enums*, *Views* e componentes de interface apresentaram baixos índices. Além disso, a análise revelou a presença de classes com alta complexidade ciclomática associada a baixa cobertura, indicando potenciais riscos para manutenção e evolução do sistema.

5.2.2 Estratégia de melhoria

Com base nos resultados iniciais, foi definida uma estratégia incremental para aumento da cobertura de testes, priorizando componentes com baixa cobertura e alta reutilização, como *Enums*, modelos de domínio (*Models*), regras de validação e lógica de negócio crítica. Essa abordagem permitiu maximizar o ganho de cobertura com menor esforço de implementação.

Adicionalmente, foram implementados diversos novos testes com o objetivo de ampliar a cobertura geral do sistema, com destaque para a criação de testes do

tipo *Filament*, voltados à validação de interfaces administrativas, e testes de *Events*, responsáveis por garantir o correto disparo e comportamento de eventos dentro da aplicação. Esses tipos concentraram a maior parte dos incrementos realizados, contribuindo significativamente para a melhoria da confiabilidade e robustez do sistema. Além desses, também foram desenvolvidos testes unitários, focados na validação isolada de métodos e classes, garantindo o correto funcionamento de pequenas unidades de código.

Em conjunto, foram aplicados testes de integração, responsáveis por verificar o comportamento do sistema como um todo, incluindo rotas, controladores e interações com o banco de dados. Essa combinação permitiu uma cobertura mais abrangente, contemplando tanto aspectos internos quanto fluxos completos da aplicação. Adicionalmente, foi incorporada a prática de testes de regressão, por meio da reexecução contínua da suíte de testes automatizados após cada modificação no sistema. Essa abordagem teve como objetivo garantir que funcionalidades previamente validadas permanecessem estáveis, evitando a introdução de falhas decorrentes de alterações no código.

Por fim, foram incluídos testes relacionados a validações específicas, como *Form Requests*, políticas de autorização (*Policies*), além de testes voltados para *Jobs* e *Listeners*, assegurando o correto processamento de tarefas assíncronas e reações a eventos. Também foram considerados testes para *Observers* e *Factories*, contribuindo para a consistência dos dados e facilitando a criação de cenários controlados. Dessa forma, a diversidade de tipos de testes implementados reforça a qualidade do *software*, reduzindo riscos e aumentando a confiabilidade do sistema como um todo.

5.2.3 Evolução da cobertura e impacto na qualidade

Após a implementação dos testes adicionais, foram realizadas novas medições com o objetivo de avaliar, de forma quantitativa, a evolução da cobertura do sistema ao longo das etapas definidas. Essa análise permitiu comparar o impacto direto da estratégia incremental adotada, evidenciando os ganhos obtidos em diferentes níveis de granularidade do código, como linhas executadas, métodos testados e classes cobertas.

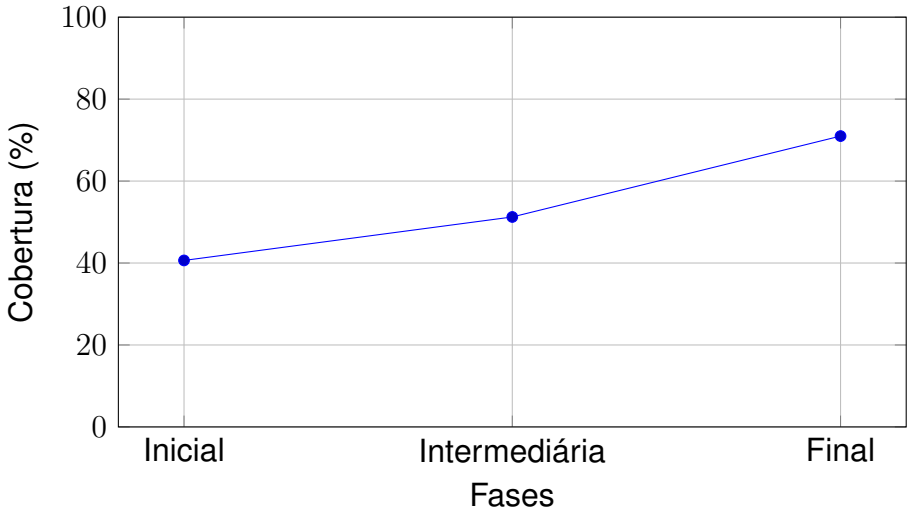
Observa-se, a partir dos dados apresentados na Quadro 18, uma evolução progressiva e consistente em todas as métricas analisadas. Na fase inicial, os valores indicavam uma cobertura ainda limitada, com maior concentração em métodos. Já na fase intermediária, nota-se um crescimento moderado, resultado da inserção dos primeiros conjuntos de testes estruturados. Por fim, a fase final apresenta um aumento expressivo, atingindo 70,98% de cobertura de linhas, 79,20% de métodos e 62,75% de classes, demonstrando a efetividade das ações implementadas.

Quadro 18 – Evolução da Cobertura de Testes

Fase	Linhas (%)	Métodos (%)	Classes (%)
Inicial	40,63	53,42	40,20
Intermediária	51,24	60,95	45,59
Final	70,98	79,20	62,75

Esse crescimento ilustrado pela figura 26 pode ser diretamente associado à priorização de componentes críticos e de alta reutilização, bem como à diversificação dos tipos de testes implementados, incluindo testes unitários, de integração, *Filament* e *Events*. Em especial, o salto observado entre as fases intermediária e final evidencia o amadurecimento da suíte de testes e a ampliação da cobertura em áreas anteriormente não testadas. Dessa forma, os resultados obtidos não apenas demonstram um aumento quantitativo da cobertura, mas também indicam uma melhoria qualitativa na confiabilidade do sistema, uma vez que mais partes do código passaram a ser validadas automaticamente. Isso contribui diretamente para a redução de falhas, maior segurança em manutenções futuras e maior robustez geral da aplicação.

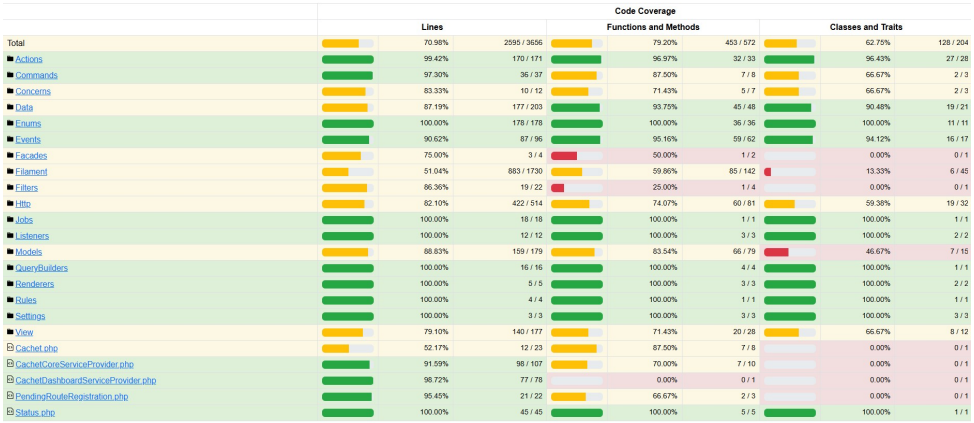
Figura 26 – Evolução da cobertura de testes ao longo das fases



Fonte: De autoria própria

A Figura 27 apresenta um recorte do último relatório de cobertura de código gerado com o auxílio do *Xdebug*, evidenciando de forma detalhada a distribuição da cobertura entre os diferentes componentes do sistema. A visualização permite não apenas a análise dos índices gerais, mas também a identificação precisa de áreas com maior ou menor nível de testabilidade. Observa-se que componentes como *Enums*, *Jobs*, *Listeners*, *QueryBuilders* e *Rules* atingem níveis elevados de cobertura, frequentemente próximos ou iguais a 100%, o que indica uma boa validação de estruturas mais objetivas e de menor acoplamento.

Figura 27 – Relatório final do Xdebug



	Code Coverage							
	Lines			Functions and Methods			Classes and Traits	
Total		70.98%	2595 / 3656		79.20%	453 / 572		62.75%	128 / 204
Actions		99.42%	170 / 171		96.97%	32 / 33		96.43%	27 / 28
Commands		97.30%	36 / 37		87.50%	7 / 8		66.67%	2 / 3
Concerns		83.33%	10 / 12		71.43%	5 / 7		66.67%	2 / 3
Data		87.19%	177 / 203		93.75%	45 / 48		90.48%	19 / 21
Enums		100.00%	178 / 178		100.00%	36 / 36		100.00%	11 / 11
Events		90.62%	87 / 96		95.16%	59 / 62		94.12%	16 / 17
Facades		75.00%	3 / 4		50.00%	1 / 2		0.00%	0 / 1
Filament		51.04%	883 / 1730		59.86%	85 / 142		13.33%	6 / 45
Filters		86.36%	19 / 22		25.00%	1 / 4		0.00%	0 / 1
Http		82.10%	422 / 514		74.07%	60 / 81		59.38%	19 / 32
Jobs		100.00%	18 / 18		100.00%	1 / 1		100.00%	1 / 1
Listeners		100.00%	12 / 12		100.00%	3 / 3		100.00%	2 / 2
Models		88.83%	159 / 179		83.54%	66 / 79		46.67%	7 / 15
QueryBuilders		100.00%	16 / 16		100.00%	4 / 4		100.00%	1 / 1
Renderers		100.00%	5 / 5		100.00%	3 / 3		100.00%	2 / 2
Rules		100.00%	4 / 4		100.00%	1 / 1		100.00%	1 / 1
Settings		100.00%	3 / 3		100.00%	3 / 3		100.00%	3 / 3
View		79.10%	140 / 177		71.43%	20 / 28		66.67%	8 / 12
Cachet.php		52.17%	12 / 23		87.50%	7 / 8		0.00%	0 / 1
CachetCoreServiceProvider.php		91.59%	98 / 107		70.00%	7 / 10		0.00%	0 / 1
CachetDashboardServiceProvider.php		98.72%	77 / 78		0.00%	0 / 1		0.00%	0 / 1
PendingRouteRegistration.php		95.45%	21 / 22		66.67%	2 / 3		0.00%	0 / 1
Status.php		100.00%	45 / 45		100.00%	5 / 5		100.00%	1 / 1

Fonte: De autoria própria

De forma semelhante, módulos como *Actions*, *Commands* e *Events* também apresentam cobertura significativamente alta, refletindo o esforço direcionado para a validação de regras de negócio e fluxos internos da aplicação. Esse resultado possui impacto direto na confiabilidade operacional do sistema, reduzindo riscos de regressões em funcionalidades responsáveis pelo gerenciamento de incidentes, notificações e execução de tarefas automatizadas. Por outro lado, ainda é possível identificar pontos com cobertura inferior, como o módulo *Filament*, além de componentes como *Facades*, *Filters* e algumas classes específicas, especialmente no que se refere à cobertura de classes e métodos.

Essas áreas, destacadas em tons de amarelo e vermelho no relatório, representam partes do sistema com maior complexidade ou forte acoplamento com interface e infraestrutura, o que dificulta a criação de testes automatizados mais abrangentes. Essa análise reforça a importância da estratégia incremental adotada, na qual a priorização inicial foi direcionada a componentes críticos e mais acessíveis à testabilidade, permitindo ganhos rápidos de cobertura. Ao mesmo tempo, evidencia oportunidades de evolução futura, especialmente na ampliação da cobertura de camadas mais complexas, como interfaces administrativas e integrações, contribuindo para o contínuo aprimoramento da qualidade do sistema.

Entretanto, embora os resultados quantitativos indiquem um crescimento consistente da cobertura de linhas, métodos e classes, é importante destacar que essa métrica não representa, isoladamente, um indicador absoluto da qualidade do software. A cobertura de testes mede essencialmente o percentual de código executado durante a execução da suíte de testes, mas não garante que os comportamentos críticos da aplicação estejam sendo validados de maneira robusta. Dessa forma, um sistema pode apresentar elevados índices de cobertura e ainda assim possuir testes frágeis, superficiais ou incapazes de detectar falhas relevantes.

Nesse contexto, os resultados obtidos neste trabalho foram analisados não ape-

nas sob a perspectiva quantitativa da cobertura alcançada, mas considerando também o impacto na qualidade do conjunto de testes na confiabilidade da aplicação. Durante a implementação dos testes, foram identificados comportamentos inconsistentes, dependências implícitas e dificuldades relacionadas ao acoplamento entre componentes do sistema. Esse processo evidenciou que os testes atuaram também como mecanismos de inspeção arquitetural e compreensão do código legado, auxiliando na identificação de limitações estruturais da aplicação.

A inclusão de testes unitários, de integração, *Filament* e *Events* permitiu validar diferentes níveis do comportamento do sistema. Já os testes de integração mostraram-se especialmente importantes em cenários fortemente dependentes da infraestrutura do *framework Laravel*, como autenticação, eventos, acesso ao banco de dados e gerenciamento de componentes administrativos. Essa combinação possibilitou uma validação mais abrangente da aplicação, aproximando os cenários de teste do comportamento real do sistema em produção.

Além disso, observou-se que a ampliação da cobertura em módulos relacionados ao gerenciamento de incidentes, componentes e notificações possui impacto direto na confiabilidade operacional do *Cachet*. Considerando que essas funcionalidades representam partes principais da plataforma de monitoramento de *status*, a validação automatizada desses fluxos reduz significativamente os riscos de regressões funcionais e inconsistências em futuras modificações do sistema. Outro aspecto relevante identificado durante o desenvolvimento dos testes foi a influência da arquitetura do *Laravel* na testabilidade do sistema.

Recursos como *Facades, Service Providers, bootstrapping* automático e o forte acoplamento proporcionado pelo *Eloquent ORM* facilitaram o desenvolvimento inicial da aplicação, mas também dificultaram o isolamento completo de determinados componentes durante os testes unitários. Em diversos cenários, tornou-se mais viável utilizar testes de integração com banco em memória do que buscar isolamento absoluto das dependências internas do *framework*. Apesar dessas limitações, a evolução da cobertura demonstrou que a adoção incremental de testes automatizados em sistemas legados é viável e pode produzir ganhos significativos de confiabilidade, manutenção e segurança na evolução contínua da aplicação.

5.2.4 Análise de desempenho e carga

Para avaliar o desempenho do sistema, foram realizados testes de carga utilizando a ferramenta *ApacheBench*, *considera*ndo diferentes níveis de concorrência com o objetivo de analisar o comportamento da aplicação sob cenários de uso moderado e intenso. No cenário de carga moderada, foram executadas 1000 requisições com nível de concorrência de 10 usuários simultâneos. Os resultados indicaram uma taxa média de 2,06 requisições por segundo, com tempo médio de resposta de aproximadamente

4,86 segundos. Não foram observadas falhas durante a execução, evidenciando a estabilidade do sistema nesse contexto.

No cenário de carga elevada, foram realizadas 2000 requisições com concorrência de 50 usuários simultâneos. Nesse caso, a taxa de requisições por segundo foi de 1,97, enquanto o tempo médio de resposta aumentou significativamente para aproximadamente 25,36 segundos. Assim como no cenário anterior, não foram registradas falhas, indicando que o sistema manteve sua estabilidade mesmo sob maior nível de carga. A análise comparativa entre os dois cenários revela que o *throughput* do sistema permaneceu praticamente constante, mesmo com o aumento significativo da concorrência.

Esses resultados obtidos demonstraram que as modificações realizadas para aumentar a testabilidade do sistema não provocaram degradação significativa no desempenho da aplicação. Mesmo com a inclusão de novas camadas de validação, reorganização de componentes e ajustes estruturais necessários para facilitar a implementação dos testes, o sistema manteve tempos de resposta estáveis e comportamento consistente durante os cenários de carga executados. Por outro lado, houve um crescimento expressivo no tempo de resposta, evidenciando que o sistema não apresenta escalabilidade linear.

Além disso, os testes de carga permitiram identificar pontos sensíveis da aplicação relacionados ao consumo de recursos e ao processamento de determinadas requisições. Esse comportamento sugere a presença de gargalos no processamento das requisições, possivelmente relacionados ao acesso ao banco de dados, ausência de mecanismos de cache ou limitações do ambiente de execução. A utilização dessa abordagem tornou possível observar não apenas a capacidade de processamento do sistema, mas também verificar se as alterações introduzidas durante a implementação dos testes automatizados impactaram o comportamento geral da aplicação.

Outro aspecto importante está relacionado à confiabilidade operacional da aplicação. Em sistemas de monitoramento de *status*, como o *Cachet*, estabilidade e previsibilidade são fatores críticos, uma vez que falhas ou lentidão durante incidentes reais podem comprometer diretamente a experiência dos usuários e a credibilidade da plataforma. Nesse contexto, os testes de desempenho contribuíram para validar que o processo de evolução arquitetural e aumento da cobertura de testes não comprometeu a capacidade operacional do sistema.

Os resultados também reforçam a importância da integração entre testes funcionais e testes não funcionais no processo de validação da qualidade de *software*. Dessa forma, conclui-se que o sistema é capaz de operar de maneira estável sob diferentes níveis de carga, mantendo taxa zero de falhas. No entanto, apresenta limitações em termos de *throughput* e escalabilidade, o que indica a necessidade de otimizações futuras para melhorar seu desempenho em cenários com maior volume de

acessos.

6 CONSIDERAÇÕES FINAIS

Segundo Beck (2003), testes automatizados permitem que o software evolua de forma mais segura, reduzindo os riscos associados à introdução de novas funcionalidades e modificações estruturais. Com base nos resultados apresentados, é possível afirmar que a estratégia incremental adotada para o aumento da cobertura de testes mostrou-se eficaz, permitindo uma evolução significativa nas métricas analisadas. A priorização de componentes críticos e de alta reutilização, aliada à diversificação dos tipos de testes implementados, contribuiu diretamente para o aumento consistente da cobertura de linhas, métodos e classes ao longo das etapas do trabalho.

A estratégia utilizada, também permitiu validar diferentes comportamentos do sistema de forma progressiva, reduzindo riscos associados a regressões e falhas em funcionalidades principais do programa. Durante o desenvolvimento dos testes, foram observadas que determinadas características arquiteturais do ecossistema *Laravel* influenciaram diretamente a complexidade do processo de testabilidade. Embora o *framework* forneça de fato recursos avançados para criação de testes, como a integração nativa com o *PHPUnit* e *Pest*, *factories* e banco em memória, alguns padrões amplamente utilizados em aplicações *Laravel* também introduzem desafios relacionados ao isolamento de componentes e à construção de testes unitários mais independentes. Um dos principais fatores observados foi o uso intensivo de *Facades*.

Apesar de oferecerem uma sintaxe simplificada e maior produtividade durante o desenvolvimento, as *Facades* abstraem dependências internas do *framework* por meio de chamadas estáticas, dificultando a identificação explícita das dependências utilizadas por determinadas classes e métodos. Além disso, a utilização de *Service Providers* e do mecanismo de *bootstrapping* automático do *Laravel* faz com que diversos serviços sejam inicializados implicitamente durante a execução da aplicação. Embora essa abordagem facilite a configuração do ambiente e a integração entre componentes, ela também torna mais complexa a separação de responsabilidades durante os testes, especialmente em cenários nos quais determinados serviços precisam ser simulados ou substituídos por *mocks*.

Outro aspecto relevante envolve o uso do *Eloquent ORM*. O forte acoplamento entre modelos, relacionamentos e acesso ao banco de dados favorece o desenvolvimento rápido da aplicação. Porém isso pode dificultar testes unitários, uma vez que muitas operações dependem diretamente da persistência de dados e do estado do banco. Em diversos casos observados neste trabalho, tornou-se mais viável utilizar testes de integração com banco em memória do que tentar isolar completamente determinados componentes do sistema.

Segundo Feathers (2004), um sistema pode ser considerado código legado

quando não possui um conjunto de testes automatizados suficientemente confiável para permitir alterações seguras em sua estrutura. Os fatores observados evidenciam que, em aplicações *Laravel* legadas, a fronteira entre testes unitários e testes de integração tende a se tornar menos rígida, exigindo estratégias híbridas de validação. Para Fowler (2018), sistemas fortemente acoplados tendem a apresentar maior dificuldade de testabilidade, especialmente em cenários que exigem isolamento de componentes e validação independente de comportamentos, conforme identificado no desenvolvimento deste trabalho.

Também devido ao forte acoplamento do sistema, testes de integração acabaram se tornando grande parte da implementação, apesar da Pirâmide de Testes de Fowler (2018) priorizar os testes unitários. Durante o desenvolvimento dos testes, foi necessário equilibrar isolamento, custo de manutenção e fidelidade do comportamento real da aplicação, priorizando abordagens que permitissem validar o funcionamento efetivo do sistema sem comprometer excessivamente a complexidade da implementação dos testes. Apesar dessas limitações, observou-se que o ecossistema *Laravel* fornece ferramentas suficientemente robustas para viabilizar a construção incremental de testes automatizados, mesmo em sistemas com forte acoplamento arquitetural.

Dessa forma, os desafios encontrados não invalidam a utilização do *framework* em aplicações de grande porte, mas reforçam a importância de boas práticas arquiteturais e redução de acoplamento para melhorar a testabilidade e a manutenção contínua do software. Segundo Feathers (2004), o processo de implementação de testes em sistemas legados também auxilia na compreensão da aplicação e na identificação de limitações na arquitetura. Esses fatores evidenciaram que a testabilidade de um sistema está diretamente relacionada à sua organização arquitetural, tornando os testes não apenas instrumentos de validação, mas também mecanismos de análise da qualidade interna do software. O que mostra que além do aumento das métricas de cobertura, a implementação dos testes proporcionou ganhos relacionados à compreensão estrutural do sistema e à identificação de limitações arquiteturais do código legado.

Entretanto, apesar dos avanços obtidos, ainda existem limitações a serem consideradas, como a dificuldade em testar determinados fluxos mais complexos ou fortemente acoplados, além do custo de manutenção da suíte de testes à medida que o sistema evolui. Conforme discutido por Fowler (2018), métricas quantitativas, como cobertura de código, devem ser interpretadas com cautela, pois apesar de representar um avanço importante no alcance da suíte de testes, essa métrica não garante a qualidade ou a eficácia dos testes desenvolvidos. Dessa forma, a cobertura mostrou-se mais adequada como indicador quantitativo de alcance do que como medida definitiva da qualidade do *software*.

Para Beck (2003), a qualidade do software não depende apenas de corrigir as falhas, mas também da capacidade de manter o sistema evolutivo, sustentável e

confiável ao longo do tempo. Esses fatores indicam que o processo de testes deve ser contínuo e adaptativo. Por isso, como trabalhos futuros, fica a sugestão da ampliação da cobertura em áreas ainda não contempladas, a adoção de testes de interface mais robustos, e a integração com ferramentas de análise contínua de qualidade de código. Além disso, a utilização de métricas complementares pode fornecer uma visão ainda mais aprofundada sobre a qualidade do software.

Também se mostra relevante a utilização de testes de mutação, onde são feitas alterações no código para verificar não apenas o comportamento do sistema, como a precisão dos testes criados. A inserção desses testes permitiria avaliar não apenas o percentual de código executado, mas também a capacidade efetiva dos testes em observar comportamentos incorretos e validar a robustez das asserções implementadas. Por fim, conclui-se que a aplicação sistemática de testes automatizados no contexto do *framework Laravel* é uma prática essencial para o desenvolvimento de *software* de qualidade, contribuindo para sistemas mais seguros, confiáveis e sustentáveis a longo prazo.

REFERÊNCIAS

Apache Software Foundation. **Apache HTTP server benchmarking tool (ab)**. [S.l.], 2024. Disponível em: <https://httpd.apache.org/docs/2.4/programs/ab.html>. Acesso em: 12 jul. 2025.

Associação Brasileira de Normas Técnicas. **NBR ISO 9000:2015 – Sistemas de gestão da qualidade – Fundamentos e vocabulário**. 2015. Rio de Janeiro: ABNT. Norma Técnica.

Associação Brasileira de Normas Técnicas. **NBR ISO 9001: Sistemas de gestão da qualidade – Requisitos**. Rio de Janeiro: ABNT, 2015. Equivalente à norma internacional ISO 9001.

BECK, K. **Test-Driven Development: By Example**. Boston: Addison-Wesley Professional, 2003.

BECK, K. **Desenvolvimento Guiado por Testes [recurso eletrônico]**. Porto Alegre: Bookman, 2010. Tradução: Jean Felipe Patikowski Cheiran. Revisão técnica: Marcelo Soares Pimenta. Dados eletrônicos.

BEIZER, B. **Software Testing Techniques**. 2. ed. New York: Van Nostrand Reinhold, 1995.

BETTONI, F. N.; FLORIAN, F.; FARINA, R. M. A importância do teste de software na garantia de qualidade em projetos de tecnologia de informação (ti) - análise comparativa entre metodologias de teste em uma empresa. **REVISTA FOCO**, v. 17, n. 12, p. e6980, dez. 2024. Disponível em: <https://ojs.focopublicacoes.com.br/foco/article/view/6980>. Acesso em: 29 jun. 2025.

CachetHQ- Documentação oficial do Cachet, em desenvolvimento ativo. **Cachet v3.x Introduction**. 2025. Disponível em: <https://docs.cachethq.io/v3.x/introduction>. Acesso em: 08 jul. 2025.

CAROSIA, J.; CASTRO, M. J. de; CALAZANS, P. H. Teste de software: Avaliação da usabilidade de software em ambiente web. In: **Congresso de Tecnologia - Fatec Mococa**. [S.l.: s.n.], 2024.

Composer Project. **Composer - Dependency Manager for PHP**. 2025. Disponível em: <https://getcomposer.org/>. Acesso em: 08 jul. 2025.

Consortium for Information & Software Quality (CISQ). **The Cost of Poor Software Quality in the U.S.: A 2020 Report**. 2020. CISQ.

DINWIDDIE, G. **The Three Amigos**. 2011. Disponível em: <https://blog.gdinwiddie.com/2011/05/03/the-three-amigos/>. Acesso em: 13 ago. 2025.

ExcellentWebWorld. **Top 7 PHP frameworks foe web development in 2025**. 2025. Disponível em: <https://www.excellentwebworld.com/best-php-frameworks/>. Acesso em: 29 jun. 2025.

FEATHERS, M. **Working Effectively with Legacy Code**. Upper Saddle River: Prentice Hall, 2004.

FOWLER, M. **Refactoring: Improving the Design of Existing Code**. 2. ed. Boston: Addison-Wesley Professional, 2018.

Git Project. **Git Documentation**. 2025. Disponível em: <https://git-scm.com/doc>. Acesso em: 08 jul. 2025.

GitHub Inc. **GitHub**. 2025. Disponível em: <https://github.com/>. Acesso em: 08 jul. 2025.

GORDIEIEV, O. *et al.* Relationship between factors influencing the software development process and software defects. In: **2023 13th International Conference on Dependable Systems, Services and Technologies (DESSERT)**. [S.l.: s.n.], 2023. p. 1–7.

HETZEL, W. C. **The Complete Guide to Software Testing**. 2. ed. Wellesley, MA: QED Information Sciences, Inc., 1987.

HEUVEL, B. vd. **Laravel Debugbar**. 2024. Disponível em: <https://github.com/barryvdh/laravel-debugbar>. Acesso em: 08 jul. 2025.

ISO/IEC. **ISO/IEC 25010:2011 — Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models**. [S.l.]: International Organization for Standardization (ISO), 2011. Geneva.

ISO/IEC JTC 1/SC 7. **ISO/IEC 25000:2014 – Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE**. [S.l.], 2014. Disponível em: <https://iso25000.com/index.php/en/iso-25000-standards>. Acesso em: 28 jun. 2025.

ISO/IEC/IEEE. **ISO/IEC/IEEE 29119-1:2022 – Software and systems engineering – Software testing – Part 1: General concepts**. 2022. 2ª edição, publicada em 4 de fevereiro de 2022.

LARAVEL. **Laravel: Documentação oficial. Versão 11**. [S.l.], 2024. Disponível em: <https://laravel.com/docs/11.x>. Acesso em: 29 jun. 2025.

Laravel: Documentação oficial. **Laravel Dusk - Browser Testing Documentation**. 2024. Disponível em: <https://laravel.com/docs/11.x/dusk>. Acesso em: 08 jul. 2025.

Laravel: Documentação oficial. **Laravel Telescope - Debug Assistant for Laravel**. 2024. Disponível em: <https://laravel.com/docs/11.x/telescope>. Acesso em: 08 jul. 2025.

LARAVEL Ecosystem Survey 2023. 2023. Disponível em: <https://laravel-news.com/laravel-survey-2023>. Acesso em: 28 jun. 2025.

MCCONNELL, S. **Code Complete: A Practical Handbook of Software Construction**. 2. ed. Redmond, WA: Microsoft Press, 2004. ISBN 978-0735619678.

Microsoft. **Visual Studio Code**. 2025. Disponível em: <https://code.visualstudio.com/>. Acesso em: 08 jul. 2025.

MOSHARRAF, B. F. e F. **Fundamentos da Ciência da Computação**. 2. ed. São Paulo: Cengage Learning, 2018.

NORTH, D. Introducing bdd. **Better Software Magazine**, March 2006. Disponível em: <https://dannorth.net/introducing-bdd/>. Acesso em: 26 jun. 2025.

Nuno Maduro et al. **Pest - Framework de Testes PHP Simples e Elegante**. [S.l.], 2024. Disponível em: <https://pestphp.com>. Acesso em: 08 jul. 2025.

OTWELL, T. **Laravel - The PHP Framework For Web Artisans**. 2025. Disponível em: <https://laravel.com/>. Acesso em: 05 jul. 2025.

PELIZZA, A. C.; BERTOLINI, C.; SILVEIRA, S. R. Um estudo sobre técnicas de teste de software no framework laravel. **Revista Eletrônica de Sistemas de Informação e Gestão Tecnológica**, v. 9, n. 3, 2018.

PRESSMAN, R. S. **Engenharia de Software: Uma Abordagem Profissional**. 8. ed. São Paulo: McGraw-Hill, 2016.

RETHANS, D. **Xdebug: PHP Debugger**. 2024. Disponível em: <https://xdebug.org/docs>. Acesso em: 08 jul. 2025.

Sebastian Bergmann and contributors. **PHPUnit**. 2025. Disponível em: <https://phpunit.de/>. Acesso em: 08 jul. 2025.

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson Education, 2011.

SOMMERVILLE, I. **Engenharia de Software**. 10. ed. São Paulo: Pearson Education, 2019.

SWEBOK. **Guide to the Software Engineering Body of Knowledge (SWEBOK)**. 3. ed. Los Alamitos: IEEE Computer Society, 2014.

SWEBOK. **The Guide to the Software Engineering Body of Knowledge (SWEBOK V4.0)**. 4. ed. New Jersey: IEEE, 2024.

ULRICH, K. T.; EPPINGER, S. D. **Product Design and Development**. 6. ed. New York: McGraw-Hill Education, 2016. Capítulo 10 define modelo/arquitetura como estrutura funcional e física do produto.