
Universidade Federal do Maranhão
Coordenação do Curso de Engenharia da Computação
Centro de Ciências Exatas e Tecnologia - CCET

Claudio Henrique Velozo Alexandre

# IMPLEMENTANDO TESTES DE SOFTWARE EM UMA APLICAÇÃO *WEB* PARA CLUBES DE LEITURA

São Luís
2025

Claudio Henrique Velozo Alexandre

# IMPLEMENTANDO TESTES DE SOFTWARE EM UMA APLICAÇÃO *WEB* PARA CLUBES DE LEITURA

Trabalho de Conclusão de Curso II, apresentado ao curso de Engenharia da Computação, como requisito para a obtenção do Título de Bacharel em Engenharia da Computação. Centro de Ciência Exatas e Tecnológicas da Universidade Federal do Maranhão

Orientador: Prof. Dr. Davi Viana dos Santos

São Luís

2025

Ficha gerada por meio do SIGAA/Biblioteca com dados fornecidos pelo(a) autor(a).
Diretoria Integrada de Bibliotecas/UFMA

Alexandre, Claudio Henrique Velozo.
Implementando Testes de Software em uma Aplicação Web para Clubes de Leitura / Claudio Henrique Velozo Alexandre. - 2025.
197 f.

Orientador(a): Davi Viana dos Santos.
Monografia (Graduação) - Curso de Engenharia da Computação, Universidade Federal do Maranhão, São Luís, 2025.

1. Qualidade de Software. 2. Testes Automatizados. 3. Testes Funcionais. 4. Testes End- To-end. 5. Clubes de Leitura. I. Santos, Davi Viana dos. II. Título.

Claudio Henrique Velozo Alexandre

# IMPLEMENTANDO TESTES DE SOFTWARE EM UMA APLICAÇÃO *WEB* PARA CLUBES DE LEITURA

Trabalho de Conclusão de Curso II, apresentado ao curso de Engenharia da Computação, como requisito para a obtenção do Título de Bacharel em Engenharia da Computação. Centro de Ciência Exatas e Tecnológicas da Universidade Federal do Maranhão

BANCA EXAMINADORA

---

**Prof. Dr. Davi Viana dos Santos**
(Orientador)
Universidade Federal do Maranhão

---

**Prof. Dr. Alex Oliveira Barradas Filho**
Universidade Federal do Maranhão

---

**Prof. Me. Bruno Carvalho da Silva**
Universidade Federal do Maranhão

São Luís

2025

# Agradecimentos

Agradeço primeiramente a Deus, por me permitir trilhar este caminho e obter sucesso nos meus objetivos.

À minha família, pelo amor, paciência e incentivo. Em especial, aos meus pais, que me apoiaram durante toda esta jornada.

À minha esposa, por ser paciente e compreensiva, e pelo seu companheirismo que me ajudou enormemente.

Ao professor Davi Viana, pela orientação durante o desenvolvimento deste trabalho. E ainda por suas belas aulas, onde absorvi conhecimentos que utilizei neste trabalho. Agradeço pela paciência e disponibilidade em ajudar quando precisei de suas instruções.

À professora Patricia Figueiredo e à Erlane Alcântara, pela oportunidade de contribuir para o desenvolvimento e melhoria da aplicação.

Ao Mateus Oliveira e à Emanuelle Lemos, por atenderem aos meus chamados quando solicitados, pela ajuda para entender as características da aplicação e por gentilmente disponibilizarem seus trabalhos de conclusão de curso, que serviram como valiosas referências para o desenvolvimento desta pesquisa.

Ao Benedito e ao Túlio, que contribuíram para a implantação da aplicação no servidor da universidade.

Aos professores, por todos os conhecimentos passados ao longo da minha trajetória acadêmica.

Aos meus colegas de graduação, pelo companheirismo e apoio durante a vida acadêmica.

# Resumo

O alto investimento das organizações no desenvolvimento de sistemas cada vez mais robustos e eficientes, associado ao aumento da exigência dos usuários por softwares de alta qualidade, promove a necessidade de processos eficientes de garantia da qualidade. Tendo isso em vista, este trabalho teve como objetivo a aplicação sistemática de técnicas de testes de *software* no aplicativo Clubes de Leitura – uma aplicação idealizada em uma dissertação de mestrado do PROFNIT/UFMA e desenvolvida em trabalhos de conclusão de curso da Engenharia da Computação/UFMA. Para isso, foram implementados testes automatizados, incluindo testes funcionais da API e testes end-to-end da interface do usuário, além da definição de um plano de testes. O trabalho ainda abrangeu a implementação de uma esteira de integração e entrega contínua (CI/CD) utilizando ferramentas como *Cypress*, *Rest Assured* e *Jenkins*. Na execução do projeto, aplicaram-se práticas de engenharia de testes fundamentadas por autores como Delamaro, Pressman, Sommerville e os materiais de estudo para certificação do ISTQB. Como resultado, o processo de desenvolvimento da aplicação tornou-se mais confiável e estruturado, com ganhos significativos em qualidade e rastreabilidade, alcançando aproximadamente 30% de cobertura dos casos de uso e 43% de cobertura dos requisitos funcionais da aplicação. No entanto, tendo em vista a limitação do tempo para a implementação de testes unitários, sugere-se a condução de um estudo de caso para a aplicação deste e de outros tipos e níveis de teste.

**Palavras-chave**: Qualidade de Software; Testes Automatizados; Testes Funcionais; Testes End-to-End; Clubes de Leitura.

# Abstract

The significant investment by organizations in developing robust and efficient systems, combined with the increasing demand from users for high-quality software, necessitates efficient quality assurance processes. With this in mind, this work aimed to apply systematic software testing techniques to the Clubes de Leitura application – a system originally proposed in a PROFNIT/UFMA master's dissertation and developed in undergraduate theses from the Computer Engineering program at UFMA. To achieve this, automated tests were implemented, including functional API tests and end-to-end tests of the user interface, along with the definition of a test plan. The work also includes the implementation of a continuous integration and continuous delivery (CI/CD) pipeline using tools such as Cypress, Rest Assured, and Jenkins. Throughout the project, software testing engineering practices were applied based on the work of authors such as Delamaro, Pressman, Sommerville, and materials used in ISTQB certification preparation. As a result, the application's development process became more reliable and structured, with significant gains in quality and traceability, achieving approximately 30% coverage of use cases and 43% coverage of the application's functional requirements. However, given the limited time available for the implementation of unit tests, a case study is suggested for the application of this and other types and levels of testing.

**Keywords**: Software Quality; Automated Testing; Functional Testing; End-to-End Testing; Book Clubs.

# Lista de ilustrações

# Lista de tabelas

# Lista de abreviaturas e siglas

| | |
|---|---|
| ABES | *Associação Brasileira das Empresas de Software* |
| API | *Application Programming Interface* |
| CD | *Continuous Delivery* |
| CI | *Continuous Integration* |
| CSCW | *Computer Supported Cooperative Work* |
| CSR | *Client-Side Rendering* |
| CT | *Caso de Teste* |
| DDD | *Domain Drive Desing* |
| DNS | *Domain Name System* |
| DSL | *Domain-Specific Language* |
| HTTP | *Hypertext Transfer Protocol* |
| HTTPS | *Hypertext Transfer Protocol Secure* |
| IEC | *International Electrotechnical Commission* |
| IEEE | *Institute of Electrical and Electronics Engineers* |
| IP | *Internet Protocol* |
| ISO | *International Organization for Standardization* |
| JRE | *Java Runtime Environment* |
| JSON | *JavaScript Object Notation* |
| NTI | *Núcleo de Tecnologia da Informação* |
| POO | *Programação Orientada à Objetos* |
| PR | *Pull Request* |
| RF | *Requisitos Funcionais* |
| REST | *Representational State Transfer* |

| | |
|---|---|
| RNF | *Requisitos Não Funcionais* |
| SO | *Sistema Operacional* |
| SOA | *Service-Oriented Architecture* |
| SPA | *Single-Page Application* |
| SQA | *Software Quality Assurance* |
| SSH | *Secure Shell Protocol* |
| SSL | *Secure Sockets Layer* |
| TI | *Tecnologia da Informação* |
| UFMA | *Universidade Federal do Maranhão* |
| UML | *Unified Modeling Language* |
| VPN | *Virtual Private Network* |
| VV&T | *Validação, Verificação e Teste* |

# Sumário

# 1 Introdução

A construção de um *software* é de certa forma complexa, há várias etapas durante o desenvolvimento e são executadas, muitas vezes, em curtos períodos de tempo. Logo, é compreensível que os *softwares* apresentem erros, falhas e inconsistências.

Nos últimos anos, a Engenharia de *Software* sofreu uma evolução significativa ao buscar definir padrões, métodos, técnicas, critérios e ferramentas que auxiliam na produção de um sistema de *software*. Essa evolução se deve ao fato da crescente utilização de sistemas com base na computação em praticamente todas as áreas da vida humana, o que, por sua vez, exige um crescimento na demanda por qualidade e produtividade (DELAMARO et al., 2007).

> A engenharia de *software* é uma disciplina relacionada a todos os aspectos da produção de *software*, desde os estágios iniciais da especificação até a manutenção depois que o sistema passa a ser usado (SOMMERVILLE, 2019, p.7).

Há várias atividades no contexto do processo de desenvolvimento de *software* que, embora sejam empregadas técnicas, métodos e ferramentas, ainda podem ocorrer erros no produto. Para mitigar esses riscos, são introduzidas atividades de Garantia de Qualidade de Software, como verificação, validação e testes, com a finalidade de minimizar erros e riscos associados ao desenvolvimento. A atividade de teste consiste na análise do *software* e tem grande relevância para a identificação e eliminação de erros (DELAMARO et al., 2007). Segundo Delamaro et al. (2007), o teste de um *software* está dividido essencialmente em quatro partes: *planejamento de testes, projeto de casos de teste, execução dos testes e avaliação dos resultados dos testes*.

Segundo a ABES (2023), o crescimento do investimento em TI alcançou US$ 3,11 trilhões em escala global em 2022, e US$ 45,9 bilhões — cerca de R$ 247,4 bilhões — no Brasil. Com isso, as empresas têm apostado em uma etapa fundamental em projetos de desenvolvimento de *software*: testes como parte da garantia de qualidade. Dessa forma, é possível encontrar falhas e erros para que sejam tratados antes que cheguem ao usuário final, evitando prejuízos de valores elevados.

> Os testes pretendem mostrar que um programa faz o que foi destinado a fazer e descobrir defeitos antes que ele seja colocado em uso (SOMMERVILLE, 2019, p. 203).

Atualmente, a etapa de testes está presente em todo ciclo de desenvolvimento de *softwares* a fim de garantir a qualidade desde as fases iniciais desse processo, como, por exemplo, na captação dos requisitos funcionais e não funcionais do *software*.

Portanto, as técnicas de testes podem, e devem, ser amplamente aplicadas para garantir que os *softwares* em desenvolvimento tenham qualidade e que não ocorram falhas quando entrarem em produção e forem utilizados pelos usuários finais. Além disso, os *softwares* também ficam obsoletos como qualquer produto de engenharia. Consequentemente, as manutenções também serão necessárias durante o ciclo de vida de um *software*, para além da correção de *bugs* encontrados por usuários, e abrangendo a garantia de um *software* fácil de manter e entender, ao longo do tempo (VALENTE, 2020).

## 1.1 Justificativa

Mesmo com o crescimento e evolução da área de testes automatizados em função da qualidade e praticidade que agrega ao desenvolvimento e ao *software*, como produto final (LUO, 2001), e das vantagens desta área destacadas na literatura, é muito comum haver projetos de *software* que não implementam a automação de testes por alguns fatores (RAFI et al., 2012), sendo um deles recursos financeiros limitados, além do curto prazo de entrega, já citado anteriormente.

Segundo Luo (2001) os testes consomem cerca de 40% a 50% dos esforços durante o desenvolvimento e esse esforço se torna maior em sistemas, onde a exigência por confiabilidade é alta, fazendo o teste ser uma parte significativa da engenharia de *software*. Ao passo que as manutenções e atualizações nos sistemas aumentam, a quantidade de testes aumenta significativamente para verificar os sistemas após a implantação das alterações. Portanto, o teste é um fator importante para avaliar a qualidade de um determinado sistema de *software*.

Com base no exposto, este projeto de pesquisa visa aplicar testes automatizados e manuais em um aplicativo *web* desenvolvido por alunos da UFMA denominado Clubes de Leitura. Esse aplicativo tem como objetivo a criação de clubes de leitura para que acadêmicos e leitores em geral se reúnam virtualmente, além do incentivo à leitura (OLIVEIRA, 2023) e à colaboração (LEMOS, 2023).

## 1.2 Objetivos

### 1.2.1 Objetivo Geral

Objetivo geral deste trabalho consiste em aplicar práticas sistemáticas de testes de *software* no aplicativo Clubes de Leitura, com ênfase na automação de testes por meio da implementação de uma pipeline de integração e entrega contínua.

### 1.2.2 Objetivos Específicos

- Analisar as regras de negócio do aplicativo

- Utilizar tecnologias modernas para implementar testes automatizados
- Validar as implementações dos requisitos funcionais do aplicativo
- Identificar e propor pontos de melhorias para aprimorar a experiência do usuário

## 1.3 Estrutura do Trabalho

O capítulo 2 apresenta a referencial teórico e tecnológico deste projeto. Neste capítulo, os principais conceitos para a aplicação da metodologia adotada são abordados.

O capítulo 3 faz parte do referencial teórico e traz informações importantes sobre a aplicação alvo deste trabalho.

O capítulo 4 apresenta a metodologia adotada para a pesquisa. Cada etapa do processo será explicada nas seções do presente capítulo.

O capítulo 5 descreve como foram executados e aplicados os passos da metodologia, bem como conceitos adicionais sobre a aplicação e ferramentas utilizadas durante o desenvolvimento.

O capítulo 6 apresenta os resultados e análises realizadas durante a pesquisa.

O capítulo 7 apresenta as conclusões do trabalho, bem como a aplicabilidade destas.

# 2 Referencial Teórico e Tecnológico

Este capítulo tem como objetivo esclarecer os leitores quanto aos conceitos relacionados a este trabalho. Então, faz-se necessária a exposição da fundamentação teórica e tecnológica utilizada neste projeto. Como cada *software* é projetado para resolver problemas específicos, buscou-se estudar os tipos e técnicas de testes para entender quais se encaixam no *software* alvo deste trabalho. Adicionalmente, serão apresentados os conceitos de CI e CD e como sua implementação auxilia durante o ciclo de desenvolvimento de um *software*.

## 2.1 Teste, Qualidade de Software e Garantia de Qualidade de Software

*Teste* consiste na execução de um sistema contendo um conjunto finito de casos, a fim de verificar se o sistema possui o comportamento esperado (VALENTE, 2020). Além disso, há uma frase frequentemente atribuída a Edsger W. Dijkstra que diz: *"Testes de software mostram a presença de bugs, mas não a sua ausência."* [1], ou seja, os testes trazem o benefício de encontrar bugs e falhas, mas também há limitações em não mostrar a ausência deles.

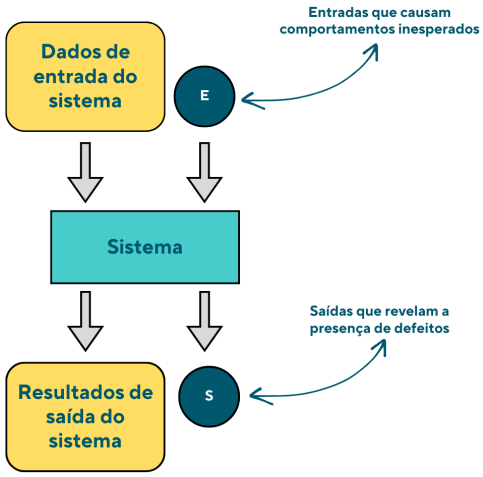
Na Figura 1 temos a ilustração dos *teste de validação*, que são os testes cujo espera-se que o sistema funcione de maneira correta quando este é submetido a casos de testes que mostram o uso previsto do sistema (SOMMERVILLE, 2019), e *teste de defeitos* que, segundo Sommerville (2019), são os testes onde os casos de teste são escritos para que seja possível revelar defeitos existentes no sistema. Apesar da diferença conceitual, não há um limite exato entre ambos os testes (SOMMERVILLE, 2019), visto que os testes de validação podem resultar em defeitos, e durante a execução de alguns testes de defeitos o sistema pode atender aos seus requisitos.

O *software* testado deve ser pensado como uma caixa, onde é aceito um conjunto de entradas e gera um conjunto de saídas(veja a Figura 1). Algumas dessas saídas estarão erradas($S$) e serão geradas pelo conjunto de entradas($E$). Sommerville (2019) explica que a prioridade do teste de defeitos é encontrar as entradas do conjunto $E$ que causam defeitos, pois evidenciam os problemas do sistema.

A *Qualidade de Software* é uma área de estudo que faz parte do processo de desenvolvimento de *software*, onde padrões são documentados com as características implícitas esperadas de todo *software* desenvolvido profissionalmente. Além disso, é avaliada a conformidade com requisitos funcionais, visuais e de desempenho do sistema. Pressman (2011) define qualidade de *software* como uma gestão de qualidade efetiva aplicada de modo a criar um

[1] DIJKSTRA, E. W. *"The humble programmer"* Comm. ACM. v.15, n. 10, 1972. p. 859-66. doi: 10.1145/355604.361591.

Figura 1 – Modelo de entrada e saída de teste de um programa




Fonte: Adaptada de (SOMMERVILLE, 2019, p. 204)

produto útil que forneça valor mensurável para aqueles que o produzem e para aqueles que o utilizam.

Já a *Garantia de Qualidade de Software* (SQA), pode ser definida como o conjunto de atividades técnicas aplicadas durante todo o processo de desenvolvimento de *software*. A SQA tem como objetivo garantir que tanto o processo de desenvolvimento quanto o produto de *software* atinjam os níveis de qualidade especificados (PRESSMAN, 2011).

## 2.2 Técnica VV&T

Delamaro et al. (2007) explicam que, apesar de muitos fatores serem identificados como causas de problemas, a maioria deles tem origem a partir do erro humano. Portanto, para auxiliar na descoberta de problemas e erros antes que o *software* seja liberado para utilização, existem algumas atividades usualmente chamadas de Técnica VV&T, que significa Validação, Verificação e Teste, e sua definição engloba várias das atividades relacionadas à SQA.

Segundo Felizardo (2010), a validação busca assegurar que o produto final corresponda aos requisitos planejados para o *software*, geralmente respondendo à pergunta: Estamos construindo o produto certo? Já a verificação tem como objetivo garantir a consistência, completude e corretude do produto em cada fase e entre fases do ciclo de vida do *software*, com a questão central sendo: Estamos construindo corretamente o produto? Por fim, o teste é definido como a atividade focada em examinar o comportamento do produto por meio de sua execução, seguindo passos predeterminados.

Delamaro et al. (2007) falam, também, que as atividades de VV&T não se restringem ao produto final. Mas, podem e devem ser utilizadas durante todo o processo de desenvolvimento do *software*, desde a sua concepção, durante a coleta dos requisitos.

## 2.3 Testes Exploratórios

Independentemente da quantidade de casos que temos no nosso conjunto de testes, ainda nos deparamos com situações inesperadas à medida que nos afastamos dos *scripts* planejados. É necessário lembrar que o usuário faz coisas fora do planejado, que os dados em produção podem e serão muito diferentes dos dados de um ambiente de testes. Logo, é inviável planejar casos de testes com antecedência para cobrir todas as possibilidades dentro do *software*.

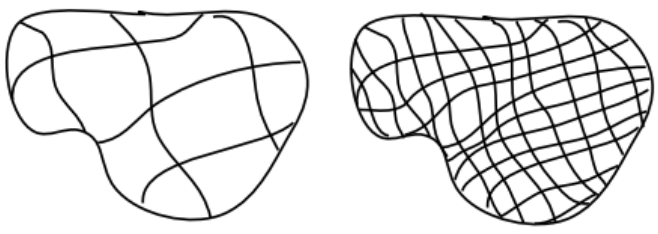
Durante o desenvolvimento de um *software* existem vários dados, variações de configurações, interações e, principalmente, o tempo. Se tentarmos desenvolver um conjunto de testes para contemplar todas as alternativas, acabaremos consumindo todo o tempo escrevendo infinitos casos de teste, deixando-nos sem tempo para sua execução.

Hendrickson (2013) explica que não é necessário um conjunto de casos de teste perfeito, mas uma estratégia de teste que responda às perguntas:

1. O software se comporta conforme as condições as quais ele deveria ser capaz de lidar?
2. Há outros riscos?

Para responder a primeira pergunta, podemos usar o conceito de *checking*, ou verificação, que nos ajuda a garantir que um *software* se comporta como ele foi planejado. Hendrickson (2013) compara os testes a uma rede de dispositivos de detecção, que são acionados sempre que o comportamento do software não está de acordo com as expectativas.

Figura 2 – Redes de testes.



$(a)$ $(b)$

Fonte: (HENDRICKSON, 2013, p.5).

Alguns conjuntos de testes podem não ser tão abrangentes quanto outros(Figura 2.a). Indicando que eles podem não cobrir todas as possíveis condições ou cenários de uso do *software*. Assim, podem haver lacunas na detecção de comportamentos inesperados ou erros e

defeitos no *software*. Quanto mais abrangentes forem os testes, mais detalhada será essa rede de detecção(Figura 2.b), ajudando a identificar e corrigir problemas de forma mais eficaz. Contudo, apesar de ter construído uma boa rede, ainda é necessário responder à segunda pergunta. E é nesse ponto que entra a exploração.

Segundo Hendrickson (2013), os testes exploratórios são focados nas áreas que a rede não cobre. Esse teste consiste em interagir com o sistema executando pequenos experimentos e usando os resultados do anterior como entrada para o próximo experimento. Enquanto a interação ocorre, são descobertos potenciais riscos, então investiga-se tais riscos mais a fundo. Nesta etapa, usa-se a capacidade de observação e análise para adaptar a investigação em tempo real. Esses experimentos fornecem evidências sobre recursos e limitações do *software* sob teste, e assim, descobre-se novas perguntas que precisam de respostas e planeja-se tipos adicionais de testes.

> A essência do teste é projetar um experimento para reunir evidências empíricas para responder a uma pergunta sobre um risco (HENDRICKSON, 2013, p.3).

A exploração proporciona um método para percorrer as inúmeras variações possíveis, permitindo uma abordagem mais adaptável para encontrar riscos que os testes pré-planejados. A variação é mais eficiente que a repetibilidade quando o assunto é descobrir surpresas adicionais (HENDRICKSON, 2013).

## 2.4 Testes Automatizados

Conforme apresentado na Seção 2.1, os testes desempenham um papel essencial na garantia da qualidade do *software*. No entanto, a execução manual de determinados tipos de testes pode comprometer a consistência e a confiabilidade do processo de teste. Isso acontece pois, durante a execução, há a possibilidade de se esquecer etapas relevantes, impactando negativamente o resultado final.

Segundo o ISTQB (2024), a automação de testes abrange a realização de testes automatizados, bem como a geração de relatórios de testes. Além disso, inclui o uso de *software* construídos exclusivamente para a implementação e execução desses testes.

Os testes automatizados são programas que são executados a cada vez que o *software* em construção precisa ser testado. Eles são mais rápidos que os testes manuais, sobretudo quando se trata de testes de regressão — explicados na Seção 2.6.3. No entanto, a automação só consegue validar se um programa faz o que foi projetado para fazer, logo, é muito difícil que um teste automatizado seja usado em sistemas que dependem de aspectos visuais, como a interface gráfica com o usuário, ou para verificar se o sistema apresenta efeitos colaterais imprevistos (SOMMERVILLE, 2019).

O ISTQB (2024), lista uma série de vantagens da automação de testes, dentre elas:

1. Proporciona a execução de mais testes em relação a testes manuais.
2. Habilita a realização de testes mais complexos do que os manuais.
3. O tempo de execução é mais rápido comparado aos testes manuais.
4. A ocorrência de erros humano é reduzida.
5. O retorno sobre a qualidade do *software* é produzido mais rapidamente.

Ainda que a automação de testes ofereça vantagens consideráveis, como salientado anteriormente, segundo o ISTQB (2024), há desvantagens que devem ser consideradas, tais como:

1. Custos adicionais na contratação de profissionais especializados e aprimoramento da infraestrutura.
2. Maior tempo necessário no desenvolvimento e manutenção da automação de testes.
3. Menor adaptabilidade dos testes automatizados às mudanças no decorrer do desenvolvimento e evolução do *software*.
4. Descoberta de mais defeitos pelos testes automatizados, aumentando o tempo para correção.

Além disso, é fundamental considerar as limitações dos testes automatizados. Nem todos os testes manuais podem ser automatizados, já que a automação está restrita a verificar apenas o que foi programado para testar. Algumas características de qualidade podem não ser testadas adequadamente com uma automação, pois os testes automatizados só conseguem verificar resultados interpretáveis por máquinas (ISTQB, 2024).

Do ponto de vista da qualidade de *software*, testes automatizados são muito importantes, visto que, a longo prazo, tais testes diminuem o custo com a execução e melhoram o processo de qualidade. Consequentemente, testes automatizados podem ser aplicados no *software* alvo, redirecionando o esforço da execução de testes para a melhoria da análise e avaliação do *software* a ser testado (REITZ et al., 2013).

## 2.5 Níveis de Teste

Durante o desenvolvimento de um *software* existem diferentes tipos de testes e eles são realizados em diferentes níveis. Os níveis de teste são grupos de atividades organizadas e geridas em conjunto e cada nível é uma instância do processo de teste, realizado em relação ao *software* em uma determinada etapa do desenvolvimento (ISTQB, 2023). Porém, Delamaro et al. (2007)

chama os níveis de testes de fases da atividade de teste, estabelecendo o teste de unidade, o teste de integração e o teste de sistemas como cada um uma fase.

### 2.5.1 Teste de Unidade

Também conhecido como teste de componente, ou teste unitário. São os testes focados em testar os componentes isoladamente (PRESSMAN, 2011). Sommerville (2019) explica que no teste de unidade são testadas partes do código, como métodos e classes. Normalmente, exige um suporte específico, como *frameworks* de testes unitários. Os testes de componentes são realizados, normalmente, pelos desenvolvedores no ambiente de desenvolvimento (ISTQB, 2023).

### 2.5.2 Teste de Integração

O teste de integração verifica se todos os componentes, após testados, funcionam quando estão interligados (ISTQB, 2023). É uma técnica sistemática na construção da arquitetura do *software*, além de realizar testes que possam identificar possíveis erros de interface, com a finalidade de desenvolver a estrutura do programa estabelecida pelo projeto a partir dos componentes testados em unidade (PRESSMAN, 2011).

### 2.5.3 Teste de Sistema

O teste de sistema concentra-se nos recursos e no comportamento do *software* como um todo. Ele verifica se os componentes são compatíveis, se interagem corretamente e se transferem os dados certos no momento certo por meio de suas interfaces (SOMMERVILLE, 2019). O teste de sistema pode ser executado por uma equipe de teste independente e é relacionado às especificações do *software* (ISTQB, 2023).

### 2.5.4 Teste de Aceite

Concentra-se na validação para a implantação, o que significa que o sistema atende aos requisitos de negócio do usuário (ISTQB, 2023). Um teste de aceitação pode se estender por semanas ou até meses, descobrindo erros cumulativos que poderiam degradar o sistema ao longo do tempo (PRESSMAN, 2011). Geralmente, são executados pelos usuários finais da aplicação.

## 2.6 Tipos de Teste

Os tipos de teste são grupos de atividades de teste destinadas a testar características específicas de um *software* (ISTQB, 2023), e a maioria dessas atividades pode ser realizadas em todos os níveis de teste.

### 2.6.1 Teste Caixa-Branca

Segundo Sommerville (2019), são testes baseados no conhecimento da estrutura do programa e seus componentes. Logo, o acesso ao código-fonte do *software* é essencial para esse tipo de teste. Pressman (2011) também chama de *teste da caixa-de-vidro*, e explica que ao aplicar esse tipo de teste, é possível criar casos de teste que garantam que todos os caminhos independentes de um módulo foram executados ao menos uma vez, exercitam todas as decisões lógicas nos seus estados verdadeiro e falso, executam todos os ciclos em seus limites, e exercitam estruturas de dados internas para assegurar a sua validade.

O teste de caixa-branca geralmente é um teste unitário, já que os responsáveis são os desenvolvedores e eles detêm o conhecimento do código-fonte do componente testado. A Figura 3 ilustra o sistema como uma caixa branca, e denota a transparência do código-fonte ao testador.

Figura 3 – Teste Caixa-Branca


Fonte: Elaborado pelo autor

### 2.6.2 Teste Caixa-Preta

Teste caixa-preta, também chamado de *teste comportamental*, é focado nos requisitos funcionais do *software* (PRESSMAN, 2011). É uma abordagem na qual os testadores não possuem acesso ao código-fonte e são derivados das especificações do *software* (SOMMERVILLE, 2019). Sommerville (2019) dá outro nome para esse teste: *teste funcional*, pois os testadores se preocupam apenas com a funcionalidade do *software* testada, e não com a sua implementação. O principal objetivo do teste caixa-preta é verificar o comportamento do sistema em relação às suas especificações (ISTQB, 2023).

A Figura 4 ilustra bem o que é o teste caixa-preta. O responsável pelo teste não tem ideia do que está por trás do sistema; para ele, o sistema é como uma caixa escura que não revela seus segredos, ou seja, o testador não tem acesso ao código-fonte do *software*.

Figura 4 – Teste Caixa-Preta


Fonte: Elaborado pelo autor

### 2.6.3 Teste de Regressão

São testes aplicados para comprovar que alterações no *software* não geraram defeitos ou falhas, até mesmo uma correção que já tenha sido aprovada após o teste. Esses defeitos e falhas podem impactar o componente que sofreu alteração, outros componentes do mesmo *software* e ainda outros sistemas conectados (ISTQB, 2023).

Delamaro et al. (2007) enfatizam a importância da realização de testes após qualquer modificação, salientando:

> A cada modificação efetuada no sistema, após a sua liberação, corre-se o risco de que novos defeitos sejam introduzidos. Por esse motivo, é necessário, após a manutenção, realizar testes que mostrem que as modificações efetuadas estão corretas, ou seja, que os novos requisitos implementados (se for esse o caso) funcionam como o esperado e que os requisitos anteriormente testados continuam válidos. (DELAMARO et al., 2007, p.4).

Os testes já executados e aprovados, sendo eles de qualquer tipo, podem descobrir novos erros após a adição de uma nova funcionalidade, e esses erros precisam ser corrigidos. A cada correção no *software*, algum elemento de configuração é alterado: o código-fonte, a documentação, os dados, etc. O teste de regressão endossa que as mudanças decorrentes das correções não geraram novos erros indesejados (PRESSMAN, 2011).

Segundo o (ISTQB, 2023), os testes de regressão são grandes candidatos a serem automatizados. Dois motivos ajudam a entender a escolha de automatizar esse tipo de teste:

1. O conjunto de testes de regressão é executado inúmeras vezes, transformando-o em uma tarefa repetitiva, justificando a automação.

2. Comumente, a quantidade de casos de teste aumenta a cada versão lançada, sendo muito oneroso executar manualmente.

## 2.7 Planejamento do Teste

Sommerville (2019) explica que o *planejamento do teste* consiste em um cronograma e definições dos recursos para todas as atividades a serem desenvolvidas no processo de teste, levando em consideração desde a definição do processo até todas as pessoas envolvidas e o tempo disponível. Assim, um plano de teste é criado, definindo o que será testado, o cronograma e como os testes serão documentados.

Para Pressman (2011), o plano de teste contém um conjunto de tarefas de teste, os artefatos a serem desenvolvidos e a maneira como os resultados devem ser avaliados, documentados e reutilizados.

Durante o desenvolvimento de um *software*, é crucial que se faça o planejamento de testes. Pois, além de estabelecer o cronograma e os procedimentos de teste, o plano de teste descreve, também, os recursos de *hardware* e *software*. Sendo vital para que os gerentes do projeto possam garantir que esses recursos estejam disponíveis para o time de testes.

Os planos de teste não são documentos estáticos, mas evoluem durante o processo de desenvolvimento (SOMMERVILE, 2019). Sendo assim, os planos de teste podem sofrer alterações devido a atrasos nas outras etapas do desenvolvimento. Se uma funcionalidade do *software* ainda estiver em desenvolvimento, então não se pode testar o *software* como um todo. Consequentemente, o plano de testes deverá ser revisado e alterado, realocando os testadores e alterando os prazos, se necessário (SOMMERVILE, 2019).

### 2.7.1 Estrutura do Plano de Teste

O plano de testes varia de acordo com o projeto, equipe de desenvolvimento e organizações, mas Sommervile (2019) define uma estrutura completa e organizada para o plano de testes:

1. **O processo de teste**: Uma descrição das principais fases do processo de teste do sistema. Isso pode ser dividido em testes de subsistemas individuais, testes de interfaces de sistemas externos, etc.

2. **Rastreabilidade de requisitos**: Os usuários estão mais interessados em que o sistema atenda aos seus requisitos e os testes devem ser planejados para que todos os requisitos sejam testados individualmente.

3. **Itens testados**: Os produtos do processo de software que serão testados devem ser especificados.

4. **Cronograma de testes**: Um cronograma geral de testes e alocação de recursos. Este cronograma deve estar vinculado ao cronograma mais geral de desenvolvimento do projeto.

5. **Procedimentos de evidências dos testes**: Não basta simplesmente executar testes; os resultados dos testes devem ser registados sistematicamente. Deve ser possível auditar o processo de teste para verificar se foi realizado corretamente.

6. **Requisitos de hardware e software**: Esta seção deve definir as ferramentas de software necessárias e a utilização estimada de hardware.

7. **Restrições ou Riscos**: As restrições que afetam o processo de testes, tais como a falta de pessoal, devem ser previstas nesta secção.

8. **Testes de sistema**: Esta seção, que pode ser completamente separada do plano de testes, define os casos de testes que devem ser aplicados ao sistema. Esses testes são derivados da especificação de requisitos do sistema.

## 2.8 Erro, Defeito e Falha

Os seres humanos cometem erros (equívocos), que produzem defeitos(*bugs*) que, por sua vez, podem resultar em falhas (ISTQB, 2023). Os erros do ser humano têm diversas origens, seja por pressão de tempo, complexidade dos processos, infraestrutura, ou apenas por estar cansado ou por não ter o devido treinamento.

Os defeitos podem ser encontrados na documentação, pode estar em uma especificação de algum requisito, pode estar num *script* de teste, no código-fonte, ou até em um objeto de suporte, como um arquivo de compilação. Ao executar o código, e este tiver um defeito, o sistema pode não fazer o que deveria fazer ou fazer algo que não é o esperado, resultando numa falha. Há defeitos que sempre que forem executados resultarão em falhas, há os que resultam em falhas apenas em circunstâncias específicas, e por sua vez, há aqueles que podem nunca resultar em falha.

Para conceituar erro, defeito e falha, Delamaro et al. (2007) cita o padrão IEEE número 610.12-1990, que diz que um **defeito**(*fault*) é um passo, processo ou definição de dados incorreta, por exemplo, uma instrução ou comando incorreto. Diz ainda que **erro**(*error*) é a diferença entre o valor obtido e o valor esperado, assim sendo, qualquer resultado inesperado na execução do código constitui um erro. E que **falha**(*failure*) é a produção de uma saída incorreta com relação às especificações.

O ISTQB (2023) também diz que as falhas nem sempre têm erros e defeitos como causa. E que condições ambientais, como a radiação ou campo eletromagnético, quando causam defeitos no *firmware*, podem resultar em falhas.

## 2.9 Gherkin

*Gherkin*[2] é uma linguagem utilizada para estruturar textos que definem uma regra de negócio ou um cenário de execução. Conta com sintaxe e semântica simples e define padrões para documentar os cenários de uso de um *software*. Assim, em vez de escrever textos longos ou uma lista extensa de requisitos e funcionalidades, opcionalmente usa-se *Gherkin* para escrever textos mais simples, em linguagem natural e que pode ser entendidos por toda a equipe envolvida no projeto, desde o time de desenvolvimento até o time de negócio.

Na **Figura 5** temos dois cenários de testes para a funcionalidade de *saque*, o primeiro cenário(linha 3) descreve os passos para um *cenário positivo*, onde o usuário consegue realizar o saque do valor solicitado. No segundo cenário(linha 9), temos um *cenário negativo*, nele o usuário solicita um valor maior que o seu saldo disponível, logo terá o saque negado. De forma clara e simples, é possível entender a funcionalidade descrita e os passos para executá-la.

Figura 5 – Cenários de teste escrito em Gherkin

```
1  Funcionalidade: Saque
2
3      Cenario: Saque com sucesso
4          Dado que eu tenho uma conta com saldo de R$ 300,00
5          Quando eu faço um saque no valor de R$ 100,00
6          Então eu vejo a mensagem "Saque realizado com sucesso"
7          E meu saldo final será de R$ 200,00
8
9      Cenario: Saldo insuficiente para saque
10         Dado que eu tenho uma conta com saldo de R$ 50,00
11         Quando eu faço um saque no valor de R$ 51,00
12         Então eu vejo a mensagem "Saldo insuficiente"
13         E meu saldo final será de R$ 50,00
```

Fonte: Elaborado pelo autor

*Gherkin* utiliza algumas palavras-chave para organizar a estrutura do texto e dar significado ao cenário. Algumas dessas palavras e seus significados estão listadas abaixo:

- **Funcionalidade**(*Feature*): é a descrição da funcionalidade/recurso e pode agrupar vários cenários.
- **Cenário**(*Scenario*): é o título de um cenário possível.
- **Dado**(*Given*): especifica uma pré-condição a ser cumprida antes da execução dos passos do cenário.
- **Quando**(*When*): é a ação que executa-se e espera-se uma resposta do sistema.

[2] Disponível em: https://cucumber.io/docs/gherkin/reference/

- **Então**(*Then*): usado para descrever um resultado ou um resultado esperado. Aqui comparamos o resultado atual com o resultado esperado.
- **E**(*And*): se for necessário muitas interações para completar um fluxo, basta acrescenter o passo usando "E".
- **Mas**(*But*): serve para a mesma funcionalidade do "E", mas normalmente é utilizado após uma validação negativa depois do "Então".

Apesar de inicialmente ter sido criado em inglês, o *Gherkin* possui tradução oficial para mais de 70 idiomas [3].

## 2.10 Integração Contínua (CI)

Em muitos projetos de *softwares*, é comum que, durante o desenvolvimento, a aplicação não esteja funcional por longos intervalos de tempo. Segundo Humble e Farley (2010), isso ocorre porque ninguém tenta executar a aplicação por completo até que ela esteja finalizada. Ainda que os desenvolvedores façam alterações no código e executem testes unitários automatizados, dificilmente executam a aplicação em um ambiente semelhante ao de produção para verificar se o sistema está funcionando corretamente.

Esse problema se intensifica em projetos que utilizam *branches* de longa duração ou deixam os testes de aceitação para o final. Nesses casos, são agendadas longas fases de integração para que os desenvolvedores tenham tempo hábil para mesclar suas alterações e garantir que o projeto esteja funcional durante a fase de testes de aceitação. Porém, essa abordagem pode mostrar que o *software* não é exatamente o que foi planejado apenas no final do desenvolvimento.

A Integração Contínua ou *Continuous Integration* (CI) surge para auxiliar o desenvolvedor a mesclar suas alterações ao repositório central sem precisar executar a aplicação todas as vezes que realizar um *commit*. A CI requer que toda vez que alguém fizer qualquer alteração, o *software* inteiro seja construído e um conjunto abrangente de testes automatizados seja executado nele (HUMBLE; FARLEY, 2010).

> The goal of continuous integration is that the software is in a working state all the time. [4] (HUMBLE; FARLEY, 2010, p.55).

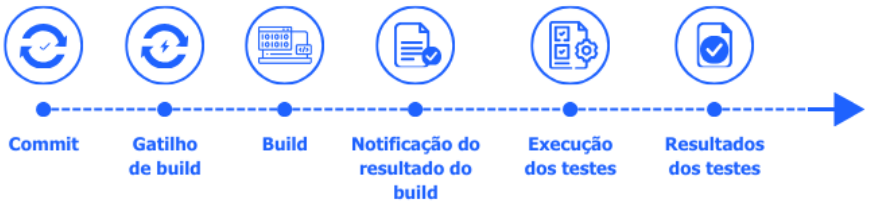
Equipes de desenvolvedores que usam a CI têm alguns benefícios em relação a equipes que não utilizam. Tais benefícios são elencados por Humble e Farley (2010, p.56):

- Auxiliam na rapidez da integração das alterações realizadas pelos desenvolvedores

[3] Disponível em: https://cucumber.io/docs/gherkin/languages/
[4] "O objetivo da integração contínua é que o software esteja sempre em um estado funcional." (tradução: autor)

- A detecção de bugs ocorre mais rápido e mais cedo ajudando a corrigir antes da entrega
- A CI otimiza o tempo de entregar do projeto de *software* e a entrega é feita muito mais rápida
- As entregas tem menos *bugs*

Figura 6 – Continuous Integration (CI)




Fonte: Elaborado pelo autor

A Figura 6 exemplifica a implementação de uma CI automatizada com etapas claras e objetivas. Na imagem, nota-se que tudo se inicia com um *commit*, ou seja, uma alteração realizada pelo desenvolvedor, que ativa o processo de CI passando pela execução do *build* da aplicação para que os testes automatizados sejam executados. Por fim, são gerados os resultados dos testes que podem ser analisados pela equipe de desenvolvimento em caso de falhas.

Com isso, nota-se que a CI é tão importante quanto o controle de versões e implementar um pipeline automatizado desobriga os desenvolvedores a executar o processo manualmente, assegurando que todas as alterações realizadas por eles sejam enviadas de maneira consistente às etapas estabelecidas.

## 2.11 Entrega Contínua (CD)

Uma prática importante que está associada à CI é a Entrega Contínua ou *Continuous Delivery* (CD), que tem como objetivo automatizar os processos de entrega e implantação da aplicação, habilitando a entrega rápida e estável de versões funcionais do *software*, além de permitir que o *software* seja implantado de maneira confiável e recorrente em vários ambientes.

Apesar disso, Humble e Farley (2010) salientam que há companhias que executam o processo de entrega de forma completamente manual, pois os aplicativos desenvolvidos possuem uma complexidade alta, acarretando na divisão da entrega em várias etapas que são delegadas a um time ou pessoa. Sabe-se que o processo de entrega manual está sujeito a falhas humanas, podendo causar falhas na implantação.

A Figura 7 exibe um exemplo das etapas executadas por um *pipeline* de Entrega Contínua. Observa-se que CD é uma extensão de CI onde realiza testes contínuos para verificar se o *software*

Figura 7 – Continuous Delivery (CD)


Fonte: Elaborado pelo autor

tem qualidade de produção por meio de um processo de lançamentos automatizado (ITKONEN et al., 2016).

Além da prática de entrega contínua ainda há uma prática chamada *Deployment* Contínuo(ou *Continuous Deployment*) que também pode ser abreviado por CD. Os conceitos de ambos os CD são bem próximos, como explica Valente (2020):

> *Deployment* é o processo de liberar uma nova versão de um sistema para seus usuários. *Delivery* é o processo de liberar uma nova versão de um sistema para ser objeto de *deployment*. (VALENTE, 2020, sec.10.4.1).

Valente (2020) ainda explica que quando é adotado o *Deployment* Contínuo, ambos os CD são automatizados e contínuos. Entretanto, com a implementação da Entrega Contínua, a entrega é frequente, mas o *deployment* é realizado de forma que depende de uma autorização manual.

Itkonen et al. (2016) elenca vários benefícios ao implementar CD no desenvolvimento de *software*, tais como:

- Aumento da produtividade já que a automação diminuiu o esforço do desenvolvedor necessário para execução de tarefas repetitivas.
- Melhor qualidade de entrega, pois a automação de testes garante menos bugs após a implantação
- Redução dos riscos de falhas de lançamentos devido a testes automatizados e ao acesso dos desenvolvedores a um ambiente semelhante ao de produção onde podem reduzir o risco de erros que podem ser descobertos apenas quando o *software* estiver em execução no ambiente de produção.

Ambos os conceitos de CD, seja a Entrega Contínua ou o Deployment Contínuo, são grandes aliados dos desenvolvedores, pois, novamente, eliminam a necessidade de o desenvolvedor realizar operações complexas manualmente, evitando o desperdício de tempo e

falhas durante o processo, podendo redirecionar esforços para a resolução de outros problemas relevantes.

## 2.12 *Pipeline*

Um *pipeline* é uma prática muito importante para uma implementação eficiente de Integração e Entrega Contínua. O *pipeline* define as etapas que um *software* percorre para ter uma nova versão funcional lançada em produção, focando na melhoria da entrega das aplicações no decorrer do ciclo de desenvolvimento de *software* através de automações (REDHAT, 2022).

Humble e Farley (2010) definem *pipeline* como sendo uma manifestação automatizada do processo de obtenção da aplicação do repositório de controle de versão para o uso por parte dos seus usuários em produção, onde as modificações realizadas no *software* passam por um processo complexo para serem lançadas.

Figura 8 – Exemplo de Pipeline de CI/CD


Fonte: RedHat (2022)

A Figura 8 apresenta um exemplo de estrutura de um pipeline. A estrutura completa é formada pela associação do CI e dos CD. Em geral, as etapas de um pipeline são (HUMBLE; FARLEY, 2010):

1. Construção do código: o primeiro estágio do pipeline ocorre quando um *commit* é realizado acionando um gatilho e criando uma nova instância do pipeline. Nesse estágio o código-fonte da aplicação é compilado, os testes unitários são executados, ocorre também a geração de instaladores.

2. Testes Automatizados: o próximo estágio é execução dos testes automatizados. Podem ser testes funcionais da API ou testes da interface do usuário, dependendo da aplicação.

3. Deploy em Ambientes de Homologação: Se os testes forem aprovados, a aplicação é disponibilizada no ambiente de testes. Em algumas pipelines, esta etapa ocorre após a aprovação manual.

4. Testes Pós-Deploy: neste estágio há a execução de mais testes automatizados, como testes de integração e validações específicas no ambiente implantado(nesse caso o ambiente de teste/homologação).

5. Deploy em Produção: Após todos os processos finalizarem com sucesso, a aplicação está pronta para ser lançada em produção. Está etapa também pode ser configurada para aguardar uma aprovação manual.

O pipeline de CI/CD configura esse processo, e sua modelagem em uma ferramenta de integração contínua e gerenciamento de lançamentos é o que permite que o desenvolvedor ou a equipe vejam e controlem os progressos de cada modificação à medida que ela passa do controle de versão, percorrendo vários conjuntos de testes e implantações, até a liberação para os usuários (HUMBLE; FARLEY, 2010).

# 3 Clubes de Leitura: uma aplicação *web*

O propósito deste capítulo é fornecer uma explicação abrangente aos leitores acerca do *Clubes de Leitura*, uma aplicação *web* desenvolvida por alunos da UFMA, que servirá como o foco principal das análises e testes de *software* deste trabalho. Portanto, é necessário apresentar seus objetivos, funcionalidades, requisitos e estrutura do banco de dados.

O *Clubes de Leitura* é definido por Alcântara (2022), em sua dissertação de mestrado, como uma ferramenta tecnológica para o gerenciamento, compartilhamento e incentivo à leitura. Ademais, o *Clubes de Leitura* é uma aplicação voltada para clubistas, fãs de leitura e acadêmicos, e tem como objetivos possibilitar a criação e o gerenciamento de comunidades literárias e/ou clubes, ser um incentivador à leitura e práticas literárias, além da possibilidade da captação de dados para uso de políticas públicas (OLIVEIRA, 2023). Adicionalmente, o aplicativo busca utilizar aspectos de sistemas colaborativos para que ofereça meios de facilitar a colaboração utilizando funcionalidades que apoiem a comunicação e a cooperação entre os membros e também a coordenação das tarefas do clubes (LEMOS, 2023).

## 3.1 Clube de Leitura

Clubes de leitura, também chamados de círculos de leitura, tertúlias literárias e grupos de leitura por Boccia (2012), são *"produtos de agregamentos sociais com necessidades ou propósitos próprios e até, não raramente, únicos"* (BOCCIA, 2012), e em geral, esses clubes têm como principal objetivo promover a apreciação da leitura e estimular discussões acerca de aspectos relevantes para o grupo.

A leitura pode ser vista como uma atividade individual, de fato, não é preciso mais que um livro para começar a ler, porém, pode ser realizada coletivamente por meio de clubes de leitura que *"configuram uma modalidade de prática leitora que se organiza em torno da partilha de textos e de pontos de vista sobre os mesmos"* (BOHM; MARANGONI, 2011). Resumidamente, os clubes são grupos de pessoas que possuem interesses literários em comum e realizam reuniões periódicas, com a liderança de um mediador que promove a organização e a participação democrática dos membros, além de fomentar discussões relevantes para as reuniões.

## 3.2 Requisitos do Sistema

Os requisitos de um sistema, segundo Sommerville (2019), são as descrições dos serviços que o sistema deve fornecer e as restrições que ditam a sua operação. Tais requisitos constituem um reflexo das necessidades dos clientes, indicando quais serviços o sistema deve ser capaz de

oferecer para cumprir um propósito específico, como o controle de dispositivos, processamento de pedidos ou recuperação de informações. A análise desses requisitos tem como resultado a especificação das características operacionais do *software* e revela a integração do *software* com outros elementos do sistema (PRESSMAN, 2011).

### 3.2.1 Requisitos Funcionais e Não Funcionais

Normalmente, os requisitos de um *software* são divididos em dois grupos: funcionais e não funcionais. Os RF são descrições do que o sistema deve fazer, e em alguns casos, do que não deve fazer. E devem descrever as funções do sistema, suas entradas, saídas e as exceções em detalhes (SOMMERVILLE, 2019). A Tabela 1 lista os Requisitos Funcionais da aplicação alvo deste trabalho.

Tabela 1 – Requisitos Funcionais da Aplicação

| # | Requisito |
|---|---|
| RF01 | Cadastro de todos os livros que o clube já leu |
| RF02 | Definição da agenda de encontros |
| RF03 | Definição da agenda de votações |
| RF04 | Criação de votação dos livros |
| RF05 | Cadastro de clube, incluindo política de ingresso |
| RF06 | Curadoria de novos clubistas |
| RF07 | Cadastro de clubistas |
| RF08 | Estante de livros do clubista |
| RF09 | Cadastro de livros |
| RF10 | Importação de base de livros |
| RF11 | Avaliação de livros |
| RF12 | Registro de progresso da leitura |
| RF13 | Cadastro de comentários sobre os livros |
| RF14 | Cadastro de notificação |

Fonte: (OLIVEIRA, 2023; LEMOS, 2023)

Os RNF, contudo, não possuem conexão direta com os serviços específicos fornecidos aos usuários pelo sistema (SOMMERVILLE, 2019). E, diferente dos funcionais, onde os usuários conseguem contornar uma funcionalidade que não os satisfaz, descumprir um requisito não funcional pode implicar na inutilização do sistema (SOMMERVILLE, 2019). A Tabela 2 elenca os Requisitos Não Funcionais da aplicação.

### 3.2.2 Casos de Uso

Segundo Pressman (2011), Casos de Uso, do inglês *Use Cases* (UC), ajudam a definir as funcionalidades do sistema e as suas características através do ponto de vista do usuário. Um

Tabela 2 – Requisitos Não Funcionais da Aplicação

| # | Requisito |
|---|---|
| RNF01 | O sistema precisa de espaço para logomarca do aplicativo |
| RNF02 | O sistema deve utilizar uma API externa para apresentar a capa e o número de páginas do livro |
| RNF03 | O sistema deve ter autenticação de usuário |
| RNF04 | A senha deve ser gravada criptografada |
| RNF05 | O sistema não deve permitir cache de senha |
| RNF06 | O sistema deve ter interface responsiva |
| RNF07 | O sistema deve ser compatível com os navegadores modernos e mais usuais (Chrome, Firefox, Opera, Safari) |
| RNF08 | O sistema deve permitir cadastro a partir de redes sociais(Google e Facebook) |

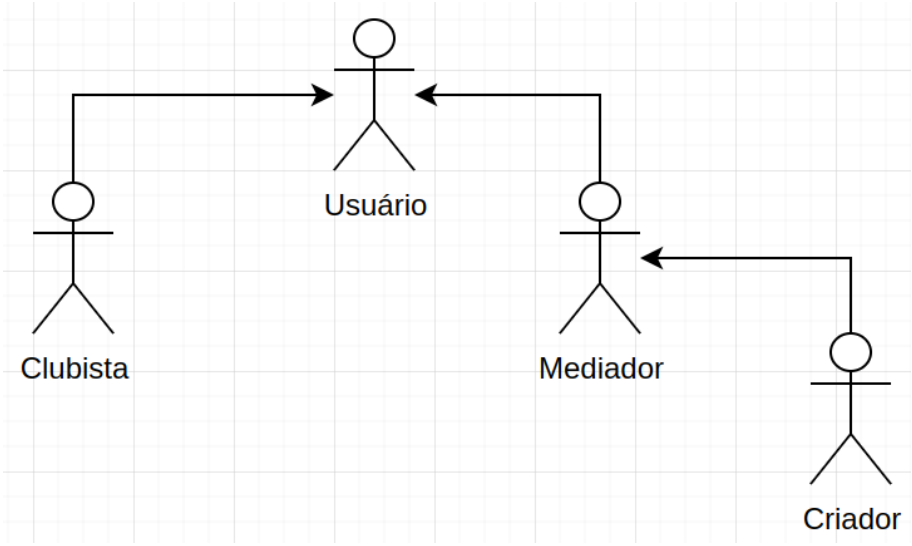
Fonte: (OLIVEIRA, 2023; LEMOS, 2023)

caso de uso descreve como um usuário(ou ator [1]) interage com o sistema seguindo os passos necessários para executar uma ação ou atingir um objetivo específico (PRESSMAN, 2011).

A UML(*Unified Modeling Language* – linguagem de modelagem unificada) é uma linguagem de fácil compreensão usada por arquitetos de *software* para criarem diagramas que ajudam os desenvolvedores a entender e especificar e, principalmente, construir o *software* (PRESSMAN, 2011).

Os casos de uso para o Clubes de Leitura contêm quatro atores, ilustrados na Figura 9 abaixo:

Figura 9 – Atores dos Diagramas de Casos de Uso




Fonte: (LEMOS, 2023)

[1] Ator não é uma pessoa específica, mas sim um papel que uma pessoa (ou dispositivo) desempenha em um contexto específico (PRESSMAN, 2011, p. 155).

Segundo Sommerville (2019), os casos de uso são utilizados basicamente para modelar interações entre o sistema e os agentes externos(outrora chamados de atores), que podem ser pessoas ou até mesmo outros sistemas.

Neste trabalho, os casos de uso da aplicação estão listados em tabelas, onde a Tabela 3 lista os casos de uso para os usuários da aplicação, a Tabela 4 lista os casos de uso para um usuário administrador do clube, a Tabela 5 contém os casos de uso para usuários clubistas e, por fim, os casos de uso para um usuário mediador estão na Tabela 6.

Tabela 3 – Casos de Uso para usuários em geral

| ID | Caso de Uso |
|---|---|
| UC01 | O Registro no sistema |
| UC02 | Visualização de Clubes de Leitura |
| UC03 | Realização de Login |
| UC04 | Visualização de Comentários |
| UC05 | Cadastro de Comentários nos Livros |
| UC06 | Visualização da Estante do Clube |
| UC07 | Visualização dos Livros |
| UC08 | Cadastro dos Livros na Estante de Usuário |
| UC09 | Busca por Livros |

Fonte: (OLIVEIRA, 2023; LEMOS, 2023)

Tabela 4 – Casos de Uso para usuário administrador

| ID | Caso de Uso |
|---|---|
| UC01 | Criação de Clube |
| UC02 | Exclusão de Clube |
| UC03 | Gerenciamento de Mediadores |
| UC04 | Cadastro de Política de Ingresso do Clube |

Fonte: (OLIVEIRA, 2023; LEMOS, 2023)

Tabela 5 – Casos de Uso para usuário clubista

| ID | Caso de Uso |
|---|---|
| UC01 | Cadastro de Avaliação em um Livro |
| UC02 | Cadastro de Voto |
| UC03 | Cadastro em um Clube |
| UC04 | Visualização de Estante Pessoal |
| UC05 | Cadastro de *like* nos Comentários |

Fonte: (OLIVEIRA, 2023; LEMOS, 2023)

Tabela 6 – Casos de Uso para usuário mediador

| ID | Caso de Uso |
|---|---|
| UC01 | Identificação de Comentário com *Spoiler* |
| UC02 | Cadastro de Votação |
| UC03 | Cadastro de Datas Votáveis |
| UC04 | Cadastro de Datas de Reunião |
| UC05 | Cadastro de Livros Votáveis |
| UC06 | Cadastro de Livro da Vez |
| UC07 | Arquivamento de Comentário |
| UC08 | Aceitação de Clubista |
| UC09 | Expulsão de Clubista |

Fonte: (OLIVEIRA, 2023; LEMOS, 2023)

## 3.3 Arquitetura da aplicação

A arquitetura da aplicação foi definida por (OLIVEIRA, 2023), onde explica que o aplicativo não contempla uma arquitetura exclusiva e sim que um conjunto de arquiteturas e padrões foram utilizados para a construção do aplicativo, tais como uma arquitetura baseada em Service-Oriented Architecture (SOA) o padrão arquitetural *Domain Drive Desing* (DDD), o estilo arquitetural *Cliente-Servidor* e o estilo(ou *design*) arquitetural *Clean Arquicture*. Além disso, o sistema é baseado no protocolo HTTP, portanto implementou-se uma *REST API*.

### 3.3.1 *REST API*

*Representational State Transfer* (REST), em português "Transferência de Estado Representacional" é um padrão de design ou estilo de arquitetura de *software* apresentado por Roy Fielding em 2000 em sua tese de doutorado em Ciência da Computação pela Universidade da Califórnia.

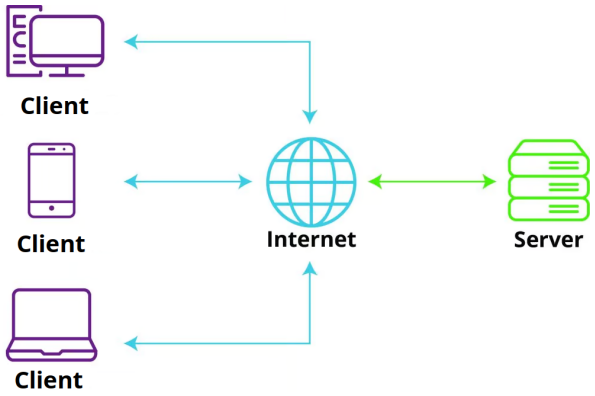
*REST* é definida por (FIELDING, 2000) como um estilo híbrido derivado de vários estilos arquitetônicos baseados em redes — para uma leitura e entendimento profundo a respeito dos estilos arquitetônicos, leia o Capítulo 3 da tese de doutorado de Roy Fielding: Architectural Styles and the Design of Network-based Software Architectures. REST combina tais estilos a um conjunto de restrições que podem ser utilizadas para a construção de serviços *web* como uma *API* ou outras aplicações. É baseado em alguns critérios, tais como *Stateless* e *Client-Server*.

#### 3.3.1.1 *Client-Server*

Segundo Fielding (2000), Client-Server(ou Cliente-Servidor) é o modelo encontrado com maior frequência em aplicativos baseados em rede. Um servidor disponibiliza um conjunto de recursos e processa solicitações relacionadas a eles. O cliente é responsável por executar as

solicitações a fim de encontrar o recurso desejado. O servidor responde ao cliente, podendo rejeitar ou executar a solicitação (FIELDING, 2000).

Figura 10 – Modelo Cliente-Servidor

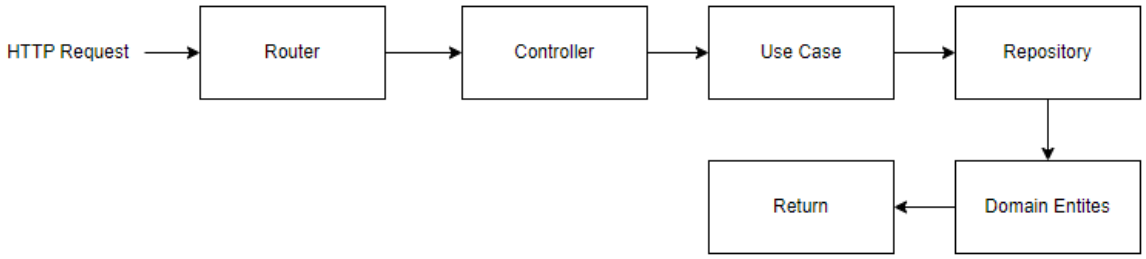



Fonte: Adaptado de Molloy (2023)

A Figura 10 ilustra as solicitações realizadas pelos clientes ao servidor através da *internet*. Observa-se que, ao separar o cliente do servidor, ganha-se algumas vantagens no que diz respeito à escalabilidade do servidor, à portabilidade da interface do usuário para várias plataformas e à evolução independente dos componentes (FIELDING, 2000).

O servidor do Clubes de Leitura foi desenvolvido seguindo o design de arquitetura em camadas, como explica Oliveira (2023, p.13), e seguindo as instruções de Fielding (2000), onde deve-se criar camadas que sejam dependentes apenas da camada mais próxima, ou das camadas imediatamente adjacentes. Além de evitar o uso de bibliotecas nas camadas mais internas, pois o *software* pode criar dependências indesejáveis (OLIVEIRA, 2023).

Figura 11 – Arquitetura do Servidor da aplicação




Fonte: (OLIVEIRA, 2023)

Oliveira (2023, p.29) define a arquitetura do aplicativo Clubes de Leitura como demonstrada na Figura 11.

O cliente, também conhecido como interface do usuário ou *frontend*, foi desenvolvido como monolítico, multiplataforma e usando o processo de renderização no lado do cliente –

Client-Side Rendering (CSR), onde as páginas da aplicação são montadas no *browser*(navegador de internet) do cliente. Além disso, o modelo de desenvolvimento Single-Page Application (SPA) está presente na aplicação, pois ela permite que seja desenvolvido apenas uma única página *web* e atualize pequenos trechos dentro dessa página (OLIVEIRA, 2023).

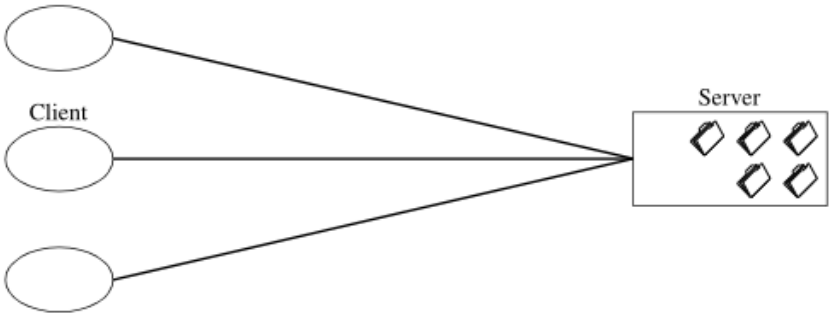
No modelo de arquitetura cliente-servidor, o *frontend* é o responsável por realizar requisições ao servidor para a obtenção de dados, além de apresentar esses dados visualmente ao usuário.

#### 3.3.1.2 *Stateless*

É uma restrição derivada do *client-server* e determina que o servidor não deve armazenar nenhum estado de sessão e que cada solicitação do cliente deve conter todas as informações necessárias para que o servidor entenda a solicitação (FIELDING, 2000). Uma vez que o cliente armazena todo o estado da sessão, nenhuma informação sobre requisições anteriores é mantida no servidor.

Essa restrição assegura que o servidor não tenha que olhar para outros dados para garantir a resposta, além dos enviados na requisição. Elimina possíveis falhas, pois o servidor vai entregar exatamente o que foi solicitado. E uma melhora na escalabilidade é observada, pois o servidor libera recursos mais rapidamente, porque não armazena os dados das solicitações.

Figura 12 – Comunicação sem estado



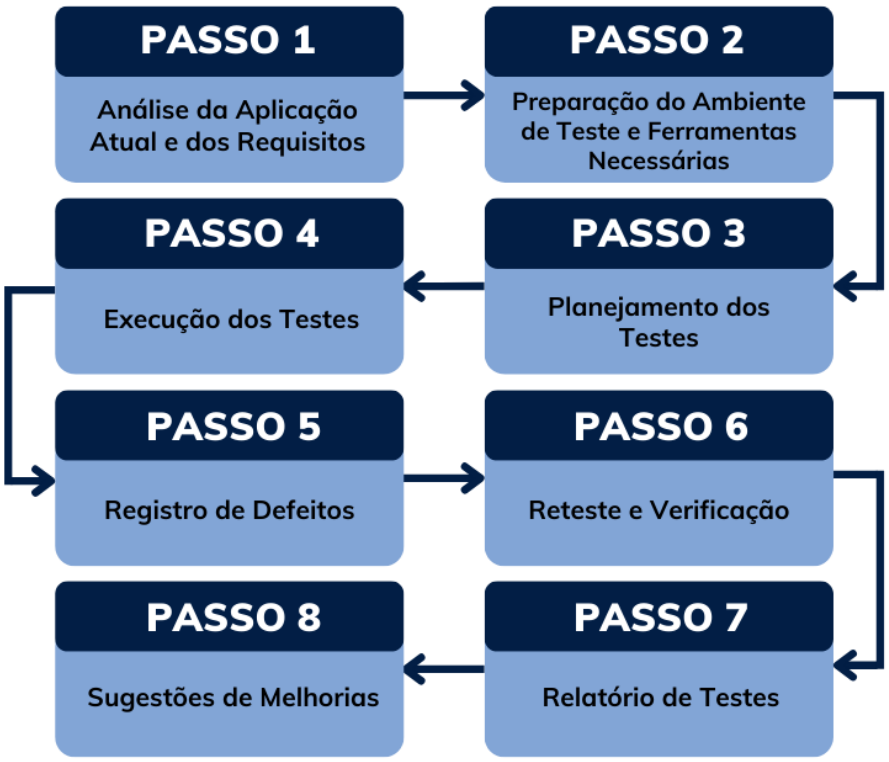

Fonte: (FIELDING, 2000)

Em contrapartida, a comunicação *stateless* apresenta uma desvantagem quanto à diminuição do desempenho da rede, pois o servidor pode ser sobrecarregado por uma série de requisições repetitivas, já que os dados das requisições não são armazenados no servidor em um contexto compartilhado (FIELDING, 2000).

# 4 Metodologia do Trabalho

Este capítulo tem como objetivo fornecer um guia para a execução dos testes, desde a análise inicial da aplicação e dos requisitos até a elaboração do relatório de testes e identificação de melhorias. Ao longo deste capítulo, serão apresentados os passos que descrevem o processo de teste, visando garantir a qualidade e a eficácia da aplicação. A Figura 13 mostra como a metodologia deste trabalho foi dividida:

Figura 13 – Metodologia do Trabalho.




Fonte: Elaborado pelo autor

## 4.1 Análise da Aplicação Atual e dos Requisitos

Antes de iniciar os testes do aplicativo *Clubes de leitura*, é essencial realizar uma análise detalhada da aplicação, conforme descrito no Capítulo 3 deste trabalho. Isso envolve revisar os requisitos do aplicativo, o código-fonte e as funcionalidades existentes para compreender a sua estrutura e seu funcionamento. A análise dos requisitos ajuda a identificar as áreas críticas da aplicação que requerem testes mais detalhados e a estabelecer uma base sólida para o planejamento dos testes. Essa etapa é fundamental para garantir uma cobertura completa durante

os testes e para identificar quaisquer falhas nos requisitos ou no funcionamento da aplicação que possam afetar sua qualidade.

## 4.2 Preparação do Ambiente de Teste

Estabelecer um ambiente de teste isolado é fundamental. Isso envolve configurar um ambiente que não interfira no ambiente de produção do aplicativo. Todas as dependências necessárias devem ser instaladas e configuradas corretamente para garantir a eficácia dos testes e a obtenção de resultados precisos. Para esse passo, tivemos o apoio do Núcleo de Tecnologia da Informação (NTI) da UFMA.

### 4.2.1 Tecnologias e Ferramentas Necessárias

- Rest Assured [1]
- Cypress [2]
- Java [3]
- JavaScript [4]
- Jenkins [5]
- Postman [6]
- Allure Report [7]
- Docker [8]
- Docker Compose [9]
- Harbor [10]
- Nginx [11]
- SonarQube Community [12]

Para a execução dos testes automatizados, foram escolhidas as ferramentas *RestAssured* e *Cypress*, que serão responsáveis pelos testes da *API* e da interface gráfica da aplicação, respectivamente. *Java* e *JavaScript* são as linguagens de programação utilizadas para escrever os *scripts* de testes. Ambas são linguagens que utilizam o paradigma de programação orientada a objetos (POO), o *JavaScript* suporta ainda a programação funcional.

O *Postman* é uma ferramenta que possibilita o envio de requisições HTTP para um servidor sem o uso de um navegador de internet. Com ele, é possível também realizar testes manuais e exploratórios na *API*, citados na *Seção 2.3* deste trabalho.

[1] Disponível em: *https://rest-assured.io/*
[2] Disponível em: *https://www.cypress.io/*
[3] Disponível em: *https://www.java.com/en/*
[4] Disponível em: *https://developer.mozilla.org/pt-BR/docs/Web/JavaScript*
[5] Disponível em: *https://www.jenkins.io/doc/*
[6] Disponível em: *https://www.postman.com/*
[7] Disponível em: *https://allurereport.org/*
[8] Disponível em: *https://www.docker.com/*
[9] Disponível em: *https://docs.docker.com/compose/install/*
[10] Disponível em: *https://goharbor.io/*
[11] Disponível em: *https://nginx.org/*
[12] Disponível em: *https://www.sonarsource.com/open-source-editions/sonarqube-community-edition/*

O *Jenkins* é um servidor de automação de código aberto que pode ser utilizado para automatizar todos os tipos de processos relacionados a *build*, testes e entrega ou implantação de *softwares*.

Para gerar as evidências de execução dos testes automatizados, escolheu-se o *Allure Report*, uma ferramenta de código aberto para geração e visualização dos relatórios de uma execução de teste. Ela captura os resultados dos testes automatizados e, com alguns comandos, gera relatórios completos que auxiliam durante a análise dos testes e também na identificação de falhas e melhorias.

Com *Docker* e *Docker Compose* é possível criar imagens e contêineres para a execução e gerenciamento dos ambientes da aplicação. E o *Harbor* combinado a essas duas ferramentas auxilia organizando as imagens em um repositório privado. O *Nginx* foi escolhido como servidor *web* e *proxy* reverso para disponibilizar a aplicação na internet.

Para análise estática e cobertura de código, optou-se por utilizar o SonarQube na versão *Community Edition*, por tratar-se de uma versão gratuita, autohospedada e que permite a configuração e gerenciamento em um servidor próprio.

### 4.2.2 Implementação do Pipeline de CI/CD

O fluxo de desenvolvimento deve ser eficiente e seguro, portanto, um pipeline de CI/CD será implementada. O pipeline será responsável pela realização de *deploys* automatizados nos ambientes de teste e produção, além da execução dos testes automatizados planejados neste trabalho. O pipeline acelera o ciclo de execução de testes, facilita a integração das alterações realizadas no código-fonte e ajuda a reduzir riscos de erros manuais.

A Figura 14 mostra a sugestão do pipeline com base nos conceitos discutidos ao longo deste trabalho.

Figura 14 – Pipeline para CI/CD


Fonte: Elaborado pelo autor

## 4.3 Planejamento dos Testes

O próximo passo é o planejamento dos testes. Isso inclui uma revisão detalhada dos requisitos do aplicativo e a identificação dos casos de teste relevantes para cada funcionalidade disponível. Os casos de teste devem abranger uma variedade de cenários, incluindo casos típicos

de uso e casos excepcionais. Esses casos de teste devem ser priorizados com base em sua importância e na frequência com que são utilizados pelos usuários.

### 4.3.1 Casos de teste

Delamaro et al. (2007) definem um caso de teste como a combinação de dados de entrada com os resultados esperados para a execução do sistema sob essas condições específicas. Adicionalmente, descreve um *conjunto de teste* como a coleção de todos os casos de teste executados durante uma atividade de teste.

Já para Ammann e Offutt (2008), um caso de teste é composto pelos dados de entrada, resultados esperados e pelos valores prefixos e pós-fixos necessários para uma completa execução e validação do sistema. Os valores prefixos são definidos como os dados necessários para que o software esteja pronto para executar o teste. Já os valores pós-fixos referem-se aos dados que precisam ser enviados para o sistema após o envio dos valores de entrada.

O ISTQB (2022) afirma que todos os casos de teste devem ser repetíveis, verificáveis e diretamente vinculados aos requisitos ou *design*(como histórias de usuário, especificações ou diagramas). Ainda segundo o ISTQB (2022) os casos de teste devem definir:

- O objetivo do teste.
- Condições prévias que incluem requisitos do projeto, ambiente de teste, planos de entrega e estado do sistema antes da execução.
- Pré-requisitos para o caso de testes e dados que devem existir para que o caso de teste seja executado no sistema.
- Resultados esperados com critérios de aprovação ou reprovação.
- Pós-condições, por exemplo, dados alterados, o estado do *software* após a execução do teste, gatilhos para pós-processamento, entre outros.

Para este trabalho, os casos de teste foram organizados com a seguinte estrutura: identificador, título, tipo de teste (end-to-end ou funcional), descrição, pré-requisitos, passos para execução, resultado esperado e status do teste (a executar, sucesso ou falha). Essa estrutura pode ser observada na Tabela 7. Os casos de teste foram criados com base nos conceitos discutidos anteriormente, além dos requisitos e casos de uso analisados no Capítulo 3.

## 4.4 Execução dos Testes

Com o planejamento em execução, é hora de executar os testes. Isso inclui uma variedade de abordagens, como testes funcionais para verificar se as funcionalidades operam conforme

Tabela 7 – Modelo da estrutura do caso de teste

| Elemento | Descrição |
|---|---|
| **ID** | Identificação única do caso de teste |
| **Título** | Nome do caso de teste |
| **Tipo** | End-to-end (UI) ou Funcional (API) |
| **Descrição** | O que o teste valida |
| **Endpoint** | Rota utilizada no teste |
| **Pré-requisitos** | Condições que devem ser atendidas antes do teste |
| **Passos para execução** | Etapas para executar o teste escrito em *Gherkin* |
| **Resultado esperado** | Resultado esperado para cada etapa |
| **Status** | Resultado do teste |

Fonte: Elaborado pelo autor

esperado, testes de interface do usuário para garantir uma experiência consistente e intuitiva, testes exploratórios para identificar possíveis falhas não previstas e testes unitários para garantir a integridade do código.

## 4.5 Registro de Defeitos

Durante a execução dos testes, é inevitável que defeitos sejam encontrados. É importante documentar cuidadosamente cada defeito encontrado, incluindo uma descrição detalhada do problema e os passos necessários para reproduzi-lo. Cada defeito deve ser atribuído a uma prioridade e gravidade específicas para auxiliar na triagem e correção.

## 4.6 Reteste e Verificação

Após a correção dos defeitos identificados, é essencial realizar o reteste para verificar se as correções foram aplicadas com sucesso e se novos problemas não foram introduzidos. Durante esse estágio, todas as funcionalidades do aplicativo devem ser verificadas novamente para garantir que estejam operando conforme o esperado.

## 4.7 Relatório de Testes

Ao concluir os testes, é necessário gerar um relatório abrangente para detalhar os resultados dos testes realizados. Isso inclui uma análise dos resultados, estatísticas de defeitos encontrados e recomendações para possíveis melhorias. Esse relatório deve ser compartilhado com a equipe de desenvolvimento e outras partes interessadas para revisão e *feedback*.

## 4.8 Sugestões de Melhorias

Por fim, é importante identificar oportunidades de melhoria para o processo de desenvolvimento e teste do aplicativo. Isso pode envolver a revisão dos processos de teste, a implementação de ferramentas de automação adicionais ou o desenvolvimento de novos casos de teste com base no *feedback* dos usuários e nos resultados dos testes realizados. O objetivo final é garantir a qualidade e a confiabilidade contínuas do aplicativo Clubes de Leitura.

# 5 Desenvolvimento

No presente capítulo, serão detalhadas as execuções dos passos da metodologia descritos no Capítulo 4. Primeiramente, apresenta-se a preparação do ambiente, que se iniciou com a configuração e implementação do servidor responsável por hospedar a aplicação. Além disso, serão descritas as ferramentas utilizadas e como elas favorecem a execução eficiente do servidor. Também serão discutidas as configurações e a implementação do *Jenkins* e do pipeline sugerida na Seção 4.2.2, bem como a configuração do *Harbor*, o repositório de imagens *Docker* da aplicação.

A preparação do ambiente iniciou-se com a disponibilização de alguns DNS (*Domain Name System*) para a aplicação por parte do NTI. A Tabela 8 lista cada DNS e seus respectivos *hostnames*, endereço de IP *(Internet Protocol Address)*, a responsabilidade de cada um e se o acesso é pela VPN:

Tabela 8 – DNS do servidor

| DNS | Responsabilidade | VPN |
|---|---|---|
| clubedeleitura.ufma.br | Ambiente de produção | Não |
| clubedeleitura-hml.ufma.br | Ambiente de testes/homologação | Sim |
| jenkins.clubedeleitura.ufma.br | Acesso aos *jobs* de pipelines | Sim |
| harbor.clubedeleitura.ufma.br | Acesso ao repositório de imagens docker | Sim |
| sonar.clubedeleitura.ufma.br | Analisador estático de código | Sim |

Fonte: Elaborado pelo autor.

## 5.1 Infraestrutura e Configuração do Servidor

A infraestrutura utilizada para hospedar a aplicação baseia-se em uma máquina virtual hospedada no *data center* da UFMA, rodando o sistema operacional Ubuntu. A máquina foi configurada por profissionais do NTI e disponibilizada para hospedar a aplicação. É possível acessar o servidor utilizando o protocolo SSH, através da VPN da universidade. As credenciais do servidor e da VPN foram fornecidas pelo NTI, possibilitando o acesso ao servidor de forma segura. Algumas informações sobre o servidor podem ser vistas na Tabela 9.

Uma prática essencial para a organização de um *software*, é a separação de ambientes de execução para cada parte do ciclo de desenvolvimento. Humble e Farley (2010) definem um *ambiente* como o conjunto dos recursos e configurações que uma aplicação precisa para funcionar, e que os ambientes são caracterizados por algumas propriedades, como os requisitos de *hardware* dos servidores que formam o ambiente e a infraestrutura de rede e também a

Tabela 9 – Requisitos de hardware da máquina virtual(Servidor)

| Componente | Informação |
|---|---|
| **SO** | Ubuntu 22.04.5 LTS |
| **Kernel** | Linux |
| **Versão do kernel** | 5.15.0-131-generic |
| **Arquitetura** | x86_64 |
| **Processador** | Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz |
| **Memória RAM** | 8GB |
| **Armazenamento** | 80GB |

Fonte: Elaborado pelo autor com base nos dados do servidor.

configuração do sistema operacional (como aplicativos e servidores *web*, servidores de banco de dados, etc.) que dão suporte para que as aplicações sejam executadas.

A ISO/IEC 27001:2022 recomenda que os ambientes devem ser separados e seguros, com o propósito de manter o controle e o isolamento do ambiente de produção para não ser afetado por dados gerados pelas atividades de desenvolvimento e testes (ISO/IEC, 2022). Com isso, para iniciar a separação e implementação dos ambientes, as ferramentas necessárias citadas na Seção 4.2.1 foram instaladas. A Tabela 10 lista essas ferramentas e suas respectivas versões.

Tabela 10 – Ferramentas de implementação do app

| Ferramenta | Versão |
|---|---|
| Docker | 28.0.1 |
| Docker Compose | 2.33.1 |
| Jenkins | 2.492.2 |
| Harbor | 2.12.2 |
| Nginx | 1.18.0 |

Fonte: Elaborado pelo autor com base nos dados do servidor.

Então, foram definidos 3 ambientes: produção, testes/homologação e desenvolvimento. A separação dos ambientes foi realizada utilizando o *Docker* onde cada um possui seus próprios contêineres e imagens isolados. Além disso, os *volumes* foram configurados para garantir que os dados do banco de dados não sejam perdidos em caso de reinicialização dos contêineres. A configuração dos volumes foi realizada conforme a Figura 15.

## 5.2 Gerenciamento de Imagens Docker com Harbor

Para o armazenamento e gerenciamento das imagens *docker* da aplicação, foi escolhido o *Harbor*, um registro privado de código aberto que protege artefatos com políticas de controle de acesso baseadas em funções, realiza a varredura das imagens para detectar vulnerabilidades e também permite a assinatura das imagens como confiáveis. Além disso, o Harbor oferece conformidade, desempenho e interoperabilidade para auxiliar no gerenciamento seguro e

Figura 15 – Configuração dos volumes

```yaml
name: cl_api
services:
  database:
    volumes:
      - nome_do_volume:/caminho/do/volume/no/container

volumes:
  nome_do_volume:
```

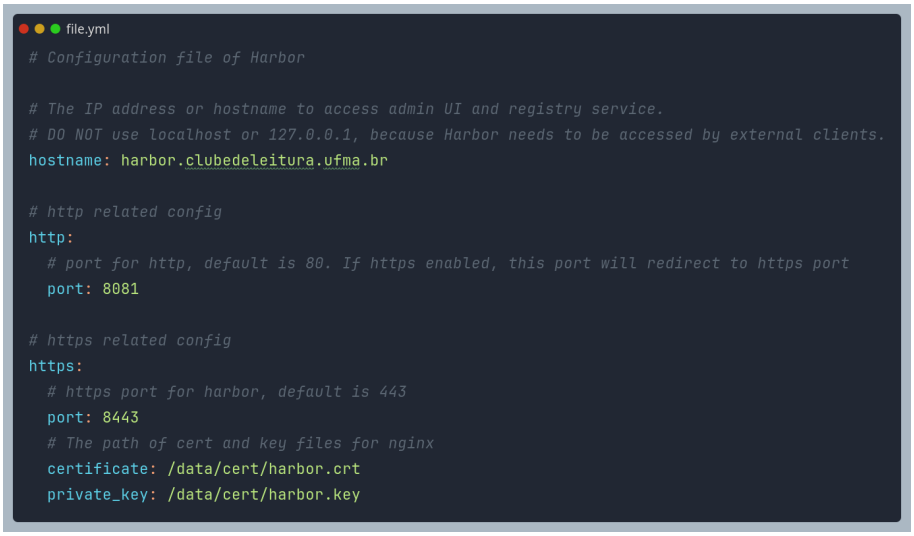
Fonte: Elaborado pelo autor.

consistente de artefatos nas plataformas de computação nativa em nuvem, como o Docker (HARBOR, 2025).

### 5.2.1 Configuração e Gerenciamento de Projetos

O Harbor foi instalado e configurado seguindo os passos recomendados na sua documentação[1], onde destacam-se o uso do DNS disponibilizado pelo NTI da UFMA, a porta em que o Harbor roda e as configurações do certificado para habilitar o protocolo HTTPS (*Hypertext Transfer Protocol Secure*). Os pontos citados podem ser verificados na Figura 16.

Figura 16 – Configuração do Harbor



file.yml

```yaml
# Configuration file of Harbor

# The IP address or hostname to access admin UI and registry service.
# DO NOT use localhost or 127.0.0.1, because Harbor needs to be accessed by external clients.
hostname: harbor.clubedeleitura.ufma.br

# http related config
http:
  # port for http, default is 80. If https enabled, this port will redirect to https port
  port: 8081

# https related config
https:
  # https port for harbor, default is 443
  port: 8443
  # The path of cert and key files for nginx
  certificate: /data/cert/harbor.crt
  private_key: /data/cert/harbor.key
```

Fonte: Elaborado pelo autor

Após a configuração, foram criados dois projetos principais para armazenar as imagens: *clube-leitura-prod* e *clube-leitura-hml*, para os ambientes de produção e de testes/homologação, respectivamente. Além disso, cada projeto possui dois repositórios para separar as imagens da interface do usuário e da API, chamados de *UI* e *API*.

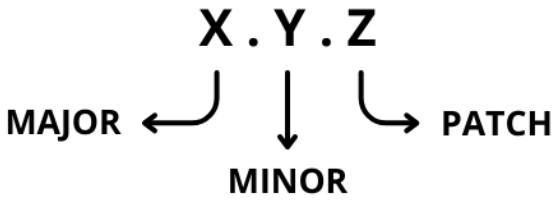
[1] Disponível em: *https://goharbor.io/docs/2.12.0/install-config/#installation-process*

### 5.2.2 Versionamento das Imagens

As imagens são versionadas usando *tags*[2], que seguem o formato de números separados por pontos e que são incrementados de acordo com algumas regras do versionamento semântico, ou *Semantic Versioning*[3]. Além disso, há um identificador chamado *latest*, usado para indicar que é a versão mais atual da imagem.

O formato da versão das imagens de produção é mostrado na Figura 17, onde cada número da versão deve ser incrementado se:

Figura 17 – Formato de versões das imagens de produção




Fonte: Elaborado pelo autor

- **MAJOR**: houver alterações que tornem a API incompatível. É a versão principal e atua como um controle de compatibilidade.
- **MINOR**: houver a adição de funcionalidades de maneira compatível com as versões passadas. Identifica a versão secúndaria, funciona como controle das funcionalidades.
- **PATCH**: houver correções de *bugs* compatíveis com as versões anteriores. Indica que defeitos foram identificados e corrigidos. É a versão de correção.

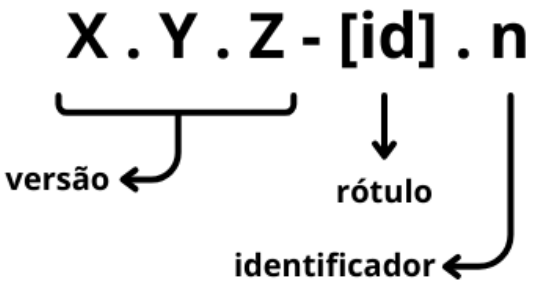
Por exemplo, a versão de uma imagem é *2.5.0* e foi preciso corrigir um defeito em produção, a versão da imagem com a correção será *2.5.1*. Posteriormente, foram implementadas novas funcionalidades, mas mantendo a compatibilidade da aplicação com as versões já lançadas anteriormente, logo a próxima versão da imagem será *2.6.0*. Por fim, a aplicação passou a usar outro servidor de banco de dados, então a versão a ser lançada será *3.0.0*, pois com a mudança haverá a incompatibilidade da aplicação com as versões anteriores.

Além disso, há *tags* que possuem um *rótulo* e um *identificador*. Os rótulos servem para indicar em que fase do desenvolvimento a imagem que recebeu a *tag* está, já o identificador indica o número de *pushes* recebidos até gerar uma imagem estável. A Figura 18 ilustra a estrutura dessas *tags*.

[2] Segundo Docker (2025c) uma *tag* é um identificador opcional usado para fazer referência a uma versão ou uma variação específica da imagem.

[3] Disnponível em: <*https://semver.org/*>. Acesso em: 18 de março de 2025.

Figura 18 – Formato de versões das imagens de teste




Fonte: Elaborado pelo autor

- **X.Y.Z**: é a versão definida na Figura 17.
- **[id]**: é um rótulo que varia entre *alpha*, *beta* ou *rc*, dependendo da fase do desenvolvimento da imagem.
- **n**: é um número inteiro usado para identificar a versão. É incrementado em relação à *tag* anterior.

O formato da Figura 18 é conhecido como pré-lançamento (*pre-release*), são candidatas a serem implantadas em produção, porém podem ser instáveis, pois podem apresentar incompatibilidades nas funcionalidades implementadas. Por isso, o rótulo é importante, porque indica a fase da imagem e consequentemente a sua estabilidade. Os rótulos indicam:

1. *rc*: abreviação de *release cadidate*. É uma imagem que já passou por toda a esteira de testes e está pronta para ser levada para produção.
2. *beta*: Indica que a imagem passou pelos testes funcionais e é menos instável, mas que ainda pode apresentar inconsistências.
3. *alpha*: As imagens alpha são as mais instáveis, pois são imagens que estão em desenvolvimento ou que foram reprovadas nos testes da esteira de testes. Portanto, são imagens que em hipótese alguma devem ser levadas para produção.

## 5.3 Gerenciamento de Contêineres e Ambientes

Os contêineres foram estruturados de acordo com o ambiente em execução, garantindo que os ambientes de produção, testes e desenvolvimento sejam isolados, como recomenda a ISO/IEC 27001:2022. Apesar da separação, todos os ambientes possuem configurações semelhantes.

### 5.3.1 Principais Contêineres

Como explicado na Seção 5.1 deste capítulo, os contêineres foram separados em ambientes diferentes, dentre eles destacam-se os ambientes de produção e testes/homologação. O ambiente de desenvolvimento é configurado de forma local. Os principais contêineres da infraestrutura da aplicação estão listados na Tabela 11.

Tabela 11 – Principais contêineres aplicação

| Contêiner | Descrição |
|---|---|
| harbor | Principal contêiner do repositório de imagens |
| cl-api-hml | API do ambiente de testes/homologação |
| cl-ui-hml | Interface do ambiente de testes/homologação |
| cl-database-hml | Banco de dados do ambiente de testes/homologação |
| cl-api-prod | API do ambiente de produção |
| cl-ui-prod | Interface do ambiente de produção |
| cl-database-prod | Banco de dados do ambiente de produção |

Fonte: Elaborado pelo autor.

### 5.3.2 Uso de Arquivos *Dockerfile*, *docker-compose* e *.env*

Os ambientes são configurados por meio de arquivos que contêm as variáveis de ambiente correspondentes ao ambiente que se deseja executar. Para facilitar a identificação do ambiente durante a execução, cada arquivo leva o nome do seu ambiente. Por exemplo, o ambiente de testes/homologação é configurado a partir do arquivo *.env.hml*, o ambiente de produção usa o arquivo *.env.prod* e o arquivo *.env.dev* é responsável por armazenar a configuração do ambiente de desenvolvimento.

Além disso, é utilizado um arquivo *Dockerfile* para gerar imagens automaticamente por meio das instruções contidas nele. O Docker (2025b) define esse arquivo como sendo *"um documento de texto que armazena todos os comandos que um desenvolvedor pode executar na linha de comando do terminal para montar uma imagem"*.

Para construir as imagens da API e do banco de dados, utiliza-se um arquivo chamado *docker-compose*. Segundo o Docker (2025a), o *compose* é um arquivo usado para configurar os serviços, redes, volumes, etc. de uma aplicação, além de criar e executar todos os serviços através de linhas de comando em um terminal usando tais configurações. Os nomes dos arquivos *compose* seguem a mesma regra de nomenclatura dos arquivos *.env* mencionados anteriormente. Então, o arquivo *docker-compose.prod.yml* é o responsável por construir as imagens e executar os contêineres do ambiente de produção.

### 5.3.3 Configuração de Redes Docker

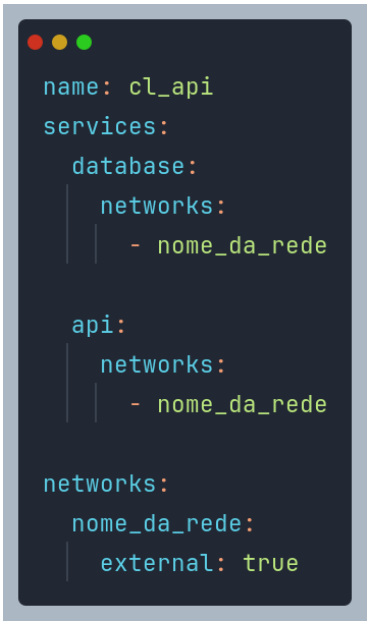
Inicialmente, cada ambiente é composto por 3 contêineres que rodam os serviços de frontend, backend e banco de dados, e, para completar a configuração do ambiente, é necessário que esses contêineres tenham a capacidade de se conectar e se comunicar entre si. Para isso, o Docker dispõe de recursos de gerenciamento de redes, como a criação, conexão, entre outros. Apesar de cada contêiner possuir uma rede habilitada por padrão, é possível configurar para que um determinado contêiner possa se conectar a uma rede específica (DOCKER, 2025a). Tendo isso em vista, podemos criar uma rede personalizada executando o comando de criação de rede do Docker (Figura 19).

Figura 19 – Comando de criação manual de rede do Docker


Fonte: (DOCKER, 2025a)

Após a criação da rede, é preciso informar ao contêiner que ele não deve usar e nem criar uma rede padrão, mas deve usar a rede criada pelo comando da Figura 19. Como queremos manter a conexão entre todos os contêineres, é preciso fazer essa configuração em cada contêiner do ambiente. Então, no arquivo *docker-compose.yml*, podemos configurar a rede conforme a Figura 20 exemplifica.

Figura 20 – Configurando os contêineres para utilizar uma rede externa




Fonte: Elaborado pelo autor.

## 5.4 Configuração do NGINX

O *NGINX* ("*engine* x") é um dos servidores *web* mais utilizados; entre os motivos para tal estão suas capacidades como balanceador de carga (DEJONGHE, 2024), além de ser um servidor web HTTP, proxy reverso, cache de conteúdo e proxy de outros protocolos da internet (NGINX, 2025).

O servidor do Clubes de Leitura utiliza o Nginx principalmente como servidor *web* e também como proxy reverso para gerenciar as requisições da aplicação. As rotas foram divididas conforme os ambientes, garantindo assim a distribuição correta do tráfego.

Conforme Nginx (2025), as configurações são realizadas utilizando um arquivo de texto, onde armazenam as configurações para o servidor. Por padrão, o arquivo é nomeado como *nginx.conf* mas para as configurações do servidor utilizaram-se os nomes dos serviços para nomear os arquivos de configuração. A Tabela 12 contém os nomes de cada arquivo e onde estão armazenados no servidor.

Tabela 12 – Arquivos de configuração do Nginx

| Caminho | Arquivo | Descrição |
|---|---|---|
| /opt/clubes-de-leitura/nginx/ | clubes-de-leitura | Ambiente de produção |
| | clubes-de-leitura.hml | Ambiente de testes/homologação |
| | jenkins | Jenkins |
| | harbor | Harbor |

Fonte: Elaborado pelo autor.

Os arquivos de configurações são formados por diretivas e seus respectivos parâmetros. As diretivas simples, aquelas com uma única linha, são finalizadas com um ponto e vírgula(;). Há diretivas que atuam como blocos que agrupam diretivas simples; os blocos são definidos usando chaves({}) (NGINX, 2025). As Figuras 21a e 21b mostram parte do conteúdo dos arquivos de configuração, onde destacam-se as diretivas *server_name* e *listen* que têm como parâmetros o DNS de cada ambiente e a porta usada para ouvir as requisições.

Figura 21 – Diretivas de configuração do servidor da aplicação

```
server {
    listen 80;
    server_name clubedeleitura.ufma.br;

    return 301 https://$host$request_uri;
}
```

(a) Configuração do servidor do ambiente de produção

```
server {
    listen 80;
    server_name clubedeleitura-hml.ufma.br;

    return 301 https://$host$request_uri;
}
```

(b) Configuração do servidor do ambiente de testes/homologação

Fonte: Elaborado pelo autor.

### 5.4.1 Configuração do HTTPS

O suporte ao protocolo HTTPS foi habilitado a fim de garantir a segurança da comunicação. O uso de certificados SSL (*Secure Sockets Layer*) protege a comunicação entre o cliente e o servidor, criptografando os dados trafegados (DEJONGHE, 2024). Essa configuração pode ser observada em todos os arquivos de configuração do servidor, e geralmente fica logo no início do arquivo. A Figura 22 mostra parte da configuração do servidor para o ambiente de produção.

Figura 22 – Habilitando HTTPS no servidor de produção


Fonte: Elaborado pelo autor

Na Figura 22 a diretiva *listen* recebe a porta encriptada 443, que é a porta frequentemente usada para ouvir as requisições HTTPS, e também recebe o parâmetro *ssl*, que é o protocolo responsável por criptografar os dados. As diretivas *ssl_certificate* e *ssl_certificate_key* definem, respectivamente, o certificado e a chave usada pelo NGINX para descriptografar requisições e criptografar as respostas (DEJONGHE, 2024).

## 5.5 Fluxo de Trabalho do *Git*

Para padronizar a escrita de código, definiu-se um fluxo de trabalho utilizando o *Git* que está representado na Figura 55 do Anexo A. Como explicam Chacon e Straub (2014), *Git*[4] é um sistema de controle de versão de código gratuito e de código aberto capaz de lidar com todos os tipos de projetos, desdes os mais básicos até os mais complexos.

Então, como funciona o fluxo de trabalho para o desenvolvimento de uma nova funcionalidade? Suponha que um desenvolvedor precisa implementar uma nova funcionalidade na aplicação. A primeira coisa a ser feita é garantir o isolamento do desenvolvimento criando uma nova ramificação chamada *feature/nova-funcionalidade* a partir da *branch develop*. Em seguida, o desenvolvedor pode começar a escrever seu código e enviar para o repositório remoto. Ao final do desenvolvimento, ele deve abrir um *Pull Request* para a *develop* para garantir uma mesclagem segura. Com a funcionalidade já desenvolvida e mesclada na ramificação alvo, o desenvolvedor deve abrir outro *Pull Request* mas dessa vez da *branch develop* para a *branch*

[4] Disponível em: <*https://git-scm.com/*>. Acesso em: 27 de maio de 2025

*hml*, que por será mesclada e a estará pronta para ser testada. Explicações mais detalhadas sobre esse fluxo serão vistas na Seção 5.6.

### 5.5.1 Ramificações e suas Responsabilidades

O *Git* trabalha com ramificações (*branches*), que são definidas como ponteiros simples e móveis que apontam para *commits* do repositório (CHACON; STRAUB, 2014). Utilizar *branches* no desenvolvimento é importante pois, com elas, pode-se isolar as implementações de novas funcionalidades ou correções de defeitos sem que uma implementação interfira na outra ou até mesmo no código original. Para este trabalho criou-se algumas *branches* que podem ser vistas no fluxo de trabalho apresentado na Figura 55. Cada ramificação é responsável por conter um contexto dentro do ciclo de desenvolvimento da aplicação e também ajuda na geração das *tags* descritas na Seção 5.2.2. Há 4 ramificações principais e 5 para apoio e desenvolvimento:

- Principais ramificações:
  1. ***main***: Responsável por salvar o histórico de *commits* do projeto
  2. ***prod***: Responsável pela versão em produção da aplicação. É uma abreviação para *production*.
  3. ***hml***: Responsável pela versão em testes/homologação. É uma abreviação para *homologation*.
  4. ***develop***: Responsável por armazenar as funcionalidades em desenvolvimento. É uma abreviação para *development*.
- Ramificações de apoio e desenvolvimento
  1. ***release/x.y.z***[5]: Responsável por preparar o lançamento da versão para produção. Deve conter a versão a ser lançada no nome para fácil identificação.
  2. ***breaking-change/*****[6]: Para desenvolvimento que torna a versão da aplicação incompatível com as versões anteriores.
  3. ***feature/*****[6]: para desenvolvimento de uma funcionalidade.
  4. ***bugfix/*****[6]: para correções menores de defeitos/bugs que ocorrem em produção ou homologação.
  5. ***hotfix/*****[6]: para correções rápidas de defeitos/bugs críticos em produção.

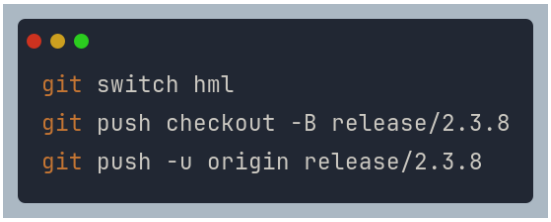
[5] x.y.z *release* deve ser substituído pela versão a ser lançada.

[6] Os (**) significam que pode conter qualquer palavra após a barra (/). Os espaços devem ser separados por (_) ou (-). Exemplo: *bugfix/correcao-hml* ou *feature/nova_funcionalidade*

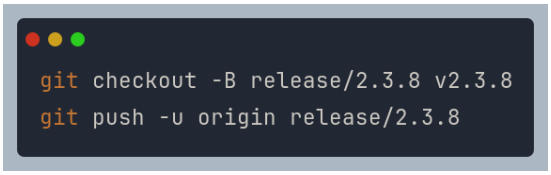
### 5.5.2 Criação do Pacote de Lançamento de Versão

O pacote de lançamento de versão nada mais é do que a organização do que será levado para produção. No fluxo de trabalho apresentado na Figura 55 são as ramificações com prefixo *release* as responsáveis por preparar os pacotes de lançamento. Há duas formas de criar um pacote de lançamento: através de linhas de comandos no terminal ou criar a ramificação diretamente no *GitHub*[7].

Figura 23 – Comandos de criação da ramificação de lançamento




(a) Criação de ramificação a partir da ramificação *hml*

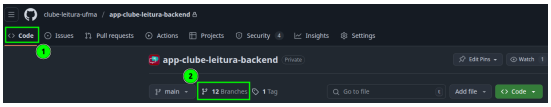



(b) Criação de ramificação a partir da última *tag* da ramificação *hml*

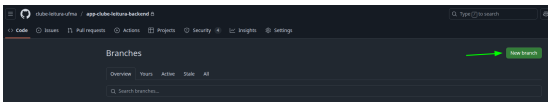
Fonte: Elaborado pelo autor.

Para criar um pacote via comandos de terminal, basta usar a linha de comando do *Git*, apontando para a ramificação *hml* (Figura 23a) ou para a *tag* que aponta para o último *commit* da ramificação *hml* (Figura 23b), ambas as opções tem o mesmo efeito.

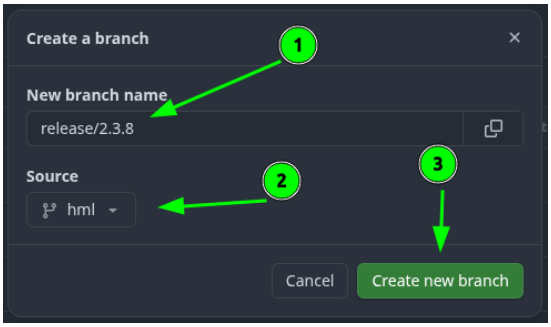
Figura 24 – Comandos de criação da ramificação de lançamento



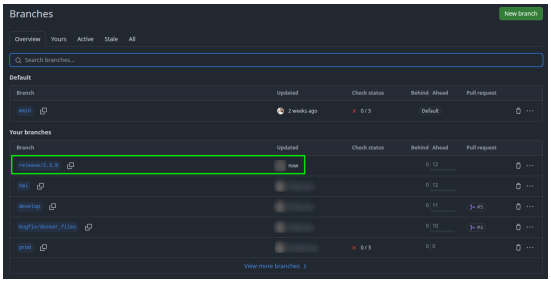
(a) Criação de ramificação a partir da ramificação *hml*



(b) Criação de ramificação a partir da última *tag* da ramificação *hml*




(c) Criação de ramificação a partir da última *tag* da ramificação *hml*




(d) Criação de ramificação a partir da última *tag* da ramificação *hml*

Fonte: Elaborado pelo autor.

É possível criar uma ramificação de *release* via interface gráfica do *GitHub*. Para isto, basta acessar o link do repositório remoto, navegar até a seção *Branches* (Figura 24a), clicar

[7] Disponível em: <*https://github.com/*>.

no botão "*Nova Branch*" (Figura 24b), preencher o nome da ramificação fazendo referência à *tag* que será implantada em produção (Figura 24c). A ramificação criada aparecerá na lista de *branches* do repositório (Figura 24d).

Após realizados os procedimentos, o pacote está pronto para ser lançado. Uma vez determinada e disponibilizada, a versão do *software* em produção não poderá ser modificada em hipótese alguma. Portanto, caso ocorra qualquer necessidade de correção de *bug*, modificações, atualizações ou implementação de novas funcionalidades, deverá ser lançada uma nova versão.

## 5.6 Implementação do Pipeline CI/CD com Jenkins

Esta seção descreverá os passos seguidos durante a configuração e implementação do pipeline, que faz parte do passo 3 da metodologia deste trabalho, descrito na Seção 4.2.2. Inicialmente, foram identificados os softwares necessários para desenvolver esta etapa, onde foi instalado e configurado o *Jenkins* (neste ponto o *Nginx* já estava instalado e configurado como descrito na Seção 5.4).

O site oficial do *Jenkins* o descreve como um software de código aberto, sendo o principal servidor de automação auto-hospedado, e que fornece uma gama de *plugins*[8] os quais auxiliam na criação, testes, implantação e automação de qualquer projeto em qualquer escala (JENKINS, 2025a). O Jenkins pode ser instalado em qualquer computador que tenha o *Java Runtime Environment* (JRE), ou ainda utilizando imagens oficiais do *Docker*. Para este trabalho, utilizou-se a instalação na máquina disponibilizada pelo NTI.

O *Jenkins* dispõe de várias formas para configurar um *pipeline* e para o projeto do Clubes de Leitura utilizou-se o arquivo de configuração *Jenksinfile* combinado com algumas configurações realizadas na interface gráfica do *Jenkins*. O *Jenkinsfile* nada mais é que um arquivo de texto que contém a definição de *scripts* onde descreve as etapas e configurações do *pipeline* e deve estar no repositório de controle de versão da aplicação (JENKINS, 2025d).

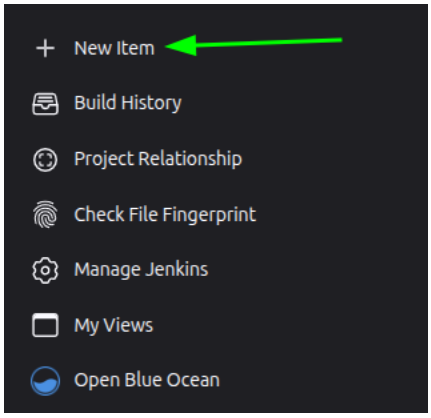
### 5.6.1 Configuração do Pipeline

Para configurar os pipelines, usou-se a opção *Organization Folders* (ou em português, Pastas de Organização), que é uma forma de monitorar uma organização inteira do *GitHub*, visto que a organização do Clubes de Leitura possui dois repositórios, um para a API e um para a interface gráfica. Além disso, esse tipo de projeto permite a criação automática de novos pipelines multi-ramificações [9] para os repositórios que têm ramificações e que contenham um *Jenkinsfile*.

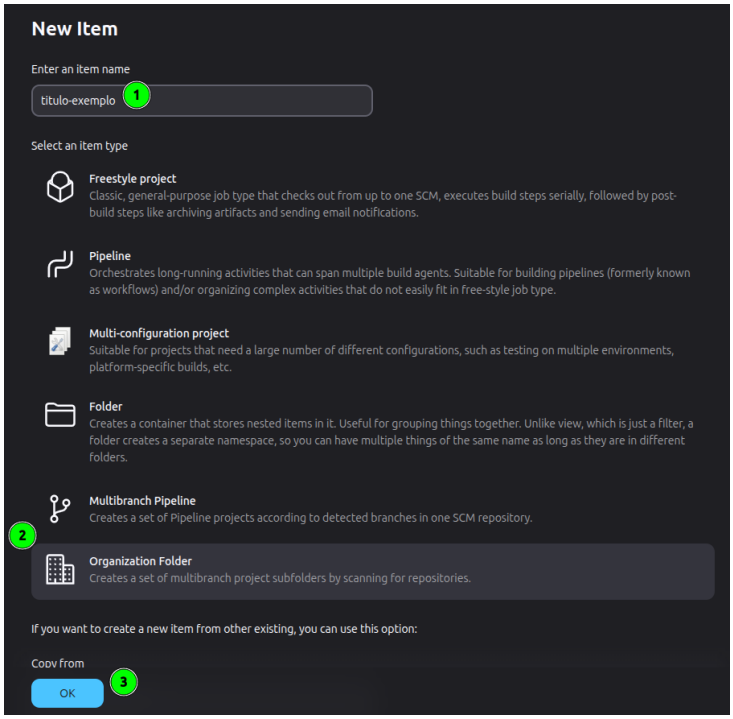
[8] Extensões que são acopladas dando novas funcionalidades ao software principal.

[9] *Multibranch Pipeline* é um tipo de projeto que é capaz de implementar vários *Jenkinsfiles* para várias ramificações no mesmo projeto. Nesse tipo de projeto, o *Jenkins* procura, gerencia e executa automaticamente os *pipelines* para cada ramificação que tenha um *Jenkinsfile* (JENKINS, 2025b).

Figura 25 – Criação de uma *Organization Folders* no *Jenkins*




(a) Menu da página inicial do *Jenkins*




(b) Escolhendo o tipo do projeto

Fonte: Elaborado pelo autor.

A Figura 25 mostra como criar um projeto do tipo *Organization Folders*. Após clicar na opção ***New Item*** na página inicial (Figura 25a), é preciso seguir os passos básicos (Figura 25b) para a criação do projeto. Com isso, nosso projeto estará pronto para ser configurado [10] e executado.

### 5.6.2 Configuração de credenciais e dados sensíveis

Algumas funcionalidades do *Jenkins* precisam de integrações com ferramentas e sistemas externos, e para o correto funcionamento é necessário informar credenciais como *chaves de API*, *tokens de acesso*, *senhas*, *caminhos de arquivos no servidor*, etc. E, por questões de segurança, o *Jenkins* disponibiliza um menu de configurações para que esses dados sensíveis sejam armazenados e não sejam vazados por descuido.

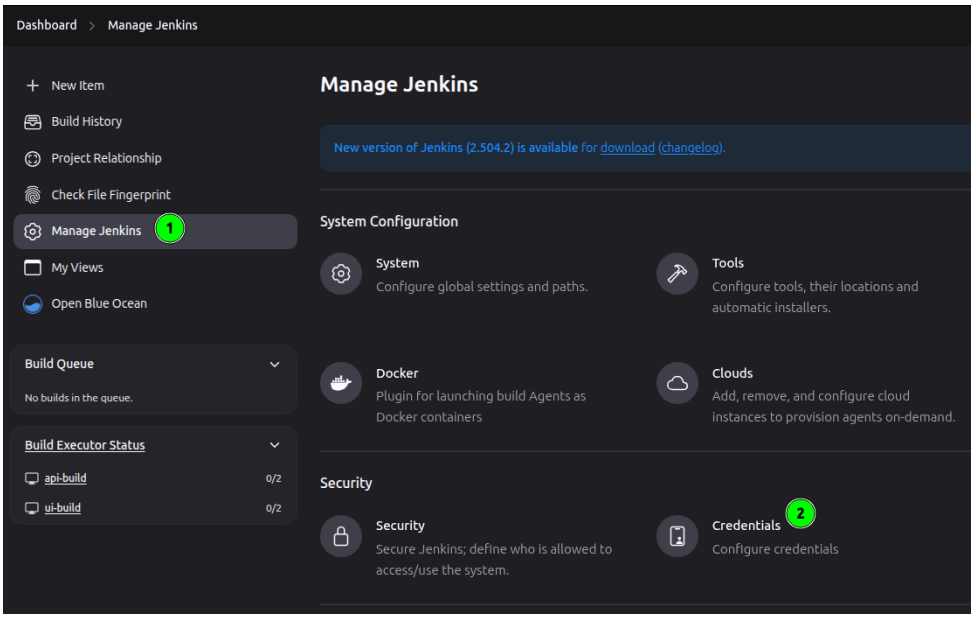
A documentação oficial do *Jenkins* recomenda o uso de credenciais: *"Use credenciais para proteger o acesso a sites e aplicativos externos que podem interagir com o* Jenkins*, como repositórios de artefatos, sistemas e serviços de armazenamento em nuvem e bancos de dados"* (JENKINS, 2025a), e ainda ressalta que o uso das credenciais é mais seguro e prático do que colocar nomes de usuário e senhas diretamente no código de cada pipeline, ou seja, o uso de

[10] Mais configurações estão disponíveis na documentação oficial. Dispinível em: *https://www.jenkins.io/doc/book/pipeline/multibranch/#creating-a-multibranch-pipeline*. Acesso em: 14 de fevereiro de 2025.

credenciais também é uma forma de reutilização desses dados em vários pipelines diferentes ou em vários pontos do mesmo pipeline.

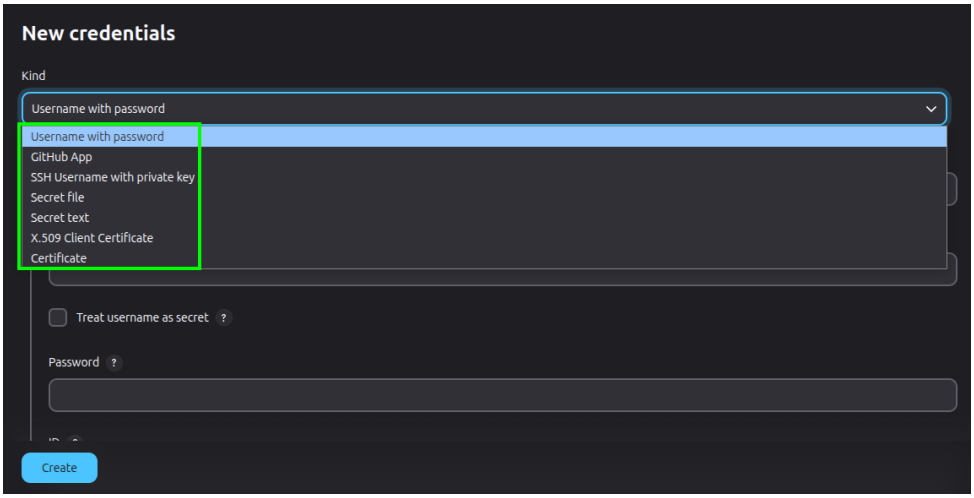
Para configurar uma credencial, é necessário acessar a *url* do *Jenkins*, em seguida, clicar em ***Manage Jenkins > Credentials > System > Global credentials > Add Credentials***[11]. A Figura 26 exibe a interface do *Jenkins* e os menus para acessar a criação de credenciais.

Figura 26 – Acessando o menu de credenciais




Fonte: Elaborado pelo autor

Figura 27 – Tipos de credenciais disponíves para criação




Fonte: Elaborado pelo autor

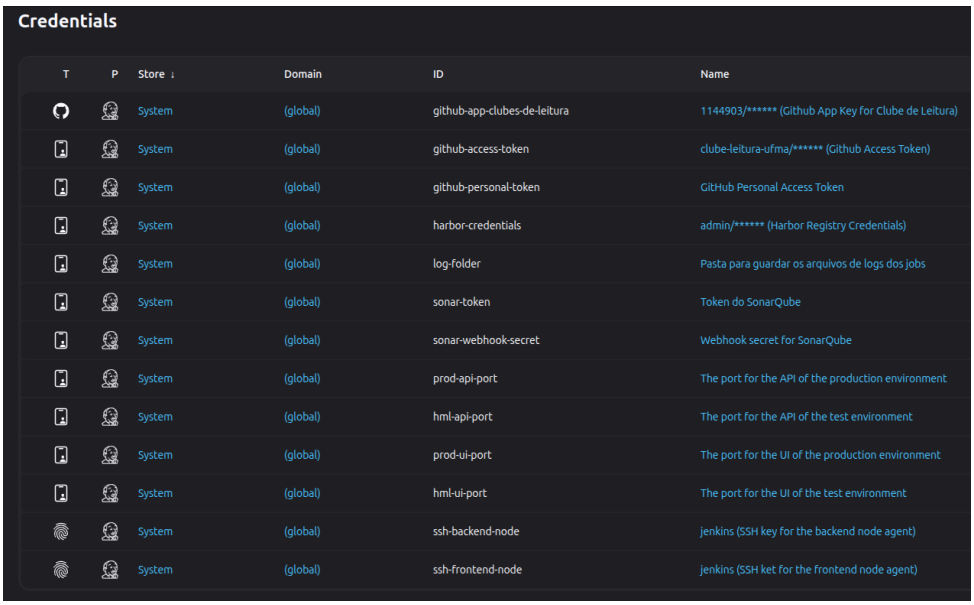
No formulário de criação, é necessário escolher o tipo de credencial (Figura 27) e, dependendo do tipo, preencher outros campos adicionais. Um campo comum a todos os tipos

[11] Para mais informações: *https://www.jenkins.io/doc/book/using/using-credentials/#adding-new-global-credentials*. Acesso em: 16 de fevereiro de 2025

de credenciais é o **ID**, que a própria interface gráfica do *Jenkins* define como um identificador exclusivo pelo qual as credenciais são identificadas nos pipelines ou em outras configurações. Segundo Jenkins (2025a), é uma boa prática padronizar os identificadores das credenciais, portanto, para o *Jenkins* do Clubes de Leitura, os **ID** são palavras que contextualizam a credencial separadas por traços, por exemplo *token-github-jenkins* seria um identificador para um token que integra o *GitHub* ao *Jenkins*.

Além disso, a documentação oficial ressalta que o campo **ID** é opcional e, se não for especificado nenhum valor, o *Jenkins* atribuirá um valor de **ID** globalmente exclusivo (GUID)[12] para o identificador da credencial. E também que, uma vez definido o **ID** de uma credencial, ele não poderá mais ser alterado. A Figura 28 traz uma lista das credenciais e dados sensíveis configurados para utilização no pipeline do Clube de Leitura.

Figura 28 – Lista de credenciais criadas para o pipeline do Clubes de Leitura




Fonte: Elaborado pelo autor

Figura 29 – Atribuição de credencial através de função

```
// Credentials
PROD_PORT = credentials("prod-api-port")
HML_PORT = credentials("hml-api-port")
```

Fonte: Elaborado pelo autor

[12] É um identificar único no formato 7c3f7cc8-d3ae-43d7-a391-bad5e6a694dd.

Para usar uma credencial no pipeline declarativo é necessário fazer referência ao identificador dado à credencial durante a sua criação. A principal forma de referenciar uma credencial em pipelines declarativos é escrever o **ID** nos parâmetros correspondentes às credenciais. Por exemplo, pode-se atribuir o valor de uma credencial a uma variável de ambiente utilizando a função *credentials(ID)* como mostrado na Figura 29. Outra forma é escrever o **ID** da credencial como parâmetro quando se utiliza o contexto *withCredentials* exemplificado na Figura 30.

Figura 30 – Atribuição de credencial através de contexto

```
withCredentials([string(credentialsId: 'github-personal-token', variable: 'TOKEN')]) {
    sh(script: """
        curl -L \
          -X POST \
          -H "Accept: application/vnd.github+json" \
          -H "Authorization: token \$TOKEN" \
          -H "X-GitHub-Api-Version: 2022-11-28" \
          ${GITHUB_API_URL}/releases \
          -d '{
            "tag_name": "${tag}",
            "name": "${tag}",
            "body": "Release criada automaticamente pelo pipeline CI/CD",
            "draft": false,
            "prerelease": false,
            "generate_release_notes": true,
            "make_latest": "true"
          }'
    """)
}
```

Fonte: Elaborado pelo autor

### 5.6.3 Estrutura do Pipeline

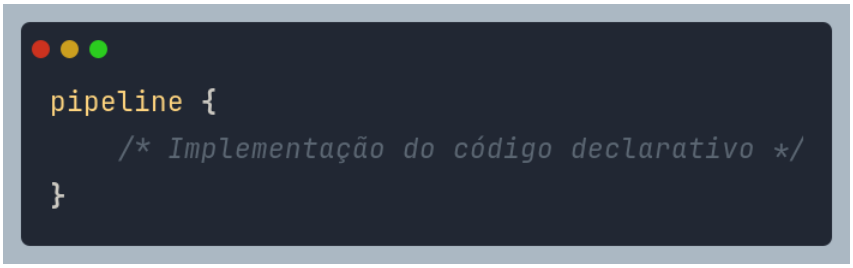
Um pipeline do *Jenkins* é uma *Domain-Specific Language* (DSL) que pode ter uma sintaxe baseada em *script* ou declarativa e segue sempre uma estrutura. Segundo Pathania (2016), pipelines baseadas em *script* são mais flexíveis e extensíveis que as pipelines declarativos, porém os pipelines declarativos são mais simples de escrever e implementar. Para o projeto do pipeline do Clubes de Leitura, escolheu-se utilizar o *pipeline declarativo*.

A sintaxe do Pipeline Declarativo é bem estruturada e uma pessoa sem muito conhecimento pode aprender mais rapidamente, o que facilita a implementação e manutenção. A sintaxe deste tipo de pipeline aceita o uso de código *Groovy*, mas exige uma abordagem bem mais estruturada. Além disso, o pipeline declarativo facilita a leitura e a compreensão do código, apesar de cenários que exigem uma personalização maior se saírem melhor com o pipeline com *script*.

> Um Pipeline é um modelo definido pelo usuário de um pipeline de CD. O código do pipeline define todo o seu processo de construção, que normalmente inclui etapas para construir um aplicativo, testá-lo e, em seguida, entregá-lo (JENKINS, 2025b).

Segundo a documentação oficial do *Jenkins*, o código de um pipeline declarativo deve estar obrigatoriamente dentro do bloco *pipeline*. Esse bloco é uma parte fundamental para a sintaxe do pipeline declarativo (JENKINS, 2025c). A Figura 31 mostra a estrutura inicial de um pipeline declarativo.

Figura 31 – Bloco *pipeline* no pipeline declarativo




Fonte: Elaborado pelo autor

O código de um pipeline declarativo dentro de um *Jenkinsfile* segue as mesmas regras da sintaxe do *Groovy*[13] com algumas exceções, segundo Jenkins (2025c), como a exigência do bloco *pipeline* (como já explicado), o não uso de ponto e vírgula para separar as instruções, pois cada instrução deve estar em uma linha. Além disso, os blocos devem ser formados apenas por *Sections*, *Directives*, *Steps* ou instruções de atribuição.

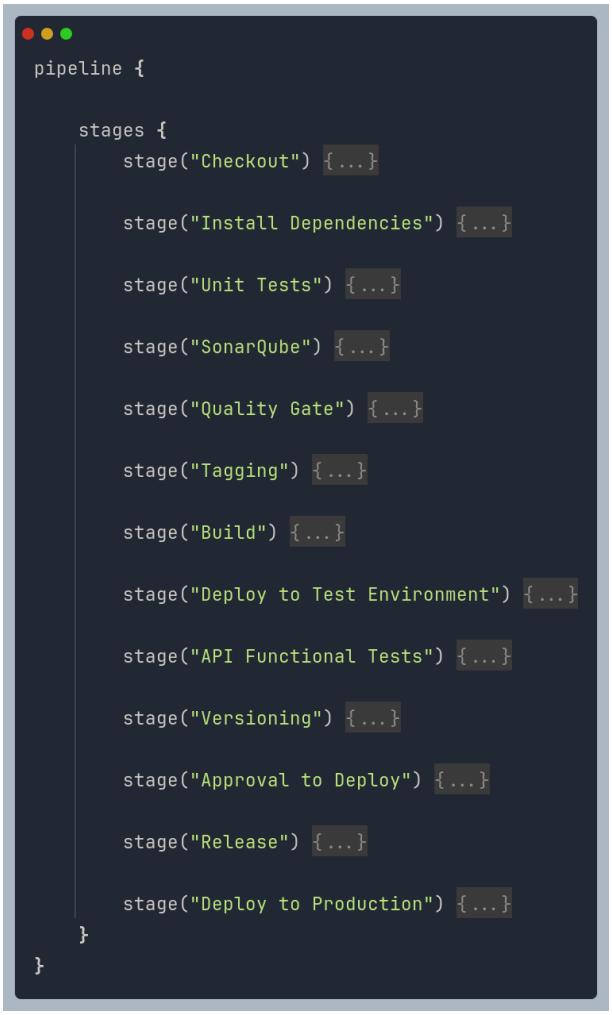
A Figura 32 apresenta a estrutura básica do pipeline declarativo implementado para o Clubes de Leitura. Ele inicia no estágio de *Checkout*, que faz a identificação da ramificação e clona[14] o código para iniciar a construção da imagem. O próximo estágio é responsável por instalar as dependências da aplicação a fim de preparar a aplicação para a execução dos testes unitários no quarto estágio *Unit Tests*. Os dois estágios seguintes são responsáveis por realizar a análise estática do código, onde são checados se o estilo do código está de acordo com o código já existente no projeto, evitar redundância de código e possíveis erros de segurança, e também um pequeno *delay* para aguardar o resultado dessa checagem, respectivamente. O sexto estágio *Tagging* é onde as *tags* das imagens são criadas e o estágio *Build* é onde ocorre a criação da imagem utilizando comandos do *Docker* com auxílio do *Docker Compose*. Todos esses estágios são comuns aos ambientes de produção e homologação (testes).

A Figura 32 ainda mostra os estágios de *deploy* para o ambiente de testes e execução dos testes da API, que são executados apenas nas ramificações *hml* e *prod* apresentadas na Seção 5.5.

[13] É uma linguagem ágil e dinâmica para a *Java Virtual Machine* (JVM) que tem suporte a DSL e outras sintaxes compactas que facilitam a leitura, compreensão e a manutenção do código. Além conter recursos de orientação à objetos e integração com todas as classes e bibliotecas do Java. Disponível em: *https://groovy-lang.org/*. Acesso em: 16 de fevereiro de 2025.

[14] Uma espécie de *download* do código fonte diretamente no servidor.

Figura 32 – Estrutura básica da implementação do pipeline para o Clubes de Leitura



```
pipeline {

    stages {
        stage("Checkout") {...}

        stage("Install Dependencies") {...}

        stage("Unit Tests") {...}

        stage("SonarQube") {...}

        stage("Quality Gate") {...}

        stage("Tagging") {...}

        stage("Build") {...}

        stage("Deploy to Test Environment") {...}

        stage("API Functional Tests") {...}

        stage("Versioning") {...}

        stage("Approval to Deploy") {...}

        stage("Release") {...}

        stage("Deploy to Production") {...}
    }
}
```

Fonte: Elaborado pelo autor

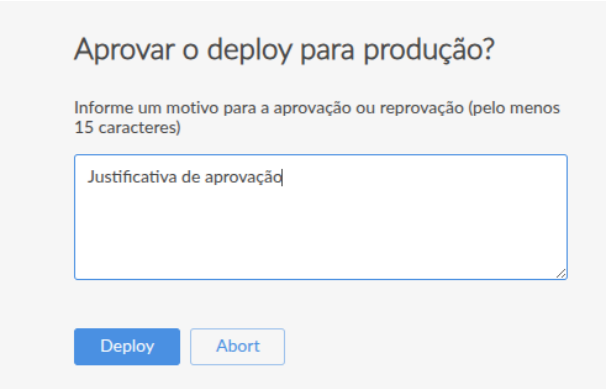
Aqui cabe uma observação, como a proposta do trabalho também estende-se à interface gráfica da aplicação, então o estágio de testes da API não existe no pipeline da interface, neste ponto, é o estágio *End-to-End Tests* que entra em execução. O estágio *Versioning* cria uma *tag* para o *commit* no *GitHub* que é marcada com a *tag* criada no sexto estágio. Os estágios *Approval to Deploy*, *Release* e *Deploy to Production* são executados apenas para a ramificação *prod*. Pois são estágios que criam o pacote de lançamento de versão e implantam as mudanças no ambiente de produção.

### 5.6.4 Aprovação Manual para Deploy

Com o propósito de garantir segurança e qualidade, implementou-se um estágio de aprovação manual do *deploy* para o ambiente de produção. O estágio *Approval to Deploy* exibe um *prompt* que aguarda a confirmação do usuário e a justificativa para prosseguir com a execução do pipeline e implantar as alterações ou novas funcionalidades em produção.

O estágio é interessante pois garante uma camada a mais de segurança, além de delegar responsabilidades quando apenas o administrador do projeto pode aprovar o *deploy*. A Figura 33 mostra o *prompt* exibido quando o estágio é alcançado.

Figura 33 – Estágio *Approval to Deploy*




Fonte: Elaborado pelo autor

Em caso de aprovação, ou seja, caso o usuário clique no botão *Deploy* mostrado na Figura 33, o pipeline segue o fluxo e executa o estágio *Release*, que cria o pacote de lançamento de versão com todas as novas funcionalidades, correções etc. implementadas durante o desenvolvimento. E, por fim, é executado o estágio *Deploy to Production*, que é onde os comandos *Docker* são executados, realizando a substituição da imagem em execução pela nova imagem gerada.

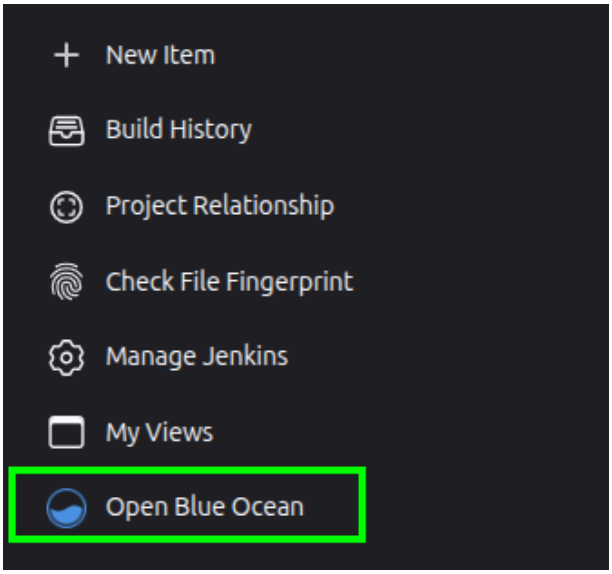
### 5.6.5 Fluxo de Execução do Pipeline

Com o pipeline configurado e implementado, é preciso entender o fluxo de execução do pipeline, e principalmente, quando o pipeline é executado. Para uma melhor visualização, utilizou-se o plugin *Blue Ocean*[15], que traz uma interface mais limpa e focada no pipeline. Para acessar as funcionalidades disponibilizadas pelo *plugin*, basta clicar na opção ***Open Blue Ocean*** no menu localizado na página inicial do *Jenkins* como mostra a Figura 34. A Figura 35a exibe uma execução de sucesso do pipeline, enquanto a Figura 35b exemplifica uma execução em que o pipeline falhou.

Na Figura 35b é possível notar ainda que o pipeline pula todos os estágios após o estágio que falhou, evitando assim execuções indesejadas. Isso é um ponto muito importante, pois, se por acaso ou erro no desenvolvimento do código do pipeline, estágios subsequentes ao estágio que apresentou falha forem executados, funcionalidades ainda em desenvolvimento ou que não foram bem testadas podem levar *bugs* e defeitos para produção. Por tanto, é imprescindível entender o fluxo do pipeline implementado, que complementa o fluxo de trabalho apresentado na Seção 5.5.

[15] Disponível em: *https://plugins.jenkins.io/blueocean/*. Acesso em: 18 de fevereiro de 2025.

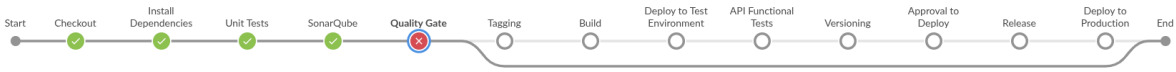
Figura 34 – Botão para abrir a visualização Blue Ocean




Fonte: Elaborado pelo autor

Figura 35 – Visualização do pipeline a partir do plugin Blue Ocean


(a) Visualização do pipeline em caso de sucesso




(b) Visualização do pipeline em caso de falha

Fonte: Elaborado pelo autor.

O *pipeline* inicia automaticamente quando o repositório configurado (Figura 36) recebe um *push* [16]. A execução básica do *pipeline* acontece nas ramificações de apoio (apresentadas na Seção 5.5) e na ramificação *develop*, onde são executados alguns estágios como *Unit Tests*, *SonarQube* e *Quality Gate*. Ou seja, o fluxo de execução do *pipeline* se inicia junto ao início do desenvolvimento de uma nova funcionalidade ou de uma correção.

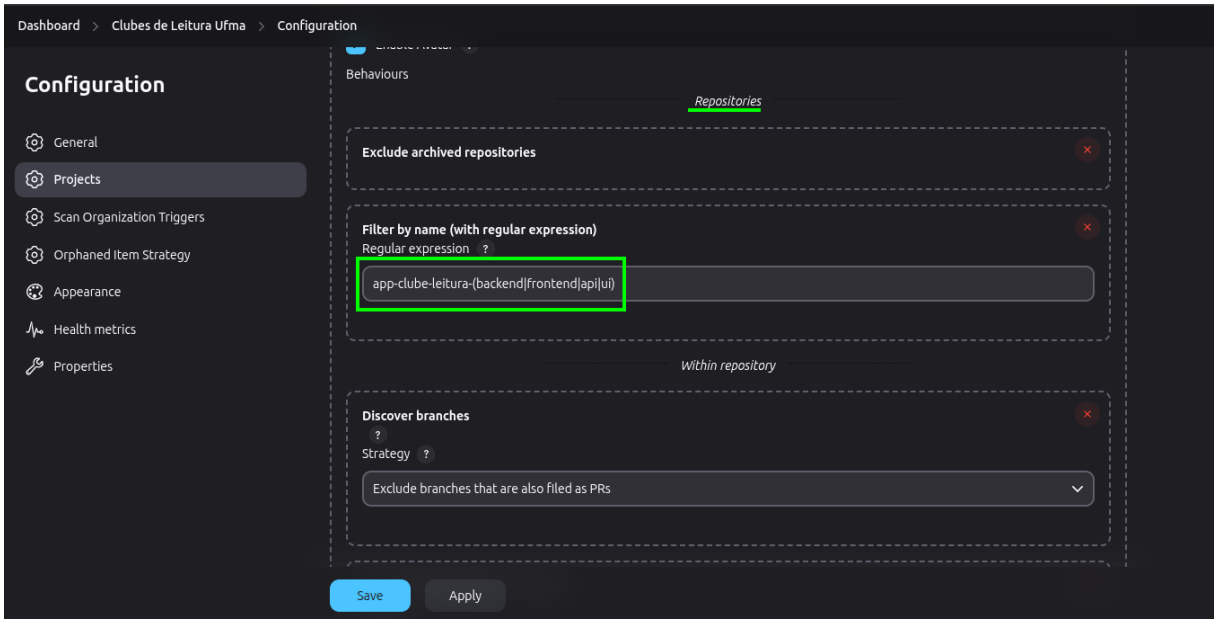
Quando o *pipeline* é executado no contexto apresentado, alguns estágios são pulados propositalmente, como explicado no início desta seção. A Figura 37 mostra a execução do pipeline para a ramificação *develop*.

Após a finalização do desenvolvimento de uma funcionalidade, o desenvolvedor precisa abrir uma solicitação de *pull* (ou *Pull Request* (PR)) [17] da sua ramificação de desenvolvimento

[16] Quando *commits* são enviados do repositório local para o remoto (CHACON; STRAUB, 2014).

[17] São solicitações de mudanças em um repositório onde pessoas podem contribuir com o desenvolvimento (CHACON; STRAUB, 2014, p. 171)

Figura 36 – Configurações do pipeline




Fonte: Elaborado pelo autor

(*feature/***, *breaking-change/***, etc.) para ramificação *develop* para que o pipeline execute e garanta que o código que foi desenvolvido esteja dentro do padrão esperado e que nenhuma funcionalidade seja "quebrada". Em caso de sucesso durante a execução do pipeline, o PR pode ser aprovado e o *merge* pode ser realizado. Quando o *merge* acontece, o pipeline executa novamente, para garantir que nenhuma alteração foi realizada durante esse processo.

Figura 37 – Execução do pipeline das *branches* de apoio e *develop*


Fonte: Elaborado pelo autor

Com isso, um PR da ramificação *develop* para a ramificação *hml* deve ser aberto, assim o pipeline irá executar novamente para analisar o PR, e novamente, com o sucesso na execução, o PR pode ser aprovado e o *merge* realizado. Com o merge concluído, o pipeline é iniciado e estágios como *Deploy to Test EnvironmentAPI* e *Functional Tests* são executados para garantir o correto funcionamento da API após as alterações. Em caso de sucesso, o estágio *Versioning* é executado finalizando o pipeline. Em caso de falha após o merge para *hml*, um estágio extra é executado, fazendo um *rollback* para a imagem anterior. A Figura 38 exibe uma execução de sucesso do pipeline para a ramificação *hml*.

O pipeline ainda tem fluxos para a ramificação de *release*, que, como já explicado na Seção 5.5, é uma ramificação que prepara as mudanças para serem levadas para produção. O fluxo para a execução continua baseado na abertura de PR, após a criação da ramificação. Com a

Figura 38 – Execução do pipeline da ramificação *hml*


Fonte: Elaborado pelo autor

execução do pipeline resultando em sucesso, o PR para a ramificação *prod* pode ser aprovado e o *merge* pode ser iniciado. Com isso, o pipeline é executado, passando por cada estágio (Figura 35a), especialmente os estágios *Approval to Deploy* (explicado na Seção 5.6.4) e *Deploy to Production*; este último executa a imagem com as novas implementações no servidor.

## 5.7 Implementação dos Testes Automatizados

A automação dos testes ocorreu em duas partes principais: testes funcionais da API e testes end-to-end da interface gráfica do usuário. Com essa separação, garantiu-se uma boa cobertura da lógica de negócio da aplicação e da experiência do usuário final. Nas seções a seguir, são detalhadas as tecnologias utilizadas, a estrutura dos testes e os principais desafios enfrentados durante o desenvolvimento.

A arquitetura do projeto de testes pode ser encontrada no Anexo B, onde a Figura 56 mostra em detalhes o processo desde as alterações feitas pelos desenvolvedores, passando pelo pipeline, repositórios de testes automatizados, até o *deploy* nos ambientes no servidor.

### 5.7.1 Testes da API

Os testes funcionais da API foram automatizados utilizando a linguagem Java com auxílio da biblioteca *RestAssured*. Essa escolha se deu pela familiaridade e experiência do autor deste trabalho com a linguagem, pela robustez da mesma, e também pela facilidade de integração do RestAssured com *frameworks* de testes como *JUnit*.

O projeto de testes foi organizado seguindo uma estrutura de pacotes e classes que correspondem aos módulos da aplicação, facilitando a manutenção e a escalabilidade do projeto. Cada classe de teste foi desenvolvida para validar um *endpoint* específico da API, cobrindo os cenários de sucesso e falha de acordo com os cenários descritos no plano de teste (Anexo C). Tais cenários abrangem:

- **Cenários positivos**: onde são realizadas requisições com dados válidos e espera-se uma resposta de sucesso.
- **Cenários negativos**: Requisições com dados inválidos ou incompletos são realizadas para simular casos de erro.

- **Validação de contrato**: Verificação da estrutura da resposta, os tipos de dados, os campos que devem ser retornados, código de status HTTP correto e mensagens de erro.

Figura 39 – Estrutura de um teste do RestAssured

```
@Test  Claudio Henrique *
@Tag("user")
@Tag("api")
@Tag("funcional")
@DisplayName("CT048 - Não deve criar um usuário com e-mail já cadastrado")
public void testNotCreateUserWithExistingEmail() {

    registeredUser = userService.createUser(newUser);

    UserBasicSave secondUser = UserFactory.createValidUser();
    secondUser.setEmail(newUser.getEmail());

    FailureResponse response = given() RequestSpecification
                                        .spec(Specification.request())
                                        .body(secondUser)
                                    .when()
                                        .post(endpoint) Response
                                    .then() ValidatableResponse
                                        .spec(Specification.response(HttpStatus.SC_CONFLICT))
                                        .body(matchesJsonSchemaInClasspath( pathToSchemaInClasspath: "schemas/failure-response-schema.json"))
                                        .extract() ExtractableResponse<Response>
                                        .response() Response
                                        .as(FailureResponse.class);

    Assertions.assertNotNull(response, message: "A resposta de erro não deve ser nula");
    Assertions.assertNotNull(response.getError(), message: "A mensagem de erro não deve ser nula");
    Assertions.assertFalse(response.getError().isEmpty(), message: "A mensagem de erro não deve ser vazia");
    Assertions.assertFalse(response.getError().isBlank(), message: "A mensagem de erro não deve ser em branco");
    Assertions.assertEquals( expected: "Esse email já está sendo usado!", response.getError(), message: "A mensagem de erro não corresponde ao esperado");
}
```

Fonte: Elaborado pelo autor

A Figura 39 mostra a estrutura de um teste escrito usando a biblioteca *RestAssured*. O teste inicia utilizando classes utilitárias como *UserFactory* para gerar dados que serão utilizados na requisição. Em seguida, é possível ver métodos do *RestAssured* como *given()*, *when()* e *then()* que seguem a linguagem *Gherkin* (explicada na Seção 2.9). É no método *given()* que o *payload* é adicionado no corpo da requisição, *when()* é o responsável por realizar a requisição e o *then()* faz algumas validações, como o formato e o *status code* da resposta. Por fim, o teste faz validações gerais do conteúdo da resposta, garantindo que o *endpoint* testado está respondendo corretamente.

Um ponto importante foi a implementação de classes utilitárias com o intuito de facilitar a criação de *payloads* e produção de dados dinâmicos para os testes, como já citado, e que reduziu a duplicação de código e aumentou a legibilidade dos testes. A Figura 40 exemplifica a classe responsável por criar dados para testes que necessitam de um usuário.

### 5.7.2 Testes da Interface

Para a implementação dos testes end-to-end da interface gráfica utilizou-se o *framework* Cypress, escrito em JavaScript. O Cypress foi escolhido por sua facilidade de configuração,

Figura 40 – Implementação da classe utilitária UserFactory

```java
public class UserFactory {  Claudio Henrique *
    private static final Faker faker = new Faker(Locale.of( language: "pt-BR"));

    public static UserBasicSave createValidUser() {  Claudio Henrique

        String name = generateFullName();
        String pass = generatePassword();
        String email = generateEmail(name);

        return new UserBasicSave(email, pass, pass, name);
    }

    private static String generateFullName() {  new *
        return faker.name().fullName();
    }

    public static String generatePassword() {  new *
        return faker.internet().password();
    }

    public static String generateEmail(String name) {  new *
        String emailPrefix = name.replace( target: ".",  replacement: "").replace( target: " ",  replacement: ".").toLowerCase();
        return faker.internet().safeEmailAddress(emailPrefix).replace( target: "example.com",  replacement: "teste.ufma.br");
    }

    // Demais métodos

    /*...*/
}
```

Fonte: Elaborado pelo autor

execução rápida e suporte à depuração visual dos testes, além de sua vasta e detalhada documentação oficial [18].

O projeto de testes é composto de arquivos separados por funcionalidades, como login, cadastro de usuários, criação de clubes, adição de livros e outras interações com formulários e demais elementos da tela. Onde os testes cobrem um fluxo específico da aplicação, simulam o comportamento real do usuário interagindo com a interface e validam os resultados.

Alguns dos pontos testados foram:

- Fluxos completos de navegação
- Validações de mensagens de erro e sucesso
- Testes de carregamento e exibição de elementos
- Integração com a API verificando os dados exibidos

É possível observar uma das formas de escrever um teste utilizando o *Cypress* na Figura 41. Na estrutura do teste, há a função *it()* que representa o teste em si, e que recebe como

[18] Dísponivel em: *https://docs.cypress.io/*

Figura 41 – Estrutura de um teste do Cypress

```
it( title: "TC010 - Deve criar um clube com informações básicas com sucesso", config: () : void ⇒ {
  cy.get( selector: "@clubData").then( fn: (club : JQuery<HTMLElement> ) : void ⇒ {
    club.literaryGenres = Utils.getTranslate(club.literaryGenres);
    cy.fillBasicCreateClubForm(club);
    cy.intercept(HttpMethods.POST, response: `**${Endpoints.CLUB}/**`).as( alias: "createClub");
    createClubPage.submit();
    cy.wait( alias: "@createClub");
  });
  cy.intercept(HttpMethods.GET, response: `**${Endpoints.CLUB}/**`).as( alias: "getMembers");
  cy.get( selector: "@createClub").then( fn: ({ request, response }) : void ⇒ {
    const club = response.body.success;
    cy.get( selector: "@user").then( fn: (user : JQuery<HTMLElement> ) : void ⇒ {
      cy.wait( alias: "@getMembers");
      clubPage.assertAlertMessage();
      clubPage.assertClubProfile(club);
      clubPage.assertUserRole(club, user);
    });
  });
});
```

Fonte: Elaborado pelo autor

*callback* uma função com os passos do teste. Os passos do teste podem ser identificados na maioria das vezes pela chamada dos métodos de variável nomeadas com o sufixo *Page*, como por exemplo *clubPage* no caso do teste em questão.

Figura 42 – Implementação de comando personalizado no Cypress

```
Cypress.Commands.add( name: "fillBasicCreateClubForm", fn: (club : any ) : void ⇒ {
  new CreateClubPage()
    .fillName(club?.name)
    .fillDescription(club?.description)
    .checkIsPrivate(club?.isPrivate)
    .fillMembersLimit(club?.membersLimit)
    .chooseLiteraryGenres(club?.literaryGenres);
})
```

Fonte: Elaborado pelo autor

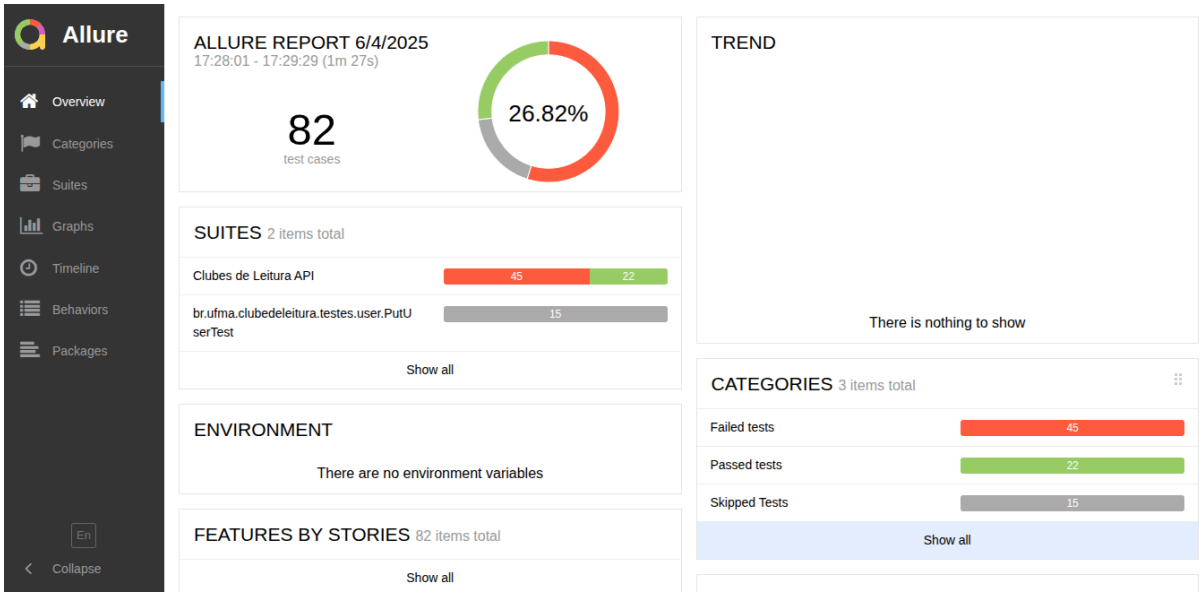
Afim de facilitar a manutenção e a criação de novos testes, foram implementados comandos customizados, que na Figura 41 podem ser identificados pela chamada *cy.fillBasicCreateClubForm(club)*, além de seletores reutilizáveis através de classes específicas, o que também reduziu a duplicação de código e aumentou a legibilidade dos testes. A Figura 42 mostra como é feita a implementação de um comando personalizado no Cypress. Neste caso, é um

comando para preencher o formulário de criação de um clube informando apenas campos básicos para criação, ou seja, preenchendo apenas os campos obrigatórios ou os julgados necessários para a identificação do clube.

## 5.8 Relatório de Teste

O relatório de testes é construído utilizando o *Allure Report*, um *framework* de código aberto que foi projetado especificamente para gerar relatórios detalhados de execuções de testes automatizados. O *Allure* constrói o relatório utilizando uma interface *web* em que os usuários podem interagir e que permite aos envolvidos no desenvolvimento da aplicação analisar os resultados dos testes de forma mais eficiente e intuitiva. A Figura 43 mostra a página inicial do relatório gerado pelo *Allure*.

Figura 43 – Página inicial do relatório gerado pelo Allure Report




Fonte: Elaborado pelo autor

O *Allure Report* oferece várias opções de visualizações dos dados resultantes das execuções de testes, organizadas em *dashboards* específicos. A página inicial apresenta uma visão geral com métricas agregadas, incluindo taxa de sucesso, distribuição de severidade dos testes e tendências históricas quando disponíveis. Cada teste individual pode ser incrementado com anexos como capturas de tela, *logs* detalhados, registros de requisições HTTP e outras evidências personalizadas, dando um contexto completo para a análise do relatório.

É possível integrar o Allure ao Jenkins utilizando o plugin oficial *"Allure Jenkins Plugin"*[19]. Ao instalar esse plugin, o Jenkins ganha a capacidade de suportar a publicação e visualização de relatórios Allure diretamente na sua interface (Figura 44).

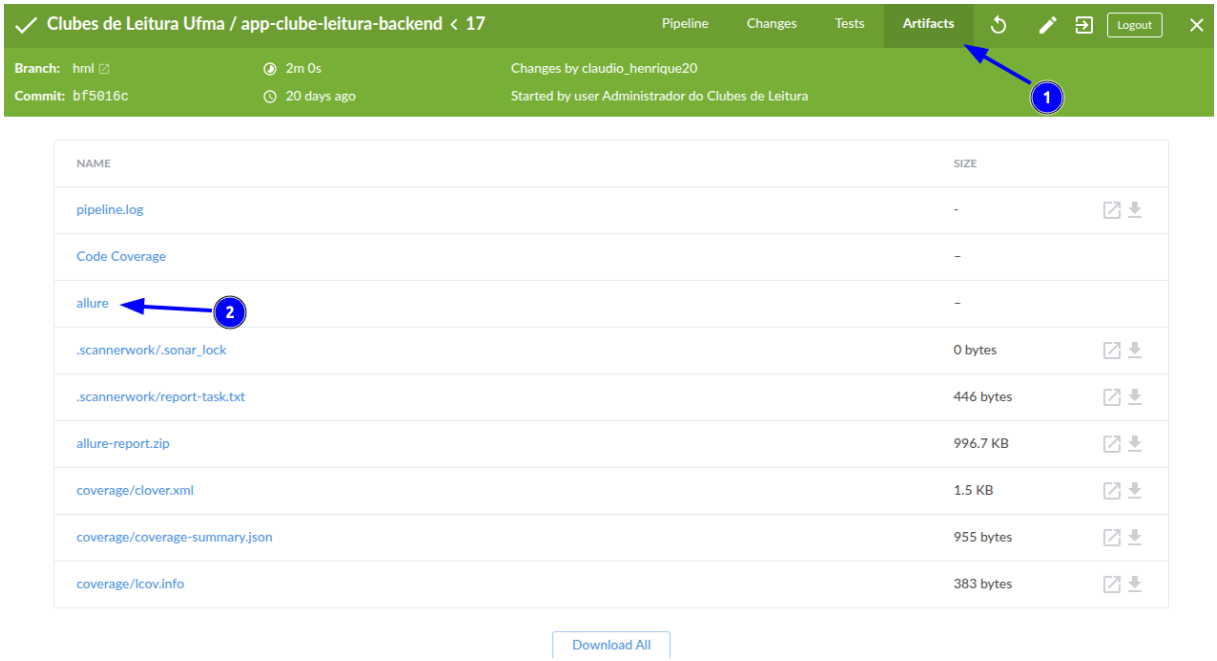
[19] Disponível em: *https://plugins.jenkins.io/allure-jenkins-plugin/*. Acesso em: 20 de maio de 2025

Figura 44 – Visualização do gráfico de tendências diretamente na interface do Jenkins


Fonte: Elaborado pelo autor

No pipeline declarativo, a integração e a geração do relatório exigem a execução dos testes precedida pela publicação dos resultados utilizando a instrução *allure*. Assim, o *plugin* detecta automaticamente os arquivos de resultado, realiza o processamento e disponibiliza o relatório através da URL específica da execução do pipeline no *Jenkins*. Além disso, na visualização *Blue Ocean* apresentada na Seção 5.6.5, é possível acessar a aba ***Artifacts*** e clicar em *allure* para abrir o relatório completo (Figura 45).

Figura 45 – Acessando o relatório de testes pela visualização Blue Ocean




Fonte: Elaborado pelo autor

Os relatórios de testes são gerados a cada execução do pipeline para as ramificações *hml*, *release* e *prod*. Esta abordagem permite que a equipe de desenvolvimento tenha acesso aos relatórios e históricos diretamente da interface do *Jenkins*, mantendo um registro completo

das execuções de teste ao longo do tempo. A funcionalidade de tendência do Allure é bastante interessante neste contexto, pois permite a identificação da queda de qualidade, aumento da instabilidade e padrões de falhas ao longo de múltiplas execuções.

## 5.9 Documentação

A documentação do *software* é um objeto muito importante, pois, de acordo com Sommerville (2019), faz parte do *software* toda a documentação necessária para os usuários da aplicação, para a equipe de garantia de qualidade e para os desenvolvedores. A documentação faz parte da definição de *software*, consistindo em informações descritivas, sejam elas impressas ou virtuais, que descrevem a operação e o uso dos programas (PRESSMAN, 2011). Além disso, a documentação facilita a manutenção e o desenvolvimento contínuo, fornecendo informações sobre a estrutura e a organização da aplicação (SOMMERVILLE, 2019).

Pressman (2011) afirma que a documentação também deve ser coberta pelos testes. No qual os testes podem ser realizados em duas etapas: a etapa de revisão técnica, onde é examinada a clareza do documento; e a etapa de testes usando a documentação em conjunto com a execução da aplicação. Neste trabalho, esse teste foi realizado de forma indireta, sendo executado durante os testes exploratórios utilizando o *Postman*.

A documentação da aplicação alvo deste trabalho limitou-se, até o momento, ao mapeamento dos *endpoints*[20], explicação das configurações do servidor e das ferramentas de apoio para o funcionamento da aplicação, além do plano de testes e dos próprios testes, como destaca Sommerville (2019), pois eles descrevem o que o código deveria fazer e que a leitura dos testes pode facilitar o entendimento do código.

O mapeamento das rotas foi integrado diretamente na API utilizando o *Swagger*, um conjunto de ferramentas de código aberto desenvolvidas com base na especificação *OpenAPI* que auxilia a projetar, construir, documentar e consumir *APIs REST* (SWAGGER, 2025). O *Swagger* foi instalado e utilizado conforme o descrito na sua documentação[21].

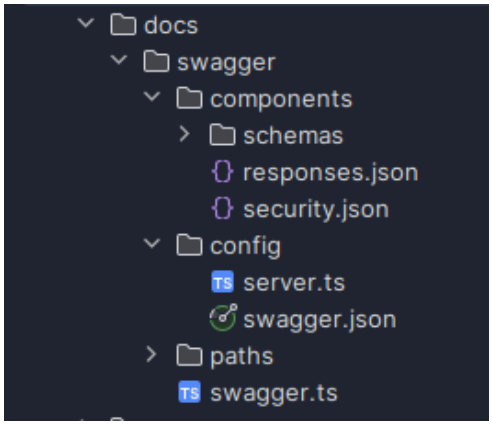
### 5.9.1 Modularização da documentação

Como a aplicação possui inúmeros *endpoints*, optou-se pela estratégia de separação dos arquivos de configuração da documentação. A configuração foi realizada através de arquivos *JSON (JavaScript Object Notation)*, o que permitiu a modularização e integração das configurações juntamente ao código-fonte da API. Isso pode ser visualizado na Figura 46, onde é mostrada a organização das pastas e arquivos da documentação.

[20] Rotas de requisições disponibilizadas para solicitar recursos do servidor.

[21] Disponível em: *https://swagger.io/docs/open-source-tools/swagger-ui/usage/installation/*. Acesso em: 10 de março de 2025.

Figura 46 – Estrutura de pastas do Swagger




Fonte: Elaborado pelo autor

Dentre os arquivos, destacam-se o *swagger.ts* (Figura 47a), que é o responsável por integrar a documentação à API e o *swagger.json* (Figura 47b), onde estão as configurações comuns do *Swagger*. As pastas *paths* e *schemas* armazenam, respectivamente, os arquivos de configuração para a documentação dos *endpoints* e a modelagem dos objetos da aplicação.

Figura 47 – Conteúdo dos arquivos de configuração

```ts
import swaggerJSDoc, { Options, SwaggerDefinition } from "swagger-jsdoc";
import { JsonObject } from "swagger-ui-express";
import swaggerBases from "./config/swagger.json"
import swaggerResponses from "./components/responses.json"
import swaggerPaths from "./paths"
import swaggerSecurity from "./components/security.json"
import swaggerSchemas from "./components/schemas"
import swaggerServer from "./config/server"

const swaggerDefinitions: SwaggerDefinition = {
  ...swaggerBases,
  servers: [ swaggerServer ],
  paths: { ...swaggerPaths },
  components: {
    schemas: { ...swaggerSchemas },
    responses: { ...swaggerResponses },
    securitySchemes: { ...swaggerSecurity },
    parameters: {  },
    examples: {  },
    requestBodies: {  },
    headers: {  },
    links: {  },
    callbacks: {  }
  },
};

const swaggerOptions: Options = {
  definition: swaggerDefinitions,
  apis: []
}

const swaggerDocs: JsonObject = swaggerJSDoc(swaggerOptions);

export default swaggerDocs; Show usages  Claudio Henrique  Henrique, 1
```

(a) Integração do Swagger com a API

```json
{
  "openapi": "3.1.0",
  "info": {
    "title": "Clubes de Leitura API",
    "description": "Documentação da API do backend do Clubes de Leitura - UFMA",
    "version": "1.0.0",
    "contact": {
      "name": "Clubes de Leitura - UFMA",
      "email": "admin@clubedeleitura.ufma.br",
      "url": "https://clubedeleitura.ufma.br"
    }
  }
}
```

(b) Configuração comuns do Swagger

Fonte: Elaborado pelo autor.

### 5.9.2 URLs de Documentação

É possível acessar a documentação dos *endpoints* acessando as URLs listadas na Tabela 13. A documentação apresentada reflete o ambiente em execução, ou seja, caso acesse a URL do ambiente de produção, os *endpoints* apresentados serão os utilizados em produção.

Tabela 13 – URLs de acesso à documentação dos endpoints

| URL | Ambiente |
|---|---|
| https://clubedeleitura.ufma.br/api/docs/swagger/ | Produção |
| https://clubedeleitura-hml.ufma.br/api/docs/swagger/ | Testes/Homologação |
| http://localhost:porta/docs/swagger | Desenvolvimento |

Fonte: Elaborado pelo autor.

## 5.10 Plano de Testes

A construção do plano de testes seguiu as orientações descritas na Seção 2.7 do Capítulo 2, onde é explicado que o plano depende do projeto, dos desenvolvedores e das empresas, variando conforme a necessidade de cada um. Sendo assim, o plano de testes do *app* Clubes de Leitura é uma adaptação da estrutura definida por Sommervile (2019).

O plano de testes encontra-se no Anexo C, e foi baseado em práticas de testes automatizados e na metodologia deste trabalho. A fim de assegurar a eficiência da cobertura das funcionalidades, o plano foi projetado utilizando diferentes níveis de teste:

1. **Testes Unitários**: Após a análise do código-fonte, identificou-se que não existiam testes unitários com uma cobertura satisfatória. Como apresentado na Seção 2.5, a implementação desses testes é normalmente responsabilidade dos desenvolvedores, porém no caso do objeto alvo deste trabalho, tem-se a necessidade de serem executados pelo analista de testes. Os testes unitários foram projetados para verificar o funcionamento isolado das funções e componentes do aplicativo. Para esses testes optou-se por utilizar o *Jest*[22], pois ele é capaz de lidar tanto com os testes unitários da API quanto os testes unitários da interface gráfica, além de ser *open-source*, gratuito e de fácil configuração.

2. **Testes Funcionais da API**: O *Rest Assured* foi o *framework* utilizado para realizar os testes funcionais da API, assegurando que os *endpoints* especificados na documentação respondam corretamente. Esses testes validam os fluxos de autenticação (*login*), manipulação de usuários, clubes e livros na aplicação.

3. **Testes E2E**: Esses testes foram implementados utilizando o *Cypress*, onde testam os fluxos da aplicação de modo geral, simulando as ações do usuário na interface gráfica.

[22] Disponível em: *https://jestjs.io/*

# 6 Resultados

Neste capítulo, serão apresentados os resultados obtidos com a implementação da automação de testes e dos pipelines CI/CD no aplicativo *web* Clubes de Leitura. Serão abordadas a eficácia desta solução, como ela impactou o ciclo de desenvolvimento, a abrangência dos testes realizados e os principais indicadores de desempenho e estabilidade observados durante as execuções dos pipelines. Além disso, serão descritas as métricas geradas pelas ferramentas utilizadas e, por fim, destacam-se algumas melhorias técnicas que podem ser incorporadas nas próximas versões da aplicação, considerando as limitações identificadas ao longo do trabalho.

## 6.1 Avaliação do Sucesso da Implementação

A implementação da automação de testes foi desenvolvida com sucesso, englobando tanto a API quanto a interface da aplicação. Tal sucesso ocorreu devido à integração dos testes funcionais e *end-to-end* às pipelines CI/CD, com ambientes separados para desenvolvimento, testes (homologação) e produção.

Como resultado, foi possível atingir uma cobertura de aproximadamente 30% dos casos de uso e aproximadamente 43% dos requisitos funcionais, com tempo médio de execução dos testes de 2 minutos e 7 segundos para os testes funcionais e 9 minutos e 35 segundos para os testes *end-to-end*. Esses dados demonstram a eficácia da automação e contribuíram para a confiabilidade do desenvolvimento. Dessa forma, o principal objetivo do trabalho foi alcançado: agregar práticas sistemáticas de testes ao ciclo de desenvolvimento do aplicativo Clubes de Leitura.

Apesar da necessidade de executar configurações extras no ambiente, fugindo um pouco do planejamento inicial, tais configurações tiveram um papel essencial para garantir a solidez da automação de testes e a execução das validações em cada estágio do fluxo de entrega.

Os testes foram executados automaticamente a cada nova alteração nas *branches* monitoradas, permitindo a verificação contínua da aplicação, o que ampliou a garantia de qualidade e promoveu mais segurança durante o desenvolvimento.

## 6.2 Testes Exploratórios e Compreensão da Aplicação

Em relação aos testes exploratórios, Tinkham e Kaner (2003) explicam que são testes em que o testador tem total liberdade para realizar mudanças à medida que eles são executados e usar os dados obtidos para aperfeiçoar o projeto de novos testes. Além disso, afirmam, categoricamente, que *"todo testador faz testes exploratórios"*. Portanto, os testes exploratórios

deste trabalho foram executados nas etapas iniciais do projeto com o objetivo de entender os fluxos e rotas da aplicação, descobrir pontos críticos e mapear funcionalidades relevantes para automação.

Os testes exploratórios contribuíram principalmente para o entendimento prático da aplicação, fornecendo conhecimento a cerca das funcionalidades presentes e das regras de negócio. Pois, como destacado por Bach (2003), os testes exploratórios se encaixam em situações como a necessidade de dar *feedback* rápido sobre uma aplicação ou funcionalidade, quando é preciso aprender sobre o produto em pouco tempo, aprimorar e diversificar os testes, encontrar defeitos e falhas críticas no menor intervalo de tempo possível, investigar um defeito específico ou investigar um risco com o objetivo de avaliar a possibilidade de implementar testes automatizados. Desse modo, os testes exploratórios realizados no Clubes de Leitura possibilitaram a identificação de:

- Fluxos e rotas mais relevantes do ponto de vista do usuário.
- Comportamentos sem documentação.
- Validações do frontend e limitações da API.
- Casos de uso mais críticos para priorização de testes automatizados.
- Defeitos e falhas, tanto do frontend quanto da API.
- Melhorias.

### 6.2.1 Cenários Explorados

A Tabela 14 descreve alguns dos cenários de testes exploratórios executados contra a interface *web* da aplicação. A tabela exibe a funcionalidade testada, o cenário e se foram encontrados defeitos ou falhas.

Como mostra a Tabela 14, algumas funcionalidades apresentaram defeitos, como tentar fazer login com formato de e-mail inválido (sem @, por exemplo), fazia o servidor quebrar e retornar o *status code 502 Bad Gateway*, o que, à primeira vista, não parecia ser algo grave, a longo prazo, poderia acarretar em mais inconsistências e até em falhas mais graves. Esse defeito ocorria pois a API tentava enviar duas respostas para requisições feitas na rota */login*, o que causava instabilidade momentânea, fazendo o servidor reiniciar, como mostra a Figura 48.

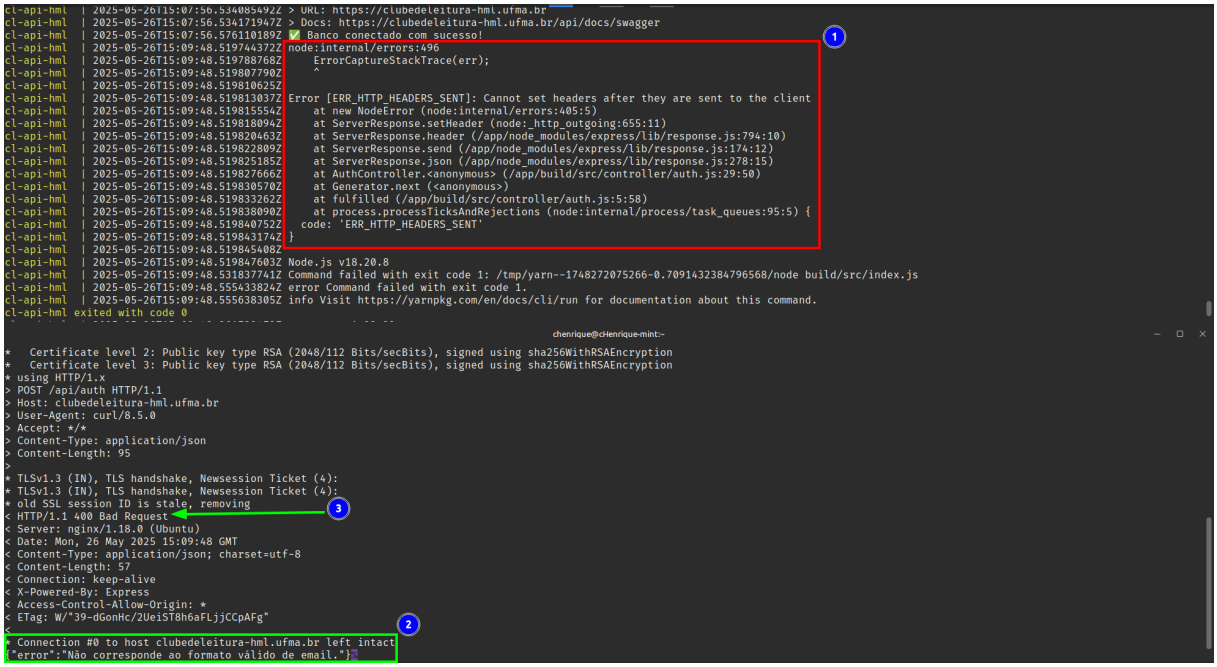
Ainda na Figura 48, é possível ver que, mesmo o servidor respondendo com o *status code* (3) e o corpo da resposta (2) corretos, a aplicação reiniciava por conta do erro destacado no retângulo vermelho (1). Apesar de não estar dentro do escopo de atividades de um analista ou engenheiro de testes, o defeito precisou ser corrigido pois ele impactava diretamente na execução dos demais testes.

Tabela 14 – Testes exploratórios executados

| Funcionalidade | Cenário | Defeito? |
|---|---|---|
| Cadastro de usuário | Todos os campos válidos | Não |
| Login | Credenciais válidas | Não |
| | Senha inválida | Não |
| | E-mail com formato inválido | Sim |
| Criação de clube | Todos os campos válidos | Não |
| | Nome maior que o permitido | Sim |
| | Imagem com formato inválido | Sim |
| | Imagem maior que 1MB | Sim |
| | Arquivo com formato inválido | Sim |
| Cadastro de livro | Todos os campos válidos | Não |
| | Título vazio | Não |
| | Sem foto de capa | Não |

Fonte: Elaborado pelo autor.

Figura 48 – Logs do servidor




Fonte: Elaborado pelo autor

Outras funcionalidades também apresentaram defeitos, como o cadastro de clubes, onde não houve validação ao tentar enviar imagens com formato inválido, com tamanho maior que o permitido e enviar o arquivo de regras com formato inválido, e em muitos casos cadastrando o clube, mas não criando a imagem ou o arquivo. O cadastro de clubes apresentou ainda uma inconsistência de informação a respeito da mensagem de alerta ao inserir um nome para o clube, onde é exibida a mensagem *"Nome deve ter entre 1 e 50 caracteres."* porém, o perfil do clube só exibe até 20 caracteres.

Quanto ao cadastro de livros e adição na estante do clube, a funcionalidade não apresentou defeitos durante testes exploratórios iniciais. Porém, ao automatizar testes dessa funcionalidade, o cadastro passou a retornar o *status code 500 Server Error*. A partir daí, qualquer tentativa de cadastro de um novo livro resultou em falha, apesar de a interface exibir uma mensagem de sucesso ao cadastrar o livro.

## 6.3 Impacto da Automação de Testes

A princípio, a aplicação não contava com estratégias formais de testes. Todas as validações e verificações eram realizadas de forma manual e eventual, sem qualquer rastreabilidade, documentação ou planejamento, onde muitas vezes eram realizadas diretamente no ambiente de produção, visto que não havia um ambiente exclusivo para testes. Sendo assim, não seguia as recomendações de separação dos ambientes da ISO/IEC (2022), cujo objetivo é padronizar o isolamento do ambiente de produção para que este não seja afetado por dados gerados pelos testes e pelo desenvolvimento de novas funcionalidades.

Um dos impactos desta implementação está em consonância com o que foi descrito por Bernardo e Kon (2008), ao apontarem que a automação de testes proporciona reprodutibilidade, permitindo que um cenário específico seja executado diversas vezes. Essa característica ajuda a garantir que etapas necessárias não sejam esquecidas devido à falha humana, além de possibilitar a identificação de comportamentos inesperados.

Adicionalmente, como os testes automatizados são escritos em códigos que são executados por um computador, Bernardo e Kon (2008) apontam que tais testes possibilitam a construção de cenários muito mais completos em comparação a testes manuais. E conclui que mudanças na aplicação podem ser realizadas com segurança, pois os testes automatizados são fáceis de serem executados.

A implementação dos testes automatizados neste projeto trouxe alguns benefícios:

- A detecção de falhas ocorre mais cedo, ainda durante o desenvolvimento e antes do *deploy* em produção.
- Redução do risco de reaparecimento de falhas que já foram resolvidas, principalmente em funcionalidades mais críticas.
- Confiabilidade durante o processo de *deploy* nos ambientes de homologação e produção.
- Facilidade de expandir o pipeline CI/CD de forma segura.

Adicionalmente, tem-se a possibilidade da utilização dos testes para exercer o papel de documentação do comportamento esperado da aplicação, facilitando a introdução da aplicação para novos desenvolvedores.

A automação impactou diretamente na cobertura de alguns dos principais fluxos de negócio da aplicação. Na API, foram criados testes funcionais para os *endpoints* de autenticação, cadastro de usuários e listagem de clubes do usuário. Na interface gráfica, foram testadas as rotas, formulários e fluxos de navegação mais utilizados, como login, cadastro de usuários e criação de clubes. Apesar da cobertura não ter sido total, atendeu às demandas funcionais essenciais do aplicativo.

Um exemplo de teste falhado no *RestAssured* e no *Cypress* pode ser visto nas Figuras 49 e 50, respectivamente. A Figura 49 exibe o resultado de um teste da API com cenário em que não deveria ser possível cadastrar um usuário com e-mail vazio, e como a imagem mostra, o teste falhou porque o usuário foi criado. A falha acontece logo ao identificar o *status code 200 Success* da resposta, quando deveria ser *400 Bad Request* e uma mensagem de falha no corpo da resposta como, por exemplo, *"E-mail inválido"* ou *"O e-mail é obrigatório"*.

Figura 49 – Indicação de falha de teste no RestAssured

```
10:05:01.612 [main] INFO br.ufma.clubedeleitura.testes.user.PostUserTest -- Verificando o status da API...
10:05:01.615 [main] INFO br.ufma.clubedeleitura.testes.user.PostUserTest -- API está online.
10:05:01.853 [main] INFO br.ufma.clubedeleitura.testes.base.AllureTestWatcher -- 🔄 Iniciando teste: CT050 - Não deve criar um usuário com e-mail vazio
10:05:03.934 [main] INFO br.ufma.clubedeleitura.utils.CurlFilter -- Você pode usar o curl abaixo para reproduzir a request de teste:

curl --location --request POST https://clubedeleitura-hml.ufma.br/api/user \
--header 'Accept: */*' \
--header 'Content-Type: application/json' \
--data-raw '{"email":"","password":"5w46e49r3g6lr1","passwordConfirm":"5w46e49r3g6lr1","fullname":"Suélen Alves"}'

10:05:03.979 [main] INFO br.ufma.clubedeleitura.testes.base.AllureTestWatcher -- ▶️ Teste concluído: CT050 - Não deve criar um usuário com e-mail vazio
10:05:03.984 [main] ERROR br.ufma.clubedeleitura.testes.base.AllureTestWatcher -- ❌ O teste falhou: CT050 - Não deve criar um usuário com e-mail vazio
java.lang.AssertionError: 1 expectation failed.
Expected status code <400> but was <200>.
> <1 internal line>
>     at java.base/java.lang.reflect.Constructor.newInstanceWithCaller(Constructor.java:502) <6 internal lines>
><85 folded frames>

java.lang.AssertionError: 1 expectation failed.
Expected status code <400> but was <200>.

> <1 internal line>
>     at java.base/java.lang.reflect.Constructor.newInstanceWithCaller(Constructor.java:502) <6 internal lines>
><85 folded frames>
```

Fonte: Elaborado pelo autor

A Figura 50 traz um cenário de teste E2E, onde testa a criação de usuários contendo *script injection*, por exemplo, inserir *"<script>alert('XXS')</script>"* no campo do nome. Neste caso, não deveria ser possível cadastrar um usuário contendo *scripts* maliciosos, porém, pode-se observar que não há validação quanto a isso, pois o usuário foi criado com sucesso. A mensagem em *AssertionError* diz que não encontrou o elemento de alerta na tela, resultando em uma falha.

## 6.4 Relatórios e Métricas

De acordo com Sommerville (2019), diversas métricas podem ser usadas para avaliar um *software* e seus atributos. E um valor para a qualidade da aplicação pode ser determinado

Figura 50 – Indicação de falha de teste no Cypress


Fonte: Elaborado pelo autor

com base nos resultados das medições. Além do mais, se um *software* tivesse alcançado o limite de qualidade exigido, não precisaria ser aprovado após teste e revisão. Adicionalmente, define métricas como características de uma aplicação que podem ser medidas.

Com relação aos tipos de métricas, Sommerville (2019) exemplifica que a quantidade de linhas de código, o número de falhas encontradas e o tempo exigido para corrigir defeitos estão entre as características que podem ser medidas em um *software*.

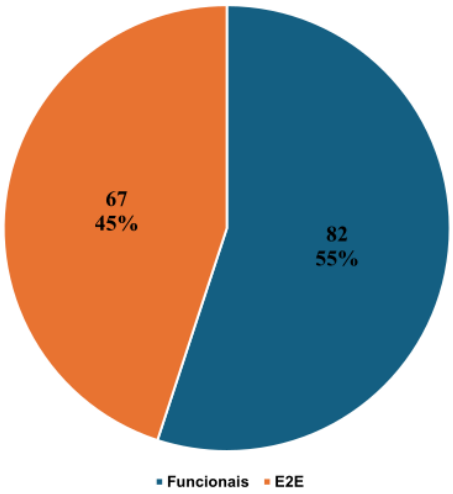
Sendo assim, a produção de relatórios e a captura de métricas que foram executadas com a utilização da ferramenta *Allure Reports*, integrada aos pipelines, possibilitou a visualização detalhada dos resultados dos testes, apresentando informações como testes executados, falhas encontradas, tempo de execução e evidências das execuções. E como explicado anteriormente, esses dados podem ser utilizados para medir a qualidade da aplicação.

Os painéis interativos gerados pelo Allure facilitaram a análise dos testes executados tanto para a API quanto para a interface da aplicação. Os relatórios são compostos por capturas de tela em casos de falha nos testes E2E com Cypress, assim como *logs* detalhados das requisições efetuadas nos testes funcionais executados com Rest Assured. As principais métricas obtidas ao longo da execução dos *pipelines* foram:

- Total de testes implementados: **149**
- Total de testes funcionais: **82**
- Total de testes E2E: **67**
- Média de tempo por execução de testes funcionais: **02 min 07 seg 721 ms**
- Média de tempo por execução de testes E2E: **09 min 35 seg**
- Taxa de sucesso nos testes funcionais: **27%**
- Taxa de sucesso nos testes E2E: **61%**

- Número total de falhas (funcionais + E2E): **86**
- Taxa de sucesso total (funcionais + E2E): **42%**

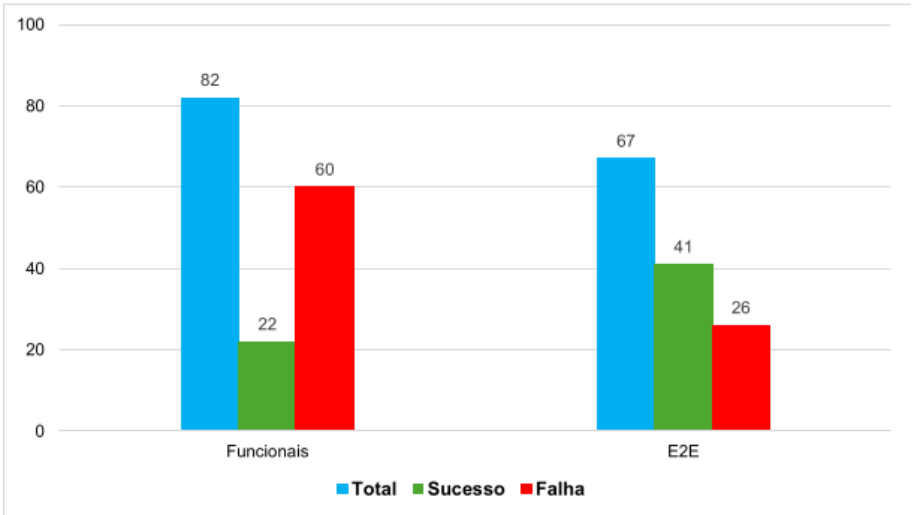
Figura 51 – Gráfico do total de testes implementados




Fonte: Elaborado pelo autor

A Figura 51 ilustra visualmente a quantidade total de testes implementados, onde o total consiste no resultado da soma de testes funcionais e testes E2E. Enquanto a Figura 52 exibe o resultado da execução dos testes e a quantidade de sucessos e falhas em relação ao total de testes implementados em cada um dos tipos de testes.

Figura 52 – Resultado das execuções dos testes



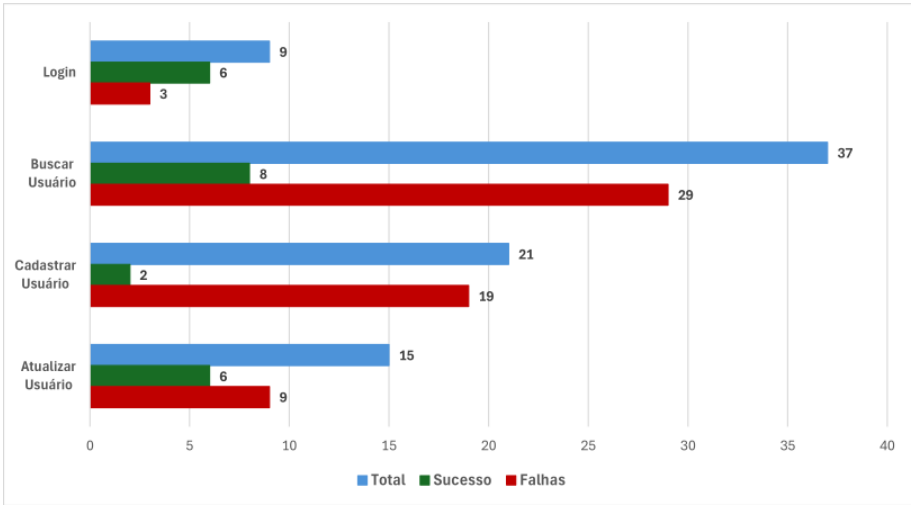

Fonte: Elaborado pelo autor

Os gráficos das Figuras 53 e 53 agrupam os resultados de testes por funcionalidade, mostrando a quantidade de sucesso e falha em relação ao total de testes implementados e executados. Com base nos dados dos gráficos, é possível perceber que a aplicação apresenta

problemas de qualidade. Os testes realizados na API indicam uma taxa de 90% de falha no cadastro e outros 78% na busca de usuário, enquanto os testes E2E evidenciam uma estabilidade, com exceção da funcionalidade de adicionar livros, que alcançou uma taxa de 67% de falha. Essa diferença revela que a interface contém mais tratamentos de dados inválidos e outras exceções em relação à API, e que os defeitos decorrentes da ausência de validação na API nem sempre são propagados para o fluxo completo do usuário.

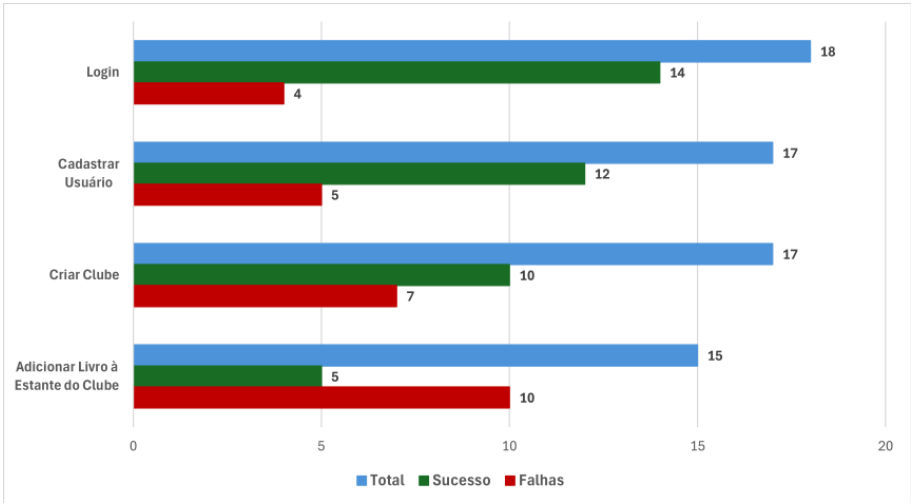
A funcionalidade que demonstrou mais consistência nos dois níveis de testes, baseando-se exclusivamente nos resultados dos testes automatizados aplicados, foi o *Login*, o que sugere que houve mais cuidado durante a implementação.

Figura 53 – Resultado das execuções dos testes funcionais por funcionalidade




Fonte: Elaborado pelo autor

Figura 54 – Resultado das execuções dos testes E2E por funcionalidade




Fonte: Elaborado pelo autor

Diante dos resultados apresentados, surge a necessidade de refatoração nas operações de cadastro, busca e edição de usuários e revisão da arquitetura de dados antes que a aplicação

entre em produção. Porém, vale citar que, "*Um teste bem-sucedido é aquele que faz com que o sistema opere incorretamente e, como consequência, expõe um defeito existente, enfatizando assim um fato importante sobre os testes*" (SOMMERVILLE, 2019), ou seja, todos os defeitos aqui relatados não são para criticar negativamente a aplicação ou os desenvolvedores, mas para auxiliar na obtenção de melhores níveis de qualidade.

Detalhes sobre os defeitos encontrados podem ser vistos no Apêndice A.

## 6.5 Sugestões de Melhorias

Os testes, principalmente os exploratórios, ajudaram a identificar pontos de melhoria na aplicação. Algumas sugestões estão listadas a seguir:

1. A página *Buscar Clubes* não possui um campo de pesquisa. A medida que os testes foram executados, a lista de clubes foi se estendendo até um ponto em que encontrar um clube específico se tornou algo demorado e cansativo. A sugestão é implementar a funcionalidade de pesquisa por nome do clube e talvez por gêneros literários, para uma busca mais refinada.

2. Na página *Buscar Livros* a lista de livros não possui paginação, dando a entender que existem apenas os livros exibidos na tela. Apesar disso, a pesquisa por título do livro funciona perfeitamente. A sugestão é adicionar paginação na lista de livros, semelhante ao que ocorre na lista de clubes.

3. Outra oportunidade de melhoria na página *Buscar Livros* é a possibilidade de buscar livros por autor. Visto que, o usuário pode não lembrar do título do livro mas do nome do autor, logo pesquisar o livro informando o nome do autor seria uma boa saída.

4. Atualmente, para um usuário que administra ou modera vários clubes trocar entre os perfis dos clubes é necessário sair da aplicação e entrar novamente, e assim escolher outro clube que deseja acessar. A sugestão de melhoria é implementar a trocar de perfis de clubes sem necessariamente fazer *login* novamente toda vez que quiser trocar de clube.

5. Em alguns momentos, como a seleção de clube ao realizar login como moderador ou administrador, o código-fonte da página exibe os dados do clube. Apesar de não serem dados considerados sensíveis, é interessante analisar onde isso acontece para remover esses dados.

As sugestões de melhorias apontadas nessa seção podem ser implementadas em versões futuras do projeto, após refinamento, classificação de prioridade e viabilidade, com a equipe de desenvolvimento, garantindo melhor usabilidade e experiência para o usuário, além de elevar a qualidade geral da aplicação.

# 7 Conclusão

A aplicação de testes em *softwares* se tornou uma etapa fundamental no ciclo de desenvolvimento, visto que a ausência de testes pode provocar prejuízos financeiros devido ao aparecimento de defeitos não encontrados durante o desenvolvimento. Além da exigência por aplicações mais confiáveis, tendo em vista o alto investimento no desenvolvimento dessas aplicações. Com um bom planejamento e uma execução eficiente, os testes podem produzir resultados expressivos. Sendo assim, quanto maior a quantidade de testes, mais erros serão encontrados e mais cedo serão corrigidos e, então, melhor e mais correta será a aplicação. Contudo, uma quantia significativa de testes possui um custo elevado. Cerca de 40% a 50% dos esforços durante o desenvolvimento são consumidos pela etapa de testes. Mesmo que isso pareça um cenário favorável para a automação de testes, é comum que projetos de *software* não utilizem essa alternativa, adotando uma execução exclusivamente manual, menos formal e menos rigorosa dos testes.

A principal proposta deste trabalho era implementar técnicas metódicas de testes de *software* automatizados com o objetivo de garantir a qualidade e o aprimoramento da aplicação *web* Clubes de Leitura. Ao longo da pesquisa, abordou-se a implementação de um pipeline de CI/CD, com foco na qualidade de *software*. O pipeline desenvolvido integrou diversas ferramentas, como Jenkins, Docker, Rest Assured, Cypress, entre outras, para construir um ambiente com robustez, sustentabilidade e escalabilidade, e que estivesse pronto para suportar o ciclo de vida de desenvolvimento de *software*.

A produção do pipeline automatizado com a capacidade de executar testes funcionais da API e end-to-end integrado ao processo de versionamento e *deploy*, é com certeza o principal resultado obtido. Adicionalmente, a geração de relatórios detalhados possibilitou a captura de métricas e o acompanhamento da execução dos testes. Dessa forma, houve melhora na prevenção de falhas e no controle de qualidade da aplicação, mesmo que não tenhamos dados anteriores para efeito de comparação, devido à falta da prática de realização de testes.

Destaca-se ainda o baixo tempo para execução dos testes automatizados, com um tempo médio de aproximadamente **2 minutos e 7 segundos** para os testes funcionais e **9 minutos e 35 segundos** para os testes end-to-end. Assim, os testes automatizados podem ser rodados frequentemente, garantindo testes de regressão que auxiliam na validação de funcionalidades prévias. É preciso ressaltar que a presença de testes automatizados não eliminou a necessidade da execução dos testes manuais e exploratórios, que foram de grande importância para o entendimento da aplicação e de seus fluxos e serviram significativamente como base para o projeto e implementação dos testes automatizados, com destaque para os testes end-to-end.

Entretanto, as etapas de Testes Unitários e Análise Estática do código não foram plenamente concluídas, visto que o esforço maior na implementação do pipeline e dos testes funcionais e E2E ocupou grande parte do tempo disponível. Porém, isso não é um resultado necessariamente negativo, já que um plano de testes é um documento que evolui durante o desenvolvimento. Com isso, o plano pode sofrer alterações devido a atrasos nas outras etapas do desenvolvimento.

Ademais, uma limitação clara observada durante o desenvolvimento deste trabalho está associada ao impedimento de uso de funcionalidades mais avançadas em ferramentas como GitHub e SonarQube, que, apesar de serem gratuitas, suprimem a possibilidade de uso de algumas funcionalidades que trariam grande ganho ao projeto, como mais opções de segurança em PRs e mesclagens de ramificações, além de mecanismos mais aprimorados de análise de código. E tais recursos estão presentes apenas em planos pagos.

Diante dos resultados de testes apresentados, a alta presença de defeitos, principalmente na API, deve-se à ausência de planejamento e execução de testes anteriores a este trabalho. Apesar disso, as principais funcionalidades como cadastro de usuário, autenticação, criação de clubes e adição de livros na estante do clube garantem a usabilidade da aplicação.

De modo geral, este projeto fornece conhecimento e resultados de forma direta para a área de Engenharia de *Software*, mais especificamente nas subáreas de qualidade e testes. Expor uma abordagem prática, funcional e replicável a respeito da automação de testes, integração e entrega contínua, fazendo uso de ferramentas utilizadas em grande escala pelo mercado, torna esta pesquisa bastante pertinente em relação ao contexto acadêmico e profissional. A experiência obtida por meio das implementações estimula a compreensão sobre a importância da etapa de testes como um dos pilares da qualidade no ciclo de desenvolvimento de *software*.

## 7.1 Trabalhos Futuros

Para a continuidade deste trabalho e melhoria contínua da aplicação testada, propõe-se criar novos casos de teste para as funcionalidades já testadas, melhorar os casos de teste implementados, planejar e implementar casos de teste para as demais funcionalidades da aplicação. Além disso, a implementação de outros tipos de testes, como testes de performance e testes de carga e *stress*, traria um grande ganho para a qualidade da aplicação como um todo.

Em vista da não implementação dos testes unitários em virtude do tempo, a adição desse tipo essencial de teste ao projeto garantiria uma melhor cobertura a nível de código e certificaria que defeitos menores sejam descobertos antes mesmo dos testes funcionais e E2E.

Outras recomendações seriam conduzir estudos para escolher a melhor forma de *deploy* e a melhor implementação de *rollback*, que permitiria que a aplicação retorne automaticamente

para uma versão estável caso ocorram falhas críticas em produção que façam o servidor ficar *offline*.

Além do exposto, a implementação de uma nova versão da aplicação, considerando a correção dos defeitos encontrados durante os testes deste trabalho, é outra sugestão de trabalho futuro, pois a falta de validações básicas ao tratar informações faltantes ou dados não cadastrados resultam em falhas no servidor, reiniciando nas tentativas de encontrar um dado que não existe.

Um estudo para a separação dos ambientes em servidores dedicados para Produção, Testes e Desenvolvimento (hoje acessado de forma local) seria de grande importância para a confiabilidade e escalabilidade da aplicação, pois garantiria que os três ambientes seriam isolados fisicamente e não apenas logicamente como é feito atualmente.

Por fim, realizar um estudo aprofundado sobre a avaliação da qualidade da aplicação, considerando características como a compatibilidade, confiabilidade, segurança, manutenção, flexibilidade, entre outros aspectos, baseado-se na norma ISO/IEC 25010[1], para corroborar a qualidade da aplicação, agregar valor ao *software* desenvolvido e garantir um produto que alcance as expectativas e necessidades de seus usuários.

[1] Um modelo de qualidade com critérios para avaliar produtos de *software*. Disponível em: *https://www.iso25000.com/index.php/en/iso-25000-standards/iso-25010*

# Referências

ABES. *Estudo Mercado Brasileiro de Software – Panorama e Tendências*. 2023. Disponível em: <https://abes.com.br/dados-do-setor/>. Citado na página 15.

ALCâNTARA, E. M. de S. *Clubes de leitura*: proposta de um aplicativo para gerenciamento e compartilhamento de leituras literárias e incentivo à leitura. 141 f. Dissertação (Programa de Pós-Graduação em Propriedade Intelectual e Transferência de Tecnologia para Inovação) — Universidade Federal do Maranhão, São Luís, 2022. Disponível em: <https://tedebc.ufma.br/jspui/handle/tede/4526>. Acesso em: 24 de julho de 2025. Citado na página 34.

AMMANN, P.; OFFUTT, J. *Introduction to Software Testing*. [S.l.]: Cambridge University Press, 2008. Citado na página 44.

BACH, J. *Exploratory testing explained*. 2003. Disponível em: <https://www.satisfice.info/articles/et-article.pdf>. Acesso em: 18 de julho de 2025. Citado na página 78.

BERNARDO, P. C.; KON, F. A importância dos testes automatizados. *Engenharia de Software Magazine*, v. 1, n. 3, p. 54–57, 2008. Citado na página 80.

BOCCIA, A. S. Clubes de leitura: a construção de sentidos em situações de leitura colaborativa. *2236-5729*, volume 2, p. 97–113, 05 2012. Citado na página 34.

BOHM, V.; MARANGONI, M. Círculo de leitura: Ressignificando experiências. *Estudos Interdisciplinares sobre o Envelhecimento*, v. 16, 01 2011. Citado na página 34.

CHACON, S.; STRAUB, B. *Pro Git*. 2. ed. Apress, 2014. 440 p. ISBN 1484200772/978-1484200773. Disponível em: <https://git-scm.com/book/en/v2>. Citado 3 vezes nas páginas 55, 56 e 66.

DEJONGHE, D. *NGINX Cookbook — Advanced Recipes for High-Performance Load Balancing*. 3. ed. [S.l.]: O'Reilly Media, Inc., 2024. ISBN 978-1-098-15844-6. Citado 2 vezes nas páginas 54 e 55.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. et al. *Introdução ao Teste de Software*. [S.l.]: Elsevier Editora Ltda, 2007. ISBN 978-85-352-2634-8. Citado 7 vezes nas páginas 15, 19, 20, 22, 25, 27 e 44.

DOCKER. *Compose file reference*. [S.l.], 2025. Disponível em: <https://docs.docker.com/reference/compose-file/>. Acesso em: 9 de março de 2025. Citado 2 vezes nas páginas 52 e 53.

DOCKER. *Dockerfile reference*. [S.l.], 2025. Disponível em: <https://docs.docker.com/reference/dockerfile/>. Acesso em: 10 de fevereiro de 2025. Citado na página 52.

DOCKER. *Reference documentation*. [S.l.], 2025. Disponível em: <https://docs.docker.com/reference/>. Acesso em: 10 de fevereiro de 2025. Citado na página 50.

FELIZARDO, K. R. *Técnicas de VV&T - Validação, Verificação e Teste*. 2010. Disponível em: <http://linhadecodigo.com.br/artigo/492/tecnicas-de-vvampt-validacao-verificacao-e-teste.aspx>. Acesso em: 7 de julho de 2023. Citado na página 19.

FIELDING, R. T. *Architectural styles and the design of network-based software architectures*. [S.l.]: University of California, Irvine, 2000. Citado 3 vezes nas páginas 38, 39 e 40.

HARBOR. *Harbor 2.12 Documentation*. [S.l.], 2025. Disponível em: <https://goharbor.io/docs/2.12.0/>. Acesso em: 14 de fevereiro de 2025. Citado na página 49.

HENDRICKSON, E. *Explore it! Reduce Risk and Increase Confidence with Exploratory Testing*. Raleigh, NC: Pragmatic Programmers, 2013. (The pragmatic programmers). ISBN 1937785025/978-1937785024. Citado 2 vezes nas páginas 20 e 21.

HUMBLE, J.; FARLEY, D. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley, 2010. (A Martin Fowler Signature Book). ISBN 9780321601919. Disponível em: <https://books.google.com.br/books?id=9CAxmQEACAAJ>. Citado 5 vezes nas páginas 29, 30, 32, 33 e 47.

ISO/IEC. *ISO 27001:2022 A 8.31 — Information security, cybersecurity and privacy protection — Information security management systems — Requirements*. Third edition. [S.l.], 2022. Citado 2 vezes nas páginas 48 e 80.

ISTQB. *Certified Tester Advanced Level Test Analyst Syllabus - CTAL-TA*. [S.l.], 2022. Disponível em: <https://bstqb.qa/ctal-ta>. Acesso em: 13 de janeiro de 2025. Citado na página 44.

ISTQB. *Certified Tester Foundation Level Syllabus - CTFL*. [S.l.], 2023. Disponível em: <https://bstqb.qa/ctfl>. Acesso em: 10 de agosto de 2023. Citado 5 vezes nas páginas 22, 23, 24, 25 e 27.

ISTQB. *Certified Tester Advanced Level Test Automation Engineering Syllabus - CTAL-TAE*. [S.l.], 2024. Disponível em: <https://bstqb.qa/ctal-tae>. Acesso em: 13 de janeiro de 2025. Citado 2 vezes nas páginas 21 e 22.

ITKONEN, J.; UDD, R.; LASSENIUS, C.; LEHTONEN, T. Perceived benefits of adopting continuous delivery practices. In: *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. New York, NY, USA: Association for Computing Machinery, 2016. (ESEM '16). ISBN 9781450344272. Disponível em: <https://doi.org/10.1145/2961111.2962627>. Citado na página 31.

JENKINS. *Jenkins User Documentation*. [S.l.], 2025. Disponível em: <https://www.jenkins.io/doc/>. Acesso em: 14 de fevereiro de 2025. Citado 3 vezes nas páginas 58, 59 e 61.

JENKINS. *Pipeline*. [S.l.], 2025. Disponível em: <https://www.jenkins.io/doc/book/pipeline/>. Acesso em: 14 de fevereiro de 2025. Citado 2 vezes nas páginas 58 e 63.

JENKINS. *Pipeline Syntax*. [S.l.], 2025. Disponível em: <https://www.jenkins.io/doc/book/pipeline/syntax/>. Acesso em: 15 de março de 2025. Citado na página 63.

JENKINS. *Using a Jenkinsfile*. [S.l.], 2025. Disponível em: <https://www.jenkins.io/doc/book/pipeline/jenkinsfile/>. Acesso em: 14 de fevereiro de 2025. Citado na página 58.

LEMOS, E. da L. *Habilitando Aspectos de Colaboração em um Software para Clubes de Leitura*. 63 f. Monografia (Graduação em Engenharia da Computação) — CCET, Universidade Federal do Maranhão, São Luís, 2023. Citado 6 vezes nas páginas 16, 34, 35, 36, 37 e 38.

LUO, L. *Software Testing Techniques*. [S.l.], 2001. Disponível em: <https://www.cs.cmu.edu/~luluo/Courses/17939Report.pdf>. Acesso em: 21 de setembro de 2023. Citado na página 16.

MOLLOY, J. *Guide to Client-server Architecture or Model*. 2023. Disponível em: <https://www.liquidweb.com/blog/client-server-architecture/>. Acesso em: 10 de dezembro de 2024. Citado na página 39.

NGINX. *Nginx Docs - Web Server*. [S.l.], 2025. Disponível em: <https://docs.nginx.com/nginx/admin-guide/web-server/>. Acesso em: 10 de fevereiro de 2025. Citado na página 54.

OLIVEIRA, M. da S. *Aplicativo Web para o Controle de Clubes e Gerenciamento de Leituras*. 55 f. Monografia (Graduação em Engenharia da Computação) — CCET, Universidade Federal do Maranhão, São Luís, 2023. Citado 8 vezes nas páginas 16, 34, 35, 36, 37, 38, 39 e 40.

PATHANIA, N. *Learning Continuous Integration with Jenkins*. Packt Publishing, 2016. ISBN 9781785285035. Disponível em: <https://books.google.com.br/books?id=hgRwDQAAQBAJ>. Citado na página 62.

PRESSMAN, R. S. *Engenharia de Software: Uma abordagem Profissional*. 7. ed. [S.l.]: AMGH Editora Ltda., 2011. ISBN 978-0-0733-7597-7. Citado 9 vezes nas páginas 18, 19, 23, 24, 25, 26, 35, 36 e 74.

RAFI, D.; MOSES, K.; PETERSEN, K.; MäNTYLä, M. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In: . [S.l.: s.n.], 2012. p. 36–42. ISBN 978-1-4673-1821-1. Citado na página 16.

REDHAT. *O que é um pipeline de CI/CD?* 2022. Disponível em: <https://www.redhat.com/pt-br/topics/devops/what-cicd-pipeline>. Acesso em: 11 de dezembro de 2024. Citado na página 32.

REITZ, L. P.; CASTIñEIRA, M. I.; SCHUHMACHER, V. R. et al. Testes automatizados com ferramentas de software livre: um estudo de caso. 2013. Citado na página 22.

SOMMERVILE, I. *Test planning*. 2019. Disponível em: <https://software-engineering-book.com/web/test-planning/>. Citado 2 vezes nas páginas 26 e 76.

SOMMERVILLE, I. *Engenharia de software*. 10. ed. [S.l.]: Pearson Education do Brasil, 2019. ISBN 978-85-43O-2497-4. Citado 14 vezes nas páginas 15, 18, 19, 21, 23, 24, 26, 34, 35, 37, 74, 81, 82 e 85.

SWAGGER. *Swagger UI*. [S.l.], 2025. Disponível em: <https://swagger.io/tools/swagger-ui/>. Acesso em: 10 de março de 2025. Citado na página 74.

TINKHAM, A.; KANER, C. Exploring exploratory testing. 2003. Disponível em: <https://kaner.com/pdfs/ExploringExploratoryTesting.pdf>. Acesso em: 18 de julho de 2025. Citado na página 77.

VALENTE, M. T. *Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade*. [S.l.]: Editora: Independente, 2020. Citado 3 vezes nas páginas 16, 18 e 31.

# Apêndices

# APÊNDICE A – Relatório de defeitos

## A.1 Defeitos encontrados pelos testes funcionais

Tabela 15 – Detalhamento dos defeitos encontrados na API

| Funcionalidade | Caso de teste | Defeito | Descrição |
|---|---|---|---|
| Login | CT040 | Status code incorreto | Ao tentar fazer login com credenciais inválidas, o status code retornado é 400 quando deveria ser 401 Unauthorized |
| | CT041 | Status code incorreto | Ao tentar fazer login com credenciais inválidas, o status code retornado é 400 quando deveria ser 401 Unauthorized |
| | CT046 | Mensagem de erro incorreta | Ao enviar um body vazio, o endpoint deveria validar e retornar uma mensagem que indicasse isso, porém a mensagem retornada é "Senha obrigatória" |
| Buscar usuário | CT003 | Não valida o id do usuário | O endpoint não valida se o usuário com o id passado existe e tenta fazer operações com dados indefinidos, quebrando/reiniciando o servidor. |
| | CT004 | Não valida o id do usuário | O endpoint não valida se o id passado é nulo e mesmo assim consulta o banco e retorna um objeto vazio. |

Continua na próxima página

Tabela 15 – Detalhamento dos defeitos encontrados na API (Continuação)

| Funcionalidade | Caso de teste | Defeito | Descrição |
|---|---|---|---|
| Buscar usuário | CT005 | Servidor responde 502 | Algum tratamento interno de dados faz com que o servidor responda 502 e depois reinicia. |
| | CT006 | Servidor responde 502 | Algum tratamento interno de dados faz com que o servidor responda 502 e depois reinicia. |
| | CT007 | Servidor responde 502 | Algum tratamento interno de dados faz com que o servidor responda 502 e depois reinicia. |
| | CT008 | Sem tratamento para email nulo | O servidor não consegue tratar o email nulo e quebra/reinicia. |
| | CT009 | Sem tratamento para email vazio | O servidor não consegue tratar o email vazio e quebra/reinicia. |
| | CT010 | Não trata ID de usuário nulo | O servidor reinicia pois não encontra o id |
| | CT011 | Não trata ID de usuário inexistente | O servidor reinicia pois não encontra o id |
| | CT012 | Não trata ID de usuário vazio | O servidor responde com 200 porém deveria responder com 400. |
| | CT015 | Não trata o nick nulo | O servidor responde com 200 porém deveria responder com 400 pois null não deveria ser permitido |

Continua na próxima página

Tabela 15 – Detalhamento dos defeitos encontrados na API (Continuação)

| Funcionalidade | Caso de teste | Defeito | Descrição |
|---|---|---|---|
| Buscar usuário | CT016 | Não trata nick inválido | O servidor aceita nicks com caracteres especiais(inválidos) e responde com 200 quando deveria barrar para evitar dados inconsistentes |
| | CT017 | Não trata nick vazio | O servidor fazer validações sem necessariamente enviar um nick respondendo 200 quando deveria responder 400 |
| | CT018 | Não trata ID de usuário nulo | O endpoint aceita que passe um id nulo e o servidor responde 200. |
| | CT019 | Não trata ID de usuário inexistente | O endpoint aceita que passe um id inexistente e o servidor responde 200. |
| | CT020 | Não trata ID de usuário vazio | O endpoint aceita que passe um id vazio e o servidor responde 200, porém com o body vazio. |
| | CT024 | Não trata ID de usuário nulo | O endpoint aceita que passe um id nulo fazendo o servidor quebrar e responder 502. |
| | CT025 | Não trata ID de usuário inexistente | O servidor procura os clubes de um usuário com ID inexistente na base, não retorna erros, porém deveria retornar 404 já que o usuário não existe. |

Continua na próxima página

Tabela 15 – Detalhamento dos defeitos encontrados na API (Continuação)

| Funcionalidade | Caso de teste | Defeito | Descrição |
|---|---|---|---|
| Buscar usuário | CT026 | Não trada ID de usuário vazio | O servidor procura os clubes de um usuário com ID vazio na base, não retorna erros, porém deveria retornar 400 já que é uma requisição mal formada |
| | CT027 | Não trata ID de usuário nulo | O endpoint aceita que passe um id nulo fazendo o servidor quebrar e responder 502. |
| | CT028 | Não trata ID de usuário inexistente | O servidor procura os clubes de um usuário com ID inexistente na base, não retorna erros, porém deveria retornar 404 já que o usuário não existe. |
| | CT029 | Não trada ID de usuário vazio | O servidor procura os clubes de um usuário com ID vazio na base, não retorna erros, porém deveria retornar 400 já que é uma requisição mal formada |
| | CT030 | Não trata ID de usuário nulo | O endpoint aceita que passe um id nulo fazendo o servidor quebrar e responder 502. |
| | CT031 | Não trata ID de usuário inexistente | O servidor procura os clubes de um usuário com ID inexistente na base, não retorna erros, porém deveria retornar 404 já que o usuário não existe. |

Continua na próxima página

Tabela 15 – Detalhamento dos defeitos encontrados na API (Continuação)

| Funcionalidade | Caso de teste | Defeito | Descrição |
|---|---|---|---|
| Buscar usuário | CT032 | Não trada ID de usuário vazio | O servidor procura os clubes de um usuário com ID vazio na base, não retorna erros, porém deveria retornar 400 já que o usuário não existe. |
| | CT034 | Não trata ID do clube inexistente | O servidor procura os clubes de um usuário na base com ID do clube que não existe , não retorna erros, porém deveria retornar 404 já que o usuário não existe. |
| | CT035 | Não trata ID do usuário inexistente | O servidor procura os clubes de um usuário na base com ID de usuário que não existe , não retorna erros, porém deveria retornar 404 já que o usuário não existe |
| | CT036 | Não trata ID do clube nulo | O servidor tentar usar o valor nulo como ID do clube para buscar os clubes, retorna 200 e uma lista vazia, mas deveria retornar 400 |
| | CT037 | Não trata ID do usuário nulo | O servidor tenta usar o valor nulo como ID do usuário para buscar os clubes, retorna 200 e uma lista vazia, mas deveria retornar 400 |

Continua na próxima página

Tabela 15 – Detalhamento dos defeitos encontrados na API (Continuação)

| Funcionalidade | Caso de teste | Defeito | Descrição |
|---|---|---|---|
| Cadastrar usuário | CT048 | Status code incorreto | Apesar de não ser possível cadastrar um usuário com um e-mail já cadastrado, é vital que a API retorne os status corretos nas respostas |
| | CT049 | A API permite o cadastro de e-mail com formato inválido | É possível cadastrar um e-mail com formato inválido, por exemplo, sem @ ou sem domínio. |
| | CT050 | Sem validação para e-mail vazio | É possível cadastrar um usuário enviando um corpo contendo um e-mail vazio (email: “”). |
| | CT051 | Sem validação para e-mail sendo espaços em branco | É possível cadastrar um usuário enviando um corpo contendo um e-mail formado por espaços em branco (email: “ ”). |
| | CT052 | A API cadastra e-mail nulo | É possível criar um usuário com e-mail nulo. (email: null). |
| | CT053 | Não ocorre validação para senha vazia | Apesar de não ser possível cadastrar uma senha vazia(senha: “”), a validação não é a correta. O servidor deveria responder 400 pois a senha tem um formato inválido. |

Continua na próxima página

Tabela 15 – Detalhamento dos defeitos encontrados na API (Continuação)

| Funcionalidade | Caso de teste | Defeito | Descrição |
|---|---|---|---|
| Cadastrar usuário | CT054 | Não ocorre validação para senha formada por espaços em branco | Apesar de não ser possível cadastrar uma senha formada por espaços em branco(senha: “ ”), a validação não é a correta. O servidor deveria responder 400 pois a senha tem um formato inválido. |
| | CT055 | Não há validação para senha nula | O servidor não cria um usuário com senha nula, porém a resposta deveria conter o contexto de senha nula. |
| | CT056 | Não ocorre validação para senha vazia | A validação aqui ocorre apenas a comparação com a senha, porém a requisição deveria ser barrada ao enviar uma confirmação de senha vazia no corpo. |
| | CT057 | Não ocorre validação para senha formada por espaços em branco | A validação aqui ocorre apenas a comparação com a senha, porém a requisição deveria ser barrada ao enviar uma confirmação de senha formada por espaços em branco no corpo. |

Continua na próxima página

Tabela 15 – Detalhamento dos defeitos encontrados na API (Continuação)

| Funcionalidade | Caso de teste | Defeito | Descrição |
|---|---|---|---|
| Cadastrar usuário | CT058 | Não há validação para senha nula | A validação aqui ocorre apenas a comparação com a senha, porém a requisição deveria ser barrada ao enviar uma confirmação de senha nula no corpo. |
| | CT060 | Nome vazio não é tratado | Não há validação para verificar se o nome no corpo da requisição é vazio, cadastrando um usuário sem nome. |
| | CT061 | Nome composto por espaços em branco não é tratado | Não há validação para verificar se o nome no corpo da requisição é formado por espaços em branco, cadastrando um usuário nome “ ”, por exemplo. |
| | CT062 | Nome nulo não é tratado | Não há validação para verificar se o nome no corpo da requisição é nulo, cadastrando um usuário sem nome. |
| | CT063 | Não valida se os dados da requisição estão vazios | É possível enviar um corpo com as propriedades vazias e mesmo assim cadastrar um usuário. |
| | CT064 | Não valida se os dados da requisição são apenas espaços em branco | É possível criar um usuário com todos os dados sendo espaços em branco |

Continua na próxima página

Tabela 15 – Detalhamento dos defeitos encontrados na API (Continuação)

| Funcionalidade | Caso de teste | Defeito | Descrição |
|---|---|---|---|
| Cadastrar usuário | CT065 | Não valida os campos sendo todos nulo | Ao fazer uma requisição com todos os campos do corpo sendo nulo, o servidor quebra, retornando 502. |
| | CT066 | Validação incorreta | Apesar de retornar o status code correto(400), a mensagem não é clara, dando a impressão de que só realizou o cadastro pois já há um usuário com o e-mail contido no corpo da requisição (sendo que não há corpo). |
| | CT067 | Aceita requisições sem corpo | O endpoint aceita requisições sem corpo, retornando um HTML ao invés de tratar e retornar um status code (400) e uma mensagem adequada. |
| Atualizar usuário | CT068 | A propriedade *description* não foi mapeada | Ao enviar uma requisição de atualização de um usuário, e preencher o corpo da resposta com os dados necessários, o servidor quebra pois tenta atualizar um campo inexistente na tabela de usuários no banco de dados. |

Continua na próxima página

Tabela 15 – Detalhamento dos defeitos encontrados na API (Continuação)

| Funcionalidade | Caso de teste | Defeito | Descrição |
|---|---|---|---|
| Atualizar usuário | CT070 | Não há tratamento em caso de token inválido | O servidor faz a requisição e não consegue extrair as informações do token passado, pois é inválido, e isso causa a queda momentânea do servidor. |
| | CT071 | Não há tratamento em caso de token nulo | O servidor faz a requisição e não consegue extrair as informações do token passado, pois é inválido, e isso causa a queda momentanêa do servidor. |
| | CT075 | Status code incorreto | Apesar da requisição ser barrada, o status code da resposta não deveria ser 401, pois o ID nulo deveria ser tratado como 400. |
| | CT076 | Status code incorreto | Apesar da requisição ser barrada, o status code da resposta não deveria ser 401, pois o ID com espaços em branco deveria ser tratado como 400. |
| | CT077 | Status code incorreto | Apesar da requisição ser barrada, o status code da resposta não deveria ser 404, pois o ID vazio deveria ser tratado como 400. |
| | CT080 | Não há tratamento para nick nulo | O endpoint aceita o nick nulo, atualizando o usuário com dados inconsistentes. |

Continua na próxima página

Tabela 15 – Detalhamento dos defeitos encontrados na API (Continuação)

| Funcionalidade | Caso de teste | Defeito | Descrição |
|---|---|---|---|
| Atualizar usuário | CT081 | Não há tratamento para nick com espaços em branco | O servidor é derrubado momentaneamente pois tenta fazer operações com valor em branco. |
| | CT082 | Não há tratamento para nick vazio | O endpoint aceita o nick vazio, atualizando o usuário com dados inconsistentes. |

*Fonte*: Elaborado pelo autor.

## A.2 Defeitos encontrados pelos testes E2E

Tabela 16 – Detalhamento dos defeitos encontrados na interface

| Funcionalidade | Caso de teste | Defeito | Descrição |
|---|---|---|---|
| Login | CT009 | Mensagem de validação incorreta | Quando tenta fazer login sem informar email e senha, a mensagem de validação deveria ser algo como “Email e senha são obrigatórios” ou “Informe o email e a senha” |
| | CT010 | Mensagem de validação incorreta | Ao informar um e-mail incorreto, a mensagem exibida é “Server Error” quando deveria ser algo como “E-mail inválido” |
| | CT012 | Mensagem de validação incorreta | Ao informar e-mail e senha incorretos, a mensagem exibida é “Server Error” quando deveria ser algo como “E-mail ou senha inválidos” ou “Credenciais inválidas” |

Continua na próxima página

Tabela 16 – Detalhamento dos defeitos encontrados na interface (Continuação)

| Funcionalidade | Caso de teste | Defeito | Descrição |
|---|---|---|---|
| Login | CT018 | A aplicação não limita o campo de e-mail | É possível fazer login com um e-mail muito longo, quando deveria barrar a tentativa de login com email nessa condição |
| Cadastrar usuário | CT029 | Não há validação para nome inválido | É possível criar um usuário com o nome contendo caracteres especiais |
| | CT030 | Senha muito curta | É possível criar um usuário com uma senha muito curta. Para segurança, seria importante implementar uma senha com no mínimo 8 caracteres e que contenham algumas especificidades como letras maiúsculas, caracteres especiais permitidos, etc. |
| | CT032 | Aceita script injection | A aplicação permite o cadastro de nomes de usuários com script injection. É uma falha grave, pois um script malicioso pode ser executado indevidamente. |
| | CT034 | Não valida o tamanho do nome | Apesar de ser possível criar um usuário com o nome muito grande, a aplicação trunca o nome para exibição. É necessária uma limitação para evitar inconsistências. |

Continua na próxima página

Tabela 16 – Detalhamento dos defeitos encontrados na interface (Continuação)

| Funcionalidade | Caso de teste | Defeito | Descrição |
|---|---|---|---|
| Cadastrar usuário | CT035 | Não valida o conteúdo da senha | É possível criar uma senha contendo emojis. Para evitar dados inconsistentes no banco, e a quebra do servidor ao buscar tais dados, é necessário um tratamento da senha quanto a emojis. |
| Criar clube | CT044 | Não valida o formato da imagem | O endpoint aceita qualquer arquivo fazendo o servidor responder 502 e ser reiniciado. |
| | CT045 | Não valida o formato do arquivo | O endpoint aceita qualquer arquivo fazendo o servidor responder 502 e ser reiniciado. |
| | CT047 | Não valida o tamanho da imagem | É possível enviar uma imagem maior que 1MB e a aplicação aceita, mesmo sendo um requisito do sistema. |
| | CT049 | Aceita script injection | A aplicação permite o cadastro de nomes de clubes com script injection. É uma falha grave, pois um script malicioso pode ser executado indevidamente. |
| | CT050 | Aceita SQL injection | A aplicação permite o cadastro de nomes de clubes com SQL injection. É uma falha grave, pois um script malicioso pode ser executado indevidamente. |

Continua na próxima página

Tabela 16 – Detalhamento dos defeitos encontrados na interface (Continuação)

| Funcionalidade | Caso de teste | Defeito | Descrição |
|---|---|---|---|
| Criar clube | CT051 | Aceita script injection | A aplicação permite o cadastro de descrição de clubes com script injection. É uma falha grave, pois um script malicioso pode ser executado indevidamente. |
| | CT052 | Aceita SQL injection | A aplicação permite o cadastro de descrição de clubes com SQL injection. É uma falha grave, pois um script malicioso pode ser executado indevidamente. |
| Adicionar livro à estante do clube | CT053 | Não é possível cadastrar um livro | Aplicação não cadastra um novo livro, retornando status code 500 |
| | CT054 | Não é possível cadastrar um livro | Aplicação não cadastra um novo livro, retornando status code 500 |
| | CT055 | Não é possível cadastrar um livro | Aplicação não cadastra um novo livro, retornando status code 500 |
| | CT057 | Não valida a obrigatoriedade de informar o autor | A aplicação tenta cadastrar um livro sem que seja informado um autor, apesar da indicação de obrigatoriedade de preenchimento do campo. |
| | CT058 | Não valida a obrigatoriedade de informar o autor | A aplicação tenta cadastrar um livro sem que seja informado um autor, apesar da indicação de obrigatoriedade de preenchimento do campo. |

Continua na próxima página

Tabela 16 – Detalhamento dos defeitos encontrados na interface (Continuação)

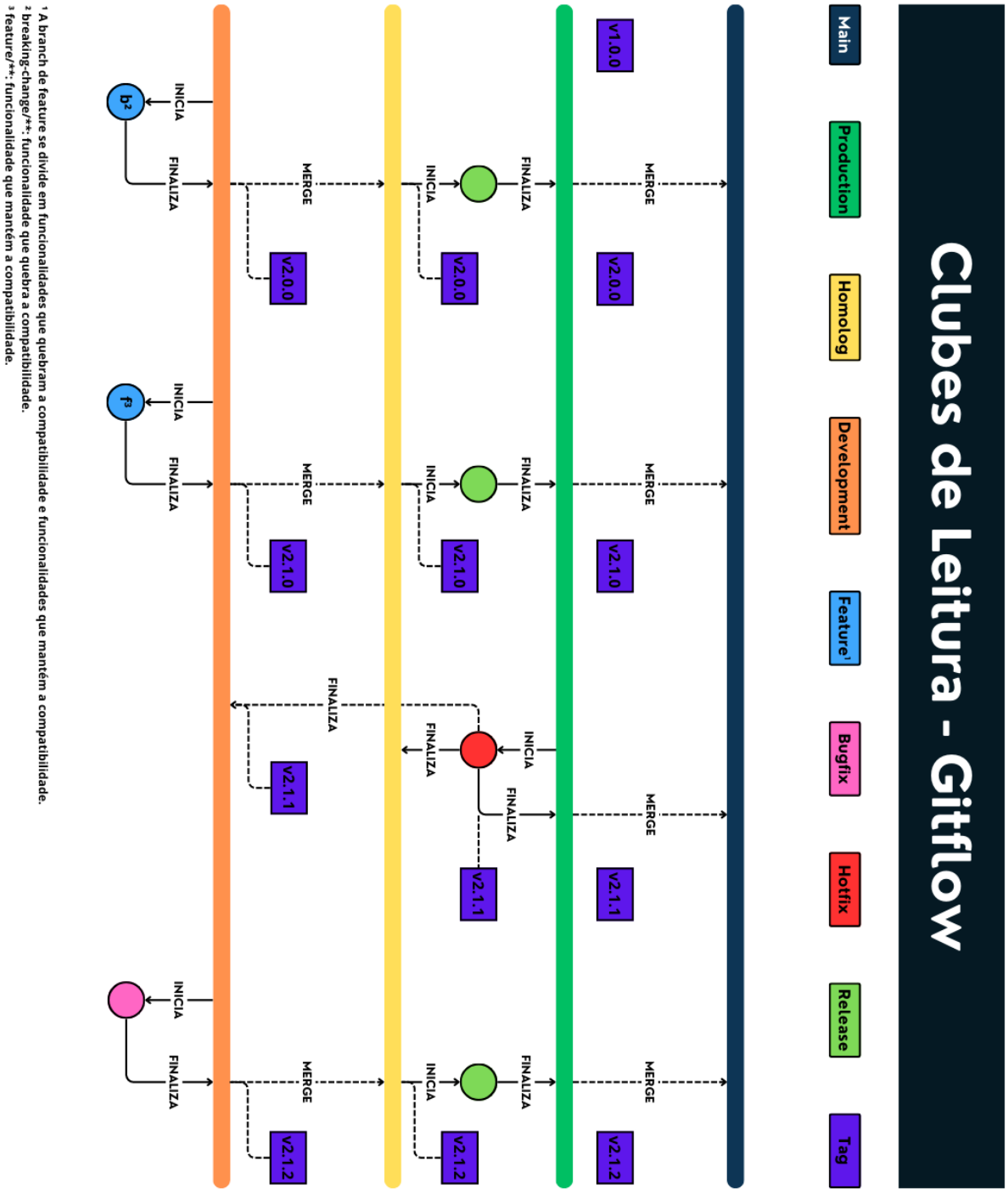
| Funcionalidade | Caso de teste | Defeito | Descrição |
|---|---|---|---|
| Adicionar livro à estante do clube | CT059 | Não é possível cadastrar um livro | Aplicação não cadastra um novo livro, retornando status code 500 |
| | CT060 | Não é possível cadastrar um livro | Aplicação não cadastra um novo livro, retornando status code 500 |
| | CT061 | Aceita script injection | A aplicação tenta cadastrar um livro com nome com script injection, só não consegue pois o cadastro de livros não está funcional. |
| | CT062 | Aceita SQL injection | A aplicação tenta cadastrar um livro com descrição com SQL injection, só não consegue pois o cadastro de livros não está funcional. |
| | CT065 | Não foi possível adicionar um livro | O Botão de adicionar livros não aparece para usuários com perfil mediador |

*Fonte*: Elaborado pelo autor.

# Anexos

# ANEXO A – Gitflow do Clubes de Leitura

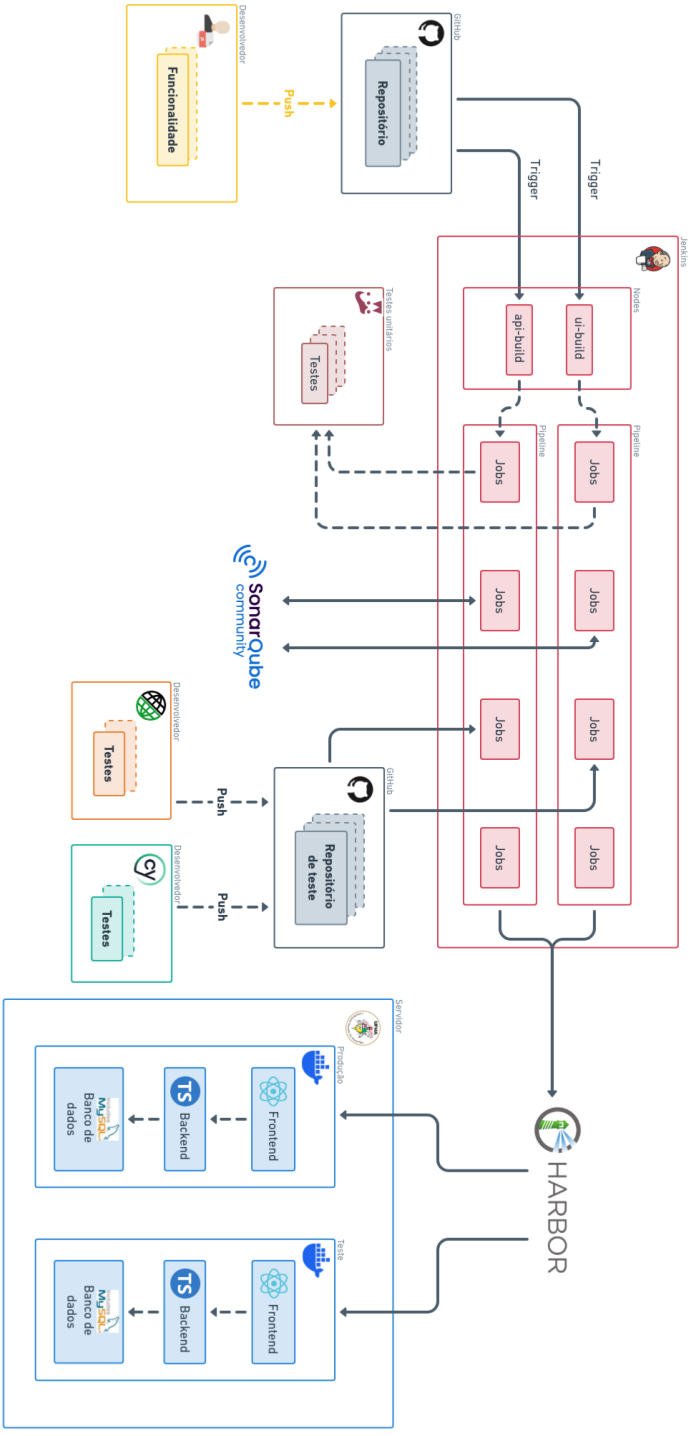
Figura 55 – Fluxo de trabalho do controle de versão




Fonte: Elaborado pelo autor

# ANEXO B – Arquitetura dos Testes Automatizados

Figura 56 – Projeto arquitetural dos testes automatizados




Fonte: Elaborado pelo autor

# ANEXO C – Plano de Testes
## Sumário

# 1 Introdução

## 1.1 Contextualização do Projeto

O objeto deste plano de testes trata-se de um aplicativo *web* desenvolvido por alunos da *Universidade Federal do Maranhão (UFMA)* que tem como foco a criação de clubes de leitura para incentivar a prática de leitura (ALEXANDRE, 2025). Além disso, tem-se a possibilidade de usar a aplicação como fonte de captação de dados a fim de aprimorar ou implantar políticas públicas (OLIVEIRA, 2022). O *Clubes de Leitura*, como o aplicativo é nomeado, tende a ser uma ferramenta tecnológica para gerenciar clubes utilizando características de sistemas colaborativos (LEMOS, 2023).

A aplicação permite a criação de usuários, autenticação baseada em funções dentro do clube, criação de clubes, criação de estantes de livros, criação de votações, criação de reuniões. Permite ainda comentar em um fórum, adicionar resenhas de livros lidos e permite expulsar um usuário de um clube, dentre outras ações. Algumas ações são protegidas por autenticação e apenas usuários específicos podem executá-las (OLIVEIRA, 2022; LEMOS, 2023).

## 1.2 Objetivo do plano

- Identificar as funcionalidades do *software* que serão testadas
- Identificar os tipos de testes que serão elaborados
- Descrever a estratégia de teste para as funcionalidades
- Identificar os recursos necessários para a execução dos testes
- Identificar os riscos envolvidos durante a execução dos testes
- Estabelecer o cronograma para eficiência do plano
- Listar os cenários que serão testados

## 1.3 Recursos Humanos

A execução dos testes será constituída por uma equipe de desenvolvimento com 4 pessoas, dentre elas um *Product Owner (PO)*, um desenvolvedor, um analista de testes (QA) e o professor orientador, que trabalha como um *Scrum Master (SM)*. Além da equipe principal, há 2

pessoas para suporte junto ao *Núcleo de Tecnologia da Informação (NTI)* da universidade. A Tabela 1 lista todas as pessoas envolvidas e suas respectivas funções.

Tabela 1 – Equipe de Desenvolvimento

| Nome | Função |
|---|---|
| Claudio Henrique | Analista de Testes |
| Davi Viana | Scrum Master |
| Erlane de Sousa | Product Owner |
| Mateus Oliveira | Desenvolvedor |
| Benedito Mendes | Suporte NTI |
| Túlio Ferreira | Suporte NTI |

Fonte: Elaborado pelo autor.

## 1.4 Escopo dos Testes

O objetivo do plano de testes é validar a qualidade da aplicação Clubes de Leitura, garantindo que suas funcionalidades atendem aos requisitos e que a aplicação se comporta de maneira correta em diferentes ambientes. Para isso, serão implementados testes manuais e automatizados cobrindo diferentes níveis da aplicação.

1. **Backend**:
    - Testes unitários: validação das funções e regras de negócio.
    - Testes funcionais: verificação da resposta da API em cenários variados.
    - Testes de integração: assegurar a correta comunicação entre os módulos.
2. **Frontend**
    - Testes unitários de componentes: garante o comportamento esperado.
    - Testes end-to-end (E2E): validação da interação do usuário com a aplicação.
3. **Pipeline CI/CD**
    - Testes automatizados nas pipelines: execução de testes regressivos.
    - Geração de relatórios de testes para análise e aprovação manual do deploy.

Com o intuito de manter o foco nos testes automatizados e implementação na pipeline, o plano de testes não cobrirá os seguintes aspectos:

- Testes de performance e carga.

- Testes avançados de segurança (como análise de vulnerabilidades).
- Testes manuais extensivos (pois o foco é a automação).

# 2 Ferramentas e Ambiente de Testes

Para a execução do plano de testes, é necessário definir as ferramentas que serão utilizadas para executar cada tipo de teste, dos exploratórios e manuais aos regressivos e automatizados. Além disso, é necessário caracterizar o ambiente de teste e como ele está implantado.

## 2.1 Ferramentas

Para assegurar a qualidade da aplicação, algumas ferramentas foram utilizadas durante o processo de teste da aplicação. As ferramentas auxiliam na automação dos testes, no gerenciamento de pipelines de CI/CD, na documentação da API e no monitoramento dos ambientes.

As ferramentas foram escolhidas de acordo com a compatibilidade com a *stack* tecnológica do projeto, suporte à integração e entrega contínua, fácil configuração e manutenção. A Tabela 2 apresenta as principais ferramentas a serem utilizadas, juntamente com suas respectivas versões e funções.

Tabela 2 – Ferramentas

| Ferramenta | Versão | Função |
|---|---|---|
| Docker | 28.0.1 | Gerenciamento de contêineres |
| Jenkins | 2.492.2 | Gerenciamento de pipelines CI/CD |
| RestAssured | 4.4.0 | Automatizar testes funcionais da API[1] |
| Cypress | 14.5.0 | Automatizar testes E2E |
| Jest | 29.7.0 | Automatizar testes unitários da *UI*[2] |
| | | Automatizar testes unitários da *API* |
| Allure Reports | 2.34.0 | Geração de relatórios e captura de métricas |
| Postman | 11.15.0 | Testes exploratórios e manuais da API |

Fonte: Elaborado pelo autor.

## 2.2 Ambientes de Teste

A execução dos testes ocorrerá em diferentes ambientes para certificar que a aplicação funcione corretamente antes de ser liberada para os usuários finais:

[1] Interface de Programação de Aplicações (do inglês Application Programming Interface)
[2] Interface do Usuário (do inglês User Interface)

- Desenvolvimento: Ambiente local utilizado para a implementação dos testes, além de testes preliminares e validação inicial do código.
- Testes/Homologação: Ambiente de testes automatizados e de aceitação e integração contínua.
- Produção: Para monitoramento e validação pós- *deploy*.

O ambiente de testes(ou ambiente de homologação) pode ser acessado através das URLs *clubedeleitura-hml.ufma.br* para a interface de usuário e *clubedeleitura-hml.ufma.br/api* para a *API*, garantindo que seja um ambiente isolado em relação aos ambientes de produção e desenvolvimento, como recomenda a ISO/IEC 27001:2022.

O ambiente local(ou ambiente de desenvolvimento) do analista de teste será composto por um computador, navegador de *internet*, Ambiente de Desenvolvimento Integrado (do inglês *Integrated Development Environment (IDE)*) para edição de código e implementação dos testes automatizados. A PO usará um computador e um navegador de *internet* para executar testes exploratórios e de aceitação. A Tabela 3 agrupa as informações sobre os ambientes utilizados por componentes da equipe responsáveis pela execução dos testes.

Tabela 3 – Ambientes

| **Analista de Testes** | | | **Product Owner** | | |
|---|---|---|---|---|---|
| **Nome** | **Versão** | **Descrição** | **Nome** | **Versão** | **Descrição** |
| Linux Mint | 22 | Sistema Operacional | Windows | 11 | Sistema Operacional |
| Mozila Firefox | 136.0.1 | Navegador de Internet | Google Chrome | - | Navegador de internet |
| WebStorm | 2024.3.5 | IDE para testes do frontend | Android | - | Sistema Operacional |
| Intellij IDEA | 2024.2.3 (*Community Edition*) | IDE para testes do backend | Chrome Mobile | - | Navegador de internet |

Fonte: Elaborado pelo autor

# 3 Estratégia de Teste

As estratégias de testes foram definidas após a análise realizada pelo analista de testes em conjunto com a priorização dos fluxos elencados pela *product owner*. As estratégias serão explicadas na Tabela 4 a seguir:

Tabela 4 – Estratégias de Teste

| Objetivo | Técnicas | Critérios de conclusão |
|---|---|---|
| **Testes Unitários** | | |
| Garantir que as funções e componentes executem suas responsabilidades como o esperado | Caixa branca e automatizado | Todas as funções e componentes realizam sua atividade com sucesso e sem corrupção ou perda de dados |
| **Testes de Persistência** | | |
| Assegurar que os dados sejam persistidos corretamente e que os acessos ao banco ocorram dentro do esperado | Caixa preta e automatizado | Todos os métodos de acesso ao banco de dados funcionam de acordo com o esperado e sem corromper os dados |
| **Testes Funcionais** | | |
| Assegurar a boa navegação do software, a entrada, processamento e recuperação dos dados | Caixa preta, automatizado e manual | Todos os testes planejados foram executados e todos as falhas encontradas foram tratadas |
| **Testes E2E** | | |
| Testar os fluxos da aplicação na interface gráfica do software para garantir que os componentes funcionem como o conforme os requisitos | Caixa preta, automatizado e manual | Verificação com sucesso da navegação e que a interface permaneça consistente e dentro de um padrão aceitável |
| **Testes de Segurança e Controle de Acesso** | | |
| Verificar se o usuário tem acesso apenas às funções ou dados para os quais seu tipo de usuário tenha permissão | Caixa preta, automatizado e manual | Os tipos de usuário tem acesso apenas as funções e dados que são permitidos |

Fonte: Elaborado pelo autor.

# 4 Requisitos de Testes

1. **Teste de Unidade**

    a) Os métodos de classe devem ser testado individualmente para validar seu comportamento esperado.

2. **Teste de Persistência**

    a) O sistema deve garantir que o acesso ao banco de dados funcione corretamente para ações que exijam leitura e escrita de dados.

    b) Os dados armazenados no banco devem permanecer consistentes após operações de inserção, atualização e exclusão.

3. **Teste de Funcionalidade**

    a) As funcionalidades devem ser verificadas como um todo para garantir que a correta integração entre métodos.

4. **Teste E2E**

    a) A interface do usuário deve ser intuitiva e proporcionar um fluxo lógico de utilização.

5. **Teste de Segurança e Controle de Acesso**

    a) O login deve ser protegido, exigindo autenticação via e-mail e senha.

    b) O sistema deve validar o *token* de acesso do usuário.

    c) O controle de perfis de usuário deve restringir permissões adequadamente.

# 5 Riscos

É importante ressaltar que o desenvolvimento de um *software* está suscetível a riscos que podem afetar a qualidade do produto ou a eficiência dos testes. Então, é necessário listar tais riscos:

1. **Riscos Técnicos**
    a) Instabilidade do servidor, impactando a execução dos testes e da pipeline CI/CD
    b) Configurações erradas podem acarretar em problemas com os contêineres Docker afetando a execução das automações
    c) Alguns testes podem enfrentar incompatibilidades entre os ambientes
    d) O projeto dos testes pode ser executado de forma equivocada, influenciando em resultados incorretos
    e) Estimativa de tempo equivocada pode afetar na execução dos testes e funcionalidades críticas podem não serem testadas resultando em falhas em produção
2. **Riscos Operacionais**
    a) Testes extremamente longos podem aumentar o tempo de execução atrasando o fluxo da pipeline
    b) Partes da aplicação podem não ter a cobertura de testes adequada, ocasionando defeitos em produção.
    c) Dados de testes incorretos podem interferir na repetição dos testes
3. **Riscos Humanos**
    a) O descaso da aprovação manual do *deploy* pode levar ao lançamento de versões defeituosa
    b) A qualidade da aplicação pode ser afetada caso haja falhas durante a análise dos relatórios de testes
    c) Os testes podem não ser aplicados em sua totalidade por falha na estimativa de tempo, o que pode comprometer a cobertura dos testes.
4. **Riscos de Segurança**
    a) Dados sensíveis em *logs* ou mensagens de erro podem vazar informações confidenciais durante os testes.

b) A inexistência de testes de segurança pode ajudar na não identificação de vulnerabilidades.

5. **Riscos Relacionados ao Versionamento**

a) Versões incorretas podem ser lançadas em caso de erro durante o versionamento das imagens Docker.

b) Alterações na API podem causar incompatibilidade na interface gráfica sem que os testes possam detectar a falha.

# 6 Entregáveis

A Tabela 5 lista todos os artefatos entregáveis ao fim do desenvolvimento e testes da aplicação:

Tabela 5 – Estratégias de Teste

| **Software** | | **Documento** | |
|---|---|---|---|
| **Nome** | **Descrição** | **Nome** | **Descrição** |
| Ambiente de testes | Implementação do ambiente de testes/homologação | Plano de Teste | Documento do planejamento de implementação dos testes |
| Pipeline CI/CD | Implementação da pipeline de entrega e integração contínua | Relatório de defeitos | Documento com a lista de defeitos encontrados |
| Projeto dos testes funcionais | Repositório com o código fonte dos testes automatizados da API | Relatório de teste manuais | Documento com evidências dos testes manuais |
| Projeto dos testes e2e | Repositório com o código fonte dos testes automatizados da interface gráfica | Cenários de testes | Documento com o cenários de testes planejados |

Fonte: Elaborado pelo autor.

# 7 Cronograma

A Tabela 6 apresenta o cronograma de atividades a serem executadas durante o desenvolvimento e execução dos testes deste plano.

Tabela 6 – Cronograma de atividades

| Atividade | Data de Início | Data de Término | Duração |
|---|---|---|---|
| Planejamento | 17/03/2025 | 21/03/2025 | 5 dias |
| Projetar testes | 24/03/2025 | 28/03/2025 | 5 dias |
| Implementação dos testes | 31/03/2025 | 25/04/2025 | 20 dias |
| Execução dos testes | 28/04/2025 | 09/05/2025 | 10 dias |
| Reteste e verificação | 12/05/2025 | 16/05/2025 | 5 dias |
| Análise do relatório de testes | 19/05/2025 | 20/05/2025 | 2 dias |
| Entrega | 23/05/2025 | 23/05/2025 | 1 dia |

Fonte: Elaborado pelo autor

# Referências

ALEXANDRE, C. H. V. *Implementando Testes dd Software em uma Aplicação Web para Clubes de Leitura*. Tese (TCC(Graduação em Engenharia da Computação)) — Universidade Federal do Maranhão, UFMA, 2025. Citado na página 2.

LEMOS, E. da L. *Habilitando Aspectos de Colaboração em um Software para Clubes de Leitura*. 63 p. Tese (TCC(Graduação em Engenharia da Computação)) — Universidade Federal do Maranhão, UFMA, 2023. Citado na página 2.

OLIVEIRA, M. da S. *Aplicativo Web para o Controle de Clubes e Gerenciamento de Leituras*. 55 p. Tese (TCC(Graduação em Engenharia da Computação)) — Universidade Federal do Maranhão, UFMA, 2022. Citado na página 2.

# Apêndices

# APÊNDICE A – Casos de Teste

## A.1 API

| | |
|---|---|
| **ID** | CT001 |
| **Título** | Deve retornar todos os usuários com sucesso |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de buscar usuários |
| **Endpoint** | GET /user |
| **Pré-condição** | Ter usuários cadastros no sistema |
| **Passos** | DADO alguns usuários cadastrados no sistema<br>QUANDO uma requisição é realizada no endpoint *GET /user*<br>ENTÃO a API responde com status code 200<br>E o corpo da resposta tem uma lista de usuários |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve conter a lista de usuários cadastrados |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT002 |
| **Título** | Deve retornar um usuário com ID correspondente |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de buscar usuários |
| **Endpoint** | GET /user/{userId} |
| **Pré-condição** | Ter o ID de um usuário cadastrado |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO uma requisição é realizada no endpoint *GET /user/{userId}* informando o ID do usuário<br>ENTÃO a API responde com status code 200<br>E o corpo da resposta contém as informações do usuário |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve com a lista de usuários cadastrados<br>3. O ID da resposta deve corresponder ao ID do usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT003 |
| **Título** | Não deve retornar um usuário com o ID inexistente |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de buscar usuários |
| **Endpoint** | GET /user/{userId} |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *GET /user/{userId}* informando um ID inexistente<br>ENTÃO a API responde com status code 404<br>E deve retornar a mensagem “Usuário não encontrado” |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve ter uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT004 |
| **Título** | Não deve retornar um usuário com o ID nulo |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de buscar usuários |
| **Endpoint** | GET /user/{userId} |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *GET /user/null*<br>ENTÃO a API responde com status code 400<br>E deve retornar a mensagem “O ID do usuário inválido” |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve ter uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT005 |
| **Título** | Deve validar um e-mail cadastrado com sucesso |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de validação de email |
| **Endpoint** | GET /user/{userId}/email |
| **Pré-condição** | Email cadastrado no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO uma requisição é realizada no endpoint *GET /user/{userId}/email?email={email do usuário}*<br>ENTÃO a API responde com status code 409<br>E deve retornar a mensagem "Já em uso" |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve ter uma mensagem de validação |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT006 |
| **Título** | Deve validar um e-mail não cadastrado com sucesso |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de validação de email |
| **Endpoint** | GET /user/{userId}/email |
| **Pré-condição** | Email não cadastrado no sistema |
| **Passos** | DADO um usuário sem cadastro no sistema<br>QUANDO uma requisição é realizada no endpoint *GET /user/{userId}/email?email={email do usuário}*<br>ENTÃO a API responde com status code 200<br>E deve retornar a mensagem "válido" |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve ter uma mensagem de validação |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT007 |
| **Título** | Não deve validar um e-mail nulo |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de validação de email |
| **Endpoint** | GET /user/{userId}/email |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *GET /user/{userId}/email?email=null*<br>ENTÃO a API responde com status code 400<br>E deve retornar a mensagem "E-mail com formato inválido" |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve ter uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT008 |
| **Título** | Não deve validar um e-mail inválido |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de validação de email |
| **Endpoint** | GET /user/{userId}/email |
| **Pré-condição** | - |
| **Passos** | DADO um e-mail com formato inválido<br>QUANDO uma requisição é realizada no endpoint *GET /user/{userId}/email?email={email inválido}*<br>ENTÃO a API responde com status code 400<br>E deve retornar a mensagem "E-mail com formato inválido" |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve ter uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT009 |
| **Título** | Não deve validar um e-mail vazio |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de validação de email |
| **Endpoint** | GET /user/{userId}/email |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *GET /user/{userId}/email?email=*<br>ENTÃO a API responde com status code 400<br>E deve retornar a mensagem “E-mail com formato inválido” |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve ter uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT010 |
| **Título** | Não deve validar um e-mail com ID nulo |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de validação de email |
| **Endpoint** | GET /user/{userId}/email |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *GET /user/null/email?email={email do usuário}*<br>ENTÃO a API responde com status code 400<br>E deve retornar a mensagem “ID do usuário inválido” |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve ter uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT011 |
| **Título** | Não deve validar um e-mail com ID inexistente |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de validação de email |
| **Endpoint** | GET /user/{userId}/email |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *GET /user/{userId}/email?email={email do usuário}* com ID inexistente<br>ENTÃO a API responde com status code 400<br>E deve retornar a mensagem “Usuário não encontrado” |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve ter uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT012 |
| **Título** | Não deve validar um e-mail com ID vazio |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de validação de email |
| **Endpoint** | GET /user/{userId}/email |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *GET /user//email?email={email do usuário}*<br>ENTÃO a API responde com status code 400<br>E deve retornar a mensagem “ID do usuário inválido” |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve ter uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT013 |
| **Título** | Deve validar um nick cadastrado com sucesso |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de validação de nick |
| **Endpoint** | GET /user/{userId}/nick |
| **Pré-condição** | Nick não ser cadastrado por nenhum usuário |
| **Passos** | DADO um usuário com nick cadastrado<br>QUANDO uma requisição é realizada no endpoint *GET /user/{userId}/nick?nick={nick do usuário}*<br>ENTÃO a API responde com status code 200<br>E deve retornar a mensagem “Já em uso” |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve ter uma mensagem de validação |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT014 |
| **Título** | Deve validar um nick não cadastrado com com sucesso |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de validação de nick |
| **Endpoint** | GET /user/{userId}/nick |
| **Pré-condição** | Nick já ser cadastrado por algum usuário |
| **Passos** | DADO uma requisição realizada no endpoint *GET /user/{userId}/nick?nick={nick do usuário}*<br>ENTÃO a API responde com status code 200<br>E deve retornar a mensagem “válido” |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve ter uma mensagem de validação |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT015 |
| **Título** | Não deve validar um nick nulo |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de validação de nick |
| **Endpoint** | GET /user/{userId}/nick |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *GET /user/{userId}/nick?nick=null*<br>ENTÃO a API responde com status code 400<br>E deve retornar a mensagem “Nick inválido” |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve ter uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT016 |
| **Título** | Não deve validar um nick inválido |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de validação de nick |
| **Endpoint** | GET /user/{userId}/nick |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *GET /user/{userId}/nick?nick={nickl com caracteres especiais}*<br>ENTÃO a API responde com status code 400<br>E deve retornar a mensagem “Nick inválido” |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve ter uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT017 |
| **Título** | Não deve validar um nick vazio |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de validação de nick |
| **Endpoint** | GET /user/{userId}/nick |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *GET /user/{userId}/nick?nick=*<br>ENTÃO a API responde com status code 400<br>E deve retornar a mensagem “Nick inválido” |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve ter uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT018 |
| **Título** | Não deve validar um nick com ID nulo |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de validação de nick |
| **Endpoint** | GET /user/{userId}/nick |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *GET /user/null/nick?nick={nick do usuário}*<br>ENTÃO a API responde com status code 400<br>E deve retornar a mensagem “ID do usuário inválido” |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve ter uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT019 |
| **Título** | Não deve validar um nick com ID inexistente |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de validação de nick |
| **Endpoint** | GET /user/{userId}/nick |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *GET /user/{userId}/nick?nick={nick do usuário}* com ID inexistente<br>ENTÃO a API responde com status code 400<br>E deve retornar a mensagem “Usuário não encontrado” |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve ter uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT020 |
| **Título** | Não deve validar um nick com ID vazio |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de validação de nick |
| **Endpoint** | GET /user/{userId}/nick |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *GET /user//nick?nick={nick do usuário}* com ID inexistente<br>ENTÃO a API responde com status code 400<br>E deve retornar a mensagem “ID do usuário inválido” |
| **Resultado esperado** | 1. A API deve responder com status correto<br>2. O corpo da resposta deve ter uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT021 |
| **Título** | Deve retornar os clubes que o usuário é membro com sucesso |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de busca de clubes do usuário |
| **Endpoint** | GET /user/{userId}/clubs |
| **Pré-condição** | Usuário ter cadastro no sistema e participar de pelo menos um clube |
| **Passos** | DADO um usuário que participa de um clube<br>QUANDO uma requisição é realizada no endpoint *GET /user/{userId}/clubs*?action=member<br>ENTÃO a API responde com o status code 200<br>E deve retornar a lista de clubes do usuário |
| **Resultado esperado** | 1. A API deve retornar o status code correto<br>2. A API deve retornar as informações dos clubes do usuário<br>3. As informações dos clubes devem conter o ID do usuário correspondente |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT022 |
| **Título** | Deve retornar os clubes que o usuário é administrador com sucesso |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de busca de clubes do usuário |
| **Endpoint** | GET /user/{userId}/clubs |
| **Pré-condição** | Usuário ter cadastro no sistema e ser o criado de pelo menos um clube |
| **Passos** | DADO um usuário que criou um clube<br>QUANDO uma requisição é realizada no endpoint *GET /user/{userId}/clubs?action=admin*<br>ENTÃO a API responde com o status code 200<br>E deve retornar a lista de clubes do usuário |
| **Resultado esperado** | 1. A API deve retornar o status code correto<br>2. A API deve retornar as informações dos clubes do usuário<br>3. As informações dos clubes devem conter o ID do usuário correspondente |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT023 |
| **Título** | Deve retornar os clubes que o usuário é mediador com sucesso |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de busca de clubes do usuário |
| **Endpoint** | GET /user/{userId}/clubs |
| **Pré-condição** | Usuário ter cadastro no sistema e ser mediador de pelo menos um clube |
| **Passos** | DADO um usuário que é mediador de um clube<br>QUANDO uma requisição é realizada no endpoint *GET /user/{userId}/clubs?action=mediator*<br>ENTÃO a API responde com o status code 200<br>E deve retornar a lista de clubes do usuário |
| **Resultado esperado** | 1. A API deve retornar o status code correto<br>2. A API deve retornar as informações dos clubes do usuário<br>3. As informações dos clubes devem conter o ID do usuário correspondente |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT024 |
| **Título** | Não deve retornar os clubes que o usuário é membro com ID nulo |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de busca de clubes do usuário |
| **Endpoint** | GET /user/{userId}/clubs |
| **Pré-condição** | Usuário ter cadastro no sistema e ser membro de pelo menos um clube |
| **Passos** | DADO um usuário que é membro de um clube<br>QUANDO uma requisição é realizada no endpoint *GET /user/null/clubs?action=member*<br>ENTÃO a API responde com o status code 400<br>E deve retornar a mensagem “ID do usuário inválido” |
| **Resultado esperado** | 1. A API deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT025 |
| **Título** | Não deve retornar os clubes que o usuário é membro com ID inexistente |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de busca de clubes do usuário |
| **Endpoint** | GET /user/{userId}/clubs |
| **Pré-condição** | Usuário ter cadastro no sistema e ser membro de pelo menos um clube |
| **Passos** | DADO um usuário que é membro de um clube<br>QUANDO uma requisição é realizada no endpoint *GET /user/{userId}/clubs?action=member* com ID inexistente<br>ENTÃO a API responde com o status code 404<br>E deve retornar a mensagem “Usuário não encontrado” |
| **Resultado esperado** | 1. A API deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT026 |
| **Título** | Não deve retornar os clubes que o usuário é membro com ID vazio |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de busca de clubes do usuário |
| **Endpoint** | GET /user/{userId}/clubs |
| **Pré-condição** | Usuário ter cadastro no sistema e ser membro de pelo menos um clube |
| **Passos** | DADO um usuário que é membro de um clube<br>QUANDO uma requisição é realizada no endpoint *GET /user//clubs?action=member*<br>ENTÃO a API responde com o status code 404<br>E deve retornar a mensagem “ID do usuário inválido” |
| **Resultado esperado** | 1. A API deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT027 |
| **Título** | Não deve retornar os clubes que o usuário é administrador com ID nulo |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de busca de clubes do usuário |
| **Endpoint** | GET /user/{userId}/clubs |
| **Pré-condição** | Usuário ter cadastro no sistema e ser criador de pelo menos um clube |
| **Passos** | DADO um usuário que é criador de um clube<br>QUANDO uma requisição é realizada no endpoint *GET /user/null/clubs?action=admin*<br>ENTÃO a API responde com o status code 400<br>E deve retornar a mensagem “ID do usuário inválido” |
| **Resultado esperado** | 1. A API deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT028 |
| **Título** | Não deve retornar os clubes que o usuário é administrador com ID inexistente |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de busca de clubes do usuário |
| **Endpoint** | GET /user/{userId}/clubs |
| **Pré-condição** | Usuário ter cadastro no sistema e ser criador de pelo menos um clube |
| **Passos** | DADO um usuário que é criador de um clube<br>QUANDO uma requisição é realizada no endpoint *GET /user/{userId}/clubs?action=admin* com ID inexistente<br>ENTÃO a API responde com o status code 404<br>E deve retornar a mensagem “Usuário não encontrado” |
| **Resultado esperado** | 1. A API deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT029 |
| **Título** | Não deve retornar os clubes que o usuário é administrador com ID vazio |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de busca de clubes do usuário |
| **Endpoint** | GET /user/{userId}/clubs |
| **Pré-condição** | Usuário ter cadastro no sistema e ser criador de pelo menos um clube |
| **Passos** | DADO um usuário que é criador de um clube<br>QUANDO uma requisição é realizada no endpoint *GET /user//clubs?action=admin*<br>ENTÃO a API responde com o status code 404<br>E deve retornar a mensagem “ID do usuário inválido” |
| **Resultado esperado** | 1. A API deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT030 |
| **Título** | Não deve retornar os clubes que o usuário é mediador com ID nulo |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de busca de clubes do usuário |
| **Endpoint** | GET /user/{userId}/clubs |
| **Pré-condição** | Usuário ter cadastro no sistema e ser mediador de pelo menos um clube |
| **Passos** | DADO um usuário que é mediador de um clube<br>QUANDO uma requisição é realizada no endpoint *GET /user/null/clubs?action=mediator*<br>ENTÃO a API responde com o status code 400<br>E deve retornar a mensagem “ID do usuário inválido” |
| **Resultado esperado** | 1. A API deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT031 |
| **Título** | Não deve retornar os clubes que o usuário é mediador com ID inexistente |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de busca de clubes do usuário |
| **Endpoint** | GET /user/{userId}/clubs |
| **Pré-condição** | Usuário ter cadastro no sistema e ser mediador de pelo menos um clube |
| **Passos** | DADO um usuário que é mediador de um clube<br>QUANDO uma requisição é realizada no endpoint *GET /user/{userId}/clubs?action=mediator* com ID inexistente<br>ENTÃO a API responde com o status code 404<br>E deve retornar a mensagem “Usuário não encontrado” |
| **Resultado esperado** | 1. A API deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT032 |
| **Título** | Não deve retornar os clubes que o usuário é mediador com ID vazio |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de busca de clubes do usuário |
| **Endpoint** | GET /user/{userId}/clubs |
| **Pré-condição** | Usuário ter cadastro no sistema e ser mediador de pelo menos um clube |
| **Passos** | DADO um usuário que é mediador de um clube<br>QUANDO uma requisição é realizada no endpoint *GET /user//clubs?action=mediator*<br>ENTÃO a API responde com o status code 404<br>E deve retornar a mensagem “ID do usuário inválido” |
| **Resultado esperado** | 1. A API deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT033 |
| **Título** | Deve retornar um clube do usuário com os IDs correspondentes |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de busca de um clube do usuário |
| **Endpoint** | GET /user/{userId}/clubs/{clubId} |
| **Pré-condição** | Usuário ter cadastro no sistema e ser membro (ou criador ou mediator) de pelo menos um clube |
| **Passos** | DADO um usuário que é membro de um clube<br>QUANDO uma requisição é realizada no endpoint *GET /user/{userId}/clubs/{clubId}*<br>ENTÃO a API responde com o status code 200<br>E retorna as informação do clube |
| **Resultado esperado** | 1. A API deve responder com o status code correto<br>2. A API deve retornar informações do clube<br>3. O id do usuário e o id do clube correspondente devem estar na resposta |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT034 |
| **Título** | Não deve retornar um clube do usuário com ID de clube inexistente |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de busca de um clube do usuário |
| **Endpoint** | GET /user/{userId}/clubs/{clubId} |
| **Pré-condição** | Usuário ter cadastro no sistema e ser membro (ou criador ou mediator) de pelo menos um clube |
| **Passos** | DADO um usuário que é membro de um clube<br>QUANDO uma requisição é realizada no endpoint *GET /user/{userId}/clubs/{clubId}* com ID do clube inexistente<br>ENTÃO a API responde com o status code 404<br>E deve retornar a mensagem “Clube não encontrado” |
| **Resultado esperado** | 1. A API deve responder com o status code correto<br>2. Deve retornar uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT035 |
| **Título** | Não deve retornar um clube do usuário com ID de usuário inexistente |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de busca de um clube do usuário |
| **Endpoint** | GET /user/{userId}/clubs/{clubId} |
| **Pré-condição** | Usuário ter cadastro no sistema e ser membro (ou criador ou mediator) de pelo menos um clube |
| **Passos** | DADO uma requisição realizada no endpoint *GET /user/{userId}/clubs/{clubId}* com ID do usuário inexistente<br>ENTÃO a API responde com o status code 404<br>E deve retornar a mensagem “Usuário não encontrado” |
| **Resultado esperado** | 1. A API deve responder com o status code correto<br>2. Deve retornar uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT036 |
| **Título** | Não deve retornar um clube do usuário com ID de usuário nulo |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de busca de um clube do usuário |
| **Endpoint** | GET /user/{userId}/clubs/{clubId} |
| **Pré-condição** | Usuário ter cadastro no sistema e ser membro (ou criador ou mediator) de pelo menos um clube |
| **Passos** | DADO uma requisição realizada no endpoint *GET /user/null/clubs/{clubId}*<br>ENTÃO a API responde com o status code 400<br>E deve retornar a mensagem “ID do usuário inválido” |
| **Resultado esperado** | 1. A API deve responder com o status code correto<br>2. Deve retornar uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT037 |
| **Título** | Não deve retornar um clube do usuário com ID de clube nulo |
| **Tipo** | Funcional |
| **Descrição** | Validar o funcionamento do endpoint de busca de um clube do usuário |
| **Endpoint** | GET /user/{userId}/clubs/{clubId} |
| **Pré-condição** | Usuário ter cadastro no sistema e ser membro (ou criador ou mediator) de pelo menos um clube |
| **Passos** | DADO uma requisição realizada no endpoint *GET /user/{userId}/clubs/null*<br>ENTÃO a API responde com o status code 400<br>E deve retornar a mensagem "ID do clube inválido" |
| **Resultado esperado** | 1. A API deve responder com o status code correto<br>2. Deve retornar uma mensagem de falha |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT038 |
| **Título** | Deve fazer login com sucesso |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de autenticação de usuários |
| **Endpoint** | POST /auth |
| **Pré-condição** | Usuário ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO uma requisição é efetuada no endpoint *POST /auth*<br>ENTÃO a API responde com status code 200<br>E deve retornar as informações de login do usuário |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve realizar o login do usuário<br>3. Deve retornar o token de autenticação do usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT039 |
| **Título** | Não deve fazer login com senha incorreta |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de autenticação de usuários |
| **Endpoint** | POST /auth |
| **Pré-condição** | Usuário ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO uma requisição é efetuada no endpoint *POST /auth* com a senha incorreta<br>ENTÃO a API responde com status code 401<br>E deve retornar a mensagem “Senha inválida” |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve realizar o login do usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT040 |
| **Título** | Não deve fazer login com email incorreto |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de autenticação de usuários |
| **Endpoint** | POST /auth |
| **Pré-condição** | Usuário ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO uma requisição é efetuada no endpoint *POST /auth* com o email incorreto<br>ENTÃO a API responde com status code 401<br>E deve retornar a mensagem “Email inválido” |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve realizar o login do usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT041 |
| **Título** | Não deve fazer login com email e senha incorretos |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de autenticação de usuários |
| **Endpoint** | POST /auth |
| **Pré-condição** | Usuário ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO uma requisição é efetuada no endpoint *POST /auth* com o email e senha incorretos<br>ENTÃO a API responde com status code 401<br>E deve retornar a mensagem “Email ou senha inválidos” |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve realizar o login do usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT042 |
| **Título** | Não deve fazer login com email vazio |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de autenticação de usuários |
| **Endpoint** | POST /auth |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição efetuada no endpoint *POST /auth* sem informar o email<br>ENTÃO a API responde com status code 400<br>E deve retornar a mensagem “Email obrigatório” |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve realizar o login do usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT043 |
| **Título** | Não deve fazer login com senha vazia |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de autenticação de usuários |
| **Endpoint** | POST /auth |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição efetuada no endpoint *POST /auth* sem informar a senha<br>ENTÃO a API responde com status code 400<br>E deve retornar a mensagem "Senha obrigatória" |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve realizar o login do usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT044 |
| **Título** | Não deve fazer login com email e senha vazios |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de autenticação de usuários |
| **Endpoint** | POST /auth |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição efetuada no endpoint *POST /auth* sem informar email e senha<br>ENTÃO a API responde com status code 400<br>E deve retornar a mensagem "Email e senha obrigatórios" |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve realizar o login do usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT045 |
| **Título** | Não deve fazer login com email inválido |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de autenticação de usuários |
| **Endpoint** | POST /auth |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição efetuada no endpoint *POST /auth* informando um e-mail sem @<br>ENTÃO a API responde com status code 400<br>E deve retornar a mensagem "Não corresponde ao formato válido de email" |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve realizar o login do usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT046 |
| **Título** | Não deve fazer login com body vazio |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de autenticação de usuários |
| **Endpoint** | POST /auth |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição efetuada no endpoint *POST /auth* com o corpo vazio<br>ENTÃO a API responde com status code 400<br>E deve retornar a mensagem "O corpo da requisição não pode ser vazio" |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve realizar o login do usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT047 |
| **Título** | Deve criar um usuário com sucesso |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | Usuário não ter cadastro no sistema |
| **Passos** | DADO um usuário sem cadastro no sistema<br>QUANDO uma requisição é realizada no endpoint *POST /user*<br>ENTÃO a API responde com status code 201<br>E o usuário é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve realizar o cadastro do usuário<br>3. Deve retornar as informações do usuário cadastrado |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT048 |
| **Título** | Não deve criar um usuário com e-mail já cadastrado |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | Email ser cadastro no sistema por algum usuário |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* com um email já cadastrado<br>ENTÃO a API responde com status code 409<br>E deve retornar uma mensagem de falha “Esse email já está sendo usado”<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT049 |
| **Título** | Não deve criar um usuário com e-mail inválido |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* com um email sem @<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha "Não corresponde ao formato válido de email"<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT050 |
| **Título** | Não deve criar um usuário com e-mail vazio |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* sem email<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha "Não corresponde ao formato válido de email"<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT051 |
| **Título** | Não deve criar um usuário com e-mail sendo espaços em branco |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* informando apenas espaços em branco no email<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha "Não corresponde ao formato válido de email"<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT052 |
| **Título** | Não deve criar um usuário com e-mail nulo |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* com email nulo<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha "Não corresponde ao formato válido de email"<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT053 |
| **Título** | Não deve criar um usuário com senha vazia |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* sem informar a senha<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha "Senha inválida"<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT054 |
| **Título** | Não deve criar um usuário com senha sendo espaços em branco |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* informando apenas espaços em branco na senha<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha "Senha inválida"<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT055 |
| **Título** | Não deve criar um usuário com senha nula |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* com a senha nula<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha “Senha inválida”<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT056 |
| **Título** | Não deve criar um usuário com confirmação de senha vazia |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* sem informar a confirmação de senha<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha “Confirmação de senha inválida”<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT057 |
| **Título** | Não deve criar um usuário com confirmação de senha sendo espaços em branco |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* informando apenas espaços em branco na confirmação de senha<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha "Confirmação de senha inválida"<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT058 |
| **Título** | Não deve criar um usuário com confirmação de senha nula |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* com confirmação de senha nula<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha "Confirmação de senha inválida"<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT059 |
| **Título** | Não deve criar um usuário com senha e confirmação de senha diferentes |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* informando a confirmação de senha diferente da senha<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha “Senhas não coincidem”<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT060 |
| **Título** | Não deve criar um usuário com nome vazio |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* sem informar um nome<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha “Nome inválido”<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT061 |
| **Título** | Não deve criar um usuário com nome sendo espaços em branco |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* informando um nome preenchido com espaços em branco<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha “Nome inválido”<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT062 |
| **Título** | Não deve criar um usuário com nome nulo |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* com nome nulo<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha “Nome inválido”<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT063 |
| **Título** | Não deve criar um usuário com todos os campos vazios |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* sem informar nenhum campo<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha "Não corresponde ao formato válido de email"<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT064 |
| **Título** | Não deve criar um usuário com todos os campos sendo espaços em branco |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* preenchendo todos os campos com espaço em branco<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha "Não corresponde ao formato válido de email"<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT065 |
| **Título** | Não deve criar um usuário com todos os campos nulos |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* com todos os campos sendo nulos<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha "Não corresponde ao formato válido de email"<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT066 |
| **Título** | Não deve criar um usuário com body vazio |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* sem o corpo da requisição<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha "Corpo da requisição inválido"<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT067 |
| **Título** | Não deve criar um usuário com body sendo espaços em branco |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de cadastro de usuários |
| **Endpoint** | POST /user |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *POST /user* preenchendo o corpo da requisição com espaços em branco<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha "Corpo da requisição inválido"<br>E o usuário não é criado |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve cadastrar o usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT068 |
| **Título** | Deve editar um usuário com sucesso |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de edição do cadastro de usuários |
| **Endpoint** | PUT /user/{userId} |
| **Pré-condição** | Usuário ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO uma requisição é realizada no endpoint *PUT /user/{userId}*<br>ENTÃO a API responde com status code 200<br>E uma indicação de sucesso na operação |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve atualizar as informações do usuário<br>3. Deve retornar uma indicação de sucesso |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT069 |
| **Título** | Não deve editar um usuário sem token de autenticação |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de edição do cadastro de usuários |
| **Endpoint** | PUT /user/{userId} |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição sem token de autenticação realizada no endpoint *PUT /user/{userId}*<br>ENTÃO a API responde com status code 401<br>E deve retornar uma mensagem de falha "Você não tem permissão pra fazer essa chamada" |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve atualizar as informações do usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT070 |
| **Título** | Não deve editar um usuário com token de autenticação inválido |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de edição do cadastro de usuários |
| **Endpoint** | PUT /user/{userId} |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *PUT /user/{userId}* com token de autenticação inválido<br>ENTÃO a API responde com status code 401<br>E deve retornar uma mensagem de falha "Você não tem permissão pra fazer essa chamada" |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve atualizar as informações do usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT071 |
| **Título** | Não deve editar um usuário com token de autenticação sendo nulo |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de edição do cadastro de usuários |
| **Endpoint** | PUT /user/{userId} |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *PUT /user/{userId}* com token de autenticação nulo<br>ENTÃO a API responde com status code 401<br>E deve retornar uma mensagem de falha "Você não tem permissão pra fazer essa chamada" |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve atualizar as informações do usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT072 |
| **Título** | Não deve editar um usuário com token de autenticação sendo espaço em branco |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de edição do cadastro de usuários |
| **Endpoint** | PUT /user/{userId} |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *PUT /user/{userId}* com token de autenticação preenchido com espaços em branco<br>ENTÃO a API responde com status code 401<br>E deve retornar uma mensagem de falha "Você não tem permissão pra fazer essa chamada" |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve atualizar as informações do usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT073 |
| **Título** | Não deve editar um usuário com token de autenticação de outro usuário |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de edição do cadastro de usuários |
| **Endpoint** | PUT /user/{userId} |
| **Pré-condição** | Usuário ter cadastro no sistema |
| **Passos** | DADO uma requisição realizada no endpoint *PUT /user/{userId}* com token de autenticação pertencente a outro usuário<br>ENTÃO a API responde com status code 401<br>E deve retornar uma mensagem de falha "Você não tem permissão pra fazer essa chamada" |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve atualizar as informações do usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT074 |
| **Título** | Não deve editar um usuário com ID inválido |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de edição do cadastro de usuários |
| **Endpoint** | PUT /user/{userId} |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *PUT /user/{userId}* com ID do usuário inválido<br>ENTÃO a API responde com status code 401<br>E deve retornar uma mensagem de falha "Você não tem permissão pra fazer essa chamada" |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve atualizar as informações do usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT075 |
| **Título** | Não deve editar um usuário com ID nulo |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de edição do cadastro de usuários |
| **Endpoint** | PUT /user/{userId} |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *PUT /user/null*<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha "ID do usuário inválido" |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve atualizar as informações do usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT076 |
| **Título** | Não deve editar um usuário com ID sendo espaço em branco |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de edição do cadastro de usuários |
| **Endpoint** | PUT /user/{userId} |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *PUT /user/{userId}* com ID do usuário preenchido com espaços em branco<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha "ID do usuário inválido" |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve atualizar as informações do usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT077 |
| **Título** | Não deve editar um usuário com ID vazio |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de edição do cadastro de usuários |
| **Endpoint** | PUT /user/{userId} |
| **Pré-condição** | - |
| **Passos** | DADO uma requisição realizada no endpoint *PUT /user/*<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha "ID do usuário inválido" |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve atualizar as informações do usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT078 |
| **Título** | Não deve editar um usuário com nick já cadastrado |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de edição do cadastro de usuários |
| **Endpoint** | PUT /user/{userId} |
| **Pré-condição** | Usuário ter cadastro no sistema, Nick já ser cadastrado por algum usuário |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO uma requisição é realizada no endpoint *PUT /user/{userId}*<br>ENTÃO a API responde com status code 409<br>E deve retornar uma mensagem de falha "Já em uso" |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve atualizar as informações do usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT079 |
| **Título** | Não deve editar um usuário com nick igual ao do usuário autenticado |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de edição do cadastro de usuários |
| **Endpoint** | PUT /user/{userId} |
| **Pré-condição** | Usuário ter cadastro no sistema, Nick já ser cadastrado pelo usuário |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO uma requisição é realizada no endpoint *PUT /user/{userId}*<br>ENTÃO a API responde com status code 409<br>E deve retornar uma mensagem de falha "Já em uso" |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve atualizar as informações do usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT080 |
| **Título** | Não deve editar um usuário com nick nulo |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de edição do cadastro de usuários |
| **Endpoint** | PUT /user/{userId} |
| **Pré-condição** | Usuário ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO uma requisição é realizada no endpoint *PUT /user/{userId}* com nick nulo<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha "Nick inválido" |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve atualizar as informações do usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT081 |
| **Título** | Não deve editar um usuário com nick sendo espaço em branco |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de edição do cadastro de usuários |
| **Endpoint** | PUT /user/{userId} |
| **Pré-condição** | Usuário ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO uma requisição é realizada no endpoint *PUT /user/{userId}*<br>com nick preenchido com espaços em branco<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha “Nick inválido” |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve atualizar as informações do usuário |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT082 |
| **Título** | Não deve editar um usuário com nick vazio |
| **Tipo** | Funcional |
| **Descrição** | Validar o endpoint de edição do cadastro de usuários |
| **Endpoint** | PUT /user/{userId} |
| **Pré-condição** | Usuário ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO uma requisição é realizada no endpoint *PUT /user/{userId}*<br>sem informar o nick<br>ENTÃO a API responde com status code 400<br>E deve retornar uma mensagem de falha “Nick inválido” |
| **Resultado esperado** | 1. Deve retornar o status code correto<br>2. Deve retornar uma mensagem de falha<br>3. Não deve atualizar as informações do usuário |
| **Status** | Falhou |

## A.2 E2E

| | |
|---|---|
| **ID** | CT001 |
| **Título** | Deve fazer login com email e senha válidos com sucesso |
| **Tipo** | E2E |
| **Descrição** | Validação da tela de login com credenciais válidas |
| **Funcionalidade** | Login |
| **Pré-condição** | Usuário deve ter cadastro no sistema |
| **Passos** | DADO um usuário que tenha cadastro no sistema<br>QUANDO as credenciais são preenchidas<br>ENTÃO a página de perfil é exibida |
| **Resultado esperado** | 1. Realizar o login com sucesso<br>2. Apresentar a tela de perfil com nome do usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT002 |
| **Título** | Deve fazer login com sucesso com email e senha com tamanho mínimo |
| **Tipo** | E2E |
| **Descrição** | Validar o valor limite (mínimo) para login |
| **Funcionalidade** | Login |
| **Pré-condição** | Cadastro do usuário possuir email e senha com tamanho mínimo permitido |
| **Passos** | DADO um usuário que tenha cadastro no sistema<br>QUANDO as credenciais são preenchidas<br>ENTÃO a página de perfil é exibida |
| **Resultado esperado** | 1. Realizar o login com sucesso<br>2. Apresentar a tela de perfil com nome do usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT003 |
| **Título** | Deve fazer login com sucesso com email e senha com tamanho máximo |
| **Tipo** | E2E |
| **Descrição** | Validar o valor limite (máximo) para login |
| **Funcionalidade** | Login |
| **Pré-condição** | Cadastro do usuário possuir email e senha com tamanho máximo permitido |
| **Passos** | DADO um usuário que tenha cadastro no sistema<br>QUANDO as credenciais são preenchidas<br>ENTÃO a página de perfil é exibida |
| **Resultado esperado** | 1. Realizar o login com sucesso<br>2. Apresentar a tela de perfil com nome do usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT004 |
| **Título** | Deve fazer login como criador do clube com sucesso |
| **Tipo** | E2E |
| **Descrição** | Validar o login de usuário com cargo criador |
| **Funcionalidade** | Login |
| **Pré-condição** | Usuário deve ter cadastro no sistema e ter criado pelo menos um clube |
| **Passos** | DADO um usuário que criou um clube<br>QUANDO as credenciais são preenchidas<br>ENTÃO as opções de *"roles"* são exibidas<br>E uma lista de clubes do usuário |
| **Resultado esperado** | 1. Apresentar as opções de *"roles"*<br>2. Fazer login com sucesso<br>3. Exibir a página de perfil do clube selecionado<br>4. Exibir o cargo do usuário no clube |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT005 |
| **Título** | Deve fazer login como mediador do clube com sucesso |
| **Tipo** | E2E |
| **Descrição** | Validar o login de usuário com cargo mediador |
| **Funcionalidade** | Login |
| **Pré-condição** | Usuário deve ter cadastro no sistema e ser mediador de pelo menos um clube |
| **Passos** | DADO um usuário que é mediador de um clube<br>QUANDO as credenciais são preenchidas<br>as opções de *"roles"* são exibidas<br>E uma lista de clubes do usuário |
| **Resultado esperado** | 1. Apresentar as opções de *"roles"*<br>2. Fazer login com sucesso<br>3. Exibir a página de perfil do clube selecionado<br>4. Exibir o cargo do usuário no clube |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT006 |
| **Título** | Deve fazer login como clubista com sucesso |
| **Tipo** | E2E |
| **Descrição** | Validar o login de usuário como clubista |
| **Funcionalidade** | Login |
| **Pré-condição** | Usuário deve ter cadastro no sistema e ser criador ou mediador de pelo menos um clube |
| **Passos** | DADO um usuário que criou ou é mediador de um clube<br>QUANDO as credenciais são preenchidas<br>ENTÃO as opções de *"roles"* são exibidas<br>E uma lista de clubes do usuário é exibida |
| **Resultado esperado** | 1. Apresentar as opções de *"roles"*<br>2. Fazer login com sucesso<br>3. Exibir a página de perfil do clube selecionado |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT007 |
| **Título** | Não deve fazer login com email vazio |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* do campo de email |
| **Funcionalidade** | Login |
| **Pré-condição** | Usuário deve ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO o campo *email* não é preenchido<br>ENTÃO uma mensagem de falha é exibida na tela *"Email obrigatório"*<br>E o login não é realizado |
| **Resultado esperado** | 1. Não deve realizar o login<br>2. A mensagem de falha deve ser exibida no canto inferior direito da tela |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT008 |
| **Título** | Não deve fazer login com senha vazia |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* do campo de senha |
| **Funcionalidade** | Login |
| **Pré-condição** | Usuário deve ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO o campo *senha* não é preenchido<br>ENTÃO uma mensagem de falha é exibida na tela *"Senha obrigatória"*<br>E o login não é realizado |
| **Resultado esperado** | 1. Não deve realizar o login<br>2. A mensagem de falha deve ser exibida no canto inferior direito da tela |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT009 |
| **Título** | Não deve fazer login com email e senha vazios |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* simultâneo dos campos de email e senha |
| **Funcionalidade** | Login |
| **Pré-condição** | Usuário deve ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO os campos das credenciais não são preenchidos<br>ENTÃO uma mensagem de falha é exibida na tela *"Email e senha obrigatórios"*<br>E o login não é realizado |
| **Resultado esperado** | 1. Não deve realizar o login<br>2. A mensagem de falha deve ser exibida no canto inferior direito da tela |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT010 |
| **Título** | Não deve fazer login com email incorreto |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* do campo de email |
| **Funcionalidade** | Login |
| **Pré-condição** | Usuário deve ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO o campo *email* é preenchido com um valor incorreto<br>ENTÃO uma mensagem de falha é exibida na tela *"Email inválido"*<br>E o login não é realizado |
| **Resultado esperado** | 1. Não deve realizar o login<br>2. A mensagem de falha deve ser exibida no canto inferior direito da tela |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT011 |
| **Título** | Não deve fazer login com senha incorreta |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* do campo de senha |
| **Funcionalidade** | Login |
| **Pré-condição** | Usuário deve ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO o campo *senha* é preenchido com um valor incorreto<br>ENTÃO uma mensagem de falha é exibida na tela *"Senha inválida"*<br>E o login não é realizado |
| **Resultado esperado** | 1. Não deve realizar o login<br>2. A mensagem de falha deve ser exibida no canto inferior direito da tela |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT012 |
| **Título** | Não deve fazer login com email e senha incorretos |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* simultâneo dos campo de email e senha |
| **Funcionalidade** | Login |
| **Pré-condição** | Usuário deve ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO *email* e *senha* são preenchidos com valores incorretos<br>ENTÃO uma mensagem de falha é exibida na tela: *"Credenciais inválidas"*<br>E o login não é realizado |
| **Resultado esperado** | 1. Não deve realizar o login<br>2. A mensagem de falha deve ser exibida no canto inferior direito da tela |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT013 |
| **Título** | Não deve fazer login com senha com espaços no final |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* do campo de senha |
| **Funcionalidade** | Login |
| **Pré-condição** | Usuário deve ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO o campo *senha* correta é preenchida mas com espaços no final<br>ENTÃO uma mensagem de falha é exibida na tela *"Senha inválida"*<br>E o login não é realizado |
| **Resultado esperado** | 1. Não deve realizar o login<br>2. A mensagem de falha deve ser exibida no canto inferior direito da tela |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT014 |
| **Título** | Não deve fazer login com email sem @ |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* do campo de email |
| **Funcionalidade** | Login |
| **Pré-condição** | Usuário deve ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO o campo *email* é preenchido sem @<br>ENTÃO uma mensagem de falha é exibida na tela *"Não corresponde ao formato válido de email"*<br>E o login não é realizado |
| **Resultado esperado** | 1. Não deve realizar o login<br>2. A mensagem de falha deve ser exibida no canto inferior direito da tela |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT015 |
| **Título** | Não deve fazer login com senha com emojis |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* do campo de senha |
| **Funcionalidade** | Login |
| **Pré-condição** | Usuário deve ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO o campo *senha* é preenchido com emojis<br>ENTÃO uma mensagem de falha é exibida na tela *"Senha inválida"*<br>E o login não é realizado |
| **Resultado esperado** | 1. Não deve realizar o login<br>2. A mensagem de falha deve ser exibida no canto inferior direito da tela |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT016 |
| **Título** | Não deve fazer login com senha com SQL injection |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* do campo de senha |
| **Funcionalidade** | Login |
| **Pré-condição** | Usuário deve ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO o campo *senha* é preenchido com SQL injection<br>ENTÃO uma mensagem de falha é exibida na tela *"Senha inválida"*<br>E o login não é realizado |
| **Resultado esperado** | 1. Não deve realizar o login<br>2. A mensagem de falha deve ser exibida no canto inferior direito da tela |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT017 |
| **Título** | Não deve fazer login com email com script injection |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* do campo de email |
| **Funcionalidade** | Login |
| **Pré-condição** | Usuário deve ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO o campo *email* é preenchido com script injection<br>ENTÃO uma mensagem de falha é exibida na tela *"Email inválido"*<br>E o login não é realizado |
| **Resultado esperado** | 1. Não deve realizar o login<br>2. A mensagem de falha deve ser exibida no canto inferior direito da tela |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT018 |
| **Título** | Não deve fazer login com email muito grande |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* do campo de senha |
| **Funcionalidade** | Login |
| **Pré-condição** | Usuário deve ter cadastro no sistema |
| **Passos** | DADO um usuário com cadastro no sistema<br>QUANDO o campo *email* é preenchido com um valor muito grande<br>ENTÃO uma mensagem de falha é exibida na tela *"Email inválido"*<br>E o login não é realizado |
| **Resultado esperado** | 1. Não deve realizar o login<br>2. A mensagem de falha deve ser exibida no canto inferior direito da tela |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT019 |
| **Título** | Deve criar um usuário com sucesso |
| **Tipo** | E2E |
| **Descrição** | Validar a tela de cadastro de usuário |
| **Funcionalidade** | Cadastrar usuário |
| **Pré-condição** | Usuário não deve ter cadastro no sistema |
| **Passos** | DADO um usuário sem cadastro no sistema<br>QUANDO os campos são preenchidos com dados válidos<br>ENTÃO o usuário é criado<br>E o login é realizado<br>E a página de perfil do usuário é exibida na tela |
| **Resultado esperado** | 1. Deve cadastrar o usuário<br>2. Deve realizar o login automático<br>3. Deve exibir o perfil do usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT020 |
| **Título** | Deve criar um usuário com nome longo com sucesso |
| **Tipo** | E2E |
| **Descrição** | Validar o valor limite (máximo) do campo *Nome completo* |
| **Funcionalidade** | Cadastrar usuário |
| **Pré-condição** | Usuário não deve ter cadastro no sistema |
| **Passos** | DADO um usuário sem cadastro no sistema<br>QUANDO o campo *nome completo* é preenchido com tamanho máximo<br>ENTÃO o usuário é criado<br>E o login é realizado<br>E a página de perfil do usuário é exibida na tela |
| **Resultado esperado** | 1. Deve cadastrar o usuário<br>2. Deve realizar o login automático<br>3. Deve exibir o perfil do usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT021 |
| **Título** | Deve criar um usuário com nome de tamanho mínimo com sucesso |
| **Tipo** | E2E |
| **Descrição** | Validar o valor limite (mínimo) do campo *Nome completo* |
| **Funcionalidade** | Cadastrar usuário |
| **Pré-condição** | Usuário não deve ter cadastro no sistema |
| **Passos** | DADO um usuário sem cadastro no sistema<br>QUANDO o campo *nome completo* é preenchido com tamanho mínimo<br>ENTÃO o usuário é criado<br>E o login é realizado<br>E a página de perfil do usuário é exibida na tela |
| **Resultado esperado** | 1. Deve cadastrar o usuário<br>2. Deve realizar o login automático<br>3. Deve exibir o perfil do usuário |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT022 |
| **Título** | Não deve criar um usuário com todos os campos vazio |
| **Tipo** | E2E |
| **Descrição** | Validar os *inputs* simultâneos da página de criação de usuário |
| **Funcionalidade** | Cadastrar usuário |
| **Pré-condição** | - |
| **Passos** | DADO um usuário sem cadastro no sistema<br>QUANDO os campos não são preenchidos<br>ENTÃO mensagens de alerta são exibidos nos campos<br>E o usuário não é criado |
| **Resultado esperado** | 1. Não deve cadastrar usuário sem dados<br>2. Deve exibir mensagens de alerta em cada campo vazio |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT023 |
| **Título** | Não deve criar um usuário com nome vazio |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de nome |
| **Funcionalidade** | Cadastrar usuário |
| **Pré-condição** | - |
| **Passos** | DADO um usuário sem cadastro no sistema<br>QUANDO o campo *nome completo* não é preenchido<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o usuário não é criado |
| **Resultado esperado** | 1. Não deve cadastrar usuário sem dados<br>2. Deve exibir a mensagem de alerta no campo *nome completo* |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT024 |
| **Título** | Não deve criar um usuário com email vazio |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de email |
| **Funcionalidade** | Cadastrar usuário |
| **Pré-condição** | - |
| **Passos** | DADO um usuário sem cadastro no sistema<br>QUANDO o campo *email* não é preenchido<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o usuário não é criado |
| **Resultado esperado** | 1. Não deve cadastrar usuário sem dados<br>2. Deve exibir a mensagem de alerta no campo *email* |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT025 |
| **Título** | Não deve criar um usuário com senha vazia |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de senha |
| **Funcionalidade** | Cadastrar usuário |
| **Pré-condição** | - |
| **Passos** | DADO um usuário sem cadastro no sistema<br>QUANDO o campo *senha* não é preenchido<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o usuário não é criado |
| **Resultado esperado** | 1. Não deve cadastrar usuário sem dados<br>2. Deve exibir a mensagem de alerta no campo *senha* |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT026 |
| **Título** | Não deve criar um usuário com confirmação de senha vazia |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de confirmação de senha |
| **Funcionalidade** | Cadastrar usuário |
| **Pré-condição** | - |
| **Passos** | DADO um usuário sem cadastro no sistema<br>QUANDO o campo *confirmar senha* não é preenchido<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o usuário não é criado |
| **Resultado esperado** | 1. Não deve cadastrar usuário sem dados<br>2. Deve exibir a mensagem de alerta no campo *confirmar senha* |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT027 |
| **Título** | Não deve criar um usuário com email inválido |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de email |
| **Funcionalidade** | Cadastrar usuário |
| **Pré-condição** | - |
| **Passos** | DADO um usuário sem cadastro no sistema<br>QUANDO o campo *email* é preenchido sem @<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o usuário não é criado |
| **Resultado esperado** | 1. Não deve cadastrar usuário com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *email* |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT028 |
| **Título** | Não deve criar um usuário com confirmação de senha diferente da senha |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* simultanêo de senha e confirmação da senha |
| **Funcionalidade** | Cadastrar usuário |
| **Pré-condição** | - |
| **Passos** | DADO um usuário sem cadastro no sistema<br>QUANDO os campos *senha* e *confirmar senha* são preenchidos com valores diferentes<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o usuário não é criado |
| **Resultado esperado** | 1. Não deve cadastrar usuário com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *confirmar senha* |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT029 |
| **Título** | Não deve criar um usuário com nome inválido |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de nome |
| **Funcionalidade** | Cadastrar usuário |
| **Pré-condição** | - |
| **Passos** | DADO um usuário sem cadastro no sistema<br>QUANDO o campo *nome completo* é preenchido com caracteres especiais<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o usuário não é criado |
| **Resultado esperado** | 1. Não deve cadastrar usuário com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *nome completo* |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT030 |
| **Título** | Não deve criar um usuário com senha muito curta |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de senha |
| **Funcionalidade** | Cadastrar usuário |
| **Pré-condição** | - |
| **Passos** | DADO um usuário sem cadastro no sistema<br>QUANDO o campo *senha* é preenchido com tamanho muito pequeno<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o usuário não é criado |
| **Resultado esperado** | 1. Não deve cadastrar usuário com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *senha* |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT031 |
| **Título** | Não deve criar um usuário com confirmação de senha com espaços |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de confirmação da senha |
| **Funcionalidade** | Cadastrar usuário |
| **Pré-condição** | - |
| **Passos** | DADO um usuário sem cadastro no sistema<br>QUANDO o campo *confirmar senha* é preenchido com espaços<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o usuário não é criado |
| **Resultado esperado** | 1. Não deve cadastrar usuário com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *confirmar senha* |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT032 |
| **Título** | Não deve criar um usuário com nome com script injection |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de script injection |
| **Funcionalidade** | Cadastrar usuário |
| **Pré-condição** | - |
| **Passos** | DADO um usuário sem cadastro no sistema<br>QUANDO o campo *nome completo* é preenchido com script injection<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o usuário não é criado |
| **Resultado esperado** | 1. Não deve cadastrar usuário com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *nome completo* |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT033 |
| **Título** | Não deve criar um usuário com email com SQL injection |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de SQL injection |
| **Funcionalidade** | Cadastrar usuário |
| **Pré-condição** | - |
| **Passos** | DADO um usuário sem cadastro no sistema<br>QUANDO o campo *email* é preenchido com SQL injection<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o usuário não é criado |
| **Resultado esperado** | 1. Não deve cadastrar usuário com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *email* |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT034 |
| **Título** | Não deve criar um usuário com nome maior que o permitido |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de nome |
| **Funcionalidade** | Cadastrar usuário |
| **Pré-condição** | - |
| **Passos** | DADO um usuário sem cadastro no sistema<br>QUANDO o campo *nome completo* é preenchido maior que o permitido<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o usuário não é criado |
| **Resultado esperado** | 1. Não deve cadastrar usuário com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *nome completo* |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT035 |
| **Título** | Não deve criar um usuário com senha com emojis |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de emojis |
| **Funcionalidade** | Cadastrar usuário |
| **Pré-condição** | - |
| **Passos** | DADO um usuário sem cadastro no sistema<br>QUANDO o campo *senha* é preenchido com emojis<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o usuário não é criado |
| **Resultado esperado** | 1. Não deve cadastrar usuário com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *senha* |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT036 |
| **Título** | Deve criar um clube com informações básicas com sucesso |
| **Tipo** | E2E |
| **Descrição** | Validar a criação de clube com campos obrigatórios |
| **Funcionalidade** | Criar clube |
| **Pré-condição** | Usuário ter cadastro e estar logado no sistema |
| **Passos** | DADO que o usuário acessou a página de criação de clubes<br>QUANDO preenche todos os campos obrigatórios<br>ENTÃO uma mensagem de sucesso e instruções são exibidas na tela |
| **Resultado esperado** | 1. Deve exibir uma mensagem de sucesso<br>2. Deve exibir instruções após a criação do clube |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT037 |
| **Título** | Deve criar um clube informando todos os campos com sucesso |
| **Tipo** | E2E |
| **Descrição** | Validar a criação de clube |
| **Funcionalidade** | Criar clube |
| **Pré-condição** | Usuário ter cadastro e estar logado no sistema |
| **Passos** | DADO que o usuário acessou a página de criação de clubes<br>QUANDO preenche todos os campos<br>ENTÃO uma mensagem de sucesso e instruções são exibidas na tela |
| **Resultado esperado** | 1. Deve exibir uma mensagem de sucesso<br>2. Deve exibir instruções após a criação do clube |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT038 |
| **Título** | Deve criar um clube privado com sucesso |
| **Tipo** | E2E |
| **Descrição** | Validar a seleção do *input* de clube privado |
| **Funcionalidade** | Criar clube |
| **Pré-condição** | Usuário ter cadastro e estar logado no sistema |
| **Passos** | DADO que o usuário acessou a página de criação de clubes<br>QUANDO seleciona o campo *É Privado?*<br>ENTÃO uma mensagem de sucesso e instruções são exibidas na tela |
| **Resultado esperado** | 1. Deve exibir uma mensagem de sucesso<br>2. Deve exibir instruções após a criação do clube |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT039 |
| **Título** | Deve criar um clube com informações com valores mínimos |
| **Tipo** | E2E |
| **Descrição** | Validar o valor limite (mínimo) dos campos |
| **Funcionalidade** | Criar clube |
| **Pré-condição** | Usuário ter cadastro e estar logado no sistema |
| **Passos** | DADO que o usuário acessou a página de criação de clubes<br>QUANDO todos os campos são preenchidos com valores tamanhos mínimos<br>ENTÃO uma mensagem de sucesso e instruções são exibidas na tela |
| **Resultado esperado** | 1. Deve exibir uma mensagem de sucesso<br>2. Deve exibir instruções após a criação do clube |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT040 |
| **Título** | Deve criar um clube com informações com valores máximos |
| **Tipo** | E2E |
| **Descrição** | Validar o valor limite (máximo) dos campos |
| **Funcionalidade** | Criar clube |
| **Pré-condição** | Usuário ter cadastro e estar logado no sistema |
| **Passos** | DADO que o usuário acessou a página de criação de clubes<br>QUANDO todos os campos são preenchidos com valores de tamanhos máximos<br>ENTÃO uma mensagem de sucesso e instruções são exibidas na tela |
| **Resultado esperado** | 1. Deve exibir uma mensagem de sucesso<br>2. Deve exibir instruções após a criação do clube |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT041 |
| **Título** | Deve criar um clube com descrição com emojis |
| **Tipo** | E2E |
| **Descrição** | Validar a possibilidade de usar emojis na descrição |
| **Funcionalidade** | Criar clube |
| **Pré-condição** | Usuário ter cadastro e estar logado no sistema |
| **Passos** | DADO que o usuário acessou a página de criação de clubes<br>QUANDO o campo *descrição* é preenchido com emojis<br>ENTÃO uma mensagem de sucesso e instruções são exibidas na tela |
| **Resultado esperado** | 1. Deve exibir uma mensagem de sucesso<br>2. Deve exibir instruções após a criação do clube |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT042 |
| **Título** | Não deve criar um clube com nome vazio |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de nome do clube |
| **Funcionalidade** | Criar clube |
| **Pré-condição** | Usuário ter cadastro e estar logado no sistema |
| **Passos** | DADO que o usuário acessou a página de criação de clubes<br>QUANDO o campo *nome* não é preenchido<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o clube não é criado |
| **Resultado esperado** | 1. Não deve criar um clube sem dados<br>2. Deve exibir a mensagem de alerta no campo *nome* |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT043 |
| **Título** | Não deve criar um clube com descrição vazia |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de descrição do clube |
| **Funcionalidade** | Criar clube |
| **Pré-condição** | Usuário ter cadastro e estar logado no sistema |
| **Passos** | DADO que o usuário acessou a página de criação de clubes<br>QUANDO o campo *descrição* não é preenchido<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o clube não é criado |
| **Resultado esperado** | 1. Não deve criar um clube sem dados<br>2. Deve exibir a mensagem de alerta no campo *descrição* |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT044 |
| **Título** | Não deve criar um clube com imagem de formato inválido |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de foto do clube |
| **Funcionalidade** | Criar clube |
| **Pré-condição** | Usuário ter cadastro e estar logado no sistema |
| **Passos** | DADO que o usuário acessou a página de criação de clubes<br>QUANDO é inserido um arquivo com formato diferente de imagem no campo *foto do clube*<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o clube não é criado |
| **Resultado esperado** | 1. Não deve criar um clube com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *foto do clube* |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT045 |
| **Título** | Não deve criar um clube com arquivo de regras de formato inválido |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de arquivo de regras do clube |
| **Funcionalidade** | Criar clube |
| **Pré-condição** | Usuário ter cadastro e estar logado no sistema |
| **Passos** | DADO que o usuário acessou a página de criação de clubes<br>QUANDO é inserido um arquivo com formato diferente do permitido no campo *regras do clube*<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o clube não é criado |
| **Resultado esperado** | 1. Não deve criar um clube com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *regras do clube* |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT046 |
| **Título** | Não deve criar um clube sem nome e sem descrição |
| **Tipo** | E2E |
| **Descrição** | Validar simultaneamente os *inputs* de nome e descrição |
| **Funcionalidade** | Criar clube |
| **Pré-condição** | Usuário ter cadastro e estar logado no sistema |
| **Passos** | DADO que o usuário acessou a página de criação de clubes<br>QUANDO os campos *nome* e *descrição* não são preenchidos<br>ENTÃO as mensagens de alerta são exibidas abaixo dos campos<br>E o clube não é criado |
| **Resultado esperado** | 1. Não deve criar um clube com dados inconsistentes<br>2. Deve exibir as mensagens de alerta nos campos *nome* e *descrição* |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT047 |
| **Título** | Não deve criar um clube com imagem maior que 1MB |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de foto do clube |
| **Funcionalidade** | Criar clube |
| **Pré-condição** | Usuário ter cadastro e estar logado no sistema |
| **Passos** | DADO que o usuário acessou a página de criação de clubes<br>QUANDO é inserida uma imagem maior que 1BM no campo *foto do clube*<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o clube não é criado |
| **Resultado esperado** | 1. Não deve criar um clube com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *foto do clube* |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT048 |
| **Título** | Não deve criar um clube com arquivo de regras maior que 1MB |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de regras do clube |
| **Funcionalidade** | Criar clube |
| **Pré-condição** | Usuário ter cadastro e estar logado no sistema |
| **Passos** | DADO que o usuário acessou a página de criação de clubes<br>QUANDO é inserido um arquivo maior que 1BM no campo *regras do clube*<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o clube não é criado |
| **Resultado esperado** | 1. Não deve criar um clube com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *regras do clube* |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT049 |
| **Título** | Não deve criar um clube com nome contendo script injection |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de nome |
| **Funcionalidade** | Criar clube |
| **Pré-condição** | Usuário ter cadastro e estar logado no sistema |
| **Passos** | DADO que o usuário acessou a página de criação de clubes<br>QUANDO o campo *nome* é preenchido com script injection<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o clube não é criado |
| **Resultado esperado** | 1. Não deve criar um clube com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *nome* |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT050 |
| **Título** | Não deve criar um clube com nome contendo SQL injection |
| **Descrição** | Validar o *input* de nome |
| **Funcionalidade** | Criar clube |
| **Pré-condição** | Usuário ter cadastro e estar logado no sistema |
| **Passos** | DADO que o usuário acessou a página de criação de clubes<br>QUANDO o campo *nome* é preenchido com SQL injection<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o clube não é criado |
| **Resultado esperado** | 1. Não deve criar um clube com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *nome* |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT051 |
| **Título** | Não deve criar um clube com descrição contendo script injection |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de descrição |
| **Funcionalidade** | Criar clube |
| **Pré-condição** | Usuário ter cadastro e estar logado no sistema |
| **Passos** | DADO que o usuário acessou a página de criação de clubes<br>QUANDO o campo *descrição* é preenchido com script injection<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o clube não é criado |
| **Resultado esperado** | 1. Não deve criar um clube com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *descrição* |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT052 |
| **Título** | Não deve criar um clube com descrição contendo SQL injection |
| **Tipo** | E2E |
| **Descrição** | Validar o *input* de descrição |
| **Funcionalidade** | Criar clube |
| **Pré-condição** | Usuário ter cadastro e estar logado no sistema |
| **Passos** | DADO que o usuário acessou a página de criação de clubes<br>QUANDO o campo *descrição* é preenchido com SQL injection<br>ENTÃO a mensagem de alerta é exibida abaixo do campo<br>E o clube não é criado |
| **Resultado esperado** | 1. Não deve criar um clube com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *descrição* |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT053 |
| **Título** | Deve criar e adicionar um livro na estante do clube como criador com sucesso |
| **Tipo** | E2E |
| **Descrição** | Validar o fluxo de criação de novo livro e adição na estante do clube |
| **Funcionalidade** | Adicionar livro à estante do clube |
| **Pré-condição** | Usuário ter cadastro, estar logado no sistema como criador do clube e um livro não cadastrado |
| **Passos** | DADO que o usuário acessou a página de cadastro de livro do clube<br>QUANDO os campos são preenchidos<br>ENTÃO o livro é criado<br>E uma mensagem de sucesso é exibida na tela<br>E o livro é adicionado à estante do clube |
| **Resultado esperado** | 1. Deve criar o novo livro<br>2. Deve adicionar o livro na estante do clube<br>3. Deve exibir mensagens de sucesso ao criar o livro e adicionar na estante do clube |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT054 |
| **Título** | Deve criar e adicionar um livro na estante do clube como mediador com sucesso |
| **Tipo** | E2E |
| **Descrição** | Validar o fluxo de criação de novo livro e adição na estante do clube |
| **Funcionalidade** | Adicionar livro à estante do clube |
| **Pré-condição** | Usuário ter cadastro, estar logado no sistema como mediador do clube e um livro não cadastrado |
| **Passos** | DADO que o usuário acessou a página de cadastro de livro do clube<br>QUANDO os campos são preenchidos<br>ENTÃO o livro é criado<br>E uma mensagem de sucesso é exibida na tela<br>E o livro é adicionado à estante do clube |
| **Resultado esperado** | 1. Deve criar o novo livro<br>2. Deve adicionar o livro na estante do clube<br>3. Deve exibir mensagens de sucesso ao criar o livro e adicionar na estante do clube |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT055 |
| **Título** | Deve criar e adicionar um livro sem preencher os gêneros |
| **Tipo** | E2E |
| **Descrição** | Validar o fluxo de criação de novo livro e adição na estante do clube |
| **Funcionalidade** | Adicionar livro à estante do clube |
| **Pré-condição** | Usuário ter cadastro, estar logado no sistema e um livro não cadastrado |
| **Passos** | DADO que o usuário acessou a página de cadastro de livro do clube<br>QUANDO o campo *gêneros literários do livro* não é preenchido<br>ENTÃO o livro é criado<br>E uma mensagem de sucesso é exibida na tela<br>E o livro é adicionado à estante do clube |
| **Resultado esperado** | 1. Deve criar o novo livro<br>2. Deve adicionar o livro na estante do clube<br>3. Deve exibir mensagens de sucesso ao criar o livro e adicionar na estante do clube |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT056 |
| **Título** | Não deve criar um livro com o titulo vazio |
| **Tipo** | E2E |
| **Descrição** | Validar o fluxo de criação de novo livro e adição na estante do clube |
| **Funcionalidade** | Adicionar livro à estante do clube |
| **Pré-condição** | Usuário ter cadastro, estar logado no sistema e um livro não cadastrado |
| **Passos** | DADO que o usuário acessou a página de cadastro de livro do clube<br>QUANDO o campo *título* não é preenchido<br>ENTÃO uma mensagem de alerta é exibida abaixo do campo<br>E o livro não é criado |
| **Resultado esperado** | 1. Não deve criar um livro sem dados<br>2. Deve exibir a mensagem de alerta no campo *título* |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT057 |
| **Título** | Não deve criar um livro com o autor vazio |
| **Tipo** | E2E |
| **Descrição** | Validar o fluxo de criação de novo livro e adição na estante do clube |
| **Funcionalidade** | Adicionar livro à estante do clube |
| **Pré-condição** | Usuário ter cadastro, estar logado no sistema e um livro não cadastrado |
| **Passos** | DADO que o usuário acessou a página de cadastro de livro do clube<br>QUANDO o campo *autor* não é preenchido<br>ENTÃO uma mensagem de alerta é exibida abaixo do campo<br>E o livro não é criado |
| **Resultado esperado** | 1. Não deve criar um livro sem dados<br>2. Deve exibir a mensagem de alerta no campo *autor* |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT058 |
| **Título** | Não deve criar um livro sem titulo e autor |
| **Tipo** | E2E |
| **Descrição** | Validar o fluxo de criação de novo livro e adição na estante do clube |
| **Funcionalidade** | Adicionar livro à estante do clube |
| **Pré-condição** | Usuário ter cadastro, estar logado no sistema e um livro não cadastrado |
| **Passos** | DADO que o usuário acessou a página de cadastro de livro do clube<br>QUANDO os campos *título* e *autor* não são preenchidos<br>ENTÃO as mensagens de alerta são exibidas abaixo dos campos<br>E o livro não é criado |
| **Resultado esperado** | 1. Não deve criar um livro sem dados<br>2. Deve exibir as mensagens de alerta nos campos *título* e *autor* |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT059 |
| **Título** | Não deve adicionar um livro criado sem escolher um status |
| **Tipo** | E2E |
| **Descrição** | Validar o fluxo de criação de novo livro e adição na estante do clube |
| **Funcionalidade** | Adicionar livro à estante do clube |
| **Pré-condição** | Usuário ter cadastro, estar logado no sistema e um livro não cadastrado |
| **Passos** | DADO que o usuário preencheu o cadastro do livro<br>QUANDO o campo *E como está o processo de leitura?* não é selecionado<br>ENTÃO uma mensagem de alerta é exibida abaixo do campo<br>E o livro não é criado |
| **Resultado esperado** | 1. Não deve criar um livro sem dados<br>2. Deve exibir a mensagem de alerta no campo *E como está o processo de leitura?* |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT060 |
| **Título** | Não deve adicionar um livro criado sem escolher uma avaliação |
| **Tipo** | E2E |
| **Descrição** | Validar o fluxo de criação de novo livro e adição na estante do clube |
| **Funcionalidade** | Adicionar livro à estante do clube |
| **Pré-condição** | Usuário ter cadastro, estar logado no sistema e um livro não cadastrado |
| **Passos** | DADO que o usuário preencheu o cadastro do livro<br>QUANDO o campo *O quanto você gostou?* não é selecionado<br>ENTÃO uma mensagem de alerta é exibida abaixo do campo<br>E o livro não é criado |
| **Resultado esperado** | 1. Não deve criar um livro sem dados<br>2. Deve exibir a mensagem de alerta no campo *O quanto você gostou?* |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT061 |
| **Título** | Não deve criar um livro com titulo com script injection |
| **Tipo** | E2E |
| **Descrição** | Validar o fluxo de criação de novo livro e adição na estante do clube |
| **Funcionalidade** | Adicionar livro à estante do clube |
| **Pré-condição** | Usuário ter cadastro, estar logado no sistema e um livro não cadastrado |
| **Passos** | DADO que o usuário acessou a página de cadastro de livro do clube<br>QUANDO o campo *título* é preenchido com script injection<br>ENTÃO uma mensagem de alerta é exibida abaixo do campo<br>E o livro não é criado |
| **Resultado esperado** | 1. Não deve criar um livro com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *título* |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT062 |
| **Título** | Não deve criar um livro com descrição com SQL injection |
| **Tipo** | E2E |
| **Descrição** | Validar o fluxo de criação de novo livro e adição na estante do clube |
| **Funcionalidade** | Adicionar livro à estante do clube |
| **Pré-condição** | Usuário ter cadastro, estar logado no sistema e um livro não cadastrado |
| **Passos** | DADO que o usuário acessou a página de cadastro de livro do clube<br>QUANDO o campo *descrição* é preenchido com SQL injection<br>ENTÃO uma mensagem de alerta é exibida abaixo do campo<br>E o livro não é criado |
| **Resultado esperado** | 1. Não deve criar um livro com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *descrição* |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT063 |
| **Título** | Não deve criar um livro com nome maior que o permitido |
| **Tipo** | E2E |
| **Descrição** | Validar o fluxo de criação de novo livro e adição na estante do clube |
| **Funcionalidade** | Adicionar livro à estante do clube |
| **Pré-condição** | Usuário ter cadastro, estar logado no sistema e um livro não cadastrado |
| **Passos** | DADO que o usuário acessou a página de cadastro de livro do clube<br>QUANDO o campo *título* é preenchido com mais de 255 caracteres<br>ENTÃO uma mensagem de alerta é exibida abaixo do campo<br>E o livro não é criado |
| **Resultado esperado** | 1. Não deve criar um livro com dados inconsistentes<br>2. Deve exibir a mensagem de alerta no campo *título* |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT064 |
| **Título** | Deve adicionar um livro na estante do clube como criador com sucesso |
| **Tipo** | E2E |
| **Descrição** | Validar o fluxo de adição de livro existente na estante do clube |
| **Funcionalidade** | Adicionar livro à estante do clube |
| **Pré-condição** | Usuário ter cadastro, estar logado no sistema como criador do clube e um livro já cadastrado |
| **Passos** | DADO que o usuário selecionou um livro na página de adição de livro na estante do clube<br>QUANDO os campos são selecionados<br>ENTÃO o livro é adicionado na estante do clube<br>E uma mensagem de sucesso é exibida na tela |
| **Resultado esperado** | 1. Deve adicionar o livro na estante do clube<br>2. Deve exibir uma mensagem de sucesso |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT065 |
| **Título** | Deve adicionar um livro na estante do clube como mediador com sucesso |
| **Tipo** | E2E |
| **Descrição** | |
| **Funcionalidade** | Adicionar livro à estante do clube |
| **Descrição** | Validar o fluxo de adição de livro existente na estante do clube |
| **Funcionalidade** | Adicionar livro à estante do clube |
| **Pré-condição** | Usuário ter cadastro, estar logado no sistema como mediador do clube e um livro já cadastrado |
| **Passos** | DADO que o usuário selecionou um livro na página de adição de livro na estante do clube<br>QUANDO os campos são selecionados<br>ENTÃO o livro é adicionado na estante do clube<br>E uma mensagem de sucesso é exibida na tela |
| **Resultado esperado** | 1. Deve adicionar o livro na estante do clube<br>2. Deve exibir uma mensagem de sucesso |
| **Status** | Falhou |

| | |
|---|---|
| **ID** | CT066 |
| **Título** | Não deve adicionar um livro na estante do clube sem escolher um status |
| **Tipo** | E2E |
| **Descrição** | Validar o fluxo de criação de novo livro e adição na estante do clube |
| **Funcionalidade** | Adicionar livro à estante do clube |
| **Pré-condição** | Usuário ter cadastro, estar logado no sistema e um livro cadastrado |
| **Passos** | DADO que o usuário selecionou um livro na página de adição de livro na estante do clube<br>QUANDO o campo *E como está o processo de leitura?* não é selecionado<br>ENTÃO uma mensagem de alerta é exibida abaixo do campo<br>E o livro não é criado |
| **Resultado esperado** | 1. Não deve criar um livro sem dados<br>2. Deve exibir a mensagem de alerta no campo *E como está o processo de leitura?* |
| **Status** | Sucesso |

| | |
|---|---|
| **ID** | CT067 |
| **Título** | Não deve adicionar um livro na estante do clube sem escolher uma avaliação |
| **Tipo** | E2E |
| **Descrição** | Validar o fluxo de criação de novo livro e adição na estante do clube |
| **Funcionalidade** | Adicionar livro à estante do clube |
| **Pré-condição** | Usuário ter cadastro, estar logado no sistema e um livro cadastrado |
| **Passos** | DADO que o usuário selecionou um livro na página de adição de livro na estante do clube<br>QUANDO o campo *O quanto você gostou?* não é selecionado<br>ENTÃO uma mensagem de alerta é exibida abaixo do campo<br>E o livro não é criado |
| **Resultado esperado** | 1. Não deve criar um livro sem dados<br>2. Deve exibir a mensagem de alerta no campo *O quanto você gostou?* |
| **Status** | Sucesso |