

## **RELATÓRIO FINAL DE TESTES**

Projeto Prático – Testes e Qualidade de Software

Sistema: TaskManager UNISAGRADO – Gerenciador de Tarefas

Integrantes do grupo:

- Victor Dionisio Momesso
- Fausto Renato
- Theo Rondon
- Kevin Lopes
- Pedro Rocha

### **1. INTRODUÇÃO**

Este relatório apresenta o resumo final das atividades de teste realizadas no sistema TaskManager UNISAGRADO, desenvolvido como parte do Projeto Prático da disciplina de Testes e Qualidade de Software.

O objetivo principal foi planejar, especificar, executar e automatizar testes sobre um sistema simples, porém completo, exercitando práticas de testes funcionais, não funcionais, automação, TDD e integração contínua.

### **2. VISÃO GERAL DO SISTEMA TESTADO (SUT)**

Nome do sistema: TaskManager UNISAGRADO

Tipo: Aplicação web simples para gerenciamento de tarefas acadêmicas.

Funcionalidades principais:

- Autenticação de usuário (login e logout).
- Cadastro de tarefas (título obrigatório, descrição opcional, data de vencimento opcional).
- Listagem de tarefas do usuário autenticado.
- Edição de tarefas existentes.
- Exclusão de tarefas com confirmação.
- Marcação de tarefas como concluídas, com destaque visual.
- Filtro de tarefas por status (Todas, Pendentes, Concluídas).
- Limites de negócio:
  - Título com no máximo 100 caracteres.
  - Até 100 tarefas ativas (pendentes) por usuário.
  - Mensagens claras de sucesso e erro nas operações.

Arquitetura resumida:

- Back-end: API REST em Python + Flask, com armazenamento em memória.
- Front-end: página web HTML/CSS/JavaScript consumindo a API Flask.
- Testes automatizados de API: Python + pytest + requests.

### 3. ESTRATÉGIA E ESCOPO DE TESTES

A estratégia de testes foi baseada nas seguintes diretrizes:

- Cobrir os requisitos funcionais RF01 a RF10 e não funcionais RNF01 e RNF02.
- Focar nos fluxos principais do usuário: login, criação de tarefa, listagem, conclusão, exclusão e logout.
- Produzir artefatos formais: plano de teste, matriz de rastreabilidade, casos de teste detalhados, automação mínima e relatórios.

- Realizar pelo menos dois ciclos de testes: ciclo 1 (descoberta de defeitos) e ciclo 2 (regressão após correções).
- Automatizar casos críticos (login e fluxo E2E), além de testes de API.

Tipos de teste aplicados:

- Testes funcionais (caixa-preta) com classes de equivalência, valores-limite e tabelas de decisão.
- Testes de integração/sistema, verificando o fluxo completo front-end + API.
- Testes de aceitação baseados em cenários Dado/Quando/Então.
- Teste não funcional de desempenho simples (tempo de resposta na listagem) e usabilidade (clareza de mensagens e rótulos).
- Testes automatizados de API com pytest/requests.

#### 4. REQUISITOS E CASOS DE TESTE

Os requisitos funcionais (RF01–RF10) e não funcionais (RNF01–RNF02) foram documentados no Plano de Teste, e associados a casos de teste na Matriz de Rastreabilidade.

Principais requisitos cobertos:

- RF01 – Login de usuário.
- RF02 – Logout.
- RF03 – Cadastro de tarefa.
- RF04 – Listagem de tarefas.
- RF05 – Edição de tarefa.
- RF06 – Exclusão de tarefa.
- RF07 – Marcar tarefa como concluída.
- RF08 – Filtro de tarefas por status.

- RF09 – Limite de título e quantidade de tarefas.
- RF10 – Mensagens de feedback.
- RNF01 – Usabilidade (clareza de mensagens e botões).
- RNF02 – Desempenho simples (listagem em até 2 segundos).

Os casos de teste foram detalhados em planilha específica, contendo para cada CT:

- ID (ex.: CT-LGIN-01, CT-TASK-01, CT-E2E-01).
- Requisitos relacionados.
- Objetivo.
- Pré-condições.
- Passos de execução.
- Dados de entrada.
- Resultado esperado.
- Técnica de teste utilizada (CE, VL, TD, NF, E2E).

Dentre os casos, destacam-se:

- CT-LGIN-01: Login com credenciais válidas.
- CT-LGIN-02: Login com senha inválida.
- CT-TASK-01 a CT-TASK-05: criação e validações de tarefas, incluindo limites de título e quantidade.
- CT-FILTER-01/02: filtros por status.
- CT-USAB-01: avaliação de usabilidade.
- CT-NF-01: teste de desempenho da listagem.
- CT-E2E-01: fluxo completo login → criar → concluir → excluir → logout.

## 5. EXECUÇÃO DOS TESTES

### 5.1. Ambiente de teste

- Sistema operacional: Windows 10.
- Back-end: Python 3.x, Flask em modo desenvolvimento (porta 3000).
- Front-end: execução local via navegador (Chrome).
- Ferramentas de apoio: navegador com DevTools, terminal, planilhas.

### 5.2. Ciclo 1 – Execução inicial

No primeiro ciclo foram executados os casos de teste manuais definidos como prioritários (login, cadastro, listagem, conclusão, exclusão, filtros e limites).

Durante o ciclo 1, foram observadas e registradas as seguintes situações (exemplo para preenchimento):

- Casos executados: todos os CTs planejados.
- Alguns comportamentos foram ajustados durante o desenvolvimento, especialmente mensagens de erro e validações de título.

Defeitos encontrados neste ciclo foram registrados em issues (por exemplo, mensagens pouco claras ou ausência de validação em algum campo). Após correções, os casos foram reexecutados.

### 5.3. Ciclo 2 – Regressão

No ciclo 2, foram reexecutados:

- Todos os casos que haviam falhado no ciclo 1.
- Fluxo E2E (CT-E2E-01).

- Casos principais de login e cadastro de tarefa.

Após as correções, os casos reexecutados passaram conforme o esperado, demonstrando estabilidade nas funcionalidades principais.

## 6. DEFEITOS REGISTRADOS (VISÃO GERAL)

Os defeitos identificados ao longo do projeto foram registrados em ferramenta de controle (por exemplo, GitHub Issues), contendo:

- ID do defeito (ex.: DEF-01, DEF-02).
- Título e descrição.
- Passos para reproduzir.
- Resultado esperado vs. obtido.
- Severidade e prioridade.
- Evidências (print da tela e/ou logs).

Principais tipos de defeitos tratados:

- Mensagens de validação ausentes ou pouco claras (ex.: título obrigatório).
- Ajustes em limites de campos (comprimento do título).
- Pequenos problemas de usabilidade (rótulos de botões e feedback visual).

Após as correções, os defeitos críticos foram retestados e encerrados.

## 7. AUTOMAÇÃO DE TESTES

Atendendo ao requisito de automação mínima, foi criada uma suíte de testes automatizados de API utilizando Python, pytest e requests, localizada na pasta Automacao.

Os testes automatizados implementados foram:

- test\_login\_valido
  - Relacionado ao caso CT-LOGIN-01.
  - Verifica login com credenciais válidas, retorno HTTP 200, token e mensagem de sucesso.
  
- test\_login\_invalido
  - Relacionado ao caso CT-LOGIN-02.
  - Verifica login com senha incorreta, retorno HTTP 401 e mensagem de erro adequada.
  
- test\_fluxo\_e2e\_completo
  - Relacionado ao caso CT-E2E-01.
  - Executa todo o fluxo: login → criar tarefa → listar → concluir → excluir → logout.
  - Garante que a tarefa criada aparece na lista, pode ser concluída e é removida ao final.

A execução é realizada com o comando:

```
python -m pytest -v test_taskmanager_api.py
```

O resultado obtido (3 testes PASSADOS) foi registrado em evidência (print de tela) e vinculado aos IDs de casos de teste correspondentes.

## 8. TDD E CI/CD

Para ilustrar TDD (Test-Driven Development), foi adotada a prática de primeiro escrever testes unitários para validação de regras simples de negócio, como validação de título de tarefa (campos obrigatórios e limites de caracteres), para depois implementar/refatorar o código de produção.

Exemplo de ciclo TDD aplicado:

- Escrever um teste para garantir que títulos vazios sejam rejeitados.
- Executar o teste e observar falha (vermelho).
- Implementar a validação no back-end Flask.
- Reexecutar o teste até obter sucesso (verde).
- Refatorar mantendo os testes passando.

## 9. MÉTRICAS DE TESTE

As métricas foram obtidas a partir das planilhas de casos de teste, matriz de rastreabilidade e registros de execução.

Exemplos de métricas que podem ser apresentadas (valores a serem ajustados conforme os dados finais do grupo):

- Cobertura de requisitos:

- 100% dos requisitos funcionais (RF01–RF10) possuem pelo menos um caso de teste associado.
- 100% dos requisitos não funcionais (RNF01–RNF02) possuem pelo menos um caso de teste.

- Cobertura de casos de teste:

- Casos planejados: total de CTs definidos na planilha.
- Casos executados no ciclo 1: todos os CTs planejados.

- Casos executados no ciclo 2: CTs críticos e casos que falharam no ciclo 1.

- Resultados dos testes (exemplo):

- Ciclo 1: X% Passou, Y% Falhou.

- Ciclo 2: Z% Passou, 0% Crítico em aberto.

- Defeitos:

- Total de defeitos encontrados.

- Defeitos por severidade (Crítico, Alto, Médio, Baixo).

- Defeitos reabertos (se houver).

Essas informações podem ser extraídas diretamente das planilhas de execução e do histórico de Issues do projeto.

## 10. RISCOS E RECOMENDAÇÕES

Riscos remanescentes e pontos de atenção:

- Possível aumento de carga (mais usuários e tarefas) pode exigir testes de desempenho mais avançados e uso de banco de dados persistente em vez de memória.
- Melhorias futuras de usabilidade, como responsividade em dispositivos móveis e acessibilidade.
- Necessidade de ampliar a automação para a interface gráfica (UI), cobrindo mais cenários.

Recomendações:

- Evoluir a solução de back-end para uso de banco de dados relacional ou NoSQL, com migrações e scripts de carga de dados.
- Ampliar a suíte de testes automatizados, incluindo mais casos de negócios e testes de regressão.

- Integrar testes de interface (por exemplo, Cypress) ao pipeline de CI/CD.
- Documentar continuamente novos requisitos e manter a matriz de rastreabilidade atualizada.

## 11. CONCLUSÃO

O projeto TaskManager UNISAGRADO permitiu aplicar na prática conceitos fundamentais de Testes e Qualidade de Software, desde a definição de requisitos e casos de teste até a execução manual, registro de defeitos e automação básica com pytest.

Os principais objetivos do projeto foram alcançados:

- Todos os requisitos definidos foram mapeados em casos de teste e rastreados na matriz.
- Os testes funcionais garantiram o correto funcionamento dos fluxos principais do sistema.
- Os testes não funcionais abordaram usabilidade e desempenho simples.
- A automação de API demonstrou a viabilidade de reexecução rápida de cenários críticos (login e fluxo E2E).
- Foram exercitados conceitos de TDD e configurada uma base para integração contínua.

Como resultado, o grupo pôde vivenciar o ciclo completo de testes em um sistema real, compreendendo a importância de planejar, executar, medir e automatizar testes para garantir a qualidade de software.