Plano de Teste

OnCatalog
versão 1.1

Histórico das alterações

Data	Versão	Descrição	Autor(a)
22/07/2024	1.0	Release Inicial	Cleanio
15/09/2024	1.1	Correções de bugs e melhorias de performance	Cleanio, Silvio

1 - Introdução

Objetivo

Este plano de teste tem como principal objetivo avaliar a qualidade global do sistema OnCatalog em todas as suas fases de desenvolvimento, garantindo que o software entregue atenda aos requisitos funcionais, de segurança e desempenho esperados, e esteja preparado para escalar conforme a demanda das pequenas empresas. Além disso, visa identificar e corrigir falhas de usabilidade e integridade de dados, assegurando uma experiência consistente e eficiente tanto para os administradores (pequenas empresas) quanto para os usuários finais (clientes). Esse plano detalha os tipos de testes que serão executados (funcionais, de integração, de segurança, de usabilidade, de estresse e regressão), os critérios de aceitação de cada requisito, e as métricas para a avaliação de performance.

Através deste documento, será possível monitorar a evolução dos testes, realizar ajustes conforme o feedback contínuo, garantir conformidade com os objetivos de negócios e reduzir os riscos de falhas na produção. Além de proporcionar a melhoria contínua do sistema, o plano também auxilia na tomada de decisões estratégicas para priorizar correções e otimizações. O projeto OnCatalog, desenvolvido com o framework Django em Python, será validado em sua capacidade de oferecer uma plataforma confiável e acessível para micro e pequenas empresas (MEs e EPPs) criarem catálogos virtuais e gerenciarem suas interações com clientes de forma simples e eficiente, com foco em escalabilidade e integração com canais de comunicação populares, como o WhatsApp.

2 - Requisitos a Testar

Os principais requisitos a serem avaliados para a primeira versão em produção da aplicação incluem:

- **Autenticação e Registro**: O sistema deve permitir o cadastro, login.
- **Criação e Gerenciamento de Produtos**: O sistema deve permitir que empresas adicionem, editem e excluam produtos de seus catálogos.
- **Acessibilidade do Catálogo**: O catálogo deve estar acessível publicamente.
- **Sistema de Contato com a Empresa**: Os clientes devem ser capazes de entrar em contato com as empresas tanto via mensagem direta quanto por WhatsApp.
- **Integração com Banco de Dados**: Validação do armazenamento correto dos produtos e usuários no banco de dados.

3 - Tipos de teste

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

- **Testes de Unidade**: Focar na validação de cada função e método de forma isolada.

- **Testes de Integração**: Garantir que os módulos funcionem corretamente quando integrados, especialmente o cadastro, criação de catálogos e sistema de contato.
- **Testes de Sistema**: Validar o comportamento do sistema completo, garantindo que ele funcione conforme as expectativas dos usuários finais.
- **Testes de Regressão**: Garantir que novas funcionalidades não introduzem falhas em partes já testadas.
- **Testes de carga/estresse:** Avaliar a robustez e a capacidade do sistema sob condições extremas de carga, identificando os limites de desempenho e garantindo a estabilidade.

3.1 - Métodos da Classe

Para garantir a correta funcionalidade de cada classe, os métodos devem ser testados individualmente, verificando se estão retornando os resultados esperados. Se possível, recomenda-se a utilização de testes automatizados (incluindo uso de ferramentas CI/CD) para agilizar o processo e aumentar a confiabilidade do sistema. Seguem algumas práticas recomendadas para testar os métodos das classes:

- **Verificação de retorno esperado**: Cada método deve ser testado para garantir que retorna o valor correto com base em diferentes entradas de dados. Isso pode incluir testes com valores normais, limites, e valores inesperados (para testar a robustez do código).
- **Testes de exceção**: Além de verificar o retorno correto dos métodos, deve-se testar se eles estão lidando corretamente com exceções e erros, como entradas inválidas ou operações que podem falhar.
- **Teste de integração entre métodos**: Quando métodos de diferentes classes ou de uma mesma classe se integram, é importante verificar se essa interação ocorre corretamente, garantindo que o comportamento geral do sistema seja o esperado.
- **Automatização com frameworks**: Ferramentas como **PyTest** (para Python).
- **Cobertura de código**: Avaliar a cobertura de código pode ajudar a garantir que todos os caminhos do código (incluindo loops e condições) foram testados. Isso garante que os testes estão cobrindo o máximo possível de cenários.

3.2 - Persistência de Dados

O teste de persistência de dados é crucial para garantir a integridade e a confiabilidade das informações no sistema, especialmente em cenários onde há falha ou desligamento inesperado. Para realizar esses testes, alguns pontos precisam ser observados:

- **Teste de integridade de dados**: Verificar se os dados são gravados corretamente no banco de dados e permanecem consistentes mesmo após o desligamento do programa. O objetivo é garantir que as operações de inserção, atualização, e exclusão não causem perda ou corrupção de dados.

- **Recuperação após falha**: Testar se o sistema consegue se recuperar após falhas inesperadas, como queda de energia ou fechamento abrupto da aplicação. Isso pode ser feito simulando falhas durante operações críticas, como a gravação de dados, para garantir que o banco de dados permaneça em um estado consistente ou que mecanismos de rollback sejam aplicados corretamente.
- **Transações e atomicidade**: Se o sistema usa transações, é importante verificar se elas seguem o conceito de atomicidade (ACID). Se uma operação de transação falhar, o banco de dados deve ser capaz de desfazer todas as mudanças feitas durante a transação, garantindo que o sistema não permaneça em um estado inconsistente.
- **Teste de backup e restauração**: Garantir que os mecanismos de backup e restauração funcionem corretamente para prevenir perda de dados em casos de falhas severas. O backup deve ser testado regularmente, e a restauração deve ser verificada para garantir que os dados retornem ao estado correto.
- **Automatização dos testes**: Ferramentas como **PyTest** com bibliotecas de banco de dados (para Python), ou frameworks de teste de banco de dados específicos podem ser usadas para automatizar esses processos. Testes automatizados podem incluir:
    - Inserção de dados, seguida de fechamento forçado do programa, e verificação se os dados foram persistidos corretamente.
    - Simulação de falhas durante transações e verificação do comportamento de rollback.
    - Teste de restauração após backup simulado.

3.3 - Integração dos Componentes

Os testes de **integração de componentes** são essenciais para garantir que as diversas partes do sistema (classes, métodos e módulos) funcionem corretamente quando combinadas. O objetivo é verificar se as interações entre os componentes individuais do programa ocorrem de forma correta e sem erros durante uma sequência de ações. Para realizar esses testes, considere os seguintes pontos:

- **Fluxo de ações do programa**: Testar se a sequência de ações que envolve diferentes partes do sistema (métodos, classes e componentes) é executada corretamente. O sistema deve ser capaz de executar fluxos completos sem interrupções ou inconsistências, verificando desde a interação de interface com o usuário até o backend e persistência de dados.
- **Interdependência de métodos e classes**: Verificar se os métodos de diferentes classes estão se comunicando e se integrando conforme o esperado. A interação entre componentes, como chamadas de métodos entre objetos, deve ser testada em diversos cenários.
- **Verificação de dados entre camadas**: Em sistemas com várias camadas, como camada de interface, camada de lógica de negócios e camada de persistência, é importante testar se as informações fluem corretamente entre essas camadas. Testar,

por exemplo, se os dados coletados pela interface são processados corretamente pela lógica de negócios e persistidos no banco de dados.

- **Simulação de cenários reais**: Criar cenários que simulam o uso real do sistema para verificar como os componentes se comportam. Isso pode incluir a execução de fluxos de usuário completos, como cadastro, login, operações em um banco de dados e exibição de informações processadas.
- **Testes de interface de comunicação**: Em sistemas distribuídos ou com múltiplas APIs, deve-se testar as interfaces de comunicação entre os módulos ou serviços, assegurando que eles troquem dados corretamente.
- **Automatização dos testes**: Frameworks como **Selenium** (para testes de integração de frontend e backend) e/ou **PyTest** (para integração de métodos e classes no backend) podem ser usados para criar testes automatizados. Esses testes podem simular interações reais, verificando como os componentes funcionam em conjunto.
- **Teste de dependências externas**: Se o sistema depende de APIs ou serviços externos, é importante verificar se as integrações com esses serviços funcionam corretamente. Testes com mockups ou simuladores dessas dependências podem garantir que, mesmo em ambientes controlados, o sistema reage bem às respostas externas.
- **Manutenção de consistência**: Durante os testes de integração, deve-se verificar se o estado dos objetos e dados é mantido consistentemente entre as interações. Por exemplo, ao passar um objeto entre diferentes métodos ou classes, ele deve manter seus atributos e valores corretamente.

3.4 - Testes funcionais

Os **testes funcionais** têm o objetivo de validar se o sistema atende aos requisitos funcionais, ou seja, se cada funcionalidade está operando conforme o especificado. Esses testes focam nas ações que o sistema deve executar e são realizadas sob a perspectiva do usuário final, garantindo que cada função do sistema funcione corretamente e forneça os resultados esperados. Aqui estão os principais pontos para realizar testes funcionais:

- **Validação de requisitos**: O principal objetivo é garantir que todas as funcionalidades descritas nos requisitos do sistema estejam implementadas corretamente. Cada função (como cadastro, login, inserção de dados, etc.) deve ser testada de forma individual e em conjunto.
- **Fluxo de trabalho completo**: Simular cenários reais de uso, onde o usuário realiza várias ações sequenciais, como navegar entre páginas, preencher formulários e submeter dados. O sistema deve ser capaz de conduzir esses fluxos de trabalho sem falhas.
- **Testes de fronteira e casos extremos**: Verificar se o sistema lida corretamente com entradas nos limites, como campos com valores mínimos e máximos permitidos, ou até mesmo entradas inesperadas, como valores nulos ou incorretos.
- **Comportamento esperado em diferentes entradas**: Testar o comportamento do sistema com diferentes tipos de entrada. Verificar se as validações e restrições

funcionam corretamente, como impedir o envio de formulários incompletos ou com dados inválidos.

- **Interação com o usuário**: Garantir que a interface do usuário (UI) seja intuitiva e responda corretamente às ações dos usuários. Isso inclui verificar se os botões e links realizam as funções esperadas, se as mensagens de erro ou sucesso são exibidas corretamente e se a navegação está clara.
- **Verificação de usabilidade**: Embora os testes funcionais sejam mais focados no comportamento da aplicação, é importante garantir que a experiência do usuário não seja comprometida. Aspectos como tempo de resposta, acessibilidade e clareza das funcionalidades podem ser validados.
- **Automatização de testes funcionais**: Ferramentas de teste automatizado, como **Selenium** para interfaces web.
- **Testes em diferentes ambientes**: Verificar o comportamento do sistema em diferentes navegadores.
- **Integração com testes de regressão**: Sempre que uma nova funcionalidade for adicionada, os testes funcionais existentes devem ser reexecutados para garantir que as funcionalidades antigas ainda funcionem como esperado. Isso pode ser feito através de testes de regressão automatizados.
- **Testes de casos de uso críticos**: Priorizar testes em áreas críticas do sistema, como processo de pagamento, segurança de login, ou funcionalidades essenciais para o negócio, assegurando que falhas nessas áreas sejam evitadas.

3.5 - Testes de usabilidade

Os **testes de usabilidade** são realizados para avaliar a facilidade de uso e a qualidade da experiência do usuário (UX) ao interagir com o sistema. Eles garantem que o sistema seja intuitivo, eficiente e que os usuários consigam realizar suas tarefas sem frustrações. Esses testes focam principalmente na interface do usuário (UI) e na interação, assegurando que o design e a funcionalidade estejam alinhados com as expectativas do público-alvo. Aqui estão os principais aspectos para conduzir testes de usabilidade:

- **Facilidade de aprendizado**: Avaliar o tempo e o esforço necessários para que novos usuários entendam e utilizem o sistema de forma eficaz. A interface deve ser intuitiva, com uma curva de aprendizado mínima, permitindo que o usuário comece a utilizar o sistema sem precisar de instruções complexas.
- **Eficiência na execução de tarefas**: Testar a rapidez e a fluidez com que os usuários conseguem realizar as ações propostas no sistema. As funcionalidades principais devem ser acessíveis de forma direta e sem obstáculos desnecessários. A navegação entre páginas e menus deve ser simples e rápida.
- **Satisfação do usuário**: Medir o nível de satisfação do usuário ao utilizar o sistema. Isso pode ser feito por meio de entrevistas ou questionários após os testes, onde os usuários compartilham suas impressões sobre a facilidade de uso, estética da interface e conveniência.

- **Testes com usuários reais**: Realizar testes com usuários reais, que representam o público-alvo do sistema, para observar como eles interagem com a interface. Durante esses testes, deve-se observar:
    - A facilidade de encontrar funcionalidades importantes.
    - Se o usuário consegue completar tarefas sem ajuda.
    - Onde ele encontra dificuldades ou frustrações.
    - O tempo necessário para completar determinadas ações.
- **Fluxo de navegação**: Verificar se o fluxo de navegação é claro e coerente. O usuário deve ser capaz de ir de uma funcionalidade a outra sem se perder. Verificar se os botões, links e menus estão posicionados de forma lógica e fácil de acessar.
- **Feedback visual e sonoro**: Testar se o sistema oferece feedback adequado ao usuário. Por exemplo, após clicar em um botão, deve haver uma resposta visual clara (como uma animação ou mudança de cor) ou sonora para indicar que a ação foi recebida e está sendo processada.
- **Design responsivo e adaptabilidade**: Testar a interface em diferentes dispositivos e resoluções de tela (computadores, tablets, smartphones) para garantir que o design seja responsivo e que a experiência de uso não seja prejudicada em telas menores ou em dispositivos móveis.
- **Acessibilidade**: Verificar se o sistema é acessível para usuários com deficiências, utilizando ferramentas de leitura de tela, teclados, e se a interface tem um contraste adequado entre texto e fundo, botões de tamanho apropriado, e outras boas práticas de acessibilidade (de acordo com normas como as **WCAG** - Web Content Accessibility Guidelines).
- **Detecção de erros de usabilidade**: Identificar áreas do sistema onde os usuários podem cometer erros e se o sistema lida bem com esses erros. Isso inclui testar como o sistema informa o usuário sobre entradas incorretas e se oferece maneiras claras de corrigir o problema.
- **Automatização de testes de usabilidade**: Embora muitos testes de usabilidade sejam melhor realizados manualmente com usuários reais, ferramentas de análise de usabilidade automatizada, como **Google Lighthouse**, podem ser usadas para avaliar aspectos técnicos como tempos de carregamento, acessibilidade e responsividade.
- **Testes de cenários reais**: Criar cenários que simulam o uso real do sistema em diversas situações, observando como os usuários interagem com a interface. Isso ajuda a identificar barreiras e melhorias na usabilidade que podem não ser detectadas em testes isolados.

3.5 - Testes de carga/estresse

Os testes de estresse têm como objetivo avaliar como o sistema se comporta sob condições extremas de uso, como picos de tráfego, grandes volumes de dados ou processamento intensivo. Esse tipo de teste ajuda a identificar os limites do sistema, revelando falhas, gargalos de desempenho e comportamentos inesperados que podem ocorrer quando a carga ou a demanda excede o que é considerado normal. Aqui estão os principais aspectos de testes de estresse:

- **Carga extrema de usuários**: Testar o sistema com um número muito maior de usuários simultâneos do que o esperado normalmente. Isso ajuda a determinar até que ponto o sistema consegue suportar o tráfego antes de apresentar degradação de desempenho ou falhas.
- **Volume de dados**: Enviar grandes quantidades de dados para o sistema, simulando cenários onde ele precisa processar ou armazenar um volume excessivo de informações, como grandes uploads de arquivos, consultas complexas em bancos de dados ou processamento massivo de dados..
- **Simulação de falhas de infraestrutura**: Testar como o sistema reage quando componentes críticos, como servidores, banco de dados ou serviços de rede, falham ou se tornam temporariamente inacessíveis. O objetivo é garantir que o sistema seja resiliente e consiga se recuperar de falhas sem perda de dados ou interrupção completa do serviço.
- **Degradação controlada**: Observar em que ponto o desempenho começa a degradar sob carga extrema e se essa degradação é gradual ou abrupta. O sistema deve ser capaz de diminuir a performance de maneira controlada, em vez de simplesmente travar ou se tornar inacessível.
- **Tempo de resposta sob estresse**: Medir o tempo de resposta do sistema enquanto ele está sob estresse. Mesmo sob uma alta carga, o sistema deve responder aos usuários em tempo aceitável. Esse teste é importante para garantir que a experiência do usuário não seja drasticamente afetada.
- **Picos de tráfego**: Simular picos repentinos de tráfego, onde o sistema precisa lidar com uma grande quantidade de requisições em um curto período. Isso é importante para entender como o sistema se adapta a mudanças abruptas na demanda, como durante promoções ou eventos de alto tráfego.
- **Monitoramento de logs e métricas**: Durante os testes de estresse, monitorar os logs do sistema, uso de CPU, memória e disco, além de métricas de rede, como latência e throughput, para identificar pontos críticos e áreas que precisam ser otimizadas.
- **Automatização dos testes de estresse**: Ferramentas como **Apache JMeter** ou **K6** podem ser usadas para simular grandes volumes de tráfego e testar a resistência do sistema sob carga. Essas ferramentas permitem configurar cenários complexos de estresse e fornecem relatórios detalhados sobre o desempenho.
- **Testes em ambientes reais**: É importante realizar os testes de estresse em ambientes que simulem o mais próximo possível o ambiente de produção, para garantir que as condições testadas sejam realistas. Se possível, testes também podem ser realizados no ambiente de produção durante períodos de menor tráfego (com controles adequados).

4 - Recursos

4.1 Ferramentas de Teste

Para garantir a eficiência e eficácia dos testes do OnCatalog, serão empregadas ferramentas de automação e gerenciamento de testes para diferentes tipos de testes.Duas

ferramentas amplamente utilizadas para testes de software são o **Pytest** e o **SonarQube** Elas têm objetivos diferentes, mas são complementares em um ambiente de desenvolvimento ágil e orientado a testes. Abaixo, vamos falar sobre cada uma delas.

O **Pytest** é uma das bibliotecas mais populares em Python para a criação e execução de testes automatizados. Ele é usado principalmente para escrever testes unitários, de integração e funcionais, permitindo que os desenvolvedores verifiquem se suas funções e módulos estão funcionando conforme o esperado.

- **Principais características do Pytest**:
    - **Simples de usar**: A curva de aprendizado do Pytest é bastante suave, pois ele facilita a escrita de testes simples com pouca configuração. O desenvolvedor pode criar um arquivo de teste e usar uma simples assertiva (assert) para verificar o comportamento esperado.
    - **Testes legíveis**: Pytest promove a escrita de testes legíveis e fáceis de entender, tornando o processo de depuração mais eficiente.
    - **Suporte a fixtures**: O Pytest oferece o conceito de **fixtures**, que são funções que preparam o ambiente de teste (como conectar a um banco de dados, criar arquivos temporários, etc.). Fixtures podem ser reutilizadas em múltiplos testes, facilitando a manutenção do código de teste.
    - **Compatibilidade com outras ferramentas**: Pytest pode ser facilmente integrado com outras ferramentas de teste, como bibliotecas de "mocking" e frameworks de comportamento (BDD), como **pytest-bdd**.
    - **Suporte a parametrização**: Ele permite que os testes sejam parametrizados, ou seja, o mesmo teste pode ser executado com várias entradas, facilitando a validação de uma gama mais ampla de casos.
    - **Plugins**: Pytest tem um ecossistema rico de plugins que estendem sua funcionalidade, permitindo integrar testes com ferramentas de CI/CD, gerar relatórios de cobertura de código, entre outros.
    - **Execução seletiva de testes**: Pytest permite executar testes específicos ou subconjuntos de testes com base em nomes de arquivos, funções ou tags, facilitando a execução de testes prioritários ou durante processos de depuração.

O **SonarQube**, comumente chamado de **Sonar**, é uma plataforma de código aberto usada para inspeção contínua da qualidade do código. Ele realiza análises estáticas no código para detectar problemas como bugs, vulnerabilidades, código duplicado, e manter o código alinhado com padrões de qualidade.

- **Principais características do Sonar**:
    - **Análise estática de código**: O Sonar realiza uma análise automática do código-fonte para detectar uma ampla variedade de problemas de qualidade, como bugs, vulnerabilidades de segurança, erros potenciais e violações de boas práticas de desenvolvimento.

- **Relatórios de qualidade**: O Sonar gera relatórios detalhados de qualidade do código, mostrando métricas como cobertura de testes, complexidade ciclomática, duplicação de código, e qualidade geral do código.
- **Cobertura de testes**: Integrado com ferramentas de teste como Pytest, o Sonar pode coletar dados de cobertura de testes, ajudando a garantir que o código seja bem testado e a identificar áreas que precisam de mais testes.
- **Monitoramento contínuo**: Sonar pode ser integrado a sistemas de integração contínua (CI/CD) como Jenkins, GitLab CI, ou TravisCI. Isso permite que as análises de qualidade sejam realizadas automaticamente sempre que o código for atualizado, prevenindo a introdução de problemas antes que eles cheguem à produção.
- **Regras personalizadas**: O Sonar permite que as regras de análise sejam personalizadas de acordo com as necessidades do projeto. Isso significa que você pode ajustar os padrões de qualidade e tolerâncias para refletir as prioridades da equipe.
- **Integração com ferramentas de segurança**: O Sonar também pode ser integrado com ferramentas específicas de segurança para detectar vulnerabilidades de segurança no código.
- **Dashboard intuitivo**: O Sonar oferece uma interface web onde é possível visualizar o status de qualidade do código com gráficos e relatórios detalhados, facilitando a visualização de métricas e o acompanhamento de melhorias ao longo do tempo.

5 - Cronograma

Tipo de teste	Duração	Data de início	Data de término
Planejar teste	1 semana	22/07/2024	25/07/2024
Projetar teste	1 semana	26/07/2024	30/07/2024
Implementar teste	contínua	12/08/2024	31/11/2024
Executar teste	contínua	12/08/2024	31/11/2024
Avaliar teste	contínua	20/08/2024	31/11/2024