

RAFAEL CARVALHO FRANÇA

**TESTES AUTOMATIZADOS EM SALESFORCE: UMA VISÃO
PRÁTICA DE COMO AUTOMATIZAR TESTES COM PYTHON,
SELENIUM E PYTEST**

CAMPINAS
2024

RAFAEL CARVALHO FRANÇA

**TESTES AUTOMATIZADOS EM SALESFORCE: UMA VISÃO PRÁTICA DE COMO
AUTOMATIZAR TESTES COM PYTHON, SELENIUM E PYTEST**

Trabalho de Conclusão de Curso apresentado
como parte dos requisitos para obtenção do
diploma do Curso de Tecnologia em Análise e
Desenvolvimento de Sistemas do Instituto
Federal de Educação, Ciência e Tecnologia
Campus Campinas.

Orientador: Profa. Zady Salazar

CAMPINAS
2024

Ficha Catalográfica
Instituto Federal de São Paulo – Campus Campinas
Biblioteca
Tatiane Salles – CRB 8/8946

França, Rafael Carvalho

F815t Testes automatizadas em salesforce: uma visão prática de como automatizar testes com Python, selenium e pytest / Rafael Carvalho França. – Campinas, SP: [s.n.], 2024.

54 f. : il.

Bibliografia: 44-46

Orientadora: Dra. Zady Castaneda Salazar

Trabalho de Conclusão de Curso (graduação) – Instituto Federal de Educação, Ciência e Tecnologia de São Paulo Campus Campinas. Curso de Tecnologia em Análise e Desenvolvimento de Sistemas, 2024.

1. Software - testes. 2. Python (Linguagem de programação de computador). 3. Framework (Arquivo de computador). 4. Software de aplicação - Desenvolvimento. I. Instituto Federal de Educação, Ciência e Tecnologia de São Paulo Campus Campinas, Curso de Análise e Desenvolvimento de Sistemas. II. Título.

CDD: 004

ATA N.º 15/2024 - TADS-CMP/DAE-CMP/DRG/CMP/IFSP

Ata de Defesa de Trabalho de Conclusão de Curso - TADS

Na presente data, realizou-se a sessão pública de defesa do Trabalho de Conclusão de Curso intitulado TESTES AUTOMATIZADOS EM SALESFORCE: UMA VISÃO PRÁTICA DE COMO AUTOMATIZAR TESTES COM PYTHON, SELENIUM E PYTEST, apresentado(a) pelo(a) aluno(a) RAFAEL CARVALHO FRANÇA (CP3013766) do CURSO DE TECNÓLOGO EM ANÁLISE DE SISTEMAS (campus Campinas). Os trabalhos foram iniciados às 16h:30 pelo(a) Professor(a) presidente da banca examinadora, constituída pelos seguintes membros:

Membros	Instituição	Presença (Sim/Não)
Zady Castaneda Salazar (Presidente/Orientador)	IFSP	sim
Cecilia Sosa Arias Peixoto (Examinador 1)	IFSP	sim
José Américo S Mendonça(Examinador 2)	IFSP	sim

Observações:

A banca examinadora, tendo terminado a apresentação do conteúdo da monografia, passou à arguição do(a) candidato(a). Em seguida, os examinadores reuniram-se para avaliação e deram o parecer final sobre o trabalho apresentado pelo(a) aluno(a), tendo sido atribuído o seguinte resultado:

☒ Aprovado(a) ☐ Reprovado(a)

Proclamados os resultados pelo presidente da banca examinadora, foram encerrados os trabalhos e, para constar, eu lavrei a presente ata que assino em nome dos demais membros da banca examinadora.

IFSP Campus Campinas - 02/12/2024

Documento assinado eletronicamente por:

- **Zady Castaneda Salazar**, PROFESSOR ENS BASICO TECN TECNOLOGICO, em 02/12/2024 17:21:30.
- **Jose Americo dos Santos Mendonca**, PROFESSOR ENS BASICO TECN TECNOLOGICO, em 02/12/2024 17:22:53.
- **Cecilia Sosa Arias Peixoto**, PROF ENS BAS TEC TECNOLOGICO-SUBSTITUTO, em 02/12/2024 17:23:14.

Este documento foi emitido pelo SUAP em 02/12/2024. Para comprovar sua autenticidade, faça a leitura do QRCode ao lado ou acesse <https://suap.ifsp.edu.br/autenticar-documento/> e forneça os dados abaixo:

Código Verificador: 852722
Código de Autenticação: a1543f3f99



Dedico este trabalho aos meus pais, pelo apoio incondicional e por sempre estarem ao meu lado durante todo o período do curso, oferecendo amor, compreensão e força para que eu pudesse seguir em frente. Agradeço, também, à minha namorada e futura esposa, que com seu carinho e incentivo, me motivou a não desistir, mesmo nos momentos mais difíceis.

AGRADECIMENTOS

Gostaria de expressar minha gratidão a todos os professores e colaboradores do IFSP - Campus Campinas, que tiveram um papel fundamental na minha formação acadêmica e pessoal. Em particular, quero agradecer à minha orientadora Zady Salazar pela confiança e pela liberdade concedida durante a elaboração desta monografia, além de ser a professora das matérias que me fizeram ter o interesse pessoal e profissional pela área de Qualidade de Software.

RESUMO

O teste de software é uma etapa crucial no desenvolvimento de sistemas, com o objetivo de garantir que os produtos atendam aos requisitos especificados e funcionem conforme esperado. Contudo, a crescente complexidade das aplicações, torna os testes manuais cada vez mais demorados e suscetíveis a erros. Nesse contexto, a automação de testes surge como uma solução eficiente, permitindo a execução rápida, consistente e repetitiva de testes, economizando tempo e recursos. Este trabalho investigou a automação de casos de teste funcionais no sistema CRM Salesforce, com a criação e execução de dez testes cobrindo integralmente o fluxo de usuário em três objetos distintos. Os scripts de teste automatizados foram desenvolvidos utilizando a linguagem de programação Python, com o auxílio do Selenium WebDriver, e os testes foram realizados no navegador Google Chrome, com a geração automática de relatórios HTML contendo o tempo total e individual de execução de cada teste, e se passaram ou não. A adoção do padrão Page Object Model demonstrou ser altamente eficaz, pois possibilitou a criação de testes escaláveis, bem estruturados e de fácil manutenção. Aliado a isso, o uso do framework Pytest, com suas funcionalidades avançadas, contribuiu significativamente para aumentar a eficiência, facilitar a execução tanto em suítes quanto individualmente, e melhorar a organização dos testes automatizados.

Palavras-chave: teste de software; automação de testes; Salesforce; Selenium; Pytest.

ABSTRACT

Software testing is a crucial stage in system development, aiming to ensure that products meet specified requirements and function as expected. However, the increasing complexity of applications makes manual testing more time-consuming and error-prone. In this context, test automation emerges as an efficient solution, enabling the rapid, consistent, and repetitive execution of tests, saving time and resources. This study investigated the automation of functional test cases in the Salesforce CRM system by creating and executing ten tests that fully covered the user flow in three distinct objects. The automated test scripts were developed using the Python programming language, with the help of Selenium WebDriver, and the tests were executed in the Google Chrome browser. Automated HTML reports were generated, detailing the total and individual execution times of each test and indicating whether the tests passed or failed. The adoption of the Page Object Model pattern proved to be highly effective, enabling the creation of scalable, well-structured, and easily maintainable tests. Additionally, the use of the Pytest framework, with its advanced functionalities, significantly contributed to enhancing efficiency, simplifying execution in both suites and individual tests, and improving the organization of automated tests.

Keywords: software testing; test automation; Salesforce; Selenium; Pytest.

LISTA DE FIGURAS

Figura 1 – Estrutura do cenário utilizando Gherkin.....	25
Figura 2 – Tela inicial do Salesforce.....	29
Figura 4 – Estrutura de pastas da aplicação na IDE.....	30
Figura 5 – Arquivo ct03_create_account.feature.....	32
Figura 6 – Algumas funções da página base.....	32
Figura 7 – Chamada da função na página de contas (account_page.py).....	33
Figura 8 – Localizador XPath utilizado para verificar a presença do título na página.....	34
Figura 9 – Página da web com o elemento localizado (destacado em verde).....	34
Figura 10 – Arquivo ct03_create_account.py.....	35
Figura 11 – Comando de execução dos scripts de teste e do relatório incluso.....	38
Figura 12 – Relatório em HTML gerado após a execução dos dez casos de testes.....	39

LISTA DE GRÁFICOS

Gráfico 1 – Comparação dos tempos de teste na execução manual e automatizada em segundos.....	40
Gráfico 2 – Proporção do tempo total gasto em testes automatizados versus testes manuais..	41

LISTA DE QUADROS

Quadro 1 – As oito estratégias diferentes de localização de elementos embutidas no Selenium WebDriver.....	21
Quadro 2 – Ferramentas e bibliotecas utilizadas.....	28
Quadro 3 – Estrutura de pastas da aplicação.....	30
Quadro 4 – Scripts de teste, BDDs correspondentes e marcadores.....	36

LISTA DE SIGLAS

BDD	<i>Behavior Driven Development</i> (Desenvolvimento Orientado a Comportamento)
CRM	<i>Customer Relationship Management</i> (Gestão de Relacionamento com o Cliente)
DOM	<i>Document Object Model</i> (Modelo de Objeto de Documento)
E2E	<i>End-to-end</i> (Ponta a ponta)
HTML	<i>Hypertext Markup Language</i> (Linguagem de Marcação de Hipertexto)
IDE	<i>Integrated Development Environment</i> (Ambiente de Desenvolvimento Integrado)
Org	Organização
PO	Product Owner (Proprietário do Produto)
POM	<i>Page Object Model</i> (Modelo de Objeto de Página)
QA	<i>Quality Assurance</i> (Garantia da Qualidade)
TDD	<i>Test-Driven Development</i> (Desenvolvimento Orientado a Testes)

SUMÁRIO

1 INTRODUÇÃO.....	13
2 JUSTIFICATIVA.....	15
3 OBJETIVOS.....	17
3.1 Objetivo geral.....	17
3.2 Objetivos específicos.....	17
4 FUNDAMENTAÇÃO TEÓRICA.....	18
4.1 Teste de Software.....	18
4.2 Teste Funcional.....	18
4.3 Testes Automatizados.....	18
4.4 Selenium WebDriver.....	19
4.5 Python.....	21
4.6 Salesforce.....	22
4.7 POM.....	23
4.8 Aqua IDE.....	23
4.9 Behavior-Driven Development (BDD).....	25
4.10 Pytest.....	26
4.10.1 <i>Pytest.fixtures</i>	26
4.10.2 <i>Pytest-html</i>	26
4.10.3 <i>Pytest.marks</i>	26
4.10.4 <i>Pytest-bdd</i>	27
5 METODOLOGIA.....	28
5.1 Levantamento bibliográfico.....	28
5.2 Configuração do ambiente de desenvolvimento.....	28
5.3 Desenvolvimento de casos de teste.....	31
5.3.1 <i>Utilização do framework de testes Pytest</i>	31
5.3.2 <i>Criação das especificações dos testes usando o BDD</i>	31
5.3.3 <i>Implementação do padrão POM</i>	32
5.4 Criação, execução e validação dos testes.....	34
6 RESULTADOS.....	38
7 CONCLUSÃO.....	42
REFERÊNCIAS.....	44
APÊNDICE A – Casos de Teste.....	47

1 INTRODUÇÃO

O teste de software é uma fase do processo de desenvolvimento que visa garantir a qualidade do software, verificando se ele atende aos requisitos especificados e funciona conforme o esperado pelo cliente. Sommerville (2011) afirma que “o processo de teste é destinado a mostrar que um programa faz o que é proposto a fazer e para descobrir os defeitos do programa antes do uso. [...] Os resultados do teste são verificados à procura de erros, anomalias ou informações sobre os atributos não funcionais do programa”.

No contexto do desenvolvimento de software, o teste desempenha um papel crucial ao buscar identificar o maior número de falhas com o menor esforço possível, assegurando que o produto final atenda aos padrões de qualidade exigidos. Entretanto, alcançar esse objetivo envolve superar desafios consideráveis, como a multiplicidade de requisitos, a interdependência entre diferentes partes do sistema, a necessidade de cobrir uma ampla gama de casos de uso e a constante evolução inerente ao processo de desenvolvimento. Essa complexidade é acentuada pela diversidade de ambientes de execução e dispositivos.

Consequentemente, o teste manual em aplicações web pode revelar-se moroso, suscetível a erros e dispendioso. Além disso, a repetição de tarefas de teste comuns pode demandar recursos consideráveis da equipe de desenvolvimento. Mas, é fundamental reconhecer a coexistência entre testes manuais e automatizados. Conforme argumentado por Sommerville:

“O uso de testes automatizados tem aumentado consideravelmente nos últimos anos. Entretanto, os testes nunca poderão ser totalmente automatizados, já que testes automáticos só podem verificar se um programa faz aquilo a que é proposto. É praticamente impossível usar testes automatizados para testar os sistemas que dependem de como as coisas estão (por exemplo, uma interface gráfica de usuário), ou para testar se um programa não tem efeitos colaterais indesejados.”
(Sommerville, 2011)

Nesse contexto, é essencial abordar também a automação de testes, esta técnica consiste na prática de utilizar ferramentas ou frameworks especializados para controlar a execução de testes de software. Esse processo é realizado com o auxílio de scripts que replicam ações manuais, testando as funcionalidades do software e verificando se os resultados obtidos estão alinhados com os esperados.

Por fim, este trabalho de conclusão de curso tem a finalidade de demonstrar como o Selenium WebDriver, integrado com a linguagem Python, pode ser utilizado para criar e manter testes automatizados de forma escalonável em aplicações web. Além disso, busca-se

explorar o uso de outras ferramentas complementares, como o gerenciador de testes Pytest, para garantir uma automação eficiente, organizada e de fácil entendimento tanto para a equipe de qualidade quanto para as equipes de desenvolvimento.

2 JUSTIFICATIVA

A automação de testes é uma prática essencial para acompanhar o desenvolvimento de software, especialmente em aplicações web. Com a prevalência dos métodos ágeis na indústria de software, o desenvolvimento está em constante andamento, o que resulta em ciclos de entrega mais rápidos. Nesse contexto, a automação de testes desempenha um papel crucial na garantia da qualidade do software, ajudando a identificar e corrigir potenciais erros e falhas rapidamente.

Ao adotar a automação de testes, as equipes de desenvolvimento podem detectar e corrigir erros mais cedo no ciclo de vida do desenvolvimento de software, reduzindo custos e tempo de entrega. Além disso, a automação de testes permite que os testes sejam executados de forma consistente e repetível, garantindo uma cobertura abrangente dos casos de teste e proporcionando uma maior confiança na qualidade do software entregue aos usuários finais.

“A automação dos testes dá segurança à equipe para fazer alterações no código, seja por manutenção, refatoração ou até mesmo para adição de novas funcionalidades. Além disso, representar casos de teste através de programas possibilita a criação de testes mais elaborados e complexos, que poderão ser repetidos identicamente inúmeras vezes e a qualquer momento.” (Bernardo e Kon, 2009)

Nesse contexto, o Selenium WebDriver surge como uma ferramenta poderosa para a automação de testes em navegadores web, oferecendo uma maneira eficiente de garantir a qualidade do software em diferentes cenários de uso. Esta ferramenta desempenha um papel fundamental na automação de testes em aplicações web, oferecendo uma maneira intuitiva e flexível de interagir com elementos da página e simular o comportamento do usuário. Sua compatibilidade com várias linguagens de programação, incluindo Python, o torna uma escolha popular entre os desenvolvedores e testadores de software.

Embora existam poucos estudos sobre automação dentro da plataforma de CRM (do inglês, *Customer Relationship Management*) Salesforce, esta foi escolhida para a implementação prática da automação, dadas suas características adequadas para o contexto. A abordagem de design *Page Object Model* (POM), estabelece uma base flexível e replicável para uma ampla gama de cenários de teste, o que é particularmente útil na Salesforce. A plataforma Salesforce, conhecida por sua padronização nas páginas e métodos de navegação, oferece um cenário propício para automação de testes. Isso ocorre porque a repetição dos elementos e padrões permite que os testes sejam facilmente aplicáveis em diversas áreas do sistema, aumentando a eficiência da validação. Além disso, os testes automatizados de ponta a

ponta (E2E) são especialmente valiosos neste contexto, pois permitem uma verificação abrangente e detalhada do sistema. Essa abordagem não só facilita a identificação precoce de erros, como também contribui significativamente para a qualidade do software. Ao detectar problemas em etapas iniciais, a automação reduz o tempo e os custos de manutenção e aprimora a experiência dos usuários finais, assegurando um sistema mais confiável e robusto.

Dessa forma, a utilização do POM na automação de testes para Salesforce não apenas otimiza o processo de validação, mas também apresenta um potencial de replicabilidade e escalabilidade elevado, podendo servir como um modelo para a automação de outras plataformas semelhantes. Isso reforça a importância de uma estrutura de automação bem planejada, que não apenas atende aos requisitos técnicos, mas também alinha-se com as necessidades estratégicas de qualidade e eficiência no desenvolvimento de software.

3 OBJETIVOS

3.1 Objetivo geral

Desenvolver uma automação de testes em ambiente web utilizando Python, Pytest e o Selenium WebDriver, com o objetivo de otimizar processos repetitivos e manuais, garantindo estabilidade e confiabilidade na execução de interações automatizadas em aplicações web.

3.2 Objetivos específicos

- a) Desenvolver casos de teste em Python utilizando o Selenium WebDriver para simular interações no CRM em nuvem Salesforce, cobrindo cenários comuns como preenchimento de formulários, navegação e validação de elementos.
- b) Implementar o padrão POM para organizar e reutilizar elementos de página, visando melhorar a manutenção e a escalabilidade dos testes automatizados.
- c) Utilizar o framework Pytest para estruturar, organizar e gerar relatórios dos testes automatizados, explorando recursos como marcadores, fixtures e do BDD (do inglês, *Behavior Driven Development*).
- d) Executar testes automatizados registrando capturas de tela como um auxílio visual para validar a correta execução das interações automatizadas.
- e) Analisar os resultados dos testes.
- f) Apresentar os resultados do trabalho de forma clara e organizada, destacando os benefícios, além de sugestões para aprimoramentos futuros.

4 FUNDAMENTAÇÃO TEÓRICA

4.1 Teste de Software

O desenvolvimento de software é dividido em diversas etapas, e uma das mais importantes é a fase de testes. Essa etapa busca identificar falhas no sistema para que suas causas sejam analisadas e corrigidas pela equipe de desenvolvimento antes da liberação do projeto para produção (Neto, 2015). Os conceitos de falha, erro e defeito são fundamentais nesse contexto. O defeito ocorre devido à implementação incorreta no código, seja por má compreensão de informações ou pelo uso inadequado de ferramentas ou métodos. Já o erro é caracterizado por um estado inconsistente ou inesperado gerado pelo defeito, que pode culminar em uma falha. A falha, por sua vez, representa um desvio entre o resultado obtido e o esperado durante a execução do sistema (Neto, 2015).

4.2 Teste Funcional

Os testes funcionais verificam os requisitos funcionais do sistema e podem ser classificados como caixa-branca ou caixa-preta. O teste caixa-branca considera o código-fonte, enquanto o caixa-preta foca nas entradas e saídas, sem levar em conta a implementação. No caso dos testes caixa-preta, os dados de entrada são fornecidos e os resultados obtidos são comparados com os valores esperados. Segundo Pressman (2011), o teste funcional caixa-preta pode identificar diversos tipos de erros, tais como: (1) funções incorretas ou ausentes, que ocorrem quando uma funcionalidade não opera conforme esperado ou está ausente no sistema; (2) erros de interface, relacionados a falhas na interação entre o usuário e o sistema, como elementos gráficos ou funcionalidades que não são intuitivas; (3) erros em estruturas de dados ou no acesso a bases de dados externas, envolvendo problemas na manipulação de dados ou na comunicação com bancos de dados; (4) erros de comportamento ou desempenho, englobando falhas no comportamento esperado do sistema ou degradação na performance, como lentidão ou travamentos; e (5) erros de inicialização e término, que ocorrem quando o sistema não inicia ou termina corretamente, comprometendo sua operação. Esse tipo de teste foca na funcionalidade do sistema sem considerar sua estrutura interna, validando os resultados com base nas entradas e saídas.

4.3 Testes Automatizados

A automação de testes usa scripts para simular atividades manuais, otimizando tempo e reduzindo custos. E como já mencionado, essa prática não substitui os testes manuais, mas complementa o processo. A automação no processo de software traz diversas vantagens para assegurar a qualidade e a eficácia dos produtos. Com a automação dos testes, as equipes de desenvolvimento têm a capacidade de executar testes de forma repetitiva, facilitando a detecção precoce de falhas e garantindo a validação contínua das funcionalidades implementadas. Isso não só acelera o ciclo de desenvolvimento, possibilitando lançamentos mais rápidos e frequentes, como também eleva a confiabilidade do software, diminuindo a ocorrência de erros nas versões em produção (Chicanelli, 2019).

No contexto do desenvolvimento ágil, é crucial contar com testes automatizados para garantir um feedback oportuno, especialmente devido aos ciclos de desenvolvimento curtos. Isso se deve ao fato de que os testes automatizados são mais confiáveis do que os testes manuais, economizando tempo e evitando a introdução de bugs no software (Myers; Badgett; Sandler, 2012, tradução própria). Os testes automatizados são fundamentais no desenvolvimento de software, garantindo a qualidade do sistema e a conformidade com os requisitos do cliente. Com a evolução da área, surgiram ferramentas como o Selenium WebDriver, que permitem automatizar os testes e aumentar a confiabilidade, reduzindo o tempo necessário para testes manuais. Para Myers, Badgett e Sandler (*op. cit.*), os testes automatizados contribuem significativamente para reduzir custos e tempo de projeto durante o desenvolvimento. Além de atender às especificações do cliente, os sistemas de software devem ser seguros, eficientes, flexíveis e de fácil manutenção. Desta forma, o Selenium WebDriver se destaca como uma ferramenta robusta e flexível para a automação de testes em aplicações web, oferecendo uma série de vantagens e possibilidades para os profissionais da área de testes.

4.4 Selenium WebDriver

Selenium é o software de automação de testes mais popular atualmente (Testing Company, 2024). Conforme descrito na documentação oficial, o Selenium WebDriver é uma API que controla o comportamento dos navegadores, com suporte para todos os principais navegadores, cada navegador possui sua própria implementação do WebDriver, conhecida como driver, que facilita a comunicação entre o Selenium WebDriver e o navegador¹ (Selenium, 2024). Essa abordagem visa incentivar os fornecedores de navegadores a

¹ Disponível em: <https://www.selenium.dev/pt-br/documentation/webdriver/getting_started/>. Acesso em: 20 out. 2024.

assumirem a responsabilidade pela implementação de seus próprios drivers. Embora o Selenium WebDriver prefira utilizar os drivers de terceiros, ele também oferece seus próprios drivers quando necessário. A estrutura do Selenium WebDriver proporciona uma interface de usuário que integra perfeitamente os diferentes back-ends de navegadores, permitindo a automação entre navegadores e plataformas de forma transparente. Com esta ferramenta, os desenvolvedores podem produzir interações do usuário em um navegador da web, como clicar em botões, preencher formulários, navegar entre páginas e realizar assertivas.

Antes de iniciar a criação dos scripts de automação com o Selenium WebDriver, é importante compreender alguns recursos específicos do framework, utilizando Python como linguagem de programação. Localizar elementos é um dos aspectos mais fundamentais do uso do Selenium WebDriver, já que as interações na página como ler, clicar, editar, devem ser realizadas nesses elementos em que selecionamos. O Selenium WebDriver oferece várias maneiras para identificar exclusivamente um elemento, todas elas feitas dentro do *Document Object Model* (DOM). O DOM é a estrutura hierárquica que representa a página web carregada no navegador. Ele organiza todos os elementos HTML (do inglês, *Hypertext Markup Language*) da página como botões, imagens, links, em forma de uma árvore de nós, onde cada nó corresponde a uma parte do documento, seja uma tag HTML, um atributo ou texto (MDN Web Docs, 2024).

Para localizar um elemento, utilizamos o método `find element`. Quando esse método é chamado na instância do driver, ele percorre o DOM e retorna a referência do primeiro elemento que corresponde ao seletor especificado. Há momentos em que é necessário encontrar todos os elementos que correspondem a um determinado localizador, e não apenas o primeiro. Como o caso de itens em uma picklist. Neste caso é utilizado o método `find elements`, que retorna uma lista contendo referências a todos os elementos que correspondem ao localizador. Se nenhum elemento for encontrado, ele retornará uma lista vazia. Em certos casos, é possível localizar elementos dentro do contexto de outro elemento pai. Para isso, podemos encadear o método `find elements` ao elemento pai, permitindo acessar todos os elementos filhos correspondentes ao critério de busca dentro do escopo desse elemento² (Selenium, 2024).

Como já citado, quando um script Selenium WebDriver precisa encontrar um elemento, ele utiliza métodos *locators* como `find element` ou `find elements` que realizam a busca através do DOM. Este processo pode ser realizado de diferentes formas, dependendo do

² Disponível em: <<https://www.selenium.dev/pt-br/documentation/webdriver/elements/finders/>>. Acesso em: 05 out. 2024.

tipo de locator utilizado que foi utilizado. Os *locators* são formas de identificar elementos numa página em forma de argumento aos métodos finders. A documentação oficial do WebDriver, recomenda que sejam utilizados apenas localizadores do tipo XPath e CSS para melhorar o desempenho³ (Selenium, 2024). A documentação também mostra todas as oito estratégias diferentes de localização de elementos embutidos no framework, como mostra o quadro 1.

Quadro 1 – As oito estratégias diferentes de localização de elementos embutidas no Selenium WebDriver

Localizador	Descrição
class name	Localiza elementos cujo nome de classe contém o valor de pesquisa (nomes de classes compostas não são permitidos)
css selector	Localiza elementos que correspondem a um seletor CSS
id	Localiza elementos cujo atributo de ID corresponde ao valor de pesquisa
name	Localiza elementos cujo atributo NAME corresponde ao valor de pesquisa
link text	Localiza elementos âncora cujo texto visível corresponde ao valor de pesquisa
partial link text	Localiza elementos âncora cujo texto visível contém o valor da pesquisa. Se vários elementos forem correspondentes, apenas o primeiro será selecionado.
tag name	Localiza elementos cujo nome de tag corresponde ao valor de pesquisa
xpath	Localiza elementos que correspondem a uma expressão XPath

Fonte: Selenium: Localizando elementos, 2024.

4.5 Python

Para construção do script de testes, será necessária a construção de algoritmos que façam essa interação com a página utilizando o DOM. Um algoritmo é um conjunto de instruções executadas em uma determinada ordem que orienta o computador sobre como executar uma determinada tarefa (Baudson, 2013). Para este projeto, a linguagem escolhida para a escrita desses algoritmos foi o Python, amplamente reconhecido por sua simplicidade e legibilidade. Segundo a documentação oficial, "Python é uma linguagem de programação de propósito geral, de alto nível e que pode ser aplicada em muitos tipos diferentes de problemas" (Python, 2024).

Os benefícios do Python na automação de testes são diversos e tornam essa linguagem uma escolha ideal para essa área. Um dos principais fatores é a sua sintaxe clara e simples, semelhante à linguagem humana, o que facilita a leitura e compreensão dos scripts de automação. Isso reduz a complexidade e permite que os desenvolvedores e testadores

³ Disponível em: <<https://www.selenium.dev/pt-br/documentation/webdriver/elements/finders/>>. Acesso em: 05 out. 2024.

escrevam testes com menos esforço e em menos tempo, aumentando significativamente a produtividade. Além disso, o Python possui uma vasta biblioteca padrão, que oferece uma grande variedade de códigos reutilizáveis para tarefas comuns, o que elimina a necessidade de desenvolver funcionalidades do zero (AWS, 2024).

4.6 Salesforce

Para execução e demonstração dos testes automatizados, foi escolhido o CRM baseado em nuvem: Salesforce. CRM significa *Customer Relationship Management* (Gestão de Relacionamento com o Cliente), é uma ferramenta estratégica que centraliza a gestão de vendas, marketing, atendimento e outros pontos de contato com o cliente. Com ela é possível realizar cadastro de contatos, armazenando informações detalhadas sobre clientes e suas interações com a empresa, como visitas a sites, e-mails e ligações. Essa integração de dados permite que as equipes tenham uma visão completa dos clientes, melhorando o atendimento e a experiência do consumidor. Entre os produtos da Salesforce, destaca-se o *Sales Cloud*, que é uma solução voltada para o gerenciamento de negócios, ideal para otimizar o processo de vendas e melhorar a organização das informações comerciais. Esta é a solução que será utilizada para execução dos testes automatizados. Um recurso interessante da Salesforce são as Sandboxes, que são ambientes isolados de desenvolvimento e teste. Esse recurso é essencial para testar novos recursos, implementar atualizações antes de lançá-las e realizar treinamentos. Também é importante falar sobre os Objetos, que são componentes fundamentais usados para armazenar e organizar os dados na plataforma. Esses objetos podem ser tanto padrões (como *leads*, contas e oportunidades) quanto personalizados, permitindo que as empresas ajustem o CRM conforme seus requisitos. A estrutura de objetos facilita a integração de informações, proporcionando uma visão completa e centralizada dos dados do cliente, além de possibilitar automações e personalizações de acordo com os processos de negócio.

A utilização do Salesforce como plataforma para realizar testes automatizados no contexto do presente trabalho, é justificada pela robustez, flexibilidade e funcionalidades que estão prontas dentro dos objetos, portanto, não será realizada nenhuma alteração na *Sandbox* de testes utilizada, uma vez que o objetivo principal é demonstrar o processo de construção de testes automatizados dentro da plataforma. Essa abordagem garante que o ambiente se mantenha fiel ao seu propósito original e permite que o foco esteja apenas na criação e execução dos testes. Também vale destacar que cada Org (organização) no Salesforce possui

suas particularidades e personalizações, dependendo dos requisitos específicos do cliente.

Cada implementação do Salesforce é adaptada às necessidades empresariais individuais, o que significa que os fluxos de trabalho, objetos e funcionalidades podem variar significativamente de uma Org para outra. Essa flexibilidade na personalização da plataforma torna os testes automatizados ainda mais relevantes, pois garante que os cenários de teste sejam adequados ao ambiente específico de cada empresa. Portanto, a escolha do Salesforce para o presente trabalho, reflete a sua capacidade de suportar testes automatizados em um ambiente seguro, ao mesmo tempo em que demonstra como construir e gerenciar esses testes de maneira eficaz, respeitando as particularidades de cada Org personalizada.

4.7 POM

O POM é um padrão de design com a proposta de simplificar a escrita de testes automatizados, proporcionando uma estrutura mais organizada e orientada a objetos, além de melhorar a manutenção de testes e reduzir a duplicação de código. Segundo a documentação oficial do Selenium, algumas das vantagens incluem a separação bem definida entre o código de teste e o código da página específica, além de existir um repositório único para os serviços ou operações que a página oferece, em vez de tê-los espalhados pelos testes⁴ (Selenium, 2024). Logo, ao utilizar o POM, evitamos com que os *locators* e seus métodos fiquem dispersos pelo código de teste, o que torna a manutenção mais fácil e menos propensa a erros. Saini (2022), diz que a criação de um modelo de objeto de página é uma maneira eficaz de promover a escrita de código limpo e bem testado.

4.8 Aqua IDE

Após a definição do framework que será utilizado para a construção dos scripts de teste, da linguagem suportada pelo framework e do ambiente em que os testes serão realizados, podemos escolher o IDE mais adequado ao contexto de testes de software. A sigla IDE refere-se a Ambiente de Desenvolvimento Integrado (do inglês, *Integrated Development Environment*). Trata-se de um software que reúne diversas ferramentas utilizadas por desenvolvedores em uma única interface gráfica, facilitando o processo de criação de aplicações (Red Hat, 2024).

Dado o contexto, o IDE selecionado será o Aqua, desenvolvido pela JetBrains. O

⁴ Disponível em:

<https://www.selenium.dev/pt-br/documentation/test_practices/encouraged/page_object_models/>. Acesso em: 20 out. 2024.

Aqua, é o primeiro IDE criado especificamente para automação de testes⁵ (JetBrains, 2024). Ela promove um espaço de trabalho versátil que oferece suporte tanto para o Selenium, quanto para o Cypress e Playwright, estes, frameworks também utilizados para automação de testes. Nesta IDE é possível escrever os testes em linguagens como Java, Python, JavaScript, TypeScript, Kotlin, além de SQL para realizar consultas e manipulações de dados, tornando-a uma solução abrangente para testes de aplicativos web e banco de dados.

O motivo da escolha da IDE para a implementação dos testes automatizados, é que com ela, é possível testar diretamente sem precisar instalar e configurar vários plugins e instalações, apesar de que algumas instalações ainda haverão de serem feitas. Por ser focada na jornada de escrita de um automatizador de testes, a IDE possui alguns recursos facilitadores. Um dos recursos da IDE que mais será utilizado, é o Web Inspector, este funciona como um navegador embutido e permite encontrar qualquer elemento da página sem mudar para outra ferramenta. Outro recurso facilitador dentro do Web Inspector, é que ele gera um seletor CSS ou XPath para um elemento da página da Web que foi selecionado e o adiciona ao código-fonte. Quando ativo, a complementação de código CSS e XPath sugere e realça localizadores para os elementos mais importantes da página Web. O Aqua facilita a visualização e execução de testes, exibindo ícones para rodar testes diretamente do editor, como os de Selenium. Ele oferece resultados detalhados, com logs e saídas de console, além de permitir a ordenação e filtragem de testes falhos ou ignorados. Além disso, o Aqua destaca o código que falhou e exibe dicas com o tempo de execução de cada etapa. A navegação entre os resultados e o código-fonte é simples, e a ferramenta de busca integrada permite encontrar testes específicos facilmente, mesmo que não estejam declarados como funções. Outra funcionalidade importante da IDE é a opção de criar um ambiente virtual logo no início do projeto, o que é essencial para o nosso contexto. Esse recurso garante que todas as instalações de pacotes fiquem isoladas em um ambiente exclusivo, evitando conflitos com outras bibliotecas ou versões instaladas no sistema. Além disso, facilita a replicação do ambiente de desenvolvimento em outras máquinas, garantindo consistência e controle sobre as dependências do projeto⁶ (JetBrains, 2024).

⁵ Disponível em: <<https://www.jetbrains.com/pt-br/aqua/>>. Acesso em: 12 out. 2024.

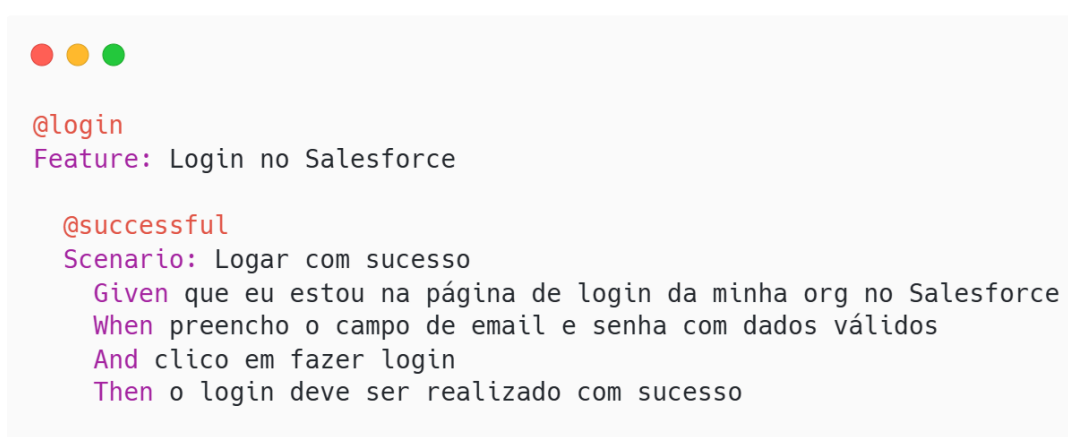
⁶ Disponível em: <<https://lp.jetbrains.com/selenium-1591/>>. Acesso em: 12 out. 2024.

4.9 Behavior-Driven Development (BDD)

O *Behavior-Driven Development* (BDD) foi introduzido por Dan North em 2003 como uma evolução do *Test-Driven Development* (TDD) de Kent Beck. O BDD surgiu para solucionar problemas de comunicação e alinhamento entre equipes durante o desenvolvimento de software, dificuldades essas observadas por North enquanto lecionava e praticava TDD (CARVALHO, 2019). O principal foco do BDD é promover uma comunicação clara entre desenvolvedores, *Product Owners* (POs), *Quality Assurances* (QAs) e outras partes envolvidas, usando linguagem natural para descrever o comportamento desejado do software.

A forma de escrever no formato BDD costuma ser formatada utilizando a linguagem Gherkin, que descreve o comportamento do software de forma clara e simples. Embora o BDD não dependa exclusivamente do Gherkin, ele é amplamente utilizado em ferramentas BDD que escrevem este formato em arquivos de formatos *.feature*. A escrita dos cenários no Gherkin pode ser dividida em três etapas principais: Funcionalidades (*features*), Descrições simples das funcionalidades (*User Stories*) e Critérios de Aceitação ou Cenários (*Scenario*), que seguem uma estrutura baseada em três elementos: Dado (*Given*) para pré-condições, Quando (*When*) para os passos do cenário, e Então (*Then*) para os resultados esperados. A palavra "E" (And) pode ser usada para adicionar contexto adicional em cada um desses elementos. É possível visualizá-la pela figura 1, com um exemplo prático de seu uso.

Figura 1 – Estrutura do cenário utilizando Gherkin



```
@login
Feature: Login no Salesforce

  @successful
  Scenario: Logar com sucesso
    Given que eu estou na página de login da minha org no Salesforce
    When preencho o campo de email e senha com dados válidos
    And clico em fazer login
    Then o login deve ser realizado com sucesso
```

Fonte: Elaborado pelo Autor (2024)

4.10 Pytest

Para gerenciamento dos testes escritos em Python, o Pytest torna fácil a configuração e criação dos suites de teste. Segundo Okken (2017, tradução própria), esta ferramenta pode ser usada para todos os tipos e níveis de testes de software, como equipes de desenvolvimento, equipes de QA, grupos de teste independente e quem pratica o TDD. O Pytest é uma ferramenta de linha de comando, que no contexto de testes web automatizados, é muito usada para executar os testes em sua totalidade, em conjuntos específicos e até individualmente. Outros pontos destacados por Okkem (2017, tradução própria), é a utilização de *fixtures* e também da geração de relatórios.

4.10.1 *Pytest.fixtures*

As *fixtures* então seriam essenciais para estruturar o código de teste para praticamente qualquer sistema de software não trivial, ou seja, que foi desenvolvido utilizando uma abordagem complexa. Elas são funções que são executadas pelo Pytest antes ou depois das funções que executam o próprio teste em si.

4.10.2 *Pytest-html*

O uso da uma extensão, como o Pytest-html, permite gerar relatórios em formato HTML após a execução dos testes, fornecendo uma visão detalhada dos resultados dos testes, incluindo informações sobre testes passados, falhas, erros e tempo de execução. A importância de gerar um relatório após a execução dos testes está em fornecer uma documentação clara e acessível sobre o estado do software em relação aos testes realizados, facilitando a identificação de problemas, o acompanhamento do progresso dos testes ao longo do tempo e a comunicação eficaz entre membros da equipe de desenvolvimento, gerenciamento e outras partes interessadas.

4.10.3 *Pytest.marks*

O Pytest também oferece um recurso chamado *marks*, que permite adicionar marcadores personalizados às funções de teste, além dos já definidos pelo próprio Pytest. Esses marcadores podem ser inseridos no topo de uma função executora do script de teste. Na linha de comando, é possível executar testes específicos usando esses marcadores, passando a flag “-m”. Outro recurso muito útil é o “mark.skip”. Ao utilizar esse marcador, quando uma suíte de testes é executada, ou mesmo quando o comando “Pytest” é utilizado para rodar todos

os testes cujo nome comece ou termine com "test", os testes com essa marcação serão ignorados e não executados.

4.10.4 Pytest-bdd

Também será utilizado junto ao Pytest, o *plugin* Pytest-bdd. Ele introduz a escrita dos testes no formato BDD, utilizando decoradores para marcar as funções que executam os testes e atrelá-las aos passos descritos dentro do arquivo de formato .feature para os critérios de aceite da funcionalidade. O Pytest-bdd, que usa a linguagem Gherkin, permite a automação desses cenários de comportamento no ambiente de testes do Pytest. Assim, o Gherkin fornece a estrutura para descrever o comportamento, e o Pytest-bdd permite executar esses cenários como testes automatizados.

5 METODOLOGIA

Para alcançar os objetivos propostos no presente trabalho foram realizadas as seguintes etapas:

5.1 Levantamento bibliográfico

Esta etapa envolveu a pesquisa e revisão de literatura relacionada ao Selenium WebDriver, automação de testes em aplicações web, Python, POM e Pytest. Também foram explorados livros, artigos científicos, tutoriais online e documentações oficiais para obter uma compreensão abrangente dos conceitos, práticas mais utilizadas e técnicas relevantes para o desenvolvimento do projeto.

5.2 Configuração do ambiente de desenvolvimento

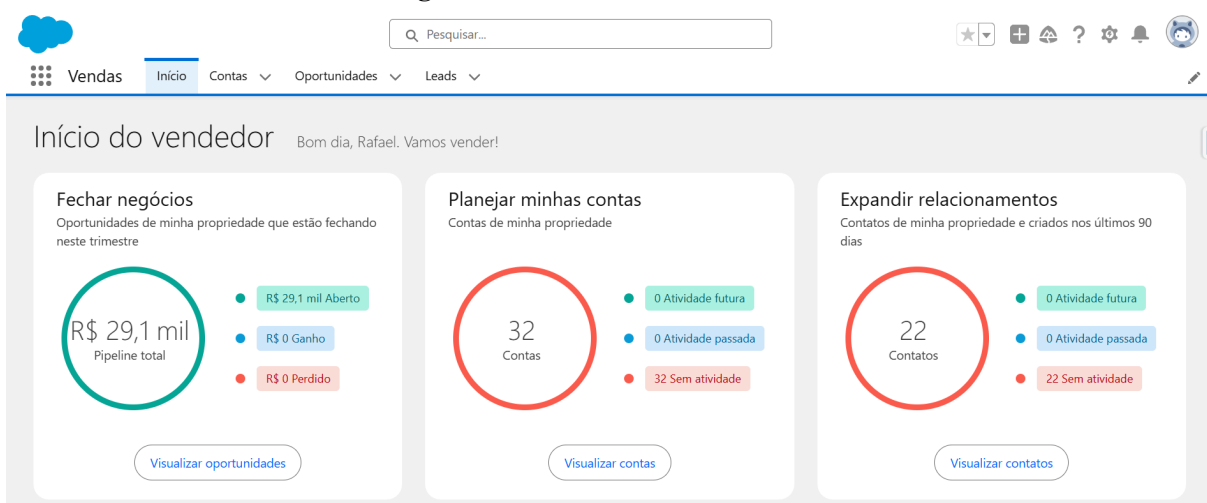
Com base no levantamento realizado na etapa anterior, nesta fase foram identificados, baixados e configurados os recursos essenciais para o desenvolvimento e execução dos testes automatizados com Selenium WebDriver em Python. Isso envolveu a instalação e configuração do Python, do Selenium, do navegador Google Chrome, além de outras bibliotecas cruciais para a construção da ferramenta. Todos esses recursos estão listados no quadro 2.

Quadro 2 – Ferramentas e bibliotecas utilizadas

Ferramenta	Categoria	Versão	Descrição
Google Chrome	Instalado via .exe	130.0.6723.92	Navegador web usado para automação de testes de interface gráfica com Selenium.
Aqua IDE	Instalado via .exe	2024.2.1	Ambiente de desenvolvimento para Python, com suporte a análise e depuração.
Python	Instalado via .exe	3.11.2004	Linguagem de programação versátil, usada para desenvolvimento de aplicações, análise de dados, etc.
Selenium	Biblioteca python instalada via pip	4.25.0	Automação de navegadores para testes, preenchimento de formulários e outras interações online.
pytest-html	Biblioteca python instalada via pip	4.1.2001	Gera relatórios em HTML para testes executados com pytest.
pytest-bdd	Biblioteca python instalada via pip	7.3.2000	Integra pytest com o BDD (Behavior-Driven Development) para testes orientados a comportamento.

Fonte: Elaborado pelo Autor (2024)

Também foi utilizado uma sandbox no Salesforce (figura 2), para execução dos testes, juntamente com a IDE Aqua, além do arquivo de configuração do Pytest, o conftest.py (figura 3).

Figura 2 – Tela inicial do Salesforce

Fonte: Elaborado pelo Autor (2024)

Figura 3 – Arquivo conftest.py

```
@pytest.fixture
def driver():
    opt = Options()
    opt.add_experimental_option(
        "prefs", {"profile.default_content_setting_values.notifications": 2}
    )
    driver = webdriver.Chrome(options=opt)
    driver.implicitly_wait(10)
    driver.maximize_window()
    driver.get("https://login.salesforce.com")

    yield driver

    driver.quit()
```

Fonte: Elaborado pelo Autor (2024)

A estrutura e o conteúdo das pastas da aplicação foram organizados para facilitar a execução e a manutenção dos testes automatizados. A raiz do projeto contém arquivos essenciais para configuração do ambiente, como `conftest.py` (figura 3), o `requirements`, listando todas as instalações via `pip` e o `.env` para variáveis sensíveis. Cada pasta desempenha um papel específico: `/pages` centraliza funções de teste por página, enquanto `/tests` subdivide-se em pastas para dados de teste (`/tests/data`), cenários (`/tests/features`), e passos de execução (`/tests/step-defs`). A pasta `/utils` organiza tanto os relatórios gerados quanto as capturas de tela, garantindo evidências visuais e resultados para cada teste executado. Essa estrutura modular permite um fluxo de trabalho eficiente e transparente, facilitando a

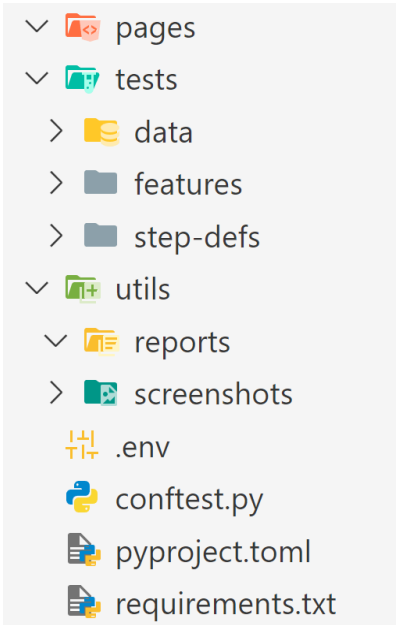
manutenção dos testes e o controle das evidências. Essas informações podem ser observadas de forma detalhada no quadro 3, assim como a estrutura da aplicação na figura 4.

Quadro 3 – Estrutura de pastas da aplicação

Pastas	Descrição
Raiz do projeto	Contém arquivos de configuração do pytest (conftest.py), Python (pyproject.toml), .env para as variáveis sensíveis e requirements.txt para instalação das dependências.
/pages	Centraliza as funções de teste por página, incluindo uma página base com funções principais do Selenium.
/tests/data	Armazena arquivos .json usados para validação de labels e preenchimento de campos.
/tests/features	Contém arquivos .feature escritos em Gherkin, definindo cenários de teste com as etapas a serem executadas.
/tests/step-defs	Armazena os arquivos de definição dos passos de teste, mapeando funções das páginas com os cenários definidos.
/utils/reports	Guarda o último relatório gerado após a execução dos testes, tanto por suíte quanto individualmente.
/utils/screenshots	Armazena evidências em .png (capturas de tela) feitas durante a execução dos testes.

Fonte: Elaborado pelo Autor (2024)

Figura 4 – Estrutura de pastas da aplicação na IDE



Fonte: Elaborado pelo Autor (2024)

5.3 Desenvolvimento de casos de teste

Foram desenvolvidos dez casos de teste para demonstrar o uso do Selenium WebDriver em Python na automação de interações em aplicações web. Esses testes automatizados visam verificar a estabilidade e a integridade das funcionalidades testadas. Os casos de teste abrangem uma variedade de cenários comuns, como preenchimento de formulários, cliques em botões, navegação entre páginas e verificação de elementos na página.

5.3.1 Utilização do framework de testes Pytest

O framework de testes Pytest foi utilizado para a execução, organização e geração de relatórios dos testes automatizados em Python. Cada script de teste está fortemente vinculado a uma fixture e a uma feature específica, garantindo modularidade e reutilização de código. No arquivo `confest.py` (figura 3), foi implementada a fixture driver, que configura o ambiente do navegador para os testes automatizados. A fixture driver inicializa uma instância do Chrome com opções específicas, como a desativação de notificações e tamanho da janela. Ela também define uma espera implícita de 10 segundos para carregar os elementos. Essa fixture carrega a página inicial para login no Salesforce e, após a execução de cada teste, fecha o navegador, assegurando a limpeza dos recursos.

5.3.2 Criação das especificações dos testes usando o BDD

Em seguida, foram desenvolvidos dez casos de teste utilizando a linguagem Gherkin, que define, etapa por etapa, os elementos e funcionalidades a serem verificados na aplicação. Cada caso de teste inclui marcadores específicos, como `@create_account` e `@successful`, que permitem a execução individual dos testes por meio de comandos no terminal. A figura 5 apresenta um exemplo do caso de teste número três, ilustrando o cenário de criação de conta com sucesso. O BDD foca no comportamento do software sob a perspectiva do usuário, garantindo que o desenvolvimento atenda às necessidades reais do negócio, agregando valor ao usuário e não apenas atendendo a especificações técnicas. Isso promove uma comunicação eficaz entre as equipes de TI e de negócios, minimizando ruídos e garantindo o alinhamento.

Figura 5 – Arquivo ct03_create_account.feature

```

@create_account
Feature: Criar conta no formulário de cadastro

    @successful
    Scenario: Criar uma conta com sucesso
        Given que estou na página de cadastro de conta
        When eu acesso o formulário de criação de conta
        And preencho todos os campos disponíveis
        And eu salvo a conta
        Then a conta deve ser criada com sucesso
        And todos os campos devem estar presentes nos detalhes da conta

```

Fonte: Elaborado pelo Autor (2024)

5.3.3 Implementação do padrão POM

O design pattern escolhido para a organização e reutilização dos elementos da página nos testes automatizados foi o POM. Nesse modelo, serão criadas classes separadas para cada página da aplicação, cada uma contendo métodos específicos para interagir com os elementos desta página. Esse enfoque facilita a manutenção e escalabilidade dos testes automatizados. Além disso, haverá um arquivo principal que centraliza as funções essenciais do Selenium, as quais serão chamadas dentro das classes de páginas. Esse arranjo é ilustrado nas figuras 6 e 7, que demonstram, respectivamente, algumas funções da classe base da página e a página.

Figura 6 – Algumas funções da página base

```

class BasePage:
    def __init__(self, driver):
        self.driver = driver

    def access_link(self, tab_link):
        self.driver.get(tab_link)

    def text_exists_on_screen(self, locator, text, timeout=10):
        try:
            self.presence_wait(locator, timeout)
            element = self.find_element(locator)

            assert text in element.text, (
                f"0 texto '{text}' não foi encontrado "
                f"no elemento com o seletor {locator}!"
            )

        except TimeoutException:
            pytest.fail(
                f"0 seletor {locator} não foi encontrado na tela!",
                pytrace=True,
            )

```


Fonte: Elaborado pelo Autor (2024)

No exemplo da figura 6, o código define o arquivo principal mencionado oferecendo funcionalidades para interagir com uma página web durante a automação de testes. O objetivo

geral dessa classe é facilitar o controle e as verificações de uma página web durante testes automatizados. Ela possui três métodos principais:

1. `__init__(self, driver)`: Inicializa a classe com um objeto driver, que representa o navegador utilizado para as interações.
2. `access_link(self, tab_link)`: Acessa um link ou URL, passando a URL como argumento para o método `get()` do driver.
3. `text_exists_on_screen(self, locator, text, timeout=10)`: Verifica se um texto específico está presente em um elemento da página. O método aguarda até que o elemento seja encontrado e compara se o texto fornecido está no conteúdo do elemento. Se o texto não for encontrado ou o elemento não aparecer a tempo, gera uma falha no teste com o uso de `pytest.fail`.

Figura 7 – Chamada da função na página de contas (account_page.py)



```
def verificar_pagina_cadastro(self):
    url = os.getenv("ACCOUNT_PAGE_URL")
    self.access_link(url)
    self.text_exists_on_screen(AccountLocators.account_page, "Contas")
```

Fonte: Elaborado pelo Autor (2024)

No exemplo da figura 7, o método verifica uma das etapas descritas para que o teste passe, herdando da `BasePage`, os métodos `access_link` e `text_exists_on_screen`. De modo que:

1. `url = os.getenv("ACCOUNT_PAGE_URL")`:
 - Este comando obtém o valor de uma variável de ambiente chamada "ACCOUNT_PAGE_URL". Essa variável geralmente é configurada em algum arquivo de configuração ou no ambiente do sistema operacional onde o código está sendo executado. Essa URL representa o endereço da página de cadastro que desejo acessar.
2. `self.access_link(url)`:
 - O método `access_link`, que foi definido na classe `BasePage`, é chamado aqui com a URL obtida na etapa anterior. Isso faz com que o navegador (controlado pelo driver) acesse a página de cadastro. Esse método simplesmente carrega a página desejada no navegador.
3. `self.text_exists_on_screen(AccountLocators.account_page, "Contas")`:

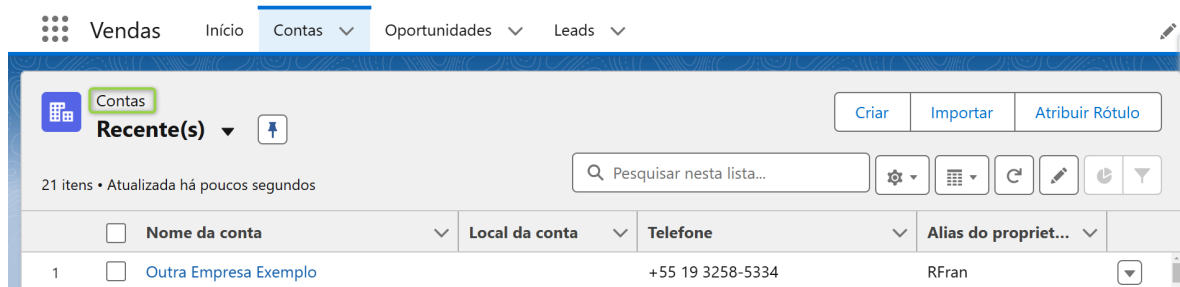
- Depois que a página é carregada, o método `text_exists_on_screen` é chamado. Ele verifica se o texto "Contas" aparece em um elemento da página localizado pelo seletor `AccountLocators.account_page`.
- O argumento `AccountLocators.account_page` representa o seletor utilizado para localizar o elemento da página (figura 8), que, neste caso, é do tipo XPath (figura 9). O elemento de interesse foi destacado em verde para facilitar a visualização.
- O método `text_exists_on_screen` aguarda até que o elemento esteja presente na tela e, em seguida, verifica se o texto "Contas" está presente nesse elemento. Caso o texto não seja encontrado, ele dispara um erro no teste, indicando que a verificação falhou.

Figura 8 – Localizador XPath utilizado para verificar a presença do título na página

```
account_page = (By.XPATH, '//span[@class="slds-var-p-right_x-small"]')
```

Fonte: Elaborado pelo Autor (2024)

Figura 9 – Página da web com o elemento localizado (destacado em verde)



Fonte: Elaborado pelo Autor (2024)

5.4 Criação, execução e validação dos testes

Como cada teste é definido por um cenário descrito utilizando a abordagem BDD, é necessário importar esses cenários para os scripts de teste. O script completo do teste é desenvolvido em conjunto com a criação das funções responsáveis por realizar as ações nas páginas a serem testadas, conforme ilustrado na figura 7. Essas funções são então importadas para o script de teste, sendo utilizadas conforme o cenário específico em execução. A figura 10 apresenta a versão final deste arquivo de teste, aplicando todos os conceitos abordados anteriormente.

Figura 10 – Arquivo ct03_create_account.py

```

@pytest.fixture
def ap(driver):
    return AccountPage(driver, screenshot=True)

@pytest.mark.account
@scenario("../features/ct03_create_account.feature", "Criar uma conta com sucesso")
def test_ct03_create_account():
    pass

@given("que estou na página de cadastro de conta")
def acessar_pagina_cadastro(driver, ap):
    login_page = LoginPage(driver)
    login_page.login_com_sucesso()
    ap.verificar_pagina_cadastro()

@when("eu acesso o formulário de criação de conta")
def acessar_formulario_criacao_conta(ap):
    ap.abrir_formulario_criacao_conta()

@when("preencho todos os campos disponíveis")
def preencher_campos_disponiveis(ap, load_account_data):
    ap.preencher_formulario_conta(load_account_data)

@when("eu salvo a conta")
def clicar_criar_conta(ap):
    ap.salvar_conta()

@then("a conta deve ser criada com sucesso")
def verificar_edicao_conta(ap, load_new_account_data):
    ap.verificar_sucesso(
        n_campos=len(load_new_account_data), campos_adicionais=1, campos_agrupados=0
    )

@then("todos os campos devem estar presentes nos detalhes da conta")
def verificar_campos_detalhes(ap, load_account_data):
    ap.verificar_campos_detalhes(load_account_data)

```

Fonte: Elaborado pelo autor (2024)

Foram escritos ao todo 10 casos de teste conforme apresentados no quadro 4. Os casos de teste encontram-se descritos de forma detalhada no apêndice A deste trabalho. Esses testes foram desenvolvidos com o objetivo de validar as funcionalidades de três objetos essenciais no Salesforce: Contas, Oportunidade e Leads. Cada um desses testes realiza verificações funcionais detalhadas, assegurando que os campos de entrada (ou *labels*) dos formulários estejam corretos, conforme especificado em arquivos JSON externos.

Quadro 4 – Scripts de teste, BDDs correspondentes e marcadores

Arquivo de Teste	BDD Correspondente	Marcador individual	Marcador de Suíte
test_ct01_valid_login.py	ct01_login.feature	login	-
test_ct02_labels_create_account.py	ct02_labels_create_account.feature	labels_create_account	account
test_ct03_create_account.py	ct03_create_account.feature	create_account	account
test_ct04_labels_create_opportunity.py	ct04_labels_create_opportunity.feature	labels_create_opportunity	opportunity
test_ct05_create_opportunity.py	ct05_create_opportunity.feature	create_opportunity	opportunity
test_ct06_labels_create_lead.py	ct06_labels_create_lead.feature	labels_create_lead	lead
test_ct07_create_lead.py	ct07_create_lead.feature	create_lead	lead
test_ct08_edit_account.py	ct08_edit_account.feature	edit_account	account
test_ct09_edit_opportunity.py	ct09_edit_opportunity.feature	edit_opportunity	opportunity
test_ct10_edit_lead.py	ct10_edit_lead.feature	edit_lead	lead

Fonte: Elaborado pelo autor (2024)

Os testes não se limitam apenas à verificação da presença e exatidão das *labels* nos formulários, mas também incluem a funcionalidade de preenchimento desses campos. Os valores a serem inseridos nos formulários são extraídos diretamente do arquivo JSON, e os testes garantem que cada campo seja preenchido corretamente com os dados fornecidos. Após o preenchimento, os testes verificam se os dados inseridos foram devidamente salvos no sistema, comparando os valores salvos com os valores originais preenchidos durante o teste. Essa etapa é crucial, pois garante que o sistema não apenas exiba corretamente os campos, mas também que os dados inseridos sejam refletidos na interface de forma precisa.

Além disso, esses testes incluem uma etapa de edição dos registros salvos. A edição é realizada com base em um outro arquivo JSON, que contém os novos valores a serem aplicados aos campos. O processo de edição permite verificar se, ao atualizar um registro existente, o sistema consegue substituir corretamente os dados antigos pelos novos, sem causar inconsistências. Após a edição, o teste valida novamente se os valores atualizados foram corretamente salvos e estão refletindo a modificação. A execução desses testes para a criação, o salvamento e a edição dos dados nos três objetos (Contas, Oportunidade e Leads) garante que as operações de manipulação de dados sejam executadas de maneira consistente e sem falhas.

A importância de testar tanto a criação quanto a edição de dados é fundamental para assegurar que o sistema funcione corretamente em ambos os cenários. Esse procedimento é especialmente relevante em sistemas complexos como o Salesforce, nos quais a precisão na inserção e atualização de dados é crucial para o funcionamento adequado das operações de negócios. Outro ponto de destaque, é a escolha de arquivos JSON como fonte de dados para as validações, já que ele contribui para uma melhor manutenção dos scripts de teste, separando os dados do código em si. Dessa forma, é possível atualizar os dados de teste de

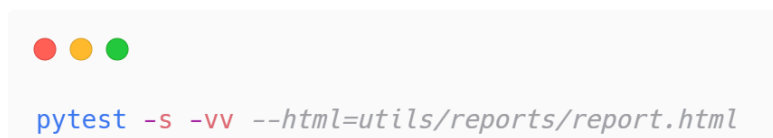
maneira mais eficiente, sem a necessidade de modificar diretamente os scripts de teste, o que facilita a adaptação do sistema a mudanças futuras e garante maior flexibilidade e escalabilidade.

6 RESULTADOS

Para a apresentação dos resultados, foi utilizada uma funcionalidade do Pytest para gerar um relatório com os resultados do último teste executado. Para rodar os testes, basta utilizar o comando `pytest -m nome_marcaador`. O parâmetro `-m` indica que o Pytest deve executar apenas os testes que contêm o marcador especificado. É possível passar um marcador exclusivo para rodar um teste individualmente, ou um marcador para rodar uma suíte de testes agrupados por características comuns. É possível também rodar apenas com o comando Pytest, sem os marcadores, o que fará com que todos os testes que se iniciam ou terminam com a palavra “test” no diretório sejam executados.

Além disso, o comando mostrado na figura 11 foi utilizado para gerar um relatório detalhado e personalizado em HTML, com informações completas sobre a execução de cada teste, principalmente se passaram ou não, além do tempo de execução total e também de cada um deles separadamente.

Figura 11 – Comando de execução dos scripts de teste e do relatório incluso

A terminal window with a light gray background and three colored window control buttons (red, yellow, green) in the top left corner. The command `pytest -s -vv --html=utils/reports/report.html` is displayed in a monospaced font, with `pytest` in blue, `-s` in red, `-vv` in red, and the rest in black.

Fonte: Elaborado pelo Autor (2024)

O parâmetro `-s` permite exibir as mensagens de saída padrão no terminal durante a execução, como logs e mensagens de depuração, enquanto o `-vv` habilita um nível de detalhe elevado nos resultados, exibindo o nome de cada teste, o status e informações adicionais. O relatório é salvo no diretório especificado na linha de comando e pode ser acessado para análise posterior. Essa abordagem facilita a identificação de possíveis falhas e organiza os resultados de forma estruturada, contribuindo significativamente para a documentação e manutenção dos testes no projeto. Após a execução, o relatório gerado pode ser aberto em formato HTML, mostrando o resultado dos testes, conforme exemplificado na figura 12.

Figura 12 – Relatório em HTML gerado após a execução dos dez casos de testes

☒ 0 Failed,
 ☒ 10 Passed,
 ☒ 0 Skipped,
 ☒ 0 Expected failures,
 ☒ 0 Unexpected passes,
 ☒ 0 Errors,
 ☒ 0 Reruns

[Show all details](#) / [Hide all details](#)

Result ▲	Test	Duration	Links
Passed	tests/step-defs/test_ct01_valid_login.py::test_ct01_valid_login	00:00:20	
Passed	tests/step-defs/test_ct02_labels_create_account.py::test_ct02_labels_create_account	00:00:34	
Passed	tests/step-defs/test_ct03_create_account.py::test_ct03_create_account	00:00:50	
Passed	tests/step-defs/test_ct04_labels_create_opportunity.py::test_ct04_labels_create_opportunity	00:00:34	
Passed	tests/step-defs/test_ct05_create_opportunity.py::test_ct05_create_opportunity	00:00:49	
Passed	tests/step-defs/test_ct06_labels_create_lead.py::test_ct06_labels_create_lead	00:00:41	
Passed	tests/step-defs/test_ct07_create_lead.py::test_ct07_create_lead	00:01:03	
Passed	tests/step-defs/test_ct08_edit_account.py::test_ct08_edit_account	00:00:56	
Passed	tests/step-defs/test_ct09_edit_opportunity.py::test_ct09_edit_account	00:00:56	
Passed	tests/step-defs/test_ct10_edit_lead.py::test_ct10_edit_lead	00:01:06	

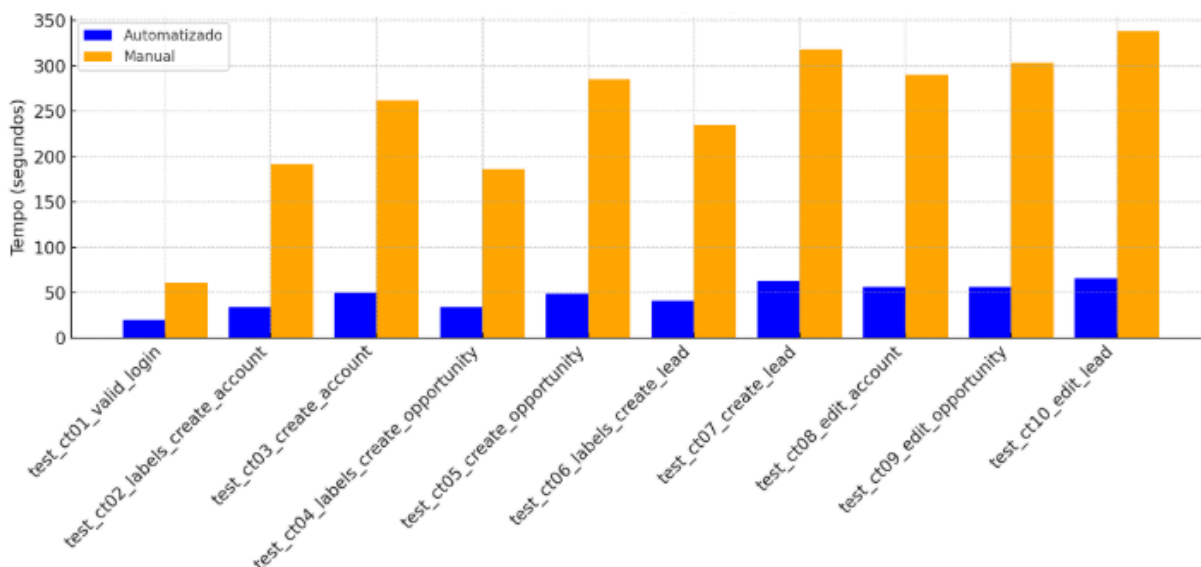
Fonte: Elaborado pelo Autor (2024)

Também foi desenvolvida uma função personalizada que registra screenshots da interface em momentos estratégicos durante a execução dos testes, organizando as capturas de tela em pastas separadas, sendo cada pasta dedicada a um script de teste específico. Essa abordagem permite coletar evidências visuais de todas as etapas realizadas de forma automatizada, garantindo um registro completo das operações executadas durante o teste. Após essas verificações, a função registra uma captura de tela do estado atual da página, utilizando a funcionalidade de captura automatizada. A captura de tela é nomeada de forma descritiva, e armazenada no diretório associado ao script de teste específico.

A inclusão de capturas de tela automatizadas nos testes representa uma importante melhoria para a documentação e a validação do sistema, especialmente em ambientes onde a interface gráfica desempenha um papel fundamental. Além disso, a organização das evidências em pastas específicas para cada script de teste contribui para a clareza e a eficiência no gerenciamento dos resultados, garantindo que as informações estejam disponíveis de forma acessível para futuras consultas ou auditorias.

Por fim, os resultados obtidos evidenciam os benefícios da automação de testes em termos de eficiência de tempo, conforme detalhado no gráfico 1 que compara os tempos de execução automatizada e manual.

Gráfico 1 – Comparação dos tempos de teste na execução manual e automatizada em segundos

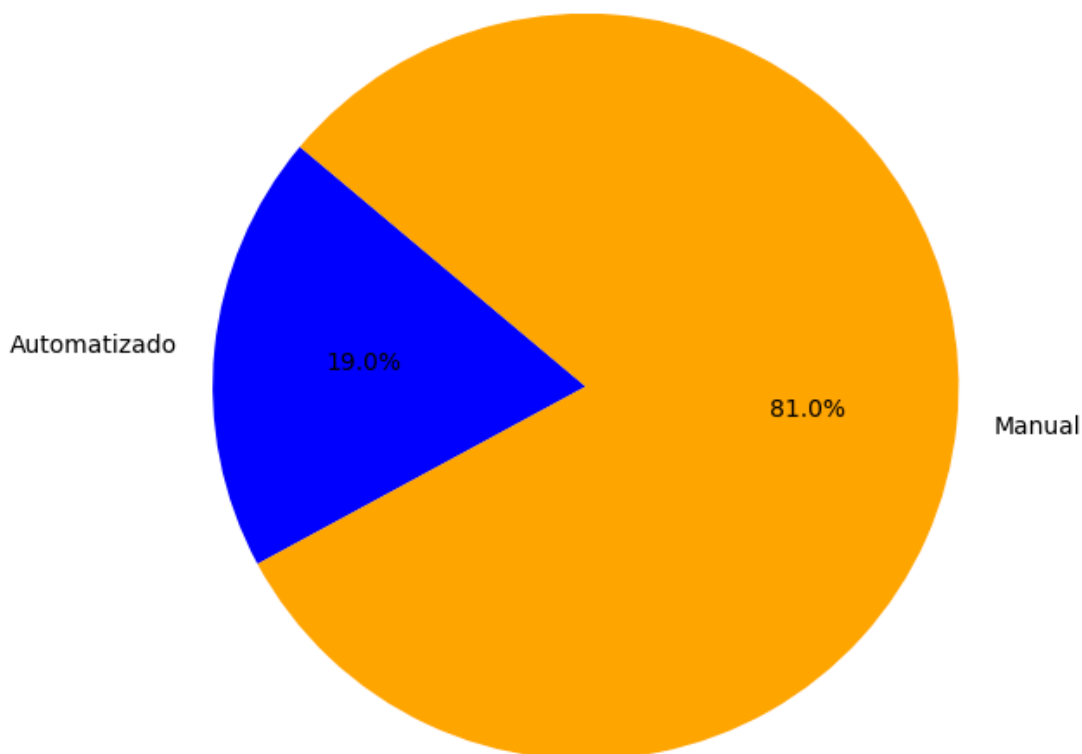


Fonte: Elaborado pelo Autor (2024)

Em todos os casos, os testes automatizados foram significativamente mais rápidos do que os testes manuais. O ganho de eficiência varia conforme a complexidade do caso de teste, mas o impacto é notável, especialmente em testes mais extensos. Por exemplo, o `test_ct07_create_lead`, que manualmente levaria algo em torno de 5 minutos e 18 segundos, foi executado automaticamente em apenas 1 minuto e 3 segundos, representando uma economia de cerca de 80% no tempo de execução.

O tempo total necessário para a execução automatizada de todos os testes foi de 7 minutos e 49 segundos, enquanto o mesmo conjunto de testes executado manualmente demandou 41 minutos e 10 segundos. Isso indica que a automação levou apenas cerca de 19% do tempo requerido para a execução manual, reforçando a eficácia da automação para otimização do tempo, conforme demonstrado no gráfico 2.

Gráfico 2 – Proporção do tempo total gasto em testes automatizados versus testes manuais.



Fonte: Elaborado pelo Autor (2024)

Isso mostra que automação não apenas reduz o tempo, mas também elimina a variabilidade e o risco de erros humanos, proporcionando consistência nos resultados. Esses ganhos de desempenho tornam a automação uma ferramenta indispensável para ambientes de teste que necessitam de repetição frequente e escalabilidade. A análise dos resultados também demonstra a viabilidade e a eficiência do uso de ferramentas como Selenium e Pytest no CRM Salesforce. Ao gerar relatórios detalhados e garantir a repetibilidade dos testes, a automação não apenas aumenta a produtividade, mas também melhora a confiabilidade das validações realizadas.

7 CONCLUSÃO

Este trabalho explorou a automação de casos de teste funcionais no sistema CRM Salesforce, com a criação de dez casos de teste que abrangem integralmente o fluxo do usuário em três objetos distintos. Esses testes foram executados no navegador Google Chrome, com o registro detalhado do tempo de execução total e individual de cada caso em um relatório HTML gerado automaticamente ao término dos testes.

A automação trouxe benefícios significativos, especialmente na otimização do tempo de execução, com testes automatizados sendo aproximadamente cinco vezes mais rápidos que os manuais. Além disso, o reuso dos testes desenvolvidos permite sua execução repetida sem a necessidade de intervenção humana, o que reduz custos e minimiza riscos de cobertura insuficiente das funcionalidades críticas do sistema. Entretanto, destaca-se que a automação não substitui completamente os testes manuais, já que certos aspectos de um sistema ainda demandam validação não automatizáveis.

As principais contribuições deste trabalho são:

1. Viabilidade de um design de projeto estruturado para testes automatizados, que prioriza a reutilização de funções, a separação entre dados e funcionalidades, e o aproveitamento das capacidades do Pytest integrado ao Selenium Webdriver;
2. Preparação de um ambiente que simplifica a escrita e execução dos testes, beneficiando futuros desenvolvedores e analistas de qualidade;
3. Prova de conceito da automação de cenários de teste no CRM Salesforce, ampliando as referências disponíveis sobre o tema, que ainda carece de abordagens técnicas na literatura e na comunidade técnica.

O projeto demonstrou que é viável criar e manter testes escalonáveis, utilizando ferramentas apropriadas. A escolha do Aqua IDE, da linguagem Python e do framework Selenium revelou-se eficaz na implementação dos testes. Além disso, a adoção do POM possibilitou uma organização mais limpa e estruturada do código, promovendo maior reutilização e legibilidade. O uso do Pytest, com funcionalidades como *fixtures* e *marks*, aliado ao Pytest-bdd, contribuiu para uma execução mais eficiente e facilitou o entendimento da equipe. Assim, este trabalho reforça que, com ferramentas adequadas e um planejamento estruturado, é possível implementar e escalar a automação de testes, garantindo a qualidade e a confiabilidade do software.

Para trabalhos futuros, destacam-se oportunidades como a integração da automação em pipelines de integração contínua, permitindo a execução automática dos testes a cada nova versão disponibilizada no ambiente. Além disso, a inclusão de testes para as versões mobile e tablet do sistema ampliaria a cobertura, garantindo validações completas em múltiplos dispositivos.

REFERÊNCIAS

AWS. **O que é Python?**. Disponível em: <<https://aws.amazon.com/pt/what-is/python/>>.

Acesso em: 05 out. 2024.

BAUDSON, A. J. G. S. **Algoritmos e programação**. Ouro Preto: Instituto Federal de Educação, Ciência e Tecnologia de Minas Gerais, 2013. *E-book*. Disponível em:

<https://nuted.ouropreto.ifmg.edu.br/capacitar/pluginfile.php/134/mod_glossary/attachment/1/algoritmos_programacao.pdf>. Acesso em: 03 abr. 2024.

BERNARDO, P. C.; KON, F. A Importância dos Testes Automatizados. [s.l.]: **Engenharia de Software Magazine**, 2008. Disponível em:

<<https://www.ime.usp.br/~kon/papers/EngSoftMagazine-IntroducaoTestes.pdf>>. Acesso em: 03 abr. 2024.

CARVALHO, D. M.; MARQUES, D.. **Proposta de uso da técnica BDD para otimizar a escrita e automação de testes no framework Scrum**. 2019. Trabalho de Conclusão de Curso - Instituto Federal de Educação Ciência e Tecnologia de São Paulo, Campus Hortolândia São Paulo, 2019. Disponível em:

<https://hto.ifsp.edu.br/portal/images/thumbnails/images/IFSP/Cursos/Coord_ADS/Arquivos/TCCs/2019/TCC_Dumas_Morais_de_Carvalho.pdf>. Acesso em: 03 abr. 2024.

CARVALHO, L.. **Automação de Testes Web com Selenium Webdriver e Python**. [s.l.]: Udemy, mar. 2023. Disponível em:

<<https://www.udemy.com/course/automacao-de-testes-selenium-webdriver-python>>. Acesso em: 04 maio 2024.

CHICANELLI, R. et al. Aspectos sociais, humanos e econômicos da utilização de testes automatizados no desenvolvimento de sistemas. **Revista Iberoamericana de Sistemas, Cibernética e Informática**, 2019. Disponível em:

<<https://www.iiisci.org/journal/pdv/risci/pdfs/CA234CS19.pdf>>. Acesso em: 17 out. 2024.

JETBRAINS. **O IDE para automação de testes**. Disponível em:

<<https://www.jetbrains.com/pt-br/aqua/>>. Acesso em: 12 out. 2024.

JETBRAINS. **Selenium support in Aqua**. Disponível em:

<<https://lp.jetbrains.com/selenium-1591/>>. Acesso em: 12 out. 2024.

MDN WEB DOCS. **DOM**. Disponível em:

<<https://developer.mozilla.org/pt-BR/docs/Glossary/DOM>>. Acesso em: 05 out. 2024.

MYERS, G. J.; BADGETT, T.; SANDLER, C. **The Art of Software Testing**. 3. ed. Hoboken, New Jersey: John Wiley & Sons, Inc, 2012. *E-book*. Disponível em:

<<https://malenezi.github.io/malenezi/SE401/Books/114-the-art-of-software-testing-3-edition.pdf>>. Acesso em: 23 mar. 2024.

NETO, Arilo. Introdução a Teste de Software. **Engenharia de Software Magazine**, nº 1, 2008. Disponível em:

<https://edisciplinas.usp.br/pluginfile.php/3503764/mod_resource/content/3/Introducao_a_Teste_de_Software.pdf>. Acesso em: 17 out. 2024.

OKKEN, B. **Python Testing with Pytest. Simple, Rapid, Effective, and Scalable**. 1. ed.

Raleigh: The Pragmatic Programmers, LLC, 2017. *E-book*. Disponível em:

<<http://www.3testing.com/doc/23.pdf>>. Acesso em: 03 abr. 2024.

PRESSMAN, Roger S. **Engenharia de software: uma abordagem profissional**. 7. ed. São Paulo: McGraw-Hill, 2011. *E-book*. Disponível em:

<<https://github.com/user-attachments/files/16191853/engenharia-de-software-pressman-9-edicao-portugues.pdf>>. Acesso em: 17 out. 2024.

PYTHON. **O que é Python?**. Disponível em:

<<https://docs.python.org/pt-br/dev/faq/general.html>>. Acesso em: 05 out. 2024.

RED HAT. **O que é IDE?**. Disponível em:

<<https://www.redhat.com/pt-br/topics/middleware/what-is-ide>>. Acesso em: 12 out. 2024.

SAINI, Manish. **Page Object Model and Page Factory in Selenium Python**. Disponível em: <<https://www.browserstack.com/guide/page-object-model-in-selenium-python#toc0>>. Acesso em: 01 abr. 2024.

SELENIUM. **Começando**. Disponível em: <https://www.selenium.dev/pt-br/documentation/webdriver/getting_started/>. Acesso em: 20 out. 2024.

SELENIUM. **Encontrando Elementos Web**. Disponível em: <<https://www.selenium.dev/pt-br/documentation/webdriver/elements/finders/>>. Acesso em: 05 out. 2024.

SELENIUM. **Localizando elementos**. Disponível em: <<https://www.selenium.dev/pt-br/documentation/webdriver/elements/locators/>>. Acesso em: 05 out. 2024.

SELENIUM. **Modelos de objetos de página**. Disponível em: <https://www.selenium.dev/pt-br/documentation/test_practices/encouraged/page_object_models/>. Acesso em: 20 out. 2024.

SMART, John F. **BDD in Action: Behavior Driven Development for the whole software lifecycle**. Manning Publications Co. NY. 2015. *E-book*. Disponível em: <<https://github.com/dashpradeep99/https-github.com-miguellgt-books/blob/master/tdd%20%2B%20bdd/bdd-in-action.pdf>>. Acesso em: 03 abr. 2024.

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011. *E-book*. Disponível em: <<https://www.facom.ufu.br/~william/Disciplinas%202018-2/BSI-GSI030-EngenhariaSoftware/Livro/engenhariaSoftwareSommerville.pdf>>. Acesso em: 03 abr. 2024.

TESTING COMPANY. **Conheça 10 ferramentas essenciais para testar seu software em 2024**. Disponível em: <<https://testingcompany.com.br/blog/conheca-10-ferramentas-essenciais-para-testar-seu-software-em-2024>>. Acesso em: 05 out. 2024.

APÊNDICE A – Casos de Teste

Quadro 1 – Caso de teste 01 (login com sucesso)

Realizar o login na plataforma com sucesso			
Nome da Feature	ct01_login_sucesso.feature	Comando para Rodar	pytest -m login
Nome do Teste	test_ct01_valid_login.py	Pré-condições	- Acesso a internet; - Preencher as credenciais de login em .env.
Nome da Página	login_page.py		
Passo	Ação	Resultado Esperado	
1	Navegar até a página de login da org no Salesforce	Página de login é exibida	
2	Preencher o campo de email com um email válido	Email inserido no campo correspondente	
3	Preencher o campo de senha com uma senha válida	Senha inserida no campo correspondente	
4	Clicar no botão "fazer login"	Ação de login é iniciada	
5	Autenticação realizada com sucesso	Login realizado com sucesso e redirecionamento para a página inicial	

Fonte: Elaborado pelo autor (2024)

Quadro 2 – Caso de teste 02 (verificação das *labels* na criação de conta)

Verificação das <i>labels</i> de criação de Conta			
Nome da Feature	ct02_labels_create_accoun nt.feature	Comando para Rodar	pytest -m labels_create_account
Nome do Teste	test_ct02_labels_create_a ccount.py	Pré-condições	- Acesso a internet; - Arquivo JSON com a especificação dos campos e picklists a serem verificados.
Nome da Página	account_page.py		
Passo	Ação	Resultado Esperado	

1	Acessar a página de cadastro de conta após o login	Página de cadastro de conta é exibida
2	Acessar o formulário de criação de conta	Formulário de criação de conta é exibido
3	Verificar se todas as <i>labels</i> de criação de conta estão visíveis	Todas as <i>labels</i> necessárias estão visíveis
4	Verificar se as picklists contêm as opções corretas	Todas as picklists contêm as opções corretas e estão visíveis

Fonte: Elaborado pelo autor (2024)

Quadro 3 – Caso de teste 03 (criação de conta e verificação dos campos)

Criação de conta e verificação dos campos			
Nome da Feature	ct03_create_account.feature	Comando para Rodar	pytest -m labels_create_account
Nome do Teste	test_ct03_create_account.py	Pré-condições	- Acesso a internet; - Disponibilidade de um arquivo JSON contendo a especificação dos campos e seus respectivos valores para preenchimento.
Nome da Página	account_page.py		
Passo	Ação	Resultado Esperado	
1	Acessar a página de cadastro de conta após o login	Página de cadastro de conta é exibida	
2	Acessar o formulário de criação de conta	Formulário de criação de conta é exibido	
3	Preencher todos os campos obrigatórios com informações válidas	Todos os campos são preenchidos com os dados fornecidos	
4	Clicar em "salvar" para criar a conta	Ação de criação de conta é iniciada	
5	Verificar se a conta foi criada com sucesso	Conta é criada e confirmação de criação é exibida	

6	Verificar se todos os campos preenchidos estão presentes nos detalhes da conta	Todos os campos aparecem nos detalhes da conta recém-criada
---	--	---

Fonte: Elaborado pelo autor (2024)

Quadro 4 – Caso de teste 04 (verificação das labels na criação de oportunidade)

Verificação das <i>labels</i> de criação de Oportunidade			
Nome da Feature	ct04_labels_create_opportunity.feature	Comando para Rodar	pytest -m labels_create_opportunity
Nome do Teste	test_ct04_labels_create_opportunity.py	Pré-condições	- Acesso a internet; - Arquivo JSON com a especificação dos campos e picklists a serem verificados.
Nome da Página	opportunity_page.py		
Passo	Ação	Resultado Esperado	
1	Acessar a página de cadastro de oportunidade após o login	Página de cadastro de oportunidade é exibida	
2	Acessar o formulário de criação de oportunidade	Formulário de criação de oportunidade é exibido	
3	Verificar se todas as <i>labels</i> de criação de oportunidade estão visíveis	Todas as <i>labels</i> necessárias estão visíveis	
4	Verificar se as picklists contêm as opções corretas	Todas as picklists contêm as opções corretas e estão visíveis	

Fonte: Elaborado pelo autor (2024)

Quadro 5 – Caso de teste 05 (criação de oportunidade e verificação dos campos)

Criação de oportunidade e verificação dos campos			
Nome da Feature	test_ct05_create_opportunity.py	Comando para Rodar	pytest -m create_opportunity

Nome do Teste	test_ct05_create_opportunity.py	Pré-condições	- Acesso a internet; - Disponibilidade de um arquivo JSON contendo a especificação dos campos e seus respectivos valores para preenchimento.
Nome da Página	lead_page.py		
Passo	Ação	Resultado Esperado	
1	Acessar a página de cadastro de oportunidade após o login	Página de cadastro de oportunidade é exibida	
2	Acessar o formulário de criação de oportunidade	Formulário de criação de oportunidade é exibido	
3	Preencher todos os campos obrigatórios com informações válidas	Todos os campos são preenchidos com os dados fornecidos	
4	Clicar em "salvar" para criar a oportunidade	Ação de criação de oportunidade é iniciada	
5	Verificar se a oportunidade foi criada com sucesso	Oportunidade é criada e confirmação de criação é exibida	
6	Verificar se todos os campos preenchidos estão presentes nos detalhes da oportunidade	Todos os campos aparecem nos detalhes da oportunidade recém-criada	

Fonte: Elaborado pelo autor (2024)

Quadro 6 – Caso de teste 06 (verificação das *labels* na criação de lead)

Verificação das labels de criação de lead			
Nome da Feature	ct06_labels_create_lead.feature	Comando para Rodar	pytest -m labels_create_lead
Nome do Teste	test_ct06_verify_labels_create_lead.py	Pré-condições	- Acesso a internet; - Arquivo JSON com a especificação dos campos e picklists a serem verificados.
Nome da Página	lead_page.py		
Passo	Ação	Resultado Esperado	
1	Acessar a página de cadastro de lead após o	Página de cadastro de lead é exibida	

	login	
2	Acessar o formulário de criação de lead	Formulário de criação de lead é exibido
3	Verificar se todas as <i>labels</i> de criação de lead estão visíveis	Todas as <i>labels</i> necessárias estão visíveis
4	Verificar se as picklists contêm as opções corretas	Todas as picklists contêm as opções corretas e estão visíveis

Fonte: Elaborado pelo autor (2024)

Quadro 7 – Caso de teste 07 (criação de lead e verificação dos campos)

Criação de lead e verificação dos campos			
Nome da Feature	ct07_create_lead.feature	Comando para Rodar	pytest -m create_lead
Nome do Teste	test_ct07_create_lead.py	Pré-condições	- Acesso a internet; - Disponibilidade de um arquivo JSON contendo a especificação dos campos e seus respectivos valores para preenchimento.
Nome da Página	lead_page.py		
Passo	Ação	Resultado Esperado	
1	Acessar a página de cadastro de lead após o login	Página de cadastro de lead é exibida	
2	Acessar o formulário de criação de lead	Formulário de criação de lead é exibido	
3	Preencher todos os campos obrigatórios com informações válidas	Todos os campos são preenchidos com os dados fornecidos	
4	Clicar em "salvar" para criar o lead	Ação de criação do lead é iniciada	
5	Verificar se o lead foi criado com sucesso	Lead é criado e confirmação de criação é exibida	

6	Verificar se todos os campos preenchidos estão presentes nos detalhes do lead	Todos os campos aparecem nos detalhes do lead recém-criado
---	---	--

Fonte: Elaborado pelo autor (2024)

Quadro 8 – Caso de teste 08 (edição de conta e verificação dos campos)

Edição de conta e verificação dos campos			
Nome da Feature	ct08_edit_account.feature	Comando para Rodar	pytest -m edit_account
Nome do Teste	test_ct08_edit_account.py	Pré-condições	- Acesso a internet; - Disponibilidade de um novo arquivo JSON contendo a especificação dos campos e seus respectivos valores para preenchimento na edição; - Conta existente para edição.
Nome da Página	account_page.py		
Passo	Ação	Resultado Esperado	
1	Acessar a página de lista de contas após o login	Página de lista de contas é exibida	
2	Acessar uma conta existente	Detalhes da conta existente são exibidos	
3	Clicar no botão de editar a conta	Ação de edição da conta é iniciada	
4	Preencher os campos do formulário de edição com as novas informações	Campos são preenchidos com as novas informações fornecidas	
5	Clicar em "salvar" para salvar as alterações	Alterações são salvas com sucesso	
6	Verificar se a conta foi atualizada com sucesso	Verificar se o número de campos atualizados bate com os campos inseridos	
7	Verificar se todos os campos refletem as novas informações nos detalhes	Todos os campos nos detalhes da conta mostram as novas informações	

	da conta	
--	----------	--

Fonte: Elaborado pelo autor (2024)

Quadro 9 – Caso de teste 09 (edição de oportunidade e verificação dos campos)

Edição de oportunidade e verificação dos campos			
Nome da Feature	ct09_edit_opportunity.feature	Comando para Rodar	pytest -m edit_opportunity
Nome do Teste	test_ct09_edit_opportunity.py	Pré-condições	- Acesso a internet; - Disponibilidade de um novo arquivo JSON contendo a especificação dos campos e seus respectivos valores para preenchimento na edição; - Oportunidade existente para edição.
Nome da Página	opportunity_page.py		
Passo	Ação	Resultado Esperado	
1	Acessar a página de lista de oportunidades após o login	Página de lista de oportunidades é exibida	
2	Acessar uma oportunidade existente	Detalhes da oportunidade existente são exibidos	
3	Clicar no botão de editar a oportunidade	Ação de edição da oportunidade é iniciada	
4	Preencher os campos do formulário de edição com as novas informações	Campos são preenchidos com as novas informações fornecidas	
5	Clicar em "salvar" para salvar as alterações	Alterações são salvas com sucesso	
6	Verificar se a oportunidade foi atualizada com sucesso	Verificar se o número de campos atualizados bate com os campos inseridos	
7	Verificar se todos os campos refletem as novas informações nos detalhes da oportunidade	Todos os campos nos detalhes da oportunidade mostram as novas informações	

Fonte: Elaborado pelo autor (2024)

Quadro 10 – Caso de teste 10 (edição de lead e verificação dos campos)

Edição de lead e verificação dos campos			
Nome da Feature	ct10_edit_lead.feature	Comando para Rodar	pytest -m edit_opportunity
Nome do Teste	test_ct10_edit_lead.py	Pré-condições	- Acesso a internet; - Disponibilidade de um novo arquivo JSON contendo a especificação dos campos e seus respectivos valores para preenchimento na edição; - Lead existente para edição.
Nome da Página	lead_page.py		
Passo	Ação	Resultado Esperado	
1	Acessar a página de lista de lead após o login	Página de lista de leads é exibida	
2	Acessar um lead existente	Detalhes do lead existente são exibidos	
3	Clicar no botão de editar o lead	Ação de edição do lead é iniciada	
4	Preencher os campos do formulário de edição com as novas informações	Campos são preenchidos com as novas informações fornecidas	
5	Clicar em "salvar" para salvar as alterações	Alterações são salvas com sucesso	
6	Verificar se o lead foi atualizada com sucesso	Verificar se o número de campos atualizados bate com os campos inseridos	
7	Verificar se todos os campos refletem as novas informações nos detalhes do lead	Todos os campos nos detalhes do lead mostram as novas informações	

Fonte: Elaborado pelo autor (2024)