UNIVERSIDADE ESTADUAL DA PARAÍBA
CAMPUS I - CAMPINA GRANDE
CENTRO DE CIÊNCIA E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE GRADUAÇÃO EM BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

HENRIQUE JOSÉ DOS SANTOS MARTINS

AUTOMATIZAÇÃO DE TESTES FUNCIONAIS: UM RELATO DE EXPERIÊNCIA

CAMPINA GRANDE
2024

UNIVERSIDADE ESTADUAL DA PARAÍBA
CAMPUS I - CAMPINA GRANDE
CENTRO DE CIÊNCIA E TECNOLOGIA

HENRIQUE JOSÉ DOS SANTOS MARTINS

AUTOMATIZAÇÃO DE TESTES FUNCIONAIS: UM RELATO DE EXPERIÊNCIA

Relatório de Estágio apresentado à Coordenação do Curso de Ciência da Computação da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de Bacharel em Computação

Área de concentração: Engenharia de Software

Orientador: Prof. Ana Isabella Muniz Leite

CAMPINA GRANDE
2024

M386a Martins, Henrique Jose dos Santos.
Automatização de testes funcionais [manuscrito] : um relato de experiência / Henrique Jose dos Santos Martins. - 2024.
54 f. : il. color.

Digitado.

Relatório de Estágio (Graduação em Ciência da computação) - Universidade Estadual da Paraíba, Centro de Ciências e Tecnologia, 2024.

"Orientação : Prof. Dra. Ana Isabella Muniz Leite, Departamento de Computação - CCT".

1. Automação de testes. 2. Testes Funcionais. 3. Qualidade de software. I. Título

21. ed. CDD 005.14

Elaborada por Bruno Rafael Freitas de Lima - CRB - 15/1021 BC

HENRIQUE JOSE DOS SANTOS MARTINS

AUTOMATIZAÇÃO DE TESTES FUNCIONAIS: UM RELATO DE EXPERIÊNCIA

Relatório de Estágio apresentado à Coordenação do Curso de Ciência da Computação da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de Bacharel em Computação

Aprovada em: 26/02/2025.

BANCA EXAMINADORA

Documento assinado eletronicamente por:

- **Sabrina de Figueirêdo Souto** (***.047.964-**), em **13/03/2025 11:48:19** com chave **33ffebfc001a11f0905006adb0a3afce**.
- **Fábio Luiz Leite Júnior** (***.848.564-**), em **12/03/2025 22:35:12** com chave **67fcc9b2ffab11ef92a41a7cc27eb1f9**.
- **Ana Isabella Muniz Leite** (***.834.864-**), em **12/03/2025 15:59:55** com chave **2fe6d252ff7411efbcdc06adb0a3afce**.

Documento emitido pelo SUAP. Para comprovar sua autenticidade, faça a leitura do QrCode ao lado ou acesse https://suap.uepb.edu.br/comum/autenticar_documento/ e informe os dados a seguir.
Tipo de Documento: Folha de Aprovação do Projeto Final
Data da Emissão: 13/03/2025
Código de Autenticação: 0a6bac

Dedico a Deus por não me deixar desistir nessa jornada. Dedico também à minha família que me deu o suporte durante essa caminhada.

AGRADECIMENTOS

A Deus, pela minha vida, por estar sempre me amparando em todos os momentos e também a Nossa Senhora, onde busquei refúgio nos momentos de atribulações. A meus pais e irmão, por está sempre ao meu lado e ser minha base durante toda a jornada acadêmica. A Profa. Ana Isabella que me acolheu em seu projeto e puxou minha orelha quando eu não atualizava o Jira. A Luana que tornou a caminhada do curso mais fácil com conversa, ajuda e conselhos. A Edilson que me indicou ao projeto e me ajudou no processo de realização dos testes.

"O aprendizado é fruto da repetição e dos erros, não do perfeccionismo." (Angela Duckworth)

RESUMO

O Teste Funcional de software é utilizado para verificar se o software em nível de usuário está operando de acordo com as especificações e se todos as entradas e saídas, estão funcionando corretamente. Considerando que as soluções baseadas em software estão se tornando cada vez mais complexas, testá-las é uma atividade difícil, intrincada e demorada. Por essa razão, os testes muitas vezes são realizados de forma inadequada ou até mesmo ignorados pelos profissionais. A automação de testes torna-se imprescendível para essa situação e assim, tornar o ambiente de produção do sistema mais ágil e menos suscetível a erros. Este trabalho apresenta observações empíricas e os desafios enfrentados por uma equipe de testes que era nova nas práticas ágeis e na automação de testes, utilizando ferramentas de teste de código aberto integradas aplicadas no contexto do projeto Senior Saúde Móvel. Os resultados obtidos contribuiram para a criação de um ambiente de teste automatizado, permitindo assim a detecção de erros mais rápidos e com menos esforço dos desenvolvedores.

Palavras-chave: automação de testes; testes funcionais; qualidade de software;

ABSTRACT

Software Functional Testing is used to verify whether the software at the user level is operating according to specifications and whether all inputs and outputs are functioning correctly. Since software-based solutions are becoming increasingly complex, testing them is difficult, intricate, and time-consuming. For this reason, testing is often performed inadequately or even overlooked by professionals. Test automation becomes essential in this scenario, making the system's production environment more agile and less prone to errors. This work presents empirical observations and the challenges faced by a test team new to agile practices and test automation, using integrated open-source testing tools within the Senior Saúde Móvel project. The results obtained contributed to creating an automated testing environment, enabling faster error detection with less effort from developers.

Keywords: test automation; functional test; software quality.

SUMÁRIO

Página

1 INTRODUÇÃO 10

2 FUNDAMENTAÇÃO TEÓRICA 12
2.1 Testes de Software . . . 12
2.1.1 Testes Funcionais . . . 13
2.1.2 Automação de Testes . . . 14

3 METODOLOGIA 17
3.1 Fase 1: Automação com Selenium WebDriver . . . 17
3.2 Fase 2: Automação com Cypress . . . 18

4 AUTOMATIZAÇÃO DOS TESTES FUNCIONAIS COM O SELENIUM 20
4.1 Desafios . . . 20
4.2 Configuração Inicial . . . 21
4.2.1 Criação de testes . . . 24
4.3 Desenvolvimento com Selenium WebDriver e JUnit . . . 24
4.3.1 Configuração do Projeto . . . 25
4.3.2 Implementação dos Testes . . . 26
4.3.3 Execução e Validação . . . 28
4.4 Integração Contínua com Jenkins . . . 29
4.4.1 Criação de Pipelines . . . 29
4.4.2 Configuração de Jobs . . . 30
4.5 Gerenciamento de Testes com AIO Test no Jira . . . 31

5 AUTOMATIZAÇÃO DOS TESTES FUNCIONAIS CO O CYPRESS 33
5.1 Desafios . . . 33
5.2 Processos de Automação . . . 34
5.2.1 Instalação e Configuração do Cypress com Cucumber . . . 34
5.2.2 Estruturação dos Testes . . . 35
5.2.3 Desenvolvimento dos Step Definitions . . . 37
5.2.4 Execução e Integração com Jenkins . . . 38
5.2.5 Integração com AIO Test no Jira . . . 40

6 DISCUSSÃO 41

7 CONCLUSÃO 43

REFERÊNCIAS 44

A APÊNDICE A - AIO tests para o Selenium 45

B APÊNDICE B - AIO tests para o Cypress 50

1 INTRODUÇÃO

Nos ambientes de desenvolvimento de software, a qualidade do software é um aspecto fundamental que influencia diretamente a satisfação do usuário final e a viabilidade do produto no mercado (Atoum et al. 2021). A garantia da qualidade de software envolve várias práticas e processos que visam assegurar que o produto final esteja conforme os requisitos especificados e livre de defeitos. Uma das práticas fundamentais para garantir a qualidade do software é a realização de testes (Bhanushali 2023). O teste de software acompanha todo o processo de desenvolvimento, ajudando a garantir que tudo funcione como esperado. Mas é na fase final que ele ganha ainda mais importância, pois essa etapa que dá o aval para o produto ser entregue ao usuário com confiança e qualidade. O teste de software é um processo de avaliar e verificar se o software está funcionando como esperado e atendendo aos requisitos propostos para então, assim, atestar a qualidade do produto. A atividade de teste de software envolve a execução de uma implementação do software com os dados de teste e a análise de suas saídas e de seu comportamento operacional, a fim de verificar se foi executada conforme o esperado (Ian 2011), tais como testes unitários, testes de integração, teste de sistema e teste funcional, todos focados em diferentes aspectos do software. Acordo com (Atoum et al. 2021), a implementação de testes ao longo de um desenvolvimentode sofware contínuo é trabalhoso e custoso, sendo muitas vezes repetitivos e propensos a erros humanos. Por isso faz-se necessário a realização da automação dos testes.

A automação dos testes torna-se essencial para aumentar a eficiência, garantindo maior cobertura e consistência na execução, o que contribui diretamente para a confiabilidade do projeto. Além disso, reduz significativamente a probabilidade de erros humanos que podem ocorrer durante a execução manual dos testes (Ian 2011), assegurando que os testes sejam realizados de forma precisa e uniforme. Com esse objetivo, foram utilizados frameworks para a gravação dos passos e criação dos testes, permitindo a automatização dos testes funcionais previamente escritos e reforçando a qualidade do processo

Diante desse cenário, este trabalho visa, conforme o GQM (Goal-Question-Metric), *automatizar* os testes funcionais *com o propósito de* melhorar a qualidade (confiabilidade) do projeto, *com respeito a* detecção de falhas imediata *a partir do ponto de* vista do time de desenvolvimento e testadores *no contexto* do projeto da Senior Saúde Móvel[1].

A sênior é uma plataforma voltada para a inovação no cuidado com a saúde de idosos, utilizando tecnologia para aprimorar o monitoramento e a qualidade dos atendimentos fisioterapêuticos. Para oferecer um acompanhamento contínuo e mais eficiente, foi desenvolvido um sistema de monitoramento remoto que capta informações do paciente 24 horas por dia por meio de dispositivos inteligentes, como relógios e pulseiras. Essa tecnologia

[1] https://www.seniorsaudemovel.com.br/

permite a coleta de dados em tempo real, auxiliando na predição de condições adversas à saúde e na tomada de decisões mais assertivas por parte dos profissionais de fisioterapia.

Este trabalho descreve os passos seguidos para a implementação da automação dos testes funcionais, os benefícios observados em termo de qualidade de software no tocante a funcionalidade, os desafios encontrados durante a implementação e como foram superados e as recomendações práticas para as equipes que desejam implementar. Mais especificamente, descrever o caminho percorrido para a implementação da automação dos testes funcionais e as integrações necessárias para a geração dos relatórios no software de monitoramento das tarefas.

Para isso, este trabalho adota uma abordagem de relato de experiência para descrever o processo da automação de testes funcionais com Selenium-Jenkins-AIO tests e Cypress-Jenkins-AIO tests na geração de relatórios. O trabalho foi conduzido em um ambiente de desenvolvimento de software real voltado para a relatar o impacto da automação de testes funcionais no processo de desenvolvimento de software na Sênior Saúde móvel. Os casos de testes foram especificados com base nas principais funcionalidades do ambiente da Senior. A seleção foi orientada a partir do que se esperava e o que seria realizado pela função. A seleção dos testes foi documentada e organizada em casos de teste, sendo dividida em duas categorias: casos funcionais, destinados à automação, e casos de usabilidade. Com os testes devidamente selecionados, iniciou-se o processo de automação.

Os resultados obtidos demonstraram uma melhor documentação dos testes funcionais, os relatórios criados contribuíram para uma melhor organização, manutenção e confiabilidade do projeto com a detecção de erros. Além disso, um processo de testes foi implementado no projeto, a partir da implementação de um modelo padronizado e um ambiente já configurado para a produção dos testes funcionais futuros. Mostra também, de forma prática, o caminho para a implementação de ambientes de testes completo, bem como suas dificuldades e melhorias.

O presente trabalho está estruturado da seguinte forma: os capítulos 1, 2 e 3 compõem a introdução, abordando a apresentação do trabalho, a metodologia utilizada e a fundamentação teórica. Na segunda parte, nos capítulos 4 e 5, é desenvolvido o processo de automação dos testes para cada ferramenta. Por fim, são apresentados os resultados e a conclusão, destacando os principais achados e aprendizados obtidos ao longo do estudo.

2 FUNDAMENTAÇÃO TEÓRICA

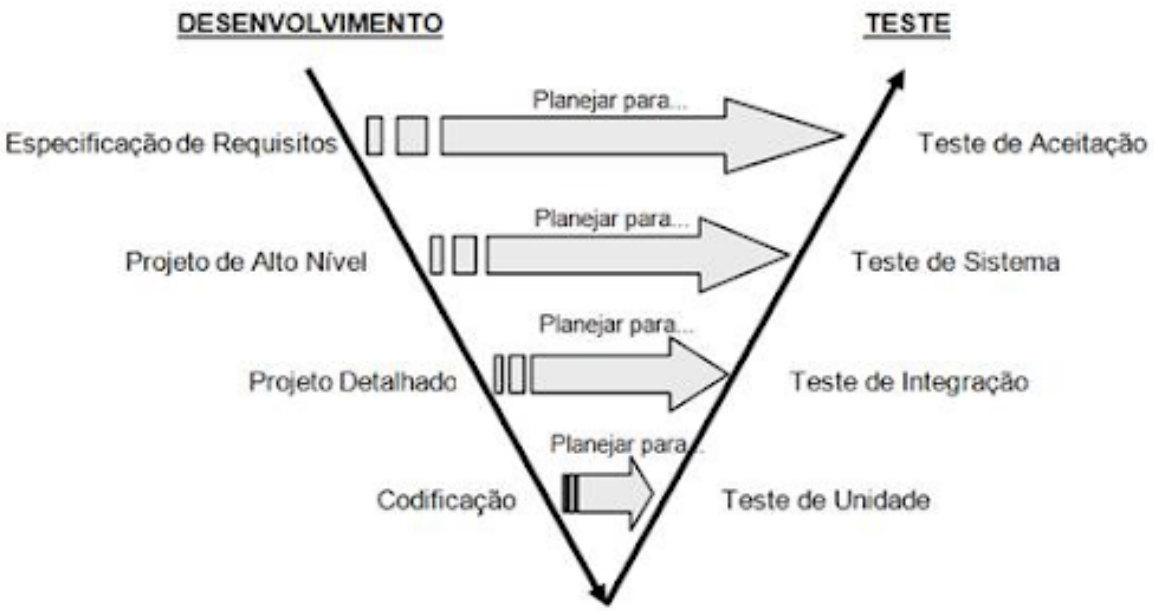
A fundamentação teórica abordará os conceitos de teste de software, com um foco aprofundado nos testes funcionais e na automação de testes. Esta seção examinará como esses conceitos são tratados na literatura e aplicados no desenvolvimento de software, oferecendo uma base teórica sólida para a pesquisa e situando-a no contexto acadêmico e prático atual.

2.1 Testes de Software

Testes de software são um processo, ou uma série de processos, projetados para garantir que o código de computador faça o que foi planejado para fazer e, inversamente, que não faça nada não intencionado. O software deve ser previsível e consistente, sem apresentar surpresas aos usuários. Testar é o processo de executar um programa com a intenção de encontrar erros.(Myers Tom Badgett 2012) Os testes de software são uma atividade essencial no ciclo de desenvolvimento, cujo objetivo é garantir a qualidade e a funcionalidade do sistema desenvolvido. O objetivo do teste é encontrar erros, e um bom teste é aquele que tem alta probabilidade de encontrar erros.(Pressman e Maxim 2021) As quatro atividades básicas para o processo de desenvolvimento de software: especificação, desenvolvimento, validação, e evolução. O teste de software está presente na validação. A parte da validação utiliza técnicas para investigar falhas em um sistema.(Ian 2011)

O processo V&V (Verificação e validação), imagem 1 é um processo que visa garantir a qualidade e a confiabilidade do produto final.(Sommerville, 2007) O processo de verificação é entendido como a avaliação de um sistema para determinar se ele atende aos requisitos proposto na documentação, enquanto a validação refere-se a confirmação de que o sistema está atendendo a expectativa do cliente.(Ian 2011) A figura 1 mostra o diagrama em V e onde cada tipo de teste é realizado. O estudo é focado nos teste de sistema e também nos de aceitação onde os testes funcionais se encaixam.

Figura 1 – Diagrama em V (Fonte: Adaptação de questão do Cebraspe com conceitos de metodologias de desenvolvimento de software.



Fonte: Adaptação de questão do Cebraspe com conceitos de metodologias de desenvolvimento de software.

- Teste de Unidade: é o processo de testar a menor unidade, possível, do sistema. O teste unitário é o processo de testar os componentes de um programa, como métodos ou classe do objeto.(Ian 2011)
- Teste de Integração: Ele procura falhas nos componentes quando estão conectados um com o outro.(Ian 2011)
- Teste de Sistema: Os teste de sistema verifica se os componentes são compatíveis, se ocorre a interação esperada e o envio correto de dados. É um teste onde o sistema já está completamente integrado, ele garante se a interface funciona como o esperado.(Ian 2011)
- Teste de Aceitação: É um teste usado para imitar o usuário final onde simula o uso do software para verificar o comportamento correto do sistema.

2.1.1 Testes Funcionais

Também conhecido como teste de caixa-preta, os teste funcionais focam no atendimento de requisitos funcionais do software. O teste caixa-preta não é uma forma diferente para o caixa-branca, já que não analisa a estrutura interna do software, mas sim complementar. Os testes funcionais são uma categoria de testes de software que se concentra na validação das funcionalidades do sistema em relação aos requisitos especificados. Os

testes funcionais verificam se cada função do software opera conforme as especificações, abordando aspectos como entradas e saídas, comportamento esperado e manuseio de erros. São feitos a partir da interface estável do software, com a visão do usuário final o teste verifica se as funcionalidades requeridas estão sendo atendidas na interface. Teste de caixa-preta funciona a partir de um dado de entrada é fornecido e uma saída de um dado é esperada, essa saída é comparada com o resultado esperado. É chamado caixa preta 2 devido a não ser necessário visualizar o que acontece na parte interna do software e sim sua funcionalidade na tela do usuário.

Figura 2 – Teste Caixa Preta (Fonte: https://www.testbirds.com/en/blog/black-box-testing-enhancing-user-experience/)

Fonte: https://www.testbirds.com/en/blog/black-box-testing-enhancing-user-experience/

2.1.2 Automação de Testes

Para se testar um software é necessário que esses testes sejam eficazes na detecção de defeitos, mas os testes devem ser feitos com o menor custo de tempo e econômico possível.(Graham e Fewster 1999) A automação de testes é um componente essencial em equipes ágeis, proporcionando uma maneira eficaz de garantir a qualidade contínua do software, "a automação libera as pessoas para fazerem o seu melhor trabalho".(Crispin e Gregory 2009) Isso acontece porque os testes manuais, além de demorados, estão sujeitos a erros humanos, enquanto os testes automatizados podem ser repetidos com precisão e rapidez, permitindo que a equipe se concentre em outras tarefas, como testes exploratórios e análise de falhas.

"Os testes automatizados de regressão fornecem uma rede de segurança"(Crispin e Gregory 2009), permitem que as equipes detectem rapidamente problemas introduzidos por novas funcionalidades ou mudanças. Esse tipo de feedback imediato é essencial em ciclos de desenvolvimento ágeis, onde mudanças frequentes são comuns e o risco de introdução de bugs aumenta.

Além disso, os testes automatizados ajudam a documentar o sistema. Quando implementados corretamente, eles servem como uma fonte de documentação viva, refletindo o

comportamento esperado do sistema em diferentes cenários. Conforme Crispin e Gregory, "os testes são uma excelente documentação"(Crispin e Gregory 2009), ajudando a equipe a entender o funcionamento do sistema em diversas camadas. Entretanto, implementar a automação de testes não é uma tarefa simples. Um dos maiores desafios enfrentados pelas equipes é superar a curva de aprendizado inicial. "A curva de aprendizado" é uma barreira significativa para a automação, especialmente para equipes que não têm experiência prévia com as ferramentas e práticas envolvidas.(Crispin e Gregory 2009) Além disso, a automação requer um investimento inicial significativo, tanto em termos de tempo quanto de recursos, para configurar o ambiente de testes e desenvolver os scripts iniciais.

Para o processo de automatização, foram selecionadas ferramentas para esse processo. Inicialmente, foi utilizado o Selenium, junto com o Jenkins e AIO tests; logo após, o Selenium foi substituído pelo Cypress e Cucumber.

o Selenium WebDriver se destaca como uma API orientada a objetos que interage diretamente com os navegadores, simulando o comportamento de um usuário real. Essa ferramenta permite a criação de scripts de teste mais complexos e robustos, oferecendo maior controle sobre as interações com a interface do usuário.(Selenium WebDriver documentação 2024)

O Jenkins é um servidor de automação de código aberto que facilita a construção, teste e implantação contínua de projetos de software. Ele oferece centenas de plugins que suportam a construção, implantação e automação de qualquer projeto, permitindo que as organizações acelerem o processo de desenvolvimento de software por meio da automação.(Documnetação do Jenkins 2024)

O Cypress é uma estrutura de automação de teste end-to-end construída e projetada para aplicações web modernas. e se concentra em eliminar inconsistências nos testes, garantindo que você possa escrever, depurar e executar testes no navegador sem precisar de configuração adicional ou pacotes extras.(Mwaura 2021)

Cypress foi construído para a web, tendo sido otimizado para rodar em navegadores. A arquitetura do Cypress permite que ele execute testes de forma eficiente, ao mesmo tempo que supera os desafios do WebDriver. Enquanto o Cypress tem o poder de rodar no navegador, o WebDriver interage com o navegador usando o protocolo HTTP, causando assim atrasos e eventos de espera desconhecidos durante a execução dos testes. O Cypress também é direcionado a engenheiros de QA e desenvolvedores que procuram escrever testes sem se preocupar com a infraestrutura subjacente e a limitação de uma única biblioteca de asserções e linguagem de programação.(Mwaura 2021)

O Cucumber, com sua sintaxe Gherkin, permite que os cenários de teste sejam descritos em uma linguagem natural, padronizada e de fácil compreensão. Isso garante que os testes sigam uma estrutura uniforme, facilitando a leitura e entendimento por toda a equipe, incluindo membros não técnicos. A documentação criada se torna executável, servindo simultaneamente como uma especificação do comportamento esperado do sistema e como

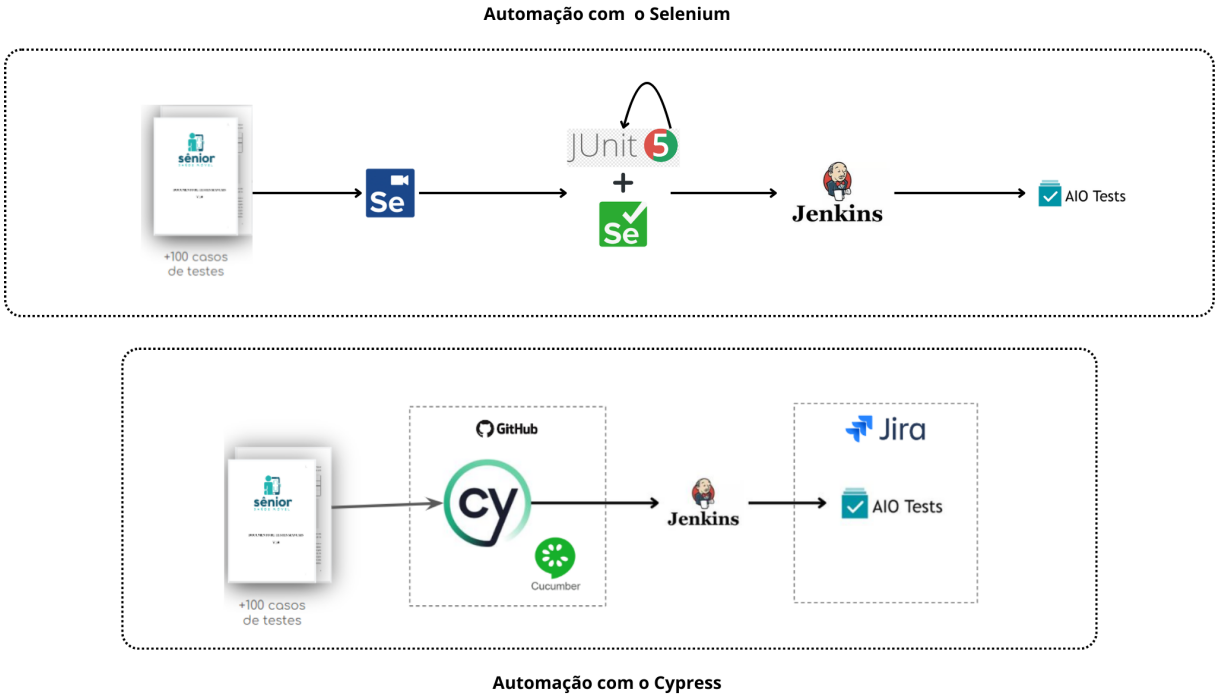
um meio de automatizar os testes. Dessa forma, garante-se que as funcionalidades estejam sempre alinhadas com os requisitos do projeto. Os testes Cucumber compartilham o benefício dos documentos de especificação tradicionais por poderem ser escritos e lidos por partes interessadas do negócio, mas têm uma vantagem distinta: você pode entregá-los a um computador a qualquer momento para saber o quão precisos eles são. Na prática, isso significa que sua documentação, em vez de ser algo que é escrito uma vez e depois gradualmente se torna obsoleta, torna-se algo vivo que reflete o verdadeiro estado do projeto.(Wynne Matt; Hellesøy 2012)

A automação de testes é uma prática fundamental no desenvolvimento moderno de software, contribuindo para a melhoria da qualidade, eficiência e confiabilidade dos sistemas. A implementação eficaz de automação de testes, combinada com uma estratégia robusta de testes funcionais, pode proporcionar um impacto significativo na entrega de software de alta qualidade.

3 METODOLOGIA

As atividades realizadas para a concretização desse trabalho, foram desenvolvidas seguindo o método ágil SCRUM[1], com reuniões de acompanhamento semanais e com Sprints de 2 semanas. Estas foram realizadas em duas fases (1) Automação com Selenium WebDriver e (2) Automação com Cypress, ambas integradas ao Jenkins para integração contínua e ao AIO Test no Jira para rastreamento e gerenciamento dos resultados de teste. Uma vez que, todas as atividades foram planejadas e acompanhadas pelo Jira[2].

Figura 1 – Ambientes de testes criados



Fonte: Elaborado pelo autor, 2024

3.1 Fase 1: Automação com Selenium WebDriver

O primeiro passo foi analisar o processo de testes manual existente e identificar os fluxos críticos e cenários repetitivos que seriam automatizados. Com base na análise, foram selecionados os cenários que mais se beneficiariam da automação. Esses cenários incluíram validações de login, navegação entre telas, cadastro de usuários e interações com formulários. Com o uso da extensão do Selenium IDE os passos para os testes foram gravados e exportados em código Junit, logo após os scripts de teste já exportados para JUnit, foram organizados e revisados. Cada caso de teste foi implementado para simular interações reais do usuário com a aplicação, como preenchimento de cam-

[1]https://www.scrum.org/
[2]https://www.atlassian.com/br/software/jira

pos, cliques em botões e validações de mensagens. Para otimizar o tempo de execução e garantir que os testes funcionassem em diferentes navegadores e plataformas, foi configurado o Selenium Grid.[3]O Selenium Grid é uma extensão poderosa que permite a execução paralela de testes. Isso é feito distribuindo casos de teste em vários nós com configurações únicas de navegador. Isso acelera os testes e reduz o tempo de execução. (Selenium grid: all you need to know 2024) Foi configurado um pipeline no Jenkins para monitorar o repositório GitHub e acionar a execução dos testes automatizados. O Jenkins executava os testes por meio do Selenium Grid, garantindo que a automação fosse feita continuamente. Os resultados dos testes executados no Jenkins foram automaticamente integrados ao AIO Test no Jira. Isso permitiu um rastreamento eficaz dos resultados e o monitoramento em tempo real das falhas e execuções bem-sucedidas.

3.2 Fase 2: Automação com Cypress

Após a conclusão dos testes com Selenium WebDriver, foi iniciada a segunda fase do processo, utilizando o Cypress e Cucumber para trocar os testes automatizados e otimizar a execução de testes front-end.

Foi realizada uma nova análise dos cenários de teste para identificar quais fluxos de interface poderiam ser otimizados com Cypress. Esse framework foi escolhido para testar principalmente componentes de UI, validar elementos dinâmicos e realizar testes mais rápidos e de feedback instantâneo. O Cypress foi configurado no projeto com uma instalação simples utilizando JavaScript. Por ser uma ferramenta que trabalha diretamente com o DOM da página, não foi necessária a configuração de drivers adicionais, o que facilitou o setup.

Os testes foram estruturados usando o modelo Page Object Model (POM) para garantir que o código fosse modular e reutilizável. Cada página da aplicação foi representada por uma classe, encapsulando as interações com os elementos da interface.[4] Os testes automatizados foram desenvolvidos focando em fluxos de UI e componentes dinâmicos da aplicação. Esses testes foram projetados para serem executados rapidamente e oferecerem feedback imediato, complementando os testes mais complexos de Selenium. Um novo pipeline foi configurado no Jenkins para a execução dos testes de Cypress. Este pipeline rodava paralelamente ao de Selenium, garantindo que os dois conjuntos de testes fossem executados de forma contínua. Assim como os testes de Selenium, os resultados dos testes de Cypress foram enviados automaticamente para o AIO Test no Jira. Isso garantiu um rastreamento unificado de todos os testes automatizados, permitindo uma visão centralizada dos resultados e facilitando a identificação de falhas.

A metodologia aplicada foi adequada às intervenções realizadas, visando otimizar a execução dos testes de regressão e melhorar a cobertura dos testes funcionais. O proce-

[3]https://www.selenium.dev/documentation/grid/applicability/

[4]https://www.browserstack.com/guide/cypress-page-object-model

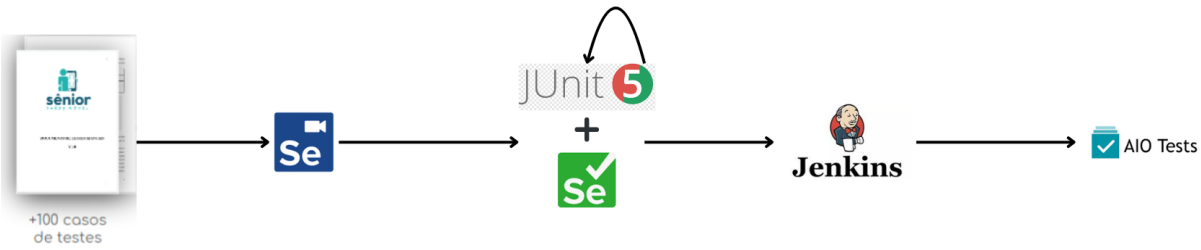
dimento adotado é apresentado em detalhes nos capítulos subsequentes, onde o capítulo 4 aborda o uso do Selenium WebDriver, e o capítulo 5 detalha a implementação dos testes utilizando Cypress. Cada uma dessas etapas foi ajustada às necessidades do projeto, permitindo maior eficiência, repetibilidade e rastreabilidade dos resultados.

4 AUTOMATIZAÇÃO DOS TESTES FUNCIONAIS COM O SELENIUM

O capítulo mostra a implementação de um fluxo automatizado de testes utilizando o Selenium, desde a captura inicial de interações no Selenium IDE, a estruturação dos testes no Selenium WebDriver e JUnit. Como a integração desses testes em um ambiente de integração contínua com Jenkins, garantiu a execução automatizada e o armazenamento de relatórios no AIO Test do Jira.

O fluxo de trabalho que combina o uso de Selenium IDE para gravação inicial de testes, Selenium WebDriver com JUnit para a implementação de testes automatizados, Jenkins para integração contínua e AIO Test no Jira para gerenciamento de testes foi montado para automatizar o processo.

Figura 1 – Passos para a implementação da automação dos testes funcionais utilizando Selenium IDE e AIO Tests.



Fonte: Elaborado pelo autor, 2024

A imagem 1 demonstra os passos feitos e necessários para a implementação da automação dos testes funcionais utilizando Selenium IDE e o uso do AIO tests para um fornecimento e armazenamento de reports.

4.1 Desafios

- Desafios no Uso do Selenium IDE
  - Captura de alertas temporários: O Selenium IDE não consegue gravar a presença de um alerta quando ele é disparado.
  - Exportação para JUnit: Apesar da facilidade de gravar e exportar os passos para código Java com JUnit, foi necessário revisar e limpar os testes para que fossem executáveis.
- Desafios no JUnit
  - Organização modular do código: Para melhorar a manutenção e a legibilidade, foi necessário estruturar os testes de forma mais organizada.

  - Conhecimento aprofundado do Selenium: O desenvolvimento de testes robustos exigiu um entendimento detalhado das bibliotecas e APIs do Selenium.

- Desafios na Manutenção dos Testes Automatizados
  - Resiliência a mudanças na interface: Pequenas alterações no layout ou estrutura HTML podem quebrar os testes, exigindo manutenção contínua.
  - Eficiência e desempenho: Com suítes de testes grandes, garantir que os testes rodem de forma rápida e eficiente tornou-se um desafio.

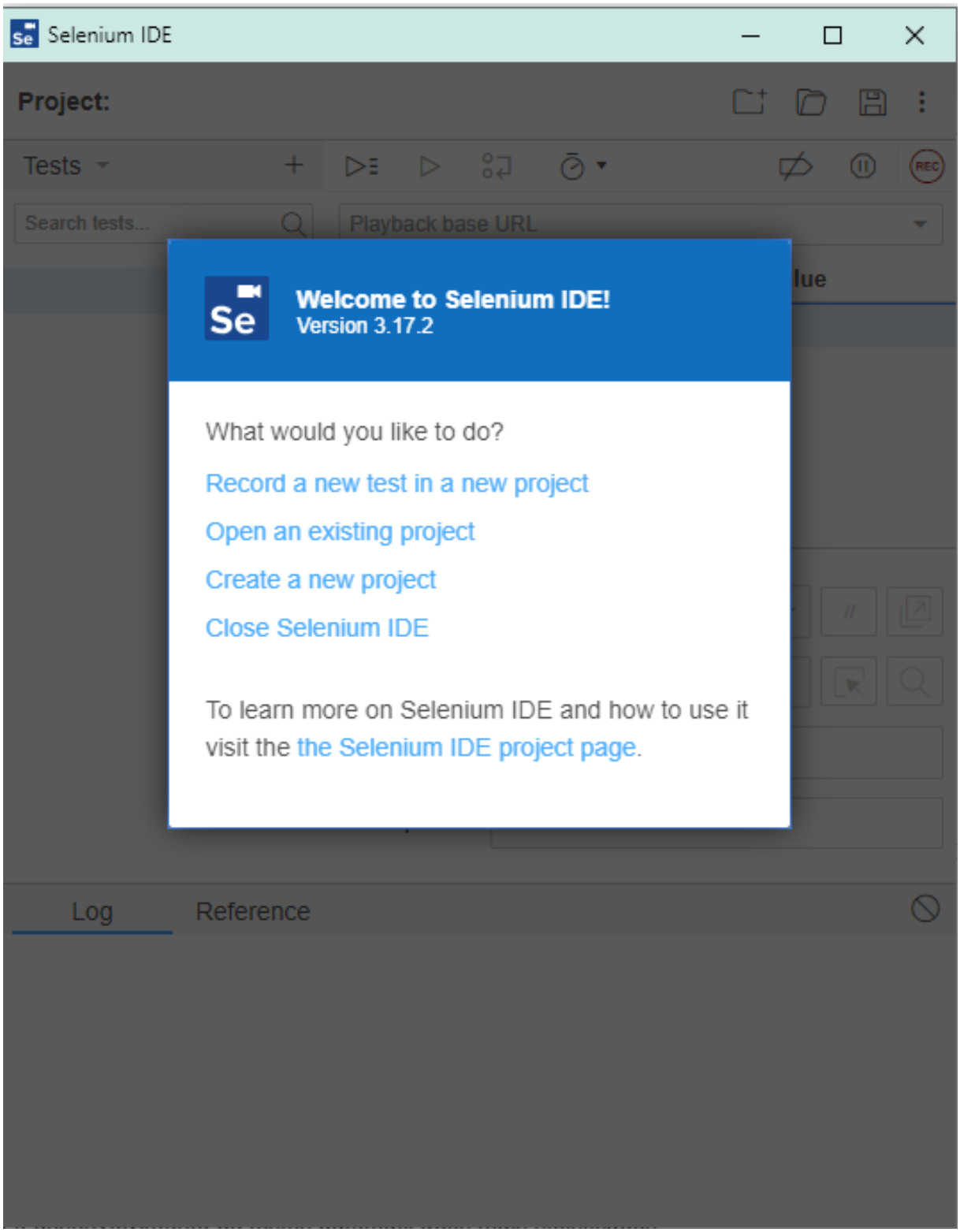
- Desafios no Uso do Selenium Grid e WebDriver
  - Criação de múltiplas contas: O sistema, por motivos de segurança, impedia logins repetitivos com a mesma conta, exigindo a criação de novos usuários para os testes.
  - Manutenção do estado de login: O Selenium Grid não conseguia manter os cookies de login entre testes, o que exigia novos logins para cada caso de teste.

4.2 Configuração Inicial

O Selenium IDE é uma ferramenta que permite a criação de testes com o formato de script sem a necessidade, inicial, de uso de código. Ele apresenta uma interface gráfica para gravar, reproduzir e salvar as ações na interface do usuário.

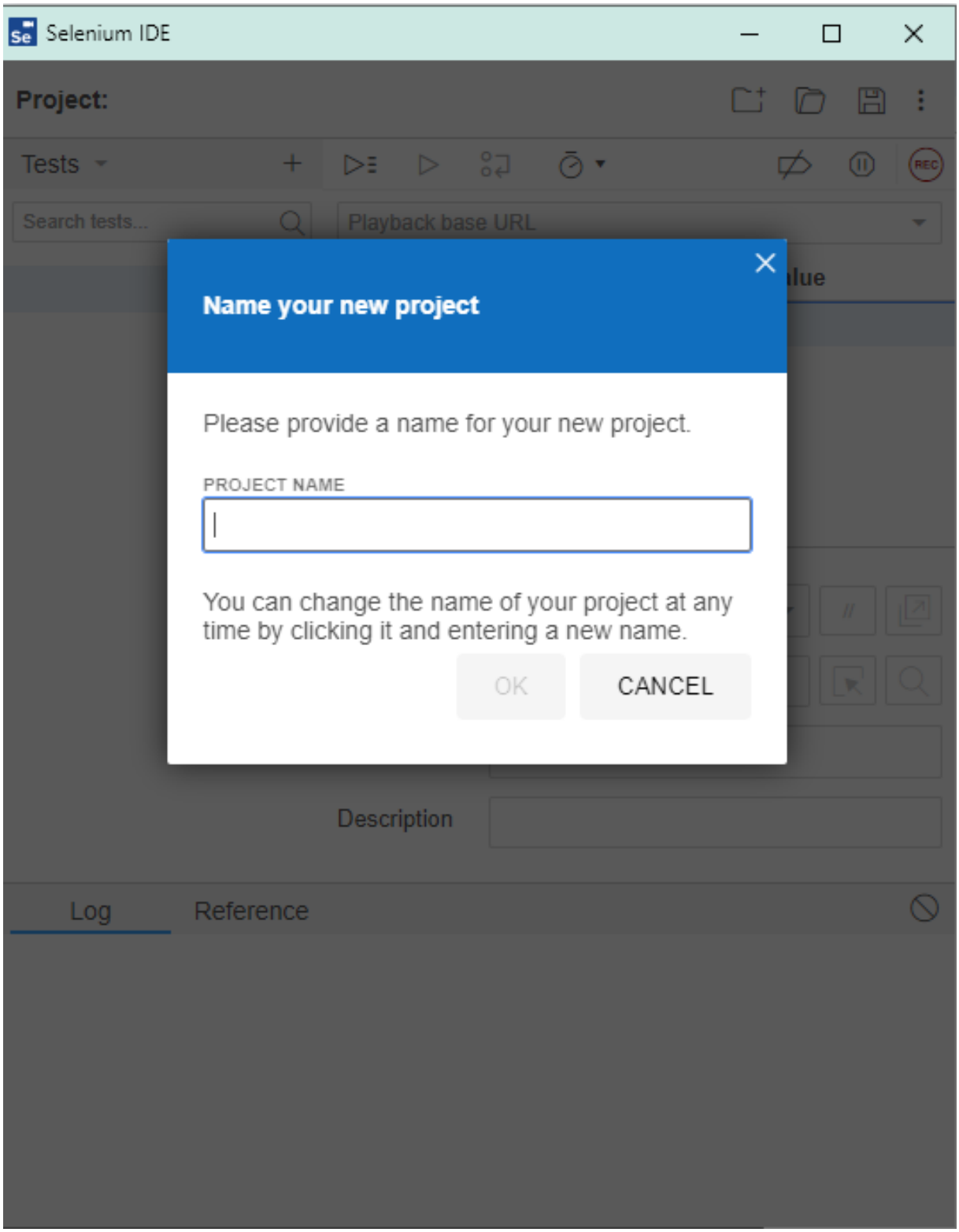
Para a utilização do Selenium IDE, foi necessária a instalação da extensão no navegador que será utilizado, nesse caso, Chrome. A extensão pode ser encontrada na loja de extensões do próprio navegador. Com a extensão instalada, o Selenium IDE fica integrado ao navegador e pode ser acessado para iniciar a gravação dos passos, as imagens 2 e 3 mostram o início da configuração inicial, com a abertura da extensão e a criação de um novo projeto. É necessário informar que a extensão do Selenium IDE não se encontra mais disponível na loja de extensão do Chrome, podendo, então, ser substituída por outra extensão como o Katalon Recorder ou usar outro navegador como Edge e Firefox.

Figura 2 – Menu da extensão do Selenium IDE.



Fonte: Elaborado pelo autor, 2024

Figura 3 – Etapa para a criação do projeto de teste.

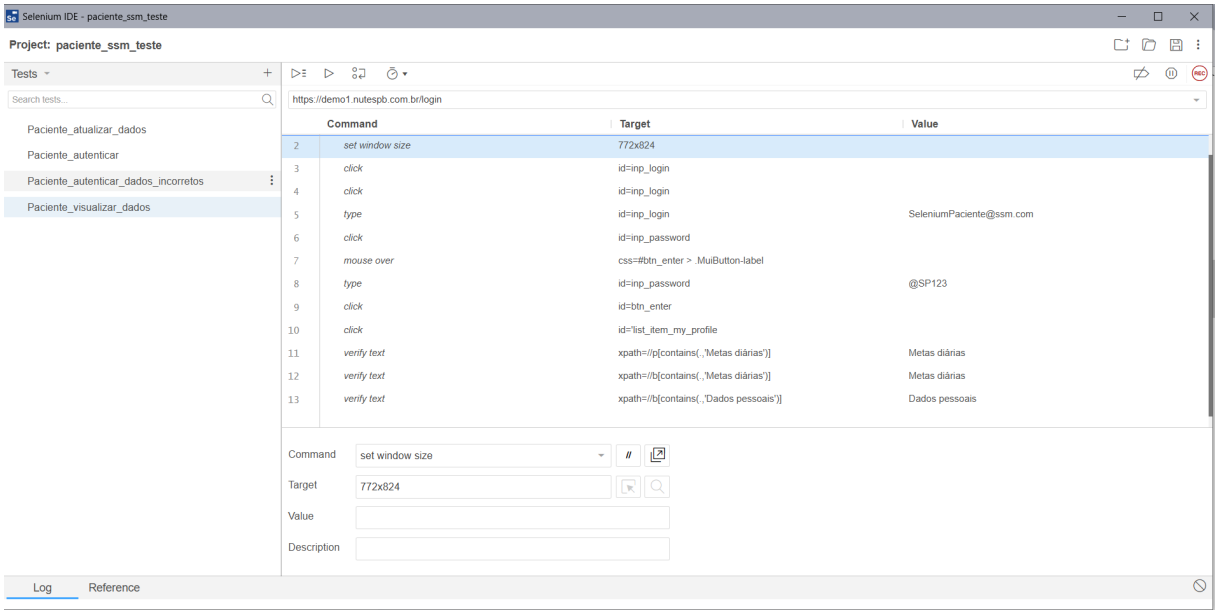


Fonte: Elaborado pelo autor, 2024

4.2.1 Criação de testes

A criação dos testes no Selenium IDE envolve a navegação através da aplicação web, enquanto o Selenium registra cada ação realizada, como cliques, inserção de dados em formulários, e navegação entre páginas. Essa funcionalidade é especialmente útil para capturar os principais fluxos de trabalho da aplicação, garantindo que os casos de uso críticos sejam testados. Ao final da gravação, o Selenium IDE permite a execução imediata do script para validar se as ações foram capturadas corretamente. A imagem 4 é um exemplo de um projeto com quatro scripts de teste na visão de um tipo de usuário.

Figura 4 – Interface do Selenium IDE exibindo um teste automatizado de autenticação.

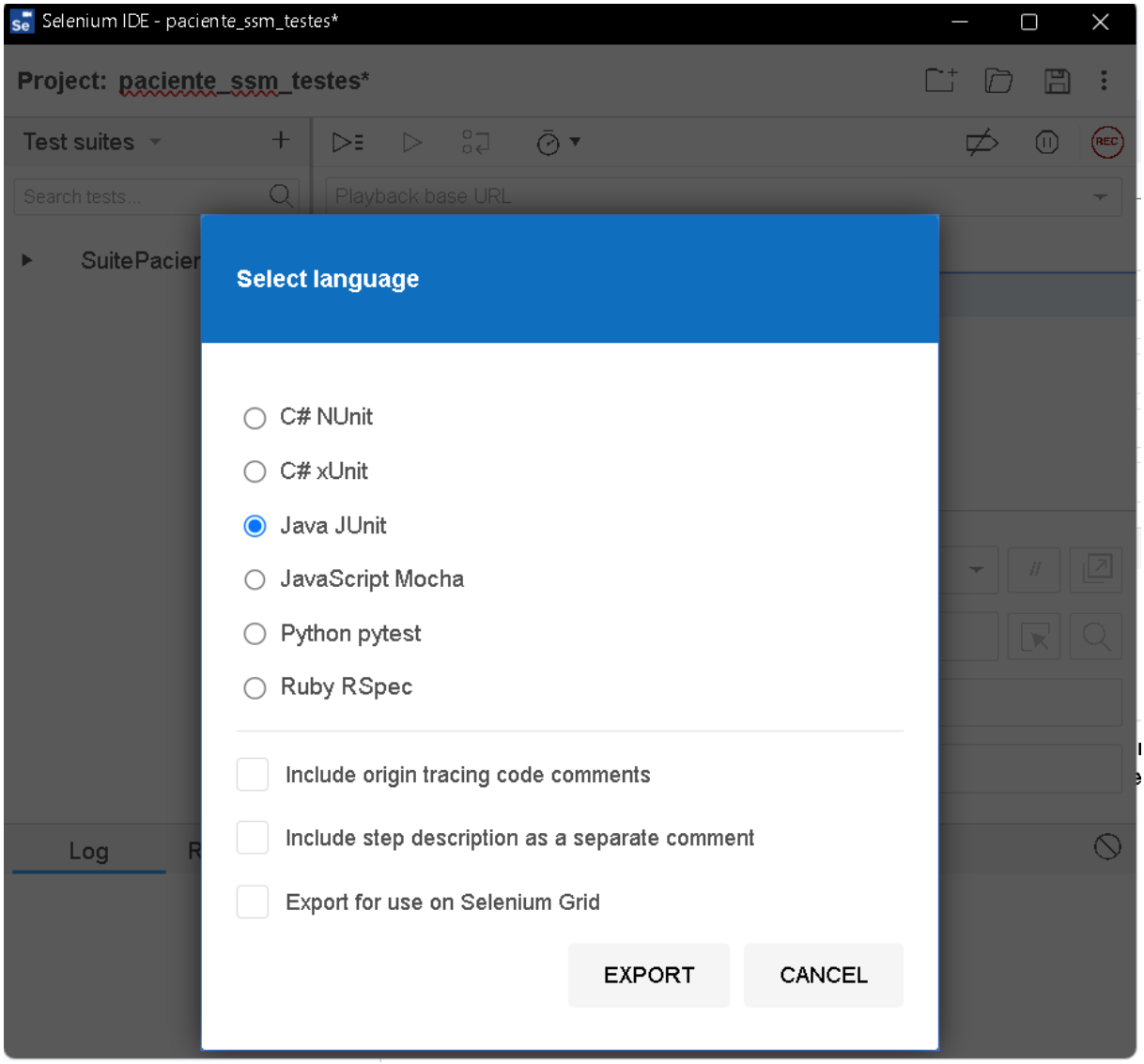


Fonte: Elaborado pelo autor, 2024

4.3 Desenvolvimento com Selenium WebDriver e JUnit

Embora o Selenium IDE seja eficaz para testes básicos, ele possui limitações quando se trata de testes mais complexos. Para superar essas limitações, os scripts gravados foram exportados para a linguagem de programação Java, para automação dos testes, com Selenium WebDriver e JUnit. Isso permitiu que os scripts gravados fossem utilizados como base para o desenvolvimento de testes automatizados mais sofisticados.

Figura 5 – Exportar scripts de testes no formato JUnit.



Fonte: Elaborado pelo autor, 2024

A próxima etapa foi a utilização do código JUnit(Java) para revisão e limpeza dos scripts exportados do selenium IDE. Configurado ao Selenium WebDriver foi possível executar uma classe e/ou todas as classes de testes presente no projeto localmente, ou seja, sem a necessidade de colocar em um repositório para saber o resultado dos testes.

4.3.1 Configuração do Projeto

A configuração dessa parte envolve a configuração inicial de um projeto de automação de testes com Selenium WebDriver e JUnit, envolve a criação de um ambiente de desenvolvimento integrado (IDE) como Eclipse ou IntelliJ IDEA. Foi necessário configurar um sistema de build no Maven para o gerenciamento das dependências do projeto, como Selenium WebDriver, Junit e Selenium Grid. O Selenium Grid permite a execução de

testes em diferentes navegadores e sistemas operacionais paralelamente, utilizando nós distribuídos. A configuração do Selenium Grid envolve a criação de um nó principal (hub) e vários nós (nodes) que serão utilizados para executar os testes. É necessário configurar um sistema de build como Maven ou Gradle para gerenciar as dependências do projeto, que incluirão o Selenium WebDriver, JUnit, e outras bibliotecas necessárias.

Figura 6 – Configuração do sistema Maven.

```
    <groupId>selenium</groupId>
    <artifactId>selenium-test</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.version>3.9.6</maven.version>
        <surefire.version>3.2.5</surefire.version>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <selenium.version>4.17.0</selenium.version>
        <junit.version>5.8.2</junit.version>
    </properties>

    <dependencies>
        <!-- Selenium WebDriver -->
        <dependency>
            <groupId>org.seleniumhq.selenium</groupId>
            <artifactId>selenium-java</artifactId>
            <version>4.17.0</version>
            <scope>test</scope>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-grid -->
        <dependency>
            <groupId>org.seleniumhq.selenium</groupId>
            <artifactId>selenium-grid</artifactId>
            <version>4.17.0</version>
            <scope>test</scope>
        </dependency>
```

Fonte: Elaborado pelo autor, 2024

4.3.2 Implementação dos Testes

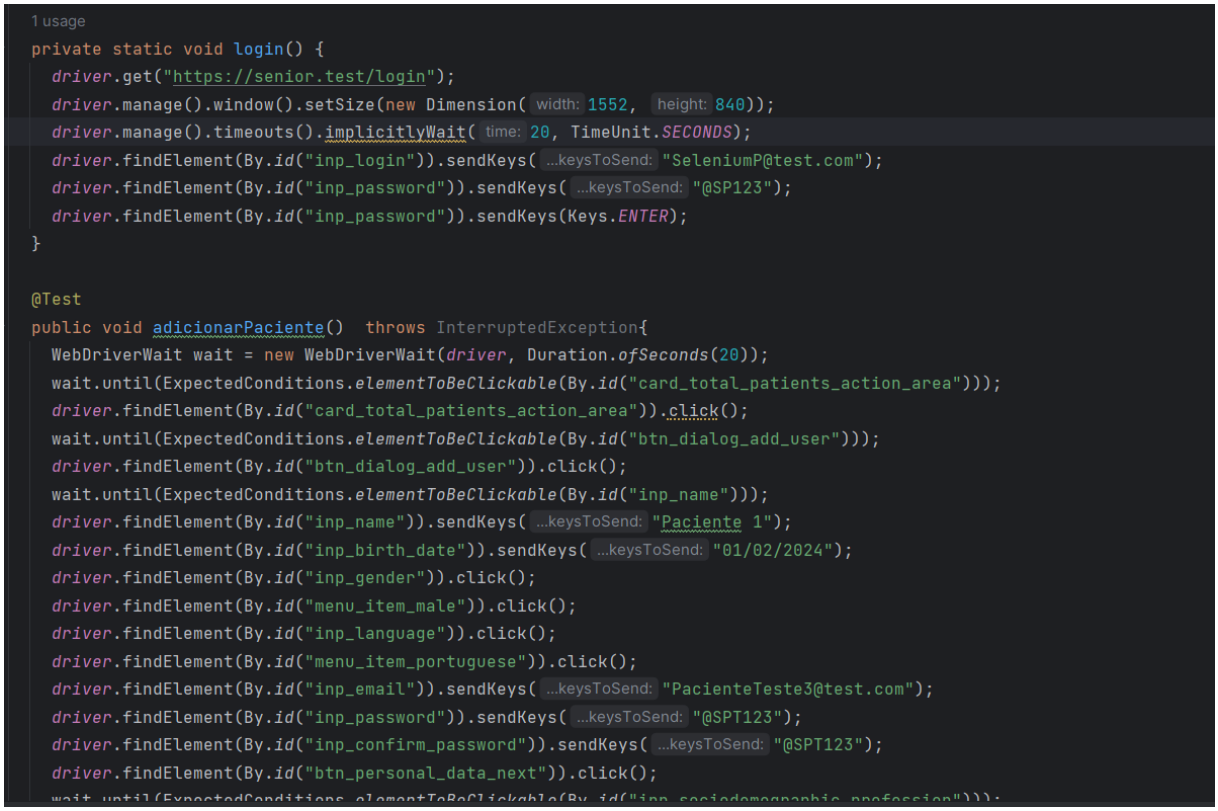
Cada caso de teste é escrito como um método em uma classe de teste. Quando exportado do Selenium IDE para o JUnit, o projeto criado torna-se a classe e cada script de teste um método. O JUnit fornece anotações como @Test, @Before, e @After, que ajudam a organizar e gerenciar a execução dos testes. A anotação @Test indica que um método é um caso de teste, enquanto @Before e @After são usados para configurar e limpar o ambiente de teste, respectivamente.

Figura 7 – Exemplo da estrutura de uma classe de Teste no JUnit.

```
20    private static JavascriptExecutor js;
21
22    @Before
23    public static void setUp() {
24      driver = new ChromeDriver();
25      ChromeOptions options = new ChromeOptions();
26      js = (JavascriptExecutor) driver;
27      vars = new HashMap<>();
28
29      options.addArguments("--start-maximized");
30      options.addArguments("--ignore-ssl-errors=yes");
31      options.addArguments("--ignore-certificate-errors");
32      options.addArguments("--disable-extensions");
33      options.addArguments("--disable-infobars");
34      options.addArguments("--no-sandbox");
35      options.addArguments("--disable-dev-shm-usage");
36
37      login();
38    }
39
40    @After
41    public static void tearDown() {
42      if (driver != null) {
43        driver.quit();
44      }
45    }
46
```

Fonte: Elaborado pelo autor, 2024

Figura 8 – Exemplo da estrutura dos scripts de testes no JUnit.



```
1 usage
private static void login() {
  driver.get("https://senior.test/login");
  driver.manage().window().setSize(new Dimension( width: 1552,  height: 840));
  driver.manage().timeouts().implicitlyWait( time: 20, TimeUnit.SECONDS);
  driver.findElement(By.id("inp_login")).sendKeys( ...keysToSend: "SeleniumP@test.com");
  driver.findElement(By.id("inp_password")).sendKeys( ...keysToSend: "@SP123");
  driver.findElement(By.id("inp_password")).sendKeys(Keys.ENTER);
}

@Test
public void adicionarPaciente()  throws InterruptedException{
  WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(20));
  wait.until(ExpectedConditions.elementToBeClickable(By.id("card_total_patients_action_area")));
  driver.findElement(By.id("card_total_patients_action_area")).click();
  wait.until(ExpectedConditions.elementToBeClickable(By.id("btn_dialog_add_user")));
  driver.findElement(By.id("btn_dialog_add_user")).click();
  wait.until(ExpectedConditions.elementToBeClickable(By.id("inp_name")));
  driver.findElement(By.id("inp_name")).sendKeys( ...keysToSend: "Paciente 1");
  driver.findElement(By.id("inp_birth_date")).sendKeys( ...keysToSend: "01/02/2024");
  driver.findElement(By.id("inp_gender")).click();
  driver.findElement(By.id("menu_item_male")).click();
  driver.findElement(By.id("inp_language")).click();
  driver.findElement(By.id("menu_item_portuguese")).click();
  driver.findElement(By.id("inp_email")).sendKeys( ...keysToSend: "PacienteTeste3@test.com");
  driver.findElement(By.id("inp_password")).sendKeys( ...keysToSend: "@SPT123");
  driver.findElement(By.id("inp_confirm_password")).sendKeys( ...keysToSend: "@SPT123");
  driver.findElement(By.id("btn_personal_data_next")).click();
```

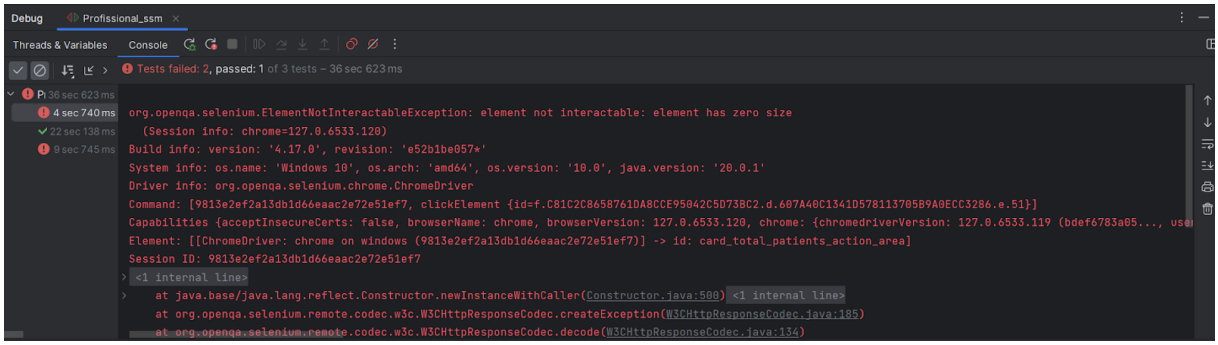
Fonte: Elaborado pelo autor, 2024

4.3.3 Execução e Validação

Para a limpeza e validação dos testes, foram realizadas execuções diretamente da IDE, sem a necessidade de utilizar a configuração do Selenium GRID e do Selenium Web Driver. Onde a configuração primeiro é feita para executar localmente utilizando o link da página de produção de teste sendo disponibilizada no próprio caso de teste, mostrado na imagem 7.

Os testes então quando executados, localmente, geram um report detalhado quando falham, mostrando onde teve o erro e o porquê de ter acontecido.

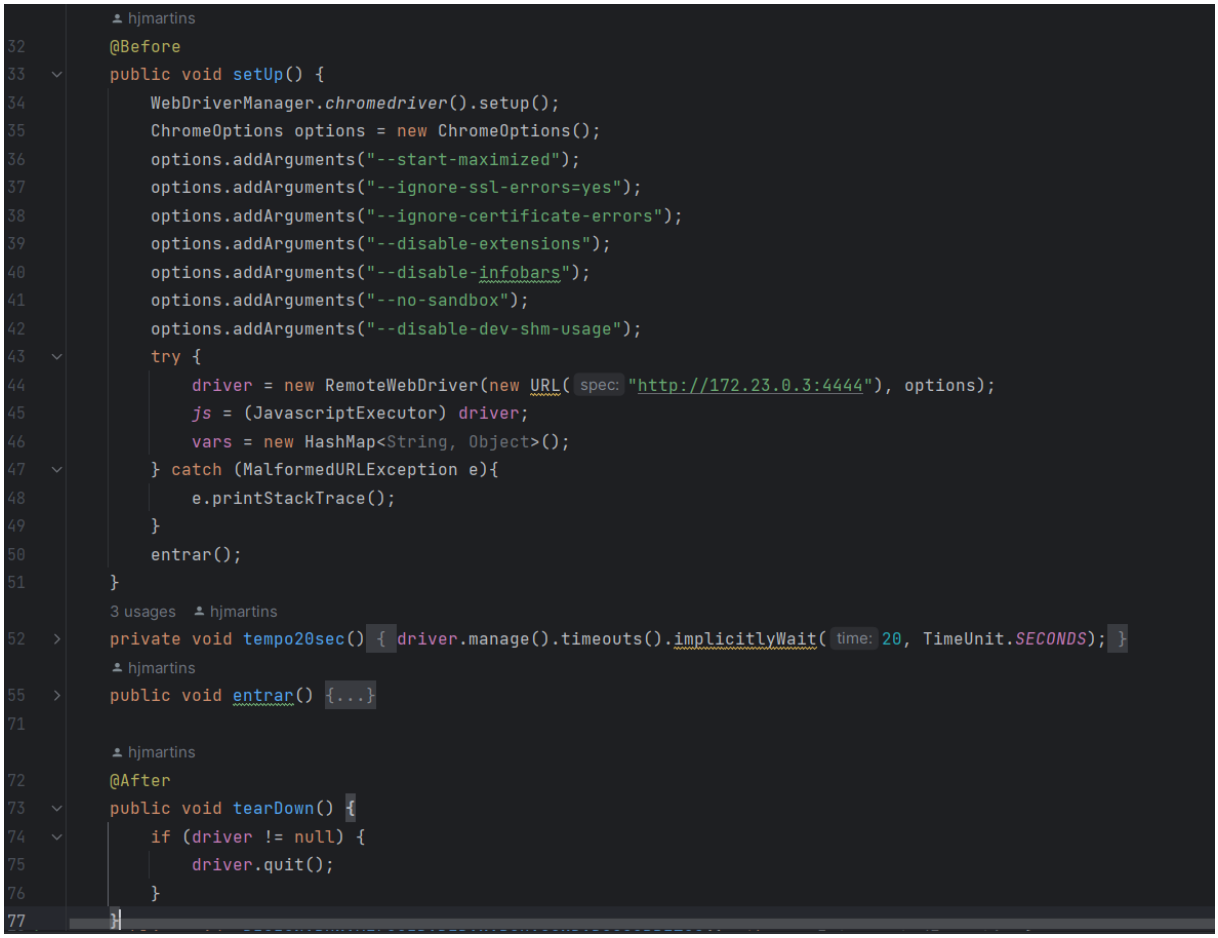
Figura 9 – Exemplo do report durante a execução local.



Fonte: Elaborado pelo autor, 2024

Quando revisados e validados os testes é realizado a configuração, em cada classe, do Selenium WebDriver e do Selenium Grid para utilizar a URL do HUb do Selenium Grid e depois é realizado o commit para suas respectivas branches para fazer a execução real, onde acontecerá o report e as builds automáticas no Jenkins.

Figura 10 – Configuração do JUnit para ser utilizado via Selenium Grid.



```
hjmartins
@Before
public void setUp() {
    WebDriverManager.chromedriver().setup();
    ChromeOptions options = new ChromeOptions();
    options.addArguments("--start-maximized");
    options.addArguments("--ignore-ssl-errors=yes");
    options.addArguments("--ignore-certificate-errors");
    options.addArguments("--disable-extensions");
    options.addArguments("--disable-infobars");
    options.addArguments("--no-sandbox");
    options.addArguments("--disable-dev-shm-usage");
    try {
        driver = new RemoteWebDriver(new URL( spec: "http://172.23.0.3:4444"), options);
        js = (JavascriptExecutor) driver;
        vars = new HashMap<String, Object>();
    } catch (MalformedURLException e){
        e.printStackTrace();
    }
    entrar();
}
3 usages  hjmartins
private void tempo20sec() { driver.manage().timeouts().implicitlyWait( time: 20, TimeUnit.SECONDS); }
hjmartins
public void entrar() {...}

hjmartins
@After
public void tearDown() {
    if (driver != null) {
        driver.quit();
    }
}
```

Fonte: Elaborado pelo autor, 2024

4.4 Integração Contínua com Jenkins

Jenkins é um servidor de automação open source autônomo que pode ser usado para automatizar todos os tipos de tarefas relacionadas à construção, teste e entrega ou implantação de software.(Documnetação do Jenkins 2024) A integração contínua com Jenkins permitiu que os testes automatizados sejam executados automaticamente para a geração de reports. Para configurar o Jenkins, foi necessário instalar a ferramenta em uma máquina virtual hospedada em um servidor. Para poder ter acesso às classes de testes foi usado o github como forma de intermediador. A configuração essencial para este processo inclui o plugin de integração com JUnit e Git.

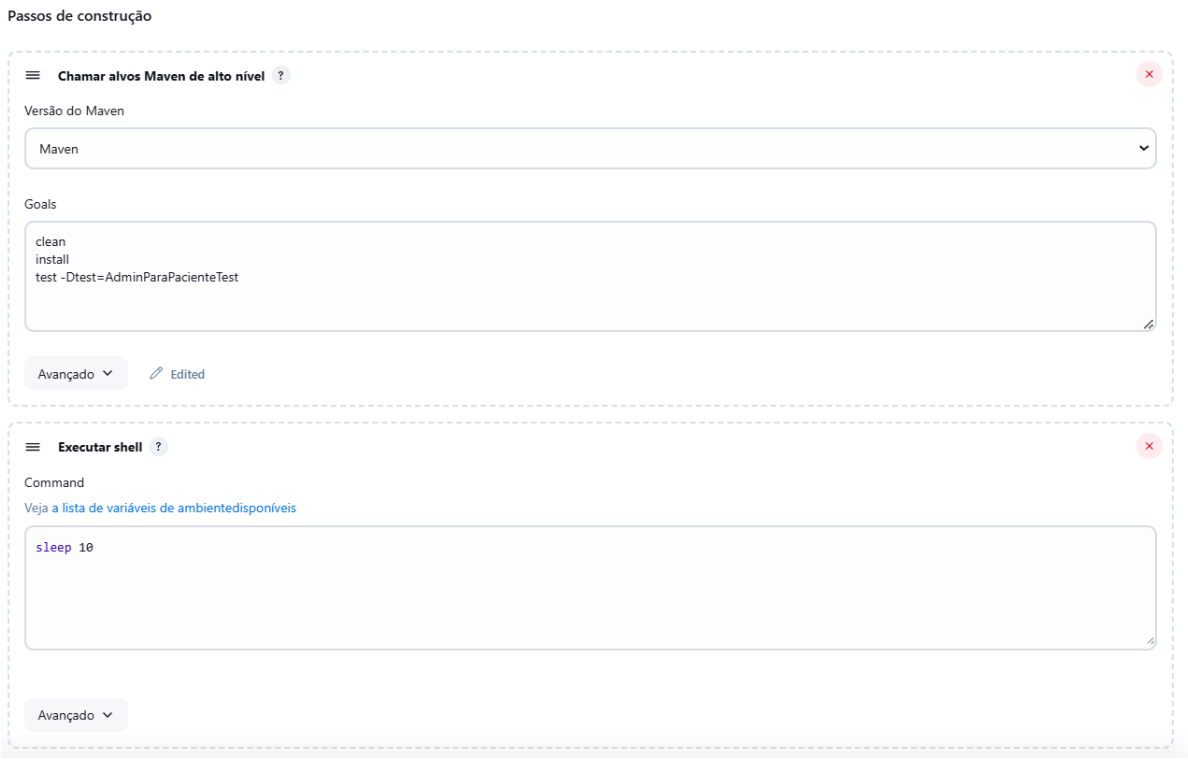
4.4.1 Criação de Pipelines

No Jenkins, um pipeline é um conjunto de etapas que definem o processo de build e teste. O pipeline para automação de testes inclui as seguintes etapas:

- Checkout do Código: O código-fonte do repositório Git é clonado.

- O projeto é compilado e as dependências são resolvidas.
- Execução dos Testes: Os testes automatizados são executados utilizando Selenium WebDriver, Selenium Grid e JUnit.
- Geração de Relatórios: Resultados dos testes são gerados e publicados.

Figura 11 – Exemplo da organização da pipeline no Jenkins

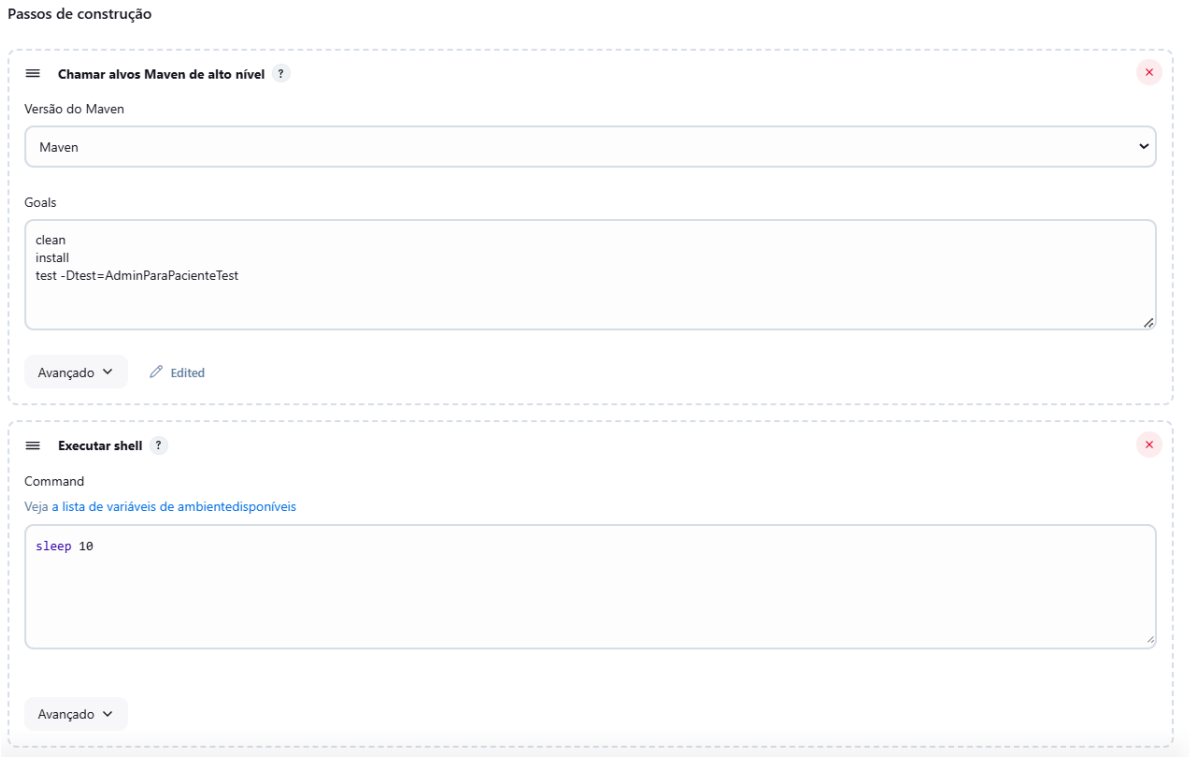


Fonte: Elaborado pelo autor, 2024

4.4.2 Configuração de Jobs

Além da pipeline, o Jenkins permite a configuração de jobs agendados para execução em um dia específico a cada sete dias. Isso garante que os testes sejam executados continuamente, ajudando a identificar problemas o mais cedo possível no ciclo de desenvolvimento.

Figura 12 – configuração do Job no Jenkins



Fonte: Elaborado pelo autor, 2024

4.5 Gerenciamento de Testes com AIO Test no Jira

O AIO Test é um plugin do Jira que oferece um conjunto completo de ferramentas para o gerenciamento de casos de teste, execução de testes e relatórios de resultados. A integração com o Jenkins permite que os resultados dos testes automatizados sejam automaticamente vinculados aos tickets do Jira, facilitando o rastreamento de defeitos e a gestão de qualidade.

No AIO Test, os casos de teste foram criados e organizados em folders conforme as respectivas branchs dos testes, para organizar os testes para uma melhor integração com o projeto JUnit e uma melhor organização visual.

A integração entre Jenkins e AIO Test é configurada por meio do plugins permitindo que os resultados dos testes sejam automaticamente atualizados no Jira. Para isso foi fornecido a utilização do token do projeto e a chave do projeto para serem inseridos em cada tarefa criada. Quando completa a configuração, os testes, quando executados automaticamente geram um report que será entregue ao AIO com a criação de um Cycle. O Cycle tem todos os testes presente na branch mostrando todos os resultados dos testes e o report de cada caso de teste. O Apêndice A mostra como o report se comporta no AIO Test.

Com o report completo, os testes são analisados. Para ser qualificado como aceito, o ciclo de testes, todos os testes devem estar marcados como ”passed”. Quando não ocorre

de todos os testes estarem como aceito, o report é analisado e verifica-se o(s) teste(s) que falharam. É feito então um levantamento se ocorreu alguma mudança, se foi falha no script de teste ou teve outra influência como atraso de resposta do WebDriver e falhas sem motivos aparente.

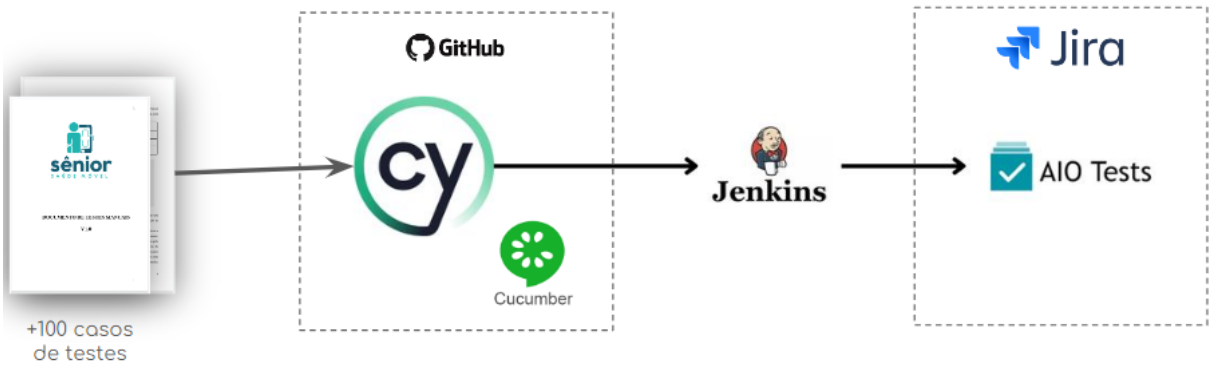
5 AUTOMATIZAÇÃO DOS TESTES FUNCIONAIS CO O CYPRESS

A automação dos testes no Cypress tem a mesma estrutura do Selenium, mas a parte de gravação dos passos imitando o usuário é retirada, os scripts de testes são feitos diretamente no Cypress juntamente com o Cucumber. Neste capítulo, exploramos todo o processo, desde a configuração até a execução e manutenção dos testes e como o Cypress e o Cucumber se comporta na integração com o Jenkins e AIO tests.

O Cypress foi escolhido como a ferramenta de automação de testes para esta segunda parte do projeto devido à sua capacidade de oferecer uma experiência de desenvolvimento ágil e eficiente, especialmente quando combinado com o Cucumber. Com a finalidade de conseguir ultrapassar as barreiras encontradas durante a primeira fase, com o Selenium.

A integração do Cypress com o Cucumber possibilita a geração de relatórios completos e personalizados, que incluem os passos de cada cenário, os resultados dos testes (passou/falhou) e capturas de tela das falhas. Essa visibilidade detalhada facilita a identificação de problemas e a análise da causa raiz. A manutenção dos testes mais simples e eficiente. Ao utilizar a linguagem Gherkin, as alterações nos requisitos podem ser refletidas diretamente nos cenários de teste, reduzindo o esforço necessário para revisar e manter os testes atualizados. A figura mostra qual foi o caminho dos passos para a automação dos testes e a divulgação do relatório final.

Figura 1 – caminho para a realização do teste



Fonte: Elaborado pelo autor, 2024

5.1 Desafios

Para superar os desafios encontrados com o uso do Selenium, o Cypress se mostrou a escolha mais adequada. As dificuldades incluíam a incapacidade de detectar mensagens de alerta, a configuração complexa, a lentidão na execução dos testes — que frequentemente ultrapassavam 40 minutos para serem concluídos —, a instabilidade nos testes, com falhas

intermitentes entre ciclos de execução, e a entrega de relatórios pouco detalhados e mal organizados.

Embora seja uma ferramenta relativamente recente, o Cypress conta com diversos pacotes disponíveis que facilitam e aprimoram a criação de testes. Além disso, é constantemente atualizado para acompanhar as inovações no desenvolvimento de aplicações modernas, garantindo maior eficiência e compatibilidade. Entre suas principais vantagens, destaca-se a configuração simples, que não exige a instalação de drivers adicionais, sendo fácil de configurar e começar a usar; a execução rápida e estável, com testes mais rápidos e menos falhas intermitentes devido à execução direta no navegador; a depuração facilitada, por meio de uma interface interativa que permite visualizar e depurar os testes em tempo real, inspecionando o DOM durante a execução; o acesso simplificado ao DOM e ao ambiente do navegador, que facilita o controle direto dos elementos e eventos do navegador, possibilitando a escrita de testes mais complexos; o suporte a aplicações modernas, sendo especialmente eficiente para frameworks como React, Angular e Vue; a escrita de testes intuitiva, graças à sua API com comandos encadeados e tratamento automático de sincronização e esperas, reduzindo a necessidade de configurações manuais; e a integração com ferramentas modernas, com suporte nativo para relatórios, plugins como Cucumber e fácil integração com pipelines de CI/CD, como o Jenkins.

5.2 Processos de Automação

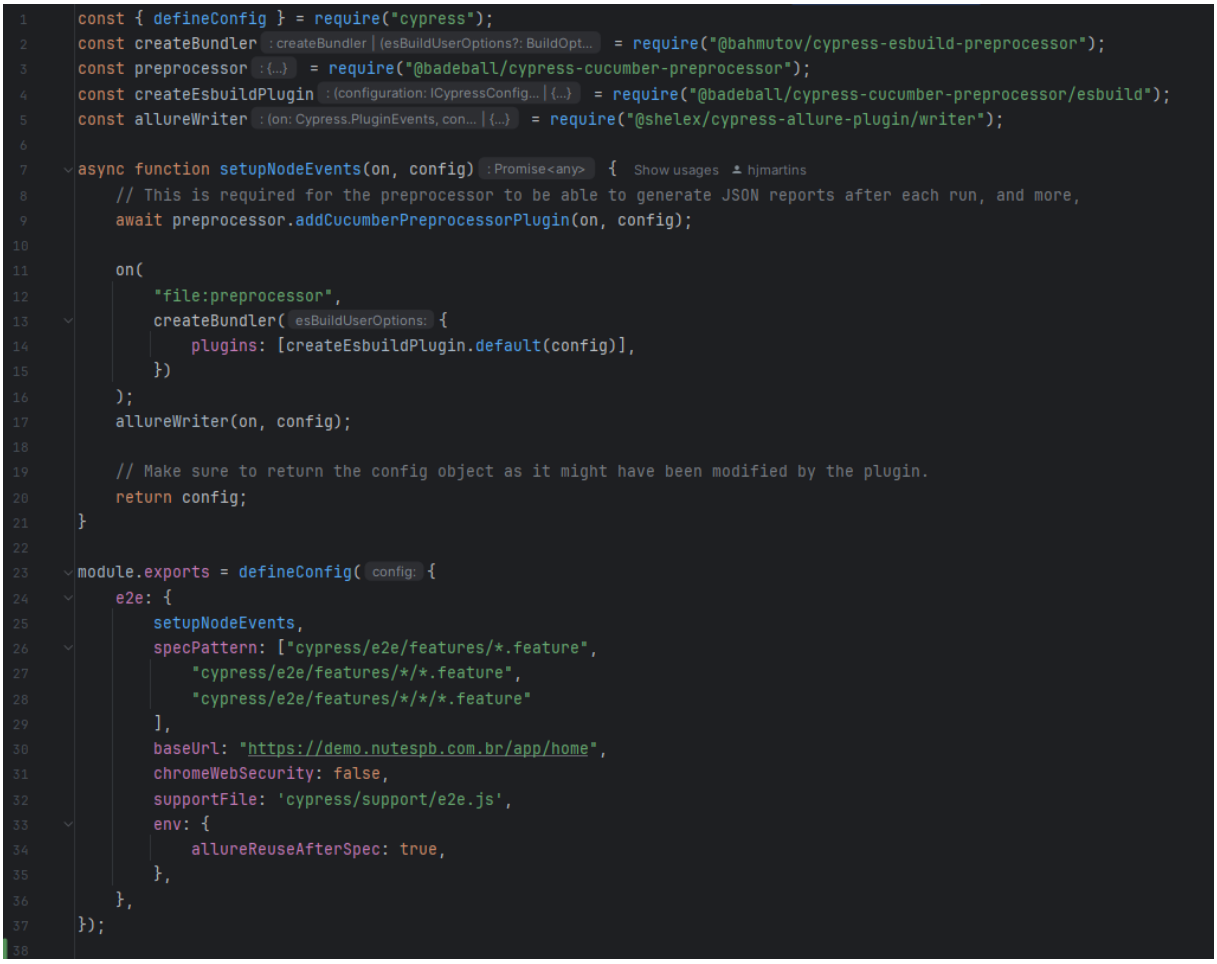
5.2.1 Instalação e Configuração do Cypress com Cucumber

Para iniciar o processo de automação de testes, o primeiro passo foi configurar o ambiente de testes, garantindo que tanto o Cypress quanto o Cucumber pudessem ser utilizados integradamente. O uso do cucumber permitiu que os testes fossem escritos em formato .feature, facilitando a compreensão e produzindo uma documentação dentro do teste.

Utilizando o npm, o Cypress e o Cucumber a instalação das dependencias forma instaladas no projeto: *npm install cypress @badeball/cypress-cucumber-preprocessor –save-dev*

Para habilitar o suporte ao Cucumber, o arquivo de configuração foi ajustado para utilizar o preprocessor adequado. Essa alteração foi essencial para que a estrutura dos testes fosse direcionada aos arquivos .feature, permitindo que a execução seguisse a ordem definida nesses arquivos. Além disso, essa configuração favorece a reutilização dos testes Cypress em diferentes cenários, promovendo maior eficiência e organização no processo de automação.

Figura 2 – arquivo de configuração cypress.config.js



```
const { defineConfig } = require("cypress");
const createBundler = require("@bahmutov/cypress-esbuild-preprocessor");
const preprocessor = require("@badeball/cypress-cucumber-preprocessor");
const createEsbuildPlugin = require("@badeball/cypress-cucumber-preprocessor/esbuild");
const allureWriter = require("@shelex/cypress-allure-plugin/writer");

async function setupNodeEvents(on, config) {
    // This is required for the preprocessor to be able to generate JSON reports after each run, and more,
    await preprocessor.addCucumberPreprocessorPlugin(on, config);

    on(
        "file:preprocessor",
        createBundler({
            plugins: [createEsbuildPlugin.default(config)],
        })
    );
    allureWriter(on, config);

    // Make sure to return the config object as it might have been modified by the plugin.
    return config;
}

module.exports = defineConfig({
    e2e: {
        setupNodeEvents,
        specPattern: ["cypress/e2e/features/*.feature",
            "cypress/e2e/features/*/*.feature",
            "cypress/e2e/features/*/*/*.feature"
        ],
        baseUrl: "https://demo.nutespb.com.br/app/home",
        chromeWebSecurity: false,
        supportFile: 'cypress/support/e2e.js',
        env: {
            allureReuseAfterSpec: true,
        },
    },
});
```

Fonte: Elaborado pelo autor, 2024

5.2.2 Estruturação dos Testes

Os cenários de teste foram escritos em Gherkin, utilizando uma estrutura clara. Essa abordagem facilitou a escrita dos testes e proporcionou uma maneira mais natural de documentar as funcionalidades testadas. Gherkin é um conjunto de regras gramaticais que torna o texto simples estruturado o suficiente para que o Cucumber possa entendê-lo.(Cucumber Documetation 2024)

Os Testes em Gherkin descrevem o fluxo de teste de forma legível e padronizada, com as palavras-chave given, then e When permitindo que todos na equipe compreendam o comportamento esperado. Junto a isso a documentação dos testes, feitas previamente, tem a mesma estrutura. A imagem 3 mostra um exemplo de como está estruturado a documentação.

Figura 3 – caso de teste encontrado na documentação

TC DSHB-003 AUTENTICAR COM DADOS INVÁLIDOS
Dado que o usuário cadastrado na plataforma esteja na tela de login do SSM.
Quando inserir e-mail e/ou senha inválidos e clicar no botão "ENTRAR".
Então a plataforma exibirá a seguinte mensagem: "Email com formato inválido".

Fonte: Elaborado pelo autor, 2024

Figura 4 – .feature exemplo

```
Feature: Diagnosis Page

  Background:
    Given that the user is on the dashboard screen. - 2PH

  Scenario: ADD A NEW DIAGNOSIS
    When Click on Diagnosis button. - 1PH
    Then the diagnosis page will appear
    When click on Add button
    Then a dialog window will appear
    When the user fill the diagnosis information and click on save button
    Then a confirmation alert message will appear

  Scenario: SEE A DIAGNOSIS
    When Click on Diagnosis button. - 2PH
    Then the diagnosis page will appear. - 2PH
    When click on diagnosis card
    Then the diagnosis information will appear

  Scenario: DELETE A DIAGNOSIS
    When Click on Diagnosis button. - 3PH
    Then the diagnosis page will appear. - 3PH
    When click on delete button
    Then a dialog window will appear. - 3PH
    When click on YES button
    Then a confirmation alert for the exclusion will appear
```

Fonte: Elaborado pelo autor, 2024

A figura 5 mostra como um exemplo da estrutura de um arquivo .feature.

Feature: Descreve a funcionalidade ou área do sistema que está sendo testada (neste exemplo, a página de diagnóstico).

Scenario: Descreve um caso de teste específico (neste caso, o usuário está em uma página anterior a de diagnóstico).

Given: Descreve o contexto inicial ou a pré-condição do cenário.

When: Descreve a ação que o usuário executa.

Then: Descreve o resultado esperado após a ação.

5.2.3 Desenvolvimento dos Step Definitions

Para cada passo descrito no arquivo .feature, é necessário implementar uma definição do passo com mesmo nome, chamada de step definition. Esses steps são implementados em arquivos JavaScript na pasta step_definitions.

Os step definitions foram implementados para que o Cypress pudesse executar as ações descritas no arquivo .feature. Cada passo foi mapeado para uma função que simula as ações do usuário. Os arquivos step definitions são arquivos .js e é lá que acontece toda ação. No step definitions é necessário que a frase esteja igual a do arquivo .feature e não pode ocorrer repetição da mesma frase, dentro dos arquivos, pois ocorrerá choque entre qual função será executada.

Figura 5 – exemplo do step definition

Fonte: Elaborado pelo autor, 2024

Com os passos dos arquivos .feature e do step definitions alinhados, os testes foram

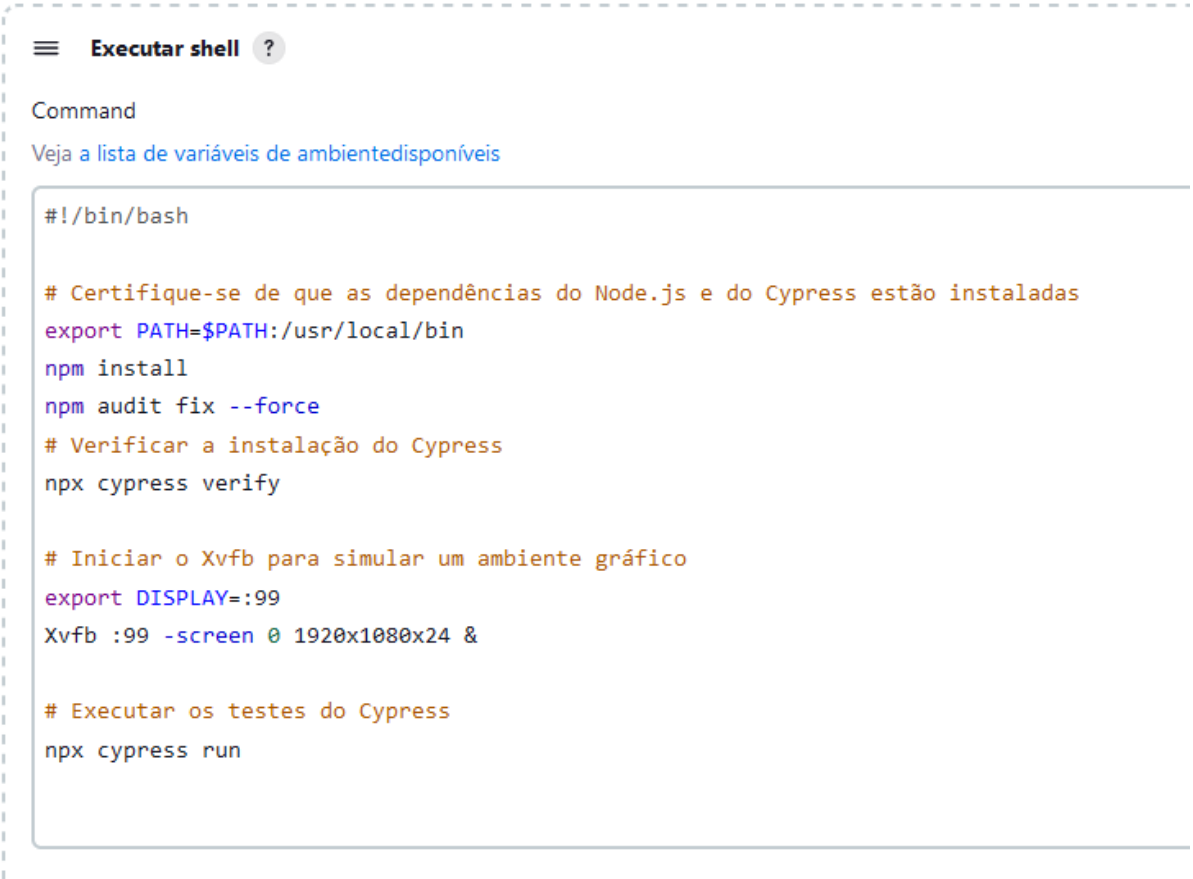
executados localmente para manutenção antes que suba para a branch designada. Quando executados localmente e validados, ocorre, então, o commit.

5.2.4 Execução e Integração com Jenkins

Quando os testes já estão alocados no repositório, ocorre a integração com o Jenkins. Essa integração foi um passo crucial para garantir que os testes fossem executados automaticamente a cada início de semana.

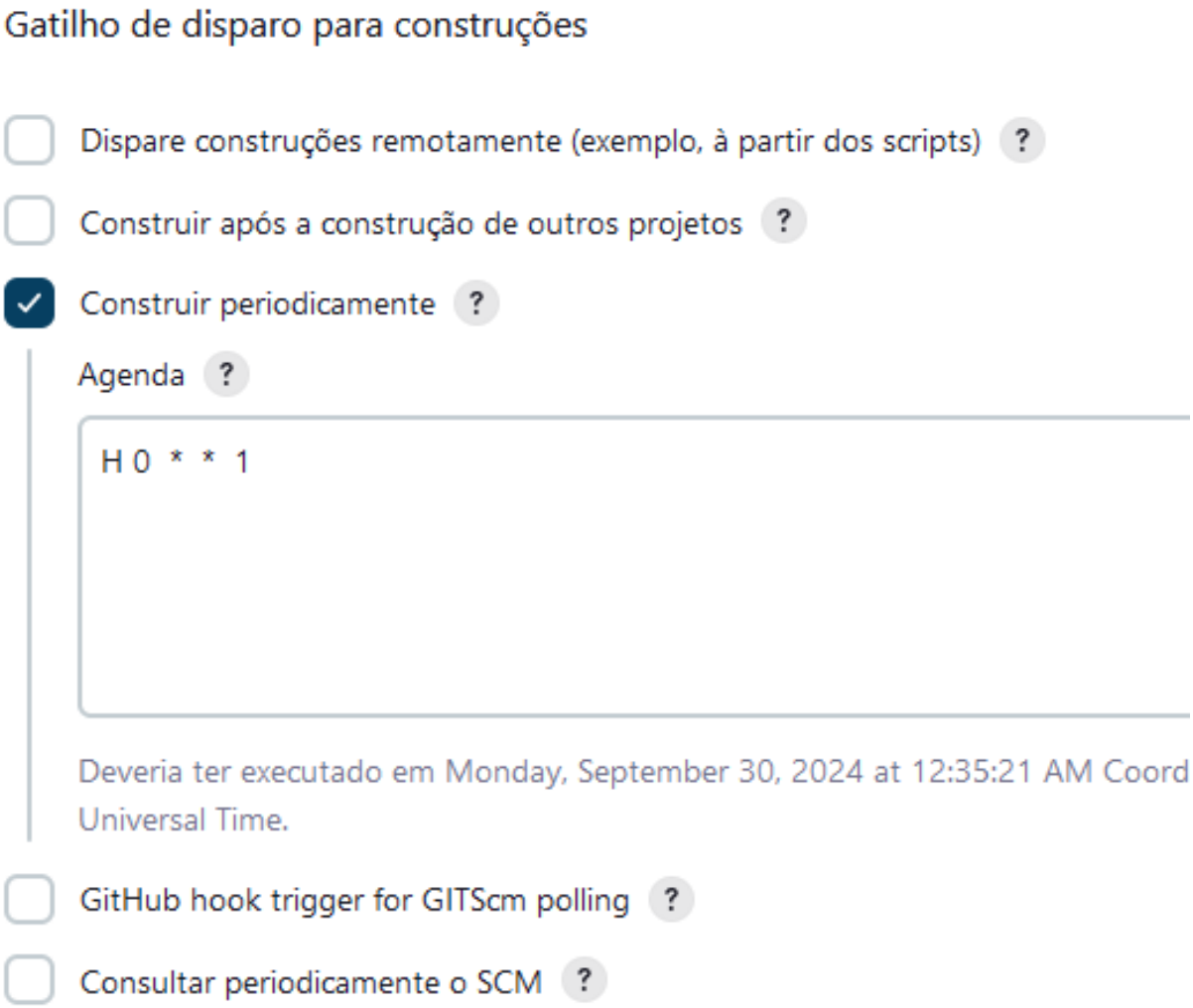
A execução dos testes no Jenkins foi feita em modo headless, garantindo que os testes rodassem sem a necessidade de uma interface gráfica. A configuração do pipeline 6 no Jenkins foi feita para monitorar o repositório GitHub e rodar os testes automaticamente com a configuração de gatilho.7.

Figura 6 – Pipeline de execução do Jenkins para o Cypress



Fonte: Elaborado pelo autor, 2024

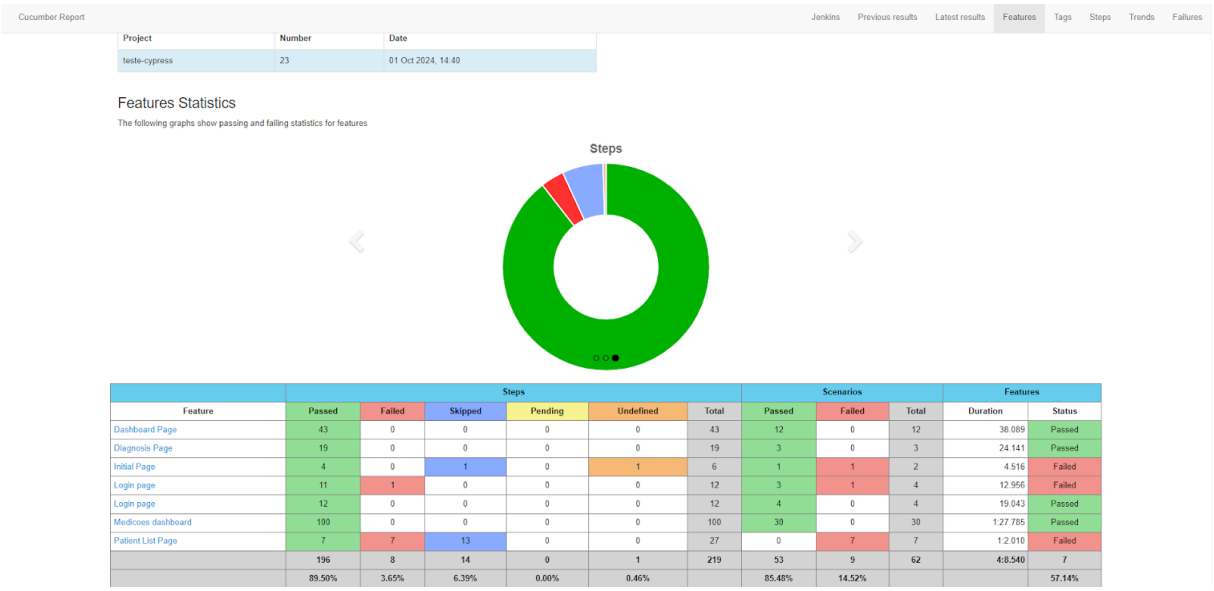
Figura 7 – Gatilho de disparo para rodar automaticamente



Fonte: Elaborado pelo autor, 2024

A integração Jenkins e Cucumber permite a geração automática de relatórios HTML a cada build executado. Isso é possível por meio do plugin Cucumber Reports, disponível no Jenkins, que processa os resultados dos testes escritos em Gherkin e gera relatórios detalhados e visuais. Esses relatórios são atualizados a cada execução do pipeline, oferecendo uma visão clara do status dos testes, com detalhes sobre quais cenários passaram ou falharam, facilitando a análise de resultados e o rastreamento de falhas no sistema.

Figura 8 – relatório criado pelo plugin do cucumber no Jenkins



	Steps						Scenarios			Features	
Feature	Passed	Failed	Skipped	Pending	Undefined	Total	Passed	Failed	Total	Duration	Status
Dashboard Page	43	0	0	0	0	43	12	0	12	38.089	Passed
Diagnosis Page	19	0	0	0	0	19	3	0	3	24.141	Passed
Initial Page	4	0	1	0	1	6	1	1	2	4.516	Failed
Login page	11	1	0	0	0	12	3	1	4	12.956	Failed
Login page	12	0	0	0	0	12	4	0	4	19.043	Passed
Medicoes dashboard	100	0	0	0	0	100	30	0	30	1:27.785	Passed
Patient List Page	7	7	13	0	0	27	0	7	7	1:2.010	Failed
	196	8	14	0	1	219	53	9	62	4:8.540	7
	89.50%	3.65%	6.39%	0.00%	0.46%		85.48%	14.52%			57.14%

Fonte: Elaborado pelo autor, 2024

Posteriormente, quando executado no Jenkins, os testes são enviados para o AIO com a integração do Plugin presente no Jenkins e AIO Tests.

5.2.5 Integração com AIO Test no Jira

A integração com o AIO Test no Jira foi um dos pontos mais importantes do processo, por permitir rastrear e gerenciar todos os resultados dos testes de forma centralizada. Após cada execução dos testes no Jenkins, os resultados eram automaticamente enviados para o AIO Test, facilitando o gerenciamento e a análise. Os resultados dos testes Cypress, juntamente com os cenários do Cucumber, eram apresentados detalhadamente, com informações sobre cada passo do teste, quais falharam, e quais passaram com sucesso. O Apêndice B mostra como os reports eram exibidos no AIO Test.

Essa integração garantiu uma documentação clara e rastreável de cada execução, permitindo que os participantes do projeto acompanhassem o progresso dos testes e entendessem o comportamento, além disso, permite que pessoas de fora do projeto consigam entender sem muita complicação.

6 DISCUSSÃO

A implementação da automação de testes funcionais utilizando as ferramentas Selenium-JUnit-Jenkins-AIO e Cypress-Cucumber-Jenkins-AIO permitiu uma análise aprofundada dos impactos e desafios associados à adoção dessas tecnologias no contexto de testes de software. Os resultados obtidos demonstraram uma evolução significativa na qualidade e eficiência dos testes, bem como a consolidação de um ambiente estruturado para o monitoramento e a rastreabilidade dos resultados.

A integração dessas ferramentas viabilizou um fluxo de testes automatizados alinhado às boas práticas de desenvolvimento ágil, favorecendo a detecção precoce de falhas e garantindo maior estabilidade ao sistema. O uso do AIO Test no Jira aprimorou o gerenciamento dos resultados dos testes, proporcionando um controle mais eficiente e permitindo a análise detalhada das falhas identificadas.

O Selenium, em conjunto com o JUnit, demonstrou-se eficaz na execução de testes automatizados, assegurando a preservação da funcionalidade do sistema mesmo após sucessivas atualizações. No entanto, a complexidade inerente à configuração do Selenium Grid, somada ao tempo elevado de execução dos testes, impactou a agilidade do processo. Ademais, a impossibilidade de reutilização de sessões de login dentro de um mesmo ciclo de testes comprometeu a eficiência do processo, resultando na necessidade de múltiplos logins e consequente aumento da carga computacional.

A transição para o Cypress evidenciou ganhos substanciais em termos de desempenho e usabilidade. A execução direta no navegador eliminou a latência observada no Selenium, permitindo testes mais rápidos e precisos. A adoção do Cucumber, com a sintaxe Gherkin, proporcionou uma padronização na escrita dos casos de teste, facilitando a compreensão por parte de todos os envolvidos no processo de desenvolvimento, incluindo aqueles sem expertise técnica.

O Cypress conseguiu resolver eficazmente o problema da captura e validação de alertas, algo que o Selenium não conseguia fazer. Isso tornou a automação de testes muito mais completa e precisa. Além disso, os testes ficaram muito mais rápidos, com o Cypress reduzindo de 15 a 20 minutos o tempo de execução, em comparação com o Selenium. Essa melhoria foi especialmente visível em suítes de testes mais complexas. Outro ponto positivo foi a eliminação da instabilidade do Selenium, que costumava apresentar falhas nos ciclos de execução, algo que foi completamente resolvido com o uso do Cypress.

A geração de relatórios estruturados e detalhados no Jenkins, aliada à integração com o AIO Test, otimizou o rastreamento dos testes e a documentação dos resultados, permitindo uma visibilidade aprimorada sobre a qualidade do software. Entretanto, o Cypress apresentou desafios no que tange à curva de aprendizado para a implementação inicial, especialmente na integração com ferramentas de gerenciamento de testes e na

adaptação de scripts para cenários específicos.

De forma geral, os resultados obtidos evidenciaram que a automação de testes funcionais representa um diferencial significativo na melhoria da qualidade do software, reduzindo o tempo de identificação de falhas e aumentando a confiabilidade das entregas. A escolha entre Selenium e Cypress deve considerar as especificidades do projeto e os requisitos do ambiente de desenvolvimento, uma vez que cada ferramenta apresenta vantagens e limitações distintas. O estudo conduzido demonstrou que, enquanto o Selenium se destaca pela robustez e compatibilidade com diferentes navegadores, o Cypress se sobressai pela eficiência e facilidade de uso, tornando-se uma alternativa viável para a automação de testes em aplicações web modernas.

Assim, os aprendizados obtidos ao longo deste processo oferecem diretrizes valiosas para equipes de desenvolvimento que buscam otimizar suas estratégias de automação de testes, promovendo maior confiabilidade e eficiência no ciclo de vida do software.

7 CONCLUSÃO

O objetivo proposto neste projeto foi plenamente atingido, evidenciando os benefícios e desafios associados à implementação da automação de testes funcionais. A adoção de Selenium e Cypress permitiu não apenas a validação sistemática das funcionalidades do software, mas também uma melhoria significativa na rastreabilidade dos testes, proporcionando maior controle sobre o produto.

Entre os principais benefícios observados, destaca-se a redução do tempo de execução dos testes, a minimização de erros humanos e a geração de relatórios detalhados para análise e acompanhamento das execuções. Além disso, a padronização dos casos de teste com o uso do Cucumber e a integração com ferramentas como Jenkins e AIO Test no Jira favoreceram a colaboração entre as equipes e a eficiência na detecção de falhas.

No entanto, alguns desafios foram enfrentados ao longo do projeto. A complexidade inicial na configuração do Selenium Grid e a necessidade de múltiplos logins impactaram a execução dos testes, enquanto o Cypress apresentou uma curva de aprendizado considerável, especialmente na adaptação de scripts para diferentes cenários e na integração com outras ferramentas.

Para trabalhos futuros, o ambiente do Jenkins, configurado para o AIO, pode ser facilmente integrado com outros tipos de testes para automação, como testes de performance, carga e até mesmo testes de segurança. Essa flexibilidade permite que diferentes tipos de testes sejam executados de forma contínua e eficiente, garantindo uma cobertura completa e assegurando uma maior cobertura do software. Além disso, o Cypress oferece uma plataforma robusta para a automação de testes, não se limitando somente a testes funcionais. Todo o ambiente montado para o Cypress pode ser aproveitado também para testes de API e de usabilidade, proporcionando uma solução abrangente que cobre várias frentes de teste de maneira ágil e eficaz.

A principal contribuição deste trabalho reside na demonstração prática da viabilidade da automação de testes funcionais no contexto do desenvolvimento ágil. O estudo comparativo entre Selenium e Cypress fornece subsídios para que outras equipes possam tomar decisões informadas sobre a escolha da ferramenta mais adequada às suas necessidades. Além disso, a implementação bem-sucedida reforça a importância da automação como estratégia para aumentar a confiabilidade, a eficiência e a qualidade do software desenvolvido.

REFERÊNCIAS

ATOUM, I. et al. Challenges of software requirements quality assurance and validation: A systematic literature review. *IEEE Access*, IEEE, v. 9, p. 137613–137634, 2021.

BHANUSHALI, A. Ensuring software quality through effective quality assurance testing: Best practices and case studies. *International Journal of Advances in Scientific Research and Engineering*, v. 26, n. 1, 2023.

CRISPIN, L.; GREGORY, J. *Agile testing: A practical guide for testers and agile teams*. [S.l.]: Pearson Education, 2009.

CUCUMBER Documetation. 2024. Acessado em: 8 de novembro de 2024. Disponível em: ⟨https://cucumber.io/docs/guides/overview/⟩.

DOCUMNETAçãO do Jenkins. 2024. Acessado em: 8 de novembro de 2024. Disponível em: ⟨https://www.jenkins.io/doc/⟩.

GRAHAM, D.; FEWSTER, M. *Software Test Automation: Effective Use of Text Execution Tools*. [S.l.]: Addison Wesley, 1999.

IAN, S. *Engenharia de software*. [S.l.]: 6a. edição, Addison-Wesley/Pearson, 2011.

MWAURA, W. *End-to-End Web Testing with Cypress*. [S.l.]: Packt Publishing, 2021.

MYERS TOM BADGETT, C. S. G. J. *THE ART OF SOFTWARE TESTING*. [S.l.]: John Wiley Sons, Inc, 2012.

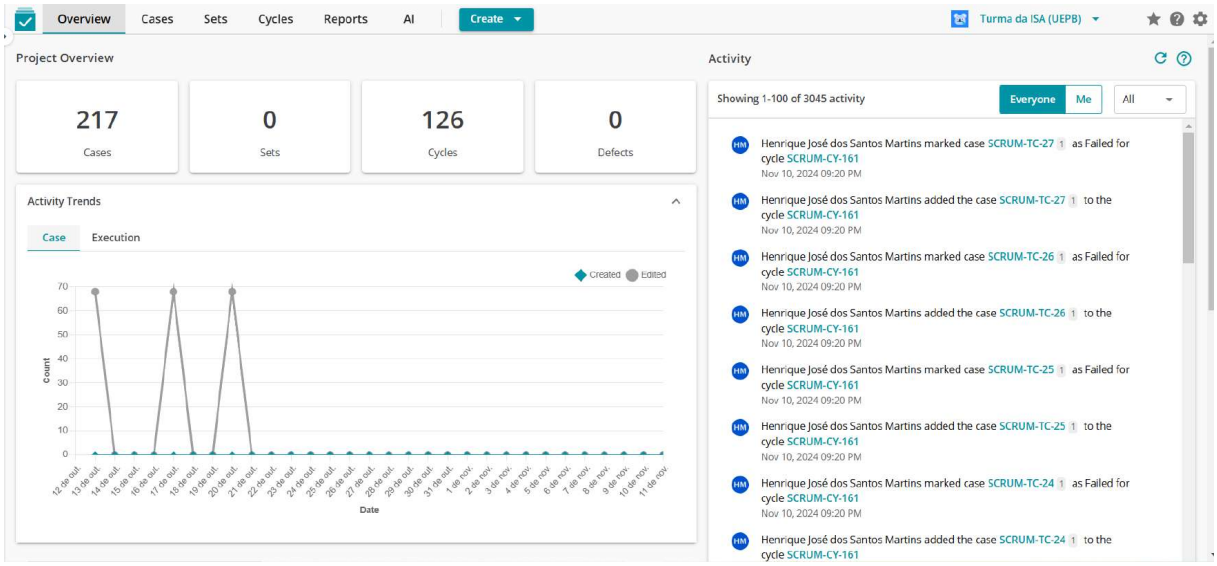
PRESSMAN, R. S.; MAXIM, B. R. *Engenharia de software-9*. [S.l.]: McGraw Hill Brasil, 2021.

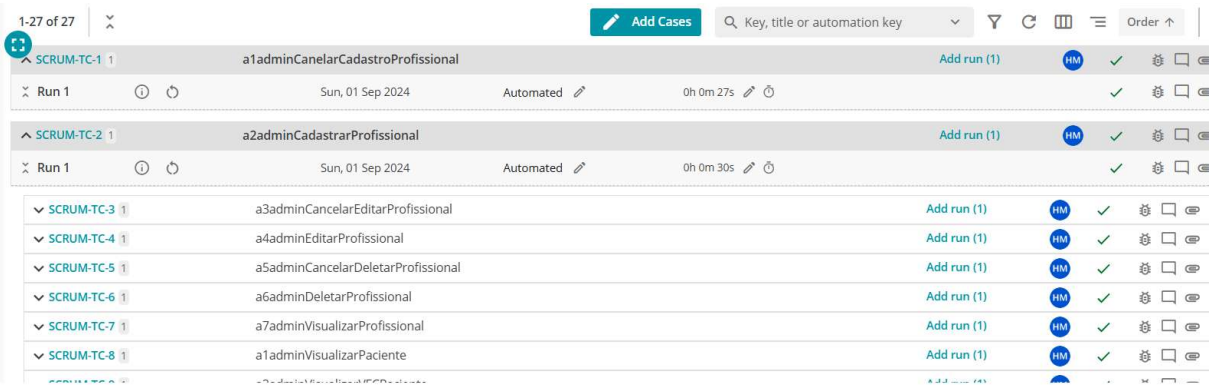
SELENIUM grid: all you need to know. 2024. Acessado em: 8 de novembro de 2024. Disponível em: ⟨https://decode.agency/article/selenium-grid-guide/⟩.

SELENIUM WebDriver documentação. 2024. Acessado em: 8 de novembro de 2024. Disponível em: ⟨https://www.selenium.dev/documentation/webdriver/⟩.

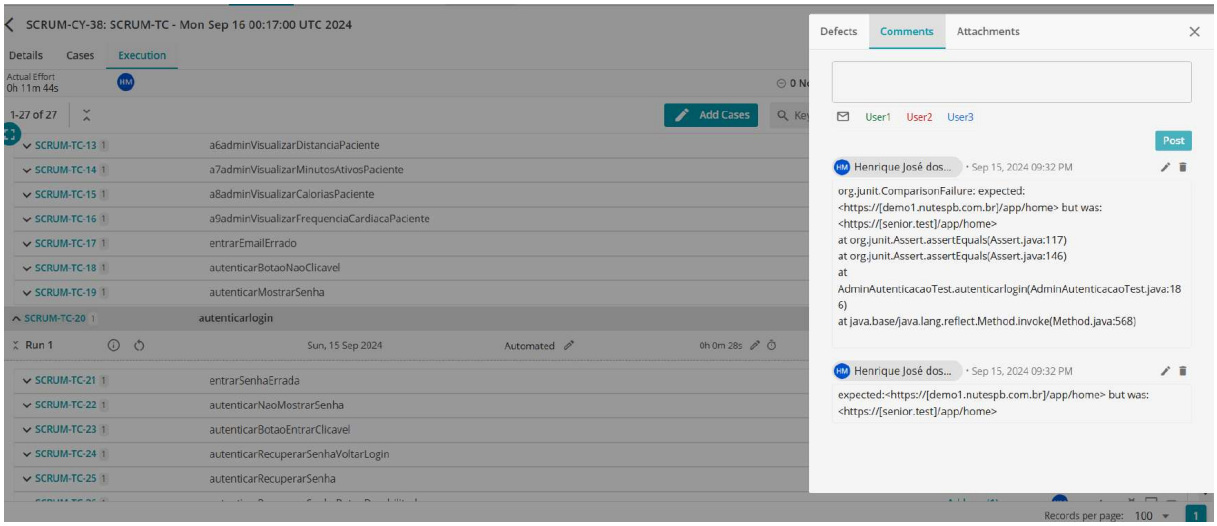
WYNNE MATT; HELLESøY, A. *The Cucumber Book*. [S.l.]: Pragmatic Programmers, 2012.

APÊNDICE A - AIO tests para o Selenium

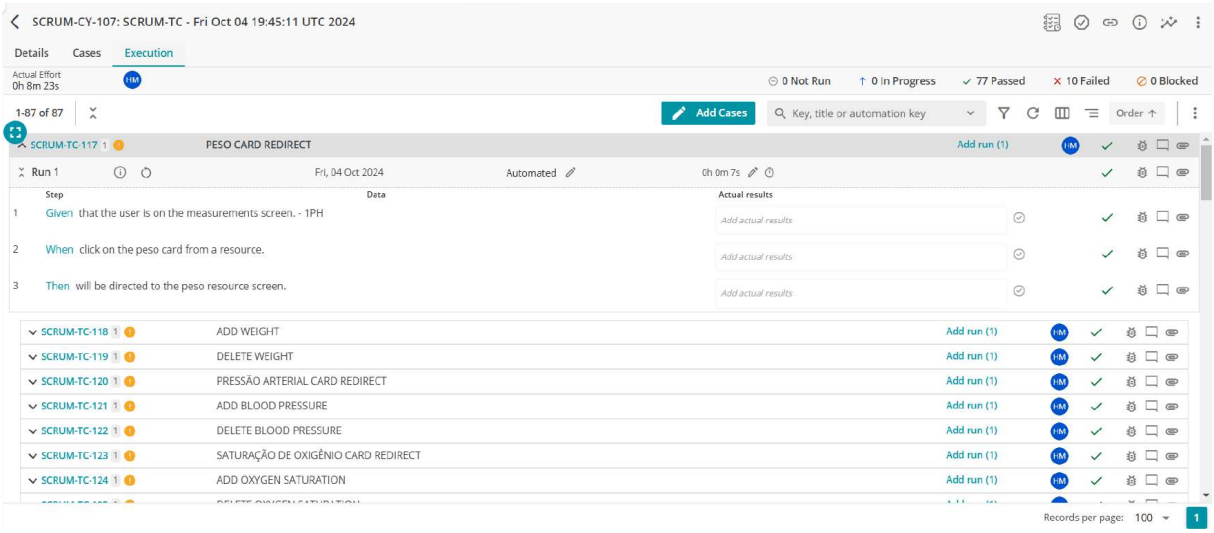

Overview
Cases
Sets
Cycles
Reports
AI
Create
Turma da ISA (UEPB)
Project Overview
217
Cases
0
Sets
126
Cycles
0
Defects
Activity Trends
Case
Execution
Created
Edited
Count
Date
Activity
Showing 1-100 of 3045 activity
Everyone
Me
All
Henrique José dos Santos Martins marked case SCRUM-TC-27 1 as Failed for cycle SCRUM-CY-161
Nov 10, 2024 09:20 PM
Henrique José dos Santos Martins added the case SCRUM-TC-27 1 to the cycle SCRUM-CY-161
Nov 10, 2024 09:20 PM
Henrique José dos Santos Martins marked case SCRUM-TC-26 1 as Failed for cycle SCRUM-CY-161
Nov 10, 2024 09:20 PM
Henrique José dos Santos Martins added the case SCRUM-TC-26 1 to the cycle SCRUM-CY-161
Nov 10, 2024 09:20 PM
Henrique José dos Santos Martins marked case SCRUM-TC-25 1 as Failed for cycle SCRUM-CY-161
Nov 10, 2024 09:20 PM
Henrique José dos Santos Martins added the case SCRUM-TC-25 1 to the cycle SCRUM-CY-161
Nov 10, 2024 09:20 PM
Henrique José dos Santos Martins marked case SCRUM-TC-24 1 as Failed for cycle SCRUM-CY-161
Nov 10, 2024 09:20 PM
Henrique José dos Santos Martins added the case SCRUM-TC-24 1 to the cycle SCRUM-CY-161

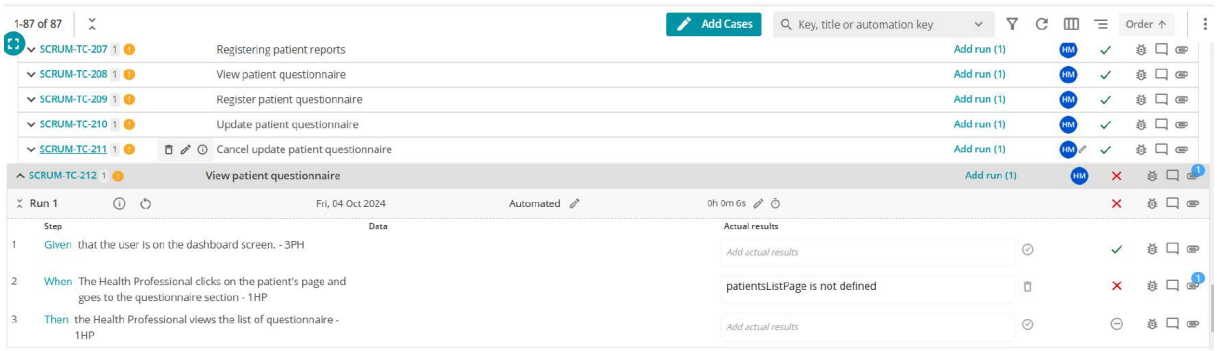

1-27 of 27
Add Cases
Key, title or automation key
Order
SCRUM-TC-1 1
a1adminCanelarCadastroProfissional
Add run (1)
HM
Run 1
Sun, 01 Sep 2024
Automated
0h 0m 27s
SCRUM-TC-2 1
a2adminCadastrarProfissional
Add run (1)
HM
Run 1
Sun, 01 Sep 2024
Automated
0h 0m 30s
SCRUM-TC-3 1
a3adminCancelarEditarProfissional
Add run (1)
HM
SCRUM-TC-4 1
a4adminEditarProfissional
Add run (1)
HM
SCRUM-TC-5 1
a5adminCancelarDeletarProfissional
Add run (1)
HM
SCRUM-TC-6 1
a6adminDeletarProfissional
Add run (1)
HM
SCRUM-TC-7 1
a7adminVisualizarProfissional
Add run (1)
HM
SCRUM-TC-8 1
a1adminVisualizarPaciente
Add run (1)
HM

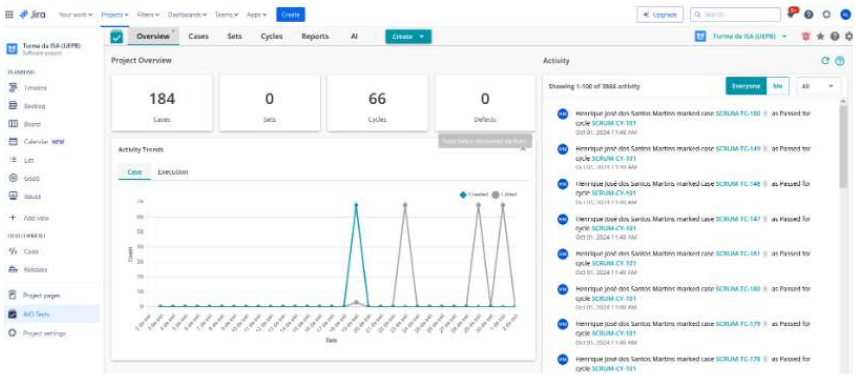

SCRUM-TC-16 1
a9adminVisualizarFrequenciaCardiacaPaciente
Add run (1)
SCRUM-TC-17 1
entrarEmailErrado
Add run (1)
SCRUM-TC-18 1
autenticarBotaoNaoClicavel
Add run (1)
SCRUM-TC-19 1
autenticarMostrarSenha
Add run (1)
SCRUM-TC-20 1
autenticarlogin
Add run (1)
Run 1
Sun, 15 Sep 2024
Automated
0h 0m 28s
SCRUM-TC-21 1
entrarSenhaErrada
Add run (1)
SCRUM-TC-22 1
autenticarNaoMostrarSenha
Add run (1)
SCRUM-TC-23 1
Add run (1)

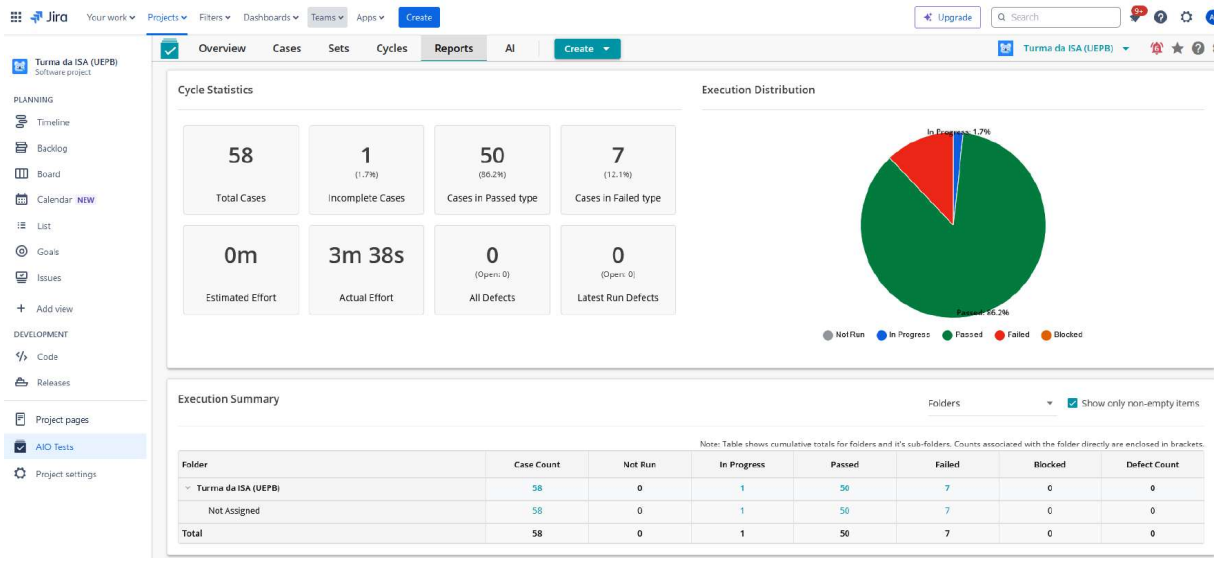

SCRUM-CY-38: SCRUM-TC - Mon Sep 16 00:17:00 UTC 2024
Details
Cases
Execution
Actual Effort
0h 11m 44s
1-27 of 27
Add Cases
SCRUM-TC-13 a6adminVisualizarDistanciaPaciente
SCRUM-TC-14 a7adminVisualizarMinutosAtivosPaciente
SCRUM-TC-15 a8adminVisualizarCaloriasPaciente
SCRUM-TC-16 a9adminVisualizarFrequenciaCardiacaPaciente
SCRUM-TC-17 entrarEmailErrado
SCRUM-TC-18 autenticarBotaoNaoClicavel
SCRUM-TC-19 autenticarMostrarSenha
SCRUM-TC-20 autenticarlogin
Run 1
Sun, 15 Sep 2024
Automated
0h 0m 28s
SCRUM-TC-21 entrarSenhaErrada
SCRUM-TC-22 autenticarNaoMostrarSenha
SCRUM-TC-23 autenticarBotaoEntrarClicavel
SCRUM-TC-24 autenticarRecuperarSenhaVoltarLogin
SCRUM-TC-25 autenticarRecuperarSenha
Defects
Comments
Attachments
User1
User2
User3
Post
Henrique José dos...
• Sep 15, 2024 09:32 PM
org.junit.ComparisonFailure: expected:
<https://[demo1.nutespb.com.br]/app/home> but was:
<https://[senior.test]/app/home>
at org.junit.Assert.assertEquals(Assert.java:117)
at org.junit.Assert.assertEquals(Assert.java:146)
at
AdminAutenticacaoTest.autenticarlogin(AdminAutenticacaoTest.java:18
6)
at java.base/java.lang.reflect.Method.invoke(Method.java:568)
Henrique José dos...
• Sep 15, 2024 09:32 PM
expected:<https://[demo1.nutespb.com.br]/app/home> but was:
<https://[senior.test]/app/home>
Records per page: 100
1

APÊNDICE B - AIO tests para o Cypress


SCRUM-CY-107: SCRUM-TC - Fri Oct 04 19:45:11 UTC 2024
Details
Cases
Execution
Actual Effort
0h 8m 23s
0 Not Run
0 In Progress
77 Passed
10 Failed
0 Blocked
1-87 of 87
Add Cases
Key, title or automation key
Order
SCRUM-TC-117
PESO CARD REDIRECT
Add run (1)
Run 1
Fri, 04 Oct 2024
Automated
0h 0m 7s
Step
Data
Actual results
1 Given that the user is on the measurements screen. - 1PH
2 When click on the peso card from a resource.
3 Then will be directed to the peso resource screen.
Add actual results
SCRUM-TC-118 ADD WEIGHT
SCRUM-TC-119 DELETE WEIGHT
SCRUM-TC-120 PRESSÃO ARTERIAL CARD REDIRECT
SCRUM-TC-121 ADD BLOOD PRESSURE
SCRUM-TC-122 DELETE BLOOD PRESSURE
SCRUM-TC-123 SATURAÇÃO DE OXIGÊNIO CARD REDIRECT
SCRUM-TC-124 ADD OXYGEN SATURATION
Records per page: 100
1


1-87 of 87
Add Cases
Key, title or automation key
Order
SCRUM-TC-207 Registering patient reports Add run (1)
SCRUM-TC-208 View patient questionnaire Add run (1)
SCRUM-TC-209 Register patient questionnaire Add run (1)
SCRUM-TC-210 Update patient questionnaire Add run (1)
SCRUM-TC-211 Cancel update patient questionnaire Add run (1)
SCRUM-TC-212 View patient questionnaire Add run (1)
Run 1 Fri, 04 Oct 2024 Automated 0h 0m 6s
Step Data Actual results
1 Given that the user is on the dashboard screen. - 3PH Add actual results
2 When The Health Professional clicks on the patient's page and goes to the questionnaire section - 1HP patientsListPage is not defined
3 Then the Health Professional views the list of questionnaire - 1HP Add actual results


Overview
Cases
Sets
Cycles
Reports
AI
Create
Project Overview
184
Cases
0
Sets
66
Cycles
0
Defects
Activity Trends
Case
Execution
Activity
Everyone
Me
All


Jira
Your work
Projects
Filters
Dashboards
Teams
Apps
Create
Upgrade
Search
Turma da ISA (UEPB)
Software project
PLANNING
Timeline
Backlog
Board
Calendar NEW
List
Goals
Issues
Add view
DEVELOPMENT
Code
Releases
Project pages
AIO Tests
Project settings
Overview
Cases
Sets
Cycles
Reports
AI
Create
Turma da ISA (UEPB)
Cycle Statistics
58
Total Cases
1
(1.7%)
Incomplete Cases
50
(86.2%)
Cases in Passed type
7
(12.1%)
Cases in Failed type
0m
Estimated Effort
3m 38s
Actual Effort
0
(Open: 0)
All Defects
0
(Open: 0)
Latest Run Defects
Execution Distribution
In Progress: 1.7%
Passed: 86.2%
Not Run
In Progress
Passed
Failed
Blocked
Execution Summary
Folders
Show only non-empty items
Note: Table shows cumulative totals for folders and it's sub-folders. Counts associated with the folder directly are enclosed in brackets.
Folder
Case Count
Not Run
In Progress
Passed
Failed
Blocked
Defect Count
Turma da ISA (UEPB)
58
0
1
50
7
0
0
Not Assigned
58
0
1
50
7
0
0
Total
58
0
1
50
7
0
0