UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Matheus da Silva Cristino

Análise de Ferramentas de Teste E2E: Um Estudo de Caso em Projetos Web utilizando Cypress, Selenium e Playwright

Uberlândia, Brasil

2024

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Matheus da Silva Cristino

Análise de Ferramentas de Teste E2E: Um Estudo de Caso em Projetos Web utilizando Cypress, Selenium e Playwright

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Sistemas de Informação.

Orientador: Fabiano Azevedo Dorça

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2024

Resumo

A automação de testes oferece uma solução eficiente para lidar com os desafios de testar aplicações web, especialmente à medida que sua complexidade aumenta. Testes manuais podem ser exaustivos e demorados, enquanto a automação permite a execução de testes de forma mais precisa e sem a sobrecarga humana associada aos processos manuais. No cenário competitivo do desenvolvimento de software, garantir a qualidade e a confiabilidade das aplicações web tornou-se uma tarefa cada vez mais desafiadora, devido ao aumento da complexidade das aplicações e às crescentes exigências dos usuários. Testes *end-to-end (E2E)* desempenham um papel fundamental na validação das funcionalidades, especialmente no *front-end*, e a automação desses testes ajuda a tornar o ciclo de desenvolvimento mais ágil e eficiente. No entanto, a escolha da ferramenta adequada para esses testes pode influenciar significativamente a eficácia do processo. Este estudo tem como objetivo auxiliar na escolha de ferramentas de teste automatizado, comparando três opções populares: Cypress, Selenium e Playwright.

Serão avaliados critérios-chave como instalação, configuração, codificação, facilidade de uso, desempenho e manutenibilidade. A coleta de dados foi realizada por meio de testes em uma aplicação web, oferecendo *insights* sobre as vantagens e desvantagens de cada ferramenta. Espera-se que este estudo forneça orientações valiosas para desenvolvedores que buscam implementar uma estratégia eficaz de automação de testes em seus projetos de aplicações web.

Palavras-chave: Cypress, Selenium, Playwright, Automação, Testes End-to-End.

Lista de ilustrações

Figura 1 – Aplicação Web React . . . 17
Figura 2 – Tela de Boas Vindas do Cypress. . . . 19
Figura 3 – Estrutura de Pastas do Cypress . . . 19
Figura 4 – Testes de Exemplo do Cypress . . . 20
Figura 5 – Estrutura do projeto Cypress . . . 20
Figura 6 – Suit de testes Cypress . . . 21
Figura 7 – Commands Cypress . . . 22
Figura 8 – Locators Cypress . . . 23
Figura 9 – Execução da Suit de testes no modo visual Cypress . . . 24
Figura 10 – Execução modo headless Cypress . . . 25
Figura 11 – Resultado tempo médio das execuções no Cypress . . . 25
Figura 12 – Dependências Java . . . 27
Figura 13 – ChromeDriver . . . 27
Figura 14 – Estrutura projeto Selenium com Java . . . 29
Figura 15 – Página de contas Selenium . . . 30
Figura 16 – Página de extrato Selenium . . . 30
Figura 17 – Página de home Selenium . . . 31
Figura 18 – Página de login Selenium . . . 31
Figura 19 – Página de movimentações Selenium . . . 32
Figura 20 – Suíte de testes Selenium . . . 32
Figura 21 – Execução modo visual Selenium . . . 33
Figura 22 – Execução modo Headless Selenium . . . 33
Figura 23 – Resultados das execuções no Selenium . . . 34
Figura 24 – Estrutura do projeto Playwright . . . 36
Figura 25 – Conta Page Playwright . . . 37
Figura 26 – Home Page Playwright . . . 37
Figura 27 – Login Page Playwright . . . 38
Figura 28 – Movimentações Page Playwright . . . 38
Figura 29 – Suíte de testes Playwright . . . 39
Figura 30 – Configuração modo visual e navegador no Playwright . . . 40
Figura 31 – Resultado execuções no Playwright . . . 40

Lista de tabelas

Tabela 1 – Comparação entre Cypress, Selenium e Playwright 47

Lista de abreviaturas e siglas

E2E - End-to-End (De ponta a ponta)

CI/CD - Continuous Integration/Continuous Delivery (Integração Contínua/Entrega Contínua)

UAT - User Acceptance Testing (Testes de Aceitação do Usuário)

BDD - Behavior Driven Development (Desenvolvimento Orientado por Comportamento)

WEB - World Wide Web

OWASP - Open Web Application Security Project

ZAP - Zed Attack Proxy

API - Application Programming Interface

UI - User Interface (Interface do Usuário)

URL - Uniform Resource Locator

VSCode - Visual Studio Code

JS - JavaScript

Npm - Node Package Manager

NodeJs - Node JavaScript

JDK - Java Development Kit

IDE - Integrated Development Environment (Ambiente de Desenvolvimento Integrado)

UI - User Interface (Interface do Usuário)

CI/CD - Continuous Integration/Continuous Delivery (Integração Contínua/Entrega Contínua)

GUI - Graphical User Interface (Interface Gráfica do Usuário)

DOM - Document Object Model

API - Application Programming Interface

CI/CD - Continuous Integration/Continuous Delivery (Integração Contínua/Entrega Contínua)

Selenium Grid - Sistema de Execução Paralela do Selenium

Sumário

1 INTRODUÇÃO 8

2 REVISÃO BIBLIOGRÁFICA 10
2.1 Qualidade de software 10
2.2 Tipos de testes de software 12
2.3 Testes de Software End-to-End (E2E) 13
2.4 Ferramentas de Teste E2E (Cypress, Selenium e Playwright) 15

3 DESENVOLVIMENTO 17
3.1 JavaScript com Cypress 18
3.1.1 Codificação dos cenários de testes 20
3.2 Java com Selenium 26
3.2.1 Codificação dos cenários de testes 28
3.3 Javascript com Playwright 34

4 RESULTADOS E DISCUSSÕES 41
4.1 Setup e configuração 41
4.2 Desempenho 42
4.3 Linguagens suportadas 42
4.4 Ferramentas de depuração 42
4.5 Navegadores suportados 43
4.6 Suporte a Espera Automática 44
4.7 Re-tentativas (Para Flake Test) 44
4.8 Suporte a várias abas 45
4.9 Suporte a testes isolados 45
4.10 Escalabilidade 45
4.11 Suporte para gravação e reprodução 46

5 CONCLUSÃO 48

REFERÊNCIAS 49

1 Introdução

No cenário dinâmico e altamente competitivo do desenvolvimento de software, garantir a qualidade e a confiabilidade das aplicações web é essencial para o sucesso de um produto, especialmente em ambientes ágeis, onde entregas frequentes e incrementais têm se tornado um padrão (FOWLER; HIGHSMITH, 2001).

Para assegurar que o software apresente o mínimo possível de defeitos durante o uso pelos usuários, os testes de software desempenham um papel crucial. Eles são uma etapa essencial no ciclo de desenvolvimento, responsáveis por garantir que uma aplicação funcione conforme o esperado e atenda aos requisitos especificados. Os testes envolvem a validação de funcionalidades, a identificação de falhas e a verificação da conformidade do software com as especificações técnicas e de negócio. Existem várias abordagens para os testes, sendo os testes manuais e automatizados as mais comuns (MYERS; SANDLER; BADGETT, 2011).

Embora os testes manuais sejam fundamentais, eles podem ser exaustivos e suscetíveis a erros humanos. Quando realizados manualmente, os testadores seguem um conjunto de etapas predefinidas para verificar se o software funciona corretamente. No entanto, à medida que os projetos se tornam mais complexos e os ciclos de desenvolvimento mais curtos, a repetição dos testes manuais pode se tornar uma tarefa árdua, exigindo muito tempo e esforço. Além disso, a natureza repetitiva desses testes pode levar a lapsos de atenção, resultando em falhas não detectadas ou resultados inconsistentes.

A exaustividade e a propensão a erros nos testes manuais destacam a necessidade de alternativas mais eficientes, como a automação de testes, que busca mitigar esses problemas, proporcionando maior precisão, rapidez e repetibilidade no processo de validação do software (GAROUSI; FELDERER; MäNTYLä, 2019).

A automação de testes é uma prática que utiliza ferramentas e *scripts* para executar casos de teste de forma automática, sem a necessidade de intervenção humana constante. Essa prática se tornou fundamental no desenvolvimento de software moderno. Especificamente, a automação de testes *E2E* permite a execução de vários cenários de teste de forma rápida e repetitiva, cobrindo desde as interações básicas do usuário até fluxos complexos envolvendo múltiplas funcionalidades. Ela é especialmente valiosa no *front-end* das aplicações, onde a interação direta com o usuário ocorre e onde os defeitos podem ter um impacto significativo na satisfação do cliente.

No entanto, a implementação eficaz da automação de testes *E2E* não é trivial. A escolha da ferramenta adequada pode ser desafiadora, pois cada opção disponível no mercado possui suas próprias características, pontos fortes e limitações. Ferramentas po-

pulares como Cypress, Selenium e Playwright oferecem diferentes abordagens e recursos, tornando a decisão sobre qual utilizar uma tarefa complexa que requer uma análise cuidadosa.

Este estudo se propõe a facilitar essa escolha, comparando as três ferramentas mencionadas com base em critérios pré-definidos. Serão avaliados aspectos como facilidade de instalação e configuração, sintaxe e legibilidade do código, velocidade de execução dos testes, recursos disponíveis e manutenibilidade ao longo do tempo. Essa análise foi realizada por meio de testes em uma aplicação web real, permitindo a coleta de dados empíricos e a extração de *insights* valiosos sobre o desempenho de cada ferramenta em cenários práticos.

Os resultados deste estudo visam fornecer orientações valiosas para estudantes, desenvolvedores e testadores que buscam implementar uma estratégia eficaz de automação de testes *E2E* em seus projetos. Ao comparar as principais opções disponíveis no mercado, espera-se capacitar os profissionais a tomar decisões informadas sobre qual ferramenta melhor se adapta às necessidades específicas de seus projetos, contribuindo assim para a entrega de aplicações web de alta qualidade em um ambiente de desenvolvimento ágil.

Este capítulo apresentou o contexto geral do trabalho, seus objetivos e justificativas, destacando a importância dos testes de software, especialmente a automação de testes *E2E*, no desenvolvimento de aplicações web. O próximo capítulo, Revisão Bibliográfica, abordará os principais conceitos relacionados à qualidade de software e os diferentes tipos de testes que podem ser realizados ao longo do ciclo de desenvolvimento, com foco nas ferramentas de automação de testes *E2E*, como Cypress, Selenium e Playwright. O Capítulo 3, Desenvolvimento, detalhará a aplicação utilizada para os testes, as ferramentas escolhidas, a codificação dos cenários e suas respectivas complexidades e usabilidades na criação de testes *E2E*. O Capítulo 4, Resultados e Discussões, apresentará os resultados obtidos, comparando os pontos fortes e fracos de cada ferramenta, seguido pelo Capítulo 5, Conclusão, que sintetiza as descobertas do trabalho e oferece recomendações para o uso das ferramentas de testes E2E, auxiliando desenvolvedores e equipes de qualidade na escolha da ferramenta de automação de testes mais adequada ao seu contexto.

2 Revisão Bibliográfica

Neste capítulo serão descritos os principais conceitos relacionados a qualidade de software e tipos de testes que podem ser realizado no ciclo de desenvolvimento. Serão apresentando também a ideia de testes *End To End* e ferramentas para auxiliar a automação de testes de aplicações web como Cypress, Selenium e Playwright.

2.1 Qualidade de software

Qualidade de Software é a área no desenvolvimento de software que visa garantir a conformidade de um produto de software com seus respectivos requisitos, especificações e padrões pré-definidos. Tem-se o objetivo de garantir também características como confiabilidade, usabilidade, eficiência, manutenibilidade, portabilidade e segurança de tudo que gira em torno de um software. O foco na Qualidade de Software vem aumentando cada vez mais devido à alta demanda, à competição no mercado de desenvolvimento de software e às rigorosas exigências dos usuários. No entanto, a atenção à qualidade de software remonta às décadas de 1960 e 1970, quando os primeiros programas de computador começaram a ser desenvolvidos em larga escala (PRESSMAN, 2014).

Para as organizações, desenvolver produtos excepcionais pode ser desafiador, mas garantir o desempenho deles ao longo do tempo é ainda mais complexo; isso é o que significa assegurar uma boa qualidade de software. As empresas precisam ter a certeza de que podem atender às expectativas de seus clientes todos os dias. No contexto de testes, inicialmente, o foco estava na correção de erros (debugging) e na conformidade com os requisitos. Com o tempo, abordagens mais sistemáticas, como a Engenharia de Software, foram desenvolvidas para aprimorar a qualidade do software (MYERS; SANDLER; BADGETT, 2011).

A área tem visto muitos avanços nos últimos anos, como exemplificado pelos seguintes pontos:

Integração Contínua/Entrega Contínua (CI/CD): Essa abordagem trata da automação dos processos de desenvolvimento, testes e implantação, o que melhora significativamente a qualidade do software, reduzindo erros introduzidos durante alterações no código e agilizando o processo de correção se ocorrerem problemas (LAUKKANEN; ITKONEN; LASSENIUS, 2017). Ferramentas como Jenkins, GitHub Actions e GitLab CI/CD têm sido amplamente utilizadas para implementar pipelines de CI/CD, permitindo feedback rápido e maior confiabilidade no ciclo de desenvolvimento (SHAHIN; BABAR; ZHU, 2017).

Métodos Ágeis: Práticas ágeis, como Scrum e Kanban, promovem a colaboração, a adaptabilidade às mudanças e a entrega frequente de software, resultando em software mais alinhado com as necessidades dos usuários finais (FOWLER; HIGHSMITH, 2001). Além disso, frameworks como SAFe (Scaled Agile Framework) têm sido adotados para escalar práticas ágeis em grandes organizações, garantindo alinhamento estratégico e qualidade em projetos complexos (LEFFINGWELL, 2020).

Testes Automatizados: Ferramentas de teste automatizado, incluindo testes unitários, de integração e de aceitação do usuário, ajudam a identificar problemas rapidamente, permitindo correções antes que afetem os usuários (MESZAROS, 2007). Ferramentas modernas como Cypress, Playwright e Selenium têm se destacado por sua capacidade de realizar testes de ponta a ponta (E2E) de forma eficiente e integrada a pipelines de CI/CD.

DevOps: A cultura DevOps integra desenvolvimento e operações, enfatizando a colaboração e a comunicação entre equipes para melhorar a qualidade, velocidade e confiabilidade do desenvolvimento de software (GAROUSI; FELDERER; MäNTYLä, 2019). Ferramentas como Docker e Kubernetes têm sido fundamentais para a implementação de práticas DevOps, permitindo a criação de ambientes consistentes e escaláveis para desenvolvimento e produção (KIM et al., 2016).

Testes de Acessibilidade: Com a crescente preocupação com a inclusão digital, ferramentas como axe-core e Pa11y têm sido amplamente utilizadas para garantir que aplicações estejam em conformidade com as diretrizes de acessibilidade, como as WCAG (Web Content Accessibility Guidelines) (SYSTEMS, 2021).

Testes Baseados em IA: A inteligência artificial tem sido aplicada para melhorar a eficiência e a precisão dos testes de software. Ferramentas como Applitools e Testim utilizam IA para identificar mudanças visuais e prever falhas, reduzindo o esforço manual e aumentando a cobertura de testes (APPLITOOLS, 2023).

Esses avanços têm transformado a área de qualidade, tornando os processos mais rápidos, precisos e alinhados às necessidades modernas de desenvolvimento de software.

Alguns Exemplos de Abordagens e Ferramentas:

Testes Unitários: *Frameworks* populares como *JUnit, NUnit* e *pytest* são usados para testes unitários em várias linguagens de programação (MESZAROS, 2007).

Integração Contínua: Ferramentas como *Jenkins*, *Travis CI* e *GitLab CI* automatizam a integração contínua, permitindo a detecção precoce de problemas de integração (LAUKKANEN; ITKONEN; LASSENIUS, 2017).

Testes de Aceitação do Usuário (*UAT*): Ferramentas como *Cucumber* e *SpecFlow* permitem escrever casos de teste em linguagem natural, facilitando a colaboração entre desenvolvedores e partes interessadas (GAROUSI; FELDERER; MäNTYLä, 2019).

Monitoramento de Aplicações: Ferramentas como *New Relic* e *Datadog* monitoram o desempenho da aplicação em tempo real, identificando problemas rapidamente (PRESSMAN, 2014).

Revisões de Código: Plataformas como *GitHub* e *Bitbucket* oferecem funcionalidades para revisões colaborativas de código, melhorando a qualidade do código por meio da detecção de *bugs* e do estabelecimento de boas práticas (PRESSMAN, 2014).

2.2 Tipos de testes de software

Testes de software são uma etapa essencial no que diz respeito ao processo de desenvolvimento de software. Consiste em validar se o software atende aos requisitos definidos e se também funciona como o esperado. Para isso, podem ser utilizados diversos tipos de testes, cada um com seu respectivo objetivo, como por exemplo: testes unitários que visam garantir que cada método do nosso teste esteja coberto com testes, ou seja, valida a menor unidade possível de uma função, como métodos, funções e *controllers*, etc. Existem também os testes de integração, que têm o foco em validar integrações de fato, seja ela entre métodos e classes na própria aplicação ou integrações externas (MYERS; SANDLER; BADGETT, 2011).

Outro tipo de testes são os de sistema, que validam a aplicação como um todo, e também os testes de aceitação, que são realizados geralmente na etapa final do desenvolvimento, onde os *stakeholders* validam se o que foi pedido foi realmente entregue, por exemplo (PRESSMAN, 2014).

A prática de realizar testes de software começou logo após a preocupação com software de qualidade, conforme mencionado anteriormente. Inicialmente, os testes eram realizados manualmente, mas à medida que os sistemas de software se tornaram mais complexos, surgiram técnicas automatizadas e ferramentas de teste. Apesar disso, os testes manuais ainda são essenciais. Essas ferramentas surgiram para acelerar o ciclo de desenvolvimento do software (MYERS; SANDLER; BADGETT, 2011).

Alguns exemplos de tipos de testes de software incluem:

Testes Unitários: Verificam unidades individuais de código para garantir que funcionem conforme o esperado (MESZAROS, 2007).

Testes de Integração: Testam a interação entre diferentes unidades ou módulos no sistema (MYERS; SANDLER; BADGETT, 2011).

Testes de Sistema: Avaliam o comportamento do sistema como um todo, verificando se ele atende aos requisitos (PRESSMAN, 2014).

Testes de Aceitação do Usuário: São realizados pelos usuários finais para validar se o sistema atende aos requisitos do usuário (GAROUSI; FELDERER; MäNTYLä, 2019).

Testes de Regressão: Garantem que novas alterações no código não afetem funcionalidades existentes (MESZAROS, 2007).

Testes de Desempenho: Avaliam o desempenho do software sob diferentes condições de carga (PRESSMAN, 2014).

Testes de Segurança: Identificam vulnerabilidades de segurança no software (PRESSMAN, 2014).

Além disso, avanços notáveis na área de testes de software incluem:

Ambientes de Integração Contínua Automatizados (CI): A automação de testes em ambientes de CI permite a detecção precoce de problemas, garantindo que as alterações no código não quebrem funcionalidades existentes (LAUKKANEN; ITKONEN; LASSENIUS, 2017).

Testes de Regressão Automatizados: Ferramentas como Selenium para testes de interface do usuário e *frameworks* de teste de unidade como *JUnit* e *pytest* permitem a execução automatizada de testes de regressão para garantir que novas alterações não causem regressões em funcionalidades previamente testadas (ZHAO; ZHANG; ZHANG, 2014).

Testes Baseados em Comportamento (*BDD*): Ferramentas como *Cucumber* e *SpecFlow* permitem que casos de teste sejam escritos em linguagem natural, melhorando a comunicação entre desenvolvedores, testadores e partes interessadas (GAROUSI; FELDERER; MäNTYLä, 2019).

Testes de Desempenho e Carga: Ferramentas como *Apache JMeter* e *LoadRunner* são usadas para simular cargas de usuários e avaliar o desempenho do software sob diferentes condições (PRESSMAN, 2014).

Testes de Segurança: Ferramentas como *OWASP ZAP* e *Burp Suite* são usadas para identificar vulnerabilidades de segurança em aplicações web, ajudando a garantir a proteção contra ameaças cibernéticas (PRESSMAN, 2014).

2.3 Testes de Software End-to-End (E2E)

Os Testes de Software *End-to-End (E2E)* são uma técnica que avalia um aplicativo de ponta a ponta, simulando o comportamento do usuário real em um ambiente de teste. Os testes E2E verificam se o software funciona corretamente desde o início até o final do processo de desenvolvimento, incluindo a interação entre diferentes componentes do sistema. Por exemplo, em uma aplicação web de *e-commerce*, um teste *E2E* pode envolver o login do usuário, a seleção de um produto, a adição ao carrinho, o processo de *checkout* e a confirmação do pedido (GAROUSI; FELDERER; MäNTYLä, 2019). Dentro desse fluxo, podem existir diversas variações, como a aplicação de cupons ou a inserção de

dados incorretos em alguma etapa do processo, o que torna os testes *E2E* essenciais para garantir a robustez do sistema.

Inicialmente, esses testes eram realizados manualmente. No entanto, com o avanço da tecnologia e o aumento na complexidade das aplicações, tornou-se crucial automatizar esses testes para economizar tempo e garantir uma cobertura abrangente dos cenários de teste (MESZAROS, 2007). É importante notar que os testes *E2E* demandam um esforço maior em comparação a outros tipos de testes, pois cobrem toda a aplicação.

Para auxiliar nesse tipo de teste, existem ferramentas que facilitam a criação, execução e documentação dos testes. A metodologia *BDD (Behavior Driven Development)* é uma delas, que consiste em escrever casos de teste *E2E* em linguagem natural, de forma que o usuário final realizaria suas ações na aplicação. Isso promove a colaboração entre equipes de desenvolvimento, testes e negócios (GAROUSI; FELDERER; MäNTYLä, 2019). Uma ferramenta amplamente utilizada é o *Cucumber*, que, através da linguagem *Gherkin*, utiliza palavras-chave como "Dado", "Quando", "Então"e "E"para facilitar o entendimento de um determinado caso de uso. Por exemplo, em um fluxo de login de *e-commerce*, o teste escrito utilizando essa técnica ficaria assim:

- **Dado** que o usuário acessa a página inicial;
- **E** digita um e-mail cadastrado no sistema;
- **E** digita a senha cadastrada;
- **Quando** clica no botão de login;
- **Então** o usuário acessa a área logada do sistema.

Dessa forma, o teste fica muito mais claro tanto para a equipe técnica quanto para a equipe de negócios (FOWLER; HIGHSMITH, 2001). Outras ferramentas que auxiliam os testes *E2E* são as de testes automatizados, que consistem na criação de *scripts* que podem ser executados em diferentes navegadores e plataformas. Além disso, essas ferramentas podem ser executadas várias vezes a uma velocidade que um humano não conseguiria, garantindo consistência nos resultados e eliminando o risco de erros humanos (MESZAROS, 2007).

A automação de testes *E2E* tem proporcionado uma eficiência significativa no ciclo de desenvolvimento de software. Por exemplo, considerando um cenário de teste que abrange diversas variações e precisa ser validado em vários navegadores e plataformas, o esforço manual seria extremamente alto e propenso a erros. Com a automação, esse processo se torna mais ágil e confiável (GAROUSI; FELDERER; MäNTYLä, 2019).

Alguns avanços recentes na área de automação de testes *E2E* incluem:

- **Integração com Estruturas de Teste:** A automação de testes *E2E* foi integrada a estruturas de teste modernas, como *TestNG* e *JUnit*, facilitando a execução e o relatório de testes *E2E* juntamente com outros tipos de testes (MESZAROS, 2007).
- **Implementação em Nuvem:** Serviços em nuvem, como *BrowserStack* e *Sauce Labs*, permitem que os testes *E2E* sejam executados em uma variedade de dispositivos e navegadores, garantindo a compatibilidade com diferentes ambientes (ZHAO; ZHANG; ZHANG, 2014).
- **Headless Browsing:** Ferramentas como Puppeteer, Selenium e Cypress oferecem suporte para navegação "headless", o que significa que os testes podem ser executados em um ambiente sem interface gráfica, tornando a execução dos testes mais rápida e eficiente (ZHAO; ZHANG; ZHANG, 2014).
- **Integração com Ferramentas de Orquestração:** Ferramentas de orquestração, como *Kubernetes* e *Docker Swarm*, permitem a execução e escalabilidade eficiente de testes *E2E* em ambientes de contêineres, facilitando a integração dos testes com pipelines de CI/CD (LAUKKANEN; ITKONEN; LASSENIUS, 2017).

2.4 Ferramentas de Teste E2E (Cypress, Selenium e Playwright)

Ferramentas de teste *End-to-End (E2E)* são usadas para avaliar a funcionalidade de um aplicativo em um ambiente realista, simulando a interação do usuário com a aplicação, seja ela na web ou em dispositivos móveis. Estas ferramentas são cruciais para testar o fluxo completo da aplicação, desde a entrada do usuário até a saída do aplicativo (ZHAO; ZHANG; ZHANG, 2014).

O Selenium é uma das ferramentas de teste *E2E* mais antigas e amplamente utilizadas. Lançado em 2004, é uma ferramenta de código aberto que suporta várias linguagens de programação, incluindo Java, Python, Ruby e JavaScript (ZHAO; ZHANG; ZHANG, 2014). Em contraste, o Cypress é uma ferramenta de teste *E2E* mais recente, lançada em 2014, e é exclusivamente voltada para JavaScript/TypeScript (GAROUSI; FELDERER; MäNTYLä, 2019). Já o Playwright, lançado em 2019, é uma ferramenta de código aberto que suporta diversas linguagens de programação, incluindo JavaScript, Python, Java e .NET.

Tanto o Cypress quanto o Selenium e o Playwright são utilizados para realizar testes *E2E* das funcionalidades de aplicações web, abrangendo aspectos como navegação, entrada e saída de dados, e a interface (GAROUSI; FELDERER; MäNTYLä, 2019). Com os avanços tecnológicos, é agora possível realizar testes *E2E* também em dispositivos móveis e tablets utilizando essas ferramentas. Outros avanços notáveis incluem o suporte a vários navegadores, como Chrome, Firefox, Safari e Edge (ZHAO; ZHANG; ZHANG,

2014). Além disso, essas ferramentas integram-se facilmente com plataformas de Integração Contínua e Entrega Contínua (CI/CD), podendo ser integradas a plataformas como *GitHub Actions*, *Jenkins*, e também a serviços em nuvem como *Azure DevOps* e *AWS* (LAUKKANEN; ITKONEN; LASSENIUS, 2017).

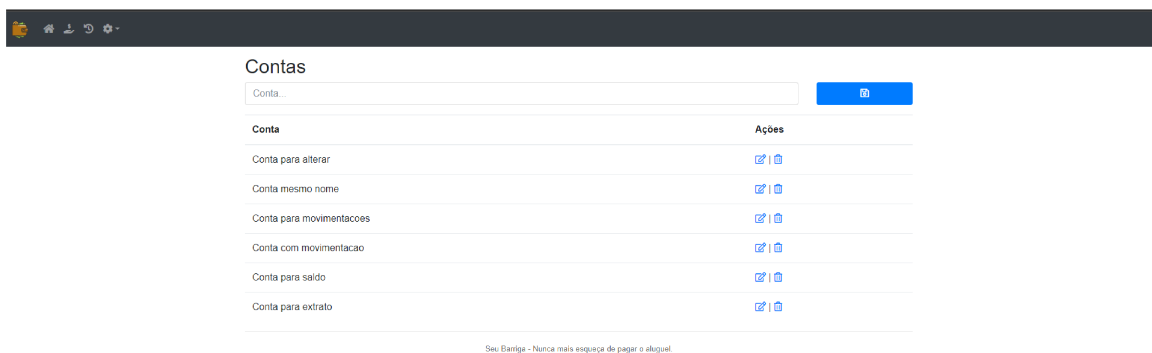
3 Desenvolvimento

Neste capítulo, será apresentado o desenvolvimento do trabalho proposto, que inclui a apresentação da aplicação a ser utilizada, o detalhamento das ferramentas, bem como sua complexidade e usabilidade na criação de testes de *E2E* em aplicações web. Além disso, serão discutidas as linguagens de programação utilizadas.

Tem-se como base de aplicação um site desenvolvido em *React*, disponível no seguinte endereço: https://barrigareact.wcaquino.me/. Esta aplicação é voltada para treinamentos em ferramentas de automação, oferecendo funcionalidades relacionadas à criação e ao gerenciamento de contas bancárias, conforme mostrado na Figura 1. Foram realizados os seguintes cenários de teste pelas ferramentas:

1. Validar a inserção de uma nova conta;
2. Validar a alteração de uma conta já existente;
3. Validar a exibição de erro ao tentar inserir uma conta repetida;
4. Validar a inserção de uma movimentação financeira em uma conta;
5. Validar o cálculo do saldo de uma conta;
6. Validar a remoção de uma movimentação financeira de uma conta.

Figura 1 – Aplicação Web React



Fonte Adaptado de WCAQUINO (https://barrigareact.wcaquino.me/, acessado em setembro de 2024)

3.1 JavaScript com Cypress

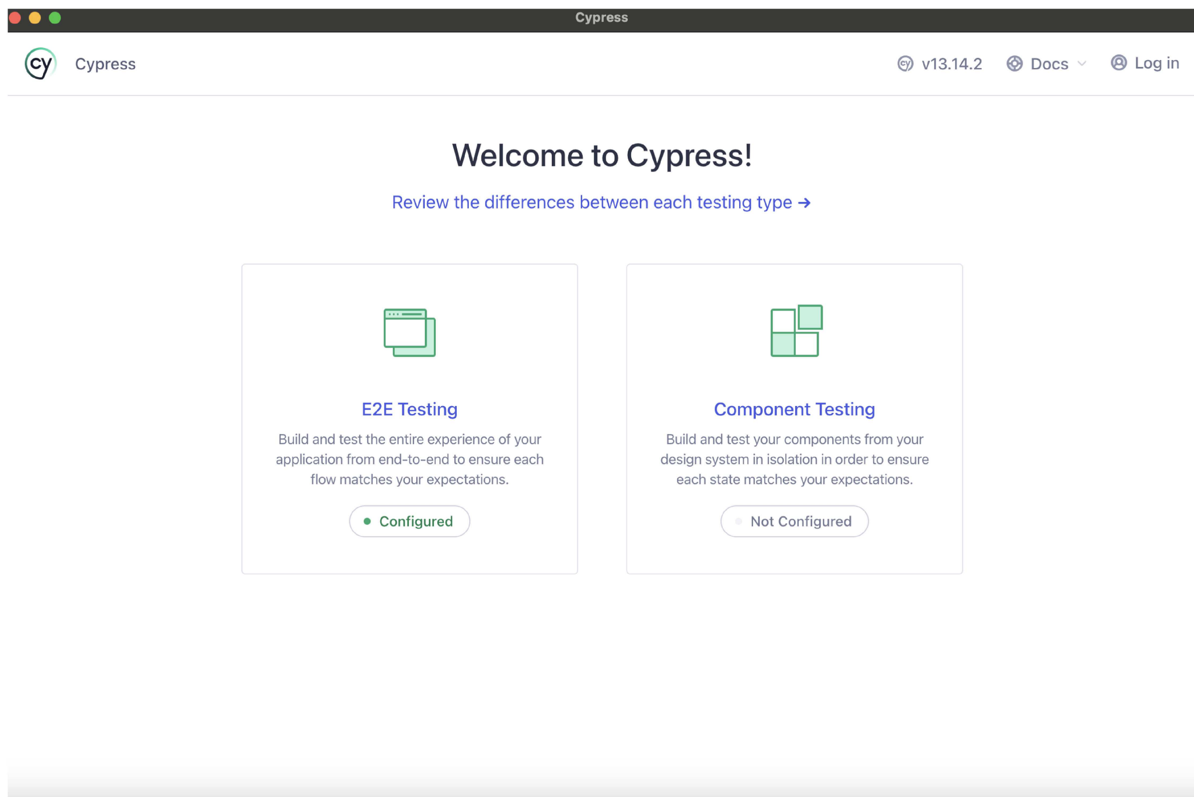
JavaScript foi criado em 1995 por Brendan Eich, enquanto trabalhava na Netscape. Desde então, evoluiu significativamente e agora é padronizado sob o nome ECMAScript (EICH, 1995). A linguagem é amplamente utilizada para criar interatividade em páginas da web. De acordo com Eich (1995), "JavaScript é uma das três tecnologias fundamentais da web, junto com HTML e CSS, e é suportada por todos os navegadores modernos"(EICH, 1995). Originalmente desenvolvida como uma linguagem de *script* do lado do cliente, ela se expandiu para também ser usada no lado do servidor, com o Node.js, por exemplo.

O Cypress é um *framework* de testes de ponta a ponta (*end-to-end*) baseado em JavaScript, desenvolvido para simplificar o processo de teste de aplicações web modernas. Construído sobre o Mocha, um *framework* de testes JavaScript, o Cypress é projetado para funcionar diretamente no navegador, o que permite uma execução de testes mais rápida e confiável (CYPRESS.IO, 2024). Para configurar o ambiente e utilizar o Cypress com JavaScript, siga estes passos:

1. Instale o Node.js e o NPM;
2. Instale um editor de texto, como o VSCode (Visual Studio Code);
3. Instale o Cypress.

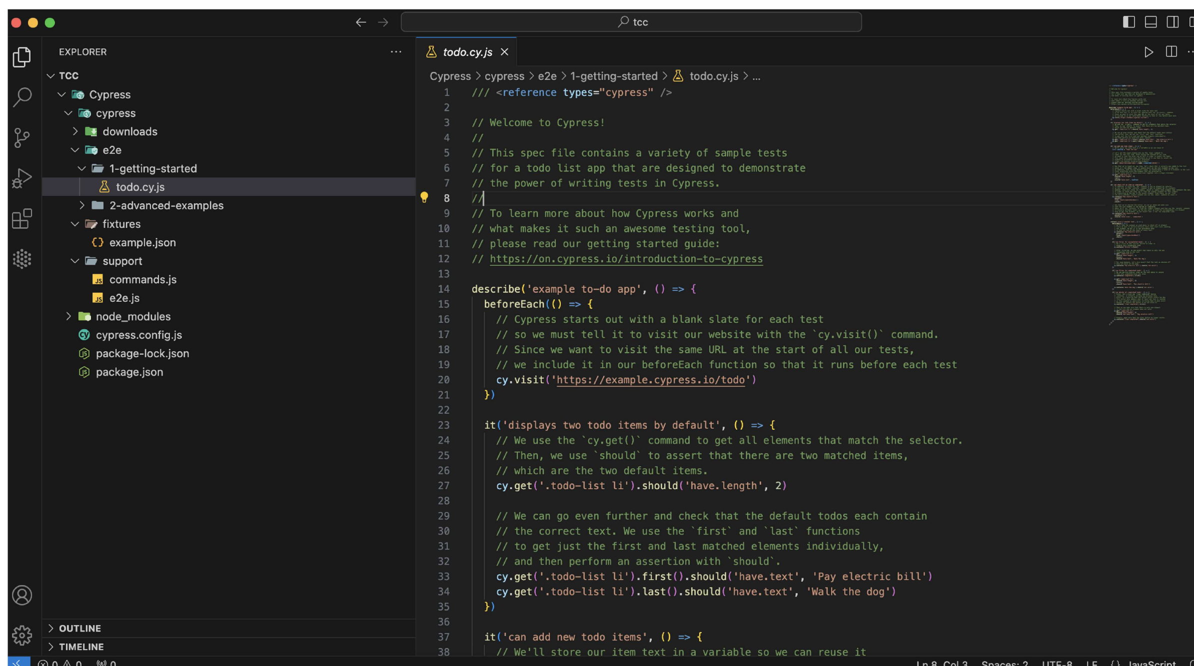
Quando o Cypress é iniciado pela primeira vez, é exibida uma tela de boas-vindas, como pode ser visto na Figura 2. Nessa tela, é possível realizar as configurações iniciais padrão. Durante essa configuração, o Cypress cria uma estrutura de pastas e fornece alguns testes de exemplo para facilitar o início com o *framework*, conforme mostrado na Figura 3. Na Figura 4 , é possível verificar os exemplos de testes criados por meio da execução no modo visual no navegador.

Figura 2 – Tela de Boas Vindas do Cypress.



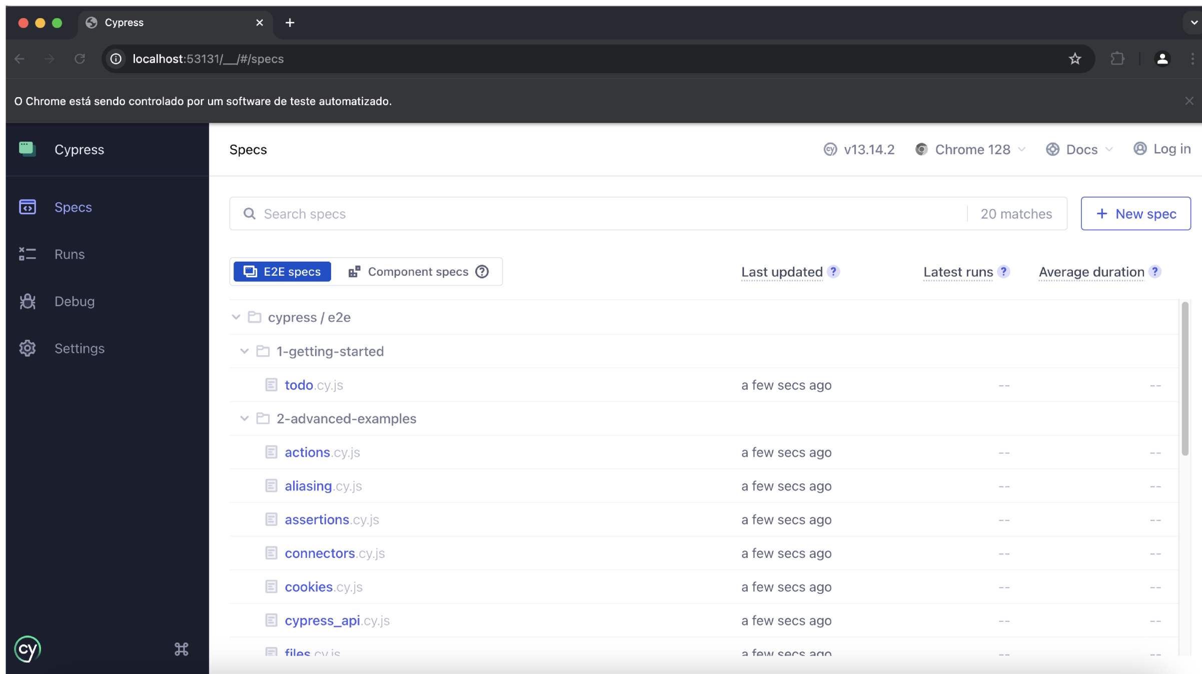
Fonte: Autor.

Figura 3 – Estrutura de Pastas do Cypress



Fonte: Autor.

Figura 4 – Testes de Exemplo do Cypress

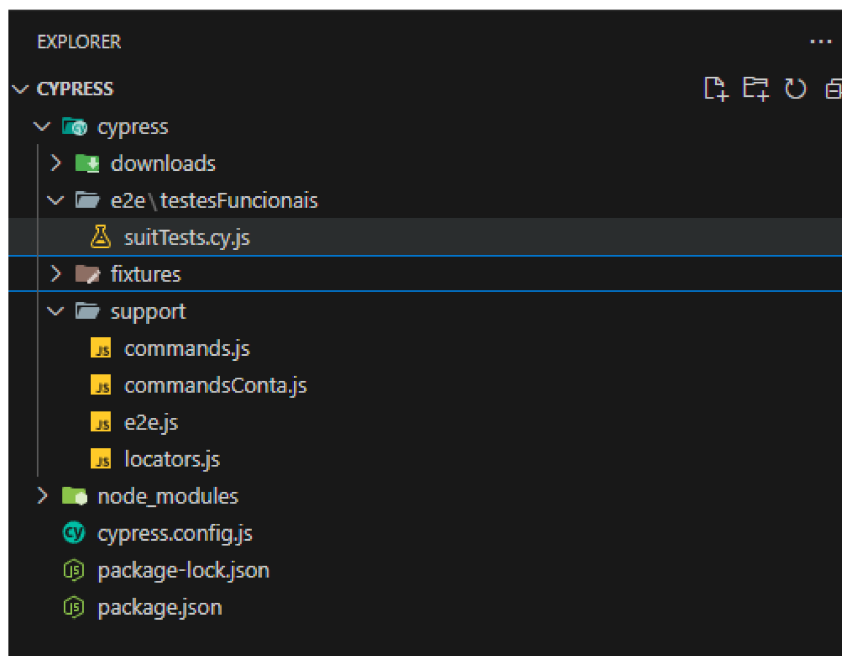


Fonte: Autor.

3.1.1 Codificação dos cenários de testes

A estrutura do projeto segue o padrão fornecido pelo Cypress, onde as pastas mais importantes para o funcionamento são `e2e` e `support`. Na pasta `e2e`, há uma subpasta chamada `testesFuncionais`, que contém todos os cenários de teste da aplicação, organizados no arquivo `suitTest.cy.js`, conforme mostrado na Figura 5.

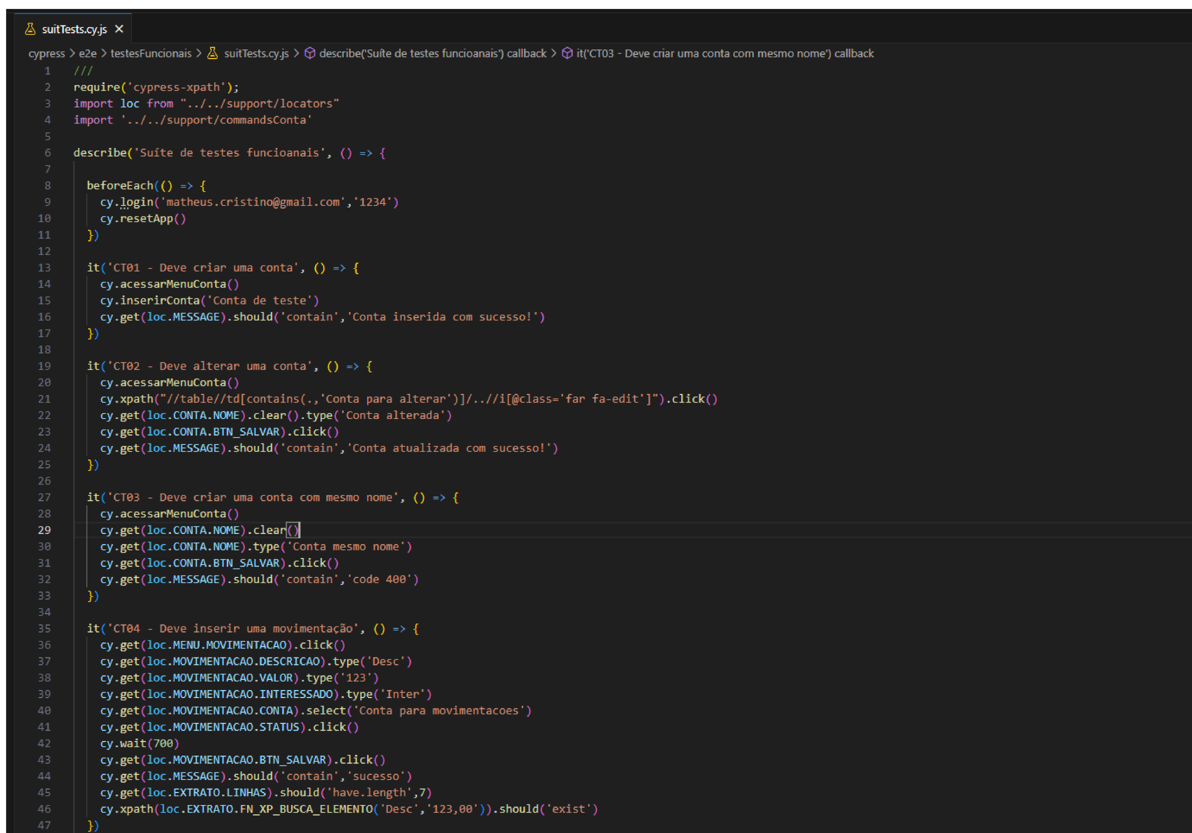
Figura 5 – Estrutura do projeto Cypress



Fonte: Autor

Na Figura 6, é mostrada a codificação de todos os cenários de teste, onde a palavra reservada do Cypress `describe` define o nome da suíte de testes, e `it` define os cenários que pertencem a esta suíte de testes.

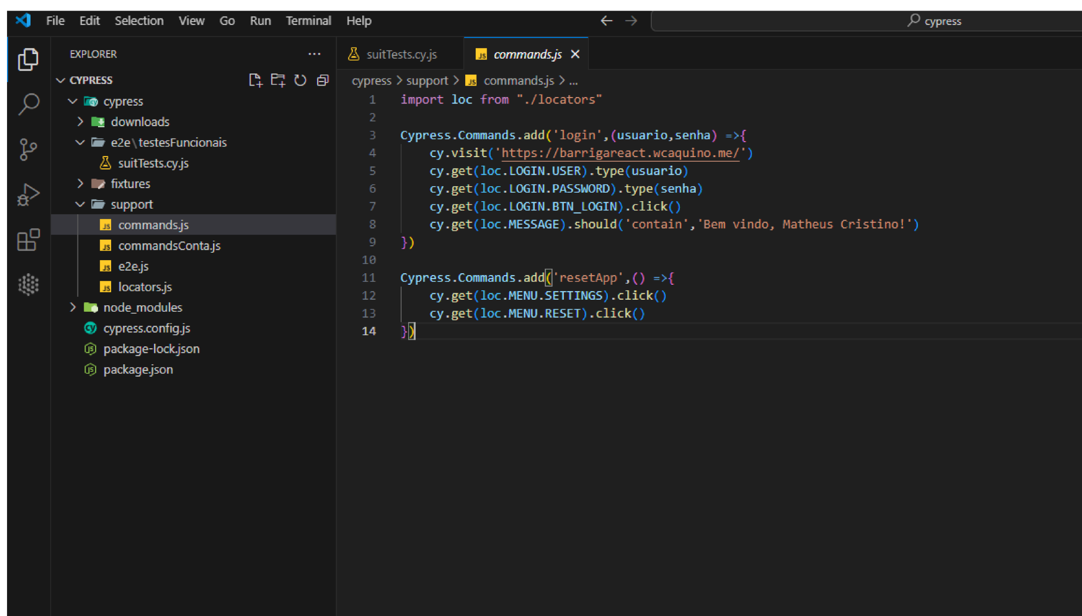
Figura 6 – Suit de testes Cypress



Fonte: Autor

Na pasta `support`, estão os arquivos `commands.js` e `commandsConta.js`, que têm a função de criar comandos personalizados, conforme mostrado na Figura 7. Esses comandos são funções reutilizáveis que podem estender o conjunto de comandos nativos do Cypress, permitindo simplificar os testes ao agrupar interações comuns e evitar a duplicação de código. Além disso, esses comandos ficam disponíveis globalmente em todos os testes.

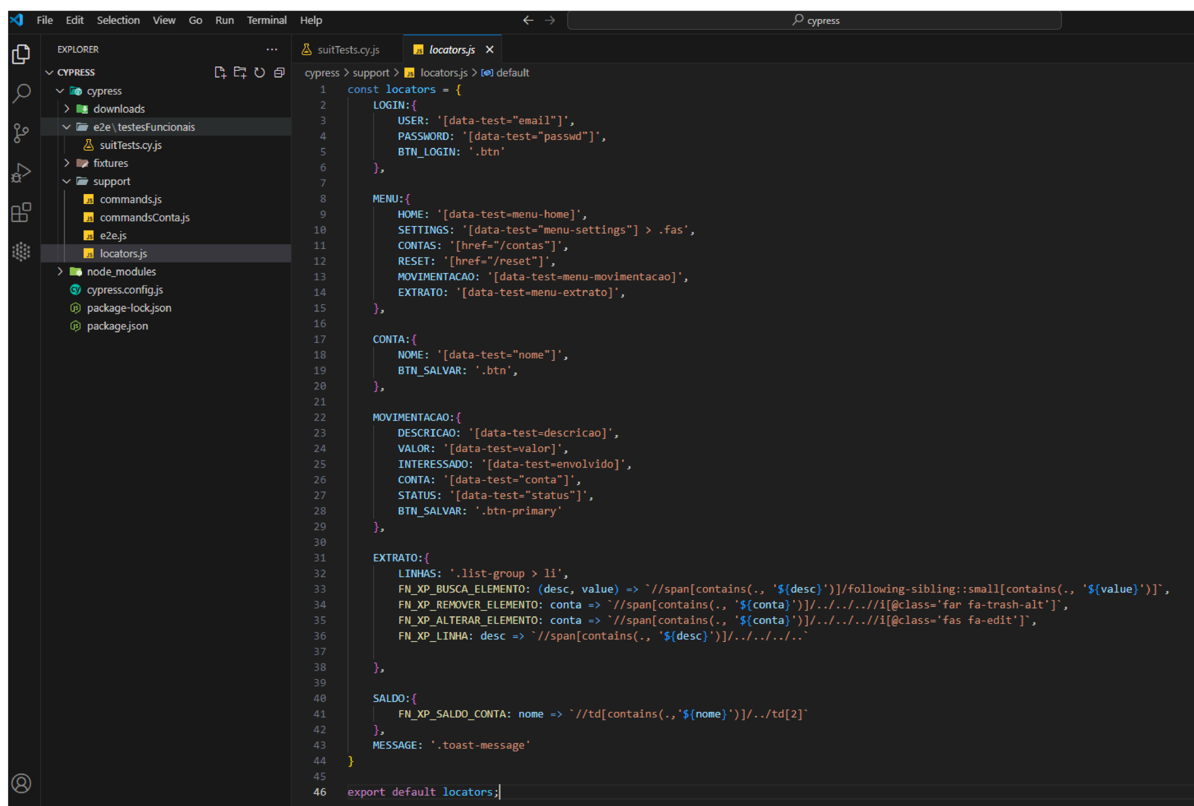
Figura 7 – Commands Cypress



Fonte: Autor

Por fim, há o arquivo `locators.js`, onde todos os elementos utilizados nos testes estão mapeados e organizados de acordo com suas respectivas páginas, seguindo um padrão similar ao Page Objects, conforme mostrado na Figura 8. Esse padrão organiza o código, separando a lógica de interação da interface dos testes, tornando-o mais modular e fácil de manter. No padrão Page Objects, cada página da aplicação é representada por uma classe ou objeto que contém os elementos e métodos de interação específicos dessa página. No entanto, no Cypress, esse padrão é utilizado de forma um pouco diferente, em que os locators são colocados em arquivos `.js` e as páginas são divididas em objetos. Como destacado, "o Page Object Model refere-se ao uso de objetos/classes para representar todos os localizadores e funções relacionadas a uma determinada página em uma aplicação web"(NAYANAJITH, 2022).

Figura 8 – Locators Cypress

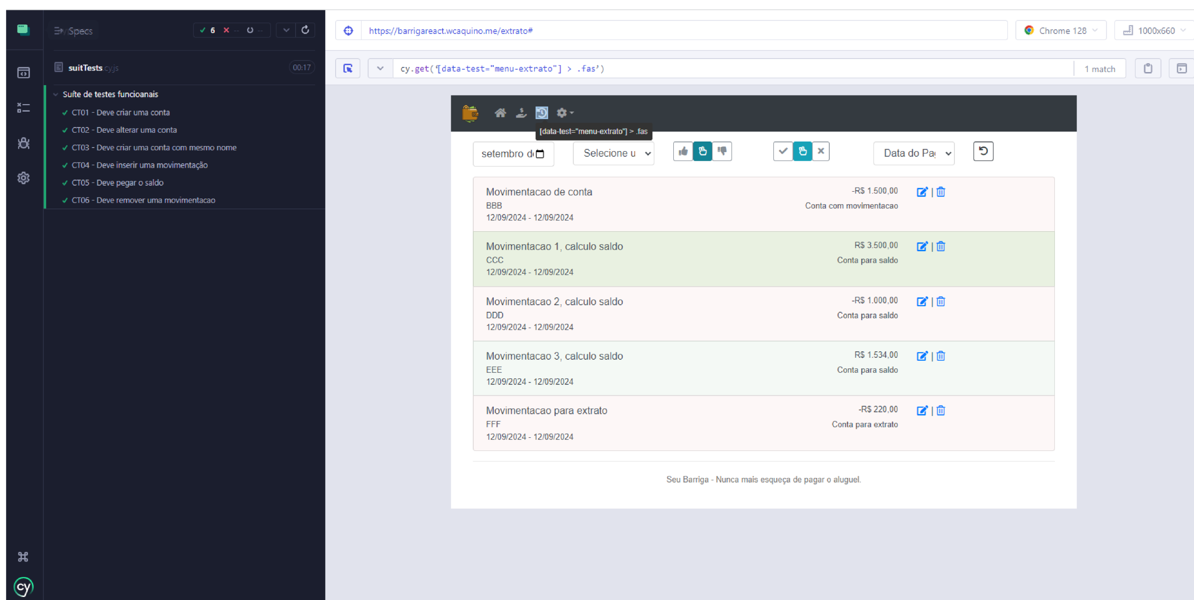


Fonte: Autor

Para execução dos testes no Cypress existem duas maneiras principais: no modo visual (com navegador aberto) e no modo headless (com navegador fechado). Cada um tem suas vantagens dependendo do seu objetivo.

No modo visual, também chamado de modo interativo, é útil para desenvolver e depurar testes. Nele, você pode ver os testes sendo executados em tempo real dentro do navegador, o que permite inspecionar cada etapa e verificar os elementos da página. Para executar, basta estar na raiz do projeto e rodar o comando `npx cypress open`. Ao executar esse comando, o Cypress abre sua interface visual (Test Runner), onde é possível selecionar o navegador e o arquivo de teste para execução, conforme pode-se notar na Figura 9 após a realização de uma execução neste modo.

Figura 9 – Execução da Suit de testes no modo visual Cypress



Fonte: Autor

Outra maneira de executar os testes no Cypress é por meio do modo headless, utilizado principalmente em pipelines de CI/CD ou quando não é necessário acompanhar a execução visualmente. Nesse modo, o Cypress executa os testes sem abrir uma janela do navegador, conforme mostrado na Figura 10. Para rodar o Cypress nesse modo, basta executar o comando `npx cypress run` na raiz do projeto. Por padrão, esse comando executa os testes no modo headless usando o navegador Electron (integrado ao Cypress). No entanto, é possível especificar outro navegador, como o Chrome, utilizando o comando `npx cypress run -browser chrome`.

Figura 10 – Execução modo headless Cypress

```
Running:  suitTests.cy.js                                                    (1 of 1)


Suíte de testes funcioanais
  √ CT01 - Deve criar uma conta (4907ms)
  √ CT02 - Deve alterar uma conta (2630ms)
  √ CT03 - Deve criar uma conta com mesmo nome (2666ms)
  √ CT04 - Deve inserir uma movimentação (3011ms)
  √ CT05 - Deve pegar o saldo (2040ms)
  √ CT06 - Deve remover uma movimentacao (2067ms)


6 passing (20s)


(Results)

  Tests:        6
  Passing:      6
  Failing:      0
  Pending:      0
  Skipped:      0
  Screenshots:  0
  Video:        false
  Duration:     20 seconds
  Spec Ran:     suitTests.cy.js


====================================================================================================

(Run Finished)


     Spec                                      Tests  Passing  Failing  Pending  Skipped
  √  suitTests.cy.js                  00:20        6        6        -        -        -

  √  All specs passed!                00:20        6        6        -        -        -
```

Fonte: Autor

Ao executar os testes no modo visual, o tempo médio de execução obtido foi de 16 segundos e o desvio padrão 0,92, considerando 10 execuções. Já ao executar os testes no modo headless, o tempo médio de execução foi de 21 segundos e o desvio padrão 1,18, também levando em conta 10 execuções, conforme pode ser visto na Figura 11.

Figura 11 – Resultado tempo médio das execuções no Cypress

	A	B	C	D	E
1		Modo Visual/Tempo em segundos			Modo Headless/Tempo em segundos
2		17			20
3		17			22
4		16			21
5		14			21
6		15			20
7		15			20
8		16			21
9		15			21
10		15			20
11		16			24
12	Total:	16		Total:	21
13	Desvio Padrão:	0,92		Desvio Padrão:	1,18

Fonte: Autor

3.2 Java com Selenium

Java é uma linguagem de programação amplamente utilizada e uma plataforma de desenvolvimento que foi criada pela Sun Microsystems em 1995, e atualmente é mantida pela Oracle (QUALIDADE. . . , 2024). Java é reconhecida como uma linguagem de propósito geral e orientada a objetos, projetada para minimizar dependências de implementação, o que a torna popular para o desenvolvimento de aplicações robustas e seguras.

Selenium é uma ferramenta amplamente utilizada para automação de testes de aplicações web (**??**). A ferramenta permite a automação de testes em aplicações web, simulando ações de usuários, como clicar em botões e preencher formulários, e é amplamente utilizada para testes de interface de usuário e aceitação.

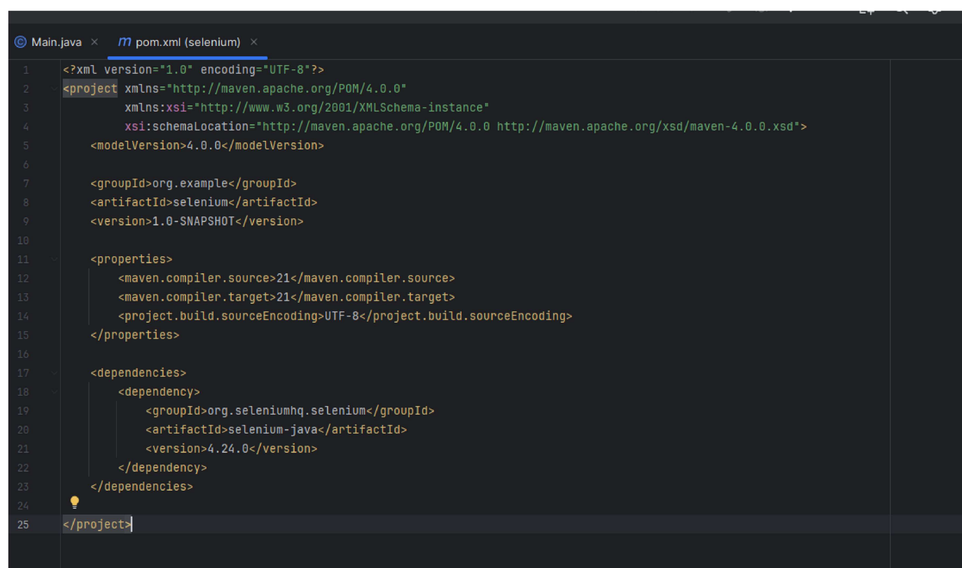
Para configurar o ambiente e utilizar o Selenium com Java, é necessário instalar o JDK, uma IDE, o Selenium e o `Chrome Driver`. Os passos para essa configuração são:

1. Instalar o *JDK - Java Development Kit*
2. Instalar uma IDE - Ambiente de desenvolvimento integrado
3. Instalação do Selenium
4. Instalação do `Chrome Driver`

O *JDK (Java Development Kit)* é necessário para criar e executar projetos na linguagem *Java*. Além disso, é preciso utilizar uma IDE (Ambiente de Desenvolvimento Integrado); neste projeto, foi utilizada o IntelliJ, da empresa JetBrains. O projeto foi criado utilizando o *Maven*, uma ferramenta de gerenciamento de dependências e automação de *build* para projetos *Java*. O *Maven* também auxilia no gerenciamento do ciclo de vida do desenvolvimento de software, incluindo etapas como compilação, teste, empacotamento e implantação.

Com o projeto *Maven* criado, basta adicionar a dependência do Selenium no arquivo `pom.xml` para começar a codificar o projeto, conforme pode ser visto na Figura 12.

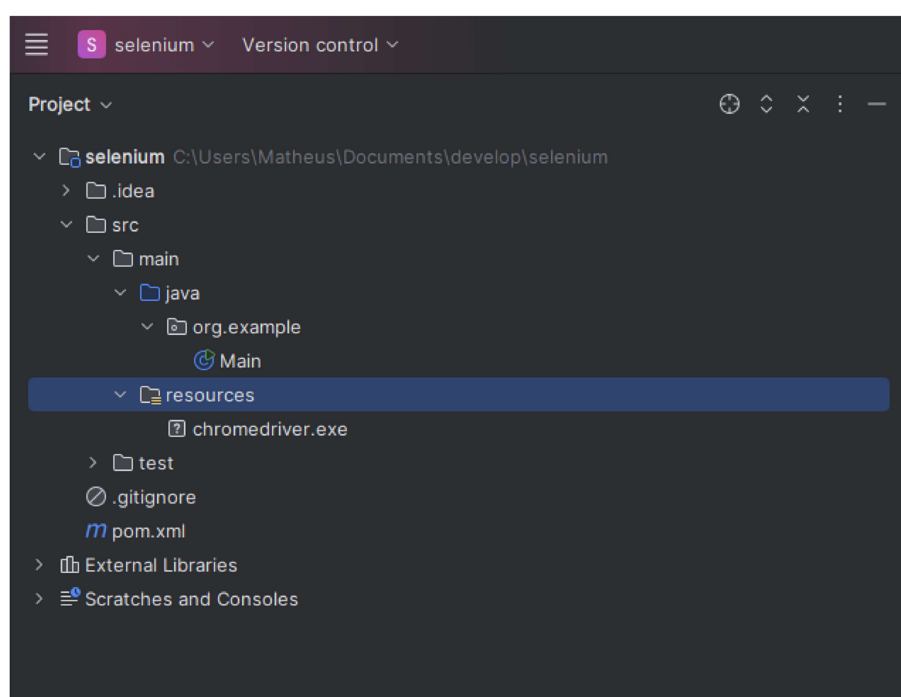
Figura 12 – Dependências Java



Fonte: Autor

Diferentemente do Cypress, que consegue detectar automaticamente os navegadores instalados na máquina para executar os testes, o Selenium requer o download de um driver específico para o navegador que se deseja utilizar na automação, como o `ChromeDriver`, para rodar os testes. O `ChromeDriver` é o driver responsável por automatizar interações com o Google Chrome em testes automatizados. Para integrá-lo ao Java, basta adicionar o driver na pasta *resources* na raiz do projeto, conforme ilustrado na Figura 13.

Figura 13 – ChromeDriver



Fonte: Autor

3.2.1 Codificação dos cenários de testes

Para a estruturação do projeto, foi utilizado o padrão de projeto Page Objects. O padrão Page Objects é amplamente utilizado em automação de testes para organizar o código de forma modular e reutilizável. Segundo Freeman e Pryce (2009), "o padrão Page Objects separa a lógica de interação da interface dos testes, tornando o código mais fácil de manter e evoluir"(FREEMAN; PRYCE, 2009). Nesse padrão, cada página ou parte significativa da aplicação é representada por uma classe separada no código. Essas classes contêm o mapeamento dos elementos da página, como botões, campos de texto, etc., além dos métodos de ação, que são responsáveis por realizar tarefas como preencher formulários, clicar em botões e navegar entre páginas. De acordo com Meszaros (2007), "a separação entre a lógica de interação e os testes permite que os testes sejam mais legíveis e menos suscetíveis a mudanças na interface do usuário"(MESZAROS, 2007).

No pacote `Tests`, há a classe `TestSuite`, que reúne todos os cenários de teste implementados e utiliza os métodos das classes presentes no pacote `Pages`. Conforme ilustrado na Figura 14, e seguindo o padrão de projeto Page Object, este pacote contém classes separadas para cada página da aplicação. Essas classes são responsáveis pelo mapeamento dos elementos da interface e pelo encapsulamento dos métodos que interagem com esses elementos. Nas próximas imagens, será detalhada a implementação de cada uma dessas classes.

Nas Figuras 15, 16, 17, 18 e 19, é possível visualizar a implementação das classes ContasPage, ExtratoPage, HomePage, LoginPage e MovimentacoesPage.

A classe `ContasPage` encapsula a lógica para interagir com a página de contas através dos métodos `inserirConta(String nomeContaNova)`, que localiza o ícone de edição de uma conta, altera o nome da conta e clica no botão de salvar, e o método `getMessage()`, que retorna a mensagem de `feedback` exibida após a alteração da conta.

A classe `ExtratoPage` encapsula a lógica para interagir com a página de extrato através dos métodos `excluirMovimentacao()`, que localiza o ícone de exclusão de uma movimentação específica e clica nele para removê-la, e o método `getMessage()`, que retorna a mensagem de `feedback` exibida após a exclusão da movimentação.

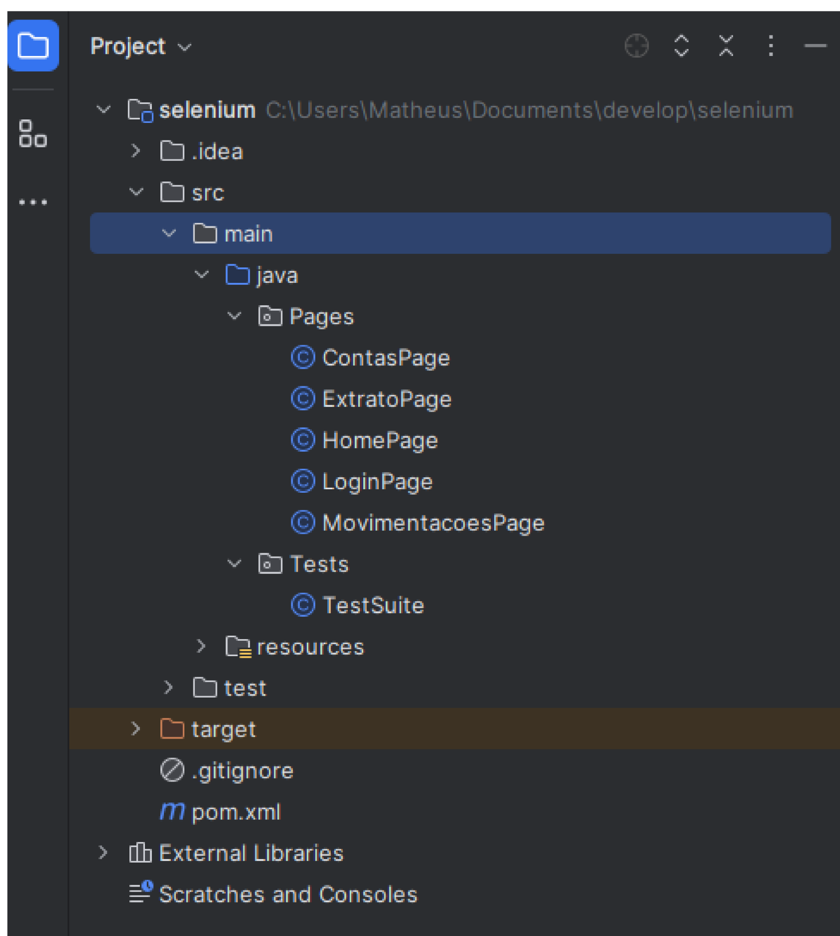
A classe `HomePage` encapsula a lógica para interagir com a página inicial, que possui funcionalidades como acessar configurações, restaurar dados, acessar contas, cadastrar movimentações e visualizar extratos. A interação ocorre através dos métodos `acessarSettings()`, que clica no ícone de configurações, `acessarReset()`, que clica no link de `resetar`, `acessarContas()`, que clica no link de contas, `acessarMovimentacoes()`, que clica no ícone de movimentações financeiras, `acessarExtrato()`, que clica no ícone de extrato, e `getSaldo(String conta)`, que retorna o saldo de uma conta específica, localizando o saldo com base no nome da conta.

A classe `LoginPage` encapsula a lógica para realizar o login na aplicação, através do método `login(String email, String password)`, que preenche os campos de e-mail e senha e clica no botão de login.

A classe `MovimentacoesPage` encapsula a lógica para interagir com a página de movimentações financeiras. Utiliza os métodos `inserirMovimentacao()`, que preenche os campos de descrição, valor, interessado, conta, marca o status e submete a movimentação. O método `getMessage()` retorna a mensagem de `feedback` exibida após a inserção da movimentação. O método `movimentacaoExiste()` verifica se uma movimentação com uma descrição e valor específicos está presente na página.

Na Figura 20, é apresentada a codificação da classe `TestSuite`. Nessa classe, todos os cenários de teste mapeados na aplicação estão separados pela anotação `@Test` do Java. Neste arquivo, todas as classes implementadas no pacote Pages são instanciadas e inicializadas. Através dos métodos implementados nessas classes, é possível realizar as interações na página de acordo com cada cenário e efetuar as validações necessárias.

Figura 14 – Estrutura projeto Selenium com Java



Fonte: Autor

Figura 15 – Página de contas Selenium

```java
package Pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

import java.time.Duration;

public class ContasPage {
    private WebDriver driver;

    private By nomeContaField = By.cssSelector("[data-test='nome']");
    private By salvarButton = By.cssSelector(".btn");
    private By message = By.cssSelector(".toast-message");
    private By btnAlterarConta = By.xpath("//table//td[contains(text(),'Conta para alterar')]/..//i[@class='far fa-edit']");
    public ContasPage(WebDriver driver) {
        this.driver = driver;
    }

    public void inserirConta(String nomeContaNova) {
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
        By editIconLocator = btnAlterarConta;
        WebElement editIcon = wait.until(ExpectedConditions.visibilityOfElementLocated(editIconLocator));
        editIcon.click();
        driver.findElement(nomeContaField).clear();
        driver.findElement(nomeContaField).sendKeys(nomeContaNova);
        driver.findElement(salvarButton).click();
    }

    public String getMessage() {
        return driver.findElement(message).getText();
    }
}
```

Fonte: Autor

Figura 16 – Página de extrato Selenium

```java
package Pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

import java.time.Duration;

public class ExtratoPage {
    private WebDriver driver;
    private By excluir = By.xpath("//span[contains(., 'Movimentacao para exclusao')]/../../..//i[@class='far fa-trash-alt']");
    private By message = By.cssSelector(".toast-message");

    public ExtratoPage(WebDriver driver) {
        this.driver = driver;
    }

    public void excluirMovimentacao(){
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
        WebElement element = wait.until(ExpectedConditions.visibilityOfElementLocated(excluir));
        driver.findElement(excluir).click();
    }

    public String getMessage() {
        return driver.findElement(message).getText();
    }


}
```

Fonte: Autor

Figura 17 – Página de home Selenium

```java
import java.time.Duration;

public class HomePage {  2 usages
    private WebDriver driver;  10 usages

    private By settings = By.xpath( xpathExpression: "//i[@title='settings']\n");  2 usages
    private By resetar = By.xpath( xpathExpression: "//a[text()='Resetar']");  2 usages
    private By contas = By.xpath( xpathExpression: "//a[text()='Contas']\n");  2 usages
    private By movimentacoes = By.xpath( xpathExpression: "//i[@title='Cadastrar movimentação']");  1 usage
    private By extrato = By.xpath( xpathExpression: "//i[@title='extrato']");  1 usage

    public HomePage(WebDriver driver) {  1 usage
        this.driver = driver;
    }

    public void acessarSettigns() {  5 usages
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
        WebElement element = wait.until(ExpectedConditions.visibilityOfElementLocated(settings));
        driver.findElement(settings).click();
    }

    public void acessarReset() {  2 usages
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
        WebElement element = wait.until(ExpectedConditions.visibilityOfElementLocated(resetar));
        driver.findElement(resetar).click();
    }

    public void acessarContas() {  3 usages
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
        WebElement element = wait.until(ExpectedConditions.visibilityOfElementLocated(contas));
        driver.findElement(contas).click();
    }

    public void acessarMovimentacaoes() {  1 usage
        driver.findElement(movimentacoes).click();
    }

    public void acessarExtrato(){  1 usage
        driver.findElement(extrato).click();
    }
```

Fonte: Autor

Figura 18 – Página de login Selenium

```java
package Pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

public class LoginPage {  2 usages
    private WebDriver driver;  4 usages

    private By emailField = By.cssSelector("[data-test='email']");  1 usage
    private By passwordField = By.cssSelector("[data-test='passwd']");  1 usage
    private By loginButton = By.cssSelector(".btn");  1 usage

    public LoginPage(WebDriver driver) {  1 usage
        this.driver = driver;
    }

    public void login(String email, String password) {  1 usage
        driver.findElement(emailField).sendKeys(email);
        driver.findElement(passwordField).sendKeys(password);
        driver.findElement(loginButton).click();
    }
}
```

Fonte: Autor

Figura 19 – Página de movimentações Java

```java
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;

public class MovimentacoesPage {
    private WebDriver driver;

    private By menuMovimentacao = By.cssSelector("[data-test='menu-movimentacao']");
    private By descricaoField = By.cssSelector("[data-test='descricao']");
    private By valorField = By.cssSelector("[data-test='valor']");
    private By interessadoField = By.cssSelector("[data-test='envolvido']");
    private By contaSelect = By.cssSelector("[data-test='conta']");
    private By statusCheckbox = By.cssSelector("[data-test='status']");
    private By salvarButton = By.cssSelector(".btn-primary");
    private By message = By.cssSelector(".toast-message");

    public MovimentacoesPage(WebDriver driver) {
        this.driver = driver;
    }

    public void inserirMovimentacao(String descricao, String valor, String interessado, String conta) {
        driver.findElement(descricaoField).sendKeys(descricao);
        driver.findElement(valorField).sendKeys(valor);
        driver.findElement(interessadoField).sendKeys(interessado);
        driver.findElement(contaSelect).sendKeys(conta);
        driver.findElement(statusCheckbox).click();
        driver.findElement(salvarButton).click();
    }

    public String getMessage() {
        return driver.findElement(message).getText();
    }


    public boolean movimentacaoExiste(String descricao, String valor) {
        String xpath = "//span[contains(., '" + descricao + "')]/following-sibling::small[contains(., '" + valor + "')]";
        WebElement movimentacao = driver.findElement(By.xpath(xpath));

        return movimentacao.isDisplayed();
    }
}
```

Fonte: Autor

Figura 20 – Suíte de testes Selenium

```java
public class TestSuite {
    public void setUpMethod() {
    }
    @Test
    public void CT01_deveCriarConta() throws InterruptedException {
        homePage.acessarSettigns();
        homePage.acessarContas();
        contasPage.inserirConta("Conta de teste");
        contasPage.getMessage().contains("Conta inserida com sucesso!");
    }
    @Test
    public void CT02_deveAlterarConta() {
        homePage.acessarSettigns();
        homePage.acessarContas();
        contasPage.inserirConta("Conta alterada");
        contasPage.getMessage().contains("Conta atualizada com sucesso!");
    }
    @Test
    public void CT03_deveCriarContaComMesmoNome() {
        homePage.acessarSettigns();
        homePage.acessarContas();
        contasPage.inserirConta("Conta mesmo nome");
        contasPage.getMessage().contains("code 400");
    }
    @Test
    public void CT04_deveInserirMovimentacao() {
        homePage.acessarMovimentacaoes();
        movimentacoesPage.inserirMovimentacao("Desc", "123", "Inter", "Conta para movimentacoes");
        Assert.assertTrue(movimentacoesPage.getMessage().contains("sucesso"));
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
        wait.until(ExpectedConditions.presenceOfAllElementsLocatedBy(By.cssSelector(".list-group .list-group-item")));
        List<WebElement> rows = driver.findElements(By.cssSelector(".list-group .list-group-item"));
        Assert.assertEquals(rows.size(), 7, "O número de linhas na tabela não é o esperado.");
        Assert.assertTrue(movimentacoesPage.movimentacaoExiste("Desc", "123,00"));
    }
    @Test
    public void CT05_devePegarSaldo() {
        Assert.assertEquals(homePage.getSaldo("Conta para saldo"), "R$ 534,00");
    }
    @Test
    public void CT06_deveRemoverMovimentacao() {
        homePage.acessarExtrato();
```

Fonte: Autor

Para a execução dos testes no Selenium, existem duas maneiras principais: o modo visual e o modo headless. No modo visual, o Selenium utiliza o driver (como o `ChromeDriver`) para abrir o navegador localmente e executar os testes, permitindo que você veja os testes sendo executados em tempo real dentro do navegador.

No modo headless, os testes são executados com o navegador fechado, sem interface gráfica. Isso é útil para executar testes em ambientes onde a interface gráfica não está disponível ou para acelerar a execução.

Assim como no Cypress, o modo visual permite desenvolver e depurar os testes, visualizar a execução em tempo real, inspecionar cada etapa e verificar os elementos da página. Para executar dessa forma, basta especificar o caminho do driver no código e usar o comando `driver.get` para abrir o navegador, conforme ilustrado na Figura 21.

Figura 21 – Execução modo visual Selenium

```
24
25    @BeforeClass
26    public void setup() {
27        System.setProperty("webdriver.chrome.driver", "src/main/resources/chromedriver");
28
29        ChromeOptions options = new ChromeOptions();
30        options.addArguments("--window-size=1920,1080");
31
32        driver = new ChromeDriver(options);
33        driver.manage().window().maximize();
34        driver.get("https://barrigareact.wcaquino.me/");
35
36        loginPage = new LoginPage(driver);
37        contasPage = new ContasPage(driver);
```

Fonte: Autor

Para rodar no modo sem abrir o navegador no Selenium, basta instanciar o `ChromeOptions` antes de iniciar os testes e adicionar o argumento `-headless`, conforme ilustrado na Figura 22.

Figura 22 – Execução modo Headless Selenium

```
@BeforeClass
public void setup() {
    System.setProperty("webdriver.chrome.driver", "src/main/resources/chromedriver");

    ChromeOptions options = new ChromeOptions();
    options.addArguments("--headless");
    options.addArguments("--window-size=1920,1080");

    driver = new ChromeDriver(options);
    driver.manage().window().maximize();
    driver.get("https://barrigareact.wcaquino.me/");

    loginPage = new LoginPage(driver);
```

Fonte: Autor

Executando os testes no modo visual, o tempo médio de execução foi de 12,344 segundose o desvio padrão foi de 0,17, considerando 10 execuções. No modo headless, o tempo médio de execução foi de 12,266 segundos e o desvio padrão 0,18, também considerando 10 execuções, conforme mostrado na Figura 23.

Figura 23 – Resultados das execuções no Selenium

	A	B	C	D	E
1		Modo Visual/Tempo em segundos			Modo Headless/Tempo em segundos
2		12,297			12,376
3		12,536			12,566
4		12,269			12,418
5		12,448			12,282
6		12,390			12,249
7		12,740			12,225
8		12,205			11,882
9		12,442			12,464
10		12,280			12,201
11		12,130			12,210
12	Total:	12,344		Total:	12,266
13	Desvio Padrão:	0,17		Desvio Padrão:	0,18

Fonte: Autor

3.3 Javascript com Playwright

Assim como o Cypress, o Playwright também utiliza JavaScript e Node.js para o desenvolvimento de testes, portanto as configurações iniciais são bastante semelhantes. O Playwright é uma poderosa ferramenta de automação de testes de navegador criada pela Microsoft. Ele é projetado para realizar testes *end-to-end (E2E)* e é capaz de automatizar várias tarefas em aplicativos web, incluindo navegação, interação com elementos e verificação de resultados em diferentes navegadores e plataformas (MICROSOFT, 2024).

Para configurar o ambiente e utilizar o Playwright com JavaScript, siga os seguintes passos:

1. Instalar o Node.js e o npm.
2. Instalar um editor de texto; neste caso foi utilizado o VSCode (Visual Studio Code).
3. Instalar o Playwright.

O Node.js é necessário porque o Playwright é uma biblioteca JavaScript que precisa de um ambiente para ser executado fora do navegador. O npm, por sua vez, facilita a instalação e o gerenciamento de pacotes. A instalação do Playwright é bem simples; com o Node.js e npm instalados, basta criar uma pasta para o projeto e digitar os comandos:

1. `npm init -y`, para iniciar um projeto Node.js com as configurações iniciais.

2. `npm install playwright`.
3. `npx playwright install`, para instalar os navegadores que o *Playwright* utiliza, caso não estejam instalados na máquina.
4. `node "arquivo de teste"` para executar os testes.

Para a estruturação do projeto, foi utilizado o padrão de projeto `Page Objects`, com uma estrutura similar à utilizada no Selenium. Na pasta `Page`, estão os arquivos correspondentes às páginas e suas implementações. Na raiz do projeto, encontram-se o arquivo de configuração do Playwright, o arquivo `package.json`, que contém as dependências, e o `suiTest.spec.js`, que contém os testes implementados, conforme mostrado na Figura 24.

Na Figura 25, é apresentada a codificação da classe `ContaPage`, que encapsula a lógica para interagir com a página de contas. Essa página contém um formulário de criação de contas e um `dropdown` para seleção de contas. A interação é realizada por meio dos métodos `createAccount(name)`, que preenche o campo de nome e submete o formulário, e `selectAccount(accountName)`, que seleciona uma conta no `dropdown`.

Na Figura 26, é apresentada a codificação da classe `HomePage`, que é usada para interagir com a página `Home`. Ela encapsula a lógica para manipular elementos da página relacionados às ações de restaurar dados da aplicação, acessar contas, abrir movimentações e extratos, além de capturar mensagens de notificação `(toast messages)`. Os métodos incluem: `resetData()`, que clica no botão de `reset` e no link de `resetar`; `goToAccounts()`, que clica no botão de `reset` e no link de contas; `openMovimentacao()`, que clica no botão de movimentação financeira; `getToastMessage()`, que retorna o elemento que contém a mensagem de notificação exibida no momento atual; e `openExtrato()`, que clica no botão de extrato.

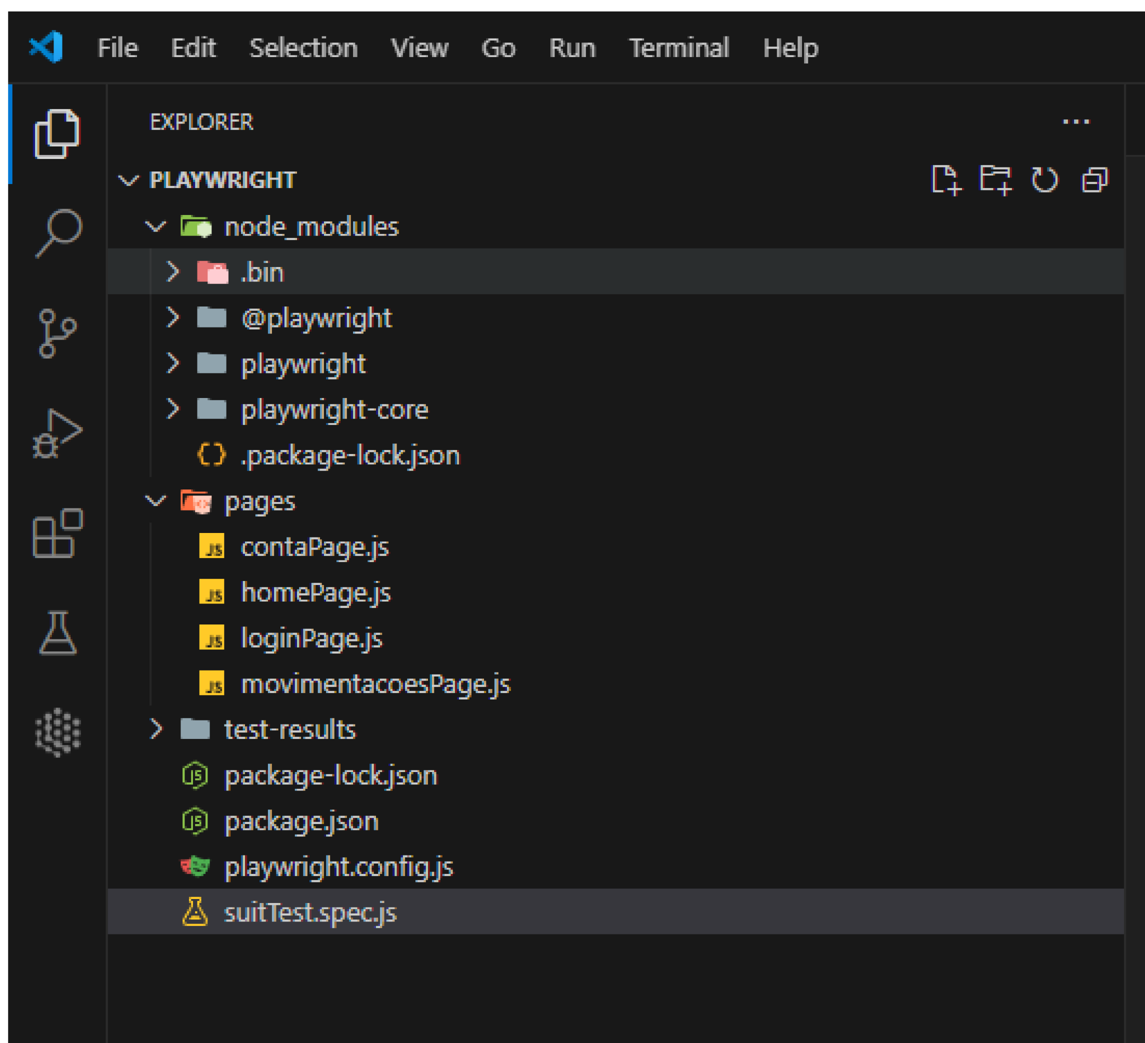
Na Figura 27, é apresentada a codificação da classe `LoginPage`, que encapsula a lógica para interagir com a página de login. O método `login(email, password)` recebe os valores de e-mail e senha, preenche-os na aplicação e clica no botão de login.

Na Figura 28, é apresentada a codificação da classe `MovimentacoesPage`, que encapsula a lógica para interagir com a página de movimentações financeiras. Ela possui o método `createMovimentacao(descricao, valor, envolvido, conta)`, que preenche os campos de descrição, valor, envolvido, seleciona a conta e o status, e clica no botão de salvar para criar uma movimentação.

Todas essas classes são exportadas para serem utilizadas no arquivo de cenários de teste `suitTest.spec.js`, juntamente com a codificação dos cenários de teste, conforme ilustrado na Figura 29. Neste arquivo, encontram-se os cenários de testes automatizados

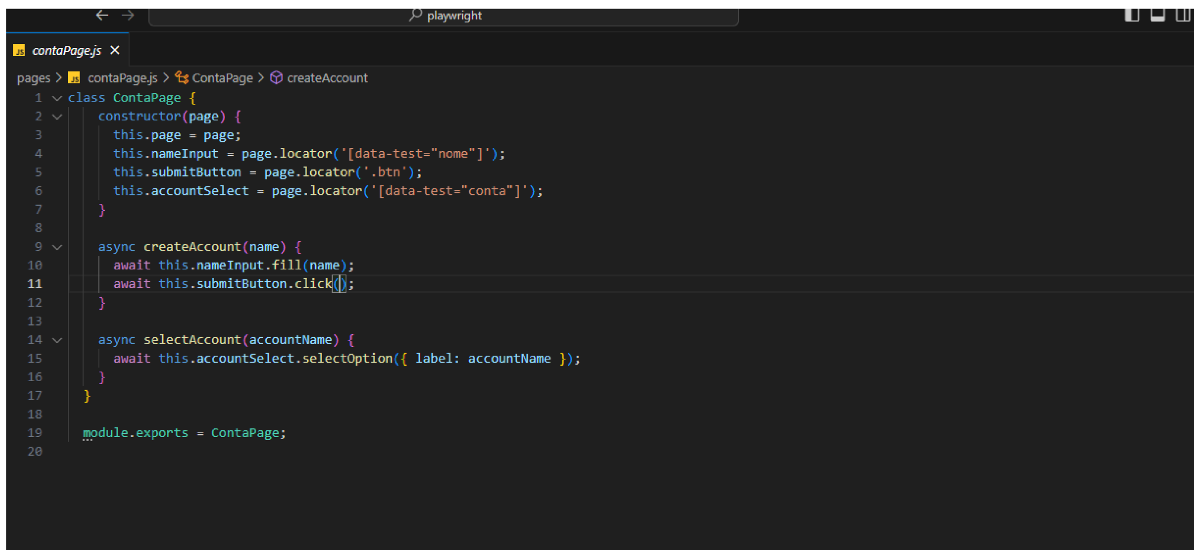
que verificam as funcionalidades de criação, alteração e exclusão de contas e movimentações financeiras. Os cenários de teste são identificados pela palavra reservada *Test*.

Figura 24 – Estrutura do projeto Playwright



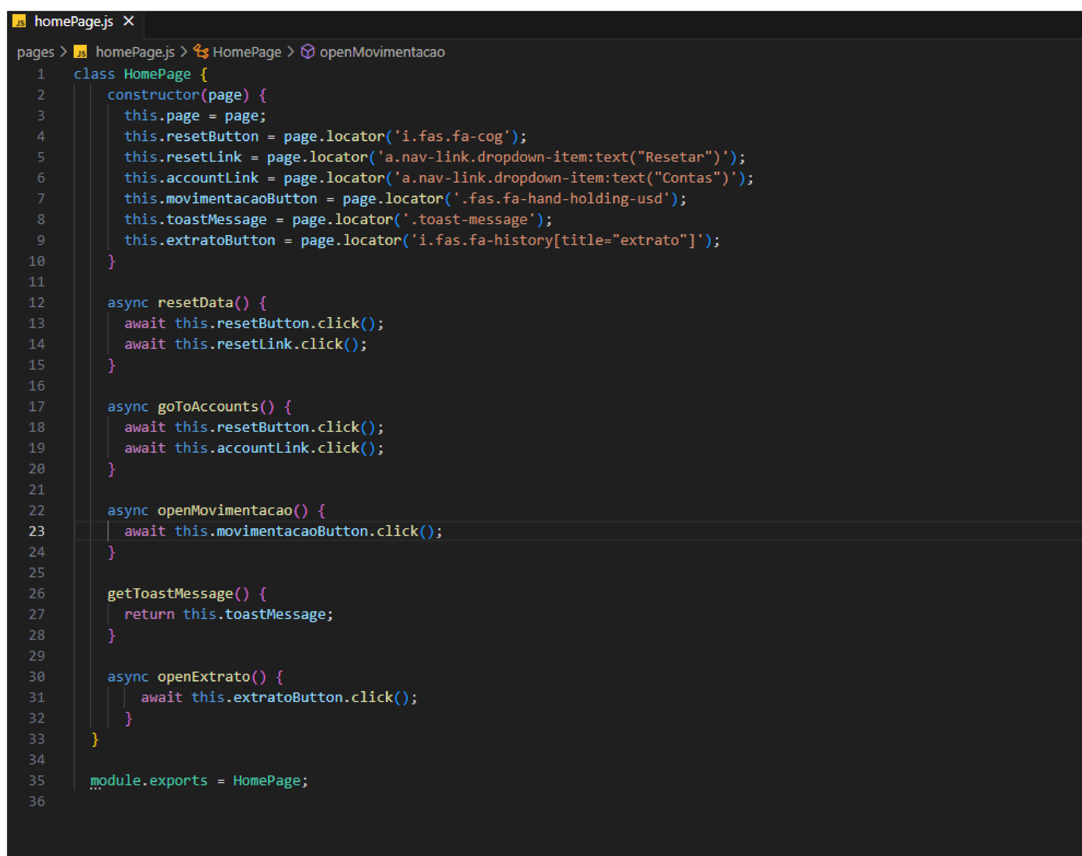
Fonte: Autor

Figura 25 – Conta Page Playwright



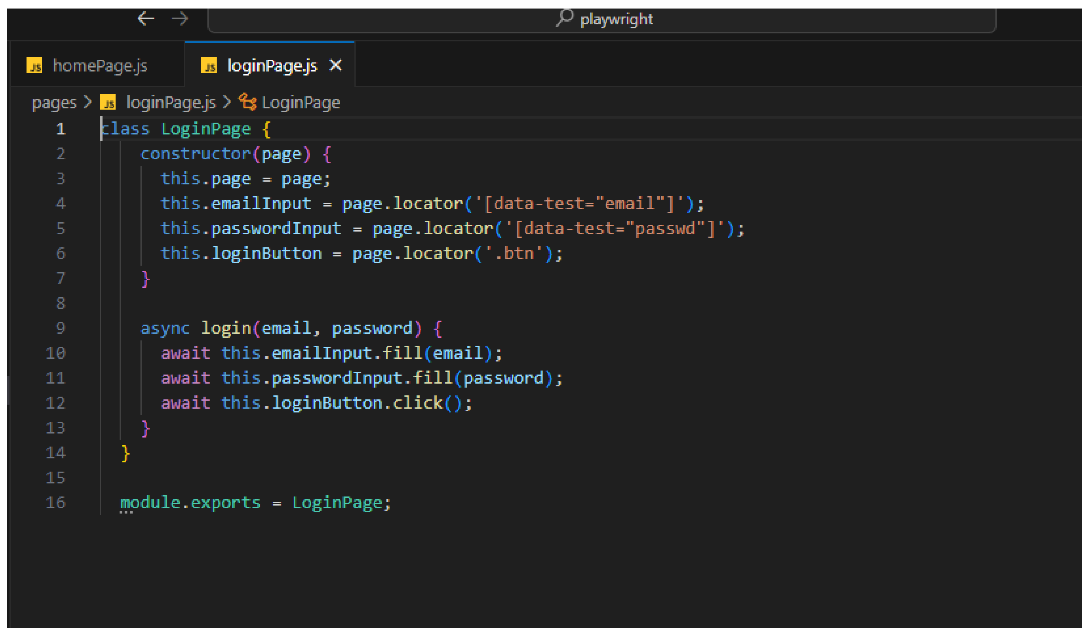
Fonte: Autor

Figura 26 – Home Page Playwright



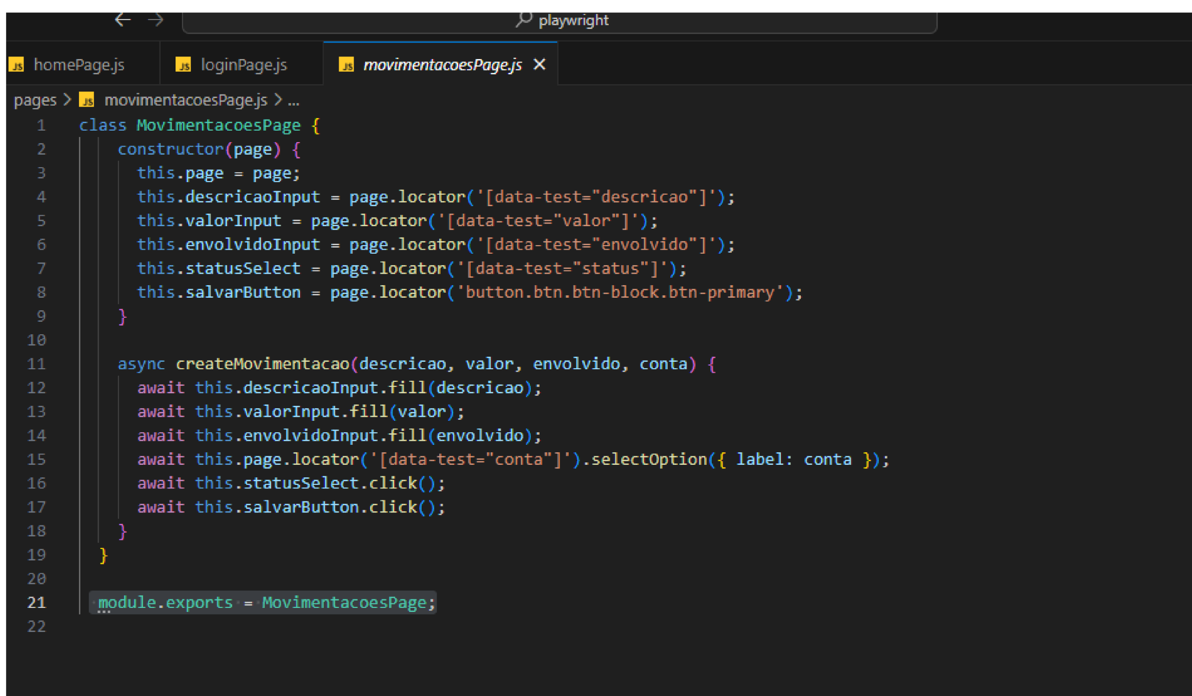
Fonte: Autor

Figura 27 – Login Page Playwright



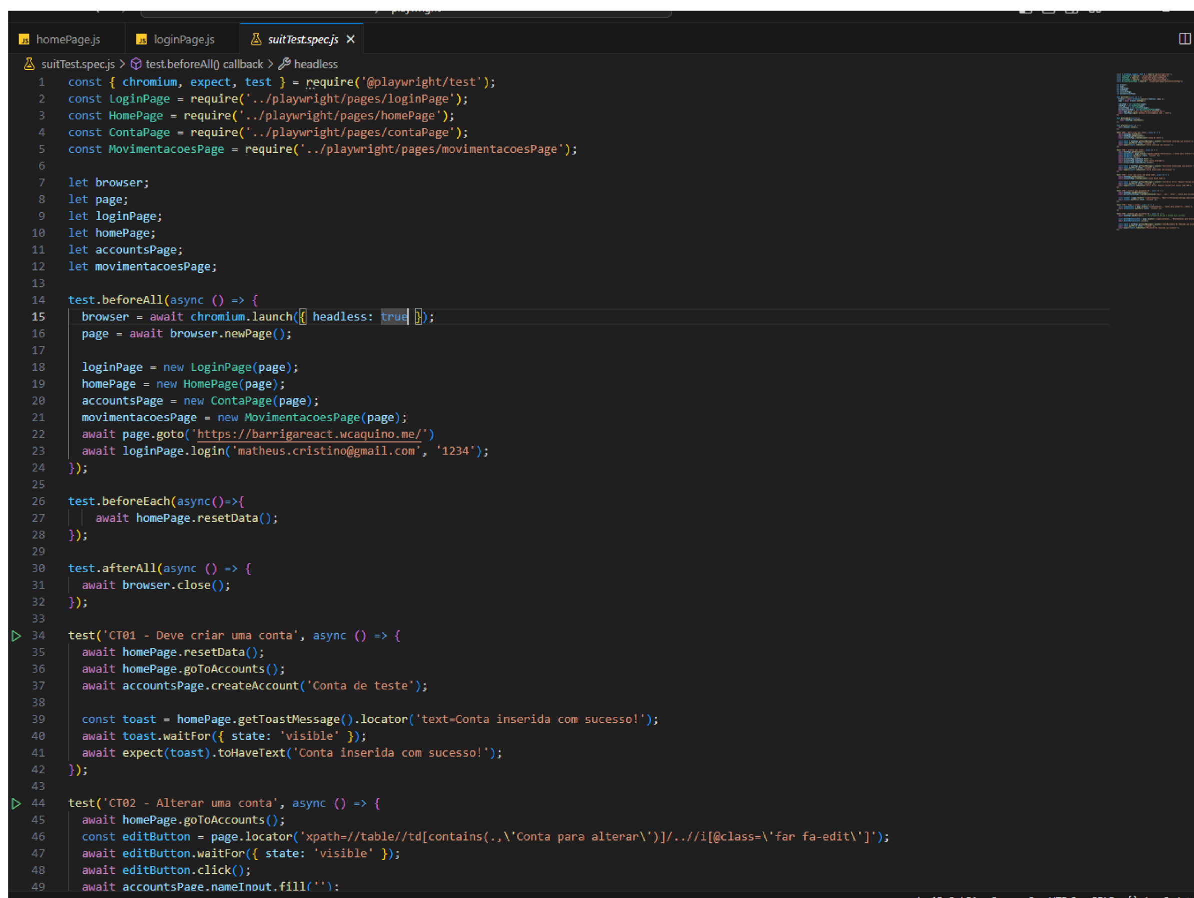
Fonte: Autor

Figura 28 – Movimentações Page Playwright



Fonte: Autor

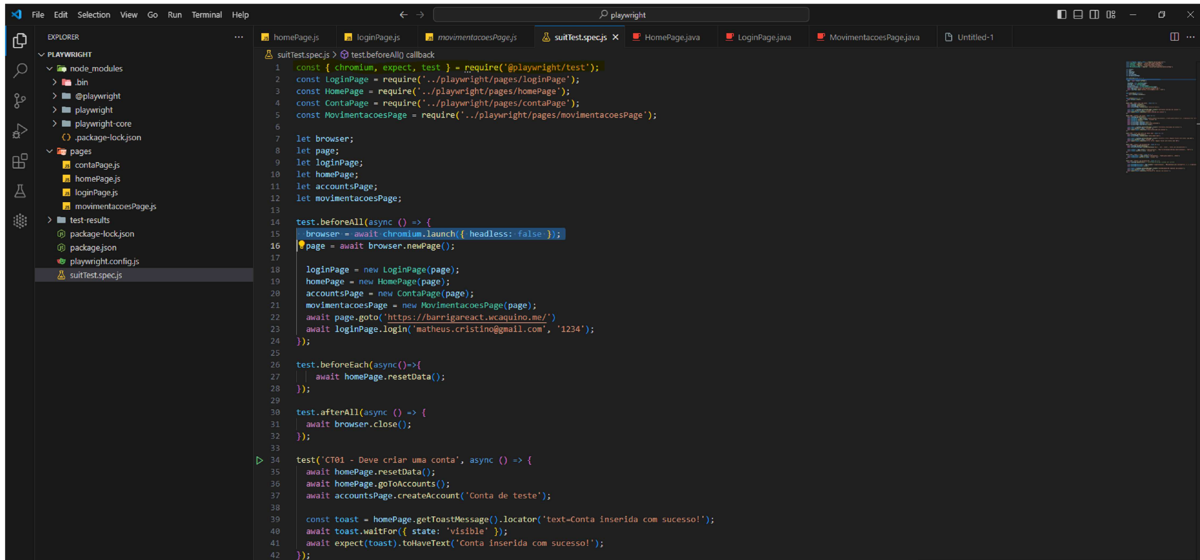
Figura 29 – Suíte de testes Playwrightt



Fonte: Autor

Para a execução dos testes no Playwright, existem duas maneiras principais: o modo visual e o modo headless. Por padrão, o Playwright está configurado para executar testes no modo headless. Para rodar os testes no modo visual, é necessário alterar a propriedade headless para false no código. Quanto aos navegadores, o Playwright suporta a execução de testes nos principais navegadores por padrão: Chromium, Firefox e WebKit (baseado no Safari). Para fins de comparação, o código foi configurado para rodar apenas no Chromium, assim como nos outros frameworks, conforme ilustrado na Figura 30.

Figura 30 – Configuração modo visual e navegador no Playwright



Fonte: Autor

Executando os testes no modo visual, o tempo médio de execução foi de 15,7 segundos e o desvio padrão de 0,68, considerando 10 execuções. No modo headless, o tempo médio de execução foi de 14,8 segundos e o desvio padrão 0,90, também considerando 10 execuções, conforme mostrado na Figura 31.

Figura 31 – Resultado execuções no Playwright

	A	B	C	D	E
1		Modo Visual/Tempo em segundos			Modo Headless/Tempo em segundos
2		16,4			17,7
3		15,7			14,9
4		15,5			14,8
5		15,9			14,5
6		15,5			14,6
7		16,5			14,6
8		15,6			15,0
9		15,4			14,7
10		17,7			15,4
11		15,5			14,8
12	Total:	15,7		Total:	14,8
13	Desvio Padrão:	0,68		Desvio Pa	0,90

Fonte: Autor

4 Resultados e Discussões

Neste capítulo, serão apresentados e discutidos os resultados obtidos a partir da análise comparativa das ferramentas de teste E2E Cypress, Selenium e Playwright, aplicadas em projetos web. A análise foi conduzida com base em critérios como configuração e setup, desempenho, linguagens de programação suportadas, ferramentas de depuração, navegadores compatíveis, suporte a espera automática, re-tentativas de execução, suporte a múltiplas abas, testes isolados, escalabilidade e funcionalidades de gravação/reprodução. Os resultados obtidos permitem avaliar o desempenho e a eficiência de cada ferramenta, destacando suas vantagens e limitações no contexto da automação de testes em aplicações web. A seguir, são detalhados os principais achados, seguidos de uma discussão crítica sobre as implicações desses resultados para a escolha da ferramenta mais adequada em diferentes cenários de desenvolvimento.

4.1 Setup e configuração

O Cypress requer Node.js e oferece um processo de instalação relativamente simples. Após a instalação do Cypress usando npm, o Cypress Test Runner orienta na seleção dos tipos de teste (por exemplo, ponta a ponta) e navegadores. O Cypress também gera automaticamente arquivos de configuração e testes de amostra (CYPRESS.IO, 2024).

O Selenium exige a instalação do JDK e das dependências do Selenium, e também requer que o driver do navegador utilizado nos testes esteja baixado e configurado na máquina do desenvolvedor. Ao contrário do Cypress, o Selenium não cria automaticamente a estrutura de diretórios e arquivos de teste iniciais; o desenvolvedor precisa configurar isso manualmente (SELENIUMHQ, 2024).

O Playwright simplifica seu processo de instalação exigindo apenas a linguagem de programação relevante, como Node.js para JavaScript. Um comando simples instala o Playwright e baixa automaticamente os navegadores necessários. Após a instalação, o Playwright fornece um teste de amostra no diretório tests e cria automaticamente um arquivo de configuração do Playwright (MICROSOFT, 2024).

No geral, cada *framework* de teste oferece um processo de instalação amigável ao usuário. No entanto, Playwright e Cypress aumentam ainda mais a conveniência do usuário por meio de configurações automatizadas e geração de testes de amostra.

4.2 Desempenho

O desempenho é crucial em testes, pois tempos de execução mais rápidos resultam em ciclos de feedback mais curtos. Com base nos benchmarks apresentados no capítulo anterior, que consistiram em automatizar 6 cenários de ponta a ponta e executar esses testes 10 vezes em cada *framework*, o Selenium obteve o melhor desempenho, com 12,34 segundos de tempo médio de execução no modo visual e 12,66 no modo *headless*, ambos com desvio padrão abaixo de 1. Em segundo lugar, o Playwright registrou 15,07 segundos de tempo médio de execução no modo visual e 15,08 no modo *headless*, também com desvio padrão abaixo de 1. Por último, o Cypress apresentou 16 segundos de tempo médio de execução no modo visual e 21 segundos no modo *headless*, sendo que, neste último, o desvio padrão ultrapassou o valor de 1.O A superioridade do Selenium em desempenho pode ser atribuída à sua simplicidade e integração nativa com os navegadores por meio do WebDriver. Por outro lado, o Cypress, apesar de sua facilidade de uso e estabilidade, sofre com o overhead imposto por sua arquitetura no mesmo ciclo do navegador. Já o Playwright, equilibrando modernidade e funcionalidade, encontra-se entre os dois, sendo ligeiramente mais lento que o Selenium, mas oferecendo maior flexibilidade e suporte.

4.3 Linguagens suportadas

Entre as linguagens suportadas, o Selenium se destaca, com suporte para C Sharp, Python, Ruby, Kotlin, JavaScript e Java. Em seguida, o Playwright oferece suporte para JavaScript, Python, .NET e Java. Por último, o Cypress é suportado apenas pelo JavaScript.

4.4 Ferramentas de depuração

O Cypress é equipado com um conjunto abrangente de ferramentas de depuração. Ele fornece informações detalhadas de erro e rastreamentos de pilha, além de permitir a visualização da execução de comandos em tempo real. O Cypress também integra a funcionalidade de pausa, permitindo que os usuários percorram o código ou retomem comandos no log de comando. Além disso, o comando *debug*() e o uso da instrução *debugger* nas ferramentas de desenvolvimento do navegador Chromium melhoram os recursos de depuração (CYPRESS.IO, 2024).

As ferramentas de depuração do Selenium, por outro lado, são menos sofisticadas. Elas dependem principalmente de capacidades de registro para fornecer informações adicionais para auxiliar na solução de problemas. O desenvolvedor precisa usar técnicas de depuração padrão, como definir pontos de interrupção em *IDEs* ou editores, para passar

pelos testes, de maneira semelhante à depuração de programas regulares (SELENIUMHQ, 2024).

O Playwright oferece ferramentas de depuração abrangentes, incluindo integração com o depurador do *VSCode*, permitindo testes diretos no popular editor de código para uma solução de problemas eficiente. As ferramentas de depuração avançadas adicionais que aprimoram o processo de depuração do Playwright incluem:

- **Playwright Inspector**: Uma interface gráfica que permite percorrer seus testes, editar localizadores ao vivo ou visualizar registros de ações.
- **Visualizador de Rastreamento**: Esta ferramenta GUI permite examinar os rastreamentos gravados dos seus testes no *Playwright*.
- **Ferramentas do Desenvolvedor do Navegador**: Executar o *Playwright* no modo de depuração habilita o objeto `playwright`, útil para inspecionar o DOM, visualizar logs do console ou verificar a atividade da rede.
- **Logs de API Detalhados**: Ao definir a variável de ambiente `DEBUG`, o *Playwright* registra interações detalhadas da API.
- **Modo Orientado**: Os testes podem ser executados visualmente, com uma opção para desacelerar a execução.

Quando se trata de depuração, o Playwright lidera com sua ampla variedade de ferramentas de depuração, que atendem a diversas necessidades de depuração. As ferramentas de depuração do Selenium, por outro lado, são menos sofisticadas. Elas dependem principalmente de capacidades de registro para fornecer informações adicionais para auxiliar na solução de problemas. O desenvolvedor precisa usar técnicas de depuração padrão, como definir pontos de interrupção em *IDEs* ou editores, para passar pelos testes, de maneira semelhante à depuração de programas regulares.

4.5 Navegadores suportados

O Playwright oferece um suporte robusto a vários navegadores, incluindo janelas de visualização móveis, com flexibilidade nas atualizações e instalações de navegadores. Isso o torna adequado para testes em diversos ambientes (MICROSOFT, 2024).

O Cypress fornece suporte estável para navegadores Chromium e Firefox, com suporte experimental para WebKit. Ele é eficaz quando a compatibilidade com WebKit não é crucial e suporta configuração fácil de viewport móvel (CYPRESS.IO, 2024).

O Selenium é compatível com todos os principais navegadores, incluindo navegadores mais antigos, como o Internet Explorer. No entanto, a configuração de janelas de

visualização móvel requer codificação manual (SELENIUMHQ, 2024). Para projetos que necessitam de ampla compatibilidade entre navegadores e dispositivos móveis, o Playwright é a melhor escolha.

4.6 Suporte a Espera Automática

A espera automática em *frameworks* de teste garante que as ações sejam pausadas até que os elementos se tornem interativos (clicáveis, visíveis, habilitados) antes de prosseguir. Sem a espera automática, os testadores dependem de esperas implícitas (configurações globais) ou esperas explícitas (condições específicas), oferecendo vários graus de controle.

O Cypress suporta espera automática, monitorando ativamente eventos de carregamento/descarregamento de página, animações de elementos, mudanças de visibilidade e cobertura. Ele espera inteligentemente até que as condições necessárias sejam atendidas antes de prosseguir (CYPRESS.IO, 2024).

O Selenium não possui uma espera automática integrada, dependendo de esperas implícitas definidas globalmente. Embora reduzam erros, essas esperas exigem configuração manual e podem levar a resultados menos previsíveis (SELENIUMHQ, 2024).

O Playwright oferece uma espera automática robusta por meio de métodos localizadores como `getByRole()`, `page.getByLabel()` e `page.getByAltText()`. Ele verifica a visibilidade do elemento, estabilidade, estado habilitado e editabilidade antes de agir, abortando se as condições não forem atendidas dentro de um tempo limite (MICROSOFT, 2024).

4.7 Re-tentativas (Para Flake Test)

Um *flake test* (ou teste intermitente) é um teste automatizado que não é consistentemente reproduzível. Ou seja, ele pode passar em algumas execuções e falhar em outras, sem que haja uma mudança clara no código ou no ambiente que justifique a diferença. Esses testes são problemáticos porque introduzem incerteza e podem mascarar problemas reais no código.

Para lidar com *flake tests*, a repetição dos testes é uma prática valiosa. Repetir os testes várias vezes ajuda a identificar e mitigar testes instáveis. Além disso, a implementação de esperas automáticas pode ajudar a estabilizar os testes, garantindo que o código esteja completamente carregado e pronto para interação antes de prosseguir. Essas abordagens ajudam a atenuar os problemas associados a *flake tests* e a melhorar a confiabilidade dos testes automatizados.

O Cypress suporta métodos localizadores com novas tentativas integradas e permite configurações para detectar testes instáveis (CYPRESS.IO, 2024). O Selenium, por outro lado, não possui mecanismos de repetição integrados, tornando o manuseio de testes instáveis mais desafiador (SELENIUMHQ, 2024).

O Playwright suporta novas tentativas automáticas para métodos localizadores como `page.getByRole()` e `page.getByLabel()`, que incluem espera automática (MICROSOFT, 2024). Para testes instáveis, novas tentativas globais podem ser configuradas, ou configurações específicas de novas tentativas podem ser ajustadas usando o parâmetro `-retries`.

4.8 Suporte a várias abas

Manipular várias páginas ou guias durante os testes é essencial para simular com precisão as interações do usuário no mundo real. O Cypress não tem suporte para múltiplas abas do navegador, o que restringe sua capacidade de testar cenários complexos de usuários (CYPRESS.IO, 2024). O Selenium tem suporte a múltiplas abas, embora não diferencie entre abas e janelas (SELENIUMHQ, 2024). O Playwright habilita múltiplas abas dentro de contextos de navegador, simplificando interações simultâneas com várias páginas (MICROSOFT, 2024).

4.9 Suporte a testes isolados

O isolamento de testes, essencial em testes de ponta a ponta, garante que cada teste seja executado em um ambiente separado, evitando interferências. O Cypress oferece suporte ao isolamento de testes redefinindo automaticamente os contextos do navegador antes de cada teste (CYPRESS.IO, 2024). O Selenium, por outro lado, não fornece suporte integrado para isolamento de testes (SELENIUMHQ, 2024). O Playwright se destaca no isolamento de testes através do uso de BrowserContexts, que funcionam como perfis de usuário separados, mas de maneira mais leve e rápida (MICROSOFT, 2024).

4.10 Escalabilidade

A escalabilidade é essencial para ferramentas de teste, especialmente ao gerenciar grandes suítes de testes.

O Cypress oferece suporte a testes paralelos, mas para utilizar esse recurso, é necessário dividir os testes em vários arquivos. O Cypress distribui cada arquivo de especificação entre as máquinas disponíveis, utilizando uma estratégia de balanceamento de carga (CYPRESS.IO, 2024). Também é possível executar testes em paralelo em uma

única máquina, embora a própria documentação do Cypress desaconselhe isso devido ao alto consumo de recursos.

O Selenium pode ser escalado através do Selenium Grid, que permite a execução de testes em paralelo em múltiplas máquinas remotas (SELENIUMHQ, 2024). O *Selenium Grid* roteia os comandos do cliente de teste para diferentes instâncias de navegador em servidores remotos, otimizando o desempenho e acelerando a execução dos testes.

O Playwright, por padrão, executa automaticamente os arquivos de teste em paralelo em todos os núcleos de *CPU* disponíveis, gerando processos de trabalho. Ele também permite a paralelização de testes dentro de um único arquivo. Para suítes maiores, o Playwright oferece suporte à fragmentação de testes, distribuindo-os em várias máquinas para reduzir o tempo de execução (MICROSOFT, 2024).

Se o projeto requer escalabilidade, seja em uma única máquina ou em múltiplas, Playwright, Cypress e Selenium são boas opções.

4.11 Suporte para gravação e reprodução

A funcionalidade de gravação e reprodução permite que o desenvolvedor interaja com um aplicativo manualmente e então gere *scripts* executáveis com base nessas interações, simplificando o processo de criação de testes.

O Cypress fornece funcionalidade de gravação e reprodução por meio de um recurso experimental chamado *Cypress Studio*. Embora promissor, esse recurso ainda está em desenvolvimento e pode não ser tão maduro quanto outras opções (CYPRESS.IO, 2024).

O Selenium também oferece suporte para gravação e reprodução por meio do Selenium IDE, uma extensão do navegador que permite aos usuários gravar interações e gerar código *Selenium WebDriver* (SELENIUMHQ, 2024).

O Playwright oferece suporte robusto para gravação e reprodução por meio de seu recurso codegen, que gera executáveis com base nas interações do usuário com o aplicativo (MICROSOFT, 2024).

Embora todas as estruturas suportem gravação e reprodução até certo ponto, Playwright e Selenium oferecem soluções mais maduras e estáveis.

Os resultados da análise comparativa das ferramentas de teste E2E Cypress, Selenium e Playwright são sintetizados na Tabela 1. Nela, são apresentados os principais critérios avaliados, como configuração e setup, desempenho, linguagens de programação suportadas, ferramentas de depuração, navegadores compatíveis, entre outros. A tabela permite uma visão clara das vantagens e limitações de cada ferramenta, facilitando a escolha da solução mais adequada para diferentes cenários de desenvolvimento.

	Cypress	**Selenium**	**Playwright**
Setup e configuração (Complexidade)	Simples	Grande	Simples
Performance	Médio	Ótimo	Bom
Linguagens Suportadas	JavaScript	C#, Python, Ruby, Kotlin, JavaScript, Java	JavaScript, Python, .NET, Java
Ferramentas de depuração (Qualidade)	Ótimo	Ruim	Ótimo
Navegadores Suportados	Chromium, Firefox, Webkit (experimental), Edge	Chromium, Firefox, Webkit, Edge	Chromium, Firefox, Webkit, Edge
Suporte à espera automática	Possui	Não possui	Possui
Retentativas (Para Flake Test)	Possui	Não possui	Possui
Suporte a várias abas	Não possui	Possui	Possui
Suporte a testes isolados	Possui	Não possui	Possui
Escalabilidade (Qualidade)	Ótimo	Ótimo	Ótimo
Suporte para gravação e reprodução	Possui	Possui	Possui

Tabela 1 – Comparação entre Cypress, Selenium e Playwright

Com base nos resultados apresentados neste capítulo, é possível concluir que as ferramentas de teste E2E Cypress, Selenium e Playwright possuem características distintas que as tornam mais adequadas para diferentes cenários de desenvolvimento. O Cypress se destaca pela facilidade de configuração e suporte a espera automática, sendo uma excelente escolha para projetos que priorizam a simplicidade e a integração com JavaScript. O Selenium, por sua vez, oferece a maior compatibilidade com diferentes linguagens de programação e navegadores, sendo ideal para projetos que exigem flexibilidade e suporte a uma ampla gama de ambientes. Já o Playwright combina o melhor dos dois mundos, com uma configuração simples, suporte robusto a múltiplos navegadores e ferramentas avançadas de depuração, além de ser altamente escalável. Dessa forma, a escolha da ferramenta mais adequada dependerá das necessidades específicas do projeto, como a linguagem de programação utilizada, a complexidade dos cenários de teste e os requisitos de escalabilidade e compatibilidade com navegadores.

5 Conclusão

A análise comparativa das ferramentas de teste E2E Cypress, Selenium e Playwright revelou que cada uma delas possui características únicas que as tornam mais adequadas para diferentes cenários de desenvolvimento. O Cypress se destaca pela simplicidade de configuração e suporte a espera automática, sendo uma excelente escolha para projetos que priorizam a integração com JavaScript e a facilidade de uso. No entanto, sua limitação em relação ao suporte a múltiplas abas e a compatibilidade com navegadores pode restringir sua aplicabilidade em cenários mais complexos.

Por outro lado, o Selenium oferece a maior flexibilidade, com suporte a uma ampla gama de linguagens de programação e navegadores, incluindo versões mais antigas. Isso o torna ideal para projetos que exigem compatibilidade com diversos ambientes e maior controle sobre a configuração dos testes. Todavia, sua configuração inicial é mais complexa e suas ferramentas de depuração são menos sofisticadas em comparação com as outras opções.

O Playwright surge como uma solução intermediária, combinando a simplicidade de configuração do Cypress com a flexibilidade e robustez do Selenium. Ele oferece suporte a múltiplos navegadores, ferramentas avançadas de depuração e escalabilidade, sendo uma excelente escolha para projetos que exigem testes mais complexos e paralelização eficiente.

Dessa forma, a escolha da ferramenta mais adequada dependerá das necessidades específicas de cada projeto, como a linguagem de programação utilizada, a complexidade dos cenários de teste, os requisitos de escalabilidade e a compatibilidade com navegadores. Este estudo contribui para a tomada de decisões mais informadas por parte de desenvolvedores e equipes de qualidade de software, auxiliando na otimização do processo de automação de testes e na melhoria da qualidade do software entregue.

A importância deste estudo reside no fato de que ele oferece uma análise detalhada e prática das principais ferramentas de teste E2E disponíveis no mercado, auxiliando desenvolvedores e equipes de qualidade de software a tomarem decisões mais informadas. Ao compreender as vantagens e limitações de cada ferramenta, os profissionais podem escolher a solução que melhor se adapta às necessidades específicas de seus projetos, otimizando o processo de automação de testes e, consequentemente, a qualidade do software entregue. Além disso, este trabalho contribui para o avanço do conhecimento na área de testes automatizados, servindo como base para futuras pesquisas e aprimoramentos no uso dessas ferramentas.

Referências

APPLITOOLS. **AI-Powered Visual Testing**. 2023. <https://applitools.com/>. Citado na página 11.

CYPRESS.IO. **Cypress Documentation**. 2024. Acessado em: setembro de 2024. Disponível em: <https://www.cypress.io/>. Citado 7 vezes nas páginas 18, 41, 42, 43, 44, 45 e 46.

EICH, B. **JavaScript: The Definitive Guide**. [S.l.]: Netscape Communications, 1995. Citado na página 18.

FOWLER, M.; HIGHSMITH, J. **The Agile Manifesto**. 2001. Disponível em: <https://agilemanifesto.org/>. Disponível em: <https://agilemanifesto.org/>. Citado 3 vezes nas páginas 8, 11 e 14.

FREEMAN, S.; PRYCE, N. **Growing Object-Oriented Software, Guided by Tests**. [S.l.]: Addison-Wesley, 2009. Citado na página 28.

GAROUSI, V.; FELDERER, M.; MäNTYLä, M. V. The need for automated end-to-end testing in agile development: A survey. **IEEE Software**, v. 36, n. 2, p. 48–55, 2019. Discusses the importance of automated E2E testing in Agile environments. Citado 6 vezes nas páginas 8, 11, 12, 13, 14 e 15.

KIM, G.; HUMBLE, J.; DEBOIS, P.; WILLIS, J. **The DevOps Handbook: How to Create World-Class Agility, Reliability, Security in Technology Organizations**. [S.l.]: IT Revolution, 2016. Citado na página 11.

LAUKKANEN, E.; ITKONEN, J.; LASSENIUS, C. Problems, causes and solutions when adopting continuous delivery—a systematic literature review. **Information and Software Technology**, v. 82, p. 55–79, 2017. Explores challenges and solutions in adopting CI/CD practices. Citado 5 vezes nas páginas 10, 11, 13, 15 e 16.

LEFFINGWELL, D. **SAFe 5.0 for Lean Enterprises**. [S.l.]: Scaled Agile, Inc., 2020. Citado na página 11.

MESZAROS, G. **xUnit Test Patterns: Refactoring Test Code**. [S.l.]: Addison-Wesley Professional, 2007. Provides insights into unit testing frameworks like JUnit and NUnit. Citado 6 vezes nas páginas 11, 12, 13, 14, 15 e 28.

MICROSOFT. **Playwright Documentation**. 2024. Acessado em: setembro de 2024. Disponível em: <https://playwright.dev/>. Citado 6 vezes nas páginas 34, 41, 43, 44, 45 e 46.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. 3rd. ed. [S.l.]: Wiley, 2011. Covers various types of software testing, including unit and integration testing. Citado 3 vezes nas páginas 8, 10 e 12.

NAYANAJITH, G. **Cypress Page Object Model: Tutorial**. 2022. Accessed: 2024-09-16. Disponível em: <https://www.browserstack.com/guide/cypress-page-object-model>. Citado na página 22.

PRESSMAN, R. S. **Software Engineering: A Practitioner's Approach**. 8th. ed. [S.l.]: McGraw-Hill Education, 2014. Discusses software quality and testing methodologies. Citado 3 vezes nas páginas 10, 12 e 13.

QUALIDADE de Software: Conceitos e Características. 2024. Acesso em: 2024-09-16. Disponível em: <https://www.devmedia.com.br/qualidade-de-software-engenharia-de-software-29/18209>. Citado na página 26.

SELENIUMHQ. **Selenium Documentation**. 2024. Acessado em: setembro de 2024. Disponível em: <https://www.selenium.dev/>. Citado 5 vezes nas páginas 41, 43, 44, 45 e 46.

SHAHIN, M.; BABAR, M. A.; ZHU, L. **Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices**. [S.l.: s.n.], 2017. v. 5. 3909-3943 p. Citado na página 10.

SYSTEMS, D. **axe-core Accessibility Testing**. 2021. <https://www.deque.com/axe/>. Citado na página 11.

ZHAO, W.; ZHANG, L.; ZHANG, Y. A survey of end-to-end testing tools for web applications. **Journal of Software Engineering**, v. 9, n. 3, p. 123–135, 2014. Compares various E2E testing tools, including Selenium and Cypress. Citado 3 vezes nas páginas 13, 15 e 16.