

Automação de Testes Funcionais: Uma Análise Técnica dos Frameworks Cypress e Selenium

Alex A. Cândido Silva¹, Dennys Barros¹, Rodrigo Cunha¹, Wylliams Santos^{1,2}

¹ CESAR School – Avenida Cais do Apolo, 77, Recife – PE

²Universidade de Pernambuco (UPE), Recife – PE

alexscandido10, dennys.sbarros, rcalvesmoreira@gmail.com, wbs@upe.br

Resumo. Com o aumento da velocidade de criação e complexidade dos softwares, faz-se necessário investimentos em automação de testes para manter uma boa qualidade dos produtos desenvolvidos. Em consequência disso, vários frameworks de testes foram criados e disponibilizados no mercado, dificultando assim a escolha por aquela ferramenta que melhor se adapta a determinado projeto. Este trabalho realizou uma análise entre o Selenium, uma das ferramentas mais tradicionais, e o Cypress, uma ferramenta bastante promissora e que vem sendo cada vez mais utilizada. Para esta análise, foram desenvolvidos scripts de testes utilizando uma aplicação web. Foi então realizado uma comparação entre esses scripts a partir de determinadas características. Por fim, são apresentados os prós e contras entre cada um desses frameworks.

Abstract. With the increase in the development speed and complexity of softwares, investments in test automation are necessary to maintain a good quality of the developed products. As a result, several test frameworks were made available on the market, making it difficult to choose the tool that best suits a given project. This work carried out an analysis of Selenium, one of the most traditional tools, and Cypress, a very promising tool that has been increasingly used. For this analysis, test scripts were developed using a web application. A comparison was then carried out between these scripts based on certain characteristics. Finally, are presented the pros and cons between these frameworks.

1. Introdução

A execução de testes de software é crucial para garantir a qualidade no ciclo de desenvolvimento de software. De acordo com [Molinari 2010], ao nos dedicarmos aos testes, vamos obter um retorno direto em qualidade. Para [Maldonado et al. 2001], as atividades de teste de software permitem a verificação e validação de uma aplicação. A verificação é fruto da análise dinâmica da aplicação, assim, checagens são realizadas durante a sua execução de modo a encontrar possíveis erros e/ou nos casos em que os erros não são encontrados, essa execução possibilita ao time de desenvolvimento um sentimento de confiança de que a aplicação está ou continua funcionando como esperado.

A tecnologia nos dias atuais permite um rápido acesso a informação. Em questão de minutos é possível se comunicar com pessoas em diferentes lugares do mundo, realizar compras através de sites de *e-commerce* sem sair de casa, bem como realizar serviços

em empresas privadas e públicas de maneira prática. Tudo isso só é possível através da transformação digital proporcionada pelas aplicações web modernas e suas operações digitais. Os aplicativos web tem facilitado a vida das pessoas. Esses aplicativos tem se tornado cada vez mais complexos devido a riqueza dos recursos computacionais utilizados, que tornam a experiência do usuário mais fluída e dinâmica. Essas características elevam a complexidade ao testar os aplicativos web modernos.

O curto prazo para um lançamento e desenvolvimento de aplicativos cada vez mais complexos, faz com que testes automáticos se tornem extremamente necessários para garantir qualidade durante o ciclo de desenvolvimento iterativo e incremental. A automação de testes se caracteriza como o uso de uma ferramenta que replica os comportamentos de um ser humano na condição de usuário de uma dada aplicação. A utilização de ferramentas de automação permite a execução dos casos de testes de forma autônoma, o que proporciona a redução do tempo de execução dos testes, o aumento na velocidade do feedback e confiança para a equipe de desenvolvimento, bem como na qualidade e cobertura do projeto. Como reforçado por [Pressman 2001], “o teste de software é um elemento crítico da garantia de qualidade de software e representa a última revisão da especificação, do projeto e da codificação”. Assim, notamos o quão importante é a execução de testes automáticos durante todo o ciclo de desenvolvimento de um projeto, e não apenas ao seu término.

Atualmente existem diversos tipos de frameworks para automação de testes, e escolher entre um deles pode ser uma tarefa difícil. Uma das ferramentas mais tradicionais para automação de testes em plataformas web é o [Selenium 2021b], que é bastante consolidado e utilizado, lidando bem com interfaces de aplicações web clássicas, através da simulação de simples interações de cliques via mouse e manipulação de valores por meio do teclado. Entretanto o Selenium foi desenvolvido em 2005 onde as aplicações web eram mais simples que as desenvolvidas atualmente, onde as aplicações contém muitos elementos web dinâmicos, impondo dificuldades ao Selenium e reduzindo assim a performance na execução dos testes bem como a ocorrência de execuções de teste *flaky* ou inconsistentes.

Diante dessa problemática, esse trabalho apresenta um framework de testes que surge como alternativa para lidar com automação de aplicações web dinâmicas: Cypress. A escolha desse framework se dá pelas suas interessantes características em relação ao Selenium, mas também por alguns outros pontos relevantes a serem considerados tais como ser open source, pela sua adoção por grandes empresas, a quantidade de estrelas em seu repositório no GitHub e a expressiva quantidade de downloads nos últimos anos, chegando a superar o Selenium, como visto na Tabela 1 e Figura 1.

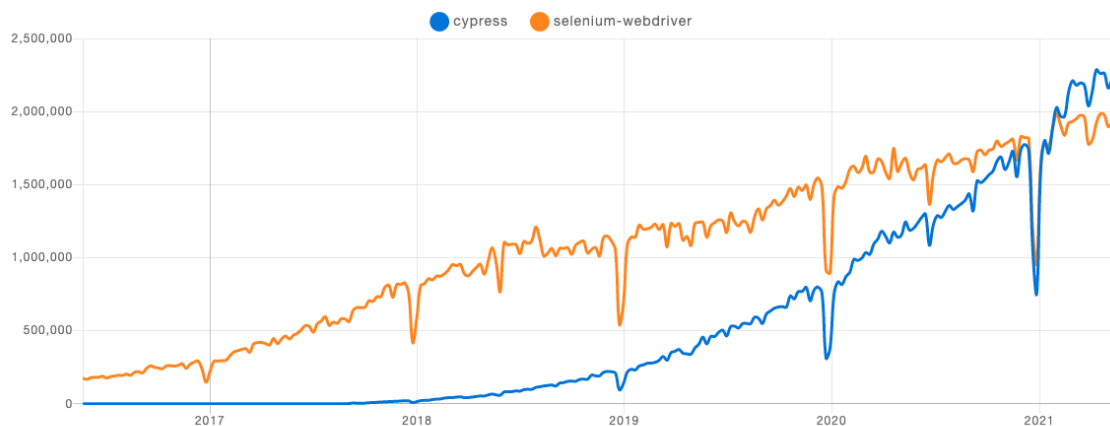
Tabela 1. Dados sobre o framework Cypress.

Framework	Linguagem	Empresas que usam	Estrelas no GitHub
Cypress	Javascript	Adobe, Johnson e Johnson, DHL, etc	31.2K

Atualmente o Cypress suporta a maioria dos browsers disponíveis no mercado, como Firefox, Chrome, e os browsers baseados no Chrome como Edge, Electron e Brave.

Este trabalho apresenta o framework Cypress através do desenvolvimento de um projeto de automação de testes funcionais. O desenvolvimento desse projeto permite ao

Downloads in past 5 Years ▾



Stats



	stars 🌟	issues 🐛	updated 🔄	created 🗓	size 📦
 cypress	31,283	1,403	May 24, 2021	Mar 3, 2015	
 selenium-webdriver	20,856	188	May 24, 2021	Jan 14, 2013	minzipped size: 114.6 KB

Figura 1. Quantidade de downloads de cada framework nos últimos 5 anos.

Fonte: <https://www.npmtrends.com/cypress-vs-selenium-webdriver>

leitor conhecer melhor as principais característica e vantagens do Cypress ao compará-lo com o Selenium no momento de decidir qual ferramenta utilizar antes de começar um projeto de automação.

O site Demo QA (<https://demoqa.com/>) foi utilizado como exemplo de aplicação web dinâmica devido a grande quantidade de elementos dinâmicos que são renderizados.

Dentre os objetivos do trabalho estão o levantamento das principais funcionalidades e limitações de cada framework, realização das respectivas configurações de ambiente, elaboração da arquitetura do projeto de automação seguindo as boas práticas de engenharia de software no tocante a automação de teste de software, desenvolvimento do mesmo conjunto de scripts de testes em cada framework, execução e validação do experimento e por fim, a elaboração de um relatório com os resultados obtidos.

Este trabalho está organizado em quatro Seções. A Seção 2 contém a revisão da literatura sobre os frameworks escolhidos e trabalhos relacionados. A Seção 3 contém o planejamento e execução do experimento, onde iremos apresentar as principais características e diferenças entre os frameworks. Por fim, a Seção 4 encerra o trabalho com a conclusão do experimento realizado.

2. Revisão da Literatura

Uma das ferramentas de automação mais comuns é o Selenium que é considerada uma ferramenta de automação robusta e confiável devido a sua longa existência no mercado que data de mais de 10 anos [Mobaraya and Ali 2019].

Selenium é um projeto abrangente para uma variedade de ferramentas e bibli-

otecas que permitem e dão suporte à automação de navegadores da web. Ele fornece extensões para emular a interação do usuário com os navegadores, um servidor de distribuição para dimensionar a alocação do navegador e a infraestrutura para implementações da especificação W3C WebDriver que permite escrever código intercambiável para todos os principais navegadores da web. No núcleo do Selenium está o WebDriver, uma interface para escrever conjuntos de instruções que podem ser executados alternadamente em muitos navegadores [Selenium 2021a].

Entretanto, [Mobaraya and Ali 2019] destaca que [Vila et al. 2017], [Bulla and Brunda 2016], [Mane et al. 2016] e estudos recentes com desenvolvedores de automação, mostram uma significativa limitação do Selenium no manuseio renderização de elementos dinâmicos, carregamentos de página e janelas pop-up na aplicações modernas atuais.

O Cypress é um framework open source de automação de testes end-to-end usando Javascript [Berçam 2019]. É um framework que vem ganhando adeptos por ser bastante completo, contendo bibliotecas de asserções, possibilidade de realizar mocking e stubs, debug de execução, execução rápida e sem flakes, além de possuir uma boa documentação e também por não ter o selenium em sua arquitetura, deixando assim problemas relacionados com ele [Cypress.io 2019a]. Em poucas palavras o Cypress é uma ferramenta de teste de última geração para front-end criada para a web moderna. Abordamos os principais pontos problemáticos que desenvolvedores e engenheiros de controle de qualidade enfrentam ao testar aplicativos modernos. Tornando possível configurar, escrever, executar e debugar testes.

O Cypress é mais frequentemente comparado ao Selenium; no entanto, Cypress é fundamentalmente e arquitetonicamente diferente. Cypress não é limitado pelas mesmas restrições que o Selenium. Isso permite que você escreva testes mais rápidos, fáceis e confiáveis [Cypress.io 2019c].

A [Thoughtworks 2020] na edição 22 do seu Tech Radar recomenda a adoção do Cypress. Segundo a empresa, ele ainda é um favorito entre os times nos quais as pessoas desenvolvedoras gerenciam os testes de ponta-a-ponta, como parte de uma pirâmide de testes saudável, é claro. Decidimos destacar novamente neste Radar porque as versões recentes do Cypress adicionaram suporte ao Firefox, e sugerimos fortemente testar em múltiplos navegadores. O domínio dos navegadores Chrome e baseados no Chromium levou os times a uma tendência preocupante de aparentemente testar apenas no Chrome, o que pode levar a surpresas desagradáveis.

Testes inconsistentes geram muita dor de cabeça, [Cowan 2019] menciona que um dos momentos mais desafiadores enfrentados foi o erro de compilação devido a um teste de automação que falhou apenas para que ela passasse magicamente apenas executando novamente a compilação. Esse fenômeno ou automação de testes zumbis costuma ser chamado de *flake*. O principal problema com o flake é que ele não é determinístico, o que significa que um teste pode exibir um comportamento diferente quando executado com as mesmas entradas em momentos diferentes. Você pode observar a confiança em seu conjunto de testes de regressão aumentar quando o número de testes não determinísticos aumenta. Provavelmente, um teste flake está relacionado ao tempo, latência e assincronia que estamos tentando resolver com os nossos assistentes *Thread.sleep* e *waitFor* que

precisamos continuar escrevendo.

A principal vantagem de usar o Cypress é que ele simplifica os testes assíncronos. Ele também define uma opção de espera automática no framework, onde acontecem esperas automáticas pelo carregamento completo de elementos DOM ou pela completa renderização de qualquer animação, após a conclusão dos carregamentos é que o framework inicia a busca por um dado elemento web. Essa é uma das principais limitações do Selenium, uma vez que os automatizadores precisam definir esperas explícitas e implícitas para aguardarem o carregamento completo de cada elemento da página que se deseja interagir. Quando essas esperas não são bem definidas, acarreta na adição de threads de tempo que seguram/congelam a execução dos testes por uma quantidade fixa de tempo, o que ocasiona um acréscimo significativo na performance e no tempo de execução de suítes de testes.

2.1. Trabalhos relacionados

O trabalho *Technical Analysis of Selenium and Cypress as Functional Automation Framework for Modern Web Application Testing* [Mobaraya and Ali 2019], realiza um estudo semelhante ao executado neste trabalho no que se refere a analisar diferenças entre Selenium e Cypress.

Duas métricas utilizadas pelo trabalho de Mobaraya são similares às utilizadas neste estudo: Comparar a quantidade de linhas de código e o tempo de execução dos scripts. Mobaraya mensura a quantidade de linhas de código necessárias para automatizar uma determinada funcionalidade e nomina essa métrica como eficiência de teste. O sistema utilizado por Mobaraya para o desenvolvimento dos scripts foi o site de comércio eletrônico AliExpress. Mobaraya selecionou duas funcionalidades do sistema, criou cinco cenários para cada, totalizando um montante de 10 cenários nos quais os scripts de teste foram desenvolvidos em Selenium e Cypress, e assim os comparativos foram realizados.

O experimento executado neste trabalho, seleciona um conjunto de cenários distintos que permitem explorar variados tipos de interações, possibilitando exercitar os frameworks Selenium e Cypress em diferentes circunstâncias. Este trabalho também realiza uma comparação a cerca da quantidade de linhas de código e tempo de execução dos scripts, tendo em vista que um novo experimento, com uma amostra diferente nos permite reforçar ou contestar resultados de outros trabalhos anteriores como o de Mobaraya. Mas este trabalho vai além e não se restringe apenas a comparação dessas métricas, ao trazer uma análise detalhada de outras características importantes que foram evidenciadas durante a jornada de desenvolvimento dos scripts, possibilitando auxiliar o leitor na tomada de decisão da ferramenta a ser utilizada antes de começar um projeto de automação.

3. O experimento

Esse trabalho foi conduzido através de um experimento, onde os frameworks Cypress e Selenium foram estudados e utilizados para elaborar um projeto de automação web. A elaboração das soluções para os desafios escolhidos tem por intuito a identificação das principais características de cada framework durante o processo de desenvolvimento dos scripts, permitindo através da exploração das tecnologias, a descoberta de vantagens, deficiências, complexidade de implementação, quantidade de linhas de código para desenvolver a mesma solução, tempo de execução, entre outros aspectos. As seguintes etapas foram realizadas durante o experimento:

1. Definição da aplicação web que será automatizada;
2. Definição do conjunto de scripts de testes a ser implementado na aplicação selecionada;
3. Configuração do ambiente do framework Cypress;
4. Configuração do ambiente do framework Selenium;
5. Desenvolvimento do conjunto de scripts de testes em cada framework;
6. Execução e validação do experimento (executar os mesmos cenários de testes em cada framework);
7. Conclusão do experimento (análise dos resultados e elaboração de um relatório com os resultados obtidos).

3.1. Arquitetura dos frameworks

Selenium e Cypress são as ferramentas de automação utilizadas nesse experimento para desenvolver os scripts de teste. O Selenium foi selecionado por ser uma das ferramentas mais poderosas e consolidadas na automação de aplicativos da web, enquanto que o Cypress foi selecionado por ser um framework com bastante adoção recentemente, além de oferecer uma nova maneira de automatizar aplicativos da web modernos.

Ambos os frameworks suportam adoção de padrões de projetos em sua arquitetura e organização de scripts de automação, como o popular padrão de design POM (*Page Object Model*) [Stewart 2015], que destina-se a aumentar a capacidade de manutenção e modularidade do código, tornando assim os testes mais legíveis, requerendo apenas as diferentes adaptações de acordo com as sintaxe das linguagens de programação utilizadas por cada framework. Nesse projeto o Cypress está implementado usando JavaScript [Gorej 2018] enquanto que o Selenium está usando Java. Entretanto, o blog oficial do Cypress sugere a utilização de um padrão chamado Ações de Aplicações (*App Actions*) [Cypress.io 2019b] que surge como uma alternativa para resolver algumas limitações e dificuldades do POM, tornando os testes de ponta a ponta mais rápidos e produtivos. No experimento realizado nesse trabalho nenhum padrão de projeto foi implementado, uma vez que não era relevante para os objetos de estudo definidos.

A Figura 2 ilustra a arquitetura do framework do Selenium e a Figura 3 mostra uma comparação entre o Cypress e demais frameworks, destacando as diferentes dependências necessárias.

Como visto na Figura 3 o Cypress possui um conjunto de várias bibliotecas empacotadas em sua arquitetura, o que torna mais fácil sua instalação bem como utilização por parte do usuário final, uma vez que não se faz necessário configurações adicionais. O Cypress também funciona de dentro do navegador e se comunica diretamente com o aplicativo sob teste (*UAT - Application Under Test*), enquanto que o Selenium instancia um WebDriver que atua como um terceiro para manipular as ações do usuário no aplicativo sob teste.

3.2. Definição da aplicação

Foi decidido utilizar a aplicação web Demo QA (<https://demoqa.com>) como visto na Figura 4, que se trata de uma plataforma de demonstração para certificação e treinamento do framework Selenium. Essa aplicação possui diversos tipos de desafios que nos permite exercitar os frameworks nos diversos tipos de interações existentes, fazendo com que

Selenium WebDriver Architecture

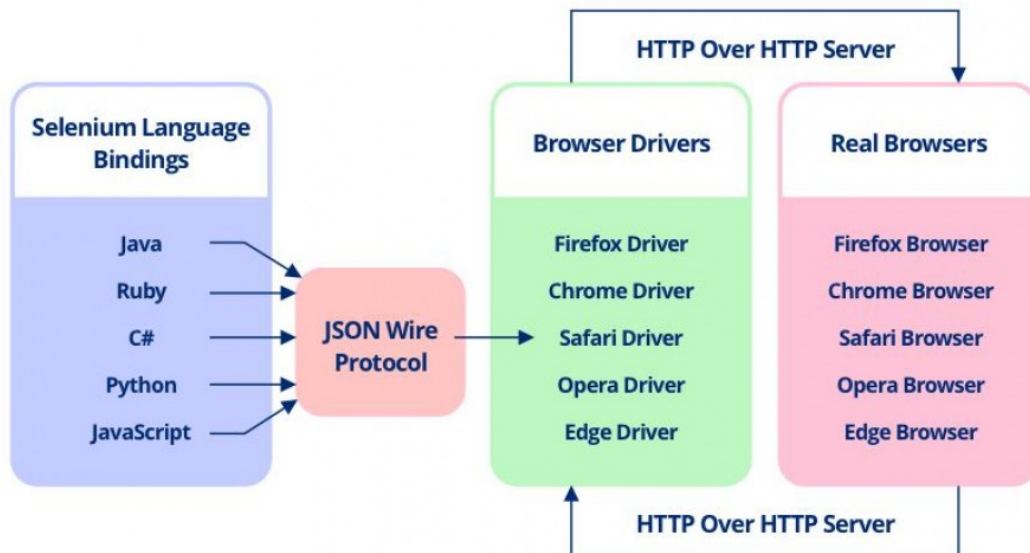


Figura 2. Arquitetura do framework Selenium [Sadiq 2021]

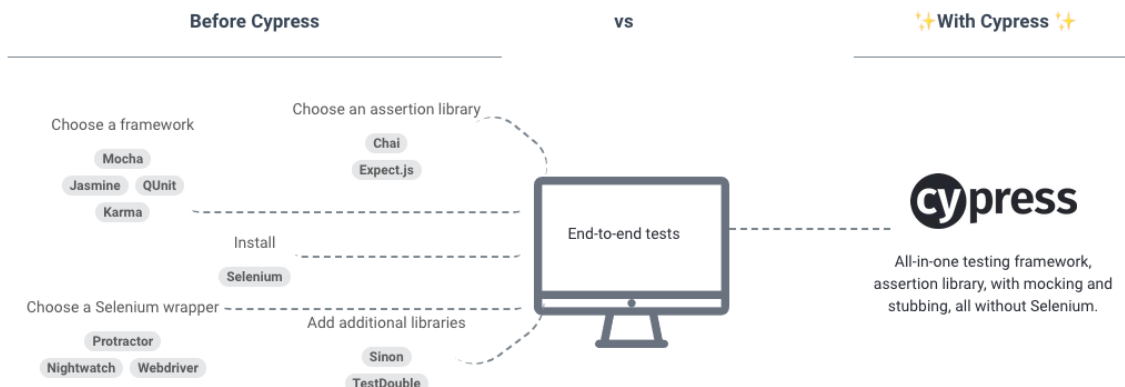


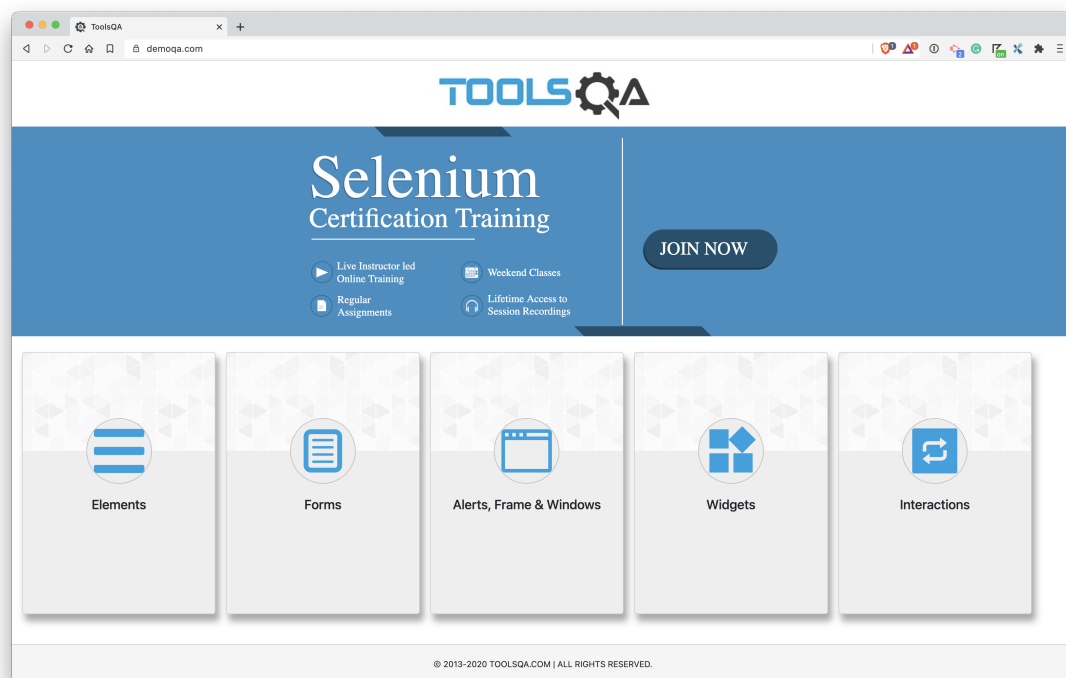
Figura 3. Comparação entre os demais frameworks e o Cypress [Cypress 2021]

o experimento contenha uma variedade de exemplos que são encontrados em diversas situações em aplicações reais.

3.3. Definição do conjunto de scripts

A aplicação Demo QA contém diversos tipos de elementos que possibilitam os mais variados tipos de interações/manipulações que são em sua grande maioria encontrados em aplicações reais. A Figura 5 a seguir contém as categorias de interações disponíveis na aplicação Demo QA e suas respectivas opções.

Entretanto, algumas dessas opções lidam com alguns tipos de interações que não são tão comuns e removemos da análise, baseado em sua aplicabilidade/utilidade para o contexto de validação de testes funcionais. Essas opções de interação foram marcadas como N/A (não aplicáveis), tais como *Broken Links - Images*, *Upload and Download*,



Group	Action	Applicable?	Selenium assignee	Selenium Status	Cypress assignee	Cypress Status	Comment	Source code & xPath / css selector
			Complete	100,00%	Complete	100,00%		
Elements	Text Box	<input checked="" type="checkbox"/>	Dennys	Done	Alex	Done		https://github.com/alexandidos/cypress-sample-project/blob/master/cypress/integ
	Check Box	<input checked="" type="checkbox"/>	Dennys	Done	Alex	Done		https://github.com/alexandidos/cypress-sample-project/blob/master/cypress/integ
	Radio Button	<input checked="" type="checkbox"/>	Dennys	Done	Alex	Done		https://github.com/alexandidos/cypress-sample-project/blob/master/cypress/integ
	Web Tables	<input checked="" type="checkbox"/>	Dennys	Done	Alex	Done		https://github.com/alexandidos/cypress-sample-project/blob/master/cypress/integ
	Buttons	<input checked="" type="checkbox"/>	Dennys	Done	Alex	Done		https://github.com/alexandidos/cypress-sample-project/blob/master/cypress/integ
	Links	<input checked="" type="checkbox"/>	Dennys	Done	Alex	Done	Alex: Aqui tem umas possibilidades de validar chamdas de API q. Dennys: Pra validar chamada de API com Selenium acredito que Alex: Acho que nesse caso, poderíamos deixar o Selenium como	https://github.com/alexandidos/cypress-sample-project/blob/master/cypress/integ
	Broken Links - Images	<input type="checkbox"/>		N/A		N/A	Alex: Sem relevância para o escopo de testes funcionais.	
	Upload and Download	<input type="checkbox"/>		N/A		N/A	Alex: Sem relevância para o escopo de testes funcionais.	
Forms	Dynamic Properties	<input type="checkbox"/>		N/A		N/A	Alex: Sem relevância para o escopo de testes funcionais.	
	Practice Forms	<input checked="" type="checkbox"/>	Dennys	Done	Rodrigo	Done		https://github.com/alexandidos/cypress-sample-project/blob/master/cypress/integ
Alerts, Forms & Windows	Browser Windows	<input checked="" type="checkbox"/>	Dennys	Done	Alex	Not Supported	Alex: O Cypress parece ter algumas limitações nesse sentido de novas janelas/abas.	
	Alerts	<input checked="" type="checkbox"/>	Dennys	Done	Alex	Done	Alex: Acho que por default ele não suporta, ele sugere que vc abra o link da página/aba que deseja validar, então não faz muito sentido, se co	https://github.com/alexandidos/cypress-sample-project/blob/master/cypress/integ
	Frames	<input type="checkbox"/>		N/A		N/A	Alex: Sem relevância para o escopo de testes funcionais.	
	Nested Frames	<input type="checkbox"/>		N/A		N/A	Alex: Sem relevância para o escopo de testes funcionais.	
	Modal Dialogs	<input checked="" type="checkbox"/>	Dennys	Done	Alex	Done		https://github.com/alexandidos/cypress-sample-project/blob/master/cypress/integ
Widgets	Accordion	<input checked="" type="checkbox"/>	Dennys	Done	Rodrigo	Done		https://github.com/alexandidos/cypress-sample-project/blob/8d53c9416d1cb185
	Auto Complete	<input checked="" type="checkbox"/>	Dennys	Done	Rodrigo	Done		https://github.com/alexandidos/cypress-sample-project/blob/8d53c9416d1cb185
	Date Picker	<input checked="" type="checkbox"/>	Dennys	Done	Rodrigo	Done		https://github.com/alexandidos/cypress-sample-project/blob/master/cypress/integ
	Slider	<input checked="" type="checkbox"/>	Dennys	Done	Alex	Done	Alex: Interessante, parece desafioador.	https://github.com/alexandidos/cypress-sample-project/blob/master/cypress/integ
	Progress Bar	<input checked="" type="checkbox"/>	Dennys	Done	Alex	Done	Alex: Interessante, parece desafioador. Porém talvez não seja tão relevante.	https://github.com/alexandidos/cypress-sample-project/blob/master/cypress/integ
	Tab	<input checked="" type="checkbox"/>	Dennys	Done	Alex	Done		https://github.com/alexandidos/cypress-sample-project/blob/master/cypress/integ
	Tool Tips	<input checked="" type="checkbox"/>	Dennys	Done	Alex	Done		https://github.com/alexandidos/cypress-sample-project/blob/master/cypress/integ
	Menu	<input checked="" type="checkbox"/>	Dennys	Done	Rodrigo	Done	Alex: Interessante, parece desafioador.	https://github.com/alexandidos/cypress-sample-project/blob/master/cypress/integ
Select Menu	<input checked="" type="checkbox"/>	Rodrigo	Done	Alex	Done		https://github.com/alexandidos/cypress-sample-project/blob/master/cypress/integ	

Dynamic Properties, *Frames* e *Nested Frames* e estão destacadas em amarelo na tabela exibida na Figura 5.

3.4. Configuração do ambiente do framework Cypress

Tabela 2. Lista de interações escolhidas para serem automatizadas.

Script ID	Descrição
Script_01	Verificar a interação com abas
Script_02	Verificar a interação com alertas - alerta é mostrado após 5 segundos
Script_03	Verificar a interação com alertas - clicar no botão para ver um alerta
Script_04	Verificar a interação com alertas - preencher prompt
Script_05	Verificar a interação com alertas de confirmação - botão cancelar
Script_06	Verificar a interação com alertas de confirmação - botão OK
Script_07	Verificar a interação com barra de progresso
Script_08	Verificar a interação com botões - botão direito e clique duplo
Script_09	Verificar a interação com caixas de texto
Script_10	Verificar a interação com diferentes janelas e abas
Script_11	Verificar a interação com elemento checkbox
Script_12	Verificar a interação com elemento radio button
Script_13	Verificar a interação com elementos de menu
Script_14	Verificar a interação com elementos de select menu
Script_15	Verificar a interação com elementos de slider
Script_16	Verificar a interação com elementos em tabelas
Script_17	Verificar a interação com elementos tooltip
Script_18	Verificar a interação com formulários e submetê-los
Script_19	Verificar a interação com links e endpoints
Script_20	Verificar a interação com modais
Script_21	Verificar a interação com seletores de data
Script_22	Verificar a interação com widgets accordion
Script_23	Verificar a interação com widgets de auto-complete

3.4.1. Instalação

Instale o Cypress usando *npm*:

1. Abra o terminal e instale o *npm* através do comando: *npm install*
2. Vá para a pasta onde será criado o projeto: *cd /caminho/do/projeto*
3. Execute o comando: *npm install cypress --save-dev*

Esse comando irá instalar o Cypress localmente como uma dependência de desenvolvimento em seu projeto. Certifique-se que você já rodou o comando *npm init*, ou tenha o diretório *node_modules*, ou o arquivo *package.json* no diretório raiz do seu projeto para garantir que o Cypress seja instalado no diretório correto.

3.4.2. Abrindo o Cypress

Se você usou *npm* para instalação, o Cypress foi instalado no diretório *./node_modules*, com seus binários executáveis acessíveis em *./node_modules/.bin*.

Abra o terminal e execute o comando: *./node_modules/.bin/cypress open* e então você terá aberto o Cypress do seu projeto principal. Após um momento, o Cypress Test Runner será aberto, como mostra a Figura 6.

3.4.3. Trocando navegadores

O Cypress Test Runner tenta encontrar todos os navegadores compatíveis na máquina do usuário. O campo de seleção suspenso para selecionar um navegador diferente pode ser encontrado no topo lateral direito do Test Runner, como ilustra a Figura 6.

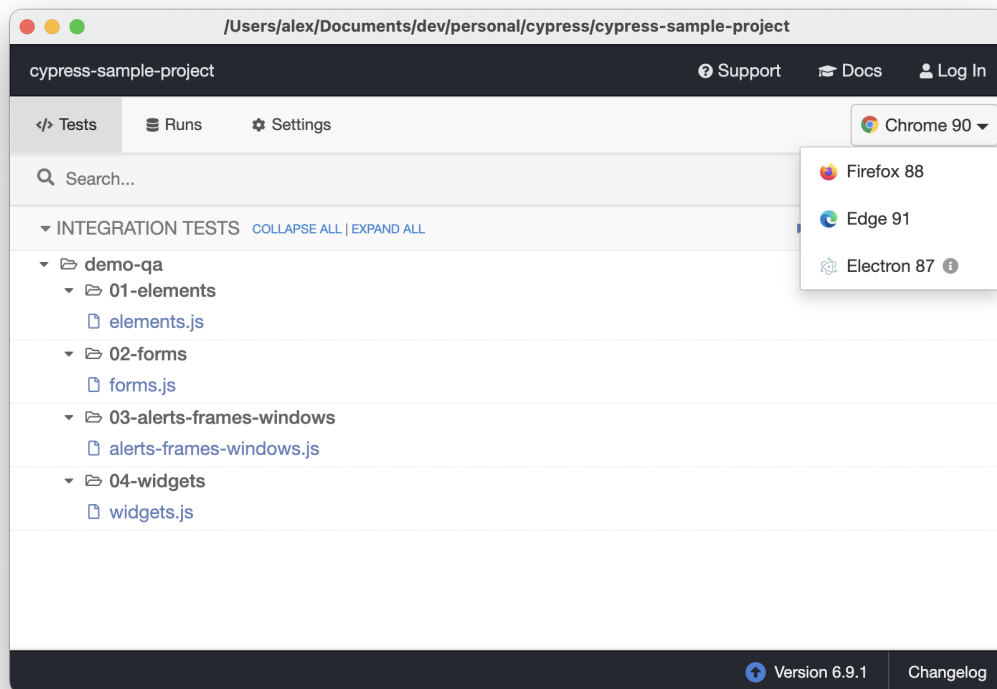


Figura 6. Test Runner com a lista de navegadores suportados.

O Cypress atualmente suporta o Firefox e navegadores da família Chrome (incluindo Edge e Electron).

3.4.4. Executando os testes

Após selecionar o navegador desejado no Test Runner, basta clicar em um dos arquivos de especificação de casos de teste (*specs*) terminados em .js que uma nova janela do navegador irá abrir e a execução do teste irá acontecer automaticamente como visto na Figura 7. O Test Runner embutido permite ao usuário verificar o tempo de execução do teste, navegar entre os estados da aplicação durante a execução (uma espécie de linha do tempo), reexecutar o teste, inspecionar elementos com a ajuda de seletor que permite copiar os elementos inspecionados já na sintaxe que será utilizada no código, entre outras funções bastante valiosas para o desenvolvedor de scripts de testes.

3.5. Configuração do ambiente do framework Selenium

Para a realização do experimento com o Selenium, as seguintes tecnologias foram utilizadas:

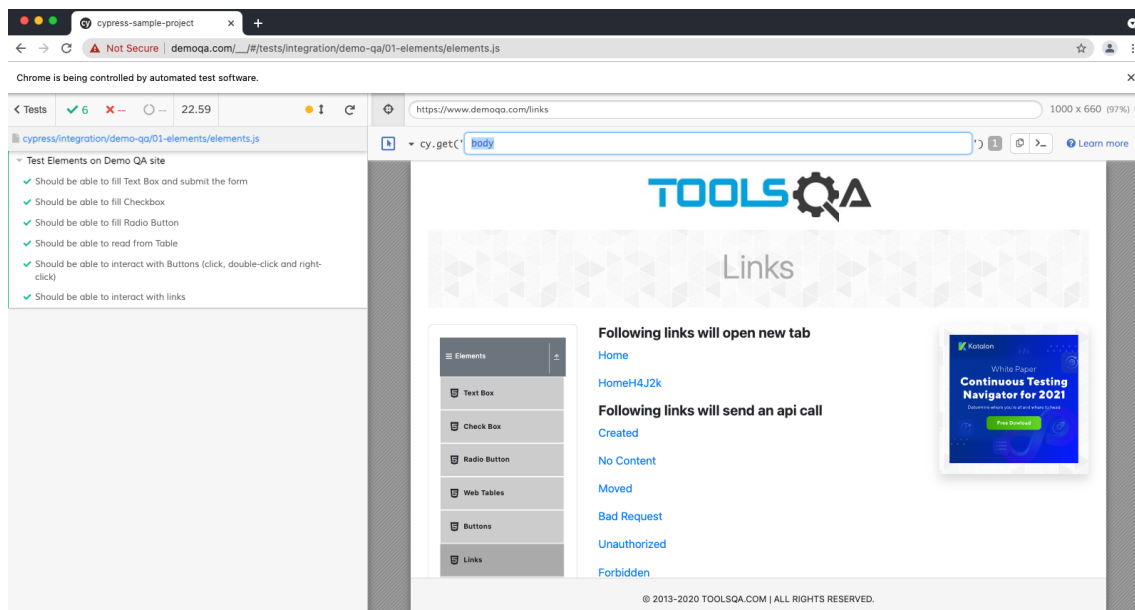


Figura 7. Navegador contendo Test Runner embutido durante a execução de um caso de teste no Cypress.

- IntelliJ IDEA CE - IDE utilizada para auxiliar o desenvolvimento;
- Maven 8 - Ferramenta utilizada para gerenciar dependências do projeto;
- Selenium 3.141.59;
- Java 1.8;
- JUnit 4.13-beta-1;

3.5.1. Instalação

Instale o ambiente Java seguindo o passo a passo a seguir:

1. Baixe IntelliJ IDEA CE disponível no website <https://www.jetbrains.com/> ;
2. Crie um novo projeto Maven;
3. Adicione o arquivo pom.xml com o seguinte conteúdo;

```

1
2     <?xml version="1.0" encoding="UTF-8"?>
3 <project xmlns="http://maven.apache.org/POM/4.0.0"
4         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
6         http://maven.apache.org/xsd/maven-4.0.0.xsd">
7     <modelVersion>4.0.0</modelVersion>
8
9     <groupId>org.example</groupId>
10    <artifactId>tcc_cesar</artifactId>
11    <version>1.0-SNAPSHOT</version>
12
13    <properties>
14        <java.version>1.8</java.version>
15        <junit.version>4.12</junit.version>
16        <cucumber.version>6.1.1</cucumber.version>

```

```

16     <selenium.version>3.141.59</selenium.version>
17     <maven.compiler.source>8</maven.compiler.source>
18     <maven.compiler.target>8</maven.compiler.target>
19 </properties>
20
21 <dependencies>
22     <dependency>
23         <groupId>org.seleniumhq.selenium</groupId>
24         <artifactId>selenium-java</artifactId>
25         <version>${selenium.version}</version>
26     </dependency>
27
28     <dependency>
29         <groupId>junit</groupId>
30         <artifactId>junit</artifactId>
31         <version>${junit.version}</version>
32         <scope>test</scope>
33     </dependency>
34
35     <dependency>
36         <groupId>junit</groupId>
37         <artifactId>junit</artifactId>
38         <version>4.13-beta-1</version>
39         <scope>compile</scope>
40     </dependency>
41     <dependency>
42         <groupId>org.junit.jupiter</groupId>
43         <artifactId>junit-jupiter</artifactId>
44         <version>RELEASE</version>
45         <scope>test</scope>
46     </dependency>
47 </dependencies>
48
49 </project>

```

4. Execute o processo de construção do *maven* para baixar e instalar todas as dependências
5. Baseado no seu navegador local, baixe e adicione o driver específico na raiz do projeto para que o script possa ser executado

Após seguir os passos citados da configuração do *maven*, o projeto está totalmente configurado com o ambiente necessário para desenvolver os scripts de teste.

3.6. Execução

Neste tópico vamos falar de algumas particularidades observadas na criação e execução dos scripts nos diferentes frameworks.

A Tabela 3 mostra um resumo das principais diferenças entre Cypress e Selenium encontradas na execução do experimento. As subseções a seguir mostram mais detalhes a respeito dessas diferenças.

Tabela 3. Resumo das diferenças encontradas entre Cypress e Selenium

Característica	Selenium	Cypress
Simplicidade	Necessidade de chamar bibliotecas adicionais para fazer asserts.	Métodos de assert nativos.
Chamadas HTTP	Selenium não contém funcionalidades nativas de validação de requisições HTTP.	É possível validar requisições HTTP.
Uso de <i>Wait</i>	Esperas precisam ser explicitamente definidas.	Comandos esperam automaticamente um estado de ação.
Métodos Javascript	É preciso saber exatamente o código Javascript para poder executá-lo.	Possui métodos nativos que auxiliam o desenvolvimento.
Múltiplas abas	Possui suporte a múltiplas abas e janelas.	Não possui suporte.
Tempo de execução	Scripts executados em 110.443s.	Scripts executados em 91.823s.
Linhas de código	Scripts desenvolvidos em 1512 linhas.	Scripts desenvolvidos em 569 linhas.
Documentação	Site oficial com diversos exemplos nas linguagens suportadas, 87K perguntas no stackoverflow e comunidade ampla.	Site oficial com diversos exemplos e vídeos, 4.4K perguntas no stackoverflow, comunidade em crescimento.
Artefatos de teste	Não tem suporte por padrão, precisa do desenvolvimento de bibliotecas adicionais.	Grava por padrão vídeos da execução dos testes.
Suporte à execução dos testes	Não tem.	Possui uma interface para execução local (Test Runner) e um serviço web (Dashboard) que estende as funcionalidades do Test Runner provendo suporte a <i>CI</i> e outras métricas de execução relevantes.

3.6.1. Simplicidade na criação dos scripts e chamada de métodos

Podemos observar que o Cypress possui diversos métodos nativos que auxiliam no desenvolvimento do script de uma maneira mais direta e simples. Além disso, eles podem

ser combinados facilmente, auxiliando assim na facilidade da implementação. Por outro lado, fez-se necessária a utilização de métodos auxiliares no Selenium para atingir o mesmo resultado.

Enquanto o Cypress já possui a possibilidade de fazer asserts sem a adição de nenhuma biblioteca, o Selenium necessita do JUnit para fazer validações. Portanto, mais métodos são utilizados.

Tomemos como exemplo o cenário "Verificar a interação com abas". Enquanto no Cypress a ação de achar o elemento na página e fazer diversas validações podem ser atingidas com somente uma linha de código, possibilitando se fazer um assert diretamente após a chamada de encontrar o elemento, o Selenium sendo utilizado em Java não nos permite essa ação.

Já que o Selenium não possui métodos de validação, faz-se necessário a chamada do JUnit para conseguir fazer assertivas, aumentando assim a quantidade métodos chamados para se atingir o mesmo resultado, como ilustram a Figura 8 e a Figura 9:

```
113 cy.xpath("//a[@id='demo-tab-what']").should('have.text', 'What').should('have.attr', 'aria-selected', 'true').and('have.class', 'nav-link active');
114 cy.xpath("//a[@id='demo-tab-origin']").should('have.attr', 'aria-selected', 'false').and('not.have.class', 'nav-link active');
115 cy.xpath("//a[@id='demo-tab-use']").should('have.text', 'Use').should('have.attr', 'aria-selected', 'false').and('not.have.class', 'nav-link active');
116 cy.xpath("//a[@id='demo-tab-more']").should('have.text', 'More').should('have.attr', 'aria-selected', 'false').should('have.class', 'nav-link disabled');
117 cy.xpath("//div[@id='demo-tabpane-what']").should('contains.text', 'Lorem Ipsum is simply dummy text of the printing and typesetting industry. ');
118 cy.xpath("//div[@id='demo-tabpane-what']").should('have.attr', 'aria-hidden', 'false').and('have.class', 'active show');
119 cy.xpath("//div[@id='demo-tabpane-origin']").should('have.attr', 'aria-hidden', 'true').and('not.have.class', 'active show');
120 cy.xpath("//div[@id='demo-tabpane-use']").should('have.attr', 'aria-hidden', 'true').and('not.have.class', 'active show');
121 cy.xpath("//div[@id='demo-tabpane-more']").should('have.attr', 'aria-hidden', 'true').and('not.have.class', 'active show');
```

Figura 8. Funções que encontram elementos e validam atributos podem ser atingidos mais facilmente, aumentando a simplicidade da escrita de scripts com Cypress

3.6.2. Validação de chamadas HTTP

Uma das vantagens do Cypress em relação ao Selenium é a possibilidade de realizar validações de chamadas de HTTP. Além de realizar ações na página, o Cypress também é capaz de obter elementos de requisições http, tais como o status code e o body, como pode ser visto na Figura 10.

O Selenium não possui esse tipo de funcionalidade. Para atingir essa funcionalidade em conjunto com Selenium, uma das possibilidades seria criar um proxy para interceptar as requisições HTTP enquanto o script escrito em Selenium percorre a tela.

3.6.3. Uso de "Wait"

Geralmente, quando executamos um script que percorre os elementos da tela, precisamos garantir que o elemento esteja pronto para sofrer interação. Por isso, precisamos esperar por esse estado. Caso o elemento não esteja pronto para receber uma ação, o script irá falhar.

O Selenium possui três tipos de *Waits*. O implícito, explícito e fluente. Falando a respeito dos dois primeiros, o *wait* implícito é usado para definir um tempo específico para esperar por determinado elemento. O *wait* explícito define uma condição específica

```

159     driver.get(Constants.TOOLS_QA_WIDGETS_TABS_URL);
160
161     Assert.assertEquals("What", widgetsTabsPage.getDemoTabWhatAttributes().get(0));
162     Assert.assertEquals("true", widgetsTabsPage.getDemoTabWhatAttributes().get(1));
163     Assert.assertEquals("nav-item nav-link active", widgetsTabsPage.getDemoTabWhatAttributes().get(2));
164
165     Assert.assertEquals("Origin", widgetsTabsPage.getDemoTabOriginAttributes().get(0));
166     Assert.assertEquals("false", widgetsTabsPage.getDemoTabOriginAttributes().get(1));
167     Assert.assertNotEquals("nav-link active", widgetsTabsPage.getDemoTabOriginAttributes().get(2));
168
169     Assert.assertEquals("Use", widgetsTabsPage.getDemoTabUseAttributes().get(0));
170     Assert.assertEquals("false", widgetsTabsPage.getDemoTabUseAttributes().get(1));
171     Assert.assertNotEquals("nav-link active", widgetsTabsPage.getDemoTabUseAttributes().get(2));
172
173     Assert.assertEquals("More", widgetsTabsPage.getDemoTabMoreAttributes().get(0));
174     Assert.assertEquals("false", widgetsTabsPage.getDemoTabMoreAttributes().get(1));
175     Assert.assertNotEquals("nav-link active", widgetsTabsPage.getDemoTabMoreAttributes().get(2));
176
177     Assert.assertTrue(widgetsTabsPage.getDemoTabPanelWhatText()
178         .contains("Lorem Ipsum is simply dummy text of the printing and typesetting industry.));
179
180     Assert.assertEquals("false", widgetsTabsPage.getDemoTabPaneWhatAttributes().get(0));
181     Assert.assertEquals("fade tab-pane active show", widgetsTabsPage.getDemoTabPaneWhatAttributes().get(1));
182
183     Assert.assertEquals("true", widgetsTabsPage.getDemoTabPaneOriginAttributes().get(0));
184     Assert.assertNotEquals("active show", widgetsTabsPage.getDemoTabPaneOriginAttributes().get(1));
185
186     Assert.assertEquals("true", widgetsTabsPage.getDemoTabPaneUseAttributes().get(0));
187     Assert.assertNotEquals("active show", widgetsTabsPage.getDemoTabPaneUseAttributes().get(1));
188
189     Assert.assertEquals("true", widgetsTabsPage.getDemoTabPaneMoreAttributes().get(0));
190     Assert.assertNotEquals("active show", widgetsTabsPage.getDemoTabPaneMoreAttributes().get(1));

```

Figura 9. Mesmo cenário da Figura 8, mas em escrito em Selenium. Faz-se necessária a utilização de Javascript e métodos auxiliares para atingir o mesmo resultado

```

133     cy.intercept('GET', '/created').as('getCreated');
134     cy.xpath("//a[@id='created']").click();
135     cy.xpath("//p[@id='linkResponse']").should('include.text', 'Link has responded with status 201 and status text Created');
136     cy.wait('@getCreated').its('response.statusCode').should('eq', 201);
137
138     cy.intercept('GET', '/no-content').as('getNoContent');
139     cy.xpath("//a[@id='no-content']").click();
140     cy.xpath("//p[@id='linkResponse']").should('include.text', 'Link has responded with status 204 and status text No Content');
141     cy.wait('@getNoContent').its('response.statusCode').should('eq', 204);

```

Figura 10. Com o Cypress é possível validar requisições HTTP. Isso não é possível com Selenium.

para a espera em conjunto com o tempo máximo. Com o *wait* explícito, o script espera até que a condição seja atingida ou que o tempo máximo termine. *Waits* em Selenium devem ser codificados e devem estar presentes no código para evitar erros, conforme mostra a Figura 11.

O Cypress não possui tal necessidade de explicitar a espera no script. De acordo com a documentação oficial do Cypress [Cypress.io 2019c], comandos de ação esperam automaticamente o elemento ficar num estado em que ele pode ser manipulado. Isso contribui para a simplicidade e facilidade para a escrita do código.

```

27     public void waitAndClick(By locator) {
28         wait.until(ExpectedConditions.elementToBeClickable(locator));
29         driver.findElement(locator).click();
30     }
31
32     public void waitAndCommandClick(By locator) {
33         wait.until(ExpectedConditions.elementToBeClickable(locator));
34         WebElement element = driver.findElement(locator);
35         action.keyDown(Keys.COMMAND).click(element).keyUp(Keys.COMMAND).perform();
36     }
37
38     public void waitAndClear(By locator) {
39         wait.until(ExpectedConditions.elementToBeClickable(locator));
40         driver.findElement(locator).clear();
41     }
42
43     public void waitAndDoubleClick(By locator) {
44         wait.until(ExpectedConditions.elementToBeClickable(locator));
45         action.doubleClick(driver.findElement(locator)).perform();
46     }
47
48     public void waitAndRightClick(By locator) {
49         wait.until(ExpectedConditions.elementToBeClickable(locator));
50         action.contextClick(driver.findElement(locator)).perform();
51     }

```

Figura 11. Métodos de *wait* criados em Selenium para esperar elementos atingirem determinado estado para poderem ser manipulados.

3.6.4. Métodos Javascript e definição de atributos dos elementos

Em algumas situações, para se atingir determinado estado na tela, faz-se necessário o uso de métodos do Javascript ou definir atributos específicos de elementos na tela.

Não existem métodos pré-definidos em Selenium que nos ajudam a atingir certos estados ou definir atributos de um elemento. Por exemplo, caso queiramos fazer um *scroll* numa tela, não há um método específico chamado *scroll()* ou *scrollTo()*. Para conseguir executar ações com a barra de rolagem, deve-se chamar a biblioteca *JavascriptExecutor*. A partir daí, pode-se chamar qualquer método Javascript. Porém, o desenvolvedor deve saber exatamente como escrever o código Javascript, como mostra a Figura 12.

```

14     public void move() {
15         JavascriptExecutor js = (JavascriptExecutor) driver;
16
17         js.executeScript("arguments[0].value='80'", driver.findElement(firstElementSlider));
18         js.executeScript("arguments[0].setAttribute('style', '--value:80;')", driver.findElement(firstElementSlider));
19         js.executeScript("arguments[0].setAttribute('value', '80')", driver.findElement(firstElementSlider));
20         js.executeScript("arguments[0].setAttribute('style', 'left: calc(80% + -6px);')", driver.findElement(secondElementSlider));
21         js.executeScript("arguments[0].setAttribute('value', '80')", driver.findElement(thirdElementSlider));
22     }
23

```

Figura 12. Chamada de métodos auxiliares em Selenium para executar chamadas Javascript.

Por outro lado, o Cypress possui métodos específicos para esses tipos de ação, facilitando e simplificando o desenvolvimento de scripts. Citando o mesmo exemplo exibido na Figura 12, o Cypress possui métodos nativos chamados *scrollTo()* e *scrollIntoView()*. Não sendo necessário, portanto, a chamada de bibliotecas adicionais. Além disso, o Cypress possui o método *invoke()* onde dentre as várias funções, auxiliam na definição de atributos dos elementos, conforme podemos ver na Figura 13.

```
91     it("Should be able to interact with Slider Widgets", () => {
92
93         cy.visit("https://www.demoqa.com/slider");
94
95         cy.xpath("//input[@type='range']").invoke('val', 80).trigger('change');
96
97         cy.xpath("//input[@type='range']").invoke('attr', 'style', "--value:80;").trigger('change');
98
99         cy.xpath("//input[@type='range']").invoke('attr', 'value', "80").trigger('change');
100
101         cy.xpath("//div[contains(@class,'range-slider__tooltip--auto')]").invoke('attr', 'style', "left: calc(80% - 6px);");
102
103         cy.xpath("//input[contains(@id,'sliderValue')]").invoke('attr', 'value', '80').trigger('change');
104
105         cy.xpath("//input[contains(@id,'sliderValue')]").should('have.attr', 'value', '80');
106
107     });
```

Figura 13. Cypress possui métodos nativos que auxiliam na realização de ações Javascript.

3.6.5. Suporte a múltiplas abas e janelas

Uma vantagem na utilização do Selenium é o suporte a múltiplas abas e janelas. Alguns links em certos sistemas são abertos em abas ou janelas diferentes. Isso faz parte da funcionalidade e deve ser testado. O Selenium possui esse tipo de suporte. Um dos métodos da classe WebDriver, chamado *switchTo()*, permite que o foco do script seja alterado para uma aba ou para uma nova janela, conforme mostra a Figura 14.

```
78     public void switchToTab() {
79         ArrayList<String> tabs = new ArrayList<String>(driver.getWindowHandles());
80         driver.switchTo().window(tabs.get(1));
81     }
82
83     public void closeTabAndReturn() {
84         ArrayList<String> tabs = new ArrayList<String>(driver.getWindowHandles());
85         driver.close();
86         driver.switchTo().window(tabs.get(0));
87     }
```

Figura 14. Métodos para manipular abas utilizando o Selenium.

Porém, isso não é possível com o Cypress. O script desenvolvido em Cypress é executado diretamente no navegador. Por causa disso, o framework fica impossibilitado de suportar abas múltiplas ou novas janelas. A falta de suporte a essa funcionalidade é uma escolha dos desenvolvedores do Cypress[Cypress.io 2019c].

3.6.6. Suporte à execução dos testes

Uma das funcionalidades de maior destaque do Cypress são o Test Runner e o seu Dashboard. Como mostrado nas Figura 6, o Test Runner tem uma interface gráfica bastante intuitiva e amigável que permite selecionar os scripts que se deseja executar, bem como o navegador onde a execução irá acontecer.

Além de executar os testes localmente através do Test Runner, o usuário tem a opção de usar o serviço do Cypress Dashboard que é um complemento opcional baseado na web para o Test Runner, ele fornece informações oportunas, simples e poderosas sobre todos os seus testes executados como visto na Figura 15, permitindo a paralelização automática e balanceamento de carga, possibilitando otimizar CI (*Continuous Integration*) e executar testes significativamente mais rápidos como descrito em sua documentação [Cypress.io 2019c].

Esse tipo de funcionalidade/interface tanto para seleção dos casos de testes a serem executados localmente (Test Runner) bem como de um dashboard na nuvem com recursos valiosos sobre histórico das execuções, resultados de execuções em ciclos de CI, e outras funcionalidades não existem no Selenium.

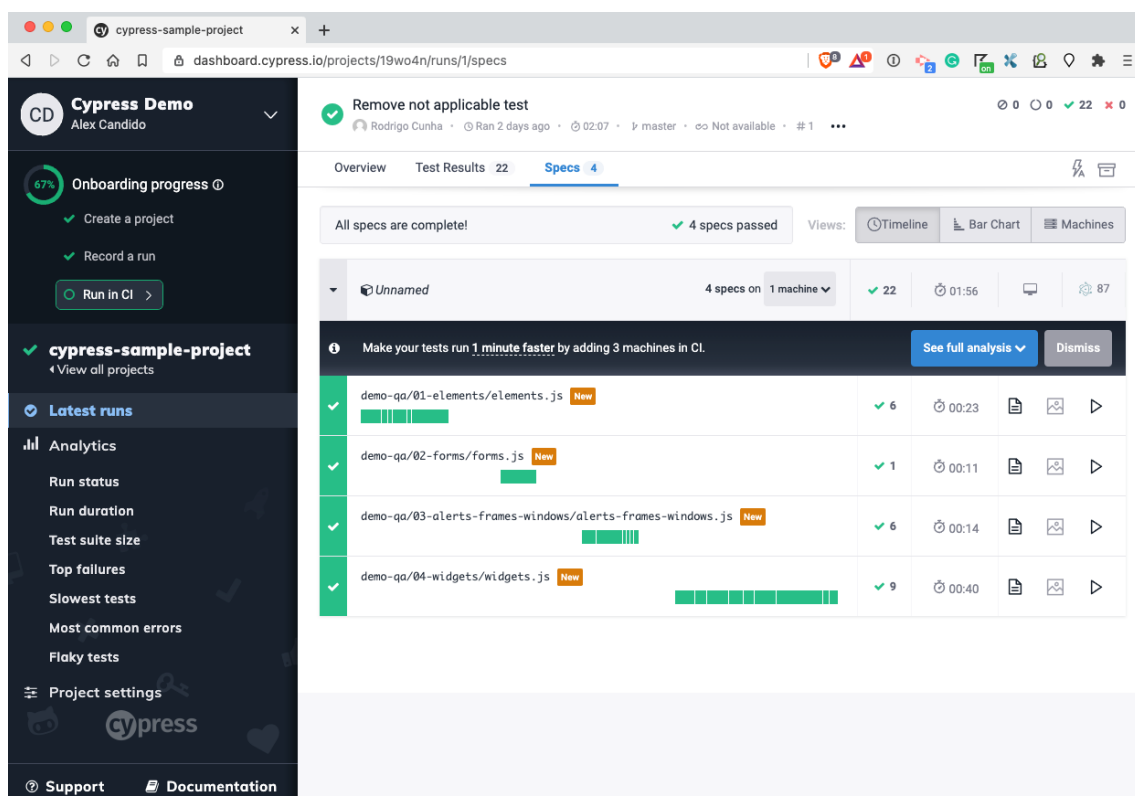


Figura 15. Cypress Dashboard.

Com paralelização automática e balanceamento de carga, você pode otimizar CI e executar testes significativamente mais rápidos.

3.6.7. Artefatos da execução dos testes

Uma das importantes características durante a automação de testes é a geração de artefatos que auxiliem a validação e comprovação dos resultados dos testes executados. O Cypress por padrão grava suas execuções em vídeos do formato “.mp4” que são armazenados na pasta “videos” localizada no diretório raiz do projeto. O Selenium não disponibiliza a gravação de vídeos ou mesmo a captura de screenshots durante a execução dos testes, sendo necessário o desenvolvimento ou integração dessa funcionalidade através de bibliotecas externas que possibilitam esse suporte.

3.6.8. Documentação

O framework do Selenium é uma das ferramentas de automação de testes mais antigas e populares disponíveis no mercado, possuindo assim uma vasta comunidade e ampla documentação disponível na internet, em pouco tempo de busca é possível encontrar exemplos de implementações e possíveis soluções para o tema desejado. Apesar de seu pouco tempo, o Cypress possui uma comunidade bastante atuante que também já possibilita encontrar soluções para dúvidas na internet. O site oficial do Cypress é a principal fonte de referências para o framework, ele é repleto de exemplos e tutoriais contendo inclusive vídeos gravados por seus mantenedores. O Selenium também possui em seu site oficial bastante conteúdo contendo exemplos de implementações de soluções em diferentes linguagens de programação suportadas pelo framework.

Ao procurarmos no *stackoverflow.com*, um dos principais fóruns de programação da internet, encontramos 87k perguntas registradas com a tag “selenium” e 4.4K perguntas registradas com a tag “cypress”.

3.7. Resultados

Alguns indicadores são necessários para medirmos o progresso da automação de testes, neste trabalho, definimos duas métricas mensuráveis que são o tempo de execução de cada script e a quantidade de linhas de código necessárias para realizar uma dada interação.

3.7.1. Tempo de Execução

A fim de evitar discrepâncias devido a divergência de ambientes de execução, o tempo de execução dos scripts foi aferido em um único equipamento com as seguintes configurações:

- **Computador (modelo):** MacBook Pro (16-inch, 2019);
- **Processador:** 2,3 GHz 8-Core Intel Core i9;
- **Memória:** 16 GB 2667 MHz DDR4;
- **Placa gráfica:** Intel UHD Graphics 630 1536 MB;
- **Sistema Operacional:** macOS Big Sur 11.3.1;
- **Navegador:** Chrome 90.

É possível notar que o Cypress teve um desempenho discretamente melhor em relação ao Selenium no que diz respeito ao tempo de execução. A Tabela 4 detalha o

tempo gasto em milissegundos para a execução de cada script e o tempo total gasto em segundos.

Tabela 4. Tempo de Execução em milissegundos (ms) de cada script. Total calculado em segundos (s).

Script	Selenium (ms)	Cypress (ms)
Script_01	5651 ms	1646 ms
Script_02	5553 ms	5793 ms
Script_03	2905 ms	3157 ms
Script_04	818 ms	995 ms
Script_05	952 ms	1158 ms
Script_06	942 ms	997 ms
Script_07	14371 ms	11854 ms
Script_08	5162 ms	1271 ms
Script_09	5587 ms	5401 ms
Script_10	7850 ms	N/A
Script_11	4158 ms	1231 ms
Script_12	4050 ms	961 ms
Script_13	4773 ms	1884 ms
Script_14	5710 ms	4025 ms
Script_15	4306 ms	3930 ms
Script_16	6203 ms	5270 ms
Script_17	4164 ms	5851 ms
Script_18	7710 ms	11566 ms
Script_19	7560 ms	8741 ms
Script_20	1603 ms	1486 ms
Script_21	7197 ms	7484 ms
Script_22	5990 ms	4531 ms
Script_23	5078 ms	2591 ms
TOTAL (em segundos)	110.443 s	91.823 s

3.7.2. Linhas de código

No que diz respeito a quantidade de linhas de código para desenvolver a mesma solução, o Cypress teve uma vantagem significativa perante o Selenium, tendo quase 2/3 a menos de linhas de código. Apenas em um dos scripts (Script_19) foi possível alcançar a mesma solução com quantidade inferior de linhas usando o Selenium. A Tabela 5 detalha o a quantidade de linhas desenvolvidas para a realizar a mesma interação/comportamento.

Tabela 5. Linhas de código para desenvolver cada script.

Script	Selenium	Cypress
Script_01	170	36
Script_02	24	12
Script_03	25	10
Script_04	29	12
Script_05	27	12
Script_06	27	12
Script_07	96	32
Script_08	43	18
Script_09	45	24
Script_10	36	N/A
Script_11	36	12
Script_12	36	10
Script_13	89	17
Script_14	100	26
Script_15	79	18
Script_16	53	41
Script_17	91	46
Script_18	130	67
Script_19	52	61
Script_20	32	18
Script_21	84	35
Script_22	114	26
Script_23	97	24
TOTAL	1512	569

4. Conclusão

Este trabalho teve como objetivo fazer uma análise técnica entre os frameworks de automação de testes Selenium e Cypress. Foi realizado um experimento por meio do desenvolvimento de scripts de testes para cada framework utilizando uma aplicação web. A intenção deste trabalho é apresentar informações relevantes que possam ajudar profissionais da área na tomada de decisão.

O Cypress demonstrou ser uma ferramenta bastante promissora, é possível criar testes com bem menos linhas de código em relação ao Selenium. O Cypress também possui métodos nativos que auxiliam na realização de diferentes ações, enquanto que no Selenium faz-se necessário importar bibliotecas externas. O Cypress teve uma performance de duração de execução dos testes levemente melhor que o Selenium. Uma das diferenças que pode ser considerada como uma limitação percebida no Cypress, foi o não suporte a interação com múltiplas abas e janelas, segundo a documentação oficial do framework, ela sugere que o script seja desenvolvido de maneira a abrir a URL de destino da aba/janela e então realizar as validações desejadas, sendo desse modo incapaz de validar se uma nova aba ou janela foi aberta após uma determinada interação como um simples clique em um botão ou hyperlink. Esse tipo de interação é suportado sem maiores dificuldades no Selenium.

Para trabalhos futuros, podemos destacar a avaliação de outros frameworks de automação de testes open source voltados para a automação de aplicações web modernas, dentre as opções temos os seguintes frameworks como boas opções de estudo e avaliação: Robot Framework [Robot 2021], o Playwright [Playwright 2021], Testcafe [TestCafe 2021], Nightwatch [Nightwatchjs 2021] e Puppeteer [Puppeteer 2021]. Essas aplicações tem tido uma significativa adoção e quantidade de estrelas em seus projetos no GitHub. Outra possibilidade seria a estruturação do projeto desenvolvido utilizando padrões de projetos como o *Page Objects*, que é bastante conhecido e consolidado, e do padrão *Actions*, esse último sendo uma das recomendações do próprio Cypress. Vale também destacar como trabalho futuro a avaliação dos frameworks em projetos reais, de maior complexidade e escala.

Referências

- Berçam, R. (2019). Um overview sobre cypress.io — framework de automação de testes end-to-end. <https://medium.com/@faelbercam/um-overview-sobre-cypress-io-framework-de-automa>
- Bulla, A. and Brunda, S. (2016). A study- automation technique using selenium web driver. *International Journal of Research*, 3:467–470.
- Cowan, P. (2019). Cypress.io: The selenium killer. <https://blog.logrocket.com/cypress-io-the-selenium-killer/>.
- Cypress (2021). How it works. <https://www.cypress.io/how-it-works>.
- Cypress.io (2019a). Cypress.io framework. <https://www.cypress.io/>.
- Cypress.io (2019b). Stop using page objects and start using app actions. <https://www.cypress.io/blog/2019/01/03/stop-using-page-objects-and-start-using-app-actions/>.
- Cypress.io (2019c). Why cypress? <https://docs.cypress.io/guides/overview/why-cypress.html>.
- Gorej, V. (2018). Page object pattern in javascript. <https://www.linkedin.com/pulse/page-object-pattern-javascript-vladim>
- Maldonado, J. C., Rocha, A., and Weber, K. (2001). Qualidade de software: teoria e prática. *São Paulo*.
- Mane, M., Bhadekar, G., and Salunkhe, S. (2016). Text and keyword driven automation testing using selenium web driver.
- Mobaraya, F. and Ali, S. (2019). Technical analysis of selenium and cypress as functional automation framework for modern web application testing. pages 27–46.
- Molinari, L. (2010). Inovação e automação de testes de software. *São Paulo: Editora Érica*.
- Nightwatchjs (2021). Nightwatch.js: End-to-end testing, the easy way. <https://nightwatchjs.org/>.
- Playwright (2021). Playwright enables reliable end-to-end testing for modern web apps. <https://playwright.dev/>.

- Pressman, R. S. (2001). Software engineering: a practitioner's approach.
- Puppeteer (2021). Puppeteer. <https://pptr.dev/>.
- Robot (2021). Robot framework. <https://robotframework.org/>.
- Sadiq, S. (2021). What is selenium webdriver? [complete guide]. <https://hackr.io/blog/what-is-selenium-webdriver>.
- Selenium (2021a). The selenium browser automation project. <https://www.selenium.dev/documentation/en/>.
- Selenium (2021b). Selenium web browser automation. <http://www.seleniumhq.org/>.
- Stewart, S. (2015). Pageobjects. <https://github.com/SeleniumHQ/selenium/wiki/PageObjects>.
- TestCafe (2021). Testcafe: End-to-end testing, simplified. <https://testcafe.io/>.
- Thoughtworks (2020). Adote: Cypress. <https://assets.thoughtworks.com/assets/technology-radar-vol-22-pt.pdf>.
- Vila, E., Nedeltcheva, G., and Todorova, D. (2017). Automation testing framework for web applications with selenium webdriver: Opportunities and threats. pages 144–150.