Implementando Testes Automatizados para o Sistema Web Speedy: Um Relato de Experiência

Allain Prost Lucena Campos[1]

Departamento de Ciências Exatas (DCX) –
Universidade Federal da Paraíba (UFPB) - Campus IV
Cep 58297-000 – Rio Tinto – PB – Brasil

allain.prost@dcx.ufpb.br

Abstract. *The construction and implementation of automated tests can become a complex task, especially when the source code is not provided and when we do not know which testing tool to use in this situation.\. Furthermore, another big challenge is the construction of automated test scripts that are easy to maintain. In this context, we present in this work an experience report regarding the process of implementing automated tests for the Speedy application using the Cypress tool. Therefore,, this article aims to support developers that need to implement automated tests for Web systems using the Cypress tool.. In order to do that, we present the process of creating and implementing the test scripts, the difficulties encountered and the lessons learned in this process were described.*

Resumo. *A construção e implementação de testes automatizados pode se tornar uma tarefa complexa, especialmente quando o código do sistema a ser testado não é disponibilizado e quando não se sabe que ferramenta utilizar para testar nessa situação. Além disso, um outro grande desafio é a construção de scripts de testes automatizados que sejam fáceis de manter. Neste contexto, neste trabalho apresentamos um relato de experiência relativo ao processo de implementar testes automatizados para a aplicação Speedy utilizando a ferramenta Cypress. Sendo assim, este artigo visa apoiar outros desenvolvedores que desejem implementar testes automáticos em sistemas Web utilizando a ferramenta Cypress. Para isso, apresentamos o processo de criação e implementação dos scripts de testes, as dificuldades encontradas e as lições aprendidas nesse processo.*

1. Introdução

Durante a evolução da computação e, consequentemente dos softwares, era bastante comum que os softwares criados fossem entregues aos clientes com a presença de defeitos, que, subsequentemente, causavam falhas [ISTQB, 2018]. Contudo, com o uso

[1] Trabalho de conclusão de curso, sob orientação da professora Ayla Débora Dantas de Souza Rebouças submetido ao Curso de Bacharelado de Sistemas de Informação do Centro de Ciências Aplicadas e Educação (CCAE) da Universidade Federal da Paraíba, como parte dos requisitos necessários para obtenção do grau de BACHAREL EM SISTEMAS DE INFORMAÇÃO.

de técnicas de testes apropriadas e mão de obra qualificada, esses defeitos podem ser reduzidos drasticamente.

Segundo Pressman (2011) o teste de software é uma das principais atividades realizadas para melhorar a qualidade de um produto ou serviço em desenvolvimento. Seu principal objetivo é revelar a presença de defeitos o mais cedo possível no desenvolvimento de softwares, buscando a minimização do custo das correções do mesmo.

Sendo assim, é notória a importância e a necessidade de se realizar testes de softwares e, se possível automatizá-los, uma vez que este processo de automatização traz um *feedback* mais rápido e mais eficiência, mas o processo de automação nem sempre é tão simples e por isso requer bastante atenção. Além disso, em muitos casos requer conhecimento técnico de programação voltado à produção de *scripts* automatizados de teste.

Para manter os *scripts* bem estruturados, é desejável que seu código seja fácil de se entender e manter caso haja alterações em componentes do sistema. Para isso, é importante sempre buscar refatorar os *scripts* de teste, pois eles em si contribuem para que refatorações no sistema possam ser feitas de forma mais segura. Segundo Martin Fowler (2019), refatoração é “o processo de mudar a estrutura interna do software sem alterar seu comportamento”. E para garantir que isso aconteça sem introduzir defeitos, Ebanhesaten et al. (2013) fala que "testar é essencial para assegurar que a refatoração não mudou o comportamento do software”. Considerando isso, o objetivo deste trabalho é apoiar pessoas que desejem implementar testes automáticos de sistemas Web através da apresentação de um relato de experiência sobre a construção de testes automáticos para um sistema Web intitulado Speedy utilizando a ferramenta Cypress, que é uma ferramenta de teste de *end-to-end* para sistemas Web. Espera-se com este trabalho descrever e relatar os passos e os desafios de construir testes automáticos com essa ferramenta durante o desenvolvimento de softwares.

Alguns objetivos específicos esperados para este trabalho são os seguintes: a criação e implementação de casos de testes, a utilização de bibliotecas para gerar dados fictícios, qual assertiva utilizar durante o desenvolvimento dos scripts e a identificação das principais lições aprendidas ao criar testes automatizados usando o Cypress.

As demais seções deste artigo estão organizadas conforme descrito a seguir: a Seção 2 apresenta a fundamentação teórica e conceitos sobre validação, verificação e teste de software, qualidade, automação de testes e ferramentas que foram utilizadas durante o desenvolvimento dos *scripts* e do artigo; a Seção 3 apresenta a metodologia utilizada na realização deste trabalho; a Seção 4 apresenta e descreve os resultados obtidos com a experiência; e, por fim, a Seção 5 apresenta a conclusão, as considerações finais e propostas de trabalhos futuros.

2. Fundamentação Teórica

Nesta seção serão apresentados conceitos relacionados a este trabalho com maior riqueza de detalhes.

2.1 Qualidade de software

A grande maioria dos erros encontrados nos softwares são causados pela falha humana durante o desenvolvimento [Delamaro, 2016].

Sendo assim, para garantir um produto de qualidade, e para evitar que os problemas e erros inseridos nas etapas de desenvolvimento não perdurem até o lançamento do software, são necessárias várias atividades, coletivamente chamadas de verificação, validação e teste de software, ou "VV&T", que possuem a finalidade de garantir a qualidade e a conformidade com os requisitos passados pelos clientes para a construção do software [Delamaro 2016]. A verificação assegura consistência, completude e corretude do produto em cada fase e entre fases consecutivas da vida do software, já a validação assegura que o produto final corresponde aos requisitos passados pelo cliente e o teste de software examina o comportamento do produto através de sua execução. Vale ressaltar que as atividades "VV&T" são fundamentais e devem ocorrer durante todo o processo de desenvolvimento de software.

Segundo David Garvin, da Harvard Business School, "qualidade é um conceito complexo e multifacetado" [Engenharia de Software, 2016]. Este autor afirma que "a qualidade depende do ponto de vista sob o qual o produto é analisado e por quem é analisado". Na visão do usuário ou cliente, se aquele determinado produto ou serviço atender às expectativas e às necessidades, ele apresenta qualidade. Já o fabricante determina a qualidade através das especificações. Caso tais especificações sejam atendidas, ele também apresenta qualidade.

Segundo Pressman (2011), no desenvolvimento de software, a qualidade de um projeto engloba o grau de atendimento às funções e características especificadas no modelo de requisitos. A qualidade de conformidade focaliza o grau em que a implementação segue o projeto e que o sistema resultante atende às suas necessidades e às metas de desempenho.

De modo geral, um software de qualidade é um produto que atende às necessidades e expectativas dos clientes e usuários, fornecendo-lhes uma boa experiência de uso. Ao mesmo tempo, também é importante que o software tenha qualidade para seus desenvolvedores, de forma que seja fácil realizar sua manutenção.

Portanto, para garantir a qualidade de *software* é necessário analisar todos os aspectos, desde o levantamento de requisitos e arquitetura até a codificação, integração de componentes e validação de funcionalidades. Para se fazer a validação das funcionalidades, é interessante tomar por base casos de teste, que, segundo o ISTQB (2021), é "um conjunto de valores de entrada, pré-condições de execução, resultados esperados e pós-condições de execução, desenvolvido para um objetivo particular ou condição de testes." Logo, o caso de teste é de extrema importância, já que serve de base para os testadores executarem os testes de forma manual e construírem os *scripts* de testes automatizados a partir das ações e resultados esperados.

2.2 Automação de testes

Segundo Maldonado (2018) "existem tarefas que são executadas de forma mais eficiente pelo computador do que pelo ser humano". Sendo assim, para evitar atividades manuais repetitivas relativas a testes, é fundamental a utilização de tecnologias que automatizam o processo. Essas tecnologias têm o intuito de facilitar e agilizar o processo de teste.

Neste caso, a automatização de testes é a utilização de tecnologias e softwares para controlar a execução de testes, aplicando regras e comparando resultados esperados com os resultados obtidos. Há vários tipos de testes automatizados. Neste trabalho, o foco será dado aos testes de caixa preta. O teste de caixa preta recebe esse nome porque o foco é apenas nas entradas e saídas do sistema, sem se preocupar com a estrutura

interna do programa. Testes automatizados, que podem ou não ser de caixa preta, servem para apoiar no processo de manutenção do sistema e algumas vezes podem ser utilizados como testes de regressão, para evitar que novos defeitos sejam inseridos durante o acréscimo ou ajuste de funcionalidades. Segundo Pressman (2011), o teste de regressão "é a reexecução de testes que já foram executados como forma de assegurar que alterações feitas não propaguem efeitos colaterais indesejáveis". Sendo assim, os conjuntos de testes de regressão são executados diversas vezes e isso geralmente leva tempo. Portanto, o teste de regressão é um forte candidato à automatização.

Dentre as ferramentas e frameworks para automatização de testes, a ferramenta escolhida para este trabalho foi o Cypress. O motivo de escolha desta ferramenta se dá por causa da familiaridade e experiência por parte do autor e por conter uma funcionalidade chamada *Selector Playground,* responsável por mapear elementos visuais de forma rápida, conforme destacado posteriormente.

3. Metodologia

O grande desafio deste trabalho foi construir testes como forma de garantir a qualidade do sistema *Speedy.* Para isso, ocorreu um processo de planejamento e construção de *scripts* de testes automatizados com a ferramenta Cypress e ao longo do processo se documentou a experiência para apoiar outros desenvolvedores de testes.

Inicialmente, o primeiro passo da metodologia adotada neste trabalho foi definir e documentar os casos de teste a serem desenvolvidos. Para isso, foi pensado e elaborado um modelo que será apresentado na Seção 4.1, com exemplos de casos de testes para o sistema. Após o planejamento dos testes, iniciou-se a construção dos *scripts* de testes.

O desenvolvimento dos *scripts* é algo que requer bastante atenção, já que o analista deve observar se aquele determinado caso de testes deve ou não ser automatizado. Os especialistas Daniel Amorim e Hugo Vares (2015) afirmam que:

> Geralmente, nós automatizamos para evitar trabalho manual repetido, obter um feedback mais rápido, economizar tempo executando testes repetidamente, e assegurar que estamos sempre executando testes de forma consistente com as mesmas precondições e expectativas.

Partindo desse pressuposto, o passo seguinte foi criar um conjunto de testes automatizado bem definido, para focar nas áreas críticas do sistema Speedy. Para alcançar este objetivo foi necessário realizar algumas reuniões informais com os *stakeholders* da aplicação, sendo eles os diretores e desenvolvedores da aplicação, para identificar quais regras de negócios possuem mais relevância e quais delas estavam sendo mais afetadas por possíveis *bugs* no sistema. Sendo assim, buscamos repassar um feedback o mais rápido possível para os desenvolvedores do sistema, para que fosse possível corrigir os bugs encontrados, já que a aplicação estava em ambiente de produção.

Posteriormente, foram realizados os estudos sobre qualidade, testes automatizados, sistema Speedy e Cypress para obtermos um melhor entendimento sobre o sistema e sobre o processo de automatização de testes. Durante esse mesmo período foi realizada a escrita de parte do referencial teórico deste trabalho. Foi então, posteriormente, iniciado o processo de testes com o desenvolvimento dos *scripts*. Por fim, foram documentados os passos necessários para este processo e foram realizados ajustes tanto no artigo quanto nos *scripts* de testes.

4. Testando o Speedy com o Cypress

Nessa seção será apresentado o Sistema Speedy, a documentação dos casos de testes e relatado o processo de construção de testes End-to-End utilizando a ferramenta Cypress, com o objetivo de apoiar outros desenvolvedores que precisam desenvolver testes desse tipo em sistemas Web.

4.1 Compreendendo o Sistema Speedy

O primeiro passo para iniciar a automatização dos testes caixa preta de um sistema Web é compreender como funciona o sistema a ser testado. Para isso, nessa seção apresentaremos o sistema Speedy[2].

O Speedy é um sistema que visa realizar a integração entre estabelecimentos do ramo alimentício e entregadores. Sendo assim, quando um estabelecimento necessita realizar uma entrega, o sistema dispara um chamado para os entregadores, e quando esse chamado é atendido, o entregador deve comparecer até o estabelecimento para coletar o pedido e realizar a entrega ao cliente.

O sistema surgiu para solucionar o problema de altas taxas de entregas que são cobradas por várias plataformas de *delivery*. O Speedy tem como objetivo reduzir custos para os estabelecimentos e gerar uma maior rentabilidade para os entregadores.

Esse sistema apresenta três interfaces principais. A primeira é direcionada ao gerenciamento dos entregadores e dos estabelecimentos e conta com diversas funcionalidades, como a geração de relatórios e controle financeiro. A segunda interface é destinada aos donos de estabelecimentos, cujos produtos devem ser entregues. Através dessa interface é possível cadastrar uma nova entrega, visualizar as solicitações, realizar agendamentos e o controle financeiro relativo às entregas. E, por último, há uma interface específica para o entregador e que corresponde a um aplicativo para dispositivo móvel, já que o mesmo deve estar em constante movimento. O aplicativo do entregador conta com a possibilidade de aceitar os chamados de entregas, gerenciar uma carteira digital (uma espécie de conta), visualizar sua pontuação e um histórico de corrida, dentre outras funcionalidades.

4.2 Documentação de Casos de Teste para o Speedy

Após compreender o funcionamento básico do sistema, o próximo é a definição e construção dos casos de teste. Para o sistema do Speedy, além de identificar os principais casos de teste, consideramos importante documentá-los. O modelo utilizado para essa documentação tem como base algumas ferramentas de testes já existentes e que são utilizadas no mercado (e.g. TestLink[3]), juntamente com alguns conceitos que analistas de testes utilizam ao falar sobre casos de testes.

Quadro 1: Caso de teste SPY-01

Caso de Teste SPY-01: Cadastro de clientes
Objetivo do Teste:

[2] "Sistema Web Speedy" https://speedy.app.br/#inicio. Acessado em 01 nov. 2021.
[3] "TestLink" https://testlink.org/. Acessado em 10 nov. 2021.

Verificar se o cadastro de um cliente no sistema ocorre sem erros.	
Pré-Condições:	
O usuário deve: ● Estar logado; ● Ter perfil de administrador.	
Ações:	**Resultados esperados:**
1. O usuário deve clicar no menu: *Comercial - Clientes*	O sistema deve exibir o menu dos clientes.
2. O usuário deve clicar no botão: *Novo Cliente*	O sistema deve exibir a tela de Novo Cliente
3. O usuário deve preencher todos os dados necessários e clicar no botão: *Salvar*	O sistema deve adicionar todos os dados do cliente no sistema e retornar para tela de cliente.
Prioridade:	**Alta**
Tipo de execução:	**Automática**

Fonte: o autor

Quadro 2: Caso de teste SPY-02

Caso de Teste SPY-02: Ativar clientes através do nome fantasia	
Objetivo do Teste:	
Realizar a ativação de um cliente que esteja cadastrado no sistema e verificar se a situação do cliente é alterada.	
Pré-Condições:	
O usuário deve: ● Estar logado; ● Ter perfil de administrador.	
Ações:	**Resultados esperados:**
1. O usuário deve clicar no menu: *Comercial - Clientes*	O sistema deve exibir o menu referente aos clientes
2. O usuário deve realizar uma busca preenchendo o campo a. *Nome fantasia* com o valor "Loja Tal"	O sistema deve exibir a tela de *Detalhes do cliente*
3. O usuário deve selecionar a opção de *cliente* e clicar no botão de: *ativar*	O sistema deve exibir uma mensagem para o usuário perguntando se tem mesmo certeza que deseja *ativar* o cliente.

4. O usuário deve confirmar que deseja, clicando no botão *ativar* novamente	O sistema deve *ativar* o cliente, alterando sua situação.
Prioridade:	**Alta**
Tipo de execução:	**Automática**

Fonte: o autor

4.3 Cypress

O Cypress é um ferramenta de automatização de testes em interfaces Web, que aborda os principais pontos problemáticos que os desenvolvedores e engenheiros de qualidade enfrentam ao testar aplicações web modernas. O mesmo tem como objetivo a facilitação do processo de configuração, execução e depuração dos testes [Cypress 2021].

Essa ferramenta pode ser usada para escrever vários tipos diferentes de testes, permitindo a criação de testes unitários, de integração e testes *end-to-end* (E2E), também conhecidos por testes de ponta a ponta. Mas o Cypress foi originalmente projetado para desenvolver e executar testes ponta a ponta em qualquer componente que seja renderizado e executado em um navegador web.

O teste de ponta a ponta é uma metodologia utilizada para testar todo o fluxo de funcionamento do software. Sendo assim, a ideia é validar o funcionamento de todas as funcionalidades e fluxos que o usuário é capaz de realizar. De forma resumida, o teste ponta a ponta é uma maneira de realizar uma verificação do sistema simulando o mais próximo possível da interação do usuário com o sistema.

A escolha do Cypress se dá pelo grande diferencial com relação a outras ferramentas de automatização de testes que operam fora do navegador e executam comandos remotos na rede. Cypress é exatamente o oposto. Cypress é executado no mesmo *loop* de execução de seu aplicativo. Essa ferramenta utiliza o Node JS como servidor e interpretador dos testes feitos em linguagem JavaScript. Com isso, o Cypress e Node JS mantêm uma sincronização e comunicação para realizar as execuções de tarefas, tornando a experiência da escrita e execução dos testes ainda mais ágil. Um outro ponto para essa tomada de decisão é o fato do autor deste trabalho ter um vasto conhecimento e experiência com essa ferramenta.

Com o Cypress o analista de qualidade pode acompanhar a execução dos testes em tempo real à medida que estes são executados, graças à interface da ferramenta. A Figura 1 ilustra a interface do Cypress exibida durante a execução de testes.

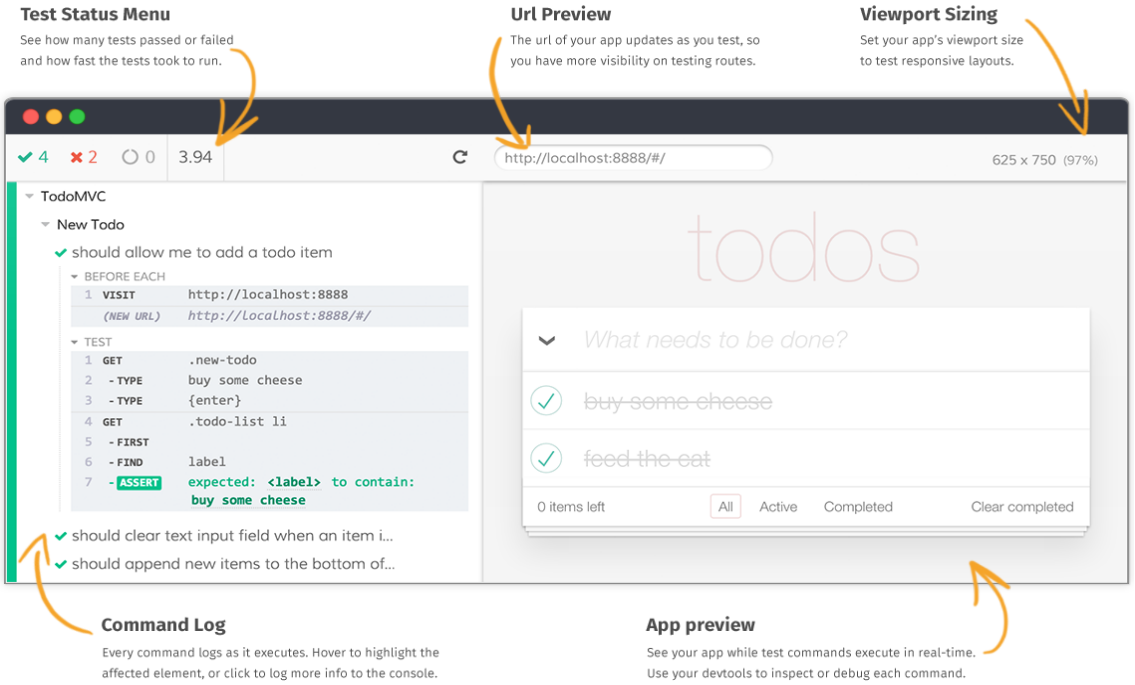

Figura 1. Interface do Cypress: Cypress.io

4.4 Preparação do ambiente de Testes com a Ferramenta Cypress

Nesta subseção serão apresentados os principais passos para preparação do ambiente de testes utilizando a ferramenta escolhida neste trabalho.

Para criar o projeto Cypress é necessário instalar o *NodeJS* e uma *IDE*. Neste trabalho a *IDE* escolhida foi o *Visual Studio Code* (*VS Code*). Na sequência, criamos um diretório para o projeto e, dentro do mesmo, realizamos a abertura do *VS Code* para executar os comandos no seu próprio terminal. O primeiro comando a ser executado foi `npm init -y`, responsável por criar o arquivo `package.json`. O próximo passo consistiu na execução do comando `npm install cypress`, utilizado para instalar a versão do Cypress mais recente no projeto. Com a instalação feita, é possível visualizar a versão instalada da ferramenta no arquivo `package.json`, e, neste mesmo arquivo, é necessário adicionar a linha de código abaixo `"cypress:open":"cypress open"`.

Isso é feito para que seja possível executar no terminal o comando `npm run cypress:open`. Este comando é o responsável por executar os *scripts* de testes e realizar a abertura de um *dashboard* chamado *The Test Runner,* ilustrado na Figura 2. Esta ferramenta do Cypress contém um *dashboard* que exibe todos os comandos que foram executados. À direita, é exibida a aplicação sendo executada de acordo com cada comando. Na parte superior, temos um menu de *status* de testes que contabiliza a quantidade de testes já aprovados, reprovados e quanto tempo durou a execução de todos os testes.

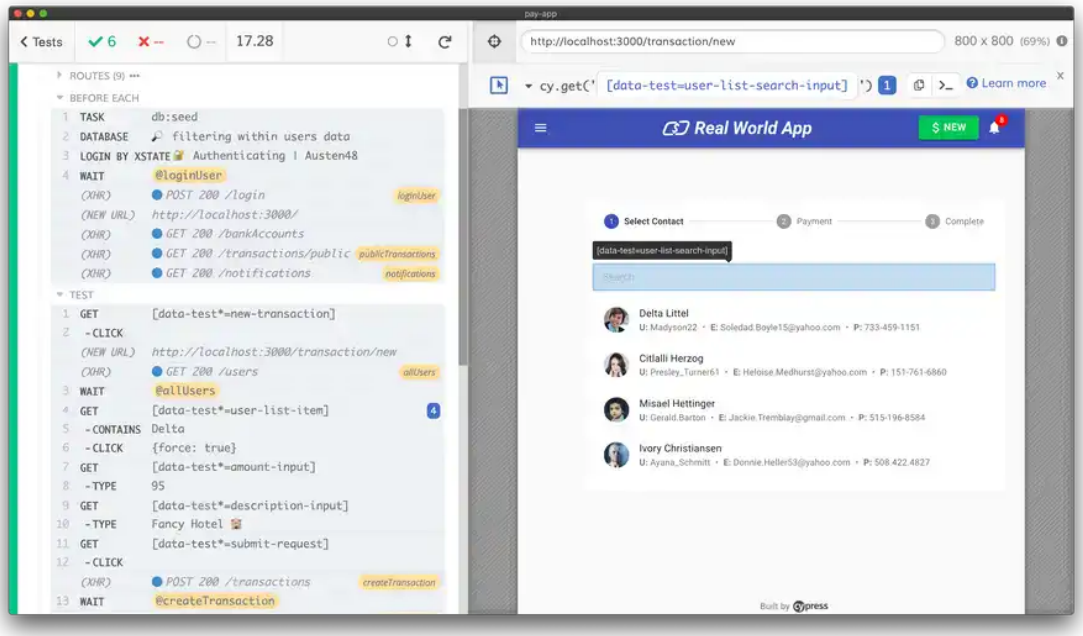

Figura 2. O *Test Runner*. Fonte: Cypress.io

Ao lado do menu temos uma função chamada *Selector Playground,* recurso interativo que auxilia o testador no mapeamento dos elementos e na geração de seletores de forma rápida e eficaz. Sendo assim, o Cypress calcula automaticamente um seletor único para o elemento escolhido, executando uma série de estratégias. Mas a ferramenta possui uma lista de prioridades, que podem ser alteradas facilmente conforme mostra a Listagem 1. Para a geração de bons seletores através do Cypress é recomendado seguir a lista de prioridades e seguir o guia de práticas recomendadas.

```
Cypress.SelectorPlayground.defaults({
  selectorPriority: ['id', 'class', 'attributes'],

})
```

Listagem 1. Exemplo de como iniciar um teste com Cypress

Para abrir o *Selector Playground*, é necessário clicar no botão próximo a *URL* na parte superior do *runner*. Passando o mouse sobre os elementos da aplicação será possível visualizar um seletor exclusivo para aquele elemento na dica de ferramenta conforme mostrado na Figura 3. Por fim, temos a visualização da *URL* e a dimensão da janela nos quais os testes estão sendo executados.

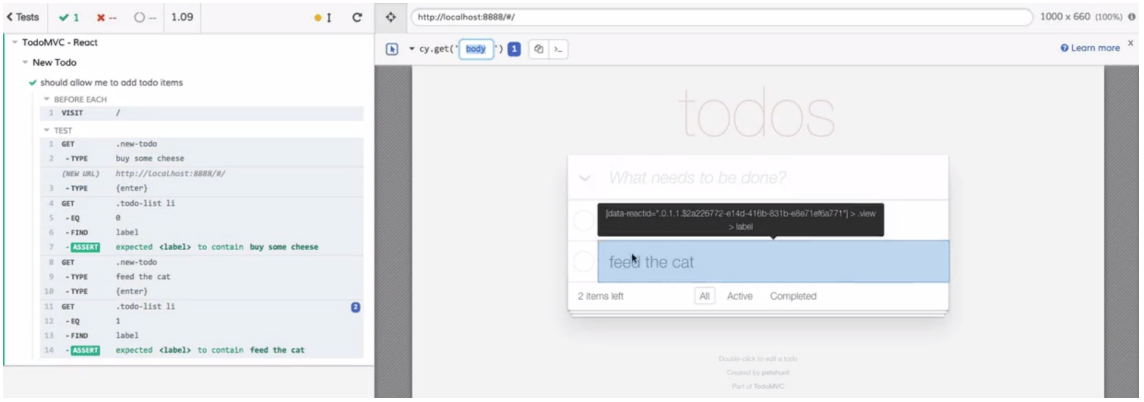

Figura 3. Exemplo do *Selector Playground*. Fonte: Cypress.io

4.5. Estrutura do Cypress

Para entendermos melhor a estruturação do código de testes, é necessário compreender a estruturação das pastas e arquivos do Cypress. Na Figura 4, no canto esquerdo, é possível visualizar a estrutura inicial de pastas e arquivos que a ferramenta cria ao ser instalada. Já no canto direito da Figura 4, é possível visualizar a estrutura atual para os testes do Speedy já com os arquivos necessários para execução dos testes.

Começando de cima para baixo, temos a pasta “fixture” que é a responsável por armazenar qualquer dado ou arquivo externo que venha a ser inserido no processo de criação dos *scripts*. Logo em seguida vem a pasta “integration”, que é o diretório responsável por armazenar todos os testes da aplicação. O diretório "plugins", como o próprio nome sugere, é um local para adicionar extensões, ou seja, injetar algumas funcionalidades a mais no Cypress. A pasta “support” é o local que armazena todos os arquivos auxiliares, mas existe um arquivo em específico que se chama “commands” que é responsável por criar comandos Cypress, transformando esses comandos em funções que podem ser referenciados através do comando “cy”. Por fim, mas não menos importante, há os diretórios "screenshots" e "videos", que são os responsáveis respectivamente por armazenar as capturas de telas e vídeos produzidos durante a execução dos testes.

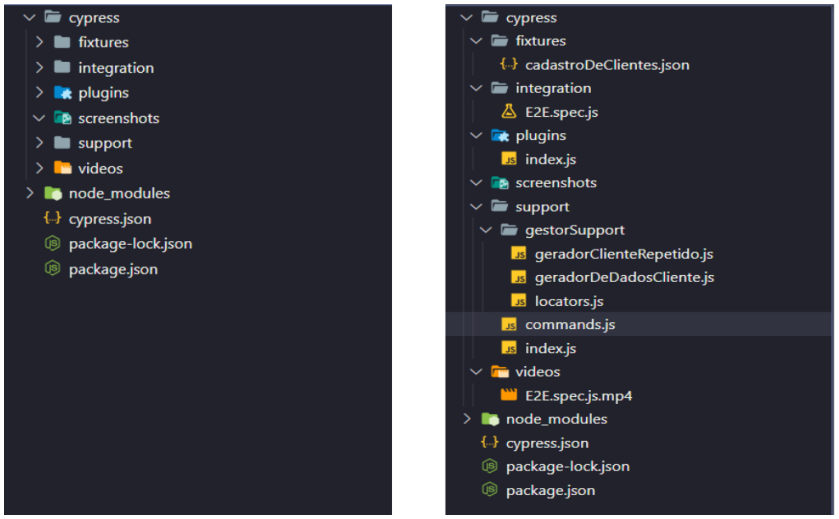

Figura 4. Estruturação do Cypress

4.5 Implementação dos Testes

Como descrito na seção 4.1, o sistema Speedy é formado por três interfaces, sendo assim, o processo de desenvolvimento dos testes tem como base as funcionalidades referente aos Clientes gerenciados por meio da interface do Gestor. O processo de implementação dos testes consistiu em analisar os casos de teste juntamente com o menu do cliente e planejar a estruturação e construção dos *scripts* de testes automatizados, visando cobrir o máximo possível de funcionalidades.

Todos os *scripts* de teste que foram construídos encontram-se em um arquivo chamado *"E2E.spec.js"*. Esse arquivo realiza a importação de alguns outros arquivos, sendo um deles o "locators.js", que é o responsável por armazenar e exportar a localização de todos os elementos com que o Cypress vai interagir no navegador para realizar determinadas ações. Um outro arquivo é o *geradorDeClientes.js*, que, como o próprio nome sugere, é responsável por gerar e exportar dados fictícios para criação de clientes.

O processo de construção e implementação dos scripts foi bastante desafiador, mas com o apoio da documentação do Cypress e de sua ampla comunidade, todas as dificuldades encontradas puderam ser superadas. Alguns dos principais desafios encontrados durante a implementação foram: a localização de alguns elementos para utilização nos scripts de teste, a interceptação de rotas, a geração de dados fictícios, a falta de acesso ao código da aplicação, as formas de gerar dados fictícios para utilizar nos testes e a dificuldade de realizar asserções eficientes.

4.6. Estrutura do Código

Escrever *scripts* de testes de forma rápida é um dos principais objetivos do Cypress. Com isso, para iniciarmos a criação de um teste automatizado com essa ferramenta, é necessário ter uma *"describe",* que é basicamente uma descrição geral daquela suíte de testes, contendo um "*it"* para descrever o que esse teste deve realizar e um ponto de partida, que é invocado por meio do *cy.visit,* recebendo como argumento uma *URL*, conforme mostrado na Listagem 2.

```
1. describe('Página inicial', () => {
2.  it('Sucesso ao carregar a página', () => {
3.    cy.visit('http://localhost:8080')//change URL to match your dev URL
4.  })
5. })
```

Listagem 2. Exemplo de como iniciar um teste com Cypress

No sistema Speedy, o ponto de partida é a tela de *login*, na qual o usuário necessita informar suas credenciais para acessar as funcionalidades do sistema. Portanto, o *login* se torna algo muito recorrente ao longo do processo de implementação dos testes e por isso surgiu a necessidade de criar uma função para evitar a repetição das linhas de código responsáveis por acessar a página de *login* e preencher todos os campos necessários.

Como dito na seção anterior, o local mais apropriado para a criação de comandos que acabam se tornando funções, é no arquivo *commands.js,* que fica dentro do diretório *support* do Cypress. Sendo assim, o código da Listagem 3 mostra a criação

do comando *login,* que simula todo o passo a passo necessário para o usuário se autenticar no sistema e ter acesso às funcionalidades. Essa função recebe como parâmetro o e-mail do usuário juntamente com a senha, visita o ponto de partida que é a tela de *login* através do *cy.visit*, mapeia os elementos correspondentes aos campos de e-mail e senha por meio do *cy.get* e preenche os campos através da função *type*. Feito isso, é necessário realizar o clique no botão de *login* por intermédio da função *click*. Por fim, é feita a interceptação da rota por meio do *cy.intercept* para que seja possível nomear essa requisição e aguarda-lá através do *cy.wait* para realizar uma asserção que visa verificar se a resposta obtida foi igual a 200, já que no *HTTP (HyperText Transfer Protocol)* esse código indica que a requisição foi bem sucedida.

```
1. import locators from '../support/gestorSupport/locators'
2. Cypress.Commands.add('login', (email, senha) => {
3.     cy.intercept('GET','https:/st-api.speedy.app.br/api/v1/regions/all/')
4.         .as('login')
5.     cy.visit('https://st-gestor.speedy.app.br/login')
6.     cy.get(locators.LOGIN.USER).type(email, {delay : 70})
7.     cy.get(locators.LOGIN.PASSWORD).type(senha,{delay :70})
8.     cy.get(locators.LOGIN.BTN_EXIBIR_SENHA).click()
9.     cy.get(locators.LOGIN.BTN_LOGIN).click()
10.
11.     cy.wait('@login').then(($res) => {
12.         expect($res.response.statusCode).to.be.equal(200)
13.    })
14.})
```

Listagem 3. Trecho de código do comando 'login'. Fonte: o autor

Como visto na Listagem 3, a função *cy.get* recebe um atributo que é importado de outro arquivo chamado *locators.js*. Esse arquivo armazena a localização de todos os elementos que são necessários até então para o Cypress realizar a execução dos testes. Os localizadores de elementos possuem uma função primordial que será abordada na próxima subseção.

Já na Listagem 4 é mostrada a função *login* sendo invocada dentro de um *beforeEach* que é responsável por executar uma série de instruções antes de cada teste. Essa função está contida em um *describe*. Sendo assim, todos os casos de testes que forem executados dentro desta função acabam realizando o *login* de forma independente do teste executado anteriormente, o que agrega valor ao processo de testes e reforça a premissa de que um teste não deve depender do outro para existir. Na seção seguinte, serão descritas lições aprendidas com maior riqueza de detalhes e como os desafios foram superados.

```
1. describe('Criação e remoção de clientes', () => {
2.     beforeEach(() => {
3.         cy.login('admin@teste.com', 'teste123')
4.     })
5.     it('Deve cadastrar um Cliente', () => {
6.        //
7.     })
8.     it('Exemplo', () => {
```

```
9.       //
10.    })
11. })
```

Listagem 4. Trecho de código para criação de um cliente: o autor

5. Desafios e lições aprendidas ao implementar testes para o Speedy

Nesta seção são apresentados os principais desafios e lições aprendidas durante o processo de desenvolvimento de criação e implementação dos testes automatizados para o sistema web Speedy. Todo o aprendizado foi adquirido por meio da experiência ao implementar os testes das funcionalidades do cliente e ao tentar superar desafios que surgiram ao longo desse processo.

5.1. Desafios e lições aprendidas com a falta de acesso ao código do sistema

Nesta subseção descrevemos de forma minuciosa as lições aprendidas através dos desafios relacionados à falta de acesso ao código do sistema. Essa falta de acesso gerou bastante dificuldade ao mapear os elementos e ao adicionar as propriedades (por exemplo, *data-cy*) dentro das tags *HTML* para auxiliar nas buscas dos elementos, o que proporciona seletores com uma boa escrita e consequentemente uma boa prática de implementação.

Um dos maiores desafios ao realizar este trabalho foi testar o sistema apesar de não termos acesso ao seu código. Isso impediu, por exemplo, a adição de atributos como o *data-cy* dentro das tags *HTML*, o que facilitaria bastante o trabalho de mapeamento dos elementos para inserir nos *scripts* de testes. Um outro ponto que dificulta um bom mapeamento e escrita dos seletores são *IDs* e nomes de classes que são gerados dinamicamente por alguns *frameworks*. Existem diversas formas de mapear e escrever os seletores, mas neste trabalho vamos nos dedicar apenas a duas.

A primeira se deu utilizando o Cypress como descrito a seguir. Partindo da execução da tela inicial (*url* base do sistema), navegamos na aplicação através do *dashboard* para mapear esses elementos por intermédio do *selector playground*. Caso essa funcionalidade gerasse um seletor muito extenso ou de difícil entendimento, optamos por realizar a escrita através do *Xpath,* conforme detalhado a seguir.

Conforme mostra a Listagem 5, o *locators.js* apresenta a definição de uma constante sendo exportada para que outros arquivos tenham acesso ao seu conteúdo interno. Essa constante armazena todos os seletores dos elementos necessários para que seja possível realizar os testes automatizados. A busca por esses elementos pode ser feita de diversas formas, mas, como dito anteriormente, o Cypress possui uma funcionalidade chamada *Selector Playground*, responsável por auxiliar os testadores a mapear de forma interativa os elementos que o *cy.get* é capaz de receber como parâmetro. No entanto, essa funcionalidade possui uma preferência de busca, favorecendo essa ordem de seletores: data-cy, data-test, data-testid, id, class, tag, attributes, nth-child, mas que pode ser alterada.

```
1. const locators = {
2.     LOGIN: {
3.         USER: `#email` ,
4.         PASSWORD: `#password`,
5.         BTN_EXIBIR_SENHA: `.MuiSvgIcon-root`,
```

```
6.         BTN_LOGIN: `.MuiGrid-grid-md-5 > .MuiButtonBase-root`
7.     }
8. }
9. export default locators
```

Listagem 5. Trecho de código do arquivo locators. Fonte: o autor

Os seletores mostrados na Listagem 5 foram gerados a partir da funcionalidade *selector playground*. Dentro da constante de locators existe um elemento chamado *login,* que contém mais quatro elementos. Os dois primeiros possuem a propriedade *id,* que facilita para a ferramenta mapear e gerar um seletor de fácil entendimento aos testadores. Os dois últimos elementos servem para exemplificar que a ausência de propriedades bem definidas como *data-cy* pode levar à geração de seletores extensos e, em alguns casos, os testadores não conseguem mapeá-los rapidamente, o que dificulta o processo de construção e até mesmo de uma futura manutenção nos *scripts* de testes.

Uma outra forma de mapear os elementos se dá através de uma linguagem de consulta chamada *Xpath*, que auxilia os desenvolvedores e testadores a navegar por documentos que usam marcadores, assim como o *HTML*. Mas é necessário ressaltar que essa metodologia não é mais indicada para mapear os elementos. A documentação e a comunidade do Cypress aconselham a utilizar o *Xpath* e outros métodos de mapeamento quando realmente não tivermos outra escolha. Com isso, buscamos ao máximo mapear os elementos através do Cypress para que ele reconheça de forma nativa, já que para utilizar o *Xpath* é preciso instalar o plugin, através do comando `npm install -D cypress-xpath`.

No *HTML* as informações são agrupadas através de tags, e o *Xpath* tem como objetivo localizar pontos específicos do documento *HTML* para gerar "endereços", que, no nosso caso, se tornam seletores. Existem diversas formas de localizar e mapear um elemento com o *Xpath*, sendo algumas ilustradas na Listagem 6.

```
1. const locators = {
2.         LOJA: {
3.               XP_BTN_DETALHES_DA_LOJA:
4.                     `//button[contains(.,"Lojas")]`
5.         },
6.         BOTAO: {
7.               FN_XP_BTN_DETALHAR_CLIENTE_CPF:
8.                     cpf => `//td[@value='${cpf}']/../td[8]`
9.         },
10. }
11. export default locators
```

Listagem 6. Trecho de código do arquivo locators. Fonte: o autor

O primeiro elemento da Listagem 6 é um botão que está contido dentro do menu das lojas que é responsável por detalhar uma loja de um determinado cliente. A nomenclatura desse seletor é formada inicialmente por uma sigla XP que faz referência ao *Xpath*, posteriormente BTN para abreviar o nome botão e, por fim, o nome daquele seletor. Para localizar esse elemento através do *Xpath* é necessário utilizar uma outra funcionalidade dos navegadores chamada *DevTools*, ilustrada na Figura 5.

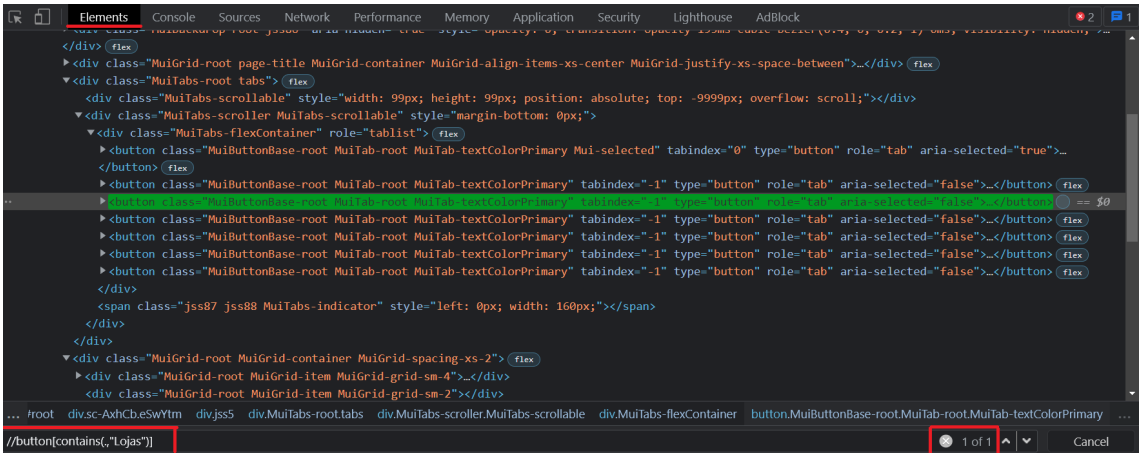

Figura 5. Utilização da ferramenta *DevTools*: o autor

Neste trabalho o navegador utilizado foi o Google Chrome. Para utilizar essa funcionalidade é preciso clicar com o botão direito e selecionar a opção “inspecionar” na tela onde o analista deseja buscar e analisar os elementos, que imediatamente será aberto o *DevTools*. Com isso, na parte superior selecionamos a opção “elements” e na parte inferior realizamos a busca através do *Xpath*. É de extrema importância que essa busca retorne um único elemento como mostrado no canto inferior direito da Figura 5, pois se for retornado mais de um elemento, o Cypress não vai ser capaz de decifrar qual elemento ele deve capturar para realizar uma determinada ação.

Para deixarmos os *scripts* de testes dinâmicos, podemos aperfeiçoar a forma na qual os seletores são mapeados. No momento que o seletor está sendo construído, o mesmo pode se tornar uma função que recebe um ou vários argumentos que são inseridos em uma determinada parte da escrita do seletor. O segundo elemento da Listagem 6 traz um exemplo que contém essas características descritas anteriormente; a sua nomenclatura segue a mesma estrutura do primeiro elemento, mas com o acréscimo na parte inicial da sigla FN que serve para abreviar a palavra função. Sendo assim, essa função recebe um parâmetro que é inserido no meio do seletor, para realizar a busca de um elemento de forma mais específica. Neste caso, esse seletor é responsável por mapear um cliente e clicar no botão de detalhar cliente através do seu CPF, que funciona como um identificador único para os clientes cadastrados no sistema Speedy.

Como dito anteriormente, toda a comunidade do Cypress e sua documentação aconselham a utilizar o que podemos chamar de seletores nativos, que são aqueles seletores que o Cypress reconhece sem o auxílio de plugins ou extensões. Este problema é resultado da falta de acesso aos sistemas, já que, caso o testador tivesse acesso seria possível adicionar as propriedades de preferência do Cypress às tags *HTML*.

5.2. Desafios e lições aprendidas ao gerar dados fictícios

Nesta subseção descrevemos de forma detalhada os desafios e as lições aprendidas relacionados à implementação de dados fictícios, que tem como intuito gerar uma massa de dados para os *scripts* de testes.

A geração de dados de testes é uma maneira de simplificar e agilizar o processo de execução dos *scripts*. Isso é interessante visto que alguns cenários de testes exigem uma grande variedade de dados. Um exemplo disso seria o formulário de cadastro dos clientes. Realizar o preenchimento deste formulário de forma manual demanda uma

certa quantidade de tempo e um trabalho repetitivo para o testador, o que reforça a importância dos testes automatizados, que buscam automatizar esse trabalho repetitivo e, consequentemente, economizar o tempo gasto com a execução de testes.

Sendo assim, para facilitar a geração de dados para testes, recorremos às bibliotecas chamadas *Faker* e *Faker-BR* para definir dados fictícios a utilizar nos testes. Para instalar essas bibliotecas como dependências do projeto, é necessário executar respectivamente os comandos *npm install faker* e *npm install faker-br*. Feito isso, é preciso referenciá-las e invocá-las como mostra a Listagem 7.

```
1.  const faker1 = require('faker/locale/pt_BR');
2.  const faker = require('faker-br');
3.  const nome_fantasia = faker1.company.companyName()
4.  const razaoSocial = faker1.name.firstName(nome_fantasia)
5.  const cpf_cnpj = faker.br.cnpj()
6.  const pessoa_p_contato = faker1.name.firstName()
7.  const telefone = faker1.phone.phoneNumber()
8.  const email = faker1.internet.email(razaoSocial)
9.  const cep = "58045100"
10. const numero = faker1.datatype.number()
11. const complemento = faker1.random.word(razaoSocial)
12. const nomeDaLoja = faker1.company.companyName()
13. const numeroTelefone = faker1.phone.phoneNumber()
14. const contatoDaLoja = faker1.name.firstName()
15. const emailLoja = faker1.internet.email(nomeDaLoja)
16. const cepLoja = "58045100"
17. const numeroDoLocal = faker1.datatype.number(numero)
18. const complementoDaLoja = faker1.random.word(razaoSocial)
19. const cliente = {
20.         razaoSocial,
21.         nome_fantasia,
22.         cpf_cnpj,
23.         pessoa_p_contato,
24.         telefone,
25.         email,
26.         cep,
27.         numero,
28.         complemento,
29.         nomeDaLoja,
30.         numeroTelefone,
31.         contatoDaLoja,
32.         emailLoja,
33.         cepLoja,
34.         numeroDoLocal,
35.         complementoDaLoja
36. }
37. export default cliente
```

Listagem 7. Código do arquivo geradorDeCliente. Fonte: o autor.

Com as bibliotecas referenciadas podemos utilizar os seus métodos para criar diversas constantes e atribuir os dados geradores às próprias constantes. A partir disso, criamos uma constante para englobar todos esses dados para formar um cliente. Com o

cliente criado podemos importar e referenciar os valores configurados nos locais desejados, como mostra a Listagem 8.

```
1. import cliente from '../support/gestorSupport/geradorDeDadosCliente'
2.
3. it.skip('Deve criar Clientes', () => {
4.   cy.get(locators.CLIENTES.INF_GERAIS.RAZAO_SOCIAL).type(cliente.razaoSocial)
5.   cy.get(locators.CLIENTES.INF_GERAIS.NOME_FANTASIA).type(cliente.nome_fantasia)
6.  // Restante do código abaixo
7. //
8. }
```

Listagem 8. Trecho de código do arquivo E2ECliente. Fonte: o autor.

5.3. Desafios e lições aprendidas ao realizar asserções

Nesta subseção descrevemos de forma detalhada os desafios e as lições aprendidas relacionados à implementação das asserções durante o desenvolvimento dos *scripts* de testes.

Podemos dizer que na computação uma asserção é um predicado inserido no programa que visa verificar uma condição que o testador supõe que seja verdadeira em determinado ponto e comparar essas expectativas com os resultados obtidos. O Cypress empacota uma biblioteca de asserções chamada *Chai* e algumas extensões como *Sinon* e *jQuery*, trazendo ao testador dezenas de asserções bem elaboradas e prontas para serem utilizadas de forma nativa. Vale ressaltar a importância que as asserções trazem para os *scripts* de teste, já que são responsáveis por averiguar se todo aquele processo que é executado através do *script* de teste será aprovado ou reprovado e quanto mais asserções houver, mais precisos e seguros serão os *scripts* de testes.

Um dos maiores desafios de construir boas assertivas é saber qual a asserção se encaixa melhor para aquele determinado *script* de teste. No Cypress existem dois tipos de asserções - as implícitas e as explícitas -. Ambas estão presentes na Listagem 9, conforme explicado a seguir.

As implícitas utilizam a função chamada *Should*, que faz uma afirmação sobre o assunto atual, ou seja, sobre o elemento atual que foi capturado pelo Cypress. Na Listagem 9, as linhas 19 e 20 nos mostram a utilização desta função, onde foi encadeado o *cy.get* juntamente *.should* para capturar a mensagem que é exibida para o usuário e verificar através do *.should* se dentro desta mensagem há um determinado texto. Caso exista esse determinado texto, dentro do elemento capturado pelo Cypress será retornado um valor verdadeiro (*true*), que significa que o teste foi aprovado. Caso retorne um valor falso (*false*), significa que o teste foi reprovado.

As asserções explícitas utilizam a função *Expect*, conforme mostrado nas linhas 15, 16 e 17. Essa função, como o próprio nome sugere, espera algo para realizar uma comparação entre o que está sendo esperado e o que foi obtido. Esse exemplo da Listagem 9 é bem interessante, já que são utilizadas diversas funções do Cypress. Na linha 14 temos um *cy.wait* que aguarda uma interceptação de uma requisição do tipo *PUT* (método *HTTP* utilizado para atualizar dados). Esta é nomeada de '@attDeEmail' através da função *.as*. O *cy.wait* foi encadeado com a função *.then* que permite ao testador trabalhar com o assunto gerado no comando anterior. Sendo assim, esse *script* de teste intercepta a requisição de atualização do e-mail, realiza todo o processo de

edição, aguarda a interceptação que instancia uma função através do *.then* que recebe como argumento uma variável *($res)* e, por fim, executa as asserções através do *expect*.

Neste exemplo vemos dois *expects*. O primeiro verifica se dentro da resposta foi retornado um *status Code* igual a 200 enquanto o segundo verifica se o e-mail passado no corpo da requisição é igual ao e-mail passado no corpo da resposta. Caso o primeiro *expect* seja falso, o *script* de teste para de ser executado e recebe o *status* de reprovado.

```
1. it('Deve editar o e-mail através do nome fantasia do cliente', () => {
2.     cy.intercept({
3.         method: 'PUT',
4.         url: `https://st-api.speedy.app.br/api/v1/customers/**`,
5.     }).as('attDeEmail')
6.
7.     cy.acessarPaginaDoCliente('AllTips')
8.     cy.xpath(locators.EDITAR.XP_BOTAO_EDITAR).click()
9.     cy.get(locators.CLIENTES.INF_GERAIS.EMAIL)
10.         .clear()
11.         .type(exemplo.teste@gmail.com', {delay : 90})
12.     cy.xpath(locators.EDITAR.XP_BOTAO_SALVAR).click()
13.
14.     cy.wait('@attDeEmail').then(($res) => {
15.       expect($res.response.statusCode).to.be.equal(200)
16.       expect($res.request.body.email)
17.          .to.be.equal($res.response.body.email)
18.     })
19.     cy.get(locators.CLIENTES.MESSAGEM).should('contain', 'Cliente
20.       atualizado com sucesso')
21.})
```

Listagem 9. Trecho de código do arquivo E2ECliente: o autor

Sendo assim, o testador deve escolher qual a melhor asserção para cada *script* de teste. Durante a tomada de decisão de qual asserção devemos utilizar, levamos em consideração a experiência do testador e a documentação do Cypress, que nos mostra os dois tipos de asserções e como cada uma delas pode ser utilizada, já que, em alguns casos, uma é mais adequada que a outra. Neste trabalho, as asserções implícitas foram utilizadas para elementos visuais; já as explícitas foram usadas para testes que realizam interceptações em *api*.

6. Considerações Finais e Trabalhos Futuros

Este trabalho objetivou descrever um relato de experiência em torno da construção de *scripts* de testes através da ferramenta Cypress, visando uma maior cobertura dos casos de testes para garantir a qualidade das funcionalidades referentes ao menu do cliente que está presente no sistema Speedy.

Levando em consideração tudo o que foi apresentado neste artigo acreditamos que o principal objetivo foi alcançado, já que foram apresentados os passos para a criação e implementação dos *scripts* de testes visando garantir a qualidade do sistema Speedy. Os principais desafios e lições aprendidas foram detalhados para que o leitor possa entender e até mesmo utilizar esses conceitos em seus projetos.

O artigo buscou contribuir para um melhor direcionamento ao leitor para criação e implementação dos *scripts* de testes automatizados usando a ferramenta Cypress. Na

experiência obtida, chegamos à conclusão de que a automatização dos testes ajuda a reduzir a probabilidade de erros, provê uma maior segurança, um feedback mais rápido e, com mais frequência, evita um trabalho repetitivo e a perda de tempo em executar diversos casos de testes. Com isso, a criação de testes automatizados se torna componente essencial no processo de desenvolvimento de software.

Acreditamos que o relato de experiência aqui apresentado pode auxiliar bastante o leitor que está a procura de conhecimento na área de testes de sistemas Web ou iniciando uma jornada na área de qualidade e até mesmo na área de automatização. O trabalho também visa contribuir para uma reflexão sobre a importância de testes automáticos, em especial com a ferramenta, visto que observamos uma escassez de trabalhos em relação ao Cypress.

Como trabalho futuro, almejamos realizar a ampliação dos casos de testes, juntamente com a construção dos *scripts* de testes, ajustar a nomenclatura de alguns seletores e solicitar acesso ao código da aplicação para adicionarmos as tags *HTML,* conforme sugerido pela documentação do Cypress para facilitar o mapeamento dos elementos. Por fim, propomos também a implementação do padrão *Page Object* e que se analise os benefícios da estratégia.

Referências

Amorim, Daniel; Vieira, Hugo Vares (2015). Página Web. Disponível em: https://www.thoughtworks.com/pt-br/insights/blog/functional-tests-how-decide-what-automate/ . Acesso em: 20 de setembro. de 2021

Cypress, (2021). Página Web. Disponível em: https://docs.cypress.io/ . Acesso em: 10 de ago. de 2021

Delamaro, M , (2016). Introdução ao Teste de Software. Grupo GEN. 9788595155732. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788595155732/. Acesso em: 2021 ago. 10.

Ebanhesaten, K.; Füßl, F.; Streitferdt, D. (2013) "Refactoring and automated testing of distributed ERP systems," (2013) IEEE 37th Annual Computer Software and Applications Conference Workshops, pp. 681-684, doi: 10.1109/COMPSACW.2013.105.

Fowler, M. (2019). Refatoração: Aperfeiçoamento e Projeto. Bookman Editora

Fowler, M. (2021). Página Web. Disponível em: https://refactoring.com/ . Acesso em: 29 de ago. de 2021

IEEE, (2021). Standard Glossary of Software Engineering Terminology. Página Web. Disponível em: http://www.mit.jyu.fi/ope/kurssit/TIES462/Materiaalit/IEEE_SoftwareEngGlossary.pdf . Acesso em: 29 de ago. de 2021

ISTQB, (2018). International Software Testing Qualifications Board

ISTQB, (2021). International Software Testing Qualifications Board. Página Web. Disponível em: https://glossary.istqb.org/en/search/. Acesso em: 10 de nov. de 2021.

Maldonado, J. Automatização de Teste de Software com Ferramentas de Software Livre, (2018). Grupo GEN, 2018. 9788595155305. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788595155305/. Acesso em: 2021 ago. 13.

Pressman, Roger S, (2011),.Engenharia de software: Uma abordagem profissional, AMGH, 7ª edição.