UNIVERSIDADE FEDERAL DE SANTA MARIA
COLÉGIO POLITÉCNICO DA UFSM
CURSO DE SISTEMAS PARA INTERNET

Eleonora Teixeira

ANÁLISE DE FERRAMENTAS DE AUTOMAÇÃO DE TESTES DE SOFTWARE

Santa Maria, RS
2022

Eleonora Teixeira

ANÁLISE DE FERRAMENTAS DE AUTOMAÇÃO DE TESTES DE SOFTWARE

Trabalho apresentado ao Curso de Sistemas para Internet como requisito parcial para a obtenção do grau de **Tecnólogo em Sistemas para a Internet** da Universidade Federal de Santa Maria (UFSM).

Orientador: Prof. Dr. Giani Petri

Santa Maria, RS
2022

Eleonora Teixeira

ANÁLISE DE FERRAMENTAS DE AUTOMAÇÃO DE TESTES DE SOFTWARE

Trabalho apresentado ao Curso de Sistemas para Internet como requisito parcial para a obtenção do grau de **Tecnólogo em Sistemas para a Internet** da Universidade Federal de Santa Maria (UFSM).

Aprovado em 04 de fevereiro de 2022:

_____

Giani Petri. Dr.
Presidente/Orientador

_____

Marcos Alexandre Rose Silva. Dr.
(UFSM)

_____

Daniel Lichtnow. Dr.
(UFSM)

Santa Maria, RS
2022

RESUMO

ANÁLISE DE FERRAMENTAS DE AUTOMAÇÃO DE TESTES DE SOFTWARE

AUTOR: Eleonora Teixeira
ORIENTADOR: Prof. Dr. Giani Petri

Com o avanço da tecnologia da informação, está se tornando cada vez maior a demanda para o desenvolvimento de sistemas *web*. Junto a isto, a busca pelo desenvolvimento de sistemas com qualidade e que satisfaça os requisitos aumenta. Através das técnicas de testes de *software*, pode-se ter uma garantia maior que, ao término do desenvolvimento, se obtenha um sistema funcional e que corresponda às especificações. No entanto, muitas vezes a aplicação dessas técnicas se dá de forma manual, fazendo-se necessário muitos testes acerca de uma funcionalidade, a fim de certificar a não ocorrência de erros, tornando-se um processo custoso. A implementação da automação de testes surge para minimizar o custo decorrente dos testes manuais, testando uma funcionalidade diversas vezes, de forma automática. Desta maneira, este trabalho de conclusão de curso tem por objetivo analisar as vantagens e desvantagens quanto ao uso de ferramentas de testes automatizados que auxiliam no processo de controle da qualidade de software. Foram analisados os *frameworks Capybara* e *Cypress* em termos de configuração, estruturação, linhas de código, depuração e tempo de execução e os resultados mostram que o *Cypress* se sobressai em relação a configuração, estruturação, depuração e tempo de execução, enquanto o *Capybara* se sobressai em relação as linhas de código geradas.

Palavras-chave: Qualidade de Software. Testes automatizados. Teste de Software.

ABSTRACT

ANALYSIS OF SOFTWARE TESTING AUTOMATION TOOLS

AUTOR: Eleonora Teixeira
ORIENTADOR: Prof. Dr. Giani Petri

With the advancement of information technology, the demand for the development of web systems is becoming even greater. Along with this, the search for the development of quality systems that satisfy the requirements increases. Through software testing techniques, it is possible to guarantee that at the end of development, a functional system that matches the specifications is obtained. However, many times the application of these techniques is done manually, making it necessary to do many tests about a functionality, in order to certify the non-occurrence of errors, making it a costly process. The implementation of test automation arises to minimize the cost of manual testing, testing a functionality several times, automatically. Thus, this course conclusion work aims to analyze advantages and disadvantages regarding the use of automated testing tools that help in the software quality control process. To identify the pros and cons and which tool is best suited in the context of automating web systems. Capybara and Cypress frameworks were analyzed in terms of configuration structure, line of code, debugging and runtine and the results show that Cypress excels in terms of configuration, structuring, debugging and runtime, while Capybara excels in relation to the generated line of code.

Keywords: Software Quality. Automated Testing. Software Testing.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
BDD	*Behavior driven development*
Cmder	*Console Emulator*
GQM	*Goal – Question - Metric*
GUI	*Graphical User Interface*
IBM	*International Business Machines*
IDE	*Integrated Development Environment*
RFT	*Rational Functional Tester*
VSCode	*Visual Studio Code*
VV&T	Validação, Verificação e Teste

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento de sistemas *web* pode-se tornar bastante complexo, pois depende muito das características e funcionalidade do sistema a ser desenvolvido. Sendo assim, este se torna suscetível a diversos tipos de erros, podendo resultar na obtenção de um produto diferente do esperado e especificado (DELAMARO; JINO; MALDONADO, 2013).

A qualidade de *software* visa encontrar defeitos no software que está sendo desenvolvido, assim vários cenários são executados, a fim de avaliar o comportamento do sistema e verificar se esse está em conformidade ao que foi especificado em seu escopo (BARTIÉ, 2002). Neste contexto, a Engenharia de Software detém de várias técnicas de teste, para o controle de qualidade, que se aplicadas durante o desenvolvimento, podem assegurar que ao final dele tenha-se um sistema funcional (PRESSMAN; MAXIM, 2016). Muitas vezes, a realização destes testes é feita de forma manual, seguindo uma série de atividades, chamadas de "Validação, Verificação e Teste" ou "VV&T" (DELAMARO; JINO; MALDONADO, 2013).

Durante o processo de controle da qualidade de software, também há a implementação de testes de software automatizados, que podem reduzir o esforço necessário para a realização de testes em uma funcionalidade específica ou até mesmo em um sistema completo. Esse tipo de teste permite aumentar a quantidade de testes a serem realizados em um curto espaço de tempo (FEWSTER; GRAHAM, 1999).

Desta maneira, pode-se observar em como a automação é benéfica e menos custosa que os testes manuais, pois além de rápida, se torna muito mais produtiva (FEWSTER; GRAHAM, 1999).

1.1 CONTEXTUALIZAÇÃO DO PROBLEMA

A realização de testes manuais faz com que sejam necessários muitos testes acerca de uma funcionalidade do sistema, apenas para se certificar que esteja livre de erros. Desta forma, o teste de software manual é eficaz na detecção de erros, porém acaba se tornando pouco eficiente por ser um processo demorado e custoso. Em comparação, a execução de testes manual, que podem levar horas, os testes automatizados podem ser executados em minutos, tornando o processo muito mais rápido, eficiente e menos custoso (FEWSTER; GRAHAM, 1999).

No contexto de testes automatizados, há diversas ferramentas que auxiliam na automação. Tipicamente essas ferramentas são programas ou scripts que, ao serem programados, executam casos de testes acerca de cada funcionalidade do sistema, simulando situações específicas, evitando que passos importantes sejam ignorados e facilitando a identificação de possíveis falhas no sistema (BERNARDO; KON, 2008). Para programar os testes, pode-se usar, por exemplo, o *Capybara* escrito em *Ruby*, o *Selenium Webdriver* que utiliza o *Java* dentre outras que utilizam linguagens como *Python*, *Javascript* e C# (PAPITO, 2020).

1.1.1 Problema

Esse trabalho aborda como os testes automatizados podem trazer mais rapidez e produtividade para a realização de testes de software, em contrapartida aos testes manuais que são muito mais custosos e demorados, podendo assim deixar passar despercebido algum erro. Considerando que a realização de testes automatizados oferece diversas ferramentas para o auxílio na atividade de testes. Neste contexto, a pergunta que norteia este trabalho é: Quais são as características das ferramentas de automação de testes existentes e como escolher a mais adequada para a realização da automação?

1.2 JUSTIFICATIVA

No início do desenvolvimento de software, a realização de testes muitas vezes era uma tarefa realizada apenas a fim de corrigir problemas existentes já conhecidos, sendo feita pelo próprio desenvolvedor (BARTIÉ, 2002).

A partir de 1957, houve mudanças em relação ao conceito do que é teste de software, conseguindo ampliar o valor que este tem na detecção de erros. Porém, este ainda era realizado de forma tardia, somente no final do processo de desenvolvimento, tornando a correção destes erros muito mais custosa e demorada (BARTIÉ, 2002).

No início dos anos 80, surgiram os primeiros conceitos de qualidade de software, onde havia o trabalho conjunto entre desenvolvedores e testadores, durante todo o processo de desenvolvimento do software. Apenas nos anos 90 se deu início a produção de ferramentas de testes, essas trariam alta produtividade e qualidade ao processo de teste (BARTIÉ, 2002).

A pesquisa justifica-se pela proposta de uma análise dessas ferramentas de automação de testes e o benefício que elas podem trazer durante o processo de desenvolvimento de

software. Sendo assim, somente nos últimos anos que houve uma intensificação na importância dessa para utilização no auxílio do controle da qualidade do software.

1.3 OBJETIVOS

1.3.1 Objetivo Geral

Analisar as vantagens e desvantagens quanto ao uso de ferramentas de testes automatizados que auxiliam no processo de controle da qualidade de software.

1.3.2 Objetivos Específicos

Para atingir o objetivo principal deste trabalho os seguintes objetivos específicos são relacionados:

- Realizar uma análise na literatura sobre as técnicas de testes de software;
- Pesquisar acerca das principais ferramentas de automação de testes;
- Analisar as ferramentas de automação de testes dentro dos seus contextos de uso;
- Avaliar e identificar os prós e contras de cada ferramenta de automação de testes analisada.

1.4 METODOLOGIA

Este trabalho adotará uma metodologia com abordagem multimétodo, dividida nas seguintes etapas:

Etapa 1 – Nesta etapa é feita uma pesquisa e análise da literatura, acerca de conceitos importantes à compreensão deste trabalho, através da realização das seguintes atividades:

1.1 - Análise teórica sobre o processo de desenvolvimento do software;

1.2 - Análise teórica sobre testes de software;

1.3 - Análise teórica identificando as técnicas de testes manuais;

1.4 - Análise teórica acerca de testes automatizados; Pesquisa e apresentação sobre algumas ferramentas de testes automatizados.

Etapa 2 – Nesta etapa é realizada uma busca e análise de ferramentas e artigos relacionados que comparam as ferramentas de testes automatizados, através da realização das seguintes atividades:

2.1 - Buscar as principais ferramentas de automação de testes relatadas em publicações científicas;

2.2 - Analisar o contexto de uso dessas ferramentas de automação de testes;

2.3 - Buscar publicações científicas que comparam essas ferramentas de automação;

2.4 - Analisar os critérios utilizados nas comparações e análises das ferramentas encontradas.

Etapa 3 – Nesta etapa será realizada uma avaliação para as vantagens e desvantagens de cada ferramenta de automação de testes, seguindo a realização das seguintes atividades:

3.1 – Selecionar duas ferramentas de automação de testes;

3.2 – Definir questões e métricas de análise utilizando a abordagem GQM (BASILI et al., 1994);

3.3 – Realizar a automação de testes em uma aplicação web;

3.4 – Analisar e relatar os resultados obtidos para cada ferramenta de automação de testes.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os conceitos teóricos relacionados a área de qualidade de software e testes, manuais e automatizados, com o objetivo de proporcionar um melhor entendimento e compreensão da temática abordada no presente trabalho.

2.1 O PROCESSO DE DESENVOLVIMENTO DE SOFTWARE

O desenvolvimento de software é uma atividade que objetiva alcançar um propósito específico de negócio. Softwares profissionais, geralmente são desenvolvidos por equipes e são alterados e mantidos durante toda sua vida (SOMMERVILLE, 2011).

À medida em que há um aumento na importância do software, há a busca por tecnologias que tornem mais fácil, rápido e mais barato o processo de desenvolver e manter softwares de qualidade (PRESSMAN; MAXIM, 2016).

O processo de software detém modelos de processo, cada um destes modelos representa uma perspectiva particular de um processo (SOMMERVILLE, 2011). Os modelos de processo fornecem um guia, definindo o fluxo de todas as atividades, ações e tarefas. Os principais modelos genéricos são modelo cascata, incremental e concorrente. O modelo cascata tem uma abordagem sequencial e sistemática para o desenvolvimento do software. O modelo incremental é um processo iterativo em que se produz várias versões do software rapidamente. O modelo concorrente permite a representação de elementos iterativos e concorrentes de qualquer modelo de processo. Devido às constantes mudanças no desenvolvimento de software, surgiu a necessidade da criação do processo de desenvolvimento ágil que envolve ciclos iterativos e incrementais do produto (PRESSMAN; MAXIM, 2016).

Independentemente do modelo adotado, em geral, o processo de desenvolvimento de software, deve seguir uma série de atividades metodológicas, que não tem relação com o tamanho e complexidade do software a ser desenvolvido, para que assim ao final se tenha um produto de software. Uma metodologia genérica é composta de cinco atividades (PRESSMAN; MAXIM, 2016):

- Comunicação: entende os objetivos do projeto e reúne os requisitos que irão definir os recursos e funcionalidades do software;
- Planejamento: cria o plano de projeto de software, onde são descritas as tarefas a serem conduzidas, os riscos que possam surgir, os recursos necessários, o produto resultante e um cronograma de trabalho;

- Modelagem: cria os modelos que ajudarão no entendimento das necessidades do software e o projeto que irá atender essas necessidades;
- Construção: gera o código e os testes necessários, a fim de revelar possíveis erros;
- Entrega: entrega ao cliente, que irá avaliar o produto e prover feedback, acerca do que foi entregue.

Essas atividades são aplicadas repetidamente durante o processo, de acordo com as várias iterações que possam ocorrer. A cada iteração, é produzido um novo incremento que disponibilizará um recurso ou funcionalidade ao software, tornando o software cada vez mais completo (PRESSMAN; MAXIM, 2016).

Essas atividades metodológicas são complementadas por várias atividades de apoio, que ao serem aplicadas ao projeto de software, ajudam a gerenciar o andamento, a qualidade, as alterações e riscos. Tipicamente essas atividades são:

- Controle e Acompanhamento do projeto: viabiliza a avaliação do progresso em relação ao plano de projeto, podendo assim, tomar medidas necessárias para o cumprimento do cronograma;
- Administração de riscos: avalia os riscos que possam vir a afetar o resultado do projeto;
- Garantia da qualidade de software: define e realiza as atividades que garantem a qualidade do software;
- Revisões Técnicas: avalia artefatos, a fim de identificar e eliminar possíveis erros;
- Medição: auxilia na entrega de acordo com os requisitos do software;
- Gerenciamento da configuração de software: gerencia os resultados dos incrementos ao longo do processo de desenvolvimento;
- Gerenciamento da capacidade de reutilização: define critérios para a reutilização de componentes e estabelecer meios para a obtenção desses componentes reutilizáveis;
- Preparo e produção de artefatos de software: atividade em que há a criação de artefatos.

Neste trabalho será aprofundado apenas atividades de Garantia de qualidade de software, tais como o processo de testes e os tipos de testes.

2.2 GARANTIA DE QUALIDADE DE SOFTWARE

A implementação de um processo de qualidade de software tem por objetivo determinar um processo que garanta e gerencie nível de qualidade do produto e processo de desenvolvimento de software. O desenvolvimento de software inadequado eleva o custo total

de desenvolvimento significativamente, ampliando os riscos de insucesso do projeto (BARTIÉ, 2002).

O produto final do processo de desenvolvimento de software é o conjunto de decisões e ações geradas durante todo o ciclo do desenvolvimento. Para produzir software de qualidade, é essencial investir em qualidade, durante todo o processo de desenvolvimento (BARTIÉ, 2002). Ao buscar a qualidade de software em todas as atividades de desenvolvimento, há uma redução na quantidade de retrabalho, resultando em custos menores e diminuindo o tempo em que o produto será entregue e posto em produção (PRESSMAN; MAXIM, 2016).

Para obter um software de qualidade, o processo de garantia da qualidade deve ter um enfoque simultâneo no produto e no processo de desenvolvimento do software. Desta maneira, há duas dimensões fundamentais determinadas para alcançar a qualidade do software (BARTIÉ, 2002):

- Dimensão da qualidade do processo: devem ser consideradas nessa dimensão, todas as atividades e etapas do processo de desenvolvimento do software. Isso inclui todos os requisitos levantados, os modelos e especificações do negócio, arquiteturas, modelos de dados e classes, análise de custos, ou seja, todos os artefatos gerados durante a etapa do desenvolvimento do software;
- Dimensão da qualidade do produto: o principal objetivo dessa dimensão é a garantia da qualidade do produto de software gerado. É nessa dimensão que os testes de software serão realizados, a fim de encontrar possíveis erros por meio do “estresse” das telas e funcionalidades.

Dentro da garantia da qualidade temos as técnicas de Verificação e Validação. A verificação tem por objetivo garantir a qualidade no processo do desenvolvimento do software, enquanto a validação visa a garantia da qualidade do produto de software. Cada uma dessas técnicas possui objetivos distintos, mas, ao serem executados de forma complementar, fortalecem o processo de detecção de erros, aumentando a qualidade do produto (BARTIÉ, 2002).

A partir desse momento será trabalhado com enfoque em testes de software, bem como conceitos básicos de testes, níveis de testes e as técnicas utilizadas para a sua realização.

2.3 TESTES DE SOFTWARE

Testes de software são conjuntos de atividades planejadas antecipadamente e executadas de forma sistêmica que visam encontrar erros cometidos durante a fase de desenvolvimento.

Para isto é empregado um conjunto de técnicas específicas, para cada tipo de teste. Essas técnicas devem ser abrangidas desde testes de baixo nível até os testes de alto nível. Teste de baixo nível são testes estruturais, realizados a nível de código, onde é testado a menor unidade do sistema. Os testes de alto nível são testes funcionais, estes são guiados pelas especificações dos requisitos do sistema e suas funcionalidades (PRESSMAN; MAXIM, 2016).

Testar um software consiste em uma execução controlada do software a fim de validar se esse apresenta o comportamento especificado. Sendo assim, tem como principal objetivo apresentar o maior número de falhas com o mínimo de esforço possível, revelando se os resultados estão de acordo com o esperado (NETO, 2008).

Um bom planejamento de teste é fundamental, para isto cria-se uma coleção de procedimentos e casos de teste. Um procedimento de teste é um conjunto detalhado de instruções para a execução dos testes, podendo ser invocado dentro de vários casos de testes. O caso de teste é a especificação de entradas, resultados esperados e condições de execução, do artefato sendo testado, esses têm por objetivo a detecção de defeitos e não demonstrar que o programa funciona corretamente (PAULA FILHO, 2019). A Figura 1 apresenta um exemplo de um caso de teste.

Figura 1- Exemplo de caso de teste genérico

Caso de teste	**Cadastrar cliente**	
Pré-condição	**O usuário deve estar autenticado no sistema.**	
Pós-condição	**A transferência deve ser efetivada.**	
Fluxo principal		
Passo	**Ação**	**Resultado esperado**
1	Consultar saldo em conta	Exibição de saldo em conta
2	Cadastrar beneficiário	Beneficiário cadastrado
3	Realizar transferência de valor	Emitir recibo de confirmação de transferência
Fluxo alternativo		
Passo	**Ação**	**Resultado esperado**
1	Cadastrar beneficiário	Beneficiário cadastrado
2	Realizar transferência	Emitir recibo de confirmação de transferência

Fonte: Fátima et al, 2016.

Dentro das atividades de teste, temos os conceitos de defeitos, erros e falhas e estes se diferem nos seguintes aspectos (NETO, 2008):

- Defeito: Um ato inconsistente cometido por um indivíduo;
- Erro: Manifestação concreta do defeito em um artefato de software;
- Falha: Comportamento operacional que difere do esperado.

Em síntese, os defeitos são causados por pessoas e esses podem ocasionar na manifestação de erros que, consequentemente, geram falhas, que consistem em um comportamento inesperado no software. Sendo assim, testes revelam apenas as falhas no produto de software (NETO, 2008). O principal objetivo da metodologia de testes, é maximizar a cobertura desses, a fim de encontrar o maior número de falhas possíveis na sua execução (PAULA FILHO, 2019).

Desta maneira, se tem a necessidade de realizar os testes em diferentes níveis, com o objetivo de avaliar o software em diferentes perspectivas, de acordo com o artefato gerado em cada fase do ciclo de desenvolvimento (NETO, 2008).

Os principais níveis de teste de software são:

- Teste de Unidade: explora a menor unidade do projeto, ou seja, o código escrito em si. Esse teste busca gerar falhas que possam ser ocasionadas por defeitos lógicos e de implementação (NETO, 2008).
- Teste de Integração: testa a interface do conjunto de módulos de sistema, durante a integração desses, verificando seu funcionamento (HIRAMA, 2011).
- Teste de Validação: testa o software como um todo, verificando se suas funcionalidades, comportamentos e desempenho estão sendo atendidas de acordo com o especificado (HIRAMA, 2011).
- Teste de Sistema: mensura o sistema em diversos cenários, a fim de aferir se todos os seus elementos foram integrados devidamente e se estão realizando suas funcionalidades corretamente (HIRAMA, 2011).
- Teste de Aceitação: simula operações de rotina de uso do sistema, verificando se o comportamento desse está de acordo com o esperado. Geralmente sua realização é feita por um grupo de usuários finais do sistema (NETO, 2008).
- Teste de Regressão: executa, em uma nova versão, dos testes dos principais fluxos, já aplicados no sistema em versões mais antigas, verificando se este continua funcionando corretamente (NETO, 2008).

Tendo isto em vista, os diferentes níveis de teste que podem ser realizados no sistema, para auxiliar nestes, há algumas técnicas de testes de software, que podem ser aplicadas a cada um desses diferentes níveis citados anteriormente (NETO, 2008).

As técnicas de teste de software têm sua classificação de acordo com informações empregadas que determinam os requisitos dos testes. Desta maneira, acabam por contemplar as diferentes perspectivas do software impondo a necessidade de definir uma estratégia de testes. As duas técnicas de testes existentes são: funcional e estrutural, que podem ser chamados respectivamente de teste de caixa-preta e teste de caixa-branca (NETO, 2008).

- Teste de caixa-branca: avalia o comportamento interno do sistema. Por meio de casos de teste, garante que todos os caminhos foram testados independentemente pelo menos uma vez, também testam todas as decisões lógicas nos estados verdadeiro e falso, executam os ciclos em seus limites e testam as estruturas de dados internas assegurando sua validade (PRESSMAN; MAXIM, 2016).
- Teste de caixa-preta: foco nos requisitos funcionais do software. Esta técnica permite considerar múltiplas condições de entrada que utilizem todos os requisitos funcionais do sistema. O teste de caixa-preta tem por objetivo encontrar erros de funções, interface, estrutura de dados ou acesso a base de dados externos, comportamentais. Por meio das técnicas de caixa-preta é gerado uma coleção de casos de testes, que irão satisfazer os critérios funcionais do software (PRESSMAN; MAXIM, 2016).

Dentro do contexto de testes funcionais, temos os testes manuais e os testes automatizados de software. Na sequência serão tratados com mais enfoque esses testes, sendo respectivamente, apresentados os testes funcionais manuais e teste funcionais automatizados.

2.3.1 Testes funcionais manuais

Em geral, os testes funcionais têm por objetivo testar os requisitos que compõem a funcionalidade do software como um todo, por meio de condições de entrada e resultados esperados. Essas condições são transformadas em casos de testes e documentadas dentro de um plano de testes (CRESPO et al, 2004).

A execução dos testes manuais é feita por meio da execução dos casos de testes, propostos no plano de teste, onde ao término de cada execução é gerado um relatório em forma de evidência, cujo resultado é apresentado nos testes executados (CRESPO et al, 2004). Ao término da execução dos testes, pode-se perceber duas condições distintas: na primeira, as funcionalidades do sistema estão de acordo com o especificado, sendo aceito; e na segunda, há a identificação de uma ou mais falhas que são documentadas, para que posteriormente sejam corrigidas (PRESSMAN; MAXIM, 2016).

Nos testes manuais, também há a execução de testes exploratórios, que consistem em navegar pelo sistema, sem um plano definido, a fim de encontrar alguma inconsistência durante a execução, esse tipo de teste acaba se tornando uma atividade iterativa e empírica de idas e vindas, exploratória, num processo investigativo contínuo (CAETANO, 2008).

Ter um processo de testes manuais bem definido impacta positivamente no resultado de todo o processo de desenvolvimento do software. O processo de automação de testes de software que veremos na sequência, pode ser inserido como uma técnica adicional onde há a agregação de valor no mesmo e este é inserido naturalmente a medida em que há um amadurecimento no processo de testes manuais (CAETANO, 2008).

2.3.2 Testes funcionais automatizados

A automação de testes funcionais é uma técnica de testes que deve ser empregada de forma adicional ao processo de testes manuais, seu objetivo não é invalidar os testes manuais existentes, mas sim agregar valor a estes, reduzindo a execução de testes manuais repetitivos e focando em abordagens de testes complementares, tendo como benefício o aumento na amplitude e profundidade dos testes (CAETANO, 2008).

A automação de testes acaba se tornando uma combinação entre teste e desenvolvimento de software, pois é feita através da criação de scripts de testes, podendo ser encarada como um projeto com características próprias, exigindo um planejamento detalhado (CAETANO, 2008).

Automatizar cem por cento dos testes manuais, nem sempre é a melhor prática. Quando os testes têm uma complexidade alta, ou exigem interações intersistemas, a automação acaba se tornando pouco eficaz, por isso, estes tipos de testes devem permanecer manuais (CAETANO, 2008).

A seleção dos casos de testes a serem automatizados, são agrupados em três áreas distintas:

- *Smoke Tests*: conjunto mínimo de casos de testes, que tem o objetivo de validar um *build* ou liberação, antes de um novo ciclo de testes;
- Teste de Regressão: casos de testes que tem o objetivo de executar retestes de uma funcionalidade ou da aplicação como um todo;
- Funcionalidades Críticas: casos de teste que contém funcionalidades críticas ao sistema, que possam trazer riscos de negócio.

Os testes são automatizados do ponto de vista da interface gráfica e com o objetivo de automatizar as ações dos usuários finais, desta maneira é essencial a incorporação de

testabilidade às aplicações. Essa testabilidade refere-se a atributos que determinam a capacidade de uma aplicação ser testada (CAETANO, 2008).

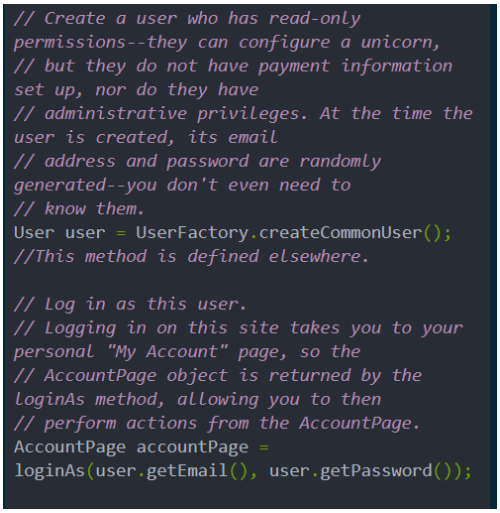
2.3.2.1 Ferramentas de testes automatizados

A realização da automação de testes dispõe de diversos *frameworks* que auxiliam no processo. *Frameworks* que auxiliam na execução de cenários de testes, utilizam da metodologia *Behavior Driven Development* (BDD), a fim de escrever testes dirigidos ao comportamento do usuário. A principal ferramenta, que implementa o BDD aos testes automatizados, é o *Cucumber* utilizando a linguagem *Gherkin* (WYNNE; HELLESØY, 2012; GRAPE, 2016).

A escrita dos scripts de automação de testes pode ser feita em diversas linguagens de programação e seus respectivos *frameworks* de automação de testes, algumas dessas ferramentas, como o *Capybara*, *Watir*, *DalekJS* e *Selenide*, têm algo em comum, que é a utilização da API (*Application Programming Interface*) do *Selenium Webdriver*, é ele que conduz um navegador nativamente e emula a interação do usuário com o navegador. Esse permite a emulação de atividades comuns de usuários finais. Ele é usado principalmente para automação de testes de *front-end* e a escrita dos testes geralmente segue três etapas:

- Configurar os dados: Nesta etapa são definidos os dados que serão utilizados para o cenário de teste (SELENIUM.DEV, 2021). A Figura 2 apresenta um exemplo de uma configuração dos dados. Essa configuração se faz através dos parâmetros dos casos de testes, de maneira onde se cria um usuário com um tipo específico de permissão, *read-only*, sem informações de pagamento e com e-mail e senha criados randomicamente, desta maneira, nunca se saberá de fato essa informação.

Figura 2 - Exemplo de criação de dados com *Selenium Webdriver.*



```
// Create a user who has read-only
permissions--they can configure a unicorn,
// but they do not have payment information
set up, nor do they have
// administrative privileges. At the time the
user is created, its email
// address and password are randomly
generated--you don't even need to
// know them.
User user = UserFactory.createCommonUser();
//This method is defined elsewhere.

// Log in as this user.
// Logging in on this site takes you to your
personal "My Account" page, so the
// AccountPage object is returned by the
loginAs method, allowing you to then
// perform actions from the AccountPage.
AccountPage accountPage =
loginAs(user.getEmail(), user.getPassword());
```

Fonte: SELENIUM.DEV, 2021.

- Realizar um conjunto discreto de ações: nesta etapa, são realizadas uma série de ações do ponto de vista do usuário final, dentro do contexto das páginas do sistema (SELENIUM.DEV, 2021). A Figura 3 apresenta um exemplo de uma automação de ações, nesta etapa são atribuídas informações ao objeto unicórnio, posteriormente há a navegação até a página de configuração de unicórnio, onde é feita criação do unicórnio, passando o objeto previamente criado, através do preenchimento de um formulário e clicando em *submit*.

Figura 3 - Exemplo da automação das ações.

```
// The Unicorn is a top-level Object--it has
attributes, which are set here.
// This only stores the values; it does not
fill out any web forms or interact
// with the browser in any way.
Unicorn sparkles = new Unicorn("Sparkles",
UnicornColors.PURPLE,
UnicornAccessories.SUNGLASSES,
UnicornAdornments.STAR_TATTOOS);

// Since we are already "on" the account
page, we have to use it to get to the
// actual place where you configure unicorns.
Calling the "Add Unicorn" method
// takes us there.
AddUnicornPage addUnicornPage =
accountPage.addUnicorn();

// Now that we're on the AddUnicornPage, we
will pass the "sparkles" object to
// its createUnicorn() method. This method
will take Sparkles' attributes,
// fill out the form, and click submit.
UnicornConfirmationPage
unicornConfirmationPage =
addUnicornPage.createUnicorn(sparkles);
```

Fonte: SELENIUM.DEV, 2021.

- Avaliar os resultados: Neste passo é feita a certificação de que as ações realizadas anteriormente realmente funcionaram (SELENIUM.DEV, 2021). A Figura 4 apresenta um exemplo de uma certificação da funcionalidade, através de uma assertiva onde é passado um objeto e uma condição e espera-se que o retorno dessa seja *true*.

Figura 4 - Exemplo de certificação acerca da funcionalidade.

```
// The exists() method from
UnicornConfirmationPage will take the
Sparkles
// object--a specification of the attributes
you want to see, and compare
// them with the fields on the page.
Assert.assertTrue("Sparkles should have been
created, with all attributes intact",
unicornConfirmationPage.exists(sparkles));
```

Fonte: SELENIUM.DEV, 2021.

Atualmente, alguns dos *frameworks* para o auxílio da automação de testes utilizados, são o *Selenide*, *Capybara* e *Cypress* (SEMPRE IT, 2021).

Selenide: O *Selenide* é uma estrutura para automação de testes escrita em *Java*, construída em cima do *Selenium Webdriver*, foi criado para resolver problemas de

instabilidades de testes causados por conteúdo dinâmico, *Javascript*, Ajax, tempo limite, etc. Tem métodos para controles operacionais como *textfield*, *radiobutton*, *selectbox* e pesquisa de elementos por texto, e também tem métodos para achar elementos, verificar o estado do elemento, métodos-ações sobre o elemento e métodos para obter status de elementos e valores atribuídos (SELENIDE.ORG, 2021). A Figuras 5 apresenta um exemplo de automação de um formulário, onde a o preenchimento de um nome e a seleção de campos como *Radio Buttons* e caixas de seleção por opção e por texto.

Figura 5 -Exemplos métodos convenientes para controles operacionais.

```
@Test
public void canFillComplexForm() {
  open("/client/registration");
  $(By.name("user.name")).val("johny");
  $(By.name("user.gender")).selectRadio("male");
  $("#user.preferredLayout").selectOption("plain");
  $("#user.securityQuestion").selectOptionByText("What is my first car?");
}
```

Fonte: SELENIDE.ORG, 2021.

A Figura 6 apresenta um exemplo do método utilizado para achar um elemento em tela, utilizando os atributos *id* (representados pelo "#") e classe, (representado pelo ".") dos elementos.

Figura 6 - Exemplo método achar elemento.

```
$("#header").find("#menu").findAll(".item")
```

Fonte: SELENIDE.ORG, 2021.

A Figura 7 apresenta um exemplo onde há a verificação do estado de um elemento, testando se o elemento existe, se está visível e se ele apresenta o texto informado.

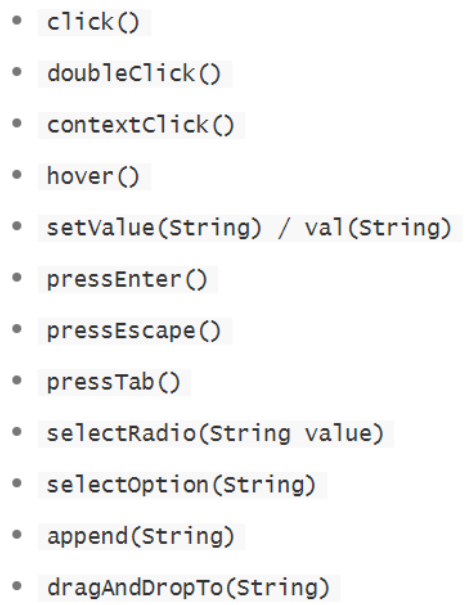
Figura 7 - Exemplo método para verificar estado do elemento.

```
$("input").should(exist);
$("input").shouldBe(visible);
$("input").shouldHave(exactText("Some text"));
```

Fonte: SELENIDE.ORG, 2021.

A Figura 8 apresenta alguns exemplos de ações que podem ser realizadas na automação, como cliques, duplos cliques, *hover*, pressionar *enter*, seleção de *Radio Button*, seleção de opções e etc.

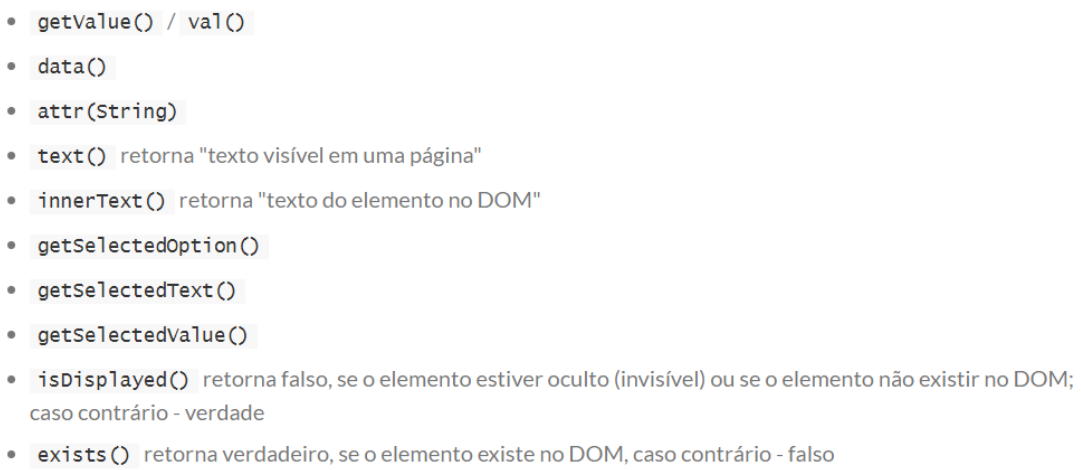
Figura 8 - Exemplo método-ação sobre elementos.



Fonte: SELENIDE.ORG, 2021.

A Figura 9 apresenta exemplos do uso de métodos para obter o *status* e valores atribuídos aos elementos.

Figura 9 - Exemplos métodos para obter *status* dos elementos e valores atribuídos.



Fonte: SELENIDE.ORG, 2021.

Capybara: O *Capybara* é uma biblioteca escrita em Ruby que facilita a simulação da interação do usuário com a aplicação. Pode se comunicar com diversos *drivers* que executam os testes através de uma interface limpa e simples. Dentre os *drivers* que o *Capybara* usa estão o *Selenium Weddriver* e *Webkit*. Vem com suporte a rack::Teste (*Ruby* puro) e *Selenium* embutidos, no entanto o *Webkit* é suportado através de uma *gem* externa. Com esse, é possível interagir com a aplicação seguindo links e botões, seguindo automaticamente quaisquer redirecionamentos e enviando formulários associados a botões, há também as possibilidades de interação com formulários, opções para consultar, achar e manipular certos elementos na página. (TEAMCAPYBARA, 2021). A Figura 10 apresenta exemplos de interações de cliques em elementos, sejam clique em *links* e em um botão com um determinado texto.

Figura 10 - Exemplo clicando em links e botões.

```
click_link('id-of-link')
click_link('Link Text')
click_button('Save')
click_on('Link Text') # clicks on either links or buttons
click_on('Button Value')
```

Fonte: TEAMCAPYBARA, 2021.

A Figura 11 apresenta um exemplo de preenchimento de campos através do *fill_in*, ou seleção de opções de *Radio Button*, *checkboxes*, anexo de arquivos e seleção de múltiplas opções através de um *select*.

Figura 11 - Exemplo interagindo com formulários.

```
fill_in('First Name', with: 'John')
fill_in('Password', with: 'Seekrit')
fill_in('Description', with: 'Really Long Text...')
choose('A Radio Button')
check('A Checkbox')
uncheck('A Checkbox')
attach_file('Image', '/path/to/image.jpg')
select('Option', from: 'Select Box')
```

Fonte: TEAMCAPYBARA, 2021.

A Figura 12 mostra exemplos de consulta de elementos, se estes existem em tela, através de atributos como *xpath*, *css*, ou algum conteúdo específico.

Figura 12 - Exemplo de consulta de elementos.

```
page.has_selector?('table tr')
page.has_selector?(:xpath, './/table/tr')

page.has_xpath?('.//table/tr')
page.has_css?('table tr.foo')
page.has_content?('foo')
```

Fonte: TEAMCAPYBARA, 2021.

A Figura 13 apresenta exemplos de como achar e manipular os elementos através do *find*, esses elementos podem ser encontrados através de atributos como *id's*, classes, *placeholders*, texto e após serem encontrados, pode-se realizar ações como cliques, obter o valor atribuído a ele e verificar sua visibilidade.

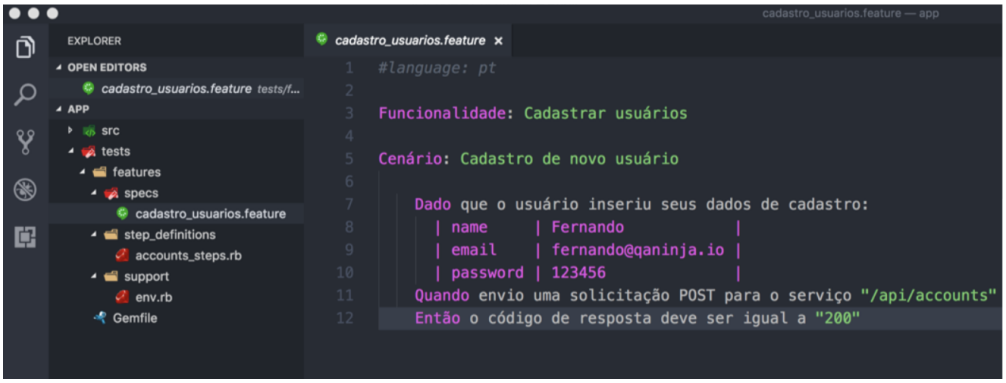
Figura 13 - Exemplo de achar e manipular elementos.

```
find_field('First Name').value
find_field(id: 'my_field').value
find_link('Hello', :visible => :all).visible?
find_link(class: ['some_class', 'some_other_class'], :visible => :all).visible?

find_button('Send').click
find_button(value: '1234').click

find(:xpath, ".//table/tr").click
find("#overlay").find("h1").click
all('a').each { |a| a[:href] }
```

Fonte: TEAMCAPYBARA, 2021.

Um exemplo do funcionamento desta ferramenta, *Capybara*, é representado nas Figuras 14 e 15, desde a escrita de um caso de teste, utilizando o *Cucumber*, até a escrita dos *scripts* de teste, usando o padrão *PageObject* (PAPITO, 2017). A Figura 14 apresenta um exemplo de escrita de um caso de teste da funcionalidade de cadastro de um usuário utilizando o *Cucumbe*r, onde esse guiará a automação de testes.

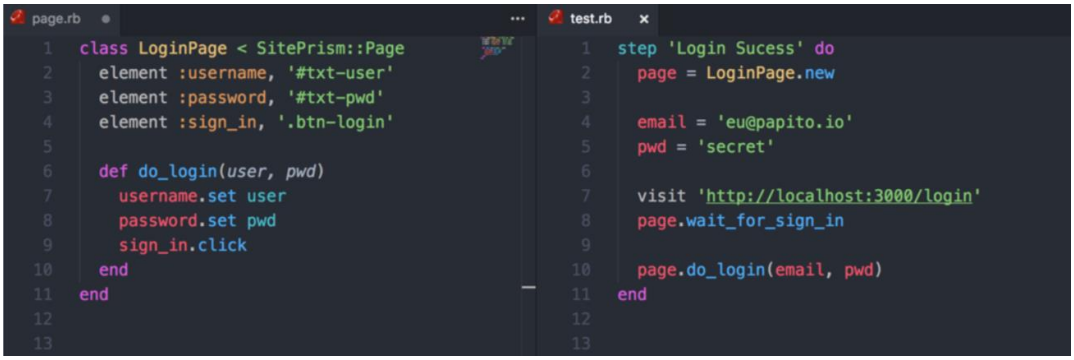
Figura 14 - Exemplo caso de teste com *Cucumber*.



Fonte: PAPITO, 2017.

A Figura 15 mostra um exemplo da escrita dos *scripts* de testes onde há um arquivo *page* com a classe de *LoginPage* e um segundo arquivo onde há os passos que serão percorridos. No arquivo *page* é feito todos os métodos de interação com o sistema enquanto no arquivo de passos é feita as chamadas desses métodos presentes no arquivo *page*.

Figura 15 - Exemplo escrita de scripts de teste em *Ruby* e *Capybara*.



Fonte: PAPITO, 2017.

Cypress: O *Cypress* é uma ferramenta de automação de testes, que diferente do *Selenide* e *Capybara*, não utiliza o driver de execução do *Selenium Webdriver*. Desta maneira, enquanto o *Selenium Webdriver* executa comandos remotos através da rede, no *Cypress* é executado no mesmo *loop* de execução que a aplicação. Os testes são escritos em *JavaScript* fazendo com que o código de testes seja compilado dentro do próprio navegador, não tendo uma vinculação de idiomas ou *drivers*, havendo apenas o *JavaScript* (CYPRESS.IO, 2021).

O *Cypress* permite a realização de testes de ponta a ponta, tendo os testes executados dentro do navegador, executa as ações através da interface do usuário, da mesma forma que o usuário real (CYPRESS.IO, 2021). A Figura 16 apresenta um exemplo de utilização do Cypress, onde há a interação com elementos através do *get*, desta maneira há o preenchimento de uma lista e por fim são feitas interações de *check* e clique em tela.

Figura 16 - Exemplo de utilização do *Cypress*.

```
describe('TodoMVC', function () {
  beforeEach(function () {
    cy.visit('http://localhost:8888/')

    cy.get('.new-todo')
      .type('buy some cheese {enter}')
      .type('feed the cat {enter}')
      .type('book a doctors appointment {enter}')
  })

  it.only('hides "Clear Completed" with nothing checked', function () {
    cy.get('.todo-list li').eq(1).find('.toggle').check()
    cy.get('.clear-completed').should('be.visible').click()
    cy.get('.clear-completed').should('not.exist')
  })
})
```

Fonte: CYPRESS.IO, 2021.

3 REVISÃO DA LITERATURA

Para a análise de trabalhos relacionados ao contexto desta pesquisa, utilizou-se a plataforma *Google Scholar*, onde foram encontrados trabalhos referentes ao tema proposto. Dentre os trabalhos encontrados, cinco deles foram selecionados para realizar uma análise sobre o tema. A análise dos trabalhos relacionados objetiva identificar os principais temas abordados a fim de encontrar critérios para a realização da análise das ferramentas de automação de testes. O Quadro 1 apresenta a relação dos trabalhos selecionados para a análise e discussão.

Quadro 1 - Pesquisas sobre o uso de ferramenta de automação de testes.

Referência	**Título**	**Local**	**Tipo de produção**
(MOBARAYA & ALI, 2019)	*Technical Analysis of Selenium and Cypress as functional automation framework for modern web application testing*	*AGI Institute*	Artigo
(OKEZIE, 2019)	*A Critical Analysis of Software Testing Tools*	*Journal of Physics Conference Series*	Artigo
(GRAPE, 2016)	*Comparing Costs of Browser Automation Test Tools with Manual Testing*	*Linköpings Universitet*	Dissertação
(KASTEGÅRD, 2015)	*Automated testing of a web-based user interface*	*Institutionen för datavetenskap*	Dissertação
(SHETH & SINGH, 2015)	*Software Test Automation-Approach on evaluating test automation tools*	*International Journal of Scientific and Research Publications*	Artigo

Fonte: Elaborado pela autora, 2022.

3.1 ANÁLISE DOS ESTUDOS SELECIONADOS

O artigo de Sheth & Singh (2015) tem por objetivo examinar e estabelecer um mecanismo que avalie efetivamente ferramentas de automação de teste, através do desenvolvimento de métricas a serem usadas na avaliação dessas ferramentas e da comparação entre resultados obtidos dos testes manuais e automatizados. No artigo, antes de realizarem a automação dos casos de teste, estes foram testados primeiramente de forma manual e

posteriormente, como parte de um teste de regressão, os mesmos casos foram codificados em *scripts* de teste e executados.

As ferramentas de automação de testes, selecionadas pelos autores foram:

- RFT: é uma ferramenta que simula os passos do usuário final da aplicação e compara recursos entre as ferramentas e realiza os testes em ambientes web.
- *Janova*: é uma ferramenta baseada na nuvem, com suporte a testes de ambientes web.
- *Ranorex*: é uma ferramenta muito usada para testes em ambiente web.

Para a comparação dessas ferramentas, foram utilizados os seguintes critérios de comparação: funcionalidades, facilidade na depuração, progresso de automação, suporte ao processo de teste, usabilidade e requisitos. As ferramentas também foram avaliadas em relação ao suporte *web*. Com os critérios de comparação definidos, a primeira métrica avaliou as funcionalidades que cada ferramenta apresentou. A segunda métrica a ser aplicada foi a usabilidade, que avaliou desde a instalação da ferramenta, interface do usuário, ajuda com erros até tutoriais de como usar a ferramenta.

A pesquisa concluiu que as ferramentas de teste são muito diferentes entre si e o uso dessas depende do contexto de uso. O autor conclui que o RFT é uma ótima ferramenta para testes de regressão, enquanto *Janova* é a melhor ferramenta devido a ser baseada na nuvem e pode ser acessada de qualquer máquina de teste e *Ranorex* é a melhor ferramenta para testes *web*. No entanto, os autores recomendam que as métricas utilizadas no artigo, sejam aplicadas na avaliação de outras ferramentas de teste que existem no mercado.

A dissertação de Kastegård (2015) investigou como a automação de testes de sistemas web pode ser implementada. Para isso, testa métodos e uma seleção de *frameworks* e ferramentas relevantes avaliados de acordo com os requisitos propostos. Dentre os requisitos levantados pela autora estão: a gratuidade do *framework* ou ferramenta, a possibilidade de realizar testes de caixa preta e a simulação do comportamento do usuário, a não dependência de outros *frameworks* ou ferramentas, a habilidade de desenvolver *scripts* de teste do zero e a habilidade de escrever testes em *Java*. A autora lista alguns *frameworks* e ferramentas de automação, sendo estes:

- *Selenium WebDriver*: uma ferramenta bem documentada e o framework do *WebDriver* é utilizado como base para diversas outras ferramentas de automação de testes.
- *Sahi*: uma ferramenta que apresenta uma versão grátis e uma versão pro, onde os testes são configurados através de uma interface gráfica.

- *DalekJS*: é uma ferramenta *open source* baseada em *NodeJS*, que utiliza o *JavaScript* para programar os testes.
- *Jasmine*: é uma ferramenta com foco em testes de código *JavaScript*, não tendo suporte a testes em ambiente web.

Por fim, a escolha final da autora foi o *Selenium Webdriver* que acabou suprindo todos os requisitos listados.

A pesquisa de Grape (2016) teve como objetivo a comparação de custo entre a execução de testes automatizados e testes manuais. O autor apresenta as vantagens na realização dos testes automatizados que são: o tempo de execução dos testes que acaba sendo menor em relação ao manual, o reuso de código ao executar novamente casos de teste, executar casos de teste padrão ao realizar testes de regressão e validar se o sistema está funcionando de acordo com o esperado. Por outro lado, o autor também apresentou as desvantagens dos testes automatizados que são: o custo dos testes automatizados, seja em licença de *software* como em aprendizado de uma ferramenta e criação e manutenção dos testes, nem todo caso de teste é adequado para automação e que os testes automatizados não acharão todos os bugs existentes em um sistema.

O autor apresenta uma série de ferramentas de automação de testes e suas particularidades, essas ferramentas são:

- *Selenium WebDriver*: uma ferramenta *open source* implementada na *WebDriver API* e que possibilita a escrita de scripts em diversas linguagens de programação.
- *Watir Webdrive*: também implementado em cima da *WebDriver API* e *open source* tendo como linguagem de programação o *Ruby*.
- *DalekJS*: outra implementação *WebDriver* mas baseada em *NodeJs open source* com linguagem de programação *JavaScript* e *CoffeeScript.*
- *CasperJS*: construído em cima do navegador *headless PhantomJS*, que por sua vez, suporta *WebDriver*. É uma ferramenta *open source* com linguagem de programação *JavaScript* e *CoffeeScript.*
- *Capybara*: ferramenta de testes de aceitação *open source* para aplicações *web* escrito em *Ruby*, que pode utilizar mais de um drive de teste.
- *Robot Framework*: framework *open source* de testes de aceitação genérico, que usa a abordagem *KeyWord-Driven Testing*, construído em Python, utiliza diferentes *test drivers*, incluindo uma biblioteca *WebDriver-based*.

Para o trabalho, Grape escolheu utilizar as ferramentas *Selenium WebDriver* com *Python*, *Capybara* e *Cucumber*, a fim de abordar o BDD, junto com o *test drive* do *Selenium*

WebDriver e o *Robot Framework*, com a finalidade de abordar o *KeyWord Driven Testing*, utilizando a biblioteca *Selenium2Library*. Para o estudo de caso, os testes foram conduzidos na aplicação *StoredSafe*. Durante essa fase é apresentado como os casos de testes foram executados e como os dados foram coletados, sendo o estudo de caso dividido nas etapas de implementação, execução e manutenção do código.

A fim de testar a aplicação, uma série de casos de testes foi definida baseada em suas especificações, para que juntas possam ser um cenário de caso de uso, emulando um conjunto de interações com a aplicação. A fase da implementação percorre dois passos. O primeiro passo inclui o aprendizado de como utilizar os *frameworks* selecionados e implementar dois casos de teste. Após o primeiro passo foram implementados mais nove casos de testes para cada *framework* em ordens misturadas, fazendo com que cada um dos *frameworks* passasse a ser o primeiro, segundo e terceiro a ser implementado.

Para o desenvolvimento dos casos de teste em *Selenium WebDriver* e *Capybara* o autor usou o padrão *PageObject* a fim de facilitar a interação com a aplicação web e apontou que o *Robot Framework* apresentou dificuldades em adaptar o *page objects* à estrutura de teste, na verdade, o gerenciamento de *web pages* específicas e elementos específicos, são acumulados em arquivos de recursos específicos, a fim de deixar o projeto enxuto.

Para mensurar o tempo, Grape (2016) calculou os tempos que levou usando cada um dos três *frameworks*, ele mensura esse tempo em: tempo de aprendizado, *set up* e implementação de casos de testes e o tempo que leva entre a execução dos testes manualmente e automaticamente. A mensuração do custo de execução, foi calculado o tempo de execução, para isso, todos os *frameworks* e testes manuais foram executados no mesmo computador usando o mesmo navegador. Em relação aos custos de manutenção, o autor não conseguiu obter resultado, devido ao sistema não ter recebido atualizações durante o processo. No entanto, foi feito uma estimativa desses custos baseando-se nos custos de implementação.

A fim de apresentar os resultados obtidos, o autor apresenta três modelos econômicos que são usados para calcular o impacto econômico da automação dos casos de teste. Os modelos apresentados são: *Ramler* e *Wolfmaier, Hauptmann* et al. e *Cui e Wang*. Cada um destes modelos apresenta um cálculo a fim de mensurar o impacto econômico, cálculos apresentados e aplicados pelo autor. Através do resultado obtido pelos cálculos realizados, onde se tem o número de vezes que é necessário à execução de um caso de teste para quebrar os testes automatizados mesmo com testes manuais.

Okezie (2019) conduziu um estudo a fim de examinar diversas ferramentas de automação de testes, baseadas em métricas de usabilidade como facilidade de uso, suporte

técnico e facilidade de instalação e configuração. No decorrer do estudo, a autora faz uma avaliação acerca das ferramentas de automação de testes que considera mais relevantes, identificando suas diferenças e fazendo uma comparação entre elas, apresentando seus pontos fortes e pontos fracos, para, por fim, tecer suas recomendações. As ferramentas apresentadas pela autora são:

- *Selenium Webdriver*: uma ferramenta *open source* que performa testes baseados em web, funciona com diferentes navegadores e sistemas operacionais e pode ser composta por diferentes linguagens de programação.
- *Test Complete*: uma ferramenta de teste utilizada para testes funcionais. Permite a criação de testes para aplicações *Windows*, *Web*, *Android* e *iOS*.
- *Ranorex*: uma ferramenta de testes, que aplica testes do ponto de vista do usuário, tem compatibilidade com aplicações desktop, web e mobile. É capaz de conduzir testes de regressão e garante a reusabilidade das atividades de teste.
- *Appium*: uma ferramenta *open source* que realiza testes de aplicações *mobile*, tem suporte *cross-platform*, possibilitando a escrita dos testes tanto para Android quanto para *iOS*.
- *Quick Test Professional*: uma ferramenta de testes baseada no *Windows*, usada para testar aplicações *desktop* e *web*. Oferece automação de testes de regressão e funcional.
- *OpenScript*: uma IDE de testes baseada no Eclipse que ajuda a criação de testes funcionais.
- *Janova*: uma ferramenta baseada na web, que executa os testes na nuvem. Não há necessidade de escrita de scripts de testes. É uma ferramenta rápida e simples devido a utilização na nuvem.
- RFT: uma ferramenta de testes que envolve testes de regressão, funcionais e de interface, trabalha com Java e realiza os testes em ambientes *web*.

Junto a essas ferramentas também são apresentados os requisitos que a autora acha pertinente para a escolha de uma ferramenta de automação de testes, essas são: *open source*, necessidade de licença, suporte *mobile*, suporte *web*, suporte *desktop*, facilidade de aprendizado e uso, habilidade de programação, reusabilidade de código, resultado dos testes, gravação e *playback*.

Dentre as ferramentas listadas, apenas três são *open source*, sendo elas: *Selenium*, *Appium* e RFT. Dentre as três, apenas duas suportam automação de teste *web*, *Selenium* e RFT.

Os critérios de comparação selecionados pela autora, foram: se a ferramenta é *open source*, se é licenciada, o suporte a plataformas *mobile*, suporte a plataforma *web*, a facilidade de uso e aprendizado, habilidade de programação, reusabilidade de código, *report* de resultado de testes e gravação e reexecução. Por fim é concluído que todas as ferramentas de automação de testes são eficientes, mas dependem do contexto, algumas tendem a ser mais eficientes do que outras.

O artigo de Mobaraya e Ali (2019) apontou o quão complexo os sistemas web estão se tornando, devido a digitalização da informação. Desta forma, testar esses sistemas modernos, acaba se tornando um desafio e a automação de testes surge utilizando de ferramentas que replicam o comportamento do usuário real, executando testes autonomamente, com o objetivo de reduzir o tempo de execução de testes e aumentar a cobertura desses. Os autores citam o *Selenium* como a ferramenta de automação de testes mais utilizada atualmente, porém indica que essa ferramenta foi desenvolvida em 2005, numa época em que os sistemas web eram muito mais simples, fazendo com que o *Selenium* possa ter algum problema em lidar com elementos dinâmicos, que acaba reduzindo a performance da execução dos testes. Dessa maneira, em resolução ao problema, é proposto a apresentação do framework de automação *Cypress*, que atende a automação dos sistemas web dinâmicos. Os autores denotam que a vantagem desse framework é a simplificação de testes assíncronos.

A pesquisa de Mobaraya e Ali (2019) tem por objetivo: criar *scripts* de automação em *Selenium* e *Cypress*, desenvolver testes de regressão, comparar a execução dos testes em *Selenium* e *Cypress* e comparar a eficiência dos testes nos dois *frameworks*, utilizando o sistema AliExpress como objeto de estudo, automatizando duas funcionalidades chave, a conta do usuário e o acompanhamento de pedidos, tendo um total de dez casos de teste, cinco para cada funcionalidade.

Os autores escolheram três métricas como indicador à pesquisa, essas métricas são: tempo total de execução dos testes, eficiência de execução dos testes e requisitos para a cobertura dos testes. O tempo total de execução dos testes ajuda a observar o progresso da execução dos testes no geral. A eficiência dos testes mede a efetividade do custo dos testes, o esforço total na escrita dos scripts de teste. Os requisitos para a cobertura dos testes medem o número de requisitos que foram cobertos pelos casos de teste, se a cobertura de testes não for 100%, significa que há buracos nos testes e são necessários mais casos de teste para cobrir todos os requisitos.

A pesquisa conclui que através das métricas de teste coletadas, não houve muita diferença de tempo total de execução entre *Selenium* e *Cypress*. Da parte do *Selenium*, a maior

parte do tempo consumido na execução foi ao instanciar o *WebDriver* e inicializar a aplicação. No entanto, há uma diferença significativa na eficiência entre *Selenium* e *Cypress*, onde o *Cypress* produziu 45 linhas de código a menos que o *Selenium* nos testes de conta de usuário e 69 linhas a menos nos testes de acompanhamento de pedidos, indicando que a escrita de scripts de teste em *Cypress*, necessita de menos esforço, aumentando a eficiência. Para a última métrica de testes requisitos para a cobertura dos testes, tanto *Selenium* quanto *Cypress* cobriram 100% dos casos de teste. Foi detectado uma limitação no *Cypress*, que é a capacidade de abrir e navegar entre novas abas. Enquanto o *Selenium* consegue contornar esse problema ao usar comando do teclado para alternar entre as abas, *Cypress* acaba não tendo suporte para isso. Outras dificuldades encontradas no *Cypress* foram, acesso de segurança da *web* restrito e dificuldade na interação de elementos escondidos em *iFrames*. Também foi descoberto que no *Selenium* a execução dos casos de teste é feita por ordem alfabética, se não houver uma priorização nos casos de teste, enquanto no *Cypress* os testes são executados em sequência. Os comandos em *Cypress* são mais legíveis em relação ao *Selenium*, no entanto, é necessário um conhecimento mais técnico para entender os comandos usados no *Cypress*. Também se apontou que *Cypress* se torna mais fácil ao ser depurado, ao indicar, no código, onde há ocorrência de erros.

A pesquisa conclui que apesar do *Selenium* ser uma ferramenta de testes poderosa com uma grande comunidade e suporte, o *Cypress* traz uma visão promissora do futuro da automação de testes. Sendo ele um framework significantemente fácil e simples de configurar, e produzindo um código melhor e mais limpo.

3.2 DISCUSSÃO

Dentre os cinco trabalhos selecionados na literatura, três deles convergiram em relação ao que tange as métricas de comparação das ferramentas de automação de testes analisadas (Mobaraya & Ali, 2019; Grape, 2016; Sheth & Singh, 2015). Por outro lado, outros dois trabalhos (Okezie, 2019; Kastegård, 2015) tratam de requisitos pré-estabelecidos para o auxílio na escolha de uma ferramenta de automação de testes.

Em seu artigo Mobaraya e Ali (2019) indicam três métricas de comparação: tempo total de execução, eficiência de execução dos testes e requisitos para a cobertura dos testes. Grape (2016), em sua dissertação, lista as seguintes: tempo, custo de execução, custo de manutenção. Em seu artigo Sheth e Singh (2015) apresentam-as como: funcionalidades que cada ferramenta apresentou e a usabilidade. Pode-se observar que dentre as métricas listadas por cada autor,

Mobaraya & Ali (2019) e Grape (2016) entendem que o tempo de execução e realização são pertinentes aos testes, os dois autores também apresentam métricas relacionadas ao custo e eficiência da execução dos testes, concordando que é algo a se ter em vista. Sheth e Singh (2015) apresentam a usabilidade, que pode ser relacionada tanto ao tempo de execução e realização dos testes quanto a execução desses, pois essas avaliam o custo e tempo de instalação da ferramenta, o quão amigável e fácil de compreender a ferramenta é e a facilidade de acesso a documentação dessa.

No que se relaciona aos requisitos pré-estabelecidos para o auxílio na escolha de uma ferramenta de automação de teste, em seu artigo Okezie (2019) lista os seguintes requisitos: se é *open source*, necessidade de licença, suporte mobile, suporte web, suporte desktop, facilidade de aprendizado e uso, habilidade de programação, reusabilidade de código, resultado dos testes, gravação e playback dos testes. Já os requisitos apresentados por Kastegård (2015) em sua dissertação são: *frameworks* ou ferramentas gratuitas, a possibilidade de teste de caixa preta e o uso de simuladores de comportamento do usuário, a não dependência de outros *frameworks* ou ferramentas, habilidade de desenvolver casos de teste do zero e habilidade de escrever testes em Java. Esses convergiram no quesito de a ferramenta ser *open source* ou licenciada, também podemos relacionar o requisito onde é necessário a habilidade de programação que Okezie (2019) indica e a habilidade de escrever testes em *Java*, apontados por Kastegård (2015). De acordo com os requisitos apontados pelas autoras, nota-se que a escolha desses acaba sendo muito mais em relação a uma necessidade particular e ao que cada uma buscava em uma ferramenta ou *framework* de automação de testes. No Quadro 2 é apresentado a comparação das métricas e requisitos que convergiram entre os autores.

Quadro 2 - Comparação dos critérios utilizados pelos autores

(continua)

Critérios/ Trabalhos	**MOBARAYA; ALI, 2019**	**OKEZIE, 2019**	**GRAPE, 2016**	**KASTEGÅRD, 2015**	**SHETH; SINGH, 2015**
Tempo	✓		✓		✓
Custo	✓		✓		✓

Quadro 2 - Comparação dos critérios utilizados pelos autores

(conclusão)

Open Source		✓		✓	
Scripts de testes				✓	
GUI		✓			✓

Fonte: Elaborado pela autora, 2022.

Em síntese, a análise desses trabalhos relacionados com ênfase nos critérios utilizados para a comparação das ferramentas e *frameworks* de automação de testes, contribuem nas definições para a sequência da pesquisa.

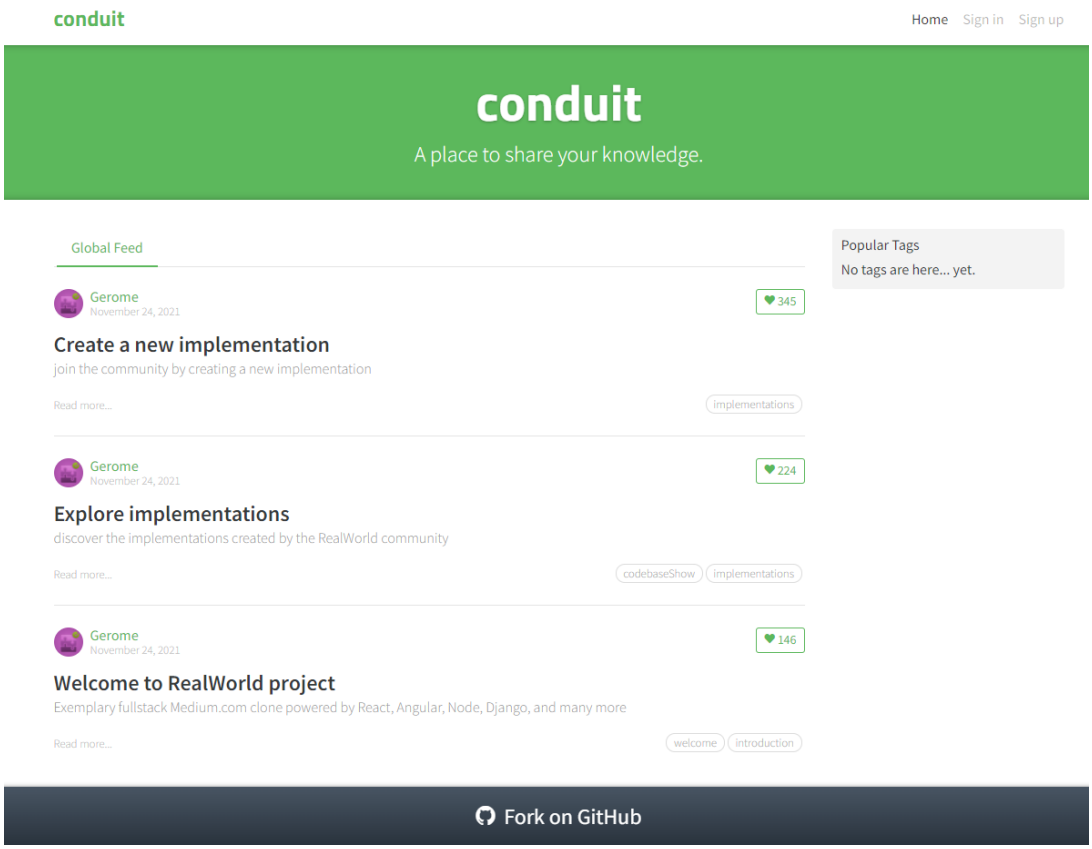
4 AVALIAÇÃO DE FERRAMENTAS DE TESTES AUTOMATIZADOS

Neste capítulo é realizada uma avaliação e comparação entre os *frameworks Capybara* e *Cypress* no contexto de automação de testes *end to end* em aplicações *web*, de modo a identificar as vantagens e desvantagens dos referidos *frameworks*. A escolha dos dois *frameworks* foi baseada nos seguintes critérios: tanto *Capybara* quanto *Cypress* são *frameworks open source*, de sintaxe simples e linguagem de tipagem dinâmica, que possibilita a utilização do padrão *page objects* e também a integração ao *cucumber* para escrita e execução de cenários de testes em *Gherkin*. Desta maneira, o *Capybara* que utiliza como linguagem o *Ruby* será configurado para rodar os testes através do *Selenium Webdriver* e o *Cypress* escrito em *JavaScript* terá os testes executados localmente através do navegador.

4.1 APLICAÇÃO DE TESTE

Para realização da comparação, foi selecionado o sistema *web Conduit* (demo.realworld.io/), um sistema construído com a finalidade de ser um clone do sistema *Medium.com* (THINKSTER, 2021). A escolha deste sistema deu-se pelo fato de ser um sistema *open source*, de interface simples e intuitiva e com funcionalidades automatizáveis. O sistema possibilita ao usuário realizar cadastro, autenticação, publicação, visualização e detalhes dos artigos. A Figura 17 apresenta um *screenshot* da página inicial da aplicação *Conduit*.

Figura 17 - Página inicial da aplicação Conduit



Fonte: *Home — Conduit (realworld.io)*

Desta maneira foram identificados alguns casos de testes para cada uma dessas funcionalidades. No Quadro 3 é apresentado um plano de testes para as funcionalidades do sistema, bem como os casos de testes identificados.

Quadro 3 - Plano de testes das funcionalidades do sistema Conduit

(continua)

Funcionalidade	**Caso de Teste**
Cadastro de usuário	*Username* inválido
	Email inválido
	Password inválido
	Cadastro de usuário com sucesso
Autenticação de usuário	*Email* inválido
	Password inválido
	Autenticação com sucesso

Quadro 3 - Plano de testes das funcionalidades do sistema *Conduit*

(conclusão)

Publicar artigos	Sem preencher campo obrigatório - Título
	Sem preencher campo obrigatório - Descrição
	Sem preencher campo obrigatório - Corpo
	Publicar artigo com sucesso

Fonte: Elaborado pela autora, 2022.

4.2 DEFINIÇÃO DA AVALIAÇÃO DOS FRAMEWORKS CAPYBARA E CYPRESS

Para a realização da avaliação de comparação, é adotada a abordagem *Goal - Question - Metric* (GQM), que consiste em uma abordagem orientada a metas para a mensuração de produtos e processos de software. Suporta a identificação de métricas úteis e relevantes tal qual suporta a análise e interpretação dos dados obtidos (BASILI et al., 1994). Desse modo, com base na abordagem GQM é possível definir o objetivo da avaliação, a partir do objetivo, definir questões de análise e métricas, que serão coletadas e as questões respondidas durante a execução dos projetos de automação.

Seguindo o *template* GQM para definição da meta de avaliação, o objetivo da avaliação é: Analisar os *frameworks Capybara* e *Cypress* com o propósito de identificar vantagens e desvantagens com respeito ao tempo de execução, custo de execução e complexidade de configuração do ponto de vista de testador de software no contexto da aplicação *Conduit*. Estes fatores foram definidos com base nos resultados da revisão da literatura, planejamento dos testes e identificação dos casos de testes. O Quadro 4 apresenta uma definição para os fatores de análise considerados na avaliação.

Quadro 4 - Definição dos fatores de análise considerados na avaliação.

(continua)

Fator de análise	**Definição**
Complexidade de Configuração	Nível de complexidade de configurar as ferramentas e dependências necessárias para a criação e configuração

Quadro 4: Definição dos fatores de análise considerados na avaliação.

(conclusão)

Complexidade de Configuração	de um projeto de automação em cada *framework* e estruturação do projeto
Custo de execução	Quantidade de linhas necessárias para escrita de scripts de testes e o acesso ao log e depuração desses testes
Tempo execução	Refere-se ao tempo de execução de uma *feature* e cenários de teste e execução da automação por completa

Fonte: Elaborado pela autora, 2022.

A partir do objetivo da avaliação, questões de análise e métricas são definidas para cada fator de análise. O Quadro 5 apresenta as questões de análises e as métricas coletadas para respondê-las.

Quadro 5 - Métricas e questões de análise para mensuração.

(continua)

Fator de Análise	**Questão de Análise**	**Métrica**
Complexidade de Configuração	**QA1. Qual a complexidade de configurar um projeto de automação**?	Etapas necessárias para a iniciação e configuração de um projeto de automação com o *framework Capybara*. Etapas necessárias para a iniciação e configuração de um projeto de automação com o *framework Cypress*.
	QA2. Qual a facilidade de estruturação e execução de um projeto no *framework*?	Nível de facilidade de Estruturação do projeto no *framework Capybara*. Nível de facilidade de Estruturação do projeto no *framework Cypress*.

Quadro 5 - Métricas e questões de análise para mensuração.

(continuação)

Custo de execução	**QA3. Quantas linhas de código são geradas por funcionalidade?**	Número de linhas de código gerada em cada funcionalidade executada no *framework Capybara*. Número de linhas de código gerada em cada funcionalidade executada no *framework Cypress*
	QA4. Qual a facilidade de acesso ao log e depuração dos testes?	Nível de facilidade de acesso ao log e depuração dos testes no *framework Capybara*. Nível de facilidade de acesso ao log e depuração dos testes no *framework Cypress*.
Tempo de execução	**QA5. Quanto tempo leva a execução de um cenário de testes de uma *feature*?**	Tempo (em segundos) que levará para a execução de um cenário testes de uma *feature* no *framework Capybara*. Tempo (em segundos) que levará para a execução de um cenário testes de uma *feature* no *framework Cypress*.
	QA6. Quanto tempo leva a execução de todos os cenários de testes de uma *feature*?	Tempo (em segundos) que levará para a execução de todos os cenários testes de uma *feature* no *framework Capybara*. Tempo (em segundos) que levará para a execução de todos os cenários testes de uma *feature* no *framework Cypress*.
	QA7. Quanto tempo leva a execução de todas as *features* de testes?	Tempo (em segundos) que levará para a execução de cenário testes de todas as *features* no framework *Capybara*. Tempo (em segundos) que levará para a execução de cenário testes de todas as

Quadro 5 - Métricas e questões de análise para mensuração.

(conclusão)

		features no framework *Cypress*.

Fonte: Elaborado pela autora, 2022.

Os dados que responderão essas questões são obtidos por meio da configuração, escrita de cenários, escrita de scripts e execução dos testes automatizados, que são apresentados na sequência.

4.3 EXECUÇÃO

Para a implementação dos projetos de automação, é utilizado um notebook com processador Intel i7-7700HG CPU 2.80GHz de 64bits, 8GB de RAM e sistema operacional *Windows 10 Home Single Language*. O acesso à rede se dá de forma cabeada, sendo uma rede fibra óptica de 200MB e todos os testes são executados no navegador *Google Chrome* (versão 96.0). A codificação das automações, é por meio da IDE *VSCode* (*Visual Studio Code)* e é utilizado o *Console Emulator | Cmder* para execução de linhas de comando.

Conforme as definições estabelecidas, nessa etapa é realizada a escrita dos cenários de testes obtidos através do plano de testes elaborados para o sistema *Conduit* (Quadro 3), bem como a configuração dos projetos de automação para os *frameworks Capybara* e *Cypress*, estruturação e pôr fim a escrita dos *scripts* e testes, para cada um dos cenários de testes identificados.

4.3.1 Cenários de teste

Para a escrita dos cenários de teste, é utilizada a abordagem *Behavior Driven Development* (BDD) através da linguagem *Gherkin* e integração com o *framework Cucumber*. Dessa forma, os cenários escritos poderão ser utilizados tanto no projeto de automação em *Capybara*, quanto no projeto de automação em *Cypress*. O Quadro 6 apresenta a relação dos cenários de testes e suas definições para cada uma das *features* identificadas no plano de testes.

Quadro 6 - Relação de cenários de testes identificados por *feature*.

(continua)

Feature	**Definição**	**Cenário**	
Cadastro de usuário	**Como** usuário do sistema **Quero** realizar cadastro **Para** ter acesso as funcionalidades do sistema	Utilizar nome de usuário já utilizado na plataforma	**Dado que** acesso a tela de cadastro do sistema **Quando** informo um usuário inválido **Então** preencho os outros campos com dados válidos **Quando** clico em “Sign up” **Então** vejo erro “username has already been taken”
		Utilizar um e-mail inválido	**Dado que** acesso a tela de cadastro do sistema **Quando** informo um e-mail inválido **Então** preencho os outros campos com dados válidos **Quando** clico em “Sign up” **Então** vejo erro “email has already been taken”
		Não informar uma senha	**Dado que** acesso a tela de cadastro do sistema **Quando** preencho os campos *username* e *e-mail* **E** clico em “Sign up” **Então** vejo erro “password can’t be blank”
		Cadastrar usuário com sucesso	**Dado que** acesso a tela de cadastro do sistema **Quando** preencho todos os campos de cadastro **E** clico em “Sign up” **Então** sou redirecionado ao dashboard do sistema
Autenticação de usuário	**Como** usuário do sistema **Quero** me autenticar **Para** acessar meu perfil de usuário	Inserir um e-mail incorreto	**Dado que** estou na tela de autenticação **Quando** informo um e-mail inválido **Então** informo uma senha válida **Quando** clico em “Sign in” **Então** vejo erro “email or password is invalid”
		Inserir uma senha incorreta	**Dado que** estou na tela de autenticação **Quando** informo um e-mail e uma senha válida **E** clico em “Sign in”

Quadro 6 - Relação de cenários de testes identificados por *feature.*

(continuação)

			Então vejo erro "email or password is invalid"
		Autenticar usuário com sucesso	**Dado que** estou na tela de autenticação **Quando** informo um e-mail válido **Então** informo uma senha inválida **Quando** clico em "Sign in" **Então** sou redirecionado ao dashboard do sistema
Publicar artigos	**Como** usuário do sistema **Quero** acessar a área de criação de artigos **Para** poder realizar a publicação de artigos	Não preencher campo obrigatório - Título	**Dado** que estou no dashboard do sistema **E** clico em "New Article" **Quando** não preencho o Título **Então** preencho os outros campos obrigatórios **Quando** clico em "Publish Article" **Então** vejo erro "title can't be blank"
		Não preencher campo obrigatório - Descrição	**Dado** que estou no dashboard do sistema **E** clico em "New Article" **Quando** não preencho a Descrição **Então** preencho os outros campos obrigatórios **Quando** clico em "Publish Article" **Então** vejo erro "description can't be blank"
		Não preencher campo obrigatório - Corpo	**Dado** que estou no dashboard do sistema **E** clico em "New Article" **Quando** não preencho o Corpo **Então** preencho os outros campos obrigatórios **Quando** clico em "Publish Article" **Então** vejo erro "body can't be blank"
		Publicar artigo com sucesso	**Dado** que estou no dashboard do sistema **Quando** clico em "New Article" **Então** preencho todos os campos obrigatórios.

Quadro 6 - Relação de cenários de testes identificados por *feature.*

(conclusão)

			Quando clico em “Publish Article” **Então** vejo o artigo publicado

Fonte: Elaborado pela autora, 2022.

Através da identificação dos cenários de teste, observa-se que alguns cenários possuem o mesmo fluxo, porém, com a inserção de dados diferentes. Sendo assim, é possível a utilização de *Scenario Outilne* ou Esquema de Cenário, onde esses cenários possuem mais de um exemplo, agrupados em uma tabela. Esses exemplos são executados de acordo com cada linha da tabela, e a cada execução, são utilizadas as informações da linha que está sendo executada, dessa forma, consegue-se reduzir a quantidade de cenários escritos para cada *feature*, sendo realizado um esquema de cenário para os cenários inválidos e um cenário válido, ficando na seguinte estrutura.

- ***Cadastro de usuário***

A Figura 18 apresenta a estruturação de como serão os cenários de teste da funcionalidade de Cadastro de Usuário, escrito em *Gherkin*, já no *framework Cucumber*. Onde o primeiro cenário é um esquema de cenário, que testará as três variações de cenários inválidos, sendo esses o de usuário inválido, e-mail inválido e senha inválida e o último cenário percorre o caminho feliz, onde ocorre um cadastro com sucesso.

Figura 18 - Cenários de teste para *feature* de cadastro de usuário

```
#language: pt

Funcionalidade: Cadastro de Usuário
Como usuário do sistema
Quero realizar cadastro
Para ter acesso as funcionalidades do sistema


Esquema do Cenário: Cadastrar usuário preenchendo informações inválidas
    Dado que acesso a tela de cadastro do sistema
    Quando informo um <username>, um <email> e uma <password>
    E clico em 'Sign up'
    Então vejo erro <error_message>
    Exemplos:
        | username  | email     | password   |         error_message           |
        |'invalido' |'valido'   |'valido'    |'username has already been taken'|
        |'valido'   |'invalido' |'valido'    |'email has already been taken'   |
        |'valido'   |'invalido' |'invalido'  |"password can't be blank"        |

Cenário: Cadastrar usuário com Sucesso
Dado que acesso a tela de cadastro do sistema
Quando preencho todos os campos de cadastro
E clico em 'Sign up'
Então sou redirecionado ao dashboard do sistema
```

Fonte: Elaborado pela autora, 2022.

- **Autenticação de usuário**

A Figura 19 apresenta a estruturação de como serão os cenários de teste da funcionalidade de Autenticação de Usuário, escrito em *Gherkin*, já no *framework Cucumber*. Onde o primeiro cenário é um esquema de cenário, que testará as três variações de cenários inválidos, sendo esses o de e-mail inválido e senha inválida e o último cenário percorre o caminho feliz, onde ocorre a autenticação com sucesso.

Figura 19 - Cenários de teste para a *feature* de autenticação de usuário

```
#language: pt

Funcionalidade: Autenticação de usuário
Como usuário do sistema
Quero me autenticar
Para acessar meu perfil de usuário


Esquema do Cenário: Autenticar usuário preenchendo informações inválidas
    Dado que estou na tela de autenticação
    Quando informo um <email> e uma <password>
    E clico em 'Sign in'
    Então vejo erro <error_message>
    Exemplos:
        | email                 | password   |         error_message           |
        |'peter.parker@bol.com' |'123,'      |'email or password is invalid'   |
        |'peter.parker@mail.com'|'12345'     |'email or password is invalid'   |


Cenário: Autenticar usuário com Sucesso
Dado que estou na tela de autenticação
Quando preencho dados válidos de email e senha
E clico em 'Sign in'
Então sou redirecionado ao dashboard do sistema
```

Fonte: Elaborado pela autora, 2022.

- **Publicar artigos**

A Figura 20 apresenta a estruturação de como serão os cenários de teste da funcionalidade de Publicação de artigo, escrito em *Gherkin*, já no *framework Cucumber*. Onde o primeiro cenário é um esquema de cenário, que testará as três variações de cenários inválidos, sendo esses o de não preenchimento do campo título, descrição e corpo e o último cenário percorre o caminho feliz, onde ocorre a publicação de um artigo com sucesso.

Figura 20 - Cenário de teste para *feature* de publicação de artigos

```
#language: pt

Funcionalidade: Publicação de artigo
Como usuário do sistema
Quero acessar a área de criação de artigos
Para poder realizar a publicação de artigos

@login
Esquema do Cenário: Publicar artigos não preenchendo campos obrigatórios
    Dado que estou no dashboard do sistema
    Quando acesso página de  publicação de artigos
    Então preencho os campos <title>, <description> e <body>
    Quando clico em 'Publish Article'
    Então vejo erro <error_message>
    Exemplos:
        | title                            | description  | body         |      error_message          |
        |''                                |'Lorem ipsum' |'Lorem ipsum'|"title can't be blank"       |
        |'Automação de testes com Capybara' |''           |'Lorem ipsum'|"description can't be blank"|
        |'Automação de testes com Capybara' |'Lorem ipsum' |''           |"body can't be blank"       |

@login
Cenário: Publicar artigo com sucesso
Dado que estou no dashboard do sistema
Quando acesso página de  publicação de artigos
Então preencho todos os campos obrigatórios
Quando clico em 'Publish Article'
Então vejo o artigo publicado
```

Fonte: Elaborado pela autora, 2022.

4.4 ANÁLISE DOS RESULTADOS

Após a execução dos casos de testes utilizando os *frameworks Capybara* e *Cypress*, as questões de análise (Quadro 5), são sistematicamente respondidas.

QA1. Qual a complexidade de configurar um projeto de automação?

As configurações apresentadas aqui, serão baseadas apenas no sistema operacional *Windows* 10.

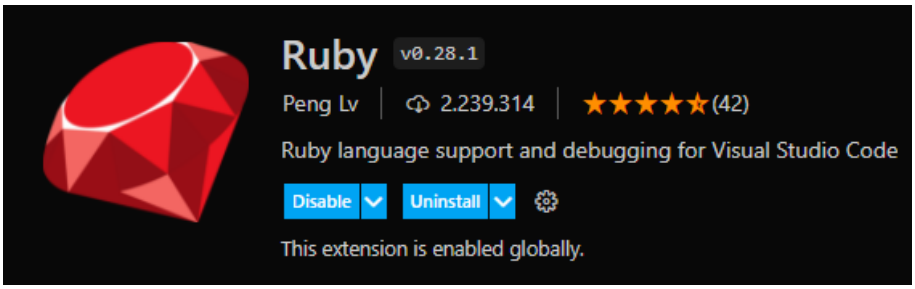
Capybara

Para configuração e utilização do *Capybara*, é necessário fazer a instalação do *Ruby+Devkit*, o *devkit* pode ser baixado através do site *Ruby Installer* (rubyinstaller.org). Também deve-se fazer o download do *chromedriver* (ChromeDriver - WebDriver for Chrome - Downloads (chromium.org)), e extrair seu conteúdo dentro da pasta "c:/windows". O *chromedriver* é o responsável por levantar o navegador ao executar a automação de testes com o *Capybara*.

No *VSCode* é necessário instalar a extensão *"Ruby",* para que ele possa reconhecer todo código gerado em *Ruby*. Também é necessário instalar a extensão *"Cucumber (Gherkin)",* para

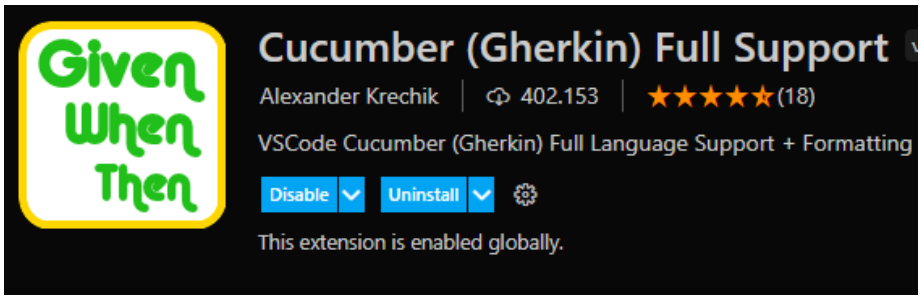
que sejam reconhecidas as *features* do *Cucumber* como linguagem *gherkin*. As Figuras 21 e 22 apresentam as extensões a serem instaladas dentro do *VSCode*.

Figura 21 - Extensão "*Ruby*" no *VSCode*



Fonte: Elaborado pela autora, 2022.

Figura 22 - Extensão "*Cucumber (Cherkin)*" no *VSCode*

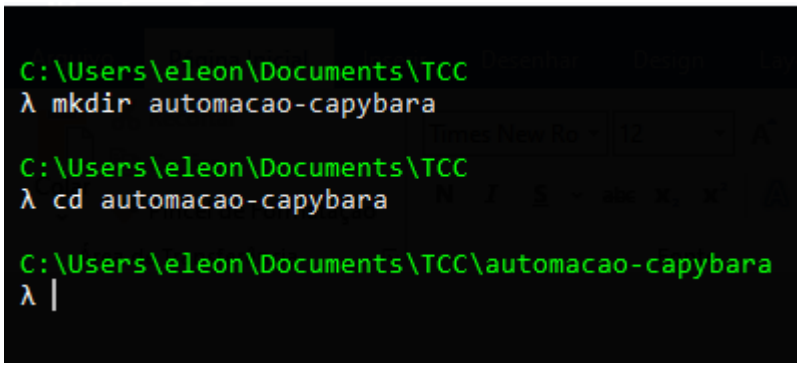


Fonte: Elaborado pela autora, 2022.

Após instalar o *Devkit* do *Ruby* e ter extraído o *chromedriver* na pasta c:/windows, o ambiente está pronto para iniciar a configuração de um novo projeto de automação com *Capybara*.

Através do Cmder é criado um novo diretório e esse é acessado, é nesse que será feito todo projeto de automação em *capybara*. A Figura 23 apresenta o comando de criação de um novo diretório bem como o acesso a esse através dos comandos *mkdir* (criação de novo diretório pelo terminal) e *cd* (comando para acessar diretórios).

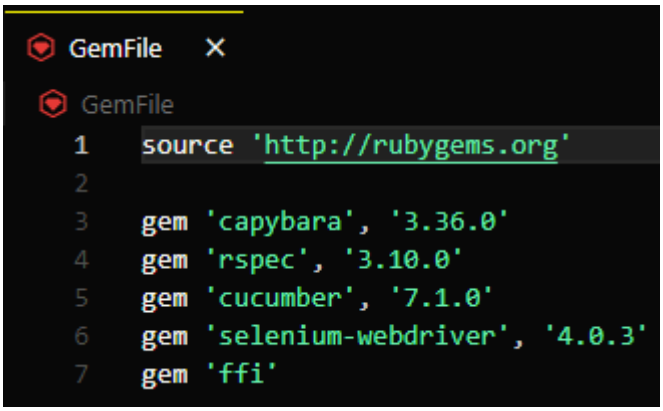
Figura 23 - Criação e acesso ao diretório do projeto de automação em *Capybara*



Fonte: Elaborado pela autora, 2022.

Após a criação do diretório, já é possível acessá-lo através do *VSCode*. No *VSCode* é necessário a criação de um arquivo *"GemFile"*. O *"GemFile"* é quem gerencia a instalação de todas as dependências de um projeto em *Ruby* através do *Bundler*. As dependências no *Ruby* são chamadas de *gems*, dessa forma, a cada novo recurso necessário, é preciso adicionar a *gem* ao *GemFile* essas *gems* e suas versões, podem ser encontradas na página rubygems.org. A Figura 24 apresenta o arquivo *GemFile* onde são informadas as *gems* necessárias para a configuração e execução do projeto de automação, essas *gems* estão definidas pelo seu nome e a versão que será instalada.

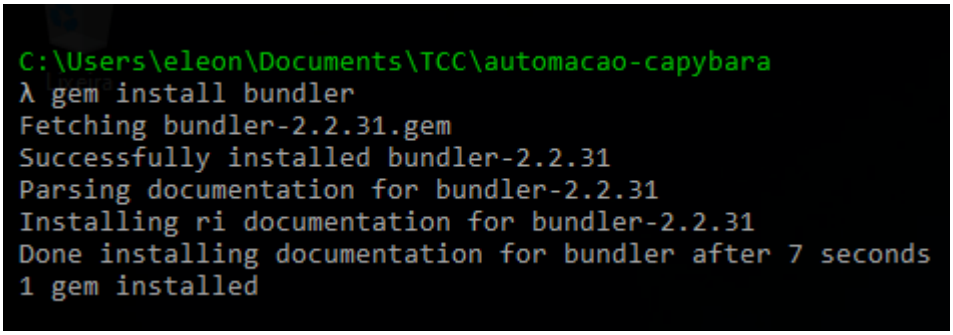
Figura 24 - *GemFile* e *gems* necessárias para configuração do projeto



Fonte: Elaborado pela autora, 2022.

Dado que as dependências do projeto são instaladas através do *Bundler*, primeiramente se faz necessária a instalação do mesmo, dentro do projeto, para que posteriormente possa se fazer a instalação das *gems*. A Figura 25 apresenta a execução do comando *"gem install Bundler"*, que consiste na instalação da *gem* do *Bundler* ao projeto de automação.

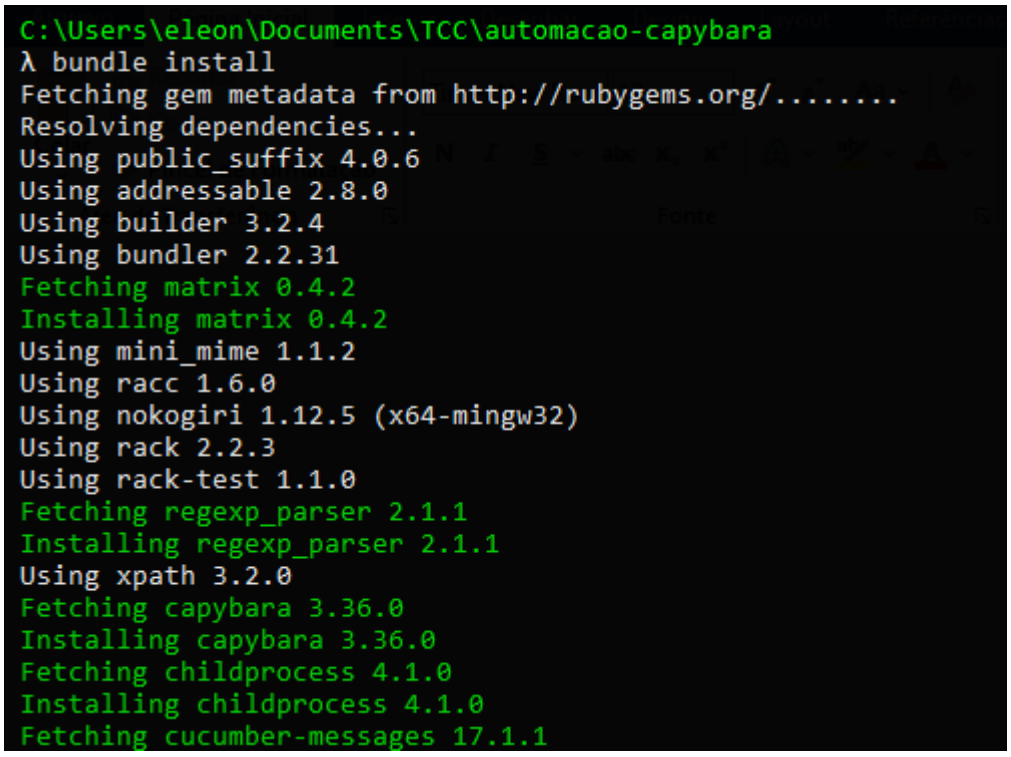
Figura 25 - Instalação do *Bundler*



Fonte: Elaborado pela autora, 2022.

A Figura 26 apresenta a execução do comando *"bundle install"*, esse é o comando responsável pela execução do *Bundler* instalado anteriormente, fazendo com que as dependências sejam instaladas no projeto.

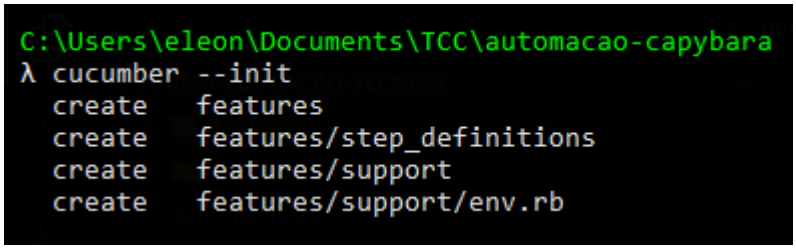
Figura 26 - Instalação das dependências do projeto



Fonte: Elaborado pela autora, 2022.

Após a instalação das dependências, é necessário rodar o comando *"cucumber --init"*, através desse, será criado a estrutura inicial padrão do projeto de automação. A Figura 27 apresenta a execução do comando *"cucumber --init"*, esse acaba por adicionar os diretórios *features*, *features/step_definitions*, *features/support* e *features/support/env.rb*, nesse último arquivo *env.rb* é que será feita a configuração do ambiente.

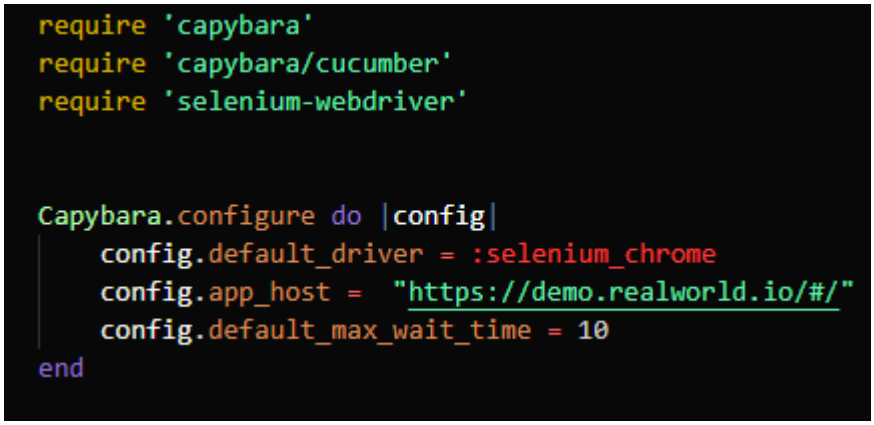
Figura 27 - Iniciando projeto



Fonte: Elaborado pela autora. 2022.

A Figura 28 apresenta a configuração inicial padrão para o arquivo *env.rb*, onde "*require"*, trata da palavra reservada para a importação das dependências que serão utilizadas. E é feita a configuração através da função "*Capybara.configure"*, onde o *"default_driver"*, define o navegador chrome, para execução padrão, utilizando do *Selenium Webdriver*, *app_host*, define a URL da aplicação que será testada e *default_max_wait_time*, define o tempo máximo de espera de carregamento da tela, em segundos.

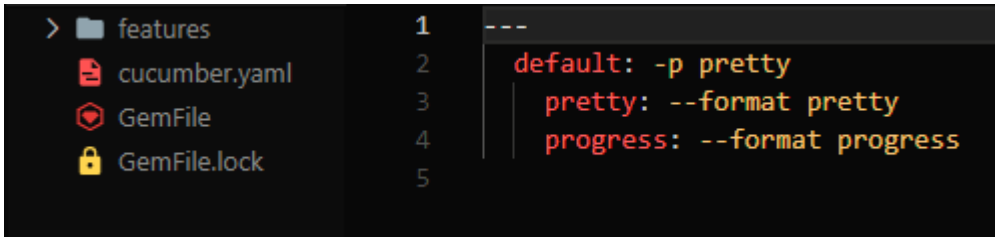
Figura 28 - Configuração inicial env.rb



Fonte: Elaborado pela autora, 2022.

Por fim, é adicionado a raiz do diretório, o arquivo *"cucumber.yaml"*, esse é o responsável por deter todas as configuração da execução do projeto de automação *Capybara* com *Cucumber*. A Figura 29 apresenta a configuração inicial do arquivo *"cucumbe.yaml"*, onde em *default* é inserido tudo aquilo que será executado por padrão no projeto, nesse contexto inicial, apenas está sendo definida um formato *pretty* para apresentação dos resultados da execução da automação. Nesse arquivo, pode-se definir qual ambiente (quando houver mais de um), navegador (se houver a necessidade de trabalhar com multi *browsers*) e configuração de *reports*, que serão executados por padrão junto à automação.

Figura 29 - Arquivo cucumber.yaml

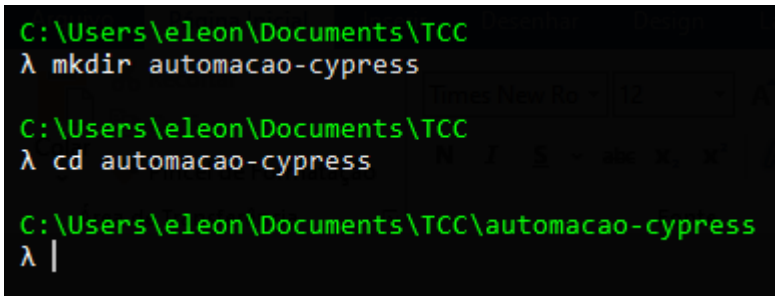


Fonte: Elaborado pela autora, 2022.

Cypress

Para configuração e utilização do *Cypress*, é necessário fazer a instalação do *Node.js*, esse pode ser baixando através do site do *Node* (Node.js (nodejs.org)). Através do cmder é criado um novo diretório para o projeto de automação em *Cypress*, após sua criação esse é acessado. A Figura 30 apresentam o comando de criação de um novo diretório bem como o acesso a esse através dos comandos *mkdir* e *cd*.

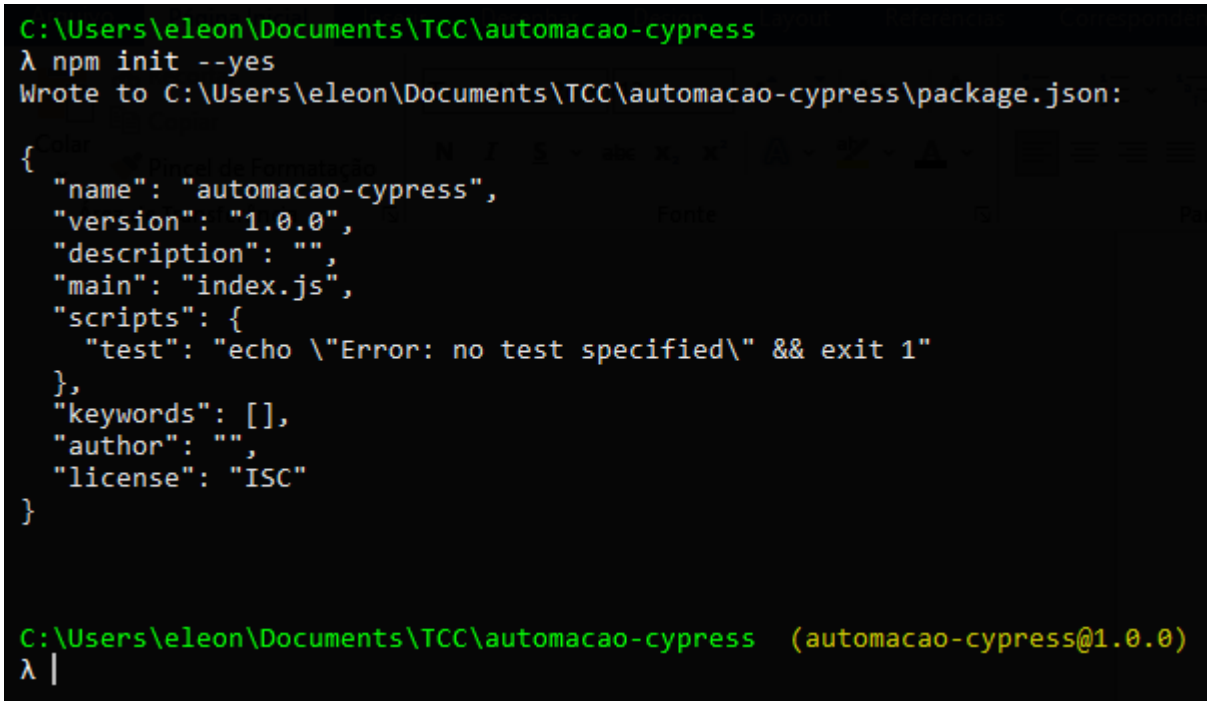
Figura 30 - Criação e acesso ao diretório do projeto de automação com *Cypress*



Fonte: Elaborado pela autora, 2022.

Para dar início a configuração do *Cypress*, primeiro é necessário iniciar um novo projeto node. A Figura 31 apresenta o comando *"npm init --yes"*, utilizado para iniciar um novo projeto em node.js, "npm" trata-se da palavra reservada para o gerenciador de pacotes do node, "*init"*, refere-se à invocação da inicialização de um novo projeto node e *"--yes"* é uma configuração passada, para que seja por padrão definido sim como reposta padrão para todas as perguntas de inicialização do projeto.

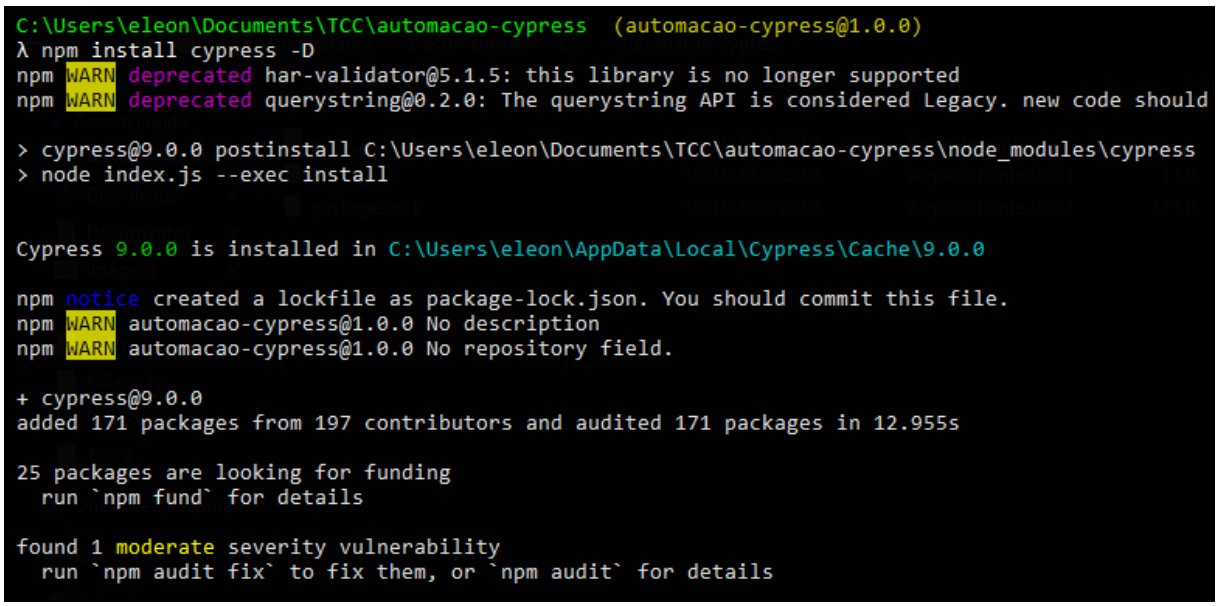
Figura 31 - Iniciando um novo projeto node



Fonte: Elaborado pela autora, 2022.

Após a inicialização do projeto em node.js, é necessário instalar o pacote do *Cypress* e o pacote do *Cucumber*, para se utilizar junto ao *Cypress.* A Figura 32 apresenta a instalação do pacote do *Cypress*, através do comando *"npm install cypress -D"*, onde "-D" refere-se a indicação de que esse pacote é uma dependência de desenvolvimento.

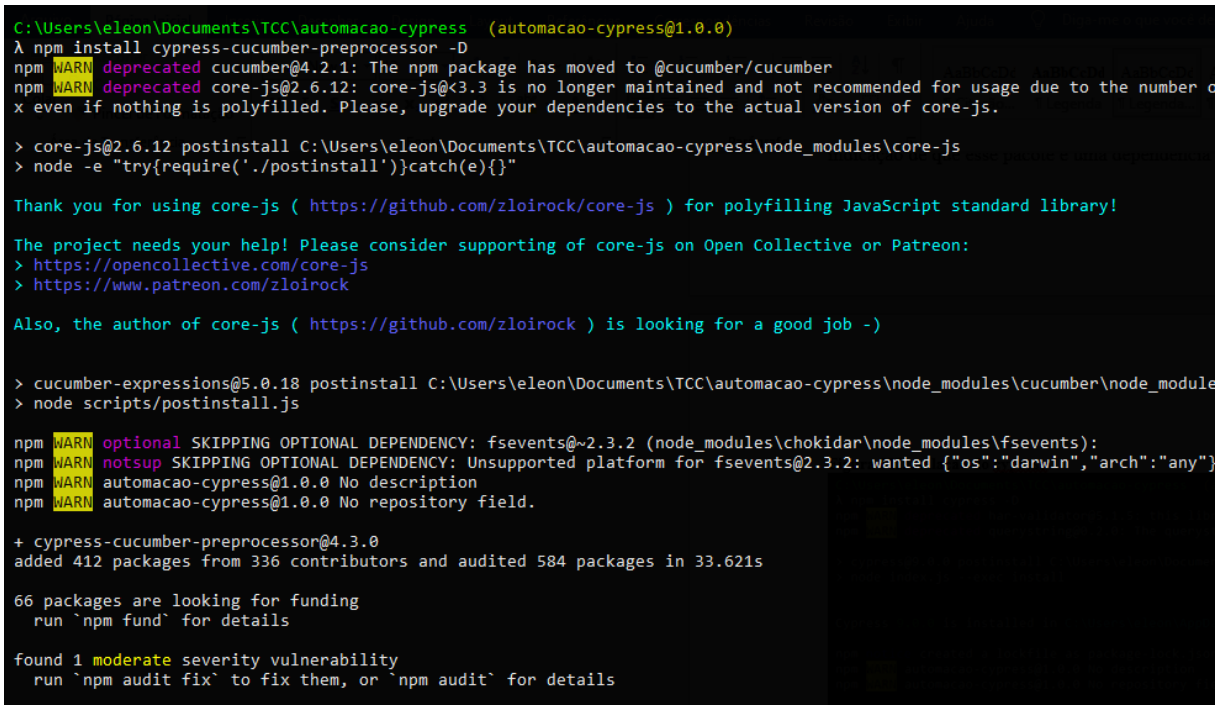
Figura 32 - Instalação do *cypress*



Fonte: Elaborado pela autora, 2022.

A Figura 33 apresenta a instalação do pacote do *Cucumber* com *Cypress*, através da execução do comando *"npm install cypress-cucumber-preprocessor -D"*.

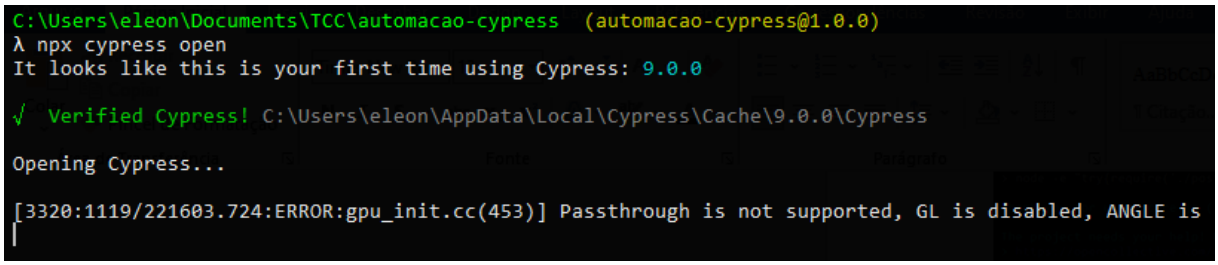
Figura 33 - Instalação do pacote do *cucumber* ao *cypress*



Fonte: Elaborado pela autora, 2022.

No cmder é necessário executar o comando de inicialização do projeto em *Cypress*. Esse comando é executado toda vez que for executar os testes, ao ser executado pela primeira vez, o *Cypress* cria o diretório inicial para a escrita dos scripts de automação e levanta o console de execução de testes. A Figura 34 apresenta a execução do comando *"npx cypress open"*, crie a estrutura padrão no projeto e inicie o console de execução de testes.

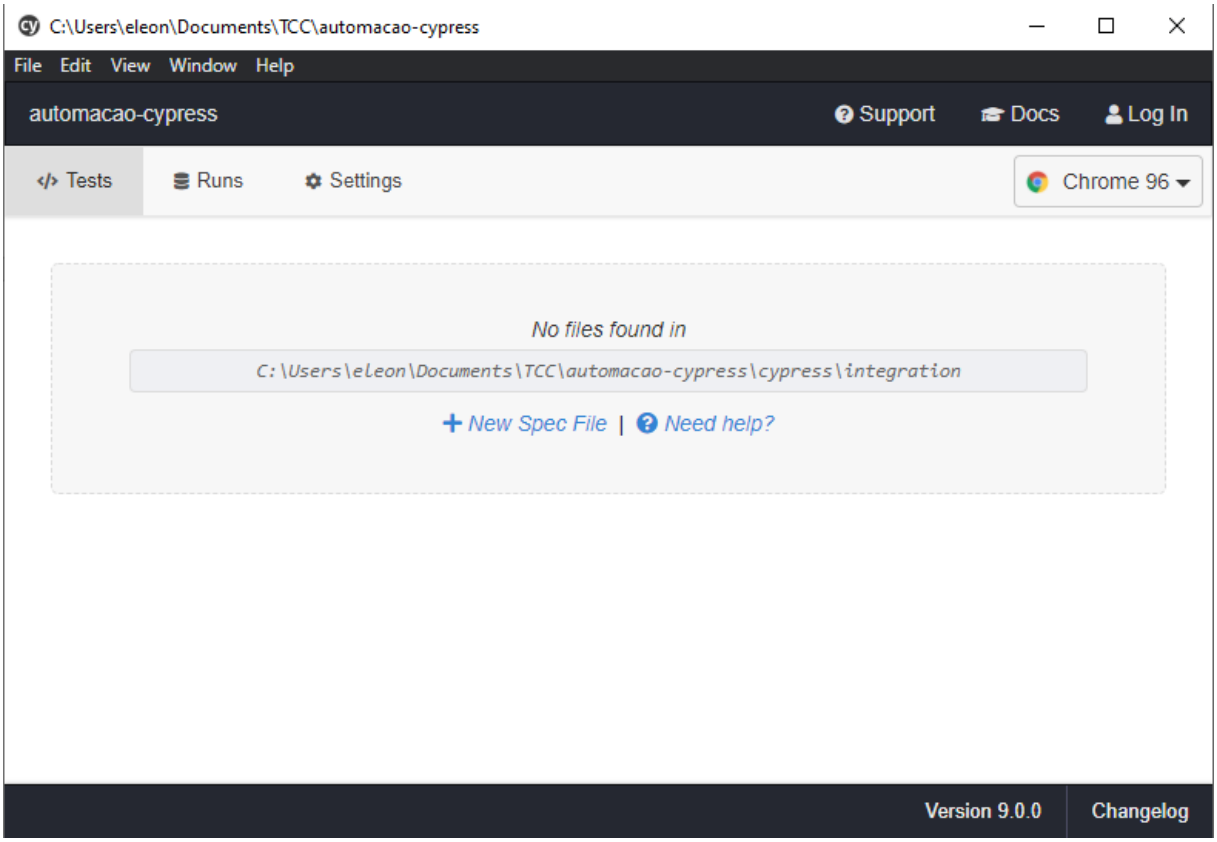
Figura 34 - Inicialização do *cypress*



Fonte: Elaborado pela autora, 2022.

A Figura 35 apresenta o console de execução de testes que o *cypress* abre ao executar o comando *"npx cypress open"*, nesse console serão listados todos os cenários de testes à medida que forem sendo criados, bem como também se faz a seleção do navegador utilizado para a execução dos testes.

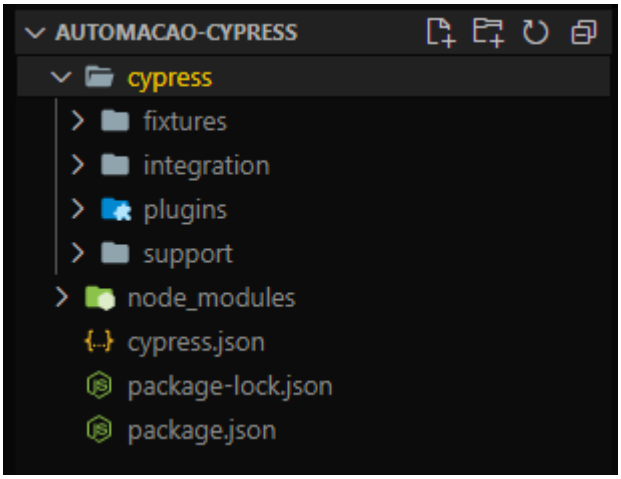
Figura 35 - Console de execução de testes



Fonte: Elaborado pela autora, 2022.

A Figura 36 apresenta a estrutura inicial, que é criada ao executar o comando *"npx cypress open"* no terminal, sendo criado o diretório *"cypress"* e todos seus subdiretórios.

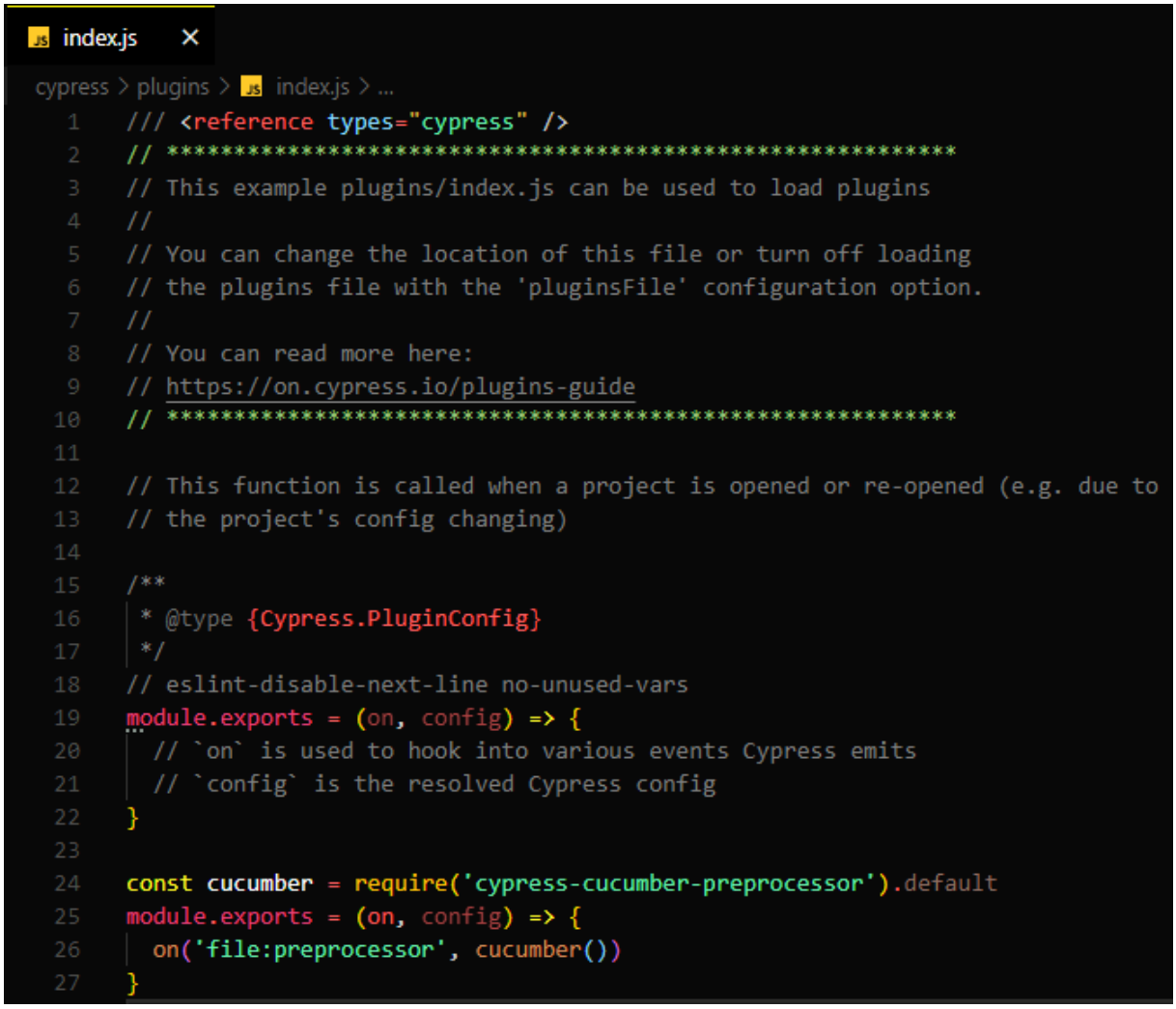
Figura 36 - Estrutura inicial projeto de automação com cypress



Fonte: Elaborado pela autora, 2022.

Agora é necessário adicionar no arquivo *cypress/plugins/index.js*, um *script* de importação do *Cucumber*, para que assim o *Cucumber* possa ser utilizado em conjunto com o *Cypress*. A Figura 37 apresenta o script que deve ser adicionado ao arquivo *"index.js"*.

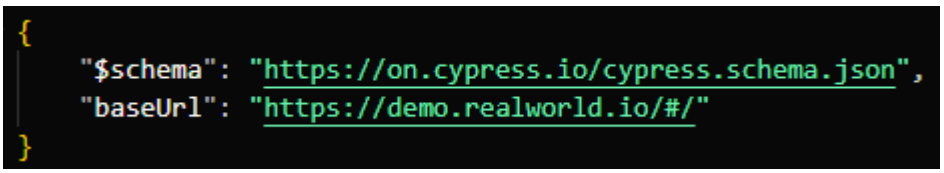
Figura 37 - Importação do plugin do cucumber para ser utilizado em conjunto com o cypress



Fonte: Elaborado pela autora, 2022.

Por fim, após a inicialização do projeto em *Cypress* temos o arquivo *"cypress.json"*, esse é o arquivo responsável pelas configurações do *cypress*. A Figura 38 apresenta a configuração inicial do *"cypress.json"*, onde *"$schema"* refere-se a um comando que traz todas as opções de comando do *Cypress* e *"baseUrl"* é a definição da URL base de acesso a aplicação onde serão automatizados os testes.

Figura 38 - Configurações iniciais cypress.json



Fonte: Elaborado pela autora, 2022.

Tendo a configuração inicial dos dois projetos de automação realizada, consegue-se observar que o *Cypress*, acaba tendo uma configuração mais simples do que o *Capybara*, onde nesse percebe-se que acaba ocorrendo uma dependência muito grande entre ferramentas. Tanto *Cypress* quanto *Capybara* necessitam da instalação dos seus executores de ambiente de suas linguagens, no caso do *Cypress* o node.js e no *Capybara* o ruby devkit. Além disso, o projeto em *Capybara* ainda necessita do *Chromedriver* e da instalação de mais um gerenciador, o B*undler*, e esse sim acaba sendo o responsável por gerenciar as dependências do projeto. Junto as dependências do *Capybara* notam-se que todas as dependências necessárias para a execução do projeto precisam ser definidas manualmente uma a uma. Também há uma dependência muito grande na iniciação e execução do projeto, através do *Cucumber*, onde tudo acaba ficando amarrado, através do *"cucumber --init"* ou até mesmo do *"cucumber.yml"*. Dessa forma, nota-se que a configuração do ambiente acaba sendo de certa forma mais burocrática de forma que no arquivo de ambiente *"env.rb"* se faz necessária a importação das dependências e também a definição do próprio navegador a ser utilizado. Enquanto no *Cypress*, basta instalar as respectivas bibliotecas através do gerenciador de pacotes do node, iniciar o executor do *Cypress*, onde esse inicia a estrutura inicial do projeto e através disso a configuração inicial do projeto está pronta. A única dependência que deve de fato ser importada, é a do *Cucumber*, através de um *script* de importação de plugin, no arquivo *"index.js"*.

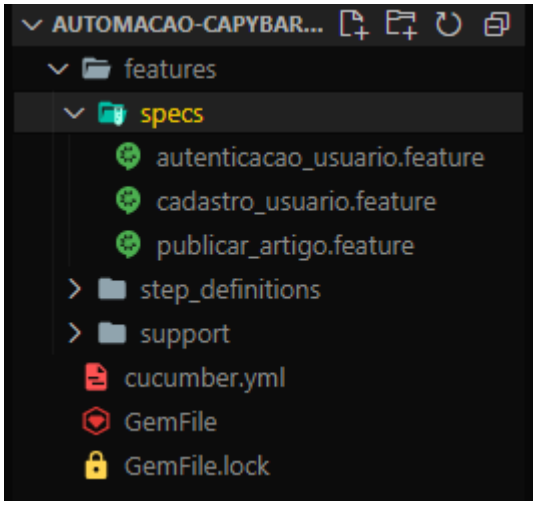
QA2. Qual a facilidade de estruturação e execução de um projeto no *framework*?

Após finalizar a configuração inicial dos *frameworks Capybara* e *Cypress* é necessário fazer algumas alterações na arquitetura dos projetos, para que estes fiquem melhor estruturados facilitando a escrita e execução dos *scripts*.

Capybara

Dentro da pasta *features*, é necessário criar uma nova pasta, chamada *"specs"*, é dentro desta pasta que ficarão os arquivos *".feature"* do *Cucumber*. A Figura 39 apresenta a pasta *specs*, dentro da pasta *features*, dentro da pasta *specs* que ficam os arquivos *.feature*, do *Cucumber*, que consistem nos cenários de testes.

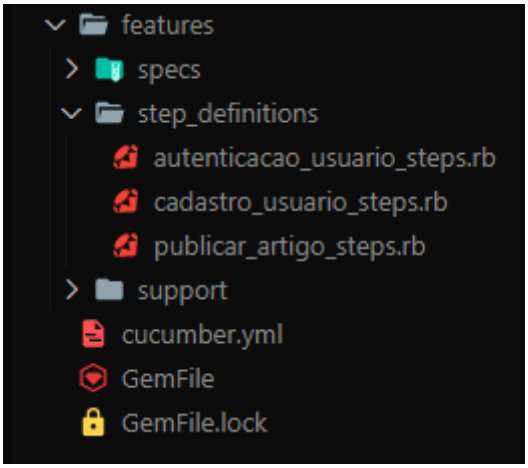
Figura 39 - Pasta specs e arquivos de cenários de testes



Fonte: Elaborado pela autora, 2022.

Dentro da pasta *step_definitions*, são criados todos os arquivos *rb* com os passos de cada caso de teste. A Figura 40 apresenta a pasta *step_definitions* com os arquivos *steps.rb* que são criados com os passos de cada um dos casos de testes, obtidos através dos arquivos *feature* do *cucumber*.

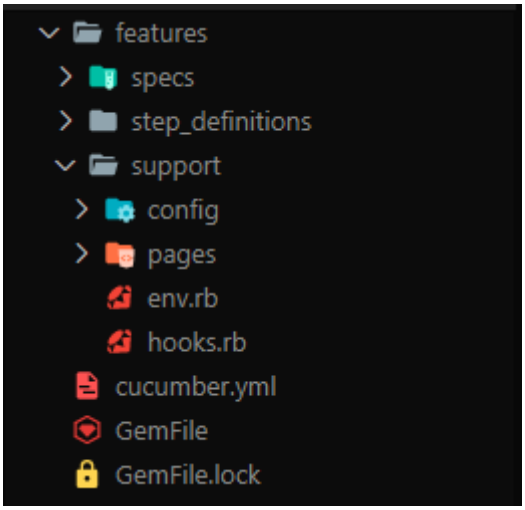
Figura 40 - Arquivos rb com os passos para cada cenário



Fonte: Elaborado pela autora, 2022.

Dentro da pasta *support*, criam-se duas pastas, *config* e *pages*. Na pasta *config* ficam os arquivos de configurações de ambiente, possibilitando a configuração de n ambientes para se rodar a automação. Na pasta *pages* é onde ficam os arquivos com as classes e métodos, sendo utilizado o padrão de projeto *PageObjects*. A Figura 41 apresenta as duas pastas *config* e *pages* dentro da pasta de *support*.

Figura 41 - Pastas config e *pages* dentro de *support*

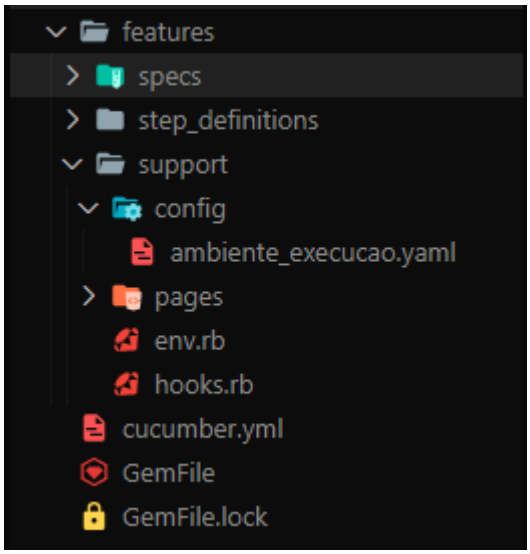


Fonte: Elaborado pela autora, 2022.

Como citado anteriormente, dentro da pasta *config*, podem-se ser criados n arquivos "*yaml"* com configurações padrões de diferentes ambientes, esses ambientes podem ser de desenvolvimento, homologação, produção, *DevOps*, etc. Após a criação da pasta *config* há a criação de um arquivo *yaml* com o nome do ambiente de execução, no caso, foi utilizado um

nome genérico, pois não trabalharemos com múltiplos ambientes. A Figura 42 apresenta o arquivo “ambiente_execucao.yaml” criando dentro da pasta *config*.

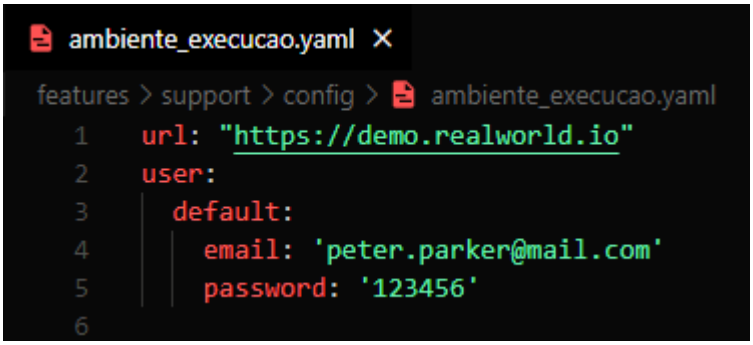
Figura 42 - Arquivo yaml com a configuração do ambiente de execução



Fonte: Elaborado pela autora, 2022.

A Figura 43 apresenta as configurações padrões do ambiente em que será executada a automação de testes, sendo estes, uma *url* padrão de execução, e um usuário padrão para quando for necessário a realização de autenticação na aplicação.

Figura 43 - Configurações de ambientes padrões que serão executadas

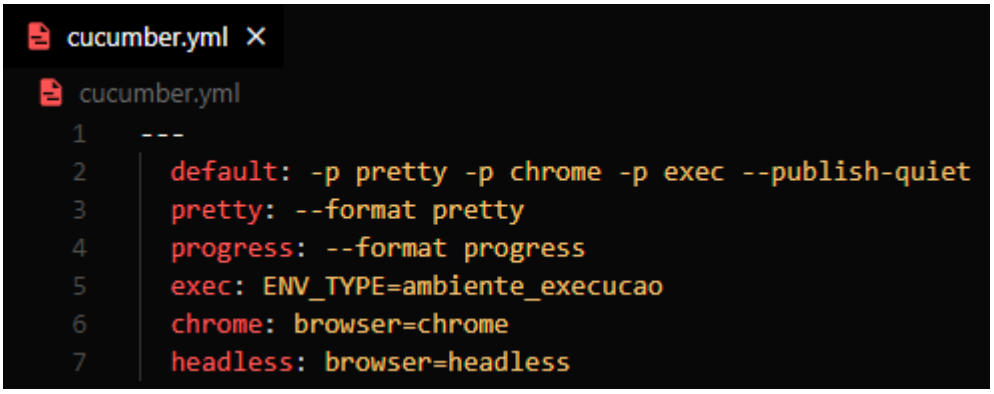


Fonte: Elaborado pela autora, 2022.

Como pode-se ter mais de um arquivo de configurações de ambiente, o arquivo *cucumber.yml*, acaba sendo o responsável por definir qual ambiente será executado, desta maneira, é ele que envia para o arquivo *env.rb* qual o *“ENV_TYPE”* que será utilizado, também é nesse arquivo que há a definição do navegador em que a automação irá ser executada. A Figura 44 apresenta a criação da definição do ambiente que será executado, através de *exec*,

que possui um *ENV_TYPE* que recebe o nome “ambiente_execucao”, também é definido o *chrome* e *headless*, que possuem a definição de um *browser* um desses *browser* recebe *chrome* e o outro *headless*, respectivamente. Desta maneira, essas configurações acabam sendo adicionadas a linha *default* e sendo iniciadas por padrão, toda vez que a automação é executada.

Figura 44 - Definição do ambiente de execução no arquivo cucumber.yml



Fonte: Elaborado pela autora, 2022.

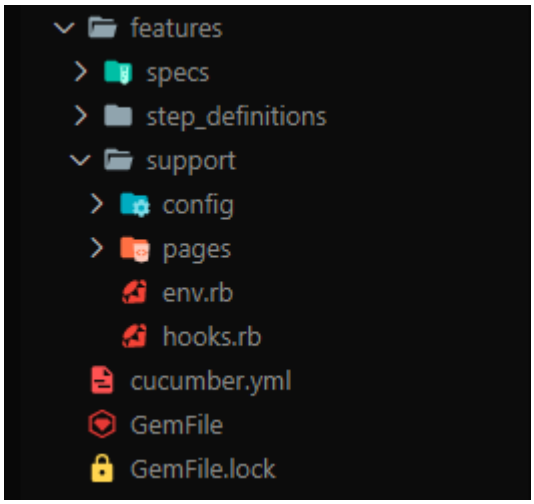
Para que essas configurações sejam utilizadas de fato, no arquivo *env.rb*, deve-se criar uma variável global, onde serão carregadas todas as informações do arquivo yaml, podendo ser utilizada sempre que necessário, também se cria um condicional onde é feita a comparação do tipo de *browser* que está sendo recebido através do c*ucumber.yml* para que assim, se faça a criação e definição do *driver* do navegador de execução. A Figura 45 mostra a variável global *CONFIG*, que recebe o caminho do arquivo “ambiente_execucao.yaml”, também apresenta o *app_host* recebendo a *url* padrão, através da variável *“CONFIG[‘url’]”*. Também mostra a criação do *driver* de execução do navegador e esse sendo carregado na variável *default_driver*.

Figura 45 - Variável de ambiente config carregando os dados do arquivo yaml

Fonte: Elaborado pela autora, 2022.

Por fim é necessário a criação do arquivo *hooks.rb*, dentro da pasta *support*¸ esse arquivo terá o papel de um gancho, fazendo a inicialização de todos os arquivos *pages* que serão utilizados na automação. A Figura 46 apresenta o arquivo *hooks.rb*, que é criado dentro da pasta *support*.

Figura 46 - Arquivo *hooks* dentro da pasta *support*



Fonte: Elaborado pela autora, 2022.

A Figura 47 apresenta a configuração de dois *Before* dentro do *hooks.rb*, o primeiro é a instanciação das classes *page* para dentro de uma variável global, para que essas possam ser

utilizadas sempre que necessário na automação. O segundo é a automação de uma autenticação na aplicação, que carrega as informações de usuário padrão da variável *CONFIG*, para os cenários de testes necessários e que possuam a *tag* definida *"@login"*.

Figura 47 – Configuração do *hooks.rb*

hooks.rb

features > support > hooks.rb

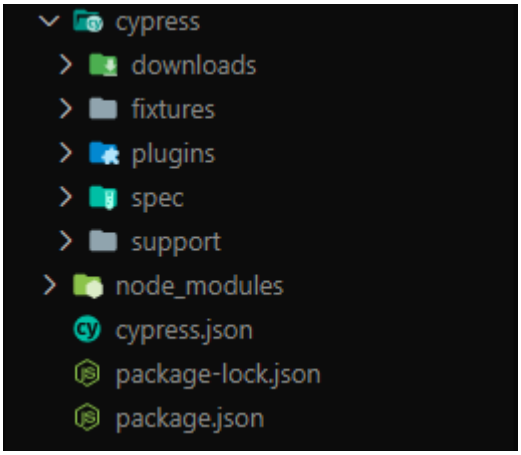
```ruby
Before do
    @cadastro = CadastroUsuarioPage.new
    @common = CommonPage.new
    @autentica = AutenticaoPage.new
    @publica = PublicArticlePage.new
end

Before("@login") do
    user = CONFIG['user']['default']
    @autentica.go_to_signin
    @autentica.fill_signin_info(user['email'], user['password'])
    @common.click_action_button('Sign in')
end
```

Fonte: Elaborado pela autora, 2022.

Cypress

Primeiramente, altera-se o nome da pasta *integration*, para *spec*, é dentro dessa pasta que serão criados os arquivos *".feature"* do *cucumber*. Para que essa alteração seja reconhecida pelo *Cypress*, é necessário adicionar uma configuração dentro do arquivo *cypress.json*, indicando que a pasta *integration* agora se chama *spec*. A Figura 48 apresenta a pasta *spec*, anteriormente chamada de *integration* dentro da pasta c*ypress*.

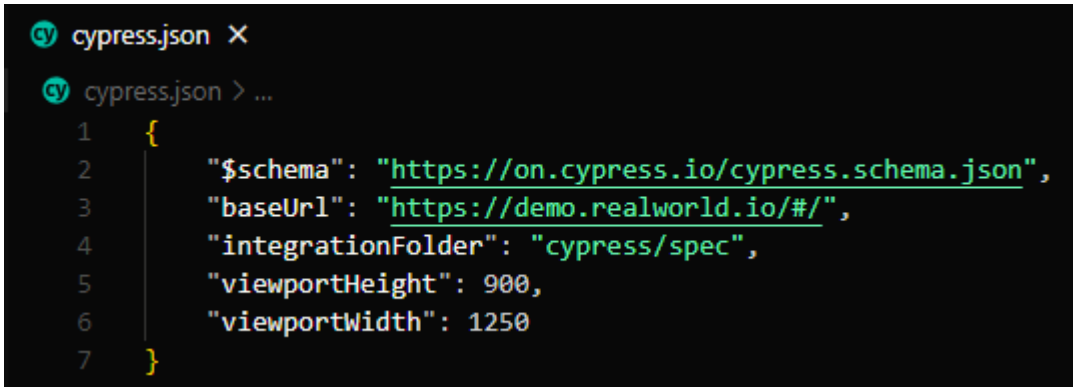
Figura 48 - Pasta spec dentro de *cypress*



Fonte: Elaborado pela autora, 2022.

A Figura 49 apresenta a configuração necessária a se fazer no arquivo c*ypress.json*, para que a pasta *spec* seja reconhecida como a pasta *integration*. Para isso se indica que *"integrationFolder"* é igual ao caminho da pasta *spec*, *"cypress/spec"*.

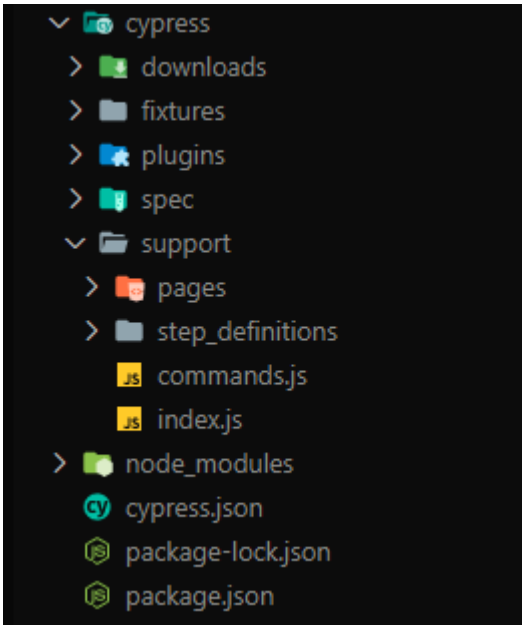
Figura 49 - Configuração da pasta *integration*



Fonte: Elaborado pela autora, 2022.

Dentro da pasta *support* é feito a criação de duas pastas, *step_definitons* e *pages*. Dentro da pasta *step_definitions*, são criados todos os arquivos *js* com os passos de cada caso de teste e dentro da pasta *pages* são criadas as pastas de *pages* que contém os arquivos com as classes, métodos e elementos, sendo utilizado o padrão de projeto *PageObjects*. A Figura 50 apresenta a estrutura de pastas *step_definitions* e *pages* dentro da pasta *support*.

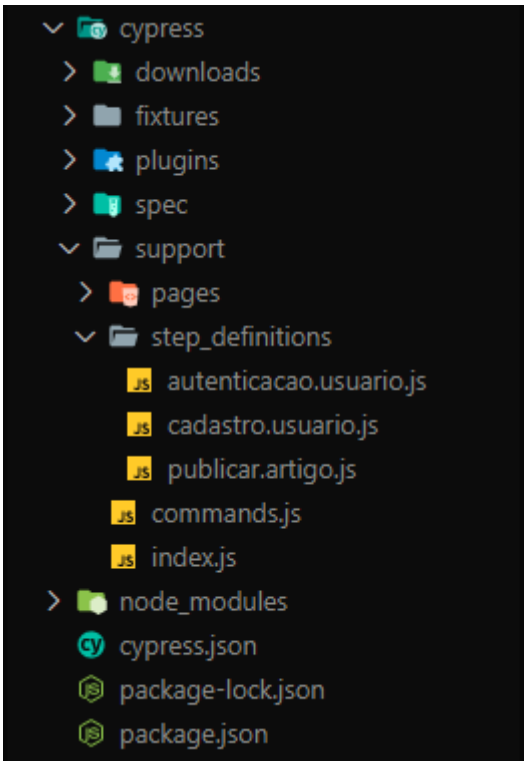
Figura 50 - Pastas step_definitons e pages



Fonte: Elaborado pela autora, 2022.

A Figura 51 apresenta os arquivos *js*, onde ficam os passos de cada um dos casos de testes da automação.

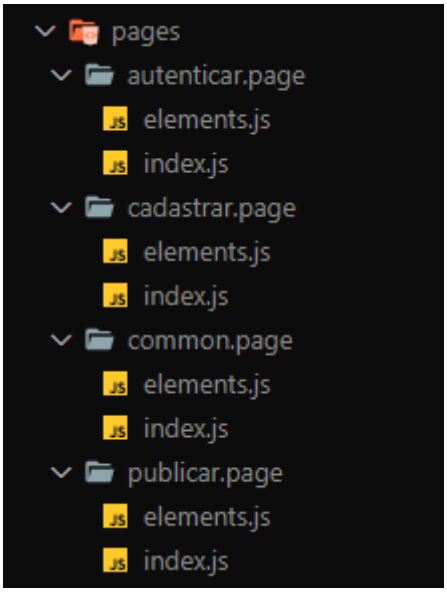
Figura 51 - Arquivos de passos para cada cenário de testes



Fonte: Elaborado pela autora, 2022.

A Figura 52 apresenta a estrutura da pasta *pages*, onde há a criação de uma pasta *.page* para cada page necessário, dessa forma se faz uma abstração entre os elementos e as classes e métodos.

Figura 52 - Pasta com os pages utilizados na automação



Fonte: Elaborado pela autora, 2022.

Após realizar a estruturação dos projetos de automação, novamente se observa que o *Cypress* apresenta uma estruturação mais simples que o *Capybara*, no entanto apesar de ter mais pontos a se ajustar dentro da estruturação do *Capybara*, percebe-se que após feito, o processo de automação torna-se muito mais fluído e produtivo. A integração do *Capybara* com o *Cucumber* é algo que facilita a utilização de múltiplos ambientes, dado que para isso basta fazer a criação de um arquivo *yaml* para cada ambiente e apenas definir sua execução no c*ucumber.yml*. Em contrapartida, no *Cypress* essa definição de ambiente é feita apenas no c*ypress.json* sendo definida apenas uma *baseUrl*. Também se percebe no *Capybara* a facilidade com que se pode definir as credenciais de quantos usuários necessários, para quando precisa realizar autenticação na aplicação, podendo assim ter essas credenciais por ambiente e deixar a automação mais dinâmica através da utilização de um *hooks* global, já no *Cypress*, não há essa possibilidade, tornando esse processo mais estático.

QA3. Quantas linhas de código são geradas por funcionalidade?

Para a realização da automação de testes, foram automatizados os mesmos cenários de testes tanto para os *frameworks Capybara* quanto para o *Cypress*, e também foi utilizado o

padrão de projeto *PageObjects*, a fim de deixar a estrutura das automações o mais semelhante entre si. Desta maneira, a contagem de linhas de código é separada em duas etapas, quantas linhas são geradas entre os *steps* de cada funcionalidade e em relação aos *pages* de cada funcionalidade, através da utilização do *PageObjects*, foi gerado um *page* de métodos em comum, utilizados pelas três funcionalidades, sendo esse definido apenas como *Common.*

Capybara

O Quadro 7 apresenta a relação entre as linhas de código geradas para cada funcionalidade automatizada no *framework Capybara*, estando separado em linhas dos arquivos de *steps* e *pages* de cada uma das funcionalidades e ao final quantas linhas foram geradas para os métodos do *page* em comum utilizado pelas funcionalidades.

Quadro 7 - Linhas de códigos geradas para o *framework Capybara*

Funcionalidade	***Steps***	***Pages***
Cadastro Usuário	23 linhas	39 linhas
Autenticação Usuário	11 linhas	18 linhas
Publicação de Artigos	19 linhas	27 linhas
Common	-	16 linhas

Fonte: Elaborado pela autora, 2022.

No quadro foi apresentado a relação de linhas por arquivos *steps* e *pages*, no entanto, se for levar em conta que o *Capybara* ainda possui um arquivo *hooks.rb*, pode-se adicionar mais 13 linhas de código geradas.

Cypress

O Quadro 8 apresenta a relação entre as linhas de código geradas para cada funcionalidade automatizada no *framework Cypress*, sendo separado entre a linhas dos arquivos de *steps* e as linhas dos arquivos de *pages* de cada uma das funcionalidades e quantas linhas foram geradas para os métodos do *page* em comum utilizado pelas funcionalidades.

Quadro 8 - Linhas de códigos geradas para o *framework Cypress*

(continua)

Funcionalidade	***Steps***	***Pages***
Cadastro Usuário	32 linhas	37 linhas

Quadro 9 - Linhas de códigos geradas para o *framework Cypress*

(conclusão)

Autenticação Usuário	17 linhas	20 linhas
Publicação de Artigos	25 linhas	50 linhas
Common	-	17 linhas

Fonte: Elaborado pela autora, 2022.

Tendo o projeto de automação implementado nos dois *frameworks*, observa-se que no geral, o *Capybara* acaba se saindo melhor em relação ao *Cypress*, tendo menos linhas geradas tanto nos *steps* quanto nos *pages*. Ao todo, o *Capybara* apresenta 53 linhas totais em relação aos *steps* e 100 linhas totais em relação aos *pages,* enquanto o *Cypress* apresenta 74 linhas totais em relação aos *steps* e 124 linhas totais em relação aos *pages*. Se for avaliar o total de linhas de código geradas, *Capybara* obtém um resultado de 153 linhas, no entanto se adicionar a essa conta a linhas geradas no arquivo *hooks* temos um total de 166 linhas, ainda assim acaba sendo uma diferença significante em relação ao *Cypress* que totalizando *steps* e *pages* temos 198 linhas de código geradas.

A integração entre *Cypress* e *Cucumber* acaba por não ser das melhores e pode se tornar um pouco trabalhosa, enquanto no *Capybara* o *Cucumber*, reconhece quando ainda não há um passo implementado dentro da pasta *step_defitions* informando isso ao usuário e entregando esse passo estruturado. No *Cypress* o usuário tem que ter o cuidado de criar passo por passo e se certificar que estes estão presentes dentro de *step_defitions*, caso contrário, a automação irá quebrar sem ao menos informar onde está ocorrendo o erro. Em relação aos *step_definitions* também é necessário indicar dentro dos arquivos de cada passo, a utilização do *Cucumber* através informando *"/* global Given, When, Then*/"*, enquanto no *Capybara*, tudo acontece automaticamente. No *Capybara* há a possibilidade de utilizar no *Cucumber* em múltiplos idiomas, no entanto, no *Cypress* é reconhecido apenas o inglês.

Em relação a utilização do padrão *PageObjects* o *Cypress* através da sintaxe do javascript, acaba deixando o usuário muito limitado em sempre precisar exportar o *page* dentro do arquivo de *pages* e importar os *pages* que serão utilizados nos *steps_definitions*, no entanto a estruturação dos *pages* acaba sendo muito produtiva, através da abstração entre os métodos e os elementos de cada tela, fomentando a reutilização e facilitando mais ainda a manutenção de código. Com o *Capybara* a utilização de pages é muito simples, porém não há essa abstração dos elementos da mesma forma que o *Cypress* apresenta, podendo tornar a manutenção de código um pouco mais trabalhosa. No entanto a forma de instanciação e utilização se torna

menos trabalhosa, já que isso é sempre feito através do arquivo *hooks*, por meio da criação de variáveis globais que podem ser chamadas e utilizadas a qualquer momento em qualquer arquivo do *step_definitions*.

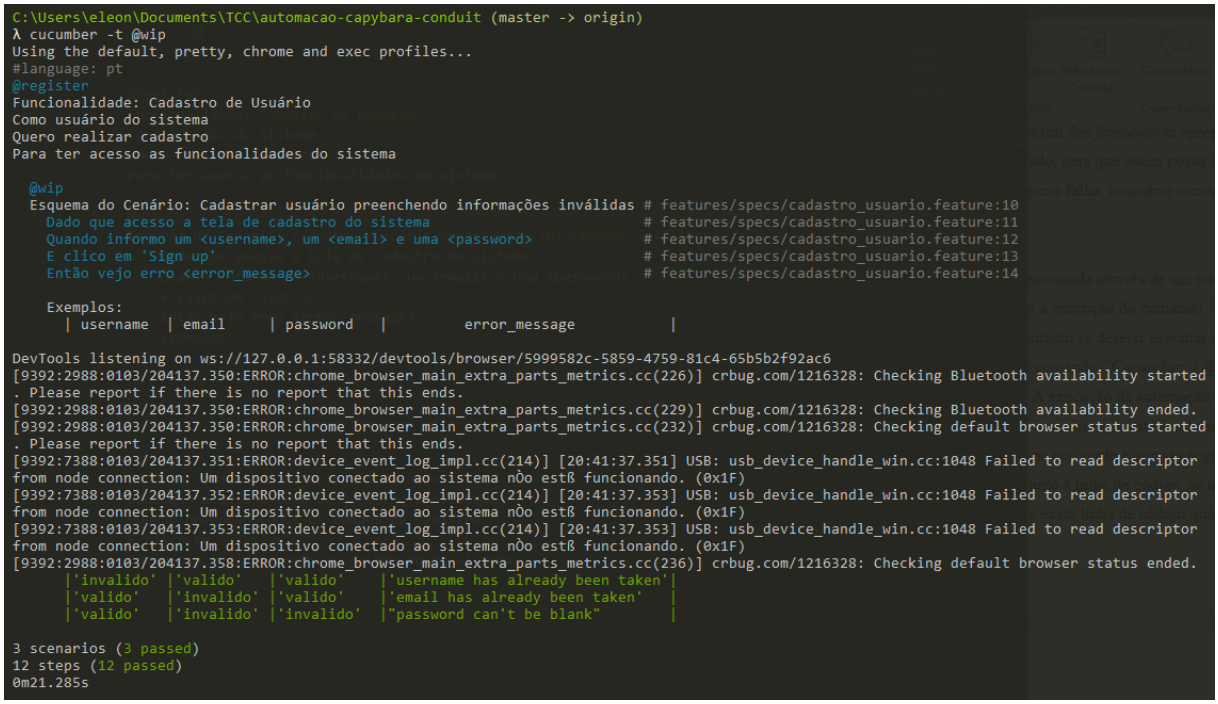
QA4. Qual a facilidade de acesso ao log e depuração dos testes?

Ao executar a automação de testes, cada um dos *frameworks* apresenta um log, a fim de mostrar ao usuário o que está sendo executado, para que assim possa se ter um acompanhamento dos passos e caso ocorra alguma falha, o usuário consiga identificar onde essa ocorreu.

Capybara

A automação no *Capybara* é sempre executada através de um terminal de comando, no caso o *Cmdr*, dessa maneira, sempre que há a execução do comando *"cucumber"* é iniciada a execução de toda a automação. No entanto, se desejar executar apenas um ou mais cenários selecionados, é necessário executar o comando *"Cucumber -t @tag"* (onde *@tag* refere-se a uma chave definida, pelo usuário). A execução da automação é feita cenário por cenário, desta forma, é informado no terminal o cenário que está sendo executado e à medida que os passos vão passando esses vão ficando na cor verde e o passo seguinte é apresentado e ao seu lado é sempre apresentado o caminho junto à linha de código, se um dado passo quebra, esse fica na cor vermelha e apresenta a exata linha de código que ele estava passando de onde ocorreu a quebra e a razão, neste ponto a automação desse cenário é interrompida, passando para o próximo cenário da fila. Ao final da execução, é apresentado quantos cenários foram executados, quantos passos e o tempo de execução. A Figura 53 apresenta o log de execução no terminal, com todos os cenários em verde indicando que a automação foi executada e finalizada com sucesso.

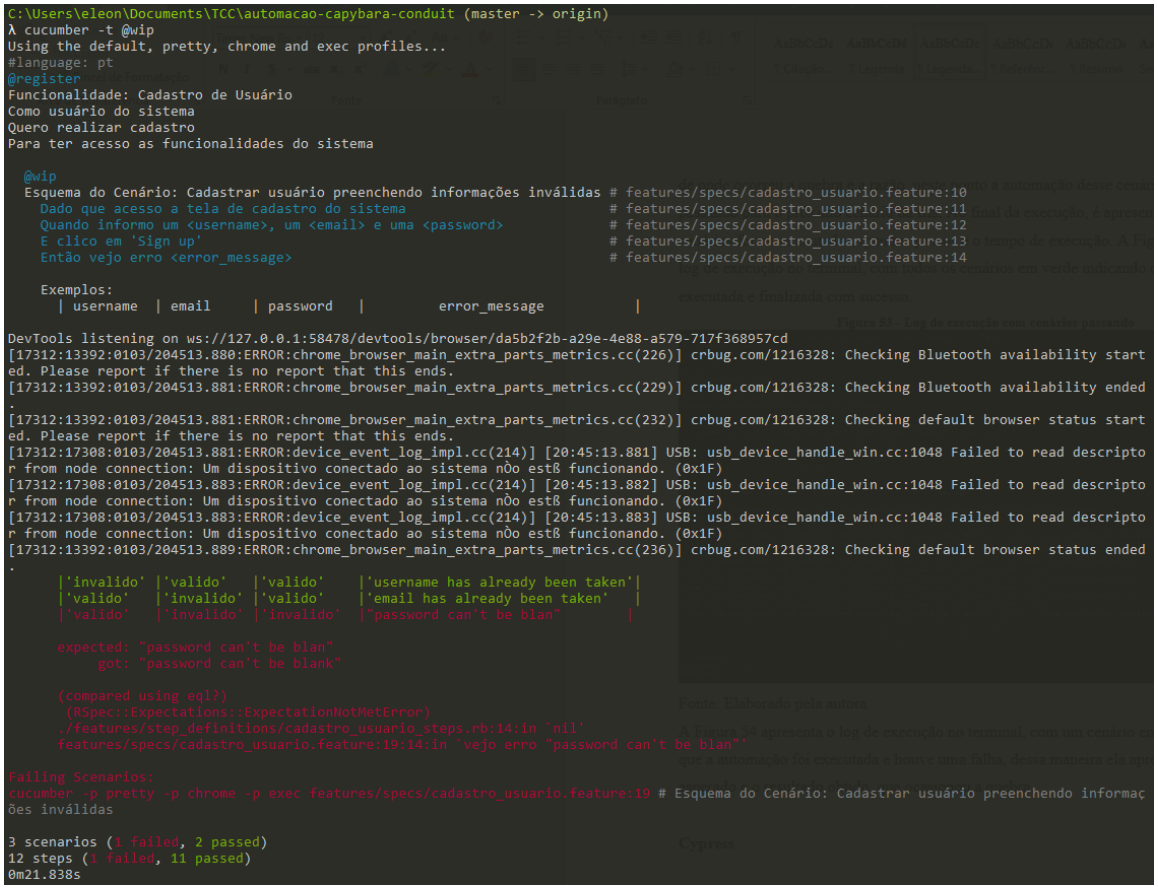
Figura 53 - Log de execução com cenários passando no *framework Capybara*



Fonte: Elaborado pela autora, 2022.

A Figura 54 apresenta o log de execução no terminal, com um cenário em vermelho indicando que a automação foi executada e houve uma falha, dessa maneira ela apresenta o resultado esperado e o resultado obtido, que ocasionou a quebra.

Figura 54 - Log de execução com falha na execução no *framework Capybara*

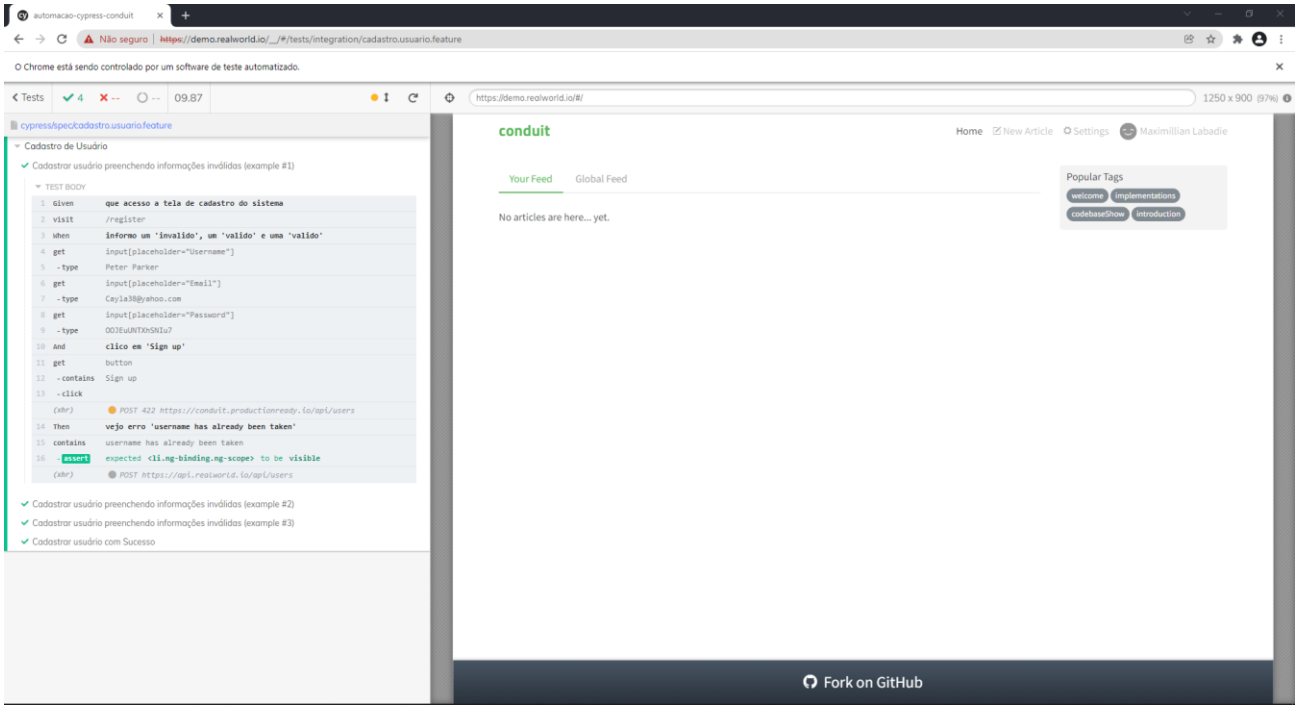


Fonte: Elaborado pela autora, 2022.

Cypress

O *Cypress* possui um terminal próprio de execução dos seus testes, desta forma basta selecionar a *feature* desejada para execução que a automação é executada inteiramente dentro do navegador. Há também a possibilidade de executar todas as *features* presentes dentro da pasta *integration* que foi renomada para *spec*. Ao iniciar a execução dos testes automatizados, é aberto o navegador selecionado (*Chrome*), e listado os cenários de testes, a cada cenário executado com sucesso, é apresentado um *check* ao lado do nome do cenário, quando a alguma falha na automação é apresentada um erro em vermelho e o caminho de onde esse erro está ocorrendo, bem como o que foi obtido e o esperado. Também é apresentado quantos cenários passaram e quantos quebraram, junto ao tempo de execução. A Figura 55 apresenta o log de execução no navegador *Chrome*, com todos os cenários em verde indicando que a automação foi executada e finalizada com sucesso.

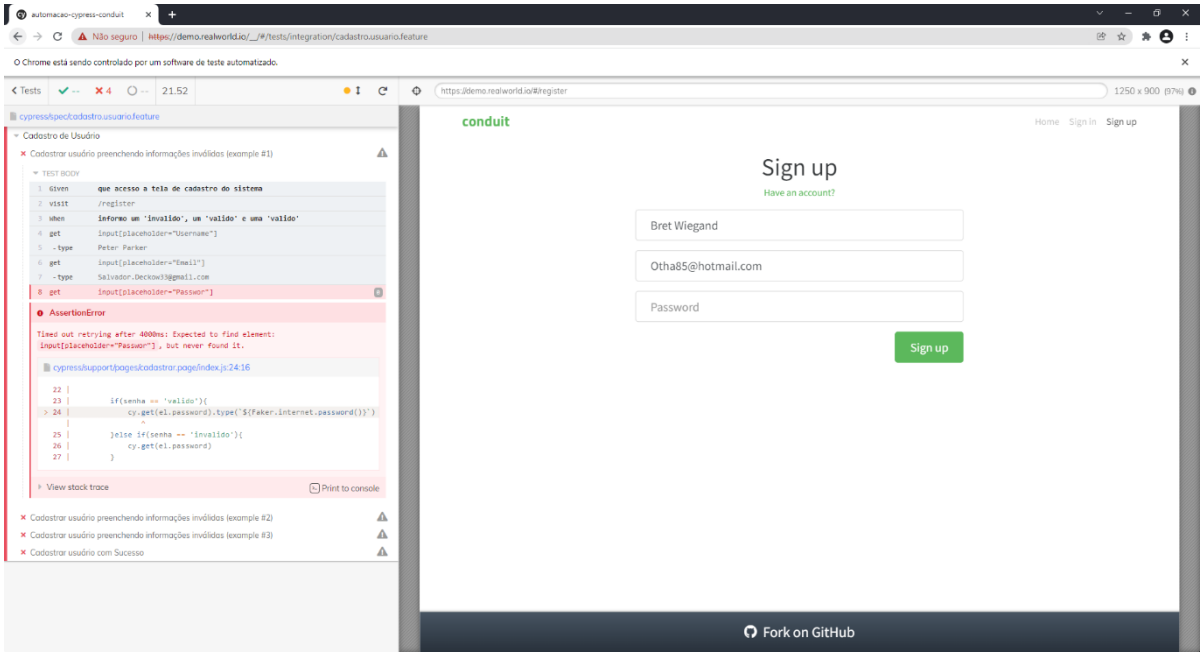
Figura 55 - Log de execução com cenários passando no *framework Cypress*



Fonte: Elaborado pela autora, 2022.

A Figura 56 apresenta o log de execução no terminal, com os cenários apresentando um "x" em vermelho indicando que a automação foi executada e houve falha nos cenários, dessa maneira ela apresenta o resultado esperado e o resultado obtido, que ocasionou a quebra.

Figura 56 - Log de execução com cenários com falha no *framework Cypress*



Fonte: Elaborado pela autora, 2022.

Uma das maiores diferenças entre o *Capybara* e o *Cypress*, acaba sendo o fato de o *Cypress* não ter sido construído para ser executado em cima do motor do *Selenium WebDriver,* e por ter um painel próprio e único de execução, enquanto a execução da automação com *Capybara* ocorre diretamente no terminal de comando. A maior diferença identificada em relação a isso, é a facilidade visual de identificar os erros que ocorrem na automação, dado que ao longo da execução da automação no *Cypress*, é apresentado a todo momento na tela os passos executados, com o resultado esperado e obtido, tanto quando há sucesso, tanto quando há falha, e ao término da execução, essas imagens não são perdidas, fazendo que quando o usuário passar o mouse sobre um passo, é apresentado exatamente o que estava ocorrendo na execução, àquele momento. No *Capybara*, apesar de ser bem detalhado o que está sendo executado no terminal, mesmo a automação sendo executada no navegador, ao término de sua execução, este é fechado, e acaba se perdendo visualmente o que estava acontecendo em tela no momento que cada passo é executado, dessa forma isso pode acabar por atrasar a compreensão do que ocorreu durante a execução de determinado passo, se houver falha.

QA5. Quanto tempo leva a execução de um cenário de testes de uma *feature*?

Através da execução de cada um dos cenários de testes, obtém-se o tempo estimado que se levou para a finalização da mesma.

Capybara

O Quadro 9 apresenta a relação de tempo de execução em segundos, para cada um dos cenários de testes de cada *feature* de teste no *framework Capybara*. Esse tempo é obtido através da execução da automação, no terminal de comando e ao final é obtido o tempo total da execução.

Quadro 10 - Tempo de execução por cenários *framework Capybara*

(continua)

Feature	**Cenário**	**Tempo**
Cadastro de usuário	Cadastrar usuário preenchendo informações inválidas	**28.447s**
	Cadastrar usuário com Sucesso	**12.972s**

Quadro 11 - Tempo de execução por cenários *framework Capybara*

(conclusão)

Autenticação de usuário	Autenticar usuário preenchendo informações inválidas	**10.024s**
	Autenticar usuário com Sucesso	**6.281s**
Publicar Artigo	Publicar artigos não preenchendo campos obrigatórios	**24.481s**
	Publicar artigo com sucesso	**9.920s**

Fonte: Elaborado pela autora, 2022.

Cypress

O *Cypress* não tem a opção de execução de cenários de testes de forma isolada, impossibilitando de se obter o tempo médio que cada cenário de teste leva para ser executado.

QA6. Quanto tempo leva a execução de todos os cenários de testes de uma *feature*?

Através da execução de todos os cenários de testes de uma única *feature*, obtém-se o tempo estimado que se levou para a finalização da mesma.

Capybara

O Quadro 10 apresenta a relação de tempo de execução em segundos, para cada uma das *features* de teste no *framework Capybara*. Esse tempo é obtido através da execução da automação, no terminal de comando e ao final é obtido o tempo total da execução.

Quadro 12 -Tempo de execução de cada *feature framework Capybara*

Feature	**Tempo**
Cadastro de usuário	**24.116s**
Autenticação de usuário	**13.587s**
Publicar Artigo	**31.442s**

Fonte: Elaborado pela autora, 2022.

Cypress

O Quadro 11 apresenta a relação de tempo de execução em segundos, para cada uma das *features* de teste no *framework Cypress*. Esse tempo é obtido através da execução da automação, através do painel de execução do *Cypress*, ao término dessa.

Quadro 13 - Tempo de execução de cada *feature framework Cypress*

Feature	**Tempo**
Cadastro de usuário	**15.40s**
Autenticação de usuário	**07.63s**
Publicar Artigo	**18.28s**

Fonte: Elaborado pela autora, 2022.

QA7. Quanto tempo leva a execução de todas as features de testes?

Através da execução de todas as *features* de teste, obtém-se o tempo estimado que levou para a finalização da mesma.

Capybara

Após realizar a execução de todas as *features* de automação, para o *Capybara*, obteve-se um tempo médio de execução de **64.432**s.

Cypress

Após realizar a execução de todas as *features* de automação, para o *Cypress*, obteve-se um tempo médio de execução de **34.63s.**

(QA5, QA6 e QA7) Em um contexto geral, nota-se que o *Cypress* se sai melhor em tempo de execução de seus testes automatizados, em relação ao *Capybara*. O único contexto que de fato não pode ser comparado com clareza é o caso de execução de cenários de testes isoladamente, onde no *Capybara* isso é permitido e no *Cypress* não. Como já apresentado anteriormente, devido ao *Cypress* não ter sua execução baseada no *driver* do *Selenium Webdriver*, é demandado menos recursos e componentes externos, para que seja iniciada a execução de seus testes. No momento que é selecionado a funcionalidade que será executada ou até mesmo optado por executar todas as funcionalidades, é aberto um único navegador e todos os testes são executados ali. Já no *Capybara*, por estar configurado para ser executado através do *Selenium Webdriver* e também necessitar de um *driver* externo do navegador, leva-se muito tempo apenas para inicializar o *Chrome* e após a execução de cada cenário de testes o navegador é derrubado, tendo que ser inicializado de novo, o que acaba tomando um tempo de execução maior.

4.5 DISCUSSÃO DOS RESULTADOS

Após a implementação dos dois projetos de automação de testes e dos resultados obtidos, observa-se que o *framework Cypress* é superior em relação a facilidade de configuração e estruturação de um projeto de automação, esse também se sobressai em relação a depuração e *log* e o tempo de execução dos testes. Em contrapartida o *framework Capybara* acaba tendo seus pontos fortes em relação a integração e escrita de scripts de automação utilizando o *Cucumber* e também em relação a quantidade de linhas de código geradas, no entanto, mesmo tendo uma estruturação mais robusta, após finalizada, acaba por dar mais liberdade e dinamicidade ao processo de criação dos *scripts* de automação. O Quadro 12 apresenta a relação entre as vantagens e desvantagens entre cada um dos *frameworks* comparados.

Quadro 14 - Relação entre vantagem e desvantagens de cada um dos *frameworks*

Framework	**Facilidade de Configuração**	**Estruturação**	**Integração com *Cucumber***	**Linhas de Código Geradas**	**Depuração e *Log* de Execução**	**Tempo de Execução**
Capybara	⊖	⊖	✓	✓	⊖	⊖
Cypress	✓	✓	⊖	⊖	✓	✓

Fonte: Elaborado pela autora, 2022.

5 CONSIDERAÇÕES FINAIS

Este trabalho analisou os *frameworks Capybara* e *Cypress* com o propósito de identificar vantagens e desvantagens com respeito ao tempo de execução, custo de execução e complexidade de configuração do ponto de vista de testador de software no contexto da aplicação *Conduit*.

Através da configuração de dois projetos de automação, para cada um dos diferentes *frameworks* de automação de testes *web*, foi possível observar que cada um dos *frameworks*, *Capybara* e *Cypress*, tem seus pontos fortes em alguns aspectos específicos.

Em questão de configuração, o *Cypress* se sai melhor, por ter uma configuração mais simples e rápida e que não demanda tanto conhecimento técnico, enquanto o *Capybara* acaba por ter muitas pontas a serem amarradas, para que tudo seja executado corretamente.

O mesmo ocorre em relação a estruturação dos projetos, onde no *Cypress* é tudo feito de forma muito simples, no entanto, apesar de o *Capybara* ter uma estruturação mais complexa, após finalizada se tem muito mais liberdade para definição de ambientes de execução e gerenciamento de usuários, tendo uma dinamicidade maior.

Quando se leva em conta a quantidade de linhas geradas, o *Capybara* se sai melhor, onde acaba por gerar menos linhas de código e em relação ao *Cypress*.

No entanto, quando falamos em *log* e tempo de execução, *Cypress* acaba se tornando um bom *framework,* pois seu tempo de execução de testes é menor do que o *Capybara* e o *log* é apresentado de forma visual, facilitando e agilizando a identificação de falhas que possam ocorrer.

A escolha de cada um dos *frameworks* depende do contexto do que se quer automatizar e como se quer automatizar, o *Capybara* é um *framework*, que se integra muito bem ao *Cucumber*, no entanto, isso não ocorre da mesma maneira com o *Cypress* que acaba tornando esse processo de integração e utilização bem trabalhoso. O *Cypress* é um *framework* de fácil configuração e utilização, tendo mais performance de execução e um *log* de fácil compreensão, em contrapartida, o *Capybara* é um *framework* mais maduro, que acaba por ter uma sintaxe mais produtiva e fluída.

REFERÊNCIAS

BARTIÉ, Alexandre. **Garantia da Qualidade de Software**. Rio de Janeiro: Elsevier, 2002.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. Goal, **Question Metric Paradigm**. In: MARCINIAK, J. J. (Ed.). Encyclopedia of Software Engineering. John Wiley & Sons, p. 528-532, 1994.

BERNARDO, Paulo Cheque; KON, Fabio. A importância dos testes automatizados. **Engenharia de Software Magazine**, v. 1, n. 3, p. 54-57, 2008.

CAETANO, Cristiano. Melhores Práticas na Automação de Testes. **Engenharia de software magazine**, 5ª edição, p42-47, 2008.

CRESPO, Adalberto N; SILVA, Odair J da; BORGES, Carlos Alberto; SALVIANO, Clênio F; ARGOLLO JUNIOR, Miguel de T; JINO, Mario. **Uma Metodologia para Teste de Software no Contexto da Melhoria de Processo**. Campinas, São Paulo, 2004.

CYPRESS.IO. **Cypress**. 2021. Disponível em <JavaScript End to End Testing Framework | cypress.io> Acesso em: 03 de agosto de 2021.

DELAMARO, Marcio; JINO, Mario; MALDONADO, José. **Introdução a Teste De Software.** Rio de Janeiro: Elsevier, 2013.

FEWSTER, Mark; GRAHAM, Dorothy. **Software Test Automation**. Edinburgh Gate, 1999.

GRAPE, Victor. **Comparing Costs of Browser Automation Test Tools with Manual Testing.** Thesis (Master Degree Computer and Information Science) Linköpings universitet, 2016.

HIRAMA, Kechi. **Engenharia de Software**. Rio de Janeiro: Elsevier, 2012.

KASTEGÅRD, Sandra. **Automated testing of a web-based user interface.** Final thesis (Master Degree Computer and Information Science) Linköpings universitet Institutionen för datavetenskap, 2015.

MOBARAYA, Fatini; ALI, Shahid. **Technical Analysis of Selenium and Cypress as functional automation framework for modern web application testing.** ICCSEA, WiMoA, SCAI, SPPR, InWeS, NECO - 2019 pp. 27-46, 2019.

NETO, Arilo. Introdução a Teste de software. **Revista Engenharia de Software Magazine**, Devmedia Editora, Rio de Janeiro: 1ª Edição, Ano I, pp. 54-59, publicado em mar. 2008, disponível em: <https://edisciplinas.usp.br/pluginfile.php/3503764/mod_resource/content/3/Introducao_a_Teste_de_Software.pdf> Acesso em 16 de julho de 2021,

OKEZIE, F.; Odun-Ayo, I; Bogle, S. A Critical Analysis of Software Testing Tools**. Journal of Physics**: Conference Series. 1378. p. 1-11, 2019.

PAPITO, Fernando. **Porque migrei meus testes automatizados com Java e CSharp para Ruby**. Medium, 2017. Disponível em: <Por que migrei meus testes automatizados com Java e CSharp para Ruby? | by Papito | qaninja | Medium> Acesso em 11 de Jul de 2021.

_____. **Selenium, Watir ou Capybara**. Medium, 2020. Disponível em: <Selenium, Watir ou Capybara. Qual o framework de testes que devo… | by Papito | qaninja | Medium> Acesso em 11 de Jul de 2021.

PAULA FILHO, Wilson P. **Engenharia de software:** produtos. Rio de Janeiro: LTC, 2019.

PRESSMAN, Roger S.; MAXIM, Bruce R. Engenharia de software: uma abordagem profissional. 8 ed. Porto Alegre: AMGH, 2016.

SELENIDE.ORG. **Selenide: concise UI tests in Java**. 2021. Disponível em <Selenide: concise UI tests in Java > Acesso em 11 de Jul de 2021.

SELENIUM.DEV. **WebDriver: Documentation for Selenium**. 2013-2021. Disponível em <Getting started :: Documentation for Selenium> Acesso em 11 de Jul de 2021.

SEMPRE IT**. Automação de testes: As ferramentas mais utilizadas.** 2021. Disponível em <Automação de Testes: as ferramentas mais utilizadas | Sempre IT> Acesso em 12 de Ago de 2021.

SHETH, Tarik; SINGH, K Santosh. **Software Test Automation- Approach on evaluating test automation tools.** International Journal of Scientific and Research Publications, Volume 5, 2015.

SOMMERVILLE, Ian. **Engenharia de Software**. São Paulo: Person, 2011.

TEAMCAPYBARA. **Capybara**. 2021. Disponível em <Capybara (teamcapybara.github.io)> Acesso em 11 de Jul de 2021.

THINKSTER. **RealWorld**. 2021. Disponível em < GitHub - gothinkster/realworld: "The mother of all demo apps" — Exemplary fullstack Medium.com clone powered by React, Angular, Node, Django, and many more > Acesso em 30 de Out de 2021.

WYNNE, Matt; HELLESØY, Aslak. **The Cucumber Book: Behaviour-driven Development for Testers and Developers**. Pragmatic Bookshelf, 2012.

APÊNDICE

APÊNDICE A – CADASTRO DE USUÁRIOS

Feature cadastro de usuários – *Capybara*

```
#language: pt

@register
Funcionalidade: Cadastro de Usuário
Como usuário do sistema
Quero realizar cadastro
Para ter acesso as funcionalidades do sistema


Esquema do Cenário: Cadastrar usuário preenchendo informações inválidas
    Dado que acesso a tela de cadastro do sistema
    Quando informo um <username>, um <email> e uma <password>
    E clico em 'Sign up'
    Então vejo erro <error_message>
    Exemplos:
        | username  | email     |
password   |         error_message           |
        |'invalido' |'valido'   |'valido'    |'username has already been
taken'|
        |'valido'   |'invalido' |'valido'    |'email has already been
taken'   |
        |'valido'   |'invalido' |'invalido'  |"password can't be
blank"        |


Cenário: Cadastrar usuário com Sucesso
Dado que acesso a tela de cadastro do sistema
Quando preencho todos os campos de cadastro
E clico em 'Sign up'
Então sou redirecionado ao dashboard do sistema
```

Steps cadastro de usuários – *Capybara*

```
Dado("que acesso a tela de cadastro do sistema") do
    @cadastro.go_to_signup
end

Quando("informo um {string}, um {string} e uma {string}") do |username, email,
senha|
    @cadastro.invalid_signup(username, email, senha)
end

Quando("clico em {string}") do |button|
    @common.click_action_button(button)
end

Então("vejo erro {string}") do |error_message|
    expect(@common.alert_message).to eql error_message
end

Quando('preencho todos os campos de cadastro') do
    @cadastro.valid_signup
end

Então('sou redirecionado ao dashboard do sistema') do
    expect(@common.successed_signup).to eql true
end
```

Page cadastro de usuários – *Capybara*

```
require 'faker'

class CadastroUsuarioPage
    include Capybara::DSL

    def go_to_signup
        visit '/'
        find('a[href="#/register"]').click
    end

    def invalid_signup(username, email, senha)
        case username
        when 'valido'
            find('input[placeholder="Username"]').set Faker::Name.name
        when 'invalido'
            find('input[placeholder="Username"]').set 'Peter Parker'
        end

        case email
        when 'valido'
            find('input[placeholder="Email"]').set Faker::Internet.free_email
        when 'invalido'
            find('input[placeholder="Email"]').set 'peter.parker@mail.com'
        end

        case senha
        when 'valido'
            find('input[placeholder="Password"]').set Faker::Internet.password
        when 'invalido'
            find('input[placeholder="Password"]').set ''
        end
    end

    def valid_signup
        find('input[placeholder="Username"]').set Faker::Name.name
        find('input[placeholder="Email"]').set Faker::Internet.free_email
        find('input[placeholder="Password"]').set Faker::Internet.password
    end
end
```

APÊNDICE B – AUTENTICAÇÃO DE USUÁRIOS

Feature autenticação de usuários – *Capybara*

```
#language: pt

Funcionalidade: Autenticação de usuário
Como usuário do sistema
Quero me autenticar
Para acessar meu perfil de usuário


Esquema do Cenário: Autenticar usuário preenchendo informações inválidas
    Dado que estou na tela de autenticação
    Quando informo um <email> e uma <password>
    E clico em 'Sign in'
    Então vejo erro <error_message>
    Exemplos:
        | email                 |
password   |         error_message           |
        |'peter.parker@bol.com' |'123,'      |'email or password is
invalid'   |
        |'peter.parker@mail.com'|'12345'     |'email or password is
invalid'   |


Cenário: Autenticar usuário com Sucesso
Dado que estou na tela de autenticação
Quando preencho dados válidos de email e senha
E clico em 'Sign in'
Então sou redirecionado ao dashboard do sistema
```

Steps autenticação de usuários – *Capybara*

```
Dado('que estou na tela de autenticação') do
    @autentica.go_to_signin
end

Quando('informo um {string} e uma {string}') do |email, senha|
    @autentica.fill_signin_info(email,senha)
end

Quando('preencho dados válidos de email e senha') do
    @autentica.valid_autentication
end
```

Page autenticação de usuários – *Capybara*

```
class AutenticaoPage
    include Capybara::DSL

    def go_to_signin
        visit '/'
        find('a[href="#/login"]').click
    end

    def fill_signin_info(email,pass)
        find('input[placeholder="Email"]').set email
        find('input[placeholder="Password"]').set pass
    end

    def valid_autentication
        find('input[placeholder="Email"]').set 'peter.parker@mail.com'
        find('input[placeholder="Password"]').set '123456'
    end
end
```

APÊNDICE C – PUBLICAÇÃO DE ARTIGOS

Feature publicação de artigo – *Capybara*

```
#language: pt

Funcionalidade: Publicação de artigo
Como usuário do sistema
Quero acessar a área de criação de artigos
Para poder realizar a publicação de artigos

@login
Esquema do Cenário: Publicar artigos não preenchendo campos obrigatórios
    Dado que estou no dashboard do sistema
    Quando acesso página de  publicação de artigos
    Então preencho os campos <title>, <description> e <body>
    Quando clico em 'Publish Article'
    Então vejo erro <error_message>
    Exemplos:
        | title                              | description  |
body        |      error_message         |
        |''                                  |'Lorem ipsum' |'Lorem
ipsum'|"title can't be blank"      |
        |'Automação de testes com Capybara' |''            |'Lorem
ipsum'|"description can't be blank"|
        |'Automação de testes com Capybara' |'Lorem ipsum'
|''           |"body can't be blank"       |

@login
Cenário: Publicar artigo com sucesso
Dado que estou no dashboard do sistema
Quando acesso página de  publicação de artigos
Então preencho todos os campos obrigatórios
Quando clico em 'Publish Article'
Então vejo o artigo publicado
```

Steps publicação de artigo – *Capybara*

```
Dado('que estou no dashboard do sistema') do
    expect(@common.successed_signup).to eql true
end

Quando('acesso página de  publicação de artigos') do
    @publica.go_to_new_article
end

Então('preencho os campos {string}, {string} e {string}') do |title, desc,
body|
    @publica.invalid_article_publication(title, desc, body)
end

Então('preencho todos os campos obrigatórios') do
    @publica.valid_article_publication
end

Então('vejo o artigo publicado') do
    expect(@publica.article_pulished).to eql true
end
```

Page publicação de artigo – Capybara

```
require "date"

class PublicArticlePage
    include Capybara::DSL


    def go_to_new_article
        find('a[href="#/editor/"]').click
    end

    def invalid_article_publication(title, desc, body)
        find("input[ng-model='$ctrl.article.title']").set title
        find("input[ng-model='$ctrl.article.description']").set desc
        find("textarea[ng-model='$ctrl.article.body']").set body
    end

    def valid_article_publication
        @articleName = "Automação de testes com Capybara - #{DateTime.now}"
        find("input[ng-model='$ctrl.article.title']").set @articleName
        find("input[ng-model='$ctrl.article.description']").set 'Lorem ipsum'
        find("textarea[ng-model='$ctrl.article.body']").set 'Lorem ipsum'
    end

    def article_pulished
        page.has_text?(@articleName)
    end
end
```

APÊNDICE D – PAGE COMMON PAGES – CAPYBARA

```ruby
class CommonPage
    include Capybara::DSL

    def click_action_button(button)
        click_button button
        sleep 3
    end

    def alert_message
        find('list-errors li').text
    end

    def successed_signup
        page.has_css?('a[href="#/editor/"]')
    end
end
```

APÊNDICE E – ARQUIVO *HOOKS.RB – CAPYBARA*

```
Before do
    @cadastro = CadastroUsuarioPage.new
    @common = CommonPage.new
    @autentica = AutenticaoPage.new
    @publica = PublicArticlePage.new
end

Before("@login") do
    user = CONFIG['user']['default']
    @autentica.go_to_signin
    @autentica.fill_signin_info(user['email'], user['password'])
    @common.click_action_button('Sign in')
end
```

APÊNDICE F – CADASTRO DE USUÁRIOS

Feature cadastro de usuários – Cypress

```
Feature: Cadastro de Usuário
    As usuário do sistema
    Want realizar cadastro
    To ter acesso as funcionalidades do sistema

Scenario Outline: Cadastrar usuário preenchendo informações inválidas
    Given que acesso a tela de cadastro do sistema
    When informo um <username>, um <email> e uma <password>
    And clico em 'Sign up'
    Then vejo erro <error_message>
    Examples:
        | username  | email     |
password   |         error_message           |
        |'invalido' |'valido'   |'valido'    |'username has already been
taken'|
        |'valido'   |'invalido' |'valido'    |'email has already been
taken'   |
        |'valido'   |'invalido' |'invalido'  |"password can't be
blank"        |

Scenario: Cadastrar usuário com Sucesso
Given que acesso a tela de cadastro do sistema
When preencho todos os campos de cadastro
And clico em 'Sign up'
Then sou redirecionado ao dashboard do sistema
```

Steps cadastro de usuários – Cypress

```
/// <reference types="cypress" />

import { Given, When, Then } from 'cypress-cucumber-preprocessor/steps';
import register from '../pages/cadastrar.page'
import common from '../pages/common.page'


/* global Given, When, Then*/

Given('que acesso a tela de cadastro do sistema', () => {
    register.go_to_signup()
})

When('informo um {string}, um {string} e uma {string}', (username, email,
senha) => {
    register.invalid_signup(username, email, senha)
})

And('clico em {string}', (button) => {
    common.click_action_button(button)
})

When('vejo erro {string}', (message) => {
    common.alert_message(message)
})

Then('preencho todos os campos de cadastro', () => {
    register.valid_signup()
})

When('sou redirecionado ao dashboard do sistema', () => {
    common.successed_signup()
})
```

Page cadastro de usuários – Cypress

```
const el = require('./elements').ELEMENTS
const Faker = require('faker')

class CadastroPage {

    go_to_signup(){
        cy.visit('/register')
    }

    invalid_signup(username, email, senha){
        if(username == 'valido'){
            cy.get(el.username).type(`${Faker.name.firstName()}
${Faker.name.lastName()}`)
        }else if(username == 'invalido'){
            cy.get(el.username).type('Peter Parker')
        }

        if(email == 'valido'){
            cy.get(el.email).type(`${Faker.internet.email()}`)
        }else if(email == 'invalido'){
            cy.get(el.email).type('peter.parker@mail.com')
        }

        if(senha == 'valido'){
            cy.get(el.password).type(`${Faker.internet.password()}`)
        }else if(senha == 'invalido'){
            cy.get(el.password)
        }
    }

    valid_signup(){
        cy.get(el.username).type(`${Faker.name.firstName()}
${Faker.name.lastName()}`)
        cy.get(el.email).type(`${Faker.internet.email()}`)
        cy.get(el.password).type(`${Faker.internet.password()}`)
    }
}

export default new CadastroPage()
```

Elements cadastro de usuários – Cypress

```
export const ELEMENTS = {
    register_button: 'a[href=#/register]',
    username: 'input[placeholder="Username"]',
    email: 'input[placeholder="Email"]',
    password: 'input[placeholder="Password"]'
}
```

APÊNDICE G – AUTENTICAÇÃO DE USUÁRIOS

Feature autenticação de usuários – Cypress

```
Feature: Autenticação de usuário
    As usuário do sistema
    Want me autenticar
    To acessar meu perfil de usuário


Scenario Outline: Autenticar usuário preenchendo informações inválidas
    Given que estou na tela de autenticação
    When informo um <email> e uma <password>
    And clico em 'Sign in'
    Then vejo erro <error_message>
    Examples:
        | email                 |
password   |         error_message           |
        |'peter.parker@bol.com' |'123,'      |'email or password is
invalid'   |
        |'peter.parker@mail.com'|'12345'     |'email or password is
invalid'   |


Scenario: Autenticar usuário com Sucesso
Given que estou na tela de autenticação
When preencho dados válidos de email e senha
And clico em 'Sign in'
Then sou redirecionado ao dashboard do sistema
```

Steps autenticação de usuários – Cypress

```
/// <reference types="cypress" />

import auth from '../pages/autenticar.page'

/* global Given, When, Then*/

When('que estou na tela de autenticação', () => {
    auth.go_to_signin()
})

When('informo um {string} e uma {string}', (email, password) => {
    auth.fill_signin_info(email, password)
})

When('preencho dados válidos de email e senha', () => {
    auth.valid_autentication()
})
```

Page autenticação de usuários – Cypress

```
const el = require('./elements').ELEMENTS

class AutenticarPage {

    go_to_signin() {
        cy.visit('/login')
    }

    fill_signin_info(email,pass) {
        cy.get(el.email).type(email)
        cy.get(el.password).type(pass)
    }

    valid_autentication() {
        cy.get(el.email).type('peter.parker@mail.com')
        cy.get(el.password).type('123456')
    }
}

export default new AutenticarPage()
```

Elements autenticação de usuários – Cypress

```
export const ELEMENTS = {
    email: 'input[placeholder="Email"]',
    password: 'input[placeholder="Password"]'
}
```

APÊNDICE H – PUBLICAÇÃO DE ARTIGO

Feature publicação de artigo – Cypress

```
Feature: Publicação de artigo
    As usuário do sistema
    Want acessar a área de criação de artigos
    To poder realizar a publicação de artigos


Scenario Outline: Publicar artigos não preenchendo campos obrigatórios
    Given que estou no dashboard do sistema
    When acesso página de  publicação de artigos
    Then preencho os campos <title>, <description> e <body>
    When clico em 'Publish Article'
    Then vejo erro <error_message>
    Examples:
        | title                              | description  |
body        |      error_message         |
        |''                                  |'Lorem ipsum' |'Lorem
ipsum'|"title can't be blank"      |
        |'Automação de testes com Capybara' |''            |'Lorem
ipsum'|"description can't be blank"|
        |'Automação de testes com Capybara' |'Lorem ipsum'
|''           |"body can't be blank"       |


Scenario: Publicar artigo com sucesso
Given que estou no dashboard do sistema
When acesso página de  publicação de artigos
Then preencho todos os campos obrigatórios
When clico em 'Publish Article'
Then vejo o artigo publicado
```

Step publicação de artigo – Cypress

```
/// <reference types="cypress" />

import publica from '../pages/publicar.page'

/* global Given, When, Then*/

When('que estou no dashboard do sistema', () => {
    publica.go_to_dashboard()
})

Then('acesso página de  publicação de artigos', () => {
    publica.go_to_new_article()
})

When('preencho os campos {string}, {string} e {string}', (title, description,
body) => {
    publica.invalid_article_publication(title, description, body)
})

Then('preencho todos os campos obrigatórios', () => {
    publica.valid_article_publication()
})

Then('vejo o artigo publicado', () => {
    publica.article_pulished()
})
```

Page publicação de artigo – Cypress

```
const el = require('./elements').ELEMENTS
const articleName = 'Cypress automation' + new Date().getTime()

import auth from '../autenticar.page'

class PublicarPage {

    go_to_dashboard() {
        cy.visit('/login')
        auth.valid_autentication()
        cy.get('button').contains('Sign in').click()
    }

    go_to_new_article() {
        cy.get(el.linkNovoArtigo).click()
    }

    invalid_article_publication(title, desc, body){
        if(title == '') {
            cy.get(el.inputTitle)
        }else{
            cy.get(el.inputTitle).type(title)
        }

        if(desc == '') {
            cy.get(el.inputDescription)
        }else{
            cy.get(el.inputDescription).type(desc)
        }

        if(body == '') {
            cy.get(el.inputBody)
        }else {
            cy.get(el.inputBody).type(body)
        }
    }

    valid_article_publication() {
        cy.get(el.inputTitle).type(articleName)
        cy.get(el.inputDescription).type('Lorem ipsum')
        cy.get(el.inputBody).type('Lorem ipsum')
    }

    article_pulished() {
        cy.contains(articleName).should('be.visible')
        cy.get('h1').should('have.text', articleName)
    }
```

```
}

export default new PublicarPage()
```

Elements publicação de artigo – Cypress

```
export const ELEMENTS = {
    linkNovoArtigo: 'a[href*=editor]',
    inputTitle: '[ng-model$=title',
    inputDescription: '[ng-model$=description]',
    inputBody: '[ng-model$=body]',
    inputTags: '[ng-model$=tagField]'
}
```

APÊNDICE I – *COMMONS*

Common page – Cypress

```
const el = require('./elements').ELEMENTS

class CommonPage {
    click_action_button(button) {
        cy.get('button').contains(button).click()
    }

    alert_message(message) {
        cy.contains(message).should('be.visible')
    }

    successed_signup(message) {
        cy.contains('No articles are here... yet').should('be.visible')
    }
}

export default new CommonPage()
```

Common elements – Cypress

```
export const ELEMENTS = {
    error_message: 'list-errors li',
    signed_up: 'a[href="#/editor/"]'
}
```