UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

THIAGO SANTOS DE MOURA

AUTOMAÇÃO DE TESTES EM APLICAÇÕES WEB UTILIZANDO UMA ABORDAGEM AD-HOC

CAMPINA GRANDE - PB

2021

THIAGO SANTOS DE MOURA

AUTOMAÇÃO DE TESTES EM APLICAÇÕES WEB UTILIZANDO UMA ABORDAGEM AD-HOC

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Cláudio de Souza Baptista

CAMPINA GRANDE - PB

2021

M929a Moura, Thiago Santos de.
Automação de testes em aplicações web utilizando uma abordagem AD-HOC. / Thiago Santos de Moura. – 2021.

13 f.

Orientador: Prof. Dr. Cláudio de Souza Baptista.

Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Desenvolvimento de software. 2. Aplicações web - testes. 3. Framework de testes automatizados. 4. Testes end-to-end. 5. Testes ad-hoc. 6. Testes de softwares. 7. Automação de testes em aplicações. I. Baptista, Cláudio de Souza. II. Título.

CDU:004.415.53(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

THIAGO SANTOS DE MOURA

AUTOMAÇÃO DE TESTES EM APLICAÇÕES WEB UTILIZANDO UMA ABORDAGEM AD-HOC

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Cláudio de Souza Baptista

Orientador – UASC/CEEI/UFCG

Professora Dra. Eliane Cristina de Araújo

Examinador – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 25 de maio de 2021.

CAMPINA GRANDE - PB

ABSTRACT

In web applications, often no time and resources are invested in performing automatic tests, and, commonly, only manual tests are performed by the developers themselves. Such problems can introduce application failures, which can take a long time to detect, resulting in rework, and harming the user experience when the failures reach the production environment. To contribute to a faster detection of these failures, a tool was developed that can generate test code that simulates a user going through the usage flows, simply and objectively, filling out and submitting data. Execution is done automatically by an automated testing framework. The tool was evaluated through the development of a web application for registration, in which the tests generated had their interface coverage checked, and their usefulness in indicating failures was evaluated, as well as their ability to be used as regression tests.

Automação de Testes em Aplicações Web Utilizando uma Abordagem Ad-Hoc

Thiago Santos de Moura
thiago.moura@ccc.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba

Cláudio de Souza Baptista
baptista@computacao.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba

Hugo Feitosa de Figueirêdo
hugo.figueiredo@ifpb.edu.br
Instituto Federal da Paraíba
Campina Grande, Paraíba

RESUMO

Em aplicações web, muitas vezes, não são investidos tempo e recursos na realização de testes automáticos, além de ser comum que apenas testes manuais sejam realizados pelos próprios desenvolvedores. Tal problemática pode introduzir falhas nas aplicações, que podem demorar muito tempo para serem detectadas, resultando em retrabalho, além de prejudicar a experiência do usuário quando as falhas chegam ao ambiente de produção. Para contribuir na detecção mais ágil dessas falhas, foi desenvolvida uma ferramenta capaz de gerar o código de testes que simula um usuário percorrendo os fluxos de utilização, de forma simplória e objetiva, efetuando o preenchimento e submissão de dados. A execução é feita de forma automática por um *framework* de testes automatizados. A avaliação da ferramenta foi feita através do desenvolvimento de uma aplicação web para cadastros, na qual os testes gerados tiveram sua cobertura de interfaces averiguada, e foi avaliado sua utilidade em indicar as falhas, além da capacidade de utilização como testes de regressão.

Palavras-Chave

Testes end-to-end, testes ad-hoc, cypress, ant design

Links Úteis:

https://github.com/mourats/cytestion

https://github.com/mourats/expense-organizer

https://gist.github.com/mourats/cytestion.spec.js

1. INTRODUÇÃO

No contexto de desenvolvimento de *software*, metodologias de desenvolvimento foram criadas visando sempre a qualidade do produto final. Essa qualidade está intimamente ligada à validação e verificação, em que um dos principais instrumentos é o uso de testes. Esses denominados testes de *software* [1] são diversos, cada um com seus benefícios e formas de concepção e execução. Entretanto, quando falamos de testes de sistema [2], podemos distinguir entre testes manuais, comumente realizados por testadores, com o objetivo de validar aquilo que está sendo desenvolvido, e testes automáticos, que são escritos por desenvolvedores e executados pela máquina.

Atualmente, devido à ampla utilização de sistemas distribuídos no desenvolvimento de aplicações *web*, surgiu uma nova técnica de testes, denominada de teste *end-to-end* (E2E) [3]. Esta técnica consiste em testes automáticos, em que o principal objetivo é testar a experiência do usuário final, simulando cenários reais de uso, validando o sistema em teste e garantindo a integração dos componentes e integridade de dados. Frequentemente, os testes E2E e de sistema podem ser considerados iguais, mas isso não é verdade, pois ambos são formas diferentes de teste com uma cobertura de testes diferente [4]. Enquanto o teste E2E verifica um fluxo de atividades do zero até o final do sistema, cobrindo todos os sistemas dependentes, o teste de sistema verifica a mesma funcionalidade com um conjunto diferente de entradas para avaliar a resposta.

Contudo, testes manuais são executados muitas vezes alinhados aos testes automáticos, utilizando diferentes abordagens, mas sempre visando detectar defeitos que não foram apontados pelos testes automatizados, visto que esse apontamento está profundamente atrelado ao código de teste implementado e seus objetivos. Essas abordagens foram de testes ad-hoc [5] e exploratório [6]. O teste ad-hoc, também conhecido por teste comportamental, é uma atividade não estruturada e improvisada de pesquisa de defeitos não planejados, utilizando-se de um método de tentativa e acerto de encontrar um *bug* no sistema. Por sua vez, o teste exploratório é uma metodologia criteriosa para o teste ad-hoc, estruturada, na qual um testador aprende sobre o sistema à medida que o explora e, eventualmente, desenvolve os testes usando o conhecimento adquirido.

Muitas vezes repetitivos, alguns desses testes manuais, especialmente os ad-hoc, poderiam ser automatizados, resultando numa maior completude e mais segurança. Além disso, quando os testes são feitos por desenvolvedores, é humanamente possível esquecer de testar alguma parte ou pular algum fluxo de utilização, ao mesmo tempo em que pode ser humanamente oneroso testar todos os fluxos de utilização para sistemas complexos. Esses fluxos de utilização são as sequências de passos que um usuário comumente faz em um sistema de cadastros. Por exemplo, ao chegar em uma determinada página de listagem de dados, é normal a navegação para página de edição de um item listado, ou de inserção de um novo item de dados. Ter testes automáticos com uma abordagem ad-hoc, garante que o usuário final não terá surpresas ao realizar tais passos, considerando, claro, que o sucesso na execução desses testes seja um fator determinante para a finalização de uma determinada *feature*.

Por outro lado, no contexto do desenvolvimento de aplicações web, as tecnologias predominantemente utilizadas são HTML, CSS e Javascript [7]. Essas tecnologias são conhecidas por promover uma interação amigável do usuário com a aplicação por

meio de interfaces. No entanto, na maioria dos casos, não se faz uso dessas tecnologias de forma nativa e/ou isolada, fazendo com que sejam utilizadas bibliotecas durante o processo de desenvolvimento da aplicação. Uma destas bibliotecas é o *React* [8], que utiliza-se da atual metodologia orientada a componentes, visando obter uma grande margem de reutilização, tornando o desenvolvimento mais ágil. Entretanto, tal abordagem torna perigoso realizar modificações, sejam melhorias ou correções, em componentes amplamente utilizados no sistema. Testes automatizados, quando são escritos para toda a aplicação, podem mitigar as consequências dessas alterações, dando mais confiança ao desenvolvedor ao realizá-las. Porém, existe um custo alto no desenvolvimento desses testes, principalmente quando alinhados com o desenvolvimento da aplicação web. Devido a tal custo, o time de desenvolvedores pode, por muitas vezes, decidir não adotar essa metodologia de testes.

O objetivo deste trabalho é melhorar a qualidade de aplicações web, cujos códigos fonte não tiveram testes realizados desde o começo do desenvolvimento, e dessa forma prover mais segurança na realização de modificações em código que podem impactar em várias partes do projeto, mediante metodologia de geração de testes ad-hoc automatizados. Essa metodologia não distingue aplicações que possuem ou não uma documentação de requisitos, pois na varredura realizada, sua finalidade é encontrar falhas, ou seja, um comportamento inesperado causado por problemas no desenvolvimento, e não defeitos no aspecto de enquadramento com requisitos.

Dessa forma, tendo uma definição das tecnologias utilizadas durante o desenvolvimento e realizando um estudo sobre o produto resultante de sua utilização, foi possível criar uma ferramenta capaz de identificar elementos selecionáveis, gerar um arquivo com código de testes que faça a seleção deles, e que executa as ações necessárias para simular a utilização de um usuário. Todavia, devido à possível complexidade atrelada a essa utilização, inicialmente foi reduzido o foco dos testes para garantir o sucesso em ações costumeiramente realizadas num sistema de cadastros, onde o usuário percorre um caminho da tela inicial até um formulário, o preenche e realiza a submissão.

O restante deste artigo está organizado como segue. A seção 2 aborda a solução proposta. A seção 3 discorre sobre a avaliação da solução proposta. Finalmente, a seção 4 conclui o trabalho e discorre sobre futuros trabalhos.

2. SOLUÇÃO PROPOSTA

No escopo deste trabalho, focamos inicialmente numa arquitetura que utiliza no *front-end* as seguintes tecnologias: (i) HTML, utilizada na construção de páginas na web; (ii) a linguagem de programação *JavaScript*, juntamente com a biblioteca de criação de interfaces *React*; e (iii) o uso de componentes do *Ant Design* [9]. Apesar dessa solução ter sido projetada para uma arquitetura seguindo esse modelo, nada impede que uma interface que utilize outras tecnologias, ou até HTML, CSS e *JavaScript* puro, possa usufruir dessa solução, desde que a interface siga uma forma padronizada dos elementos HTML, contendo um atributo identificador único.

Utilizando corretamente os componentes do *Ant Design,* pode-se ter uma identificação única para ser usada em testes, em tags selecionadas e diretamente relacionadas aos componentes que as originaram. Por exemplo, ao utilizar o componente *Input* [10], pode-se informar a propriedade *id*, e garantir que a tag input resultante é localizada pelo valor informado. Com isso, e baseado na análise do código HTML encontrado após cada ação realizada, é possível construir testes que percorrem todos os fluxos de utilização possíveis na aplicação web alvo. Também é possível utilizá-los para avaliar o código fonte atual, tanto por desenvolvedores durante a implementação, quanto pela execução em um *pipeline* de *continuous integration* (ou CI) [11] de um gerenciador de repositório de *software* baseado em *git*.

A solução proposta por este trabalho é um projeto de interface de linha de comando (CLI), chamado de **Cytestion**, capaz de gerar um arquivo de testes, que realiza testes ad-hoc na aplicação web alvo, de forma automática. Os testes utilizam-se de funções existentes num *framework open-source* de testes automatizados, chamado *Cypress* [12], que proporciona a criação de testes E2E, de integração e de unidade, de uma forma fácil e rápida, usando *JavaScript*. É possível criar diversos testes por tela do sistema, visando uma varredura completa, além da execução automática de todos esses testes, por linha de comando ou utilizando uma interface gráfica provida pelo *framework*. As funções utilizadas são aplicadas no desenvolvimento dos testes E2E e permitem visitar um endereço URL, selecionar, clicar, limpar e preencher elementos, além da escrita em arquivos e execução de comandos de sistema. Um exemplo dessa utilização pode ser visto na *Figura 1,* no qual a função *get* seleciona o campo identificado por *email1*, o aciona utilizando a função *click*, limpa seu conteúdo com a função *clear* e digita o email de exemplo através da função *type*. Além disso, podemos observar o uso da função *writeFile* para criação de um arquivo contendo o valor do campo *email1 e,* finalizando esse cenário de teste, a criação de uma cópia por meio do comando *cp*, executado pela função *exec*.

```
describe('My First Test', () => {
  it('Visits, gets, clear, types and create files', () => {
    cy.visit('https://example.cypress.io/commands/actions')
    .then(() => {
      cy.get('[id="email1"]').click().clear()
        .type('example@email.com');

      cy.get('[id="email1"]')
        .invoke('val')
        .then((email) => {
          cy.writeFile('email1.txt', email);
        });

      cy.exec('cp email1.txt email1-copy.txt');
    });
  });
});
```

Figura 1: Demonstração de algumas funções no *Cypress*

Com o intuito de garantir uma qualidade de código nos testes gerados, foi necessária a criação de funções personalizadas no *Cypress*, algo possível e encorajado, pois a redução de código é extremamente impactante, especialmente quando são criadas funções especializadas em realizar ações repetitivas, como selecionar e clicar em um determinado elemento apenas se este existir e seja visível. Outro cenário de impacto é o do preenchimento de campos *input*, que será feito de forma mais

efetiva, visando facilitar as ações necessárias para diferentes tipos de *input* existentes.

```
Cypress.Commands.add('writeContent', (actualId) => {
  cy.window().then((win) => {
    cy.writeFile(`tmp/${actualId.join('->')}`,
      win.document.body.outerHTML);
  });
});
Cypress.Commands.add('clearThenType', { prevSubject: true },
  (subject, text) => {
    cy.wrap(subject).clear().type(text)
      .should('have.value', text);
  }
);
Cypress.Commands.add('clickIfExist', (element) => {
  cy.get('body').then((body) => {
    if (body.find(element).length > 0) {
      cy.get(element).first().then(($id) => {
          if ($id.is(':visible')) {
            $id.click();
            cy.wait(200);
          }
        });
    }
  });
});
Cypress.Commands.add('fillInput', (element, value) => {
  cy.get('body').then((body) => {
    if (body.find(element).length > 0) {
      cy.get(element).then(($id) => {
        if ($id.is(':visible')) {
          cy.get(element).click().clearThenType(value);
        }
      });
    }
  });
});
Cypress.on('window:before:load', (win) => {
  cy.stub(win.console, 'error', (msg) => {
    cy.now('task', 'error', msg);
    throw new Error(msg);
  }).as('consoleError');
});
```

Figura 2: Principais funções personalizadas implementadas e definição do evento para detecção de falhas

As principais funções implementadas podem ser vistas na *Figura 2*, na qual há a função *writeContent,* responsável por escrever o conteúdo atual do HTML em um arquivo temporário, nomeado pelos identificadores percorridos pelo teste que a invocou. Além dela, temos a função *clearThenType*, que é utilizada para limpeza, preenchimento e asserção futura do que foi digitado em campos de *input* alfanuméricos. A função *clickIfExist,* a mais utilizada nessa metodologia proposta, cumpre o objetivo de selecionar e clicar em identificadores informados, existentes e visíveis, possibilitando a navegação na aplicação. Por último, temos as funções de preenchimento de campos, como a *fillInput,* que facilita as ações necessárias para preenchimento de um *input,* validando sua existência e visibilidade.

Essas funções são necessárias para execução dos testes gerados e, por conta disso, o próprio Cytestion garante sua existência no início de sua execução. Além dessas funções, a existência de um evento que dispara quando a página começa a carregar é fundamental para a detecção de falhas. Como visto na *Figura 2,* nesse evento criamos um *stub,* que funciona como uma forma de modificar uma função e delegar o controle de seu comportamento ao programador, para observar a chamada de *console.error* pela aplicação web, atribuindo isso como falha nos testes. Essa é uma forma eficaz de encontrar problemas quando utilizamos o *React,* pois quando erros de implementação, que ocasionam ou não na quebra da aplicação, são identificados, os mesmos são sempre revelados através dessa chamada de erro no console. Essa chamada é facilmente vista pelo programador, podendo então ser utilizada como indicador de falhas nos testes. Dessa forma, definimos que essas falhas podem ocorrer em três casos: quando um erro é lançado pela aplicação após sua quebra; quando o *assert* de que o *consoleError* não foi chamado resulta em falha, indicando que ocorreu um erro, porém não chegou a quebrar a aplicação; e quando o *assert* do que foi inserido no *input* falha.

O arquivo de testes é gerado utilizando técnicas de construção e manipulação de strings com *Node.js* [12]. O conteúdo do arquivo é construído no decorrer da execução do Cytestion, sendo incluído novos trechos de código de teste por cada fluxo corrente, em cada fase de geração completa, ou seja, não interrompida por se deparar com alguma falha. Nesse processo de investigação, os novos testes possuem o fluxo percorrido pelo seu precedente e o novo caminho a seguir. Dessa forma, os testes anteriores sempre são desabilitados na fase de geração seguinte, anulando a repetição deles, mas não descartando do resultado final, pois no caso de falha, aquele fluxo não prossegue, mas é importantíssimo para indicar onde se encontra um problema.

Para o entendimento do processo de concepção do arquivo de testes, foram definidos quatro etapas: o ponto inicial, análise, geração e limpeza. O ponto inicial e a limpeza são executados uma vez enquanto as etapas de análise e geração podem ser executadas diversas vezes até que não existam mais novos fluxos de utilização. Esse processo pode ser visto na *Figura 3*. No ponto inicial, o arquivo contendo o teste inicial é criado e executado, resultando em um arquivo com o HTML encontrado na página. Após isso, inicia-se o estágio de análise, onde o arquivo é analisado, extraindo e processando as informações necessárias para a execução do estágio de geração, que por sua vez, gera novos testes percorrendo a aplicação, que são executados, gerando novos arquivos com o HTML encontrado em cada teste. Esses dois estágios são repetidos até que o estágio de geração não consiga gerar testes sem evitar duplicação, ocasionando na execução do estágio de limpeza, que trata o arquivo de teste, removendo os testes que são subconjuntos de outros testes, como também comentários utilizados no processo e finalizando com a formatação do código contido no arquivo, resultando no arquivo dos testes finais.

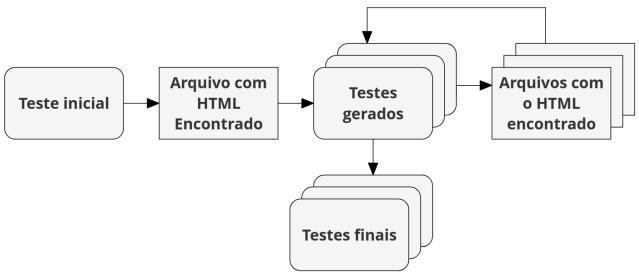

Figura 3: Visão geral do processo de geração dos testes

2.1 Ponto Inicial

O Cytestion utiliza um gerenciador de pacotes e projeto chamado *Yarn* [14]. Com o Yarn, temos a possibilidade de adicionar dependências ao projeto facilmente, além de criar comandos que funcionam como um atalho de execução no terminal, utilizando sempre o prefixo *yarn*. Nesse caso, foi criado o comando *generate-test* para execução do arquivo de entrada, chamado de *entrypoint.js*. Na sua primeira execução, são realizadas etapas de configurações locais, com uma disposição realizada exatamente como proposto pelo *Cypress*. Inicialmente, o endereço URL da aplicação web deve ser informado, podendo estar na própria máquina, em *localhost,* por exemplo. Essa informação é armazenada em um arquivo de configuração do *Cypress*, chamado de *cypress.json*.

Em seguida, outros dois arquivos de configuração, denominados *command.js* e *index.js*, são copiados de suas bases devidamente implementadas. O *command.js* contém todas as funções personalizadas necessárias, enquanto que o *index.js* contém o evento para fornecer a detecção de falhas. Idealmente, essa configuração é realizada na primeira execução da ferramenta, não sendo necessário realizar modificações. Contudo, o desenvolvedor tem total liberdade de adicionar funções, por exemplo, para realizar alterações nos testes gerados, tornando-os ainda mais específicos para sua aplicação.

Ao realizar as etapas anteriores, mais uma checagem é realizada. O arquivo de testes é criado sempre em um diretório nomeado com a data da sua geração. Assim, ao gerar mais de uma vez na mesma data, o Cytestion questiona sobre a possibilidade de sobrescrita do arquivo ou cancelamento da geração. Após isso o arquivo com o teste inicial é criado e executado, utilizando outro comando criado, chamado *test-file <nome-arquivo>*. Seu código vem de um arquivo base, contendo a princípio a definição do *beforeEach,* indicada pelo *Cypress* como função executada antes de cada teste. Nesse caso, é utilizada para visitar o endereço URL informado nas configurações locais. No corpo do testes, temos um *array* com os identificadores percorridos, que no momento é apenas *root*, a escrita do HTML atual em um arquivo temporário e a validação de que não houveram erros no console. O arquivo pode ser visto na *Figura 4*.

```
describe(`Automatic generated test file to click
        on elements on the page`, () => {
  beforeEach(() => {
    cy.visit('/');
    cy.wait(200);
  });
  //--CODE--
  it('Visits index page', () => {
    const actualId = ['root'];
    cy.writeContent(actualId);
    cy.get('@consoleError').should('not.be.called');
  });
  //--CODE--
  after(() => {
    cy.exec('yarn start-generate', { timeout: 600000 });
  });
});
```

Figura 4: Arquivo base contendo o teste inicial

Além disso, existem pontos importantes que podem ser vistos no código do arquivo base, como a utilização da função *wait* com parâmetro *time* de duzentos milissegundos (*time=200ms*), para que a página seja montada e renderizada antes da escrita do HTML. Isso é necessário exclusivamente pela forma como o *React* constrói e manipula os seus componentes, pois quando a página é visitada, ela foi montada, mas não necessariamente todos os componentes foram renderizados. É muito comum que modificações pontuais ocorram no HTML após a finalização de uma requisição ao *back end*, por exemplo. Definimos o valor do parâmetro *time* em *200ms* devido ao fato desse valor ser superior ao valor médio do tempo necessário para essas modificações na aplicação web na qual a ferramenta foi avaliada. Outro ponto importante é a apresentação do comentário, que funciona como um delimitador entre os testes, para que o Cytestion diferencie os blocos de teste no arquivo. Como último ponto, temos a definição do *after,* conhecida como a função do *Cypress* executada após a finalização de todos os testes do arquivo. Nesse caso, essa função está sendo utilizada como uma das pontas do laço que amarra o ciclo de geração, executando o comando *start-generate*, responsável por iniciar o estágio de análise e geração.

2.2 Análise

A análise é realizada em cada arquivo temporário encontrado, em cada fase da geração. Após a execução do teste inicial, apenas um arquivo é analisado. Em seguida, múltiplos arquivos podem ser criados, demandando uma grande quantidade de processamento, principalmente levando em consideração o tamanho do HTML contido nesses arquivos. Por conta disso, foi escolhida uma implementação pública [16] do algoritmo de Aho-Corasick [15], visando diminuir o overhead causado pelas pesquisas de strings, que são a parte majoritária deste estágio. Esta fase é realizada com tempo otimizado com a utilização desse algoritmo. Essa dependência proporciona um *array* com todas as ocorrências e posições encontradas, ficando necessário apenas o processamento desse *array* para remoção de duplicados, filtragem baseada na prioridade dos identificadores e pré-processamento do tipo geração.

As strings pesquisadas são *data-cy=*, *id=* e *class=*, possuindo exatamente essa ordem como preferência na seleção dos identificadores. Para alguns componentes do *Ant Design*, o identificador *id* é colocado nas tags de forma involuntária, utilizando outra propriedade bastante conhecida pelos desenvolvedores que já trabalharam com *React*, chamada *key*. Entretanto, para a maioria dos componentes, é necessário informá-la manualmente durante o desenvolvimento, para que seja possível uma identificação única no HTML final. Todavia, o *Cypress* descreve, como boas práticas, o uso de um identificador personalizado para ser utilizado nos testes, escolhido este, como sugerido, *data-cy* como identificador personalizado padrão e opção preferencial na utilização dos testes.

Por outro lado, foi necessária a inclusão do identificador *class,* especificamente para busca de utilização da classe utilizada em botões no *Ant Design,* chamada *de ant-btn.* O motivo desta inclusão é que existem componentes que apresentam botões *default*, como o *Modal* [17], caso no componente não seja informado a propriedade *footer*, ocasionando a inexistência de

identificadores únicos. Além disso, é válido considerar que possam existir botões pela aplicação sem identificadores atribuídos, por uma não adoção dessa boa prática, mas que ainda assim serão alcançados pelo Cytestion.

Após a pesquisa dessas strings, esta etapa realiza o trabalho de remover as ocorrências duplicadas nos seguintes aspectos:

1. Quando existe a ocorrência do mesmo identificador e seu respectivo valor em diferentes tags, indicando que o efeito de uma ação naquele elemento deve ser o mesmo em suas ocorrências;
2. A ocorrência de múltiplos identificadores, independente de seus valores, em uma mesma tag, onde nesse caso a filtragem é baseada na preferência dos identificadores, garantindo a unicidade de seleção;

Por último, é realizado um pré-processamento nas tags identificadas em campos *input*, visando identificar qual tipo de *input* se trata, baseado em qual classe utiliza e comparando com as contidas em campos de texto, números, datas e seleção de opções, gerados no uso dos seus respectivos componentes do *Ant Design*. Isso facilita o processo realizado no estágio de geração, para preenchimento correto dos dados.

2.3 Geração

Nesta etapa, tendo posse do processamento realizado na etapa de análise, é realizada a geração dos testes da próxima fase. Como demonstrado no arquivo base, os testes possuem um array dos identificadores percorridos, que é usado na nomeação do seu referente arquivo temporário, funcionando como uma forma do Cytestion identificar de qual teste partiram os novos identificadores encontrados no arquivo analisado. Com essa informação, conseguimos localizar o código do teste, que será copiado em um novo bloco, substituindo exatamente o local que criou o arquivo temporário, pelo novo percurso gerado. O resultado desse processo na fase seguinte à execução do teste inicial pode ser visto na *Figura 5.*

```
//--CODE--
it.skip('Click on element root->rc-tabs-1-tab-user', () => {
  const actualId = ['root', 'rc-tabs-1-tab-user'];
  cy.clickIfExist('[id="rc-tabs-1-tab-user"]');
  cy.writeContent(actualId);
  cy.get('@consoleError').should('not.be.called');
});
//--CODE--
```

Figura 5: Teste gerado ao encontrar o id=”rc-tabs-1-tab-user” após executar o teste inicial

Analisando a figura 5, podemos perceber que o título remete à última ação realizada pelo novo teste, ou seja, indica o clique no elemento encontrado, assim como o *array actualId* possui seu o identificador. Também, a chamada da função de escrita do conteúdo foi substituída pela função personalizada de clique e depois novamente a escrita do HTML. Outro detalhe importante nesse teste exemplificado é a utilização do *skip,* que serve para o *Cypress* pular o teste durante a execução. Nesta etapa, essa função é aplicada para garantir que os testes realizados anteriormente não sejam executados, independente se seus fluxos foram ramificados.

Essa abordagem de geração é realizada para todos os identificadores encontrados nos novos arquivos temporários, em cada etapa. Podemos perceber que estamos criando uma árvore de fluxos possíveis, e precisamos prevenir que um novo nó não seja igual a um nó existente, para isso vamos realizar verificações de existência do identificador. A existência nesse caso, seria tanto em testes anteriores, como nos novos testes que estão sendo gerados, especificamente no último identificador contido no *actualId*. Dessa forma, garantimos que surgiu um novo fluxo.

O objetivo da geração é agregar novos blocos de códigos no arquivo de testes, que possam percorrer a aplicação web através dos cliques em botões ou links, para verificar se existe algum erro em seus usos. Além disso, ao encontrar identificadores referentes a campos *input* em uma mesma página, estes são agrupados e seus preenchimentos são feitos em um único teste, finalizando com a submissão do possível formulário presente na página. Para realizar esse preenchimento dos campos alfanuméricos, é utilizado uma dependência chamada *faker [18],* para produzir valores randômicos e digitá-los nesses campos. Além de alfanuméricos, são suportados os campos de data e seleção de opções, que possuem um comportamento semelhante de clicar no campo da página e escolher uma das datas/opções possíveis, mas utilizam funções personalizadas diferentes. Um teste contendo esses preenchimentos pode ser visto na *Figura 6*.

```
it(`Filling values root->rc-tabs-1-tab-despesa->edit-button->
  nome-descricao-valor-usuarioId-tipoPagamentoId-parcelas-periodo
  and submit`, () => {
  cy.clickIfExist('[id="rc-tabs-1-tab-despesa"]');
  cy.clickIfExist('[data-cy="edit-button"]');
  cy.fillInput('[id="nome"]', 'Bike');
  cy.fillInput('[id="descricao"]', 'Nebraska');
  cy.fillInput('[id="valor"]', '300');
  cy.fillInputSelect('[id="usuarioId"]');
  cy.fillInputSelect('[id="tipoPagamentoId"]');
  cy.fillInput('[id="parcelas"]', '5');
  cy.fillInputDate('[id="periodo"]',
    '[class="ant-picker-cell-inner"]');
  cy.submitIfExist('.ant-form');
  cy.get('@consoleError').should('not.be.called');
});
```

Figura 6: Teste gerado com preenchimento de campos *input*

Como última função realizada nesta etapa, quando pelo menos um identificador com um novo fluxo é encontrado, temos novamente a chamada para execução do arquivo de teste, utilizando o mesmo comando citado no ponto inicial, chamado *test-file <nome-arquivo>*. Aqui fechamos o laço do ciclo de geração, no qual, ao final da execução, uma nova fase de análise e geração se inicia.

2.4 Limpeza

Quando a geração não encontra nenhum identificador com um novo fluxo, inicia-se a etapa final de limpeza. Os arquivos temporários criados por cada teste, contendo o HTML encontrado no percurso de geração, são criados em um diretório chamado *tmp,* que termina sendo limpo nesse estágio. Após isso, é realizada uma filtragem nos testes, eliminando aqueles que possuem os identificadores percorridos como subconjunto de algum outro teste. Dessa forma, removemos os testes repetidos e diminuímos as linhas de código, cobrindo exatamente os mesmos fluxos. Só após essa etapa, é de fato realizada uma limpeza nos testes

restantes, removendo artifícios de geração do seu conteúdo. Mesmo se tratando de testes diferentes, podemos observar a ausência desses detalhes nas *Figuras 5 e 6*. Primeiramente, os testes finais não vão possuir o *skip* em sua definição que é removido juntamente com o *array* dos identificadores, a chamada da função *writeContent* e os comentários para delimitação dos blocos de teste.

Dado que a ferramenta utiliza manipulação de strings para gerar o código, é válido ressaltar que a atividade de assegurar a indentação correta possui uma certa dificuldade. Por mais que a indentação não seja algo obrigatório para a interpretação do código *JavaScript*, um código indentado contribui positivamente na legibilidade por parte do desenvolvedor. Dessa forma, esta etapa utiliza uma dependência chamada *prettier* [19], para realizar a formatação do arquivo com os testes finais, resultando finalmente no encerramento da execução do Cytestion.

2.5 Ferramenta em Uso

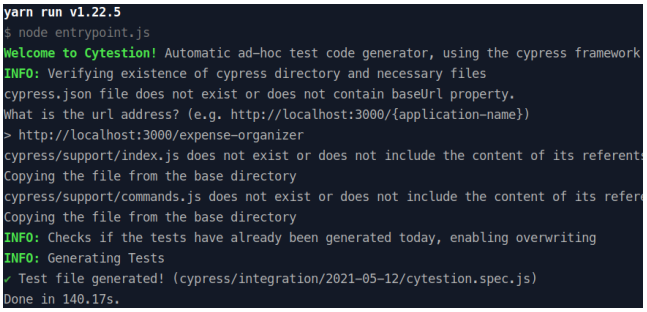

Figura 7: Demonstração de utilização do Cytestion

Compreendido o funcionamento do Cytestion, vamos demonstrar seu uso. A ferramenta foi desenvolvida pensando em sua utilização como *submodule* [20] num sistema de controle de versão como *git*, durante o desenvolvimento de aplicações web. Dessa forma, ela pode ser utilizada localmente gerando os testes após a finalização de uma *issue,* por exemplo. Para apresentar esse uso, vamos executar a geração em uma aplicação web desenvolvida como prova de conceito desse trabalho, na qual será mais detalhada na seção 3. Como pode ser visto no guia escrito no repositório da ferramenta apontado nos links úteis, é necessário ter o *Node.js* e *Yarn* instalados na máquina. Inicialmente, temos que instalar as dependências da ferramenta, executando o comando *yarn install* ou apenas *yarn*. Após realizar a instalação, basta executar o Cytestion através do comando *yarn generate-test* e seguir os passos apresentados no CLI. A Figura 7 demonstra a execução do comando para geração dos testes, conferindo exatamente os passos de configuração descritos no ponto inicial e finalizando apontando a localização do arquivo gerado. Esse arquivo pode ser acessado nos links úteis desse trabalho.

3. AVALIAÇÃO

A avaliação da ferramenta foi feita a partir do desenvolvimento de uma simples aplicação web, que possibilita o cadastro de informações para organização financeira, chamada *Expense Organizer*. Nos links úteis encontra-se o link para seu repositório. A aplicação possui as funcionalidades de CRUD (Inclusão/ Consulta / Edição e Exclusão) para as entidades usuário, tipo de pagamento, despesa e renda, demonstrando um balanço mensal na página inicial. Sua arquitetura no *front end* utilizou exatamente as tecnologias definidas como escopo da ferramenta, além de possuir todos os componentes do *Ant Design* alcançáveis pelo Cytestion, no que diz respeito a precisão e qualidade dos testes gerados. Não foi necessário hospedar a aplicação, pois a execução em ambiente de desenvolvimento local foi suficiente para avaliar a ferramenta e os testes gerados.

Após utilizar a ferramenta para geração dos testes, uma análise de cobertura de interfaces foi aplicada nos testes resultantes, visando mensurar sua capacidade de alcançar todas as telas e ações disponíveis para o usuário. Além disso, foi avaliada a aplicabilidade da ferramenta no uso de detecção de falhas após modificações realizadas no código fonte. Por último, a utilização dos testes gerados como testes de regressão. Uma vez que o processo de geração de testes passou a ser uma atividade automatizada e que, por sua vez, possui um custo zero de desenvolvimento, é válido e até mesmo indicado que o sucesso na execução dos mesmos seja levado em consideração durante o processo de integração contínua e evolução do código fonte.

3.1 Cobertura de interfaces

Para o entendimento dessa avaliação, precisamos inicialmente definir o significado de interfaces para esse contexto. Uma interface pode ser definida como uma tela que apresenta informações e interatividade para com o usuário, onde qualquer mudança na funcionalidade de uma interação, implica em uma interface diferente. Por exemplo, uma tela de inclusão e uma tela de edição de uma mesma entidade, embora possuam formulários quase idênticos, são interfaces distintas. No *Expense Organizer*, temos uma estrutura baseada em abas, onde a primeira aba, *Welcome*, apenas exibe informações do balanço de cada mês, organizado também em abas. Essa funcionalidade apenas proporciona a navegação entre os meses, podendo ser considerado um fluxo curto na aplicação. Por outro lado, temos as abas de Usuário, Tipo de Pagamento, Despesa e Renda, cada uma possuindo as mesmas funcionalidades de CRUD, implicando no mesmo fluxo de utilização, em entidades diferentes.

Quando acessamos a aba de Usuário, temos três possibilidades de interação. Podemos clicar no botão de exclusão, edição e criação de um registro. Ao clicar em excluir, temos a opção de clicar em cancelar a deleção ou em efetivá-la. Ao clicar no botão de criar, temos a opção de preencher os campos e realizar a submissão do formulário ou de clicar em submeter sem preencher os campos, a fim de reproduzir um cenário de erro na aplicação. Da mesma forma, no botão de editar, temos as mesmas opções do botão de criar, com a diferença de que o preenchimento dos campos está alterando um dado existente e, consequentemente, realizando outra operação, nesse caso a de edição.

No arquivo de teste gerado, disponível nos links úteis, podemos constatar que todas as possibilidades descritas anteriormente foram alcançadas, para todas as entidades do sistema, resultando em 100% da cobertura de interfaces da aplicação web desenvolvida. Os fluxos curtos de navegação entre os meses são percorridos, assim como todos os fluxos de CRUD nas demais abas. Na *Figura 8*, temos o fluxo na aba Usuário como exemplo

do que ocorre com as demais abas. Iniciando com o identificador root do teste inicial, é clicado no identificador *rc-tabs-1-tab-user*, alterando a interface para a tabela de listagem de usuários. Então temos três ramificações: o clique no botão de editar identificado pelo nó *edit-button;* o clique botão de criar, identificado pelo *new-button;* e, por último, o clique no botão excluir, mostrado pelo nó *delete-button*. Cada uma dessas três ramificações criam duas novas, sendo semelhantes as ramificações dos fluxos de edição e criação (nós *edit-button* e *new-button, respectivamente)*. Os testes gerados por estes fluxos preenchem os campos identificados por nome, sobrenome e email e realizando a submissão; e, seguindo um fluxo sem preenchimento dos campos e apenas clicando no *submit-button*. Segue-se, pois, a ramificação do fluxo de exclusão, identificada pelo nó *delete-button*, que clica no botão de cancelar, identificado pela sua *class ant-bnt* e confirma a ação através do clique no botão da class *ant-btn ant-btn-dangerous.*

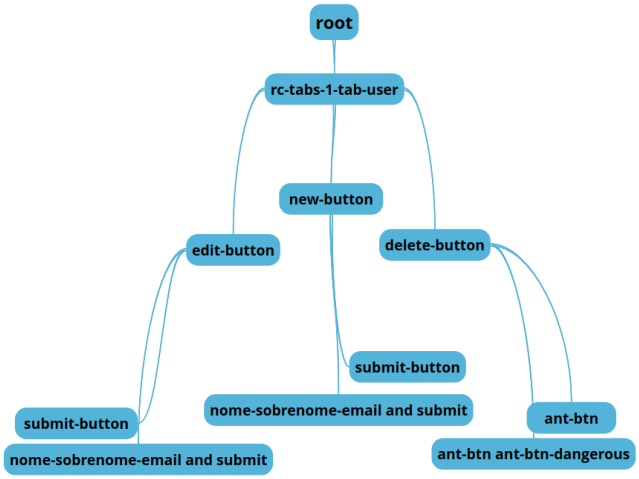

Figura 8: Fluxo de utilização alcançado na entidade usuário

3.2 Detecção de falhas

Para realizar a avaliação de detecção de falhas por meio da execução do Cytestion, iremos introduzir falhas no código fonte da aplicação web. Essas falhas são erros comuns de programação ocorridos durante o processo de desenvolvimento de software, como, por exemplo, a chamada de uma função inexistente ou o acesso a propriedades de um objeto sem checar sua existência, dentre outros. Nesse caso, na aba de Despesa, vamos inserir uma falha denominada intratável, que causa a quebra da aplicação. A falha será inserida através da mudança de nome da função responsável por retornar o botão de edição na tabela de listagem da página. Além dessa falha, no formulário da aba de Tipo de Pagamento, vamos modificar o nome da função chamada na submissão, causando uma falha tratável na aplicação. Essas alterações podem ser visualizadas na *Figura 9.*

```
render: (row) => {
  return (
    <div className='actions'>
      {this.getDefaultEdit(row)}
      {this.getDefaultEdit2(row)}
      {this.getDefaultDelete(row)}
    </div>
  );
```

```
<Form
  {...layout}
  onFinish={() => store.save()}
  onFinish={() => store.save2()}
  initialValues={store.object}
  onFieldsChange={(_, allFields) =>
    store.updateObject(fieldsToObject
  }
```

Figura 9: Erros introduzidos na aplicação web

Com as falhas inseridas, ao executar o Cytestion o comando de geração retorna um erro, indicando que pelo menos um dos testes falhou durante a execução, mas o arquivo de teste foi gerado. Realizando uma comparação entre o arquivo gerado anteriormente (onde não havia falhas no sistema) e o arquivo com falhas, através da execução por CLI, com o comando *yarn test-file <arquivo>,* podemos notar uma quantidade reduzida de testes. A diminuição na quantidade de testes produzidos ocorre em função do fluxo seguindo a aba de Despesa ter sido eliminado quando o primeiro teste clicou nela e a aplicação quebrou. Além disso, outros três testes relacionados à submissão feita no formulário do Tipo de Pagamento falharam. Foi constatado que os motivos das falhas são indicados de forma detalhada, anteriormente ao resultado final. A comparação entre os resultados pode ser vista na Figura 10.

```
(Run Finished)

    Spec                                  Tests  Passing  Failing
  ✔ 2021-05-12/cytestion.spec.js  00:53     30       30        -
  ✔ All specs passed!             00:53     30       30        -

(Run Finished)

    Spec                                  Tests  Passing  Failing
  ✖ 2021-05-13/cytestion.spec.js  00:59     25       21        4
  ✖ 1 of 1 failed (100%)          00:59     25       21        4
```

Figura 10: Comparação entre o resultado dos testes gerados antes e depois da inserção de falhas

3.3 Teste de regressão

Conforme demonstrado, a execução do Cytestion para detecção de erros é eficaz, porém ela é mais apropriada quando estamos desenvolvendo novas *features* na aplicação, uma vez que essas *features* ainda não possuem testes gerados. Quando se trata de modificações e os testes já foram gerados anteriormente, eles passam a ser utilizados como teste de regressão, indicando se houve algum erro inserido devido à atualizações no código fonte, apenas executando a suíte de testes gerada novamente. Com o intuito de reproduzir o cenário de regressão descrito, vamos submeter a aplicação que teve erros inseridos intencionalmente (seção 3.1) à execução do arquivo que foi gerado com a aplicação em seu estado estável através da interface gráfica disponibilizada pelo *Cypress*. Podemos então ver as ações descritas no código sendo realizadas visualmente, e com isso perceber ainda mais rápido como um erro ocorreu. *Na Figura 11* temos a mensagem do erro encontrado em todos os testes referentes ao fluxo atualmente quebrado da aba de Despesas.

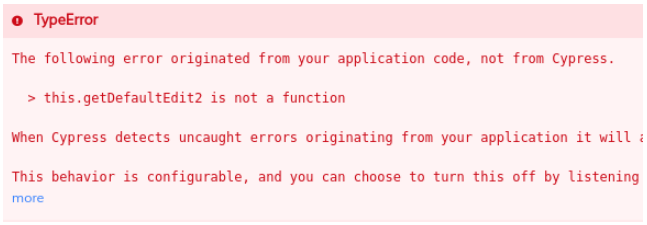

Figura 11: Mensagem do erro indicando função inexistente

4. CONCLUSÃO

O desenvolvimento da ferramenta iniciou com projeto e arquitetura da aplicação web na qual ela foi avaliada. Nessa fase, foi realizado um estudo sobre os componentes e tecnologias do escopo da ferramenta, visando identificar formas de criar testes que garantisse sua usabilidade. Ao finalizar a implementação do sistema, aliado ao estudo do *framework Cypress*, foram criados manualmente esboços de testes para extrair características genéricas do código de teste, com o objetivo de automatizar sua geração. Nesse estudo, foi constatado a necessidade de utilizar uma abordagem investigativa em fases, a fim de executar uma análise minuciosa e evolutiva, onde os testes são enriquecidos a cada fase. Dessa forma, buscou-se meios de integrar execução de testes e geração de código, de forma recursiva, concluindo ao atingir o fim de todos os fluxos possíveis. Encontramos como opção viável a execução de comandos por CLI do *Cypress* e a comunicação entre processos baseada em arquivos temporários.

Uma limitação encontrada durante o desenvolvimento foi a necessidade do uso da função *wait* nos testes para aguardar a renderização completa da página. Todas as abordagens existentes no *Cypress* [21] para contornar seu uso partem do princípio do conhecimento do efeito colateral de uma ação, a fim de validar sua ocorrência para dar continuidade nos testes. Como seguimos com uma abordagem de não conhecer a aplicação alvo, a utilização da função *wait* foi a única opção encontrada.

Futuramente, pode-se aprimorar o Cytestion fazendo com que a ferramenta seja independente do uso da biblioteca de componentes que foi escolhida, a Ant Design. Para tal, será necessário uma análise mais profunda do HTML, visando extrair e processar características próprias da aplicação para uso nos testes. Uma forma de contornar a necessidade de uso da função *wait* é fazer uso de sinalizadores, que indicam a finalização da renderização da página [22]. Outra maneira para contornar o uso dessa função seria a implementação de reconhecimento do código fonte de uma aplicação que faz uso das tecnologias do escopo desse trabalho, a fim de inferir o comportamento esperado em cada ação que será realizada nos testes.

Diante do exposto, concluímos que o uso do Cytestion pode ser um forte aliado no desenvolvimento de aplicações web simples, principalmente as baseadas em CRUDs. A ferramenta proporciona a detecção de falhas rapidamente, além de garantir a usabilidade da aplicação. Tudo isso associado a um custo zero na criação de testes, uma vez que os mesmos são gerados automaticamente, sem *overhead* no processo de desenvolvimento.

5. AGRADECIMENTOS

Agradeço a Deus, pela minha vida, e por ter permitido que eu tivesse saúde e determinação para não desanimar durante todos os meus anos de estudos. Agradeço a todo corpo docente do curso de Ciência da Computação, da Universidade Federal de Campina Grande, por me ensinarem e compartilharem seus conhecimentos durante todo o período da graduação, fundamentais para minha formação acadêmica. Agradeço principalmente ao meu orientador Prof. Dr. Cláudio de Souza Baptista por todas as oportunidades e experiências compartilhadas que me permitiram apresentar um melhor desempenho no meu processo de formação profissional ao longo do curso. Agradeço ao meu co-orientador e gerente Prof. Dr. Hugo Feitosa de Figueirêdo pelas correções e ensinamentos que contribuíram imensamente na realização deste trabalho.

Agradeço também aos meus pais, por nunca terem medido esforços para me proporcionar um ensino de qualidade durante todo o meu período escolar e compreenderem a minha ausência enquanto eu me dedicava à realização deste trabalho. Agradeço a minha esposa Ingrid Rodrigues pela paciência, apoio e amor, demonstrados ao longo dos últimos quatro anos que me conduziram até este momento. Agradeço ao meu irmão Márcio Torres por todo suporte, que muito contribuíram para a realização deste trabalho. E por fim, agradeço aos meus amigos adquiridos ao longo do curso, especialmente Francisco Edeverton, Mateus Cunha e Pedro Wanderley, que foram fundamentais para conclusão deste trabalho.

6. REFERÊNCIAS

[1] Teste de Software - The Importance of Software Testing. https://www.karllhughes.com/posts/testing-matters

[2] Teste de Sistema - What is System Testing in Software Testing? https://www.softwaretestinghelp.com/system-testing/

[3] Teste E2E - What is End-to-End (E2E) Testing? https://www.katalon.com/resources-center/blog/end-to-end-e2e-testing

[4] Teste de Sistema vs Teste E2E - System Testing Vs End-To-End Testing: Which One Is Better To Opt? https://www.softwaretestinghelp.com/system-vs-end-to-end-testing/

[5] Teste Ad-Hoc - Ad-Hoc Testing: How To Find Defects Without A Formal Testing Process. https://www.softwaretestinghelp.com/ad-hoc-testing/

[6] Teste Exploratório - What Is Exploratory Testing In Software Testing. https://www.softwaretestinghelp.com/what-is-exploratory-testing/

[7] JavaScript - o que é, para que serve e como funciona o JS? https://kenzie.com.br/blog/javascript

[8] React - Uma biblioteca javascript para criar interfaces de usuário. https://pt-br.reactjs.org

[9] Ant Design - An enterprise-class UI design language and React UI library. https://ant.design/docs/react/getting-started

[10] Input - A basic widget for getting the user input is a text field. https://ant.design/components/input/

[11] Pipeline - GitLab Docs - CI/CD pipelines https://docs.gitlab.com/ee/ci/pipelines

[12] Cypress - What Cypress is and why you should use it. https://docs.cypress.io/guides/overview/why-cypress

[13] Node.js - O que é, como funciona e quais as vantagens. https://www.opus-software.com.br/node-js

[14] Yarn - Yarn is a package manager that doubles down as project manager. https://yarnpkg.com/

[15] Aho-Corasick - Alfred Aho and Margaret Corasick. 1975. Efficient string matching: An aid to bibliographic search. Commun. ACM 18 (06 1975), 333–340. https://cr.yp.to/bib/1975/aho.pdf

[16] Implementação pública - Implementation of the Aho-Corasick string searching algorithm. https://www.npmjs.com/package/ahocorasick

[17] Modal - Modal to create a new floating layer over the current page to get user feedback or display information. https://ant.design/components/modal

[18] Faker - generate massive amounts of fake data in the browser and node.js. https://www.npmjs.com/package/faker

[19] Prettier - Formats your JavaScript using prettier followed by eslint --fix. https://www.npmjs.com/package/prettier-eslint

[20] Submodules - Git Tools Submodules. https://git-scm.com/book/en/v2/Git-Tools-Submodules

[21] Cypress Blog - When Can The Test Start? https://www.cypress.io/blog/2018/02/05/when-can-the-test-start/

[22] Better world by better software - Set flag to start tests. https://glebbahmutov.com/blog/set-flag-to-start-tests/