Student's perception of Cypress software testing

1st Juscélio de Oliveira Reis
Computer Engineering Master Degree
Universidade da Beira Interior
Covilhã, Portugal
juscelio.reis@ubi.pt

2nd Rene Jerez
Computer Engineering Master Degree
Universidade da Beira Interior
Covilhã, Portugal
rene.jerez@ubi.pt

***Abstract*—This report explores the application of Cypress, a modern web testing framework, within the context of an insurance sales software hosted on Azure Cloud. The software, developed with Vue.js for the frontend and C# .NET for the backend, simplifies the sale of insurance claims to brokers through a concise four-step process. The report explore Cypress's features and functionality, highlighting its advantages over others testing tools. By the end of this report, readers will have a solid grasp of Cypress and how it can enhance software testing processes.**

***Index Terms*—Testing, Automation, Tool, Framework**

I. Introduction

In today's rapidly evolving world of software development, the importance of software testing cannot be underestimated. The effectiveness and efficiency of web application testing can greatly impact the overall quality of a product. This report sets out on a journey to delve into the capabilities of Cypress, an innovative and user-friendly testing framework. Cypress has garnered a strong following for its unique approach to end-to-end testing, providing developers and testers with a seamless experience in writing, debugging, and executing tests, while also offering real-time feedback. Through our exploration of Cypress, this report offers a comprehensive overview of its key features, highlighting its advantages over traditional testing frameworks and its ability to streamline the testing process. Our goal is to equip readers with valuable insights into the world of Cypress.

The primary goal of this report is to help you understand Cypress and how it works in software testing. Our journey will take us through the myriad of features, functionalities, and advantages that set Cypress apart from other testing tools. As we delve into real-world testing scenarios, we will also share invaluable insights and best practices for implementing Cypress effectively. Furthermore, we will examine case studies and use cases that demonstrate how Cypress can bolster web application testing, from small development teams to large-scale projects. Upon completion of this report, readers will possess a robust understanding of Cypress and its potential to enhance web application testing processes, making it an invaluable resource for developers, testers, and quality assurance professionals.

Identify applicable funding agency here. If none, delete this.

II. Application context

This report covers an example of insurance sales software deployed in the Azure Cloud. It is built with Vue.js for user interface and has a backend developed in C# .NET, as you can see in Figure 1. The main purpose of this application is to simplify the process of selling insurance policies to brokers. It does so in four straight steps. The first part involves validation of policyholder information, the second step involves insurance party data, the third step focuses on assessing the risk associated with the offer, and the closing step do the four comply with the financial side of the offer

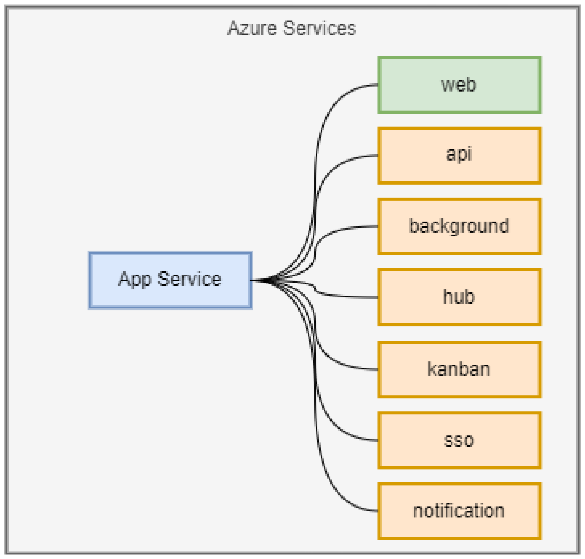


Fig. 1. Application architecture

III. Test Scenario

Test scenarios are of utmost importance in the domain of software quality assurance, as they serve as a critical factor in verifying the expected performance of a software application or system. Test scenarios consist of organized sequences of actions and conditions that are employed to assess the performance and dependability of software across a range of settings. The aforementioned scenarios serve as simulations of real-world usage, facilitating the identification of problems, weaknesses, and places for development within the software. Test scenarios play a crucial role in the testing process as they offer a methodical technique to verify that the software adheres to the necessary criteria of quality and performance. This paper presents a test scenario that is centered around the registration of insurance proposals. The purpose of this scenario is to provide an informative example of how test scenarios are utilized to improve the quality and dependability of software systems.

A. Context

In the role of an insurance broker, it is essential to validate the insurance proposal registration process, ensuring that they are filled out correctly and sent to the insurance company's system. This test scenario encompasses a complete flow, from authentication in the platform to the completion of the proposal registration. The goal is to ensure that the system functions flawlessly and that proposals are processed successfully.

B. Actions

After authentication, the user selects the option to create a new proposal. Data is provided for the policyholder, the insured party, and details related to the desired insurance's risk. Subsequently, financial data is input for the chosen proposal. The system displays a background processing screen, where various validation engines determine whether the proposal can be accepted. Finally, the result of the proposal registration is presented on the screen, demonstrating a successful registration.

This test scenario comprehensively addresses the insurance proposal registration process, ensuring that all steps are executed successfully. It plays a vital role in validating the system and ensuring that the proposal registration workflow functions as expected.

IV. Cypress

Since its first release in September 2017, Cypress has rapidly established itself as a newcomer in the realm of software testing. It presents several advancements over its well-known counterpart, Selenium, offering a fresh perspective on automated testing.

A. Architecture

The architecture of Cypress is unique and efficient. According to Jonas Verhoelen [8], Cypress is split into a server side and a client side that runs inside a browser. This browser is where Cypress sets itself apart—it embeds both the application under test and the Cypress framework within an iFrame. This allows the tests to run directly alongside the application, facilitating direct communication with the Node.js server through WebSockets and the ability to execute shell commands. A pivotal feature of this architecture is the proxy server, which enables Cypress to intercept and wait for specific network requests, thus optimizing test performance. Verhoelen's insights provide a window into the strategic design of Cypress's architecture as you can see in figure 2.

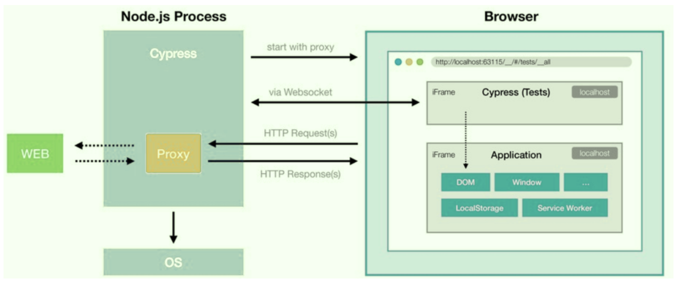


Fig. 2. Cypress architecture

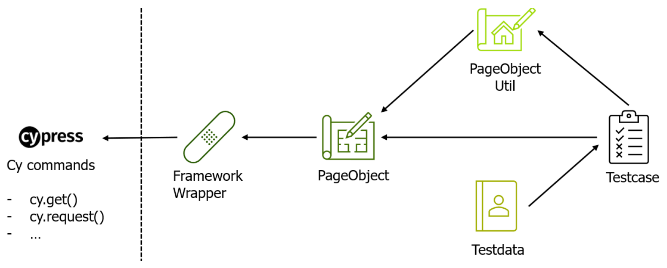


Fig. 3. Cypress Diagram

B. Diagram

Matthias Baldi [9] further explains the Cypress execution process, which is designed to be lean and maintainable with the use of selectors and test files. This design significantly reduces the overhead traditionally associated with keeping tests up to date during the development cycle. It also promotes the integration of testers into the development team, allowing for a more collaborative and agile workflow. When tests are coded, the approach taken by Cypress ensures that they are self-documented—as long as the code is written with meaningful method names and parameters, as recommended by Baldi. This approach not only makes tests easier to understand but also turns them into a form of documentation that describes the expected behavior of the application as you can see in figure 3.

These insights from Verhoelen and Baldi highlight the thoughtful design and user-centered approach that Cypress brings to software testing, marking it as a significant tool for modern development teams.

C. End to End Testing

According to the Cypress documentation [10], End-to-End (E2E) testing with Cypress is a strategic testing method that validates the entire application from the front end to the back end. It is designed to emulate real user behavior, covering the complete application flow to ensure that every function performs as intended when executed in a real-world scenario. Cypress excels in this domain by providing a robust platform that not only runs tests against a browser but also allows for the observation of network traffic and automatic execution of commands within the context of the application. By automating the user interactions and verifying the outcomes, Cypress's E2E testing confirms the integrated performance of all the application's components, thus safeguarding the user experience against potential disruptions.

D. Test Automation Insurance Aplication

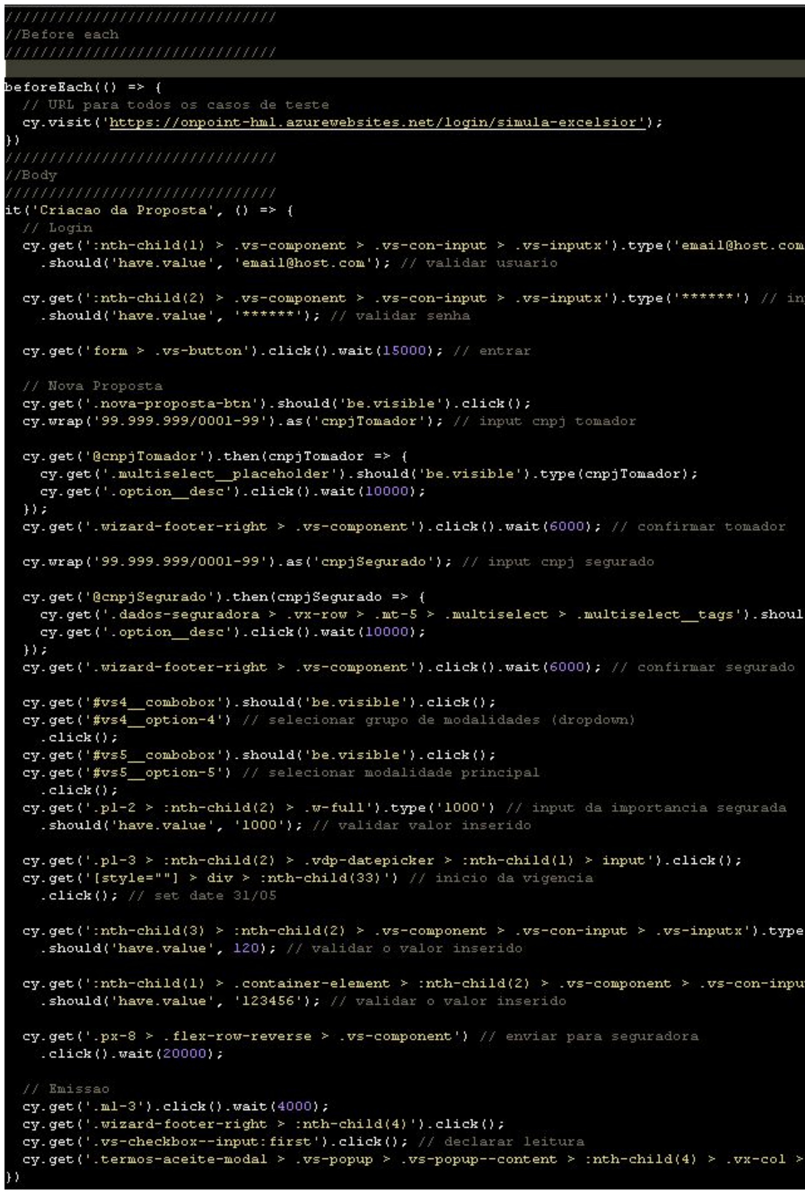
In the Insurance aplication we did a test according the scenario said before and the steps was:

1) The test initializes by starting a block with it 'Criacao da Proposta' which defines a single test case.
2) It performs a login operation by filling in a user email and password, then validates the input values.
3) The script waits for 15 seconds after clicking the login button to ensure the subsequent page loads.
4) It then starts the process of creating a new proposal by clicking the 'nova-proposta-btn'.
5) The test inputs and confirms a ID company number called "CNPJ" for 'Tomador'
6) It waits for 10 seconds, untill the page or a service to respond, then confirms the input.
7) Similarly, it inputs and confirms a ID company number called "CNPJ" for 'Segurado'.
8) The test selects options from dropdown menus for a group of modalities and the main modality.
9) It enters a value for 'importancia segurada' (insurance amount), and sets a date for 'inicio da vigencia' (start of validity).
10) The test inputs a time frame in days and an 'edital' number (notice number), validating each input.
11) It sends the proposal to the insurance company and waits for 20 seconds.
12) For the emission process, the test performs a series of clicks to navigate, declare reading of terms, and finalize the emission.

After the test steps we can handle post-test activities such as logging results, checking for visual differences with a tool like 'Eyes-Cypress', and potentially throwing an error if differences are detected. Eyes-Cypress is an integration of Applitools Eyes into the Cypress test automation framework. Applitools Eyes is a visual testing and monitoring tool that provides automated visual UI testing by capturing screenshots during tests and comparing them to baseline images to detect changes or anomalies. This integration allows developers and QA engineers to perform visual testing as part of their Cypress E2E tests.

E. Code Snnipet

The code is written in JavaScript, which is the programming language used by the Cypress testing framework. According LambdaTest [11] JavaScript is a versatile, high-level scripting language that is widely used for web development, among other things. Within the context of Cypress, JavaScript is used to write test scripts that automate interactions with a web application to verify its functionality. Folowing you can see a code wrote:



```
///////////////////////////////
//Before each
///////////////////////////////

beforeEach(() => {
  // URL para todos os casos de teste
  cy.visit('https://onpoint-hml.azurewebsites.net/login/simula-excelsior');
})
///////////////////////////////
//Body
///////////////////////////////
it('Criacao da Proposta', () => {
  // Login
  cy.get(':nth-child(1) > .vs-component > .vs-con-input > .vs-inputx').type('email@host.com
    .should('have.value', 'email@host.com'); // validar usuario

  cy.get(':nth-child(2) > .vs-component > .vs-con-input > .vs-inputx').type('******') // in
    .should('have.value', '******'); // validar senha

  cy.get('form > .vs-button').click().wait(15000); // entrar

  // Nova Proposta
  cy.get('.nova-proposta-btn').should('be.visible').click();
  cy.wrap('99.999.999/0001-99').as('cnpjTomador'); // input cnpj tomador

  cy.get('@cnpjTomador').then(cnpjTomador => {
    cy.get('.multiselect__placeholder').should('be.visible').type(cnpjTomador);
    cy.get('.option__desc').click().wait(10000);
  });
  cy.get('.wizard-footer-right > .vs-component').click().wait(6000); // confirmar tomador

  cy.wrap('99.999.999/0001-99').as('cnpjSegurado'); // input cnpj segurado

  cy.get('@cnpjSegurado').then(cnpjSegurado => {
    cy.get('.dados-seguradora > .vx-row > .mt-5 > .multiselect > .multiselect__tags').shoul
    cy.get('.option__desc').click().wait(10000);
  });
  cy.get('.wizard-footer-right > .vs-component').click().wait(6000); // confirmar segurado

  cy.get('#vs4__combobox').should('be.visible').click();
  cy.get('#vs4__option-4') // selecionar grupo de modalidades (dropdown)
    .click();
  cy.get('#vs5__combobox').should('be.visible').click();
  cy.get('#vs5__option-5') // selecionar modalidade principal
    .click();
  cy.get('.pl-2 > :nth-child(2) > .w-full').type('1000') // input da importancia segurada
    .should('have.value', '1000'); // validar valor inserido

  cy.get('.pl-3 > :nth-child(2) > .vdp-datepicker > :nth-child(1) > input').click();
  cy.get('[style=""] > div > :nth-child(33)') // inicio da vigencia
    .click(); // set date 31/05

  cy.get(':nth-child(3) > :nth-child(2) > .vs-component > .vs-con-input > .vs-inputx').type
    .should('have.value', 120); // validar o valor inserido

  cy.get(':nth-child(1) > .container-element > :nth-child(2) > .vs-component > .vs-con-inpu
    .should('have.value', '123456'); // validar o valor inserido

  cy.get('.px-8 > .flex-row-reverse > .vs-component') // enviar para seguradora
    .click().wait(20000);

  // Emissao
  cy.get('.ml-3').click().wait(4000);
  cy.get('.wizard-footer-right > :nth-child(4)').click();
  cy.get('.vs-checkbox--input:first').click(); // declarar leitura
  cy.get('.termos-aceite-modal > .vs-popup > .vs-popup--content > :nth-child(4) > .vx-col >
})
```

Fig. 4. Cypress code

V. Results

A. Execution Time

The script ran in a total time of 117,285 milliseconds (about 2 minutes and 4 seconds), with a significant portion of this duration likely attributed to the use of explicit waits.

To run the script without error, we had to insert instances of .wait() commands, such as wait(15000), wait(10000), wait(6000), and wait(20000). These commands are intended to pause the execution for a fixed duration, ensuring that web elements are loaded and ready for interaction. However, this approach also introduces a fixed delay, which occurs regardless of whether the elements are ready before the wait time elapses. The total wait time in the script sums up to 71,000 milliseconds, which is around 60.54% of the total

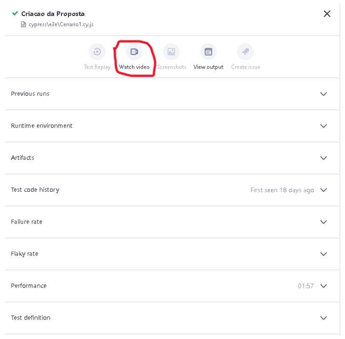


Fig. 5. Watch video feature

execution time.

Given the breakdown of the total execution time, where the test body consumes 103,454 milliseconds, it is evident that a substantial portion of the test duration is consumed by these wait periods. The use of explicit waits, especially with long durations, can inflate the execution time of the script significantly. This not only affects the efficiency of the testing process but also might not accurately reflect the application's performance under normal conditions.

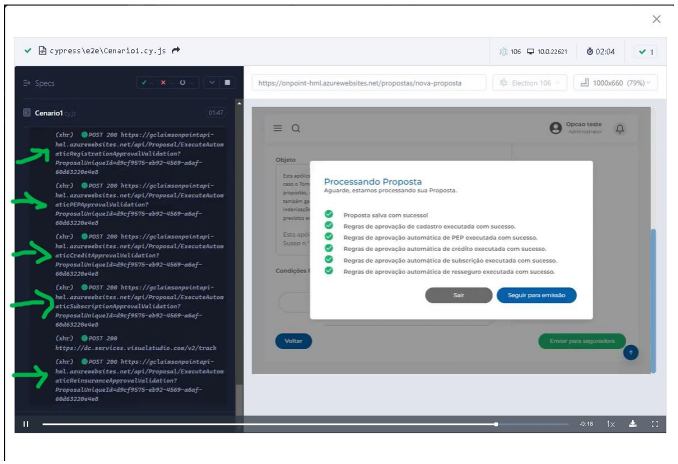
B. Run Tests

When running a test with Cypress, we can utilize a feature called video recording, which serves as a valuable tool for capturing the execution of tests. This offers significant benefits for both error analysis and team collaboration, as outlined in the Cypress documentation, figure 5.

Cypress's video recording feature, primarily supporting Chromium-based browsers like Chrome, Edge and more, is a versatile tool in software testing. Although disabled by default, it can be activated by setting video to true in the configuration. Once enabled, recorded videos are stored in the cypress/videos folder and are subject to processing and compression, especially when the –record flag is used, resulting in their upload to Cypress Cloud after each spec file execution. This encoding into a commonly digestible format, however, can impact the test duration; particularly for longer spec files, a noticeable delay may occur during cypress run as Cypress encodes and uploads the video. Despite this, the video recording feature stands out as a robust tool for documenting test executions, aiding in error analysis, and enhancing team collaboration by providing visual evidence for debugging and facilitating a deeper understanding of test scenarios, particularly in identifying causes of failures or unexpected behaviors, figure 6. n a recent instance, we attempted to rerun the same Cypress script on an alternate computer, but encountered an error midway through the process. This was due to an object not appearing in time for a required click action, figure 7.

VI. Conclusion

The utilization of Cypress software in our testing process has been a game-changer for our end-to-end testing methodologies. Through the automation of intricate workflows, as



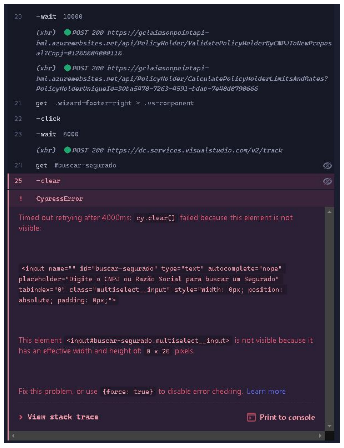
```
(xhr) ● POST 200 https://gclaimsonpointapi-
hml.azurewebsites.net/api/Proposal/ExecuteAutom
aticRegistrationApprovalValidation?
ProposalUniqueId=d9cf9575-eb92-4569-a6af-
60d63220e4e8
(xhr) ● POST 200 https://gclaimsonpointapi-
hml.azurewebsites.net/api/Proposal/ExecuteAutom
aticPEPApprovalValidation?
ProposalUniqueId=d9cf9575-eb92-4569-a6af-
60d63220e4e8
(xhr) ● POST 200 https://gclaimsonpointapi-
hml.azurewebsites.net/api/Proposal/ExecuteAutom
aticCreditApprovalValidation?
ProposalUniqueId=d9cf9575-eb92-4569-a6af-
60d63220e4e8
(xhr) ● POST 200 https://gclaimsonpointapi-
hml.azurewebsites.net/api/Proposal/ExecuteAutom
aticSubscriptionApprovalValidation?
ProposalUniqueId=d9cf9575-eb92-4569-a6af-
60d63220e4e8
(xhr) ● POST 200
https://dc.services.visualstudio.com/v2/track
(xhr) ● POST 200 https://gclaimsonpointapi-
hml.azurewebsites.net/api/Proposal/ExecuteAutom
aticReinsuranceApprovalValidation?
ProposalUniqueId=d9cf9575-eb92-4569-a6af-
60d63220e4e8
```

Processando Proposta

Aguarde, estamos processando sua Proposta.

- ✓ Proposta salva com sucesso!
- ✓ Regras de aprovação de cadastro executada com sucesso.
- ✓ Regras de aprovação automática de PEP executada com sucesso.
- ✓ Regras de aprovação automática de crédito executada com sucesso.
- ✓ Regras de aprovação automática de subscrição executada com sucesso.
- ✓ Regras de aprovação automática de resseguro executada com sucesso.

Fig. 6. running video test and passed step



```
20  -wait  10000
    (xhr) ● POST 200 https://gclaimsonpointapi-
    hml.azurewebsites.net/api/PolicyHolder/ValidatePolicyHolderByCNPJToNewPropos
    al?Cnpj=01265684000116
    (xhr) ● POST 200 https://gclaimsonpointapi-
    hml.azurewebsites.net/api/PolicyHolder/CalculatePolicyHolderLimitsAndRates?
    PolicyHolderUniqueId=30ba5478-7263-4591-bdab-7e48d8790666
21  get  .wizard-footer-right > .vs-component
22  -click
23  -wait  6000
    (xhr) ● POST 200 https://dc.services.visualstudio.com/v2/track
24  get  #buscar-segurado
25  -clear
 !  CypressError
Timed out retrying after 4000ms: cy.clear() failed because this element is not
visible:

<input name="" id="buscar-segurado" type="text" autocomplete="nope"
placeholder="Digite o CNPJ ou Razão Social para buscar um Segurado"
tabindex="0" class="multiselect__input" style="width: 0px; position:
absolute; padding: 0px;">

This element <input#buscar-segurado.multiselect__input> is not visible because it
has an effective width and height of: 0 x 20 pixels.

Fix this problem, or use {force: true} to disable error checking. Learn more

> View stack trace                                          Print to console
```

Fig. 7. Error when re-run the script

demonstrated in our 'Criacao da Proposta' test script, we've been able to enhance our testing efficiency significantly. Cypress empowers us to meticulously test essential features of our web application, covering aspects like user login, data input, and form submissions. This approach has streamlined our testing process, enabling us to cover more ground with greater accuracy.

A. Achived Objectives

A key strength of Cypress is its capability to replicate real-world user interactions within our application, guaranteeing the proper functionality of each component. By utilizing Cypress commands such as cy.get() and cy.type(), we have achieved a high level of precision in simulating user behavior, leading to more reliable and thorough testing outcomes. Furthermore, the ability to execute tests repeatedly without the need for manual input represents a substantial advancement in our testing efficiency. This consistency in test execution is vital for swiftly identifying and addressing potential issues, thereby allowing us to perform more extensive testing more frequently. This, in turn, has played a crucial role in elevating the overall quality of our software.

While the script does demonstrate a reliance on explicit wait times, a common practice in automation, it's important to recognize the balance between script simplicity and dynamic response handling. Future enhancements could focus on optimizing these wait times or employing more adaptive waiting strategies to further improve the efficiency and reliability of the tests.

B. Limitations of Cypress

The use of hardcoded wait times in the provided code introduces several interconnected issues. Firstly, these non-dynamic waits fail to adapt to the actual load times of web elements or server responses, leading to a rigidity that can cause unnecessary delays in test execution or even test failures if the wait time is not adequate. This inflexibility is closely linked to the issue of flakiness and reliability. Fixed wait times heighten the risk of producing flaky tests, which may pass or fail inconsistently, influenced by variable external factors such as network speed or server load.

Moreover, the practice of excessive waiting, as seen in the code, can significantly extend the total execution time of the test suite. This not only reduces overall efficiency but also affects the scalability of the testing process. As the application's complexity grows, managing these fixed wait times becomes increasingly cumbersome and error-prone, thus negatively impacting the scalability and maintainability of the test framework.

In essence, the approach of using static wait times in this testing scenario creates a chain of limitations, each exacerbating the other, from reduced adaptability and reliability to decreased efficiency and scalability.

References

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

[8] Jonas Verhoelen, "Green test pyramids with Cypress-UI testing of the future". https://www.codecentric.de/wissens-hub/blog/cypress-ui-end2end-testing. Accessed on: Nov. 8, 2023.

[9] Matthias Baldi, "Best practices for E2E Tests with Cypress and Angular". https://lambda-it.ch/blog/best-practices-with-cypress. Accessed on: Nov. 8, 2023.

[10] Cypress.io, "End-to-End Testing," in Cypress Documentation, 2023. [Online]. Available: https://docs.cypress.io/guides/overview/why-cypress. Accessed on: Nov. 9, 2023.

[11] LambdaTest, "Cypress JavaScript Tutorial: A Step-by-Step Handbook For Beginners", Available: https://www.lambdatest.com/blog/cypress-javascript-tutorial/. Accessed: Nov. 9, 2023.