Using Test Automation Frameworks to Ensure the Stability of Graphical User Interface Application

Anh Cao

Haaga-Helia University of Applied Sciences

Degree Program in Business Information Technology

Bachelor’s Thesis

2022

Abstract

Author(s)
Anh Cao

Degree
Bachelor of Business Administration

Thesis Title
Using Test Automation Frameworks to Ensure the Stability of Graphical User Interface Application

Number of pages and appendix pages
52 + 2

This thesis was written as a product thesis for Research and Development team at Nokia. The scope of this thesis was created a new End-to-end (E2E) testing system with a modern, fast, and reliable E2E testing framework called Cypress. This framework was considered to replace the old E2E testing framework called Protractor which had been planned to be deprecated since Angular 15. In addition, the goal of this project was implemented to enhance the CI pipeline's performance which was using Robot Framework and Python to automate the testing system during the daily run and nightly run.

The theory part of this thesis was split into 2 chapters in order to support the author to gain and widen his knowledge about background technologies. If the first chapter covered various technologies which were Angular, JavaScript, TypeScript, Node.js, Python, and Robot Framework, the second chapter did research about the background, the crucial role of Software Testing, its process, and its methods. This part also pointed out the reasons why Cypress was considered the new automation testing framework for the GUI application.

This thesis resulted in the positive and comprehensive enhancements in 2 tasks. At the decisive point of this thesis, Cypress had replaced Protractor totally and the E2E test with Cypress was executed in the CI pipeline. Secondly, the author did the enhancements for the report in CI pipeline to be more meaningful and have the possibility to execute in both parallel order and sequence order. In CI pipeline, Robot Framework was being used as an automation tool to run all the test and there were no problems in the CI pipeline when Robot Framework automated the testing job. Therefore, the author considered to do the enhancements on what was existing.

After this thesis, instead of spending hours on manual testing, GUI developers were able to scale the application without having any worries that the application would be crashed by their changes. The features that developers built were covered by automation tests which were more robust and accurate. According to the nightly and daily report, the test results were stable and efficient when it can provide details of failed tests including the screenshot of the failure cases.

Keywords
Software Development, Graphical User Interface Application, Test Automation Framework, Cypress, Robot Framework

Table of contents

Terms and Abbreviations 1
1 Introduction 2
1.1 About the company 2
1.2 Goal and Objectives 2
1.3 Out of Scope 3
2 Theoretical Background 4
2.1 Angular 4
2.2 JavaScript and TypeScript 4
2.2.1 JavaScript 5
2.2.2 TypeScript 5
2.3 Node.JS 6
2.4 Python 7
2.5 Robot Framework 8
3 Software Testing 10
3.1 Software Testing process 10
3.2 Software Testing methods 11
3.2.1 Unit Testing (Component Testing) 13
3.2.2 Integration Testing 13
3.2.3 End-to-end Testing 13
3.3 Cypress: End-to-end Testing Framework 14
3.3.1 Why Cypress? 15
3.3.2 Cypress Dashboard 16
4 Software Solution 19
4.1 Set up Cypress End-to-End testing framework 19
4.1.1 Set up Cypress for Angular project & remove Protractor E2E framework 20
4.1.2 Set up Cypress for Angular project only 20
4.1.3 Verify Cypress installation 20
4.1.4 Cypress Execution mode 21
4.1.5 Cypress Structure in Angular Project 22
4.1.6 Set up configuration for Cypress 22
4.2 Build up End-to-End testing in Cypress 26
4.2.1 Build the sharing state between tests 26
4.2.2 Assertion 27
4.2.3 Creating Cypress custom commands 28
4.2.4 Conditional 29

4.3 Enhance in CI environment 31
4.3.1 Pabot and Robot Framework 31
4.3.2 Install Pabot 33
4.4 Architecture for parallel execution and sequence execution 33
4.5 Build up custom library in Robot Framework 35
4.5.1 Set up the system for executions 36
4.5.2 Run tests 37
4.5.3 Combine results and print report 40
5 Conclusion 42
5.1 Achieved Objectives 42
5.1.1 Implement new E2E testing system 42
5.1.2 Enhance the CI pipeline performance 42
5.2 Results of the project 43
5.3 Limitations of the software solution 44
5.4 Lessons learned 45
5.5 What next? 45
References 47
Table of Figures 50
Appendices 53
Appendix 1. Complete setup configuration for Cypress E2E testing framework 53
Appendix 2: Complete execute_gui_test in custom library 54

Terms and Abbreviations

AST	Abstract Syntax Class
BDD	Behavior-Driven Development
CI	Continuous Integration
CD	Continuous Development
CDP	Chrome Devtools Protocol
CSS	Cascading Style Sheets
DOM	Document Object Model
E2E	End-to-end testing
GUI	Graphical User Interface
HTML	Hypertext Markup Language
JS	JavaScript
LTS	Long-Term Support
npm	Node Package Manager
npx	Node Package Execute
OS	Operating System
pip	Package installer for Python
QA	Quality Assurance
RF	Robot Framework
R&D	Research and Development
SPA	Single Page Application
TS	TypeScript
UI	User Interface
URL	Uniform Resource Locator
RAM	Random Access Memory
SDLC	Software Development Life Cycle

1 Introduction

Software Testing acts as a crucial role in Software Development and applications cannot be delivered to customers without testing, so does Graphical User Interface (GUI) Development. In GUI development, end-to-end (E2E) testing is a vital role for the whole application.

This project is considered as a solution for GUI development in Research and Development (R&D) team at Nokia Oyj which is expected to bring an easier way to build and maintain end-to-end tests, more flexibility to test with multiple browsers: Chrome, Microsoft Edge, Firefox, etc. Moreover, this project will investigate how to improve the current E2E testing in the CI pipeline which will reduce the running time and informative content display.

1.1 About the company

This thesis was completed during the time I was in commission with Nokia Oyj. Nokia is a Finnish Corporation that has around 130 countries of operation and offices around the world with more than 155 years in business. In order to deliver the company's strategic ambitions and become a market leader in specific areas, the company has four business departments:

- **Mobile Networks**
- **Network Infrastructure**
- **Cloud and Network Services**
- **Nokia Technologies**

1.2 Goal and Objectives

Building up and maintaining the E2E testing for GUI application is a vital thing that needs to be done. Unfortunately, the E2E testing framework which is currently used in GUI application will be deprecated by Angular at the end of 2022.

The goal of the thesis is to study and then include the knowledge to build up a E2E testing system by using new Testing Framework Cypress that has a visualization view to help the Graphical User Interface team to ensure the Quality Assurance (QA) of the application. The desired outcome is not only to help developers in GUI team make it easier to follow and maintain the tests, but also enhance the testing performance in the Continuous Integration / Continuous Delivery (CI/CD) pipeline.

To reach the goal of the project, I have to complete these primary objectives:

1. **Set up Cypress test framework for Angular project**
2. **Implement testcases to Cypress and increase the testing coverage**
3. **Figure out how to switch the tests between development and product phase**
4. **Make the new test framework to run in CI environment**
5. **Improve the test performance in CI environment**

I have a lot of opportunities to learn, sharpen my knowledge in automation testing skills, expand my knowledge about programming languages and modern technologies and frameworks during this project. Moreover, I believe I had a great chance to understand more about the GUI application when I implemented the automation test to cover mostly existing functionalities.

1.3 Out of Scope

The thesis will describe how I approached, and how I built the solution for this project. This will not go into the code base of the product. I believe that in the same situation if people do not know the solutions at the beginning, they can discover and improve their logic in programming massive further by reading others' experiences and then build their own solutions based on what they learned. This is a better way compared to only focusing on reading others' ready-made solutions.

The area of this project is about to create a new End-to-End testing system for Graphical User Interface and further improve the behavior of the new test system in the CI environment. I will not mention anything about the old E2E framework, which is no longer used, even the differences between two testing frameworks. Any other changes or improvements in GUI development or in Backend development are not included in this project.

There will be also no discussions about Automation testing and Manual testing which is better, or which one should be used. In this project, I respect both Manual testing and Automation testing, each of them has their own strengths and weakness and it depends on the way we consider and use them.

2 Theoretical Background

This section comprises the technical backgrounds which are used during this project. It covers what technologies are used and their core characteristics in the project. I will not go into details about what these technologies do, and how they can do because there is a lot of information and forum discussions available on the Internet.

2.1 Angular

Nowadays, web applications have richer User Interface (UI), User Experience (UX) and more complicated functionalities than in the old days and it will be further growing in the future (Datey Meera 2021). Therefore, Single Page Application (SPA) came up and has been used more in recent years. It facilitates developers to build an application which can load, fetch data more dynamic and faster, allow to build no internet connection mode, and more powerful technologies (Smith Steve 2022, 8).

One of the three most popular common SPA frameworks besides React and Vue is Angular, which was released by Google in 2009. It has been developed and maintained until now and the latest enhanced version of Angular is version 14. Angular is not only a Long-Term Support (LTS) product from Google but it is also a framework for large-scale applications and suitable for all types of business due to its ecosystem, trustworthiness, and opinionate which has been proven over the years (Gawron Karolina & Fluin Stephen 2018).

Google LLC (2022) said that the essential cores of the Angular framework, which make the code more consistent, and usability and enhance the application performance are building blocks called Components, and its own Templates, Directives, and Dependency Injection. Developers can build efficient and scalable applications if they have a fine grasp of the Angular essential cores.

2.2 JavaScript and TypeScript

The software solution for this product utilizes JavaScript and TypeScript, so in the following subchapters, we will briefly talk about them. Moreover, we will move deeper into the advantages of TypeScript and figure out the reason TypeScript would be the ideal choice for building not only Angular applications but also Cypress E2E testing system. An application written in TypeScript brings a vast number of benefits to developers during their work when they know exactly the data types of everything they are dealing with (Stempniak Adam & Świstak Tomasz 2022).

2.2.1 JavaScript

JavaScript is flexible and has great various features for professional developers to deal with every day. According to Stack Overflow's survey (2022) which had 71 547 responses, JavaScript was marked as one of the most popular scripting languages in a 10-year row which consisted 65.36 % of respondents.

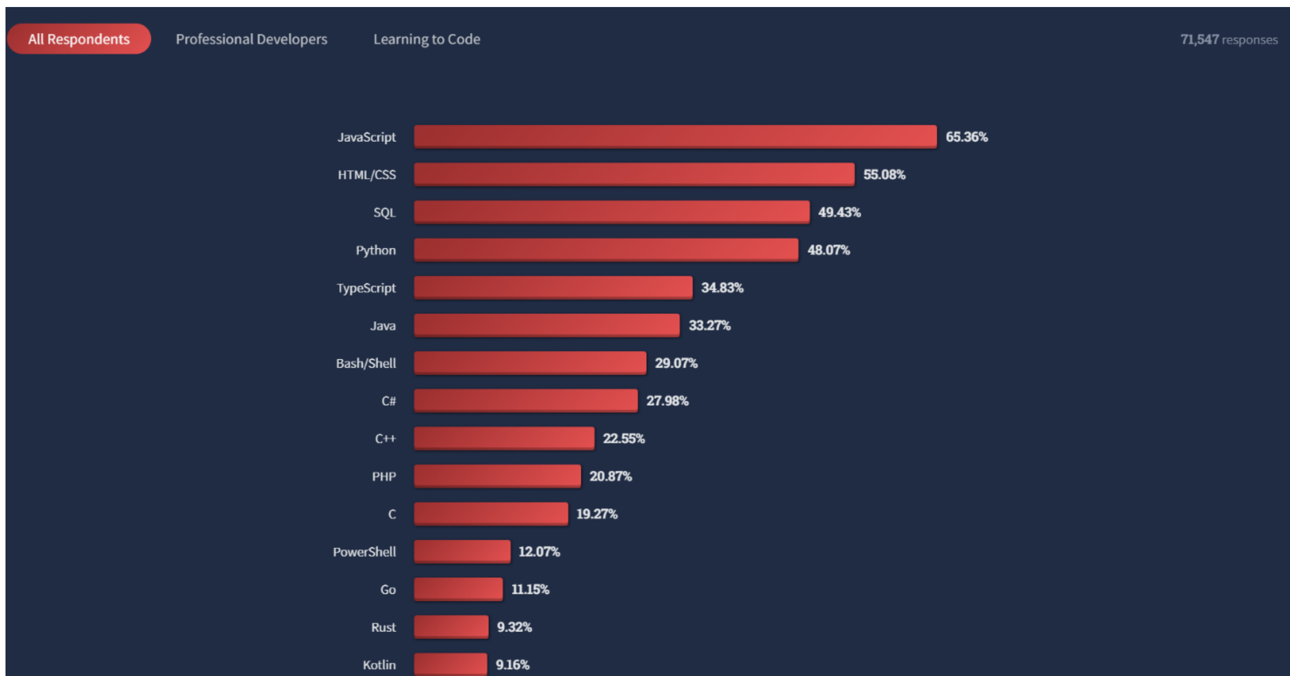


Figure 1. The most commonly programming languages in 2022 (Stack Overflow's survey 2022)

If we consider a website as a human body and start to "dissect" its anatomy of it, we can see that HTML is seen as the skeleton, and structure of the website, and CSS is skin which means it improves the outlook of the website. Finally, JavaScript is considered the brain of web applications (Ubah Kingsley 2021). When JavaScript was created by Eich Brendan in 1995, it makes web applications more dynamic, with behaviors, and interactions between a webpage and users (Nology Team 2020).

2.2.2 TypeScript

JavaScript has a lot of communities that love to use it. However, it is a weak-typed programming language that could make some experienced developers from other strongly typing languages

despise JavaScript when they have to use it. Fortunately, Microsoft announced TypeScript in 2012 and it became a great option for professional developers.

TypeScript is known as a superset of JavaScript that not only offers developers full JavaScript features but also an additional layer: type checker (Microsoft 2022). This is the enhanced benefit of TypeScript when we compare it to JavaScript, TypeScript itself can detect unexpected behavior in developers' code. Therefore, it will reduce the percentage of bugs, and unknown typing of variables then reduce wasting time debugging the code. In Figure 2 and 3 below show the behavior between TypeScript and JavaScript when a developer tries to do something that is not correct (Goldberg Josh 2022, Chapter 1).

```
const firstName = "Georgia";
const nameLength = firstName.length(); // JavaScript does not complain anything
```

Figure 2. No complaints from JavaScript (adapted from Goldberg Josh 2022, Chapter 1)

```
(property) String.length: number
Returns the length of a String object.
This expression is not callable.
  Type 'Number' has no call signatures. ts(2349)
View Problem    No quick fixes available

const firstName = "Georgia";
const nameLength = firstName.length();
```

Figure 3. TypeScript reporting an error (adapted from Goldberg Josh 2022, Chapter 1)

2.3 Node.JS

Before Node.JS was released, it is impossible for developers to use JavaScript as the back-end programming language because JavaScript is only able to run in a browser (Wassell Shaun 2021). Thanks to Node.JS, nowadays, developers with JavaScript can easily leverage their skill on developing back-end side. They are not strictly on front-end development anymore.

Node.js is fast, lightweight, extendible and can do the same jobs efficiently as other server-side languages due to its non-blocking, event-driven I/O architecture. Moreover, Node.JS has the Node Package Manager (npm) which is known as the largest package register in the world (Buckler Craig 2022). NPM has more than one million modules that allow developers to use, share and contribute to open-source projects without requiring any costs from developers.

2.4 Python

Python was created in 1990 by Guido van Rossum and over 30 years, Python has become one of the best choices for building up applications, testing systems, operating systems and so on. According to the TIOBE Programming Community index (2022), which is specialized in the assessment of the quality of software, and programming languages, Python recently became the leader in the top common programming languages with ratings of 15.74 % that overcame the C programming language with 13.96% since September 2022.

Sep 2022	Sep 2021	Change	Programming Language	Ratings	Change
1	2	^	Python	15.74%	+4.07%
2	1	v	C	13.96%	+2.13%
3	3		Java	11.72%	+0.60%
4	4		C++	9.76%	+2.63%
5	5		C#	4.88%	-0.89%
6	6		Visual Basic	4.39%	-0.22%
7	7		JavaScript	2.82%	+0.27%
8	8		Assembly language	2.49%	+0.07%

Figure 4. The Programming Languages Ranking for September 2022 (TIOBE Index 2022)

Python offers high productivity for all development phases of the Software Development Life Cycle (SDLC) such as analysis, designing, coding, testing, maintaining, and debugging, however, Python and its syntax is not difficult or complex to learn, get familiar with, and solve the problems.

Python is not only used in building up applications, and operating systems but it is also used for analyzing and working around data, machine learning, and more IT fields. Due to its consistency, own rich standard libraries, and a vast number of third-package libraries, Python is simpler, faster to process, and more productive when it can collaborate with other components, languages than the classic high-level languages such as C, C++, or Java.

If TypeScript or JavaScript has NPM package for installing and using their own third-party libraries, so does Python. Python has its own package manager called Package Installer for Python (pip) which provides massive libraries that developers can easily install and use for their own purposes.

2.5 Robot Framework

Robot Framework is an open-source automation framework which is supported and maintained by developers from big companies around the world such as Cisco, Finnair, Nordea, Nokia etc called Robot Framework Foundation (Robot Framework. 2022). This automation tool can be used for acceptance testing, acceptance test-driven development (ATDD), and robotic process automation (RPA). Robot Framework was developed based on Python so that it has a simple syntax and has the possibility to extend with generic and custom libraries. With flexibility and reliability, Robot Framework is a great automation framework for developing tests in the CI pipeline which we can create our own custom libraries and automate the way we perform our testing system. Because Robot Framework is built in Python, it is not only convenient for QA developers who mainly work on testing, but it is also easy and useful for other software developers when they already had knowledge about Python during the backend works.

Instead of using some existing libraries which is not compatible with my solution, I created and imported a custom library "**gui_e2e_handling**" which has custom functions to pull the latest code, to execute all E2E tests into a sample Robot Framework script by calling the keyword `Library` (Figure 5).

```
*** Settings ***
Documentation  Robot Framework Sample Test
Library    gui_e2e_handling.guiE2EHandling    WITH NAME    guiE2EHandling
Force Tags      gui_e2e
Suite Setup     clean_previous_report_data


*** Test Cases ***
GUI_E2E_set_up_the_latest_version
  [tags]    GUI_PARALLEL
  [Documentation]   ..........................................................
  ...               - Get the latest version of AGUI
  initialize

GUI_E2E_execute_all_tests
  [tags]    GUI_PARALLEL
  [Documentation]   ..........................................................
  ...               - This test will execute all E2E tests in GUI project
  execute_all_test


*** Keywords ***
clean_previous_report_data
  clear_report_content
```

Figure 5. Sample Robot Framework script with importing custom library

3 Software Testing

Software testing no longer belongs only to QA teams or software testers, even if the system is big or small. This becomes a compulsory responsibility for all software developers to ensure that software bugs are not allowed to exist. Nowadays, even though the main job of developers is building up software systems, they are also responsible for the quality of software what they are building and aim to deliver it to customers. As software developers, we need to remember three keys which are to ensure the quality of what we produce, testing is a tool that supports us to ensure the quality, and the testing will be effective and systematic when we know the right collection of techniques in testing (Aniche Mauricio 2022, Chapter 1).

If an application does not have any software testing system to cover its functionalities, behaviors, that means the business who owns that application can face several damages not only related to themselves, but also to customers, or even to society like dominoes when collapse. Software Testing is not an easy job and takes a lot of time to build up and maintain because it requires people to really need to spend time learning, and familiarizing themselves with the concepts, fundamentals, and architectures of the testing framework. Despite testing playing a vital role and cannot deliver a product without it, it is quite time-consuming, and most developers do not enjoy writing tests. We will dive into software testing in the following subchapters which are the keys for these questions:

- **What does software process?**
- **How many types of automation testing?**
- **Why did we choose Cypress? What are the benefits that Cypress could bring?**

3.1 Software Testing process

You might think that software testing is easy and could be redundant stuff which is an obstacle for your developer career. Unfortunately, this is not right that software testing requires more work than just creating a test. An example about creating an automation test, you have to first know what the function you test can do, then do the test manually to ensure it works, and then get a plan for how you will integrate it into the automation test.

Software testing has initial thought that it sounds like a sequential/ waterfall process, however, it is more iterative. This can be explained in a way that in the morning, a developer finishes a test for a feature that he implemented before, and he believes that this test covered rigorously all functionalities. After a few hours, he realized that the decision he made in the morning was not covered all

and it was still missing some cases that could happen or have bugs. Then he has to go back, re-write, and add more conditions to enhance the test. Quite commonly, after a while, since the last time writing the test case, the developer looks back and sees that the test is not ideal or not easy for others to maintain this case then he decides to redesign or sometimes refactor the code again.

Figure 6 below shows the relationship between the development and testing phase which starts from the collecting requirements and goes through to the production (Hambling Brian & Samaroo Angelina 2009).

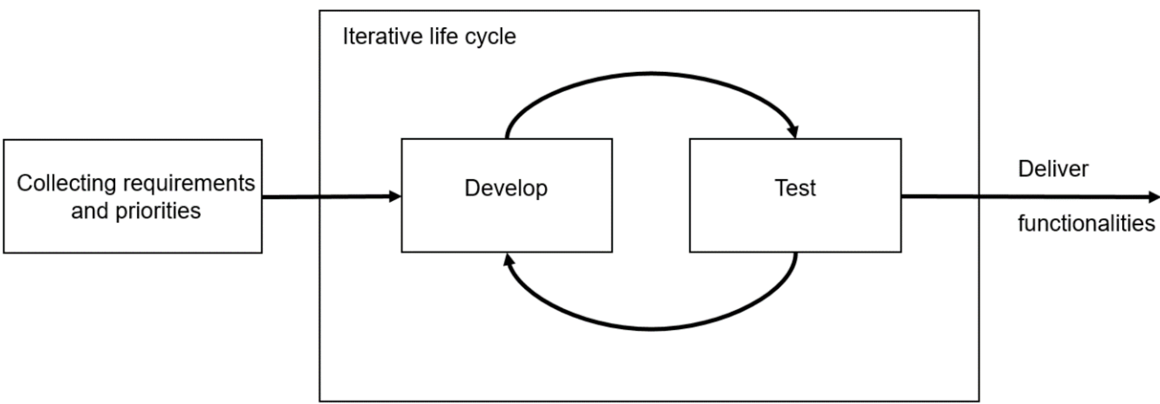


Figure 6 Testing Activities in the life cycle (adapted from Hambling Brian & Samaroo Angelina 2009, Chapter 1)

In daily work as a software developer, you absolutely need to spend time testing applications and quite often you will meet some difficulties whether you test manually. Moreover, sometimes you could also miss some situations that could happen, but automation test is not. If the automation test was created correctly and not flaky, automated test is wonderful at pointing out the problems in codebase. The flaky situation means when the whole test case or just one test will fail if we execute several times without changing the code. For instance, if someone implements some changes that breaks something, automation test can point out the problem and save a lot of time for developer to debug manually.

3.2 Software Testing methods

In software testing, there are two kinds of test methods: manual testing and automation testing. Both of these methods are needed in the software testing process, and I believe that the automation test cannot cover all functionalities by itself, nor does the manual test. However, as I

mentioned earlier (Chapter 1.3: Out of Scope), this product is only focused on automation tests, so the theoretical background will only dive in and research automation testing.

Automation testing in software is a massive workload and unfortunately, there are no shortcuts or shortage of tools to test the software application included backend and frontend. In the initial steps of automated test, developer or a group of developers will plan up cautiously a plan for building a test system that could covers all scenarios that could happen, and their application need to pass all tests. Therefore, to ensure the high-quality software and reduce the time required for testing after times developer introduce some new changes to the code, Mike Cohn (2009) divided the software testing into 3 different layers in a testing pyramid (Figure 7).

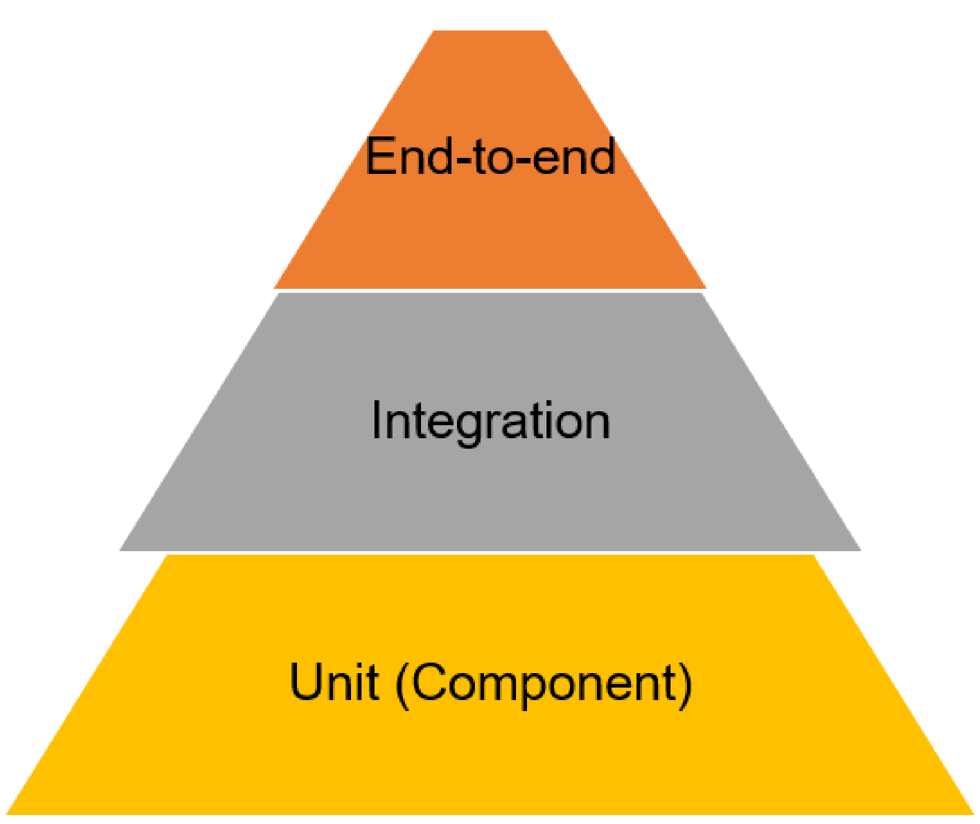


Figure 7. The Pyramid of testing (adapted from Mike Cohn 2009)

3.2.1 Unit Testing (Component Testing)

According to the pyramid graph above (Figure 7), Unit test, sometimes this test method can be called Component test is the lowest stage of the testing pyramid which means it ensures the unit, and component of the applications. The tests in this stage are built up to test the individual components or their functionalities to validate that it works as same as the desired goals and avoid isolated conditions. It is important to have a vast number of scenarios in unit tests such as positive conditions, handling errors, etc. Because of the largest stage, the unit test suite must be written to run as quickly as possible or right away after the developer finishes his recent changes. The more features that the application has, the more unit tests are required to have.

3.2.2 Integration Testing

If the Unit test is used for ensuring the quality of the function, or component, the Integration test will combine and test various functions or components logically together to point out the possible flaws. So that the Integration test can be seen as a complicated version of Unit testing. During the Integration testing, even if it does not require a complete application, it is still able to test how different components work, interact with each other to reach the desired functionalities (Loubser Nico 2021, Chapter 8).

3.2.3 End-to-end Testing

End to End testing is a methodology that is used in software development life cycle to test all the functionalities and also test the performance of an application when it is in live settings. The E2E testing's goal is to simulate situations that could happen in real life when a user starts to use an application from the beginning until the end. Therefore, the E2E testing not only helps the GUI team to ensure the QA of the code, but also helps new coming developers know about the application. The developer only needs to execute the automation test command and it will handle the rest.

E2E testing also ensures that the application would work and behave correctly as expected. E2E testing does exactly the same things that developer does while testing the application: looking, clicking, typing, and moving around the application. Therefore, E2E testing usually takes the longest time to run.

According to Keen Yee Liau, a developer from Angular team, to understand about the current E2E testing trend, Angular team conducted a survey in January 2021 which was successful when they

received nearly 1000 responses and a lot of feedback from communities. Cypress and Protractor shared the first and the second position who comprised the most, at 64 % and 19.5 % (Figure 8). As Angular team said after this survey, they realized that even though there were a lot of E2E testing tools, there was no E2E testing tool that could fit all Angular projects.

What e2e testing tools do you use?

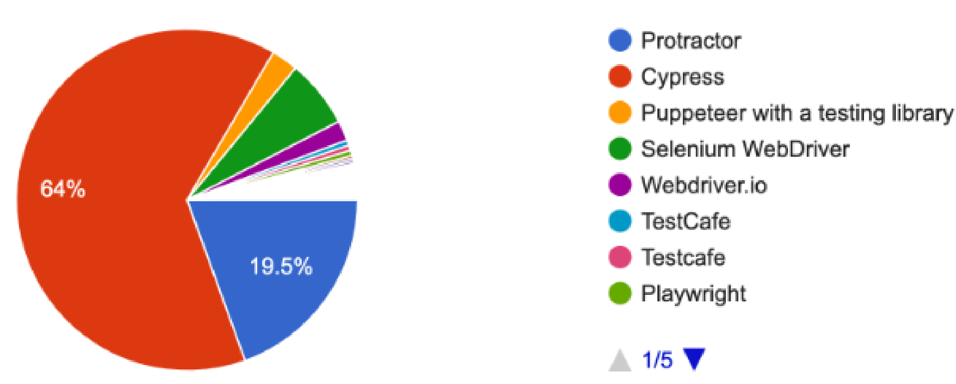


Figure 8. E2E testing tools trend (Angular's survey 2021)

Each of these E2E framework has their own strength and weakness, and depend on my commission's needs, I agreed that Cypress will become the upcoming E2E testing to replace Protractor which was planned to end development at the end of 2022 and deprecate by Angular in version 15 (Angular 2021). The following subchapter will describe Cypress and why it is worth becoming the next E2E framework for GUI application.

3.3 Cypress: End-to-end Testing Framework

We discussed about End-to-end testing and why it acts a crucial role in Software Development. There are a lot of automation tools for doing E2E job and Cypress is one of the most popular tools due to its speed, easy, reliable and the execution results are exceedingly stable for modern applications.

3.3.1 Why Cypress?

Cypress is the JavaScript end-to-end testing framework which is built, engineered, and optimized for testing modern web applications on browser purposes. Initially, the first reason why Cypress is chosen as the next E2E test framework because it is built in JavaScript, TypeScript and only supports the use of JavaScript and TypeScript to develop tests. So that it is extremely compatible with frontend Angular application. In addition, Cypress has adopted Behavior-Driven Development (bdd) syntax from Mocha which is a feature-rich testing framework in JavaScript running on Node.js and browser.

Thirdly, developers with JS or TS background can easily onboard, write the tests, debug, and also understand the errors in Cypress because they are similar to those in JS or TS application (Waweru Mwaura 2021, Chapter 2). With Cypress, developers can write, debug, and run tests directly on the browser. It is designed and built by talented developers so that it works as a standalone application and can be used in multi operating system: Windows, MacOS and Linux either using as a desktop application or command-line to test in browsers such as: Chrome, Firefox, Edge, Electron, and other browsers based on Chromium. (Cypress 2022a).

Nevertheless, Cypress's Architecture is a factor that makes Cypress outstand with other automation testing tools and become the desired and potential choice for my project. Unlike other testing tools which utilize WebDriver and interact with Document Object Model (DOM), Cypress relies on Chrome DevTools Protocol (CDP) and Cypress uses CDP only to manage the tests for its application. This allows Cypress's test cases to run directly in the browser without the need to ask for network requests which means it will directly access the DOM without having to go via CDP or WebDriver anymore. In other words, Cypress operates directly to the browser which can be seen as the browser executing the code itself (Rahul Shetty 2019, Chapter 1). This makes the tests faster, more stable, and more reliable during the execution. Figure 9 describes the work architecture of Cypress and gives an overview of how Cypress handles the test.

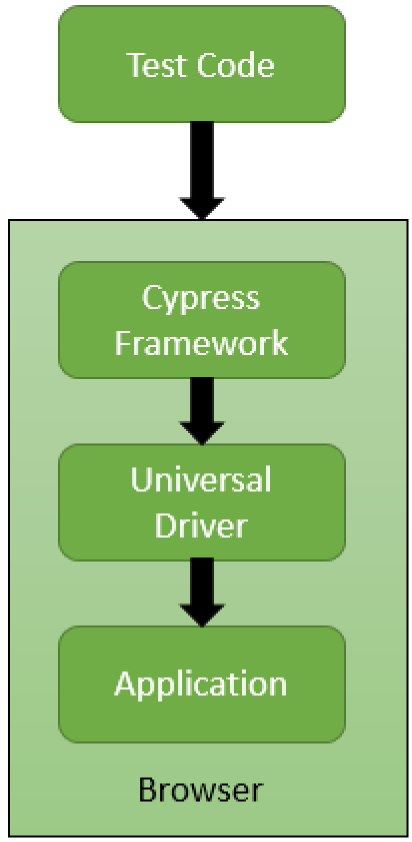


Figure 9. Cypress test execution architecture (adapted from Mwaura Waweru, Chapter 2)

Finally, Cypress has a built-in algorithm that can detect and know when need to wait, retry for the browser, and page events. For example, Cypress can wait for the network request to respond without needing to define the timeout or time to wait like automation tools such as Selenium (Mwaura Waweru 2021, Chapter 2).

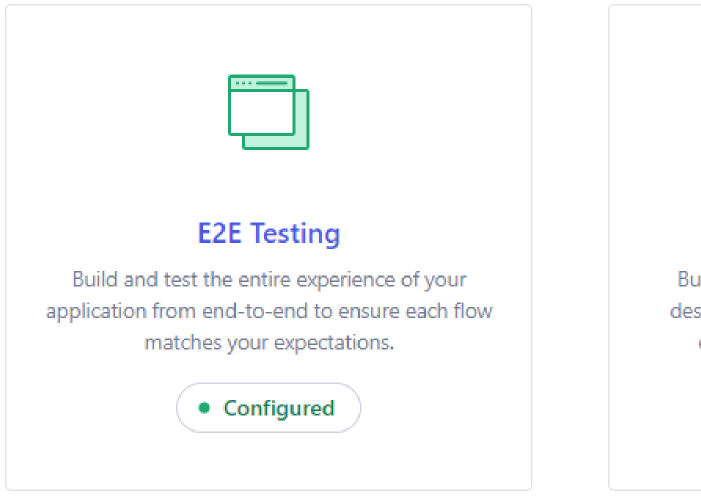
3.3.2 Cypress Dashboard

One of the reasons Cypress should be your E2E testing system is its dashboard which makes Cypress not only an e2e testing framework but also an application that users and developers can easily interact with and follow the data, and performance of the test. To open Cypress, we can execute the command `cypress open` (Figure 10).

```
$ npx cypress open
```

Figure 10. Open Cypress launchpad

Immediately, Cypress Dashboard (or Launchpad) will appear (see Figure 11 below). In the dashboard, users can easily choose which type of testing method they want to set up and use such as End-to-End testing or Component testing. At the time this thesis is implemented, I use the Cypress version 10.8.0.



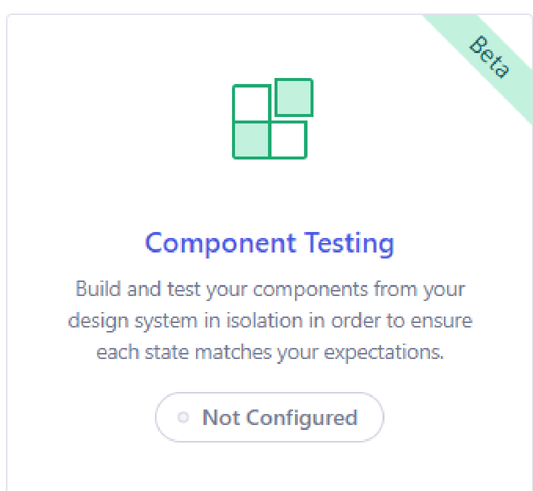


Figure 11. Cypress Dashboard

Now, we will dive into the E2E testing via Cypress Dashboard by clicking on the E2E Testing. Cypress will take you to the Launch Browser step (Figure 12). In this step, you are able to choose which browser you want to test your application and the number of available browsers depending on how many browsers your machine has and whether Cypress supports or not. Finally, it is a suitable time to move on to the implementation section of this thesis in the coming chapter.

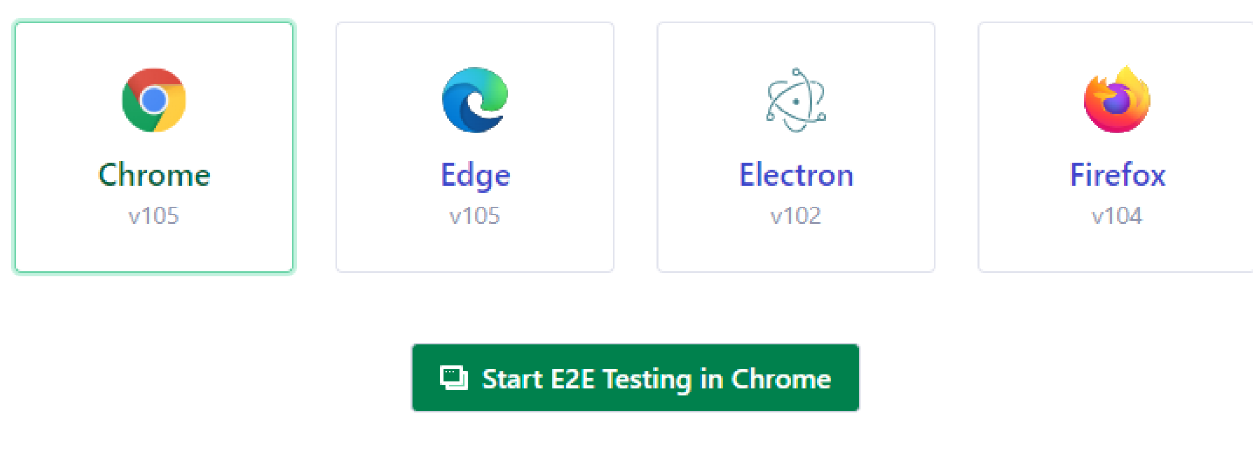


Figure 12. The available browsers on machine that Cypress supports

4 Software Solution

The project includes 2 main tasks which is create a new E2E testing system for GUI application then do enhancements to the CI pipeline performance. These two main tasks are split into 5 stages to help me easy to follow and build the software solution. This project is considered to reach the goal when all 5 stages below are finished:

- **Set up Cypress for Angular application**
- **Build up the test cases to cover all functionalities**
- **Configure the CI environment**
- **Design the testing architect for parallel execution and sequence execution**
- **Build custom library for Robot Framework to execute in parallel and sequence order**

The first stage starts with the installation and setup of the configuration for Cypress in the Angular project, which allows the test to have some specific modifications and can be able to run consistently in multiple operating systems such as Windows and Linux machines without receiving any warning. The GUI team use Windows 10 machines to develop features and run automation test by themselves. When everything is ready, the code will be merged to the master branch and in the CI pipeline, the Operating System of the environment is Linux CentOS. So, this project is implemented and tested only with Windows 10 and Linux CentOS.

After the setup stage, the next following stage will show the building up of the Cypress test cases, configure the CI environment and design the architecture for multiple execution mode: parallel and sequence order. Finally, the software solution will be completed after I do enhancements such as the report being more informative, and parallel execution to reduce the execution time by building up a custom library for Robot Framework.

4.1 Set up Cypress End-to-End testing framework

As I mentioned in section 1, the solution for this project is to create a new e2e testing system to replace the current one which will be deprecated by Angular at the end of 2022. There are 2 ways to install Cypress:

- **Replace Protractor, the current E2E testing and install Cypress**
- **Install Cypress on an Angular project with no prior E2E testing system**

4.1.1 Set up Cypress for Angular project & remove Protractor E2E framework

Based on Cypress's recommendations for migrating from Protractor to Cypress, installing the NPM package `cypress/schematic` will install Cypress and add the scripts for running Cypress in headful mode and headless mode (Cypress 2022c). Nevertheless, this package also removes the current Protractor package, update the `ng e2e` command to use Cypress for the E2E testing and keep only the Protractor's test cases for migrating later.

```
$ ng add @cypress/schematic
```

Figure 13. Installing Schematic package from Cypress

4.1.2 Set up Cypress for Angular project only

If the Angular application does not have any E2E testing system installed, we can install Cypress via `npm`. Installing with `npm` is recommended by Cypress because Cypress is versioned similarly to other dependencies so that the developer can easily understand how to download and upgrade the version of Cypress later and also simplifies to run Cypress in the Continuous Integration (Cypress 2022d).

```
$ npm install --save-dev cypress
```

Figure 14. Install Cypress via npm

4.1.3 Verify Cypress installation

After completing the installation of Cypress to our application, we need to ensure that Cypress is installed successfully. The reason we need to check whether Cypress is installed or not is sometimes, due to some conflicts with company proxy, or configurations on our machine, the NPM package cannot be installed. To verify, we only need to run `cypress verify` command and Cypress will verify itself and also the current system for us.

```
$ npx cypress verify

[STARTED] Task without title.
[TITLE]  Verified Cypress!       C:\Users\anhcao\AppData\Local\Cypress\Cache\10.8.0\Cypress
[SUCCESS]  Verified Cypress!       C:\Users\anhcao\AppData\Local\Cypress\Cache\10.8.0\Cypress
```

Figure 15. Cypress verification process

4.1.4 Cypress Execution mode

After the installation and verification process that ensure Cypress is installed successfully in Angular application, we can start to run Cypress in `open` mode or `run` mode. The `open` mode in Cypress has some other names that can be called Headful mode or Headed mode. In this execution mode, Cypress will open its launch pad and users can select which browser they want to test. This mode helps us easier, faster to figure out the reason why the test failed and can fix faster compared to the `run` mode. There are some available commands for executing this mode and each command has its own purpose.

At the moment, it might take some time to execute all the tests in `open` mode because it does not have any options for running all tests. The more test cases we have, the more time we need to test each single test case. To open Cypress and test the application without having any need code changes, we can use the command:

```
$ npx cypress open
```

Figure 16. Open Cypress Dashboard only

If you want to develop some changes for a feature in the application and after that, you also want to build a test case to cover that feature in the development phase and ensure there are no breaks before pushing to the branch Master, you can use the following command in Figure 17. Initially, this command will build the Angular development server as same as `ng serve` and then open the Cypress Runner.

```
$ ng e2e
```

Figure 17. Build Angular server and open Cypress Runner

Looking on the other hand, the `run` mode is known as a command that allows Cypress to run in the headless mode which means running the test without any UI browser, Cypress Dashboard, and printing the output to the terminal. This mode is faster because it takes fewer memory footprints on the browser, and at default, it will run all the tests which we cannot do the same thing in the headful mode.

```
$ npx cypress run
```

Figure 18. Running Cypress in headless mode

After the execution, Cypress will print out the report of the whole test which was executed. The report is informative that includes the number of tests, the passing, failing, pending, and skipped

tests (Figure 19). So that the developers can easily track and do the enhancements. Moreover, in this headless mode, Cypress provides to this mode a lot of powerful features which as options for recording and taking screenshots when the tests are not passing. To run a specific test, we need to add `--spec` to command line option with the path to the specific test file:

```
npx cypress run --spec "cypress/e2e/01_loginLogout.e2e.cy.ts"
```

```
(Run Finished)

     Spec                                          Tests  Passing  Failing  Pending  Skipped
  ┌──────────────────────────────────────────────────────────────────────────────────────┐
  │ ×  01_loginLogout.e2e.cy.ts            00:03       2        -        1        -        1 │
  └──────────────────────────────────────────────────────────────────────────────────────┘
    ×  1 of 1 failed (100%)                00:03       2        -        1        -        1
```

Figure 19. The sample output of the specific Cypress test in headless mode

4.1.5 Cypress Structure in Angular Project

It is good to understand Cypress and the purpose of each folder inside the Cypress root. Since Cypress version 10, after installing successfully, Cypress will automatically create one file called *cypress.config.ts* located in the root directory and create a folder called **cypress** which contains subfolders: **E2E**, **Fixtures**, **Plugins** and **Support** like the hierarchy below (Figure 20).

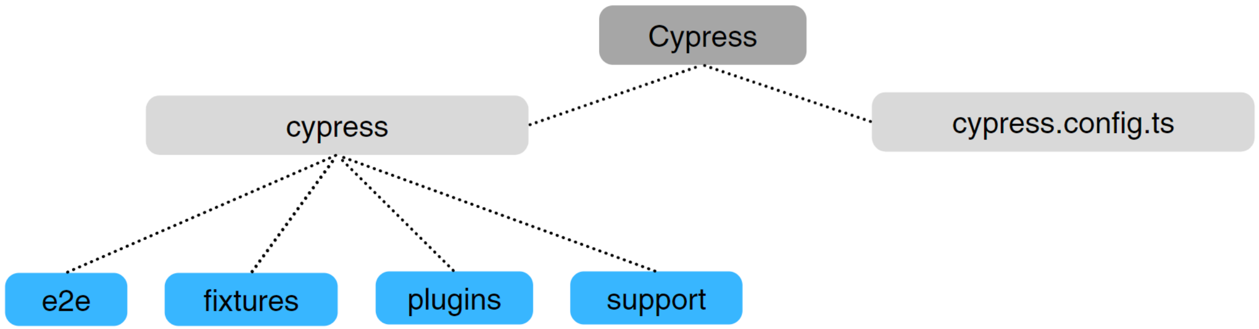


Figure 20. Cypress Hierarchy diagram

4.1.6 Set up configuration for Cypress

Initially, we need to define config variables in the **cypress.config.ts** so that we can set where videos, captures will be stored in Cypress after each execution (Figure 21).

```
import { defineConfig } from 'cypress';

export default defineConfig({
  video: false,
  screenshotOnRunFailure: true,
  screenshotsFolder: 'cypress/screenshots',
  videosFolder: 'cypress/videos',
  fixturesFolder: 'cypress/fixtures',
  downloadsFolder: 'cypress/downloads',
```

Figure 21. Define the location of subfolders

At default, Cypress has a scroll algorithm that always tries to scroll on top and this sometimes makes users confused because they cannot see the correct position of the cursor. Therefore, I assigned `scrollBehavior` to the value: "nearest" from the default value: “top.” This value will allow Cypress to scroll to the nearest position which we are checking (Figure 22).

```
scrollBehavior: 'nearest',
```

Figure 22. Define the scroll behavior

In test automation and especially in E2E tests, sometimes there are still some situations that hard to detect what is the root problem that makes the test fail and become flaky. These behaviors often occur due to unpredictable conditions such as random network errors, the servers taking more time than the timeout, API calls, etc (Cypress 2022f). Cypress has option `retries` that allow itself to retry the failed tests in both `open` mode and `run` mode. This will reduce the percentage of flakiness and save time for the development team so that they can focus on developing and enhancing some matters. In Figure 23, I set the `retries` variable to 2 which will make Cypress understand if the first execution failed, then it will try to execute 2 times more.

```
retries: 2,
```

Figure 23. Increase the number of retries when the test failed in both headed and headless mode

Defining `baseUrl` will help Cypress to know which URL you want Cypress to navigate and do the tests (Figure 24). I set this config variable to `null` because I want to make Cypress more dynamic when it is able to test with 2 URLs, one is `localhost,` and the other is product URL.

```
baseUrl: null,
```

Figure 24. Set baseUrl to null make testing system more flexible in switching between Develop and Product phase

Cypress is not only focused on performance, and enhancing the workflow for developers, but also focuses on security. Based on the way Cypress is architected, Cypress will error you when we navigate to some internal sites, super domains, or insecure sites. Even though this behavior is a great tool when it can highlight serious security, we still need to turn it off when we develop the tests for internal products.

```
chromeWebSecurity: false,
```

Figure 25. Allow Cypress to work with super domains, cross-origin

Cypress has an architecture that whenever one test in a suite is finished, it will clear all cache and cookies. This will prevent a state which will be shared between tests (Cypress 2022g). However, the drawback of this architecture is it will increase the time of each test. For example, you are developing a todo app and instead you login and do all functions such as: create a list, modify that list, and finally delete list in a row. The following code will illustrate what I just described:

```
describe('test todo app functionalities', () => {
  before(() => {
    // login to the todo app
  });

  it('create a todo list', () => {
    // create a list
  });

  it('modify a todo list', () => {
    // modify a list
  });

  it('delete a todo list', () => {
    // delete a list
  });
});
```

Figure 26. Sample code about testcases in todo app

However, in Cypress, due to its architecture, first Cypress will login and the ‘create a todo list’ then clear all cookies and the rest of the test suite will fail due to unauthorized. To solve the problem, we can create `beforeEach` instead of `before`. Unfortunately, this will make the test longer when it first login → create a list → clear all cookies, then login → modify a list → clear all cookies, finally login again → delete a list → clear all cookies. To solve this, the initial step is to enable `experimentalSessionAndOrigin` to “true” inside *e2e* object to tell Cypress allowing us to work

around with `Session` and `Origin` (Figure 27). The solution and how we approach with `Ses-sion` and `Origin` in Cypress will be discussed more in the following chapter (Chapter 4.2).

```
experimentalSessionAndOrigin: true,
```

Figure 27. Keep the memory after each test in a suite

Finally, to ensure that the Developer tools are always open when Cypress opens the browser, I will call `setupNodeEvents` function which will allow us to tap into, modify or extend the internal behavior between Cypress and browsers inside *e2e* object. In addition, this function is vastly powerful when it can help us to solve the conflict between Chrome, Linux, and Cypress.

```
setupNodeEvents(on, config) {
  Complexity is 8 It's time to do something...
  on('before:browser:launch', (browser, launchOptions) => {
    // `args` is an array of all the arguments that will
    // be passed to browsers when it launches

    if (browser.family === 'chromium' && browser.name !== 'electron') {
      // auto open devtools
      launchOptions.args.push('--auto-open-devtools-for-tabs');
    }

    if (browser.name === 'chrome' && browser.isHeadless) {
      launchOptions.args.push('--disable-gpu');
      launchOptions.args.push('--disable-software-rasterizer');
      launchOptions.args.push('--no-sandbox');
    }

    if (browser.family === 'firefox') {
      // auto open devtools
      launchOptions.args.push('-devtools');
    }

    if (browser.name === 'electron') {
      // auto open devtools
      launchOptions.preferences.devTools = true;
    }

    // whatever you return here becomes the launchOptions
    return launchOptions;
  });
}
```

Figure 28. Set up browser before execution

These codes above are setting up the config for Cypress to work in a great behavior and reduce the flakiness with multi browsers such as Chrome, Edge, Firefox, Electron, and any other browsers based on Chromium and also with multi-operating systems without receiving any warning from the Windows or Linux, especially CentOS.

4.2 Build up End-to-End testing in Cypress

The goal of this building up Cypress is to provide in detail how Cypress is designed and implemented, what are the most important and useful commands for developers to build an efficiency and following the best practices from Cypress instructions. Following the best practices from Cypress is a crucial thing that I will follow in the whole project because this will make the solution more stable and easier to maintain in the future.

Each test script has a common structure that contains test cases that cover all functionalities in a specific site. Due to Cypress's architecture, it will clear everything including cookies, and cache session after each test. This will prevent the working state being shared across test cases. To handle Cypress's architecture, I create a template for building all E2E tests which is similar to the way how we use web applications in real life: login to the application, wait for the page fully loaded, then navigate to the page which we are looking for and wait for the page to be loaded. Finally, we automate tests to cover functionalities and logout or close the application (Figure 29).

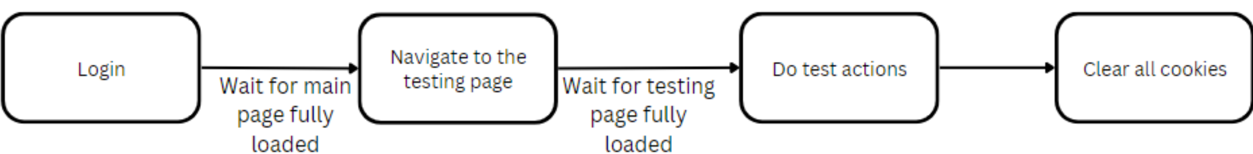


Figure 29. Structure for End-to-end testing of a specific functionality

4.2.1 Build the sharing state between tests

In order to share the state between tests, we will have to turn off `chromeWebSecurity` to "false" (Figure 25) and turn on `experimentalSessionAndOrigin` to "true". Setting up these variables will enable Cypress to work and share the state between tests which can reduce the testing time. Instead of login before each test, Cypress now can cache the cookies, local storage, and session storage in the first test, and in the following tests, Cypress will load them back to log in to our application (Figure 30).

```
cy.session('targetSession', () => {
  cy.visit('/login');
  cy.get('input[name="username"]').type(LOGIN_USERNAME);
  cy.get('input[name="password"]').type(LOGIN_PWD);
  cy.get('#kc-login').click();
});
```

Figure 30. Caching session to login application by using Session

From the function setup above, we first visit the login page and then try to look for the input element which has an attribute inside called *"username"* or "password" and type the credentials. Finally, Cypress will search for the HTML element which has an id named "kc-login," click on that element and we are successfully logged in. All the steps will be cached and stored by Cypress Session for later test usages.

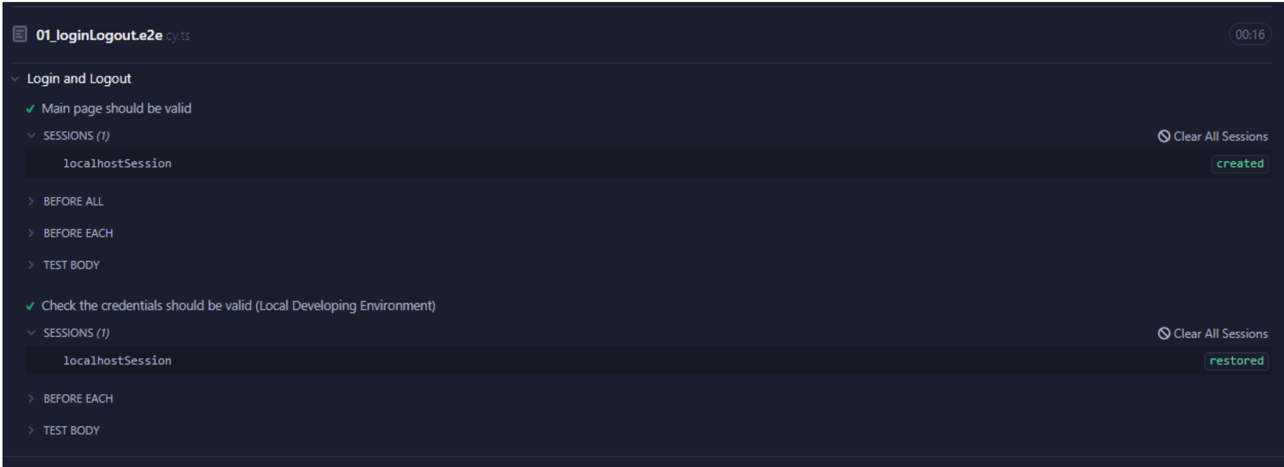


Figure 31. Demo the create and load session

According to Figure 31, which is the demo for the code above, the first test tries to login and Cypress will create a session which it will store all cookies, local and session storage. Then in the second test, instead of logging in once again to do the test, Cypress now will restore the session from the previous test to access the application and run tests.

4.2.2 Assertion

Cypress Session will cache all session data immediately after the function setup completes, it is recommended to give the assertions to ensure the session data is available to cache. Therefore, the login function above will need to have some assertions to ensure that after navigating to URL login and after login successfully, Cypress will wait for the HTML element totally visible, the session actually exists to be cached (Figure 32).

```
cy.session('targetSession', () => {
  cy.visit('/login');
  cy.get('#kc-login').should('contain', 'Sign In');
  cy.get('input[name="username"]').type(LOGIN_USERNAME);
  cy.get('input[name="password"]').type(LOGIN_PWD);
  cy.get('#kc-login').click();
  cy.url().should('contain', '/login-successful');
});
```

Figure 32. Login function with assertions

Manipulating assertions is the best practice when we use Cypress because we can reduce using `wait` which is evaluated as unnecessary and complicated by Cypress. In this project, I do not only use assertion in the login function, but I also apply assertions to all test cases and reduce the usage of `cy.wait(<amount_of_time>)` to achieve the best practice for building up Cypress testing system (Cypress 2022h). This will allow Cypress more flexibility and depend on the backend servers, if the test is faster or slower, there is no need for developers to spend time to estimate how long the wait time is needed to be inserted to the test case.

4.2.3 Creating Cypress custom commands

The more tests are built, the more complexity in lines of test code we will have. Therefore, Cypress has an excellent feature that allows us to create a custom command that we can give a name that is meaningful. This will help other developers quickly adapt and understand what the test does, easily follow, maintain and debug if something is added, or removed from the code base. Moreover, with naming test command meaningful, we can easily understand and reuse the code without needing to create a new test command with executing the same functionalities.

To create a custom command, firstly we will put the name of the test command inside this structure `Cypress.Commands.add(<test_name>, () => {})`. Then we have to declare the name of the login test command as a function called `login` inside the namespace declaration (Figure 33).

```
declare global {
  // eslint-disable-next-line @typescript-eslint/no-namespace
  ...
  namespace Cypress {
    ...
    interface Chainable<Subject>{
      login(): void;
    }
  }
}

Cypress.Commands.add('login', () => {
  cy.session('targetSession', () => {
    cy.visit('/login');
    cy.get('#kc-login').should('contain', 'Sign In');
    cy.get('input[name="username"]').type(LOGIN_USERNAME);
    cy.get('input[name="password"]').type(LOGIN_PWD);
    cy.get('#kc-login').click();
    cy.url().should('contain', '/login-successful');
  });
});
```

Figure 33. Create and define a login custom command

After defining the login custom command, now we can call it into our spec file in the `beforeEach` block (Figure 34).

```
describe('Login and Logout', () => {
  beforeEach(() => {
    cy.login();
  });

  it('Main page should be valid', () => {
    // do tests for checking the main page
  });

  it('Logout should be valid', () => {
    // do tests for log out
  });
});
```

Figure 34. Sample of manipulating the custom command

4.2.4 Conditional

Conditional statements make the application more dynamic, more flexible when it can analyze and handle different inputs and print desired outputs. So does testing applications. However, it is not easy when we do conditional statements in software testing. With human testing, if something takes longer or less around 10ms and 100ms, we never notice this delay or at least we will assume the delay is none. Otherwise, to software testing, machine has no intuition which means it does not care about the delay, it will mark the test fail if the test exceeds the time out, even just only billions

of clock cycle. Therefore, when doing conditional statements in automation testing in GUI application, we have to make sure the DOM have to be fully consistent and cannot change to something else if we do not want the test to become flaky (Cypress 2022i).

For instance, Figure 35 describes a sample situation when there is no logs data, the GUI application will return the text “No activity logs”. Otherwise, it returns all active data.

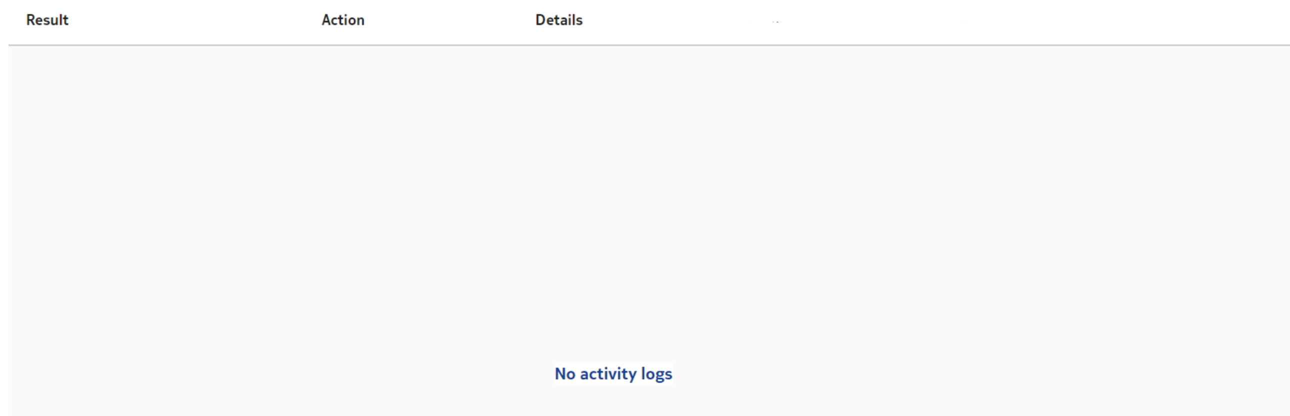


Figure 35. Sample table with no activity logs from GUI application

To satisfying the requirements from Cypress about doing conditional testing, I combine the `get` and `then` command to ask Cypress to find the table body which located inside the data-cy attribute “activity-table” then get all data from that attribute to do conditions (Figure 36).

```
cy.get('[data-cy="activity-table"] tbody')
Complexity is 3 Everything is cool!
.then(body => {
  if(body.find('span:contains(No activity logs)').length === 1){
    cy.get('[data-cy="table-info-text"]').should('contain', 'No activity logs');
  } else {
    cy.get('[data-cy="first-column"]').should('contain', 'Active');
  }
});
```

Figure 36. Sample test command with conditional statement

The command from Figure 36 first has 4 seconds default timeout to find and get the element with the attribute “activity-table” and then navigates inside that element to find “tbody” with 4 seconds default timeout. Then the `then` command will take the data as a parameter and call the callback function to do an if-else statement. If the value inside the parameter has a span element that contains the text “No activity logs,” Cypress will understand that it will find and check whether the “No activity logs” exists in the element which has a data-cy attribute “table-info-text.” Otherwise, Cypress will be asked to search for the element which has a data-cy attribute called “first-column” and contains the text “Active.”

4.3 Enhance in CI environment

At the point of this thesis, the CI environment is currently using Robot Framework to automate the testing system and it performs the testing consistently and efficiently. The greatest point of Robot Framework is the scalability which allows developers, and QA developers can create their own libraries by using built-in functionalities that Robot Framework provides, Python, and all Python's libraries. I do not have any plan for changing to other automation tools while Robot Framework has been built and maintained for several years. Instead of thinking of migration to other automation tools, I think it will be more beneficials to do enhancements for the current CI pipeline.

About the GUI application, its E2E testing is executed in the CI pipeline, however it is executed only on the nightly run by Robot Framework and still missing some testing in the daily run. So that, in order to enhance the quality and make sure the GUI application is still stable after developers do changes, I will use Robot Framework to execute the GUI test in nightly build sequentially and use Pabot - Parallel executor in Robot Framework to execute the GUI test in daily build parallelly.

According to Smith Peter (2011), daily run and nightly run describe the scenario in which software is tested fully automated many times per day. Daily run in this thesis can be understood that whenever a software developer develops a new feature under his own branch and then decides to merge it into the Master branch, Robot Framework will run the GUI E2E tests to verify that all current GUI functionality in the application is stable and nothing has been broken due to the changes. Having the daily run gives instant feedback to software developers about the latest code quality. So does the nightly run. However, the difference between the daily run and the nightly run is that the nightly run is executed once a day and only executed at night while the daily run can be executed unlimited times during the whole day.

4.3.1 Pabot and Robot Framework

Pabot is a parallel executor in Robot Framework and in this project, and it is considered to be used because it is inherited from Robot Framework which supports all Robot Framework arguments in its command line. This means with the Pabot command we only need to replace the `robot` with `pabot` keyword and keep the rest of the arguments (Figure 37).

```
# Robot Framework execution command
robot suites

#  Pabot - parallel execution command
pabot suites
```

Figure 37. Sample commands between Robot Framework and Pabot (adapted from Pabot 2022a)

This is a remarkable thing because I only have to develop one custom library which is suitable for both Parallel and Sequence execution. Then I just need to change the keyword in the command line then Robot Framework and Pabot will do their job.

Pabot is chosen not only it is inherited from Robot Framework, but it is also free and stable. Due to the benefits that Pabot has, Pabot is used to perform the parallel execution that replace the Cypress parallel execution which requires paying cost. Pabot allows developers to split one execution into multiple parallel executions and then combine all results from parallel threads into 1 report for Robot Framework.

Robot Framework and Pabot offer a range of useful arguments for various purposes that fit with mostly all applications. However, Pabot requires its arguments need to put before Robot Framework arguments, otherwise, Pabot will return errors. The arguments which are used in Pabot for this project will be explained below.

Table 1. Descriptions of arguments which are used in this project (adapted from Robot Framework and Pabot)

Argument	Description	Automation tools
`--verbose`	Print more output which is exactly what Robot Framework prints	Pabot
`--ordering`	Give execution order from a file. This argument is helpful for this solution because we can tell Pabot first pull the latest code, then run 3 suites parallelly and after 3 suites finish, combine the result and print the report	Pabot
`--pythonpath`	This argument helps developers to communicate with Robot Framework which modules are needed in order to execute the tests	Pabot & Robot Framework
`--loglevel`	Robot Framework has 3 log levels which are INFO, DEBUG, TRACE. The TRACE level	Pabot & Robot Framework

	which is used mostly in this project will print all details of the tests.	
`--outputdir` `--output`	Define in which the folder contains log.html, report.html and output-xml will be stored after the execution. With this argument, we can easily control deleting, or merge the folder after and before each execution.	Pabot & Robot Framework
`--include`	Execute the tests which have the included tag. We can define which tests we want to run, and which tests we do not want to run. For instance: in the daily run, it is not a clever idea to execute a test which lasts more than 2 hours. It is more ideal to execute in the Nightly run when the time is not a matter.	Pabot & Robot Framework

4.3.2 Install Pabot

Pabot only supports Robot Framework from version 4 or higher versions, we need to ensure the version of Robot Framework is greater than 4. To check the Robot Framework version, we can execute the command `robot --version`.

```
[                    ]$ robot --version
Robot Framework 5.0.1 (Python 3.6.8 on linux)
[                    ]$
```

Figure 38. Checking the version of Robot Framework on machine

If the version of Robot Framework is compatible with Pabot, then we can install Pabot by using Python package installer (pip) to install.

```
pip install robotframework-pabot
```

Figure 39. Pabot installation

4.4 Architecture for parallel execution and sequence execution

As I mentioned in chapter 4.3.1, because of the relationship between Pabot and Robot Framework, I can build one custom library that is consistent with both 2 executing methods sequence and

parallel. However, to make sure both sequence execution and parallel execution are working, we need to design the whole test based on what Pabot and Robot Framework can do, how it will run and how we build our test in Robot Framework.

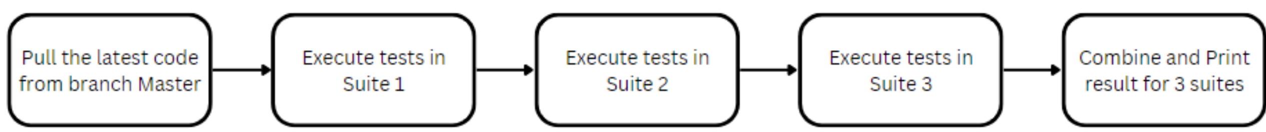


Figure 40. Sequence Execution with Robot Framework for GUI testing in CI pipeline

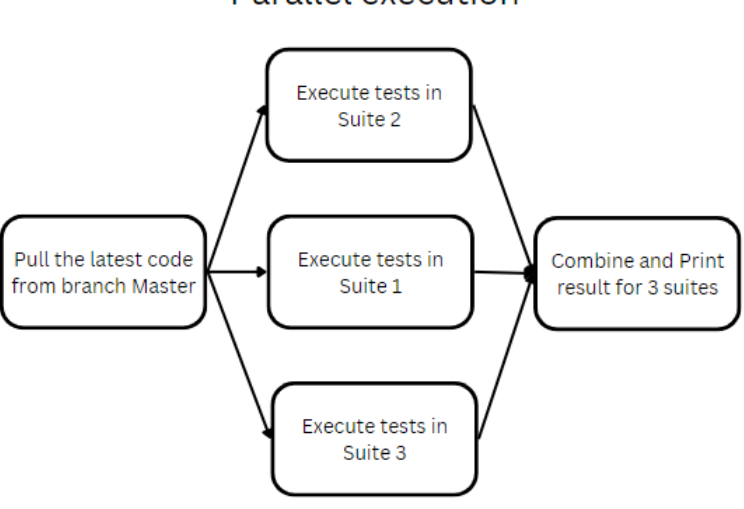


Figure 41. Parallel Execution with Pabot for GUI testing in CI pipeline

According to Figures 40 and 41, both figures have the same containers with the same functionalities. However, they are different testing workflows. If Figure 40 describes the sequence execution by Robot Framework which executes the tests step by step from pulling the latest code, to executing each single 3 suites to combine and finally printing out the result of 3 test suites, with Figure 41, after pulling the latest code, instead of running 3 suites step by step, we will run 3 suites in 3 parallel threads which will be controlled by Pabot then combine and return result.

To make Pabot understand our workflow, I need to declare the order of the files in file **.pabotsuitenames-ordering-wait** and mention this file following the `--ordering` argument:

```
--ordering .pabotsuitenames-ordering-wait
```

```
--suite Gui E2E.00 Gui E2E Setup
#WAIT
--suite Gui E2E.01 Gui E2E Suite
--suite Gui E2E.02 Gui E2E Suite
--suite Gui E2E.03 Gui E2E Suite
#WAIT
--suite Gui E2E.99 Gui E2E Report
```

Figure 42. The order of parallel in .pabotsuitenames-ordering-wait

4.5 Build up custom library in Robot Framework

According to Figure 40 and 41 in previous subchapter, each suite file will stand for one Robot Framework suite with different purposes which is:

- **Set up the system for executions: 00_gui_e2e_setup**
- **Run the tests: 01_gui_e2e_suite, 02_gui_e2e_suite, 03_gui_e2e_suite**
- **Combine result and print report: 99_gui_e2e_report**

Before we start to build the custom library for executing the test in Robot Framework which also has a possibility to run tests parallelly, I will explain why we build the custom library. Building the custom library will allow us to have the possibility to do what we want. For instance, even though Cypress offers a built-in function for parallel execution and wait for Cypress to fix bugs if they are existed, I can build my own parallel execution and do other jobs as I want.

In addition, I can reuse and do enhancements on the existing code which has been implemented to achieve a stable situation for many years. After achieving our own custom library, we can automate, scale the test, the report, and do our own customization depending on our needs. This will not only allow me but also other developers to have the freedom to build useful things to execute E2E testing in CI and not have to wait for Cypress.

Firstly, we need to create a custom library for our Robot Framework in order to execute the E2E test in GUI application. In this case, I create a folder called **gui_e2e** which will contain everything related to executing the E2E in GUI application. Inside this folder, I created another called **libraries** and one more down level, I created a python file which I named **gui_e2e_handling.py**. This python file will contain all custom functions that we are going to build for the above architectures.

```
gui_e2e
  libraries
    _pycache_
    gui_e2e_handling.py
```

Figure 43. Directory of custom libraries

4.5.1 Set up the system for executions

The set up phase always act a crucial role in a whole testing system so that the purpose of the Robot Framework script which is shown in Figure 44 is get the latest code from branch Master and execute to ensure the stability of the application. It will meaningless if the tested code is old one and not up-to-date.

```
*** Settings ***
Documentation   GUI Setup
Library     gui_e2e_handling.guiE2EHandling     WITH NAME     guiE2EHandling
Force Tags      gui_e2e
Suite Setup     clean_previous_report_data


*** Test Cases ***
GUI_E2E_set_up_the_latest_version
    [tags]     GUI_PARALLEL
    [Documentation]   ..........................................................
    ...               - Get the latest version of GUI application
    initialize


*** Keywords ***
clean_previous_report_data
    clear_report_content
```

Figure 44. Clear previous data and pull the latest code in Robot Framework script

The implementation how the set up phase is built is shown in the Figure 44 above. We import the custom library **gui_e2e_handling** with its class **guiE2EHandling** and rename it to “guiE2EHandling” by using `WITH NAME` keyword as the first step. In the testcase with description “GUI_E2E_set_up_the_latest_version”, I call the **initialize** function from our custom library. This function will check whether the GUI folder exists or not by using OS module in Python. If not, it will clone the latest source code and run `npm install`. If the GUI folder exists, this function will pull the latest version of the code base (Figure 45). The OS module in Python which provides functions for developers to interact with the Operating System is the standard module in Python.

```
def initialize(self):
    if os.path.exists(AGUI_FOLDER):
        os.system(self.cmd_pull)
    else:
        os.system(self.cmd_clone)
        os.system(self.cmd_install)
```

Figure 45. Check and pull the latest code for working project

At the top of Robot Framework script, there is a keyword `Suite Setup` that has a function **clean_previous_report_data**. The purpose of this keyword is before every time our **00_gui_e2e_setup** suite is executed, the function inside it will be executed first. This function will open and clear the text file where the results of each test in suites 1,2 and 3 are written after their executions from the previous executions (Figure 46). There is a question that comes up in this state: Why do I have to write all suite results to a text file?

```
def clear_report_content(self):
    TARGET_FOLDER = "reports/"
    target_filename = TARGET_FOLDER + "gui_e2e_reports.txt"
    if os.path.exists(TARGET_FOLDER):
        target_file = open(target_filename, 'w')
        target_file.close()
```

Figure 46. Clean up to avoid duplicated data before whole test is executed

According to Figure 46, we also use the "OS" module in Python to check whether the path of the text file which we use to store all results data exists or not. If the path exists, we will clean the data in Python way that opens the text file in write mode 'w' and close it. Otherwise, we can continue to the next steps of the whole test system.

Moreover, I will explain why I have to open and write the results of tests to a text file before each execution and why I have to clear everything in a text before executing the whole test when we are implementing in the final suite.

4.5.2 Run tests

After completing building the setup suite **00_gui_e2e_setup**, the next 3 suites will have the same structure that we call function **execute_gui_test** with 2 parameters to execute the specific test and when all tests in a suite are completed, function **print_result_for_suite_index** with the parameter suite index will be called to do 2 jobs. Initially, it will print the result for each suite then it will write results to a text file (Figure 47).

```
*** Settings ***
Documentation   Suite 1
Library     gui_e2e_handling.guiE2EHandling    WITH NAME    guiE2EHandling
Test Timeout        4 minutes
Suite Teardown      print_result_table
Force Tags          gui_e2e


*** Test Cases ***
GUI_E2E_test_site_1
    [tags]     GUI_PARALLEL
    [Documentation]   .........................................................
    ...               - Site 1
    execute_gui_test  1  1

GUI_E2E_test_site_2
    [tags]     GUI_PARALLEL
    [Documentation]   .........................................................
    ...               - Site 2
    execute_gui_test  1  2

GUI_E2E_test_site_3
    [tags]     GUI_PARALLEL
    [Documentation]   .........................................................
    ...               - Site 3
    execute_gui_test  1  3

GUI_E2E_test_site_4
    [tags]     GUI_PARALLEL
    [Documentation]   .........................................................
    ...               - Site 4
    execute_gui_test  1  4


*** Keywords ***
print_result_table
    print_result_for_suite_index  1
```

Figure 47. Sample script of Suite 1 in Robot Framework

The **execute_gui_test** firstly will open the terminal to execute a specific test based on those parameters. The execution command is more complex than usual because in the CI pipeline, the operating system which is Linux has some conflict between Linux and Chrome when we try to open from the terminal and as default, Cypress console output is not readable in CLI console. The reason is Cypress is using the ANSI color escape characters to format its output and this is not compatible with my log viewer. So that the command line for execute a specific test need to have the argument “NO_COLOR=1” as Figure 48.

```
cmd_single_exec = "(cd " + AGUI_FOLDER + " && NO_COLOR=1 xvfb-run -a npm run cypress:run:" + index_spec_file + ")"
```

Figure 48. Execution command for single test in CI

The result of the command will be stored in a variable and after that, we will split the result into single lines and start to check if the result is pass or fail by checking the name of the test and the status “✕” or “✓” (Figure 49).

```python
result = os.popen(cmd_single_exec).read()

if not result:
    raise ValueError("GUI E2E execute fail (no result) failed!")

result_splitted = result.split('\n')
failure = False
execution_result = ''
for line in result_splitted:
    if cypress_spec_name in line:
        if '✕' in line:
            failure = True
            execution_result = line
            break
        if '✓' in line:
            execution_result = line
            break
```

Figure 49. Read and check the result

Each Cypress result is collected initially from as a string format with many white spaces and 8 types of data which stand for:

- Status
- Test name
- Execution time
- Amount of test
- Passed amount
- Failed amount
- Pended amount
- Number of skipped tests.

We call the **beautify_result_into_array** function to do regular expression and store it as an array with exactly 8 elements. If there is a situation that the length of data which is collected is not equal to 8 then we mark it as an error case. Moreover, depending on which suite the test belongs, the handled result will be stored in its own suite by calling the function **append_result_into_suite** with 2 parameters suite index and the handled result (Figure 50).

```
handled_execution_result = self.beautify_result_into_array(execution_result)
if len(handled_execution_result) == 8:
    self.append_result_into_suite(suite_index, handled_execution_result)
    if (failure):
        BuiltIn().log_to_console("%s"%result)
        raise ValueError('GUI E2E test failed! - Spec name: ' + cypress_spec_name)
    else:
        debug("%s"%('GUI E2E test passed! - Spec name: ' + cypress_spec_name))
else:
    error_result = ['ERROR', cypress_spec_name + '.e2e.cy.ts', '00:00', '-', '-', '-', '-', '-']
    self.append_result_into_suite(suite_index, error_result)
    BuiltIn().log_to_console("%s"%result)
    raise ValueError("Something went wrong during execution! - Please check the report for more information!")
```

Figure 50. Handle and append the test result into its specific suite

4.5.3 Combine results and print report

Before we move to implement the final suite, I will explain why the results of suites will be written to a text file is easy to handle the order of parallel execution. Let's imagine we have 4 parallel suites that are executed to run as the same. How can we control and communicate the results if each suite is executed in different processes of our machine? The solution at the point of this thesis is after each suite is executed, the results will be written in the same text file then we can easily handle it by arrange in the order, find the missing tests, check the whole test system passed, or failed or error.

To implement the fifth suite, we call **get_report_of_suites** function which will open a text file which contains all the result from suites then handle it. This function will initially arrange the order of the test in alphabetical order. If we run in the sequence order, the order of the result is absolutely in the ascending order. However, if we execute with parallel order, the order of the result will be in random order depending on which test is finished first.

After sorting the tests for the whole test, this **get_report_of_suites** will check whether there any missing tests or not. If there is some missing, failed, or error, the final report will print out missing tests (Figure 51). Otherwise, it will combine all results and print the final reports.

```
****************************************************************************************************************************
****************************************************************************************************************************
*** GUI E2E STATUS :: OK | Conclusion: *** All specs passed! *** | Total Time (Cypress): 00:07:36
****************************************************************************************************************************
*** GUI E2E RESULT :: Total Executed Test Cases: 75 cases | Passed: 75 cases | Failed: 0 cases | Pended: 0 cases | Skipped: 0 cases.
****************************************************************************************************************************
****************************************************************************************************************************
The following E2E testcases were not executed:
01_loginLogout
****************************************************************************************************************************
```

Figure 51. A situation when there is a test is missing after parallel execution

In this project, there are 4 types of status: “OK“, “FAILED“, “ERROR“, “FAILED and ERROR“. If the whole test is passed and no matter if there are some missing tests, the status is “OK”. If there is some test failed due to actual testing reasons, we print “FAILED”. Otherwise, if some tests failed due to system errors or technical errors, we print “ERROR”. Moreover, if there are failed cases and error cases exist, we print “FAILED and ERROR”.

```
*** Settings ***
Documentation   Suites Report
Library     gui_e2e_handling.guiE2EHandling     WITH NAME     guiE2EHandling
Force Tags          gui_e2e


*** Test Cases ***
GUI_E2E_print_the_final_report
    [tags]     GUI_PARALLEL
    [Documentation]    ..........................................................
    ...                - Table of 3 seqeuence/parallel suites
    get_report_of_suites
```

Figure 52. Get the final result in Robot Framework script

5 Conclusion

The goal of this thesis is building the new E2E testing system and do enhancements in the CI pipeline with Robot Framework. The empirical part of this thesis is divided into 2 main tasks. If the initial task shows the particular way from install, set up, verify Cypress to build up Cypress, the second task improves the result of GUI testing in the CI pipeline's report and also create a custom library which provide the possibility to execute CI pipeline in parallel order and sequence order.

This project is useful for people who are looking for another option to run the test parallelly and do not want to use the parallel function from Cypress which is required paying cost. I hope that this thesis not only brings benefits to my commission party, but also be useful for other parties, undergraduates and developers who are interested in developing Quality Assurance (QA) for GUI application.

5.1 Achieved Objectives

At the final stage of this thesis, all objectives that were initialized at the beginning of this project were implemented and met:

5.1.1 Implement new E2E testing system

- Learn Cypress.
- Set up Cypress for Angular application, stable in execution through various Operating System such as Windows 10 and Linux CentOS.
- Figure how to switch testing between local development phase and production phase by setting Cypress variable `baseUrl`.
- Replace the old framework Protractor by new framework Cypress and build test cases.

5.1.2 Enhance the CI pipeline performance

- Learn Python and Robot Framework.
- Design and implement the report of GUI E2E testing in CI pipeline.
- Design and build the architecture that E2E testing can be run in parallel execution and sequence execution depending on nightly run or daily run.

5.2 Results of the project

According to our nightly reports, GUI application is now more consistent with the new automation framework Cypress compared to the old framework Protractor. Cypress solves the flaky situation from the previous E2E automation framework. In addition, our GUI developers now can easily create an automation test cases right away after they develop new features to GUI application when they are already familiar with TypeScript code which is used to write test code in Cypress.

After GUI E2E test was implemented and applied to the CI pipeline, it did not only enhance the status of the whole GUI E2E testing in the Nightly run, but also in the Daily run. For now, with the daily run in the CI pipeline, the GUI application has to pass the whole E2E test which is executed in parallel by Pabot. If everything passes, the code will be accepted to merge into the branch master. If not, developers need to check what is wrong and debug then push the fixes.

About the execution time, Pabot has a built-in function that will print the running time between parallel and sequence order to the end user. Based on the GUI testing report, the difference between the running time in parallel execution and sequence execution is approximately a half. According to Figure 53, there is a variable called Elapsed time has value is 7 minutes and 7.10 seconds which is stand the parallel execution time recorded by Pabot and the Total testing with and 13 minutes and 57.10 seconds is the execution time when the test is executed in sequence order.

```
Total testing: 13 minutes 57.10 seconds
Elapsed time:  7 minutes 7.10 seconds
```

Figure 53. The comparison time between parallel order and sequence order

Both parallel execution and sequence execution time are including the setup and execution time that Robot Framework and Pabot take to do the testing job, therefore, they are not similar to the total time (Cypress) which is shown in Figure 55. The total time (Cypress) combines all running time of every single test in Cypress and then presents the sum of the execution time.

According to Figure 54 and Figure 55, there are the old GUI E2E report and the new E2E test report which show the result with different ways of displaying the final result. We can easily see that now the new GUI report has a more meaningful report that contains the status of all tests: “OK”, “FAILED”, “ERROR” or “FAILED & ERROR”, total time (Cypress), the number of tests, passing, failing, pending, and skipped cases.

```
Executed 51 of 51 specs SUCCESS in 2 mins 1 sec.
```

Figure 54. Final GUI E2E report in CI pipeline before this thesis

```
*** GUI E2E STATUS :: OK | Conclusion: *** All specs passed! *** | Total Time (Cypress): 00:07:37
*** GUI E2E RESULT :: Total Executed Test Cases: 77 cases | Passed: 77 cases | Failed: 0 cases | Pended: 0 cases | Skipped: 0 cases.
```

Figure 55. Final GUI E2E report in CI pipeline after this thesis

5.3 Limitations of the software solution

On the other hand, this project still has a few limitations with Cypress in GUI application. On Cypress's side, there is some conflict with the proxy settings on the Windows machine. So, when developers face the proxy problem, they have to reload the page with their testing URL several times to navigate successfully. However, this only applies to Windows machines when developers develop new tests. In the CI pipeline, we use Linux and there is no similar error occurring. Cypress mentioned that if their instructions do not solve the situation, then try to ask the network admin team to allow Cypress to go through and this is not an ideal thing to follow and this is out of scope of this project.

Secondly, since the versions of Cypress 10, Cypress takes more memory than the previous versions and unfortunately, this is not compatible with Chrome since version 103. If the tests take over 4GB of memory RAM in Chrome, Chrome will run out the memory errors and stop our test. To solve this situation, we have to refactor or split the tests into smaller tests which take less memory. Otherwise, we can set `numTestsKeptInMemory` to 0 in the **cypress.config.ts** file. This variable will ask Cypress not to record anything and just run the whole test. This can be useful in the headless mode when we do not care about the debugging, however, in the headful mode, when we want to see why the tests failed, enable the `numTestsKeptInMemory` variable is not an ideal solution.

Finally, the last limitation of this project is when the whole test executed in parallel by Pabot, sometimes there are some testcases are skipped and are not executed in parallel threads. According to Pabot's community, this is a limitation from this package and need to be improved by Pabot development team (Pabot 2022b). Fortunately, there is the possibility to rerun the failed, skipped cases by using 2 built-in parameters from Robot Framework `--rerunfailed` and `–merge`.

5.4 Lessons learned

During the time developing this project, I have a terrific opportunity to learn and touch with many modern and exciting technologies, and automation frameworks such as Cypress, and Robot Framework. I am given a chance to study and responsible to build the new testing system from scratch to replace the old one which will be deprecated in Angular 15. This means that I learn how to effectively build E2E testing systems which are clean, easy to maintain and do scalability. Therefore, I have improved my skills a lot in writing test in TypeScript code.

In addition, I have never imagined that I would someday learn and use Python to build a custom library in Robot Framework. I learned basic Python before I started, but I have to say that my knowledge of Python is greater after this project. I know how to collect data in array format then print the table for the data, know how to call and use existing functions to support my work then write clean and efficient code in Python also.

About project management, I have learned to divide tasks into smaller and more manageable ones. This helps me to control the time frame to build up, maintain and do the fixtures for my project so that I am able to finalize the project on time with the initial plan.

5.5 What next?

This project completed its initial goals when the testing system itself in GUI application is stable and so execute Nightly run and Daily run. However, there are some enhancements that I could implement in the following versions after this thesis. Now, the E2E testing works quite good and consistently. However, it could be improved to be more dynamic and robust. For instance: Cypress creates something in the first test and for some reason, Cypress could "forget" to delete that role. This will make the following execution fail because when the case "Create something" was executed already from the previous and was not deleted by Cypress.

Regarding the stability of the result in CI pipeline, sometimes the testing system with more than 80 test cases can fail in 1 or 2 tests and this is not a promising idea to re-execute the whole test. Fortunately, Robot Framework has 2 arguments to support the rerun with the failed cases which are `--rerunfailed` and `--merge`. The first argument will read the output from the previous result then rerun the failures and print the output for the rerun. And the second argument will merge the first output with the recent output and return the new result of the whole test. However, this is just an initial thought for next steps after this thesis and need to spend time to study and build up functions in the custom library.

References

Angular. 2021. Future of Angular E2E & Plans for Protractor. URL: https://github.com/angular/protractor/issues/5502. Accessed: 19 September 2022.

Aniche Mauricio. 2022. Effective Software Testing. O'Reilly Media, Inc 2022.

Bose Shreya. 2020. Testing Pyramid: How to jumpstart Test Automation. URL: https://www.browserstack.com/guide/testing-pyramid-for-test-automation. Accessed: 19 September 2022.

Buckler Craig. 2022. Node.js: Novice to Ninja. O'Reilly Media, Inc 2022.

Cohn Mike. 2009. Succeeding with Agile. Addison-Wesley, 2010.

Cypress. 2022a. Installing Cypress. URL: https://docs.cypress.io/guides/getting-started/installing-cypress. Accessed: 17 September 2022.

Cypress. 2022b. Migrating from Protractor to Cypress. URL: https://docs.cypress.io/guides/end-to-end-testing/protractor-to-cypress#What-you-ll-learn. Accessed: 17 September 2022.

Cypress. 2022c. Migrating from Protractor to Cypress. URL: https://docs.cypress.io/guides/end-to-end-testing/protractor-to-cypress#Recommended-Installation. Accessed: 20 September 2022.

Cypress. 2022d. Installing Cypress. URL: https://docs.cypress.io/guides/getting-started/installing-cypress#npm-install. Accessed: 20 September 2022.

Cypress. 2022e. Writing and Organization tests – Test statueses. URL: https://docs.cypress.io/guides/core-concepts/writing-and-organizing-tests#Test-statuses. Accessed: 20 September 2022.

Cypress. 2022f. Test Retries. URL: https://docs.cypress.io/guides/guides/test-retries. Accessed: 21 September 2022.

Cypress. 2022g. clearCookies. URL: https://docs.cypress.io/api/commands/clearcookies. Accessed: 21 September 2022.

Cypress. 2022h. Session. URL: https://docs.cypress.io/guides/references/best-practices#Unnecessary-Waiting. Accessed: 4 October 2022.

Cypress. 2022i. Conditional Testing. URL: https://docs.cypress.io/guides/core-concepts/conditional-testing. Accessed: 5 October 2022

Datey Meera. 2021. Why building modern web applications are so darn complex? URL: https://www.linkedin.com/pulse/why-building-modern-web-applications-so-darn-complex-meera-datey?trk=articles_directory. Accessed: 13 September 2022.

Gawron Karolina & Fluin Stephen. 2018. Angular, the Successor of AngularJS, Is Thriving and Has Big Plans for the Future. An Interview with Stephen Fluin. URL: https://www.monterail.com/blog/angular-development-google-stephen-fluin. Accessed: 14 September 2022.

Goldberg Josh. 2022. Learning TypeScript. O'Reilly Media, Inc 2022.

Google LLC. 2022. What is Angular? URL: https://angular.io/guide/what-is-angular. Accessed: 14 September 2022.

Hambling Brian & Samaroo Angelina. 2009. Software Testing - An ISEB Intermediate Certificate. O'Reilly Media, Inc 2022.

Laster Brent. 2020. Continuous Integration vs. Continuous Delivery vs. Continuous Deployment, 2nd Edition. O'Reilly Media, Inc 2022.

Loubser Nico. 2021. Software Engineering for Absolute Beginners: Your Guide to Creating Software Products. O'Reilly Media, Inc 2022.

Martelli Alex, Martelli Anna Ravenscroft, Holden Steve, McGuire Paul. 2022. Python in a Nutshell, 4th Edition. O'Reilly Media, Inc 2022.

McKinney Wes 2022. Python for Data Analysis, 3rd Edition. O'Reilly Media, Inc 2022.

Microsoft. 2022. TypeScript for JavaScript Programmers. URL: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html. Accessed: 15 September 2022.

Mwaura Waweru. 2021. End-to-End Web Testing with Cypress. O'Reilly Media, Inc 2022.

Nokia Oyj. 2021. Nokia Strategy. URL: https://www.nokia.com/about-us/company/nokias-strategy-2021/. Accessed: 11 September 2022.

Nokia Oyj. Company. URL: https://www.nokia.com/about-us/company/. Accessed: 11 September 2022.

Nology Team. 2020. HTML, CSS and JavaScript: The Anatomy of A Website. URL: https://www.nology.io/news/html-css-and-javascript-the-anatomy-of-a-website/. Accessed: 15 September 2022.

NPM. 2022. TypeScript. URL: https://www.npmjs.com/package/typescript. Accessed: 16 September 2022.

Pabot. 2022a. Pabot. URL: https://pabot.org/. Accessed: 3 October 2022.

Pabot. 2022b. Pabot got stuck even after Test timeout was mentioned in the test case. URL: https://github.com/mkorpela/pabot/issues/321. Accessed: 17 October 2022.

Robot Framework. 2022. Robot Framework. URL: https://robotframework.org/. Accessed: 27 September 2022.

Shetty Rahul. 2021. Cypress - Modern Automation Testing from Scratch + Framework. O'Reilly Media, Inc 2022.

Smith Peter. 2011. Software Build Systems: Principles and Experience. Addison-Wesley Professional.

Smith Steve. 2022. Architect Modern Web Applications with ASP.NET Core and Microsoft Azure. .NET Docs. Accessed: 13 September 2022.

Stack Overflow. 2022. Developer Survey. URL: https://survey.stackoverflow.co/2022/#technology-most-popular-technologies. Accessed: 15 September 2022.

Stempniak Adam & Świstak Tomasz. 2022. What Is TypeScript? Pros and Cons of TypeScript vs JavaScript. URL: https://www.stxnext.com/blog/typescript-pros-cons-javascript/. Accessed: 15 September 2022.

TIOBE. 2022. TIOBE Index for September 2022. URL: https://www.tiobe.com/tiobe-index/. Accessed: 27 September 2022.

Ubah Kingsley. 2021. Learn Web Development Basics – HTML, CSS, and JavaScript Explained for Beginners. URL: https://www.freecodecamp.org/news/html-css-and-javascript-explained-for-beginners/. Accessed: 15 September 2022.

Wassell Shaun. 2021. Node.js Essentials. O'Reilly Media, Inc 2022.

Table of Figures

Figure 1. The most commonly programming languages in 2022 (Stack Overflow's survey 2022) 5

Figure 2. No complaints from JavaScript (adapted from Goldberg Josh 2022, Chapter 1) 6

Figure 3. TypeScript reporting an error (adapted from Goldberg Josh 2022, Chapter 1) 6

Figure 4. The Programming Languages Ranking for September 2022 (TIOBE Index 2022) 7

Figure 5. Sample Robot Framework script with importing custom library 9

Figure 6 Testing Activities in the life cycle (adapted from Hambling Brian & Samaroo Angelina 2009, Chapter 1) 11

Figure 7. The Pyramid of testing (adapted from Mike Cohn 2009) 12

Figure 8. E2E testing tools trend (Angular's survey 2021) 14

Figure 9. Cypress test execution architecture (adapted from Mwaura Waweru, Chapter 2) 16

Figure 10. Open Cypress launchpad 16

Figure 11. Cypress Dashboard 17

Figure 12. The available browsers on machine that Cypress supports 18

Figure 13. Installing Schematic package from Cypress 20

Figure 14. Install Cypress via npm 20

Figure 15. Cypress verification process 20

Figure 16. Open Cypress Dashboard only 21

Figure 17. Build Angular server and open Cypress Runner 21

Figure 18. Running Cypress in headless mode 21

Figure 19. The sample output of the specific Cypress test in headless mode 22

Figure 20. Cypress Hierarchy diagram 22

Figure 21. Define the location of subfolders 23

Figure 22. Define the scroll behavior 23

Figure 23. Increase the number of retries when the test failed in both headed and headless mode 23

Figure 24. Set baseUrl to null make testing system more flexible in switching between Develop and Product phase 23

Figure 25. Allow Cypress to work with super domains, cross-origin 24

Figure 26. Sample code about testcases in todo app 24

Figure 27. Keep the memory after each test in a suite 25

Figure 28. Set up browser before execution 25

Figure 29. Structure for End-to-end testing of a specific functionality 26

Figure 30. Caching session to login application by using Session 27

Figure 31. Demo the create and load session 27

Figure 32. Login function with assertions 28

Figure 33. Create and define a login custom command 29

Figure 34. Sample of manipulating the custom command 29

Figure 35. Sample table with no activity logs from GUI application 30

Figure 36. Sample test command with conditional statement 30

Figure 37. Sample commands between Robot Framework and Pabot (adapted from Pabot 2022a) 32

Figure 38. Checking the version of Robot Framework on machine 33

Figure 39. Pabot installation 33

Figure 40. Sequence Execution with Robot Framework for GUI testing in CI pipeline 34

Figure 41. Parallel Execution with Pabot for GUI testing in CI pipeline 34

Figure 42. The order of parallel in .pabotsuitenames-ordering-wait 35

Figure 43. Directory of custom libraries 36

Figure 44. Clear previous data and pull the latest code in Robot Framework script 36

Figure 45. Check and pull the latest code for working project 37

Figure 46. Clean up to avoid duplicated data before whole test is executed 37

Figure 47. Sample script of Suite 1 in Robot Framework 38

Figure 48. Execution command for single test in CI 38

Figure 49. Read and check the result 39

Figure 50. Handle and append the test result into its specific suite 40

Figure 51. A situation when there is a test is missing after parallel execution 40

Figure 52. Get the final result in Robot Framework script 41

Figure 53. The comparison time between parallel order and sequence order 43

Figure 54. Final GUI E2E report in CI pipeline before this thesis 43

Figure 55. Final GUI E2E report in CI pipeline after this thesis 44

Appendices

Appendix 1. Complete setup configuration for Cypress E2E testing framework

```
import { defineConfig } from 'cypress';

export default defineConfig({
  video: false,
  screenshotOnRunFailure: true,
  screenshotsFolder: 'cypress/screenshots',
  videosFolder: 'cypress/videos',
  fixturesFolder: 'cypress/fixtures',
  downloadsFolder: 'cypress/downloads',
  env: {···
  },
  viewportHeight: 1000,
  viewportWidth: 1600,
  scrollBehavior: 'nearest',
  chromeWebSecurity: false,
  retries: 2,
  e2e: {
    baseUrl: null,
    experimentalSessionAndOrigin: true,
    experimentalStudio: true,
    Complexity is 9 It's time to do something...
    setupNodeEvents(on, config) {
      Complexity is 8 It's time to do something...
      on('before:browser:launch', (browser, launchOptions) => {
        // `args` is an array of all the arguments that will
        // be passed to browsers when it launches

        if (browser.family === 'chromium' && browser.name !== 'electron') {
          // auto open devtools
          launchOptions.args.push('--auto-open-devtools-for-tabs');
        }

        if (browser.name === 'chrome' && browser.isHeadless) {
          launchOptions.args.push('--disable-gpu');
          launchOptions.args.push('--disable-software-rasterizer');
          launchOptions.args.push('--no-sandbox');
        }

        if (browser.family === 'firefox') {
          // auto open devtools
          launchOptions.args.push('-devtools');
        }

        if (browser.name === 'electron') {
          // auto open devtools
          launchOptions.preferences.devTools = true;
        }
        // whatever you return here becomes the launchOptions
        return launchOptions;
      });
    }
  },

});
```

Appendix 2: Complete execute_gui_test in custom library

```
def execute_gui_test(self, suite_index, index_spec_file):
    cypress_spec_name = CYPRESS_LIST[int(index_spec_file) - 1]
    cmd_single_exec = "(cd " + AGUI_FOLDER + " && NO_COLOR=1 xvfb-run -a npm run cypress:run:" + index_spec_file + ")"

    result = os.popen(cmd_single_exec).read()

    if not result:
        raise ValueError("GUI E2E execute fail (no result) failed!")

    result_splitted = result.split('\n')
    failure = False
    execution_result = ''
    for line in result_splitted:
        if cypress_spec_name in line:
            if '✕' in line:
                failure = True
                execution_result = line
                break
            if '✓' in line:
                execution_result = line
                break

    handled_execution_result = self.beautify_result_into_array(execution_result)
    if len(handled_execution_result) == 8:
        self.append_result_into_suite(suite_index, handled_execution_result)
        if (failure):
            BuiltIn().log_to_console("%s"%result)
            raise ValueError('GUI E2E test failed! - Spec name: ' + cypress_spec_name)
        else:
            debug("%s"%('GUI E2E test passed! - Spec name: ' + cypress_spec_name))
    else:
        error_result = ['ERROR', cypress_spec_name + '.e2e.cy.ts', '00:00', '-', '-', '-', '-', '-']
        self.append_result_into_suite(suite_index, error_result)
        BuiltIn().log_to_console("%s"%result)
        raise ValueError("Something went wrong during execution! - Please check the report for more information!")
```