UNIVERSITETI I EVROPËS JUGLINDORE
УНИВЕРЗИТЕТ НА ЈУГОИСТОЧНА ЕВРОПА
SOUTH EAST EUROPEAN UNIVERSITY

POSTGRADUATE STUDIES – SECOND CYCLE

THESIS:

“E2E Web Application Testing”

CANDIDATE:

Blerand Zendeli

MENTOR:

Prof. dr. Artan Luma

Tetovë, 2021

Contents

Proofreading Confirmation

I, Kujtim Ramadani, certify that the manuscript below was edited/proofread for proper English Language, including grammar, punctuation, spelling and overall style.

In Tetovo, on 02.02.2021

Declaration of original work

I certify that I am the original author of this thesis. I have not copied from any other students' work or from any other sources apart from reviewed references in accordance with the rules of referencing.

Acknowledgements

First, I would like to express my special gratitude to my supervisor, Prof. Dr. Artan Luma for his extreme support and motivation. His inspiring suggestions have been precious for the development of the content of this thesis.

In addition, I would like to express my sincere gratitude to all my family members and friends for their support and patience during the preparation of this study. I have been very lucky to have such a great family and friends.

Abstract

The main intention of this master thesis is to make a comparison between two end-to-end testing frameworks such as TestCafe and Cypress.

In this thesis, we have implemented end-to-end testing using these two frameworks on a web application built with reactJs and firebase.

By implementing these two frameworks on this web application, we have analyzed all the details related to implementation, documentation, benefits, pros and cons, performance testing, CI integration, Cloud testing integration, selectors playground, browser support.

In this thesis, we have compared performance testing of two testing frameworks on two different web browsers such as Chrome headless and Firefox headless whereupon there was a big difference between Cypress and TestCafe in terms of running time.

In this thesis, we have carried out the Continuous integration of two testing frameworks on GitLab CI/CD, as well as the integration on Cloud testing platforms such as LambdaTest.

We have also compared browser support between two frameworks and we have analyzed selector playground.

Abstrakt

Qëllimi kryesor i kësaj teze master është të bëjë një krahasim midis dy sistemeve të testimit end-to-end si TestCafe dhe Cypress.

Në këtë tezë kemi zbatuar testimin "end-to-end" duke përdorur këto dy sisteme në një ueb aplikacion me ane te këtyre dy kornizave reactJs, firebase.

Duke zbatuar këto dy korniza ne kemi analizuar të gjitha detajet në lidhje me zbatimin, dokumentacionin, përfitimet, avantazhet dhe dizavantazhet, testimin e performancës, integrimin e CI, integrimin "Cloud", fushën për të zgjedhur selektorët dhe mbështetjen e shfletuesit.

Në këtë tezë është bërë krahasimi në testimin e performancës ne mes dy kornizave të testimit, në dy shfletues ("browser") të ndryshëm të internetit si "Chrome headless" dhe "Mozilla firefox headless" ku ka pasur një ndryshim të madh midis "Cypress" dhe "TestCafe" në kohën e ekzekutimit.

Në këtë tezë është bërë integrimi i vazhdueshëm i dy kornizave të testimit në GitLab CI / CD, gjithashtu integrimi në platformën e testimit "Cloud" si LambdaTest.

Në këtë tezë është bërë edhe krahasimi i mbështetjes së shfletuesit midis dy kornizave dhe po ashtu kemi analizuar fushën për t'u zgjedhur selektorët.

Абстракт

Главната намера на оваа магистерска теза е да се направи споредба помеѓу две рамки за тестирање од крај до крај, како што се TestCafe и Cypress.

Во оваа теза спроведовме тестирање од крај до крај со користење на овие две рамки на веб-апликација изградена со реактивни мрежи и огнена база.

Со имплементирање на овие две рамки на оваа веб-апликација ги анализиравме сите детали поврзани со имплементацијата, документацијата, придобивките, добрите и лошите страни, тестирање на перформансите, интеграција на CI, интеграција на тестирање на облак, игралиште за селектори, поддршка на прелистувачот.

Во оваа теза е направена споредба за тестирање на перформансите на две рамки за тестирање на два различни веб-прелистувачи, како што се Chrome headless и Mozilla firefox headless, каде што постоеше голема разлика помеѓу Cypress и TestCafe во времето на траење.

Во оваа теза се прави континуирана интеграција на две рамки за тестирање на GitLab CI / CD, исто така, интеграција на платформата за тестирање на облак, како што е LambdaTest.

Во оваа теза е направена и споредба на поддршката на прелистувачот помеѓу две рамки, исто така, направивме анализи на игралиштето за селектори.

List of Figures

List of Tables

Chapter 1. Introduction

Software testing is the most critical phase of software development. Software under test goes through various phases like test analysis, test planning, test case, test execution and bug logging/tracking. Testing is still the primary means for quality assurance today. The integration-testing phase is the most time consuming and expensive part of testing. It is common to find software project development with only 50% to 60% of effort in testing. There is a lot of research, which has been done so far to optimize the overall testing process with the intent of improving quality of software at the least amount of time.

One of the tools for end-to-end testing is Cypress, which is a framework that does not use Selenium most end-to-end tools use selenium, which is the reason why all those tools are facing the same issue. Cypress is built on an architecture whereas Selenium executes remote commands through the network Cypress runs in the same run-loop as the application. The other tool that is used to compare with Cypress is TestCafe, which is a pure node js end-to-end solution for testing web applications. It takes care of all stages such as starting the browser, running tests, gathering test results and generating test results.

This research automates a dynamic web application – WeOffer. WeOffer is chosen as it contains various dynamic elements and rendering, and it is created only to demonstrate the difference between two testing frameworks such as TestCafe and Cypress.

The goals of this research are as follows:

- To create an automation script using TestCafe and Cypress.
- To develop a regression automation suite for WeOffer main business flow which are customer's, admin's and service creator's account and order checkout flow.
- To compare the test execution time between TestCafe and Cypress.
- To compare the test efficiency and test coverage between TestCafe and Cypress.
- To compare the difficulty of writing tests between TestCafe and Cypress.

There are several environments for the execution of the E2E automated tests. By simulating the user flow from start to finish, the completion of this testing will not only validate the system under test but will also ensure that all other systems work and behave as expected. It should also be noted that with E2E tests, we do not need to check all possible scenarios. This is because much of the test coverage will already have been done with the unit tests. The idea here is that we want to check that those units all work together as they should, as an integrated user flow. Therefore, before we proceed to develop E2E tests, we need to choose an appropriate framework that will satisfy our needs. In the remainder of the article, let us look at the JavaScript-based testing frameworks and wrappers. The reason for focusing on JavaScript is that most companies nowadays are making a shift-left move for testing, i.e. moving left in the project timeline and performing tests earlier in the development lifecycle. Therefore, developers can develop the E2E tests as part of their development practices using the language they are very familiar with.

Problem statement

Choosing the right e2e framework for testing is a bit difficult since there are some features, which are supported by one of them and are not supported by the other.

The main thing that developers notice when they look for a framework tool to integrate is the web browsers that these testing frameworks support, i.e. if they support javascript or typesrcipt.. In this thesis, a web application has been used to demonstrate the integration of both end-to-end testing frameworks, such as Cypress and TestCafe, and for each of them it has been explained in details what features they support, writing down the advantages and disadvantages and also the cloud integration of the created end-to-end tests to CI.

Research Field

The focus of this project centers on comparing and demonstrating the advantages and disadvantages of two end-to-end testing frameworks such as Cypress and TestCafe on a web application built with reactJS library and firebase.

There are many front-end application development frameworks and libraries out there for developing a web application. One of them is ReactJs library which, for the time being, is one the most recent web technologies. It focuses on the view part of the MVC pattern and is being widely adopted for big scale application development. First, it has been developed by Facebook for their internal use but since it has proved an efficient and fast library compared to other technologies, they made it an open source. When it comes to dealing with large amounts of data and users, it has been quite successful in providing better user experiences. Alongside Facebook, some other big organizations and applications are also using ReactJS and React Native for their development. Instagram, Netflix, Airbnb are a few of the big names serving smoothly enormous numbers of users worldwide. Those big names prove that ReactJS is serving them quite well. However, as we know, an application without tests is not preferable and for that part, we have chosen to compare two end-to-end testing frameworks, which would help that application in the long term.

E2E testing is a technique that tests entire applications from the beginning to the end to ensure the application flow behaves as expected. The main purpose of the end-to-end (E2E) testing is to test the application from the end user's experience simulating the real user scenarios and validating the system under test and its component integration and data integrity. By end-to-end (e2e) tests many major risks can be avoided. Software systems nowadays are complex and interconnected with numerous subsystems. If any of these subsystems fails, the whole system could crash.

E2E testing is still the primary means for quality assurance today. However, in practice, integration testing is often the most time consuming and expensive part of testing. E2E testing is a testing methodology to test an application flow from start to end.

Aim of the research

The primary aim of this project is to present a freelancing web application, whereupon we will demonstrate two testing frameworks such as Cypress and TestCafe[11]. We also aim to present a

consolidated view of the challenges while implementing e2e testing frameworks and reveal which one of these has better performance on a real life application. In other words, we will point out the advantages and disadvantages of these two testing [12] frameworks.

The following are the constructive (concrete) steps to achieve the aim:

- Examining the benefits of integrating E2E tests in a project.
- Identifying challenges related to the integration of the E2E testing framework.
- Identifying the advantages and disadvantages of the Cypress testing framework.
- Identifying the advantages and disadvantages of the TestCafe testing framework.
- Identifying which is the most promising framework for our web application, TestCafe or Cypress.
- Identifying which framework performs better in our application, TestCafe or Cypress.
- Identifying the differences between open-source TestCafe and TestCafe studio.
- Identifying web and mobile browsers, which support TestCafe and Cypress.

Importance of thesis

End to end testing is a very common testing methodology where the objective is to test how an application works by checking the flow from start to end. Not only the application flow under develop environment is tested, but the tester also has to check how it behaves once integrated with the external interface. The importance of thesis is to get the best of two testing frameworks, to get the pros and cons of both testing frameworks and to ensure which one is more suitable for integration on a real-life project.

By using written tests on javascript, we will introduce the challenges that we will face from the beginning to the end of the integrating and testing process on both frameworks.

Hypotheses

Based on the reviewed literature and the defined boundaries, we formulated our primary research questions to narrow the field of investigation further:

- RQ1: Does Cypress or TestCafe work on CI provider?
- RQ2: Does Cypress or TestCafe require us to make changes on the existing code in order to create automated tests?
- RQ3: Are these two frameworks, Cypress and TestCafe, open source?
- RQ4: What are the benefits of integrating E2E tests on web applications?
- RQ5: Does it cost more by integrating E2E tests and is it worth it?

The following hypotheses are intended to highlight different aspects using reactjs combo with firebase. They provide structure and detail to the enquired topic by serving as more particular “implementations” of research questions. Each hypothesis is followed by the rationale, as to why we chose to create the respective hypotheses.

H1: Cypress provides a better learning curve.

H2: TestCafe offers a wide range of features compared to Cypress.

H3: In the frame of application, Cypress is faster.

Structure of thesis

Chapter 1. Introduction. This chapter provides an introduction of the research and sets the research within a context. It summarizes the problems, importance, the reasons that have led to the development of the thesis and gives an overview of the aim of the research, the research field and the hypothesis raised in this thesis.

Chapter 2. Literature Review. In this chapter, the end-to-end testing concept is briefly explained along with the frameworks of end-to-end testing. It is an overview of different testing frameworks and different research methodologies.

Chapter 3. Research Methodology. This chapter illustrates the way research has been conducted by presenting the steps that need to be undertaken to be able to compare Cypress and Testcafe testing frameworks. General methods to analyze the two testing frameworks have also been mentioned here.

Chapter 4. Implementation. In this chapter, the steps that have been undertaken to obtain the results of running Cypress and TestCafe tests have been explained in more details. Each step has been described, and then a summary statistic of the collected data has been given.

Chapter 5. Results and discussion. This chapter presents the results achieved in this thesis. The results have been shown and discussed. The author's findings based on the raised hypothesis have been discussed in detail.

Chapter 6. Recommendations. This chapter discusses the author's recommendations in terms of which tool offers better documentation, hence which framework to use.

Chapter 7. Conclusion. Chapter 7 summarizes the importance of the topic and presents the conclusions from the main findings and the results achieved in this thesis. The main results and findings have been summarized.

Bibliography

Chapter 2. Literature Review

There has been enormous work done in the area of end-to-end testing frameworks but scientific papers about the Cypress and TestCafe testing frameworks are lacking, since these are new end to end testing frameworks. Several researches regarding the most widely used end-to-end testing frameworks have been presented as well as comparative studies targeting the advantages and limitations of end-to-end testing frameworks.

End to end testing is a technique used to test whether the flow of an application right from the start is behaving as expected. In simple terms, end-to-end testing is a methodology to test an application from start to end.

The main purpose of end-to-end testing is to test the complete application flow, mimic the actual production scenario, and test application integrity with its interface. There is a traditional way of E2E testing, which is is usually performed in an application using selenium (JAVA). It covers the end-to-end of different applications flow in staging environment used for regression testing before release.

E2E tests are written and executed by a quality analysis (QA) team and if there [11] are any issues, they report the bug. There has been a lot of discussion about using JS and not Java. Currently, the QA process involves E2E testing with selenium (JAVA). However, since frontend-development is mostly based on JS frameworks such as ReactJS, Angular, Vue, etc, there is a need to identify a JS based framework so that the developers can write E2E tests along with development.

Developers can write end-to-end tests for a feature implemented along with unit tests in the development phase.

Benefits of E2E tests written by developers:

- Payload regression changes: E2E tests give us a bigger picture when multiple payloads are involved across the feature end-to-end.
- Visual regression changes: E2E tests can report that the latest design changes are failing in Chrome but are passing in Safari and Firefox.

Available JS frameworks for E2E testing:

- WebdriverJS
- Protractor
- WebDriverIO
- NightWatchJS
- Cypress
- TestCafe

In the image below, details for each of JS testing frameworks have been provided:

Categories	WebdriverJS	Protractor	WebdriverIO	NightwatchJS	Cypress	TestCafe
Architecture	Implementation of W3C web driver JSON wire protocol	Wrapper around WebdriverJs	Custom implementation of W3C web driver JSON wire protocol	Custom implementation of W3C web driver JSON wire protocol	Interacts directly with the browser	Interacts directly with the browser
Selenium-based	Yes	Yes	Yes	Yes	No	No
Supported Browsers	Edge, Chrome, Safari, Firefox, Opera, IE	Edge, Chrome, Safari, Firefox, Opera, IE	Edge, Chrome, Safari, Firefox, Opera, IE	Edge, Chrome, Safari, Firefox, Opera, IE	Chrome, Electron	Edge, Chrome, Safari, Firefox, Opera, IE
Inbuilt test runner	Available	Available	Available	Available	Available	Available
Supported testing frameworks	Jasmine, Mocha, Cucumber	Jasmine, Mocha, Cucumber	Jasmine, Mocha, Cucumber	Mocha, Inbuilt framework	Custom implementation	Custom implementation
Parallel execution	Supported	Supported	Supported	Supported	Supported	Supported
Cloud Execution - Sauce Labs, BrowserStack, Testing bot	Supported	Supported	Supported	Supported	N/A	Supported
Mobile Support (APPIUM)	Supports mobile browsers	Supports mobile browsers	Supports mobile browsers	Supports mobile browsers	No	Supports mobile browsers
CI	Supported	Supported	Supported	Supported	Supported	Supported
Retrying flaky tests	Possible	Possible	Possible	Possible	Possible	Possible
TS support	Yes	Yes	Yes	No	Yes	Yes

Figure 1 Details for each of JS testing frameworks

Selenium-based frameworks use web driver approaches to interact with the browsers whereas the non-selenium-based frameworks can interact with the browser directly. This is a major win for the non-selenium-based frameworks over the former as the overhead to install drivers is not there.

TestCafe is a pure node-js end-to-end solution for testing web applications. It takes care of all stages such as starting the browser, running tests, gathering test results and generating results.

Related Work

In his paper, the author Fransiskus Anindita Kristiawan Pramana Gentur Sutapa “Review of Automated Testing Approach for Software Regression Testing” discuses that the most important part of software development life-cycle is software testing, one of which is [1] regression testing. According to the author, this method is not efficient because it is time-consuming, not reusable and prone to errors. The results of his research show that the automated testing approach is suitable to enhance the regression testing with some plausible options of tools such as Selenium, SAHI and robot framework. The author further concludes that the parallel execution method is considered as a promising choice to conduct the most efficient testing process.

Authors Jyotsna, Mukul Varshney, Shivani Garg, Abha Kiran Rajpoot on their research paper “Automated Testing: An Edge Over Manual Software Testing” explain that software testing is a process of finding errors while executing a program so that we get zero-defect software. [2] They also claim that the software testing is aimed at evaluating the capability or usability of a program. Software testing is an important means of accessing quality of software. According to them, manual testing involves a lot of effort measured in person per month. These efforts can be reduced by using the automated testing with specific tools.

In his paper, “E-Commerce testing framework”,, Denislav Lefterov presents the development of an automation-based testing framework which supports and helps to implement easily new tests related to Web platforms analogous to e-commerce applications. [3] Denislav Lefterov created scripts that represents automated acceptance, functional and non-functional tests in which the Page Object pattern is used to separate the tests into individual fragments and subsequently to call them in a different order according to the test requirements and business logic. In addition, tests can be numerous, but the elements visualized in a given functionality are similar in order to reuse the key iterations.

Da Zhang in his research paper “End to end testing [4] using integrated tools” describes an environment for testing with Selenium and Nagios, as well as customization that he develops to incorporate Selenium script into a Nagios executable library. In his research paper, he explains how he combined the Nagios monitoring tool and Selenium testing tool to realize end-to-end testing using integrated tools.

“Automated Software Testing Framework ‘STASSY’” by author Denislav Lefterof presents the idea to develop an automation-based testing framework “Stassy” - System Table Testing which supports and helps to implement easily new tests related to web applications.[5]

This implementation includes the incoming structure: Object repository, functional libraries, global variables and constants, data provider, test scripts, configuration files, recovery scenarios etc. with the advantages of page object pattern and using following technologies: Java programming language, JUnit, TestNG extended libraries. The created scripts represent automated acceptance, functional and non-functional tests in which page object is used to separate tests into individual fragments and subsequently call them in a different order according to the automation requirements and business logic. According to the author, the proposed idea of the framework reduces the required time to write and run test cases and increases their pass percentage rate by covering all the main steps in applications of this kind. It also reduces vulnerable workload of testers.

Authors W.T. Tsai, Xiaoying Bai, Ray Paul, Weiguang Shao, Vishal Agarwal in their research paper “End-To-End Integration Design” present a systematic E2E testing design [6] approach where they include test specification, test case generation and tool support. The authors’ approach has the following characteristics: It uses both black-box and white-box testing techniques, which provides sufficient information for functional test case design, coverage analysis, result analysis, defects identification and software evaluation. In addition, this approach supports remote project management and distributed collaboration so that engineers and project managers can work together via the Internet.

In their case study “Evaluation of an Automated Testing Framework”, authors Abel Mendez-Porras, Jorge Alfaro-Velasco, Alexandra Martinez reveal that the behavior of mobile applications [7] is affected by different types of user events: events produced through GUI events generated by the device hardware platform and events from the internet. These types of events are likely to generate bugs in mobile applications. In addition, the use of historical bug information to find bugs in mobile applications is complex because it requires storing information about all the bugs detected each time that other applications are tested. As for difficulties, they listed the following: The first difficulty is to know when there is enough information to infer, when to enter a user interaction during the testing process; the

second difficulty is that it is necessary to obtain applications that are being developed and have not yet been tested to increase the probability of storing bugs associated with user interface features. In their research, they have used a top-down technique to design the automated testing framework. The advantage of using this technique, according to them, was that through a formal process, the architecture of framework was designed to foster further research progress. This architecture was organized into four components: an exploration environment, an inference system, a bug analyzer, and a test storage database. In addition, in the future they plan to develop their own extrapolation environment because they will need more control over the event-sequences automatically entered in the applications under test.

In “Comparative review of the literature of automated testing tools”, authors [8] Anand Singh Gadwal and Dr. Lalju Prasa selected twelve tools that are frequently used in automation testing of web-based applications. They performed a comparative analysis on the basis of their characteristics. According to them, in selecting tools, if the project cost is to be given higher consideration, open-source tools such as selenium, are a better option. In their research, they conclude that thorough research is needed to improve the quality of tools in various aspects. However, there is no single solution available by which we can achieve complete automation testing. However, tools can be used in integration to accomplish testing requirements.

According to authors Fatini Mobaraya and Shahid Ali, in their research paper “Technical Analysis of Selenium and Cypress as Functional Automation Framework for Modern [9] Web Application Testing“, Selenium framework is undeniably a powerful tool due to its huge community and support as it has been on the market for many years. However, Cypress also gives a promising view of how the future of the automating testing will be. It significantly eases and simplifies the automation configuration processes and produces a better and cleaner code. With the right number of resources and support, Cypress can be used to achieve much more. Their research relies heavily on stackoverflow, github and Cypress official page to develop the automation scripts in Cypress. It might not be the best industry practice yet, as it is conducted on the basis of self-study.

Chapter 3. Research Methodology

The work carried out in this thesis has to do with analyzing the two end-to-end testing frameworks, Cypress and TestCafe, and understanding the advantages, disadvantages and the difficulty level upon their first-time implementation on a web application built in reactjs and firebase. In addition, the integration of an end-to-end test in CI is part of our activities.

ReactJs

React is an open-source JavaScript library that is used for building user interfaces specifically for single-page applications. It is used for handling the view layer for web and mobile apps. React also allows us to create reusable UI components. React was first created by Jordan Walke, a software engineer working for Facebook. React was first deployed on [10fi] Facebook's newsfeed in 2011 and on Instagram.com in 2012.

React allows developers to create large web applications that can change data, without reloading the page. The main purpose of React is to be fast, scalable, and simple. It works only on user interfaces in the application. This corresponds to the view in the MVC template. It can be used with a combination of other JavaScript libraries or frameworks, such as Angular JS in MVC.

React JS is also called simply React or React.js.

Now, the main question is why one should use React. There are so many open-source platforms for making the front-end web application development easier, like Angular. Let us take a quick look at the benefits of React over other competitive technologies or frameworks. With the front-end world-changing daily, it is hard to devote time to learning a new framework – especially when that framework could ultimately become a dead end. So, if you are looking for the next best thing but you are feeling a little bit lost in the framework jungle, I suggest checking out React.

1. Simplicity

ReactJS is just simpler to grasp right away. The component-based approach, well-defined lifecycle, and use of just plain JavaScript make React very simple to learn, build a professional web (and mobile applications), and support it. React uses a special syntax called JSX, which allows you to mix HTML with JavaScript. This is not a requirement; developers can still write in plain JavaScript but JSX is much easier to use.

2. Easy to learn

Anyone with some basic previous knowledge in programming can easily understand React while Angular and Ember are referred to as 'Domain-specific Language', implying that it is difficult to learn them. To react, you just need basic knowledge of CSS and HTML.

3. Native Approach

React can be used to create mobile applications (React Native). Moreover, React is a diehard fan of reusability, meaning extensive code reusability is supported. Therefore, we can make IOS, Android and Web applications at the same time.

4. Data Binding

React uses one-way data binding and an application architecture called Flux. It controls the flow of data to components through one control point – the dispatcher. It is easier to debug self-contained components of large ReactJS apps.

5. Performance

React does not offer any concept of a built-in container for dependency. You can use Browserify, Require JS, ECMAScript 6 - modules that we can use via Babel, ReactJS-di to inject dependencies automatically.

6. Testability

ReactJS applications are super easy to test. React views can be treated as functions of the state, so we can manipulate with the state we pass to the ReactJS view and take a look at the output and triggered actions, events, functions, etc.JSX

In React, instead of using regular JavaScript for templating, JSX is sused. JSX is a simple JavaScript that allows HTML quoting and uses these HTML tag syntaxes to render subcomponents. HTML syntax is processed into JavaScript calls of React Framework. We can also write in pure old JavaScript.

Single-Way data flow

In React, a set of immutable values are passed to the component's renderer as properties in its HTML tags. The component cannot directly modify any properties but can pass a call back function with the help of which we can do modifications. This complete process is known as "properties flow down; actions flow up".

Firebase

Firebase is a Backend-as-a-Service — BaaS — that started as an YC11 startup and grew up into a next-generation app-development platform on Google Cloud Platform.

Firebase frees developers to focus on designing fantastic user experiences. You do not need to manage servers. You do not need to write APIs. Firebase is your server, your API and your datastore, all written so generically that you can modify it to suit most needs. You will occasionally need to use other bits of the Google Cloud for your advanced applications. Firebase cannot be everything to everybody. However, it gets close.

1. Realtime Databases

Real-time data is the way of the future. Nothing compares to it.

Most databases require you to make HTTP calls to get and synchronize your data. Most databases give you data only when you ask for it.

When you connect your app to Firebase, you are not connecting through normal HTTP. You are connecting through a WebSocket. WebSockets are much, much faster than HTTP. You do not have to make individual WebSocket calls, because one socket connection is plenty. All of your data syncs automatically through that single WebSocket as fast as your client's network can carry it.

Firebase sends you new data as soon [18] as it is updated. When your client saves a change to the data, all connected clients receive the updated data almost instantly.

2. File Storage

Firebase Storage provides a simple way to save binary files — most often images, but it could be anything — to Google Cloud Storage directly from the client!

Firebase Storage has its own system of security rules to protect your GCloud bucket from the masses, while granting detailed write privileges to your authenticated clients.

3. Authentication

Firebase auth has a built-in email/password authentication system. It also supports OAuth2 for Google, Facebook, Twitter and GitHub. We will focus on email/password authentication for the most part. Firebase's OAuth2 system is well documented and mostly copy/pasted.

If you have ever written an authentication system, let us commiserate for a moment. Custom authentication is terrible. I will never write an auth system again for as long as I live. I fell in love with Firebase Auth at first sight, and the flame has never wavered. Sometimes I get frustrated. Sometimes we fight. But I never forget the cold, dark abyss of a custom auth system. I count my blessings. 4. Hosting

Firebase includes an easy-to-use hosting service for all your static files. It serves them from a global CDN with HTTP/2.

And to make your development particularly painless, Firebase hosting utilizes Superstatic, which you can run locally for all your testing.

The BrowserSync + Superstatic development environment is slick. BrowserSync handles reloading your development app across all connected devices and Superstatic replicates Firebase hosting locally in such a way that you can deploy straight to Firebase for production use.

5. Fully Featured App Platform

The Firebase team has integrated a bunch of new and existing Google products with Firebase. I do not plan to cover these features in detail quite yet…

A bunch of these features applies to iOS and Android but not to web.

- Remote Config
- Test Lab
- Crash
- Notifications
- Dynamic Links
- AdMob

6. Firebase Pros & Cons

Pros

- Email & password, Google, Facebook, and Github authentication
- Realtime data
- Ready-made api
- Built in security at the data node level
- File storage backed by Google Cloud Storage
- Static file hosting
- Treat data as streams to build highly scalable applications
- Do not worry about your infrastructure!

Cons

- Limited query abilities due to Firebase's data stream model
- Traditional relational data models are not applicable to NoSQL; therefore, your SQL chops will not transfer

No on-premise installation

Selected tools

This research uses two automation tools to develop the automation scripts which are TestCafe and Cypress. TestCafe is selected as it is one of the open sourced tools in automating web application while Cypress is selected as it offers a new way in automating modern web application. Cypress is initially a primary work of Brian Mann, a developer who felt testing dynamic websites has been tedious due to inefficient automation test execution.

He then conducted a survey on the challenges automation developers faced while testing current web application. Based on the collected data, automation developers expressed that most of the debugging time was spent on synchronizing wait with page loads, though the time should

actually be spent on writing more test scripts. Due to these concerns, Cypress was developed and founded in year 2015.

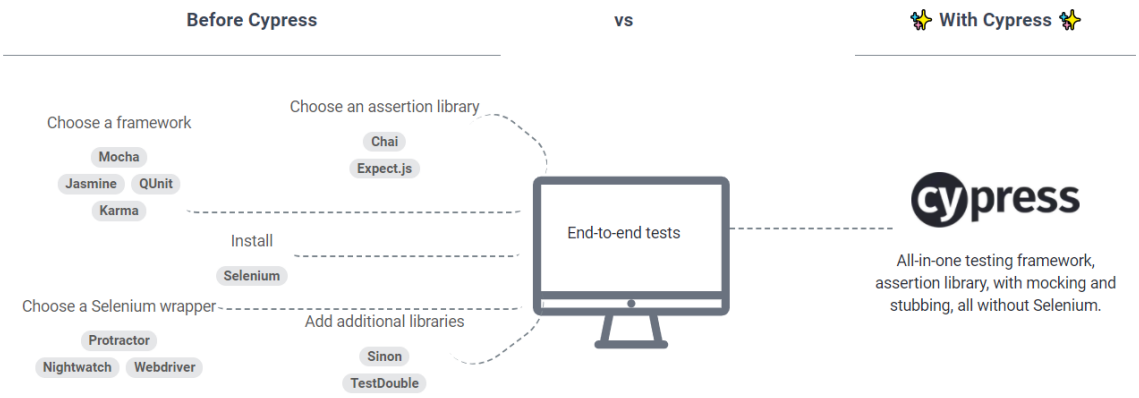

Figure 2 Benefits of Cypress

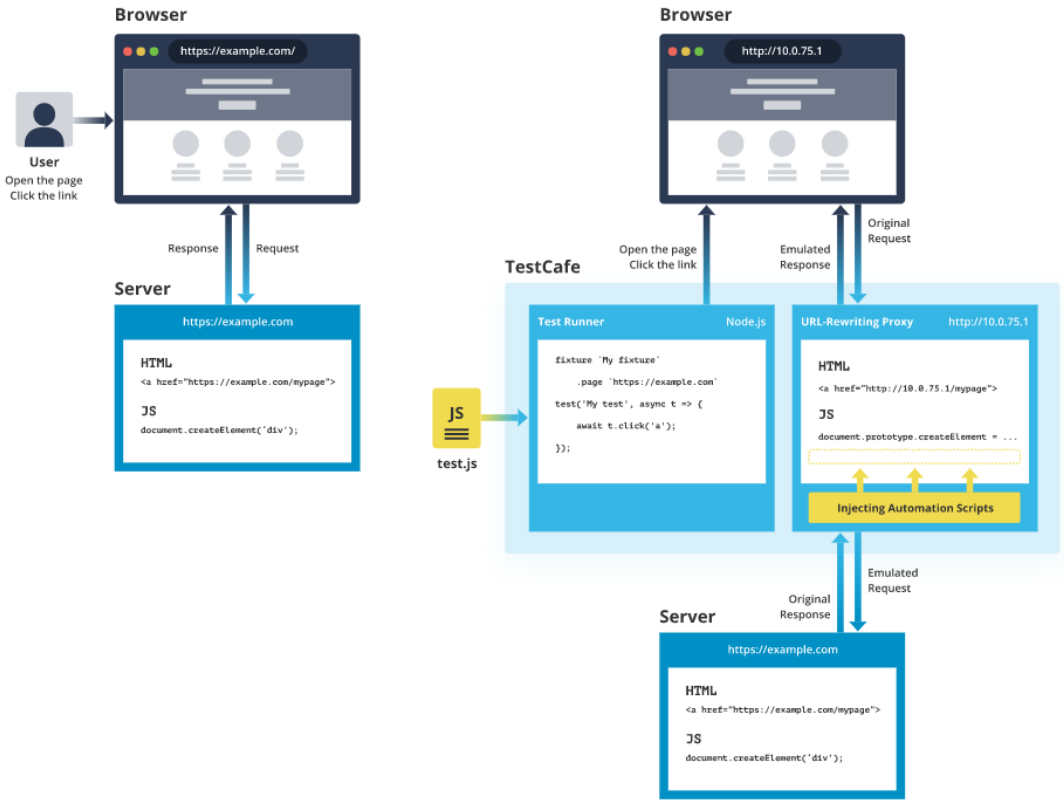

Figure 3 Benefits of Testcafe

Chapter 4. Implementation

The conducted steps were defined in the third Implementation chapter, while in this chapter there is a detailed description regarding how the steps were conducted during the implementation phase.

Project Execution

There are some requirements before the implementation of the E2E tests such as system requirements. Cypress is a desktop application that is installed on your machine and supports the following operating systems:

- macOS 10.9 and above (64bit only)
- Linux Ubuntu 12.04 and above
- Windows 7 and above

In this project, we have used npm and for this Cypress supports Node.js 10, 12 and above versions. Installation of Cypress is very easy npm install Cypress --save-dev this command will install Cypress locally as a dev dependency for your project. Best practice to install Cypress is with npm also this approach is recommended by Cypress documentation itself because Cypress is versioned like any other dependency and it simplifies running Cypress in Continuous integration.

Opening Cypress

npx Cypress open but in our case we have edited package json script and for that reason in this project we can open it with a simple command like: npm run Cypress-headless which will run the Cypress headless which means hides the browser instead showing the browsers, and by default Cypress headless run in electron.

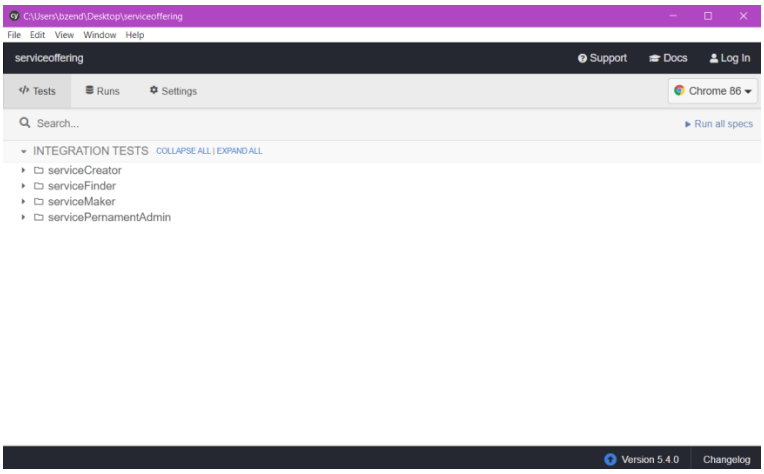

Figure 4 Cypress Runner

This is the interface of Cypress when you run it with the command npm run Cypress which gives us a user friendly interface where all our test scenarios folder are listed.

In addition, Cypress gives you a simple way to see the project settings by just clicking the settings link, which opens a new tab like the picture below:

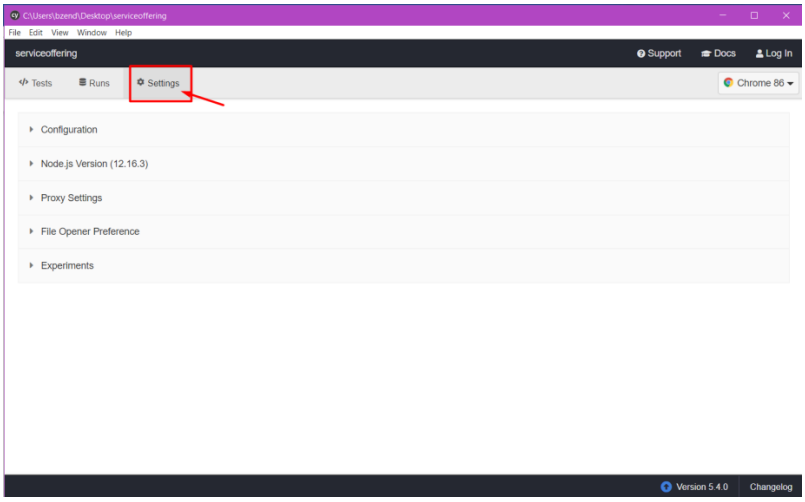

Figure 5 Cypress Settings

Here you can edit project configuration, node.js version, proxy settings, file opener preferences and experiments.

Opening TestCafe

You can install TestCafe from npm globally or locally in your project.

Local installation should be preferred for continuous integration systems, Node.js applications and other scenarios where global installation is not required.

Local installation makes your project setup easier: npm install executed in the project directory installs all dependencies including TestCafe.

Different projects can use different local TestCafe versions.

You can also run TestCcafe without prior installation. However, this is not recommended for regular use.

Global installation of TestCafe: npm install -g TestCafe

Local installation of TestCafe: npm install –save-dev TestCafe

Test Structure

By collecting information about different open source projects on GitHub the test structure for TestCafe is like below:

Each test has been separated in functions, by this we ensure that we can reuse each executable test that has been written.

```
export const basicNavigation = {
  url: 'http://localhost:3000',
  getCurrentLocation: ClientFunction(() => document.location.href),
};      You, 2 months ago • update latest changes e2e

export const userCredentials = {
  registerButton: Selector('#register span').withText('Register'),
  emailInput: Selector('#root').find('[name="email"]'),
  nameInput: Selector('#root').find('[name="fullName"]'),
  avatarInput: Selector('#root').find('[name="avatar"]'),
  passwordInput: Selector('#root').find('[name="password"]'),
  confirmPasswordInput: Selector('#root').find('[name="passwordConfirmation"]'),
  registerUserButton: Selector('button').withText('Register'),
  loginButton: Selector('#navbar-menu a').withText('Login'),
  signInButton: Selector('#root button').withText('Sign In'),
  logoutButton: Selector('#navbar-menu span').withText('Logout'),
  emailText: `testcafeUser${Math.floor(
    Math.random() * 1000000 + 1
  )}@gmail.com`,
  userName: 'TestcafeUser',
  serficeFinderUser: `testcafeFinder${Math.floor(
    Math.random() * 1000000 + 1
  )}@gmail.com`,
  pngImage:
    'https://cdn.pixabay.com/photo/2016/08/08/09/17/avatar-1577909_1280.png',
  userPassword: `B123456b`,
};
```

Figure 6 Selectors

User credentials are the selectors for the login screen and logout screen such as password, username etc. In this way, we have implemented for all the screens and we have implemented it in Cypress in the same way.

```
import { registrationForm, selectors } from '../utils/selectors';
import Config from '../config';
const { username, serviceFinder, admin } = Config;
export const NewServiceCreatorAccount = () => {
  cy.get(selectors.mainMenu).click();
  cy.get(registrationForm.registerButton).click();
  cy.get(registrationForm.emailInput)
    .type(username)
    .should('have.value', username);
  cy.get(registrationForm.userNameInput)
    .type(registrationForm.userName)
    .should('have.value', registrationForm.userName);
  cy.get(registrationForm.avatarInput)
    .type(registrationForm.avatarLink)
    .should('have.value', registrationForm.avatarLink);
  cy.get(registrationForm.passwordInput).type(
    registrationForm.passwordInputText,
    {
      log: false,
    }
  );
  cy.get(registrationForm.passwordConfirmationInput).type(
    registrationForm.passwordConfirmationInputText,
    {
      log: false,
    }
  );
  cy.get(registrationForm.authButtons).should('have.text', 'Register').click();
  cy.url().should('include', '/');
};
```

Figure 7 Service Creator Selector

Test scenarios

Test scenarios of this research are defined as in Table 1 below while Table 2 shows the detailed test steps for test execution. As mentioned earlier, this research will automate the two key functionalities of WeOffer, the service creator and service finder. Thus, the following test scenarios are derived to cover the functionality of each feature.

Table 1 Test Scenarios

Service Creator	**Successfully creates new user account**
	Successfully navigate to FAQ screen
	Create new service and checkout my services page
	Creator successfully accepts offer and signs out
	Service creator joins the collaboration and sent message
Service Finder	Successfully creates new user service finder account
	Successfully sign in with new account and makes an offer
	Service finder starts collaboration and signs out
Admin	Admin Deletes Account

Table 2 Executed test steps and results for successfully creates new user account

Test Scenarios	Test Steps	Test Data	Expected Result	Actual Result	Status
Successfully creates new user account	1. Open browser 2. Navigate to we offer url 3. Click hamburger menu icon 4. Click username input. 5. Type random username 6. Click password input 7. Click register button. 8. Logout	Localhost:3000 Email: Cypressrandom email that is generated. password: B123456b	Service creator successfully create new account	As expected	Pass

Table 3 Executed test steps and results for Successfully navigate to FAQ screen

Test Scenarios	Test Steps	Test Data	Expected Result	Actual Result	Status
Successfully navigate to FAQ screen	1. Open browser 2. Navigate to we offer url/ faq.	localhost:3000/faq	Successfully navigate to faq screen	As expected	Pass

Table 4 Executed test steps and results for Create new service and checkout my services page

Test Scenarios	Test Steps	Test Data	Expected Result	Actual Result	Status
Create new service and checkout my services page	1. Open browser 2. Navigate to we offer url 3. Sign is as user creator with the newly created account. 4. Navigate to my service page. 5. Create new service. 6. Navigate to home page 7. Logout	Service creator email and password, Service title, service price, service url.	Successfully creates new service and navigates to home page.	As expected	Pass

Table 5 Executed test steps and results for Creator successfully accepts offer and signs out

Test Scenarios	Test Steps	Test Data	Expected Result	Actual Result	Status
Creator successfully accepts offer and signs out	1. Open browser 2. Navigate to we offer url 3. Sign in 4. Navigate to offer screen 5. Accept offer	-	Successfully accepts offer and signs out	As expected	Pass

	6. Sign out				

Table 6 Executed test steps and results for Service creator joins the collaboration and sent message

Test Scenarios	Test Steps	Test Data	Expected Result	Actual Result	Status
Service creator joins the collaboration and sent message	1. Open browser 2. Navigate to we offer url 3. Sign in 4. Navigate to offer screen 5. Join collaboration 6. Send message 7. Logout	- message	Successfully creator joins the collaboration and sent message	As expected	Pass

Table 7 Executed test steps and results for successfully creates new user service finder account

Test Scenarios	Test Steps	Test Data	Expected Result	Actual Result	Status
Successfully creates new user service finder account	1. Open browser 2. Navigate to we offer url 3. Click hamburger menu icon 4. Click	Localhost:3000 Email: Cypressrandomfinder email that is generated. password: B123456b	Successfully creates new user service finder account	As expected	Pass

	username input. 5. Type random username 6. Click password input 7. Click register button. 8. Logout				

Table 8 Executed test steps and results *Successfully sign in with new account and makes an offer*

Test Scenarios	**Test Steps**	**Test Data**	**Expected Result**	**Actual Result**	**Status**
Successfully sign in with new account and makes an offer	1. Open browser 2. Navigate to we offer url 3. Click specific service 4. Make the offer 5. Log out	Offer title, offer description and offer price	Successfully sign in with new account and makes an offer	As expected	Pass

Table 9 Executed test steps and results for Service finder starts collaboration and signs out

Test Scenarios	Test Steps	Test Data	Expected Result	Actual Result	Status
Service finder starts collaboration and signs out	1. Open browser 2. Navigate to we offer collaboration screen 3. Starts collaboration 4.Logout	-	Service finder starts collaboration and signs out	As expected	Pass

Table 10 Executed test steps and results for admin

Test Scenarios	Test Steps	Test Data	Expected Result	Actual Result	Status
Admin	1. Open browser 2. Navigate to we offer url 3. Click hamburger menu icon 4. Click username input. 5. Type random username 6. Click password input	Username and password from env variables	Successfully deletes created users	As expected	Pass

Running scripts

Custom scripts are created in order to run E2E tests. For running Cypress test we have created a bash script from where we generate new users and service title. Meanwhile, in TestCafe we generate users while running tests by the help of selectors.

runCypress.sh

You, a few seconds ago | 1 author (You)

```
PARAM=$1
NEW_UUID=$(cat /dev/urandom | tr -dc 'a-zA-Z0-9' | fold -w 32 | head -n 1)
GMAIL="@gmail.com"
TITLE="Cypress Event"
echo "Generating new users and run cypress tests"
if [[ "$*" == *run* ]]
then
    export CYPRESS_FINDER=finder$NEW_UUID$GMAIL
    export CYPRESS_TITLE=$TITLE$NEW_UUID
    npx cypress run --headless --browser firefox --env USERNAME=creator$NEW_UUID$GMAIL
else
    export CYPRESS_FINDER=finder$NEW_UUID$GMAIL
    export CYPRESS_TITLE=$TITLE$NEW_UUID
    npx cypress open --env USERNAME=creator$NEW_UUID$GMAIL

fi      You, a month ago • update scripts and create test for service creation
```

Figure 8 Cypress Custom Script

Debug

```
"scripts": {
  "start-dev": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject",
  "start": "node server.js",
  "heroku-postbuild": "npm run build",
  "e2e": "testcafe firefox tests",
  "cypress": "bash ./runCypress.sh",
  "cypress-headless": "bash ./runCypress.sh run",      You, a month ago • update scripts and create test f
  "e2e:service-offer": "testcafe firefox tests/testcafe/fixtures/serviceOffer",
  "e2e:service-finder": "testcafe firefox tests/testcafe/fixtures/serviceFinder"
},
```

Figure 9 Package Json Script

There are more than one way where you want to run E2E test, either locally or on any other environment such as production or develop(testing environment).

With npm run e2e we are able to run all the TestCafe tests except for the admin test scenarios. In this case, we need to SET our environment variables for admin email and admin password in order to run all

tests via npm run e2e command. The same method is also implemented in Cypress tests, which we run with npn run Cypress-headless command.

Difference between test scenarios

The same logic is used for both testing frameworks in order to get the best results:

1. Successfully creates new user account code in TestCafe

```
You, 2 months ago | 1 author (You)
1  import {selectors,basicNavigation} from '../../utils/selectors';
2  // import basicNavigation from '../../utils/selectors';
3  import {CreateServiceOfferUser} from '../../utils/registerUser'
4  fixture`Service Creator User And Simple Faq Navigation`.page(basicNavigation.url);
5
6  test(`Successfully creates new user account`, async (t) => {
7    await CreateServiceOfferUser(t)
8  });
9
```

Figure 10 Testcafe New User Flow

Here we call a function which is placed at Utils folder:

```
import { basicNavigation, userCredentials } from './selectors';

export const CreateServiceOfferUser = async (t) => {      You, 2 months ago • update late
  await t
    .click(userCredentials.registerButton)
    .expect(basicNavigation.getCurrentLocation())
    .contains(`${basicNavigation.url}/register`)
    .click(userCredentials.emailInput)
    .typeText(userCredentials.emailInput, userCredentials.emailText)
    .click(userCredentials.nameInput)
    .typeText(userCredentials.nameInput, userCredentials.userName)
    .click(userCredentials.avatarInput)
    .typeText(userCredentials.avatarInput, userCredentials.pngImage)
    .click(userCredentials.passwordInput)
    .typeText(userCredentials.passwordInput, userCredentials.userPassword)
    .click(userCredentials.confirmPasswordInput)
    .typeText(
      userCredentials.confirmPasswordInput,
      userCredentials.userPassword
    )
    .click(userCredentials.registerUserButton)
    .click(userCredentials.logoutButton);
};
```

Figure 11 Testcafe flow for creation of service

The same logic has been used also with Cypress scenario:

```
it('Create new service creator account', () => {
  NewServiceCreatorAccount();
  Logout();
});
```

Figure 12 Cypress create new service creator account

```
const { username, serviceFinder, admin } = config;
export const NewServiceCreatorAccount = () => {      You, a month ago • cypress integration
  cy.get(selectors.mainMenu).click();
  cy.get(registrationForm.registerButton).click();
  cy.get(registrationForm.emailInput)
    .type(username)
    .should('have.value', username);
  cy.get(registrationForm.userNameInput)
    .type(registrationForm.userName)
    .should('have.value', registrationForm.userName);
  cy.get(registrationForm.avatarInput)
    .type(registrationForm.avatarLink)
    .should('have.value', registrationForm.avatarLink);
  cy.get(registrationForm.passwordInput).type(
    registrationForm.passwordInputText,
    {
      log: false,
    }
  );
  cy.get(registrationForm.passwordConfirmationInput).type(
    registrationForm.passwordConfirmationInputText,
    {
      log: false,
    }
  );
  cy.get(registrationForm.authButtons).should('have.text', 'Register').click();
  cy.url().should('include', '/');
};
```

Figure 13 Cypress for creator logic on Utils folder

2. Navigation to faq screen:

TestCafe:

```
test(`Sucessfully navigate to FAQ screen`, async (t) => {
  await t
    .click(selectors.faqNavigation)
    .expect(basicNavigation.getCurrentLocation())
    .contains(`${basicNavigation.url}/faq`);
});
```

Figure 14 Faq Navigation Testcafe

Cypress:

```
it('Navigates to faq screen', () => {
  cy.get(selectors.faqNavigation).should('have.text', 'Faq').click();
  cy.get(selectors.mainMenu).click();
  cy.url().should('include', '/faq');
});
```

Figure 15 Faq Navigation Cypress

3. CreateServiceFinderUser

TestCafe:

```
export const CreateServiceFinderUser = async (t) => {        You, 2 months ago • update l
  await t
    .click(userCredentials.registerButton)
    .expect(basicNavigation.getCurrentLocation())
    .contains(`${basicNavigation.url}/register`)
    .click(userCredentials.emailInput)
    .typeText(userCredentials.emailInput, userCredentials.serficeFinderUser)
    .click(userCredentials.nameInput)
    .typeText(userCredentials.nameInput, userCredentials.userName)
    .click(userCredentials.avatarInput)
    .typeText(userCredentials.avatarInput, userCredentials.pngImage)
    .click(userCredentials.passwordInput)
    .typeText(userCredentials.passwordInput, userCredentials.userPassword)
    .click(userCredentials.confirmPasswordInput)
    .typeText(
      userCredentials.confirmPasswordInput,
      userCredentials.userPassword
    )
    .click(userCredentials.registerUserButton)
    .click(userCredentials.logoutButton);
};
```

Figure 16 New service creation TestCafe

Cypress:

```
export const NewServiceFinderAccount = () => {       You, a month ago • refactor testcafe tests
  cy.get(selectors.mainMenu).click();
  cy.get(registrationForm.registerButton).click();
  cy.get(registrationForm.emailInput)
    .type(serviceFinder)
    .should('have.value', serviceFinder);
  cy.get(registrationForm.userNameInput)
    .type(registrationForm.userName)
    .should('have.value', registrationForm.userName);
  cy.get(registrationForm.avatarInput)
    .type(registrationForm.avatarLink)
    .should('have.value', registrationForm.avatarLink);
  cy.get(registrationForm.passwordInput).type(
    registrationForm.passwordInputText,
    {
      log: false,
    }
  );
  cy.get(registrationForm.passwordConfirmationInput).type(
    registrationForm.passwordConfirmationInputText,
    {
      log: false,
    }
  );
  cy.get(registrationForm.authButtons).should('have.text', 'Register').click();
  cy.url().should('include', '/');
};
```

Figure 17 Cypress creation of service

4. Find desired service:

TestCafe:

```
export const FindDesiredService = async (t) => {       You, a month ago • r
  const ourService = findSpecificService.exists;
  await t
    .expect(ourService)
    .ok()
    .click(findSpecificService)
    .click(serviceSelectors.makeAnOfferButton)
    .click(serviceSelectors.noteInput)
    .typeText(serviceSelectors.noteInput, 'Im a full stack developer')
    .click(serviceSelectors.serviceTime)
    .typeText(serviceSelectors.serviceTime, '50')
    .click(serviceSelectors.saveChanges);
};
```

Figure 18 Desired Service Finder TestCafe

Cypress:

```
export const FindServiceAndSentOffer = () => {
  cy.get('h4').contains(serviceTitle).click();
  cy.get('.is-medium').should('have.text', 'Make an offer').click();
  cy.get(':nth-child(1) > .input')
    .type('Im a full stack developer')
    .should('have.value', 'Im a full stack developer');
  cy.get(':nth-child(2) > .input').type('50').should('have.value', '50');
  cy.get(
    '[style="background-color: rgb(207, 219, 49); color: white;"]'
  ).click();
};      You, a month ago • refactor testcafe tests
```

Figure 19 Desired Service Finder Cypress

5. Collaboration

TestCafe:

```
You, a month ago | 1 author (You)
import { basicNavigation } from '../../utils/selectors';
import { Login, UserLogout, ServiceCreatorLogin } from '../../utils/login';
import { NavigateToOfferTabAsCreator,NavigateToSentOfferTabAndCollaborateAsFinder, ServiceCreatorMessageCollab } from '../../utils/collabor
fixture`Communicate for collaboration`.page(basicNavigation.url);

fixture`Service Creator Service Collaboration`
  .page(basicNavigation.url)

test(`Creator successfully accepts offer and signs out`, async (t) => {
  await Login(t);
  await NavigateToOfferTabAsCreator(t);
  await UserLogout(t);
});

test('Service finder starts collaboration and signs out', async (t) => {
  await ServiceCreatorLogin(t);
  await  NavigateToSentOfferTabAndCollaborateAsFinder(t)
  await UserLogout(t)
});

test('Service creator joins the collaboration and sent message', async (t) => {
  await Login(t);
  await ServiceCreatorMessageCollab(t)
  await UserLogout(t)
});      You, a month ago • finish all testcafe tests integration
```

Figure 20 Collaboration Testcafe

Cypress:

```
You, a month ago | 1 author (You)
import {
  ServiceCreatorAccountSignIn,
  Logout,
  ServiceFinderAccountSignIn,
} from '../../utils/auth';
import { selectors } from '../../utils/selectors';
import {
  SignInAndAcceptCollaborationAsCreator,
  SignInAsCreatorAndSendMessage,
  SignInAndStartCollaborationAsFinder,
} from '../../utils/general';
context('Service Collaboration Tests', () => {
  beforeEach(() => {
    cy.visit(selectors.baseUrl);
  });
  it('Accpet offer and sign out', () => {
    ServiceCreatorAccountSignIn();
    SignInAndAcceptCollaborationAsCreator();
    Logout();       You, a month ago • refactor testcafe tests
  });

  it('Logs in as service finder and start collaboration', () => {
    ServiceFinderAccountSignIn();
    SignInAndStartCollaborationAsFinder();
    Logout();
  });
  it('Logs in as service creator and send first message', () => {
    ServiceCreatorAccountSignIn();
    SignInAsCreatorAndSendMessage();
    Logout();
  });
});
```

Figure 21 Collaboration Cypress

Difference between Cypress and TestCafe

Both of these frameworks are open source written in javascripts. They run partially in the browser and partially in node.js. Cypress and TestCafe are transparently retry assertions, which eliminates a lot of the flakiness associated with Selenium based tests.

Both frameworks are under heavy development with very responsive developers and similar sized communities.

Cypress framework uses Mocha for running tests with Chai for assertions and sinon for mocking. By this, it allows developers to feel like they are in their home. TestCafe uses its own test runner, which is a bit strange, because it refers to a group of tests as a fixture. In addition, TestCafe makes you to use promises async and await to manage execution.

There are different ways on how these two frameworks serve the test site. TestCafe works by serving via a proxy server, where the server injects scripts into the webpage, which can inspect and control all elements on page. It also supports the interaction between native alerts, which means TestCafe also works in mobile devices and cloud services like Browserstack and Lambdatest. On the other hand, Cypress works by controlling the web browser via its proprietary automation APIs. Cypress runs test code in the web browser process whereas TestCafe runs it in node, which means Cypress has access to the real DOM elements. In TestCafe communication between tests and DOM must be serialized.

TestCafe tests run in Node, and by this you can call out to parts of your node server application directly from the tests. This is useful and helps clean database fixtures or events starting and stopping the test server. In Cypress, we are limited to communication with the app via HTTP or executing shell commands.

Selector difference between Cypress and TestCafe

TestCafe uses standard CSS selectors to find elements, meanwhile Cypress uses jQuery selectors which have some extra capabilities such as :first, :parent. On the other side, TestCafe has framework specific extensions for React, Angularm Aurelia, VUE that let you use component names as selectors.

TestCafe offers automation selector finder. TestCafe-Studio is a cross-platform IDE for end-to-end web testing. You can record tests visually within your favorite browser, edit scripts in its IDE-like interface, and execute tests across different browsers, platforms, and devices. This platform is free only for 30 days; meanwhile Cypress offers you selector finder for free and is more efficient.

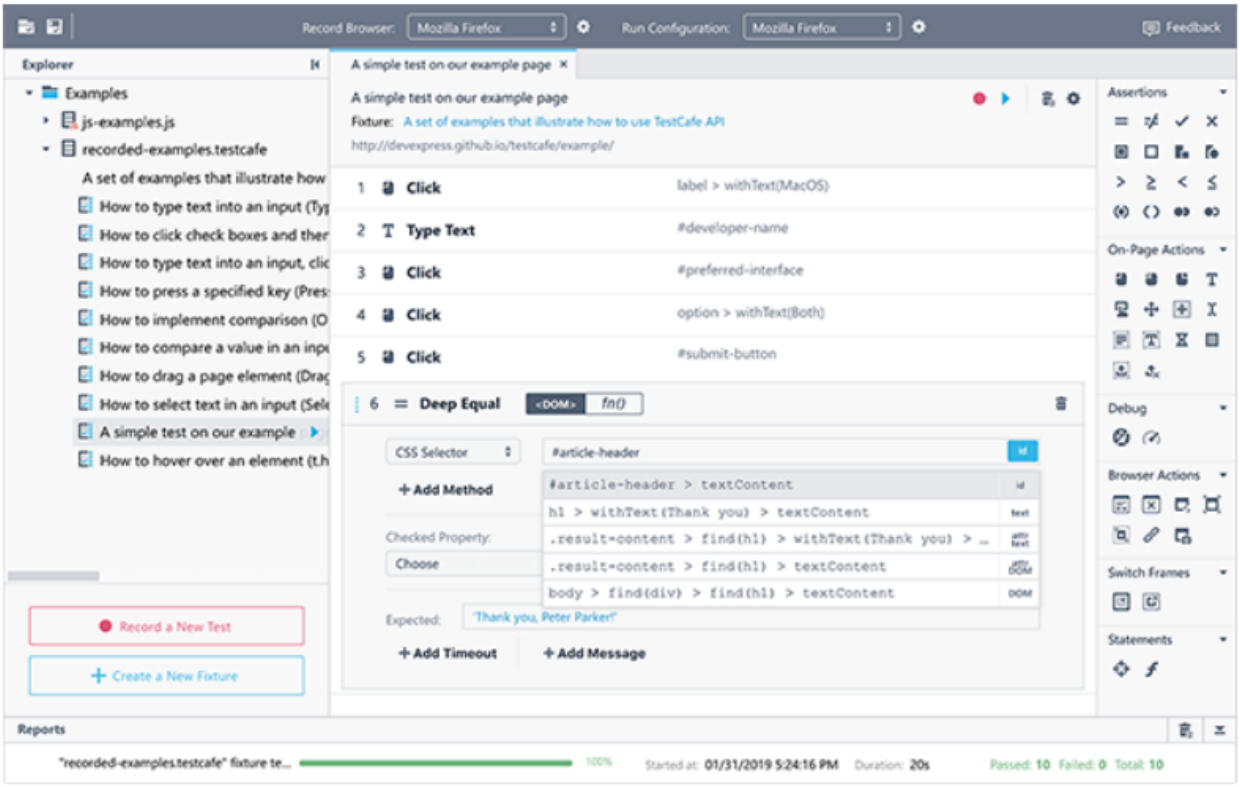

Figure 22 TestCafe Studio

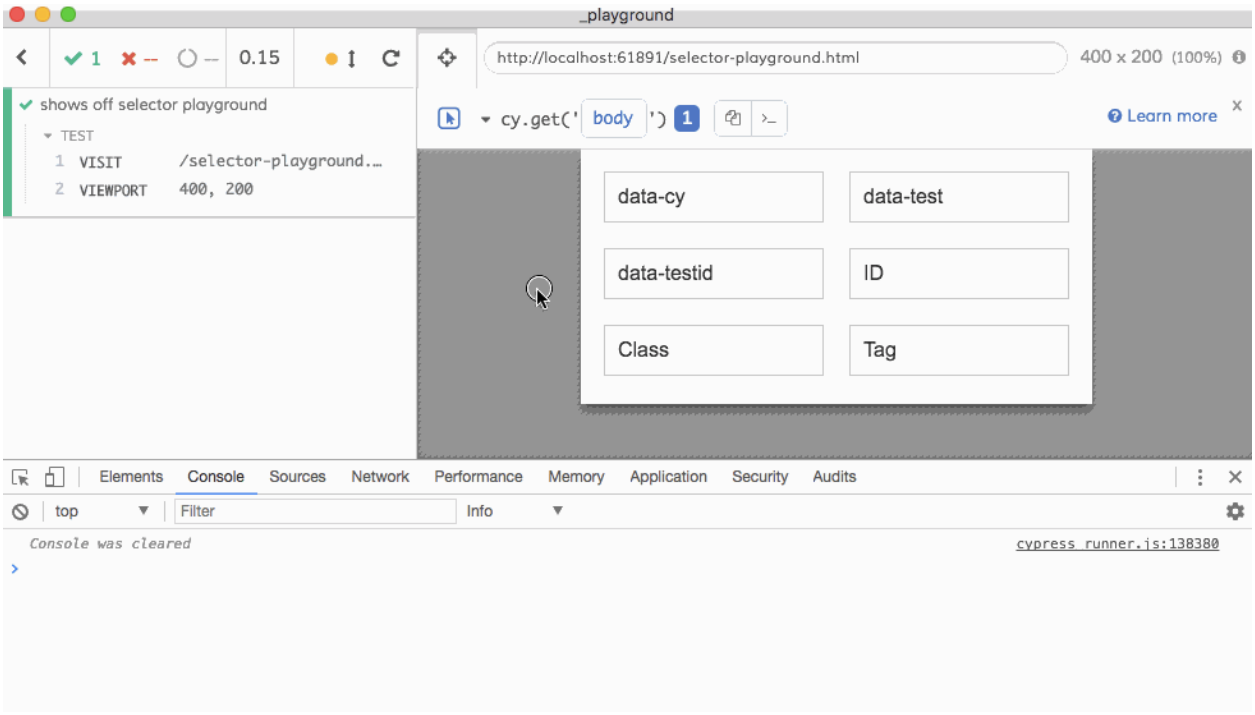

Figure 23 TestCafe Studio Demo

Console Output

TestCafe has a nice console output for test failures, which shows the exact place where the tests have failed.

```
√ Creator successfully accepts offer and signs out
√ Service finder starts collaboration and signs out
√ Service creator joins the collaboration and sent message

Service Admin
× Admin Deletes Account

  1) The "text" argument is expected to be a non-empty string, but it was undefined.

     Browser: Firefox 82.0 / Windows 10

        33 |
        34 |export const AdminLogin = async (t) => {
        35 |  await t
        36 |  .click(userCredentials.loginButton)
        37 |  .click(userCredentials.emailInput)
      > 38 |  .typeText(userCredentials.emailInput, process.env.ADMIN_USERNAME)
        39 |  .click(userCredentials.passwordInput)
        40 |  .typeText(userCredentials.passwordInput, process.env.ADMIN_PASSWORD)
        41 |  .click(userCredentials.signInButton)
        42 |  .expect(basicNavigation.getCurrentLocation())
        43 |  .contains(`${basicNavigation.url}/admin`);

        at <anonymous> (C:\Users\bzend\Desktop\serviceoffering\tests\testcafe\utils\login.js:38:4)
        at AdminLogin (C:\Users\bzend\Desktop\serviceoffering\tests\testcafe\utils\login.js:34:14)
        at <anonymous> (C:\Users\bzend\Desktop\serviceoffering\tests\testcafe\fixtures\servicePernamentAdmin\deleteUser.js:8:9)
        at test (C:\Users\bzend\Desktop\serviceoffering\tests\testcafe\fixtures\servicePernamentAdmin\deleteUser.js:7:1)
        at <anonymous> (C:\Users\bzend\Desktop\serviceoffering\node_modules\testcafe\src\api\wrap-test-function.js:17:28)
        at TestRun._executeTestFn (C:\Users\bzend\Desktop\serviceoffering\node_modules\testcafe\src\test-run\index.js:295:19)
        at TestRun.start (C:\Users\bzend\Desktop\serviceoffering\node_modules\testcafe\src\test-run\index.js:345:24)

 1/9 failed (1m 28s)
npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! weOffer@0.1.0 e2e: `testcafe firefox tests`
npm ERR! Exit status 1
npm ERR!
npm ERR! Failed at the weOffer@0.1.0 e2e script.
npm ERR! This is probably not a problem with npm. There is likely additional logging output above.

npm ERR! A complete log of this run can be found in:
npm ERR!     C:\Users\bzend\AppData\Roaming\npm-cache\_logs\2020-11-23T10_57_43_488Z-debug.log
```

Figure 24 TestCafe Console Output

Cypress has an extra feature by having a dedicated electron app that shows your tests side by side with a site under test. By this, we can have more details about which exact step the test scenario has failed in.

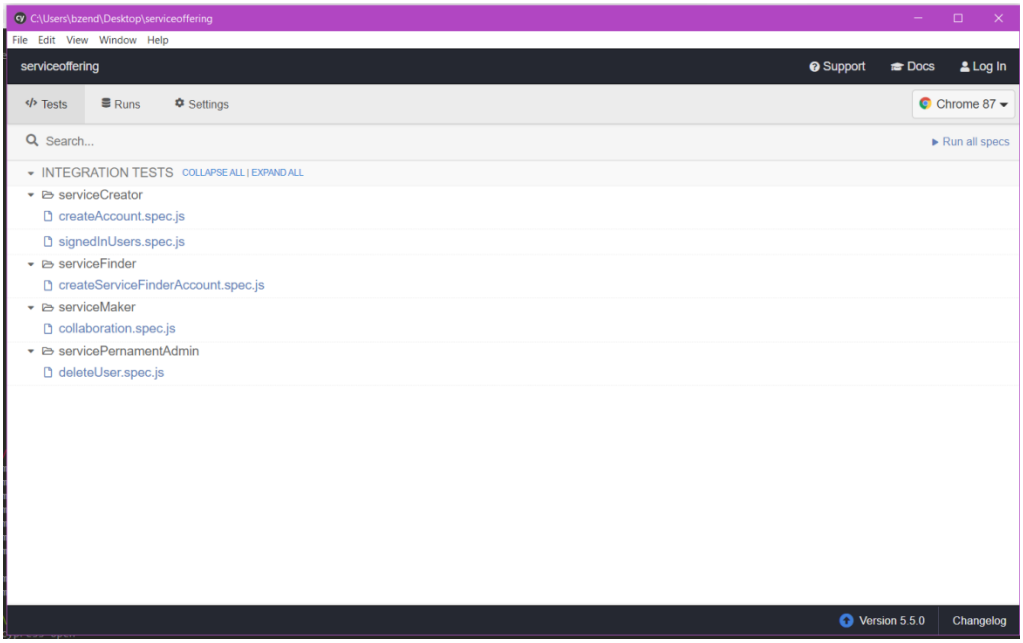

Figure 25 Cypress Interface

Supported browser

Officially supported web browser by TestCafe:

- Google Chrome: Stable, Beta, Dev and Canary
- Internet Explorer (11+)
- Microsoft Edge (legacy and Chromium-based)
- Mozilla Firefox
- Safari
- Google Chrome mobile
- Safari mobile

Officially supported web browser by Cypress

- Canary.
- Chrome.
- Chromium.
- Edge.
- Edge Beta.
- Edge Canary.
- Edge Dev.
- Electron.
- Firefox.
- Firefox Developer Edition
- Firefox Nightly

Advantages and disadvantages between Cypress and TestCafe

TestCafe

Table 11.

Advantages	**Disadvantages**
Really good documentation	The only disadvantage is that it is still under development
Huge community and contributors	
Fast and reliable	
No selenium WebDriver needed to run tests	
Standard css selectors	
React selector extension	
Parallel execution	

Synchronization handled by framework	
Live reload/retest	
Allow interacting with native alerts	
Debug command for easy test debugging	
Screenshot/Videos on fail	
Javascript errors	
Cross browser support	
Jenkins error reporting integrated	
Headless browser	
Integrates easily with Borwserstack and SauceLabs	
TestCafe studio- record and playback tool	

Cypress

Table 12.

Advantages	**Disadvantages**
Really good documentation	No integration with Browserstack or SauceLabs
Huge community.	Error reporting needs more improvement
Fast and reliable	
No selenium WebDriver needed to run tests	
JQuery selectors	
Easy debugging in the Cypress UI	
Parallel execution	
Synchronization handled by framework	
Live reload/retest	

Headless browser	
Debug command for easy test debugging	
Screenshot/Videos on fail	
Javascript errors	
Cross browser support	
Open source	
Possibility to go to previous state (visually)	

TestCafe Pros **Cypress Pros**

Table 13.

Really good documentation	**Possibility to go to previous state (visually)**
Huge community and contributors	Really good documentation
Fast and reliable	Huge community.
No selenium WebDriver needed to run tests	Fast and reliable
Standard css selectors	No selenium WebDriver needed to run tests
React selector extension	JQuery selectors
Parallel execution	Easy debugging in the Cypress UI
Synchronization handled by framework	Parallel execution
Live reload/retest	Synchronization handled by framework
Allow interacting with native alerts	Live reload/retest
Debug command for easy test debugging	Headless browser
Screenshot/Videos on fail	Debug command for easy test debugging
Javascript errors	Screenshot/Videos on fail
Cross browser support	Javascript errors

Jenkins error reporting integrated	Cross browser support
Headless browser	Open source
Integrates easily with Borwserstack and SauceLabs	
TestCafe studio- record and playback tool	

Based on the above-mentioned, we come to a conclusion that TestCafe offers a more similar approach to pure JS, because you get the values from a page and then you assert that those values are correct. On the other hand, Cypress offers a more user oriented approach, when you select the element you want to interact with and you have to do the assertion on the spot.

Continuous Integration

Running cypress in continuous integration is similar to running Cypress locally in our terminal. In general, all what we have to do is to install cypress and run cypress.

Depending on what CI provider you are using, there are different ways to create a config file. Typically, first we boot a local server before running the server. A wait-on module has been installed in order to block the cypress run command from being executed until the server has booted: npm start & wait-on http://localhost:3030

Code snipped:

```
cypress_e2e:
  stage: cypress_e2e
  image:
    name: cypress/base:10
  script:
    - echo $CI_SERVER_HOST
    - npm install cypress
    - npm install wait-on
    - npx wait-on https://weoffer.tech
    - bash ./runCypress.sh run
```

```
artifacts:
  when: always
  paths:
    - cypress/videos/**/*.mp4
    - cypress/screenshots/**/*.png
  expire_in: 1 day
only:
  - develop
```

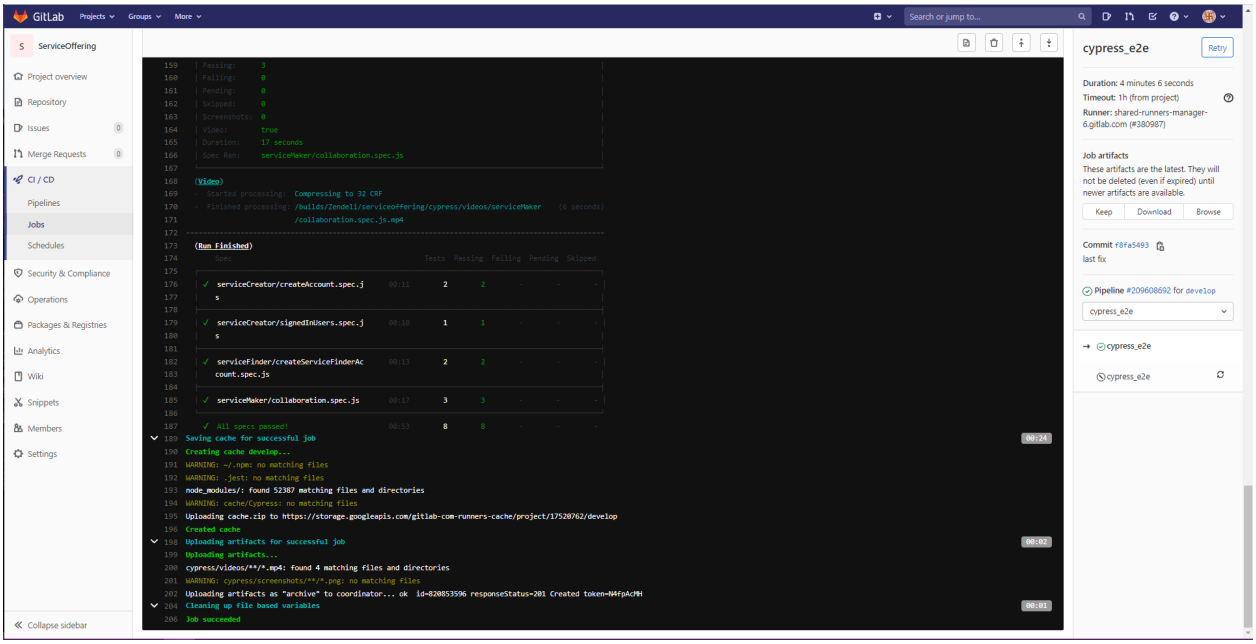

Figure 26 Cypress Pipeline

Same as Cypress we can integrate TestCafe

```
testcafe_e2e:
  stage: testcafe_e2e
  image:
    name: testcafe/testcafe
    entrypoint: ['/bin/sh', '-c']
```

script:

- echo $CI_SERVER_HOST

- npm install wait-on

- npx wait-on https://we-offer.herokuapp.com/

- /opt/testcafe/docker/testcafe-docker.sh "chromium:headless --no-sandbox" -a "npm run start-dev" tests/testcafe/

only:

- develop

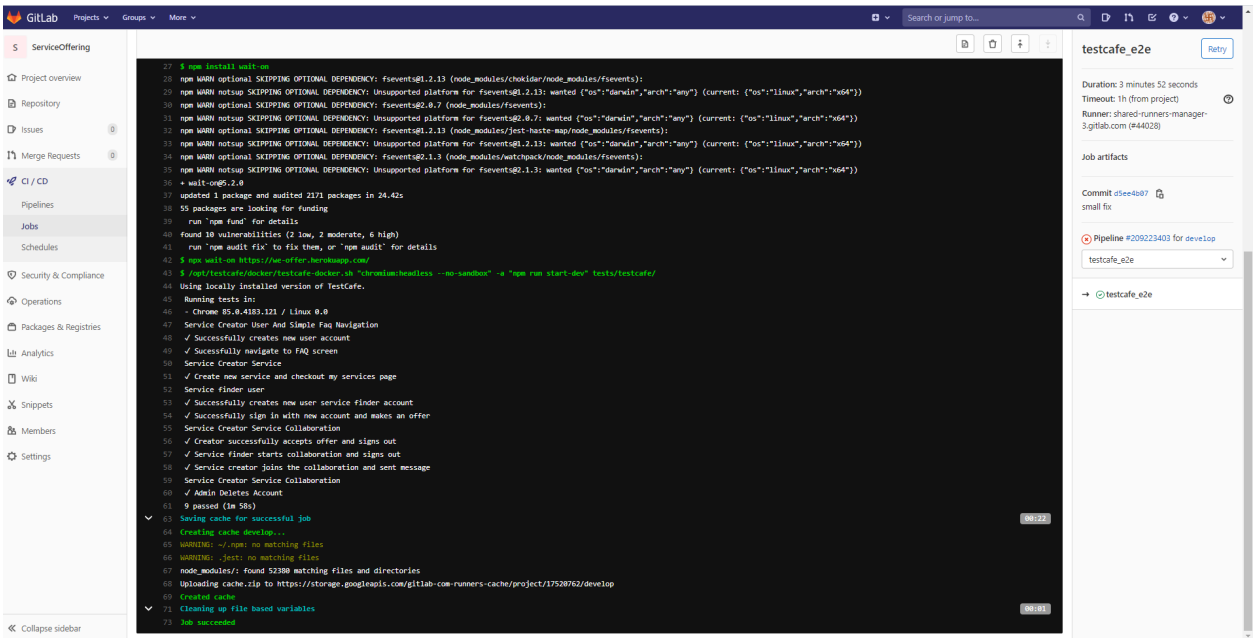

Figure 27 TestCafe Pipeline

Cloud Testing

Cloud Testing is a type of software testing in which the software application is tested using cloud-computing services. The purpose of Cloud testing is to test the software for functional as well as non-functional requirements using cloud computing which [19] ensures faster availability with scalability and flexibility to save time and cost for software testing.

Cloud computing is an internet-based platform that renders various computing services like hardware, software and other computer related services remotely.

From the selected frameworks in our thesis, only TestCafe supports full integration of cloud testing such as LambdaTest or BrowserStack. In addition, the integration of Lambdatest has been made in this thesis.

LambdaTest, a cloud-based, cross browser testing platform is out with an npm plugin that would allow you to integrate TestCafe with your LambdaTest account. That way, you can expand your test coverage using LambdaTest Selenium Grid of 2000+ real browsers, and browser versions running across various operating systems for mobile, desktop, and tablets. Similar to TestCafe, LambdaTest Selenium Grid also allows you to perform parallel testing.

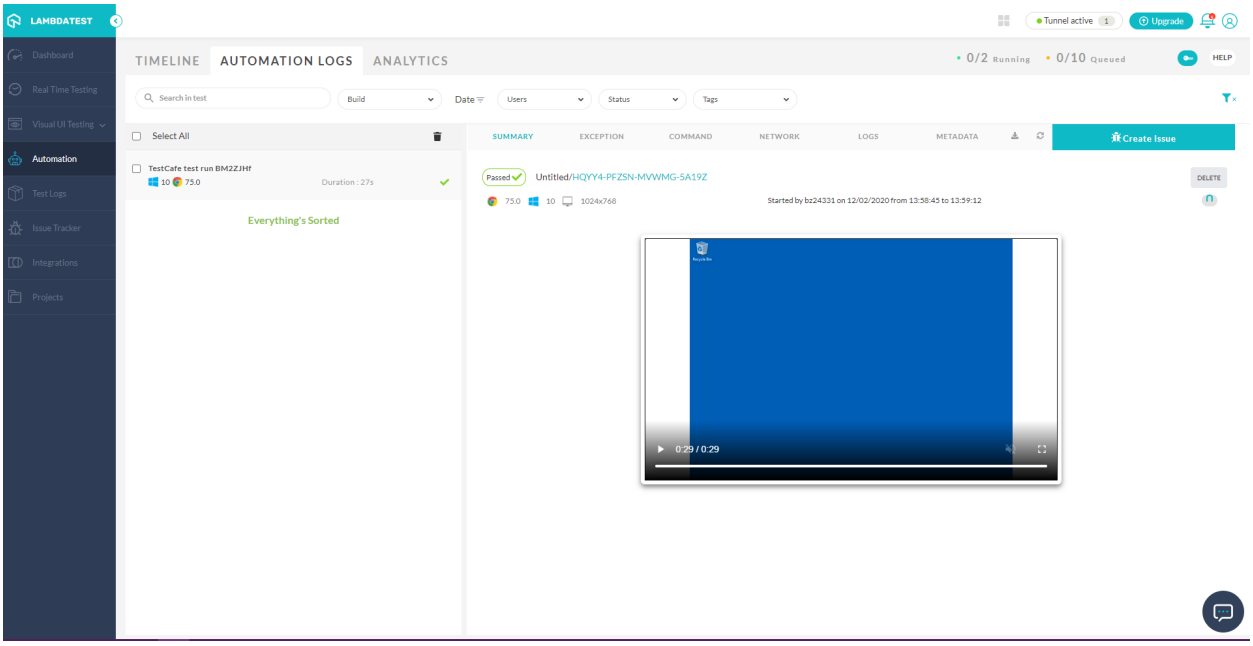

Figure 28 Lambdatest Dashboard

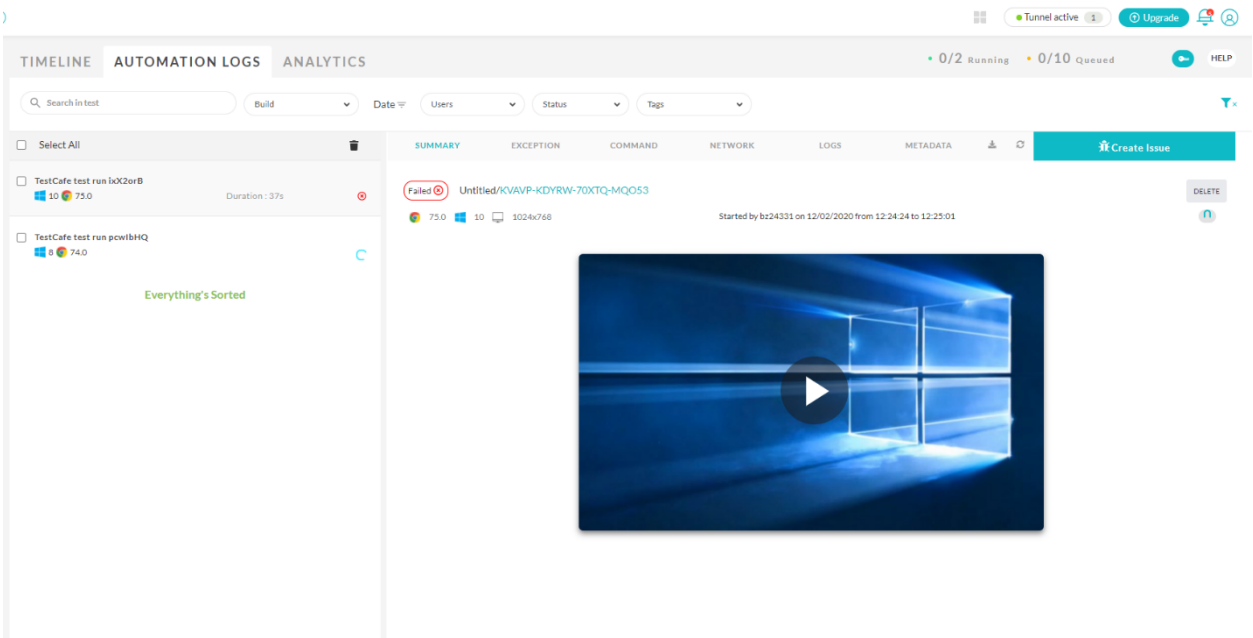

Figure 29 Lambdatest List

Chapter 5. Results and discussion

With the completion of the implementation phase (collecting the necessary data and applying sentiment analysis), for these test scenarios only a specific laptop has been used with the following performances:

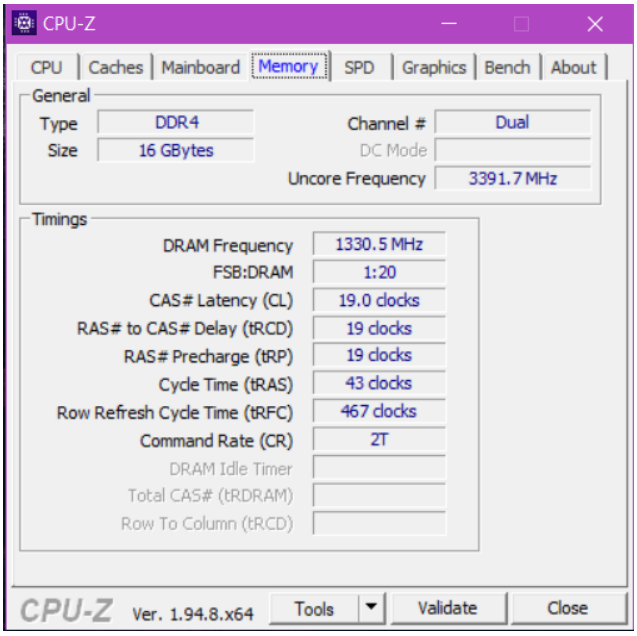

Figure 30 Laptop Memory

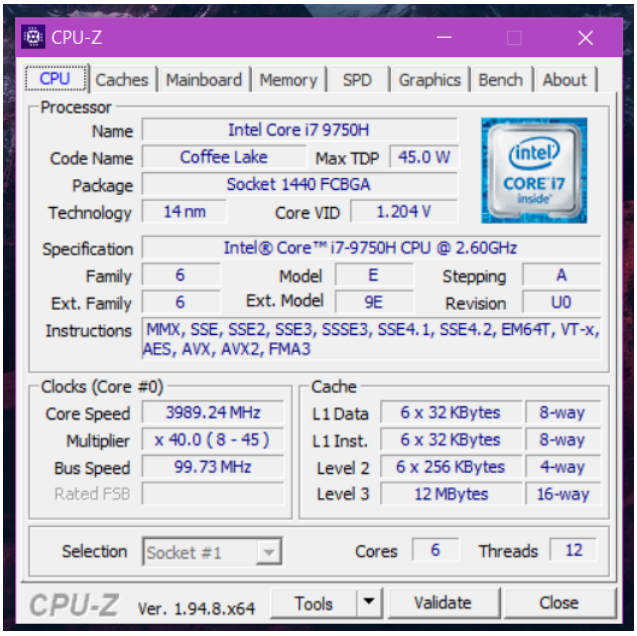

Figure 31 Laptop CPU

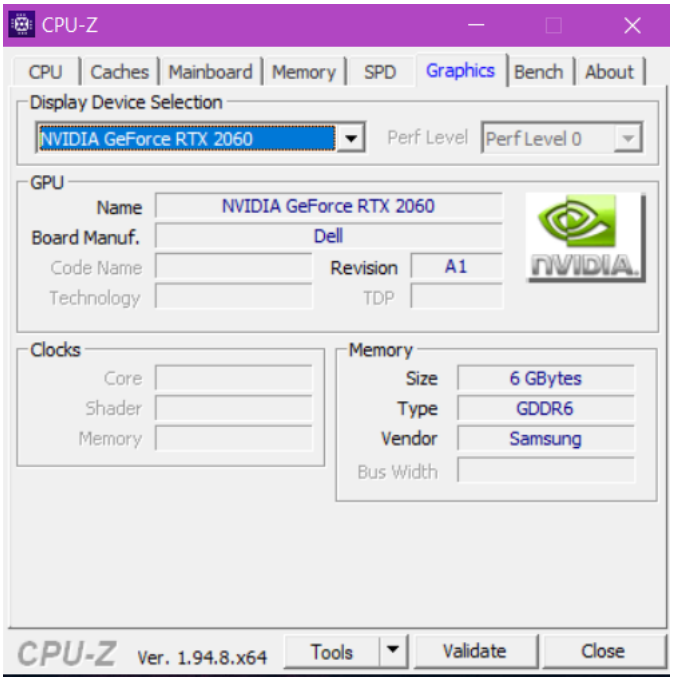

Figure 32 GPU

TestCafe (headless Chrome)

Table 14.

Successfully creates new user account(test file)	Successfully navigate to FAQ screen	Create new service and checkout my services page	Successfully creates new user service finder account	Successfully sign in with new account and makes an offer	Creator successfully accepts offer and signs out	Service finder starts collaboration and signs out	Service creator joins the collaboration and sent message	Admin Deletes Account
10036	936	11066	9361	13288	12286	17334	11074	7102
10036	3987	11043	8846	13335	13326	19279	13376	11805
10080	3956	10773	8924	10448	13020	14407	11476	10203
10122	3972	11425	9458	13407	12074	19531	11424	10249
10063	1114	11433	9041	14293	12257	17302	11156	10046
10062	3968	11128	11219	10358	12348	17322	11084	10139
7120	3959	11439	9357	13400	12158	17569	10906	10676

10040	4035	11241	9644	15135	14757	17302	11593	10263
10128	1098	11528	9017	13445	12118	17170	8556	10158
10461	3980	11194	9006	13407	12365	17377	11462	10188

In table 14 tests were run 10 times in order to get the average speed of each test scenario by table 14 and table 15 we can se that each test scenario on each test run is approximately close to 10 time result. By running testcafe test scenarios, we can see that for test scenario number one which is “Successfully creates new user account (test file)”, the average for running the test is 9.814 seconds. For the second test scenario, “Successfully navigate to FAQ screen”, the average is 3.1 seconds, which means this is the lightest test scenario on this project; it only contains the navigation to faq screen. “Create new service and checkout my services page” is the third test scenario where service creator creates its own first service and it was executed on an average of 11.227 seconds, which is somewhat long for a test file.

The fourth test scenario, “Successfully creates new user service finder account” produced an average of 9.387 seconds after being run for ten times. On the fifth one, “Successfully sign in with new account and makes an offer”, we have an average of 13.051 sec. “Creator successfully accepts offer and signs out” averages at 12.670 seconds; “Service finder starts collaboration and signs out” - 17.459 seconds; “Service creator joins the collaboration and sent message” - 11.210 seconds; “Admin Deletes Account” - 10.082 seconds.

After running TestCafe test for 10 times, we have the following results for each test that was executed:

Table 15.

Test name:	**Successfully creates new user account(test file)**	**Successfully navigate to FAQ screen**	**Create new service and check out my services page**	**Successfully creates new user service finder account**	**Successfully sign in with new account and makes an offer**	**Creator successfully accepts offer and signs out**	**Service finder starts collaboration and signs out**	**Service creator joins the collaboration and sent message**	**Admin Deletes Account**
Average	9814.8	3100.5	11227	9387.3	13051.6	12670.9	17459.3	11210.7	10082.9
Total Sec.	9.814	3.100	11.227	9.387	13.051	12.670	17.459	11.210	10.082

The arithmetic average test speed for TestCafe is 1.633 min.

The geometric average test speed for tescafe is 0.750 min.

Cypress (headless chrome)

In the table below are listed the results of Cypress tests in MS (milliseconds).

Table 16.

Test name:	**Successfully creates new user account(test file)**	**Successfully navigate to FAQ screen**	**Create new service and check out my services page**	**Successfully creates new user service finder account**	**Successfully sign in with new account and makes an offer**	**Creator successfully accepts offer and signs out**	**Service finder starts collaboration and signs out**	**Service creator joins the collaboration and sent message**	**Admin Deletes Account**
	5638	399	6905	5628	3965	5068	3408	3637	3901
	5662	401	6846	5624	3955	5109	3359	3693	3883
	5626	398	6936	5668	4023	4993	3521	3693	3818
	5709	404	7027	5610	4081	5000	3369	3869	3838
	5714	390	7063	5638	3848	5319	3393	3705	3816
	5754	399	7066	5602	3995	5022	3408	3996	3819
	5688	395	7239	5726	3925	5086	3382	3726	3867
	5688	400	7239	5726	3925	5086	3382	3726	3867
	5646	400	6903	5548	4036	4991	3367	3712	4264
	5646	395	6903	5548	4036	4991	3367	3712	4264

On the table 14, tests were run 10 times in order to get the average speed of each test scenario. From table 16 and table 17 we can see that each test scenario on each test run on Cypress is approximately close to 10 time result. By running Cypress test scenarios, we can see that for test scenario number one, which is "Successfully creates new user account (test file)", the average for running the test was 5.677 seconds. For the second test scenario, "Successfully navigate to FAQ screen" the average is 0.391 seconds, which means this it is the lightest test scenario on this project; it only contains the navigation to faq screen. The third, "Create new service and checkout my services page" is the test scenario where service creator creates its own first service and it was executed on a average 7.012 seconds which for a test file is a little bit too much.

The fourth, "Successfully creates new user service finder account" we got the average of 5.631 seconds after running the same test scenario for the 10th time. On the fifth one, "Successfully sign in with new account and makes an offer" we have an average of 3.978 sec. "Creator successfully accepts offer and signs out" achieved an average of 5.066 seconds; "Service finder starts collaboration and signs out" - 3.395 seconds; "Service creator joins the collaboration and sent message" - 3.749 seconds; "Admin Deletes Account" - 3.933 seconds.

After running TestCafe test 10 times we have the following results for each test that was executed: Table 17.

Test name:	**Successfully creates new user account(test file)**	**Successfully navigate to FAQ screen**	**Create new service and check out my services page**	**Successfully creates new user service finder account**	**Successfully sign in with new account and makes an offer**	**Creator successfully accepts offer and signs out**	**Service finder starts collaboration and signs out**	**Service creator joins the collaboration and sent message**	**Admin Deletes Account**
Average	5677.1	398.1	7012.7	5631.8	3978.9	5066.5	3395.6	3746.9	3933.7
Total in seconds	5.677	0.391	7.012	5.631	3.978	5.066	3.395	3.749	3.933

The arithmetic average test speed for cypress is 0.707min.

The geometric average test speed for cypress is 0.711min.

Difference between Cypress and TestCafe in chrome headless

Table 18.

Test name:	Successfully creates new user account(test file)	Successfully navigate to FAQ screen	Create new service and check out my services page	Successfully creates new user service finder account	Successfully sign in with new account and makes an offer	Creator successfully accepts offer and signs out	Service finder starts collaboration and signs out	Service creator joins the collaboration and sent message	Admin Deletes Account
TestCafe	9.814	3.100	11.227	9.387	13.051	12.670	17.459	11.210	10.082
Cypress	5.677	0.391	7.012	5.631	3.978	5.066	3.395	3.749	3.933
Difference	4.137	2.709	4.215	3.756	9.073	7.604	14.064	7.461	6.149

Table 19.

Total	TestCafe	Cypress
S(seconds)	98	42.422
Min(minutes)	1.633	0.707

By running all tests both Cypress and Testcafe headless in Chrome browser on our web application, we can clearly see the difference between these two.

On the first test case, “Successfully creates new user account” Testcafe was able to run the test for 4.137 seconds slower than Cypress with 5.677 seconds. One of the main differences is that cypress runs test code in browser process whereas TestCafe runs it in Node. This means Cypress tests have access to real DOM elements but in TestCafe communication between your tests and the DOM must be serialized.

Successfully navigate to FAQ screen was run by TestCafe for 3.100 seconds and Cypress 0,391 seconds with a huge difference of 2.709 seconds. In “Create new service and checkout my services page”, TestCafe is slower than Cypress by 4.215 seconds. In “Successfully creates new user service finder

account", TestCafe run tests for 9.387 seconds and Cypress for 5.631 seconds, even on this test scenario TestCafe is slower than Cypress by 3.756 seconds.

In "Successfully sign in with new account and makes an offer", Cypress is faster than TestCafe by 9.073 seconds, Moreover, on all the 3 remaining test scenarios, Cypress was faster than Testcafe. In "Creator successfully accepts offer and signs out", Cypress was faster by 7.604 seconds; In "Service finder starts collaboration and signs out" it was faster by 14.065 seconds; In "Service creator joins the collaboration and sent message" it was faster by 7.461 and in running the admin tests, Cypress was faster than TestCafe by 6.149 seconds.

By running all test scenarios in both Cypress and Testcafe, we can conclude that Cypress is faster than Testcafe by 55.578 seconds.

TestCafe (headless Firefox)

In the table below are listed the results of TestCafe tests in MS (milliseconds).

Table 20.

Successfully creates new user account(test file)	**Successfully navigate to FAQ screen**	**Create new service and checkout my services page**	**Successfully creates new user service finder account**	**Successfully sign in with new account and makes an offer**	**Creator successfully accepts offer and signs out**	**Service finder starts collaboration and signs out**	**Service creator joins the collaboration and sent message**	**Admin Deletes Account**
9045	4379	13	6949	14049	10621	17605	11986	16421
7915	4082	13267	10226	15780	13133	18203	9204	11837
9112	4169	13435	7041	13991	14701	18931	12354	12606
11810	4107	13963	12788	12007	13071	19392	12373	12600
10988	5376	16952	6938	15225	12861	17786	12531	14505
11518	1279	13043	6910	13165	14854	28438	24049	14433
7951	1175	13199	6671	11374	10244	17697	12263	12786
8044	4152	13515	6946	11812	9978	15485	9430	9983
7871	4049	13150	6714	14380	12931	15087	9639	10416
8185	1015	13209	6866	15000	10442	18027	9389	10100

After running TestCafe 10 times, we have the following results for each test that was executed:

Table 21

Test name:	**Successfully creates new user account(test file)**	**Successfully navigate to FAQ screen**	**Create new service and checkout my services page**	**Successfully creates new user service finder account**	**Successfully sign in with new account and makes an offer**	**Creator successfully accepts offer and signs out**	**Service finder starts collaboration and signs out**	**Service creator joins the collaboration and sent message**	**Admin Deletes Account**
Average	9243.9	3378.3	13703.4	7804.9	13678.3	12283.6	18665.1	12321.8	12568.7
Total in sec.	9.243	3.367	13.703	7.804	13.678	12.283	18.665	12.321	12.568

On table 20, tests were run 10 times in order to get the average speed of each test scenario. From table 20 and table 21, we can see that each test scenario on each test run on TestCafe is approximately close to 10 time result. By running TestCafe test scenarios, we can see that for test scenario number one which is “Successfully creates new user account (test file)”, the average for running the test was 9.243 seconds. For the second test scenario, “Successfully navigate to FAQ screen”, the average was 3.367 seconds, which means this it is the lightest/fastest (???) test scenario on this project; it only contains the navigation to faq screen. The third, “Create new service and checkout my services page” is the test scenario where service creator creates its own first service and it was executed at an average of 13.703 seconds, which for a test file is a little long.

The fourth, “Successfully creates new user service finder account” after running the same test scenario for ten times, we got the average of 7.804 seconds. In the fifth one, we have “Successfully sign in with new account and makes an offer” with an average 13.678 seconds. “Creator successfully accepts offer and signs out” had an average of 12.283 seconds. “Service finder starts collaboration and signs out” - 18.665 seconds; “Service creator joins the collaboration and sent message” - 12.321 seconds; “Admin Deletes Account” - 12.568 seconds.

The average test speed for TestCafe run on Firefox headless is 1.727 min

Cypress (headless Firefox)

In the table below are listed the results of TestCafe tests in MS (milliseconds).

Table 22.

Successfully creates new user account(test file)	Successfully navigate to FAQ screen	Create new service and checkout my services page	Successfully creates new user service finder account	Successfully sign in with new account and makes an offer	Creator successfully accepts offer and signs out	Service finder starts collaboration and signs out	Service creator joins the collaboration and sent message	Admin Deletes Account
5556	635	7021	5255	4663	4706	4440	4789	3219
5846	883	7097	5310	4828	4668	4734	4768	3052
5782	1025	7071	5303	4709	4898	4737	5132	3150
5873	909	7000	5402	4567	4678	4933	4969	3262
5808	946	6855	5298	4654	4942	4848	4903	3179
5945	1016	6911	5372	5304	4752	4809	4845	3318
5511	633	6800	5273	4642	5004	4705	4883	3142
5725	1019	6925	5251	4991	5146	4350	4487	3166
5701	894	7059	5228	4812	4455	4310	4920	3042
5530	635	6969	5727	5292	4724	4193	4815	3185

On table 22, tests were run 10 times in order to get the average speed of each test scenario. From table 22 and table 23 we can see that each test scenario on each test run on Cypress results are approximately close to 10 time result. By running Cypress test scenarios, we can see that for test scenario number one, which is “Successfully creates new user account(test file)” the average for running the test was 5.727 seconds. For the second test scenario, “Successfully navigate to FAQ screen”, the average was 0.859 seconds, which means that this is the lightest/fastest (???) test scenario on this project, it only contains the navigation to faq screen. The third, “Create new service and checkout my services page” is a test scenario where service creator creates its own first service and it was executed at an average of 6.970 seconds, which for a test file is a little too long.

The fourth, “Successfully creates new user service finder account” we got the average of 5.341 seconds after running the same test scenario for ten times. On the fifth one, “Successfully sign in with new account and makes an offer” we have an average of 4.846 seconds. “Creator successfully accepts offer and signs out” got an average of 4.797 seconds. “Service finder starts collaboration and signs out” - 4.605 seconds; “Service creator joins the collaboration and sent message” - 4.851 seconds; “Admin Deletes Account” - 3.171 seconds.

After running Cypress headless test 10 times, we have the following results for each test that was executed:

Table 23.

Test name:	**Successfully creates new user account(test file)**	**Successfully navigate to FAQ screen**	**Create new service and checkout my services page**	**Successfully creates new user service finder account**	**Successfully sign in with new account and makes an offer**	**Creator successfully accepts offer and signs out**	**Service finder starts collaboration and signs out**	**Service creator joins the collaboration and sent message**	**Admin Deletes Account**
Average	5727.7	859.5	6970.8	5341.9	4846.2	4797.3	4605.9	4851.1	3171.5
Total in seconds	5.727	0.859	6.970	5.341	4.846	4.797	4.605	4.851	3.171

The arithmetic average test speed for Cypress is 0.686 min

The geometric average test speed for Cypress is 0.690 min

Difference between Cypress and TestCafe in Firefox headless

Table 24.

Test name:	Successfully creates new user account(test file)	Successfully navigate to FAQ screen	Create new service and check out my services page	Successfully creates new user service finder account	Successfully sign in with new account and makes an offer	Creator successfully accepts offer and signs out	Service finder starts collaboration and signs out	Service creator joins the collaboration and sent message	Admin Deletes Account
TestCafe	9.243	3.367	13.703	7.804	13.678	12.283	18.665	12.321	12.568
Cypress	5.727	0.859	6.970	5.341	4.846	4.797	4.605	4.851	3.171
Difference	3.516	2.508	6.733	2.463	8.832	7.486	14.06	7.47	9.397

Table 25.

Total	TestCafe	Cypress
S(seconds)	103.632	41.167
Min(minutes)	1.727	0.686

Difference running test scenarios between chrome and firefox

Table 26.

Total	TestCafe	Cypress
Chrome	1.633	0.707
Firefox	1.727	0.686

On the table above, we can see that Cypress test run time results are slower in Chrome and faster in Firefox.

Testcafe test run time is faster in Chrome.

Based on all of these results, we can see that Cypress is faster than TestCafe for 0.926 minutes even in Firefox browser. Note: all these tests were runin headless mode in Chrome and Firefox, which are the most widely used browsers to date.

Chapter 6. Recommendations

After reading the documentation of both frameworks, Cypress and TestCafe, and going through different open source projects, I started to implement them into my web application built in ReactJS and Firebase. The best way to handle end-to-end test structure is by dividing them into functions, which provides the easiest way to read the code.

DRY

I have followed the best principle of programming: Write DRY (do not repeat yourself); for example, we have a scenario where a user wants to log out from application and we have eight same cases where we want to log out via end-to-end tests. In this case, there is no need to write the same logic eight times but we can use only one function and call that whenever we need it. This can be done on both Cypress and TestCafe.

Another thing is that when you are writing end-to-end tests, always put selectors in another file; from there, you can split into classes –we had the same case on this project where we have an external file that only deals with selectors. We have divided all the desired classes there.

Browser Support

In terms of browser compatibility, TestCafe support more browsers than Cypress. If you feel insecure about how your application will act in different browsers, the multi-browser support in TestCafé will be a big plus for you. TestCafé is able to run the tests in the following browsers (when installed on your system):

Google Chrome: Stable, Beta, Dev and Canary, Internet Explorer (11+), Microsoft Edge (legacy and Chromium-based), Mozilla Firefox, Safari, Google Chrome mobile, Safari mobile.

Besides running the tests in the local browsers on a developer's machine, TestCafé is able to run the tests headless in a pipeline and even on the cloud services like SauceLabs or Browserstack.

On the other side, Cypress only supports Chrome, Electron, Chromium and Firefox.

Speed

There is no need to think about speed in these two frameworks. It all depends on what you expect from an end-to-end test. Because Testcafe has only one extra feature, which is Testcafe Studio, in comparison to Cypress. also In addition, it is not that open source that has been mentioned on the TestCafe documentation. On the other side, Cypress is entirely free. In addition, the Cypress playground is one of the best features that Cypress has for picking selectors in the easiest way. On my web application, Cypress was faster and the main reason for that is that Cypress runs test codes in the browser process, whereas TestCafe runs it in Node. This means that Cypress has access to real DOM elements but in TestCafe communication between tests and DOM must be serialized.

If you want to run end-to-end tests only in limited browsers, say Chrome and Firefox, the best framework for you is Cypress. But if you plan to run you end-to-end tests in more browsers and also

integrate Browserstack, then you can use Testcafe which is a little bit more complicated than Cypress; also, selecting selectors gives you harder time to select all the elements on your web application.

Selector Playgrounds

In terms of using the tools to get a selector the easiest way, Cypress has Selector Playground and TestCafe has TestCafe Studio.

In comparison to Cypress selector playground, which is free to use, TestCafe Studio offers only 30 days trial period and afterwards you have to pay in order to use TestCafe Studio.

An extra feature that I liked in TestCafe Studio is test recording, which allows us to generate test code in an automated way.

In my opinion, the Cypress selector playground is better than TestCafe Studio.

Live reloading (hot reload)

A very handy feature of Cypress is the live reloading capability. This means that as you write your testscript and hit ‘save’, the Test Runner picks up the file and reruns the test, even if this means breaking off the already running test. This gives you almost instant feedback on the test you are writing. In TestCafe, this is implemented a bit less intuitively. When you edit and save the test file while your test is already running, you have to abort your test by clicking ctrl-z (but then you have to start TestCafe all over again) or you have to wait for the test run to finish and then hit save again. Therefore, TestCafe listens to changes in the test file only when the Runner is not running a test.

Chapter 7. Conclusion

It is important to note that E2E tests do not replace component test coverage, but unlock a holistic testing scheme. This approach drives higher product quality and maintainability, promotes an atmosphere of ownership, leads to faster development, and reduces operating costs.

Both frameworks are great choices, and nothing can get wrong if you choose one of them. In this research paper we have analyzed the implementation of both frameworks, meaning by this the performance, documentation, browsers support, CI integration, Cloud testing integration, test runners, code syntax and from all of these we have collected the required data to come in a conclusion for which framework is better to use and more easy to implement.

The first hypothesis raised in this thesis claimed that: Cypress provides a better learning curve, its partially true, according to the analyses of documentations, tutorials that both frameworks offers we came to a conclusion that both frameworks have a better learning curve.

The second hypothesis: TestCafe offers a wide range of features compared to Cypress. According to the analyses that we have done on previous chapters, we came to a conclusion that TestCafe offers a wide range of features, such as TestCafe Studio, which allow users to record test cases and automatically generates an js file with the code in it. Browser support: Testcafe, compared to Cypress, offers more support for Safari, Firefox and other web browsers that we have listed on the previous chapters. Cloud testing suite support: TestCafe supports a lot of cloud testing suites in comparison with Cypress. Since Cypress is newer than TestCafe, support for Cloud testing is in its alpha and beta phase.

The third hypothesis: In the frame of application, Cypress is faster. Based on all the analyses and performance tests that we did on the previous chapters, we can clearly come to a conclusion that Cypress is much faster than TestCafe in the frame of application. The main reason for this is that Cypress runs your actual test code in the browser process whereas TestCafe runs it in Node.

Bibliography

1. Sutapa.F.A.K.P.G (2020), Reveiw of Automated Testing Approach for Software Regression Testing, IOP Conference Series: Materials Science and Engineering, Volume 846, Issue 1, pp. 012042 (2020)
2. Jyotsna, Varshney.M,Garg.Sh,Rajpoot.A.K (2017), Automated Testing: An Edge Over Manual Software Testing, ResearchGate
3. Lefterov.D (2019), E-Commerce testing framework, ResearchGate
4. Zhang.D (2012), End to end testing using integrated tools, Ohiolink
5. Lefterov.D, Enkov.S (2019), Automated Software Testing Framework "Stassy", ResearchGate
6. Tsau.W, Bai.X, Paul.R.A, Shao.W (2001), End-to-end integration testing design
7. Mendez-Porras.A, Alfaro-Velasco.J, Martinez.A (2020), Evaluation of the Automated Testing Framework: A case study, ResearchGate
8. Gadwal.A.S, Prasad.L (2020), Comparative review of the literature of automated testing tools, ResearchGate
9. Mobaraya.F, Ali.Sh (2019), Technical Analysis of Selenium and Cypres as Functional Automation Framework for Modern Web Application Testing, ResearchGate
10. Naim.N (2017), ReactJS: An open Source JavaScript Library for front-end development, scribd.com
11. S. Katalon (2020), What is End-to-End (E2E) Testing? All You Need to Know, katalon.com
12. Chowdhury.A (2018), All You Need to Know About End to End Testing, lambdaest.com
13. TechnoArch Softwares (2017), React is a JS library for building user interfaces https://www.technoarchsoftwares.com/
14. Pandit.N (2020), What and why react.js
15. Godlewski.M (2016), Top 6 Reasons Why we love React
16. Faheem (2018), Why I choose react?
17. Appoennix (2018), ReactJs website development
18. Adel. M, The complete Firebase and Amazon S3 with JavaFX course.
19. Guru 99 (2020), What is Cloud testing.
20. CypressIO Documentation (2019)
21. Testcafe Documentation(2017)
22. LambdaTest Documentation
23. Oreilly, CI, https://www.oreilly.com/library/view/continuous-integration-delivery/9781787286610/13a652d5-9d26-45dd-99f0-f79576bba30c.xhtml (12/24/2020)