Automated Testing Tools for Front-End

Project Report

Anna Beluskova

Student No.: 20069358
Supervisor: Caroline Cahill

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

School of Science & Computing

A report submitted in fulfilment of the requirements for the Higher Diploma in Computer Science

ABSTRACT

A current trend in a professional Software Development industry is for the team of developers to be able to deliver their product with a high confidence that its components are fully functional and faultless, so that the customer would not loose their trust in the product. This is the reason why the software product needs to be tested in every single way possible and as often as new code is implemented in the application.

While the unit tests are fast and simple, the functional tests, related to the front end part of the application, that is visible to a potential user, are very costly, because it takes a lot of developer's time to write them, run them and to maintain them. Such tests are very fragile and each small change in the user's interface might cause such test to fail.

Agile methodologies (such as Scrum) emphasise the importance of continuous deployment of working software with small, improving iterations, in collaboration with customers paying a particular attention to a business value of the product.

The Continuous Integration is a process of a constant verification of a newly deployed code into the code repository. Upon each new push into the code repository, a new build of the whole application system starts and each component of the application is newly rebuilt and tested to verify that a proper functionality of the application is retained.

In order for the rebuild to happen, Docker containers are build and they run to provide for all of the needed services. In the GoCD server all parts of such integration are set and a built-in graphical user interface displays the results of each such restart.

5 automated, open-source front-end testing tools have been chosen for the purpose of this project. They have been installed and tested, following the Comparison Criteria that have been set. These Criteria were based not only on a developer's experience with the testing tools, but also on their possibility to run in multiple browsers and also on their possibility to be implemented in the process of Continuous Integration. For that reason, one part of the testing tools' functionality to verify was, its possibility to run in a virtual environment of the Docker container and subsequently, to run tests automatically, triggered by the changes in the code repository.

Five gradually-building Iterations of a simple test called "Login Add Funds" have been set and then a testing script in each of the chosen tools has been written. The testing tools were afterwards running the same test according to their own script. The results of each tool have been observed and marked, especially in terms of their log output in the terminal console. Also Docker image for each of the testing tools was created and the test was implemented in the created Pipelines of the company's GoCD server.

All findings and observations are written in the Chapter 5 of this project report. A simple table displays all comparison results in a convenient 5-star way. In the end, one of the tools was chosen and fully implemented in the company's development process.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Caroline Cahill for her guidance and advice.

I would also like to thank my work-placement mentor Mick O'Brien and rest of the RouteMatch Software staff that helped with my project.

Special thanks go to Leigh Griffin, PhD. for his continuous reviews, comments and his guidance. A tremendous support he has shown is invaluable.

Declaration of Authenticity

I declare that the work which follows is my own, and that any quotations from any sources (E.g. books, journals, the internet) are clearly identified as such by the use of 'single quotation marks', for shorter excerpt and identified italics for longer quotations. All quotations and paraphrases are accompanied by [reference number] in the text and a fuller citation in the bibliography. I have not submitted the work represented in this report in any other course of study leading to an academic award.

Student..... Date.....

Word Count

This project contains 14026 words.

Statement of Copyright

TABLE OF CONTENT

Abstract **2**

Acknowledgements **3**

Declaration of Authenticity **4**

Word Count **4**

Statement of Copyright **4**

Table of Content **5**

List of Figures **7**

List of Tables **8**

Abbreviations and Acronyms **9**
Acronym: Full Term *9*

Chapter 1: INTRODUCTION **10**
1.1 Introduction *10*
1.2 Goals and Objectives *11*
1.3 Background *11*

Chapter 2: PROJECT ANALYSIS & SPECIFICATION **13**
2.1 Introduction *13*
2.2 Methodology *13*
2.3 Behaviour Driven Development *17*
2.4 Test automation runtimes *18*
2.5 User Requirements *21*

Chapter 3: PROJECT STRUCTURE **24**
3.1 Introduction *24*
3.2 Test iterations *24*
3.3 Project Management *26*

Chapter 4: ANALYSIS **29**
4.1 Introduction *29*
4.2 Test Overview *29*
4.3 Dockerisation *32*
4.4 GoCD *34*
4.5 Selenium Webdriver *36*

Chapter 5: COMPARISON & RESULTS **38**
5.1 Testing tools overview *38*
5.2 Console outputs *41*
5.3 Comparison table *44*
5.4 Unresolved issues *44*

Chapter 6: CONCLUSION **46**
6.1 Comparison Results *46*
6.2 Key Skills *46*
6.3 Further work *49*
6.4 Conclusion *49*

BIBLIOGRAPHY **50**

APPENDICES **52**

Testing Script written in different testing tools: **52**

1. *DalekJS* *52*
2. *CasperJS* *53*
3. *Protractor* *55*
4. *NightwatchJS* *59*

LIST OF FIGURES

Figure 1 Waterfall Lifecycle [10] **14**
Figure 2 Agile methodology[9] **14**
Figure 4 Continuous Integration [10] **20**
Figure 6 Typing in login credentials **30**
Figure 7 Customer's home page **30**
Figure 8 Adding funds from new card **31**
Figure 9 Updated balance **31**
Figure 10 Initial start of building the image **33**
Figure 11 Continuing image build **33**
Figure 12 Output of the "docker images" command **34**
Figure 13 GoCD Pipelines page **34**
Figure 14 Protractor Pipeline **35**
Figure 15 Failed Pipeline console output **35**
Figure 17 How the client gets to his destination by taxi [14] **37**
Figure 18 The Selenium Webdriver explained using the taxi analogy [14] **37**
Figure 19 CasperJS console ouput: failed **42**
Figure 20 NightwatchJS console output: passed **43**

LIST OF TABLES

Table 1 Comparison results 44

Abbreviations and Acronyms

Acronym: Full Term

UI: User Interface

UX: User Experience

CSS: Cascading Style Sheets

SCSS: Syntatically Awesome Stylesheets

HTML: Hyper text Markup Language

HTTP: Hypertext Transfer Protocol

I/O: Input/Output

DB: Database

E2E: End-to-End

API: Application Programming Interface

DB: Data Base

NPM: Node Package Manager

GUI: Graphical User Interface

BDD: Behaviour Driven Development

CHAPTER 1: INTRODUCTION

1.1 INTRODUCTION

A current trend in a professional Software Development industry is for the team of developers to be able to deliver their product with a high confidence that its components are fully functional and faultless, so that the customer would not loose their trust in the product. This is nowadays achieved through the application of principles of Continuous Integration and Continuous Deployment. The main idea behind these principles is to test the application on various levels by writing automated testing scripts. These scripts then run automatically, triggered by the changes in the application's source code; depending on the developer's settings for the test coverage.

The unit tests are the simplest tests and there should be a unit test for each method or function in the back-end part of the application, so typically an average application would have over a hundred unit tests implemented in the development environment.

The front-end functional tests are, on other hand, very comprehensive, long, time-consuming scripts. They test user's interaction with the application, which reflects the calls of many back-end services and their mutual interactions. These tests are very costly, because they take a lot of developer's time to write and they can easily fail, so they also need a lot of maintenance and it's hard to measure a return of investment of such tests. Thus the companies are very slow to adopt and implement such tests in their development process.

However, the great advantage of such tests is that they, if properly written and configured, simulate user's behaviour and verify that the application responds in an expected pattern, checking on the functionality and interaction where the unit tests fail to cover such functionality (especially covering the mutual front-end-back-end interaction) or the unit tests are not capable to discover possible bugs in the application, because some of the bugs are due to browser's response to the application's call.

This is also other important factor which the unit tests fail to cover - the user's browser choice. There are many different browsers available for end users and they can respond differently to the user's behavior, very often reflecting how the application is set and whether it is able to recognise and adjust to different browser behaviour.

Thus, a front-end testing tool must be chosen in a way to ensure that its functionality is able to cover for the company's functional testing needs, while bearing in mind that it also needs to be reliable and flexible. Because it is time-consuming to set-up testing environment for each front-end testing tool and also to write testing code for each test, a correct choice of a proper testing tool is necessary. Such testing tool does not only need to be flexible (for the developer to be able to quickly reflect to the changes in the code) and reliable (the test results must be accurate, correct and realistically reflect possible user's behaviour), but they also must be able to simulate user's interaction in different browsers. On top of that they should also automatically run any time a code is changed in the code repository, and at any given time, because this is the way to achieve a high quality application the user can put a trust in.

Those were the main reasons why I have introduced this project and I have successfully implemented its outcomes in the company's development environment. I have tried five most popular free front-end testing tools have been tried in a way of running exactly same test written in each of the testing framework's script and compared its results and outcomes with regard to the company's development needs – bearing in mind that few different aspects of the tests should be covered by the chosen testing tool. For that reason, I have set Comparison Criteria and I have evaluated each of the given criteria for each of the testing tools by giving it a mark (from 1 to 5 stars), depending on my personal judgment and unbiased observations.

In the end, I chose the most satisfying testing tool and I have written many more comprehensive test scripts in that testing tool and put into the use by the company.

1.2 Goals and Objectives

The following are the project goals:

1. To install open-source front end testing tools to test the user's experience with the web application.
2. Write automated test script to be able to perform the automated tests in the UI/UX environment automatically, based on the test settings.
3. Deploy the test script to the automated Continuous Integration server.
4. Ensure that the chosen testing tool is the best available by comparison with other open-source tools and the tools' outcome is beneficial to the company.

The motivation behind this project is to help the team of developers to achieve a constant monitoring of the front end part of the application, so that the failure in the test would help the team to spot it as soon as it occurs, without the minimal impact on the end users. In order to achieve this, not only the test script itself is important, but also the reliability of such automated scripts must be taken into account.

1.3 Background

The Project objectives were outlined as a part of my work experience in the company called RouteMatch Software. The company is developing a full (mobile and web) application for their American client. The main goal of this application is for the Agencies (each Agency provide a bus transport for people in needs, such as disabled or elderly ones) to be able to serve their customers with a special type of card called SmartCard, which is to be used as a means of payment for the bus trips so that no cash or a credit card is needed when the customer takes a bus tip. The web application serves an Agency to manage their customer's accounts, trips and the SmartCards, the Customer's part is to help the customers to manage their accounts independently, depending on their personal and financial situation.

For the front end functional part of the web application JavaScript programming language is used, more specifically the AngularJS framework. 'AngularJS is a toolset for building the framework most suited to your application development. It is fully extensible and works well with other libraries. Every feature can be modified or replaced to suit your unique development workflow and feature needs' [1]. AngularJS has a model-view-whatever framework implemented into the front-end part of the application and there are also other advantages of using this framework, such as a range of templates and the advantage of two-way data binding

The pleasant user's experience is achieved by using the CSS, Sass and HTML 5 framework. These are currently most popular styling frameworks, very commonly used, and easily customised. While the CSS and HTML 5 have been used by most of the front-end software devlopers for many years, the Sass is a new framework that is becoming very popular, because its basically a very simple CSS framework, but with many useful added features.

The back end part of the whole application is based on Node.js. 'Node.js is an open-source, cross-platform runtime environment for developing server-side Web applications. Although Node.js is not a JavaScript framework, many of its basic modules are written in JavaScript, and developers can write new modules in JavaScript. Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world' [2].

The company uses express.js framework, which is considered to be a standard server framework for node.js

All important details are stored in many separate databases (even for the same customer). The company uses mongo DB databases.

CHAPTER 2: PROJECT ANALYSIS & SPECIFICATION

2.1 INTRODUCTION

Testing means 'developers think hard of all relevant inputs that a program (or subcomponent) can receive and subsequently, write code that executes the program with those inputs, while checking that the program behaves correctly' [3]. In order to decide on a suitable testing tool for a specific application, methodology, framework and test automation runtimes need to be taken into account.

2.2 METHODOLOGY

'Methodologies are defined sets of processes with the aim of making software development more predictable and more efficient' [8]. All software development methodologies are divided into two main types – heavyweight (or traditional) ones and the lightweight (or agile) ones. These two types (and their principles) are seen as the opposites of each other.

2.3.1 WATERFALL

Waterfall methodology is considered to be a typical example of a traditional methodology. These methodologies are based on a sequential series of steps, such as requirements definition, solution building, testing and deployment. Each step consists on a definite set of activities and deliverables that must be accomplished before the following phase can begin. Heavyweight methodologies require defining and documenting a stable set of requirements at the beginning of a project [8].

Typical characteristics of a waterfall (and generally every other heavyweight) methodology are:

- Predictive approach – a large part of software is carefully planned for a long span of time at the beginning of development process
- Comprehensive documentation – all requirements are well documented prior to writing any code. The documents are then signed off by the customer to limit and control all future changes.
- Process oriented – a process is well defined for whoever will be using it. Each person (employee) has defined set of tasks and for each of these tasks there are well defined procedures.
- Tool oriented – for each project there is a well defined set of tools that must be used.

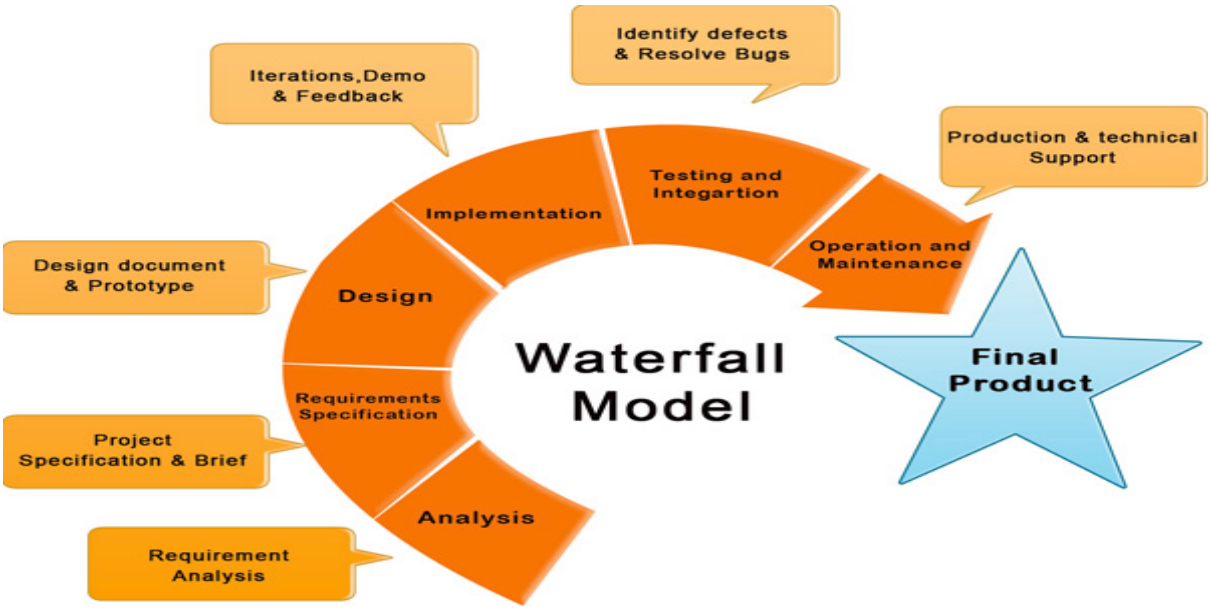

Figure 1 Waterfall Lifecycle [10]

2.3.2 AGILE

In February 2001, seventeen representatives from the different agile methods decided to form an Agile Alliance to better promote their views and what emerged was the Agile 'Software Development' Manifesto. Most of the agile techniques have been used by developers before the alliance but it is not till after the alliance that these techniques were grouped together into a workable framework. The focal values honored by the agilists are presented in the following:

We are uncovering better ways of developing software
by doing it and helping others do it.
Through this work we have come to value:
***Individuals and interactions** over processes and tools*
***Working software** over comprehensive documentation*
***Customer collaboration** over contract negotiation*
***Responding to change** over following a plan [8]*

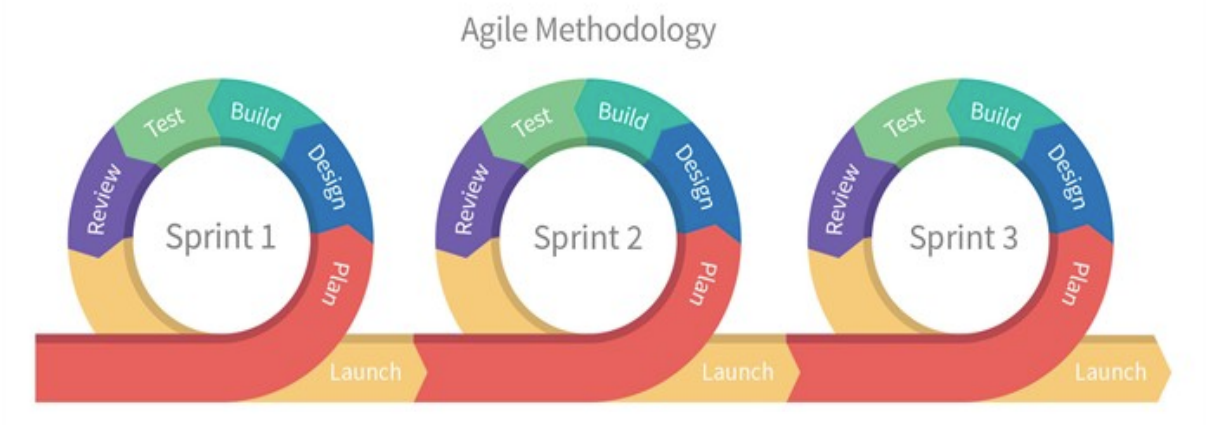

Figure 2 Agile methodology[9]

Scrum

'Scrum is a framework within which a company can employ various processes and techniques. It employs an iterative, incremental approach to optimise predictability and control risk' [8]. Its empirical process control is based on three principles: transparency, inspection and adaptation.

Transparency in this context means that when someone who inspects the process believes it's done; it must fulfill their definition of being done. The inspection ensures that various aspects of the process are constantly checked for possible bugs and errors in the application. The adaptation means that all findings from inspection are resolved as soon as possible.

'The Scrum framework consists of a set of Scrum Teams and their associated roles: Time-Boxes, Artifacts, and Rules' [3].

The Scum Team consist of people with 3 different roles: Scrum Master is ensuring that the team adheres to Scrum values, practices and rules, especially in terms of self-management and cross-functionality. Product Owner is responsible for maximising the value of product, he also prioritises the items of work. The Team (of developers) who does the work, team members, are expected to be self-motivated and to learn new skills if they're needed for the product increments to be potentially shippable at the end of every Sprint; thus the team is required to be a cross-functional group of people who, among them, have all necessary skills to deliver each increment of the product.

The Time-Boxes in Scrum are the Release Planning Meeting, the Sprint, the Sprint Planning Meeting, the Sprint Review, the Sprint Retrospective, and the Daily Scrum. The Release planning requires estimating and prioritising the Product Backlog for the Release, its purpose of release planning is to establish a plan and goals that the Scrum Teams and the rest of the organisations can understand and communicate, also the major risks, and the overall features and functionality that the Release will contain. Products are built iteratively using Scrum, wherein each Sprint creates an increment of the product, starting with the most valuable and riskiest.

Thus a Sprint is a time-framed iteration that lasts a specific period of time, but no longer than one month, it occurs one after another and it is very unusual for the Sprint to be cancelled rather than to be finished by achieving the pre-defined Sprint Goal. In the Sprint Planning Meeting (at the beginning of each Sprint) the Sprint Goal is defined, which is an objective that will be met through the implementation of the Product Backlog; in this meeting it is decided "what" needs to be achieved and "how" it should be done.

The Sprint Review meeting is held at the end of each Sprint, it is a discussion between the Team and the Product Owner about the results of the Sprint and possible problems that the Team run into. In the Sprint retrospective it is inspected how the last Sprint went in regards to people, relationships, process and tools, especially highlighting what went well and what could've been done better if-done-differently. The main purpose of this Retrospective is to identify improvements that can be implemented in the next Sprint. Daily Scrum is a 15-minute long

daily meeting where each member says what he/she did since the last daily scrum, what she/he is going to do before the next meeting and what difficulties she/he run into.

Scrum Artifacts include:

a) the Product Backlog,
b) the Release Burndown,
c) the Sprint Backlog, and
d) the Sprint Burndown.

a. The Product Backlog (for which the Product Owner is responsible) is a list of requirements for the product that the Team is developing, it represents everything necessary to develop and launch a successful product. It is a list of all features, functions, technologies, enhancements, and bug fixes that constitute the changes that will be made to the product for future releases. Changes in business requirements, market conditions, technology, and staffing cause changes in the Product Backlog.
b. The Release Burndown graph records the sum of remaining Product Backlog estimated effort across time. The Product Owner keeps an updated Product Backlog list/ Release Backlog Burndown posted at all times. A trend line can be drawn based on the change in remaining work.
c. The Sprint Backlog consists of the tasks the Team performs to turn Product Backlog items into a “done” increment.
d. Sprint Backlog Burndown is a graph of the amount of Sprint Backlog work remaining in a Sprint across time in the Sprint.

Rules bind together Scrum’s Time-Boxes, Roles, and Artifacts. A completely “done” Sprint increment includes all of the analysis, design, refactoring, programming, documentation and testing for the increment and all Product Backlog items in the increment. Testing includes unit, system, user, and regression testing, as well as non-functional tests such as performance, stability, security, and integration [3].

For the Scrum framework to be successfully implemented, everyone in the Scrum Team must know what the definition of “done” is, otherwise the other two principals of empirical process control don’t work. When someone describes something as “done”, everyone must understand what “done” means.

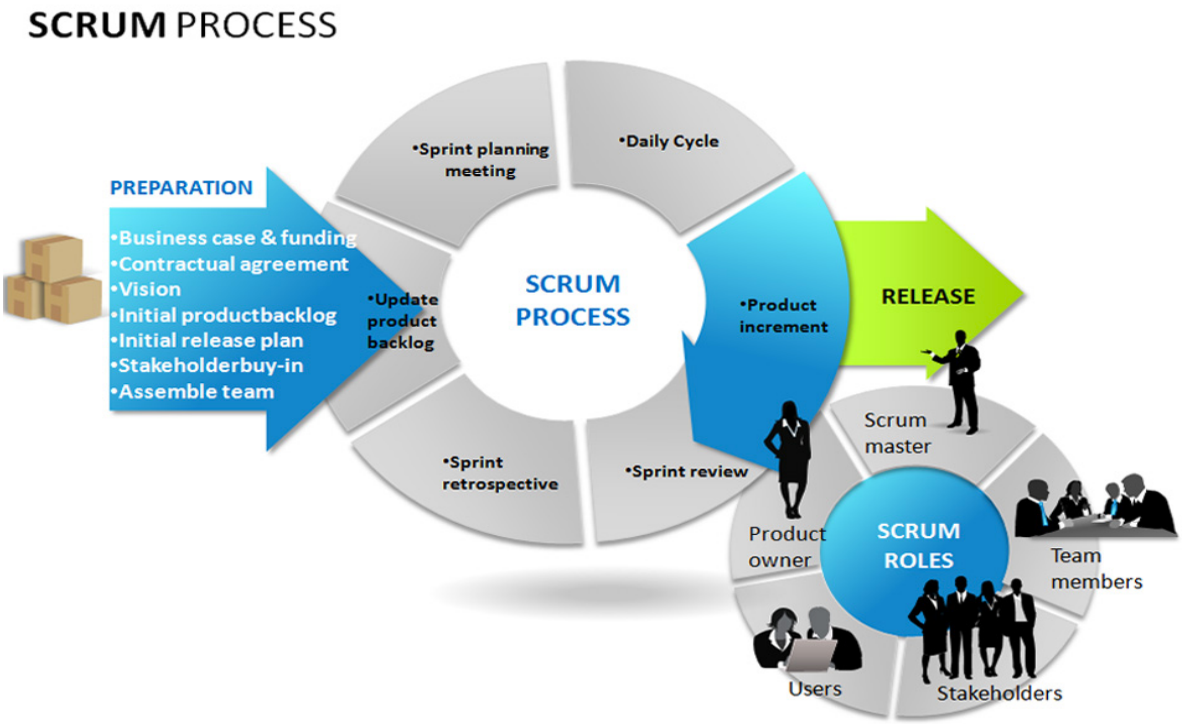

Figure 3 Scrum process [10]

2.3 Behaviour Driven Development

The company has adopted principles of Scrum methodology and it follows and implements Scrum principles in the development process and testing approach. This also means that the testing approach is following the BDD principles. BDD is a current trend in the software development industry, partially (also) because it's based on the Agile methodology.

Behavior-driven developers use their native language in combination with the ubiquitous language of domain-driven design to describe the purpose and benefit of their code. This allows the developers to focus on why the code should be created, rather than the technical details, and minimizes translation between the technical language in which the code is written and the domain language spoken by the business, users, stakeholders, project management, etc. [15]

Each test case in BDD is written describing the desired behavior of the application (or a unit test) and words "it should…" are usually used to express the expected result of the running test.

In such development the UI is the first piece of implemented code, which is then tested to ensure that it behaves in a correct way. Each further piece of code is then tested regarding the GUI that is visible to a potential customer. This approach is reflecting the principle of business value of the product being the most important one.

Following the BDD development principles, I chose the following testing tools to be tried and evaluated for this project:

Selenium IDE

The Selenium IDE testing tool is a simple, user-friendly tool where no programming experience is needed. The user can simply click on the record button and then visit the page of the choice, perform all required actions and stop recording. The recorded test can then be manually repeated as often as possible.

DalekJS

DalekJS is an open source UI testing tool written in JavaScript, it allows the user to write a tailor made testing script. The test scripts can run repeatedly without the need to use the visible web browser, which makes the test really quick to run [4].

CasperJS

'CasperJS is a navigation scripting & testing utility for the PhantomJS (WebKit) and SlimerJS (Gecko) headless browsers, written in Javascript. The test scripts can be written for the functional tests as well as for the unit tests' [5].

Protractor

'Protractor is an end-to-end test framework for AngularJS applications. It is built on top of WebDriverJS, which uses native events and browser-specific drivers to interact with the application as a user would' [6]. In order to run the Protractor, the web driver manager must be in place. Protractor can run on multiple browser, so it can, for example, simulate a chat system and it can also run with a PhantomJS headless browser.

NightwatchJS

'NightwatchJS is an easy to use *Node.js* based End-to-End (E2E) testing solution for browser based apps and websites' [7]. It runs with a help of Selenium server (which can be disabled) and it uses JavaScript programming language in its syntax.

2.4 Test automation runtimes

Based on Agile principle it is necessary for every new feature of functionality to be tested as soon as it becomes available. This is very important for the bugs to be discovered as soon as possible, because the the earlier the stage is, the less effort is needed for fixing. The approach RouteMatch takes regarding to this is a development practice called Continuous Integration.

2.4.1 Continuous Integration

The main principle of Continuous Integration (CI) is to integrate code into a shared repository several times a day. Upon each such integration an automated build is verified. This allows the teams to detect problems early.
CI is backed by several principles and practices [11].

The Practices include:

- *Maintain a single source repository*
- *Automate the build*
- *Make your build self-testing*
- *Every commit should build on an integration machine*
- *Keep the build fast*
- *Test in a clone of the production environment*
- *Make it easy for anyone to get the latest executable*
- *Everyone can see what's happening*
- *Automate deployment*

How to do it:

- *Developers check out code into their private workspaces.*
- *When done, commit the changes to the repository.*
- *The CI server monitors the repository and checks out changes when they occur.*
- *The CI server builds the system and runs unit and integration tests.*
- *The CI server releases deployable artefacts for testing.*
- *The CI server assigns a build label to the version of the code it just built.*
- *The CI server informs the team of the successful build.*
- *If the build or tests fail, the CI server alerts the team.*
- *The team fix the issue at the earliest opportunity.*
- *Continue to continually integrate and test throughout the project.*

Team Responsibilities:

- *Check in frequently*
- *Don't check in broken code*
- *Don't check in untested code*
- *Don't check in when the build is broken*
- *Don't go home after checking in until the system builds*

Continuous Deployment is closely related to Continuous Integration and refers to the release into production of software that passes the automated tests.

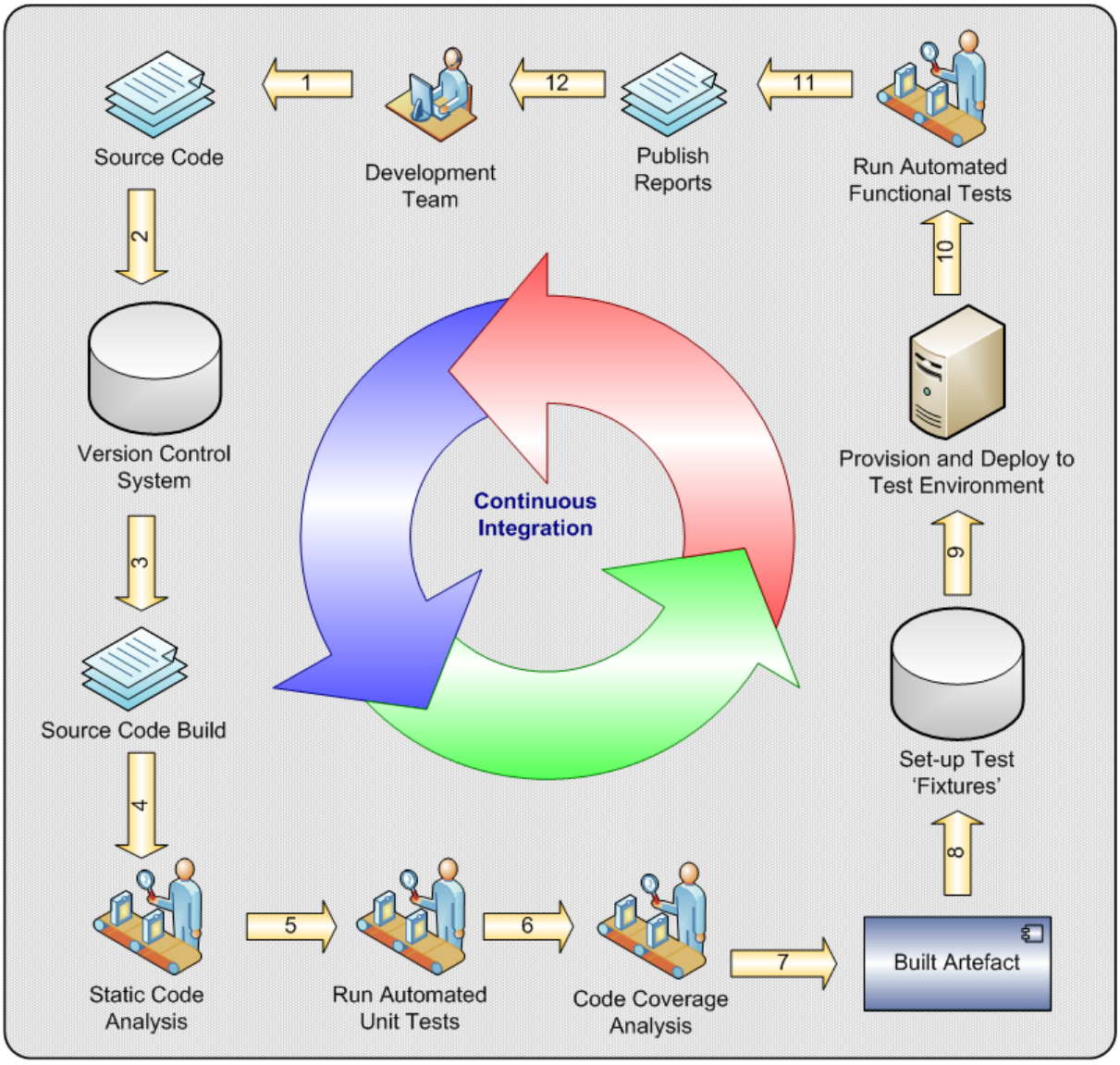

Figure 4 Continuous Integration [10]

The versioning through the shared repository is provided by the tool called GitLab. Continuous Integration in the RouteMatch Software Ireland is provided by the tool called GoCD (described below). Each separate repository from the GitLab is treated as a separate build pipeline in the GoCD tool. A functional front-end test is specifically related to the front-end part of the application and thus all automated UI tests are deployed in this repository. In order to run these tests automatically, a Docker container is built to run the tests in the Pipeline.

2.4.1.1 GitLab

GitLab is a company's code repository where all projects are stored in separate folders. All local updates are pushed to the GitLab masters branch. For the purpose of this Project separate branches were created for the testing tool to be implemented.

2.4.1.2 GoCD

GoCD is an open-source tool developed by ThoughtWorks and its main purpose is to implement the principles of continuous integration in the Git code repository. CD is very dependent on automation. Instead of manually re-building and re-testing every part of the application as soon as the new code is pushed to the Git repository, the GoCD application takes care of automatically setting up the new or modified application environment and also its testing. As a result, the team of RouteMatch Software developers builds a high level of confidence that their product stays reliable even with the new updates.

2.4.2 Docker

Docker containers wrap up a piece of software in a complete filesystem that contains everything it needs to run: code, runtime, system tools, system libraries – anything you can install on a server. This guarantees that it will always run the same, regardless of the environment it is running in [12]. Docker containers are considered to be lightweight (Docker images are constructed to be of a very small size), open (they support every infrastructure) and secure (they isolate applications from each other which provides for an added layer of protection). Dockers are usually compared to the Virtual Machines, but unlike them, they share the kernel with other containers and run as isolated processes in user space on the host operating system. Unlike with the Amazon Web Services (AWS), a developer can easily take copies of his/her live environment and run on any new endpoint running Docker. Comparatively to the AWS, containers can spin up and down in seconds, at any time, to satisfy a customer's demand. Because each container is isolated from each other, it is easy to identify the issues and fix them, causing less disruption to the running services than with the traditional methods.

In the company's private GoCd server, the Docker containers (whether individual ones or a set of the containers) are included in each environment (and in each Pipeline) where they run the processes necessary for the application to be constantly running and available to the users.

2.5 User Requirements

There are three types of Users: Customers, Drivers and Agencies.

2.5.1 Customers

The customers can sign up to the web application using a signup page. Then they can login to the web application by the email and password they had provided in their signup form.

Once the customer is logged in, the home page is displayed where the customer can see his/her contact details (full address and a telephone number) and the balance the customer has available on his/her SmartCard. The active SmartCard (and the SmartCard's number) is also

displayed on the home page. The balance can be increased by clicking on the Add Funds button and further by the option of adding funds through the credit card, either a new one or an already registered one.

The (physical) SmartCard is delivered to the customer by an agency and in order to be assigned to the customer, it must be activated by the customer. The balance on the account is then transferred to the SmartCard and can be used for any trips taken by the agency's bus. In case the SmartCard is lost, the customer has an option to de-activate the SmartCard by visiting his/her profile page where also all SmartCards used to date are displayed. De-activated SmartCard can't be re-activated afterwards. The new SmartCard can also be activated (added), in such case the previous SmartCard is automatically de-activated.

The customer can also register a credit card in order to add funds to the SmartCard. There is an option to have more than one credit card registered and when the funds are to be added to the account, the customer can not only use all of his/her already registered credit cards (which are given as a list of cards to choose from) but also to add funds through a new (not yet registered) credit card, which he has an option to save for a further use. All registered credit cards are displayed on the customer's profile page and there's also an option to delete any of them.

The customer can also see the list of payments on his/her home page. If the customer has a reason to apply for a refund for the paid trip, they can do so by clicking on trip "sandwich menu" option. Once the refund is approved by the agency, the refund amount is returned to the customer and the balance on his/her account increases.

2.5.2 Agencies

The Agency can login to their account by the way of special Email and Password allocated to them. Once logged in, the home page is displayed. The agency can choose from the list of option in the menu bar, headed by the Home option on top, this brings the Agency employee back to the home page.

Users option displays all customers, listed with their email address, customer ID, their first name and their last name, phone number, date when they signed up to the application and the amount that is available on their account balance.

Reports option is not implemented yet.

CreditCard Reconciliation option displays list of all credit card payments along with the customer's id, date, credit card number (in a masked form), RmPay's invoice number, PSP reference number, authorisation number and the payment amount. This list can be filtered by chosen dates.

Total Payments option displays the list of all payments to the company is along with the date and time when the payment was made, payment type (credit card, cash, fare), customer's id and the payment amount. The list can be filtered by chosen dates.

Refunds Report option displays list of all refunds, along with the date and time when the refund request was made, date and time when the refund request was last updated, customer's id, refund's status (created, in progress, closed) approval's status (currently true or false), amount requested (to be refunded) and amount approved. This list can be filtered by chosen dates.

Refunds option displays list of all requested refunds, along with the date of request, customer's id, RmPay invoice number, refund amount (requested by the customer), refund's status (created, in progress, closed), approval status (true or false) and the option for the Agency to approve/decline the refund or to review the receipt.

Manage Smartcards option displays list of all SmartCards that are managed by the agency. The list displays the date the SmartCard was added to the system, date the SmartCard was last time modified, SmartCard number, description and status (whether it's in stock, activated or suspended). This option also includes an Add Batch button, by which a new batch of SmartCards can be added into the system, with an initial status of being in stock.

The Management menu options include Configuration, Settings and Logout and have not been implemented yet (with the exception of the Logout option).

2.5.3 DRIVERS

The bus drivers use a special Android (Samsung) tablet app to process the payment from a customer. This app is solely for the purpose of using the SmartCard. When the payment amount is typed in the app, an option to include additional Extras (such as bicycle, bags, extra passengers etc.) is available. The Customer's id is brought in through the SmartCard reader and the total amount can be paid in few ways including cash, credit card or a SmartCard. If the cash payment is the type of payment, the amount due back to the customers is not given in a form of cash, the SmartCard's balance is increased by this amount instead.

CHAPTER 3: PROJECT STRUCTURE

3.1 INTRODUCTION

As mentioned in a previous Chapter, I have initially created the tests as very basic test cases and added as a Story in the JIRA's active Sprint. Once I had the basic test successfully built, more different testing options related to the same functionality were added to the basic test. Each added testing option of the same basic test was considered to be a new iteration and for that reason I always created a new Story for each iteration in JIRA. When eventually a complex test covered each case of the same functionality, a new series of tests could start testing a different functionality. I also needed to remember that each fully complex test is running independently any time the changes are made in the code repository, resulting starting a new series of tests in the GoCD server and thus to I needed to ensure that initial conditions are met for each test to succeed (such as to add funds before we can apply for a refund to make sure there is a transaction we can relate to or a balance to reduce).

However, for the purpose of this Project, I only used one part of the fully comprehensive test and its iterations are described below.

3.2 TEST ITERATIONS

I have approached the test case with following iterations:

TEST: To Log in a Customer and Add Funds:

Iteration 1

As a Tester
I want to test that when a customer types in
https://rmpayqa.routematch.com/login?a=ky_wheels
It is possible to see the page Title "Login – RM Pay"

Get the Browser to direct to the desired Url
Check whether the text of the Url Title page is "Login – RM Pay"

Iteration 2

As a Tester
I want to test that when I login as a Customer with given credentials
It is properly redirected to the Customer's home page

Get the Browser to direct to the desired Url
Check whether the text of the Url Title page is "Login – RM Pay"
Type into the email field customer's email address
Type into the password field customer's password

Click the Log in button
Check whether the name displayed on the Customer's home page is the Customer's one
Log out the customer

Iteration 3

As a Tester
I want to test that when I check the user's balance
It is properly displayed in the console output

Get the Browser to direct to the desired Url
Check whether the text of the Url Title page is "Login – RM Pay"
Type into the email field customer's email address
Type into the password field customer's password
Click the Log in button
Check whether the name displayed on the Customer's home page is the Customer's one
Get the testing tool to display the customer's balance amount
Log out the customer

Iteration 4

As a Tester
I want to test that when the customer clicks to Add Funds
It is possible to fill out the given form and Add the desired amount with a click of a button

Get the Browser to direct to the desired Url
Check whether the text of the Url Title page is "Login – RM Pay"
Type into the email field customer's email address
Type into the password field customer's password
Click the Log in button
Check whether the name displayed on the Customer's home page is the Customer's one
Get the testing tool to display the customer's balance amount
Click the Add Funds button
When the new window pops-up chooses to add a funds from a new card
Fill out all new card fields
Fill out the top up amount of $5.12
Click on the Add Funds button from within the form
Manually check that the balance amount has increased
Log out the customer

Iteration 5

As a Tester
I want to test that when the customer adds funds through the form
It is properly increasing the balance amount

Get the Browser to direct to the desired Url
Check whether the text of the Url Title page is "Login – RM Pay"
Type into the email field customer's email address
Type into the password field customer's password
Click the Log in button
Check whether the name displayed on the Customer's home page is the Customer's one
Get the testing tool to display the customer's balance amount
Click the Add Funds button
When the new window pops-up, choose to add a funds from a new card
Fill out all new card fields
Fill out the top up amount of $5.12
Click on the Add Funds button from within the form
Get the testing tool to compare the expect balance amount with a resulting balance value
Check whether two values are the same
Log out the customer

3.3 Project Management

For each front-end testing tool chosen for comparison in this Project I followed the following steps:

- Download and install the testing tools into the local folder within the project.
- Download all additional software tools for the testing tool to be able to run on the local machine.
- Set up the configuration file, if such file was provided.
- Create an example test to ensure the testing tools runs in a proper way.
- Build up the "Login Add Funds" test in the small continuous iterations.
- Run the final test and observe the behaviour of the testing tool during the testing process.
- Build up the Docker image for this specific testing tool.
- Run the testing script by using the given testing tool from inside of the Docker
- Implement the script for the given testing tool in the GoCD Pipeline including three stages: build the Docker image, run the test (with a help of newly built Docker container), clean up (remove the Docker image)
- Evaluate the testing tool based on the comparison criteria that have been set.

3.3.1 Comparison Criteria

In accordance with the company's internal needs but also bearing in mind general outcomes of this Project as a further guidance, I have set the following Comparison Criteria:

Easiness of initial set-up and installation

My initial experience with the testing tool has been evaluated, emphasising the installation of the testing tool and amount of additional tools needed for the testing tool to be able to run.

Accuracy of the guidance documents and easiness to understand these documents

My personal experience with the initial run of the testing tool has been evaluated. This is mostly down to ability being able to run the testing script only by the given guidance and whether this guidance (provided on the official website) was satisfactory or further adjustments were needed and the unexperienced developer-tester would find it difficult to run initial script using the given testing tool.

Configuration file

Some of the testing tools have had the configuration file provided. Depending on the definitions in this file, the testing tool could be adjusted to provide for more tailored testing in order to satisfy developer's need for more comprehensive testing. Some of the tools did not provide the configuration file by default.

Difficulty level of writing own testing script

Here's my experience of being able to write the script to follow the steps of each subsequent iteration. If testing tool provided for a possibility of additional (JavaScript) to be injected within the test, how easy was to write and implement the additional script.

Element locators

The elements of the website need to be found by a testing tool through the way of calling the locators. While some of the testing tools could only search for the elements in a limited way, other had various ways of finding the right web element.

Difficulty level of running the script

Some of the testing tools were able to run automatically, others needed to have additional tools running before the test was able to run.

Usefulness of test output

Each testing tool has provided an output in the console from which it was running. In case of a test failure, some of the tools provided a full stack trace of en error while other tools' reports were hard to follow. Some of the testing tools have also provided additional useful information and a various colorful output.

API variety

Each testing tool came with its own API library – set of commands to be used within the testing framework in order to achieve the desired output. While some of the API's were rich and comprehensive, other were very limited and/or difficult to use.

Browser capabilities

Some of the testing tools were only able to run in a headless (invisible to a human eye) browser such as PhantomJS, others were only able to run in a visible browser and there also were testing tools able to run in both types of browsers.

Docker image

Docker images are usually built as simple as possible. When it comes to the testing tools compared here, the Docker images built for each of the testing tool would include the very simple ones but also very big ones. My evaluation was based on a personal experience with building a Docker image, its size and simplicity.

Easiness of running test within the Docker container

Even though the tests were easily running in the developer's laptop environment, some of the tools proved hard to be run in a virtualised environment of a Docker container. This is reflected in my evaluation and I also added a further report describing the obvious difficulties.

Pipeline set up

I set up a separate Pipeline for each of the testing tools. In each of those Pipelines there were 3 stages. In an initial stage the Docker image was built. In the testing stage the test was running in the testing framework of the given testing tool. In a final stage, called clean up the Docker image was removed. My personal experience building the Pipelines regarding the testing tool used is evaluated in this comparison criteria.

Pipeline outcome

Even though the Pipeline was set up properly, each change in the code repository would trigger a new run of the Pipeline. A smooth, failure-free outcome of this run was evaluated in this criteria as much as the benefits of the implemented Pipeline for other developers.

CHAPTER 4: ANALYSIS

4.1 INTRODUCTION

I've written the test in order to run it on the web application the company is building for their customer. Before the test script was written, I've manually performed the test in order to see all necessary steps to make test successful. Then I wrote the testing script in a given testing tool, following the iterations in Chapter 3. Then I did a manual check of the web application to confirm that the test did run in a required way and the console outputs were truthful. Then I built the Docker image, based on the test requirements. Afterwards I set up separate Pipeline for the given testing tool in a way of a separate, tool-related branch. Even though the three Pipeline stages were all related to the given testing tool and the specific test described in Chapter 3, due to the settings in the testing tool, the web application was tested through this Pipeline (meaning that the test would start running automatically only when the changes were made in the local branches).

4.2 TEST OVERVIEW

The welcoming page should display Login – RM Pay:

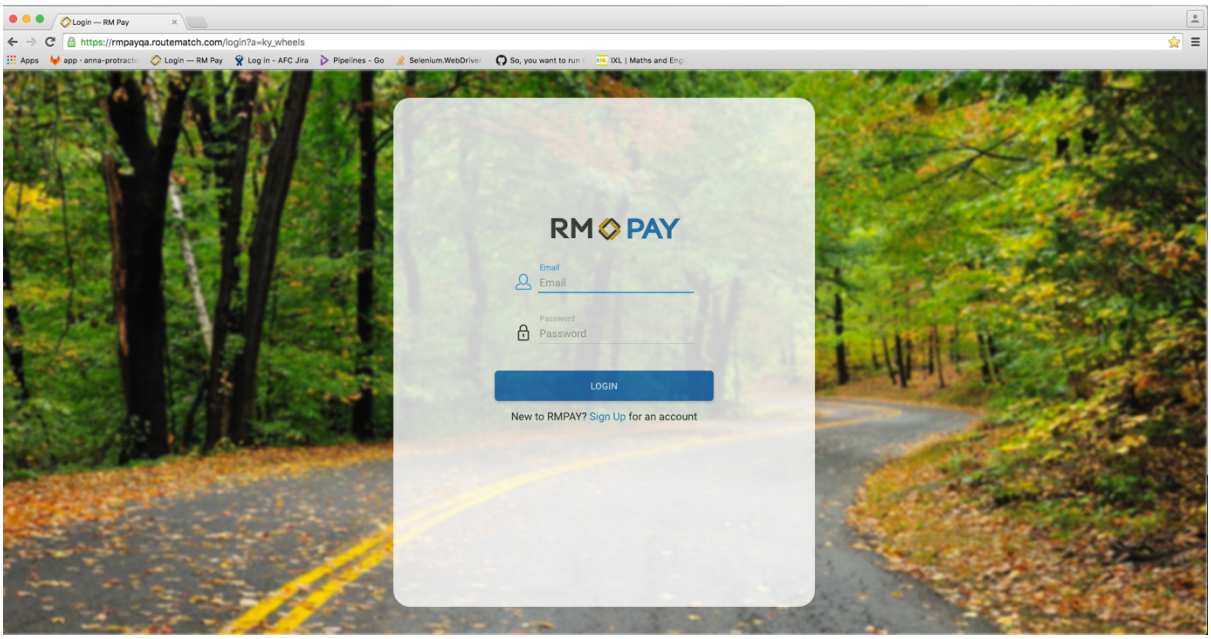

Figure 5 The welcoming page

The login credentials the testing tool should type in automatically:

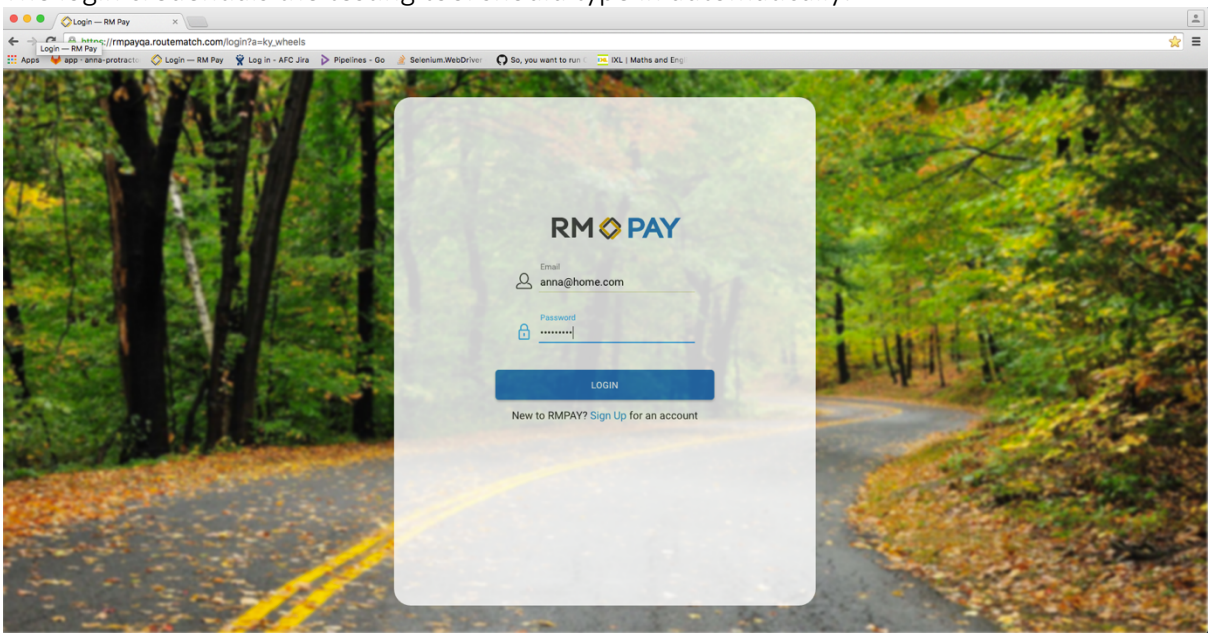

Figure 6 Typing in login credentials

The home page should be showing user's name, the balance amount is taken note of:

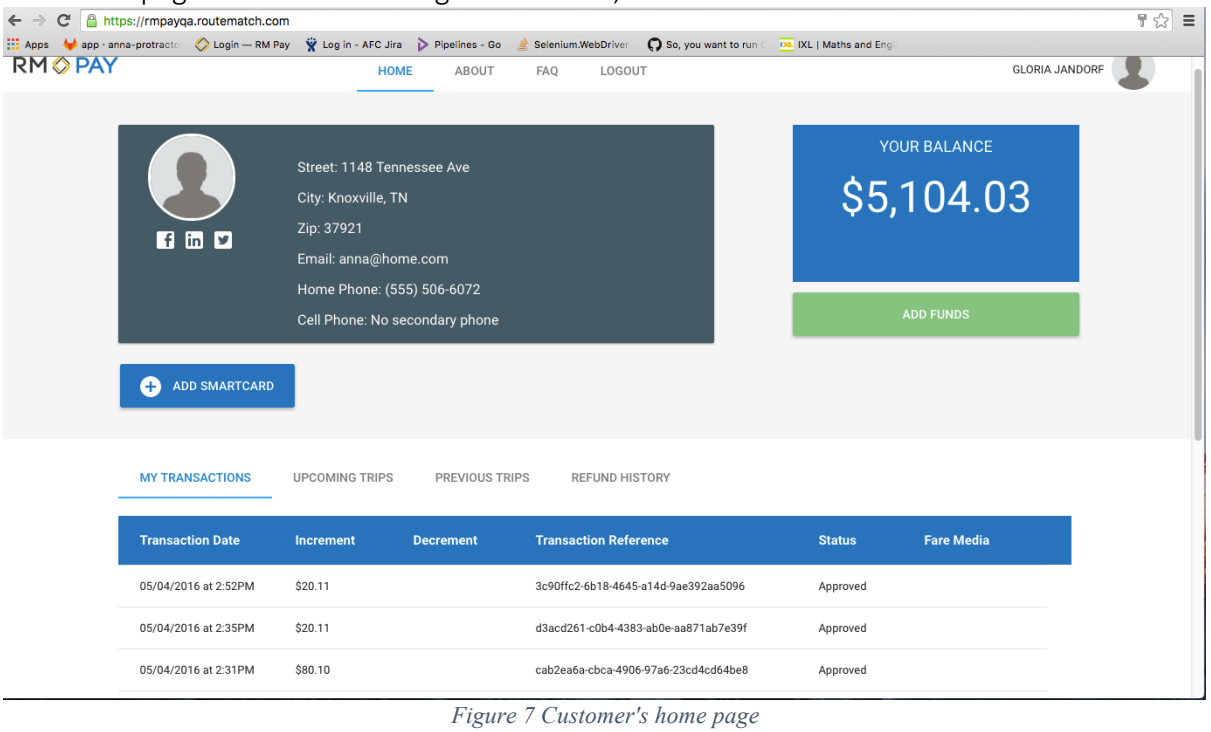

Figure 7 Customer's home page

Clicking on the Add Funds button brings up the form that the tool needs to fill out and click on the Add Fund button. Because of the PhantomJS browser the option to save credit card must remain unticked:

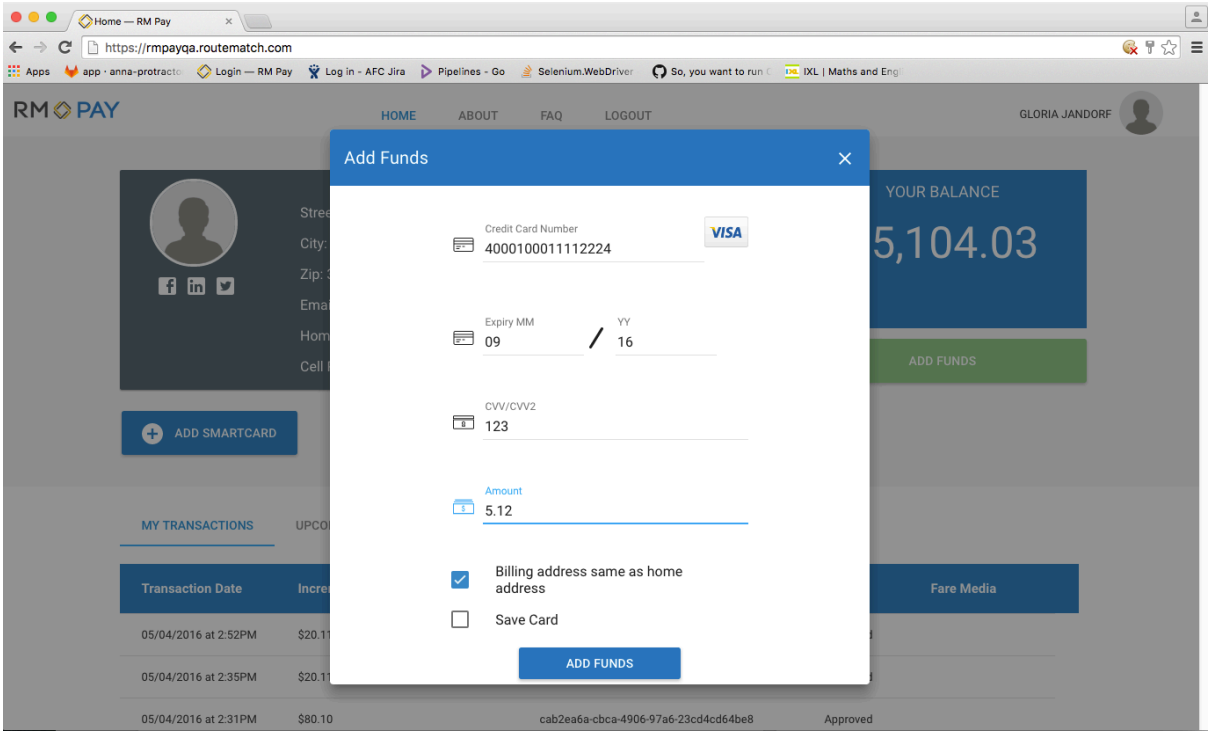

Figure 8 Adding funds from new card

and the balance amount should be updated:

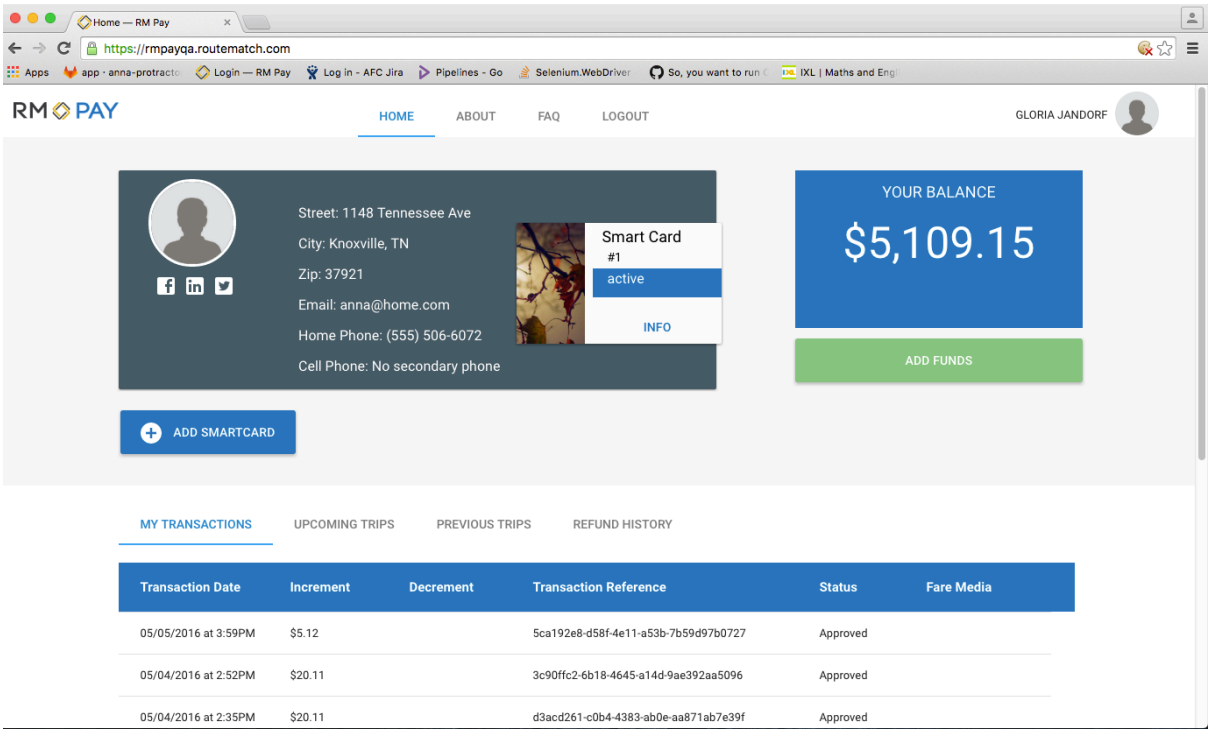

Figure 9 Updated balance

4.3 DOCKERISATION

While it is important to run all automated tests manually by the developer from the local machine, it is also important that the tests can start running automatically, any time the changes are made in the application. For that reason the Docker containers are needed as a service within the GoCD server.

Docker images are build by using a command docker build (-t dockername .) using the build source code from the file called Dockerfile. In this file the first line is FROM, followed by the name of the main image the new Docker image would be built from. Not only Operation Systems such as Ubuntu or Debian can be used, it is also possible to use source image of programming language such as Java or node. The chosen name is followed by the version number or word "stable" or "trusty" that would be referring to the latest stable (or trusty) version of the system. Next, by starting with the reserved words such as RUN, the whole Docker image is configured in the Dockerfile so that it is ready to run as a fully functional, tailor-made container, ready to serve its purpose, in case of the testing tools this includes installing the given testing tools, chosen browsers and its dependencies.

There are few different ways the tests can be running using the Docker containers, for the purpose of the Pipelines it is achieved by running the tests from within the container, so that the tests and all components of the testing tools are running independently from the local machine. There also is an option to run the tests from the image and connecting to the environment of the local computer.

There is also an option to use Docker images that have already been built and they are registered in the public Docker registry called Docker Hub. The public Docker images are free and available for anybody who can pull the images by their name and run the containers from them.

Dockers can also be built as a whole system of multi-container applications through the use of docker-compose.yaml file. In such case, the containers start running by a simple command docker-compose up, which will start the entire application (it'll start a set of running containers providing the services for the application).

Following are the outputs of the command

```
docker build -f Dockerfile -t protractor-selenium-anna .
```

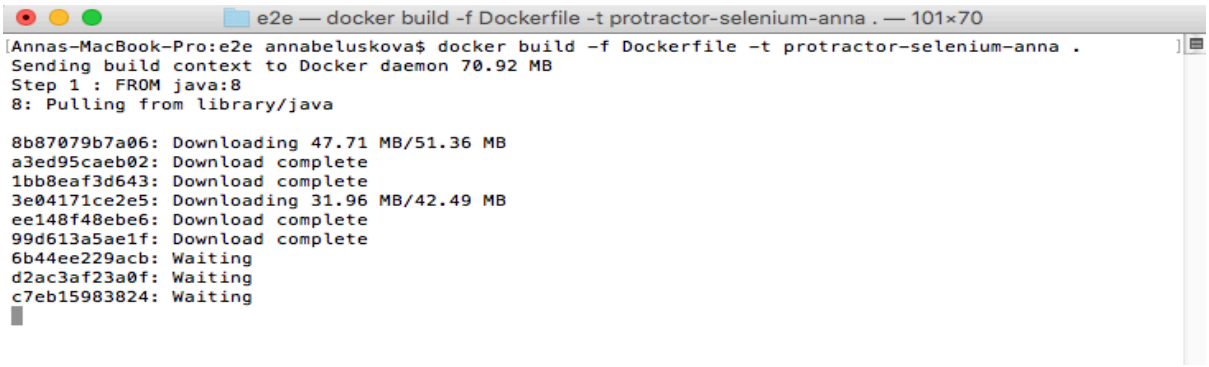

Figure 10 Initial start of building the image

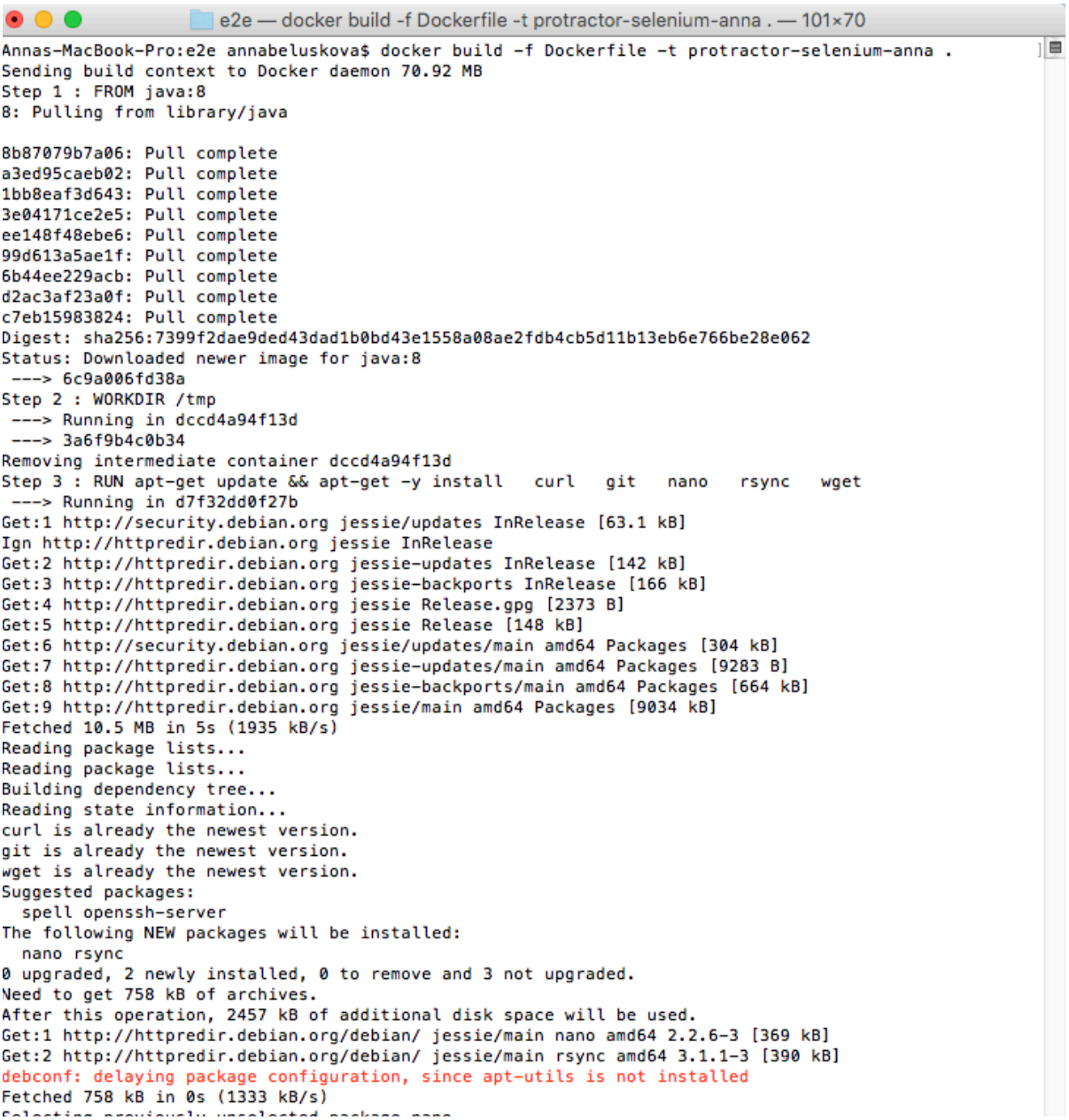

Figure 11 Continuing image build

Running 'docker images' command shows that our image protractor-selenium-anna was successfully built and in order to build it, the other image (java) was pulled from the central docker registry. Without the java image our image wouldn't be complete and it wouldn't be able to run.

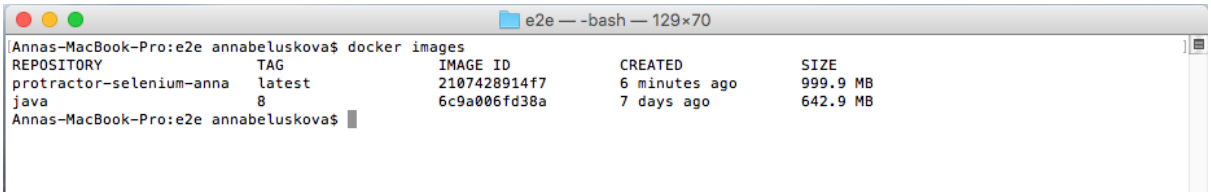

Figure 12 Output of the "docker images" command

4.4 GoCD

The GoCD is a build server for the Continuous Deployment. The verification of the new deployed version is through the series of tests running on the application, including a newly deployed code. Because the GoCD server is a virtual environment of virtual processes, Docker containers are the places where the tests run. Docker images are either built or pulled with each start of the new Pipeline run.

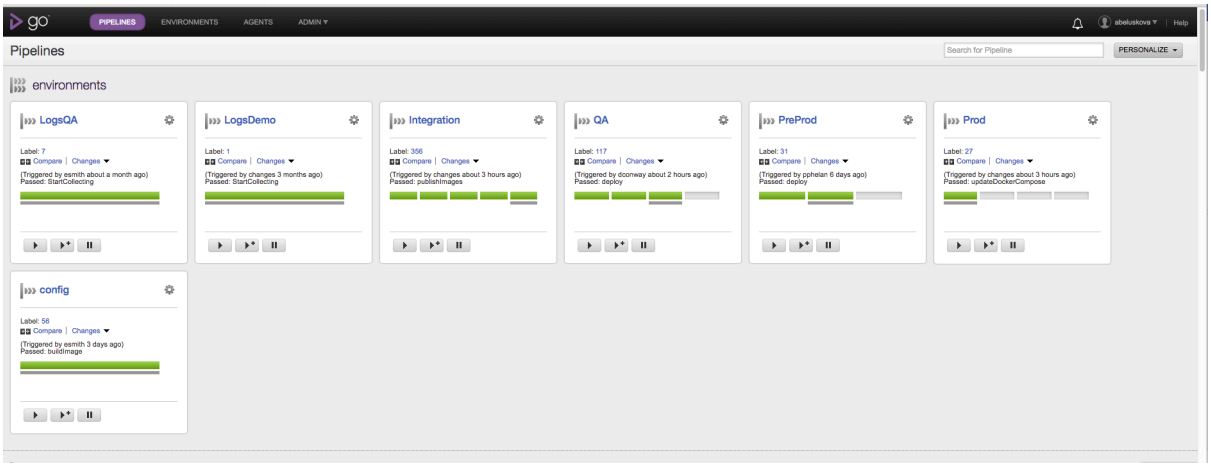

Figure 13 GoCD Pipelines page

A new Pipeline can be set up for any branch of the code repository, specifically relating to the given repository (or given branch). Pipeline consist of one or more stages, generally one stage means one bash command or a set of commands related to the stage. After the stage runs successfully, it turns green at the next stage starts running. The stage output is displayed in the stage's console's output and this is also a source of an error stack trace if the Pipeline (one of its stages) turn red, meaning that the red stage has failed. If the stage (and the whole Pipeline as a result) run fails, the team of developers are immediately notified in the Slack messenger channel.

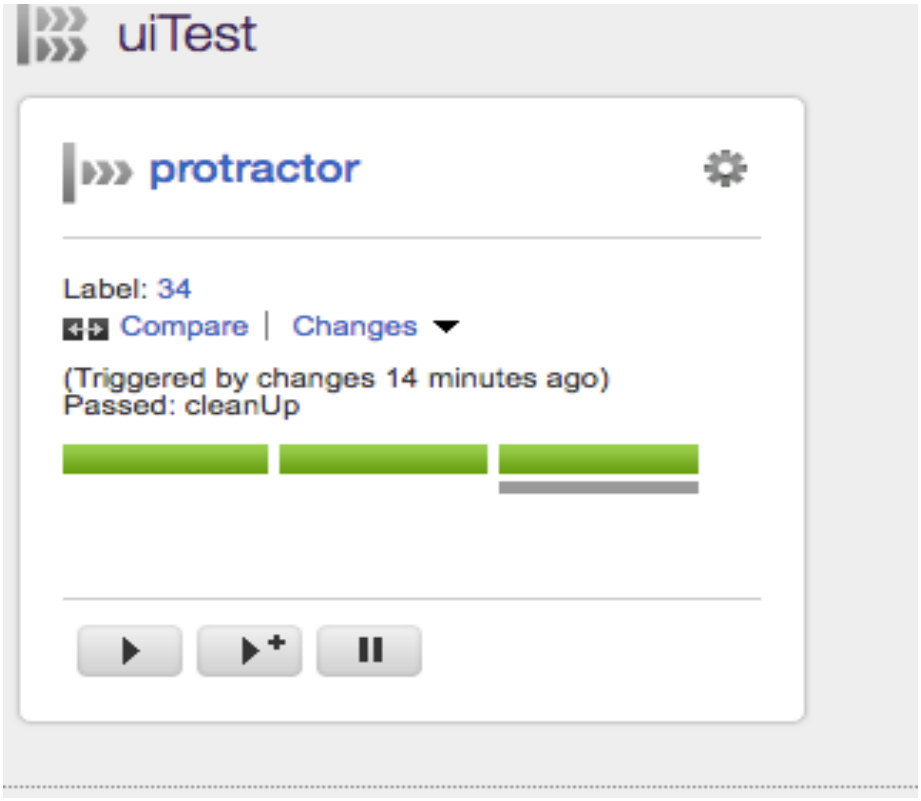

Figure 14 Protractor Pipeline

Each stage's running output is displayed in the stage's own console where it can be followed for possible error stack trace and thus it gives the developers good background information for the error to be fixed.

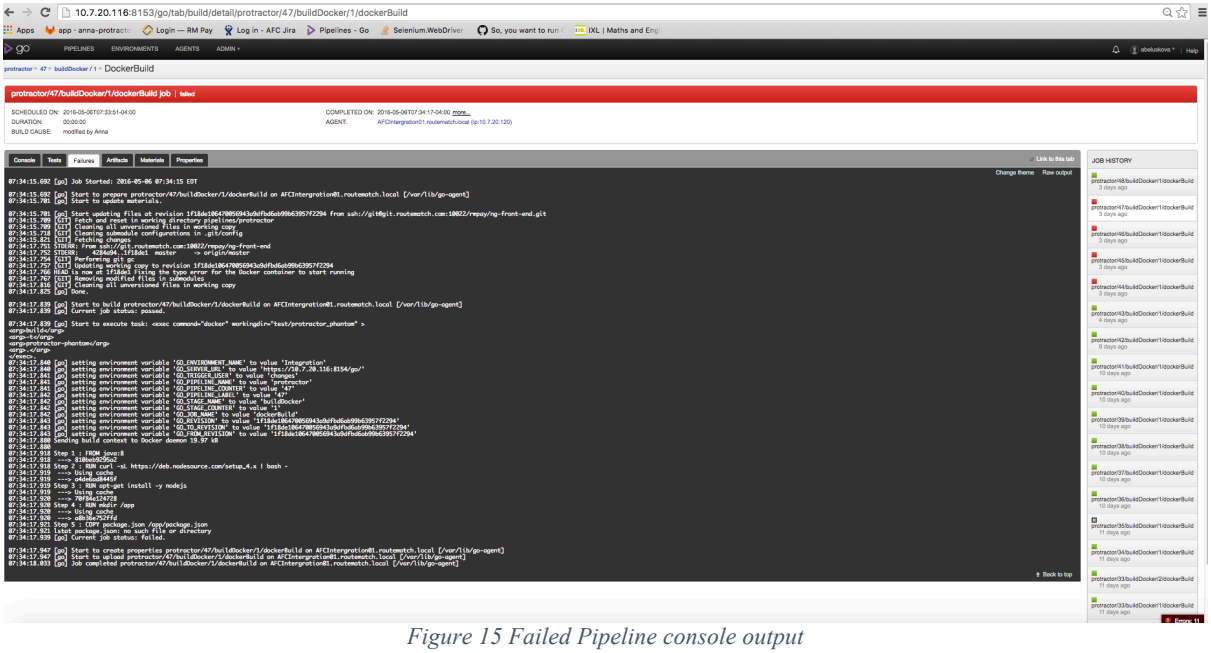

Figure 15 Failed Pipeline console output

4.5 SELENIUM WEBDRIVER

Some of the testing tools needed to use Selenium Webdriver in order to be able to run the tests in the browsers. The next picture shows how the tools get connected:

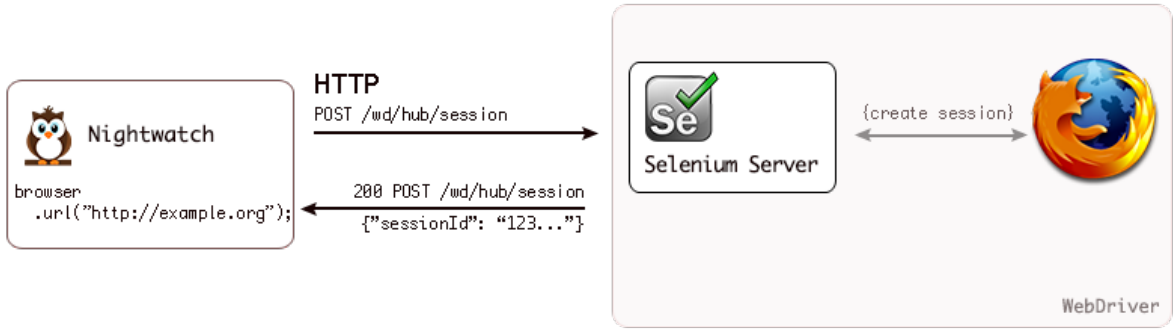

Figure 16 Testing tool talking to the Webdriver which starts the Browser session [7]

When the automation script is executed, the following steps happen:

- for each Selenium command, a HTTP request is created and sent to the browser driver
- the browser driver uses a HTTP server for getting the HTTP requests
- the HTTP server determines the steps needed for implementing the Selenium command
- the implementation steps are executed on the browser
- the execution status is sent back to the HTTP server
- the HTTP server sends the status back to the automation script [14]

And the images below help even more with their analogy to the taxi driver:

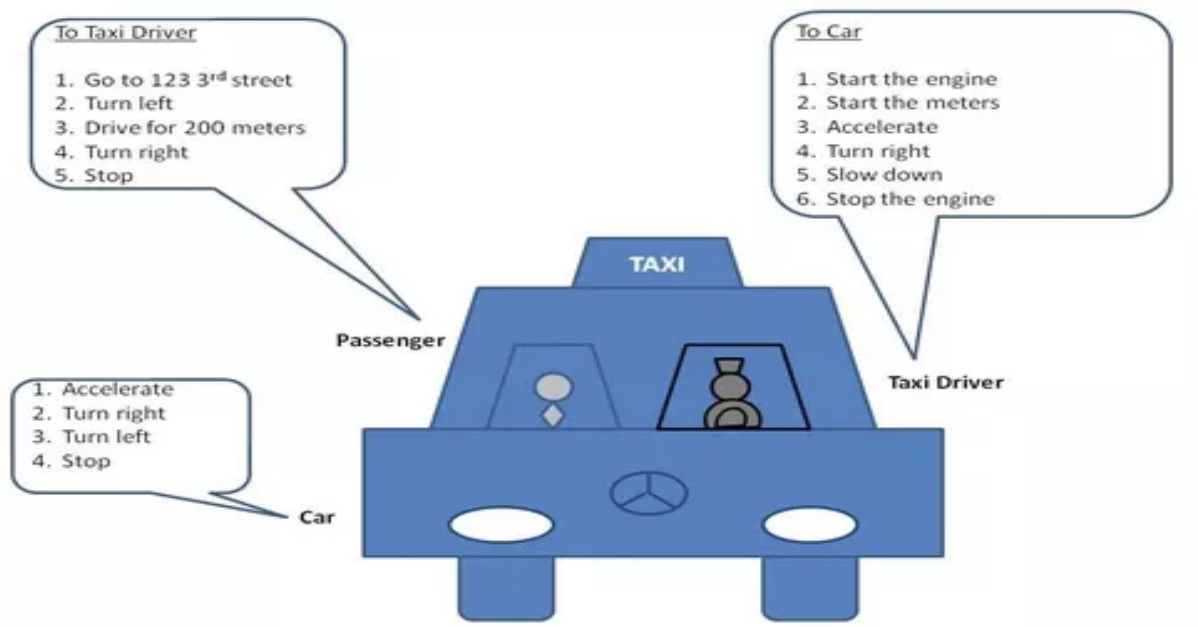

Figure 17 How the client gets to his destination by taxi [14]

- the test engineer is like a taxi client – it provides the instructions to the browser driver
- the browser driver is like a taxi driver – it provides the commands to the browser
- the browser is like a taxi – it executes the commands [14]

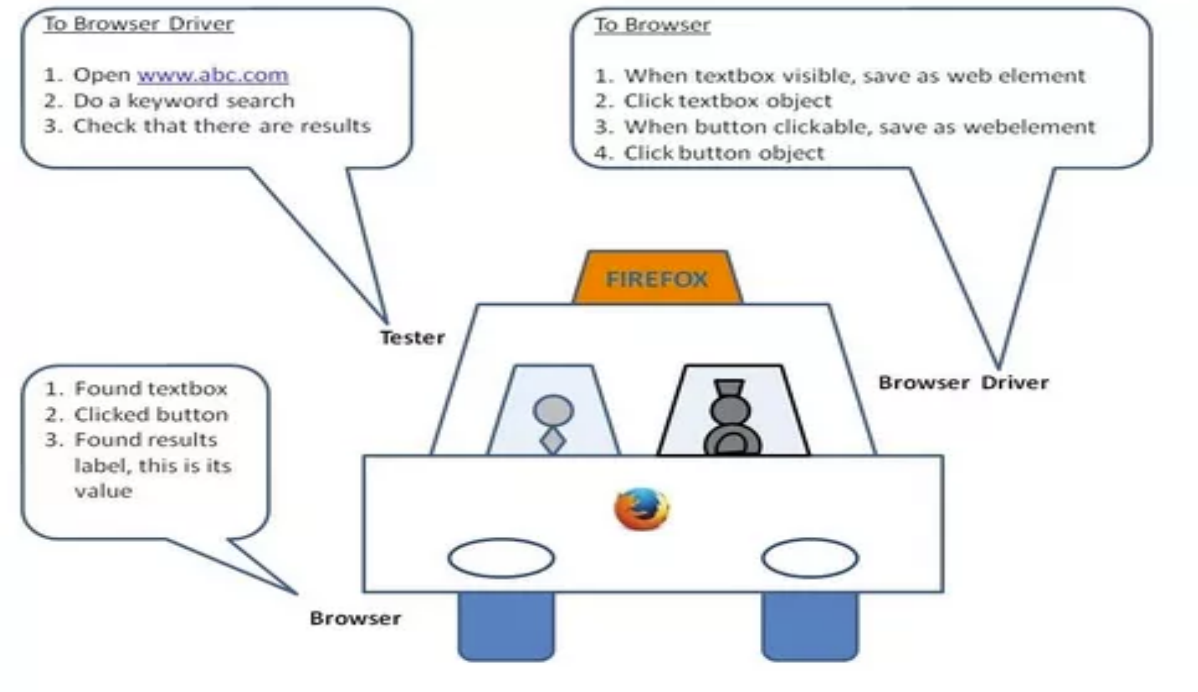

Figure 18 The Selenium Webdriver explained using the taxi analogy [14]

CHAPTER 5: COMPARISON & RESULTS

5.1 TESTING TOOLS OVERVIEW

Selenium IDE

Selenium IDE was rather easy to install and set up. It's a Firefox developer's plugin tool and once a tester becomes familiar with the principles of recoding user's supposed actions, it's relatively easy to use. Few of the configurations and settings can be changed in the Settings options (this can be represented as a GUI-type configuration file). The Selenium IDE is a GUI type of the testing tool where the main aim is to manually record a required action and then it can be repeated as often as wished by clicking on a Play button. The API is given as a list of options to choose from in a scroll down menu. The console output gives enough information for a developer to discover a possible failure. There is a variety of options how to find an element and these options can be customised in order to give a priority to a certain element attribute (such as id) over other locators (such as xpath). Selenium IDE can run in a choice of visible (only) browsers, including Android, Chrome, Firefox, Internet Explorer, iPhone and iPad.

However, this option requires a use of a Selenium Web Driver and is not fully supported yet. It can reliably run in the Chrome and Firefox browser (from a Mac laptop), depending on the settings, but a default browser is Firefox.

The biggest disadvantage of this testing tool is that its based on a GUI interface and recording possibilities and while this might be handy for a non-experienced user, a developer might this way of testing very annoying. Especially because when test is recorded, it can rarely be simply played back, most of the time additional amendments need to be done. A typical example is to add in waiting times for a page to be loaded or automatically generated id's are picked up on and they need to be replaced by something unchangeable. For this reason, which is based on a large amount of time wasted by modifying an already recorded test rather than simply typing up steps to achieve the desired testing action, I have soon abolished this tool and its implementation in Docker and the Pipelines has not even been considered.

However, it is a great learning tool for all beginners to help them understand how the automated front end tools work and which steps need to be followed in a particular testing script. Another interesting feature is to schedule the recorded (and saved) tests for a repeated run at a set time and day giving a one week scheduling calendar. I have not checked this tool for its behaviour on repeated elements and whether such elements can be selected in no particular order or choice (such as the third one or the n-th one by a choice).

DalekJS

DalekJS was very easy to set up and run. Its installation was easy, with a help of the npm it was just a one step process. I've written the first testing script in no time and there weren't any further difficulties to run the script. Its very simple API was easy to understand and use. The console output was also easily readable and understandable, the colourful output included a

green or red bar in test result. A web element can be found by its id. DalekJS runs in a headless browser (by default), but can run in Chrome, Internet Explorer, Firefox and Safari when adding an additional argument in console. The configuration file is not set by default, but it can be created in the folder and few of the configuration settings can be passed on with this file. The Docker image was easily build and the Pipeline tests were also running successfully.

DalekJS was a testing tool used by the RouteMatch at the time of my start in the company and one initial script (to log in the customer) was running in the Pipeline. However, I have soon discovered that DalekJS is buggy and gives falsely positive results. Meaning that when (for example) two variables were compared, the final assertion has found them equal, even though they were different. Following a further examination into this matter I have discovered that some of the methods from the API library are not working. Thus, the further use of this testing tool was abolished by me and by the company

CasperJS

CasperJS had an installation page on its website that had guided through the process of installation and along with writing and running initial (example) test script has made an initial experience a very pleasant one. CasperJS has a rich and various API library, which make this tool really simple to use, especially when the elements can be found by its id, css selector or by xpath. CasperJS has no configuration file and only few of the default settings can be changed thorugh the additional command line arguments. On other hand, the tool offers a variety of colourful console outputs to choose from. CasperJS can only run in a headless browser, either PhantomJS or SlimerJS have to be installed in a local folder of the testing tool. The Docker image was easily build and the Pipeline tests were also running without a problem.

The biggest disadvantage of the CasperJS testing tool is that it can only run using a headless browser and a lack of configuration file. Due to a fact that PhantomJS is used has initially led the author of this Project into difficulties because of the PhantomJS browser not being able to clear its local storage by default after each test run. As a result, the local storage has kept the previous session open even after the test has stopped, thus the customer would be kept logged in and the welcoming page could not be retrieved. It is possible to manually delete the local storage file, or to write a simple bash script to resolve this issue and eventually, this problem has been solved. However, it would be appreciated if the authors of this tool have had the reference to this issue somewhere on their webpage as this was a significant issue and took time to resolve. It was also very difficult and cumbersome to write author's own testing script.

There are also two different ways to write a testing script, which results in two different ways of running the testing scripts, which was very confusing . Other disadvantage of this testing tool is that it only runs in a headless browser which behaves slightly different as oppose to the visible browsers. Even though it might be advantageous to use a headless browser in the virtual environment of a Continuous Integration, companies like RouteMatch would prefer a testing tool that helps to spot the bugs that a real user can come across.

Protractor

Protractor provides a full guidance not only towards the installation steps but also offers a short tutorial on how to write the first test and set up the configuration file. Even the example test was referring to its own simple website, which made the whole initial experience

delightful. The API library is big and more than one reserved words can be concatenated to create one command. The elements can also be found (located) in different ways and in case of a repeated element, exact place of repeated element can be chosen.

The console output was rather simple (no rich colours), a test success is marked by a simple green dot, a failure by a red F, but a full stack trace of a failure is provided, which makes errors easy to spot and resolve. Protractor comes with a direct reference to a configuration file, in which all important options can be set. In this file there's also a list of all testing scripts (files), which then run by using the config file as a default file to run. This makes the tester's life a lot easier as only chosen script(s) can run.

Protractor declares to run with all most widely used visible browsers, but a headless PhantomJS can also be used. The Docker image was successfully built and the Pipelines created and the chosen tests can run automatically whenever the changes in the code repository are made. A big advantage was the possibility to build the page objects – an own library of the script that was repeating in different places of the script. Thanks to the page objects, the repeated code was only written in a separate file within the page folder and the one-line reference to this script in the main testing script would allow for the whole repeated script to run as a part of the test. The Selenium web driver needs to be started before the tests can run, however, the newest version of Protractor starts the web driver automatically with a start of the test.

Even though Protractor was a rather pleasant tool to work with (especially due to visibility of the test and a full error stack trace), sometimes the default behaviour was impossible to use and different ways of achieving same action needed to be sought. This was especially case when the third party elements were plugged into the application and a certain way needed to be implemented for the Protractor to deal with such elements. A typical example was when the widely-used command .click() didn't work in same places of the application and a lot more complex script needed to replace the simple click() command in order to achieve the same mouse-click simulation of the user's behaviour.

I wasn't able to run Protractor tests in all browsers that are listed in the official documentation. I need to do more investigation to find out whether this is an unresolved bug within the Protractor or the developer's environment needs to be modified.

The other difficulty was also to build a Docker image that would be able to run a headless chrome in sync with Selenium web driver manager. This issue was solved by choosing PhantomJS browser for the Protractor's implementation in the virtual environment of the GoCD server.

NightwatchJS

The installation of this testing tool was very simple, the node package manager (npm) would do it in one simple step. The configuration file is clearly set on the first page and the sample tests are easy to follow too. The API referencing is well divided and easy to understand and follow. It was also easy to write own testing script and include a JavaScript script in it. The elements can be officially found in two different ways, but additional, simpler locators were working too. NightwatchJS runs in three visible browsers (chrome, Firefox, IE) and doesn't offer an option to run headless.

There are few positive extras packaged with this testing tools. One is, that same as Protractor, page objects can be set up to avoid repeating same code. This helpful methodology was very welcome in the long, comprehensive tests, when the parts such as login customer, are repeated very often. The other positive thing about this tool was that the selenium reports are automatically sent to the text file which came handy when the error stack trace needed to be inspected.

The above mentioned lack of option to run the tests in a headless browser have caused difficulties to run this test from within the Docker container. Due to lack of time and the docker-unsolved-issue constraint, the Pipeline set up has not been tried yet. The other downfall of this testing tool is that the tool would automatically start running all tests from the folder, even though only one of them would be specified to run in the command. Also the instructions about how to run the tests were unclear and it was only due to author's experience from other tools that they would eventually run. Another disadvantage was that on order to run the tests, a web driver manager needed to be started beforehand, from a separate console window.

5.2 Console outputs

Following were the console outputs from the running tests:

CasperJS

A not fully successful test run from the local machine:

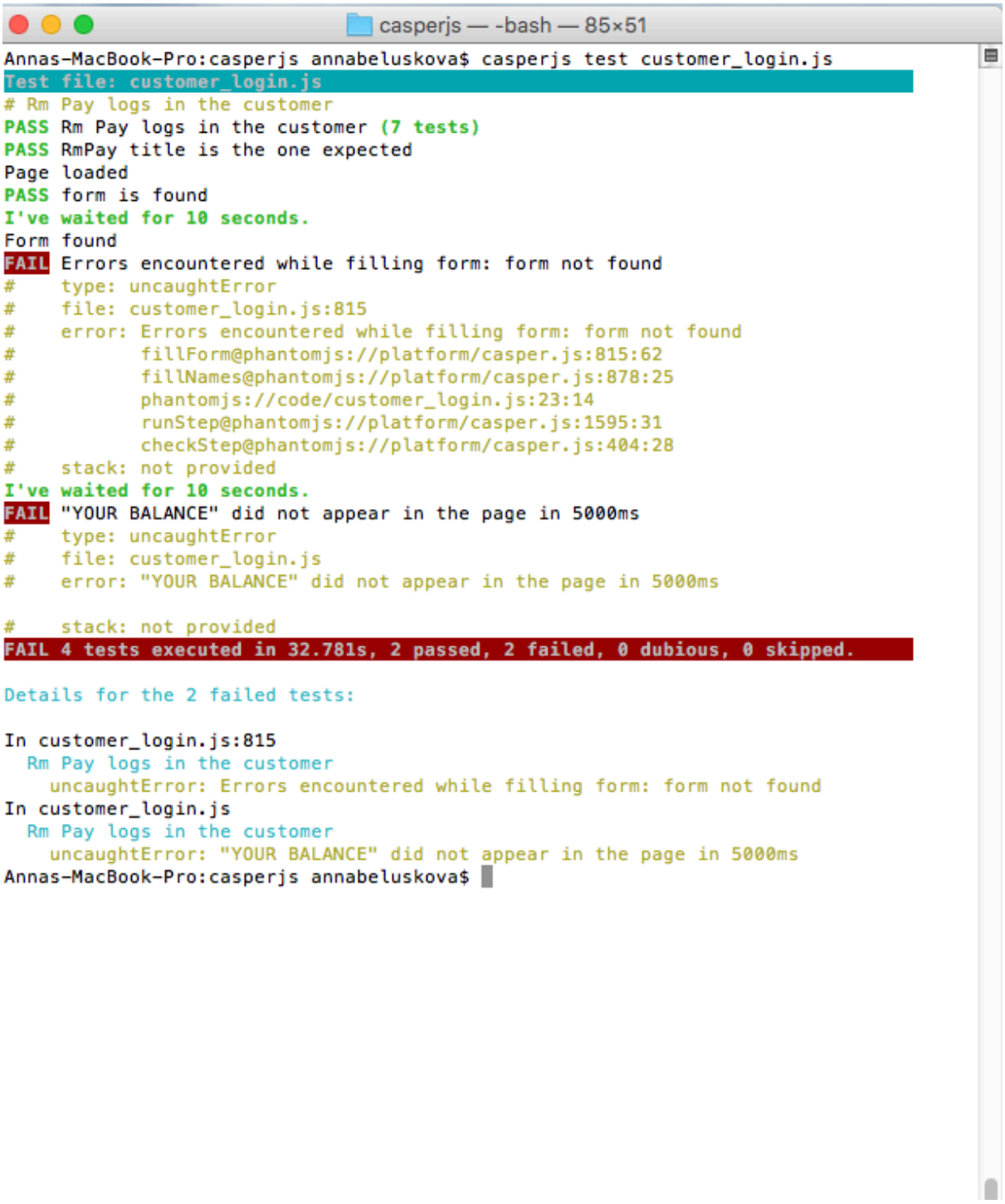

Figure 19 CasperJS console ouput: failed

NIGHTWATCHJS

A successful test run in Chrome from the local machine:

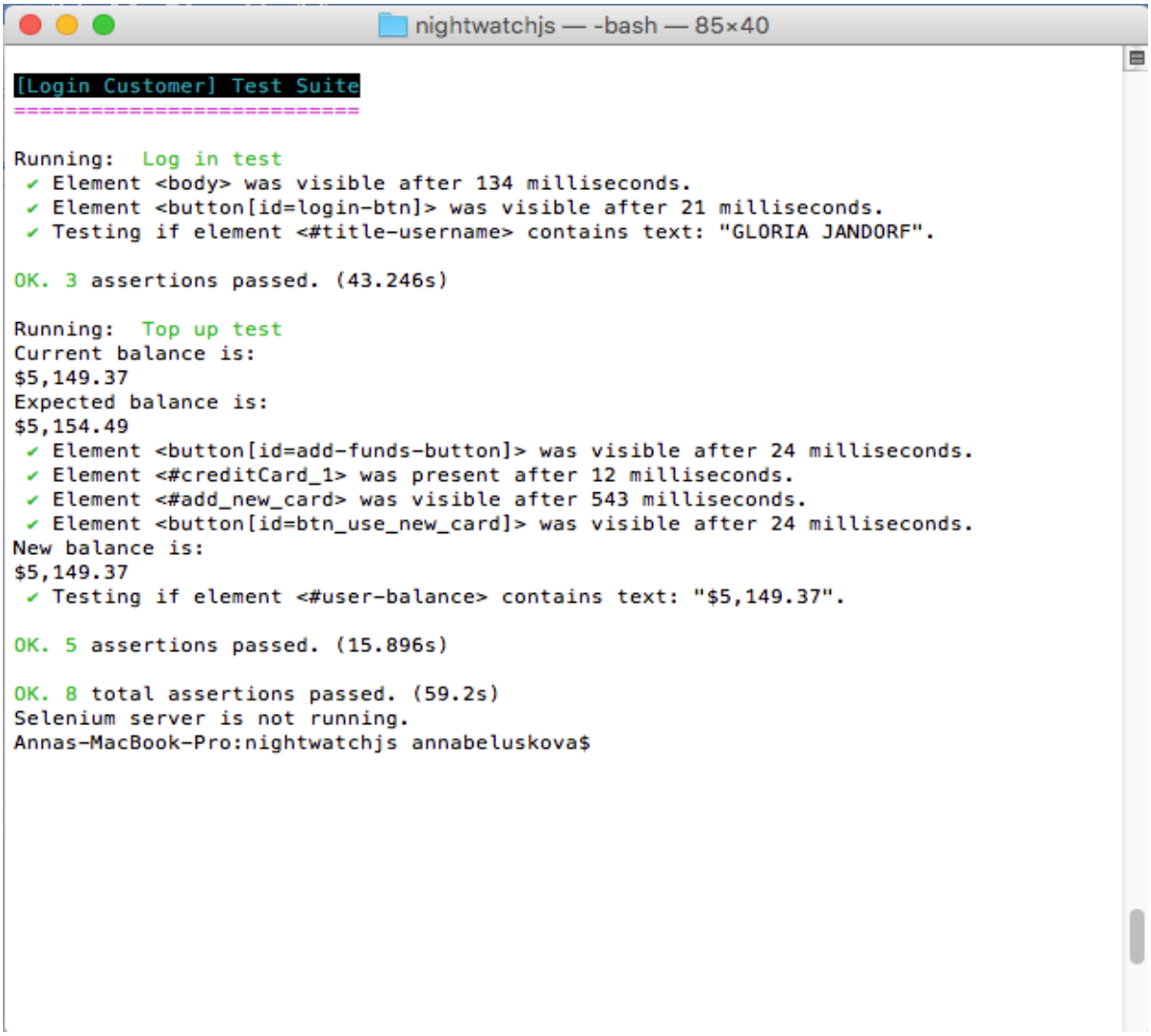

Figure 20 NightwatchJS console output: passed

5.3 COMPARISON TABLE

The comparison table below displays comparison results with a 5-star rating system.

	Selenium IDE	DalekJS	CasperJS	Protractor	NightwatchJS
Installation	****	*****	*****	****	*****
Guidance docs	*	****	****	*****	*
Config file	****	**	*	****	*****
Writing scripts	*	****	*	*****	****
Element locators	****	*	***	*****	*****
Running scripts	**	*	**	*****	***
Test outputs	*	***	***	***	*****
API variety	****	**	****	****	*****
Browsers	**	***	*	***	***
Docker image	N/A	*****	*****	****	***
Docker container	N/A	*****	*****	****	N/A
Pipeline setup	N/A	*****	*****	****	N/A
Pipeline outcome	N/A	*	*	****	N/A

Table 1: Comparison results

5.4 UNRESOLVED ISSUES

A problem has occurred while running the testing tools from inside the Docker container, thus simulating the GoCD environment.

The issue is related to the fact that the chrome is a visible browser, so it needs to have a display output related to it. The solutions seems to be a use of the Virtual Frame Buffer Tool (xvfb) but on top of that, the selenium browser needs to be running, all of which together is hard to achieve inside of the virtual environment of the Docker .

The other possible problem might be small virtual memory size of the Docker container. However, the nightwatchJS creates a selenium output file, from which it is obvious that the selenium server is running and so is the chrome/Firefox browser, it only seems that they can't reach outside of the container.

More investigation into this problem is needed and due to time constraint it was impossible to find a solution before the closure of this Project. However, I'm planning to keep working on it, until the solution is found.

CHAPTER 6: CONCLUSION

6.1 COMPARISON RESULTS

As a result of this comparison Project, I would strongly advise against using front-end testing tools that run in a headless browser such as PhantomJS. The main reason is not only that the test performance is not visible to the user (which can be bypassed by the screenshot commands in the testing scripts), but also because the PhantomJS browser behaves differently than a visible browser such as chrome and thus it gives falsely negative results of the tests and the user's possible experience with the application.

This has been observed when the Protractor's settings were changed from the Chrome to the PhantomJS and while in the Chrome the test would run without the failure, in the PhantomJS it couldn't deal with certain angular code that works in sync with the browser's (chrome) tools.

The other important aspect for the testing tool is to have the flexibility to run the tests in the different browsers, because again, even though the application might be performing well in the Chrome browser, this doesn't have to be the case for other browsers and a good company should provide for the application to be able to run in all available (visible) browsers.

While other aspects of the testing tool (such as whether the error stack trace is provided and it is useful or how hard it is to locate elements with the given testing tool) need to be taken into consideration, a special attention is needed for the ability of the testing tool to run in the virtual environment of the Continuous Integration tools.

As a result of the above-mentioned parameters, the Protractor has been chosen to be used for the front-end automated tests and further, more comprehensive scripts have been written and put into company's use. These scripts are now being used by the company and they do find a lot of bugs in the frontend environment that otherwise would have not been spotted so quickly. Even though the manual run of the fully functional tests needs to be preformed in order to gain full benefits of the tests.

The testing scripts for each of the testing tools, along with the Dockerfiles are attached to this Project in the Appendix section.

6.2 KEY SKILLS

Following new skills have been developed during the work on this project and the Internship:

- JavaScript programming language – how to write scripts in this programming language

- AngularJS programming language – the principles of the Angular framework, how it fits within JavaScript and Angular specific language

- Protractor testing tool – how to write and run testing scripts to perform any kind of an end-to-end functional testing, primarily in Protractor. The gained knowledge and skills were helpful when other testing tools were used. I have also learned how to set up the configuration file and how significant the setting files are.

- Synchronous and asynchronous executions – in the synchronous executions the tasks are performed in an order and the first task must be finished before the next task in a queue can start running. Asynchronous execution means that more tasks can start running independently of each other, so the second one can start before the previous one has finished. This was very important thing to remember from the Protractor point of view, as it's based on and asynchronous execution.

- Write a testing script for the functional testing – what are the basic parts of a functional test that have to be followed in each testing tool. How a JavaScript script can be inserted into tools' own framework library. Each test should be finished with an assertion call, which compares a real value and an expected one.

- Virtualisation – Docker and GoCD are both based on a virtualisation, which means that the processes don't run on the local machine, but on a virtual machine. This means that each part of the process needs to be virtualised, too. This was important thing to remember when the tests were running inside of a virtual Docker container and thus a use of "headless" browser was necessary.

- Docker tool – how to build a tailor-made Docker image from scratch. How to run Docker containers and how to get "inside" of this image and make further inspection or adjustments on it. How to run processes using Docker containers and how to run processes from inside of the Docker container. How to modify basic settings of the Docker image such as to increase its shared memory size. How to get Docker to use the local machine's resources. How to create a local docker hub and link docker containers together. How to build the docker compose system. How to check the list of docker images and running dockers. How to stop the running dockers and how to delete docker images from the local machine, including those with the child images and/or running containers. How to use the Docker Centralised Registry. How to build and use the Dockers for the front-end testing tools.

- GoCD – how to set up GoCD tool from scratch. This means to set up a Server, an Agent and to create the Environment. How to create a Pipeline so that it is specifically related

to a specific part of the code repository. How to create stages in the Pipeline and how to assign them a job (a task) to do. How to create a full series of inter-related stages in one Pipeline. How to investigate and fix a failure in the Pipeline.

- Bash scripting – even though I've used bash command line before, I've learned how powerful bash scripting is. More importantly, I have learned how to create bash script files which contain a command (or a list of commands) to be executed from the terminal. Instead of writing up a long command or set of commands, a filename is written in the bash terminal and this will execute the command(s).

- Error stacktrace – error stacktrace gives an important information to a developer, stating what went wrong and where exactly it went wrong, which part of the code failed and why it failed. I've learned how to read this stacktrace to find a cause of a problem, which meant that I was able to quickly resolve the issue.

- Ssh login – there is set of Agents performing assigned jobs in the GoCD server. Some of those Agents would occasionally fail and in order to fix the failure, it was necessary to remotely login to the Agent's machine's IP address. I've learned how to remotely login to these machines and execute further commands to fix a problem which caused the machine to stop running and/or performing their tasks.

- Robomongo – I've learned how to use this tool to access the mongo DB database in order to manipulate the data used for testing purposes. This includes, but is not limited to, how to find, edit, delete or update the entries in the mongo DB database tables.

- Git working with branches – each of the testing tool had its own branch so that the code wouldn't affect the performance of the application's front end interface. While pushing own testing script code to the branch, it was also important to keep up to date with the main (master) branch. I've learned how to merge in the master code into the branch, while leave the branch code outside of the master.

- SublimeText packages – I've read and written all scripts in SublimeText. SublimeText is a powerful text editor which basically displays a testing script in different colours based on the filename extension. However, additional packages can help a developer to compile the script with regards to the prescribed syntax of a given programming language. I've learned how to install and use packages to spot the JavaScript syntax errors so that they can be fixed before the script is executed.

- JIRA – Jira is a software (project management) tool that reflects the team's Scrum methodology. Each new task is created as an Issue and based on its meaning it can be

a Story, a Bug, an Epic or a Spike. In the Story there are additional entries which define the work Done. I've learned how to use the JIRA and all of its components.

6.3 Further work

As mentioned in the paragraph 5.4, the possibility of running the Protractor tests in the "headless' chrome has still not been optimised and I intend to keep working on the issue until it is resolved. This includes gaining more in-depth knowledge of the Docker environment.

A further investigation is also needed for the Protractor tool not being able to run in other than the two currently available browsers (Chrome and Firefox). I'm hoping to look into the matters in the next few weeks.

All further updates will be available on the website that I'm planning to create, using the Ionic framework. The website should provide all important details about the testing tools mentioned in this Project, along with the detailed description of all issues that were found and how they were resolved. There should also be an option for the registered users to add their comments to the list of known issues or to give their review of the testing tool. Following the website's functional skeleton, I'm hoping to create a mobile application with the same range of functionality.

Another interesting task that I'm planning to start working on (and eventually to implement it in the developer's environment), is a tool called SonarQube. This tool automatically checks the code for a quality of a new code in a process called Continuous Inspection.

6.4 Conclusion

All parts of this research project are my own work, including the code scripts and the findings.

The Project goals have been met and the chosen testing tool has been proven to be beneficial to the company for which it has been tried, tested and chosen to write and run further functional tests. A simple internal wiki doc has been written for other colleagues to be able to install and run Protractor tests from their local machine and modify the settings to suit their own testing needs.

The tests now can run by automatic trigger in the GoCD Pipeline, and they also are widely used by the developers to be manually run from their local machine and thus provide for a full functioning verification of the application.

Docker and CI tools (such as GoCD) are currently the latest and most popular technologies that most companies are on a journey to embrace. The author of this Project is delighted to have had the privileges to learn how to use them with the hands-on experience and hopes to continue in the learning process until the outstanding issue is fully resolved.

BIBLIOGRAPHY

1) Google, 2016, "AngularJS — Superheroic JavaScript MVW framework," in AngularJS - Superheroic JS*, 2008. [Online]. Available: https://angularjs.org/. Accessed: May 9, 2016.

2) Joyent, Inc, 2016, "Node.js," in Node.js, 2016. [Online]. Available: https://nodejs.org/en/. Accessed: May 9, 2016.

3) K. Schwaber, Scrumguide. Scrum Alliance, 2009.

4) S. Golasch, "Automated cross browser testing with JavaScript,". [Online]. Available: http://dalekjs.com/. Accessed: May 9, 2016.

5) N. Perriault and C. Contributors, "CasperJS, a navigation scripting and testing utility for PhantomJS and SlimerJS," 2011. [Online]. Available: http://casperjs.org/. Accessed: May 9, 2016.

6) "Protractor - end to end testing for AngularJS," in http://angular.github.io/protractor/#/. [Online]. Available: http://angular.github.io/protractor/#/. Accessed: May 9, 2016.

7) Nightwatchjs.org, "Nightwatch.js | Node.js powered End-to-End testing framework," inNightwatch.js, 2015. [Online]. Available: http://nightwatchjs.org/. Accessed: May 9, 2016.

8) M. A. Awad, A Comparison between Agile and Traditional Software Development Methodologies. Australia: The University of Western Australia, 2005.

9) admin, "Mobile App development and web design Glasgow | digital design Glasgow," Screenmedia, 2016. [Online]. Available: http://www.screenmedia.co.uk. Accessed: May 9, 2016.

10) Cycosys is a leading provider of software development services and innovative IT solutions," 2015. [Online]. Available: http://www.cyclosys.com/Practices/MethodologiesFramework. Accessed: May 9, 2016.

11) M. Fowler, C. Scientist, and ThoughtWorks, "Continuous integration," 2016. [Online]. Available: https://www.thoughtworks.com/continuous-integration. Accessed: May 9, 2016.

12) Docker, "Docker," Docker, 2016. [Online]. Available: https://www.docker.com/. Accessed: May 9, 2016.

13) "The poor state of software testing today," 2015. [Online]. Available: http://dslab.epfl.ch/blog/2015/07/22/the-poor-state-of-software-testing.html. Accessed: May 9, 2016.

14) A. Sim, "How does the Selenium WebDriver work?," in *Quora*, 2016. [Online]. Available: https://www.quora.com/How-does-the-Selenium-WebDriver-work. Accessed: May 12, 2016.

15) B. Rice, R. Jones, and J. Engel, "Behavior driven development — behave 1.2.5 documentation,"2014.[Online].Available:https://pythonhosted.org/behave/philosophy.html. Accessed: May 12, 2016.

APPENDICES

TESTING SCRIPT WRITTEN IN DIFFERENT TESTING TOOLS:

1. DALEKJS

```
module.exports =

{
  'Adding funds from existing card': function(test)
   {
    var url = 'https://rmpayqa.routematch.com/login?a=ky_wheels';
    var name = 'GLORIA JANDORF';
    var balance_after;
    var payment = 5.12;
    test
      .open(url)
      .type('#login-email', 'anna@home.com')
      .type('#login-password', 'XXXXXXXXX')
      .click('.login-btn')
      .wait(3000)
      .assert.text('#title-username').is(name)
      .log.dom('#user-balance')
      .execute(function()
      {
        this.data
        ('balance_before', parseFloat((document.getElementById
        ("user-balance")
        .innerHTML.slice(1)).replace(',','')));
      })
      .log.message('Current balance is:')
      .log.message(function()
      {
        return test.data('balance_before');
      })
      .execute(function()
      {
        this.data
        ('balance_after', (Number
        (parseFloat((document.getElementById("user-balance")
        .innerHTML.slice(1)).replace(',','''))) + Number(5.12))) ;
      })
      .log.message('Topped up: $5.12')
      .log.message('Expected new balance is:')
      .log.message(function(){
        return test.data('balance_after');
      })
      .click('#add-funds-button')
      .wait(5000)
      .click('#creditCard_1')
      .wait(5000)
      .click('#add_new_card')
```

```
        .wait(5000)
        .type('use_new_card_1','4000100011112224')
        .type('use_new_card_2','09')
        .type('use_new_card_3','16')
        .type('use_new_card_4','123')
        .type('use_new_card_5','5.12')
        .wait(5000)
        .click('#btn_use_new_card')
        .wait(5000)
        .log.message('Funds added from a new card')
        .log.message(function()
        {
          return test.data('balance_after');
        })
        .log.message('New balance is')
        .log.dom('#user-balance')
        .assert.val('#user-balance', balance_after);
        done();
    }
};
```

2. CASPERJS

```
var mouse = require("mouse").create(casper);
var colorizer = require('colorizer').create('Colorizer');

casper.test.begin('Rm Pay logs in the customer', 7, function
suite(test) {
    casper.start('https://rmpayqa.routematch.com/login?a=ky_wheels',
    function()
    {
        test.assertTitle("Login - RM Pay",
        "RmPay title is the one expected");
        console.log("Page loaded");
        this.test.assertExists('form', 'form is found');

    }, true);
  });

casper.wait(10000, function() {
    this.echo("I've waited for 10 seconds.", "INFO");
});

casper.waitForSelector ("form", function()
 {
    this.echo('Form found');
    this.fill('form#login-form', {
       'email' : 'anna@home.com',
       'password' : 'XXXXXXXXXXXX'
    }, true);
});

casper.wait(10000, function() {
    this.echo("I've waited for 10 seconds.", "INFO");
});
```

```
casper.waitForText("YOUR BALANCE”, function()
 {
    this.echo('Home page redirected');
 });

casper.waitForSelector("#title-username", function()
{
   this.test.assertExists('#title-username');
});

casper.then(function(){
    this.test
    .assertTextExists('GLORIA JANDORF', 'home page is displaying name
     Gloria Jandorf');
});

casper.then(function getText() {
    var target = this.fetchText('#user-balance');
    this.echo(target);
    var payment = parseFloat((5.12).toFixed(2));
    var balance1 = balance = target;
    var balance_before = parseFloat(balance1.slice(1).replace(',',''));
    var balance_expected = (Number( Number (balance_before)
                          +  Number(payment))).toFixed(2);
    var expected = balance_expected.replace(/\B(?=(\d{3})+\b)/g, ",");
    var value_after1 = value_after = "$" +  expected;
    console.log('Expected balance is:');
    console.log(value_after1);
});


casper.then(function(){
  this.click("#add-funds-button");
  console.log('Clicked on add funds button');
});


casper.wait(5000, function() {
    this.echo("I've waited for 5 seconds.", "INFO");
});

casper.then(function(){
  this.click("#add-funds-button");
  console.log('Clicked on add funds button');
  this.waitForSelector('#creditCard_1', 1000)
  this.click('#creditCard_1');
  this.waitForSelector('#add_new_card', 1000);
  this.click('#add_new_card')
  this.wait(1000);
  this.fillSelectors('form#login-form', {
      'input[id=use_new_card_1]': '4000100011112224',
      'input[id=use_new_card_2]': '09',
      'input[id=use_new_card_3]': '16',
      'input[id=use_new_card_4]': '123',
      'input[id=use_new_card_5]': '5.12'
      }, true);
  this.wait(3000);
  this.waitForSelector('button[id=btn_use_new_card]', 1000);
```

```
  this.click('button[id=btn_use_new_card]');
  this.wait(5000);
});

casper.then(function getText()
      {
       var new_balance1 = new_balance
                        = this.fetchText('#user-balance');
       console.log('New balance is:');
       console.log(new_balance1);
       test.assertEval(function() {
            return __utils__
            .findOne('#user-balance')
            .textContent === new_balance;
        });
      });



casper.then(function()
{
    this.click("#logout");
    console.log("Logged out the customer");
});

casper.wait(10000, function() {
    this.echo("I've waited for 10 seconds.");
});

casper.then(function(){
     this.test.
     assertTitle("Login - RM Pay", "RmPay title is the one expected");
});

 casper.then(function(){
    localStorage.clear();
    });

casper.run( function () {
    this.test.done();
    casper.exit();
});
```

3. PROTRACTOR

```
describe('Customer TopUp', function () {
  jasmine.DEFAULT_TIMEOUT_INTERVAL = 350000;

  describe("Access Login Page", function() {
    it("should display the correct title", function() {
      browser.get(browser.params.baseUrl);
      browser.sleep(5000);
      expect(browser.getTitle()).toBe('Login - RM Pay');
    });
  });
```

```
describe("Perform Login", function() {
  it ("should login David Brown", function() {
    var loginCustomer = require ('../index/pages/login_customer.js');
    loginCustomer.loginDavidBrown();
  });
});

describe ("decision", function() {
  it ("should decide which way to top up", function() {

    element(by.id('add-funds-button')).click();
    browser.sleep(1000);
    element(by.id('use_new_card_1')).isPresent()
    .then(function(result)
    {
    if (result)
    {
      browser.get('https://rmpayqa.routematch.com/');
      browser.sleep(5000)

      .then(function()
      {
            console.log('\n Displaying home page');
            browser.sleep(1)
            .then(function()
            {
                  console.log('Adding funds from new card');
                  element(by.id('user-balance')).getText()
                  .then(function(text)
                  {
                  console.log('Read users balance' );
                  var payment = parseFloat((5.12).toFixed(2));
                  var balance_before =
                  parseFloat(text.slice(1).replace(',',''));
                  var balance_expected =
                  (Number(Number(balance_before)
                  + Number(payment))).toFixed(2);
                  var expected =
                  balance_expected.replace(/\B(?=(\d{3})+\b)/g, ",");
                  var value_after = "$" +  expected;

                  console.log('\n Payment amount:');
                  console.log(payment);

                  //information for user what is expected:
                  console.log('The old balance was:');
                  console.log(text);
                  console.log("We're expecting balance to be:");
                  console.log(value_after);

                  element(by.id('user-balance')).getText();
                  element(by.id('add-funds-button')).click();
                  browser.sleep(2000);
                  element(by.id('use_new_card_1'))
                 .sendKeys('4000100011112224');
                  element(by.id('use_new_card_2')).sendKeys('09');
                  element(by.id('use_new_card_3')).sendKeys('16');
```

```
                element(by.id('use_new_card_4')).sendKeys('999');
                element(by.id('use_new_card_5'))
                .sendKeys(payment.toString());
                browser.waitForAngular();
                browser.sleep(2000)
                .then(function()
                {
                      element(by.id('btn_use_new_card')).click();
                      browser.waitForAngular();
                      browser.sleep(5000)
                      .then(function()
                      {
                          //to get information about current balance
                          console.log('new balance is:');
                          element(by.id('user-balance')).getText()
                          .then(function(text)
                          {
                            console.log(text);
                          });

                        expect(element(by.binding('balanceAmount'))
                        .getText()).toBe(value_after);
                        }, 20000);
              });
        });
       browser.element(by.id('logout')).click();
    });
  });
}

      else
      {
       console.log('\n Credit card already present');
       browser.get('https://rmpayqa.routematch.com/');
       browser.sleep(1500)
       .then(function()
       {
         browser.sleep(1)
         .then(function()
           {
             element(by.id('user-balance')).getText()
             .then(function(text)
             {

                      var payment = parseFloat((5.12).toFixed(2));
                      var balance_before =
                     parseFloat(text.slice(1).replace(',',''));
                      var balance_expected = (Number
                      ( Number (balance_before) +
                     Number(payment))).toFixed(2);
                      var expected =
                     balance_expected
                      .replace(/\B(?=(\d{3})+\b)/g, ",");
                      var value_after = "$" +  expected;
                      console.log('\n Payment amount:');
                      console.log(payment);

                      //information for user what is expected:
```

```
                                console.log('The old balance was:');
                                console.log(text);
                                console.log("We're expecting balance to be:");
                                console.log(value_after);

                                element(by.id('user-balance')).getText();
                                element(by.id('add-funds-button')).click();

                               browser.sleep(2000);
                                element(by.id('creditCard_1')).click();
                                browser.sleep(1000);
                                element(by.id('add_new_card')).click();
                                browser.waitForAngular();
                                browser.sleep(5000);
                                element(by.id('use_new_card_1'))
                                .sendKeys('4000100011112224');
                                element(by.id('use_new_card_2'))
                                .sendKeys('09');
                                element(by.id('use_new_card_3'))
                                .sendKeys('16');
                                element(by.id('use_new_card_4'))
                                .sendKeys('123');
                                element(by.id('use_new_card_5'))
                                .sendKeys(payment.toString());
                                browser.waitForAngular();
                                browser.sleep(2000)
                                .then(function()
                                {
                                     element(by.id('btn_use_new_card')).click()
                                     .then(function()
                                     {
                                          browser.waitForAngular();
                                          browser.sleep(5000);
                                          console.log('New balance is:');
                                          element(by.id('user-balance'))
                                          .getText()
                                          .then(function(text)
                                          {
                                             console.log(text);
                                          });

                                        expect(element(by.binding('balanceAmount'))
                                        .getText()).toBe(value_after);
                                         });
                                });
                           });
                      element(by.id('logout')).click();
                      });
                });
          }
        });
     });
  });
});
```

PROTRACTOR LOGIN_CUSTOMER.JS FILE:

```
var loginCustomer = function() {

  this.loginDavidBrown = function() {
    element(by.id('login-email')).sendKeys('anna@home.com');
    element(by.id('login-password')).sendKeys('XXXXXXXXXXX');
    browser.waitForAngular();
    element(by.id('login-btn')).click();
    browser.sleep(5000);
    browser.waitForAngular();
    expect(element(by.id('title-username'))
    .getText()).toBe('GLORIA JANDORF');
  }

}

module.exports = new loginCustomer();
```

4. NIGHTWATCHJS

```
module.exports = {
  'Log in test' : function (browser) {
    browser
      .url('https://rmpayqa.routematch.com/login?a=ky_wheels')
      .waitForElementVisible('body', 1000)
      .setValue('input[type=email]', 'anna@home.com')
      .setValue('input[type=password]', 'XXXXXXXXXX')
      .waitForElementVisible('button[id=login-btn]', 1000)
      .click('button[id=login-btn]')
      .pause(5000)
      .assert.containsText('#title-username', 'GLORIA JANDORF')
  },

  'Top up test' : function (browser) {
    browser
      .getText("#user-balance", function(result) {
       console.log('Current balance is:') ;
       console.log(result.value);
       var payment = parseFloat((5.12).toFixed(2));
       var balance1 = balance = result.value;
       var balance_before =
       parseFloat(balance1.slice(1).replace(',',''));
       var balance_expected = (Number( Number (balance_before)
                              + Number(payment))).toFixed(2);
       var expected = balance_expected
                      .replace(/\B(?=(\d{3})+\b)/g, ",");
       var value_after1 = value_after = "$" +  expected;
       console.log('Expected balance is:');
       console.log(value_after1);
      });
    browser
```

```
      .waitForElementVisible('button[id=add-funds-button]', 1000)
      .click('button[id=add-funds-button]')
      .pause(5000)
      .waitForElementPresent('#creditCard_1', 1000)
      .click('#creditCard_1')
      .waitForElementVisible('#add_new_card', 1000)
      .click('#add_new_card')
      .pause(1000)
      .setValue('input[id=use_new_card_1', '4000100011112224')
      .setValue('input[id=use_new_card_2]', '09')
      .setValue('input[id=use_new_card_3]', '16')
      .setValue('input[id=use_new_card_4]', '123')
      .setValue('input[id=use_new_card_5]', '5.12')
      .pause(3000)
      .waitForElementVisible('button[id=btn_use_new_card]', 1000)
      .click('button[id=btn_use_new_card]')
      .pause(5000)
      .getText("#user-balance", function(result)
      {
       var new_balance1 = new_balance = result.value;
       console.log('New balance is:');
       console.log(new_balance1);
       browser.assert.containsText('#user-balance', new_balance);
      });
    browser
      .end();
  }
};
```