

SET UP OF AUTOMATED USER INTERFACE TESTING SYSTEM



Bachelor's thesis

Electrical and Automation Engineering

Valkeakoski

Spring 2020

Oanh Do

Electrical and Automation Engineering
Valkeakoski

Author	Oanh Do	Year 2020
Subject	Set up of automated user interface testing system	
Supervisor(s)	Juha Sarkula	

ABSTRACT

Automation has evolved dramatically in the past decade, one aspect of this being software automation testing. The system used in this thesis is Nightwatch.js which is a Node.js-based framework solution for web applications and websites. With the demonstration conducted in this paper, the author aims to specify the vital role of automation testing framework in the software industry.

The commissioning party was Quux Oy, a software company located in Valkeakoski (Finland). The testing in the company are still done manually and there is an imperative need for the setup of an automated user interface testing system.

The thesis project included a theoretical review using online sources such as articles, forums, electronic sources, and the main website of the Nightwatch itself. The theory was focused on the definition of automation and on defining Nightwatch.js system along with all the required features to set up the system. The implementation part was where the whole set up process was being examined and recorded.

The outcome of the thesis project is a test suite where all the components required to be tested were covered. The targets of the thesis were achieved, with a justification for using this testing system in Quux Oy and its whole set up process.

Keywords automation testing, Nightwatch.js, Jenkins

Pages 34 pages with appendices 3 pages

CONTENTS

1	INTRODUCTION	1
1.1	Commissioner and the assignment.....	1
1.2	Outline and research objectives of thesis.....	2
1.2.1	Outline	2
1.2.2	Research questions and research objectives	2
2	THEORETICAL REVIEW	3
2.1	End-to-end automation testing.....	3
2.2	Nightwatch.js.....	4
2.2.1	API Reference	5
2.2.2	An example	6
2.3	Guideline in writing test scripts	7
2.4	About its dependencies.....	8
3	DESIGN AND IMPLEMENTATION	9
3.1	Implementation.....	9
3.1.1	Configuration	9
3.1.2	Writing tests	11
3.2	Further enhancement	16
3.2.1	Using command line	16
3.2.2	Defining constant	16
3.2.3	Before[Each] and after[Each] hooks.....	17
3.2.4	Custom command	17
3.2.5	Page object	21
3.2.6	HTML Reporter	23
3.2.7	Unit testing	24
3.2.8	Headless mode	25
3.2.9	Continuous integration and continuous delivery with Jenkins	26
4	LIMITATIONS.....	30
5	CONCLUSION	31
	REFERENCES.....	32

Appendices

Appendix 1 DP-1952.js IN EARLY STAGE

Appendix 2 testForRightAndWrongAccount.js PAGE OBJECT FILE

Appendix 3 DP-1952.js AFTER BEING CUSTOMISED

LIST OF FIGURES

Figure 1 Enter Google Gmail scenario	3
Figure 2 Enter Password scenario	3
Figure 3 Operation of Nightwatch (Theory of operation, 2019)	4
Figure 4 An example of a Nightwatch test script	6
Figure 5 Result of the example.....	7
Figure 6 Package.json file	9
Figure 7 Nightwatch.conf.js file	10
Figure 8 A description of a test story in JIRA (JIRA Software features, 2019).....	12
Figure 9 Test issue in JIRA (JIRA Software features, 2019).....	12
Figure 10 A testing folder's structure	13
Figure 11 Result in terminal window.....	15
Figure 12 Defining constant.....	17
Figure 13 loginAsAUser.js file	18
Figure 14 The test script when using custom command.....	19
Figure 15 The "custom_commands_path" property.....	19
Figure 16 An example of node-postgres connection (Bodnar, 2019)	21
Figure 17 Standard structure of a page object file	22
Figure 18 Commands property's structure	23
Figure 19 HTML Test Result	23
Figure 20 Testing pyramid (Jackson, 2017)	24
Figure 21 Headless mode setup	26
Figure 22 CI process (Pathania, 2017)	27
Figure 23 The Jenkins dashboard (McAllister, 2015)	28
Figure 24 Jenkinsfile (Declarative Pipeline) (Pipeline, 2020)	29

1 INTRODUCTION

In an era where software development is rapidly evolved, automation testing plays a crucial part when enhancing the testing quality and productivity not only by reducing a great amount of error but also by being a considerably great stand-in for manual testing with its versatility. Automation user interface testing is a process requiring an automated tool to implement test case suite for the web application, simulate user behavior and generate detail reports compared to expected results (Teixeira, 2013). The automation testing approach shorten development cycle with its reliability by being able to perform the same operation precisely every time – eliminate human error such as wrong data input and show how the application would react under repeated execution; its comprehensive by also being able to be reused on different versions of the web application to run more tests in less time. It is essential to understand that automation testing cannot completely replace manual testing, it is not a safe alternative since there are some scenarios that do not necessary to employ an automated test and manual works are preferable. Thereby, both manual and automated tests should be used coordinately - the testers/developers are advised to begin with manual testing and by using an automated tool for regression test purpose for better results (Screenster, 2018).

1.1 Commissioner and the assignment

Quux Oy, or Let's Do !T is a company located in the municipality of Valkeakoski and founded in 2014 with specialization in Information Technology & Services industry (Let's Do !T: About , 2020). In this field, Quux Oy focuses on the development of digital business and in the field of system development, software development and data center services related to the development and production of digital cloud services, integration, and marketing automation (Let's Do !T). In the dedicated software development team, there are backend and front-end developers; maintenance department has testers who run manual and unit tests. At this point, the strongly required automation testing system is End-to-end testing, also called as Automated User Interface testing. Compared to unit testing – a system testing a small unit of the code by breaking the code into functions then test each function to ensure that it behaves as expected (What is the Purpose of Unit Testing?, 2014), end-to-end testing helps ensuring the basic user interactions on the entire web application work well and does not contain ill effects on end users (Pittet, End-to-end tests, 2020). In short, although covering all the source code with unit testing system to verify whether written functions work correctly, it does not guarantee the functionalities of the final product – a combination of all the built functions - work together and create a

complete website. According to the introduction from the system's website, Nightwatch is an integrated End-to-end testing solution which examines real user scenarios of a web application from start to finish which can benefit the software development process in Quux Oy (Nightwatch.js, 2019). Thereby, this thesis is dedicated to examining the concept of a User Interface testing system – Nightwatch.js – by examining its benefits and drawbacks, along with strategies for the system setup.

1.2 Outline and research objectives of thesis

1.2.1 Outline

In order to clearly explain the concepts of this automation system, there are five chapters covering specific fields in this thesis, as follows:

Chapter one introduces the automation testing approach, the commissioning party and the assignment that needed to be carried out.

Chapter two takes a closer look to an automation testing system with Nightwatch.js system.

Chapter three is where the implementation takes place along with further enhancement for the overall system.

Chapter four discusses the drawbacks that the system has.

Chapter five is for the conclusion of the whole thesis.

1.2.2 Research questions and research objectives

The purpose of this thesis project was to give an inclusive answer through both theory research and design to the research questions: “How can End-to-end testing boost testing performance?” and “How can automation testing make the testing system of Quux Oy more flexible?”. The author of the thesis wished to follow these following objectives throughout the whole process:

- A thorough theory of an End-to-end testing system – Nightwatch.js.
- A justification of why to include the End-to-end testing framework to the testing system in Quux Oy.
- A complete set up process for the framework.

2 THEORETICAL REVIEW

2.1 End-to-end automation testing

End-to-end automation testing is a software quality assurance methodology ensuring applications have the correct functioning and performance as desired across the overall external interface. With end-to-end testing system, testers are able to test the complete functionality of the application in end user's point of view, because a complete end-to-end testing suite performs every behaviours of users on the application's interface. In which, one test script is written to verify functionality of a particular page from the system (Screenster, 2018). Figure 1 and Figure 2 demonstrate the action of signing in to a Google account, in which, the user will navigate through two pages and conduct these four interactions on the webpage's UI in order to be signed in the Gmail. A complete end-to-end test for signing in will contains test steps representing these four steps.

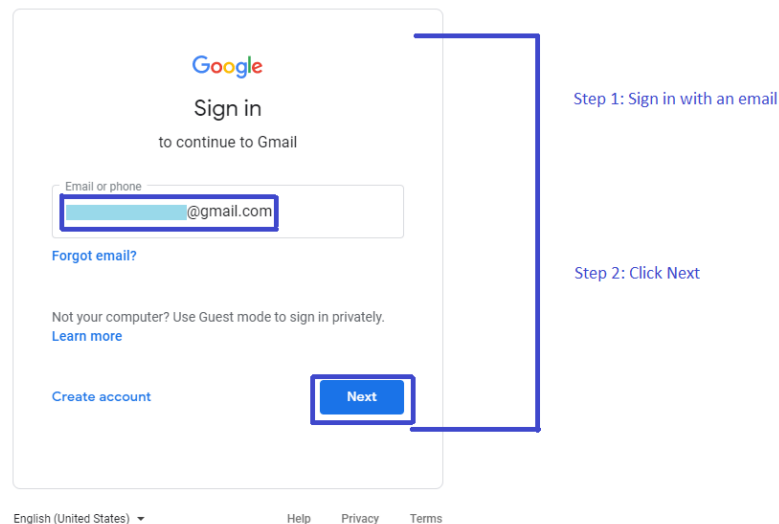


Figure 1 Enter Google Gmail scenario

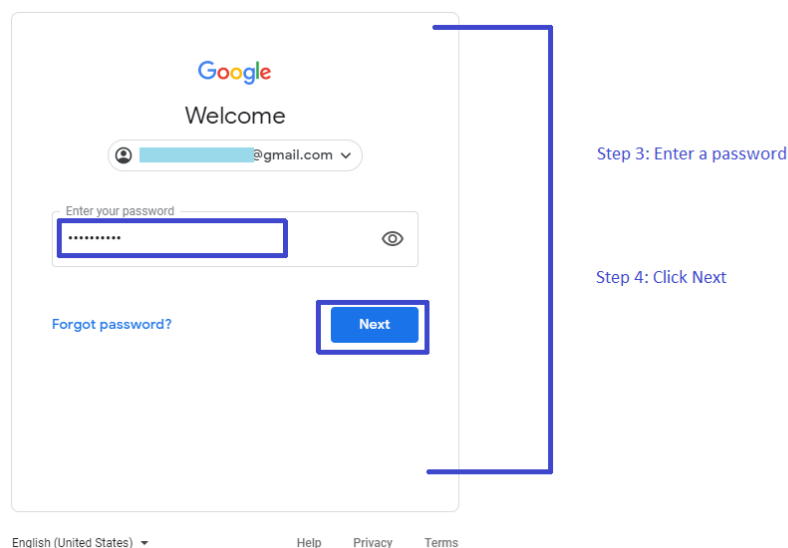


Figure 2 Enter Password scenario

2.2 Nightwatch.js

There are a vast number of end-to-end automation testing frameworks. Each of them proposes different process for generating tests along with different guidelines and installations. To choose which is the most suitable for the developer team, the testers should consider their needs and priority. Previously, at Quux Oy, testers used Robot framework for testing. This keyword-driven testing framework is implemented by using Python with PyCharm integrated development environment (IDE) to execute tests. However, for building a website, JavaScript is being used; in order to make the whole system more flexible and harmonize in a consistent way, a Node.js based framework as well as being written in Visual Studio Code IDE – Nightwatch.js is considered a better choice. With the benefits come with it, Nightwatch can help improve the development process of the team. As being stated in Collins English Dictionary, a tool is anything that can be used to perform an operation or to achieve an end (Collins Dictionary, 2020). In this thesis paper, the author will introduce the concepts of End-to-end automation testing, the tool – Nightwatch.js by describing to the reader how to use and boost the tool's functions. Thus, what is Nightwatch.js?

Nightwatch.js is a User Interface (UI) automated testing framework for web applications and websites running on top of Node.js and uses the Selenium WebDriver API to carry out commands and assertion, which means it allows testers to test an application through simple commands in a node.js environment efficiently by communicating through Hyper Text Transfer Protocol Application Program Interface – or HTTP API - with a Selenium/WebDriver server (Nightwatch.js, 2019). According to public information available on its official site, W3C WebDriver API drives browsers such as Chrome, Safari, and Firefox by controlling them remotely by standardizing browser automation (Webdriver, 2019).

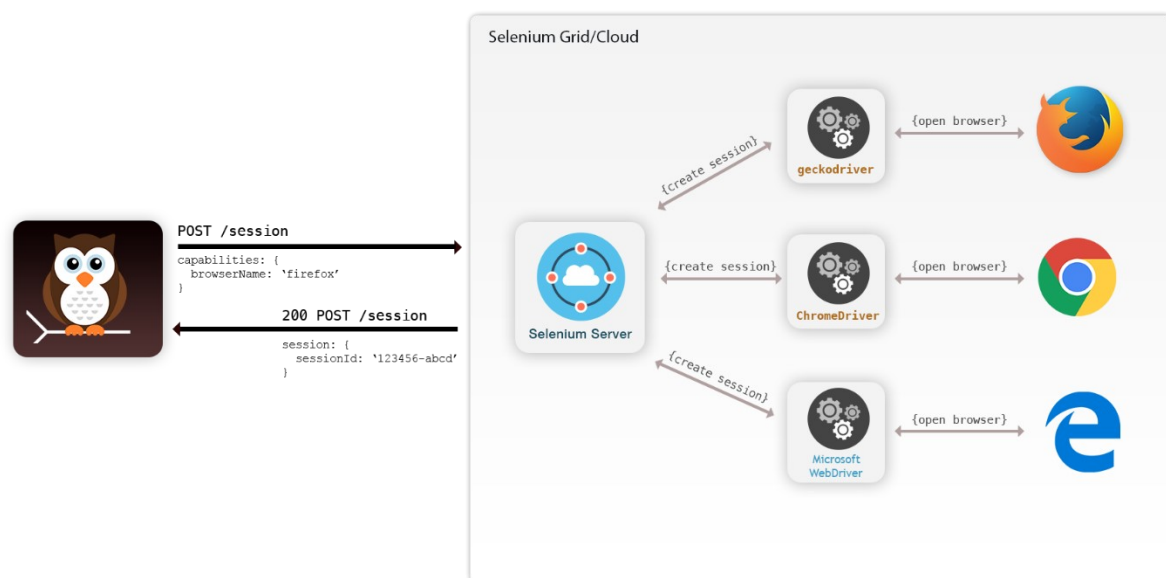


Figure 3 Operation of Nightwatch (*Theory of operation, 2019*)

Figure 3 shows how the Nightwatch.js system works. Usually Nightwatch sends two requests to the WebDriver server in which the first one is for locating an element with a given CSS selector/ XPath expression, and the second one is to perform the command on the element or assertion about it. The figure above show how Nightwatch communicates with Selenium Grid or Cloud system, however, it can still work with only W3C WebDriver API such as ChromeDriver.

Nightwatch.js was originally published in January 2014, the creator was Andrei Rusu, whose vision was to create a system which can write automated UI tests explicitly while the need of configuration and additional libraries is all but little (About Nightwatch, 2020). This testing framework uses page objects, commands, assertions with simple but powerful syntax for browser-based tests which make the test code clean and understandable. The useful build-in test runner function helps testing more flexible with the ability of running tests with groups of tags or single test module, and tests can be run successively or parallel. Nightwatch is also a Cascading Style Sheets (CSS) and Xpath support system. An automated UI test works by identifying and locating elements from the web page which mostly had been assigned attributes as Name, IDs or Class. However, there are circumstances where this practice is not achievable, using CSS selectors and Xpath for verifying the location of elements comes in handy. In short, by using Nightwatch, testers are not necessary to learn all the internal source code but can still test user interaction of the overall website and web applications with high performance of the automation execution. Furthermore, there are cloud server support, and continuous integration systems support (Nightwatch.js, 2019).

2.2.1 API Reference

As being introduced on the official website, Nightwatch's Application Programming Interface (API) reference has a remarkably long page full of commands that allow testers to use for writing test scripts in favor of manipulate the browsers. It is divided into four sections: Assert and Expect are used for validating the test, Page Object for advance tests and the most convenient syntax for manipulating users' behaviors are in the last section called API commands. The Assert library is used for implementing assertions on elements; there are also verify command for similar purpose; however, while the test ends without further performance when assertion of assert command fails, the verify command is used to log the failure and the tests are continued. For instance, `assert.title('expected value', 'message')` command are often used thanks for its ability to check whether the page title correct compared to given value then log the result. For Expect assertion, Nightwatch has the agility of Behavior-Driven Development (BDD) – a method to test the behavior of the system from the perspectives of users. The BDD interface of Nightwatch is based on Expect api from the Chai framework; an example for this assertion is

`browser.expect.element('body').to.be.present.before(1000)`. The purpose of develop Expect assertion is to have a more flexible and understandable language compared to Assert; however, even in the latest update, there is no support to chain assertions or optional messages. The third API used in Nightwatch which is an important and powerful functionality is Page Object. Its goal is to reduce the amount of duplication in the test code as much as possible by grouping pages or parts of page into object. Further information will be covered in this thesis while implementing the project due to its extensive assistance; for now, it is adequately to know API Page Object module consists of URL property, elements object and arrays of commands. The last introduced section is called API Commands, these WebDriver protocol mappings help improving the writing tests with a more readable and comprehensible syntax. There are composite commands for cases such as checking whether an element is displayed: `getValue()`, `isVisible()`; or some simple, basic commands such as `url()` or `execute()`. Additionally, Nightwatch also allows testers to manipulate the browser by providing commands functioning as `fullscreenWindow()`, `maximizeWindow()` or `openNewWindow`, etc (API Reference, 2020).

2.2.2 An example

In this chapter, a simple example of how Nightwatch.js works is given; the website being tested is `nightwatchjs.org` itself with a built-in assert library.

```

module.exports = {
  // First test case
  '@tags': ['example'],
  'Example': function (browser) {

    browser
      .url('http://nightwatchjs.org/')
      .waitForElementPresent('body', 1000)
      .assert.title('Nightwatch.js | Node.js powered End-to-End testing framework')
      .waitForElementPresent('body')
      .assert.visible('#navbar')
      .assert.containsText('#navbar', 'Getting Started')
      .assert.attributeContains('#navbar > ul > li:nth-child(2) > a', 'href', '/gettingstarted')
      .saveScreenshot('./screenshots/assert-home.png')
      .click('#navbar > ul > li:nth-child(2) > a')
      .assert.title('Getting Started | Nightwatch.js')
      .pause(2000)
      .end();
  }
};

```

Figure 4 An example of a Nightwatch test script

To begin with, a new Java Script (JS) file named “Assert.js” was made in a separate folder for the tests. In Figure 4, Assert.js starts with “module.exports”, same as every other JS files, to expose the objects as a module and a “@tags” property is added in order to make test more flexible with the ability to easily target tests based on their own one or multiple tags. Additionally, Nightwatch allows testers to test multiple test cases with different tags or to skip tags that does not need to be tested,

this will be discussed more deeply in following part of the paper. Because Nightwatch searches for keys on exported object, those keys will be test cases; and because the “browser” is a global object that being passed as an argument to the test, it is used for calling functions or accessing variables globally along with Nightwatch commands API. To navigate to a URL of the object’s web page, “.url()” is used. Then, the assert library is used to check whether the page title is equal to the given value or the “#navbar” – the navigation bar of the page – is available, etc. To capture the current state of the page, “.saveScreenShot” help taking the screenshot and save it to the given file. To simulate end user’s click action, the tester needs to locate the place with CSS selector/ Xpath o when using “.click”. Lastly, “.pause” is used for waiting and the browser session closes properly with “.end()” method. Further explanation on how to write an automated UI test script will be carried out in following chapters.

```
[Example] Test Suite
=====
| Connecting to localhost on port 9515...
DevTools listening on ws://127.0.0.1:64518/devtools/browser/a6c59037-3401-41f1-aa8a-3ee40983953a
i Connected to localhost on port 9515 (4880ms).
  Using: chrome (78.0.3904.108) on Windows NT platform.

Running: Example

✓ Element <body> was present after 26 milliseconds.
✓ Testing if the page title equals 'Nightwatch.js | Node.js powered End-to-End testing framework' (15ms)
✓ Element <body> was present after 18 milliseconds.
✓ Testing if element <#navbar> is visible (58ms)
✓ Testing if element <#navbar> contains text 'Getting Started' (63ms)

OK. 7 assertions passed. (7.253s)
```

Figure 5 Result of the example

It can easily be seen in Figure 4 and Figure 5 that by using commands from the API reference, testers are able to manipulate the web page and simulate user behavior; additionally, from what is shown in Figure 3, Nightwatch.js can generate the result of the process. The displayed result in Figure 3 shows the success of the assertion along with the amount of time it takes to perform. By using this end-to-end testing system, the tester can retest multiple times with less redundant manual steps and less be erroneous.

2.3 Guideline in writing test scripts

To ensure that there are no erroneous circumstances with the test, the following rules or guidelines below can be applied.

- A test should start a workflow with opening a browser, navigating to a certain website; this is a simulation of how a user use a web page.
- A test should be work independent without being linked to other test suites; this is a good way to avoid redundant failures.

- A test should only tests one feature or a test suite can have many test cases covering one feature individually – in other words, each test should have single purpose.
- Tests should end properly.

All the test suites, of any form of test automation, should follow these guidelines in order to prevent error-prone circumstances. From a small amount of test suites to a total of hundreds or thousands of tests, writing them with these rules, which were concluded by Raghavendra Prasad Mg in his book - Learning Selenium Testing Tools Third Edition, helps the structure clean and consistent, prevents having small issues that can ruin large parts of tests (MG, 2015).

Janet Gregory and Lisa Crispin also provided some more rules in More Agile Testing: Learning Journeys for the Whole Team, helping to create explicit test suites (Design Principles and Patterns, 2009).

- A test should be DRY – a short term for “Don’t Repeat Yourself” in automation testing wise – for easy modification.
- Use business readable names instead of plain elements copying for the web page.
- Avoid database access to prevent slowing down the process.
- The results from the test should always run green.

2.4 About its dependencies

To install Nightwatch.js on one’s machine, first, this automated testing framework works on Visual Studio Code IDE (VS Code); therefore, VS Code is required. Then, it is necessary to install Node.js – according to the introduction from the web page, Node.js is a platform which is built on Chrome’s JavaScript runtime, design for a fast and extensible network application – along with node package manager (NPM) (About Nodejs, 2020). The whole purposes of NPM is to make managing packages and dependencies easy; it assorts modules in order which helps the node find them more efficiently. Next, use NPM command line tool to install Nightwatch framework itself and Selenium WebDriver(s). Selenium WebDriver is used to accept commands, from the test suite, using Client API, and send them to a browser controlled and launched by a driver class. Nightwatch.js can be used for many browsers with specific WebDriver server listed on the website such as GeckoDriver, Microsoft WebDriver, SafariDriver and ChromeDriver. These have similar ways to install which is by downloading directly online or by using NPM. In this project, ChromeDriver will be used as based on the customer’s need. In these past few years, Nightwatch.js has many upgrades and the version will be used in the thesis is the latest one which is version 1.3.1. With these updates, it is no longer necessary to use Selenium Server to manage browser drivers including Chromedriver – the priority of this project; however, to those testers who use old browsers such as Internet Explorer, download the

latest Selenium Server Standalone package is a must and it should be placed in a file along with the desired browser driver.

3 DESIGN AND IMPLEMENTATION

3.1 Implementation

3.1.1 Configuration

All the programming test scripts of Nightwatch are written in Visual Studio Code software developed by Microsoft and the configuration is inspired by the work of Domenico Gemoli – a collaborator of Nightwatch (Gemoli, 2019). To begin with, creating a new Git repository for testing from an empty folder by typing “git init” is recommended; for this thesis, the new created folder is called ThesisDemo. Next, a key element file - package.json – which normally placed in the project root is also needed for Nightwatch.js system. JSON is the abbreviation of JavaScript Object Notation, it stores and transports data from server to web page. This package.json carries a great number of information related to the project which help NPM to analyse the project and manage its dependencies. By typing “npm init-y”, this default file is created.

```
{
  "name": "thesis-demo",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "nightwatch"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "chromedriver": "^78.0.1",
    "nightwatch": "^1.3.1"
  }
}
```

Figure 6 Package.json file

Figure 6 shows that inside the file, there are properties such as name – the project’s name or, in this case, the name of the folder – thesis-demo; version of the package; etc.

The first and the main dependency that need to be installed next is Nightwatch, which can be done by typing “npm install nightwatch –save-dev” into the terminal window. In this command, “–save-dev” helps

saving Nightwatch as a devDependency, a package that is used in the development process, in the “package.json” file created previously. From here, the “test” in “scripts” property in the package.json needs to be changed into “nightwatch”. Additionally, a file named “package-lock.json” along with the “node_module” file, where the downloaded libraries from NPM are placed, are also created. “Node_module” is where the chromedriver will be installed into – “npm install chromedriver --save-dev”, whereas “package-lock.json” is created automatically whenever npm is used. This package-lock JSON file is advantageous for its ability of stabilizing the process with consist install and in harmony dependencies.

According to Nightwatch’s website, to complete the configuration process, a file named “nightwatch.conf.json” is created manually by copying the config file shown on the website. This configuration file should look as Figure 7 illustrates:

```
module.exports = {
  "src_folders" : ["tests"],

  "webdriver" : {
    "start_process": true,
    "server_path": "node_modules/chromedriver/lib/chromedriver/chromedriver",
    "port": 9515
  },

  "test_settings" : {
    "default" : {
      "desiredCapabilities": {
        "browserName": "chrome"
      }
    }
  }
}
```

Figure 7 Nightwatch.conf.js file

The file presents src_folder, which is the folder all the nightwatch.js testing file will be placed in – in this case, the folder named “tests”; webdriver, an crucial protocol which powers tests and allows testers to manipulate end user’s behaviour on browsers by communicating with the browser drivers, contains Webdriver configuration options; test_settings, which are properties for test environments, containing a default environment node (such as chrome) or multiple different ones.

Finally, to install chromedriver, testers will need to type “npm install chromedriver --save-dev” in the terminal window. Some changes should be made in webdriver object settings depending on the testers’ computer. Server_path shows the location of the binary web driver file. In order to allow Nightwatch to manage the browsers, a path to web drivers should be stated. Therefore, if the testers are working with Windows computers, the path should be “node_modules/chromedriver/lib/chromedriver/chromedriver”

(chromedriver is used for this thesis). Otherwise, if the used local workstations are macOS, the copy from the website can be kept, which is "node_modules/.bin/chromedriver".

3.1.2 Writing tests

At Quux Oy, a tool developed by Atlassian - an Australian Company – called JIRA is being used. The purpose of using this tool is that it helps tracking and managing issues and bugs related to the web development process. Thus, this tool is highly beneficial for the workflow and team collaboration. To begin the testing process or even a web development process, testers or programmers should check the issues from JIRA beforehand. JIRA has a very powerful classification system for the workflow, these schemes are Issue having types as User Story, Epic, Sub-task, Bug, Test, etc (JIRA, 2019).

In testing process wise, testers focus mainly on Story, Bug and Test Story. In this project-friendly agile methodology, User Story is where placed the projects' requirements from end user's point of view; they are each made with a brief-but-comprehensive one-line summary describing what the story is about and a clear description of how the projects are expected to work. From what described in the Story, testers can presume how the final product should be and start creating Test-type issue; draft Nightwatch.js test scripts are also written in this stage. When the product is finished in its own branch in git, testers will check it by using "git pull" and "git fetch" to retrieve updates from the remote repository and test it in local repository. Git is an open source system and is the modern version control; it is used widely for tracking changes in source code to manage projects or set of files with their changes by allowing users to commit their work locally and merge the copy of the repository to server (What is git: become a pro at Git with this guide, 2019). At this stage, testers can rewrite the test script and run the automation test. If any bugs appear, inform the team by creating Defect under Sub-task option in JIRA; this will be covered in later section of this chapter. Automation test will be run again after bugs are fixed and will be merged from the local branch to remote repository with "git push". Two figures shown below – which are made by the maintenance manager of Quux Oy and named respectively "DP-1952: As a user, I want to login to system, so that I can use the service" and "DP-1953: Test As a user, I want to login to system, so that I can use the service" - are examples of what testers will be working with in JIRA, which contain necessary information for the testing process. Additionally, for example, "DP-1952" is the unique issue key of the task which its benefit is easy navigation, while "As a user, I want to login to system, so that I can use the service" is basically a brief summary of the task.

▼ Description

To be implemented:

Login page where are LOGIN button and fields for username and password.

Acceptance criteria:

- 1) User should be able to type any kind of characters to username field
- 2) User should be able to type any kind of characters to password field
- 3) User should see error message if username or password is wrong
- 4) User should see application selection page after successful login
- 5) Login should work with special characters in username or password field (i.e. !"#&/()=?)

Figure 8 A description of a test story in JIRA (JIRA Software features, 2019).

	Step	Data	Expected Result	Attachments
1	Go to login page		Login page opened	+ ...
2	Try to login with wrong password	Type correct username but wrong password	Login doesn't succeed. Error text is shown for user.	+ ...
3	Try to login with wrong username	Type wrong username but correct password	Login doesn't succeed. Error text is shown for user.	+ ...
4	Try to login with correct username and password	Type correct username and password	Login succeeds. Application selection page is loaded	+ ...

Figure 9 Test issue in JIRA (JIRA Software features, 2019).

Figure 8 shows that, in most cases, a description of a Story consists of two parts: to be implemented and the acceptance criteria. "To be implemented" shows what a tester should implement, how a test should be carried out; however, it is not necessary to follow every written step precisely, a test is qualified if its result meets the acceptance criteria below. Both a manual test and an automated UI test should be conducted to ensure that the product is able to perform all the tasks according to "acceptance criteria". After checking the issue with a success manual test, testers now create a test issue in JIRA which can be seen from an example shown in Figure 9. In this test details section, the steps are created based on "to be implemented" and "acceptance criteria" in Figure 8 where "steps" indicates action or behaviour of the end user's, "data" is the value

that need to be typed out and “expected result” is basically the desired result that the web page or web application should show. Therefore, the test scripts for automation testing are written accordingly. In addition, there are also issue operations section, under the issue key and summary, containing options for testers to edit, comment or assign the issue to JIRA users and sub-task option to create necessary type of sub-task including defect introduced above. There are also issue status where JIRA users can define the current status of the task to be from open, to do, coding, testing to ready, etc.

As mentioned before, a folder called “ThesisDemo” was used for this thesis project; inside it, there are subfolders such as node_modules, nightwatch.conf.js, package-lock.json, package.json and a directory called tests – which correspond with the property value in src_folders stated in nightwatch.json file – used for holding test modules. Figure 10 shows the structure of the testing folder.

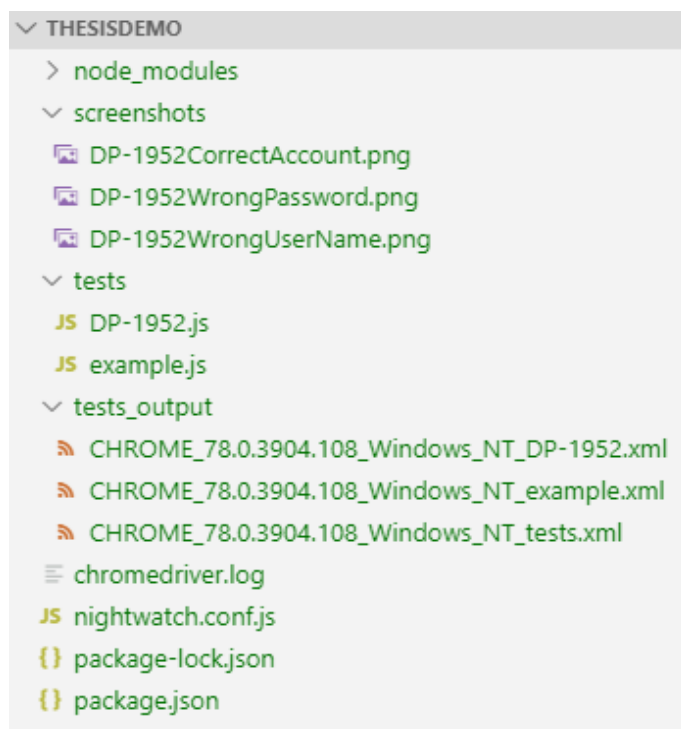


Figure 10 A testing folder's structure

To start writing any test script, a new Javascript file (.js) is created within “tests”. For better management, the name of this file should be the same as the testing task’s key – for example, “DP-1952”. Additionally, there will be a file called “tests_output”, which is where display all the .xml result logs file of Nightwatch, after running the tests. According to an article discusses about .xml file or Extensible Markup Language file, this is a markup language created to define syntax for encoding document by using tags in order to make the documents readable to both humans and machines (What is XML, 2018). There will be a file called “Screenshot” if

testers choose to take screenshot of steps in the tests – this works by using “.saveScreenshot” command.

The following steps in the writing test script process are similar with what was shown in the previous example in this thesis. Therefore, the tag of this test became “DP-1952” (the same as the task’s key and its automated test file’s name). According to the test issue of this login task in JIRA, testers should test whether the login system web page works well with the correct input of both the username and password or fails with the wrong input of either the username or the password by login many times with different inputs. Moreover, before executing any user’s actions, locating an element should be the first step in writing a test process. Nightwatch is a CSS/Xpath selector support system, which means that to locate an element on the page, testers can copy the CSS/Xpath from the page and paste it on the tests script’s command. In order to do so, testers should move the mouse over and right-click on the element needed to be spotted and choose “inspect” on the pop-up menu (or by press Ctrl+Shift+I on the clipboard). This action will display a Chrome DevTools window and the “Elements” tab will be opened with a tree-format HTML along with a highlighted element. Validating the XPath and CSS can be done by pressing Ctrl+F to use the search box for searching desired elements. To copy the CSS or XPath selectors, simply right-click on the element and choose “copy XPath” for XPath or “copy selector” for CSS option in the pop-up menu. The return result is pasted to the clipboard and it is an element instance representing the web page’s actual HTML element.

- “Step one: Login with wrong password” is the stage when by typing the wrong password, one cannot login to the website.

To begin with, the value of “step one” property is a callback function which receive browser as a property. To navigate the Uniform Resource Locator or URL which is the website’s address, use the command `.url()`. When the page is loading, the command `.waitForElementVisible()` is needed. Its purpose is to give an element a period of time(in milliseconds) to appear before any further performances. Next act is to set username and password with `.setValue()` command. This command usually requires a location – which can be get by using CSS or Xpath - to put the value into and the value itself. For achieving this step’s goal, the password input should be a wrong value. To simulate clicking the log in button, testers can use `.click()` command and locate the button. Usually whenever a click event is happened, the web should need a moment to load to another page; therefore, tests can use `.waitForElementVisible()` or use `.pause()` command and give the test some time to suspend. Usually when a wrong value is set, the page will not let user to log in and will generate a message saying about the reason. To verify whether this message appears on the web page, the command `verify.elementPresent()` from the build-in verify library is useful for this situation. With this command, testers can check whether an element exists by using CSS or Xpath to locate the element

and testers can also optionally give a log message to display in the output – in this circumstance, the message can be “Login failed”.

- “Step two: Login with wrong username” is the stage when by typing the wrong username, one cannot login into the website.

Using `.url()` from step two helps simulating the retrieving current page’s URL event. From here all the step is the same as step one; however, the username is now the one with wrong value. Testers still also need to use `.verify.elementPresent()` to check whether the page generate a message if the input is wrong.

- “Step three: Login with correct account” is the last step where using the real account to login to the page.

All the steps are the same from the steps above, from retrieving to the URL to login with the correct username and password to the click button event, all the commands can still be re-used. However, when the inputs are correct, the page will load to the main page which only the real user account can control, which will lead to some changes at the verify command line. The CSS or Xpath selector of the declared location at this step is copied from one of the elements show in the main page, and the message will be different with the change of “Login failed” to “Login successfully”.

To run the test, simply write “npm test” in the terminal window. “npm test” executes tests by knowing the stated value spotted in “tests” key in “scripts” property within the “package.json” file in the root of the project. Nightwatch, then, connects to local host on port 9515 which is where the chromedriver is in order to manipulate the browser. Figure 11 shows the result that Nightwatch generates after running the test script.

```
[DP 1952] Test Suite
=====
/ Connecting to localhost on port 9515...

i Connected to localhost on port 9515 (5128ms).
  Using: chrome (78.0.3904.108) on Windows NT platform.

Running:  step one: Login with wrong password

✓ The page is loaded
✓ Login failed (1079ms)

OK. 2 assertions passed. (11.231s)
Running:  step two: Login with wrong username

✓ Login failed (1625ms)

OK. 1 assertions passed. (4.956s)
Running:  step three: Login with correct account

✓ Login successfully (51ms)

OK. 1 assertions passed. (8.017s)

OK. 4 total assertions passed (30.043s)
```

Figure 11 Result in terminal window

3.2 Further enhancement

The automation test runs well with the structure above. However, there are ways to make the test more advance but readable and easier to manage. The following section of the thesis is about how to improve test scripts with more depth.

3.2.1 Using command line

The build-in test runner of Nightwatch is a very useful function which helps the testing process runs smoothly to generate useful results. This support comes with several advantageous run-time options; to view all, simply run `"npm test --help"`. There are lots of great and useful options such as running tests in groups, in tags or singularly, etc. As being discussed earlier, `"--tag"` or `"--skiptags"` are convenient with its advantage of filtering tests by tags to run or skip tests that have specified tags. Moreover, Nightwatch's `"--group"` support has similar influence where it allows tests to be organised into groups and run them as a single unit; by doing so, directly place necessary tests into a sub-folder while the folder's name is the group's name. The same as `"--skiptags"`, there is also `"--skipgroup"`. In order to block a test module from being executed, inside the module, set `"@disabled: true"`. Unfortunately, there is no official support to disable a test cases, however, modifying the test method into a string is a great method for overcoming this issue. Additionally, Nightwatch is still updatable, to check which version the testers have installed, simply use `"npm test --version"`. The `"--"` is necessary before adding extra Nightwatch switches because it helps separating the params passed to npm command and params that passed to the script.

3.2.2 Defining constant

In JavaScript, `"const"` is a way to define a constant in which the constant is a value that cannot be changeable (Const, 2019). When using `"const"` in Nightwatch, a CSS/Xpath selector of an element or an element itself can be named, which means the constants can be used multiple times without writing the values of the constants. As being mentioned above, to simulate the act of typing username and password, two `.setValue()` commands are used successively along with long lines of CSS/Xpath selectors. It can easily be seen that with this structure, the script will look chaotic; and when using this technique time and again, it will lead to unreadable situation if lots of identical commands are used repeatedly. Therefore, `"const"` is an excellent choice for a better script when elements which will be interacted with through commands and assertions are defined. As shown in the following snippet of code – Figure 12, the test will be more explicit if testers declare the location of the username and password field and theirs value with a name, or even with the location of the login button: the Xpath selector of where to input the username and password is now

“userAccountInput” and “userPasswordInput”, while the username and the wrong password value are simple “userAccount” and “wrongUserPassword”, the .click() command simulating click event now appears as “.click(submitButtonSelector)”.

```
.setValue(userAccountInput, userAccount)
.setValue(userPasswordInput, wrongUserPassword)
.click(submitButtonSelector)
```

Figure 12 Defining constant

As being shown above, a good programming convention is expected with meaningful and consistent constant names. With proper meaningful constant names, the scripts are readable and uncomplicated to modify later on.

3.2.3 Before[Each] and after[Each] hooks

This test hooks technique - before[Each] and after[Each] hooks – is inspired by Mocha test framework, which is used for setting up preconditions and closures to the tests (Using before[Each] and after[Each] hooks, 2019). There are two types of hooks: before/after and beforeEach/afterEach. The former run once before and after the execution of the test suite, while the latter run before and after each test case. Hooks can be appealed to with optional descriptions or named functions which helps locating errors in the tests easier. For example, Before hook is used for navigating the URL and the After hook is for ending the test execution.

3.2.4 Custom command

One of the motivations for this paper is how to execute this automation UI test system to its fullest. With its vast number of functions, Nightwatch allows testers to simplify the test script in advance ways; in which, one is so called Custom command function. When testing the web application of Quux Oy’s customer, which is built by Quux Oy developer team, almost most of the product require authentication - member’s account login - steps. Having a mindset that there will be scenarios where identical commands keep being repeated, Nightwatch simplifies the testing process by allowing testers to reduce these disadvantages with the ability to abstract away these easy-to-be-repeated commands and reuse them anywhere across the test suite (Writing Custom Commands, 2019).

When the testers decide to use Custom Command (CC), a “custom_command_path” property should be created in the “nightwatch.conf.js” file, next to “scr_folders” property, then specify the path indicating the location of the commands. Therefore, having a

separate file dedicated to them created within the root folder of the project is needed; because there will be sub-folders that hold different defined custom commands individually, the file can be named as “custom_commands”. As being introduced in the document, there are two types of custom command which are Function-style Commands and Class-style Commands (Nightwatch.js, 2019). The former is the simplest way to form a command where the command module exports a command function and call at least one Nightwatch’s own API command. The latter is how Nightwatch’ commands are written where command module exports a class constructor.

For upgrading the test script’ structure, the first type of this function is used. In the above demonstration wise, the custom command is about login event; therefore, the file is named “loginAsAUser.js”. The scenarios are login with three different input: the first one is wrong password, the second one is wrong username and the last is a correct account input. Consequently, the command module of “loginAsAUser.js” exports a function used as a class of “userAccount” and “userPassword” which means this command will take different constants of each steps into used. The following snippets of code will show what inside the command file and how the test will be when using this technique.

```
exports.command = function(userAccount,userPassword) {
  this
    .useXpath()
    .url(url)
    .setValue(userAccountInput, userAccount)
    .setValue(userPasswordInput, userPassword)
    .click(submitButtonSelector)
    .pause(load_speed)
  return this;
};
```

Figure 13 loginAsAUser.js file

```

module.exports = {
  '@tags': ['Dp-1952'],
  'step one: Login with wrong password': function (browser) {
    const userAccount = 'aWrongAccoutnInput';
    const userPassword = 'correctInput';

    browser
      .windowMaximize()
      .loginAsAUser(userAccount, userPassword)
      .verify.elementPresent('//*[id="root"]/div/div/div[1]/div/h6[2]', 'Login failed')
      .saveScreenshot("screenshots/DP-1952WrongPassword.png")
  },

  'step two: Login with wrong username': function (browser) {
    const userAccount = 'correctInput';
    const userPassword = 'aWrongPasswordInput';

    browser
      .loginAsAUser(userAccount, userPassword)
      .verify.elementPresent('//*[id="root"]/div/div/div[1]/div/h6[2]', 'Login failed')
      .saveScreenshot("screenshots/DP-1952WrongUserName.png")
  },

  'step three: Login with correct account': function (browser) {
    const userAccount = 'correctInput';
    const userPassword = 'correctInput';

    browser
      .loginAsAUser(userAccount, userPassword)
      .verify.elementPresent('//*[id="fuse-layout"]', 'Login successfully')
      .saveScreenshot("screenshots/DP-1952CorrectAccount.png")
      .end();
  }
}

```

Figure 14 The test script when using custom command

As mentioned earlier, the scenario above is when the user login with different inputs; therefore, the custom command considers the constant input of each step as a variable to modify it accordingly. With this custom command, tests for multiple user accounts process can be boosted. This command can still be reused in tests that require login event once per test suite; in order to achieve it, the correct account username and password are set inside the custom command file. Consequently, the main test script requires only to call that custom command. When the scale of the test gets bigger, the need of CC is more demanding. Therefore, many custom commands will be made and placed in their individual file. Inside the "nightwatch.conf.js" file, the "custom_commands_path" is now an array of the paths direct to those made custom command files. The Figure 15 below is how the property for the thesis demonstration looks like.

```

"custom_commands_path": [
  "./custom_commands/customer",
  "./custom_commands/user"
],

```

Figure 15 The "custom_commands_path" property

Additionally, although one of Janet Gregory and Lisa Crispin's rule mentioned above is about avoiding database access, there are cases where testing database value is necessary, which is when custom command comes in handy. Nightwatch is not specifically designed for related database testing, thus using it with this purpose results in unstable process. At Quux Oy, pgAdmin4 is a tool that used to manage the company database - PostgreSQL. pgAdmin4 is a graphical user interface, or simply considered as a web-based user interface, used for PostgreSQL administration. pgAdmin4 is an extension that helps managing, monitoring the schema, and executing SQL queries for PostgreSQL. PostgreSQL is a sophisticated top-picked open source relational database management system (RDMS) for its simplicity while installation and configuration, its rich extensions (pgAdmin, 2020). The relational model of database management was first introduced in the book *A Relational Model of Data for Large Shared Data Banks* by Edgar Frank Codd; which is a collection of tables, or relations in RDMS wise, represents a database where store organized data for the purpose of being retrieved by users (Relational model concepts, 2019). Structured Query Language (SQL) is a standardized declarative language used for managing and querying in RDMS (UAS, 2015), which is also used in Nightwatch whenever database access is required.

Because the demonstration of this paper is about testing whether the user identification function of the web page works well or not, there is no need of a database check-up. Therefore, the following figure is an example found during the research. Nightwatch is a Node.js based framework meaning, to access to database, Nightwatch works similar as how Node.js would work; Figure 16 shows how to make a connection to database from a normal node.js to get exist columns' names, therefore, to use it in Nightwatch, simply modify it to suit the Custom Command feature's structure. First, type "npm install pg" to install a non-blocking PostgreSQL client that used for Node.js, node-postgres, to connect the database (Carlson, 2019); and, optionally, install Ramda library for a more functional programming style to work with the data (aromano, 2019) - "npm install ramda".


```

column_names.js

const pg = require('pg');

const cs = 'postgres://postgres:s$cret@localhost:5432/ydb';

const client = new pg.Client(cs);

client.connect();

client.query('SELECT * FROM cars').then(res => {

    const fields = res.fields.map(field => field.name);

    console.log(fields);

}).catch(err => {
    console.log(err.stack);
}).finally(() => {
    client.end()
});

```

Figure 16 An example of node-postgres connection (*Bodnar, 2019*)

To include the modules, define constant, for example, “pg” and “R” as `require('pg')` and `require('ramda')`, or with only “`const pg = require('pg')`” is enough for the process. Then, define “cs” as a PostgreSQL connection string, the string in this case is “`postgres://postgres:s$cret@localhost:5432/ydb`”, to build the connection between the system and the database. Next, create a new object “client” and connect it with database through “`connect()`”. The “SELECT” query and the asterisk “*” are used to include all the attributes, or columns, of the table. “`res.fields`” attribute is used to retrieve the columns’ names along with the “`map`” method that used for creating a new array. “`console.log`” is used to output the results and “`catch`” clause is for output errors, if there is any and, finally, the process ends with “`end()`”. The result is all the columns’ names of the table.

3.2.5 Page object

Similar patterns will emerge when the automated test source script gets bigger; beside Custom Command technique, a very well-known pattern for reducing the duplication of test commands in automation UI testing is Page Object. Nowadays, Page Object model (POM) becomes more popular in automation framework and is being used in many projects, because its function of basically bundling pages or page parts into objects make it enhance the testing system’s maintenance with clearer structures. POM is similar to the technique introduced above; except they are bundles of custom commands that used for specific UI component (*Working with Page Objects, 2019*).

Configuring POM is similar to CC, Nightwatch reads the page objects from the folder (or folders) specified in the `page_objects_path` configuration property inside “nightwatch.conf.js”. The property can also be an array of folders, if the page objects are split into smaller groups.

```
module.exports = {
  url: '',
  elements: {},
  commands: [{}]}
}
```

Figure 17 Standard structure of a page object file

Figure 17 shows what a defined page object in its own module – the demonstration file is named “testForRightAndWrongAccounts.js” - contains: a string for URL property, an object, or an array of objects, for elements and an array of objects for commands. The URL property is basically the URL of the page that needed to be tested; it can be a string or also a function if the URL is a dynamic URL. Elements property are, at its simplest form, objects of UI elements’ location, which can be interacted from the page, with their identified names; these are used within commands called from the page object and their values are either CSS selector or Xpath selector. One advantage of POM is that switching CSS to Xpath selector is now handled internally by simply specifying “locateStrategy: 'xpath” – CSS selector is set at default - and the API “.useXpath” or “.useCss” is unnecessary to be written in the main test scripts. Having the same functionality as CC, POM allows testers to define own commands, and it is also by where Nightwatch commands and assertions API is inherited; therefore, the commands property is a list of objects containing functions, that are encapsulated, used for simulating users’ behavior on the web page.

To reference page object in the main test, define an instance for this JS object such as, for example, “const page = browser.page.testForRightAndWrongAccounts ();”. Whenever the “testForRightAndWrongAccounts” factory function is called, a new instance is created. Thus, to navigate the URL defined in URL property, use this object to navigate: “page.navigate()”; this function is what browser object does not have and testers do not need to rewrite the URL stated in the page object module when using it. This means that whenever “browser” object is called, global commands and assertions are used, and “page” object is called for page object’s custom commands and assertions only which also is passed as an argument. Another useful property but not as common as others is “sections” property; sections are used to help organizing the test scripts into logical groups and performing element-level nesting.

```

commands: [{
  loginWrongUserName(userAccount,userPassword) {
    return this
    .setValue('@userAccountInput', userAccount)
    .setValue('@userPasswordInput', userPassword)
    .click('@submitButton')
  },
  validate() {
    return this
    .verify.elementPresent('@error', 'Login failed');
  }
}]

```

Figure 18 Commands property's structure

As simple as CC, the Figure 18 shows that commands in POM are made by encapsulating commands and assertions of Nightwatch; this technique thus also helps the main test scripts gain more simplicity and flexibility along with prevents them from being flakey. One difference between these two models is that, in POM, to refer to an element in the commands, call its name with the “@” prefix rather than selector, this is shown clearly in the snippet of code above. Additionally, a privilege of using POM is that a custom command can also be called inside the page object’s commands property.

3.2.6 HTML Reporter

For a more professional performance, Nightwatch also supports HTML Reporter for result reports. Theses HTML reports will be generated under “tests_output” folder with “.html” type. Thanks to the work of Denis Denisov, a complete setup for this system was established (Denisov, 2020). Firstly, run “npm install handlebars” for semantic templates; “npm install fs” and “npm install path”. Then, copy “html-reporter.js” and “html-reporter.hbs” to project-based directory – which can be found on Denisov’s post. Lastly, to run the result in HTML, use “npm test -- --tag Dp-1952 --reporter html-reporter.js”.

Test Results

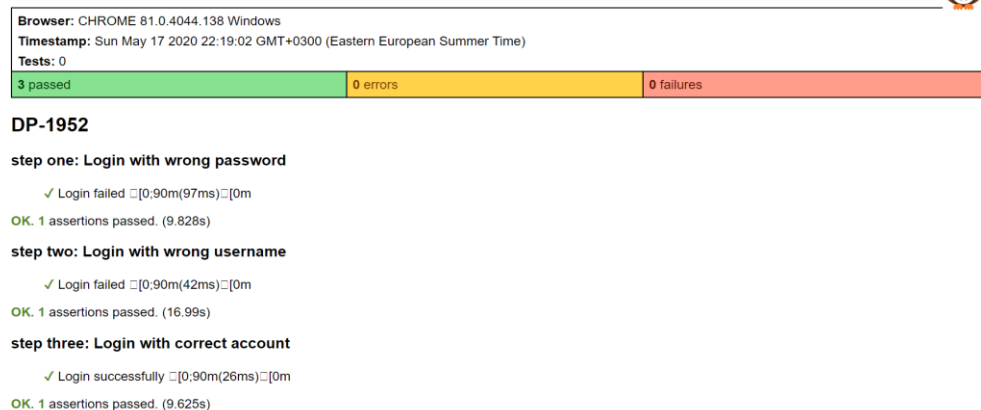


Figure 19 HTML Test Result

Figure 19 above is a HTML Test Result for this thesis demonstration, which contains all necessary information of the test including steps, number of assertion and how long the assertions take. It also supports user to customize the report's theme and structure via "html-reporter.hbs" directory.

3.2.7 Unit testing

Unit testing is a different type of testing compared to End-to-End framework which will not be covered deeply in this paper; however, a comprehensive idea about this additional function of Nightwatch is introduced. According to what Erik Dietrich has stated in his book - Starting to Unit Test: Not as Hard as You Think, unit testing is a software testing system that cover smallest individual components or modules of the source code, typically a method or a function (What is the Purpose of Unit Testing?, 2014). Its purpose is to validate whether each testable unit of the source code performs as desired with generated results of either pass or fail; additionally, the results can be time out or inconclusive depending on the setup of the used testing tools. Unit test is considered beneficial for maintaining or modifying codes with its simple and localized structure along with its advantage of allowing developers to fix bugs in early stages of development; it is commonly written by developer team and not testers, thus the knowledge of the internal source code is required.

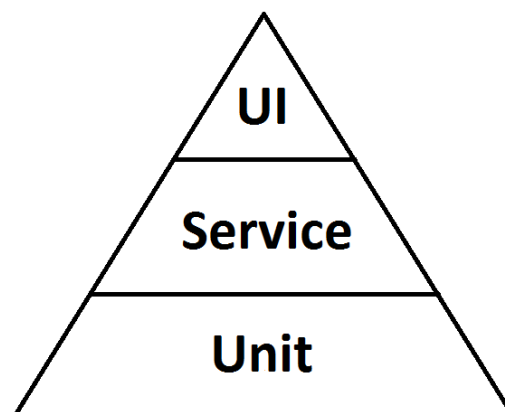


Figure 20 Testing pyramid (*Jackson, 2017*)

Figure 20, courtesy of Nic Jackson (Building Microservices with Go, 2017), is a figure created by Nic to demonstrate the original testing concept of Mike Cohn which Cohn explained in his book - Succeeding with Agile. The concept was that the bottom of the pyramid, which is Unit testing, is the testing system having a vast number of code and is the least time consuming; while the UI testing system, which is at the top of the pyramid, has least test scripts but consumes lots of time to create them, and the Service system is not as crucial (Cohn, 2009).

This leads to a result of many projects done without UI testing. However, through many years, this approach is proven to not flexible and not suitable for all situations. Technology has grown dramatically this decade and changed this assumption; the original pyramid is modified into different flexible alternative models for all situations but still being kept the original idea for guiding teams in getting the most value from their testing system. Gradually, with the enhancements of technology, it is acknowledged that one single testing tool cannot assure the quality of the source code (Lisa Crispin, The Dangers of Putting Off Test Automation, 2009). According to Gemoli, an experienced developer who has worked with end-to-end testing and an official collaborator of Nightwatch, even if the source code is all covered with unit test, it is not guarantee that the produced web page has required interactable features or the logic of functions behind those features do not correctly work; therefor, a little amount but powerful UI test can help solve this problem (Gemoli, Why is (end-to-end) testing important (to me)? And what does the testing pyramid actually mean?, 2019).

In order to enable the Unit test mode in Nightwatch, there are two ways. The first one is similar to Custom command and Page Object introduced above, inside “nightwatch.conf.js” file, a property “unit_test_mode” need to be created and set to “true”; this way of setting is for global. The second way is for individual test suites; set the “@unitTest” property in “module.exports” to true. For unit test, the object passed as an argument is not “browser” or “page” but the “done” callback (Unit Testing with Nightwatch, 2019).

3.2.8 Headless mode

Whenever being used in a Continuous Integration environment, Nightwatch.js should be set up with headless mode. Browser headless mode, or headless mode, is a condition of a web browser without its graphical user interface. Headless mode allows tester to be in control of the testing normally, execute the tests programmatically and without rendering any visible UI shell . Currently, there are many headless browser types, such as Chrome, Firefox and so on (To, 2018). The Figure 21 below is the additional part of “nightwatch.conf.js” directory, which can be used for setting up headless mode. In which, the configuration implies that Chrome is run in headless mode with the resolution of 1920 x 1080; disable gpu is needed when running in Windows; and the binary path of Chrome.

```

"test_settings" : {
  "default" : {
    "desiredCapabilities": {
      "browserName": "chrome",
      "chromeOptions" : {
        "args": [
          "window-size=1920,1080",
          "headless",
          "disable-gpu",
        ],
        "binary" : "C:/Program Files (x86)/Google/Chrome/Application/chrome.exe"
      }
    }
  },
},

```

Figure 21 Headless mode setup

Headless browser is quite difficult for debugging; therefore, this is where “.saveScreenShot” command comes in handy. Without the visible UI, tester cannot spot the error when the test fails instantly. However, the result in the console shows where the error occurs - for example, a “.click” command cannot interact with an element. Then, tester can re-write the test script by adding “.saveScreenShot” before the “.click” command to take the screen shot before the test fails.

3.2.9 Continuous integration and continuous delivery with Jenkins

Continuous Integration, also famous as CI for short, is a software development practice monitoring the automation and continuous process of from integration and testing to delivery and deployment of a product (What is CI/CD?, 2020). This development practice requires integrated commits from developers’ personal branch to common work branch (or remote branch) regularly. After every commit, the developer team can detect errors immediately and as early as possible with the verifying help of building and testing the committed code from an automated build of CI (Pathania, 2017). In other words, CI, in its simplest form, detects a change in the source code as the soonest, then compiles and tests the application; the tool will notify developers whenever there is an issue that needed to be fixed. With a good CI infrastructure, the development environment has less erroneous situation, less redundant repeated actions; are able to deliver more real value product as soon as possible while improving the working quality of the development team; and the health of the system is now measurable (Nguyen, 2017). Figure 20 depicts steps that are conducted in a CI environment.

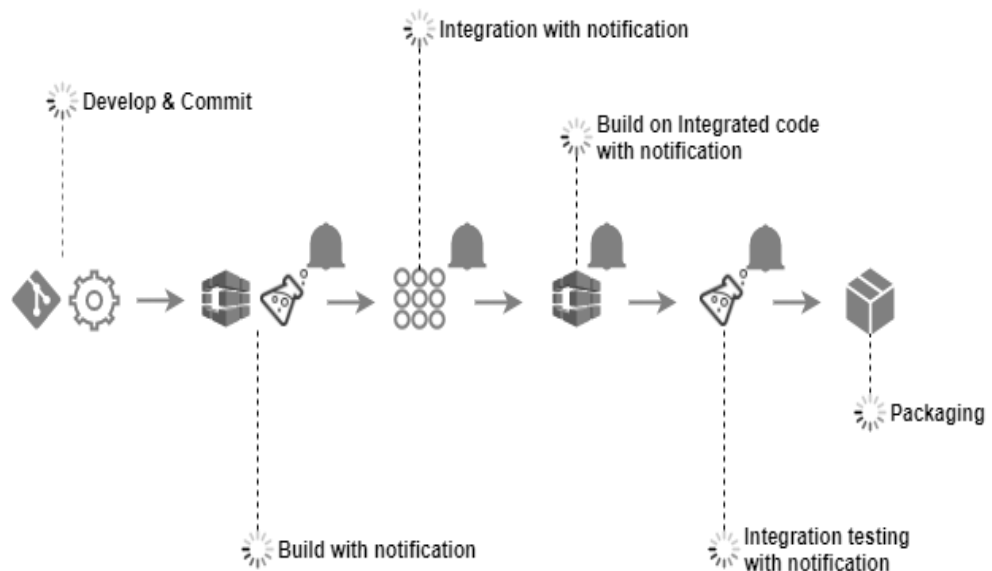


Figure 22 CI process (Pathania, 2017)

One of the most well-known CI tools is Jenkins. Jenkins, historically called Hudson, is an open source and Java- written automation server. Jenkins is made available as a tool for automating tasks such as building, deploying, compiling source code, generating test executions and scheduling builds; also is available to be used on different platforms including Windows, Mac OS, Linux etc. (Rajkumar, 2019). John Ferguson Smart, creator of Jenkins, also stated in his book that Jenkins is simple, well documented and has visually appealing and friendly user interface (Smart, 2011). This tool is widely used with its continuous integration expertise; ability in fast feedback, periodically and automatically in build scheduling; and its intelligence of identifying issues from the beginning (Angler, 2019). Therefore, Jenkins is useful for both developers and testers-wise.

For automated testing wise, Jenkins is useful with many great features. To begin with, there is the ability to schedule tests and allow testers to run them at a specific time. Jenkins is well-known with its test result trends displaying on the home page of each project, which let the users to see the overview of the tests' current state. Additionally, Jenkins also provides summary for tests' results including the number of executed tests along with their executing time, etc. The build time trend feature of Jenkins also shows the amount of time needed for tests to run using graph. Furthermore, Jenkins is also able to show the details from a test failure such as error message and stack trace. There is also email notification support, which means after every execution of the tests, Jenkins will send emails automatically to whom needed to be announced when the tests is completed (Saxena, 2016).

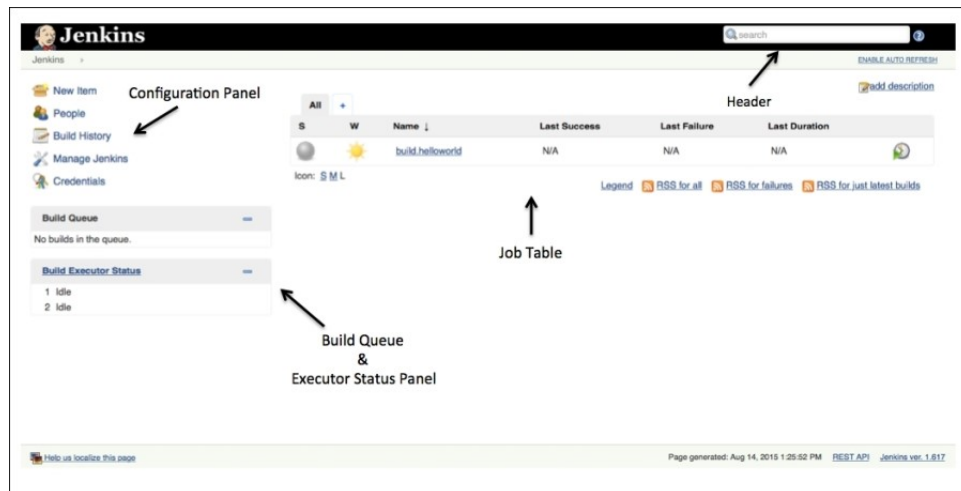


Figure 23 The Jenkins dashboard (McAllister, 2015)

Figure 21 shows the friendly and visually appealing user interface of Jenkins. In the *Mastering Jenkins* book, McAllister divided Jenkins main dashboard into four sections, which is Header, Job Table, Configuration Panel and Build Queue & Executor Status Panel. The header section includes a breadcrumbs system, which shows user's current page location; an add description link – an option with the ability to add description to the dashboard or build; an Enable Auto Refresh switch for enable or disable the auto refreshing page feature; and, lastly, a search bar, which displayed on the top-right corner. Next section is the configuration panel on the top-left corner and below the header of the dashboard. Additionally, the configuration panel will change in the subpages with different configuration options. On the main dashboard, the default options on the configuration panel are New item – used for creating new Jenkins jobs, People - including all known “users”, login identities and people mentioned in commit messages, Build history – a list of already executed builds along with their statuses, Manage Jenkins – most used by administrators to manage and configure Jenkins to satisfy individuals requirements, and Credentials – used for managing user account credentials. The main and most important section of Jenkins user interface is the job table. This table is a list of jobs such as build job, deployment job or smoke test job and so on. There is status of the most recent build, where red is for failure, blue is for success and unstable status is yellow; weather report – a feature for a combined report of recent builds; name - name of the job; last success and last failure columns are for how long ago the last successful and failed execution of the build was; last duration column is for run time length of the latest build; table footer that contains RSS feeds providing job's status; legend – a link of a graphical legend containing all dashboard's icon and their definitions. Last but not least, Jenkins main dashboard also has the build queue and executor status panel. In which, Build Queue displays waiting triggered job and Build Executor Status lists master executor and slave nodes (Belmont, 2018).

To create a build job, choose New item menu link on the Configuration panel which is the entry point into job creation. The user will then be navigated to the item's configuration page containing Item name bar and project type options. There are two most used project type: Freestyle project and Pipeline project. In which, freestyle project allows user to custom the build job freely and mostly is used for running simple job, while Pipeline project is for large project with the need of continuous delivery pipeline. Pipeline can break the jobs into stages which each stage is a job executing the commands as the user desire. Therefore, this ability helps user to see the problem clearer in each stage. An example for this type of project is build -> unit test -> delivery -> testing -> deploy. Pipeline can be built through a scripted file – Jenkinsfile or a web interface – Blue Ocean (ramz, 2019).

Continuous delivery (CD) is an extension of continuous integration for getting software from version control – local machine – to team members or customers with speed in a sustainable way. As being mentioned earlier, this process involves building, testing, and deploying the software through stages. This automated expression is usually distinguished with the ability of allowing user to release the software daily, weekly or any time that suit business requirements (Pittet, 2020).

As being claimed in the Jenkins official website, Blue Ocean was a project that was being developed to evolve the user experience with software continuous delivery through a new improve clarity and less clutter user interface. It is designed to have a sophisticated visualization for CD by being visualized on screen with steps and logs to create a CD pipeline from start to finish. Blue Ocean is comparable with both Freestyle and Pipeline projects; however, it is recommended to choose Pipelines when using Ocean Blue because of its ability to let users easily observe the execution and spot the problems with the Pinpoint Precision feature with ease and speed (Dumay, 2016). Created by Blue Ocean or by Jenkins New Item method, Jenkins pipeline is a great practice for continuous delivery, which is defining the entire build process, including stages for building an application, testing it, and then delivering it.

```

pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                //
            }
        }
        stage('Test') {
            steps {
                //
            }
        }
        stage('Deploy') {
            steps {
                //
            }
        }
    }
}

```

Figure 24 Jenkinsfile (Declarative Pipeline) (Pipeline, 2020)

Figure 24 shows how Declarative Pipeline syntax of Jenkinsfile defines stages and steps of the work. “agent any” is how Pipeline execute its stages. “Build” is a stage where its work is performed under “steps{...}”, which is the same to “Test” and “Deploy”. Thanks to the syntax block “stage”, installation and setup stages are separated from the actual test execution. Therefore, user can quickly spot which part failed; then, if the “e2e tests” stage fails, user directly know the error is in the test stages and not installation issue. By using Jenkins Pipeline, nightwatch.js end-to-end test is easily tracked and executed in a continuous delivery environment (Pipeline Syntax , 2020).

Due to the fact that the author has reached the commissioner’s requirement for this testing system, no further implementation for CD with Jenkins will be conducted in this paper. However, this work discussed profoundly about automated testing system, the author believes that this paper will provide a thorough overview for those who would make further research of automated testing related to Jenkins Pipeline topic.

4 LIMITATIONS

Automation end-to-end testing systems are getting more recognition days by days for their testing methodology of checking the workflow of a product from the start to the end on an external interface. Despite of their advantage as to the convenience and flexibility aspects for the testing team, there are also limitations.

Firstly, end-to-end testing relies on developing time. An automation testing process can only be completed if the components work as required and are bug-free. The work of a testing team usually starts when a webpage is announced as ready for being tested before it is released into the production. Testers then check the webpage manually for bugs (or also called as errors, flaws or faults) and write manual test cases into JIRA whether there are bugs or not. Bugs are inevitable in the development process; testers should inform the developer team by creating a bug issue in JIRA if there are any. When the bugs are announced fixed, testers conduct tests manually once again for validation and write end-to-end scripts for testing their functions with multiple different values.

Secondly, if the source code evolves, the old test suites have a high chance of failure. An end-to-end testing system, specifically Nightwatch.js, uses XPath and CSS selector to locate an element on the external interface of the webpage by using Inspector; therefore, if the code shown in the Element tab got changed, the test scripts related to that element should be checked and fixed to guarantee performance. This results in the need for test maintenance which is considered costly.

Thirdly, there is a limited number of scenarios that needs automation testing. Nightwatch.js has a huge amount of API references that are all useful for software testing. However, there are circumstances where its commands cannot simulate desired end user behaviour. An automation test is used to help the testing process become faster and less redundant errors; it is not developed to cover all the basic to highly advanced webpage components but to ease the adverse presenting in this field.

Through all the mentioned limitations above that the author of this paper encountered, there should be further research to discard these challenges in the future. Nightwatch.js is still in its early development stages, therefore, new features may be developed to perfect and advance the system.

5 CONCLUSION

The comprehensive theory of the automated end-to-end testing system provided in this dissertation was a firm foundation for the author to validate the benefits of the system. Nightwatch.js was proved to be suitable for Quux Oy with its fully browser testing solution and consistent settings with development section. It is also useful for raising the testing performance by checking the complete flow of the system, and especially its great performance in reducing error from human behaviour.

A demonstration of Node.js-based test script was created for testing a circumstance where the authorization function occurs, along with methods to evolve the script for a DRY (do not repeat yourself) and clean structure. The major contributions of this work were presented in Chapters 2 and 3. In Chapter 2, the theory of end-to-end automated testing system and Nightwatch.js has been studied and, in Chapter 3, the complete setup and several additional extension features have been examined.

In a nutshell, although there are limitations in the Nightwatch.js system, the target of this thesis, which is to set up an automation testing system from scratch, was achieved. The test of this automated user interface system gave a positive result and the enhancements was successfully functioned. This thesis also studies the theoretical possibilities of automation testing in a continuous integration and continuous delivery environment. The author believes that further research about these features would be conducted and there would be plenty of practical demonstrations, while this dissertation resolves all remaining questions regarding the commissioner's requirements.

REFERENCES

- (2018, September 25). Retrieved from Screenster: <https://screenster.io/end-to-end-testing/>
- About Nightwatch. (2020). Retrieved from nightwatch.js: <https://nightwatchjs.org/about>
- About Nodejs. (2020). Retrieved from Nodejs: <https://nodejs.org/en/about/>
- Angler. (2019). *AUTOMATED UI TESTING WITH JENKINS*. Retrieved from https://www.angleritech.com/case_studies/automated-ui-testing-jenkins/
- API Reference. (2020). Retrieved from nightwatchjs: <https://nightwatchjs.org/api>
- aromano, b. M. (2019). *ramda-npm*. Retrieved from npmjs: <https://www.npmjs.com/package/ramda>
- Belmont, J.-M. (2018). Chapter 8: Building Pipelines with Jenkins. In J.-M. Belmont, *Hands-On Continuous Integration and Delivery*. Packt Publisher.
- Bodnar, J. (2019). *The node-postgres first example*. Retrieved from zetcode: <http://zetcode.com/javascript/nodepostgres/>
- Carlson, B. (2019). *pg-npm*. Retrieved from npmjs: <https://www.npmjs.com/package/pg>
- Cohn, M. (2009). Succeeding with Agile. In M. Cohn.
- Collins Dictionary. (2020). *Definition of Tool*. Retrieved from Collins English Dictionary: <https://www.collinsdictionary.com/dictionary/english/tool>
- Const. (2019). Retrieved from Mozilla: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const>
- Denisov', D. (2020, March 20). *Nightwatch HTML Reporter*. Retrieved from BLocks: <https://bl.ocks.org/denji/204690bf21ef65ac7778>
- Design Principles and Patterns. (2009). In J. G. Lisa Crispin, *More Agile Testing: Learning Journeys for the Whole Team*.
- Dumay, J. (2016, May 26). *Introducing Blue Ocean: a new user experience for Jenkins*. Retrieved from Jenkins: <https://www.jenkins.io/blog/2016/05/26/introducing-blue-ocean/>
- Gemoli, D. (2019). Retrieved from <https://github.com/coding-with-dom/intro-to-nightwatchjs>
- Gemoli, D. (2019). Why is (end-to-end) testing important (to me)? And what does the testing pyramid actually mean?
- Introduction. (2020). Retrieved from <https://robotframework.org/>
- Jackson, N. (2017). In N. Jackson, *Building Microservices with Go*. Packt Publishing.
- JIRA. (2019). Retrieved from Atlassian : <https://www.atlassian.com/software/jira>
- JIRA Software features. (2019). Retrieved from Atlassian: <https://www.atlassian.com/software/jira/features>
- Let's Do IT: About . (2020). Retrieved from LinkedIn: <https://www.linkedin.com/company/let's-do-t-bonuseurope-o%C3%BC/about/>
- Lets Do IT! (n.d.). Retrieved from <https://letsdoit.fi/en/main-page/>
- Lisa Crispin, J. G. (2009). The Dangers of Putting Off Test Automation. In J. G. Lisa Crispin, *More Agile Testing: Learning Journeys for the Whole Team*.
- McAllister, J. (2015). The Jenkins user interface. In *Mastering Jenkins*. PACKT Publisher.
- MG, R. P. (2015). *Learning Selenium Testing Tools - Third Edition*.

- Nguyen, S. (2017, May). *Continuous Integration with Jenkins*. Retrieved from <https://viblo.asia/p/continuous-integration-with-jenkins-bai-1-gioi-thieu-ve-ci-va-jenkins-OeVKBggEZkW>
- Nightwatch.js*. (2019). Retrieved from <https://nightwatchjs.org/>
- Pathania, N. (2017). *Learning Continuous Integration with Jenkins - Second Edition*.
- pgAdmin* . (2020). Retrieved from pgAdmin: <https://www.pgadmin.org/>
- Pipeline*. (2020). Retrieved from Jenkins.io: <https://www.jenkins.io/doc/book/pipeline/>
- Pipeline Syntax* . (2020). Retrieved from Jenkins: <https://www.jenkins.io/doc/book/pipeline/syntax/>
- Pittet, S. (2020). *Continuous integration vs. continuous delivery vs. continuous deployment*. Retrieved from Atlassian: <https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment>
- Pittet, S. (2020). *End-to-end tests*. Retrieved from Atlassian: <https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing>
- Rajkumar. (2019). *Selenium Continuous Integration with Jenkins*. Retrieved from <https://www.softwaretestingmaterial.com/selenium-continuous-integration/>
- ramz. (2019, March 24). *Freestyle and Pipeline Jenkins Jobs*. Retrieved from techieatom: <https://techieatom.com/freestyle-and-pipeline-jenkins-jobs/>
- Relational model concepts . (2019). In A. V. Salahaldin Juba, *Learning PostgreSQL 11: A beginner's guide to building high-performance PostgreSQL database solutions, 3rd Edition*.
- Saxena, M. (2016). *Automation Testing with Jenkins: The Game Changer in Test Automation*.
- Screenster. (2018, April 5). *Manual vs automation testing of the UI*. Retrieved from Screenster: <https://screenster.io/manual-vs-automation-testing/>
- Smart, J. F. (2011). *Jenkins The Definitive Guide*. In J. F. Smart. O'Reilly Media;.
- Teixeira, P. (2013). Enter the automation era. In P. Teixeira, *Using Node.js for UI Testing*. Packt Publisher.
- Theory of operation*. (2019). Retrieved from Nightwatch.js: <https://nightwatchjs.org/gettingstarted>
- To, V. A. (2018, September 19). *Headless mode*. Retrieved from <https://vananhtooo.wordpress.com/2018/09/19/tim-hieu-ve-headless-browsers-trong-selenium-webdriver/>
- UAS, M. (2015). *SQL*.
- Unit Testing with Nightwatch*. (2019). Retrieved from Nightwatch.js: <https://nightwatchjs.org/guide/unit-testing-with-nightwatch/>
- Using before[Each] and after[Each] hooks*. (2019). Retrieved from Nightwatch.js: <https://nightwatchjs.org/guide#using-before-each-and-after-each-hooks>
- Webdriver*. (2019, November 24th). Retrieved from <https://www.w3.org/TR/webdriver/>
- What is CI/CD?* (2020). Retrieved from Red Hat: <https://www.redhat.com/en/topics/devops/what-is-ci-cd>
- What is git: become a pro at Git with this guide*. (2019). Retrieved from Atlassian: <https://www.atlassian.com/git/tutorials/what-is-git>
- What is the Purpose of Unit Testing?* (2014). In E. Dietrich, *Starting to Unit Test: Not as Hard as You Think*.
- What is XML*. (2018). Retrieved from <https://www.howtogeek.com/357092/what-is-an-xml-file-and-how-do-i-open-one/>

Working with Page Objects. (2019). Retrieved from Nighytwatchjs:
<https://nightwatchjs.org/guide/working-with-page-objects/>
Writing Custom Commands. (2019). Retrieved from Nightwatchjs:
<https://nightwatchjs.org/guide/extending-nightwatch/#writing-custom-commands>

DP-1952.js IN EARLY STAGE

```

module.exports = {
  '@tags': ['Dp-1952'],
  'step one: Login with wrong password': function (browser) {
    browser
      .url(url)
      .setValue('/*[@id="username"]', 'aWrongUserInput')
      .setValue('/*[@id="password"]', 'correctInput')
      .waitForElementVisible('body')
      .saveScreenshot("screenshots/accpw.png")
      .click('/*[@id="login-btn"]')
      .pause(5000)
      .verify.elementPresent('/*[@id="error-alert"]', 'Login failed')
  },
  'step two: Login with wrong username': function (browser) {
    browser
      .url(url)
      .setValue('/*[@id="username"]', 'correctInput')
      .setValue('/*[@id="password"]', 'aWrongPasswordInput')
      .saveScreenshot("screenshots/accpw.png")
      .click('/*[@id="login-btn"]')
      .pause(5000)
      .verify.elementPresent('/*[@id="error-alert"]', 'Login failed')
  },
  'step three: Login with correct account': function (browser) {
    browser
      .url(url)
      .setValue('/*[@id="username"]', 'correctInput')
      .setValue('/*[@id="password"]', 'correctInput')
      .saveScreenshot("screenshots/accpw.png")
      .click('/*[@id="login-btn"]')
      .pause(5000)
      .frame(0)
      .verify.elementPresent('/*[@id="terms-service-modal"]/div/div/div[1]/h2', 'Login successfully')
      .saveScreenshot("screenshots/DP-1952CorrectAccount.png")
      .end();
  }
}

```

testForRightAndWrongAccount.js PAGE OBJECT FILE

```

module.exports = {
  url: 'url',
  elements:{
    userAccountInput : {selector: '//*[@id="username"]', locateStrategy: 'xpath'},
    userPasswordInput : {selector: '//*[@id="password"]',locateStrategy: 'xpath'},
    submitButton : {selector: '//*[@id="login-btn"]',locateStrategy: 'xpath'},
    error : {selector: '//*[@id="error-alert"]/p', locateStrategy: 'xpath'}
  },
  commands: [{
    loginWrongUserName(userAccount,userPassword) {
      return this
        .setValue('@userAccountInput', userAccount)
        .setValue('@userPasswordInput', userPassword)
        .click('@submitButton')
    },
    loginWrongPassword(userAccount,userPassword) {
      return this
        .setValue('@userAccountInput', userAccount)
        .setValue('@userPasswordInput', userPassword)
        .click('@submitButton')
    },
    loginCorrectAccount(userAccount,userPassword) {
      return this
        .setValue('@userAccountInput', userAccount)
        .setValue('@userPasswordInput', userPassword)
        .click('@submitButton')
    },
    validate() {
      return this
        .verify.elementPresent('@error', 'Login failed');
    }
  }]
}

```


DP-1952.js AFTER BEING CUSTOMISED

```

module.exports = {
  '@tags': ['Dp-1952'],
  'step one: Login with wrong password': function (browser) {
    const page = browser.page.testForRightAndWrongAccounts();
    const userAccount = 'aWrongAccountInput';
    const userPassword = 'correctInput';

    page
      .maximizeWindow()
      .navigate()
      .loginWrongUserName(userAccount, userPassword)
      .validate()
      .saveScreenshot("screenshots/DP-1952WrongPassword.png")

    browser
      .end();
  },
  'step two: Login with wrong username': function (browser) {
    const userAccount = 'correctInput';
    const userPassword = 'aWrongPasswordInput';

    browser
      .loginAsACustomer(userAccount, userPassword)
      .verify.elementPresent('//*[id="error-alert"]', 'Login failed')
      .saveScreenshot("screenshots/DP-1952WrongUserName.png")
  },
  'step three: Login with correct account': function (browser) {
    const userAccount = 'correctInput';
    const userPassword = 'correctInput';

    browser
      .loginAsACustomer(userAccount, userPassword)
      .frame(0)
      .verify.elementPresent('//*[id="terms-service-modal"]/div/div/div[1]/h2', 'Login successfully')
      .saveScreenshot("screenshots/DP-1952CorrectAccount.png")
      .end();
  }
}
}

```