Timo Kurkinen

Testing automation to increase product quality

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication Technology

Thesis

10 April 2018

Author Title Number of Pages Date	Timo Kurkinen Testing automation to increase product quality 32 pages 10 April 2018
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Communication Networks and Applications
Instructors	Pekka Kinnunen, CTO Markku Nuutinen, Principal Lecturer

The purpose of the thesis was to create almost fully automated testing environment, so the company could free up man power for more productive working areas. The company had two test engineers who manually tested the application and it takes time from them to actually develop the application.

The plan was to get to know different testing frameworks that were the most compatible for the company's application. The application was coded with Ember, thus the key point was to find JavaScript based framework while also doing research about Python and Java. After that the plan was to code different kinds of tests for the application and thus more bugs can be found earlier and faster.

The outcome of the thesis was that the testing automation was running in the company and it covers as many parts of the code as possible. The company was also wiser about automated testing and the testers were closer to become full-time developers.

The results were fine considering how much time there was to write tests. While also doing manual testing and developing at the same time, there was still much learnt about testing automation and different frameworks and it is now on a right track.

Keywords	JavaScript, Testing, Ember, QUnit, Software development

Tekijä Otsikko Sivumäärä Aika	Timo Kurkinen Tuotteen laadun parantaminen testauksen automatisoinnin avulla 32 sivua 10.4.2018
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Tietoverkot ja sovellukset
Ohjaajat	CTO Pekka Kinnunen Yliopettaja Markku Nuutinen

Insinöörityön tarkoituksena oli luoda lähes täysin automatisoitu testausympäristö, sillä asiakasyrityksessä oli kaksi testaajaa, jotka hoitivat manuaalisesti kaikki testit. Tavoitteena oli saada enemmän työntekijöitä ohjelmoimaan, sillä koodareita oli viisi, joista kaksi hoiti myös testausta.

Työssä tutustuttiin erilaisiin testauskehikkoihin, jotka sopivat parhaiten yrityksen sovellukselle. Sovellus ohjelmoitiin Emberillä, joten tärkeintä oli löytää JavaScriptille sopivin kehikko, mutta samalla perehdyttiin myös Pythonin ja Javan tarjontaan. Tämän jälkeen kirjoitettiin erilaisia, mahdollisimman kattavia testejä, jotka käyvät mahdollisimman suurta osaa koodista läpi. Tämän ansiosta virheitä saattaa löytyä enemmän ja niitä pystyy helpommin korjaamaan.

Insinöörityön lopputuloksena testauksen automatisointi saatiin pitkälle yrityksessä ja testit kävivät läpi suurinta osaa koodista. Yritys oli myös tässä vaiheessa viisaampi testauksen automatisoinnin saralla, ja testaajat toimivat yhä enemmän sovelluksen kehittämisen parissa.

Tulokset olivat hyviä ottaen huomioon, kuinka paljon aikaa oli kirjoittaa testejä. Myös samanaikaisesti tapahtuvat manuaalinen testaaminen ja kehittäminen huomioiden testauksen automatisoinnista opittiin paljon lisää ja se on nyt oikealla tiellä.

Avainsanat	JavaScript, Testaus, Ember, QUnit, Ohjelmistokehitys

Contents

List of Abbreviations

1 Introduction 1

1.1 About company 1

1.2 Structure of thesis 2

2 Theory 2

2.1 Evolution of programming and testing 2

2.2 Importance of product quality 4

2.3 Profitability of test automation 5

2.4 Different testing techniques and classifications 7

2.4.1 Acceptance tests 9

2.4.2 Integration tests 10

2.4.3 Unit tests 10

2.5 About testing frameworks 11

2.6 Different options for different purposes 11

3 Choosing framework 12

3.1 Criteria for choosing framework 13

3.2 JavaScript testing frameworks 14

3.2.1 QUnit 14

3.2.2 Mocha 15

3.2.3 Jasmine 15

3.3 Limiting the top two 16

4 Implementation of framework 17

4.1 Setting up the framework 18

4.1.1 Tools 18

4.1.2 Management process with tools 20

4.1.3 Installation of QUnit 21

4.2 Getting started with integration tests 22

4.3 Impressing with acceptance tests 27

4.4 Finetuning with unit tests 28

4.5 Summary of testing classifications 31

5 Conclusions 31

References 33

List of Abbreviations

SaaS – Software as a Service

API – Application Programming Interface

GUI – Graphical User Interface

AD – Anno Domini

QA – Quality Assurance

ISTQB - International Software Testing Qualifications Board

UAT – User acceptance testing

BAT – Business acceptance testing

CMD – Command Prompt

CLI – Command language interpreter

URL - Uniform Resource Locator

BDD – Behaviour-driven development

HTML – Hypertext Markup Language

NPM - Node Package Manager

DOM - Document Object Model

EECS - Electrical Engineering and Computer Science

1 Introduction

The purpose of this thesis is to plan and create an automated testing environment using different testing frameworks and comparing them. The idea is to promote the company’s two test engineers to be almost full-time developers. The reason there is an ”almost”, is that manual tests are still needed because the application requires new features weekly to keep the customers happy and acquire more of them. There will be more about the company and application in the next subchapter.

The company has been planning to automate the testing for a long time and to keep focusing more on developing the application. There has been some small tests that are already automated but they are actually never used. Now is the perfect time to plan, do research and develop these tests as final thesis.

1.1 About company

The company is Movenium Oy which is part of Visma Software Oy. It is a SaaS company which was founded in 2005. At the end of 2016, Visma Software Oy bought the company and now it is merging into an even bigger corporation. Movenium has two offices; one in Finland and the other in Sweden. There are around 20 employees in Finland and five in Sweden.

Movenium has developed an application for all contractors for managing construction sites. In the old times, everything was done with papers for example adding worktime or adding user to a worksite. The application is a smart tool to store all data into cloud and contractors no longer need to fill out hundreds of papers. It is also easier for accounting companys to get all the data through integrations. In addition, the application also inclu-des taxman reporting, which fetches automatically all the information about worker from ”Tilaajavastuu”.

1.2 Structure of thesis

This thesis has five chapters. First chapter gives information about company and introduction of the thesis. Second chapter is about the theory of computer programming, testing and automated testing. It includes a brief overview about the evolution of computer programming and why has testing become so important. There is also information about different testing classifications and frameworks. Third chapter is the planning phase, which includes ratings of different testing frameworks and choosing what could be the best for this specific application. There is also comparison of different frameworks from other useful coding languages and limiting the options to two of the best. Fourth chapter is about implementing the framework to the service and actually making these tests with examples. It also includes the tools used in the process and summary table of the results. Final chapter includes analysis of test results and what was done and learnt during these months.

2 Theory

In the course of time, there has been many occasions where it is proven that an individual human being can make a lot of mistakes. There is not a perfect human, who manages to do all kinds of things accurately and conveniently. That is why engineers and developers of different times began to do research about programming and tried various tests to get to know the the behaviour of machines and what they are capable of.

2.1 Evolution of programming and testing

The very first programming observation was over 1200 years AD, when an inventor called Al-Jazari introduced first versions of automaton. Automaton is basically a self-operating machine that could do a very simple movement like waving a hand or turning head. The word comes from Greek and can be translated as ”acting of one’s own will”. This simple but revolutionary observation inspired many programmers and inventors in the future to take this way of thinking much further.

In the 19th century, computer programming took huge steps forward as in early years of the century a device called ”Jacquard loom” was invented that could produce entirely different weaves by changing the program. However the real breakthrough was in 1843 when the first algorithm was published that could calculate a sequence of Bernoulli numbers. In the later years of the century, machines were able to store data in machine-readable form and in 1940s a first version of electronic computer was declared. The developing of this series of inventions continues to this date.

At the same time as machines grew bigger and more complicated and consisted of much more code and mistakes, testing became more and more important. Back in the days programming wasn’t so easy and people were not educated as much as they are today. Testing has always been around and its meaning has increased; people have become more aware of the failures of programs. People have been testing programs for a pretty long time and it started to become general in 1940s; at the same time the first electronic computer was developed. The testing was done in small development teams and the key point then was to create small projects because in that way more testing could be done before releasing them. In the very first years of computing, the programs were not planned to work on many different platforms and that made testing less important because users’ computers were pretty similar. (1)

Automating these tests became more general in 1990s when the most ambitious developers began to think about the opportunities of computer testing itself. The first automation tests were burned in diskettes which were sold by these developers. There were lots of them and each one included a certain test. But the fun didn’t last very long because that became a big problem later on due to the all time growing knowledge of computer programming. As mentioned earlier, the first testing efforts were quite simple because the programs were so similar and at that time, automated tests could have worked very well. But in the modern era of programming, the problem became visible with automated tests and it was that every time a developer changed the program, the automated test should have changed also. Since it was automated, it ran the same algorithm and failed the test. Later on became many different testing frameworks that were able to test specific coding languages and were much easier to use by writing test scripts instead of using diskettes. (2)

2.2 Importance of product quality

Quality of software is very important nowadays when there is so much competition between companys. Who has got the best application or best tools to work with, will usually get customers easier. Although there are many more other areas, like marketing and selling a product, those areas become much easier, when there is a steady, well-programmed and functional product.

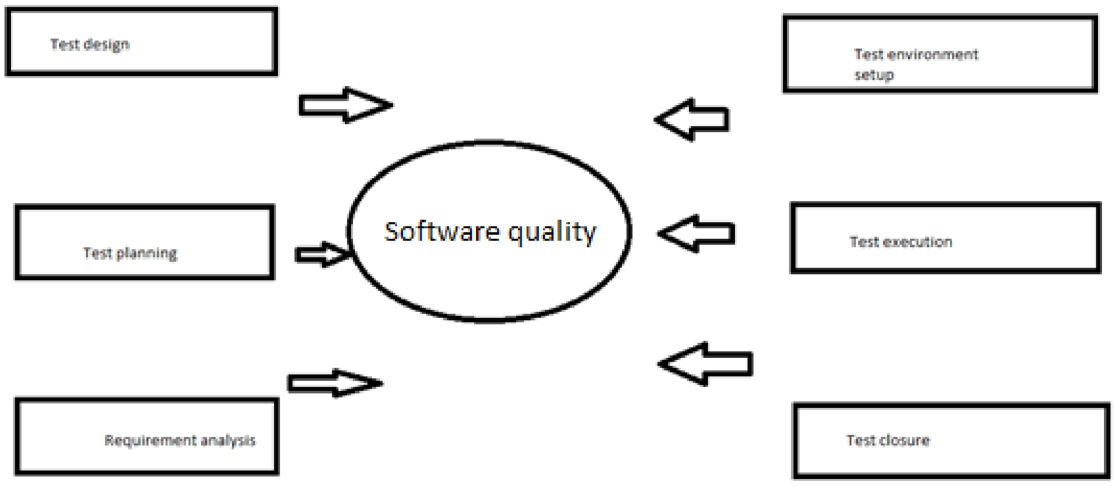


Figure 1. Software Quality Process (3)

In the Figure 1 is listed the main features of software quality process. It all begins with requirement analysis. At this part, company's QA team gathers documents and full analysis from the product owner and customers, what needs to be done and dealt with. (3)

Test planning is more architectural. Planning is probably the most important part of the process because a good plan is half of the job done. It includes the choosing and implementing of the right testing framework and thinking about the parts of the software that needs the automation and are important for the quality. At this phase, the outcome is a test plan and also the estimated time to make tests. (3)

In addition to the planning part, the team has to design the tests and at this phase the test engineer has to think about different testing techniques such as black box and white box testing to create test cases. If there is a need to make test scripts, it is also decided at this phase and how many lines of code does the test script use and how to design the code itself to be clear to modify and read. According to Doug Hoffman, ”a good test case design is neither too simple nor too complex” (4, p.41).

Code coverage means basically how many lines of code is the test covering. If a line has never been executed, it usually tells that the script didn't catch any bugs lurking in it. This type of coverage is usually called ’statement coverage’.” (5). A good general example of code coverage is that when a person is cleaning the house but forgets to clean the toilet for example, the toilet is still dirty and uncovered, so the same thing goes with coding and not testing all code.

When requirement analysis is done, and plan and design phase is in a good track, the next phase is to setup the environment. Installing the needed tools and frameworks and getting to know how they work and what they can do, is the main goal here.

Now that the installation succeeded and tester knows the basic features of the testing framework and potentially has some clue how to write tests with it; he can now begin to execute the tests. This phase may take a lot of time when framework is not maybe so familiar and the tester may have to do a lot of research how to test different kinds of fields. In addition, the knowledge of the application’s features and code has to be recognized. Outcome of this phase are execution of test cases and defect reports (3)

The last part of the process is to be confident that the tests are doing what they should do and afterwards presenting it to the product owner. The tester also has to analyze the results and finetune the code if needed to cover the code as much as possible. (6, p. 6; 3)

2.3 Profitability of test automation

”Testing has become the most popular verification and validation method in industry. To meet the market demands of producing high quality systems at low costs, testing should become more efficient and faster.” (7) There has been many analyses of why companys should automate testing and what the profit is doing so.

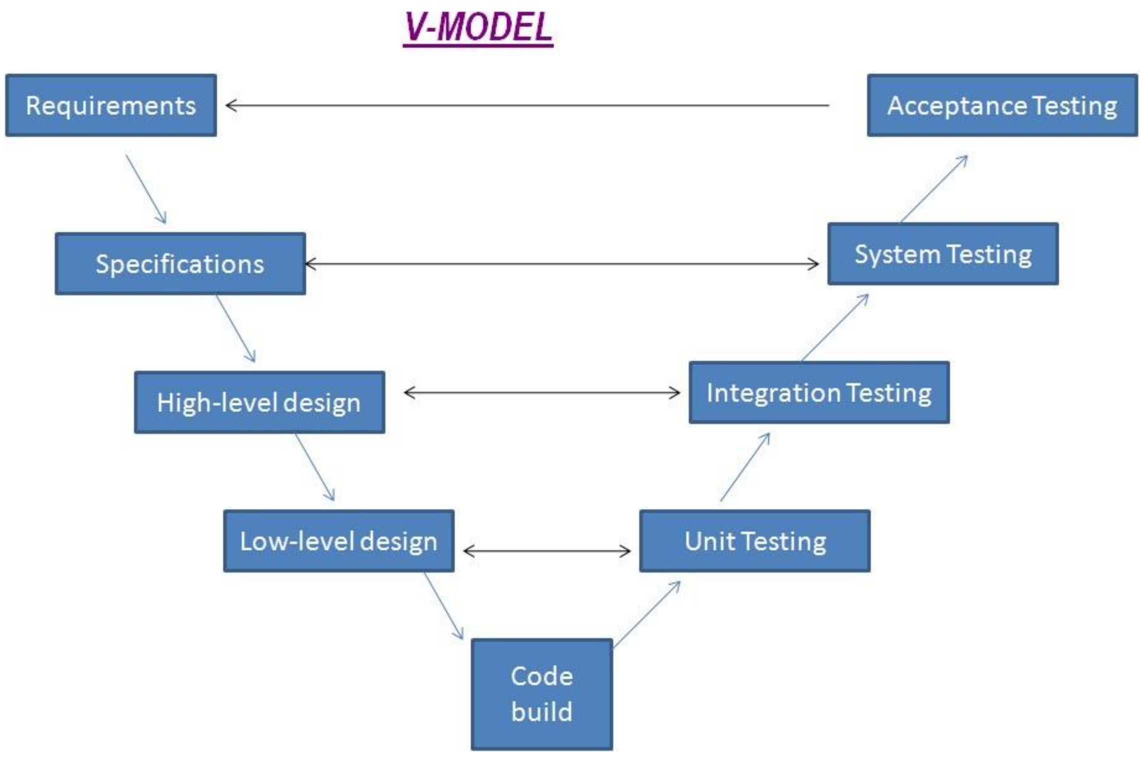


Figure 2. The V-Model [8]

The V-Model is a good illustration of how automated testing reduces errors in product and thus makes the product more profitable. It also frees up test engineers for more productive working areas due to the reduction in manual testing.

At the lowest level in V-Model is the coding itself and that has to be done before moving on to the actual tests. While going up the V-Model, upcomes unit tests at the lowest level and plan for them. The profitability in testing units is to reduce potential bugs and issues right away.

Following up are the integrations tests and plans to be able to test the relationships between these units. The profitability is to ensure that the working units are also working together like they are meant to be.

Lastly there are acceptance tests at the top of the V-Model and the profitability to test the application as a whole is to ensure that it is working in a live environment and it is doing what the customer wants it to do. (9)

When the tester doesn't have to do these steps manually, time is saved and thus also money. From the company's view this is naturally a great thing financially. On the other hand, from the tester's view it is also a good thing because there is no need to run the same tests all over again and in that way some errors can go through without noticing them. When the testing scripts have been written properly, tester can also take a break from manual testing in certain cases.

2.4 Different testing techniques and classifications

There are many efficient testing techniques and classifications which should be noted before jumping into the testing automation world.

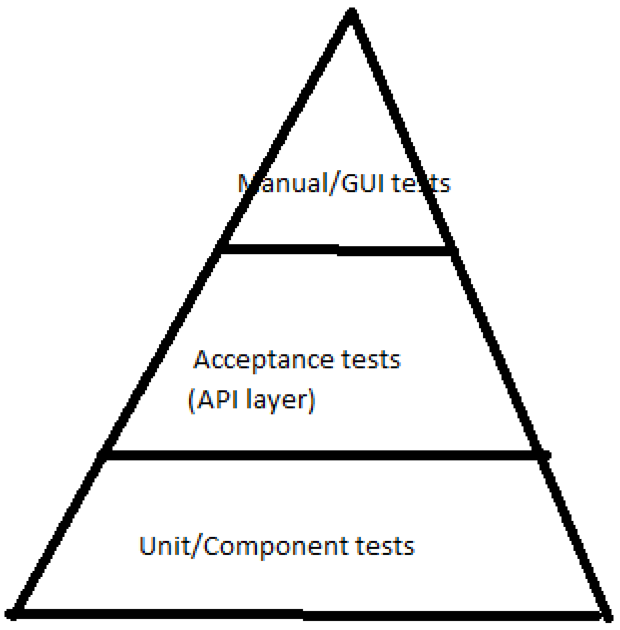


Figure 3. Testing classifications

In Figure 3, the importance of three layers of testing are classified in a pyramid. First comes the manual and GUI tests which are basically user level testing manually and testing different features – usually the most imporant ones - of the product, to be confident that they are working well. Next comes the acceptance tests to test the product as a whole to meet the customers' demands which are gathered in requirement analysis. And at the bottom layer are unit and component tests to test the functional part of the product; does every function work properly, does every field allow correct values and does the page give error.

Testing combines techniques that focus on five different dimensions:

- **Tester**: *Who* does the testing. User testing is focused on testing by members of the target market and also the tester must be able to test it like a normal user.
- **Coverage**: *What* gets tested. For example in integration tests it's the components and in unit tests it's the functions and other units.
- **Potential problems**: *Why* you are testing.
- **Activities**: *How* you test.
- **Evaluation**: *How to tell whether the test passed or failed.* (7, chapter 3 lesson 48)

These are the main questions to think when considering testing techniques. Testing tasks are often assigned on one dimension but the tester must be able to use all five dimensions. For example, company may want the tester to do function testing and it tells that every function must be tested. Following the dimensions, the tester has to think who shall do the testing, why should it be done and how, and whether the test passed or failed. By keeping these dimensions in mind, the tester is more capable of doing better choices and thus the quality of product increases.

There are few different types of testing that can and should be automated:

- **Functional**: operations perform as expected
- **Regression**: behaviour of the system has not changed
- **Exception:** goal is to make errors in system
- **Stress**: putting the product in its absolute limits
- **Performance**: testing that the performance rates doesn't drop and the product fills its requirements
- **Load**: determining the bullet points where software upgrade is needed (10)

The higher focus is on functional and regression testing which will be explained in the next chapter.

2.4.1 Acceptance tests

Acceptance test, often referred as smoke test is a test suite which goal is to check the basic functionality of the build. If the build fails the test, it is declared so unstable that it is not worth testing. (7, lesson 176) Typically, when for example a new release is done, the tester runs the acceptance test and if it fails, the developers correct the bugs and the smoke test is run again. When the test goes through without errors, the company can either release the update or continue to the next phase of testing.

The ISTQB defines acceptance as ”formal testing with respect to user needs, requirements, and business processes conducted to determine whether a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system”. (11). This means basically that the requirement analysis is fulfilled and user can use the product without problems.

There are two categories of acceptance testing:

- User acceptance testing
- Business acceptance testing (12, p.450)

User acceptance testing or UAT is conducted by the customers for them to accept the product so it fulfills their needs. The actual testing part is done either with third party company or by the customer itself but it is not very usual. Business acceptance testing or BAT is done in the company that developed the product and usually test engineers or developers do the acceptance tests before it is passed to the UAT. The development team executes test cases from the client’s contract, which includes the acceptance criteria.

Acceptance criteria is needed in any contractual agreement. The key point is to think about what criteria must the system meet in order to be acceptable. It consists of many quality attributes and the most important ones are:

- **Functional correctness and Completeness**: Does the product do what it supposed to do?
- **Accuracy**: will it give correct results?
- **Data integrity**: does the value of data remain unchanged when operations are done in later time?
- **Usability**: how easy it is to use?

- **Performance:** how good remains the performance of product when code is changed? (12, p. 452-455)

2.4.2 Integration tests

Integration testing, also known as feature or function integration testing is defined as testing different components and several functions of the system to see how they work together. (7, lesson 50) According to EECS, there are three different approaches of integration testing; based on: functional decomposition, call graphs or paths.

Functional decomposition includes that the product is created in a functional hierarchy and the problem is broken up into functions. It has the following strategies to do these tests:

- **Top-down**: first the functions that are closer to user interface and more important are tested
- **Bottom-up**: the exact opposite of top-down; the testing begins from the bottom level functions and the functions closer to user inteface are tested last
- **Sandwich**: combines top-down and bottom-up strategies
- **Call graph**: calling relationships between functions (13)
- **Big Bang**: testing only after integrating all modules (14, p. 12)

Integration testing can also be considered as a *process check*. If integration test fails or gives a lot of errors, it gives a signal that the unit testing may be in place. And if the errors occur in interfaces between the modules that should be working, the tester can trace the problem to interface specifications. (14, p. 4)

2.4.3 Unit tests

As clarified at the beginning of subchapter, a unit can be a function or a method of a class. Even the class itself can be considered as a program unit. (12, p. 51) So, the concept of unit testing is basically testing every single function and method of the code to be sure they return the right values. In integration testing, the principle is to test that the functions work properly together when calling them but in unit testing the focus is more on one function at the time.

2.5 About testing frameworks

Before choosing and comparing different testing frameworks that were considered in the process, it’s good to know what a testing framework really means. It’s a set of guidelines used for creating and designing test cases. (15) According to University of Colorado, testing framework can also be defined as ”the set of assumptions, concepts, and practices that constitute a work platform or support for automated testing”. (16, p. 9) It includes commands and tools for the testers to test more efficiently.

It’s almost impossible to make automated tests without a framework because it is responsible for:

- **Defining** the format
- **Creating** a mechanism to hook the application under test
- **Executing** the tests
- **Reporting** results (16 p. 9)

Without the four steps, testing automation cannot be done and today the testing frameworks are so well configured that it is easy to build the automation environment around the application.

2.6 Different options for different purposes

Each coding language requires a specific testing framework because the codes are naturally different and so are the libraries and tools in frameworks. It’s important for tester to acknowdge what code should be tested and thus choosing the right framework.

The biggest and the most popular coding languages like JavaScript, Java and Python offer many options to choose from. Especially JavaScript has testing frameworks for every coding framework there is. For example Ember.js framework has support for QUnit and it can be implemented with few commands, Jasmine has become very popular for testing Angular.js applications and Jest might be the best choice for testing React.js applications. (17)

Maybe the most well-known and proven frameworks for Java are JUnit and TestNG and those are compared in OverOps blog. (18) The most used testing framework is JUnit with 62% presence in all Java projects whereas 6% is the equivalent percentage for TestNG. There are a lot of frameworks to choose from for Java too. When in doubt, Selenium is always a good choice to begin with as it has a support for every language.

Python, however doesn’t have so many testing frameworks but the ones that exist are very effective and have a specific meaning to execute the tests. According to ”The Hitchhiker’s Guide to Python” (19), unittest and pyunit are specifically designed to do unit tests and doctest is designed to do integration and component tests. Pytest can also be noted as it allows also to test the application as a whole; so it is capable for acceptance testing.

3 Choosing framework

It is very important and at the same time difficult process to choose a testing framework that fits right for the service. The development team has to think about company’s own requirements and not necessarily choose blindly the framework that has the best reviews. Thus the first step is to start evaluating company’s requirements. (20, p. 249) The other approach is to start testing different kinds of frameworks and after that making the decision which one fits the best for the service. (21, p. 10)

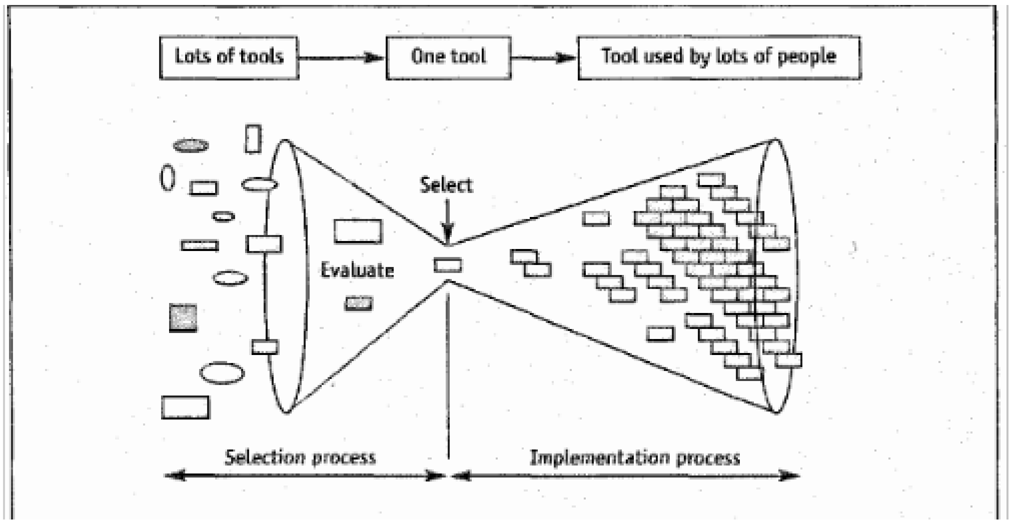


Figure 4. The tool selection and implementation process (20, p. 248)

There are five different test automation frameworks that should be noted, according to IBM; test script modularity, test library architecture, keyword-driven/table-driven testing, data-driven testing and hybrid test automation. (21, p. 1) In addition to the list, also behaviour-driven development testing is useful. (22)

These framework types were compared and investigated at the beginning chapters. Test script modularity and test library architecture were not comprehensive enough because they are directed for small tests, like testing a calculator. The difference between these two are that one creates test scripts to be executed while the other uses library files to call these scripts. (21, p. 2-5) Keyword-driven and data-driven testing are also similar to each other. They use more components of the system to create a bigger test case. For example the calculator example, it divides different clicking actions and returns the result. The difference between them are that the other uses script to test while the other keeps the test data in external files, like SQL and XML files. (21, p. 6-7; 22) Hybrid driven testing is the combination of all the other types and it is the most used type of test automation framework. This approach was basically what was chosen with the team in the company. Not to mention the last type, behaviour-driven development, it basically creates a platform that allows all the participants to actively test the product. (22)

Choosing the testing framework can also go wrong multiple times. The tester may be a beginner with the product or development overall, there may be bad planning and choosing the framework based on only reviews or starting the automation process in the wrong place. That is why testing automation should be a long process and should be planned carefully with many team members. (23) In the company, the choosing and planning process was discussed with product owner, one developer and two test engineers while also taking advice from senior developers.

3.1 Criteria for choosing framework

There are many things that the development team should have an eye on when choosing the best framework for the company. Besides choosing the framework specifically for the coding language, the team has to think about what testing framework would be the easiest to implement and fulfills the requirements. The team already knows how big the application is, so too small or too complex frameworks should be dropped. Also what type the application is; for example cloud-based application needs the cloud support in the framework. (24)

When the team has discovered options that are the most potential frameworks for the product, it’s good to compare their attributes and quality of test. The most important goal is to increase the code coverage which isn’t necessarily in an adequate level when testing manually. If some framework is so suitable for the service that it covers almost every part of the code with minor effort, then it’s an easy choice. Also one important element is how clearly the framework gives the results and reporting. Does it have a visual reporting of tests or is it only readable in CMD or some other CLI. (25)

It’s also good to be aware of wrong choices and incapability of the testing framework. The key point to handle the automation is to avoid mistakes as much as possible. The most usual and critical mistake would be that all tests are executed through user interface. In a long run, it slows down the service which is obviously not a good thing. Also keeping the testing and developing in different branches, so there won’t be any mismatches and suddenly neither one is working. The automated testing is basically coding and a good rule in coding is to keep the code in a readable form, so others can understand it and it is also easier to debug. (26)

3.2 JavaScript testing frameworks

It’s time to focus on testing frameworks that are relevant for the company. There are tens of testing frameworks aimed for JavaScript code, like QUnit, Mocha, Jasmine, Selenium, Chai, Sinon and Jest. (17) The next sections are focusing on three of the most suitable options for the company and at the end dropping one off to explicitly compare two of the best.

3.2.1 QUnit

When the research began at the beginning, test engineers used the first approach of the choosing process: discussed with the developers if they had skills or experience of testing Ember application. The first testing framework that rose up was QUnit, mainly because it is the easiest to implement and has all the testing packages built in. As a name, it may sound like it is capable for only unit testing but in addition to that, acceptance and integration testing are also possible. QUnit is heavily relying on jQuery which can be seen in the commands.

The basic principle of QUnit is to use its effective and simple commands to do actions in different pages. The most important ones are: visit, find, click, fillIn and assert. Basically the testing begins with function ”test” and continues with visiting different URLs in the service using the keyword ”visit”. When there are different text areas, checkboxes and dropdowns in the page, the tester can find these fields from the templates of the service and activate these fields using the ”click” and ”fillIn” actions to simulate the testing automatically. Finally the ”assert” actually does the test and reports it in web browser.

3.2.2 Mocha

Mocha is also a very effective JavaScript testing framework and it was the second discussion topic with the team. It is used for unit and integration testing and it’s a good candidate for BDD. It can also be implemented to test Ember applications but not as easily as QUnit. Mocha is very popular for testing Angular applications as it is the most suitable for them.

The basic principle of Mocha is to ”describe” what to test and after that using ”it” to assume what the expected result is. Mocha is using Chai library to ”expect” different results which are then compared to the tests themselves. Mocha reports the results in web browser too but it has to be initialized at the beginning of the test and then requested at the end. (27)

3.2.3 Jasmine

Jasmine was the third framework that could be suitable for the company. Like the other two, it is a good and functional testing framework and also can be implemented to test Ember application. It is very similar to Mocha as it is also aimed for BDD and has very similar commands. The difference is that the ”expect” action doesn’t need the Chai library to work and with Jasmine the test code and equal action can be written in one line while in Mocha it is written separately.

Jasmine is also capable of testing Ruby and Python as it has extensions for them too. It was previously JsUnit, a port of Java testing framework JUnit to JavaScript. The aim of Jasmine is to be easy to read, write and implement to different kinds of JavaScript projects. (28)

3.3 Limiting the top two

Based on the easy implemention and discussion with the team members, QUnit was a clear choice to be taken as one testing framework. So the other one to be also investigated had to be either Mocha or Jasmine. Because of the limited time, easy implementation and positive reviews, Mocha was the second testing framework to be inspected.

The good part about these two testing frameworks is that it’s very simple to implement from one to another. There are two ways to do this: using the Mocha-QUnit interface or changing the keywords to match the code.

The interface option means that the tester first runs a command ”npm install mocha-qunit-ui –save-dev” to install the interface. After that the tester loads the file using Node.js which is a JavaScript runtime environment to execute the code in server. The command is shortened as ”mocha –ui mocha-qunit-ui <test-file>”. Then the tester adds ”ui: qunit” in both test file and HTML file, where the template is. Finally the test is ready to run under Mocha. (29)

The second option is to simply convert the QUnit commands to be using Mocha commands or vice versa. The key point is to know what commands do the same things in different frameworks. For example, the most important commands in QUnit are ”module”, ”test” and ”assert”, which convert in Mocha as ”describe”, ”it” and ”expect”. Using this option requires the tester to be very careful especially when these is a lot of testing code. (30)

Table 1. Comparing the features of QUnit and Mocha with Ember application

	QUnit	**Mocha**
Popularity (31)	1 157 total commits 3 630 Github stars	2 263 total commits 12 480 Github stars
Setup	Easy installation with one line and built-in support for test helpers	A bit harder installation with few lines and requires libaries like Chai
Commands	Simple and straightforward, like visit, find, click	Needs the Chai library for expect action but basic

		commands simple, like describe and it
Reporting	Clear reporting in web browser	URL needs to be initialized in code as well as request to web browser, clear reporting
Ease-of-use	Very easy to use along with the actual code	Easy to use once the setup is done

The basic features of the testing frameworks have been compared in the Table 1. That will give a good abstract based on reviews and own experience where to continue with the automated testing.

4 Implementation of framework

Based on the Table 1, research and discussion with team members, the best choice to start with this application was QUnit. There has already been some small testing in the company with QUnit but now it needs to be taken more seriously. As seen on the Figure 4 about the choosing and implementation process, there are first lots of testing frameworks as mentioned in Section 3.2. Then there is that one tool which gets to be chosen after the choosing process. Now the implementation process reverses it in a way that there is one tool used by many people.

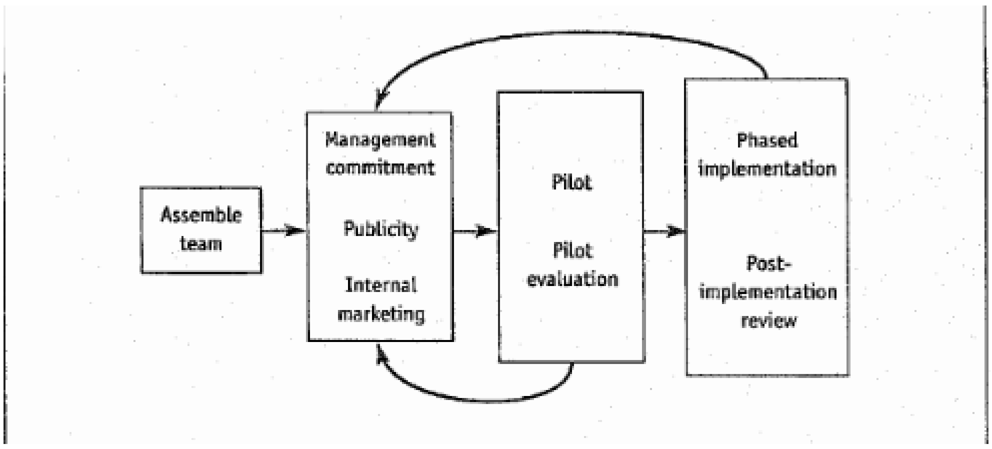


Figure 5. Overview of the implementation process (20, p. 284)

The implementation process itself is not so complex event. Once the testing team knows the testing framework and what commands are useful, they can begin to write tests with it. Figure 5 shows how the implementation process could be done in a medium to large company. First the testing team declares what framework they are choosing. Then they train other developers or team members if necessary how to use it. After that they implement the framework to the service and receive a commitment from the management team.

In a smaller company like Movenium, the basic principle was similar but the development team worked much closer and more together and the test engineers were more responsible of the process. Due to the built-in feature with Ember and QUnit, the implementation process was simple and it will be discovered in the next chapter.

4.1 Setting up the framework

As mentioned in the earlier sections and various references, QUnit matches very well with Ember. Thus the implementation and installation process is presumably simple and gives no harm. Using the knowledge of the company's developer who has done testing a little bit already and visiting QUnit's website (32) to read full instructions, the setting up could begin. First comes the discovering of the tools to work with, then getting to know the process of the whole scene and lastly installing the QUnit and beginning to write tests.

4.1.1 Tools

To be able to use any testing framework, there must be the actual code stored somewhere. At Movenium, the service is called JetBrains PhpStorm which is a popular code editor and it supports various coding languages, like JavaScript and PHP.

Figure 6. JetBrains PhpStorm [33]

All code that is included in the application that Movenium is developing is stored in PhpStorm and the UI is very simple and effective to use also in testing automation. After the installation of the testing framework, various files are created to the library that will be shown later on.

SourceTree is also playing a big part of the testing and coding process. It is a source control system that allows to edit and peek at other developers' codes and create pull requests for the senior developers. These pull requests are like a safety system that allows the more experienced developers to check the quality of code before moving it forward.

Figure 7. SourceTree [34]

The service itself looks a bit messy for users that are not so familiar with it but once the basic features are learnt, it is quite effective system. In the next section there will be a figure of how it looks regarding to the testing part.

Figure 8. Github [35]

Third tool or website is Github; a very well-known software development platform. It is basically a platform to see the changes of the files other developers have coded and also it is so widely used service so there can also be seen other people's projects and thus it allows facility to solve own problems with the help of huge community.

Figure 9. PowerShell [36]

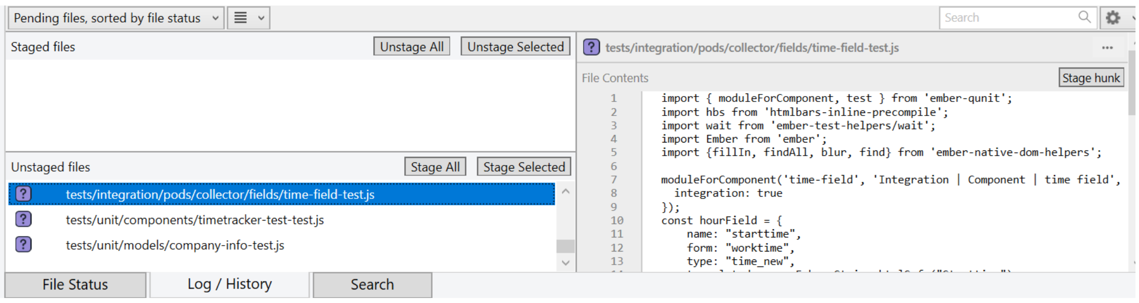
To be able to get the development environment running as well as the test server, there has to be some terminal to begin those actions. For Microsoft, an effective tool is Windows' own PowerShell which has been used constantly during the process. The whole automated testing and coding parts begin using this tool.

4.1.2 Management process with tools

The earlier section gave some vision which tools can be useful to begin with testing automation and these tools are also used to develop the application further and fixing some bugs. Learning how to use these tools together was done with help of developer.

The whole process begins with opening the PowerShell as administrator and launching the development environment using "ember serve -environment <back-end name>" command. This command allows access to the development environment where the developer can see the code changes in GUI and also the test reporting page.

Once the environment is running, SourceTree can be launched to create testing branches.



tests/integration/pods/collector/fields/time-field-test.js
tests/unit/components/timetracker-test-test.js
tests/unit/models/company-info-test.js

tests/integration/pods/collector/fields/time-field-test.js

```
import { moduleForComponent, test } from 'ember-qunit';
import hbs from 'htmlbars-inline-precompile';
import wait from 'ember-test-helpers/wait';
import Ember from 'ember';
import {fillIn, findAll, blur, find} from 'ember-native-dom-helpers';

moduleForComponent('time-field', 'Integration | Component | time field',
  integration: true
});
const hourField = {
    name: "starttime",
    form: "worktime",
    type: "time_new",
```

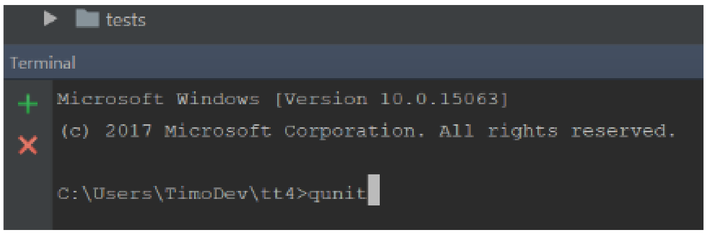
Figure 10. Usage of SourceTree

When the right branch is selected, the testing scripts can be written in PhpStorm, which allows to see the code specifically done for the branch in SourceTree. In that way, others can also look up for the test scripts written in various causes, like in Figure 10. When others pull the code tester has done from the SourceTree, they can now also use and edit the scripts. The branch can also be pushed to Github where it is stored nicely and the code is visually more readable there.

4.1.3 Installation of QUnit

The installation is quite simple and it includes few commands. After that it creates a file to the coding platform and then the test reporting page can be launched.

The first thing to do is launching PowerShell and globally installing QUnit package. The command is "npm install -g qunit", where npm is a package manager that is used in JavaScript and it is also the default package manager for Node.js. (37) The "-g" stands for global, so that the installation can be used anywhere in the system



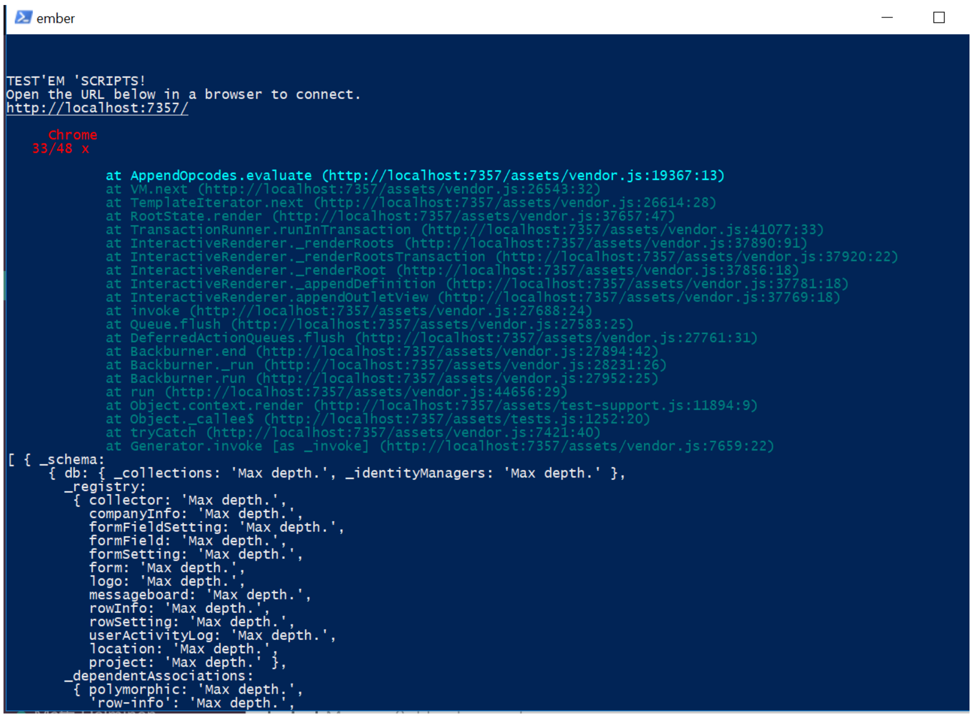
```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\TimoDev\tt4>qunit
```

Figure 11. Installing the QUnit into the service

After the installation is done, the tester should navigate to the file, where all codes have been pulled. As seen in Figure 11, the navigated file path is seen in the PhpStorm's own terminal and the tester now runs the command "qunit" to create the test files to begin with. There can also be seen the file "tests" that are created after the command has been run.

After that, it creates the file "tests" and few other files, some helpful for the testing and some not so helpful. For example ”test-helpers” is a useful file because it includes the various commands, like ”click”, ”fillIn” and ”find”. It also creates files like "index.html" and "eslintrc.js" but those are default files and there is a low priority to alter them. Index.html has been configured by the developer earlier and it includes the necessary links and scripts for reporting the tests and eslintrc.js tells the framework to test Ember specifically as it can be also used to fix linting errors.

Part of the installation is also to launch the test reporting page and check that it works. There are two ways to launch it: using PowerShell or other terminal and running a command or visiting the right web page in browser.



```
TEST'EM 'SCRIPTS!
Open the URL below in a browser to connect.
http://localhost:7357/

     Chrome
   33/48 x

            at AppendOpcodes.evaluate (http://localhost:7357/assets/vendor.js:19367:13)
            at VM.next (http://localhost:7357/assets/vendor.js:26543:32)
            at TemplateIterator.next (http://localhost:7357/assets/vendor.js:26614:28)
            at RootState.render (http://localhost:7357/assets/vendor.js:37657:47)
            at TransactionRunner.runInTransaction (http://localhost:7357/assets/vendor.js:41077:33)
            at InteractiveRenderer._renderRoots (http://localhost:7357/assets/vendor.js:37890:91)
            at InteractiveRenderer._renderRootsTransaction (http://localhost:7357/assets/vendor.js:37920:22)
            at InteractiveRenderer._renderRoot (http://localhost:7357/assets/vendor.js:37856:18)
            at InteractiveRenderer._appendDefinition (http://localhost:7357/assets/vendor.js:37781:18)
            at InteractiveRenderer.appendOutletView (http://localhost:7357/assets/vendor.js:37769:18)
            at invoke (http://localhost:7357/assets/vendor.js:27688:24)
            at Queue.flush (http://localhost:7357/assets/vendor.js:27583:25)
            at DeferredActionQueues.flush (http://localhost:7357/assets/vendor.js:27761:31)
            at Backburner.end (http://localhost:7357/assets/vendor.js:27894:42)
            at Backburner._run (http://localhost:7357/assets/vendor.js:28231:26)
            at Backburner.run (http://localhost:7357/assets/vendor.js:27952:25)
            at run (http://localhost:7357/assets/vendor.js:44656:29)
            at Object.context.render (http://localhost:7357/assets/test-support.js:11894:9)
            at Object._callee$ (http://localhost:7357/assets/tests.js:1252:20)
            at tryCatch (http://localhost:7357/assets/vendor.js:7421:40)
            at Generator.invoke [as _invoke] (http://localhost:7357/assets/vendor.js:7659:22)
[ { _schema:
     { db: { _collections: 'Max depth.', _identityManagers: 'Max depth.' },
       _registry:
        { collector: 'Max depth.',
          companyInfo: 'Max depth.',
          formFieldSetting: 'Max depth.',
          formField: 'Max depth.',
          formSetting: 'Max depth.',
          form: 'Max depth.',
          logo: 'Max depth.',
          messageboard: 'Max depth.',
          rowInfo: 'Max depth.',
          rowSetting: 'Max depth.',
          userActivityLog: 'Max depth.',
          location: 'Max depth.',
          project: 'Max depth.' },
       _dependentAssociations:
        { polymorphic: 'Max depth.',
          'row-info': 'Max depth.',
```

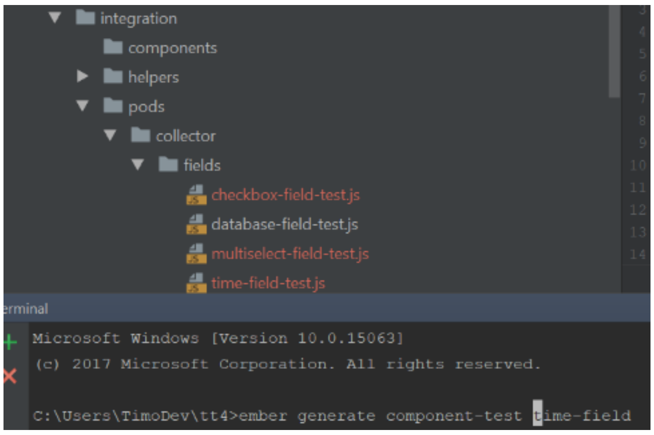
Figure 12. The outcome of running the command

The command that has been run in the Figure 12 is "ember test --server". It opens up in the PowerShell like that and also launches the browser to run all the tests that have been written in the testing files. There are currently 33 passed tests from 48 total tests because the test server has some misconfigurations that fail.

The other option is to simply open - most preferably - Chrome and navigate to the right file in the URL. In this case it is localhost:4200/tests. When hitting enter it opens the reporting page and more about that will be covered during the testing phase in the next sections.

4.2 Getting started with integration tests

Usually when companys begin their automated testing, the first type of testing is acceptance testing. That is because it is good to test first that the application is working as it should be. Due to the lack of experience in developing and testing, the process began with integration testing because it felt like an easier approach. As explained in earlier section about integration testing, it is about testing different components and how they react with each other.



```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\TimoDev\tt4>ember generate component-test time-field
```

Figure 13. Integration test files

To begin with integration testing, there has to be some files under the "tests" file where the testing scripts can be stored. A command to create these files is "ember generate component-test <file-name>".

This creates a JavaScript testing file under ”tests/integration/components” file path and it is named after the file name given with ”-test.js” suffix. In the Figure 13, the time-field test file is in fields file because it was moved there afterwards due to organizational reasons.

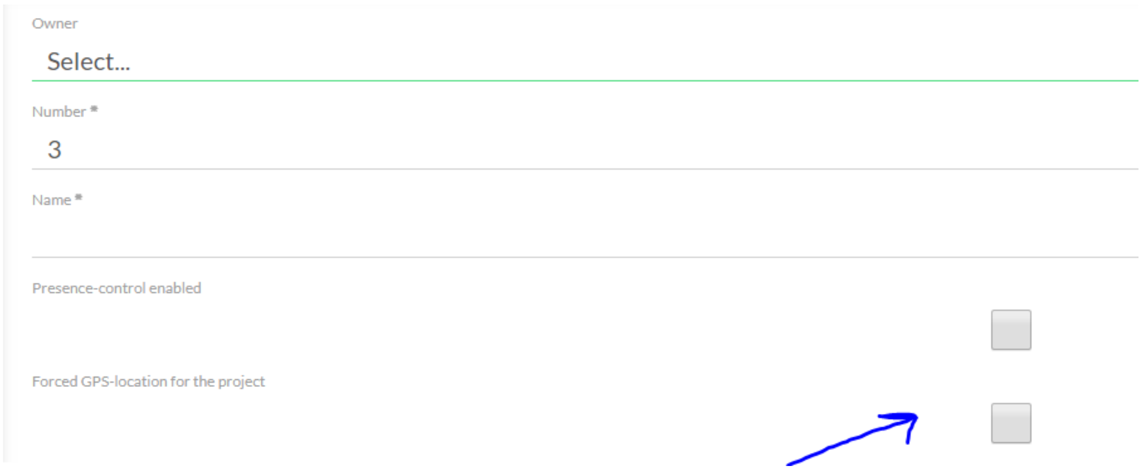
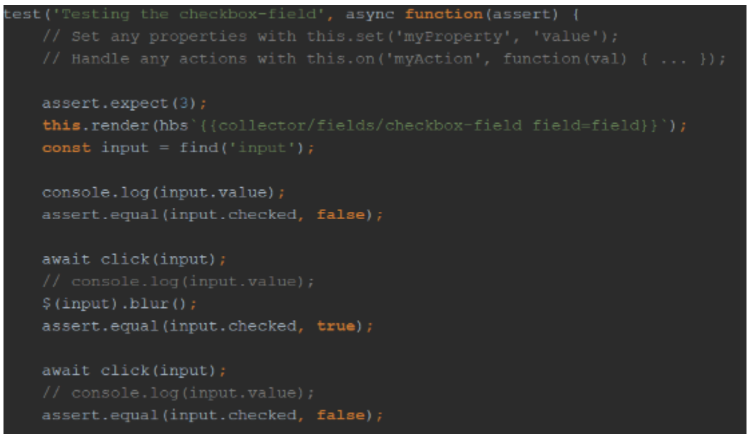


Figure 14. Checkbox field in application

In the application, it is important to test different fields, so that they are working properly. These integration or component tests are more like field tests in the company. There are many different kinds of fields in the application, like checkbox field, hour field, database field, text field or dropdown field. In Figure 14 is an example of checkbox field as seen in the application. This kind of test is very basic and a good practice to continue with more advanced testing. Testing this field is to ensure that first the checkbox field is empty, then the user clicks it and it is checked and when clicked again it is back unchecked.



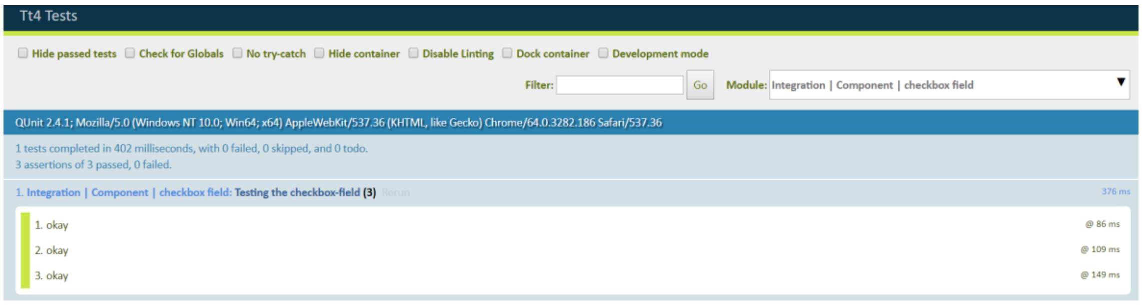
```
test('Testing the checkbox-field', async function(assert) {
    // Set any properties with this.set('myProperty', 'value');
    // Handle any actions with this.on('myAction', function(val) { ... });

    assert.expect(3);
    this.render(hbs`{{collector/fields/checkbox-field field=field}}`);
    const input = find('input');

    console.log(input.value);
    assert.equal(input.checked, false);

    await click(input);
    // console.log(input.value);
    $(input).blur();
    assert.equal(input.checked, true);

    await click(input);
    // console.log(input.value);
    assert.equal(input.checked, false);
```

Figure 15. Code used to test the checkbox field

When the commands and their actions are learnt, the actual coding could start. In Figure 15, the checkbox field test is coded using the commands that QUnit has. The test begins with "test" keyword so that the testing framework knows that the test begins here. There is also description about the test and asynchronous function with parameter ”assert”.

These are the basic features of a test. The first command of the function is ”expect” and number three. This tells the test that it should only expect three different asserts and gives an error if it is otherwise. After that, the code includes render statement and a file path to the actual checkbox field in the application. This file includes the needed codes to get the checkbox field visible in the application. The file is basically HTML code which includes few lines for the input.

Then the input field is declared in a variable using "find" keyword to actually find the input selector from the DOM. This constant is then used to first check that the input field is false when no action has been triggered. Using the "click" keyword, it triggers the action like the user is actually clicking the checkbox field in the application and then it is compared using the ”assert.equal” statement that the value is changed to true. The code includes also ”console.logs”, ”awaits” and ”blur”; console.log is used to make it easier for tester to see the events in console, await is used in asynchronous testing to wait for the earlier call to end and blur is used to focus out the event in application.


Tt4 Tests

☐ Hide passed tests ☐ Check for Globals ☐ No try-catch ☐ Hide container ☐ Disable Linting ☐ Dock container ☐ Development mode

Filter: Go Module: Integration | Component | checkbox field

QUnit 2.4.1; Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.186 Safari/537.36

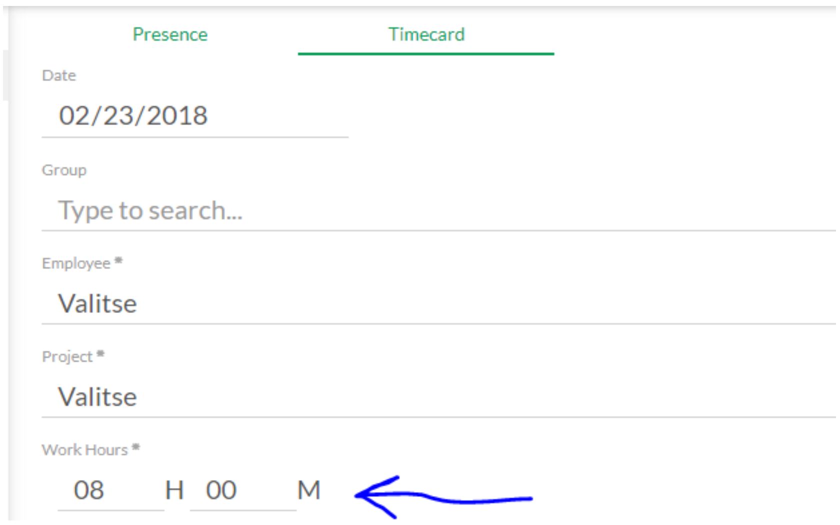
1 tests completed in 402 milliseconds, with 0 failed, 0 skipped, and 0 todo.
3 assertions of 3 passed, 0 failed.

1. Integration | Component | checkbox field: Testing the checkbox-field (3) Rerun 376 ms

1. okay @ 86 ms
2. okay @ 109 ms
3. okay @ 149 ms

Figure 16. Test server to report the results

When the test is run in browser and the right test script is selected from the dropdown, the outcome looks like in Figure 16. The reporting page includes many components for example to filter, hide passed tests or disable linting. However these are not so useful in small tests like this but in bigger tests they may be very handy. The test itself includes three parts where is text "okay". This means that in code there were three ”assert.equals” to check the change in checkbox field and they are now seen as passed tests. So the checkbox field is a working component in the application. If the test had been failed, there would be more red on the screen and report what the test expected and what was the result. More about that on the later sections.


Presence
Timecard

Date: 02/23/2018

Group: Type to search...

Employee *: Valitse

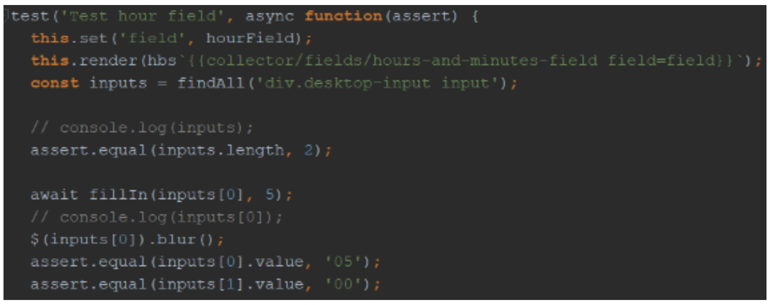
Project *: Valitse

Work Hours *: 08 H 00 M

Figure 17. Hours field in application

Another example of component test is testing the time field in application. The working ethic is similar to the checkbox field test, only the code looks a bit different. In Figure 17 is a picture what the hour field looks like in the application. There are simply two input fields, one for hours and the other for minutes. The hour field should only take numbers and it should change the one digit hours and minutes automatically to two-digit numbers.

So basically when user puts for example "8", it should be automatically changed to "08" when the focus is out of the field. It should also put automatically two zeros in minutes field when only hours field is filled.

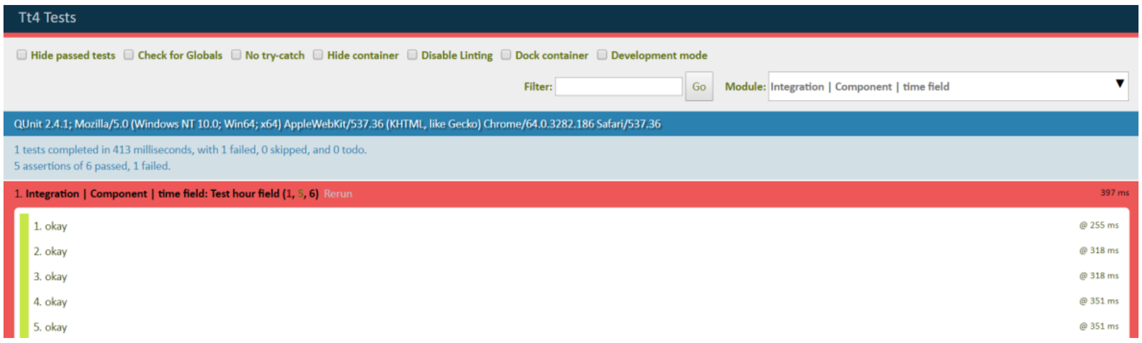


```
test('Test hour field', async function(assert) {
  this.set('field', hourField);
  this.render(hbs`{{collector/fields/hours-and-minutes-field field=field}}`);
  const inputs = findAll('div.desktop-input input');

  // console.log(inputs);
  assert.equal(inputs.length, 2);

  await fillIn(inputs[0], 5);
  // console.log(inputs[0]);
  $(inputs[0]).blur();
  assert.equal(inputs[0].value, '05');
  assert.equal(inputs[1].value, '00');
```

Figure 18. Code to test the hours input

In Figure 18 is part of the test code used to test the input fields. The beginning looks similar to the checkbox field except the field name is naturally different and also the rendered file path is targeted at the hours and minutes field in the code. After that the two input fields are found in the file which was rendered and they are in a class called ”desktop-input”. Using the ”findAll” statement, the test script tries to find all input fields under that class.

The first ”assert.equal” compares the constant that was declared earlier to ensure that there are two input fields: hours and minutes. Then the script tells the application to fill in number "5" in the first element of the array. The array has both of the inputs as elements and number "0" means the first field of the array which is the hour field. When user has filled the hour field, "blur" is called like in checkbox field test to focus out of the field. This part of the test now compares that the hour field gets the value "05" and minutes field gets the value "00" as seen in Figure 17.



Tt4 Tests

☐ Hide passed tests ☐ Check for Globals ☐ No try-catch ☐ Hide container ☐ Disable Linting ☐ Dock container ☐ Development mode

Filter: Go Module: Integration | Component | time field

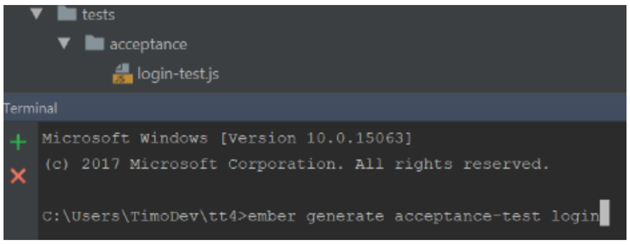
```
QUnit 2.4.1; Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.186 Safari/537.36

1 tests completed in 413 milliseconds, with 1 failed, 0 skipped, and 0 todo.
5 assertions of 6 passed, 1 failed.

1. Integration | Component | time field: Test hour field (1, 5, 6) Rerun                397 ms
  1. okay                                                                           @ 255 ms
  2. okay                                                                           @ 318 ms
  3. okay                                                                           @ 318 ms
  4. okay                                                                           @ 351 ms
  5. okay                                                                           @ 351 ms
```

Figure 19. Test server results for hours input

The reporting page is like in the checkbox field test and it is easy to test different script using the dropdown. The reporting page looks red, like in Figure 19 when some assert doesn't pass. This is only to demonstrate the different look of the reporting page and the failed test is not seen in the figure.

4.3 Impressing with acceptance tests

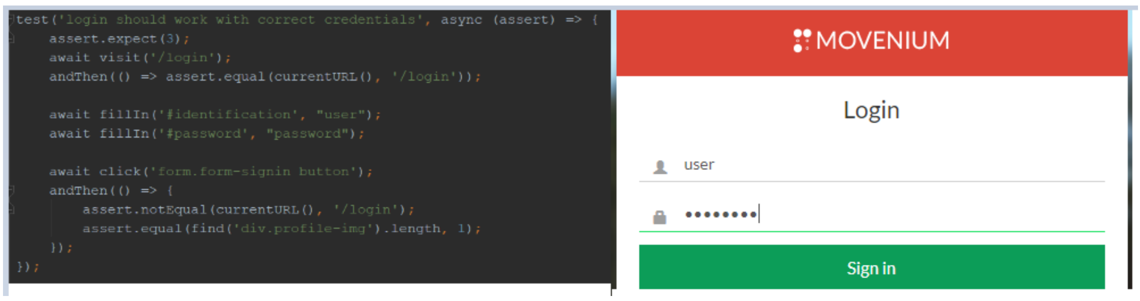
When some integration tests have been added to the system, it was time to move forward for some more challenging tasks, acceptance tests. Due to the limited time and lack of experience in developing, there is not many acceptence tests in the system yet and that will be an important issue in the spring and summer.



```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\TimoDev\tt4>ember generate acceptance-test login
```

Figure 20. Acceptance test files

However, the creation of files are shown in Figure 20 and it is very similar to the integration tests, except "component-test" is replaced with "acceptance-test". It also creates the file in different file path and in this case it's "tests/acceptance". In this example, this is how it looks when login test was created. The example of basic acceptance test can be seen in Figure 21. The login screen has two text fields like every other login screen: username and password and also the submit button to sign in.



```
test('login should work with correct credentials', async (assert) => {
    assert.expect(3);
    await visit('/login');
    andThen(() => assert.equal(currentURL(), '/login'));

    await fillIn('#identification', "user");
    await fillIn('#password', "password");

    await click('form.form-signin button');
    andThen(() => {
        assert.notEqual(currentURL(), '/login');
        assert.equal(find('div.profile-img').length, 1);
    });
});
```

Figure 21. Login test code and GUI

The test script begins similarly as in integration test, and this time the visiting page is "/login". What makes acceptance testing harder than integration testing earlier is to find the right elements from DOM. Next three lines include the selectors which can be found for example right clicking the text field and inspecting elements. In login test, it is quite easy because those two text fields have their own id and thus can be selected using the "#" mark. Once the credentials have been filled correctly, user clicks the sign in button which can also be fetched from DOM. This button doesn't have id, so the DOM has "form" class and that includes the sign in button class which is separated using dot. These selectors are basic jQuery code that QUnit is using.

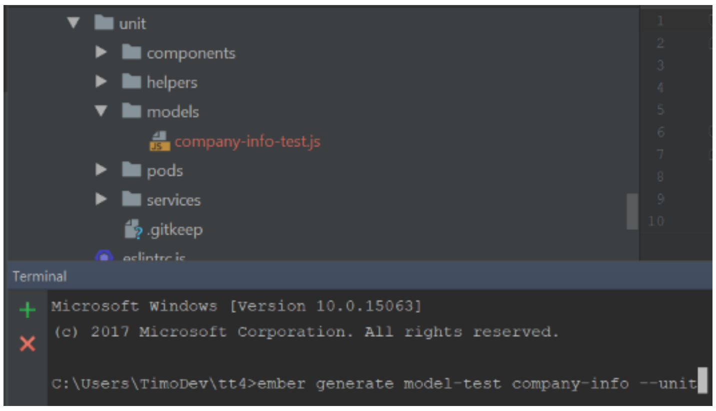
Last function compares that the user is not anymore in the login page after the click and also finds user's profile picture in the starting page. The reporting page looks similar to integration test with three passed assertions.

One important aspect of the application is to add user to the service and this kind of test could be done with acceptance testing. There was a lot of effort to automate this part of code but the test was eventually not finished. However there was a lot of research how it should be done. The adding of user happens by filling the needed information to the various fields and then saving it but the problem was that the application fetches the user information from user form and the testing script doesn’t render it on the screen automatically. The solution is to use Mirage which is a library to create sort of fake fields and then using them to render the form fields and testing it in the application environment. The acceptance testing can begin in the near future when the usability of both Mirage and also jQuery are learnt.

After that the adding of worktime, project and for example employer can begin and acceptance testing is further to be finished.

4.4 Finetuning with unit tests

Some unit tests were also added to the system and this was the last type of testing that was coded. In QUnit, unit testing can be done for controllers, components, models, services, routes and helpers. The goal was to test as many units as possible in this span of time.

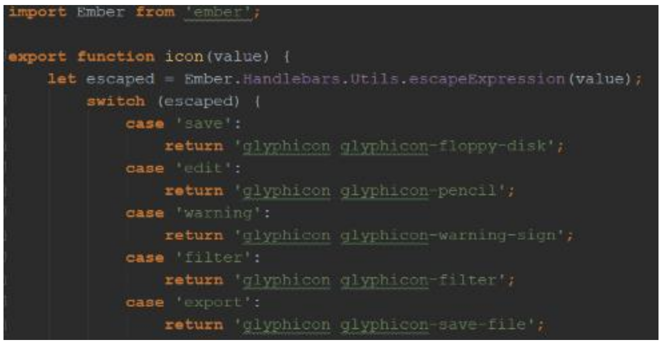


```
▼ unit
    ▶ components
    ▶ helpers
    ▼ models
        company-info-test.js
    ▶ pods
    ▶ services
      .gitkeep
  eslintrc.js
```

```
Terminal
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\TimoDev\tt4>ember generate model-test company-info --unit
```

Figure 22. Example of creating a unit test

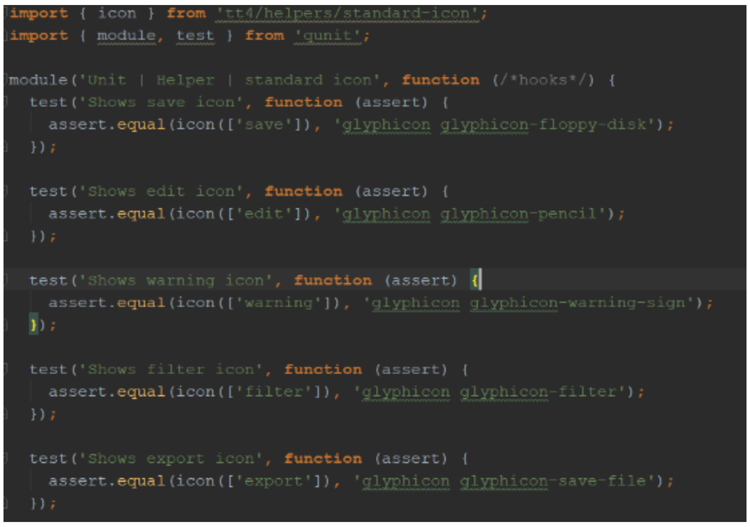
The basic principle to create unit tests can be seen in Figure 22. In this example, model test is created and it creates a test file in "/tests/unit/models". If the tester wants to test components, it is done simply by changing the word "model" to "component". The last part of the command (--unit) tells the system to create specifically unit test.



```
import Ember from 'ember';

export function icon(value) {
    let escaped = Ember.Handlebars.Utils.escapeExpression(value);
        switch (escaped) {
            case 'save':
                return 'glyphicon glyphicon-floppy-disk';
            case 'edit':
                return 'glyphicon glyphicon-pencil';
            case 'warning':
                return 'glyphicon glyphicon-warning-sign';
            case 'filter':
                return 'glyphicon glyphicon-filter';
            case 'export':
                return 'glyphicon glyphicon-save-file';
```

Figure 23. Code of icon helper

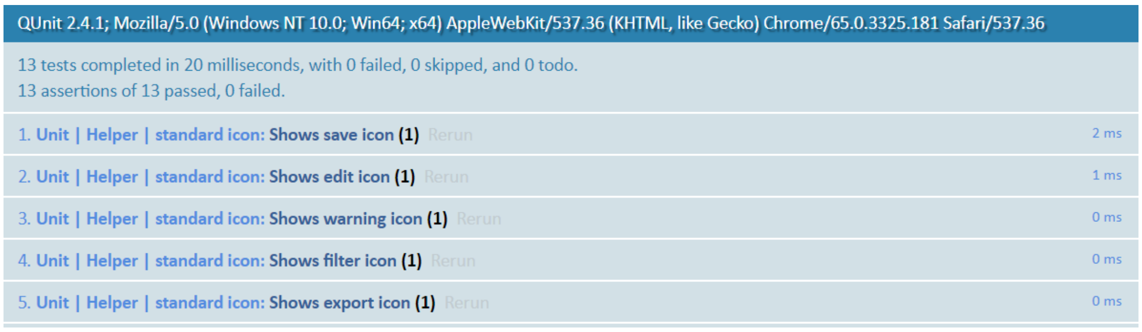
At first, a helper test was coded because it seemed like a simple approach to unit testing and good examples were found in the Internet. In Figure 23 is a code snippet of icon helper and the point was to test that the helper is working. Helpers are used to put certain codes outside of the other code to add functionality to them. This helper provides the icons to be used elsewhere in the code.



```
import { icon } from 'tt4/helpers/standard-icon';
import { module, test } from 'qunit';

module('Unit | Helper | standard icon', function (/*hooks*/) {
  test('Shows save icon', function (assert) {
    assert.equal(icon(['save']), 'glyphicon glyphicon-floppy-disk');
  });

  test('Shows edit icon', function (assert) {
    assert.equal(icon(['edit']), 'glyphicon glyphicon-pencil');
  });

  test('Shows warning icon', function (assert) {
    assert.equal(icon(['warning']), 'glyphicon glyphicon-warning-sign');
  });

  test('Shows filter icon', function (assert) {
    assert.equal(icon(['filter']), 'glyphicon glyphicon-filter');
  });

  test('Shows export icon', function (assert) {
    assert.equal(icon(['export']), 'glyphicon glyphicon-save-file');
  });
```

Figure 24. Code used to write icon helper test

The helper was tested in Figure 24. It includes same basic principles as with other testing. The function ”icon” is imported from the actual helper code where it was exported as seen in Figure 23. It is then compared that the right values of the function are equal to the right icon value. Glyphicons are components of Bootstrap and each component has its own glyphicon identifier.



```
QUnit 2.4.1; Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.181 Safari/537.36

13 tests completed in 20 milliseconds, with 0 failed, 0 skipped, and 0 todo.
13 assertions of 13 passed, 0 failed.

1. Unit | Helper | standard icon: Shows save icon (1) Rerun          2 ms
2. Unit | Helper | standard icon: Shows edit icon (1) Rerun          1 ms
3. Unit | Helper | standard icon: Shows warning icon (1) Rerun       0 ms
4. Unit | Helper | standard icon: Shows filter icon (1) Rerun        0 ms
5. Unit | Helper | standard icon: Shows export icon (1) Rerun        0 ms
```

Figure 25. Reporting page for the helper test

Reporting page shows the tests individually and reports that the tests are passed. There are 13 assertions done for 13 icons but the code and test code show only first five of them as well as the reporting page.

4.5 Summary of testing classifications

In the Table 2 is a short summary of these testing classifications and how challenging it was to do in this application using QUnit. It includes more own opinions and experiences that were confronted during the testing.

Table 2. Summary of QUnit testing

Classification	**Summary**
Acceptance test	took a lot of time and different aspects to figure out the way to do it, maybe not the best testing framework for acceptence testing.
Integration test	quite easy to write component and integrations tests, didn't take much time and was a good way to begin with.
Unit test	the most capable testing classification for Ember application, easy to write few tests but need to be continued in the future.

Table shows the experiences during testing and last section will include more analysis of the results and conclusions.

5 Conclusions

The results were good for the tests that could be done during these months. At the beginning, the goal was to automate many parts of the testing and maybe do these tests with another testing framework too. However, this goal was a bit utopistic because everything was new and there had to be much learning and research in relatively short time.

The goal was also to research the possibilities to automate testing in the company and it went well because now the whole testing automation is clearer compared to the beginning of the year.

The biggest problems were the acceptance tests because they required more knowledge about different coding techniques and they are broader than integration and unit tests. Acceptance tests are maybe the most important too and coding these tests will continue in the near future. The research was done well and they are on a good track to be coded.

Movenium is at the moment merging even more to Visma and one requirement from Visma is to automate testing. So the testing automation becomes more topical for Movenium during spring and summer. Now that the tools and commands are researched, it is easier to continue the process.

As an abstract of results, few test scripts were managed to be coded for each testing classification and the whole picture of testing automation is now clearer in the company.

References

1 Randell, Brian. 2013. The Origins of Computer Programming. ResearchGate. <https://www.researchgate.net/publication/3330487_The_Origins_of_Computer_Programming>. 21 May 2013. Accessed 15 Jan 2018.

2 Horne, Geoff. 2014. A (Very) Brief History of Test Automation. LinkedIn. <https://www.linkedin.com/pulse/20141007123253-16089094-a-very-brief-history-of-test-automation/>. 8 Oct 2014. Accessed 19 Jan 2018.

3 Software Quality Process. Soigne Technologies pvt.ltd-website. <http://www.tech-soigne.com/company/software-quality-process/>. Accessed 22 Jan 2018.

4 Hoffman, Doug.2003.A Course on Software Quality Methods. Software Quality Methods. <http://www.testingeducation.org/course_notes/hoffman_doug/test_automation/auto8.pdf>. Accessed 23 Jan 2018.

5 Marick, Brian. How to misuse code coverage?. <http://www.exampler.com/testing-com/writings/coverage.pdf>. Accessed 23 Jan 2018.

6 Tambey, Anand Avinash. End-to-end test automation – A behavior-driven and tool-agnostic approach. Infosys. <https://www.infosys.com/it-services/validation-solutions/white-papers/documents/end-test-automation.pdf>. Accessed 24 Jan

7 Kaner, Cem, Bach, James, Pettichord, Bret. 2002. Lessons learned in software testing: a context-driven approach. Accessed 12 Feb 2018.

8 Software QA Testing and Health IT -A guide for beginners. QA Talk Group. <http://qa-talk.blogspot.fi/2008/05/acceptance-testing.html>. 30 May 2008. Accessed 14 Mar 2018.

9 Powell-Morse, Andrew. 2016. V-Model: What Is It And How Do You Use It?. Airbrake. <https://airbrake.io/blog/sdlc/v-model>. 26 Dec 2016. Accessed 15 Mar 2018.

10 AdyKalra. 2017. Quality Assurance. Github. <https://github.com/AdyKalra/QA/blob/master/README.md>. 18 Apr 2017. Accessed 19 Feb 2018.

11 ISTQB. 2010. Standard glossary of terms used in Software Testing, Version 2.1. Accessed 20 Feb 2018.

12 Naik, Kshirasagar, Tripathy, Priyadarshi. 2008. Software Testing and Quality Assurance: Theory and Practice. <http://aksitha.com/Software%20Testing/STQA_book.pdf>. Accessed 6 Feb 2018.

13 EECS. 2009. Integration testing: Chapter 13. <https://www.eecs.yorku.ca/course_archive/2009-10/W/4313/slides/14-IntegrationTesting.pdf>. Accessed 26 Feb 2018.

14 Galeotti, J.P, Gorla, Alessandra. 2013. Integration, System and Regression Testing. University of Saarland. <https://www.st.cs.uni-saarland.de/edu/automatedtestingverification12/slides/18-IntegrationSystemRegressionTesting.pdf>. 31 Jan 2013. Accessed 26 Feb 2018.

15 Aebersold, Kirsten. What is a test framework?. Smartbear. <https://smartbear.com/learn/automated-testing/test-automation-frameworks/>. Accessed 20 Feb 2018.

16 Ghanakota, Gayatri. Testing frameworks. University of Colorado. <https://www.cs.colorado.edu/~kena/classes/5828/s12/presentation-materials/ghanakotagayatri.pdf>. Accessed 21 Feb 2018.

17 Zaidman, Vitali. 2017. An overview of JavaScript Testing in 2017. A Medium Corporation. <https://medium.com/powtoon-engineering/a-complete-guide-to-testing-javascript-in-2017-a217b4cd5a2a>. 19 Apr 2017. Accessed 8 Feb 2018.

18 Zhitnitsky, Alex. 2016. JUnit vs TestNG: Which Testing Framework Should You Choose?. OverOps. <https://blog.takipi.com/junit-vs-testng-which-testing-framework-should-you-choose/>. 7 Sep 2016. Accessed 9 Feb 2018.

19 Reitz, Kenneth. 2016. Testing your code. The Hitchhiker's Guide to Python. <http://docs.python-guide.org/en/latest/writing/tests/>. Accessed 9 Feb 2018.

20 Graham, Dorothy, Fewster, Mark. 1994. Software Test Automation. <http://read.pudn.com/downloads11/ebook/44105/Software%20Test%20Automation.pdf>. Accessed 14 Feb 2018.

21 Kelly, Michael. 2003. Choosing a test automation framework: IBM. <https://www.ibm.com/developerworks/rational/library/591-pdf.pdf>. 20 Nov 2003. Accessed 22 Feb 2018.

22 Rajkumar. 2018. Types of Test automation Frameworks. Software Testing Material. <https://www.softwaretestingmaterial.com/types-test-automation-frameworks>. 25 Jan 2018. Accessed 23 Feb 2018.

23 Gilmore, Tom. 2015. The ”Killer Dozen” Automation Implementation Killers: Choosing the Wrong Tool Set. Agile Testing Framework. <https://www.agiletestingframework.com/the-killer-dozen-automation-implementation-killers-choosing-the-wrong-tool-set/>. 19 Aug 2015. Accessed 23 Feb 2018.

24 McPeak, Alex. 2018. The Criteria to consider for choosing JavaScript testing frameworks. DZone. <https://dzone.com/articles/the-criteria-to-consider-for-choosing-javascript-testing-frameworks>. 19 Jan 2018. Accessed 23 Feb 2018.

25 Glow Touch Technologies. 2016. How to choose the right test automation framework: <https://www.glowtouch.com/blog/testing/how-to-choose-the-right-test-automation-framework/>. 1 Mar 2016. Accessed 24 Feb 2018.

26 Heusser, Matthew. 6 common test automation mistakes and how to avoid them. TechBeacon. <https://techbeacon.com/software-test-automation-6-common-mistakes-how-avoid>. Accessed 25 Feb 2018.

27 Sarcevic, Igor. 2015. Getting started with Node.js and Mocha. Semaphore. <https://semaphoreci.com/community/tutorials/getting-started-with-node-js-and-mocha>. 16 Mar 2015. Accessed 1 Mar 2018.

28 Burgess, Andrew. 2011. Testing your JavaScript with Jasmine. Envato Tuts+. <https://code.tutsplus.com/tutorials/testing-your-javascript-with-jasmine--net-21229>. 4 Aug 2011. Accessed 2 Mar 2018.

29 Jugglinmike. 2017. An interface for Mocha that implements QUnit's API. Github: <https://github.com/jugglinmike/mocha-qunit-ui>. 6 Sep 2017. Accessed 4 Mar 2018.

30 Bakker, Kees C. Convert QUnit test to Mocha / Chai. KeesTalksTech. <https://keestalkstech.com/2016/08/convert-qunit-test-to-mocha-chai/>. Accessed 4 Mar 2018.

31 Molsson's blog. 2017. JavaScript unit testing tools. <http://mo.github.io/2017/06/05/javascript-unit-testing.html>. 5 Jun 2017. Accessed 5 Mar 2018.

32 QUnit website. <https://qunitjs.com/>. Accessed 7 Mar 2018.

33 JetBrains website. <https://www.jetbrains.com/phpstorm/>. Accessed 7 Mar 2018.

34 SourceTree website. <https://sourcetreeapp.com>. Accessed 8 Mar 2018.

35 Github homepage. <https://github.com>. Accessed 8 Mar 2018.

36 Microsoft docs. PowerShell. <https://docs.microsoft.com/en-us/powershell/>. Accessed 8 Mar.

37 Wikipedia. <https://en.wikipedia.org/wiki/Npm_(software)>. Accessed 9 Mar 2018.