Bachelor’s thesis

Information Technology

Embedded software

2017

Nelli Tchernikova

BROWSER-BASED AUTOMATION TESTING USING SELENIUM RC FOR MONETIZE COMMERCIAL SAAS PRODUCT

TURKU AMK

TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR’S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Degree programme | Information technology

2017 | Total number of pages 48

Nelli Tchernikova

BROWSER-BASED AUTOMATION TESTING USING SELENIUM RC FOR MONETIZE COMMERCIAL SAAS PRODUCT

"HyperIn Inc is Finnish SW development company with a strong portfolio of products to manage shopping center assets such as sales data, advertisement places , tenants etc. One of the main products offered by HyperIn is Monetize. It is Software as a Service web based product offering to shopping center management the way to manage all shopping center advertisement places and selling them to end users directly or via web shop interface.

Monetize development team identified that Monetize needs to be fully covered with automatic UI tests using Selenium RC language. Thesis work goal was to implement all required automatic UI Selenium tests.

Thesis text explains agile Scrum SW development process used to develop Monetize and why automatic testing is so important for the process. It gives detailed technical overview of how to write Selenium tests for Monetize including Selenium scripts, Java helper methods, test data management and efficient ways to identify HTML components on web page in the tests. Additionally it is explained how to write localization agnostic test cases.

Thesis goal was achieved and Monetize product was100% covered with automatic Selenium UI tests. There was written about 7400 source code lines in Selenium, Java and Groovy language to implement the tests. Developed Selenium tests were taken by Monetize development team into Continuous Integration process to reduce the cost of manual regression testing.“

KEYWORDS:

Automation Testing, QA, Selenium, Selenese, Java

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Degree programme | Information technology

2017 | Total number of pages 48

Nelli Tchernikova

SELENIUM AUTOMAATTISET TESTIT KAUPALLISELLE SAAS TUOTTEELLE ”MONETIZE”

HyperIn Inc on suomalainen ohjelmistokehitykseen keskittynyt yritys, jolla on tarjolla laaja portfolio tuotteita erilaisten kauppakeskusten varojen kuten myyntitietojen, mainospaikkojen ja vuokralaisten hallintaan. Yksi HyperInin tarjoamista tuotteista on Monetize. Se on Software as a Service - verkkopohjainen tuote, joka tarjoaa kauppakeskusten johdolle mahdollisuuden hallita kaikkia kauppakeskuksen mainospaikkoja ja myydä niitä loppukäyttäjille suoraan tai verkkokaupan kautta.

Opinnäytetyö kertoo Scrum ohjelmistokehitysprosessista, joka käytettiin Monetizen kehittämiseen sekä selittää, miksi automaattinen testaus on erittäin tärkeä prosessin kannalta. Se antaa yksityiskohtaisen teknisen yleiskatsauksen siihen, miten kirjoitetaan Selenium-testit Monetizelle mukaan lukien Selenium-komentosarjat, Java-helper menetelmät, testitietojen hallinta ja tehokkaat tavat HTML-komponenttien tunnistamiseen testeissä web-sivuilla. Lisäksi opinnäytetyössä selitetään, miten tehdään lokalisoinnin testisuunnitelmia.

Opinnäytetyön tavoite oli saavutettu ja Monetize oli testattu 100 % automaattisilla Selenium -käyttöliittymän testeillä. Testien toteutusta varten kirjoitettiin noin 7 400 koodin rivija Selenium, Java- ja Groovy-kielellä. Monetizen tuotekehitysryhmä otti Selenium-testit käyttöön osaksi jatkuvaa integrointiprosessia manuaalisen regressiotestauksen kustannusten vähentämiseksi."

KEYWORDS:

Automaation Testing, QA, Selenium, Selenese, Java

CONTENTS

LIST OF ABBREVIATIONS (OR) SYMBOLS

SLOC	Source lines of code
YAML	Stands for “yet another markup language”

1 INTRODUCTION

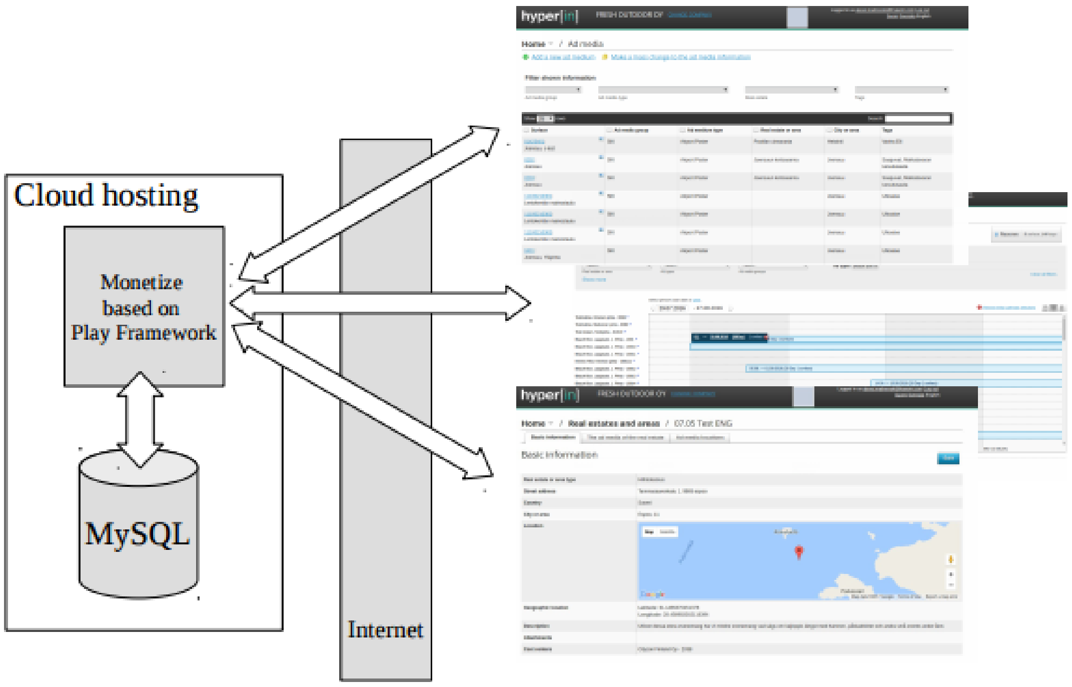
HyperIn Inc is a Finnish company with headquarters in Helsinki with mission to provide leading solution in management and monetization for shopping centers. Company products are grouped into 3 categories:

- **Manage**
  - managing shopping center tenants and sales reports
- **Connect**
  - using shopping centers tenants information in mobile apps, interactive mall directory screens and in shopping centers websites
- **Monetize**
  - managing and selling shopping centers advertisement spaces like banners, time in ad tvs etc.

Every product is a complex SW solution based on stack of SW components that requires testing before every product version goes to the market. This thesis work is focusing on Monetize product.

1.1 Monetize introduction

Monetize is a software as a service (SaaS) web based application based on Play Framework and MySQL.



Picture 1. Monetize software as a service

“MONETIZE platform gives you a variety of great tools that you can easily use for selling all the advertising and promotional spaces of your mall. Whether on your digital screens, video walls, specialty leasing places, promotional stands, voice commercials or touch-screens, you will generate more ad revenue. You can also run the same ads on your web page or consumer mobile devices.

Hyper[in] MONETIZE is a web service, which offers you or any service provider a great opportunity to efficiently sell advertising on your shopping mall and website with online reservation and payment functionality.” (Monetize 2017)

Monetize contains sophisticated web based UI for end users that allows managing ad spaces, ad space reservations , customers and many other relevant features.

As product grew it started to be very risky to extend it without good automatic UI test coverage to make sure that changes in one part of the product do not break something in other part of it.

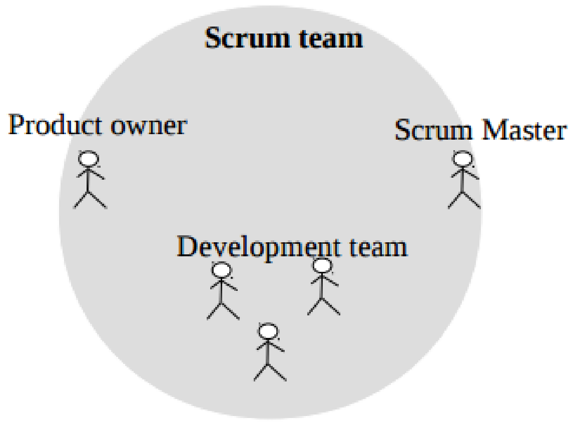
1.2 Thesis goals

Monetize UI was originally covered ~15% with automatic Selenium UI tests. The thesis goal was to achieve 100% Selenium UI automatic tests coverage of Monetize web application developed by HyperIn Inc. The work to finish the task was estimated by Monetize developers to be 3 to 6 months full time work of test automation engineer.

Thesis goal was reached and currently Monetize is 100% covered with automatic Selenium tests. This text contains details on how this was achieved.

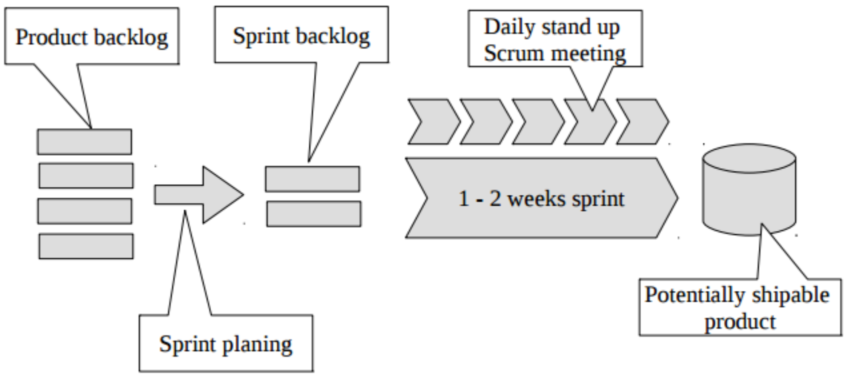
2 MONETIZE SW DEVELOPMENT PROCESS

HyperIn delivers new product features often and with good quality. Usually there is a new product version released to the customers every month. Products becoming more complex with time and it gets time consuming and expensive to test and ensure quality manually. To keep rapid delivery with expected quality HyperIn adapted agile SW development practices and effective SW testing process that includes sophisticated automatic testing steps. Monetize team is a typical Agile Scrum team as demonstrated on the picture below:



Picture 2. Agile Scrum team

Below is diagram representing Monetize development process that is similar to one explained in Agile project management with Scrum (Schwaber, 2004) or Agile software development with Scrum (Beedle & Schwaber, 2001).



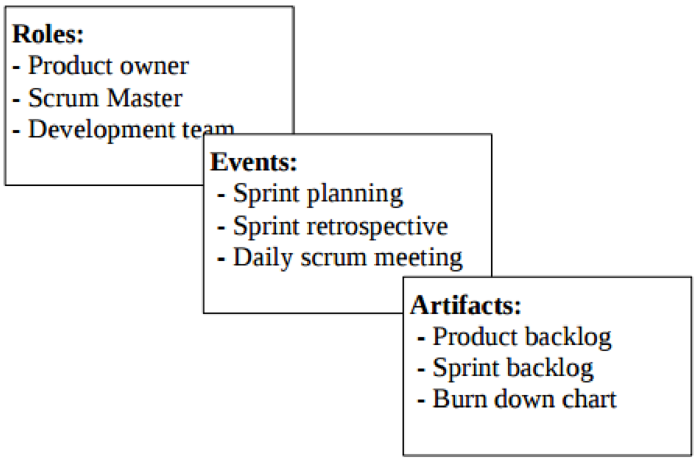
Picture 3. Monetize development process

2.1 Scrum process overview

- Scrum team agrees on Monetize releases:
  - Product owner defines requirements as Stories into “Product backlog”
  - Several Stories selected by product owner for next Monetize release
  - Release date is set based on high level estimations from the developers
- Monetize developed in a series of 1-2 weeks sprints
- At the beginning of the sprint team meets in “Sprint planning meeting” where it:
  - discusses results of the previous sprint. What went well, what not and check if release date is still feasible.
  - selects Stories it can implement in the next sprint and commit to implement them during the sprint. Stories selected for the sprint called “Sprint backlog”
- Every day team has 5 minutes stand-up meeting to check the sprint status, work impediments and check burndown chart
- Monetize stories are designed, coded and tested during the sprint
- The team is self-organized and no management involvement required during the sprint
- In case team realizes it cannot deliver features in time Product manager can decide either to postpone the release or to drop some features from the release.

2.2 Scrum framework components

The main components of Scrum framework in Monetize / HyperIn are:



Picture 4. Components of Scrum framework in Monetize

- define user Stories and what stories go to which Monetize released

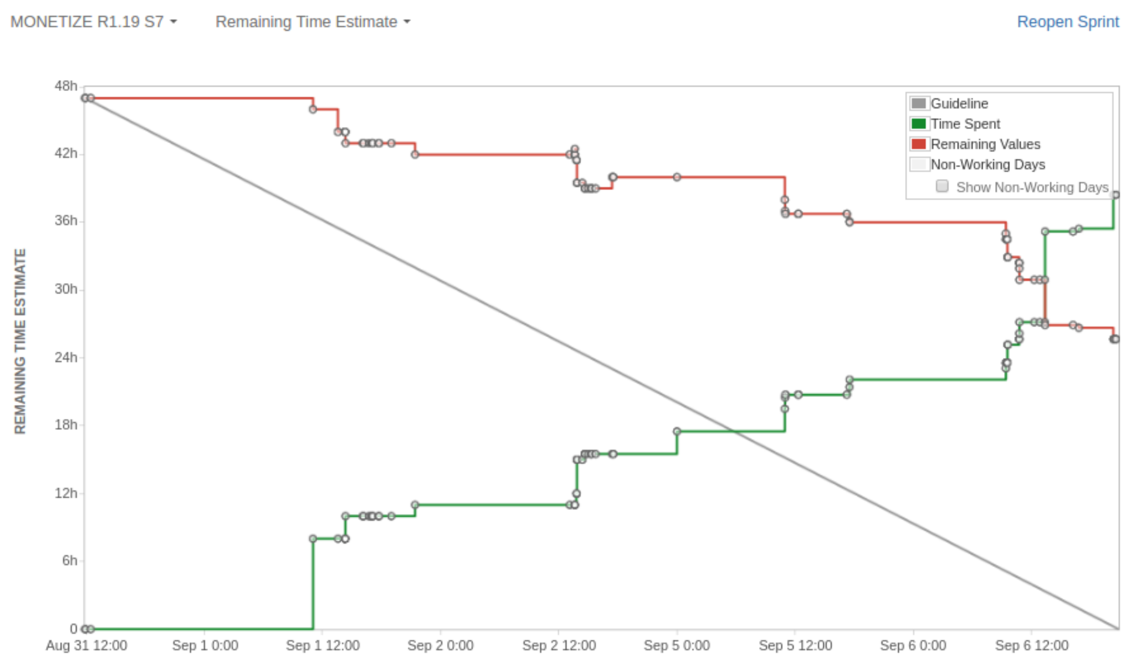
        - review, accept or reject implemented stories
    - Scrum master
        - Makes sure scrum process is followed
        - Collects and resolves impediments for the team
    - Scrum team
        - Team is self-organizing and no management is required during the sprinting
        - Consists of SW developers, SW testers and UI designers

- **Meetings**
    - Sprint planning / review
        - Review previous sprint results
        - Define next sprint goal - select Stories from Product Backlog that can be implemented and tested during sprint taking into account sprint length and team capacity
        - Define detailed tasks to implement stories and estimate each task in hours. This will allow to follow up on burn-down chart
        - Check if release date is still feasible and if not Product manager need to decide either to postpone release date or to reduce release scope.
    - Daily scrum meeting
        - 5 minutes stand up meetings with scrum master and team members
        - Check burndown chart and impediments
    - Release demo
        - When release stories are implemented those are demoed to the whole company

- **Tools**
    - Product backlog
        - List of user stories for the product.
        - Stories prioritized by product owner
    - Sprint backlog

    - List of user stories selected for implementation in the sprinting
    - Story must be small enough to fit the sprint including implementation and testing
- Burn-down chart
    - Burn-down chart represent comparison graph between worked planned at the beginning of the sprint against remaining work.
    - If remaining work is above the planned work on the graph it means scrum is late and vice versa.

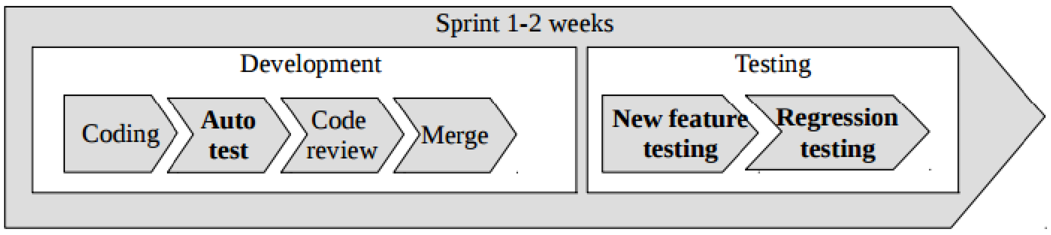
Example of typical burndown chart in Monetize presented below. In this particular case team didn't manage to reach sprint goals because remaining work presented as a red line is not zero.



Picture 5. Burndown chart in Monetize

3 QUALITY ASSURANCE IN AGILE MONETIZE TEAM

Quality Assurance of features developed during the sprint contains following stages:



Picture 6. Sprint stages

  - For every piece of newly developed / changed code there is a set of set of automatic tests that must pass before code is merged to repository
  - Automatic tests include – unit, functional and Selenium tests.
- Testing phase
  - Tests executed manually by SW testers by opening application pages and trying out functionality.
  - New feature testing – testing of newly developed functionality
  - Regression testing – testing of old functionality that might be impacted by new code.

3.1 Manual testing

Manual testing contains following events:

- Writing test plan for the user Story. It usually includes chapters for:
  - Testing of normal feature functionality
  - Testing border cases like max / min field values, mandatory fields etc
  - Testing security like trying to insert JavaScript code into input fields
  - Testing localization to all languages supported by Monetize

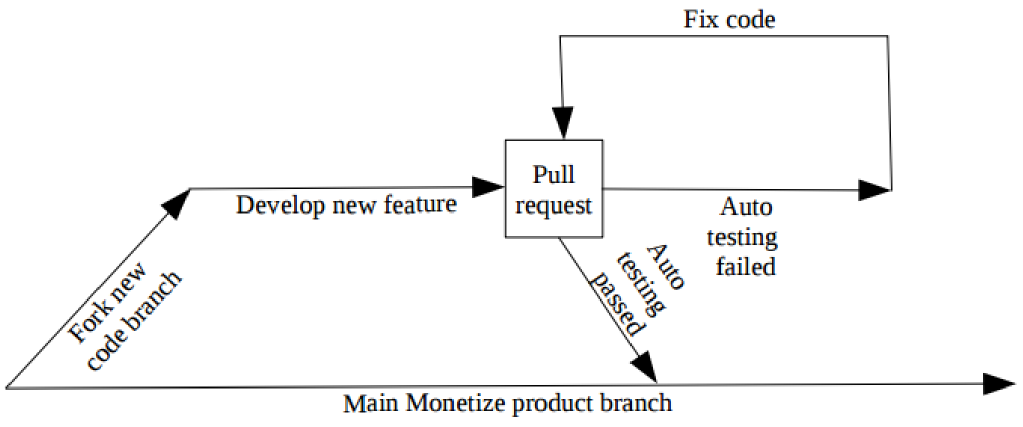
    - Testing for different browser versions from IE9 to IE12 and latest Chrome and Firefox
- Testing according to plan after user Story is implemented
    - Executing all test steps written in test plan
    - Reporting found bugs in case functionality does not follow the test plan

- Verification of the fixed bugs
    - SW developers fix found bugs immediately and testers must verify them

- Regression testing
    - While developing new features source code changes can break already functional old features.
    - Sometimes it is possible to predict what parts of product could be broken by new code and test them. Sometimes it is not possible.
    - Regression testing aims to find what existing functionality was broken by new code. It can be time consuming because Monetize is a big product.
    - Automated testing target is to reduce time spent on manual regression testing.

3.2 Automated testing

Monetize team uses git as a version control system. This is a simplified git work-flow demonstrated from automated testing point of view:



Simplified development process with automated testing steps:

Picture 7. A simplified git work-flow.

- Fork new source code branch from develop branch
- Do coding of a new functionality
- Create pull request in GitHub with a request to merge new code to develop branching
- Jenkins Continuous Integration sever triggers Pull request automated testing
- If automated tests passed Pull Request can be merged into develop branching
- If automated tests failed Pull Request is automatically rejected and developer must inspect and fix the code to make sure all automatic tests pass.
- Passing automated tests reduce regression testing that need to be done during manual testing phase.

There are several layers of automated testing implemented in Monetize and each stage helps to reduce regression testing.

3.2.1 Unit testing

During unit testing individual atomic parts of source code are tested independently of each other. It is mostly utility methods that for example process one set of data into another set of data.

This is an example of a typical unit test

```
@Test
public void getLast_nonEmptyListWithSeveralObject_returnsLastObject() {
    List<String> list = new ArrayList<>();
    list.add("First object");
    list.add("Second object");
    list.add("Third object");
    assertTrue(CollectionUtils.getLast(list).equals("Third object"));
}
```

3.2.2 Functional testing

Functional testing covers broad interconnected functionality areas to make sure those conforms to the original requirements. Play framework provides utilities that simplify writing functional tests for the products based on it. Selenium UI testing is one of the functional testing methods provided by framework.

3.2.3 Selenium UI testing

Play framework bundled with Selenium language and Selenium tests executor to create and execute automatic UI tests. Selenium is a language that allows emulating user behavior in web browser. It is possible to write script like the one below and Selenium engine will execute it step by step:

```
#{selenium "Public medium - hiding visibility to public"}
   open('/api/v1/media/${mediumId}')
   verifyTextPresent('"id":10')
   verifyTextPresent('"code":"C123"')
   open('/realty/thirdpartytomedia')
   assertLocation('*/thirdpartytomedia')
   waitForElementPresent('css=tr.testautomation_id$12')
#{/selenium}
```

Typical Selenium test contains UI control commands like “open some URL”, “click some button” and verification commands like “verify that something presented on the screen”.

Selenium tests are executed in browser window either manually or automatically.

At the beginning of this thesis work Monetize had ~15% of its functionality covered with automatic Selenium tests. There was an urgent need to cover 100% to reduce manual regression testing that is slow and costly. During this thesis work Monetize UI was 100% covered with automatic Selenium tests. Following chapters cover Selenium testing in details.

4 SELENIUM UI TESTING FRAMEWORK

There are several good UI testing frameworks on the market but Selenium is a “de-facto” most popular framework for UI testing. Probably, that's why Play Framework bundled Selenium script language and scripts executor into the platform by default.

“Selenium is a portable software testing framework for web applications. Selenium provides a record/playback tool for authoring tests without learning a test scripting language (Selenium IDE). It also provides a test domain-specific language (Selenese) to write tests in a number of popular programming languages, including C#, Groovy, Java, Perl, PHP, Python, Ruby and Scala. The tests can then be run against most modern web browsers. Selenium deploys on Windows, Linux, and OS X platforms. It is open-source software, released under the Apache 2.0 license, and can be downloaded and used without charge.” (Selenium (software) 2017)

4.1 Selenium RC scripting language

Originally Selenium was designed as a scripting linear language without loops or conditional if/then statements. The format to write scripts was quite unusual and typically written as HTML file. The script itself was formed as HTML table.

For example imagine we want to test google.com main page for search functionality.

1. Open google.com address
2. Type “What is Selenium?” to search for the answer
3. Click “Search button”
4. Wait for results

The script in original Selenium format as HTML table look very verbose and not user friendly as below. Note that elements ids cannot match actual google page and presented here only as example:

```
<tr>
```

```
    <td>open</td>
    <td>www.google.com</td>
    <td></td>
</tr>
<tr>
    <td>waitForElementPresent</td>
    <td>id=search-input</td>
    <td></td>
</tr>
<tr>
    <td>type</td>
    <td>id=search-input</td>
    <td>What is the best web platform in the world</td>
</tr>
<tr>
    <td>clickAndWait</td>
    <td>id=search-button</td>
    <td></td>
</tr>
```

The idea was that Selenium tests could run in any browsers without any need for browser simulation ensuring that you test actual web application as it is presented in the browser. As you can deduct from example test above each Selenium command contains command name and 2 parameters

1. **command name** (mandatory) for example “type” to type string into input
2. **target** (mandatory) – for example CSS selector css=#search-field or web address
3. **value –** value that must be applied for target

In other words

```
<tr>
    <td>type</td> ← command
    <td>id=search-input</td> ← target
    <td>What is the best web platform in the world</td> ← value
</tr>
```

The commands in Selenium also called Selenese commands.

4.2 Selenium RC in Play Framework

Writing Selenium tests as HTML tables was very laborious and not user friendly. From the other hands that was and is a de-facto most popular UI testing scripting language in the world. Play Framework development team decided to use Selenium for UI testing but improve the way scripts are written a lot and wrote good documentation about it here (Play selenium, 2017).

```
*{ Insert your comments here }*

#{selenium 'Test google search page'}
    open('www.google.com')
    waitForElementPresent('id=search-input')
    type('id=search-input', 'What is the best web platform in the world')
    clickAndWait('id=search-button')
#{/selenium}
```

Let's rewrite example from the previous section in Play Framework Selenium scripting language:

This code looks as real test code written in a proper conventional scripting language. It is easy to read and understand.

4.2.1 Loading testing data between tests

However, any complex tests for real web application testing usually requires some sort of database to provide data for web application. Play Framework provides mechanism to load test data before every test case. For example below is a sample Monetize Selenium test that starts with user logging into Monetize:

```
#{fixture delete:'all', load:'user-data.yml' /}

#{selenium "Login into Monetize"}
    open('/login')
    type('name=username', 'username@hyperin.com')
    type('name=password', 'password')
    clickAndWait('signin')
```

```
   open('/admin')
#{/selenium}
```

For loading test data we use #{fixture /} tag. It contains 2 directives inside:

- **delete: 'all'** – instructs to delete all previously loaded models into database. It means database schema and tables structures will remain intact but all table content will be cleared
- **load: 'user-data.yml'** – loads sample data set into tables from external file in YAML format

4.2.2 Test data YAML format overview

YAML stands for “yet another markup language” and it is a human readable data representation format (Yalm, 2017). For example in the example above we load user-data.yml file that may looks as below example:

```
Company(c1):
   name: Gweb2
   languages: [fi]
   defaultLanguage: fi
   subdomain: hyperin

User(u1):
   email: john.doe@hyperin.com
   password: klgjlskg
   companies: [c1]

User(u2):
   email: mikki.mouse@hyperin.com
   password: fdgsdfg345234rfwfdsa
   companies: [c1]
```

This yml file represents 2 tables in the database:

- **Company** – has 1 record
- **User** – has 2 records

If we look at one of the YAML pbjects in detail then:

On loading test runner identify which YAML object must go to which database table and what relationship these YAML objects has between each other (like user u2 belongs to company c1) and recreates table relationships based on this data.

User ← **table name** (u1 ← **unique id in yaml file**):
   email: john.doe@hyperin.com ← **field value**
   password: klgjlskg ← **field value**
   companies: [c1] ← **field value**

Play Framework provides in memory lightweight H2 database accessible by Play application while running Selenium tests. H2 was selected by Play Framework developers because it is much faster than MySQL to initialize and load data into it. Fast H2 database allows to achieve 2 goals:

- Reset test data between each Selenium test and make these tests independent of each other
- Reset data very fast that allows to run Selenium tests within reasonable time

As Selenium script runner runs hundreds of tests and each test needs its own set of data the batch test execution looks as below:

Reset data

#{fixture delete:'all', load:'data-test1.yml' /}

Run test 1

↓

Reset data

#{fixture delete:'all', load:'data-test2.yml' /}

Run test 2

↓

…

↓

Reset data

#{fixture delete:'all', load:'data-testn.yml' /}

Run test n

The data resetting with H2 database and YML files is very fast that allows to create independent of each other Selenium tests. If some test fails it definitely failed not due to data changed in previous test that helps to isolate and debug the problem.

4.3 Most common Selenese commands

Selenium commands are also called Selenese commands. There are many predefined Selenese commands supplied by Selenium and also Selenium can be extended with custom commands using JavaScript extension file. The commands can be grouped into 3 categories:

4.3.1 Action

Actions are commands that controls the flow and changes state of the web application under test. They represent end user behavior in the web browser. For example you action can type a text into input field or click a button or open an URL. If action fails for example because it cannot find an element test script also fails. Many actions can be extended with suffix AndWait for example you could use command click or clickAndWait. The difference is that latter will expect server to send a new web page to client and will wait for a new page to load before script continues.

The most used actions in Monetize Selenium test scripts are:

- **open(url)**
  - Opens page by URL address
  - Example: open('www.google.com')
- **type(locator, value)**
  - Types string into element. Usually element is a normal input or text field.
  - Example: type('id=search-field', 'who is mister John')
- **click(locator)**
  - Clicks on element on the page
  - Example: click('id=save')
- **clickAndWait(locator)**
  - Same as click() but also wait for next page to load after click
- **check(locator)**
  - Activates checkbox element. Use uncheck() to deactivate it
  - Example: check('id=terms-accepted')
- **select(locator, option value)**
  - Selects option from drop down
  - For example: select()
- **pause(milliseconds)**
  - Pause test execution for given amount of milliseconds. Often used to let web application under test to finish background processing before going to next test step.

4.3.2 Assertions

Assertions check the state of the web application and either fail the test or let it continue. There are 2 types of assertions starting with prefixes:

- **Verify...** – it verify condition and if it fails it log the failure but let the test to continue till the end. If any of “verify” statement failed it will fail the complete test at the end. These commands are useful to collect small non critical errors in the web applications all at once instead of terminating the whole test on minor mistake.

- **Assert…** - if condition fails it terminates test immediately and mark it as failed because failure is so critical that there is no reason to continue execution.
- **WaitFor…** - similar to Assert... but waits for 30 seconds periodically checking if condition have turned to true. If it doesn't turn true it fails the test.

The most used actions in Monetize Selenium test scripts are:

- **verifyTextPresent(text)**
  - Searches the whole page for the given text and passes if text is found.
- **verifyTextNotPresent(text)**
  - Searches the whole page for the given text and passes if text is not found.
- **verifyValue(locator, value)**
  - Searches web page for element and verify if it contains given value. Very useful for checking summary pages in Monetize.
- **assertElementPresent(locator)**
  - Passes if element found on the page. Fails whole test if element cannot be found.
- **assertElementNotPresent(locator)**
  - Passes if element not found on the page. Fails whole test if element can be found.
- **waitForElementPresent(locator)**
  - Waits maximum 30 seconds for given element to appear on web page. Passes immediately when element appeared. Fails whole test if element does not appear after 30 seconds.
- **waitForElementNotPresent(locator)**
  - Waits maximum 30 seconds for given element to disappear from web page. Passes immediately when element disappeared. Fails whole test if element is still visible after 30 seconds.

4.3.3 Accessors

Accessors check the state of the application elements and store results into variables or read them. Accessors commands were not found very useful in Monetize Selenium tests. The only useful command was the command to deactivate system browser dialogs that were otherwise not accessible by standard element selectors:

- **storeConfirmation('variable')**
  - Retrieves the message of a browse confirmation dialog generated during the previous action. After this command Monetize tests use chooseOkOnNextConfirmation() to close confirmation dialog.

Monetize Selenium tests actively use 15 - 20 Selenese commands listed above to fulfill the testing needs. However, there are many more commands Selenese available that can be used in writing test scripts. Unfortunately , official list and documentation of Selenese commands on Selenium HQ website http://www.seleniumhq.org/ is broken. However , there are plenty 3rd party sites where these commands and documentation with code examples can be found. During thesis work following web page was good source of Selenium commands examples "Selenium tutorial for beginner " (Selenium tutorial for beginners 2017)

4.4 HTML element locators

Selenium uses so called locators to locate elements on the web page. There are 4 locators types used in Monetized Selenium tests.

4.4.1 Id

Simplest locator type that uses id HTML attribute to locate elements. For example consider following HTML snippet:

```
<html>
 <body>
  <form id="login">
   <input id="username"/>
   <input id="password" name=”password.field” />
   <input id="submit"/>
  </form>
  </body>
</html>
```

Simple Selenium test in Monetize utilizing Id locator that types user name 'Nelly' into user name field will look as:

```
#{fixture delete:'all', load:'user-data.yml' /}
#{selenium "Typing user name"}
                    type('id=username', 'Nelly')
#{/selenium}
```

Syntax:

- id=value

Pros:

- Each id should be unique , at least in a well-designed page. It means locator can match only single element on the page.

Cons:

- Often elements in Monetize do not have ids and it is not possible to identify them with id locator

4.4.2 Name

Similar to id locator but uses name attribute to locate HTML element. If we consider same HTML snippet as in id locator section then simple Selenium test in Monetize utilizing Name locator that types password into password field will look as:

```
#{fixture delete:'all', load:'user-data.yml' /}
#{selenium "Typing user password"}
                    type('name=password.field', 'Nelly')
#{/selenium}
```

4.4.3 CSS

If Id or Name is not an option to select element because those are missing or not unique in web page you should prefer using CSS locators as a best alternative.

Syntax:

- css=selector

Pros:

- Easy to write and read
- Flexible
- It is claimed CSS locators are faster to execute then XPath locators

Cons:

- Sometimes, though quite seldom, CSS selectors are not flexible enough to select element. Then XPath selectors usually helps in such situations.

Typical selectors used in Monetize are:

- **css=tag**
  - Selects element by HTML tag.
  - For example to select <input></input> use locator css=input
- **css=tag[attribute=value]**
  - Selects element by HTML tag and attribute value.
  - For example to select <a href="google.com"></a> use locator css=a[href="google.com"]
- **css=tag.class**
  - Selects element by HTML tag and CSS class.
  - For example to select <div class="table-of-content"></div> use css=div.table-of-content
- **css=tag#id**
  - Selects element by HTML tag and id

  - For example to select <input id=”password”></input> use locator css=input#password
- **css=.class**
  - Selects element by class only.
  - For example to select <div class=”table-of-content”></div> use css=.table-of-content
- **css=#id**
  - Selects element by id only.
  - For example to select <input id=”password”></input> use locator css=#password
- **css=tag:contains(“text”)**
  - Selects element if it contains text. It is very convenient to select table cell with certain text using this selector.
  - For example to select <td>Some text</td> you could use selector css=td:contains(“Some text”)
- **css=parentSelector childSelector childOfChildSelector etc.**
  - Selects element by first selecting parent element by parentSelector and searching for childSelector inside. It is very useful for selecting elements in HTML tables or in complex selectors.
  - For example in this table-of

```
<html>
   <table>
                    <tr><td>row1</td><td>value</td></tr>
                    <tr><td>row2</td><td>value</td></tr>
   </table>
</html>
```

to select the bold <td> element you could use selector css=tr:contains(“row1”) td:contains(“value”)

In Monetize most of locators are CSS locators because those are fast , flexible , easy to write and read. The good CSS one page instruction that was used during Monetize Selenium scripts creation can be found in the article “XPath, CSS, DOM and Selenium: The Rosetta Stone” (Sorens, 2011).

4.4.4 Xpath

XPath is a syntax to navigate XML documents. Since HTML is practically a subset of XML – XPath syntax is supported by Selenese commands to locate HTML elements. XPath locators are generally more complex to write and difficult to read though often they provide the way to select elements that otherwise not possible to select with other selector types like CSS locator.

Syntax:

- //tag[expression]/tag[expression] etc.

Pros:

- If no other locator types works it is usually possible to construct XPath locator to select elements

Cons:

- It is difficult to construct XPath locators
- It is not user friendly and it is difficult to understand it

Typical examples XPath locators used in Monetize:

- **//span[contains(text(),"text1”)]//preceding-sibling::input**
  - Locator selects input element located before span element
- **//tr[@class="element css class"]//th[contains(text(),"table header”)]//following-sibling::td[contains(text(),"cell text”)]'**
  - Locator selects cell in table that follows after another cell and both located in table row with header that contain certain text.

As you can see those are quite complex selectors and it is quite difficult to construct them right. However, sometimes XPath selectors allows selecting elements that are otherwise not possible to select so it is important to know how to use them.

The good XPath one page instruction that was used during Monetize Selenium scripts creation can be found in the article “XPath, CSS, DOM and Selenium: The Rosetta Stone” (Sorens, 2011).

4.5 Selenium Java helper classes in Monetize

Play framework developers didn't stop on making Selenium scripts easy to write and adding good support of test data loading. They added support for calling Java class methods and Groovy script snippets to pre-process Selenium test script before sending it to test runner for test execution.

The test execution sequence in Play Framework looks as below:

test.html

(original test script with Groovy or Java source code + Play Selenium commands)

↓

Java / Groovy pre-processor

↓

test.html

(with Play Selenium commands only)

↓

Selenium preprocessor

↓

test.html

(with Selenium RC script in html tables fiormat)

↓

Script runner

Play framework does not dictate on how test developer use this possibility to pre-process test scripts with Java. Monetize SW developers took advantage of it to make Selenium tests written for monetize to be more readable and reusable and extended tests writing process with a small custom Java library called SeleniumScriptBuilder. It is easier to demonstrate how it helps to simplify Selenium script.

Let consider following real sample test script that logs in into Monetize, check that login was successful and immediately logs out.

```
#{fixture delete:'all', load:'user-data.yml' /}

#{selenium "Test login / logout"}
   // log in
   deleteAllVisibleCookies()
   createCookie(_subdomain=hyperin)
   waitForElementPresent(css=a:contains("Sign in"))
   open('/login')
   type('name=username', 'username@hyperin.com')
   type('name=password', 'password')
   clickAndWait('signin')

   // verifying successful login
   verifyElementNotPresent('css=.message.error')
   waitForElementPresent('css=div#login-status a[href$=logout]')

   // log out
   waitForElementPresent('css=a[href="/logout"]')
   click('css=a[href="/logout"]')
#{/selenium}
```

The test is written using Selenese commands in Play framework format. It is a real test and all Monetize tests can be written in this format but there is a number of problems with it:

- **Limited re-usability**
  - Every Monetize Selenium tests requires login into the Monetize service. If we use plain Selenese commands format as above we need to repeat first 7 Selenese

commands every time we want to login to Monetize. Monetize Selenium tests contains 34 places where test logs in into Monetize. It means these 7 commands needs to be repeated 34 times that is 238 lines of repeated code that generally considered as a bad design in SW engineering.

- **No configurability**
  - If test need to login with different user name / password then these values must be hard-coded into Selenese commands themselves making it very hard to change these values on the fly if needed.

- **Hard to maintain**
  - This directly comes from the first point of limited re-usability. In case Monetize login – logout code changes it will require updating all 34 places where login Selenium script is used. That is time consuming and tedious.

Let’s make login and logout methods reusable using SeleniumScriptBuilder Java library introduced by Monetize SW developers to simplify Selenium tests in Monetize.

Login.java helper class:

```
public class Login extends SeleniumScriptBuilder {
  public static Class doLogin(String subdomain,
                              String email, String password) {
    pushSteps(
            "deleteAllVisibleCookies()",
            "open('/login')",
            f("createCookie('_subdomain=%s')", subDomain));

    pushSteps(
        f("waitForElementPresent('css=a:contains(\"Sign in\")')",
    pushSteps(
            f("type('name=username', '%s')", email),
            f("type('name=password', '%s')", password),
            "clickAndWait('signin')");
    return Login.class;
  }

  public static Class assertLoginSuccess() {
    pushSteps(
            "verifyElementNotPresent('css=.message.error')",
            "waitForElementPresent('css=div#login-status a[href$=logout]')");
    return Login.class;
  }

  public static Class doLogout() {
    pushSteps(
            "waitForElementPresent('css=a[href=\"/logout\"]')",
            "click('css=a[href=\"/logout\"]')");
    return Login.class;
  }
```

Login class extends SeleniumScriptBuilder class that contains following helper methods for building Selenese commands:

- **SeleniumScriptBuilder.pushSteps(String… steps)** – pushes any string or list of string into string list in memory. String is usually a Selenese command but it could be a comment or any other string supported by Selenium script runner in Play framework.

- **f(String, string, String parameter)** – this is just a short version of Java String.format() method that allows insert one string into another string. It is used to create parametrized Selenese commands.

- Each method defined in Login.class returns class itself. It allows chained method calls as it is explained below.

Let's rewrite original login / logout test using Login.class above.

```
#{fixture delete:'all', load:'user-data.yml' /}

%{
   def Login = com.hyperin.test.greige.selenium.Login
}%

${Login
   .doLogin("hyperin", "user", "password")
   .assertLoginSuccess()
   .doLogout()
   .go()}
#{/selenium}
```

Original 11 Selenese commands login/logout test was re-factored to 3 Java method calls that construct the target Selenese script. There are following important items introduced in above re-factored script:

- Groovy script
  - You can notice new section %{ def Login = … }%. Play Framework supports executing Groovy scripts in Selenium test scripts. To access Java class in Selenium script you need to specify full Java package path to it. In order not to repeat full package path for every Java method call we can first define Groovy variable Login pointing to that Java class and then call Java class methods using that variable in the script.
  - This is a very basic example of using Groovy scripts. You can write Groovy scripts surrounded by %{ }% of any complexity in Selenium test and it is very handy in many situations.

- Template statements
  - Another new section above is surrounded by ${ statement }. Statement could be Java method call or variable. Play Framework will execute statement, collect value it returns and will replace whole ${ statement } with a return value.
  - You can see that inside these ${ statement } we execute chained Java methods one by one. Chaining is possible because inside Java methods we return reference to Java class itself as “return Login.class;”
  - The last statement is always go(); It is part of SeleniumScriptBuilder class and simply returns all strings pushed with pushSteps() collected by previous method calls. As stated above Play Framework replaces ${ statement } with a value returned inside it, so after go(); is called ${} will be replaced by Selenese commands pushed in previous Java method calls. Simple and efficient!

Good Java reference to start writing simple Java classes is (Tutorial Java, 2017).

4.6 Selenium scripts and localization

Monetize web application supports several UI languages including Finnish, Swedish, English etc.

Localizations are stored in a simple text files as key=value sequences. For example 3 sample files below contains localization for password field label:

messages.en – English localization file

```
password.field.label=Password
```

messages.fi – Finnish localization file

```
common.password=Salasana
```

messages.sv – Swedish localization file

```
common.password=Lösenord
```

Depending of current selected UI language Play Framework automatically query localization from the relevant file. Localization on the screen can change with time. For example we want to test that current web page contains label for password field. Here is test that passes:

```
#{selenium "Test password label"}
                waitForElementPresent('css=a:contains("Password")
#{/selenium}
```

But what happen if product manager decided to change label to more descriptive “Type your password”? This test will start failing even though password label is still presented on the screen. To avoid such test failures text on the screen should not be hard-coded but rather must be queried from localization database before script is run so that scrip knows correct localization string to look for.

There are 2 ways to do that. First way is to query localization by localization key directly in Selenium script using Groovy snippet as below:

```
%{
   def passwordLabelText = messages.get('password.field.label');
}%
#{selenium "Test password label"}
                waitForElementPresent('css=a:contains(${passwordLabelText})
#{/selenium}
```

Where messages() is a helper method to query localization string from localization database by string key.

The second way is to query localization key in Java helper methods as demonstrated below:

Login.java helper class:

```
import play.i18n.Messages;

public class Login extends SeleniumScriptBuilder {
   public static Class assertPasswordLabelShown() {
        String passLabel = Messages.get("password.field.label");
        pushSteps("waitForElementPresent('css=a:contains(${passLabel}'");
        return Login.class;
    }
```

Selenium script:

```
#{fixture delete:'all', load:'user-data.yml' /}

%{
    def Login = com.hyperin.test.greige.selenium.Login
}%

${Login
    .assertPasswordLabelShown()
    .go()}
#{/selenium}
```

It is recommended to never hard-code UI string values into Selenium scripts to make them localization agnostic.

4.7 Querying database in Selenium scripts

Sometime HTML elements ids or names constructed using dynamic database ids. For example below is an example of Monetize HTML source of Monetize user edit page:

```
<div class="col_3 nest">
    <input type="checkbox" name="selectedCompanyIds" value="65">
    <span data-bind="text: name">Company1 Oy</span>
</div>
<div class="col_3 nest">
    <input type="checkbox" name="selectedCompanyIds" value="66">
    <span data-bind="text: name">Company2 Oy</span>
</div>
<div class="col_3 nest">
    <input type="checkbox" name="selectedCompanyIds" value="67">
    <span data-bind="text: name">Company3 Oy</span>
</div>
```

It is a list of check boxes to select or unselect for the user. Imagine you want to select check box that user belongs to “Company2 Oy”. You could write following simple script to click on “input” element by it's value 66:

```
#{fixture delete:'all', load:'user-data.yml' /}

#{selenium "Click on company"}
                click(css=input[value=66])
#{/selenium}
```

Unfortunately this script will most probably fail the next time you run the test because input field values are dynamically generated from database ids. Next time you run the test the HTML source for “Company2 Oy” might have completely different value id 82 but test expect value 67:

```
<div class="col_3 nest">
   <input type="checkbox" name="selectedCompanyIds" value="82”>
   <span data-bind="text: name">Company2 Oy</span>
</div>
```

The solution is not to hardcode record id in the script but first to query id from H2 database and then use it in the test. Let's refactor above Selenium test case to make it independent of database id:

```
#{fixture delete:'all', load:'user-data.yml' /}

%{
   company = models.common.Company.find("byName", "Company2 Oy").fetch().get(0)
}%

#{selenium "Click on company"}
                click(css=input[value=${company.id}])
#{/selenium}
```

First we query company record from database in a form of Java model class using Groovy snippet. Then use ${company.id} to inject dynamic company id value into Selenium script. In Monetize many HTML pages contain database record ids as part of element name or id or value so this technique is used a lot in tests.

4.8 Cleaning test environment between the tests

Selenium test runner runs Selenium test cases one by one till all requested test cases are executed. In order to make each Selenium test independent from previously executed test test environment must be cleaned from the previous test results.

clean test environment from previous test

run test 1

↓

clean test environment from previous test

run test 2

…

…

clean test environment from previous test

run test N

Play Framework didn't provide automatic cleaning between the test so automation test engineer must take care of this himself.

4.8.1 Cleaning database

H2 database is created at the start of Selenium script runner. All tables created for the Selenium test case will be persisted for the next test unless manually deleted. It is a good practice to delete all tables and load test specific set of data in the beginning of the test to make the test independent from other tests.

Cleaning of database and reloading new data is done as:

```
#{fixture delete:'all', load:'user-data.yml' /}

#{selenium "Click on company"}
                click(css=input[value=66])
#{/selenium}
```

4.8.2 Cleaning web browser data

Web browser data such as local storage and cookies are also not cleaned between the tests. In case Monetize functionality depends on cookies or localStorage and test tests this functionality then these items must be cleaned between the tests.

Cleaning localStorage

There is no standard Selenium command to clean localStorage of web browser. Luckily Selenium allow executing JavaScript commands and we can access localStorage through JavaScript. The way to access localStorage from Selenium was found in one of Stack Overflow answers (Stack Overflow, 2017).

```
#{selenium "Cleaning local storage"}

    // getting reference to web browser window
                getEval('win = (this.page().getCurrentWindow().wrappedJSObject) ?
this.page().getCurrentWindow().wrappedJSObject : this.page().getCurrentWindow()')

    // cleaning local storage using reference to web browser window
    getEval('win.localStorage.clear()')

#{/selenium}
```

The complexity of cleaning localStorage is due to Selenium test runner in Play Framework v1.x in automatic test mode uses very old version of Firefox browser that does not support modern simple way of accessing the localStorage.

Cleaning cookies

Play Framework uses browser cookies to store data for end user sessions. In order tom make tests independent of each other cookies must be cleaned before each test case. Luckily Selenium RC has a dedicated command for this.

```
#{selenium "Cleaning cookies"}
   deleteAllVisibleCookies()
#{/selenium}
```

4.9 The limitation of preprocessors

As explained above Play framework executes all embedded Groovy, Java statements in the per-processing stage to get the final test for Script runner. Even though preprocessing allows to create reusable tests it has its limitations:

- it is not possible to use cycles or conditional statements during test execution
- it is not possible to query data in database during test execution

This comes from the fact that final Selenium script itself cannot contain Java or Groovy statements. All those are executed before test is sent to test runner. This let us create very complex tests but from the other hands we cannot influence test execution after it was constructed.

5 THESIS WORK RESULTS

5.1 Thesis work implementation processes

Thesis work was organized in the following steps:

SW team	Select UI module for UI test automation
Nelli	Write test plan
SW team	Review test plan
Nelli	Implement test plan comments
SW team	Approve test plan
Nelli	Implement Selenium automation tests
Nelli	Make github.com Pull Request
SW team	Review Pull Request
Nelli	Implement comments
SW team	Approve and merge pull request
SW team	Take Selenium test into SW Continuous Integration
Nelli	Go to step 1 till 100% test coverage reached

5.2 Achieved Monetize Selenium test coverage

Monetize SW development team had ~15% of Selenium tests coverage of the product at the time thesis work started. The remaining 75% of Selenium tests development were planned as the thesis work and currently Monetize is covered 100% with Selenium tests. Below is a list of application modules and Selenium test coverage status before and after thesis work.

Monetize UI module	Sub-module	Selenium coverage before thesis work **~15%**	Selenium coverage after thesis work **~100%**
Admin	Home screen	-	High
Admin	Partners	-	High
Admin	Users	-	High
Admin	User Roles	-	High
Admin	User Groups	-	High
Admin	Companies	-	High
Realty	Home screen	-	High
Realty	Ad Media	Medium	High
Realty	Real Estates	-	High
Realty	Ad media types	-	High
Realty	Reservation Calendar	-	Medium*
Realty	Offers and Orders: Listing	-	High
Realty	Offers and Orders: Order flow	-	High
Realty	Offers and Orders: Order flow (web shop)	-	High
Realty	Offers and Orders: Views	-	High
Realty	Offers and Orders: Editors	-	High
Realty	Offers and Orders: Editors (web shop)	-	High
Realty	Customers	-	High
Realty	Statistics	-	High
Realty	Email templates	-	High
Realty	Company information	-	High
Realty	Thirdparty center	Medium	High
Common	Profile app	High	High
Common	Login/Logout	High	High
Realty	Company selector	High	High
Partners	Home screen	-	High
Partners	Reservation calendar	-	High
Partners	Offers and Orders: Listing	-	High
Partners	Offers and Orders: Order flow	-	High
Partners	Offers and Orders: Views	-	High
Partners	Offers and Orders: Editors	-	High

* Reservation calendar is a very complex UI component written purely in JavaScript and it was found impossible to provide High level of testing of all its features due to Selenium RC is outdated and does not provide means to test it properly. However, it was still covered with Selenium test on a good enough level during thesis work.

5.3 Thesis work test cases statistic

Source code lines (SLOC) written:

- 700 SLOC – YML database files
- 3700 SLOC - Java code to generate Selenium commands
- 3000 SLOC - Selenium tests using Java commands
- Total SLOC: 7400

Test scripts written in files:

- 5– Data YML files
- 46 – Java classes
- 41 – Selenium test script

Average time to run all Selenium tests in Continuous integration process:

- 15 minutes

6 SUMMARY

Thesis work goal was achieved and Monetize web based SaaS product is currently covered 100% with Selenium tests. Monetize SW team took this tests into development process and it helps to reduce regression testing and to make sure that all current Monetize features work after new feature is implemented by Monetize SW development team.

REFERENCES

Stack overflow 2017. Quations. Quoted from 3.4.2017 https://stackoverflow.com/a/9986931

Schwaber, K. 2004. Agile project management with Scrum. Microsoft Press.

Schwaber, K. & Beedle, M. 2001. Agile software development with Scrum. Prentice Hall.

Selenium (software) 2017. Selenium software. Quored from 5.4.2017 www.wikipedia.org > selenium (software).

Play selenium 2017. Paly selenium. Quored from 5.4.2017 www.playselenium.com > documentation.

Yaml 2017. Yaml. Quored from 7.5.2017 www.wikipedia.org > yaml.

Tutorial selenium 2017. Tutorial selenium. Quored from 6.5.2017 www.software-testing-tutorials-automation.com > selenium tutorial.

Sorens 2017.

Tutorial Java 2017. Tutorial Java. Quored from 15.4.2017 www.tutorialspoint.com > java.

Distributed version control 2017. Distributed version control. Quored from 7.3.2017 www.wikipedia.org > Distributed version control.

Software as a service 2017. Software as a service. Quored from 6.5.2017 www.wikipedia.org > SaaS.

TERMINOLOGY

Pull Request

Contribution to a source code repository in GitHub.com. “The contributor requests that the project maintainer "pull" the source code change, hence the name "pull request". The maintainer has to merge the pull request if he or she decides the contribution should become part of the source base.“ (Destributed version control, 2017)

Continuous integration - CI

CI is a process when each developer merge new code into main source code branch several times a day. Every merge triggers automatic testing of all available automatic tests including Selenium tests written in this thesis work. CI in Monetize is implemented using Jenkins CI tool.

Jenkins

Continuous integration automation server. Allows creating automatically triggering jobs on GitHib repositories updates that automatically run automatic Selenium and other tests on the latest version of source code. Monetize team uses Jenkins to implement CI process.

Regression testing

Testing of the product to ensure that software that was previously tested and working is still working after recent source code changes in other parts of the product. Automated Selenium testing goal is to reduce manual regression testing as much as possible.

Software as a Service - SaaS

“Software as a service is a software licensing and delivery model in which software is licensed on a subscription basis and is centrally hosted. It is sometimes referred to as "on-demand software" SaaS is typically accessed by users using web browser.“ (Software as a service, 2017)

git

Command line tool to organize controlling versions of source code in a distributed development team. It does not require central source code repository though can have one if agreed in the team.

GitHub

Web service www.github.com providing storage for git based repositories. Monetize team uses github.com to store and mange Monetize source code repository.