jamk.fi

Test automation for Flex Application

Case: Landis+Gyr Oy

Sabina Hudziak

Bachelor’s thesis
May 2017
School of Technology, Communication and Transport
Degree Program in Information and Communications Technology

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s) Hudziak, Sabina	Type of publication Bachelor's thesis	Date May 2017
		Language of publication: English
	Number of pages 45	Permission for web publication: yes
Title of publication **Test automation for Flex Application** Case: Landis+Gyr Oy		
Degree programme Information and Communications Technology		
Supervisor(s) Salmikangas, Esa		
Assigned by Landis+Gyr Oy		

Abstract

Test automation has become more and more popular since it helps to save time, and with test automation, tasks impossible to carry out with manual testing can be performed. The thesis was assigned by Landis+Gyr Oy, an international company that needed automated tests for the graphical user interface for one of its web applications. The application uses Apache Flex technology, an open source framework that allows to create applications for different platforms using the same tool, programming model and codebase . The tests will be utilized for regression testing in order to check if the new functionalities break any older, well-working ones.

In the project, a test was created for one of the features of the application to present the whole test creation process. Firstly, after investigating the problem, JIRA Software was used to write the test steps and the expected results to every test case. Then, the test cases were implemented using Robot Framework with FlexSeleniumLibrary and built-in libraries. However, a custom library was needed and was created using Python 2.7.13, and its functions were used in Robot code. Finally, Git version control system was used to keep track of the code changes, and Gerrit was selected for code review. TortoiseGit tool for Windows made working with Git much faster and easier by providing a graphical representation of Git commands and the file statuses.

As a result, a documented and implemented test was created and prepared to be utilized in regression tests whenever needed. The first part of the report introduces the technologies and the tools that were used, while the second part focuses on the given problem and its solution.

Keywords/tags (subjects)
Test automation, Robot Framework, regression, JIRA, Git, Gerrit, Python

Miscellaneous

Content

Figures

Tables

Acronyms

API	Application Programming Interface
GUI	Graphical User Interface
HTML	HyperText Markup Language
IDE	Integrated Development Environment
SWF	Small Web Format
UI	User Interface

1 Introduction

Nowadays tests are an integral part of software development and companies understand their significance. Tests can be written for different parts of an application, and one of the parts is graphical user interface (GUI). With the help of GUI testing tools, a tester is able to write a test that interacts with a web page the same way as user does.

The main aim of the thesis was to create an automated GUI test for one of the case company web based application, that would be used as a regression test. However, the whole process of creating a test does not only include implementation but also investigating a problem, writing a documentation, creating test steps, expected results, and publishing the test so it can be run by a server when needed.

The web application under the test is written using Apache Flex technology, and the test was created using Robot Framework with FlexSeleniumLibrary, which is a library used to automate Flex application testing. Moreover, Python programming language was used to create custom testing libraries for Robot Framework.

2 Theoretical part

2.1 Testing

Software testing is a process related to software development. It is one of the processes for ensuring the quality of the software, and its aim is to control if the software meets the requirements and the user’s expectations. Tests can be implemented at any stage of development process – depending on the testing method used for a project. (Software Testing, 2017)

Testing itself is a very broad concept. It includes all of the actions which purpose is to find an undesirable behavior of an application, usually caused by a mistake made by a programmer. The process reveals differences between the application workflow and the expected results. The quality of the produced software can be measured by the number of occurred differences – the less differences the better the quality of the product.

Although tests can be implemented at any stage of development, it is highly recommendable to start the testing process as soon as possible. Early detection of bugs helps to save money, time and much more. Occurrence of any bug when the product has been delivered to a stakeholder and is already in use involves high fix costs and can cause customer’s confidence deprivation.

2.2 Regression testing

The software life cycle repeats until the software is no longer supported, and it is said to have reached the end-of-life. Adding a new feature to the software generally means improving its functionality. However, sometimes the new functionally can break some of the old ones that were working fine, and it is not easy to find out what has been broken.

Nowadays, the vast majority of the companies understand how important tests are. However, some of the applications have been developed many years ago, when tests were mainly done for new features, and now they start to crash on the features developed at the beginning of their life cycle, which is why regression testing is a crucial part of the whole testing process. Its aim is to make sure that an application still works properly after making modifications, fixing bugs or adding new features. Regression tests also help to find bugs that have not been found during previous tests. (What is Regression Testing, 2017)

Regression tests can be carried out both for a complete product or just for a part of it. However, regression tests should be executed after smoke or sanity tests. Smoke and sanity tests help to ensure that a new version of the application is ready to be tested and the critical functionalities are working fine. Since regression tests should be executed after every code/environment change, they are a perfect candidate for being automated.

2.3 Test automation

Manual testing is nothing else than a human sitting in front of the computer, performing all possible test scenarios and checking if they are as expected. This kind of test is usually performed only for source code or configuration changes, not for the regression tests because it would take plenty of time and many more testers. Companies cannot afford that even with the best manual testing process. (Why Test Automation, 2017)

People tend to make mistakes during monotonous manual testing, while automated tests are executed exactly as has been written. Once created, they can be used whenever needed as mentioned in the previous chapter, which is one of the reasons why they are used in regression testing. They have a capacity to compare generated results with expected ones and create reports for the tester. (Ibid.)

Moreover, test automation has a wide usage scope and it is an essential part of big software development. First of all, it helps to save time, and by saving time a company saves money as well. Once a test is written it is ready to be executed at any time and tester can focus on writing another test, rather than repeating the same procedures over and over again to ensure software quality. (Ibid.)

Secondly, an automated test can do things impossible to carry out by manual testing and can cover a greater part of the application. Tests are able to get information from the deep parts of the application (for example memory contents), and also they can perform actions that the manual tester cannot. By saving the time spent on the repetitive tests, more tests can be written at the same time, therefore, the percentage of tested parts of the application grows. (Why Test Automation, 2017)

GUI testing

The thesis focuses on graphical user interface testing. Automated tests had to be created for regression testing of one of the company’s web application.

GUI is an integral part of the software and tests for it can be automated as well as for the other parts. Its aim is to check if the application works correctly and meets all the requirements by interacting with the application in a way that normal user does – clicking on the elements, browsing data and much more. The sequence of the GUI events is important.

However, writing regression GUI tests can be more complicated than for other parts of the application, because GUI may change significantly as some elements on the page may change their location or appearance; thus, the test will fail even though it was working fine and the underlying application did not change. Since interacting with the elements on the web page is based on their IDs, when they change the test will fail as well. (Using a Goal-driven Approach to Generate Test Cases for GUIs, 2017)

2.4 Apache Flex

The thesis focuses on testing one of the company’s web applications. It is created using Apache Flex technology, a powerful, open source framework that allows to create applications for different platforms using the same tool, programming model and codebase. (About Apache Flex, 2017)

Flex has been initially developed by Macromedia, in 2005 acquired by Adobe and in 2011 Adobe decided to contribute it to the Apache Software Foundation. The technology was developed for creating Rich Internet Applications (RIA) which offer a dynamic, one-screen-application interface that eliminates the standard solutions known from HTML, such as entering data to the forms or multiple web pages reloading. The term RIA was first used by Macromedia in relation to the web pages created completely using Flash technology. It gets all the needed data from a server at the beginning of a user session, then processes and displays it. (Apache Flex, 2017)

Flex applications are compiled to the SWF files (so called “Flash files”) and can be displayed on the web page using a browser with Flash Player or Adobe AIR plugin. There are two possible ways of creating Flex applications:

- using a free Flex SDK,
- using a paid Flash Builder tool based on IDE Eclipse.

Both solutions create the same file as a result so the difference between these two options is mainly about the convenience of usage and creating the application.

Flex gained popularity due to its performance, asynchronous data transfer, creating an interface visually and high efficiency according to the effort. As mentioned before, Flex application lets to display, drill down and change data without a need to reload the page. It can pull data from multiple back-end sources and update it when the user makes any changes. (About Apache Flex, 2017)

Figure 1 shows one of the Flex component which is a compiler. It produces SWF application combining ActionScript files with MXML layout files.

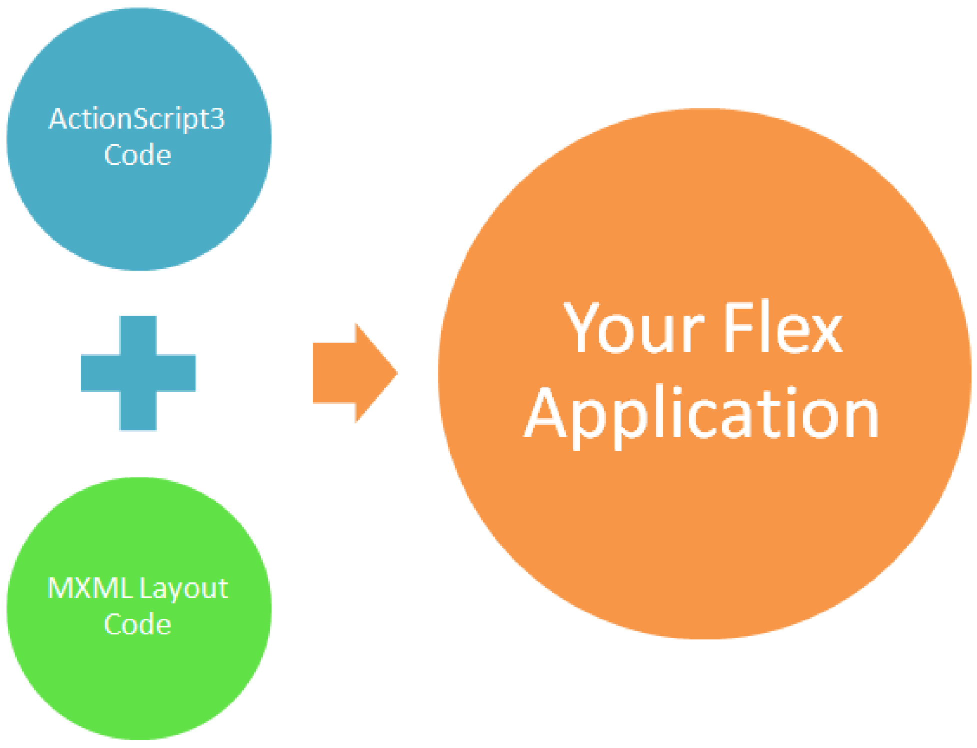


Figure 1. Flex application consists of ActionScript3 Code and MXML Layout Code (adapted from About Apache Flex, 2017).

According to Adobe’s web page (Learning ActionScript3, 2017):
“ActionScript 3 is the programming language for the Adobe Flash Player and Adobe AIR runtime environments.”

However, MXML is an XML-based user interface markup language for creating elements that the user interacts with, while browsing a web page. (About MXML, 2017)

2.5 Inspecting elements of Flex application

Testing UI bases on interacting with a web page. To perform the interaction, locating the web page’s elements is needed. The most popular way is to find the ID of an element. IDs are not visible for a user on a web page, however, they need to be found using the developer tools. For inspecting HTML objects there are tools coming with all the popular browsers. However, Apache Flex application is compiled to SWF file and using the same tools as for HTML based applications is not possible.

For inspecting elements of Flex application in the project, browser Mozilla Firefox with two plugins was used. FlashFirebug is one of the plugins, it debugs Flash/SWF files on the web and can be found at (FlashFirebug, 2017). Thus, finding the elements’ IDs is possible. Pro version of the plugin allows, for example, to see every element’s properties. In order to make the plugin work, another plugin is required and it is Firebug plugin. It delivers the developer tools and can be found at (Firebug, 2017).

2.6 JIRA

2.6.1 Scrum

Figure 2 presents a typical Scrum workflow. The information in this chapter is based on What is Scrum, 2017.

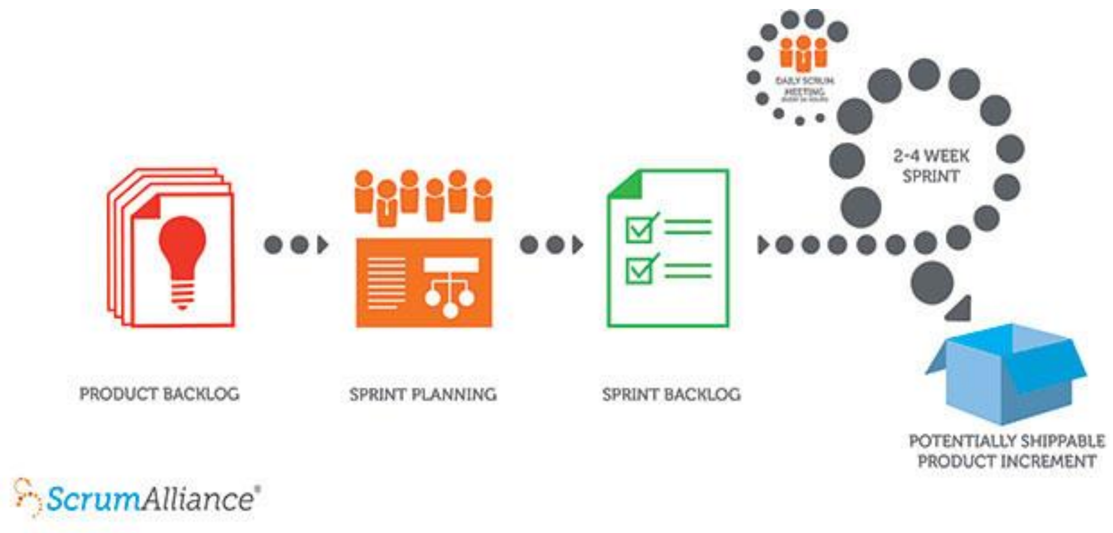


Figure 2. Scrum workflow (adapted from What is Scrum, 2017)

Scrum is an iterative, incremental agile software development framework. The main idea is to divide the whole product's development process into smaller parts (i.e. iterations). One iteration can last a maximum of one month, and it is called a "Sprint". A Sprint's goal is to deliver the next version of the product to user.

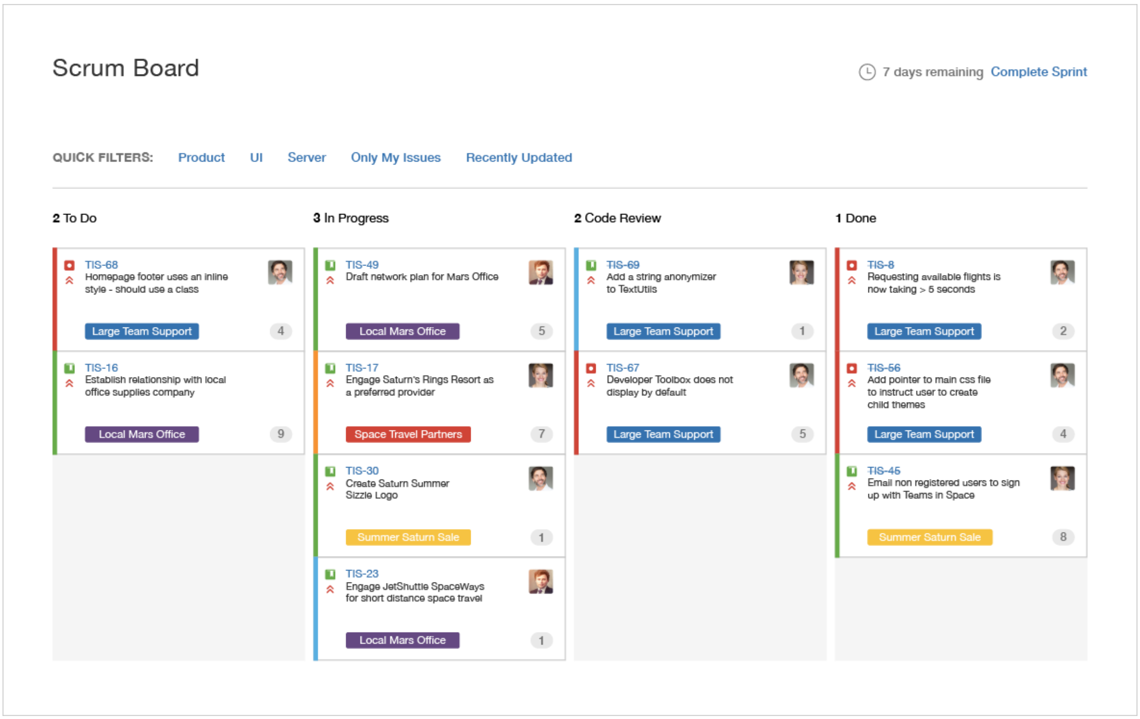
When starting to work on the product, a list of requirements is collected and every requirement is divided into smaller ones if needed. Each Sprint starts with a planning part where the most important tasks are picked up to the "Sprint Backlog", and their implementation time and complexity are estimated.

Moreover, one of the Scrum's ideas are daily, short meetings called "Daily Scrum". During these meetings, every team member says what he or she has done during the previous day, what problems occurred and what the plan for the day of the meeting is. At the end of every Sprint there is a "Sprint Review" meeting, where the results of the Sprint are presented – what has been done.

2.6.2 JIRA Software

JIRA Software is a product developed by Atlassian. It is a very powerful tool for bug tracking, issue tracking and project management functions, mostly used with agile methodologies. Nowadays, the product is in use in thousands of companies all over the world. The information in this chapter is based on JIRA Software Features, 2017.

JIRA Software comes with many tools that help to make the software development process much faster and easier for the development teams. JIRA can be used by every team member in order to plan, track and release software. Figure 3 below shows an example on how the Scrum workflow can be presented on a board.



Scrum Board

7 days remaining Complete Sprint

QUICK FILTERS: Product UI Server Only My Issues Recently Updated

2 To Do

- TIS-68 Homepage footer uses an inline style - should use a class — Large Team Support — 4
- TIS-16 Establish relationship with local office supplies company — Local Mars Office — 9

3 In Progress

- TIS-49 Draft network plan for Mars Office — Local Mars Office — 5
- TIS-17 Engage Saturn's Rings Resort as a preferred provider — Space Travel Partners — 7
- TIS-30 Create Saturn Summer Sizzle Logo — Summer Saturn Sale — 1
- TIS-23 Engage JetShuttle SpaceWays for short distance space travel — Local Mars Office — 1

2 Code Review

- TIS-69 Add a string anonymizer to TextUtils — Large Team Support — 1
- TIS-67 Developer Toolbox does not display by default — Large Team Support — 5

1 Done

- TIS-8 Requesting available flights is now taking > 5 seconds — Large Team Support — 2
- TIS-56 Add pointer to main css file to instruct user to create child themes — Large Team Support — 4
- TIS-45 Email non registered users to sign up with Teams in Space — Summer Saturn Sale — 8

Figure 3. Scrum board in JIRA (adapted from JIRA Software – Features, 2017)

The aim of the board is to display issues from one or more projects and to let user easily manage, view and report on work in progress. Using JIRA Software, it is possible to create two types of boards: Scrum board (for the tasks grouped into separate Sprints) and Kanban board (grouping tasks according to their status – to do, in progress, done). Both types of boards contain reports that present work progress in a graphical way.

The issues displayed on a board can be created by a user. By issue, JIRA means user stories, tasks and bugs. Typically, an issue has a reporter – a person who created it, an assignee – a person working on the issue. The issue can have different statuses, for example, it can be open, in progress, on hold or closed. For tracking issues, there is a JIRA Query Language for creating quick filters allowing to search using certain criteria. Since JIRA can be used by developers as well as, for example, team leaders; they may want to keep track of totally different matters connected with the issues.

One of the useful features, especially for developers, is the possibility to integrate JIRA with some development tools, such as GitHub. JIRA issue identification can be used, for example, in the commit message; thus, it will be automatically connected with the issue and the commit will be visible from JIRA issue view.

2.6.3 JIRA in the case company

The case company utilizes JIRA Software mainly for managing the Sprints' content, for example, all the issues described above connected to Scrum methodology. However, the aim of the thesis was to prepare an automated test that is to be used as a regression test for one of the company's web based application. The application has been initially developed a few years ago and now, it is still under improvement, however, in the time when the thesis is written, all the regression tests are starting to be created, to be sure that new functionalities do not break the old ones.

In this case, regression tests are not a part of the sprint content; however, JIRA issues are still created to present the test workflow, documentation and the progress on working on the issues.

2.7 Robot Framework

2.7.1 About Robot Framework

Robot Framework is an open source software, created for test automation purposes. It is easy to use, yet, a powerful tool that is operating system and technology independent. Initially developed at Nokia Networks, it is now sponsored by Robot Framework Foundation and released under Apache License 2.0. (Robot Framework, 2017)

As presented in Figure 4, Robot Framework's architecture is divided into smaller modules. Every module is independent and carries out different tasks. Test Data contains the test cases written using a tabular format and are executed when Robot Framework is run. Test Data also produces the result files for a tester using HTML format. The core framework uses Test Libraries to communicate with System Under

Test and does not know anything about it. Test Libraries communicate with the system directly or using lower level test tools as drivers. (Robot Framework User Guide, 2017)

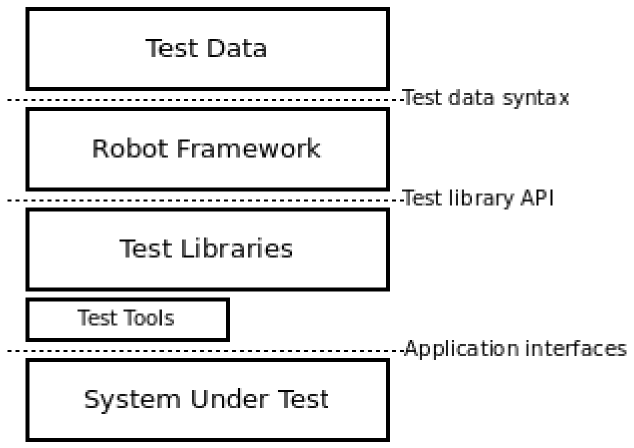


Figure 4. Robot Framework architecture (adapted from Robot Framework User Guide, 2017)

Without a doubt, the reason to write a test is to verify its result – whether it is expected or not. After test case execution, Robot Framework generates a few output files that can be easily read. The most useful are three files: output, log and report.

The output file contains the execution result in machine readable XML format, and the two other files are created based on it. The log file is presented in HTML format and is mainly used when there is a need to investigate a test workflow in detail – in case of failure it is easy to see which keyword has failed and why. Both log and report file contain a statistic information, however, for getting a higher-level overview, the report file is better. It presents the result using HTML format and colors the background with green if all critical tests pass, and red if otherwise. However, the result files can be easily configured; for instance, it is possible to disable a file creation or change the default file name. (Ibid.)

2.7.2 Creating test data

All information in this chapter is based on Robot Framework User Guide, 2017.

File format

The first step in creating test data is to choose a file format. Robot Framework supports four file formats; and they all use tabular format:

- hypertext markup language (HTML),
- tab-separated values (TSV),
- plain text,
- reStructuredText (reST).

Following the Robot Framework User Guide's recommendation, plain text format should be used when there are no special needs. Therefore, this format is used in the thesis. It is easy to edit using any popular text editor and available plugins that support syntax highlighting make it even more convenient.

Table 1 shows four types of tables the test data is structured in. In the plain text format, each table name has to start with at least one asterisk and the separator between the cells is two or more spaces or a pipe character surrounded with spaces. However, it is recommended to use four spaces as the separator, because it makes the code more readable. To use an empty cell, the ${EMPTY} variable must be used or a single backslash.

Table 1. Different test data tables (adapted from Robot Framework User Guide, 2017)

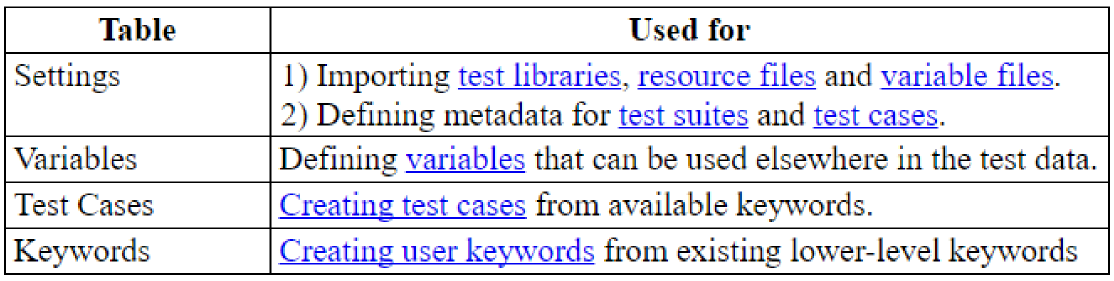


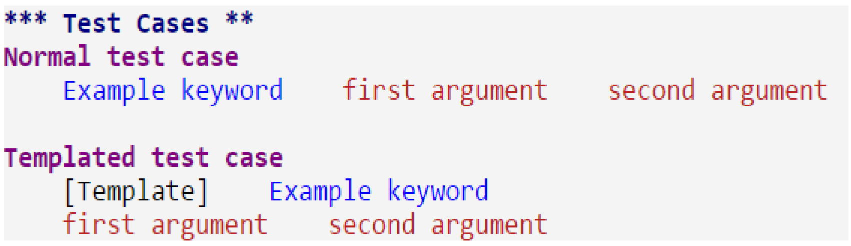
Table	Used for
Settings	1) Importing test libraries, resource files and variable files. 2) Defining metadata for test suites and test cases.
Variables	Defining variables that can be used elsewhere in the test data.
Test Cases	Creating test cases from available keywords.
Keywords	Creating user keywords from existing lower-level keywords

Test cases

The next step in creating test data is to create test cases table. The test cases utilize keywords written by a user into the keywords table or from test libraries or resource files. The first column in the test cases is its name, the second column usually contains the keyword name, but it can also contain a variable to be assigned or settings.

Every test case can have its own settings that belong only to it. The setting name is surrounded with square brackets to distinguish it from a keyword. Five possible settings can be used within a test case:

- Documentation – every test case can have its documentation, however according to Robot Framework User Guide if the test case needs to be documented, it usually means that the keywords it uses are not descriptive enough.
- Tags – help to manage different test cases. With tags it is easy to specify which test should be run, and, since they are shown on the output report, they provide good statistics overview.
- Setup/Teardown – both settings use a single keyword executed before or after the test case execution respectively. If more than one keyword needs to be used, then a higher level keyword can be created to collect them. The keyword used as a teardown behaves differently than a normal keyword as a default; even if one keyword it contains fails, the keyword continues its execution.
- Template – converts normal keyword-driven test cases into data-driven tests. The extract below shows the difference between these two approaches.



```
*** Test Cases **
Normal test case
    Example keyword    first argument    second argument

Templated test case
    [Template]    Example keyword
    first argument    second argument
```

- Timeout – is used to prevent a test case to hang endlessly. If the test case does not finish its execution after a set time, it is stopped forcefully. However, the test cases should be written so there is no need to use the timeout setting.

Keywords

As mentioned before, test cases use keywords to perform a test. There are two types of keywords: user keywords and library keywords. User keywords are higher level keywords that combine other keywords together – both other user keywords or from libraries. They have an identical syntax to the one used in test cases table. Moreover, is it possible to pass some arguments to them, return values or use variables, so they are similar to the functions in the common programming languages. Alike in test cases, keyword table have settings: Documentation, Tags, Arguments, Return, Teardown, Timeout. Again, to distinguish them from other keywords, they are surrounded with square brackets.

Library keywords are so called lower level keywords, usually implemented using a popular programming language (Python or Java) and their aim is to communicate with the system under test. There are already made built-in libraries that can be easily used by higher level keywords. However, it is possible to extend the functionality and/or write own test libraries.

Test suite

Test suite is a set of the test cases from one test case file or from multiple test cases files. It is created automatically, using all test cases' tables. Its name is also created automatically from the file or directory name; an extension is ignored, underscores are replaced with spaces and the name is title cased if it consists of more than one word.

The test case files can be grouped into directories creating higher-level test suites, or even these directories can be grouped into other directories. Settings can be used as well as for the previous examples to customize test suites.

For the test case files, the following settings are available: Documentation, Metadata, Suite Setup and Suite Teardown. Suite Setup is run before all the test cases and it can be used to check the preconditions. If the setup fails, all test cases are marked as fail. Suite Teardown, similarly to Test Teardown, is used for cleaning up and it is run after the execution of all test cases. Every keyword it contains will be run even if some of them fail.

The test suite created from a dictionary can also have customized settings. They can be put into initialization file and it can be the same as in test cases files and: Force Tags, Test Setup, Test Teardown, Test Timeout.

2.8 Python

As mentioned in the previous chapter, Python is one of the programming languages used for creating lower level keywords in Robot Framework. The keywords created in the thesis are written using Python 2.7.13.

PyCharm is the IDE for Python programming language created by JetBrains used for the project with IntelliBot plugin that supports Robot Framework and provides syntax highlighting. (PyCharm, 2017; IntelliBot, 2017)

Python is a high-level programming language with a large standard library that supports many common programming tasks. It also supports many programming paradigms: object-oriented programming, structured programming and functional programming. Its syntax is clear and easy to read; the code can be grouped into modules and packages, data types are strongly and dynamically typed, and it provides automatic memory management with garbage collection. (Python Overview, 2017)

2.9 Git, Gerrit, Tortoisegit

2.9.1 Git

Git is one of the popular version control systems, created by Linus Torvalds in 2005 as a tool for development of the Linux kernel. Nowadays, using a version control system for keeping track of the changes made on a project over time is an essential part in development process. It lets a user recall a specific version of a file or set of files at any time in the future. Moreover, it can be utilized not only in terms of producing software, but in other cases as well, since it supports almost any kind of files. (About Version Control, 2017)

There are three version control methods (About Version Control, 2017):

- Local Version Control System – the idea was to create a simple database on a local computer that stores the changes made on files. Only one user can work on a file at a time.
- Centralized Version Control System – there is one version database, for example on a server, that stores a repository, and users can connect to the database to get the newest version of a file. If the server breaks down, all data can be lost.
- Distributed Version Control System – every user has the whole version of a repository on his or her local computer, not only the newest version of the files. Comparing to the Centralized Version Control System, when a server breaks down, data can be retrieved to the server from any user's computer.

Git uses Distributed Version Control System that is presented in Figure 5. Unlikely to the other version control systems, almost every operation can be done locally, without an internet connection. Since the whole repository is stored on the user's computer, Git can easily show a project history or the differences between version of the files. Moreover, it is possible to work on the project, make changes and commit them whenever it is suitable for the user. Once the user has a network connection, the changes can be uploaded to a server. (Ibid.)

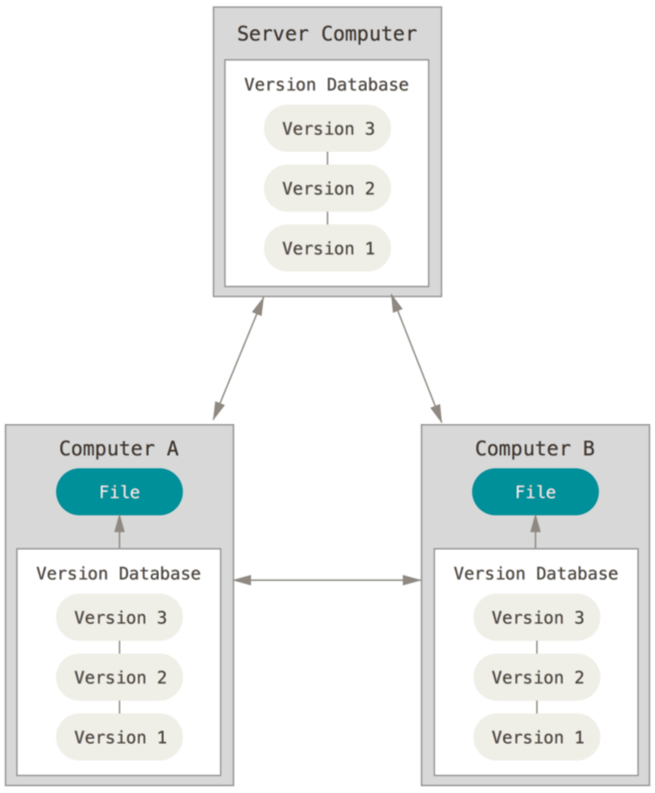


Figure 5. Distributed version control (adapted from About Version Control, 2017)

Another feature that distinguishes Git from other version control systems is the way how it stores the file changes. Most of the systems keep changes made to each file over time, whereas Git keeps the snapshots. To improve its performance, if the file has not changed, it is not saved again, only a reference to the file.

Moreover, Git has an integrity mechanism that can easily detect any changes or corruptions. Everything is check-summed before it is stored and then referred to by that checksum. SHA-1 is used for checksumming and it is a 40-hexadecimal character string that is calculated basing on the file content or directory structure in Git. (Git Basics, 2017)

One of the most important knowledge base when working with Git is to understand that the files can have three main states (Ibid.):

- committed – safety stored in a local database,
- modified – modified by a user, but not committed yet,
- staged – a modified file is marked to go into a next commit snapshot in its current version.

As presented in Figure 6, Git project is divided into three main sections: Git directory, Staging Area and Working Directory. The main part of Git is the Git directory, where the metadata and object database for the project is stored. Working directory contains files that are pulled out from the database and can be modified by the user. (Git Basics, 2017)

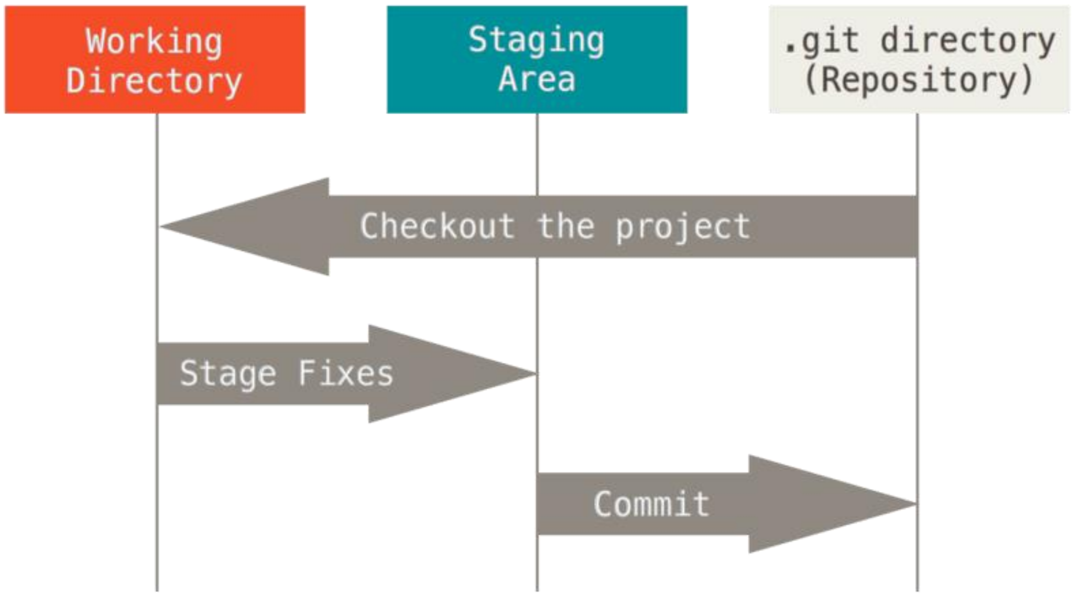


Figure 6. Working tree, staging area and Git directory (adapted from Git Basics, 2017)

Staging Area is a place, where information about what will go into the next commit is stored. Commit can be performed whenever the project reaches a state that the user wants to record. (Git Basics, 2017)

2.9.2 Gerrit

All information in this chapter is based on Gerrit Code Review, 2017. Thus, according to Gerrit’s web page:

“Gerrit is a web-based code review tool built on top of the Git version control system.”

Gerrit intends to provide a lightweight framework that helps with managing a project. As shown in Figure 7, Gerrit is in the central place of a repository. Before the changes made on the project can be merged to the code base, they should be reviewed by other team members first. The traditional approach is to gather the team together and go through all the new code line by line. Another one can be to use Gerrit and let the code be reviewed at any time before being accepted into the code base.

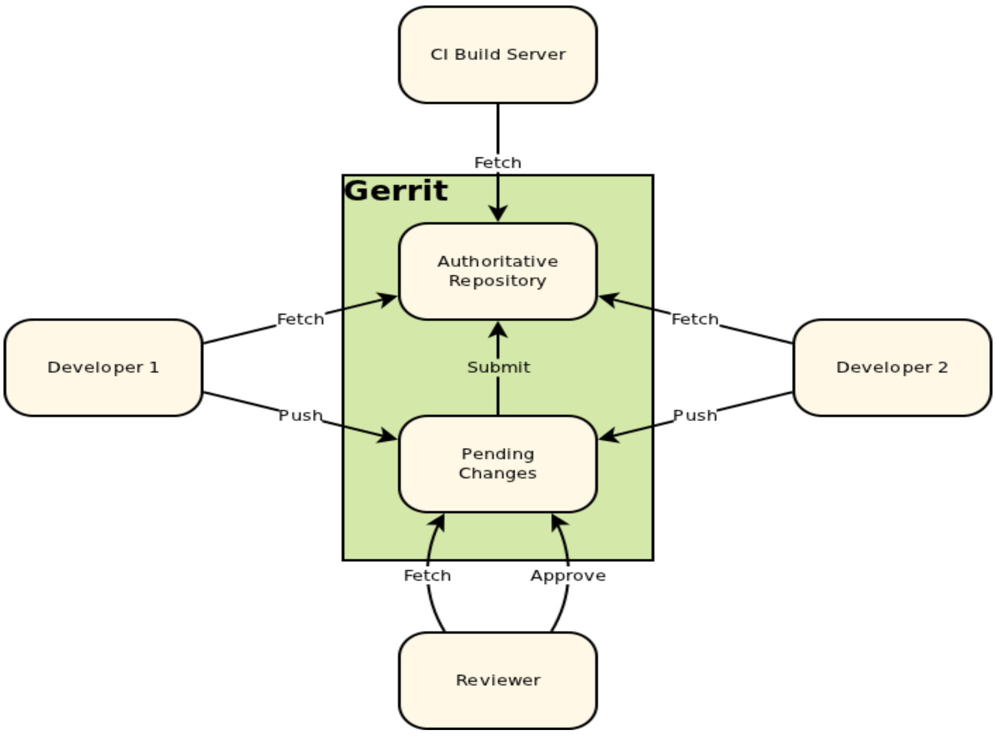


Figure 7. Gerrit in place of Central Repository (adapted from Gerrit Code Review, 2017)

Once the code changes made locally are pushed to Gerrit, its default workflow requires two checks before the change can be accepted: verification check (if the code compiles, unit tests pass etc.) and a code review (person, who is looking at the code). Verification can be done by an automated build server, using for example Gerrit Trigger Jenkins Plugin that triggers builds when a “patch set” is created.

As mentioned earlier, the code review is done by a person. The reviewer is able to put some comments to the code or give -2, -1, 0, +1 or +1 scores. The -2 and +2 are blocking or allowing the change respectively. The -1 and +1 are just an opinion and 0 is a no score. For a change acceptance, it must have at least one +2 score and zero -2 votes.

2.9.3 TortoiseGit

TortoiseGit is open source Windows Shell Interface for Git. It provides a graphical representation of Git commands. All the commands that make sens for selected file or directory are visible directly from the Windows Explorer, so there is no need to use Git Bash and remember specific commands for different operations. (About TortoiseGit, 2017)

Another functionality that makes using Git version control system much easier is the ability to see the status of the files directly in Windows explorer. As shown in Figure 8, both files and directories show their current status so a user does not need to type any special commands every time he or she wants to get this information. (Ibid.)

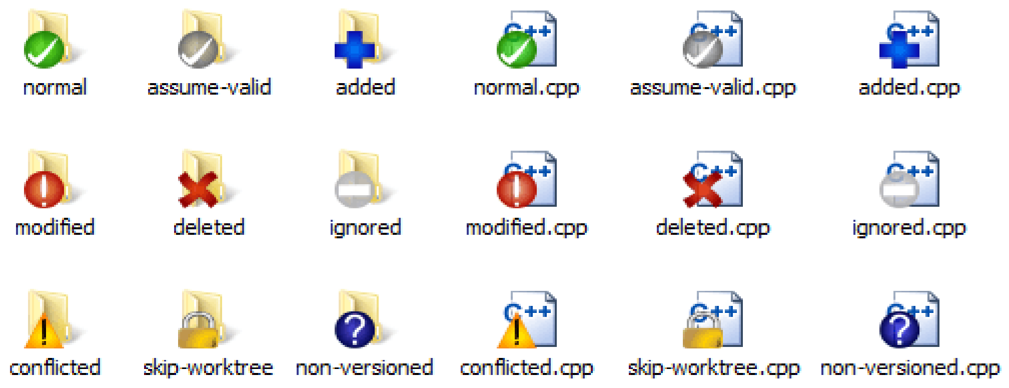


Figure 8. Overlay icons in explorer indicating the status of files and folders (adapted from TortoiseGit screenshots, 2017)

Working with branches is a very common use case in Git. TortoiseGit's Log Dialog is a tool that helps users to see a project history, see where the certain branch is relative to other branches and to all commits, and much more. Via Log Dialog it is possible to merge two commits to one for the current branch a user is working on. Moreover, this tool provides context menu commands which a user can use to get more information about the project history. (TortoiseGit screenshots, 2017)

3 Preparation and implementation

3.1 Case company

Landis+Gyr is a multinational corporation with 100+ years history and has companies all over the world (headquartered in Zug, Switzerland), in over 30 countries on all five continents. Its aim is to provide metering solutions for electricity, gas, heat/cold and water for energy measurement solutions for utilities. The company cooperates with over 3,500 energy enterprises worldwide, offering variety of energy meters and integrated smart metering solutions, enabling utilities and end-users to make better use of scarce resources, save operating costs and protect environment. One of the offices is situated in Jyskä, Finland and it is the place where the thesis was written. (Why Landis+Gyr, 2017)

3.2 Dashboard – application used in the test

Gridstream AIM is a smart metering software design to simplify the way energy companies collect and manage their metering data efficiently. It provides data management capabilities, task flow engine and validation beyond basic head end system functions. (Gridstream AIM, 2017)

One of the Gridstream AIM software is AIM Dashboard. According to Landis+Gyr's internal documentation, Dashboard is a web based application that helps with searching, viewing and controlling customer's metering. It is easy to use, but powerful tool which does not need any previous knowledge of advanced metering management in order to browse metering information stored in the AIM database. Via Dashboard the customer is able to see metering information presented in a graphical format such as usage of the energy during certain period of time. The customer's contract details, installed devices, latest measured values, relay statuses and power quality etc. are shown for every metering point. Moreover, online meter readings as well as controlling functions such as connection control can be done via Dashboard.

The thesis focuses on one of the Dashboard features which is searching for customers or metering points. The search panel is presented in Figure 9.

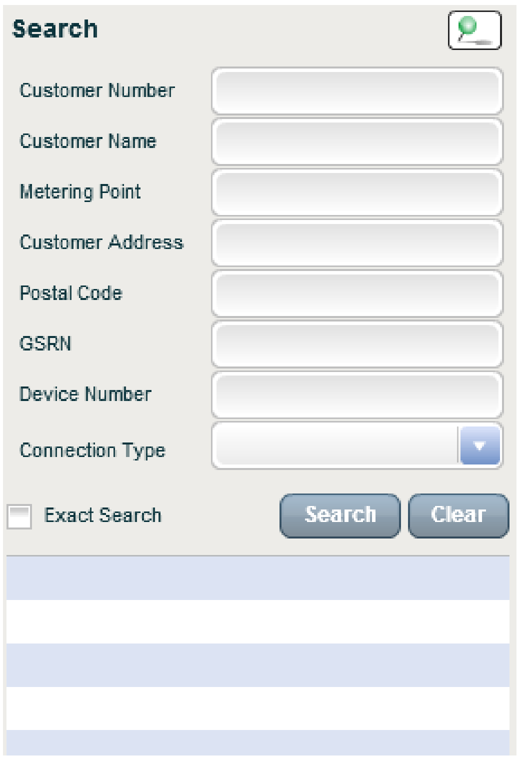


Figure 9. Search panel in Dashboard.

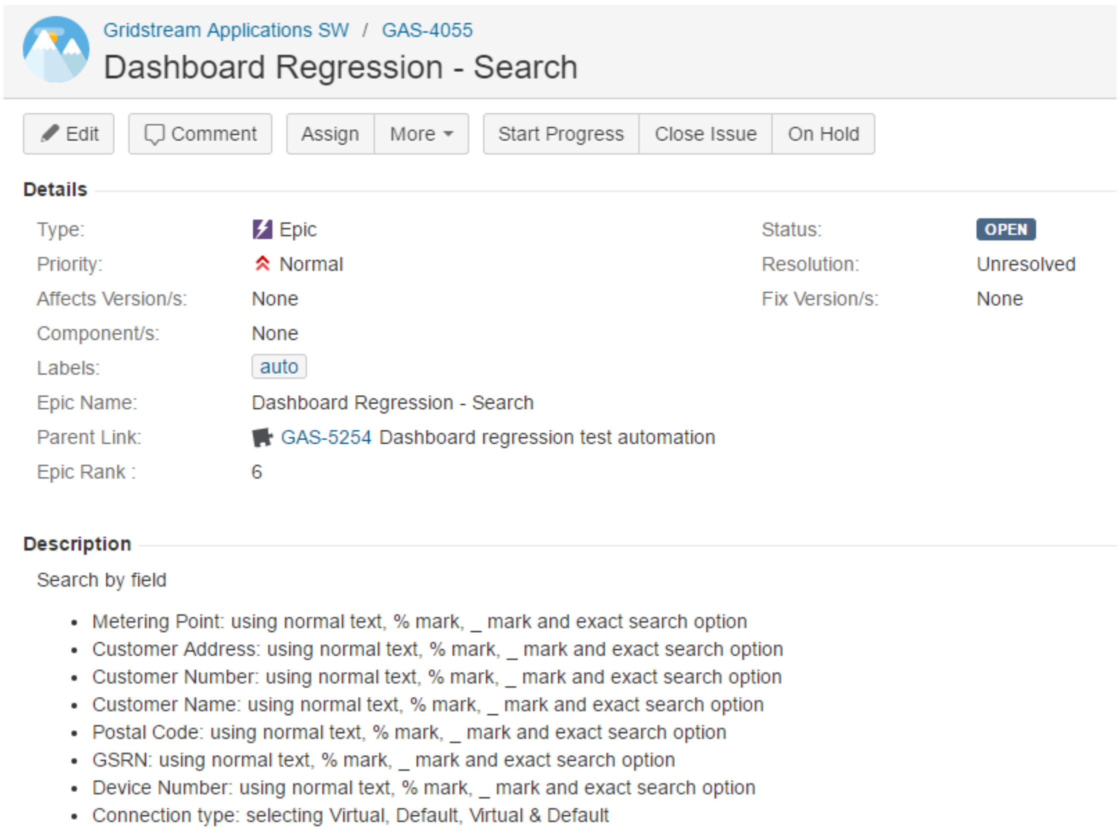
The search panel contains two parts: the first part includes input fields where search criteria can be entered, Exact Search checkbox, Search and Clear button. The second part is a list of results matching the given criteria. However, the search result list can contain up to 200 rows that match the given criteria.

3.3 Investigating the problem

The first step before creating a test that will be run automatically as a regression test for Search functionality was to investigate every possibility of input text that can be entered to one of the search fields. The result can be searched in four different ways:

- using normal text – by entering the search criterion or part of it in the entry field. Result list will contain of customers that match exactly or begin with the given criteria;
- using % mark – "%" mark replaces text in the search criterion;
- using _ mark – "_" mark replaces one character in the search criterion;
- using exact search – searches for results that exactly match the given criteria.

The second step was to create detailed test cases that include test steps and expected results. As shown in Figure 10, JIRA epic "Dashboard Regression - Search" was created to show all the test possibilities for this certain feature.



Dashboard Regression - Search

Details

Type:	Epic	Status:	OPEN
Priority:	Normal	Resolution:	Unresolved
Affects Version/s:	None	Fix Version/s:	None
Component/s:	None		
Labels:	auto		
Epic Name:	Dashboard Regression - Search		
Parent Link:	GAS-5254 Dashboard regression test automation		
Epic Rank :	6		

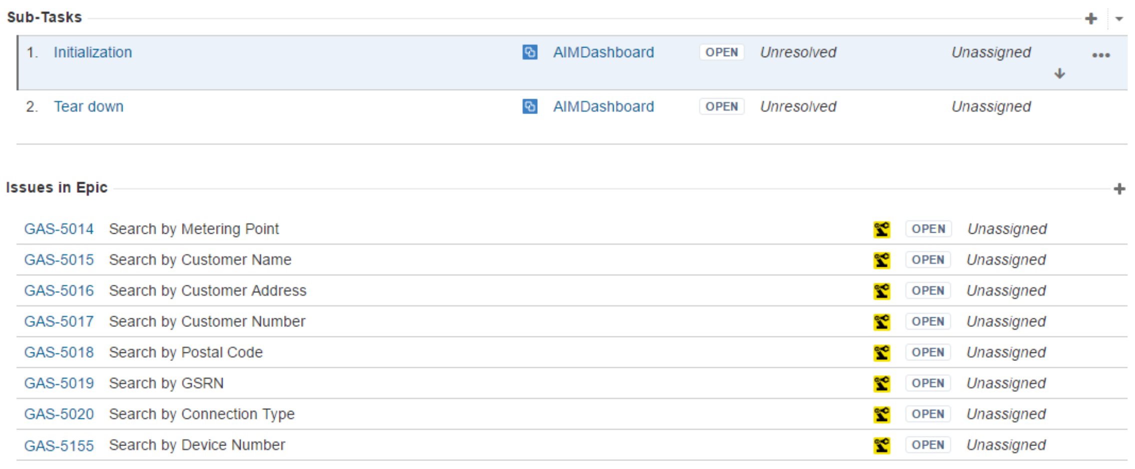
Description

Search by field

- Metering Point: using normal text, % mark, _ mark and exact search option
- Customer Address: using normal text, % mark, _ mark and exact search option
- Customer Number: using normal text, % mark, _ mark and exact search option
- Customer Name: using normal text, % mark, _ mark and exact search option
- Postal Code: using normal text, % mark, _ mark and exact search option
- GSRN: using normal text, % mark, _ mark and exact search option
- Device Number: using normal text, % mark, _ mark and exact search option
- Connection type: selecting Virtual, Default, Virtual & Default

Figure 10. JIRA issue with all possible test cases.

Because of many possible test cases, they were grouped according to the field names they test, and separate JIRA issues were created for each one and linked to the main epic as a parent so they can be easily seen from the main issue view (as shown in Figure 11).



Sub-Tasks

1.	Initialization	AIMDashboard	OPEN	Unresolved	Unassigned
2.	Tear down	AIMDashboard	OPEN	Unresolved	Unassigned

Issues in Epic

GAS-5014	Search by Metering Point	OPEN	Unassigned
GAS-5015	Search by Customer Name	OPEN	Unassigned
GAS-5016	Search by Customer Address	OPEN	Unassigned
GAS-5017	Search by Customer Number	OPEN	Unassigned
GAS-5018	Search by Postal Code	OPEN	Unassigned
GAS-5019	Search by GSRN	OPEN	Unassigned
GAS-5020	Search by Connection Type	OPEN	Unassigned
GAS-5155	Search by Device Number	OPEN	Unassigned

Figure 11. Sub-Tasks and issues connected to the main epic.

Every group has its own sub-tasks (test cases) that show the main idea for the test case workflow. The test focuses on testing Metering Point field and it has four test cases.

1. Search by Metering Point using normal text

   Test steps:

   1. Type “X” into Metering Point field
   2. Click Search button

   Expected result:

   - “X” and “XX” metering points should be visible on the result list

2. Search by Metering Point using % mark

   Test steps:

   1. Type “%” into Metering Point field
   2. Click Search button

   Expected result:

   - “X” and “XX” metering points should be visible on the result list

3. Search by Metering Point using _ mark

   Test steps:

   1. Type “X_” into Metering Point field
   2. Click Search button

   Expected result:

   - “XX” metering point should be visible on the result list

4. Search by Metering Point using exact search

   Test steps:

   1. Type “X” into Metering Point field
   2. Select Exact Search checkbox
   3. Click Search button

   Expected result:

   - “X” metering point should be visible on the result list

However, before the test can be executed, there has to be test data prepared beforehand and to be reachable from Dashboard. In this case, test data (metering points) is prepared and deleted manually so it is not a part of test automation. Due to the limitations of the search result list that can show up to 200 rows, the test data must contain unique values so the tester can control the results of the search and be sure that the search procedure will produce only this specific data.

As shown in Figure 11, under the main Dashboard Search epic there are two subtasks: Initialization and Teardown. The initialization should be done before executing the test suite and Teardown afterwards. In the Initialization part, the test data is prepared and the user is logged into Dashboard. In the Teardown part, the user is logged out from application, and the previously created metering points are deleted.

3.4 Test cases implementation

3.4.1 Project structure

It is important to have a transparent file structure within the project to easily maintain and expand it in the future. The automated test for Search in Dashboard is only one of many test suites written to test Dashboard features. Some keywords are used only by one test suite; however, some of them are shared between a few suites so the file structure should also be separated into folders.

The project has been divided into smaller parts/folders:

- Suites
    - “SearchTest.robot” - contains test cases
- Keywords
    - “SearchTest_keywords.robot” - contains keywords used in “SearchTest.robot”
    - “General_keywords.robot” - contains keywords used in “SearchTest.robot” and other test suites
- Libraries
    - “DashboardKeys.py” - Python file containing lower level keywords and using functions from FlexSeleniumLibrary
- Output
    - “log.html” - output file produced by robot framework after running the test
    - “report.html” - output file produced by robot framework after running the test
    - “output.xml” – output file produced by Robot Framework after running the test, contains an XML, machine readable test result
- “imports.robot” - is imported in suites’ robot files and collects keywords files

3.4.2 Test Cases

As mentioned earlier, four possible cases of search functionality for one input field have to be tested. The names of the test cases should be as descriptive as possible and use higher level keywords that are also easy to understand what they do. The test suite “SearchTest” includes four test cases, one for each test possibility.

On the code snippet below, “*** Settings ***” table is presented. It includes “Documentation” setting which is a brief description of the test suite - what is tested; suite setup which is executed before the whole test suite, it is a keyword “Initialize Dabo Test Suite”; suite teardown which is executed after the test executes and it is also a keyword; force tag “dabo” which is added to every test case in the test suite and a resource file that imports keywords files.

```
*** Settings ***
Documentation           Tests search functionality in Dashboard
...                     using Metering Point input field.
Suite Setup             Initialize Dabo Test Suite
Suite Teardown          Teardown Dabo Test Suite
Force Tags              dabo
Resource                ../imports.robot
```

Suite Setup can be a regular keyword as used in the test cases. However, it is executed before all test cases run and if any keyword it contains fails, all the test cases within a test suite are marked as failed as well. This property makes it a good tool for checking the preconditions that have to be met before the test can be executed.

“Initialize Dabo Test Suite” is a higher level keyword that does the following: opens a browser, goes to login page and logins to Dashboard. The keyword comes from “General_keywords.robot” file, because it can be reused in other test suites as well.

Suite Teardown behaves slightly differently; it will run every keyword it includes even if one of them fails. Teardowns are then used for cleaning up after the test. Keyword “Teardown Dabo Test Suite” is located in “General_keywords.robot” and performs log out from application and close the browser. Moreover, every test case can have its own setup and teardown, however, in this particular case there is no need for that.

As shown on the code snippet below, the “*** Test Cases ***” section includes four separate test cases with descriptive names: “Search by metering point using normal text”, “Search by metering point using % mark”, “Search by metering point using _ mark” and “Search by metering point using exact search”. As mentioned in one of the previous chapters, test data has to be prepared manually beforehand. Two metering points were created with unique values so the tester can be sure that they will be the only ones on the result list. The search criteria have to be chosen carefully as well due to the same reason.

```
*** Test Cases ***
Search by metering point using normal text
    Enter Text To Field Metering Point    ${METERING_POINT_1}
    Click Search Button
    Verify Result List Contains    ${CUSTOMER_ADDRESS_1}
    ...    ${CUSTOMER_NAME_1}    True
    Verify Result List Contains    ${CUSTOMER_ADDRESS_2}
    ...    ${CUSTOMER_NAME_2}    True
```

```
Search by metering point using % mark
    Clear Search Fields
    ${new_string}=    Replace Char In String    ${METERING_POINT_1}    -1    %
    Enter Text To Field Metering Point  ${new_string}
    Click Search Button
    Verify Result List Contains    ${CUSTOMER_ADDRESS_1}
    ...    ${CUSTOMER_NAME_1}    True
    Verify Result List Contains    ${CUSTOMER_ADDRESS_2}
    ...    ${CUSTOMER_NAME_2}    True

Search by metering point using _ mark
    Clear Search Fields
    ${new_string}=    Replace Char In String    ${METERING_POINT_2}    -1    _
    Enter Text To Field Metering Point  ${new_string}
    Click Search Button
    Verify Result List Contains    ${CUSTOMER_ADDRESS_1}
    ...    ${CUSTOMER_NAME_1}    False
    Verify Result List Contains    ${CUSTOMER_ADDRESS_2}
    ...    ${CUSTOMER_NAME_2}    True


Search by metering point using exact search
    Clear Search Fields
    Enter Text To Field Metering Point    ${METERING_POINT_1}
    Select Exact Search Checkbox
    Click Search Button
    Verify Result List Contains    ${CUSTOMER_ADDRESS_1}
    ...    ${CUSTOMER_NAME_1}    True
    Verify Result List Contains    ${CUSTOMER_ADDRESS_2}
    ...    ${CUSTOMER_NAME_2}    False
```

File “fijyvvrapp07.robot” stores server information, credentials needed for login purposes and also details of two created metering points, as these metering points had been created only to be used on this particular server. The variables with metering details are shown below:

```
*** Variables ***
# Created Metering Point 1
${METERING_POINT_1}                 uniqueMP1
${DEVICE_NUMBER_1}                  79000000
${CUSTOMER_ADDRESS_1}               uniqueAddMP1 274 4 3
${POSTAL_CODE_1}                    99991 JYSKAWICE
${GSRN_1}                           00001
${CUSTOMER_NAME_1}                  uniqueCustMP1
${CUSTOMER_NUMBER_1}                79000000

# Created Metering Point 2
${METERING_POINT_2}                 uniqueMP11
${DEVICE_NUMBER_2}                  79000001
${CUSTOMER_ADDRESS_2}               uniqueAddMP11 274 4 3
${POSTAL_CODE_2}                    99911 JYSKAWICE
${GSRN_2}                           00011
${CUSTOMER_NAME_2}                  uniqueCustMP11
${CUSTOMER_NUMBER_2}                79000001
```

It is easy to notice from the code snippet that some variables are written upper case and some lower case. It is recommended to use lower case with local variables available only inside a certain scope and upper case with others (such as global, suite, test level). Comment lines start with “#” character.

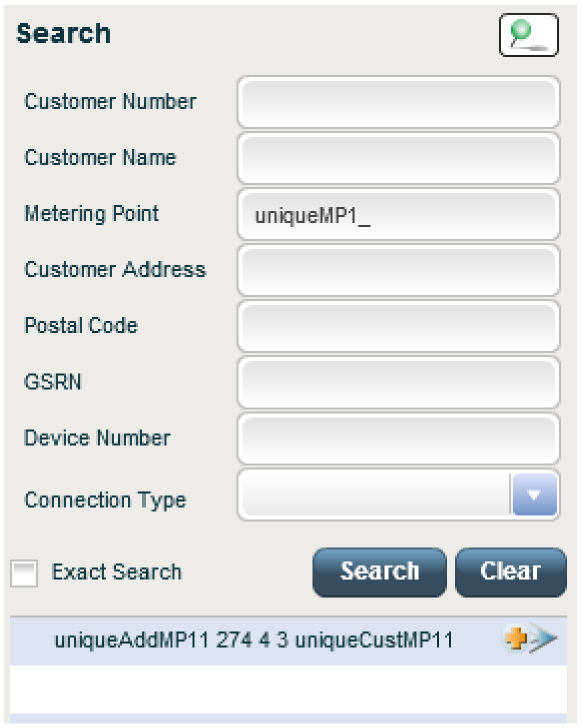
The workflow of the test cases is easy to understand even without looking into keywords, which is why no special documentation to each one is needed. In every test case the scenario is very similar: the first step is to enter the search criterion to the metering point input field, then click search button and verify if the result list contains expected results. Since the results on the list must be limited, special combination of strings basing on the metering names are created with the help of a user keyword.

The test uses two metering points: uniqueMP1 and uniqueMP11. In checking search functionality using normal text, the first metering point’s name is entered, so both meterings should be visible on the result list. This is because search fields work as to the search criterion is appended “%” mark (which replaces any text) to the end of it.

In the second test case an input string has to be modified, so the last character of the first metering point’s name is replaced by “%” mark and it gives “uniqueMP%” string. The result list should contain both metering points.

The third case presents the usage of “_” mark that replaces exactly one character. Similar to the previous case, the metering’s name is modified. However, in this case the name of the second metering is modified, and the last character is replaced by “_” mark which gives “uniqueMP1_” string. In the same test case, it is possible to do both: positive and negative test because the first metering point should not be on the result list and the second one should be there, as presented in Figure 12.

The last case uses exact search – the first metering point name is entered and the exact search checkbox is selected. Only the first MP should be visible on the result list.



Customer Number	
Customer Name	
Metering Point	uniqueMP1_
Customer Address	
Postal Code	
GSRN	
Device Number	
Connection Type	

uniqueAddMP11 274 4 3 uniqueCustMP11

Figure 12. Search panel and result list.

Each row on the search result list displays two MP's properties combined together: customer address and customer name, which is why these two values are passed as arguments to the keyword "Verify search result list contains". The third passed parameter says if the specific values should be visible (true) or should not be visible (false).

3.4.3 Higher level keywords (user keywords)

One of the most powerful features of Robot Framework are keywords. They are used in test cases and help the tester to maintain an adequate abstraction level of the test. User-defined keywords have a very similar syntax to the one used for creating test cases, it can consist of other higher level keyword or library keywords and also have arguments and return values if needed. FOR loops can be used as well.

The code snippet below shows three higher level keywords used in test cases. They use lower level keywords from built-in library and custom library.

```
Replace Char In String
    [Arguments]    ${string}    ${position}    ${char_to_replace}
    ${length}=    Get Length    ${string}
    @{chars}=    Split String To Characters    ${string}
    Run Keyword If    ${position}>=-${length} and ${position}<${length}
    ...   Set List Value    ${chars}    ${position}    ${char_to_replace}
    ${string}=    Catenate    SEPARATOR=    @{chars}
    [Return]    ${string}


Verify Result List Contains
    [Arguments]    ${value1}    ${value2}    ${expected_result}=True
    ${result}=    Search Result List Contains Value    ${value1}    ${value2}
    Should Be Equal    '${result}'    '${expected_result}'

Clear Search Fields
    Click Metering Point Clear Search Fields Button
    Unselect Exact Search Checkbox
```

The keyword “Replace Char In String” does exactly what it says – replaces one character in a given string at a given position index and is used in test cases. “[Arguments]” is used for specifying user keyword arguments and it expects 3 arguments: string to be modified, position on which char to be replaced and char to be replaced to. The first string is split into a list of characters, then the position argument is checked if it is not out of range. The “Run Keyword If” keyword runs the keyword given as one of the arguments when the condition is true. Another built-in keyword “Set List Value” sets the char of the list specified by position to the given char. Index 0 means the first position in list, -1 last and so on. “[Return]” returns a string concatenated from modified characters’ list.

Other two keywords are created for checking if the expected result is on the result list (“Verify Result List Contains”) and clearing search fields used before entering new search criteria (“Clear Search Fields”).

3.4.4 Lower level keywords (custom test libraries)

Lower level keywords are often implemented using a common programming language – Python or Java. Robot Framework delivers several standard libraries that are bundled in with the framework, such as:

- BuiltIn that provides generic, often used keywords,
- Collections that provides a set of keywords for handling Python lists and dictionaries,
- String library for generating, modifying and verifying strings.

Moreover, it is easy to implement custom test libraries if needed.

The previous chapters look almost the same as they would look when testing an HTML based application. The differences between testing HTML and Flex application are visible on the library level.

The commonly used library for test automation is Selenium2Library, however, it does not work with Apache Flex application; neither do the common developer tools for inspecting the elements on the web page.

In this project FlexSeleniumLibrary was used that can be found at (FlexSeleniumLibrary, 2017).

It helps to automate Flex application testing with Robot Framework. However, to be able to use keywords from the library the Flex application has to be built with the Selenium-Flex API as a dependency. The Selenium-Flex API can be originally found at (Selenium Flex API, 2017).

Four lower level keywords have been used in user keywords: “Click metering point search button”, “Select exact search checkbox”, “Unselect exact search checkbox” and “Search result list contains value”. The rest of the keywords comes from Built-in libraries: Collections, String and BuiltIn.

These four keywords are written in Python programming language and use functions defined in FlexSeleniumLibrary with help of previously initialized driver.

The code snippet below shows the functions used in the test cases as keywords. The first function “click_metering_point_search_button” performs a mouse click on a button with “searchButton” ID. In order to find ID of the element within Flex application two addons to Mozilla Firefox browser were needed: Firebug and FlashFirebug. They let to debug Flash/SWF files on the web and inspect elements. First, the application has to be loaded and SWF file found.

```
def click_metering_point_search_button(self):
    """ Performs mouse click on metering point search button """
    self.driver.click_button("searchButton")

def select_exact_search_checkbox(self):
    """ Selects Exact Search checkbox from Search panel """
    self.driver.set_checkbox_value("exactSearchCheckBox", True)

def unselect_exact_search_checkbox(self):
    """ Unselects Exact Search checkbox from Search panel """
    self.driver.set_checkbox_value("exactSearchCheckBox", False)
```

```
def search_result_list_contains_value(self, expected_address, ex-
pected_cust_name):
    """Checks if search result list contains specific value
    Args:
        expected_address: value displayed in the search result list
                 (meteringPointAddress)
        expected_cust_name: value displayed in the search result list
                 (customerName)
    Returns:
        true (if search result list contains expected values) or
                 false otherwise
"""
    row_count = int(self.get_search_results_list_row_count())
    for row_index in xrange(0, row_count):
        mp_address = self.get_search_result_list_value\
            ("meteringPointAddress", row_index)
        mp_cust_name = self.get_search_result_list_value\
            ("customerName", row_index)
        if mp_address == expected_address and \
                        mp_cust_name == expected_cust_name:
            return True
    return False


def get_search_result_list_value(self, field, row):
    """Gets the value from the given row in the field from search result list
        Args:
            field: name of the field to search
                 (e.g. "meteringPointAddress", "meteringPointName"...)
            row: index of the row in the list. Starts from 0
        Returns:
            value of the cell
    """
    return self.driver.get_data_grid_row_value\
        ("searchResultList", field, row)


def get_search_results_list_row_count(self):
    """ Gets the number of rows in search results list
        Returns:
            number of rows
    """
    return self.driver.get_data_grid_row_count("searchResultList")
```

Figure 13 shows how to find an element’s ID in SWF file. Once the SWF file is loaded, the ID of the element is visible and the element’s class name appears in square brackets.

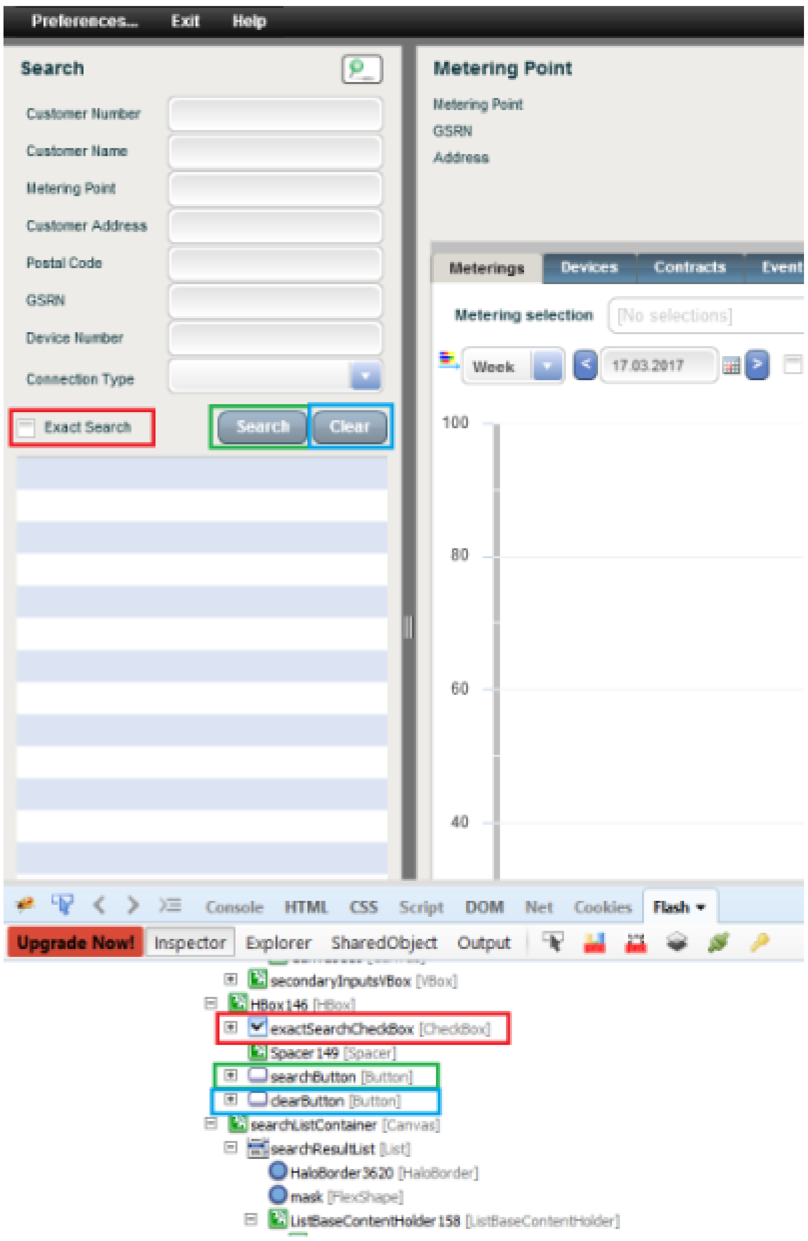


Figure 13. Inspecting elements with FlashFirebug.

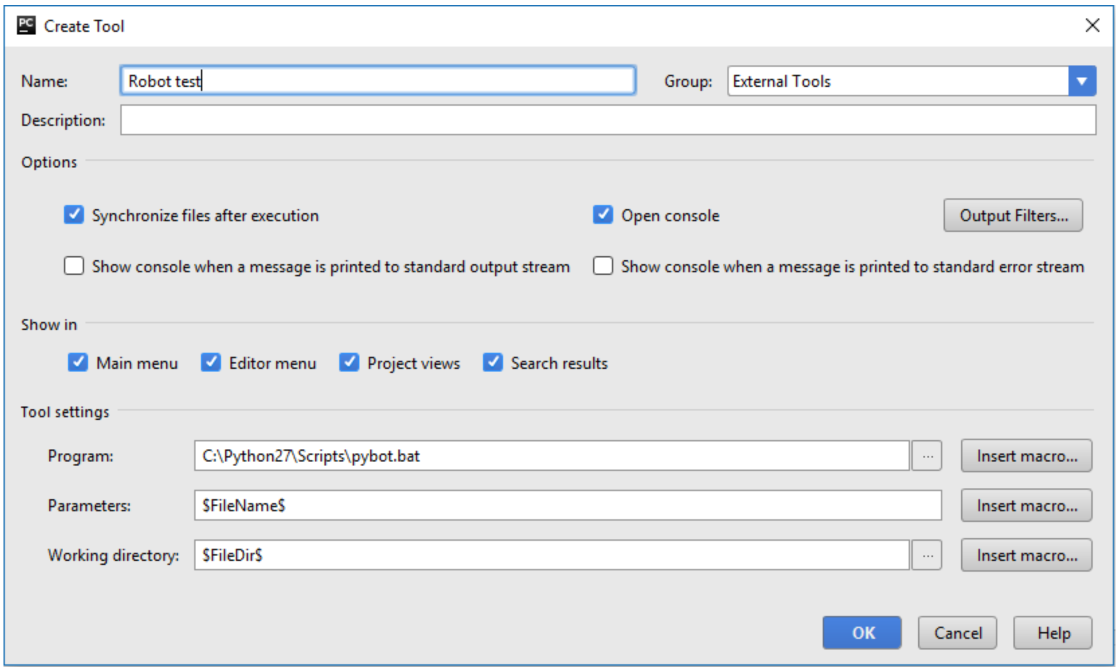
The next two functions “select_exact_search_checkbox” and “unselect_exact_search_checkbox” select or unselect checkbox below search input fields respectively.

The last function “search_result_list_contains_value” goes through the whole search result list, takes every row using “get_search_result_list_value” function and checks if the expected values passed to the function as arguments are there or not. Returns True if values are found, False otherwise.

Every function has its own documentation part that explains what the function does to make it easier to reuse it in the future. The names should also be as descriptive as possible – the same as in higher level keywords.

3.5 Running test suite

Once the test cases and keywords are done, it is time to run the test and check the results. There are a few possible ways to run the test suite. Since the whole test has been written using PyCharm IDE, one of the fastest and convenient ways was to do the following configuration (shown in Figure 14):


Create Tool

Name: Robot test
Group: External Tools
Description:

Options
- [✓] Synchronize files after execution
- [✓] Open console
- [ ] Show console when a message is printed to standard output stream
- [ ] Show console when a message is printed to standard error stream

Show in
- [✓] Main menu
- [✓] Editor menu
- [✓] Project views
- [✓] Search results

Tool settings

Program: C:\Python27\Scripts\pybot.bat
Parameters: $FileName$
Working directory: $FileDir$

Figure 14. PyCharm configuration.

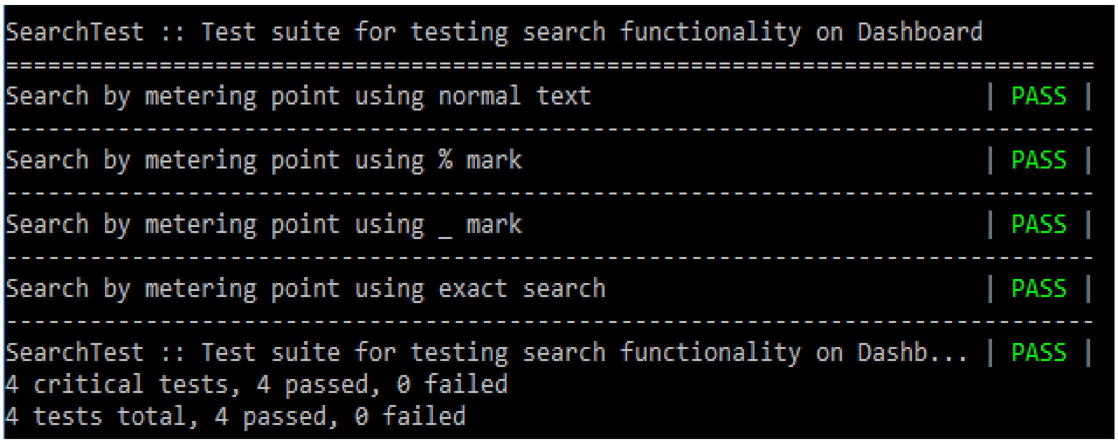
1. Go to File → Settings,
2. Choose Tools → External Tools,
3. Click on “+” sign to add a new configuration,
4. Enter some name to “Name” input field, e.g. “Robot test”,
5. In “Program” field search for path of Pybot.bat file (e.g. C:\Python27\Scripts\Pybot.bat),

6. Enter “$FileName$” to “Parameters” input field,
7. Enter “$FileDir$” to “Working directory” input field,
8. Click OK.

After the configuration is done, right click on the “SearchTest.robot” file in PyCharm. Choose External Tools → Robot test from context menu options.

3.6 Test results

The default result of the execution of Robot Framework test cases is: an XML output file and two HTML files report and log. Moreover, apart from these files the result is visible in real time in the command line when the test cases are being executed (as shown in Figure 15). When the test is run via External Tools, as shown in the previous chapter, from PyCharm IDE the real time result will be visible there.

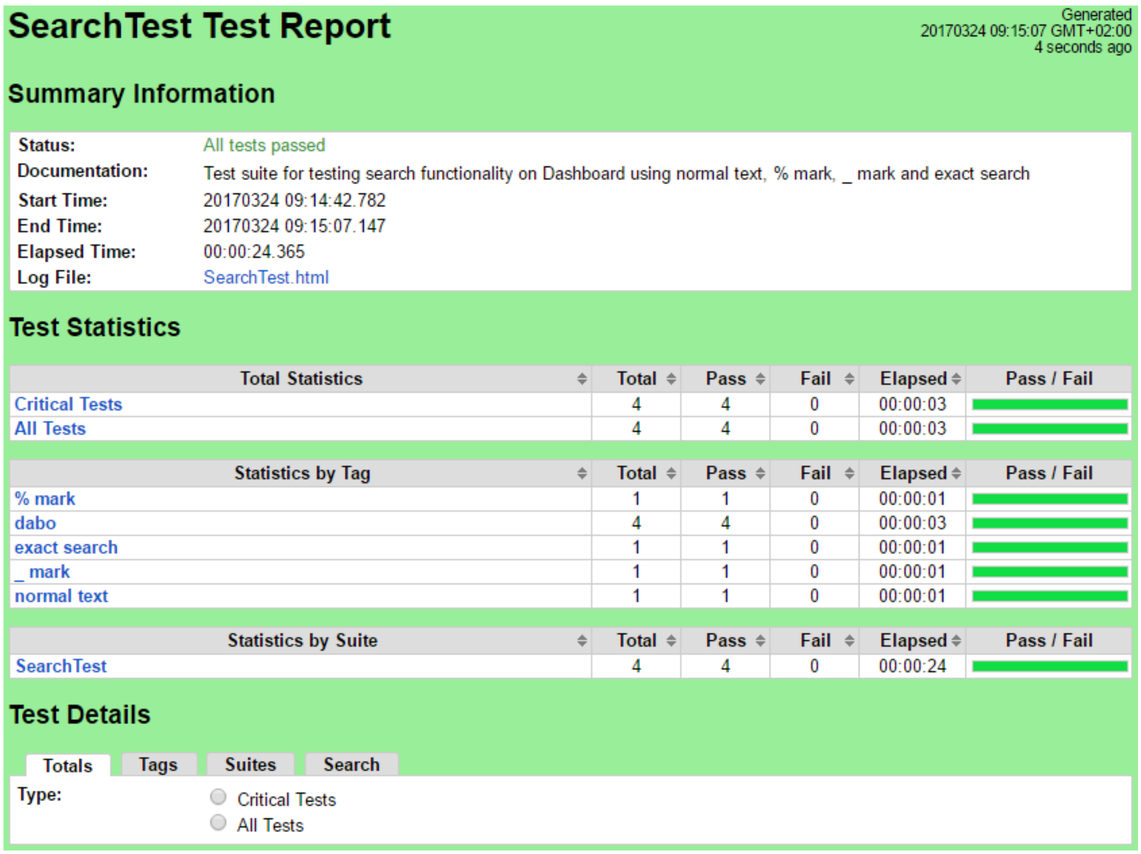


```
SearchTest :: Test suite for testing search functionality on Dashboard
==============================================================================
Search by metering point using normal text                            | PASS |
------------------------------------------------------------------------------
Search by metering point using % mark                                 | PASS |
------------------------------------------------------------------------------
Search by metering point using _ mark                                 | PASS |
------------------------------------------------------------------------------
Search by metering point using exact search                           | PASS |
------------------------------------------------------------------------------
SearchTest :: Test suite for testing search functionality on Dashb... | PASS |
4 critical tests, 4 passed, 0 failed
4 tests total, 4 passed, 0 failed
```

Figure 15. The real time result.

The first line shows the name of the test suite constructed from a file name (file extension is ignored) and a first line of the description from “Documentation” section in “Settings” table followed by the result of every test case that can be either “PASS” or “FAIL”. The last part includes a summary of the whole test suite, how many test cases passed and how many failed.

Figure 16 shows the “report.html” file. The background is green when all the test cases pass, red otherwise. The status, documentation, start time, end time, elapsed time and log file can be seen in the Summary Information part. In the Test Statistics the number of passed/failed tests is presented. Statistics by Tag are very useful when testing several of functionalities. For example, if the searching functionality using "_" mark breaks down, it will be clearly shown on the report by tag. More details are shown if needed on the Test Detail section.



SearchTest Test Report

Generated 20170324 09:15:07 GMT+02:00 4 seconds ago

Summary Information

Status:	All tests passed
Documentation:	Test suite for testing search functionality on Dashboard using normal text, % mark, _ mark and exact search
Start Time:	20170324 09:14:42.782
End Time:	20170324 09:15:07.147
Elapsed Time:	00:00:24.365
Log File:	SearchTest.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	4	4	0	00:00:03	🟩
All Tests	4	4	0	00:00:03	🟩

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
% mark	1	1	0	00:00:01	🟩
dabo	4	4	0	00:00:03	🟩
exact search	1	1	0	00:00:01	🟩
_ mark	1	1	0	00:00:01	🟩
normal text	1	1	0	00:00:01	🟩

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
SearchTest	4	4	0	00:00:24	🟩

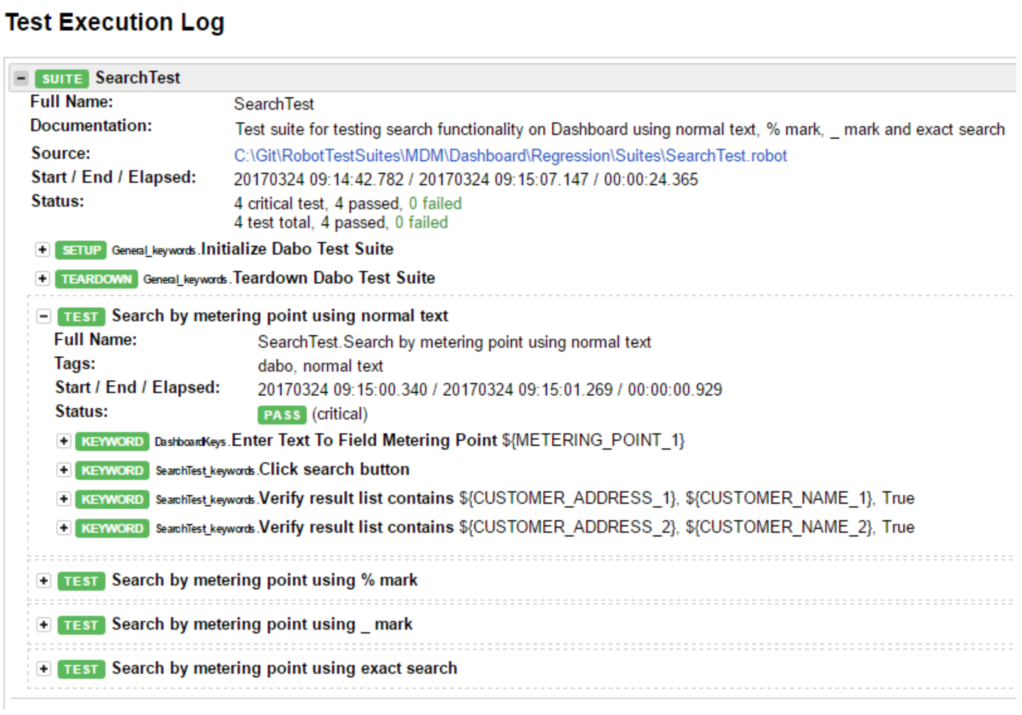
Test Details

Totals Tags Suites Search

Type: Critical Tests / All Tests

Figure 16. "report.html" file.

In "log.html" file, shown in Figure 17, the workflow of more detailed test cases is presented. Every test case can be expanded and every keyword in it can be inspected, so it is easy to know which keyword has failed in case of test case failure. Moreover, it shows where the keyword comes from – for example, Setup and Teardown come from "General_keywords.robot" file but "Verify result list contains" comes from "SearchTest_keywords.robot" file.



```
Test Execution Log

− SUITE SearchTest
  Full Name:              SearchTest
  Documentation:          Test suite for testing search functionality on Dashboard using normal text, % mark, _ mark and exact search
  Source:                 C:\Git\RobotTestSuites\MDM\Dashboard\Regression\Suites\SearchTest.robot
  Start / End / Elapsed:  20170324 09:14:42.782 / 20170324 09:15:07.147 / 00:00:24.365
  Status:                 4 critical test, 4 passed, 0 failed
                          4 test total, 4 passed, 0 failed
  + SETUP General_keywords.Initialize Dabo Test Suite
  + TEARDOWN General_keywords.Teardown Dabo Test Suite
  − TEST Search by metering point using normal text
    Full Name:              SearchTest.Search by metering point using normal text
    Tags:                   dabo, normal text
    Start / End / Elapsed:  20170324 09:15:00.340 / 20170324 09:15:01.269 / 00:00:00.929
    Status:                 PASS (critical)
    + KEYWORD DashboardKeys.Enter Text To Field Metering Point ${METERING_POINT_1}
    + KEYWORD SearchTest_keywords.Click search button
    + KEYWORD SearchTest_keywords.Verify result list contains ${CUSTOMER_ADDRESS_1}, ${CUSTOMER_NAME_1}, True
    + KEYWORD SearchTest_keywords.Verify result list contains ${CUSTOMER_ADDRESS_2}, ${CUSTOMER_NAME_2}, True
  + TEST Search by metering point using % mark
  + TEST Search by metering point using _ mark
  + TEST Search by metering point using exact search
```

Figure 17. “log.html” file.

3.7 Pushing to Gerrit Code Review

Once the test is written and running, it is time to share the code to be used for Regression Tests in the company. The whole test repository had been cloned from Gerrit before working on it. After adding/changing tests the code has to be pushed back to the Gerrit Code Review in order to be reviewed by other users before it can be merged to the main repository and be run on the main server. After the remote repository had been cloned to local repository using GIT, a new branch named according to JIRA issue name was created.

Creating a new branch for every issue to work with is a good way of managing a local repository. While working with a team, there are usually at least two team members working on the same repository and pushing their code back to Gerrit Code Review. That is why the local repository has to be up to date with the remote one before attempting a push operation. The following steps were done:

1. Commit to local repository branch,
2. Fetch changes from remote repository to local,
3. Rebase local branch that has changed,
4. Push from local branch to remote repository.

4 Results and conclusion

4.1 What has been done

The case company faced with a problem where some new features, added to one of its web based application broke the old ones. Since finding a source of the problem within the features written a few years back is very time consuming, it was decided to create the automated tests that could be utilized as the regression tests.

A regression test should cover as much application's features as possible. However, the primary goal of the thesis was to create an automated test for one of the features, and to get familiar with an entire process connected to it, that is: writing and documenting the test steps and expected results, implementing the test and preparing it to be used by the company.

The primary goal of the thesis was achieved. The theoretical part helped to familiarize with all the technologies needed for the creation process, while the practical part focused on dividing the problem into smaller parts, implementation and sharing the code. As a result, a test suite with four test cases was created for Apache Flex application.

4.2 Further development

However, now the test bases on data (metering points) that has been created manually for a particular server. The server can be used by other employees as well. Thus, it means that someone can modify or delete the data at any time, so the test will fail since it highly focuses on it.

The ideal would be to create the test data every time before a test is executed, so the test would base on exactly the same metering point every time. After the test execution, the data should be deleted, so it will not collide with other data in the future.

4.3 Conclusion

The test is running correctly and is used for regression testing in the case company. Moreover, the created detailed test cases in JIRA could be marked as closed, however, other employees can still easily browse them and see what has been done. The code written for the test has been merged to the code base, and the test is utilized in regression testing, run on a server making sure that the new functionalities do not break the old ones.

The project helped me to understand the whole process I had to go through while creating the test. I got familiar with a few tools and technologies described in the project that are popular and commonly used in other companies as well. Moreover, the thesis resulted in a beneficial solution for the company and for me. As mentioned in the theoretical part of the thesis, test automation plays a significant role in development, so I have gained the knowledge that can be utilized in my future projects for sure.

The main impediment was to work with Flex application. Nowadays, most of the web applications are created using HTML technology instead, and Flex is no longer as popular as it used to be a few years ago. Thus, no new tools and libraries are created for Flex applications, and the already-made tools are no longer supported and developed.

References

About Apache Flex. Page on Apache website. Accessed on 10.03.2017. Retrieved from http://flex.apache.org/about-whatis.html

Apache Flex. Page on Wikipedia. Accessed on 10.03.2017. Retrieved from https://en.wikipedia.org/wiki/Apache_Flex

About MXML. Page on Adobe website. Accessed on 10.03.2017. Retrieved from http://help.adobe.com/en_US/flex/using/WS2db454920e96a9e51e63e3d11c0bf5f39f-7fff.html

About TortoiseGit. Accessed on 08.04.2017. Retrieved from https://tortoisegit.org/about/

About Version Control. Page on Git website. Accessed on 18.03.2017. Retrieved from https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

Firebug. Page on Mozilla website. Accessed on 12.03.2017. Retrieved from https://addons.mozilla.org/en-us/firefox/addon/firebug/?src=dp-dl-dependencies

FlashFirebug. Page on Mozilla website. Accessed on 12.03.2017. Retrieved from https://addons.mozilla.org/en-us/firefox/addon/flashfirebug/

FlexSeleniumLibrary. Accessed on 15.03.2017. Retrieved from https://github.com/hirsivaja/FlexSeleniumLibrary

Gerrit Code Review. Accessed on 05.04.2017. Retrieved from https://gerrit-documentation.storage.googleapis.com/Documentation/2.13.7/intro-quick.html

Git Basics. Page on Git website. Accessed on 20.03.2017. Retrieved from https://git-scm.com/book/en/v2/Getting-Started-Git-Basics

Gridstream AIM. Page on Landis+Gyr website. Accessed on 10.03.2017. Retrieved from http://www.landisgyr.eu/product/gridstream-aim/

IntelliBot. Accessed on 17.03.2017. Retrieved from https://plugins.jetbrains.com/plugin/7386-intellibot

JIRA Software Features. Page on Atlassian website. Accessed on 20.03.2017. Retrieved from https://www.atlassian.com/software/jira/features

Learning ActionScript3. Page on Adobe website. Accessed on 10.03.2017. Retrieved from http://www.adobe.com/devnet/actionscript/learning.html

PyCharm. Accessed on 17.03.2017. Retrieved from https://www.jetbrains.com/pycharm/

Python Overview. Accessed on 17.03.2017. Retrieved from https://wiki.python.org/moin/BeginnersGuide/Overview

Robot Framework. Accessed on 17.03.2017. Retrieved from https://pypi.python.org/pypi/robotframework

Robot Framework User Guide. Accessed on 18.03.2017. Retrieved from http://robotframework.org/robotframework/latest/RobotFrameworkUser-Guide.html

Selenium Flex API. Accessed on 15.03.2017. Retrieved from https://code.google.com/archive/p/sfapi

Software Testing. Accessed on 04.03.2017. Retrieved from https://users.ece.cmu.edu/~koopman/des_s99/sw_testing/

TortoiseGit screenshots. Page on TortoiseGit website. Accessed on 08.04.2017. Retrieved from https://tortoisegit.org/about/screenshots/

Using a Goal-driven Approach to Generate Test Cases for GUIs. Accessed on 10.03.2017. Retrieved from https://www.cs.purdue.edu/homes/xyzhang/fall07/Papers/test-gui.pdf

What is Regression Testing. Accessed on 04.03.2017. Retrieved from https://smartbear.com/learn/automated-testing/what-is-regression-testing/?q=reg-ression

What is Scrum. Accessed on 15.03.2017. Retrieved from https://www.scrumalliance.org/why-scrum/

Why Landis+Gyr. Page on Landis+Gyr website. Accessed on 10.03.2017. Retrieved from http://www.landisgyr.com/about/landisgyr/

Why Test Automation. Accessed on 05.03.2017. Retrieved from https://smartbear.com/learn/automated-testing/