Piyusha Saran

Enhanced Test Automation of Insurance Claim Product

Metropolia University of Applied Sciences

Master of Engineering

Information Technology

Master’s Thesis

16 September 2022

Author Title Number of Pages Date	Piyusha Saran Enhanced Test Automation of Insurance Claim Product 63 pages (2 Appendices) 16 September 2022
Degree	Master of Engineering
Degree Programme	Information Technology
Instructor(s)	Sami Sainio, Lecturer, Metropolia UAS

Software Testing is an important phase of a software development life cycle. It is necessary to thoroughly test the product before releasing it to the customer. There are two different ways to carry out software testing manual testing or automation testing.

Automation testing comprises of a test automation framework, which is basically a collection of software components to execute test cases and an all-inclusive reporting of test results. There are several test automation tools available in the market like Selenium, Robot Framework, KatalonStudio, Cucumber.

The test automation framework of insurance claim handling product in the case company had a very complicated design and did not follow the market standards. Due to complicated design the tests were difficult to understand and maintain which led to the collection of obsolete tests. Hence, these stale tests resulted in high percentage of test failures in test execution and overall unreliable test reports. Therefore, a major overhaul was required to create or adopt a reliable test automation framework and hence regain the trust in the automated testing of the insurance product.

The objective of this thesis was to implement an improved test automation framework in the case company for the case software product by using latest Robot Framework libraries for example ‘Browser Library’ powered by Microsoft Playwright. Page Object Model approach was adopted, to improve maintainability and understandability of the test framework. Parallel test executions and independent test cases writing were adopted in order to reduce test execution duration. Integrations with build and test management tools were also implemented as a part of this exercise which was not possible in the old automation suite.

As a result of this study, the automation framework was improved by using new tools and new libraries, thereby improving the overall quality of test automation for the case company. Due to the new implementation, the execution time was also significantly reduced, and generated reports were used effectively to troubleshoot the problem areas and improve the feedback loop for the software developers in the case company.

Keywords	Automation Testing, Robot Framework, Browser Library, Functional Testing, Python, Jenkins.

Contents

List of Abbreviations

iOS	Operating system developed by Apple Inc.
App	Mobile Application
URL	Uniform Resource Locator
SUT	Software/System Under Test
AUT	Application Under Test
OS	Operating System
CI/CD	Continuous Integration Continuous Development
PM	Product Manager
API	Application Programming Interface
env	environment files extension
cmd	command line
QA	Quality Assurance
Python	High level, interpreted, general purpose programming language
Pycharm	Pycharm editor for writing automated tests
ASCII	American Standard Code for Information Interchange
IDE	Integrated Development Environment
JS	javascript

1 Introduction

Software testing is a process where a software product is tested after the development phase. Testing can be carried out manually or through automation. Manual testing is a tedious process [1] and consumes considerable time of a software tester. It takes considerable time [2] because it requires testers to perform testing manually step by step, and they are always deficient of time to thoroughly test the software, thereby software release cycles [3] are often delayed. Also, because of repeatability and human involvement manual testing is sometimes considered unreliable and error prone due to human errors. A major solution to the problem is by providing the software testers with test automation techniques.

Test automation [4] is necessary for the software companies because of its reliability and efficiency. It assists in creating a good quality software with lesser efforts [5] and significantly reduces the duration of the release cycle [6], where a product feature is released to the market after successful development and testing .

In order to have a successful automation for a product the test automation framework needs to be stable, robust, and simple enough to be understood by the software team comprising of testers and developers. A good automation framework based on best practices and guidelines improves the overall performance of the product by increasing the test coverage for the product. Test coverage determines how much you are testing and what you are testing specifically [7]. The test framework structure should be organised in such a way, that the different areas within the product should be clearly categorised, in order to separate different functional areas within the product like login, user accounts creation, case creation.

The case company had an insurance claim handling product, with the web portal and the mobile app. It had an old automation suite which was used to test the software product. The main goal of the case company was to have a test automation framework which was reliable, scalable and was easy to understand by the testers and developers. But the old automation suite in the case company had many flaws like complicated structure, hardcoded data usage, limited test coverage, very long test execution time, unparalleled execution. Due to these reasons, there were many problems with the old automation suite and the overall test automation of the insurance claim handling product became unreliable.

Thus, the aim of this methodological study was to identify the shortcomings in the old automation suite of the case company and implement a solution to overcome the identified shortcomings. The objective of this study was derived based on the discussions with software development team, management, and business representatives.

The identified and accepted solution was basically to create a new test automation framework using latest Robot Framework libraries. Selenium library was replaced with a new library by Robot Framework called ‘Browser Library’ powered by Microsoft Playwright to bring the latest features of automation testing apart from numerous other implementations to remediate the identified shortcomings of the old automation suite.

This thesis has been divided into 6 sections.
The first section explains the business context, business challenge and objective.

The second section details the theoretical concepts of automation and information about the fundamentals of automation testing. Also, the different types of test automation framework are described in this section.

The third section covers the project specification, where the current state analysis is done to highlight the situation with the old automation suite. It gives details about the research process used as well as the data collection methods used during this thesis. Also, the fourth section gives more information about the old automation suite, like the whole setup, and then it enlists the major drawbacks in the old automation suite.

The fifth section however explains the implementation procedure of new test automation framework, the high-level needs, the necessary setup, and configuration.

The sixth section compares the old automation suite vs new framework for comparisons and results

1.1 Business Context

This thesis was done for an insurance claim handling product company which has SaaS web portal and mobile application. The case company is one of leading provider of building claims in Nordics. The case company follows agile development style for product delivery. They developed a product which makes case for the corresponding damage happened at the customer site and after successful inspection, the details and cost estimates were sent to the insurance companies for claim.

1.2 Business Challenge

Manual testing slows down the continuity of the agile delivery flow. Therefore, product quality suffers as the development and testing of new features falls behind schedule.
Agile development is basically a school of thought where its decided on how to develop and hand over the product whereas DevOps defines the method to continuously roll out the software code by utilizing different automation tools and software. Automation is considered as the wheels of the DevOps(a combination of Development and Operations); an ineffective automation could lead to a struggling development process which also means slower deliveries and unsatisfied customers.

There are broadly two platforms of the insurance claim handling product core (web) and mobile platform. The web portal of the case company was used mainly to create case as per the damage at the customer site.

The main challenge for the case company was unreliable test automation, it had an old automation suite which had many flaws in it. The old automation suite was very complicated and extremely hard to maintain and therefore overtime due to less maintenance it lead to stale tests and eventually high number of errors.

It also means that most of the testing was actually performed manually to gain confidence in the releases but as mentioned earlier, manual testing was slowing down the release cycle significantly.
In other words, there were two chronic problems with the old automation suite:

1) Insufficiency: because testing of the product is performed manually which is a slow and tedious process resulting in higher TTM/time to market.
2) Inefficiency: the test automation is complicated and has very high failure rate.

1.3 Objective

The main objective of this thesis is to understand the flaws of the old automation suite and then implement a new framework which includes improvements and better design along with guidelines. The biggest issue with old automation suite was its complexity since it was written by a developer(s) by using Python [8] language heavily without following the test automation framework best practices, it lead to a situation that the tests were very complicated to maintain and even to understand.

Eventually, the framework started to collect obsolete tests due to less maintenance by the software team over time. The stale tests further led to huge number of failures in the test execution. It was practically impossible to debug the test results as the old automation suite was complicated and hard to understand.

The test coverage was also insufficient because it was not possible to write the automated tests for all functionalities due to competence limitations of python language in test team for the old automation suite. It was impacting the product quality over time specifically in integration areas. When new integration feature(s) were added to the product most of the testing was being carried out manually which was very time consuming. Due to time pressure, the testing was performed poorly or not completed within release cycle, thereby bringing down the overall quality of the product.

With the help of this study, old automation suite will be improved in the most efficient way as per coding guidelines so that the tests are structured and modularized. The new automation framework will be implemented following the Page Object Model approach [9], where the folders will be named as per the individual pages in the product. This approach gives the framework better readability and reusability of the methods developed. Tagging the tests will also be used majorly in the new automation framework to improve the searchability of the tests and to identify the test scope and coverage.

Overall, the main improvement from the old test automation framework was the implementation of Browser Library for Robot Framework[10] which helped in making the tests less fragile, improved the test coverage and have faster execution times to generate reports.

2 Fundamentals of Automation Testing

There are mainly two ways of testing a software product manual testing or automation testing. There are certain differences between the approach of these testing methods. Manual testing involves a software tester verifying all the functional areas of the product manually but to release the product faster, often test coverage of manual regression testing is compromised which leads to poor quality of the product. The reliability of a product increases if the testing has been done thoroughly. This work is prone to errors because of poor coverage, repeatability, and human errors when done manually. Automation testing on the other side is done using automated tools and test script to test a software product. Automated testing reduces the testing time significantly.

The different types of tools used for automation have certain advantages to be used for a certain product [11]. It is basically running the tests automatically, arranging and using the test data and analysing the results for achieving enhanced quality of the software product [12]. These automated scripts follow Standard guidelines to write code to effectively execute the important functionalities within a product. Due to the increased time spent in testing, by manual as well as automation the overall test coverage is improved greatly. This chapter includes the background to define the need of automation testing for a software product. It covers the importance of having automation apart from other types of testing and their definition.

2.1 Types of Testing in Brief

There are mainly two different types of software testing [13]:

- Manual Testing
  It means testing any software product manually without utilising any additional tool(s).
- Automation Testing
  It means testing any software product by executing the written test scripts with an automation tool.

Based on the scope of testing, it can be further classified as

- Smoke/Sanity Testing
  It means testing only the major features of a product to check if it works properly.

- Regression Testing
  It means testing the areas where a defect or bug has been fixed to check if the changes in the code did not break anything else in the total code.
- End to End Testing
  It means testing all the functionalities within the product to verify the whole software product.

2.2 Why is Test Automation needed?

Manual testing is not sufficient for any software because human errors are unavoidable during execution and the repetitive tasks decrease the productivity of a person, thereby it's often seen that automation is a much more reliable and efficient way of testing [14]. The main reason behind automation testing is that it reduces cost, decreases the time and efforts and is more reliable to schedule at desired time intervals. Initial introduction of automation involves more effort and time as compared to manual testing, therefore support from management is crucial.

Due to complexity of the product some functionalities in the insurance claims product were kept untested or poorly tested, thereby resulting in the large number of errors in that area. There were several production issues in those areas because of poor test coverage, therefore automation testing is beneficial because the complex areas can be covered thoroughly and any issues occurring can be analysed for the possible causes easily.

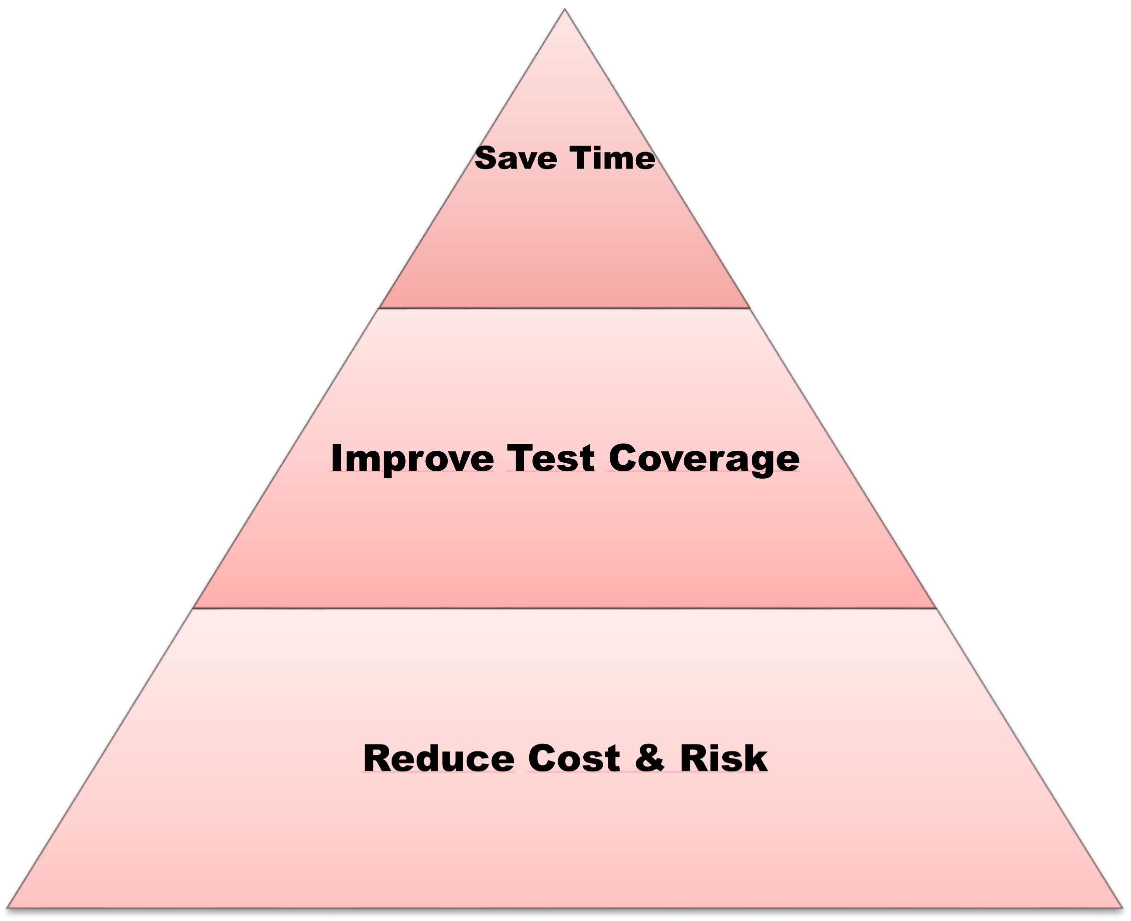


Figure 2.1 Automated Software Testing Pyramid

The above Figure 2.1 shows the automated software testing pyramid. The pyramid shows the benefits of automation testing for an organization, It is evident from the pyramid that test automation easily saves time of execution by running the scripts automatically rather than manually thereby saving time.

It improves the test coverage by adding automated tests for areas which are complex and by running them repeatedly the errors can be removed faster. It ultimately reduces the cost and risks associated with failures because those analysis results can then be used by the developers to identify the root cause behind the failures and thereby decreasing the TTM of new features of the product. Also, the overall quality of the product is also improved.

2.3 Concept of Automation Testing

Automation is an overall process for the system or product to perform in a predefined manner. The initial cost of test automation is quite high because lot of time is devoted in the tool selection, framework finalisation, programming language selection and further resources need to be planned for carrying out automation testing. There might be few times conditions where the existing resources need extra training to learn automation testing. During these stages, lot of time is spent in planning and execution.

After successful training of the resources, automation scripts are written, and framework structure is organized. The initial scripts, when written are executed multiple number of times to confirm the accuracy and the coverage of the feature for the AUT. Creation, execution, and maintenance of the automated test scripts, test environment is a challenging task and must be done periodically to implement the latest improvements in the product. All new changes in the product must be refactored in the scripts so that the scripts do not fail due to new changes in the software.

The major benefit of test automation is the cost, in addition to the cost there are several advantages of test automation as follows:

- Deeper test coverage of the AUT
- Reliable results of test execution
- Economizes Time and Cost
- Accuracy is improved.
- Lesser errors due to Efficiency
- Faster execution speed
- Re-usable test scripts
- Early TTM

The criterion for selecting the test suites for test automation are following:

1. Business critical areas of the product
2. Workflows that are repetitive
3. Workflows that are difficult to be performed manually.
4. Workflows that are time consuming.

The overall process of test automation comprises of test tool selection, scope definition, planning, test execution, reports analysis and maintenance of the test suites. The different types of tools used for automation have certain advantages to be used for a certain product.

2.4 Test Automation Process

Automation testing is a step-by-step process where an automation tool executes the test suites on certain functionality or features of the AUT, without any manual involvement. After the careful selection of the tool, the next step is to organise the automation framework in order to categorize the different areas of the product so as to keep the necessary resources and keywords at the right place.

The process initiates with firstly analysing the benefits of having about the need of automation, selecting the effective automation tool, developing the test automation framework, creation and designing the test suites, execution of the test suites, report analysis and lastly maintenance of the test suites.

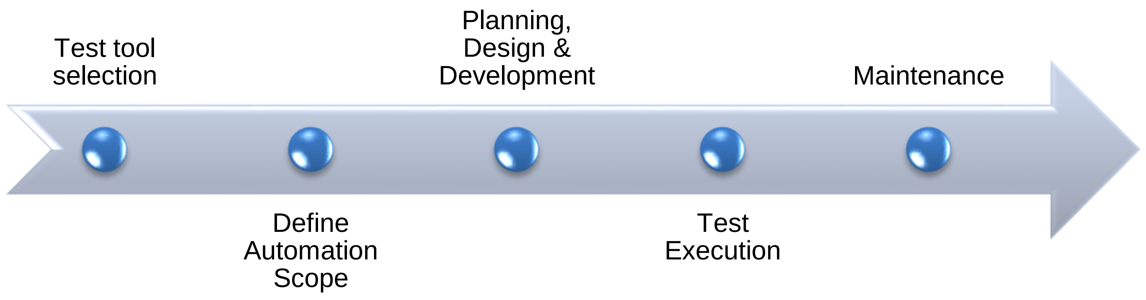


Figure 2.3 Automation Test Process

The above figure 2.3 shows how the test automation process goes in sequential form from the test tool selection to maintenance of test automation scripts. In the process we can see that it starts from the test tool to be selected depending on the feasibility and the complexity of the product to be automated. After the tool selection, comes the definition of automation scope to note down the areas to be covered for test automation. The tool selection depends majorly on which technology the AUT is running.

After selecting a tool later, a proper framework is required, which will be detailed in the next chapter. After careful framework selection then building proof of concept (POC) with end-to-end scenarios to evaluate if the tool can support automation of applications developed. The best approach to follow is to create clear folders where the necessary pages, resources and tests can be stored. In order to make the framework simpler, names should be kept as close as to the pages of the AUT, so that all the testers can quickly understand the area of the automated tests being written.

After building the POC, framework development is carried out, which is a very important step for the success of any test automation project. Framework should be designed after careful analysis of the technology used by the application also its key features. The automated scripts are developed and executed, executed results are analysed and defects are logged if any. Test scripts are maintained in version control and need to be updated according to the changes in the AUT.

2.5 Types of Automation frameworks

Testing frameworks are a very essential part of test automation. Basically, a testing framework is basically a set of rules or guidelines defined for designing the test cases. All these guidelines follow a standard practice and must be implemented always while creating a test case. It helps to ameliorate the accuracy of tests, significantly reduces the maintenance costs for the test and thereby decrease the overall cost of the project.

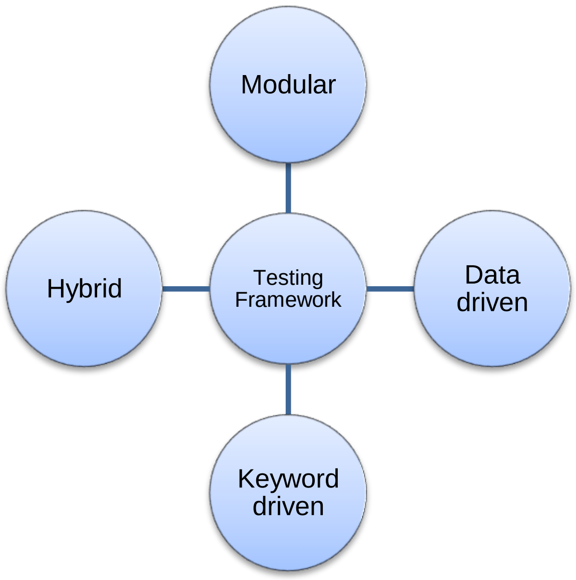


Figure 2.4 Testing framework types

The different types of Testing frameworks are shown in figure 2.4. All these different types of frameworks have their own advantages, disadvantages as well as structural architecture. The details about these are provided in the next section.

2.5.1 Data driven framework.

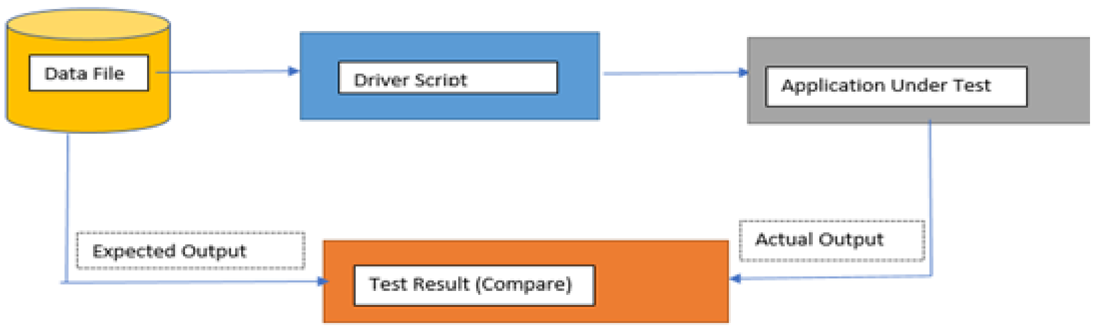


Figure 2.4.1 Data driven framework

The above figure 2.4.1 for data driven framework allows the test data to be separated from the test logic, so the data can be stored externally.
The data can be read from the external files like csv, excel, text files etc. and can be infused into variables in the test scripts.

The biggest and foremost advantage of using this approach is that any changes to the test scripts do not affect the test data. Due to this approach, it is possible to refactor the automated tests easily by just changing the data at a central place which is much quicker rather than hardcoding the data in all the automated tests. If the data is hardcoded, then during the maintenance phase it becomes a tedious work to update the tests with new data at every single place. It is very easy to miss out certain places because the tracking of hard coded data is very difficult and can be complicated at times. Also, sometimes if the values are changed then its next to impossible to remember all different places where the data could have been used to correct it.

2.5.2 Keyword driven framework

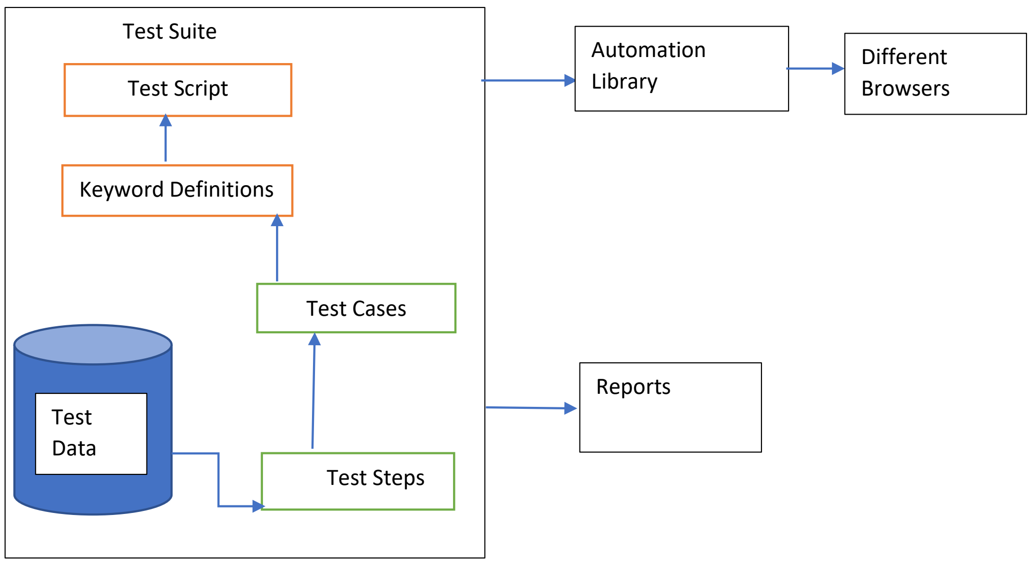


Figure 2.4.2 Keyword driven framework

As per the above figure 2.4.2 this framework requires the development of keywords and tabular data. From the above figure 2.4.2 we can see that the test data is used in the test steps. The test steps are basically the building blocks of workflow to be followed in a test case to perform certain action.

These test cases use keyword/or commonly known as user actions to do some specific tasks. With the help of the keywords, test script is organised to cover a unique functional area of the product for example, Login and then collection of these kind of test scripts are basically overall known as test suite.

This type of framework is easier for maintenance, because if there are any future changes made for the SUT then only the changes needed will have to be done in the keyword documentation. Its syntax basically lists test cases (data and action) in a table format. Also, this methodology supports reusability to a larger extent because the same keyword can be reused for different test scripts, thus saving time. This approach is the one which will be used as a basis for our newly implemented automation framework. This approach is quite easy to follow and is very modular. The benefit of this approach is that it is possible to create a keyword which is basically user action to perform a task for example, Select Inspection Assignment, this keyword clearly tells us that Inspection Assignment must be selected to a particular place from a list of options available.

2.5.3 Hybrid testing framework

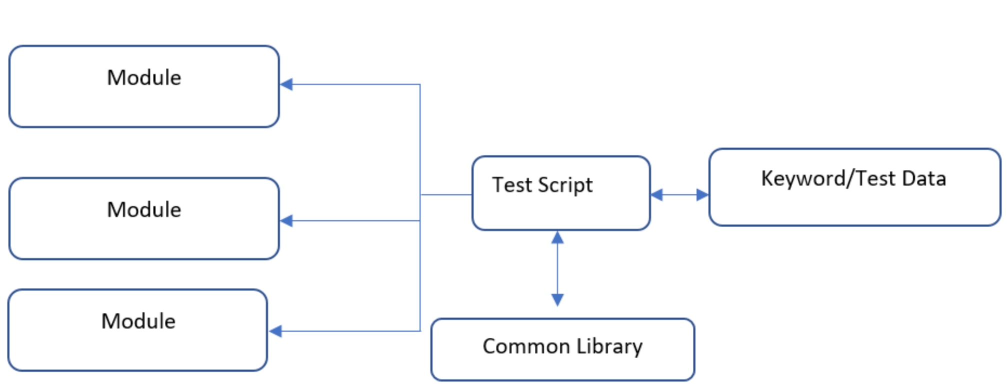


Figure 2.4.3 Hybrid Testing Framework

As stated from the figure 2.4.3 hybrid means a combination of data driven and keyword driven framework approach. We can see from the figure that several modules make pup a test script. These test scripts use a common library, which has a collection of already written keywords in it to do some action. These test scripts can also have custom made keywords which utilise test data for the test scripts. In this approach, the keywords as well as the test data are both externalized. Keywords and test data both are maintained individually in separate files for example, former is maintained in separate java class file while latter is maintained in excel file/ properties file.

2.5.4 Modular based testing framework

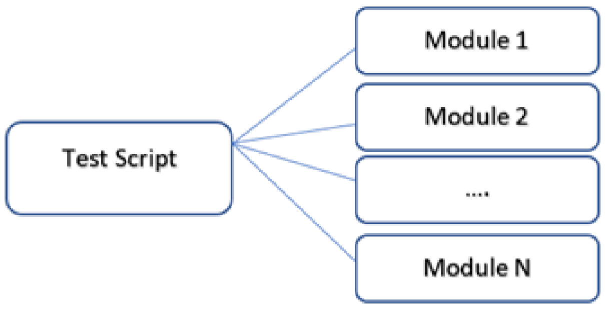


Figure 2.4.4 Modular Testing Framework

As stated from the figure 2.4.4, this framework breaks the test cases into smaller modules. It follows a non-incremental and incremental approach. The modules are tested in smaller groups and then tested as a whole. It is a reusable, scalable and efficient approach but a little complicated approach.

2.5.5 Linear Scripting

This approach is straightforward and is kind of an elementary level of testing where the test scripts are written consecutively and are run singly. It does not need custom code to be written and is simple enough to understand by the automation testers. It is based on the approach where one functionality is picked at a time, write the code, and test it. But it needs high maintenance as the scripts need to be maintained for every single change happening.

2.5.6 Test library architecture framework

In this kind of framework similarities among the test scripts are identified to be grouped under similar functions. Therefore, the library collects all the sorted functions, thereby making it highly reusable. This kind of approach is manly beneficial when the system has similar kind of functionalities across various parts of the application.

2.6 Automation Tools & Frameworks Details

There are various automation tools available in the software market and it depends upon the requirements of the company on which tool to use for their AUT. The simplest available automation tools widely used nowadays are Robot Framework with different languages like Python, Java [15] and supported by different libraries or plugins.

2.6.1 Robot Framework

Robot Framework is a test automation framework used mainly for user acceptance testing and thereby acceptance test-driven development. Robot Framework uses a keyword-driven testing framework which uses tabular test data to have a variety of test data sets to verify the boundaries of the software.

In the keyword-driven testing framework tabular test data is used for a variety of test data sets to verify the boundaries of the software. Robot Framework has built-in keywords, which are imported as library to be used by the test cases for common functions in a web page like Log, Run keyword etc. It is possible to write user-defined keywords as well which is compounding of other user defined keywords or built-in keywords. It is also possible to give arguments to these keywords which then create the user-defined keywords like functions, which can be reused later.

In the data-driven approach of Robot Framework it supports the keyword driven style and the data driven style. It basically uses the high-level keywords which are then implemented as a template to the test suite and the tests then utilize to share data with the high-level keywords defined in the template. This kind of approach helps to have a variety of data and therefore the tests can be very beneficial in terms of applications where different kind of data is expected to be used.

Behavior driven approach of Robot Framework, which is basically an extension of test-driven development, is an approach where tests are primarily based on the behavior of the system. This approach explains different methods to nurture a feature based on the behavior of the feature. Majorly, the Given-When-Then approach is used for designing the tests

2.6.2 Python

Python is a very popular high level programming language which uses object-oriented approach to aid the programmers write clear, logical programs. It has its own style of writing and is easily compatible with Robot Framework. There are 2 major versions of Python available Python 2x or Python 3x. Python 2x is becoming obsolete nowadays and many tools have stopped supporting it.

Python language is suitable for the beginners in programming as well as experienced programmers. It has comparatively better indentation support and therefore the python code is easily readable. It has a massive collection of standard libraries, and these can be used for the following areas:

- Machine learning
- GUI applications
- Image processing
- Testing frameworks
- Scientific computing

2.6.3 Selenium

Selenium is an open-source project comprising of a wide range of tools and libraries to support the browser automation. It has a unique record and playback tool to help programmers record the session activity for replay of actions. It uses the element id recognition methods mainly to capture different elements on the web page to have controlled action over the browser. Tests are written in Selenese language for Selenium.

Selenium was widely used automation technique for the element locators in the webpage. It was quite successful earlier but then it started to have limitations about the recognition of web elements.

2.6.4 Robocop

Robocop is a powerful tool that executes static code analysis of Robot Framework code. It applies Robot Framework parsing API to execute checks on the existing code looking for violations for the software program to the quality standards.

2.6.5 Git and Gitlab

Git is a version control system which is free and open source, used to handle varying scale of projects with great accuracy and efficiency.
Gitlab is a platform which integrates the development, operations, and security groups under a single application.

2.6.6 Jenkins

Jenkins is a free software tool which has many plugins for Continuous Integration. It helps to build and test the software continuously to make it quite easy for the developers/ programmers to incorporate changes in the project. With the help of Jenkins, it is possible to schedule automated tests at a desired time and reports can be generated quite easily to see the execution status thereby to be analysed and investigated for failures.

2.6.7 Pycharm

Pycharm is a special Python based Integrated Development Environment (IDE) which has multiple tools for Python programmers. It has several plugins like Robot Framework for using it with Robot Framework. It is available in Community Edition (free) and Professional Edition (free trial). Pycharm is an overall IDE for creating, executing tests. It has various features to help the software professionals in programming and running the code.

2.6.8 Pabot

Pabot is parallel executor for Robot Framework and is installed by a simple command `pip install -U robotframework-pabot`, due to Pabot the execution time can be significantly reduced. With the help of Pabot parallel execution is easily possible and the total execution time can be checked.

2.6.9 TestRail

TestRail is a web-based test case management tool which helps to manage the software testing processes. With the help of TestRail, test cases and test suites can be arranged as per the functional area of the tests.

2.6.10 Allure

Allure is a Jenkins plugin to show the Robot Framework test reports in a simplified way after each test execution.

2.6.11 Browser Library

Microsoft Playwright is a node.js based library to automate Chromium, Firefox and Web kit browsers with a single API. It is much faster in execution than Selenium library because it does not need explicit wait and sleep commands, it automatically waits for the element.

3 Project Specification

This section lists the research methods used during the study and includes the details about the old automation suite within the case company. It details the process followed by the case company for then

3.1 Initial State Analysis

This section provides the overview of the old test automation in place in the case company. The major source of information in analysing the current state of the system, was interviews, peer reviews and discussions with the existing software professionals and the business stakeholders. During the research process several meetings, interviews and discussions were carried out with the business representatives and the management, in order to understand the shortcomings in the old automation suite and its impact on the management and business.

The automation test suite comprised of complex directory structure which was very hard to understand and maintain. The main programming language used was Python2 which also needed upgrade to newer version Python3. RIDE tool was used for test execution of the automated tests which had a very limited features and had performance issues. RIDE is a very basic tool with quite limited features and is not very intuitive like modern IDEs for example Pycharm.

In the case company once the developer modified the code it was being committed to git repository without review since there was no review tool or process was in place for the test automation suite. It led to poor code quality over time and sometimes defects in the test automation suite itself.

The old automation suite had many issues like unparalleled test execution, hardcoded data use, had lots of errors and warnings in the reports, complicated tests etc.

The old automation suite theoretically provided web automation as well as mobile automation but practically was not being used as part of the testing process due to its unreliability in both areas.

The state of unreliability emerged over time and due to its design and implementation shortcomings. The test suite was developed by the developer(s) and was cantered around python instead of Robot Framework and without using testing approach. Over time it became very complex and hard to manage by the test team and eventually was not being used at all in the testing process.

Since the testing was carried out manually, the elongated testing period started to impact end user in terms of slower and poor-quality production releases of the product in the case company.

After prolonged discussions and interview sessions within the team, it was agreed that a new test automation framework is needed to address all the problems of the old automation suite.

3.2 Research Process and Data Collection

There are multiple ways to classify a research such as objective, approach, procedures, and data collection. This research applies the action research methodology with the quantitative analysis (approach) of the data being analysed and arranged during the research work. Action research is initiated to solve an immediate issue and aims to bring positive change within the organization.

This section defines the research process adapted during the analysis of the problem within the case company. It basically details the activities performed to interpret the existing problems within the case company and for that several steps were taken. For this several meetings, discussions within the team were initiated and meeting notes were recorded to assist the data collection.

Following table gives details for the different types of questions discussed during the meetings with business stakeholders of the company:

S. No	Questions/discussions
1.	How much does the current framework fulfil the needs considering the coverage?
2.	Is the framework easier to follow?
3.	What is the failure rate of night tests?
4.	How well are the robot tests structured in terms of functionality?
5.	Is there anyone responsible for refactoring the tests as and when needed?
6.	What are the important functional areas not automated?
7.	How many tests are there in the framework?
8.	Which programming language was used in the designing of robot tests and why?
9.	Are the tests searchable for specific functional area?
10.	Are the currents tests easier to debug in case of failure?
11.	Are the tests complicated to write and understand?

During the discussions couple of shortcomings within the old automation suite were identified, like complicated tests, high number of failures, unreadable reports, legacy scripts etc.

Another major flaw in the old automation suite was that it was written with Python language and used Python libraries and keywords, which made the overall test automation framework quite complicated for the test engineers to understand.

Therefore, after several discussions with the QA Manager and PM, certain conclusions were drawn to implement a new test framework which will be addressing all the major problems identified in the old framework.

4 Old Framework Setup and details

This section will give detailed information about the old automation suite setup and will further emphasize the major flaws identified within the existing framework.

4.1 Setup Description

This section describes the old automation suite setup details. In this framework there were 2 major sections mobile and web. Both these sections have their relevant automated tests within the product.

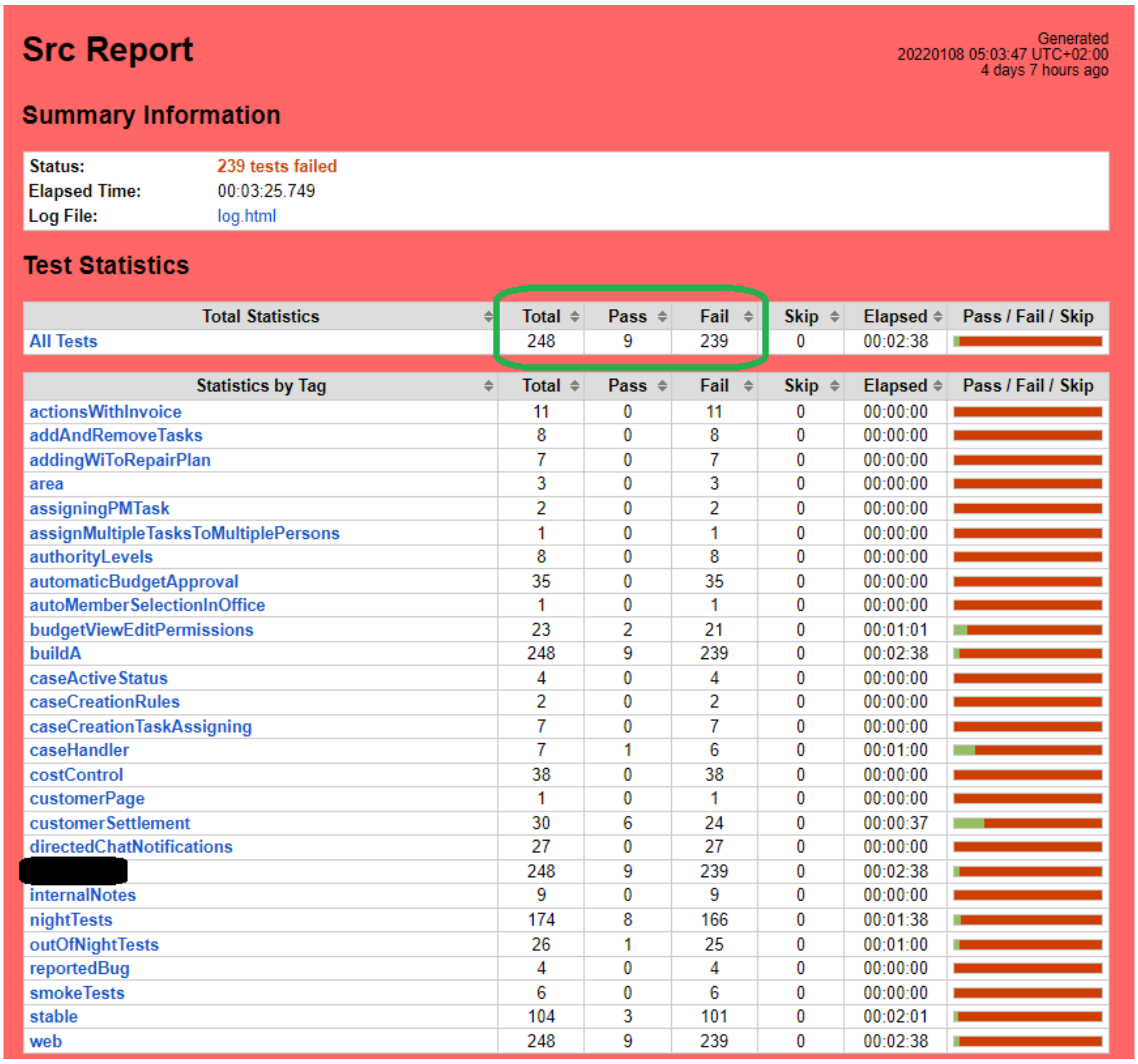
The mobile section was divided into 3 different platforms: Android, iOS and windows. The case company has a mobile app in all the 3 different OS(s). The coverage of the mobile automation was very limited because of unavailability of resources. Appium is used for the mobile automation because it supports automating native, mobile web, and hybrid applications on iOS mobile, Android mobile, and Windows desktop platforms.

Web section was also divided into different folders like main test scripts, resource folder for common keywords, documentation, and logs. All the automated tests were written in Python language and used Python libraries. Pycharm was used for writing the automated tests. The style of writing the old test was purely Python and the tests were written using custom keywords of Python.

4.2 Major flaws in the old framework setup

This section aims out the major flaws in the existing framework setup. Appendix 2 details out the main reasons for switching from old framework to the new one. Whereas some of the major problems with the old test automation framework have been detailed out below.

4.2.1 High Failure Percentage



Src Report

Generated
20220108 05:03:47 UTC+02:00
4 days 7 hours ago

Summary Information

Status:	239 tests failed
Elapsed Time:	00:03:25.749
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	248	9	239	0	00:02:38	🟩🟥

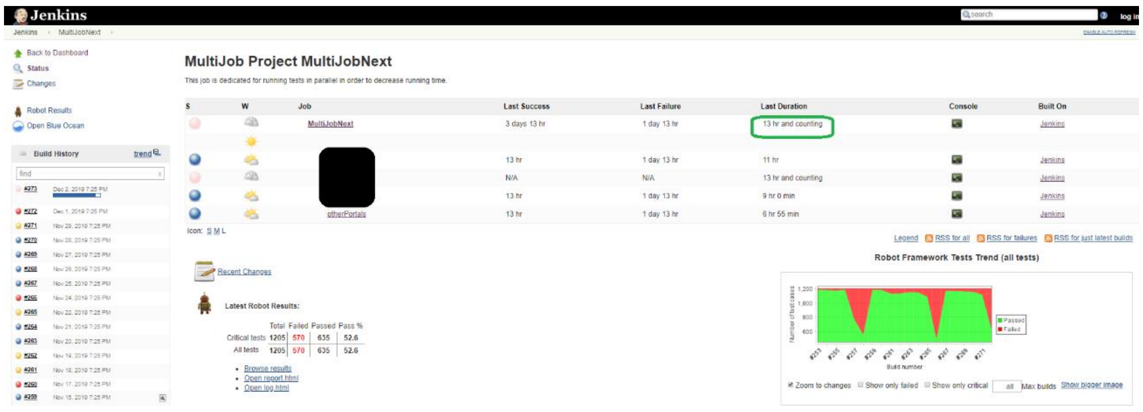
Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
actionsWithInvoice	11	0	11	0	00:00:00	🟥
addAndRemoveTasks	8	0	8	0	00:00:00	🟥
addingWiToRepairPlan	7	0	7	0	00:00:00	🟥
area	3	0	3	0	00:00:00	🟥
assigningPMTask	2	0	2	0	00:00:00	🟥
assignMultipleTasksToMultiplePersons	1	0	1	0	00:00:00	🟥
authorityLevels	8	0	8	0	00:00:00	🟥
automaticBudgetApproval	35	0	35	0	00:00:00	🟥
autoMemberSelectionInOffice	1	0	1	0	00:00:00	🟥
budgetViewEditPermissions	23	2	21	0	00:01:01	🟩🟥
buildA	248	9	239	0	00:02:38	🟩🟥
caseActiveStatus	4	0	4	0	00:00:00	🟥
caseCreationRules	2	0	2	0	00:00:00	🟥
caseCreationTaskAssigning	7	0	7	0	00:00:00	🟥
caseHandler	7	1	6	0	00:01:00	🟩🟥
costControl	38	0	38	0	00:00:00	🟥
customerPage	1	0	1	0	00:00:00	🟥
customerSettlement	30	6	24	0	00:00:37	🟩🟥
directedChatNotifications	27	0	27	0	00:00:00	🟥
[redacted]	248	9	239	0	00:02:38	🟩🟥
internalNotes	9	0	9	0	00:00:00	🟥
nightTests	174	8	166	0	00:01:38	🟩🟥
outOfNightTests	26	1	25	0	00:01:00	🟩🟥
reportedBug	4	0	4	0	00:00:00	🟥
smokeTests	6	0	6	0	00:00:00	🟥
stable	104	3	101	0	00:02:01	🟩🟥
web	248	9	239	0	00:02:38	🟩🟥

Figure 4.2.1 Jenkins failure report

As shown from figure 4.2.1 high number of test failure was observed every time the night tests were executed approximately 239 tests failed out of total 248 tests. The high failure percentage (96.3%) shows the instability of tests and insufficient information for the causes of the failures.

4.2.2 Prolonged Execution Time

Unparalleled execution while running the night tests in the old framework, causing long duration of execution times.



Jenkins

search | log in

Jenkins › MultiJobNext ›

ENABLE AUTO REFRESH

- Back to Dashboard
- Status
- Changes
- Robot Results
- Open Blue Ocean

Build History — trend

find

#273	Dec 2, 2019 7:25 PM
#272	Dec 1, 2019 7:25 PM
#271	Nov 29, 2019 7:25 PM
#270	Nov 28, 2019 7:25 PM
#269	Nov 27, 2019 7:25 PM
#268	Nov 26, 2019 7:25 PM
#267	Nov 25, 2019 7:25 PM
#266	Nov 24, 2019 7:25 PM
#265	Nov 22, 2019 7:25 PM
#264	Nov 21, 2019 7:25 PM
#263	Nov 20, 2019 7:25 PM
#262	Nov 19, 2019 7:25 PM
#261	Nov 18, 2019 7:25 PM
#260	Nov 17, 2019 7:25 PM
#259	Nov 16, 2019 7:25 PM

MultiJob Project MultiJobNext

This job is dedicated for running tests in parallel in order to decrease running time.

S	W	Job	Last Success	Last Failure	Last Duration	Console	Built On
		MultiJobNext	3 days 13 hr	1 day 13 hr	13 hr and counting		Jenkins
		[redacted]					
		[redacted]	13 hr	1 day 13 hr	11 hr		Jenkins
		[redacted]	N/A	N/A	13 hr and counting		Jenkins
		[redacted]	13 hr	1 day 13 hr	9 hr 0 min		Jenkins
		otherPortals	13 hr	1 day 13 hr	6 hr 55 min		Jenkins

Icon: S M L

Legend | RSS for all | RSS for failures | RSS for just latest builds

Recent Changes

Latest Robot Results:

	Total	Failed	Passed	Pass %
Critical tests	1205	570	635	52.6
All tests	1205	570	635	52.6

- Browse results
- Open report html
- Open log html

Robot Framework Tests Trend (all tests)

[✓] Zoom to changes ☐ Show only failed ☐ Show only critical all Max builds Show bigger image

Figure 4.2.2 Jenkins Pipeline

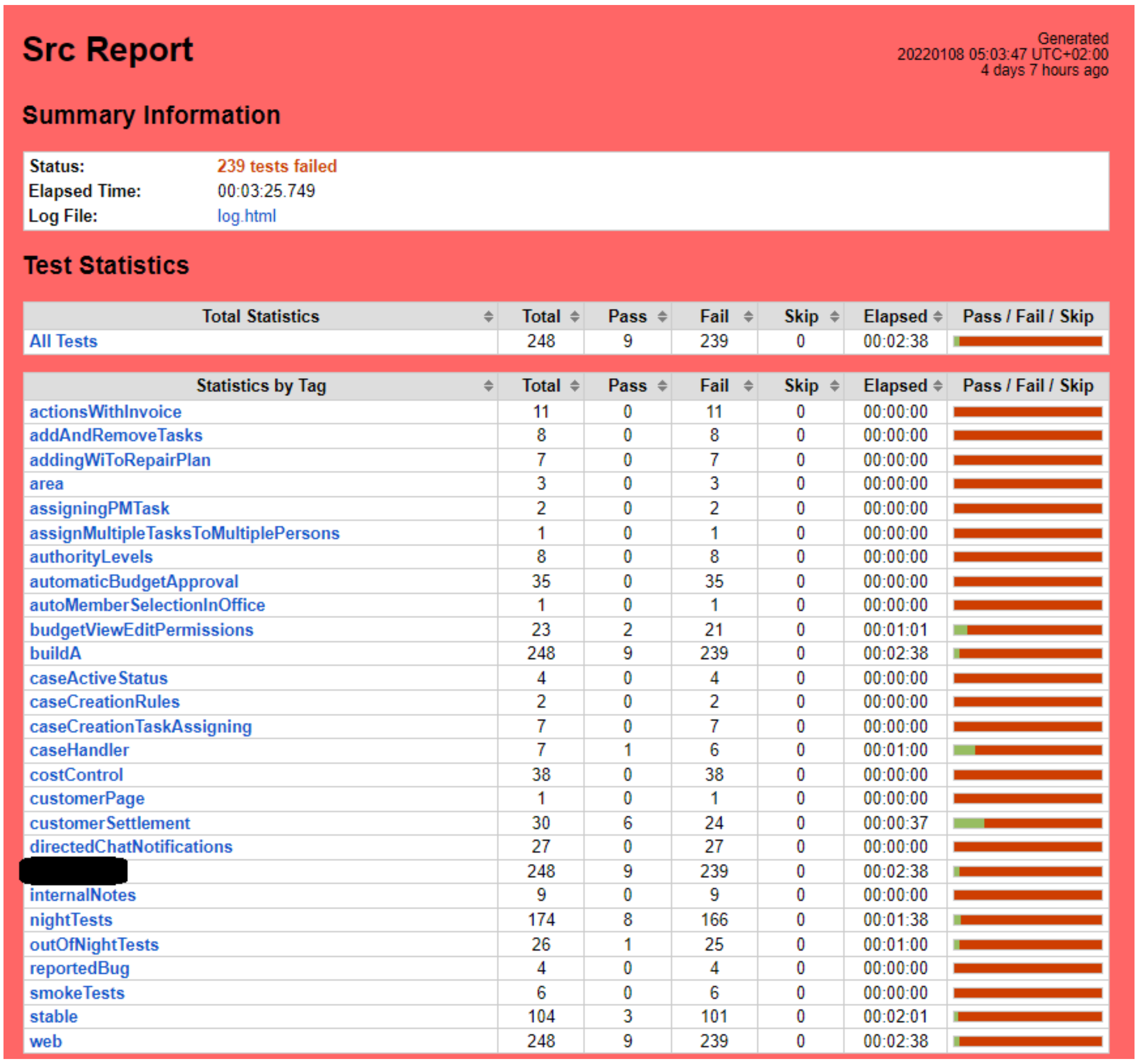
This figure 4.2.2 clearly shows the long duration of the test execution which is more than 13 hours. This is a very long duration and basically makes it practically impossible to debug the errors fast enough to be communicated to the developers.

Due to lengthy duration, the tests were forced to be run only during night because they took very long time for execution. Because of this prolonged duration, the test automation became unreliable, as it was difficult to conclude the test execution status during the Scrum meetings. Overall, no returns from the test automation suite which ideally is implemented to reduce the testing time and provide quicker feedback and quicker decisions and hence shortened period of release cycle(s).

4.2.3 Automation Quality

Below are the main factors which contributed to the poor automation quality of the old test automation suite:

4.2.3.1 Unreadable log reports



Src Report

Generated
20220108 05:03:47 UTC+02:00
4 days 7 hours ago

Summary Information

Status:	239 tests failed
Elapsed Time:	00:03:25.749
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	248	9	239	0	00:02:38	🟩🟥

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
actionsWithInvoice	11	0	11	0	00:00:00	🟥
addAndRemoveTasks	8	0	8	0	00:00:00	🟥
addingWiToRepairPlan	7	0	7	0	00:00:00	🟥
area	3	0	3	0	00:00:00	🟥
assigningPMTask	2	0	2	0	00:00:00	🟥
assignMultipleTasksToMultiplePersons	1	0	1	0	00:00:00	🟥
authorityLevels	8	0	8	0	00:00:00	🟥
automaticBudgetApproval	35	0	35	0	00:00:00	🟥
autoMemberSelectionInOffice	1	0	1	0	00:00:00	🟥
budgetViewEditPermissions	23	2	21	0	00:01:01	🟩🟥
buildA	248	9	239	0	00:02:38	🟩🟥
caseActiveStatus	4	0	4	0	00:00:00	🟥
caseCreationRules	2	0	2	0	00:00:00	🟥
caseCreationTaskAssigning	7	0	7	0	00:00:00	🟥
caseHandler	7	1	6	0	00:01:00	🟩🟥
costControl	38	0	38	0	00:00:00	🟥
customerPage	1	0	1	0	00:00:00	🟥
customerSettlement	30	6	24	0	00:00:37	🟩🟥
directedChatNotifications	27	0	27	0	00:00:00	🟥
[redacted]	248	9	239	0	00:02:38	🟩🟥
internalNotes	9	0	9	0	00:00:00	🟥
nightTests	174	8	166	0	00:01:38	🟩🟥
outOfNightTests	26	1	25	0	00:01:00	🟩🟥
reportedBug	4	0	4	0	00:00:00	🟥
smokeTests	6	0	6	0	00:00:00	🟥
stable	104	3	101	0	00:02:01	🟩🟥
web	248	9	239	0	00:02:38	🟩🟥

Figure 4.2.3.1 Jenkins test summary

As shown in above figure 4.2.3.1 these reports gave no information of the root cause of the failures and too cumbersome to spot the errors in code. These reports just give very basic information that some tests were executed and there is no categorization based on any functionality etc. In these report all the atomic groups are not grouped and therefor they form a list of 200-300 lines.

4.2.3.2 Hardcoded Data Usage

Usage of hardcoded data makes the automated tests less scalable because they depend always on the exact data that is provided, making the tests fragile and more prone to failure.

Sample code from old test suite:

```
"from
resources.web.<project_name>.useCases.zipcodeAndDistrict.Create_
and_filter_cases_with_district import MyUseCase, Variables

class Create_and_filter_cases_with_district(MyUseCase):
    def __init__(self):
        import
resources.web.<project_name>.portalVariables.portalVariables  as
_portalVariables
        import
resources.web.<project_name>.portalVariables.insuranceCompany as
_insuranceCompany

        MyUseCase.__init__(
            self,
            variables=Variables(
                browser=_portalVariables.browser,
                baseURL=_portalVariables.baseURL,

portalAdminEmail=_insuranceCompany.Office.UserLevel7.email,

portalAdminPassword=_insuranceCompany.Office.UserLevel7.password
,
                firstZipcode="12345",
                firstDistrict="<dist_name>",
                secondZipcode="12346",
                secondDistrict="Telemark",
                filterCategory="District",
            ),
        ) "
```

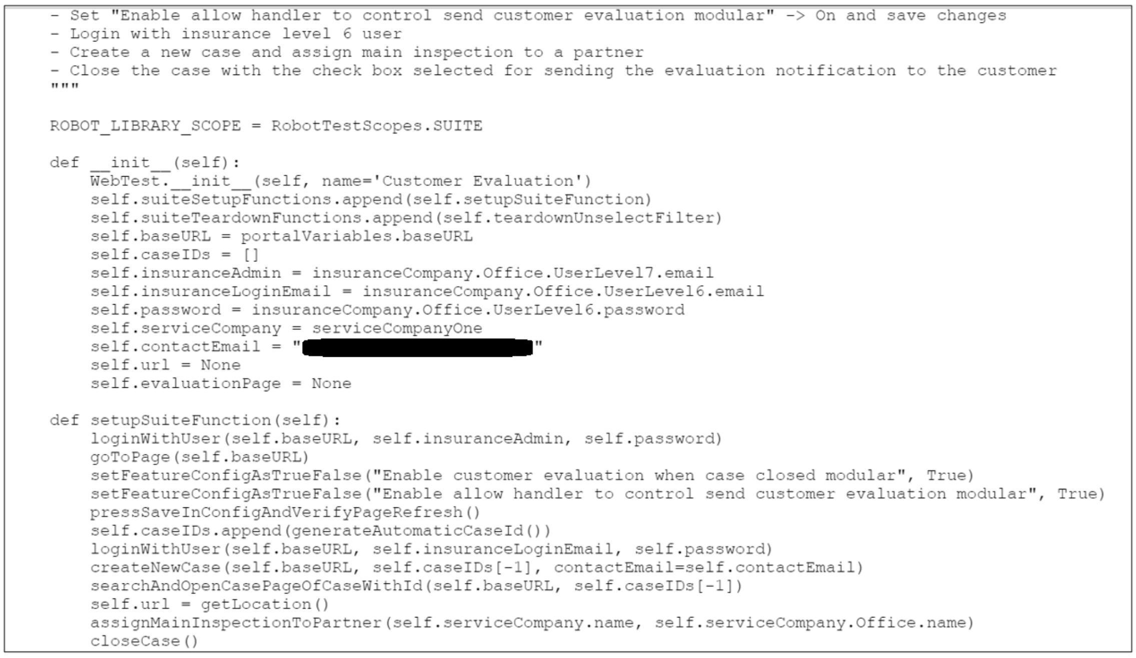
The above sample of code gives a glimpse of the old tests written in a complicated way and using hardcoded data in the tests, which makes the tests practically unscalable.

4.2.3.3 Inter dependence of the tests

The tests in the old framework were dependent on each other and therefore if one test failed then all the other tests dependent on it would also fail, thereby tests were failing mostly because of one or other reason and eventually whole test suite would fail, making it hard to debug the cause of failure.

4.2.3.4 Documentation

The documentation in old automation suite was poor and there was no information about what action is being performed. The tests had no information about the functionality being covered by the automated tests, which makes it rather complicated to understand the whole purpose of the tests. Since there was no documentation written in the tests it was much difficult to debug the tests in case of any failures.



```
    - Set "Enable allow handler to control send customer evaluation modular" -> On and save changes
    - Login with insurance level 6 user
    - Create a new case and assign main inspection to a partner
    - Close the case with the check box selected for sending the evaluation notification to the customer
    """

    ROBOT_LIBRARY_SCOPE = RobotTestScopes.SUITE

    def __init__(self):
        WebTest.__init__(self, name='Customer Evaluation')
        self.suiteSetupFunctions.append(self.setupSuiteFunction)
        self.suiteTeardownFunctions.append(self.teardownUnselectFilter)
        self.baseURL = portalVariables.baseURL
        self.caseIDs = []
        self.insuranceAdmin = insuranceCompany.Office.UserLevel7.email
        self.insuranceLoginEmail = insuranceCompany.Office.UserLevel6.email
        self.password = insuranceCompany.Office.UserLevel6.password
        self.serviceCompany = serviceCompanyOne
        self.contactEmail = "[redacted]"
        self.url = None
        self.evaluationPage = None

    def setupSuiteFunction(self):
        loginWithUser(self.baseURL, self.insuranceAdmin, self.password)
        goToPage(self.baseURL)
        setFeatureConfigAsTrueFalse("Enable customer evaluation when case closed modular", True)
        setFeatureConfigAsTrueFalse("Enable allow handler to control send customer evaluation modular", True)
        pressSaveInConfigAndVerifyPageRefresh()
        self.caseIDs.append(generateAutomaticCaseId())
        loginWithUser(self.baseURL, self.insuranceLoginEmail, self.password)
        createNewCase(self.baseURL, self.caseIDs[-1], contactEmail=self.contactEmail)
        searchAndOpenCasePageOfCaseWithId(self.baseURL, self.caseIDs[-1])
        self.url = getLocation()
        assignMainInspectionToPartner(self.serviceCompany.name, self.serviceCompany.Office.name)
        closeCase()
```

Figure 4.2.3.4

The figure 4.2.3.4 shows a code snippet where the tests have no documentation thereby giving no information about the functionality being covered by the automated test.

5 New Automation Framework Details

This section mainly gives overall details about the new framework with following sections describing the implementation method, python upgrade, IDE upgrade and necessary tools implementation details.

The discussions from the interviews, meetings with the PM and QA Manager proposed a sample methodology to be adopted and implemented in small part of code to analyse its performance with improved coding guidelines. After several meetings and discussions about the new approach a new framework was created where few workflows were targeted to be automated at first. A list of all the necessary workflows along with the agreement from the business side was created. The test coverage [16] was monitored strictly in order to cover all the important areas as identified through management discussions. The overall structure of the new automation framework was based on the Page object model methodology.

Robot Framework (Robot Framework, 2021) with a keyword driven approach was identified as solution for the new automation suite. The new framework was based on Robot Framework and Python3, and it utilizes the best practices and latest tools to enhance test automation for the case company.

With the keyword driven approach being followed for Robot Framework, at present a non-programming background is also able to write test cases. Pycharm editor was selected as the IDE for writing the robot tests, with several additional plugins installed to support the latest coding guidelines.

The robot tests were integrated with Jenkins for CI&CD to execute the tests at scheduled time automatically, in order to decrease the manual work. and identify the defects much before the production failures. With the help of Jenkins, it is possible to schedule a single test or group of tests to analyse the results and generate the test reports at any desired time. Due to the simplest approach being followed in the new automation framework, it will be easier for the test engineers to write automated scripts quickly and communicate the failure to the developer faster, to improve the overall quality of the product.

Review tools like Robocop and Pylama were also implemented and used in the new test framework to enhance the code quality for the test automation framework.

Reporting was also improved in the new test framework by the help of new tool Allure implementation. Different reports were generated to understand the failure in the tests, through which it was much easier to understand the error prone areas and the exact cause of failure with screenshots etc. These reports were simple, human readable and were used to identify the total percentage of failed tests.

The overall implementation of the new automation framework was done in steps with defined objectives so that the targets were easily achieved.

5.1 Implementation

This section provides the insight into the implementation details, tasks and activities performed in order to improve the shortcomings of old test suite.

Following major areas were improved as part of the new framework setup:

- Python upgrade
- IDE upgrade
- Review process and tool implementation
- Framework packages and plugins
- Framework Configuration
- Directory structure simplification – Page object model implementation
- Integrations enhancements – Jenkins and TestRail integration
- Keyword approach and documentation
- Parallel execution tool implementation
- Usage of tags to control selective execution
- Reporting enhancements

5.1.1 Python upgrade

Old test suite was using Python 2x, which was no longer being supported by robot framework, therefore it was necessary to upgrade from Python 2x to Python 3x.

Some key differences between Python 2 and Python 3 as shown in figure 5.2.1a below.

Parameters	Python 2	Python 3
Print command	it is a statement	it is a function
Strings to be stored in	ASCII	UNICODE
Division	integral value	floating point value
Exceptions	in notation	in parenthesis
Syntax complication	too much difficult	easy to understand
Current usage	not used anymore	in use
Compatibility	Incompatible libraries	Compatible libraries

Figure 5.2.1a Python 2 vs 3

5.1.2 IDE upgrade

Initially RIDE tool was being used for test development but it had many limitations apart from performance issues. The basic editor was replaced by modern integrated development environment tool called Pycharm from Jetbrains company.

As part of this exercise, usage of Pycharm was commonly agreed as IDE for test development and was installed on all testers machines.

5.1.3 Library Upgrade

Previously Selenium Library was being used for identifying the elements on the web page of the web portal of the case company. This library had certain limitations and was much slower in identifying the exact elements of the web page, also it needed ‘sleep’ commands to be used in the robot tests sometimes to wait for certain elements to appear on the web page. Therefore, Browser library powered by Microsoft’s Playwright, was implemented in the new test automation framework because of certain advantages as follows:

- No wait and sleep needed, automatically waits for the element
- Utilizes javascript(JS) based tech known as Playwright which directly interacts with browser API and its contents
- runs headless and does not need browser to be opened visibly

- High Performance: because pages in two separate contexts do not share cookies, sessions, or profile settings. Compared to Selenium, these do not require their own browser process. To get a clean environment a test can just open a new context. Due to this new independent browser sessions can be opened with Robot Framework Browser about 10 times faster than with Selenium by just opening a New Context within the opened browser.

5.1.4 Review process and tool implementation

All the tests written within the suite should follow certain coding guidelines and best practices as defined by Robot Framework documentation. As per best practice and to keep the code quality over long period, the test development team responsible of creating test-cases should be reviewed and later the changes should be merged into the Git repository.

Initially there was no review tool in place which overtime compromised the code quality but as part of this initiative, a tool named Robocop for checking any coding guidelines violations was implemented. This review tool is very helpful for quick debugging of the code because it can immediately locate the deviations from the coding standard.

This enabled a non-programming background test developers to contribute into the test suite creation and write test cases without affecting the code quality.

In the new framework, there was hardly a need of writing in python code but for some corners where is necessary, another tool called Pylama was also implemented**,** which helps verify the test scripts as per the Python coding guidelines.

Overall, the review tools enforced the review process automatically since it wasn’t possible to merge the code to master without fixing the violations reported by the review tools(s).

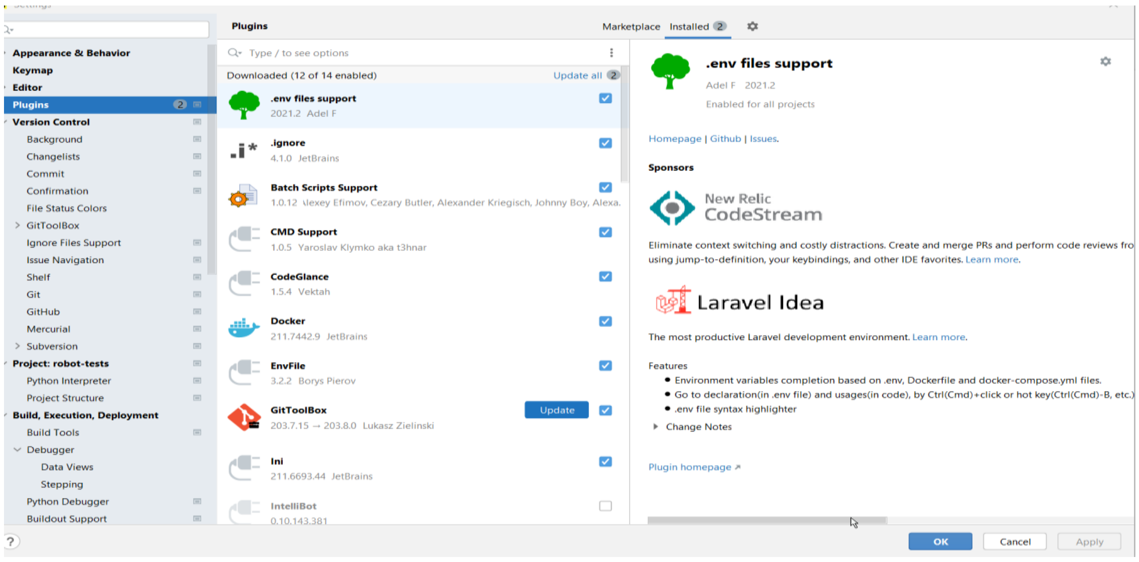
5.1.5 Framework Packages and plugins

Pycharm was upgraded with the latest Robot Framework version, different libraries, and plugins to support the latest test scripts. Several libraries were installed in Pycharm to support the latest automated tests as shown in figure 5.1.5a and 5.1.5b

- Browser library
- Env files support
- Batch script support
- Cmd support
- Robot Framework Language Server

As part of this initiative, old selenium library was replaced by latest Robot Framework browser library which is based on Microsoft playwright library. The new browser library implementation was enabled by adding the readily available plugin in Robot Framework through Pycharm.

Similarly, other plugins were included in the framework like .env files support for the .env files which were used to define the environment details like browser, url, timeout etc. Batch script support for supporting the batch files written to write commands for executing the robot test, cmd support for enabling highlighting in the test cases, and 'Robot Framework Language Server' for Robot Framework support.



Downloaded (12 of 14 enabled)

.env files support	2021.2 Adel F	✓
.ignore	4.1.0 JetBrains	✓
Batch Scripts Support	1.0.12 \lexey Efimov, Cezary Butler, Alexander Kriegisch, Johnny Boy, Alexa.	✓
CMD Support	1.0.5 Yaroslav Klymko aka t3hnar	✓
CodeGlance	1.5.4 Vektah	✓
Docker	211.7442.9 JetBrains	✓
EnvFile	3.2.2 Borys Pierov	✓
GitToolBox	203.7.15 → 203.8.0 Lukasz Zielinski	✓
Ini	211.6693.44 JetBrains	✓
IntelliBot	0.10.143.381	☐

.env files support
Adel F 2021.2
Enabled for all projects

Homepage | Github | Issues.

Sponsors

New Relic CodeStream

Eliminate context switching and costly distractions. Create and merge PRs and perform code reviews fro
using jump-to-definition, your keybindings, and other IDE favorites. Learn more.

Laravel Idea

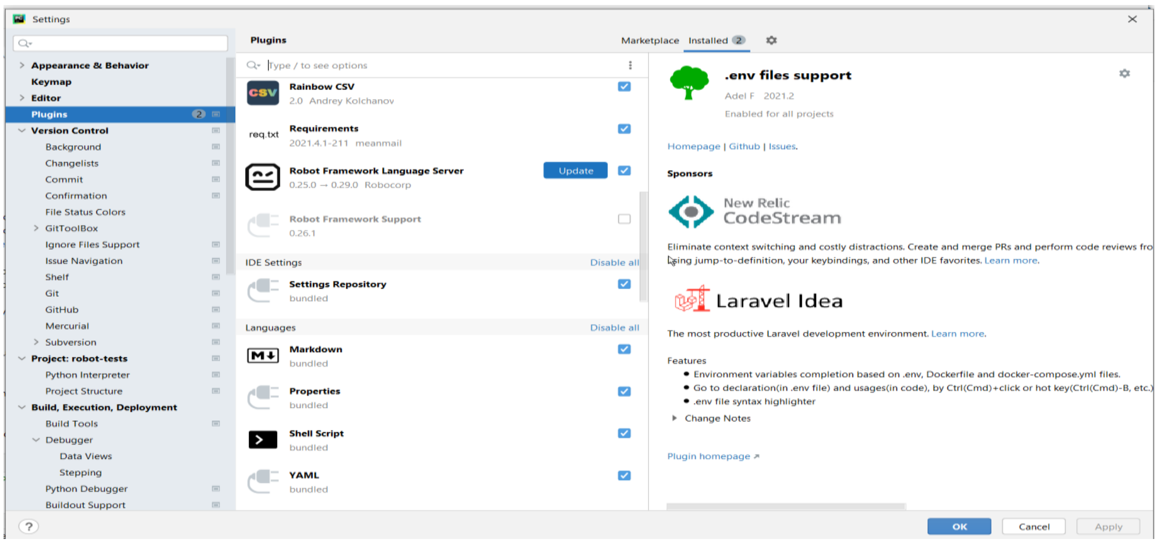
The most productive Laravel development environment. Learn more.

Features
- Environment variables completion based on .env, Dockerfile and docker-compose.yml files.
- Go to declaration(in .env file) and usages(in code), by Ctrl(Cmd)+click or hot key(Ctrl(Cmd)-B, etc.)
- .env file syntax highlighter

Change Notes

Plugin homepage ↗

Figure 5.1.5 a Plugin details



Rainbow CSV ✓
2.0 Andrey Kolchanov

Requirements ✓
2021.4.1-211 meanmail

Robot Framework Language Server ✓
0.25.0 → 0.29.0 Robocorp

Robot Framework Support ☐
0.26.1

IDE Settings

Settings Repository ✓
bundled

Languages

Markdown ✓
bundled

Properties ✓
bundled

Shell Script ✓
bundled

YAML ✓
bundled

.env files support
Adel F 2021.2
Enabled for all projects

Homepage | Github | Issues.

Sponsors

New Relic CodeStream

Eliminate context switching and costly distractions. Create and merge PRs and perform code reviews fro
using jump-to-definition, your keybindings, and other IDE favorites. Learn more.

Laravel Idea

The most productive Laravel development environment. Learn more.

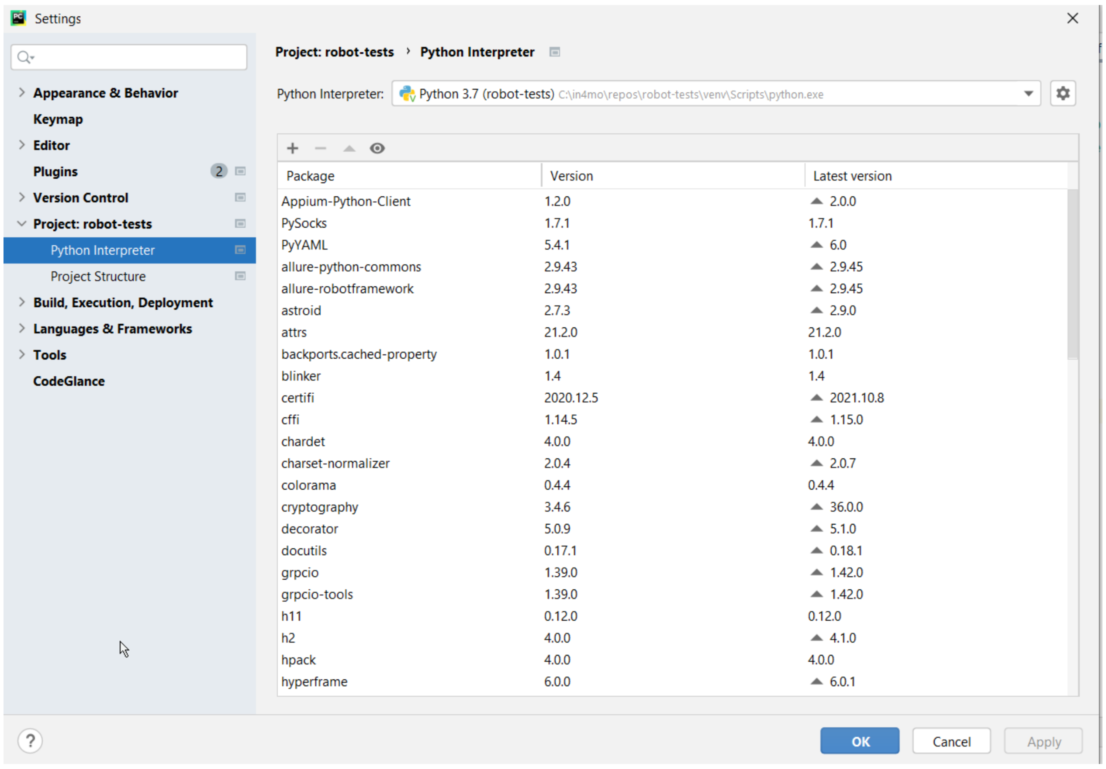
Features
- Environment variables completion based on .env, Dockerfile and docker-compose.yml files.
- Go to declaration(in .env file) and usages(in code), by Ctrl(Cmd)+click or hot key(Ctrl(Cmd)-B, etc.)
- .env file syntax highlighter

Change Notes

Plugin homepage ↗

Figure 5.1.5 b Plugin details contd.

Necessary plugins were added to Pycharm IDE to support the latest automated tests scripts. These plugins support new coding style and provide quite efficient methods to achieve solutions to difficult problems related to the framework.



Project: robot-tests › Python Interpreter

Python Interpreter: Python 3.7 (robot-tests) C:\in4mo\repos\robot-tests\venv\Scripts\python.exe

Package	Version	Latest version
Appium-Python-Client	1.2.0	▲ 2.0.0
PySocks	1.7.1	1.7.1
PyYAML	5.4.1	▲ 6.0
allure-python-commons	2.9.43	▲ 2.9.45
allure-robotframework	2.9.43	▲ 2.9.45
astroid	2.7.3	▲ 2.9.0
attrs	21.2.0	21.2.0
backports.cached-property	1.0.1	1.0.1
blinker	1.4	1.4
certifi	2020.12.5	▲ 2021.10.8
cffi	1.14.5	▲ 1.15.0
chardet	4.0.0	4.0.0
charset-normalizer	2.0.4	▲ 2.0.7
colorama	0.4.4	0.4.4
cryptography	3.4.6	▲ 36.0.0
decorator	5.0.9	▲ 5.1.0
docutils	0.17.1	▲ 0.18.1
grpcio	1.39.0	▲ 1.42.0
grpcio-tools	1.39.0	▲ 1.42.0
h11	0.12.0	0.12.0
h2	4.0.0	▲ 4.1.0
hpack	4.0.0	4.0.0
hyperframe	6.0.0	▲ 6.0.1

Figure 5.1.4 c Packages installed

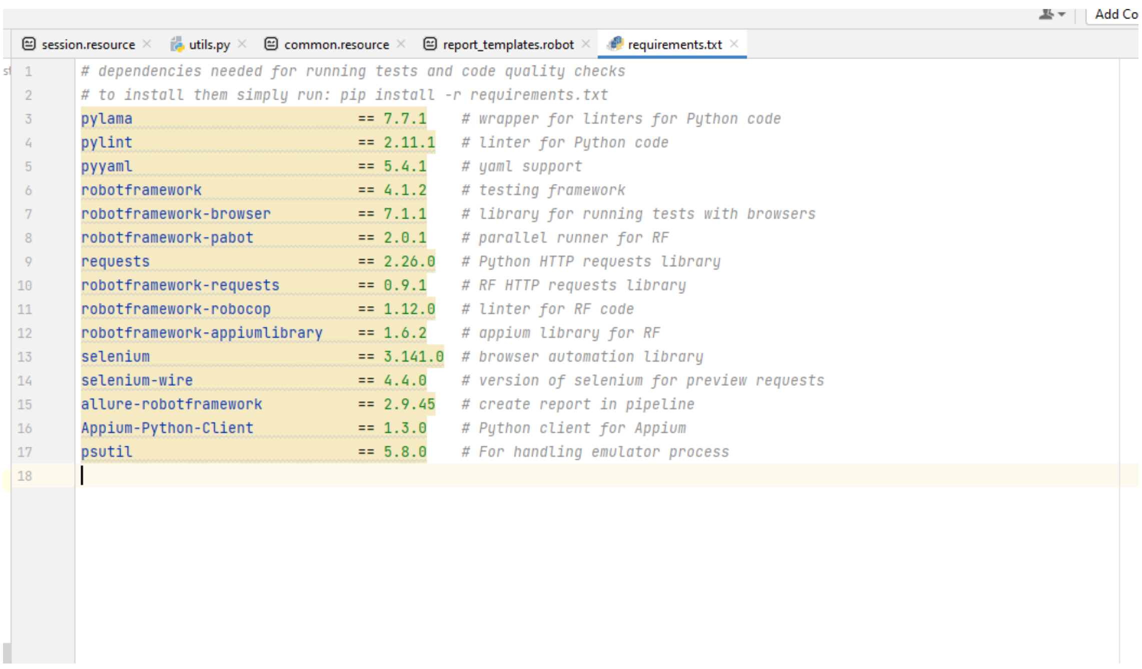
Different Packages as shown in figure 5.1.4 c above, were included through Pycharm IDE to improve the library support for writing automated tests, by utilising some important keywords from these libraries.

5.1.6 Framework Configuration

This section details the necessary configuration or versions of the different libraries or tools used in the new test automation framework:

```
# Dependencies needed for running tests and code quality checks
# to install them simply run: pip install -r requirements.txt
pylama                            == 7.7.1     # wrapper for linters for Python code
pylint                            == 2.11.1    # linter for Python code
pyyaml                            == 5.4.1     # yaml support
robotframework                    == 4.1.2     # testing framework
robotframework-browser            == 7.1.1     # library for running tests with browsers
robotframework-pabot              == 2.0.1     # parallel runner for RF
requests                          == 2.26.0    # Python HTTP requests library
robotframework-requests           == 0.9.1     # RF HTTP requests library
robotframework-robocop            == 1.12.0    # linter for RF code
robotframework-appiumlibrary      == 1.6.2     # Appium library for RF
selenium                          == 3.141.0 # browser automation library
selenium-wire                     == 4.4.0    # selenium version for preview requests
allure-robotframework             == 2.9.45    # create report in pipeline
Appium-Python-Client              == 1.3.0    # Python client for Appium
psutil                            == 5.8.0    # For handling emulator process
```

These dependencies were stored in the project under the file requirements.txt in Pycharm and it was possible to update the version of any dependency from the file itself. This way it became quite comfortable to have everything under single file stored.



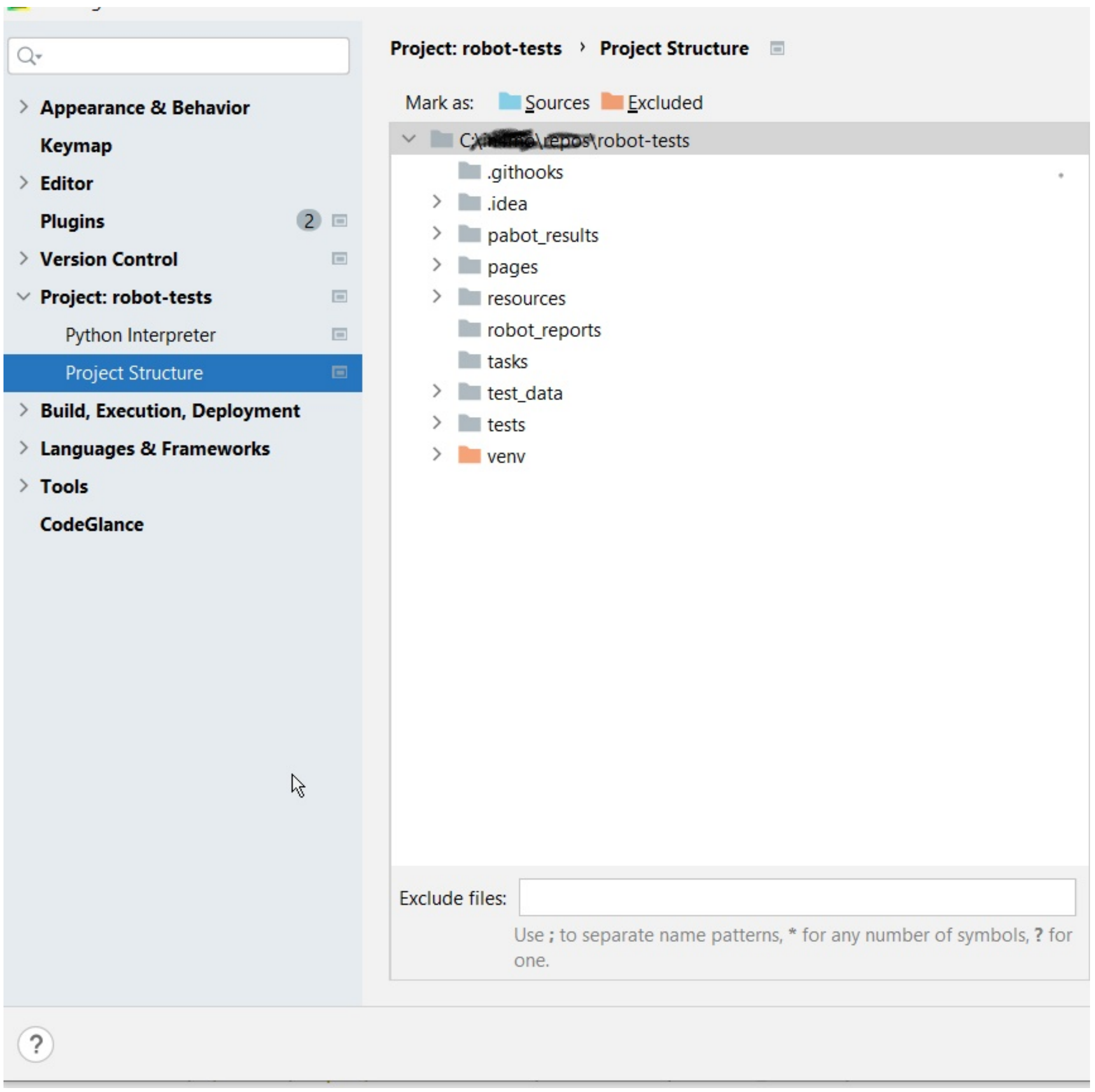
```
# dependencies needed for running tests and code quality checks
# to install them simply run: pip install -r requirements.txt
pylama                          == 7.7.1    # wrapper for linters for Python code
pylint                          == 2.11.1   # linter for Python code
pyyaml                          == 5.4.1    # yaml support
robotframework                  == 4.1.2    # testing framework
robotframework-browser          == 7.1.1    # library for running tests with browsers
robotframework-pabot            == 2.0.1    # parallel runner for RF
requests                        == 2.26.0   # Python HTTP requests library
robotframework-requests         == 0.9.1    # RF HTTP requests library
robotframework-robocop          == 1.12.0   # linter for RF code
robotframework-appiumlibrary    == 1.6.2    # appium library for RF
selenium                        == 3.141.0  # browser automation library
selenium-wire                   == 4.4.0    # version of selenium for preview requests
allure-robotframework           == 2.9.45   # create report in pipeline
Appium-Python-Client            == 1.3.0    # Python client for Appium
psutil                          == 5.8.0    # For handling emulator process
```

Figure 5.1.5 Requirements file screenshot

From the figure 5.2.2 we see the requirement file being added to the test automation framework. This file, as we see is easily maintainable by the user and gives hints to the user if the installed package needs to be upgraded to the next version. Also, it was a convenient way to maintain the installed versions of the software at a centralised place.

5.1.7 Directory Structure Simplification – Page Object Model Implementation

During the setup, the directory structure was written and finalised to support the modular approach for Test automation. It was discussed with the Quality Assurance Manager and the business stakeholders to keep the overall test framework structure as per the overall design of the case company's webpages/portal. Page Object Model approach where the folders will be named as per the individual pages in the product. This approach gives the framework better readability and reusability of the methods developed.


Appearance & Behavior
Keymap
Editor
Plugins 2
Version Control
Project: robot-tests
Python Interpreter
Project Structure
Build, Execution, Deployment
Languages & Frameworks
Tools
CodeGlance

Project: robot-tests › Project Structure

Mark as: Sources Excluded

C:\[redacted]\repos\robot-tests
.githooks
.idea
pabot_results
pages
resources
robot_reports
tasks
test_data
tests
venv

Exclude files:
Use ; to separate name patterns, * for any number of symbols, ? for one.

Figure 5.1.6 Directory structure

Here is a quick view of the directory structure as shown in the figure 5.1.6 above been developed for the new test automation framework:

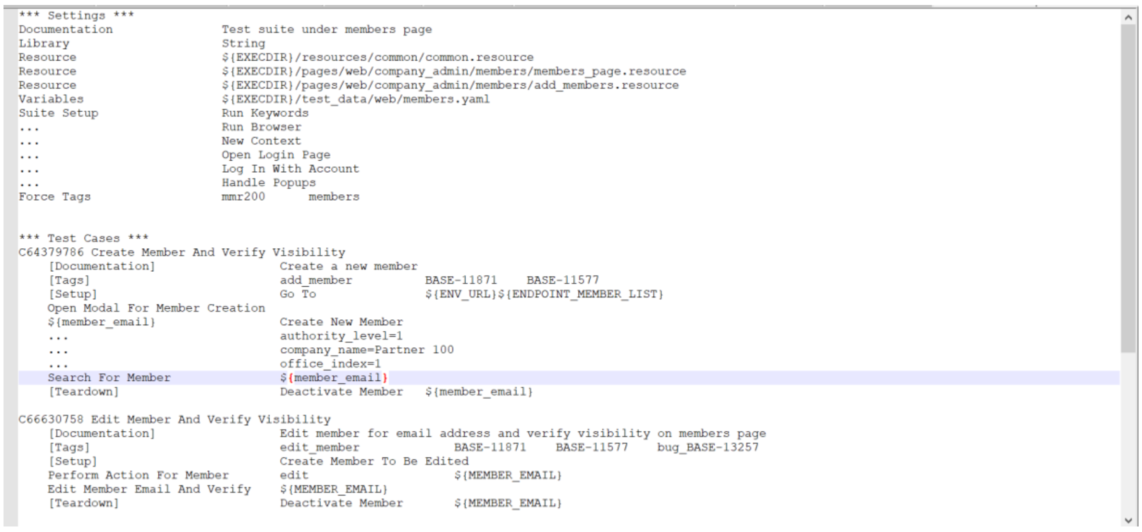
The project structure:

The root directory (robot-tests) contains several directories and files where each serve a different purpose:

```
robot-tests
├── pages                    # pages for Page Object Model pattern
│  ├── web
│  │  └──<page_name>          # container for specific webpage
│  │     ├──page_name. yaml    # file with defined locators
│ │    └── page_name.resource # file with defined keywords
│ └── mobile                  # a structure similar to web pages
│   └── ...
├── resources                 # data shared by all tests
│ ├── common                  # keywords and variables common
│ ├── environments            # run configurations as input files
│ └── mobile                 # mobile custom libraries and functions
├── tasks                     # tasks to execute test data load
├── test_data                 # place for keeping all test data
│ ├── web                     # test data for UI tests
│ └── mobile                  # test data for mobile tests
├── tests                     # main directory with tests
│ ├── mobile                  # mobile tests
│ └── web                     # UI tests
│   └── <suite_name>. robot # suite files with RF tests
├── robot_reports             # output files from test execution
└── ...                       # Other files in root like readme,
requirements etc.
```

This type of folder structure was quite helpful and efficient for the test automation engineer because this was completely as per the practical page structure of the case company’s product. Since the product contained the mobile and the web part therefore two different folders were created to store the relevant artifacts and the necessary page object resources.

The directory structure was strictly maintained and implemented using the Page Object Model Approach in the new framework. This approach was implemented in order to improve the complicated design of old test automation suite, and it was much easier to understand by the test team.



```
*** Settings ***
Documentation                 Test suite under members page
Library                       String
Resource                      ${EXECDIR}/resources/common/common.resource
Resource                      ${EXECDIR}/pages/web/company_admin/members/members_page.resource
Resource                      ${EXECDIR}/pages/web/company_admin/members/add_members.resource
Variables                     ${EXECDIR}/test_data/web/members.yaml
Suite Setup                   Run Keywords
...                           Run Browser
...                           New Context
...                           Open Login Page
...                           Log In With Account
...                           Handle Popups
Force Tags                    mmr200      members


*** Test Cases ***
C64379786 Create Member And Verify Visibility
    [Documentation]                 Create a new member
    [Tags]                          add_member          BASE-11871    BASE-11577
    [Setup]                         Go To               ${ENV_URL}${ENDPOINT_MEMBER_LIST}
    Open Modal For Member Creation
    ${member_email}                 Create New Member
    ...                             authority_level=1
    ...                             company_name=Partner 100
    ...                             office_index=1
    Search For Member               ${member_email}
    [Teardown]                      Deactivate Member   ${member_email}

C66630758 Edit Member And Verify Visibility
    [Documentation]                 Edit member for email address and verify visibility on members page
    [Tags]                          edit_member             BASE-11871    BASE-11577    bug_BASE-13257
    [Setup]                         Create Member To Be Edited
    Perform Action For Member       edit                    ${MEMBER_EMAIL}
    Edit Member Email And Verify    ${MEMBER_EMAIL}
    [Teardown]                      Deactivate Member       ${MEMBER_EMAIL}
```

Figure 5.1.6 Sample Test1

5.1.8 Integrations Enhancements – Jenkins and TestRail Integration

The test management tool TestRail was setup and the integration between the automated tests and TestRail was established. The integration was helpful because whenever the tests were created in Pycharm there was a placeholder test created for it in TestRail thereby eliminating the need for manual creation of the test case in TestRail.

The CI/CD pipeline was created with Jenkins to run the pipeline with necessary configuration and integrated with the automated tests to run them over night, regression test suites and also whenever new commit was pushed to repository. Basically, whenever a new feature/functionality was developed with automated tests after every single commit pushed to the repos, the pipeline used to run to execute all the tests in the framework, to detect any flaky tests which would fail. After careful analysis of the reports, the reason of the failing tests.

5.1.9 Keyword Approach and Documentation

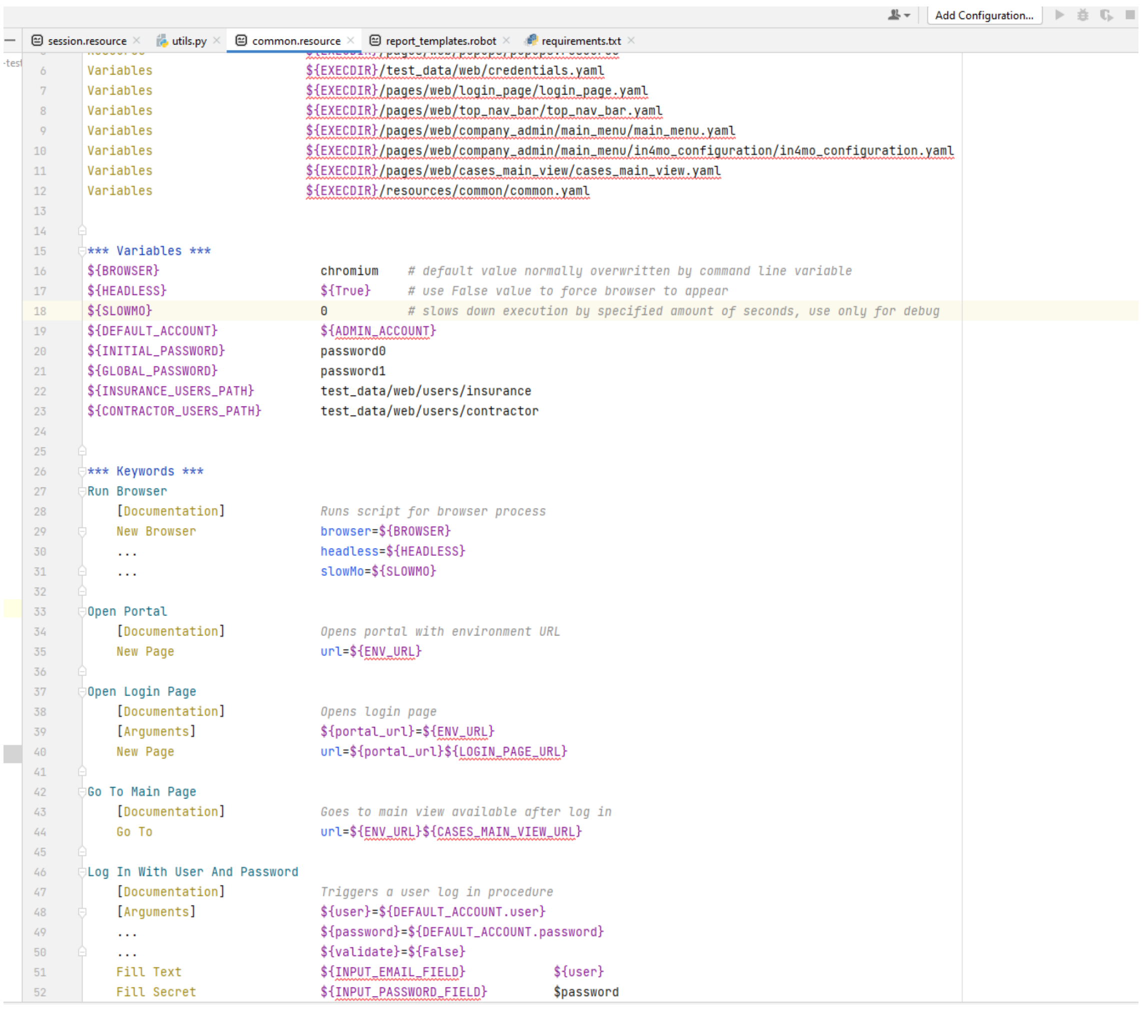
The automated tests were implemented in the new framework by using keyword driven approach [17]. This approach means that each user action to be performed will be written in the form of a customized keyword. Therefore, by using the customised keywords in the tests, a particular functionality can be easily covered by the automated scripts.

Following code snippet shows a sample test from the newly implemented test automation framework using keyword driven approach:

```
*** Settings ***
Documentation                 Test suite under members page
Library                       String
Variables                    ${EXECDIR}/test_data/web/members.yaml
Suite Setup                   Run Keywords
...                           Run Browser
...                           New Context
...                           Open Login Page
...                           Log In With Account
...                           Handle Popups
Force Tags                    members

*** Test Cases ***
Create Member And Verify Visibility
    [Documentation]                 Create a new member
    [Tags]       add_member
    [Setup]    Go To             ${ENV_URL}${ENDPOINT_MEMBER_LIST}
    Open Modal For Member Creation
    ${member_email}                 Create New Member
    ...                             authority_level=1
    ...                             company_name=Comp X
    ...                             office_index=1
    Search For Member               ${member_email}
    [Teardown]                Deactivate Member   ${member_email}
```

This code shows the test for creating a member in a company and searching for its availability after creation from the list of members available. As per the code, different keywords like “`Create Member And Verify Visibility`”, “`Search For Member`”, create new member, deactivate member are used to perform user actions. The keyword names resemble the user action that they are going to perform for e.g deactivate member will be deactivating a member been created previously. Additionally, tags are used within the tests to help with searchability according to the functionality for example, adding a member is tagged as add_member. Using this tag, it becomes very easy to search for a particular test written for this functionality in the product.

This approach was used in all the tests within the new test automation framework so that the name of the keyword would clearly define about the user action performed in contrast to the old test suite where neither the tests nor the documentation gave any hint on the user actions being performed.



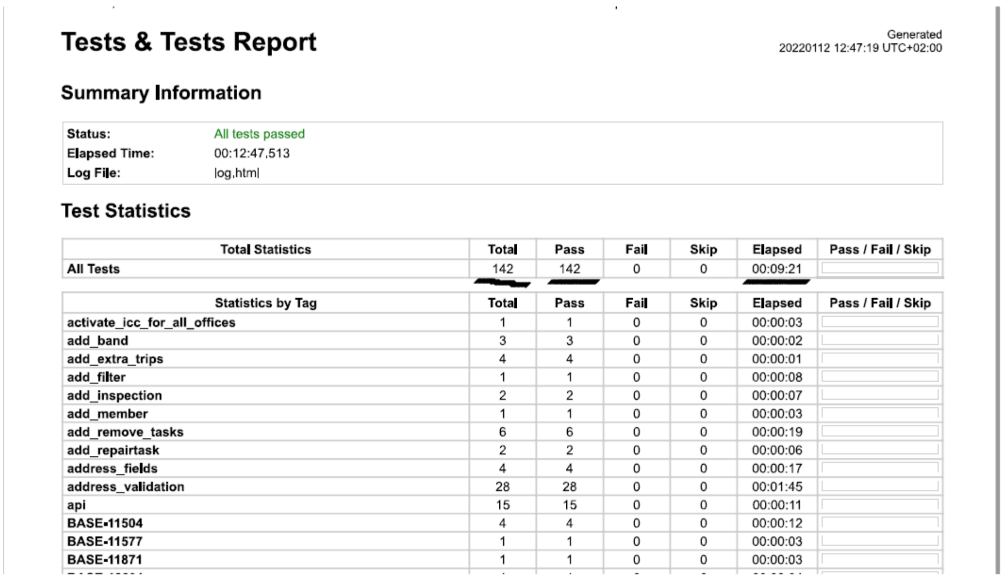
```
Variables                       ${EXECDIR}/test_data/web/credentials.yaml
Variables                       ${EXECDIR}/pages/web/login_page/login_page.yaml
Variables                       ${EXECDIR}/pages/web/top_nav_bar/top_nav_bar.yaml
Variables                       ${EXECDIR}/pages/web/company_admin/main_menu/main_menu.yaml
Variables                       ${EXECDIR}/pages/web/company_admin/main_menu/in4mo_configuration/in4mo_configuration.yaml
Variables                       ${EXECDIR}/pages/web/cases_main_view/cases_main_view.yaml
Variables                       ${EXECDIR}/resources/common/common.yaml


*** Variables ***
${BROWSER}                        chromium    # default value normally overwritten by command line variable
${HEADLESS}                       ${True}     # use False value to force browser to appear
${SLOWMO}                         0           # slows down execution by specified amount of seconds, use only for debug
${DEFAULT_ACCOUNT}                ${ADMIN_ACCOUNT}
${INITIAL_PASSWORD}               password0
${GLOBAL_PASSWORD}                password1
${INSURANCE_USERS_PATH}           test_data/web/users/insurance
${CONTRACTOR_USERS_PATH}          test_data/web/users/contractor


*** Keywords ***
Run Browser
    [Documentation]               Runs script for browser process
    New Browser                   browser=${BROWSER}
    ...                           headless=${HEADLESS}
    ...                           slowMo=${SLOWMO}

Open Portal
    [Documentation]               Opens portal with environment URL
    New Page                      url=${ENV_URL}

Open Login Page
    [Documentation]               Opens login page
    [Arguments]                   ${portal_url}=${ENV_URL}
    New Page                      url=${portal_url}${LOGIN_PAGE_URL}

Go To Main Page
    [Documentation]               Goes to main view available after log in
    Go To                         url=${ENV_URL}${CASES_MAIN_VIEW_URL}

Log In With User And Password
    [Documentation]               Triggers a user log in procedure
    [Arguments]                   ${user}=${DEFAULT_ACCOUNT.user}
    ...                           ${password}=${DEFAULT_ACCOUNT.password}
    ...                           ${validate}=${False}
    Fill Text                     ${INPUT_EMAIL_FIELD}          ${user}
    Fill Secret                   ${INPUT_PASSWORD_FIELD}       $password
```

Figure 5.1.8 Keywords sample

From the figure 5.1.8 above we can see that the tests use a keyword approach and some of the common keywords are taken from the Browser Library, while others were customised for the user action as per the product needs.

5.1.10 Parallel execution tool implementation

In the new framework, an additional tool was implemented called Pabot, the parallel executor tool, which executes the robot tests in parallel. Due to Pabot, the execution time was significantly reduced as can be seen from the figure below.



Tests & Tests Report

Generated
20220112 12:47:19 UTC+02:00

Summary Information

Status:	All tests passed
Elapsed Time:	00:12:47,513
Log File:	log,html

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	142	142	0	0	00:09:21	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
activate_icc_for_all_offices	1	1	0	0	00:00:03	
add_band	3	3	0	0	00:00:02	
add_extra_trips	4	4	0	0	00:00:01	
add_filter	1	1	0	0	00:00:08	
add_inspection	2	2	0	0	00:00:07	
add_member	1	1	0	0	00:00:03	
add_remove_tasks	6	6	0	0	00:00:19	
add_repairtask	2	2	0	0	00:00:06	
address_fields	4	4	0	0	00:00:17	
address_validation	28	28	0	0	00:01:45	
api	15	15	0	0	00:00:11	
BASE-11504	4	4	0	0	00:00:12	
BASE-11577	1	1	0	0	00:00:03	
BASE-11871	1	1	0	0	00:00:03	

Figure 5.1.9 Test Report

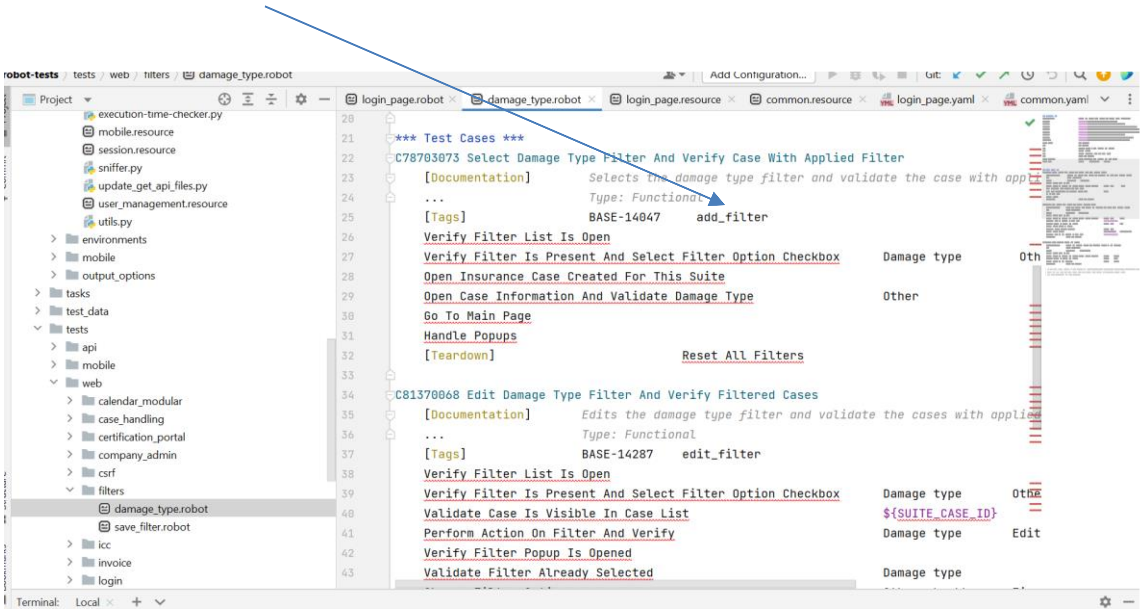
The above attached report in figure 5.1.9, shows that due to parallel execution being implemented in the new test framework, test execution time is 10 mins whereas it was 13 hours for old test automation suite.

5.1.11 Usage of tags to control selective execution

In the new test framework, robot tests were created by implementing Tags. Tags are the simplest way to run the automated tests for a specific functionality, by using the standard robot command for test execution, tags can be included so as to run the tests or search for the tests using the tags in the test reports.

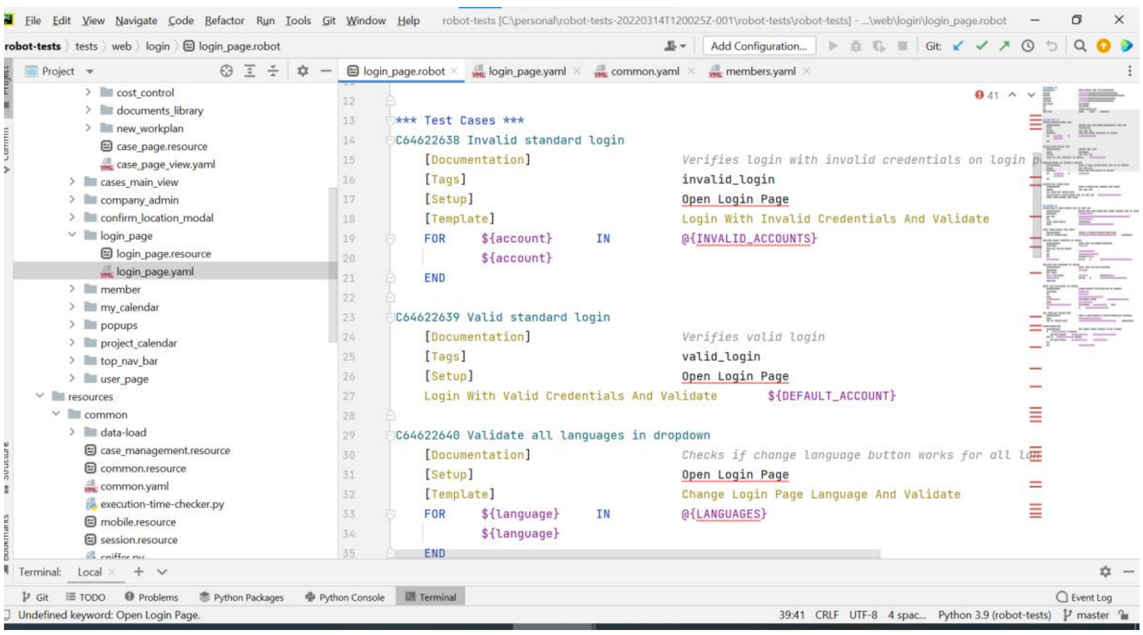
As mentioned above about tags, we can see from the example below, the following command uses ‘smoke’ tag to execute all the tests which have tags as ‘smoke’ while writing the tests.

```
robot --include add_filter --variable HOST:10.0.0.42
path/to/tests/
```



```
*** Test Cases ***
C78703073 Select Damage Type Filter And Verify Case With Applied Filter
    [Documentation]        Selects the damage type filter and validate the case with appl
    ...                    Type: Functional
    [Tags]                 BASE-14047     add_filter
    Verify Filter List Is Open
    Verify Filter Is Present And Select Filter Option Checkbox       Damage type       Oth
    Open Insurance Case Created For This Suite
    Open Case Information And Validate Damage Type                   Other
    Go To Main Page
    Handle Popups
    [Teardown]                             Reset All Filters

C81370068 Edit Damage Type Filter And Verify Filtered Cases
    [Documentation]       Edits the damage type filter and validate the cases with applied
    ...                   Type: Functional
    [Tags]                BASE-14287     edit_filter
    Verify Filter List Is Open
    Verify Filter Is Present And Select Filter Option Checkbox       Damage type      Othe
    Validate Case Is Visible In Case List                            ${SUITE_CASE_ID}
    Perform Action On Filter And Verify                              Damage type      Edit
    Verify Filter Popup Is Opened
    Validate Filter Already Selected                                 Damage type
```

Figure 5.1.10a Sample Test1



```
*** Test Cases ***
C64622638 Invalid standard login
    [Documentation]                  Verifies login with invalid credentials on login p
    [Tags]                           invalid_login
    [Setup]                          Open Login Page
    [Template]                       Login With Invalid Credentials And Validate
    FOR     ${account}      IN       @{INVALID_ACCOUNTS}
            ${account}
    END

C64622639 Valid standard login
    [Documentation]                  Verifies valid login
    [Tags]                           valid_login
    [Setup]                          Open Login Page
    Login With Valid Credentials And Validate       ${DEFAULT_ACCOUNT}

C64622640 Validate all languages in dropdown
    [Documentation]                  Checks if change language button works for all lan
    [Setup]                          Open Login Page
    [Template]                       Change Login Page Language And Validate
    FOR     ${language}     IN       @{LANGUAGES}
            ${language}
    END
```

Figure 5.1.10b Sample Test2

In these figures 5.1.10a and b, it can be clearly seen that these tests are created with suitable Tags as per the functional area of the tests for example valid_login for the verification of valid login into the product portal.

5.1.12 Reporting enhancements

For clear and simple reports, the new framework was implemented with Allure reports. These reports generated after the test execution were much convenient to understand because of clear graphs which showed the total number of tests passing or failing.

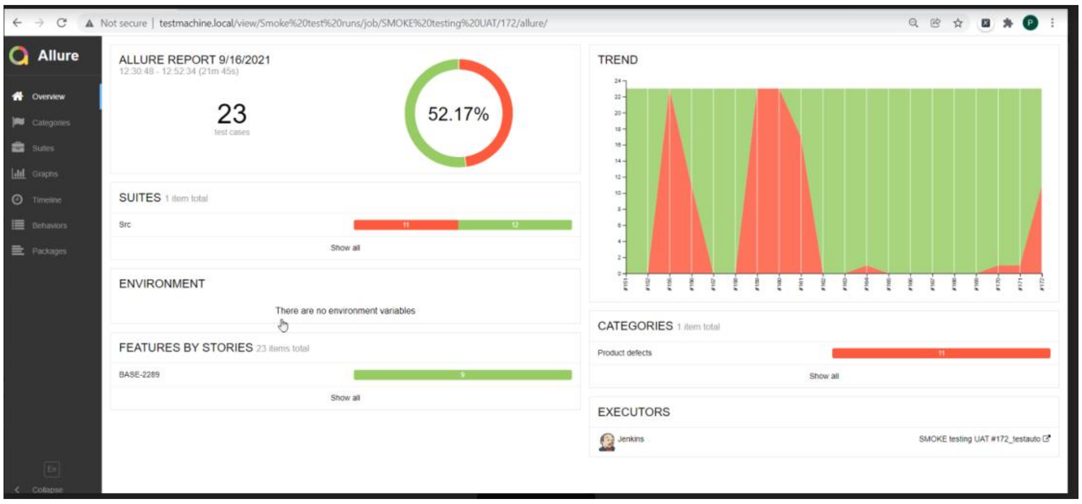


Figure 5.1.11 Allure report

Above figure 5.1.11 shows the test execution status for a particular test run. These reports were implemented using Jenkins as a plugin in the new framework.

6 Result Comparison

This section presents the comparison of old test automation suite and the new test automation framework from various perspectives:

	Old Suite	New Framework
Execution Time / Performance	Up to 13h	Less than 12mins
Parallel Test Execution	Not available	Capable (Pabot)
Review Capability	not available	Available
Independent Test Cases	0 – 10 % Only	100%
Automation Standards	Not followed	Maintained in all tests
Code Quality	Poor	Excellent
Understandability of Tests	Complicated and difficult	Easy to follow
Requirements File	not available	Available
Failure Percentage	50 – 99 %	0 – 10 %
Jenkins's integration	Available - limited extent	Available - full extent
Testrail Integration	not available	Available
Debugging	complicated	Easy (Failure screenshot captured)
Readability of Reports	Average	Excellent

Table 6a Comparison of Old vs New Framework

As can be seen from the above comparison table 6a, the new framework due to being implemented with many great features improves the overall tests usability and readability of reports.

The overall test execution time was improved by a great extent from few hours to few minutes because of the usage of parallel execution and tests were written independently in order to make parallel execution possible. The performance improvement was also brought by usage of better keywords provided by modern browser library up to certain extent.

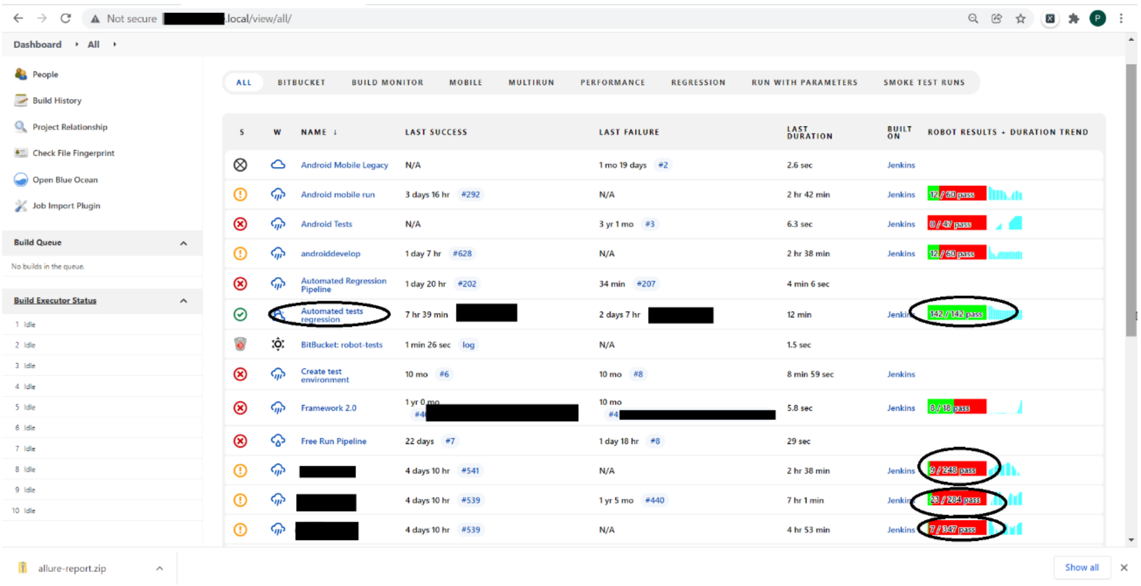
Parallel test execution saves a lot of time because all the tests can be run within a single instance using parallel process ids, with the help of Pabot the parallel execution tool used in the new test framework. It was not implemented in the old frameworks and therefore the execution time was 35 times longer than the new framework.

The new test automation framework employed the review tools called Robocop and Pylama which alerts the user for any coding guideline not been followed, which makes the automated tests aligned with automation standards and guidelines and ensuring the code quality in long run.

The code quality of the automated scripts was also improved by following the guidelines in the new framework in its implementation phase itself. The tests created were easy to script and easier to understand by the fellow team members. Any discrepancies in the guidelines, were pointed out by the code review tool Robocop in the new test framework. These steps were followed for each test pushed to the suite repository. These kinds of checks were not available for the old framework and therefore the old tests had many complications and failures.

The requirements file is being maintained in the new framework in order to enlist all the packages and the dependencies needed for running tests and code quality checks. Old framework didn’t have any such practice in place.

The requirements file was a simple .txt file which was maintained throughout the implementation and application of the new framework. This txt file has all the necessary library tools and their versions needed for the framework. If some tool needs an update, then there were hints provided in the new framework to upgrade the tool to next version. It helped to maintain the framework dependencies at a single place and whenever upgrade was needed it can be done from the single place. This file was not available in the old framework and thus tools were sometimes outdated and there was no simple way to update them for the whole framework.



Not secure [redacted].local/view/all/

Dashboard › All ›

Build Queue

No builds in the queue.

Build Executor Status

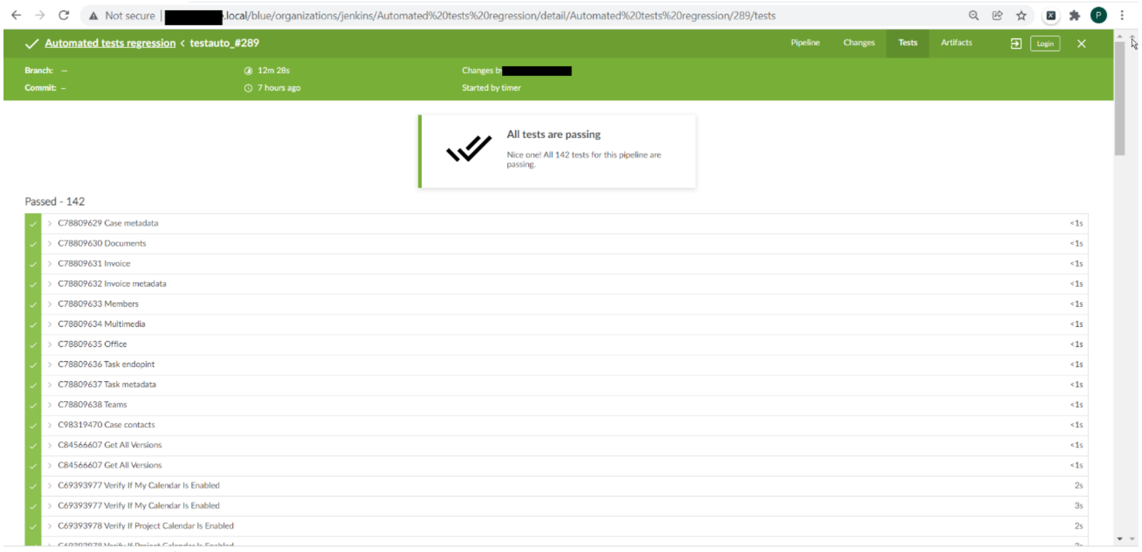
1 Idle
2 Idle
3 Idle
4 Idle
5 Idle
6 Idle
7 Idle
8 Idle
9 Idle
10 Idle

S	W	NAME ↓	LAST SUCCESS	LAST FAILURE	LAST DURATION	BUILT ON	ROBOT RESULTS + DURATION TREND
⊗	☁	Android Mobile Legacy	N/A	1 mo 19 days #2	2.6 sec	Jenkins	
!	☂	Android mobile run	3 days 16 hr #292	N/A	2 hr 42 min	Jenkins	12 / 60 pass
✗	☂	Android Tests	N/A	3 yr 1 mo #3	6.3 sec	Jenkins	0 / 47 pass
!	☂	androiddevelop	1 day 7 hr #628	N/A	2 hr 38 min	Jenkins	12 / 60 pass
✗	☂	Automated Regression Pipeline	1 day 20 hr #202	34 min #207	4 min 6 sec		
✓		Automated tests regression	7 hr 39 min [redacted]	2 days 7 hr [redacted]	12 min	Jenkins	142 / 142 pass
	☼	BitBucket: robot-tests	1 min 26 sec log	N/A	1.5 sec		
✗	☂	Create test environment	10 mo #6	10 mo #8	8 min 59 sec	Jenkins	
✗	☂	Framework 2.0	1 yr 0 mo #4[redacted]	10 mo #4[redacted]	5.8 sec	Jenkins	8 / 18 pass
✗	☂	Free Run Pipeline	22 days #7	1 day 18 hr #8	29 sec		
!	☂	[redacted]	4 days 10 hr #541	N/A	2 hr 38 min	Jenkins	9 / 248 pass
!	☂	[redacted]	4 days 10 hr #539	1 yr 5 mo #440	7 hr 1 min	Jenkins	23 / 284 pass
!	☂	[redacted]	4 days 10 hr #539	N/A	4 hr 53 min	Jenkins	7 / 347 pass

allure-report.zip

Figure 6a Total reports

The above figure 6a total report shows us that with the new automation framework the pass percentage for the tests became 100% (142/142 tests passed) while with old framework total tests it was just 4.4% (39/879 tests passed). These values clearly show us that the new approach is better, easy to follow and much more reliable as compared to old tests.



Not secure | [redacted].local/blue/organizations/jenkins/Automated%20tests%20regression/detail/Automated%20tests%20regression/289/tests

✓ Automated tests regression < testauto_#289

Branch: –
Commit: –
12m 28s
7 hours ago
Changes by [redacted]
Started by timer

All tests are passing

Nice one! All 142 tests for this pipeline are passing.

Passed - 142

✓ C78809629 Case metadata	<1s
✓ C78809630 Documents	<1s
✓ C78809631 Invoice	<1s
✓ C78809632 Invoice metadata	<1s
✓ C78809633 Members	<1s
✓ C78809634 Multimedia	<1s
✓ C78809635 Office	<1s
✓ C78809636 Task endopint	<1s
✓ C78809637 Task metadata	<1s
✓ C78809638 Teams	<1s
✓ C98319470 Case contacts	<1s
✓ C84566607 Get All Versions	<1s
✓ C84566607 Get All Versions	<1s
✓ C69393977 Verify If My Calendar Is Enabled	2s
✓ C69393977 Verify If My Calendar Is Enabled	3s
✓ C69393978 Verify If Project Calendar Is Enabled	2s

Figure 6b Detailed report

The above figure 6b shows clearly the areas covered for the automated test in the new framework. It is easily readable and gives the information quickly about execution time, functional areas, fail/pass percentage etc.

As discussed previously, all the tests in the new framework have been kept independent of each other so that if there is one test failing the others should not be affected.

The new framework strictly follows the Automation Guidelines attached as Appendix to this thesis. These guidelines were not followed in the old framework and hence lot of things were improved because of following the guidelines.

Jenkins integration was fully implemented and with the help of using tags, it was possible to run the selective tests for example smoke, tests of only of specific module or full regression suite depending upon the test scenario or requirement in the new setup using Jenkins but in the old setup only basic integration was present which was triggering the nightly tests only as full regression suite.

TestRail the test management tool was integrated with the new test framework in such a way that when the automated scripts were created in Pycharm a place holder ticket was automatically created in TestRail, so manual work was much reduced. This tool was not available with the old framework.

Reporting has been improved in the new test framework due to Allure report plugin implemented in Jenkins. These reports provide clear information about the test failures, as well as screenshots were also captured for any failures due to which debugging was very convenient. The old test suite did not have any reporting tools and screenshots were also not captured due to which debugging was very difficult.

7 Conclusion

As we see from all these comparison points, the new test framework implementation was quite helpful for the case company, and it improved the overall test automation status for the product.

These improvements were highly appreciated by the business stakeholders and the peers. The new test framework has been accepted by the company and all the old tests will eventually be rewritten as per the new framework structure. The reports are analysed every day and feedback is provided to the developers quickly for any failures occurring in the code.

The test coverage has been improved to a great extent because of the new framework and all the tests written are maintained regularly as per any latest changes in the code.

8 References

1. [1] Testing is Tedious, https://medium.com/@hanif.arkan/testing-a-sometimes-tedious-but-crucial-part-of-software-development-23923fa4aea1, Accessed 12 Sep 22
2. [2] Testing is time consuming, https://www.sciencedirect.com/science/article/pii/S1877050916001277, Accessed 12 Sep 22
3. [3] Software release life cycle, https://theproductmanager.com/topics/software-release-life-cycle/, Accessed 12 Sep 22
4. [4] Test automation, https://semaphoreci.com/blog/test-automation, Accessed 13 Sep 22
5. [5] Why automation testing is necessary https://www.cprime.com/resources/blog/why-automation-testing-necessary/. Accessed on 27 July 2021
6. [6] Sprints cycle, https://link.springer.com/article/10.1007/s11219-021-09561-2, Accessed 13 Sep 22
7. [7] Test coverage, https://stackify.com/test-coverage-in-software-testing-its-relevance-important-techniques-to-take-note/. Accessed 28 July 2021
8. [8] Python, https://opensource.com/resources/python. Accessed 15 Aug 2021
9. [9] Page Object model approach, https://testersdock.com/robot-framework-page-object-model/. Accessed 03 April 2022
10. [10] Robot Framework, https://robotframework.org/. Accessed 03 May 2022
11. [11] Web application testing tools, https://www.xenonstack.com/insights/web-application-automated-testing-tools. Accessed 31 May 2022.
12. [12] https://www.bunnyshell.com/blog/what-is-test-automation, Accessed 13 Sep 22
13. [13] Types of software testing, https://medium.com/edureka/types-of-software-testing-d7aa29090b5b, Accessed 13 Sep 22
14. [14] Reasons to do automation, https://www.cloudbees.com/blog/5-reasons-for-automated-testing. Accessed 31 May 2022
15. [15] Java, https://www.oracle.com/java/technologies/. Accessed 03 June 2022

16. [16] Test coverage,https://stackify.com/test-coverage-in-software-testing-its-relevance-important-techniques-to-take-note/. Accessed 28 June 2022

17. [17] Keyword driven testing, https://www.ranorex.com/keyword-driven-testing/. Accessed on 18 June 2022

18. Mojtaba Shahin, Muhammad Ali Babar, Liming Zhu. Continuous integration delivery and deployment: a systematic review on approaches, tools, challenges, and practices. Australia: 2017.

19. Tom Lecklider. Software testing features agile footwork. July 2014

20. Christian Bonnin, Atmel. Advances in Test Program Automation. July 2007

21. Jain Prateek, Non-functional Test Automation for Windows Phone Apps. April 2016

22. Chithra Prabha Peachi Muthu. Visibility of the project status, usage of agile methods and tools. October 2014

23. Singh Karan, Web Application Performance Requirements Deriving Methodology. May 2016

24. Venalainen Pekka, Detecting Software License Violations. May 2021

25. Bezirganoglu Sefika, Securing Cloud with Palo Alto Networks Firewalls, November 2020

26. Salama Risto, “Down with Regression”-Generating Test Suites for the Web, April 2020

27. Brinkmann Eliza, Adjustable automation for the homebrewing process, August 2021

28. Jaaksola Mikko, Software testing failures through the history and how to prepare for them, December 2018

29. Knaappila Jani, Measuring Structural Software Quality, November 2020

30. Lukkarinen Pasi, Data Center Automation- and Hybrid Cloud System Requirements, May 2020

9 Appendix 1

This appendix presents the test automation standards and guidelines documented by Robot Framework community, same has been referred in the thesis work on continuous basis.

Automation Standards and Guidelines

Pull requests and peer reviews

In Testcases repository, standard GitLab pull request process for shared repositories should be followed for merging to any protected branches (at the time of the writing, master). Following additions to GitHub PR process apply:

- Before submitting the request, rebase from master to make sure merge is clean
- Before merging, an approved review from a person other than the author is required
- Same code of conduct is followed as in software development.
- Merge to master with squash, this will keep the version history clean and linear
- Delete unneeded branches after merge
- Adding a link to a successful test run against the branch to be merged is recommended

Code review process

- Peer review of automation code is highly recommended and one of the best practices in the industry. It ensures that all members of the team adhere to the published standards and best practices with regard to developing automated tests.
- When an automation engineer is done with developing and unit testing an automated test, he/she needs to have a peer review the script.
- A reviewer should review the code/script against all standards and best practices published in the “Automation Standards and Guidelines” section of the Test Handbook.

Checklist for reviewing automated script/code:

- Folder structure should be followed
- Folder naming conventions should be followed
- Dynamic objects should be handled correctly.
- Naming conventions followed for variables
- Naming conventions followed for constants
- Naming conventions followed for environment variables
- Naming conventions followed for functions (common functions, POM)
- Function headers are created correctly (common functions, POM)
- Comments added according to standards
- Indentation followed according to standards
- Only one excel file used per application for storing test data
- Only recommended test data related functions from common function library used
- Handled known errors correctly
- Contents of configuration files is correct

- Test header is created correctly
- Re-used existing functions where applicable, instead of re-creating them
- Re-usable blocks of code have been converted into functions

Review the code according to the checklist given above and provide feedback/ comment on the merge request to the owner of the test. The owner then makes changes/modifications and submits for the review again. This cycle continues until all the items in the checklist have passed.

Git Code review process*:*

- Push your branch with created/updated test to the repository on GitLab
- Go to GitLab / project
- You'll see a new notification that you pushed changes about minute ago
- Create merge request button
- Give some description, select the reviewer and submit merge request
- After review - apply changes/give comments if needed
- Rebase your branch and commit all fixes. Make merge request again.
- After approved review, merge your test branch to master - check out on master (pull latest code) and merge with your branch. Resolve all conflicts and push to master.
- Go to GitLab and remove your test branch.

Coding standard(s)

Robot Framework coding conventions:

- Use plain text format and .robot file extension for your robot test and keyword files.
- Use robot.tidy utility (it is bundled with Robot Framework) before committing your changes:

  ```
  python –m robot.tidy --recursive
  ```

  This will harmonize the formatting of robot files.

- Every robot test suite must have the default test timeout set. A common default is defined in variable ${ROBOT_DEFAULT_TIMEOUT}. When the default value is not suitable, use your own.
- Every suite should have a documentation section where the general scope of the suite is briefly explained.
- User keywords related to a component should be kept in component-specific resource file, for example `pipeline_helpers.robot`, `project_helpers.robot`. Keyword names should be self-documenting. If the keyword has parameters or return values, these should be described in the documentation.
- Command-line variables should not be added unless absolutely necessary. Use variable files when the values of variables are environment or version specific. Browser-specific functionality must rely on the ${BROWSER} command-line variable. The tests must be kept runnable against any browser without local modifications.

- If you add comments within the robot tests, use # format, not the Comment keyword (which will also write your comment to the test log.)

Python coding conventions:

Basic rules:

- Naming conventions:
  lower_case_with_underscores
  used for :
  - function names
  - parameter names
  - package names
  - module names
  - project file and folder names
  - folder names
  - local variables

  UPPER_CASE_WITH_UNDERSCORES
  used for:
  - global variables
  - constants defined in the class

  UpperCamelCase
  used for:
  - defining the class name
  - defining exceptions (they should be classes)

- Indentations:
  - don’t mix a Tab with four spaces
  - 4 spaces are recommended
  - Tab is allowed (but if you start it is better to use spaces)
  - you can set the environment to replace the tabulator with 4 spaces

- Maximum line length:
  - limit to all lines to a maximum of 79 characters
  - the line length should be limited to 72 characters in docstrings or comments
  - only in exceptions where the xpath is very long then the maximum line length can be exceeded.

- Line spacing:
  - surround top-level function and class definitions with two blank lines
  - method definitions inside a class are surrounded by a single blank line
  - add blank line in the end of the file with code

- Imports*:*
    - imports are always at the top of the file, just after any module comments and docstrings, and before module globals and constants
    - imports should be grouped in the following order:
        1. standard library imports
        2. related third party imports
        3. local application/library specific imports

- imports should usually be on separate lines for example

```
# Correct:
import os
import sys

# Wrong:
import sys, os

# Correct:
from subprocess import Popen, PIPE
```

Whitespace in Expressions and Statements:
Avoid extraneous whitespace in the following situations:

- Immediately inside parentheses, brackets or braces:

```
# Correct:
spam(ham[1], {eggs: 2})

# Wrong:
spam( ham[ 1 ], { eggs: 2 } )
```

- Between a trailing comma and a following close parenthesis:

```
# Correct:
foo = (0,)

# Wrong:
bar = (0, )
```

- Immediately before a comma, semicolon, or colon:

```
# Correct:
if x == 4: print(x, y); x, y = y, x

# Wrong:
if x == 4 : print (x , y) ; x , y = y , x
```

- However, in a slice the colon acts like a binary operator, and should have equal amounts on either side (treating it as the operator with the lowest priority). In an extended slice, both colons must have the same amount of spacing applied. Exception: when a slice parameter is omitted, the space is omitted:

```
# Correct:
ham[1:9], ham[1:9:3], ham[:9:3], ham[1::3], ham[1:9:]
ham[lower: upper], ham[lower: upper:], ham[lower::step]
```

```
ham[lower+offset : upper+offset]
ham[: upper_fn(x) : step_fn(x)], ham[:: step_fn(x)]
ham[lower + offset : upper + offset]

# Wrong:
ham[lower + offset: upper + offset]
ham[1: 9], ham[1 :9], ham[1:9 :3]
ham[lower : : upper]
ham[ : upper]
```

- Immediately before the open parenthesis that starts the argument list of a function call:

```
# Correct:
spam(1)

# Wrong:
spam (1)
```

- Immediately before the open parenthesis that starts an indexing or slicing:

```
# Correct:
dct['key'] = lst[index]

# Wrong:
dct ['key'] = lst [index]
```

- More than one space around an assignment (or other) operator to align it with another:

```
# Correct:
x = 1
y = 2
long_variable = 3

# Wrong:
x             = 1
y             = 2
long_variable = 3
```

Other rules:

- Strings always in single or double quote. Remember to use the same style in all your code.
- Before each function use function definition (PEP 257) eg.

```
def function():
    """"Docstring documenting the function - definition"""

    print('This is the function')
```

Source documentation (PEP 8):

Following PEP-8 (https://www.python.org/dev/peps/pep-0008/) is recommended.

Keyword conventions

1. Keyword names should be descriptive and clear to be easily understandable.
2. Keywords will explain the functionality, not how it does its task(s).
3. Keywords should have different abstraction levels (for example Input Text or Administrator logs into system) to clearly define what it is doing

4. A keyword should be fully title cased or have only the first letter be capitalized.

   a) Title casing is often used when the keyword name is short (for example Input Text).

   b) Capitalizing just the first letter typically works better with keywords that are like sentences (for example Trainer logs into system) which are mainly considered higher level.

Function naming conventions:

- Function names should be named in lower_case_with_underscores or short names
- Function names should contain only alphanumeric characters (only English alphabet). Spaces and special characters should NOT be used
- Function name cannot be reserved words
- Function names cannot begin with a number
- Function names should be descriptive enough to understand what it will do
- Function names should begin with a verb such as find, get, check, verify, compare, delete, remove, and so on
- Function name should not exceed 10 words. Three to five words would be ideal
- Words in the function name should be continuous without any spaces or underscores. Each new word in the function name should begin with an upper-case letter

IMPORTANT NOTES:

- If you create variables inside functions, you must always declare them using the correct datatype to confine the scope of the variables to that function
- All function arguments in the function definition should be defined using correct datatype
- All variables used with in a function must be declared using the keyword of correct datatype.

Naming convention (files, keywords, tests, linking, tagging in robot tests):

Tagging of robot tests has two uses: reporting of test cases and selecting/excluding subsets of tests for execution (robot options - and -e). Tagging policy is summarized by the Following table.

Robot Tag	Meaning
not_ready	Test case will not be executed in CI. Test needs adaptation to SUT changes
noncritical	Unstable test case, which may fail occasionally. Needs to be fixed.
UC_Comp_6	Tests use case 6 of component Comp. Every test should have a use case tag
component	Component the test suite is testing. Every test suite should have one component tag.
positive	Positive test case
negative	Negative test case

10 Appendix 2

This appendix presents the document which was prepared and presented to the project team listing the reasons for changing and implementing a new test framework.

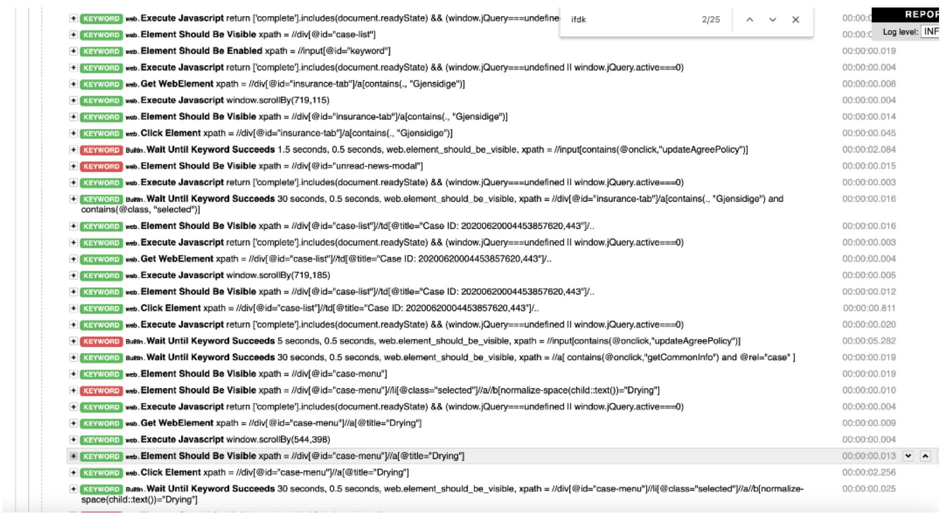
New Test Framework

New Framework will be based on Robot Framework and Python. It will utilize the best practices and latest tools to support test automation.

The main purpose is to create it in a way that will enable us to integrate with other tools and develop CI&CD.

Main reasons for switching from legacy framework to the new one

- Unreadable reports, when all the atomic steps are not grouped and follow one-by-one forming list of 200-300 lines
- Unparalleled execution
- Poor performance and efficiency – long execution time, lots of redundant waits and code duplications
- Huge result files – loading them takes time
- Lot of errors & warnings
- No abstraction level - all tests contain dozens of waits, assertions, clicks, screenshots under 1 nesting level
- Hardcoded data
- Long execution time
- Low passed rate - around 25-45%
- Too much fragmentation done to suites - no visible and understandable separation



```
+ KEYWORD web.Execute Javascript return ['complete'].includes(document.readyState) && (window.jQuery===undefine     00:00:0
+ KEYWORD web.Element Should Be Visible xpath = //div[@id="case-list"]                                                  00:00:0
+ KEYWORD web.Element Should Be Enabled xpath = //input[@id="keyword"]                                                 00:00:00.019
+ KEYWORD web.Execute Javascript return ['complete'].includes(document.readyState) && (window.jQuery===undefined II window.jQuery.active===0)   00:00:00.004
+ KEYWORD web.Get WebElement xpath = //div[@id="insurance-tab"]/a[contains(., "Gjensidige")]                           00:00:00.008
+ KEYWORD web.Execute Javascript window.scrollBy(719,115)                                                              00:00:00.004
+ KEYWORD web.Element Should Be Visible xpath = //div[@id="insurance-tab"]/a[contains(., "Gjensidige")]                00:00:00.014
+ KEYWORD web.Click Element xpath = //div[@id="insurance-tab"]/a[contains(., "Gjensidige")]                            00:00:00.045
+ KEYWORD Builtin.Wait Until Keyword Succeeds 1.5 seconds, 0.5 seconds, web.element_should_be_visible, xpath = //input[contains(@onclick,"updateAgreePolicy")]   00:00:02.084
+ KEYWORD web.Element Should Be Visible xpath = //div[@id="unread-news-modal"]                                         00:00:00.015
+ KEYWORD web.Execute Javascript return ['complete'].includes(document.readyState) && (window.jQuery===undefined II window.jQuery.active===0)   00:00:00.003
+ KEYWORD Builtin.Wait Until Keyword Succeeds 30 seconds, 0.5 seconds, web.element_should_be_visible, xpath = //div[@id="insurance-tab"]/a[contains(., "Gjensidige") and   00:00:00.016
  contains(@class, "selected")]
+ KEYWORD web.Element Should Be Visible xpath = //div[@id="case-list"]//td[@title="Case ID: 20200620004453857620,443"]/..   00:00:00.016
+ KEYWORD web.Execute Javascript return ['complete'].includes(document.readyState) && (window.jQuery===undefined II window.jQuery.active===0)   00:00:00.003
+ KEYWORD web.Get WebElement xpath = //div[@id="case-list"]//td[@title="Case ID: 20200620004453857620,443"]/..        00:00:00.004
+ KEYWORD web.Execute Javascript window.scrollBy(719,185)                                                              00:00:00.005
+ KEYWORD web.Element Should Be Visible xpath = //div[@id="case-list"]//td[@title="Case ID: 20200620004453857620,443"]/..   00:00:00.012
+ KEYWORD web.Click Element xpath = //div[@id="case-list"]//td[@title="Case ID: 20200620004453857620,443"]/..         00:00:00.811
+ KEYWORD web.Execute Javascript return ['complete'].includes(document.readyState) && (window.jQuery===undefined II window.jQuery.active===0)   00:00:00.020
+ KEYWORD Builtin.Wait Until Keyword Succeeds 5 seconds, 0.5 seconds, web.element_should_be_visible, xpath = //input[contains(@onclick,"updateAgreePolicy")]   00:00:05.282
+ KEYWORD Builtin.Wait Until Keyword Succeeds 30 seconds, 0.5 seconds, web.element_should_be_visible, xpath = //a[ contains(@onclick,"getCommonInfo") and @rel="case" ]   00:00:00.019
+ KEYWORD web.Element Should Be Visible xpath = //div[@id="case-menu"]                                                 00:00:00.019
+ KEYWORD web.Element Should Be Visible xpath = //div[@id="case-menu"]//li[@class="selected"]//a//b[normalize-space(child::text())="Drying"]   00:00:00.010
+ KEYWORD web.Execute Javascript return ['complete'].includes(document.readyState) && (window.jQuery===undefined II window.jQuery.active===0)   00:00:00.004
+ KEYWORD web.Get WebElement xpath = //div[@id="case-menu"]//a[@title="Drying"]                                        00:00:00.009
+ KEYWORD web.Execute Javascript window.scrollBy(544,398)                                                              00:00:00.004
+ KEYWORD web.Element Should Be Visible xpath = //div[@id="case-menu"]//a[@title="Drying"]                             00:00:00.013
+ KEYWORD web.Click Element xpath = //div[@id="case-menu"]//a[@title="Drying"]                                         00:00:02.256
+ KEYWORD Builtin.Wait Until Keyword Succeeds 30 seconds, 0.5 seconds, web.element_should_be_visible, xpath = //div[@id="case-menu"]//li[@class="selected"]//a//b[normalize-   00:00:00.025
  space(child::text())="Drying"]
```

Requirements

High level

- Parallel execution with flexible number of threads
- Multi-browser support (we still have to support IE 11)
- Mobile tests support (Android, iOS, Win Mobile)
- ?API tests support?
- Integration with TestRail and Jira
- Test data upload to env before testing
- Test data clean-up (redundant cases, members and other data) vs. snapshot for testing

Low level

- Code guidelines supported with "monitoring" tool (linter)
- Merge checks: static code analysis (linter), automatic execution
- Self-documenting code

BrowserLibrary vs Selenium - Rewrite robot tests to use Browser library instead of Selenium

CICD

- History of executions
- Parallel executions