TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Informatics

IDU40LT

Jana Salnikova 095163IABB

IMPLEMENTATION OF AUTOMATED INTEGRATION TESTING USING RESTFUL API ON THE EXAMPLE OF SAFFRON DIGITAL LTD

Bachelor's thesis

Supervisor: Ants Torim
PhD
Lecturer

Tallinn 2016

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut

IDU40LT

Jana Salnikova 095163IABB

AUTOMATISEERITUD INTEGRATSIOONI TESTIMISE JUURUTAMINE KASUTADES RESTFUL API-T SAFFRON DIGITAL LTD NÄITEL

Bakalaureusetöö

Juhendaja: Ants Torim
PhD
Lektor

Tallinn 2016

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Jana Salnikova

22.05.2016

Abstract

The main objective of this work was to implement structured framework for API level testing to solve Saffron Digital's challenges introduced by the currently followed test automation process.

To conclude if the implementation was successful, ROI computations were made to find out test automation value for newly implemented approach and currently used one, and compare the results.

This work showed that implementation of API level testing was successful as over the ROI period of 12 months, for every Pound Sterling invested in API test automation will return 45% and for every Pound Sterling invested in existing test automation process will cause a loss of almost 50%.

This thesis is written in English and is 42 pages long, including 6 chapters, 18 figures and 3 tables.

Annotatsioon

Automatiseeritud integratsiooni testimise juurutamine kasutades RESTful API-t Saffron Digital LTD näitel

Käesoleva töö peamiseks eesmärgiks oli juurutada struktueeritud raamistik API tasemel integratsiooni testide arendamiseks, mis aitaks lahendada ettevõttes praegusel hetkel kasutusel oleva automatiseeritud testimise protsessi puudusi.

Järeldamaks, kas raamistiku juurutamine osutus edukaks, arvutati automatiseeritud testimise tasuvus nii uuele kui ka olemasoleva protsessile, kasutades investeeringutasuvuse meetodit.

Töö tulemusena selgus, et kaheteistkümne kuu pikkuse investeeringutasuvuse perioodi jooksul saab ettevõte API integratsiooni testimiselt 45% kasu iga investeeritud naelsterlingu pealt ning peaaegu 50% kahju investeerides olemasolevasse protsessi. Antud tulemuste põhjal võib väita, et Saffron Digital-i näitel osutus API tasemel testimise juurutamine edukaks.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 42 leheküljel, 6 peatükki, 18 joonist, 3 tabelit.

List of Abbreviations and Terms

API	Application Program Interface
AST	Automated Software Testing
CI	Continuous Integration
GUI	Graphical User Interface
IDE	Integrated Development Environment
IT	Information Technology
OTT	Over-The-Top
QA	Quality Assurance
REST	Representational State Transfer
ROI	Return on Investment
SVOD	Subscription Video on Demand
UI	User Interface
URL	Uniform Resource Locator

Table of Contents

List of Figures

List of Tables

1 Introduction

In today's fast moving world, it is challenging for any company to continuously maintain and improve the quality of software development as in many cases it is very time consuming and expensive.

Over the years automated software testing (AST) has become one of the main topics related to QA in IT industry amongst software development and testing communities.

Test automation can improve the development process in many cases if applied, modified and used according to the company's needs.

But since there are so many testing methodologies and levels that can be used as part of AST, how to decide which methods should be used and how to balance them?

This document focuses on analysing current testing strategy of Saffron Digital LTD from a functional testing aspect.

1.1 Background

Saffron Digital LTD provides premium video platform that enables its clients to launch premium multi-platform over-the-top (OTT) entertainment service.

Since the company was founded in 2006 it has mainly focused on testing its software by executing manual end-to-end as well as automated unit tests. Throughout the past few years Saffron Digital has tried out many tools and programming languages that would allow its QA team to implement end-to-end UI automation tests without having much programming experience, but unfortunately it has not seen any success.

At the beginning of 2016 yet another decision was made to start automating regression test suite as end-to-end UI tests but this time using Selenium WebDriver. Amongst company's stuff it is strongly believed that simulating real user scenarios can help easily determine how a failing test would impact the user.

1.2 Objectives

The objective of this work is to introduce Mike Cohn's testing pyramid theory and based on that analyse testing strategy followed in Saffron Digital. To address raised issues with the current approach, implement and set up a structured test framework for automated API integration testing. Compute and understand the value of test automation for current UI and new API level approaches both compared to manual testing. Analyse the results and conclude if API test framework implementation was successful, is it worth future investments and Mike Cohn's theory can be applied to Saffron Digital's testing approach.

1.3 Methodology

To achieve the objectives a combination of return on investment (ROI) and action research methodology principles will be used.

Implemented test automation framework will be acting as a pilot project that will help to collect data, which will then be analysed and used in computations to determine the value of test automation for both UI and API level approaches.

1.4 Overview

The first part of the document includes an overview of test automation as well as introduction of test automation pyramid theory and its layers.

The second part introduces company's background, an SVOD service it provides as well as its testing strategy.

The third part describes the process of implementing automated test framework for API integration testing. It also includes a list of used tools with explanations and code examples why particular tools were chosen.

The final chapter of this work will focus on describing and using ROI methodology in computations of test automation values for both UI and API approaches. Acquired results will be analysed and a conclusion will be made if implementation of test automation framework on the example of Saffron Digital has been successful and should be considered for further use as part of the company's test strategy.

2 Automated Software Testing

In an automated software testing process, software tools execute pre-developed tests on a software application before it is released into production.

“The overall objective of AST is to design, develop, and deliver an automated test and retest capability that increases testing efficiencies; …” [1]

In order to gain benefits from test automation, the tests to be automated need to be carefully selected and implemented, as automated quality is independent of test quality [2].

It is also very important to understand on which level test automation should be performed. That is most commonly based on the environment, technology or simply the way the company works.

2.1 Test Automation Pyramid

An effective test automation strategy calls for automating tests at three different levels, as shown in Figure 1, which depicts the test automation pyramid [3].

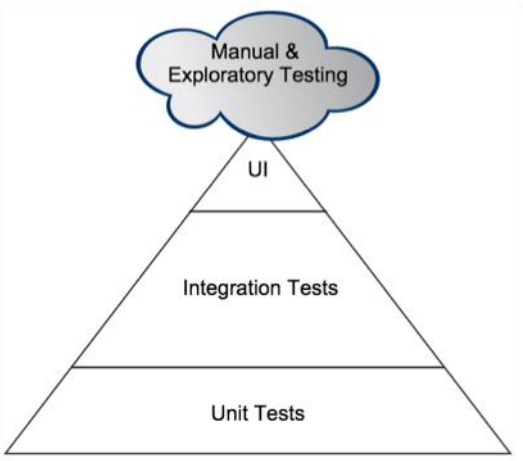

Figure 1. Test automation pyramid

According to the Figure 1 unit tests should cover the broadest area of the pyramid and then be followed by integration and UI tests. Since not all tests can be automated, a certain amount of manual testing will always be present as a cloud above the pyramid.

As with building any kind of real life object you need to have a strong foundation that will help in supporting the next levels built on top of it. The same applies to test automation pyramid – it is essential to have good testing coverage at the lower levels of your software as if this part is poorly tested, automation on higher levels will become very expensive and time consuming.

2.1.1 Unit Tests

Unit testing is a software development process in which the smallest possible piece of a program is tested individually and independently, verifying that it works as expected, without considering what the rest of the program would do. This protects each unit from inheriting bugs from mistakes made elsewhere, and makes it easy to narrow down on the actual problem [4].

Unit tests are usually written by developers, before the code, to define the functionality, however they evolve and are extended as coding progresses.

2.1.2 Integration Tests

Once program units are solid, it is necessary to test that the things that are built out of them also work correctly together, rather than in isolation [4]. This process is called integration testing.

Integration testing is represented as the middle layer of test automation pyramid (Figure 1) and is focused on testing the services of an application separately from its user interface (UI) and is also known as service-level testing [3].

Ability to access applications without UI allows testing the core, code-level functionality of the application by providing an early evaluation of its overall build strength before running any UI tests. This helps expose the small errors that can fester and become larger during the following testing stages.

As integration testing itself can be done on many different levels, the middle layer is often split in multiple layers (Figure 2).

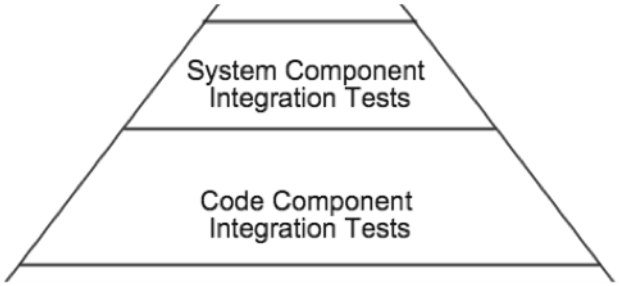

Figure 2. Middle layer split between system and code component integration tests

Integration tests at the code component level are designed to ensure that the code units or code components that need to work with each other do so in expected ways [5].

Integration tests at the system component level are designed to ensure that the system components that need to interact with each other can do so as intended. These tests are designed and executed against application programming interfaces (APIs), any interfaces exposed between system components or 3rd party services/components involved, which allows to test different variations and permutations of API calls [5].

API testing involves testing APIs directly and as part of the end-to-end transactions.

2.1.3 UI Tests

Coded UI tests are automated tests that drive applications through its user interface. These tests include functional testing of the UI controls and verify that the whole application, including its user interface, is functioning correctly. Automation on this level should be done for the functionality that requires minimal change.

Automated UI testing is placed at the top of the test automation pyramid and should be done as little as possible, as they are known to be more brittle, expensive and time consuming to write and execute [3]. Ideally only the tests that are critical for the business or can’t be covered by the lower levels should be automated using this approach.

The focus should be to minimise these automated tests by relying on and building on the successes of the testing in the layers below.

3 Test Automation of Subscription Service on the Example of Saffron Digital LTD

3.1 Company's Background

Saffron Digital is a cloud-based digital content management and delivery platform for providing premium multi-platform OTT services.

The company's state-of-the-art open virtual platform (OVP), MainStage (Figure 3), is an industry-leading end-to-end platform for the distribution of digital video, featuring content preparation including video transcoding, a backend content management system, storefront services coupled with a digital locker for consumer purchases, a secure DRM player for high-quality playback and a multi-platform application framework [6].

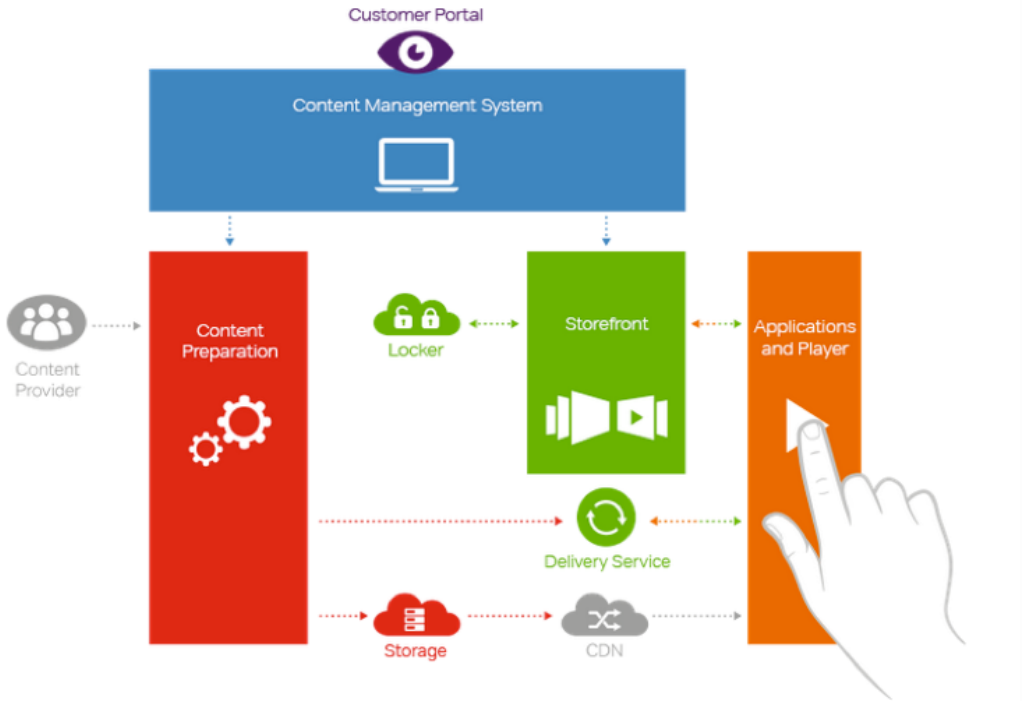

Figure 3. Saffron Digital's MainStage platform

3.2 Subscription Service

A subscription is a contract in which a customer pays a subscription price in regular intervals (monthly, yearly or seasonal) to have access to a product or a service [7]. This means that a one-time sale of a product can become a recurring sale and can build brand loyalty.

One of Saffron Digitals main business models is to provide an end-to-end platform, of their own implementation, for their clients to be able to run such a service.

Currently the company is providing an end-to-end subscription service with integrated applications across 3 platforms (web, iOS, Android) for two of their clients. The third client is using storefront subscription service APIs across 2 platforms to integrate with their own applications.

3.3 Current Test Strategy

MainStage has been built in a way where every subscription service has its own individual client-based configurations, but they are all using the same storefront API gateway by calling the same services. This allows the company to sell already existing features to multiple clients as well as set various configurations according to each client's business needs.

This type of approach is very beneficial from the development and client on-boarding point of view, but what about testing?

With manual testing approach that the company has been following, it has become very expensive and time consuming to test developed software, especially as the client base is growing and storefront is constantly changing by supporting more and more new features.

When making changes to any of the backend microservices that are used by the subscription service, the QA team needs to retest the service for all existing customers. As a result manual regression testing is taking on an average of seven man-days which doesn't leave much time for possible bug fixes and re-test cycles in a continues integration development approach where releases are made every week.

As company's QA team is small and freezing code releases is not possible, a decision was made to start automation of UI regression tests, starting with web platforms, as and when there was down time from manual testing. Selenium WebDriver and Python programming language were chosen as tools to be used.

By following this AST method Saffron Digital started moving towards hourglass test automation approach (Figure 4) having no coverage of server level testing, yet MainStage is a server based platform and the main scope of the business.

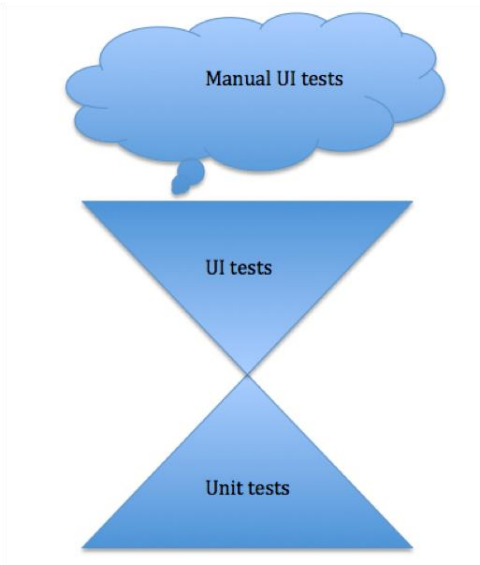

Figure 4. Hourglass test automation approach

Currently followed test automation plan is outlined in Table 1. The overall strategy is to automate all main scenarios for three existing subscription clients.

Table 1. Saffron Digital's test automation plan for subscription services using Selenium WebDriver

Client	**Average number of test cases per platform**	**Number of platforms (web, iOS, Android)**	**Number of tests that can and should be automated using Selenium**	**Total number of test cases to be automated**
Client 1	42	3	28	87
Client 2	42	2	0	0
Client 3	39	3	28	84

Based on the data presented in Table 1 there are about 171 tests to be developed in order to have good test coverage across two clients. There can be no Selenium tests done for

Client 2, as Saffron Digital provides only server side APIs and does not develop their applications, meaning the client could change the UI at any time thus breaking front end tests that rely on it.

Looking at the data above there are three main concerns that arise:

1. Too many tests to develop and later on maintain across different platforms
2. Reusing existing tests is not possible due to different implementation of the UI
3. Regression testing cannot be done for clients who are not using company's applications

4 Implementation of Automated API Integration Test Framework

A framework is considered to be a combination of set protocols, rules, standards and guidelines that can be incorporated or followed as a whole [8].

A test automation framework is an environment in which tests are automated and executed. It is a set of guidelines, coding standards, concepts, processes, practices, project hierarchies, modularity etc. that help to support automated testing [8].

Main test automation framework goals and objectives:

- Create a mechanism to drive the application under test
- Ability to develop test cases in human readable format by hiding the code logic behind the callable script/function/module
- Ability to create test cases that are independent of automated test scripts/functions/modules – no cross-impact if either one is changed
- Easy to use when developing new test cases
- Easy to execute tests
- Application independent
- Have capability to expand with the requirements of each application
- Easy to maintain
- Easy to report test results

[9]

An organized test framework helps in avoiding duplication of test cases automated across the application as well as to improve efficiency of testing.

Automation frameworks can be classified according to five broad types:

- Test script modularity framework
- Test library architecture framework
- Data driven framework

- Keyword driven framework
- Hybrid framework

[10]

Each type comes with its own advantages and disadvantages hence when choosing one clear objectives must be set. “Making the right choices in the preliminary design stage is the most critical step of the process, since this can be the differentiator between a successful framework and failed investment.” [10].

There were two primary objectives when choosing and implementing server-level test automation framework for Saffron Digital:

4. Find a solution to existing test automation concerns listed at the end of the chapter 3.3 on the page 19
5. Make the framework easy to use and understand for a user who has superb understanding of QA and a broad domain knowledge, but little programming experience

The following chapters will introduce all main elements that form the framework.

4.1 Python and PyCharm

“Python is an interpreted, object-oriented, high-level programming language with dynamic semantics.” [11]. It uses dynamic typing and by its design implements deliberately simple and readable syntax, which makes it easier to learn, understand and remember [12].

As Saffron Digital’s MainStage platform was built using Python and since it has been getting more popular as the first language to teach novices [13] it seemed only right to implement server-level test framework using the same programming language the platform itself was built in. This also allows daily support from company’s development team in case of any questions or faced difficulties, which helps to speed up QA team’s self-learning and test development process as well as save money on trainings.

PyCharm IDE version 4 is used as a programming environment for Python. It provides code analysis and graphical debugger that are necessary for a novice to get used to coding faster and avoid making mistakes.

4.2 Python pytest Tool

Although there are many known Python testing frameworks, each with their own pros and cons, it didn't take long to choose one, that right away seemed the most suitable for Saffron Digital's needs.

This chapter introduces pytest – an open-source Python testing tool. I will outline its main features that became the determining factors for picking this particular tool.

When choosing the automation testing tool there were a number of requirements to satisfy:

- Simplicity in developing tests
- Simple test discovery
- Ability to reuse already existing tests to run them for multiple clients, territories and platforms as and when needed
- Ability to run tests only if a specific condition has been met
- Ability to test for exceptions

4.2.1 Test Development

The pytest framework accepts plain Python functions as tests (Figure 5) instead of insisting that tests must be packaged inside of larger test classes (Figure 6).

```
def test_false():
    assert False == 0


def test_true():
    assert True == 1
```

Figure 5. Test examples using pytest framework

```
import unittest


class TruthTest(unittest.TestCase):
    def test_false(self):
        self.assertEqual(0, False)

    def test_true(self):
        self.assertEqual(1, True)

if __name__ == '__main__':
    unittest.main()
```

Figure 6. Test example using unittest framework

As in pytest there is no actual need for test classes, tests can be simply grouped in modules that can then be used as test suites.

4.2.2 Test Development Using Fixtures

The purpose of test fixtures is to provide a fixed baseline upon which tests can reliably and repeatedly execute [14].

Main fixture features:

- They have explicit names and are activated by declaring their use from test functions, modules, classes or whole projects
- They are implemented in a modular manner, as each fixture name triggers a fixture function which can itself use other fixtures
- Their management scales from simple unit to complex functional testing, allowing to parameterize fixtures and tests according to configuration and component options, or to re-use fixtures across class, module or whole test session scopes

[14]

Test functions can receive fixture objects by naming them as an input argument. For each argument name, a fixture function with that name provides the fixture object. Fixture functions are registered by marking functions with `@pytest.fixture` and can be further extended to include scope `@pytest.fixture(scope="module")`. By providing the scope it is possible to control the level on which the fixture will be used. If the scope is set to module it will be shared across the whole test module, if set to function it will be executed per test function etc.

Using fixture functions is a prime example of dependency injection where fixture functions take the role of the injector and test functions are the consumers of fixture objects [14].

4.2.3 Test Discovery

As per test discovery pytest provides a very simple built in solution. Starting from the directory where it is run, it will find any Python module prefixed with `test_` and will attempt to run any defined function prefixed with `test_` found inside of it.

pytest explores properly defined Python packages, searching recursively through directories that include `__init__.py` modules. Figure 7 shows pytest test discovery.

```
vouchers/
    __init__.py
    single_use_voucher.py
    multi_use_voucher.py
    test_single_use_voucher.py   checked for tests
    test_multi_use_voucher.py    checked for tests
test_data/
    pytest won't look in this package because it lacks __init__.py
    vouchers.csv
    vouchers.py
    test_vouchers.py      skipped because test_data/ lacks __init__.py
__init__.py
main.py
test_main.py checked for tests
```

Figure 7. Test discovery used in pytest framework

4.2.4 Fixture Decorators for Parameterization and skipif

Since Saffron Digital's MainStage platform is used by different clients and can be configured according to each client's specific needs, it is important to be able to execute same tests for different clients and configuration sets, avoiding test repetitiveness.

The built in `pytest.mark.parametrize` decorator enables to achieve that by simply adding it before a test function (Figure 8).

```
import pytest


@pytest.mark.parametrize("value_1, value_2", [(8, 9),
                         (5, 5), (6, 5)])
def test_values(value_1, value_2):
    assert value_1 < value_2
```

Figure 8. pytest parameterise decorator applied to a test

Figure 9 shows test results of `test_values()` function executed in PyCharm IDE.

```
.F
value_1 = 5, value_2 = 5

    @pytest.mark.parametrize("value_1, value_2", [(8, 9),
(5, 5), (6, 5)])
    def test_values(value_1, value_2):
>       assert value_1 < value_2
E       assert 5 < 5

/projects/bachelor/src/tests/test.py:28: AssertionError

value_1 = 6, value_2 = 5

    @pytest.mark.parametrize("value_1, value_2", [(8, 9),
(5, 5), (6, 5)])
    def test_values(value_1, value_2):
>       assert value_1 < value_2
E       assert 6 < 5

/projects/bachelor/src/tests/test.py:28: AssertionError
F
```

Figure 9. Test results of parameterised test execution using both positive and negative scenarios

In some cases it may be needed to skip the whole test all together (Figure 10) or when a specific condition has been met (Figure 11). In this case pytest offers `pytest.mark.skip` decorator.

```
import pytest


@pytest.mark.skip(reason="no way of currently
testing this")
def test_the_unknown():
    ...
```

Figure 10. Test using pytest.skipif decorator without condition

```
import pytest
import sys


@pytest.mark.skipif(sys.platform == 'environment',
                    reason="Test doesn't run on this env.")
def test_function():
    ...
```

Figure 11. Test using pytest skipif decorator with specified condition

4.2.5 Exception Testing

In some cases, when testing an API, it is needed to test that the code throws the right exceptions when given invalid input, or if executed in an invalid state. `pytest.raises` can be used as a context manager (Figure 12).

```
import pytest


def test_of_exception_error():
    with pytest.raises(Exception) as e_info:
        x = 1 / 0
    assert e_info.typename == 'ZeroDivisionError'
```

Figure 12. Assertion of raised exception using pytest.raises as a context manager

`e_info` is an ExceptionInfo instance, which is a wrapper around the actual exception raised.

4.3 Slumber and Curling Libraries

Slumber is a Python library that provides a convenient yet powerful object orientated interface to RESTful APIs. It acts as a wrapper around the requests library and abstracts away the handling of URLs, serialization and processing requests [15].

Curling is a REST client that wraps slumber to provide a nice interface to consume tastypie APIs in Django.

By using these two libraries together, a RESTful API can be used by the example presented in Figure 13.

```
from curling.lib import API

api = API('http://slumber.in/api/v1/')
response =
api.note.get(headers={‘authorization_headers’})
```

Figure 13. RESTful API using curling library

4.4 Library Architecture Testing Framework

Library architecture is based on common functions that are placed in a common library. These functions can then be called in the test scripts as and when required across the whole application under test. The basic concept behind the framework is to determine the common steps, group them into functions and keep them in a library [8].

Let’s look at the login functionality as an example. There is a set of login feature related tests, but apart from that, login is a necessary precondition to other tests like viewing personal details, making a payment etc. By applying library architecture concept, login steps are grouped in a function and then called in any test it is applicable for (Figure 14).

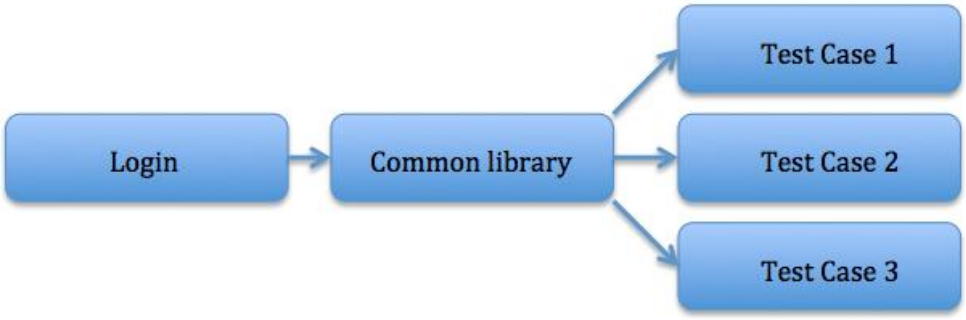

Figure 14. Simple illustration of library architecture test framework

Main pros of the framework:

- Introduces high level of modularization which leads to easier and cost efficient maintenance and scalability
- By creating common functions that can be efficiently used by the various test scripts across the whole framework a great degree of reusability is introduced

Main cons of the framework:

- The test data is submitted into the test scripts, thus any change in the test data would require changes in the test script as well

4.4.1 Saffron Digital's API Test Automation Framework

This chapter describes implemented API integration test framework structure with a UML component (Figure 15) and sequence (Figure 16) diagrams.

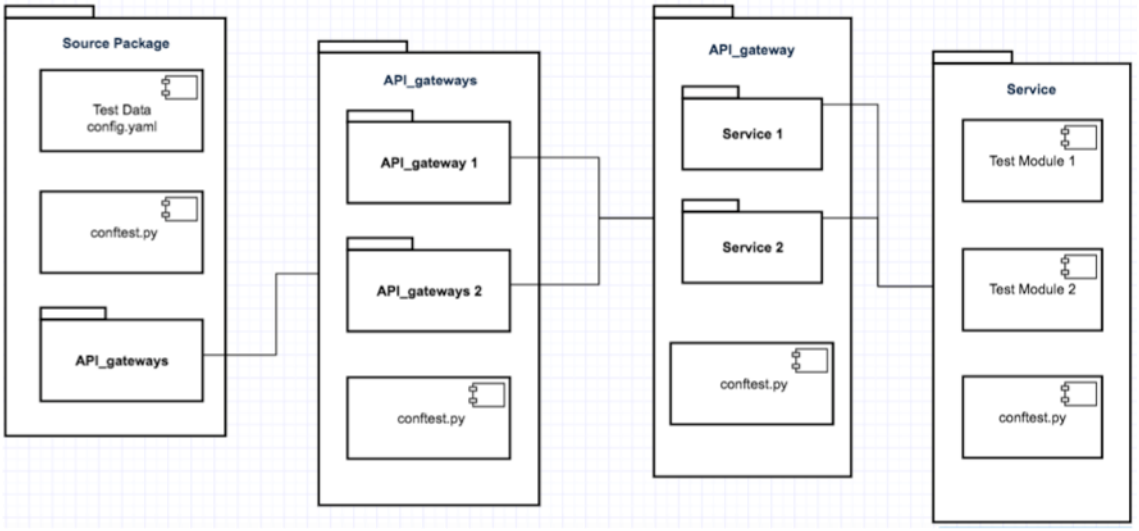

Figure 15. UML package diagram for API test framework

Figure 15 illustrates Saffron Digital's implementation of library architecture test framework by showing how packages, modules and files are placed. Modules called conftest.py are special named files that pytest looks for. They were intended for local plugins, but in this example they are also used for fixture functions hence acting like libraries. Where the conftest.py lives dictates the scope of where it applies. If present in the source package, fixture functions and setup hooks will apply to all tests in the source package as well as across all of its sub-packages. If present in a specific package, they will only apply to tests in that package and its sub-packages, but not to the tests placed in the packages on a higher level.

As it is shown in Figure 15, "master" conftest.py and test data files are placed in the source package as they are used across the whole project. The API_gateway package holds a selection of all different API gateways, each having its own sub-packages for relevant microservices in which corresponding test modules are being placed.

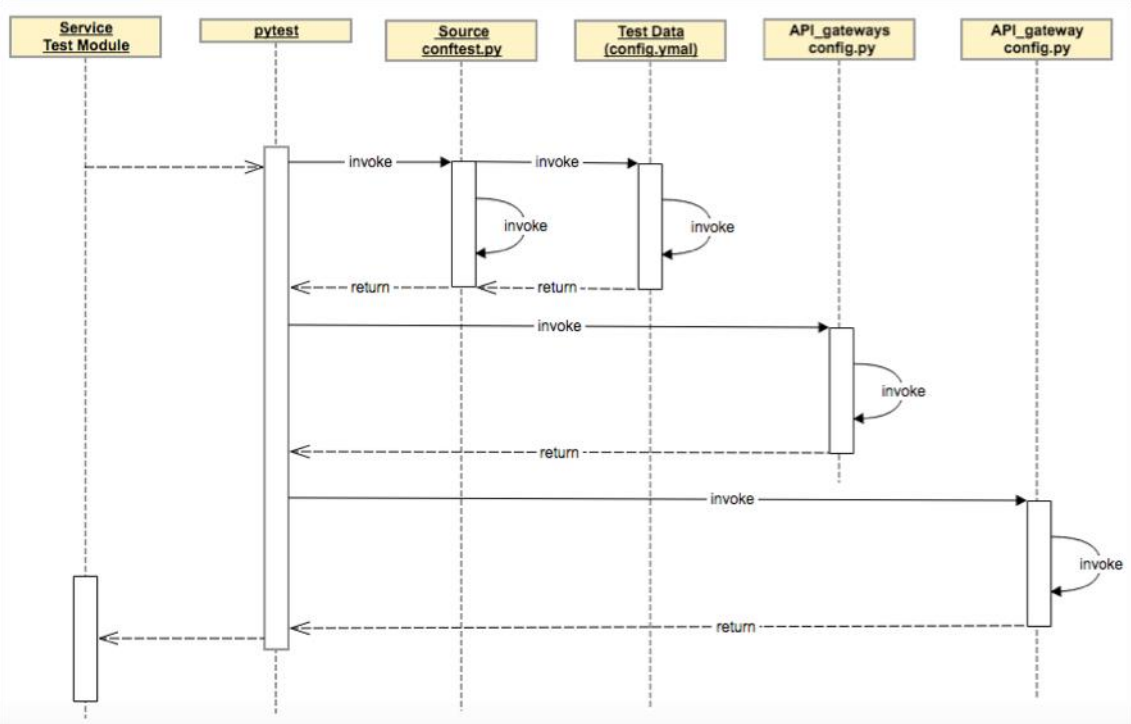

Figure 16. UML sequence diagram for API test framework

Figure 16 illustrates framework's sequence diagram by showing what is happening on the background when the test module has been triggered.

As a first thing pytest invokes all conftest.py files and creates a single instance on which all the tests that are placed inside that module are being run. This approach makes test execution faster as well as allows triggering multiple test instances of specified configuration at the same time.

Figure 17 represents an example set of tests inside a test module. Each test has its own set of fixture functions provided as arguments. When using fixtures from conftest.py file, there is no need to import different modules to the one that is using them. By triggering the test module, pytest invokes all conftest.py files that are applicable for the module after that initiates an instance on which all tests of that test module run.

```
app_name = 'name'
os = 'os'


def test_purchase_fixed_price_plan(create_user_account,
                                   get_subscription_plan,
                                   make_payment,
                                   get_transactions):
    """
    Test purchasing fixed price plan.
    Asserting subscription plan has been activated and user was charged the
    correct amount.
    """

    headers = create_user_account(app_name=app_name, os=os)['headers']

    plan = get_subscription_plan(plan_id= 'xxx',
                                 headers=headers)['plan']
    plan_price = plan['xxx']['xxx']

    make_payment(plan_id='xxx', payment_type='xxx', headers=headers)

    plan_after_purchase = get_subscription_plan(plan_id='xxx,
                                                headers=headers)['plan']
    assert plan_after_purchase['status'] == 'active'

    transactions = get_transactions().get(headers=headers)
    assert len(transactions['body']['xxx']['xxx']) == 1
    assert transactions['body']['xxx']['xxx'][0]['xxx'] == plan_price


def test_history_not_subscribed(create_user_account,
                                subscription_history):
    """
    Tests history is returned as an empty list.
    """
    headers = create_user_account(app_name=app_name, os=os)['headers']

    history = subscription_history.get(headers=headers)
    assert history['body']['xxxx'] == []
```

Figure 17. Example of two tests inside the test module

5 ROI for Automated Testing on the Example of Saffron Digital

For many people as well as companies, software testing is associated with repetitive manual process of navigating through the application by filling in and submitting different forms as well as clicking on the buttons. Uncontrollably it makes people think of testing as something simple that anyone can do without much knowledge and experience by simply having enough domain expertise.

Unfortunately this preconception affects the way people think of software test automation as well – it's testing – anyone can do it. Simply get a tool, ask testers to write automated tests that will perform required navigations without human intervention and watch how it solves the problem of test scheduling, lower the cost of testing and speed up testing processes.

In reality test automation is much more complicated than it may seem and not every approach will end up bringing value – on the contrary, it may bring more costs than benefits. It is important to realize that every test automation tool or framework is really just a specialized programming language, and developing an automated test library is a development project requiring commensurate skills and time [16].

As with any new implementation it is essential to conduct research and analysis in order to choose the best approach that will meet expectations as well as satisfy specific needs and requirements. It is important not to forget that the approach used in other's success stories may not be the best solution to go with.

In this paragraph computations of ROI values will be done for both UI test automation, currently used in Saffron Digital, as well as for API test automation that was implemented as part of this work. Results will be analysed based on which a conclusion will be made, if out of these two test automation approaches, the most suitable one was chosen as an option to begin with.

5.1 Return on Investment (ROI)

“ROI is the ultimate measure of accountability that answers the question: Is there a financial return for investing in a program, process, initiative, or performance improvement solution?” [17].

ROI is usually computed as the derived benefits divided by the costs of a given thing and is expressed by the equation ROI (%) = (Net Program Benefits / Program Costs) * 100 [17].

To get to ROI it is important to follow a four-phase process to ensure consistent and reliable results.

In the first phase, a planning of ROI evaluation should be conducted. It is important to have clear objectives that will help to develop a plan for data collection. This includes selecting the data collection instrument, identifying the source of the data as well as the period during which the data will be collected [17].

The second phase is the data collection itself, which is usually done during the program as well as after it ends, and the applied knowledge and skills becomes a routine [17].

Data analysis is done as part of the third phase. This is when the fully loaded costs are being developed, intangible benefits identified as well as ROI calculations performed [17].

The final phase is to report on the process and communicate the progress and an outcome.

The ultimate use of data generated through the ROI methodology is to show the value of programs, justify spending, gain support etc. [17].

5.2 Computations of ROI for UI and API Level Automated Testing

As part of this paper the data generated through the ROI calculations will be used to show the value of the new and existing processes as well as to justify spending that will help to decide on the process towards which future investments should be made.

In the case of Saffron Digital, the new process represents the implementation of server-level tests and existing process, proceeding with development of UI automated tests.

The main objective for using ROI methodology in case of Saffron Digital is to see how much will the company get back for every Pound Sterling invested in test automation, for each test approach.

Server-level test automation framework developed as part of this paper will be acting as a data source for automated API tests. UI test automation framework developed by Saffron Digital's QA team will be acting as a data source for automated UI tests and existing manual test cases, as a data source for manual testing. Data collection will be done while the processes are being implemented and used for the period of two months.

As two different test automation approaches will be compared to the same manual testing one, it was important to make sure that the data was collected in the similar conditions. This is the reason why data collection was done for the first two months of each process, not from the state the processes were in when starting work on this paper.

The collected data will be a calculated average, based on the results of 6 QA testers with similar testing and programming experiences. Three of them will be working on UI and the other three on an API test automation. Data will be collected for regression test suite of one SVOD client, using web as a platform.

There is a total of 42 test cases of which 28 will be used in a data collection process. The reasoning for this is the difference in automation approaches that allow implementation of different test cases as well as the fact that not all test cases can even be automated. To simplify the computations and minimise the risk of mistakes, this data will not be tracked.

In the case of Saffron Digital, test automation is being introduced after manual testing process has been in place for a while hence the cost benefits from automation will be viewed as trade-offs in comparison to manual testing.

Equation (1) will be used when computing ROI for both UI and API level automated testing. It shows a relative ROI for comparing the added benefits from automation with the added costs from automation as well as the value of automation in relation to

manual testing. It allows selecting relevant parameters according to Saffron Digital's needs and includes allocation of fixed and variable costs and benefits.

$$ROI_{automation}(in\ time\ t) = \frac{\Delta(Benefits\ from\ automation\ over\ manual)}{\Delta(Costs\ of\ automation\ over\ manual)} = \frac{\Delta B}{\Delta C} \quad (1)$$

- ΔB: The incremental benefits from automated over manual testing
- ΔC: The incremental costs of automated over manual testing

In order to use equation (1) all relevant financial costs and benefits need to be identified first. Values need to be determined for manual, automated UI and automated API testing.

Financial costs associated with automated testing can be generally split into fixed or variable costs. Fixed costs of automation are expenditures for equipment, tools, training, etc. that are not affected by the number of times tests are run or the number of tests being run. Variable costs increase or decrease based upon the number of tests that are developed or the number of times the tests are run [18].

Table 2 includes a list of some fixed and variable automation cost factors. If the cost factor is not included in Saffron Digital's test automation ROI computations then reasoning is provided.

Table 2. List of fixed and variable cost factors

Cost factors	**Fixed/variable cost values**	**Used in ROI computations**	**Reasoning if not used**
Hardware (additional or upgrades to existing)	Fixed	No	Approximately the same cost for automated and manual tests
Tool and programming language training/introduction	Fixed	No	Training costs are included in the test development and maintenance cost as team members learned as they progressed
Software licenses	Fixed	No	Approximately the same cost for automated and manual tests

Cost factors	Fixed/variable cost values	Used in ROI computations	Reasoning if not used
Automation environment design/implementation	Fixed	Yes	N/A
Scripting tools and licenses	Fixed	No	Using only freeware tools
Test case implementation	Variable	Yes	N/A
Test maintenance	Variable	Yes	N/A
Test case execution in CI environment	Variable	Yes	N/A
Test results analysis	Variable	No	Not tracked separately and is included in test case creation and maintenance costs
Defect reporting	Variable	No	Approximately the same cost for automated and manual tests
Test results reporting	Variable	No	Test result reporting is done via company's internal system hence the same cost for automated and manual tests
Data generation	Variable	No	Test data is generated as part of the tests hence is included in test case creation and maintenance costs
After-hours testing by systems	Variable	No	No after-hour testing is planned to be done for the first year

During the data collection the following assumptions have been made:

- Server deployments are done twice a week and as with every deployment at least one bug is found
- Application deployments are done once a month with the assumption that no bugs are found

- For server deployments manual regression tests are run 1,5 times a week as full set of tests is run only once. After bug fixes only main scenarios are tested
- For application deployments manual regression test suite is run once a month with the assumptions that no bugs were found
- API tests are run after every server deployment
- UI tests are run after every server and application deployment
- All manual and automated tests are run only on pre-production environment
- Total time spent for building test framework for API testing has been divided by 3 as the same framework can be reused for the other 2 clients
- Cost of automated test execution is 0 as all tests are triggered automatically by the CI tool

Data collected within a 2-month period, which is used in UI and API test automation ROI calculations is presented in Table 3.

Table 3. Saffron Digital's cost factors for ROI calculations

Factors	**Manual testing**	**UI automated testing**	**API automated testing**
Average salary per person per year (£) (London)	27000	40000	40000
Hourly salary per person	14.78	21.9	21.9
Number of working days in a year (United Kingdom)	261	261	261
Number of full weeks in one year	52	52	52
Hours in a one man-day	7	7	7
Number of test cases	28	28	28
Regression test suite runtime in hours	4.5	0.35	0.084
Number of regression test suite executions per year	90	116	104
CI tool infrastructure cost per hour (£)	N/A	0.12	0.12

Factors	Manual testing	UI automated testing	API automated testing
Test case creation time in hours per year	147	392	294
Test suite maintenance in hours per month	268	336	84
Building test framework in hours per year	N/A	14	23.3

Collected test data will be placed in an equation (1) in the way that is represented below:

$$\Delta B(in\ time\ t) = \sum(variable\ costs\ of\ running\ manual\ tests\ n_m\ times\ during\ time\ t) - \sum(variable\ costs\ of\ running\ automated\ tests\ n_a\ times\ during\ time\ t)$$

$$\Delta C(in\ time\ t) = \sum(fixed\ costs\ of\ creating\ automated\ test\ framework) + \sum(variable\ costs\ of\ creating\ automated\ tests) + \sum(variable\ costs\ of\ maintaining\ automated\ tests) + \sum(variable\ costs\ of\ executing\ automated\ tests\ in\ CI\ environment) - \sum(variable\ costs\ of\ creating\ manual\ tests) + \sum(variable\ costs\ of\ maintaining\ manual\ tests)$$

- n_m: Number of automated test executions
- n_a: Number of manual test executions
- t: Period of ROI

Computation of ROI for automation of UI tests:

$$\Delta B(in\ 12\ months) = (14{,}78 * 4{,}5 * 90) - 0 \approx 5985{,}9$$

$$\Delta C(in\ 12\ months) = \big((14 * 21{,}9) + (392 * 21{,}9) + (336 * 21{,}9) + (116 * 0{,}35 * 0{,}12)\big) - \big((147 * 14{,}78) + (168 * 14{,}78)\big) = (306{,}6 + 8584{,}8 + 7358{,}4 + 4{,}87) - (2172{,}66 + 2483{,}04) = 16\ 254{,}67 - 4655{,}7 \approx 11\ 598{,}97$$

$$ROI_{UI\ automation}(in\ 12\ months) = \frac{\Delta B}{\Delta C} = \frac{5985{,}9}{11\ 598{,}97} \approx 0{,}52$$

Computation of ROI for automation of API tests:

$$\Delta B(in\ 12\ months) = (14{,}78 * 4{,}5 * 90) - 0 \approx 5985{,}9$$

$$\Delta C(in\ 12\ months) = \big((23{,}3 * 21{,}9) + (294 * 21{,}9) + (84 * 21{,}9) + (104 * 0{,}084 * 0{,}12)\big) - \big((147 * 14{,}78) + (168 * 14{,}78)\big) = (510{,}27 + 6438{,}6 + 1839{,}6 + 1{,}05) - (2172{,}66 + 2483{,}04) = 8789{,}52 - 4655{,}7 \approx 4133{,}82$$

$$ROI_{API\ automation}(in\ 12\ months) = \frac{\Delta B}{\Delta C} = \frac{5985{,}9}{4133{,}82} \approx 1{,}45$$

5.3 ROI Value Analysis

The results of ROI computations for automated UI and API level testing differ quite a bit. It is clearly shown that in the case of Saffron Digital, investing in UI automated testing within a year will bring almost 50% loss to the company which indicates that continuing manual testing turns out to be more beneficial.

In the case of automated API tests, implementing these tests looks to be more promising with a 45% return within a year.

So why are these results so different if in both cases tests are being automated hence theoretically costs should be decreasing and benefits increasing?

When analysing the collected data we can conclude, that time spent on a UI test maintenance within a year is four times greater than the time spent on an API test maintenance. This difference is mainly caused by changes made in the code.

Application releases are being made less frequently compared to server level deployments, but since UI tests are based on an application UI, which is changing with almost every release, tests need to be refactored every time.

Weekly server deployments are mainly done for new features or bug fixes and changing existing APIs happens less frequently. This is the reason why tests are relatively stable and regression tests suite maintenance time is minimal.

Another factor why UI test maintenance takes up more time is the lack of proper testing framework, which results in code duplication across the whole project making it harder to refactor each test.

By the collected data and the ROI values we can clearly see the cost impact of missing server-level testing. Covering regression testing with only UI tests in a constantly changing environment is expensive and in this case even cheaper to be left undone.

Venn diagram in Figure 18 represents Saffron Digital's test coverage of manual versus automated UI versus automated API tests. This illustration shows the possibilities of server-level testing and actually suggests a better approach to test automation, which also ties in with Mike Cohn's testing pyramid theory.

Tests automated on the lower level of the pyramid can cover more and as proven by ROI calculation values, they are cheaper and faster to automate and maintain.

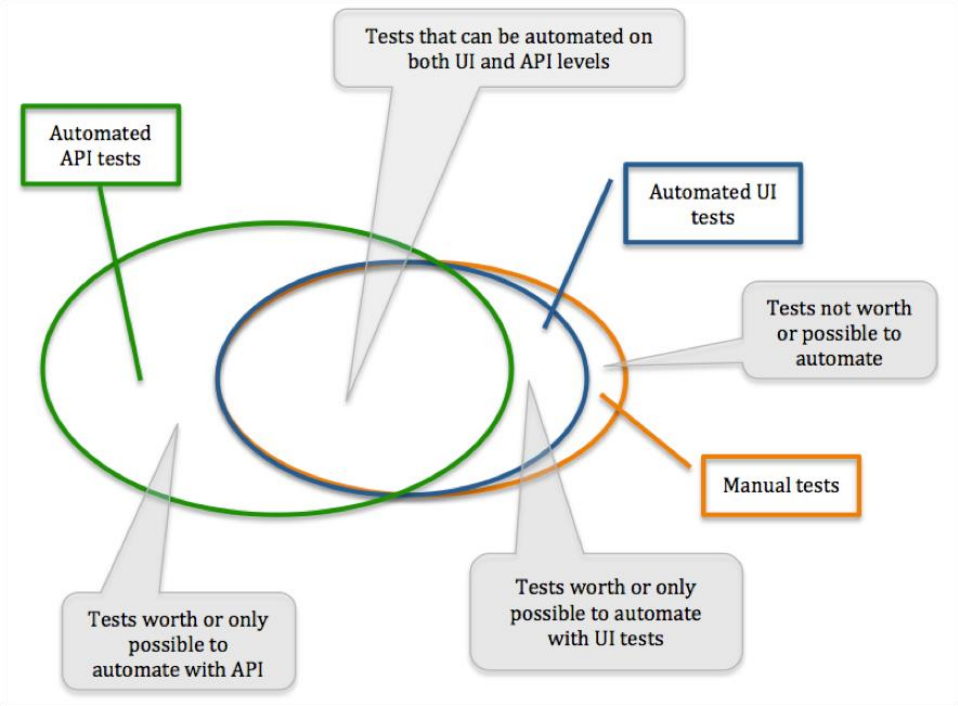

Figure 18. Venn diagram of test coverage for manual, automated UI and API tests

As it appears on the example of Saffron Digital, test automation is not always necessary, appropriate or cost efficient. It is essential to conduct research and relevant analysis in order to choose the best approach that will meet expectations as well as bring value.

This paper has shown that implementation of test automation framework for API integration testing has been successful. It has brought out a critical investment mistake towards automation of UI tests as well as given directions where test automation can benefit the company.

6 Summary

The aim of this work was to introduce testing pyramid theory and based on that detect possible problems with Saffron Digital's current testing strategy. By following the principles of the theory, try to address existing challenges and concerns by implementing structured server-level test automation framework for API integration testing.

The main objective was to understand if server-level test automation turns out to be more expensive to implement and maintain comparing to already existing UI test automation.

In achieving that, implemented test automation framework was used as a pilot project for the period of two months for test development. A relevant data was collected for calculation of costs and benefits associated with it. To be able to compare the investment required for automating API tests to the investment needed for already existing test automation process, the same type of data was collected for UI tests.

The results of utilisation of the ROI methodology in computing the costs and benefits of automated UI and API tests in comparison to manual testing showed that over the ROI period of 12 months, for every Pound Sterling invested in API test automation will return 45% and for every Pound Sterling invested in UI testing will cause a loss of almost 50%. These results allow concluding that implementation on test automation framework for API integration testing has been successful.

There are certain opportunities for continuing to develop this work by further analysing the given test data and performing further research. This could help to come up with a test strategy where tests are divided between different test automation approaches as best feasible for the business.

References

[1] E. Dustin, T. Garrett and B. Gauf, Implementing Automated Software Testing. How to Save Time and Lower Costs While Raising Quality, Upper Saddle River, NJ: Addison-Wesley, 2009.

[2] M. Fewster and D. Graham, Software Test Automation, London: Addison-Wesley, 1999.

[3] M. Cohn, "The Forgotten Layer of the Test Automation Pyramid," 17 December 2009. [Online]. Available: https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid. [Accessed 22 April 2016].

[4] D. Arbuckle, Python Testing, Birmingham: Packt Publishing, 2010.

[5] S. Ashman, "Layers of Test Automation," 28 December 2014. [Online]. Available: http://qa-matters.com/2014/12/28/layers-of-test-automation/. [Accessed 22 April 2016].

[6] Saffron Digital LTD, "About us," 2013. [Online]. Available: http://www.saffrondigital.com/company/. [Accessed 23 April 2016].

[7] Debitoor, "Subscription - What is a subscription?," 2012-2016. [Online]. Available: https://debitoor.com/dictionary/subscription. [Accessed 23 April 2016].

[8] SoftwaretestingHelp.com, "Most Popular Test Automation Frameworks with Pros and Cons of Each – Selenium Tutorial #20," 17 March 2016. [Online]. Available: http://www.softwaretestinghelp.com/test-automation-frameworks-selenium-tutorial-20/. [Accessed 30 April 2016].

[9] A. Shrivastava, "Automation Framework Architecture for Enterprise Products: Design and Development Strategy," July 2012. [Online]. Available:

http://www.oracle.com/technetwork/articles/entarch/shrivastava-automated-frameworks-1692936.html. [Accessed 01 May 2016].

[10] M. Kelly, “Choosing a test automation framework,” 20 November 2003. [Online]. Available: http://www.ibm.com/developerworks/rational/library/591.html. [Accessed 01 May 2016].

[11] The Python Software Fundation, “What is Python? Executive Summary,” 2001-2016. [Online]. Available: https://www.python.org/doc/essays/blurb/. [Accessed 17 April 2016].

[12] M. Lutz, Learning Python, 5th Edition ed., Beijing: O’Reilly Media, 2013.

[13] P. Guo, “Communications of The ACM,” 07 July 2014. [Online]. Available: http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext. [Accessed 17 April 2016].

[14] H. Krekel, “pytest fixtures: explicit, modular, scalable,” 2015. [Online]. Available: https://pytest.org/latest/contents.html. [Accessed 16 May 2016].

[15] D. Stufft, “Slumber documentation,” 2011. [Online]. Available: http://slumber.readthedocs.org/en/v0.6.0/#. [Accessed 21 April 2016].

[16] L. G. Hayes, The Automated Testing Handbook, 2nd Edition ed., Software Testing Institute, 2004.

[17] P. Pulliam Phillips and J. J. Phillips, Return on Investment (ROI) Basics, ASTD Press, 2005.

[18] D. Hoffman, “Cost Benefits Analysis of Test Automation,” 1999. [Online]. Available: https://www.agileconnection.com/sites/default/files/article/file/2014/Cost-Benefit%20Analysis%20of%20Test%20Automation.pdf. [Accessed 15 May 2016].