Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TREBALL FINAL DE CARRERA

Títol: A real example of how to implement Agile Testing

Titulació: Enginyeria Tècnica de Telecomunicació, especialitat en Sistemes de Telecomunicació

Autor: Victor Pascual Villanueva

Director: Toni Oller Arcas

Data: 18 Setembre 2015

Title: A real example of how to implement Agile Testing

Author: Victor Pascual Villanueva

Director: Toni Oller Arcas

Date: September 18th 2015

Overview

Agile development is now the standard for development teams to produce software. Testing is a very important part of this development process, and it is also important knowing how to include testing in the new agile process.

In this project we will analyse the context of a real development team and we will define a set of recommended practices adapted for this team with the goal of improving their development process by implementing Agile Testing. Not only defining what Agile Testing is but also implementing real examples of two of the main Agile Testing practices: Test Automation and Continuous Integration.

We will use open-source tools, used nowadays at an enterprise level, to automate browsers, make API calls or build a Continuous Integration system.

TABLE OF CONTENTS

INTRODUCTION

Agile development is no longer a fancy trend. It has become the standard of software development for the most successful companies in the world. Everyone is trying to implement Agile in one way or another, but where it fits Software Testing in all this new world?

Programmers have had to adapt the way they write code to be more agile, but for the whole transition to be successful, development teams should also adapt on how they test software. Agile Testing is a set of ideas and practices to bring the benefits of agile development to Software Testing.

The purpose of this project is to expose the benefits of Agile Testing and define how to implement it in a real company, Alteraid. We will analyse the existing development practices to identify pain points or bottlenecks in the software development cycle and we will try to find solutions to them by using some recommended practices from Agile Testing.

Two of these practices will be Test Automation and Continuous Integration which we will get into more detail, extending their definition and implementing real examples that apply to some of the products at Alteraid.

In the first chapter, we will briefly introduce Agile Testing and its main principles. Then we will analyse the target company, Alteraid, so we can get a better picture and some insight to later define a testing strategy based on Agile Testing methodologies and the specific context of the company. This strategy will include best practices for bug tracking, test environment and test automation among others.

In second chapter we will see the benefits of automated functional tests and we will get more practical by implementing some of them for two real products at Alteraid, a Web application and an API. We will focus on writing readable and maintainable tests that a development team could also use as documentation for the features in their products.

Third chapter will also be quite practical as besides from defining what Continuous Integration is, we will implement a Continuous Integration server to run the automated functional tests that we will have created in second chapter. Given the Alteraid context, we will also give some recommendations on how to work in a Continuous Integration workflow.

Having seen all this aspects of Agile Testing applied in a real world example will hopefully facilitate future work to people who want to implement these practices for their respective projects or companies.

CHAPTER 1. TESTING AT ALTERAID

The goal of this chapter is to define a testing strategy for the development team at an specific company. First we will see what is Agile Testing and we will analyse the specific context for the company, Alteraid. Then we will describe some best practices and recommendations to implement Agile Testing taking into account the company context.

1.1 Agile Testing

The “Agile Manifesto” was created during an informal gathering of software development experts in a ski resort in Utah in early 2001 [1]. It has now become the standard of software development with a collection of practices that try to provide an alternative to a documentation driven and heavyweight processes.

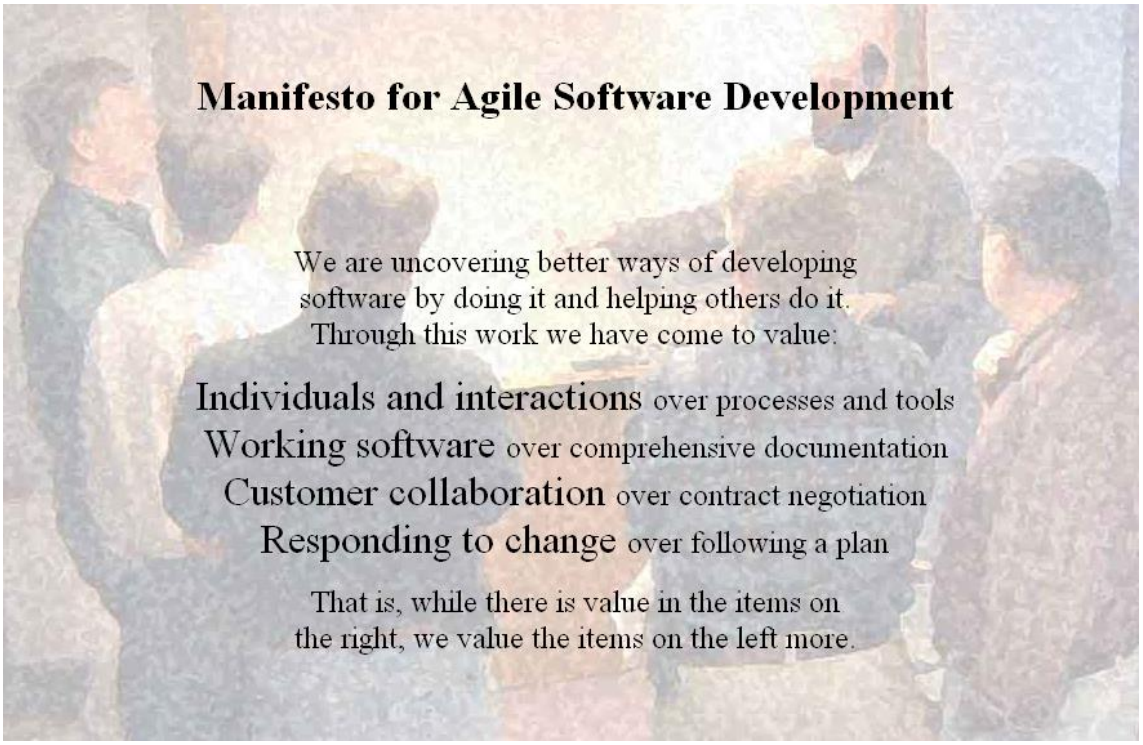

Figure 1.1 Agile Manifesto

Testing is a key element of software development. Is the process of evaluating a product by learning about it through exploration and experimentation. A product is tested so that it can be judged; for completeness, for its fitness for purpose, and for risks. It is vital because it produces information that feeds back into the creative development process.

Agile Testing is the application of the Agile Manifesto to software testing, using these values as a test strategy and defining how to perform testing activities on a project which has these values. How these values apply to software testing are described below:

Individuals and interactions over processes and tools

Testing should be driven by what is important to a user, rather than to fulfil a procedural requirement. Communication in the team and with the customer is better than maintaining the independence of the test team, that's why direct interactions should be empowered over rigid test processes and test tools.

Working software over comprehensive documentation

Many test techniques require comprehensive documentation to allow effective test design so this value has a fundamental effect on software testing.

On an agile project, due to the lack of extensive documentation we need to be able to work with software products directly; reading unit tests, exploring the system or analysing a comprehensive range of outputs.

Customer collaboration over contract negotiation

Testing is in essence taking the role of the customer in designing and executing acceptance tests, so the information that the customer can provide us with this collaboration is key on the role of extending a customer's tests to the limits of the application, acting in some cases as an undesirable user or 'Bad Customer'.

Testing only what has been agreed in a contract, reporting bugs only against fixed requirements and such strategies that lean heavily on an unchanging set of requirements may be considered against that position.

Responding to change over following a plan

An acceptance of change is at the heart of agile projects – there is no requirements freeze before coding starts, and this can cause substantial problems for testers (and coders) used to working to a fixed set of requirements.

Automated tests play an important role here as give an stability to the code and to the product that goes some way to reducing the effects of constant change.

1.2 Testing in context

The value of any practice depends on its context, that's why, first of all, we'll analyse the context where the testing practices are going to be defined.

In this analysis we'll cover some of the areas where testing can be more conditioned by, like team size or industry. We'll also get some insight from the current development practices at Alteraid using an informal tool called "The Joel Test", this will help us to define, later on the project, the testing practices that the team can adopt with less friction.

1.2.1 Organization

Alteraid is a spin-off startup of the Castelldefels School of Telecommunications and Aerospace Engineering (EETAC-UPC) [2]. It develops a cloud platform solution and its ecosystem of web and mobile applications that provide a secure communications and record-sharing infrastructure for formal and informal quality of life management.

1.2.2 Industry

Alteraid develops a software-as-a-service (SaaS) platform that stores sensible health information and manages its access at a personal level.

It's not aimed to be an Electronic Health Record [3] or similar so highly restrictive regulations that apply to other products in the Healthcare industry, like medical devices or Hospital software, do not apply to Alteraid products. Working on highly regulated environments is challenging for agile practices as some industry standard certifications, like ISO 13485:2003 for Medical devices [4], specify requirements for a quality management system where testing is considered a phase in a waterfall-like approach rather than an integrated activity in the development process where we can iterate.

The nature of the information that Alteraid products manage could be compared to the likes of fitness and wellness trackers like mobile apps for runners or sleep monitoring apps. Health, wellness or fitness data is sensible and security testing should still play an important role on the development process but not being on one of the highly regulated environments mentioned before gives us more flexibility when applying agile practices.

1.2.3 Team

The team consists of two product owners, and 6 developers. Product owners are the ones in contact with the customers and have the vision of the products that the developers build. Developers are cross-functional and there are no specialised roles. There is no dedicated tester on the team so developers self-test the code that they produce.

Development team works in different projects and the software development practices used depend on each different project. No master processes, guidelines or tools are defined across all different products.

1.2.4 Products

Alteraid is currently developing or maintaining the following products:

- **Aaaida**: cloud solution that provides a secure communications and record-sharing infrastructure for formal and informal quality of life management of those people that worry and concern us [5]. The data is accessible to users via a web portal or via public API's.

- **Virginia**: Android mobile app that allow relatives, caregivers and case managers to be involved into the care process of the people they are concerned or worried about [6]. Virginia gets Virginia Henderson's 14 components related to the elderly person and upload this information into Aaaida to be accessible by the stakeholders close to this elderly person.

- **Adheptor**: Android mobile app to track the adherence to treatment of Hepatitis C patients. The doses are uploaded to Aaaida, and in case of missed intake, an email is sent to the doctor and wife of the patient, for example [7]. Currently, it is used by the liver Spanish patients association (FNETH).

- **Dermapac/Demadoc**: Two android mobile apps, for patients and doctors respectively, that allow monitoring the evolution of skin diseases, like psoriasis, through the access of skin pictures taken by the patient, or a relative and make them accessible to the doctor [8].

Aaaida is the core product of Alteraid as other products, web or mobile apps, use that platform to share and integrate with their data.

1.2.5 Velocity of releases

From the products enumerated in the previous section, Virgina and DermaPac/DermaDoc are in maintainability mode, this means that no new

features or new releases are expected, while Adheptor and Aaaida platform are being actively developed.

Product releases in Alteraid are driven by customer requests rather than following a calendar of releases where the product can be iterated in.

A new version of Adheptor is released twice a month approximately, while Aaaida platform is considered in a stable mode with just few updates a year.

1.2.6 Business value

Being a university spin-off means that the business value of the company might be driven not only by commercial reasons but by research purposes or collaboration with public or government organizations.

Currently all products at Alteraid are offered without any charge, but in the mid-term the plan is to offer a subscription model to intensive professional users of the Aaaida platform.

Products will remain free for end users, like patients of a certain treatment, but professional organizations like healthcare providers or pharmaceutical companies might be charged following a pay-per-use API schema of the Aaaida platform.

1.2.7 The Joel Test

Back in 2000, software engineer Joel Spolsky published a blog post entitled: *"The Joel Test: 12 Steps to Better Code"* [9], that included the following questions:

The Joel Test

1. Do you use source control?
2. Can you make a build in one step?
3. Do you make daily builds?
4. Do you have a bug database?
5. Do you fix bugs before writing new code?
6. Do you have an up-to-date schedule?
7. Do you have a spec?
8. Do programmers have quiet working conditions?
9. Do you use the best tools money can buy?
10. Do you have testers?
11. Do new candidates write code during their interview?
12. Do you do hallway usability testing?

Figure 1.2 The Joel Test

Many software development organizations use this test as a sort of informal self-test to determine what they need to work on. We'll use the answers to that questions in Alteraid to have a better picture of the current development process and spot the weaknesses so in the next section we can better describe how testing can improve that process.

Here are the answers to the test given by the development manager at Alteraid:

1. **Do you use source control?**

Yes, currently we host our repositories in a self-hosted GitLab server [10]. In the past we have also used Subversion [11].

2. **Can you make a build in one step?**

Yes, we use Jenkins as a continuous integration server [12] and we can trigger builds just clicking a button. Although not all products are integrated in the Jenkins server.

3. **Do you make daily builds?**

Yes, builds for Aaaida API are triggered after pushing code to the repository. Other products that include Android apk's are generated manually on demand.

4. **Do you have a bug database?**

Yes, we use Pivotal Tracker to track the bugs in all our products [13]. In the past we've also used Mantis [14]. Although we recognize that there is not a formal process to track and resolve defects that we or our customers found in our products.

5. **Do you fix bugs before writing new code?**

Not always, each development iteration includes some bugs and new stories to work on but we do not necessary start fixing existing bugs first.

6. **Do you have an up-to-date schedule?**

We try to, although things move fast and is hard to keep things up to date. We use Trello as a tool for collaboration [15] where our high level milestones are defined and are accessible to all the team.

7. **Do you have a spec?**

Not really, user stories are defined at a high level and there is no documentation for the features of our products.

8. **Do programmers have quiet working conditions?**

Yes, working environment is quiet.

9. **Do you use the best tools money can buy?**

We try to, we benefit from research and student licenses granted to UPC.

10. **Do you have testers?**

No, programmers self-test their own code.

11. **Do new candidates write code during their interview?**

Not exactly but most of our programmers have been recruited from the classes where I lecture at UPC, so I know their coding skills beforehand.

12. **Do you do hallway usability testing?**

No, we try to follow design and usability guidelines from Google in some of our Android apps but we have not considered doing any usability testing so far.

1.3 Define Testing practices

In a young and small development team as the one at Alteraid, it is usual that products end up released with minimal or inexistent testing, that could be because we prefer having quick feedback from customers than a reliable, nice and finished product, or simply because the team does not care about customers being the ones in finding the defects on the product (beta testers).

This approach could be valid in the early stages of a company where we only have a few customers and we are probably offering our products for free, but this will not work for too long as our customer base will, hopefully, grow and users will no longer be willing to be our beta testers, rather want to get value from a usable, reliable and defect-free product.

Knowing that testing plays a key role on developing quality software and having analysed the context of Alteraid, now we will tell the story about how their

development team should implement Agile Testing to deliver the software that their customers want in a fast and flexible way.

To tell this story, we will use the Five Ws from Journalism [16] as it's a clear and structured way that enforces us to cover the different aspects of the software development process.

1.3.1 Who

If we know that the ones finding the bugs in the product can no longer be our customers, we obviously need to find these bugs before releasing.

In traditional software teams, engineers work in a similar way as they would do it in the manufacturing of a car, first, a team of engineers build the product, then a second team verify the quality of the product performing some tests. This second team of engineers are who perform the testing activities and act as the gate-keepers of quality for the product. This traditional way of developing software is known as the Waterfall model [17]:

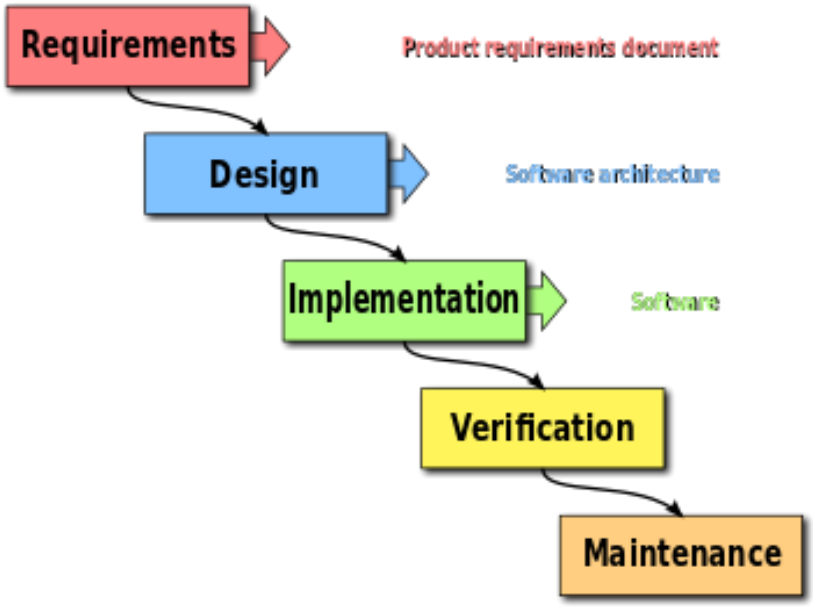

Figure 1.3 Progress flows of Waterfall Model

The Waterfall model originates in the manufacturing and construction industries with highly structured physical environments, where after-the-fact changes are prohibitively costly, if not impossible. This methodology has been proven as inefficient for software development, where customers may not know exactly what their requirements are before they see working software and so change their requirements, leading to redesign, redevelopment, and retesting, and increased costs.

In agile, the concept of a second team of testers verifying the quality of the product on a later phase is not recommendable and should no longer apply. Developers are responsible for the code that they produce, from the early stages of the design until the code is finally released to production.

Practices like Test-Driven Development (TDD) [18] and Behaviour-Driven Development (BDD) [19] help developers to build the quality in from the start by starting test activities as early as possible, creating automated tests that check the correct functionality of the code.

As we have seen, developers play a big role on producing quality products by testing the code that they produce but, unfortunately, most of the times this is not enough. Developers are excellent at testing the internals of the code they write, at a unit or integration level, but a second opinion at a higher level becomes essential to ensure that customers are going to get the value they expect from the product. Here is where another team member, usually a tester, plays the customer advocate role and helps the developer on discovering issues like uncovered edge cases or usability problems.

Testers in Agile Testing do not longer execute tests at a later phase as in traditional software development, they use their testing knowledge to build quality in the product from the start, giving their input on testability or edge cases when defining user stories, helping developers to ensure code is properly tested at different levels, building testing tools or frameworks, preparing test data and test environments or coaching the team to improve their testing skills.

At Alteraid there is no dedicated tester, so, as we have just seen, products could end up being released with big functionality gaps or defects because of the lack of this expert second opinion. In this situation, I would obviously recommend to include hiring a testing expert, or specializing an existing team member, on the mid-term roadmap.

For the current situation, with no tester in the team, developers should still be responsible for the code they write, creating automated tests and ensuring their user stories are tested at a higher level from the end-user perspective by a different team member, ideally one not involved in the technical implementation. This will ensure the valuable second opinion and will help in making quality a whole-team responsibility.

Developer and the second developer, the one playing the tester role, can collaborate together to flag things like: areas not covered by automated tests, potential performance issues in production environment, user experience problems or corner cases not considered in design. As we have seen before, this collaboration should start as early as possible and be maintained until both team members are happy with the quality of the code to be released.

1.3.2 What

Testing activities are performed for various reasons: to find bugs, to make sure the code is reliable, and sometimes just to see if the product is usable. Different types of testing should be done to accomplish different goals, a tool that can help to decide on what to test to deliver value is the “Agile Testing Quadrants” [20]:

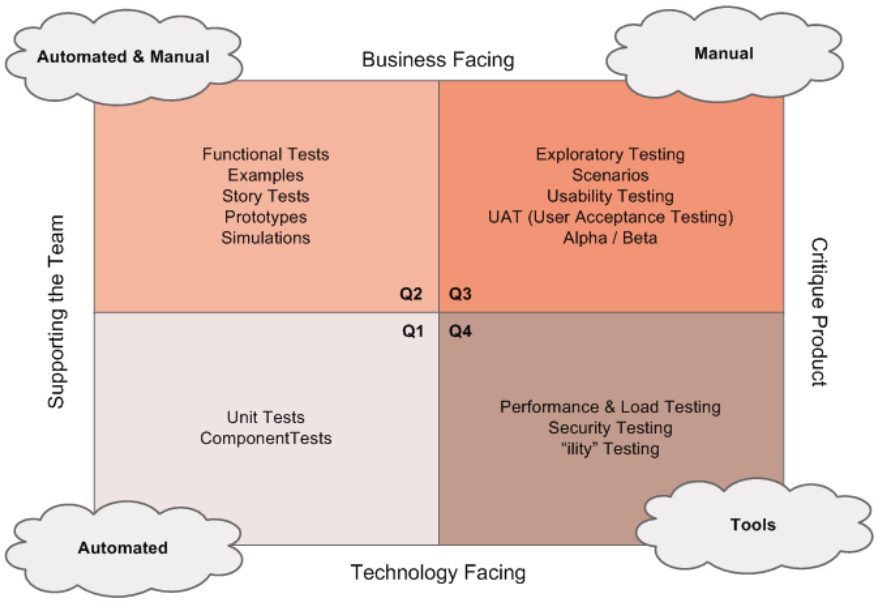

Figure 1.4 Agile Testing Quadrants

This concept was first introduced by Brian Marick, an author of the Agile Manifesto, and it was later popularized by Lisa Crispin and Janet Gregory on their book *“Agile Testing: A Practical Guide for Testers and Agile Teams”* [21].

The development team can use this matrix to decide on all the different types of tests that need to implement for a certain feature, who does which tests and how, and when testing for a user story can be considered as “done”.

Each quadrant is used to categorize tests based on their purpose (Support Team vs Critique Product) and their domain (Business vs Technology facing):

Quadrant 1

Represents test-driven development, which is a core agile development practice. Unit tests verify functionality of a small subset of the system, such as an object or method. Component tests verify the behavior of a larger part of the system, such as a group of classes that provide some service. Both are usually automated and help programmers design their code well and confidently to deliver features without worrying about making unintended changes to the system.

These tests are normally part of an automated process that runs with every code check-in, giving the team instant and continuous feedback about their internal quality.

Quadrant 2

Tests that support the work of the development team, but at a higher level. These business-facing tests define external quality for the features that the customers want and are derived from examples provided by the customers.

Most of the business-facing tests that support the development team also need to be automated. One of the most important purposes of tests in these two quadrants is to provide information quickly and enable fast troubleshooting. They must be run frequently as part of an automated continuous integration, build, and test process, in order to give the team early feedback in case any behaviour changes unexpectedly.

Quadrant 3

Business experts might overlook functionality, or not get it quite right if it isn't their field of expertise. The development team might simply misunderstand some examples. Tests in this quadrant are to exercise the working software to see if doesn't quite meet expectations, trying to emulate the way a real user would use the application. It's difficult to automate these tests because such testing relies on human intellect, experience, and instinct. However, automated tools can assist with some aspects such as test data setup.

Exploratory Testing is key to this quadrant. During exploratory testing sessions, we simultaneously design and perform tests, using critical thinking to analyse the results and to see if the "done" stories are really done to our satisfaction.

Quadrant 4

Technology-facing tests to critique product characteristics such as performance, robustness, security, maintainability, interoperability, compatibility or reliability. Should be considered at every step of the development cycle and not left until the very end. In many cases, such testing should even be done before functional testing. It is a good idea to have a checklist to make sure the team thinks about them and asks the customer how important each one is.

Automation is mandatory for some efforts such as load and performance testing.

The quadrants are merely a taxonomy to help teams plan their testing and make sure they have all the resources they need to accomplish it. Team should think about all four quadrants early and all through the development cycle to make sure all needed categories of testing are covered, choosing or creating tools that solve each testing problem and building the right test architecture that works for all team members. Should also find ways to keep customers involved in all types of testing so we can build products that meet all their expectations.

1.3.3 When

As we have seen early in this chapter, in agile development we do not longer execute tests in a later and isolated phase, we rather build quality in the product from the start. At Alteraid, products are built iteratively and in this section we will focus in one of this iterations and we will describe how testing can help from the planning phase to the successful delivery of the stories:

Planning

Before starting to design and code the stories scheduled in an iteration we need to polish their definition, estimate the effort (sizing) and also prioritize them.

We can collaborate with customers by asking questions and getting examples that will help improving definition of each story so developers can start coding straight away. For complex stories, lightweight test plans in form of checklists, mind maps or test matrices can help to define testing strategy and also visualize progress on testing tasks over the iteration.

When sizing a story, the input from the testing expert can help on analysing the story from different viewpoints, including business value, risk or how the feature will be used by the end users. Also testing efforts should be considered when estimating as no story is done until is fully tested.

When prioritizing work of an iteration, we should schedule high-risk stories early as they might require extra testing early. Other testing aspects that should be addressed early are test environments preparation and test data concerns.

Implementation

Once stories are defined and all resources are ready, we can kick-off coding, but also testing, as both activities should be part of one process during the iteration.

In Alteraid, as there is no dedicated tester, a second developer will perform the testing activities as soon as main developer starts coding. Testing

should start in this phase, guiding development, by writing a simple automated test that covers the “happy path”, when this simple test pass, more complex tests can be written so we can further guide coding.

We should test critical functionality first and, if possible, we should focus on completing one story at a time, as missing some functionality from an iteration is better than missing the entire iteration because testing couldn’t be completed on all or most stories. If needed, other team members can help as everyone can work on testing tasks.

Automated functional tests created during these phase can also be added to a regression test suite that should be scheduled to run often enough to provide adequate feedback. We will see more about that in the “Continuous Integration” chapter. After coding is finished, manual testing can also help to find missing requirements or usability problems.

Delivery

Even when all stories are “done” we might not be able to release the product to our customers. Successful delivery of a product includes more than just the application we are building: database upgrades, packaging or training documentation should be tested as well.

When possible, releasing first to a set of internal or “friendly” users is a good practice to ensure that our product will work properly in our customers environments, as they might have configurations or sets of data that we did not consider beforehand.

Monitoring metrics in the production environment will also help to identify potential issues early, even before our customers are affected. Specially for software-as-a-service solutions, like Aaaida web application, it’s also important to have a roll-back process in place, so if something goes wrong with the delivery, we can go back to a previous working version of our product.

Having travelled through an iteration we can now see how testing provides value at each stage, in order for this Agile Testing mind-set to succeed, it is key that the whole development team feels responsible for the quality of the products that they deliver, as no other team will be taking that responsibility at a later phase as in traditional software development.

1.3.4 Where

Even when testing helps from the start to the end of an story, with enough time and resources allocated, and we consider all different scenarios that we can think of, we can still deliver a broken feature for our customers because they are using

a different OS, browser, device or an slightly different set of data. When testing, we should consider which environments our customers will be using, which is their configuration and also which are the characteristics of their data, so we can mimic those as much as possible and catch, during development, the majority of the issues that might be related to our customers specifics.

Test Environments

At Alteraid, programmers usually use their local machines as sandboxes for development and testing, they might use a favourite browser, such as Chrome, and have a small set of data to run their local tests. This is ok as it keeps their environment small and lean but it is not enough when further testing needs to be done for a story as developer sandboxes might have unstable code deployed, so they are not always available for testing, and their configuration will hardly mimic our production or customer environments.

Having a shared test environment will help the development team as this will have a known build installed, with passed automated tests and always available for testing. This test environment should be as similar as possible to production, using data that reflects the real world. This environment should be updated on a regular basis, ideally in an automated way, so we always have the latest stable code available. When testing a user story, even if the main developer is not available, this environment makes possible for a second developer to go there and verify the functionality with the latest build, making team more flexible and efficient. In order to achieve that, a priority for the team is to make sure that test environment is not broken and is available for testing.

Team should consider having other test environments with specific characteristics for activities such as load, performance or stress testing, otherwise these tests won't be meaningful unless they run in an environment that mimics the production environment.

Virtual machines installed in our own hardware or cloud services like Amazon EC2 [22], Microsoft Azure [23] or Google Compute Engine [24] give us the flexibility to run test environments on demand, saving costs and configuration time.

Test Data

Test data requirements vary according to the type of testing. Automated tests usually create their own data or run against a small representational set of data that can be refreshed to a known state quickly. Exploratory testing may need a complete replica of production type data as it provides a good base for different scenarios. Production data needs to be obfuscated before it's used for testing in order to remove any sensitive information such as identification or personal health record details.

Test data tends to get obsolete and out of date over time. Older data, may no longer accurately reflect current production data. Test data needs should be continually reviewed, so we can refresh data or create it using a new approach, ideally with input from our customers, so test data is as meaningful as possible. Development team should think about test data needs when planning the iteration or release as this could be a time consuming task and we will need this data available when stories are ready for testing.

1.3.5 Why

We have been through many reasons so far on why to implement Agile Testing but one of the most obvious reasons why testing is done has not been analysed yet: finding bugs before our customers do.

Whenever we have a concern with software we call it a defect, this means that not only code errors are considered as defects, also deviation from specifications on aspects like scalability, performance or security, just to name a few. Although Agile Testing methodology attempts to prevent bugs through the development cycle it is very likely that we end up introducing some defects to our products and it is key knowing how to deal with them.

Bug Tracking

The main advantage of testing early is that you find bugs early, when they are still cheap to fix. If a defect is found while the story is still being developed, the tester should talk to the developer so this can fix it immediately, there's no need to log the bug anywhere. Although it is highly recommended, if possible, to create a unit test to reproduce the bug, if someone breaks that piece of code later, the test will catch the regression.

Not always a developer is available immediately to work on an bug, and it is possible the defect might be forgotten, in this case the issue should be tracked in the Bug Tracking system. This also applies to problems that weren't caught until after code was released, internally by the team on a subsequent iteration or in the production environment by a customer.

At Alteraid defects are tracked in Pivotal Tracker, this tool helps to manage agile projects, allowing to track user stories and bugs. To keep things simple, Pivotal Tracker's states are limited to Unstarted, Started, Finished, Delivered, Accepted and Rejected. To overcome that limitation, team can use labels in order to clearly define defect states. The following diagram shows a recommended flow for bugs tracked in Pivotal Tracker using the additional labels "needs_QA", "QA_passed" and "QA_failed":

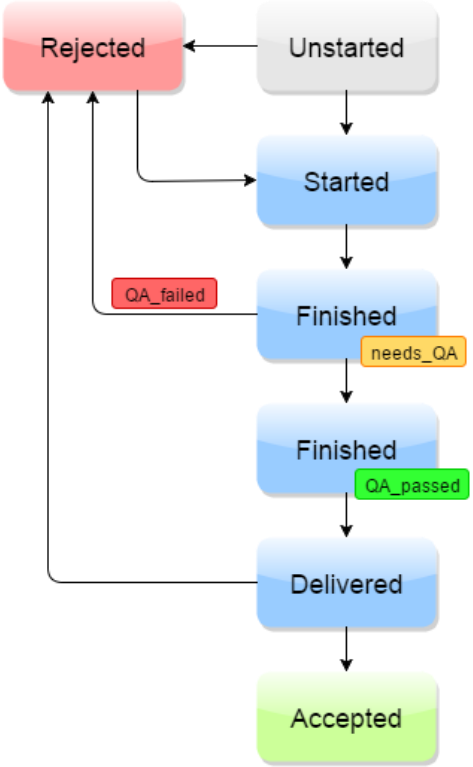

Figure 1.5 Bug lifecycle in Pivotal Tracker with custom labels

A bug is created with the “Unstarted” state by default, a Developer picks that bug to investigate and fix it so it changes state to “Started”. When the bug fix is ready to test, Developer changes state to “Finished” and adds “needs_QA” label. At this point the Tester will pick up that bug story and perform the required tests to check that the bug fix is working as expected, if so, it will replace “needs_QA” label by “QA_passed” and will add a comment with relevant information such as test environment, build number or test details. If bug is still reproducible or Tester is not happy with the fix for other reasons, it will change state to “Rejected” and will add “QA_failed” label. It is very important in this situation to clearly expose the reasons why the bug fix has been rejected, it will help the Developer to focus on the outstanding issues.

Priority vs Severity

The severity of a bug depends on the impact it has on the underlying system, it is a functional measure, and can be estimated right when opening an issue. The priority of a bug is a scheduling decision, depending on business requirements, time required to fix and several other factors.

Severity and priority are clearly two different concepts. Although they are correlated (high severity should generally mean high priority, but not always), they represent different concepts. For example, an issue priority

can change after a rescheduling, while an issue severity can hardly change unless something changes.
There are not dedicated fields in Pivotal Tracker stories for Priority or Severity but development team at Alteraid can use custom labels again for this purpose. It is good to start small and simple so I would recommend only using 3 clear values for each measure:

- Severity: Show Stopper, Work Around or Cosmetic.
- Priority: Fix now, Fix later or Don't fix.

Bug Reports

These are one of the main communication methods that development team members use, it is important to know who is going to read these bug reports, and what they are expected to do with them.

Ideally a bug report should contain all the information needed by the developer to fix the issue, this includes steps to reproduce, observed behaviour and expected behaviour as a bare minimum. It is also highly recommendable to add other information like end-user implications, screenshots, environment where issue is being reported or affected customer.

This is an example Bug Report in Pivotal Tracker of a real issue found on Aaaida website during the preparation of this project:

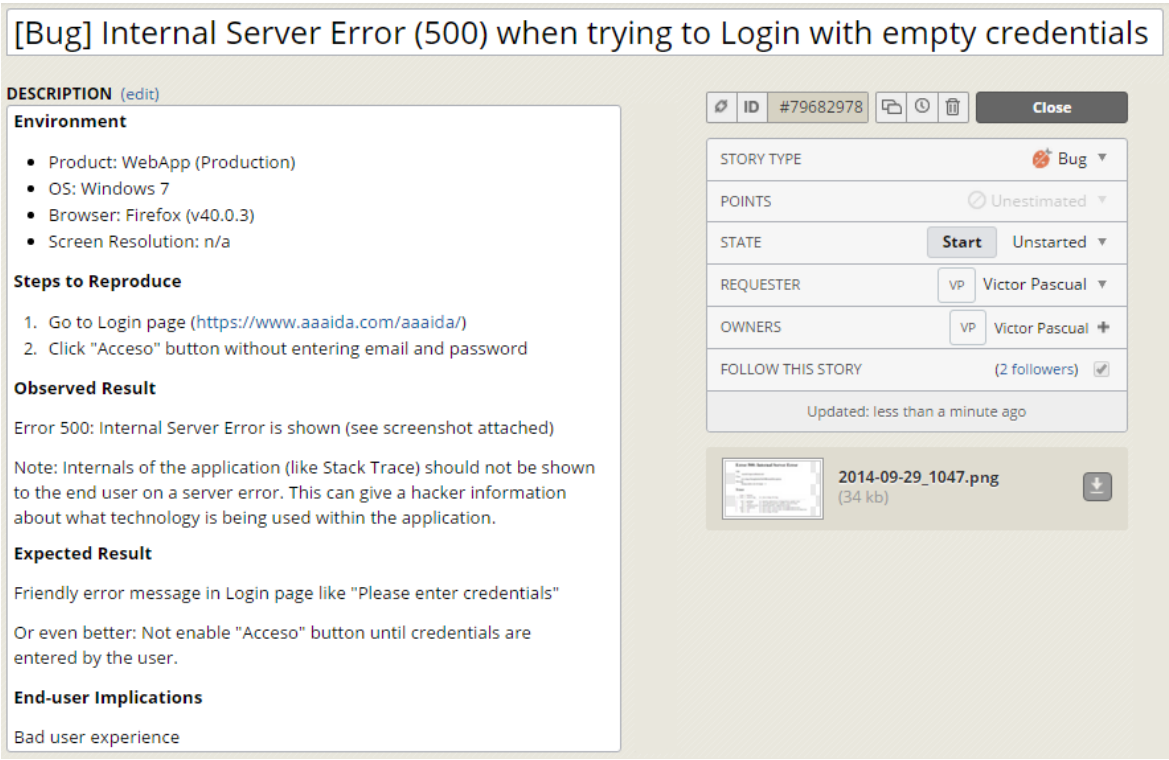

Figure 1.6 Example of Bug Report in Pivotal Tracker

Every meaningful detail in a Bug Report counts, we can see them as a tool to sell developers on the idea of spending their time and energy fixing a bug. When testing, we should prioritize quality over quantity, as the best tester isn't the one who finds the most bugs or embarrasses the most developers. The best tester is the one who gets the most bugs fixed.

CHAPTER 2. AUTOMATE FUNCTIONAL TESTS

Now that we know the theory, we will take one of the main characteristics of Agile Testing, Test Automation, and we will extend in this area. Implementing real world tests for Web and API applications. Focusing that implementation on having a useful and maintainable suite of automated tests that we could also use as documentation for the development team.

2.1 Agile Testing and Automation

Automation is key to successful agile development, it frees the team to deliver high quality code frequently. Automation is an extensive topic: source code control, automated builds, deployment, test suites or monitoring. All these development activities can be automated to a certain extend to eliminate tedium, ensure reliability and allow the team to maximize its velocity while maintaining a high standard.

In this chapter we will focus on how to automate functional tests, as we have seen in the previous chapter, functional tests are located at Quadrant 2 so they support the development team but exercise the code at a higher level than unit or integration tests. These tests are business-facing and derived from feature specifications or examples provided by customers.

The most basic reason a team wants to automate is that it simply takes too long to complete all of the necessary testing manually. As an application gets bigger and bigger, the time to execute a regression test manually grows longer and longer, sometimes exponentially, depending on the complexity of the application. An automated suite of regression tests also provides a safety net for developers, tests will tell them right away whether or not a code change broke anything, so they can go lots faster than if we rely exclusively on manual testing.

After an automated test for a certain feature passes, it must continue to pass until the functionality is intentionally changed. When we plan changes in the application, we change the tests to accommodate them. When an automated test fails unexpectedly, a regression defect may have been introduced by a code change. Running an automated suite of tests often helps ensure that regression bugs will be caught quickly. Quick feedback means the change is still fresh in some developer’s mind, so troubleshooting will be easier than if the bug was not found until some testing phase days or weeks later.

2.2 Test Automation Framework

In order to build an effective and maintainable suite of functional tests, we need to think and provide some architecture to it. Tests in general but functional tests in particular should be easy to read and understand, this is because ideally our

tests will not only check if our application is working as expected, they will also document how the application works.

Functional tests usually exercise a system end to end, like a user will do, so tests can become quite complex when it comes to setting up the test data, authentication, manage environment, navigation, etc. To make the tests simple and readable we need to move as much of this complexity as possible to the automation framework. We also want for our test suite to require minimal effort to maintain, for example, we don't want to be updating hundreds or thousands of tests if an small cosmetic change is introduced to the application under test, we should rather do a single update on our automation framework. It provides an abstraction layer between the tests and the application under test.

As there is already good knowledge of .NET technologies at Alteraid, we will use C# as the programming language and Microsoft Visual Studio 2015 (Community Edition) [25] as the development environment for the automation framework and the functional tests. Using well-known technologies and tools by the team for automation tasks will ease the learning curve and integrate better with the existing code base at Alteraid.

We will develop the framework itself in the following sections, while writing the functional tests. We do not want to anticipate and write code that then we will not use. As the YAGNI (You aren't gonna need it) mantra states: "a programmer should not add functionality until deemed necessary" [26].

For now we can create a solution in Visual Studio named "Alteraid.Automation". This solution will contain a set of C# Class Library projects for the automation framework and MSTest [27] test projects with the functional tests for Aaaida Web and Aaaida API.

2.3 Web App Functional Tests

We will now implement some functional tests for one of the products at Alteraid, the Aaaida web portal. It is a web application that allow users manage their personal health data and authorize other people to review or manage that data for them. In particular we will test some scenarios of the "Login" feature:

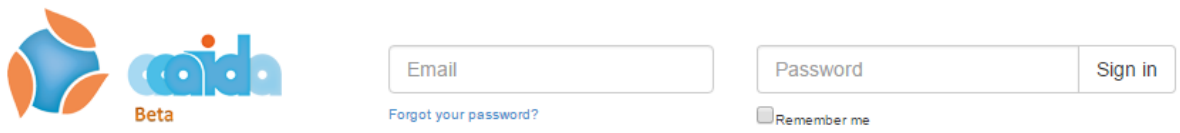

Figure 2.1 Login section in Aaaida web site

Login is a well-known and critical feature across web applications and software products in general, it allow users to authenticate and access their own data. For

Aaaida it is especially important as personal health data is even more sensitive data that other kinds of data. Additionally, Login is usually the first functionality that users will use on the product so a correct functionality and nice user experience is essential.

2.3.1 Test Plan

When writing tests it is important to take some time to think about the functionality that we need to test, this will help us to get the bigger picture to prioritize critical tests and mitigate risks. In a traditional waterfall environment, testers spent time writing bulky test plans that nobody read and nobody bothered to maintain. In agile development, we want test plans to serve our needs for the current iteration or release, keeping them concise and lightweight.

Mind maps are one of the alternatives to traditional test plans, with them we can get a clear picture of the feature to test and they are easy to consume and share with the rest of the development team and business people. The following figure shows a mind map (created with MindMup.com [28]) with things to consider about the “Login” feature:

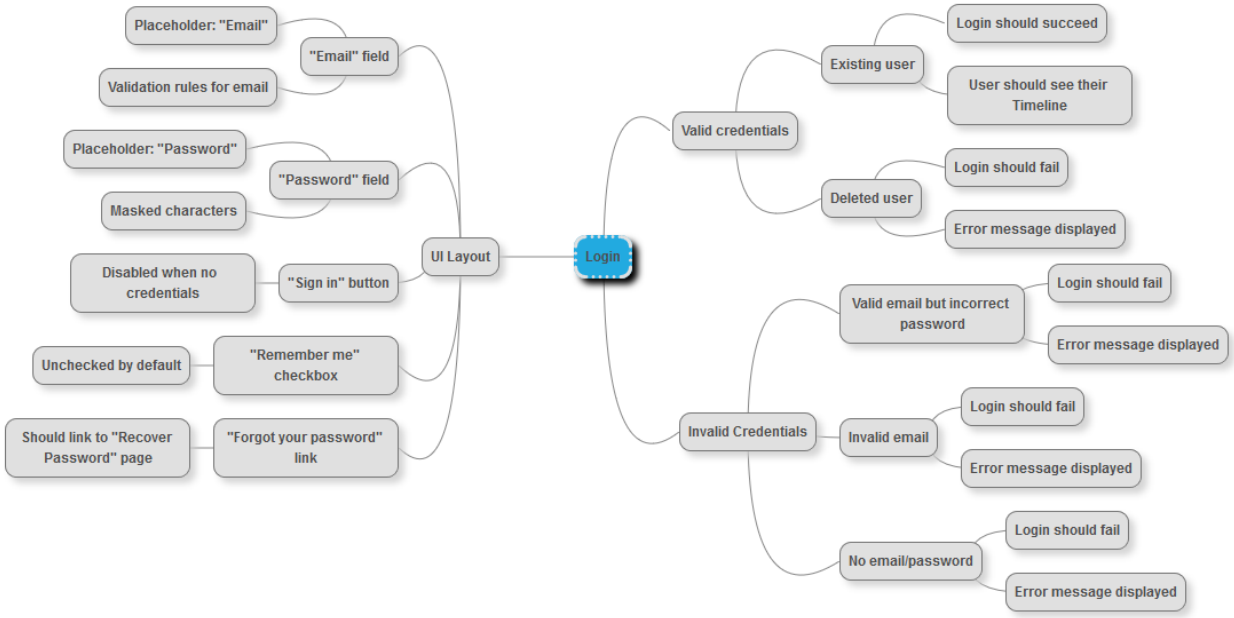

Figure 2.2 Mind map of “Login” feature in Aaaida web app

From the right hand side of this mind map we can easily derive a list of tests for the main functionality and which things we will be checking. From the left hand side, we can also derive some tests that check additional features in the UI layout.

2.3.2 Browser Automation

These functional tests will exercise the application at the higher level, with a real browser, like the end user will do. To automate the web browser we will use Selenium WebDriver [29], this tool was created in 2004 by Jason Huggins, after the years and with the support of the open-source community and main web browser vendors like Google, Microsoft or Mozilla, has become the standard for automating browsers.

To use Selenium WebDriver we just need to add the C# binaries to our project via NuGet (the standard package manager for Visual Studio [30]). The following code snippet shows a simple test that uses Selenium to navigate to Aaaida website, login with valid credentials and assert that login has been successful:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using OpenQA.Selenium;
using OpenQA.Selenium.Firefox;
using System.Threading;

namespace LoginTestsExample
{
    [TestClass]
    public class LoginTests
    {
        IWebDriver driver;

        [TestMethod]
        public void CanLoginWithValidCredentials()
        {
            //Initialize Firefox driver before each test
            driver = new FirefoxDriver();

            //Navigate to Login page
            driver.Navigate().GoToUrl("http://old.aaaida.com");

            //Populate Email
            var emailInput = driver.FindElement(By.Name("email"));
            emailInput.SendKeys("victor.testing@mailinator.com");

            //Populate Password
            var passwordInput = driver.FindElement(By.Name("password"));
            passwordInput.SendKeys("Barcelona1234");

            //Click 'Sign in' button
            var submitButton = driver.FindElement(By.CssSelector("#login-form button"));
            submitButton.Click();

            //Sleep for 1 second while login is being performed
            Thread.Sleep(1000);

            //Assert user in Home page by checking URL
            Assert.IsTrue(driver.Url.Equals("http://old.aaaida.com/aaaida/timeline/index"),
                          "Not in Home page");

            //Assert logged in User is the correct one
            Assert.AreEqual("victor.testing",
                            driver.FindElement(By.CssSelector("#header_navbar .navbar-brand")).Text,
                            "Logged in as an incorrect User");

            //Close driver after each test
            driver.Quit();
        }
    }
}
```

Figure 2.3 Simple test for successful Login

We will not get into the details and internals on how Selenium works as there is a lot of documentation online, we will rather focus on the design for maintainability and readability of the tests, so these tests can be used as documentation for the development team. There are a few things from the previous test that we can improve in order to make tests more maintainable:

Setup and Teardown

If we now write another test, we will need to repeat the code for creating the driver at the beginning of the test and to exit the driver at the end of the test. We can move this code to “TestInitialize” and “TestCleanup” methods that will run before and after each test in the test class:

```
[TestClass]
public class LoginTestsExample
{
    IWebDriver driver;

    [TestInitialize]
    public void Initialize()
    {
        //Initialize Firefox driver before each test
        driver = new FirefoxDriver();
    }

    [TestCleanup]
    public void Cleanup()
    {
        //Close driver after each test
        driver.Quit();
    }
```

Figure 2.4 Initialize and Cleanup methods

Browser abstraction layer

If some day we decide to change Selenium for another browser automation tool we will need to update all our functional tests, this is a pain. Imagine that this new tool can run tests 4x times faster and more reliably, it might still not be justifiable the effort on changing all your existing tests because it would cost so much.

In order to avoid that issue, we can create a new “BrowserManager” project in our automation framework solution to abstract interactions with the browser. Only that project will have references to the Selenium binaries, so if we need to update Selenium to a newer version or even change the browser automation tool we will just do it in one place and we will not need to update all the tests.

For the moment the “BrowserManager” project will only contain one static class to wrap some Selenium functionality:

```
using OpenQA.Selenium;
using OpenQA.Selenium.Firefox;

namespace BrowserManager
{
    public static class Browser
    {
        static IWebDriver driver;

        public static string Url
        {
            get { return driver.Url; }
        }

        public static void InitiliazeFirefox()
        {
            driver = new FirefoxDriver();
        }

        public static void Quit()
        {
            driver.Quit();
        }

        public static void GoTo(string url)
        {
            driver.Navigate().GoToUrl(url);
        }
    }
}
```

Figure 2.5 Browser static class

The class is static so we don't need to declare the "Browser" in the tests and we end up with a more readable suite of tests:

```
[TestInitialize]
public void Initialize()
{
    Browser.InitiliazeFirefox();
}

[TestCleanup]
public void Cleanup()
{
    Browser.Quit();
}

[TestMethod]
public void CanNotLoginWithWrongPassword()
{
    //Navigate to Login page
    Browser.GoTo("http://old.aaaida.com/aaaida/?lang=en");
```

Figure 2.6 Test using BrowserManager

At this stage we still cannot remove the reference to the Selenium binaries because we are using it to locate and interact with the page elements. We will overcome that limitation later in this chapter by using the Page Objects design pattern.

Explicit Waits

In the test example for a successful Login we are forcing the code to wait 1 second after submitting the username and password, this is to make sure that home page has fully loaded before moving to next step. But what if it takes 3 seconds to load? Or only 50ms? For the first, Selenium will throw an exception because the element that we are trying to interact with in the next step is not present yet. For the second, we will just be wasting almost 1 second when we do not need to.

Selenium provides a better way to wait for certain conditions to happen, the Explicit Waits [31]. To use this advanced feature we need to reference the “Selenium.Support” NuGet package in the “BrowserManager” project.

Once we wrap the advanced waiting mechanism in our “Browser” static class we can now wait for an element to be present in the page before moving forward:

```
//Wait until a certain element is visible
Browser.Wait.Until(ExpectedConditions.ElementIsVisible(By.Id("filter-form")));
```

Figure 2.7 Explicit Wait for a certain element to be visible

Using these explicit waits we will only be waiting the necessary amount of time before the condition is met. There is also a configurable timeout (10 seconds by default), so if the condition is not met in time Selenium will throw an exception.

In the example above we are waiting for a certain element to be present but Selenium provides, via de “ExpectedConditions” class [32], more conditions to be met like url to be an specific value or an element to be invisible or not present.

Logging and Screenshots

When a test fails, we want to spend as less time as possible figuring out why it failed. Logging helps to troubleshoot failed tests providing useful information about where and, ideally, why the test failed. We could implement a custom logging system just printing to Console the important steps in the test but we just rather prefer to use an existing library that fits this purpose so we don’t have to reinvent the wheel.

NLog is an open-source logging platform for .NET [33], easy to configure, can write the logs to different targets like files, console, email or database. To use this library we need to add a reference to the NLog NuGet package on the “BrowserManager” project.

Now we can create a “Logger” object in the “Browser” static class and use it to log any significant step. We will just use the “Info” logging level in this example but we also have “Trace”, “Debug”, “Warn”, “Error” and “Fatal” available [34]:

```
private static readonly Logger Logger = LogManager.GetCurrentClassLogger();

public static void InitiliazeFirefox()
{
    Logger.Info("Initalizing Firefox browser");
    driver = new FirefoxDriver();
}

public static void Quit()
{
    Logger.Info("Closing driver");
    driver.Quit();
}

public static void GoTo(string url)
{
    Logger.Info("Navigating to '{0}'", url);
    driver.Navigate().GoToUrl(url);
}
```

Figure 2.8 Logging in “Browser” class

To keep it simple we will just log the information to the console, so it will be easily discoverable when troubleshooting failed tests. We can also specify a template for the log format to include timestamp, class, level or any other important data. In order to configure all this settings we need to add an “nlog” section in the “App.config” file of the functional tests project:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="nlog" type="NLog.Config.ConfigSectionHandler, NLog" />
  </configSections>
  <nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <targets>
      <target xsi:type="Console"
              name="console"
              layout="${date} | ${uppercase:${level}} | ${logger:shortName=true} | ${message}" />
    </targets>
    <rules>
      <logger name="*" minlevel="Trace" writeTo="console" />
    </rules>
  </nlog>
</configuration>
```

Figure 2.9 NLog configuration in App.config file

If we now run our test in Visual Studio we can see all the logs in the standard output of the test run summary:

Test Name: CanLoginWithValidCredentials

Test Outcome: Passed

Standard Output

```
2015/09/05 13:08:45.307 | INFO | LoginTests | Initalizing Firefox browser
2015/09/05 13:08:49.303 | INFO | LoginTests | Navigating to 'http://old.aaaida.com/aaaida/?lang=en'
2015/09/05 13:08:52.663 | INFO | LoginTests | Populating 'victor.testing@mailinator.com' Email
2015/09/05 13:08:52.905 | INFO | LoginTests | Populating Password
2015/09/05 13:08:52.978 | INFO | LoginTests | Clicking 'Sign in' button
2015/09/05 13:08:55.709 | INFO | LoginTests | Closing driver
```

Figure 2.10 Test with logging implemented with NLog

For unit tests, a good logging system would be enough because the tests are checking only a small piece of logic, for functional tests exercising a real browser good logging is still essential but sometimes is not enough. Selenium WebDriver provides a mechanism to take screenshots of the browser at any time and save them to a file:

```
public static void TakeScreenshot(string filePath)
{
    Logger.Info("Taking Screenshot");
    var screenshot = ((ITakesScreenshot)_driver).GetScreenshot();
    screenshot.SaveAsFile(filePath, ImageFormat.Png);
}
```

Figure 2.11 Method in the “Browser” class to take an screenshot

With this method implemented we can now modify our Cleanup() method to take an screenshot of the browser every time that a test fails. We will use the TestContext class provided by Visual Studio unit test framework to determine on runtime if a test has passed or failed:

```
[TestCleanup]
public void Cleanup()
{
    if(TestContext.CurrentTestOutcome != UnitTestOutcome.Passed)
    {
        Browser.TakeScreenshot(Path.Combine(ScreenshotsPath,
                                            TestIdentifier +
                                            "_screenshot.png"));
    }

    Logger.Info("Closing driver");
    Browser.Quit();
}
```

Figure 2.12 Cleanup method taking an screenshot if test is not passed

Screenshots are not only useful when debugging failed tests, we can also take an screenshot on any key step and then manually review the library of screenshots looking for any UI issues like images not rendering or CSS layout errors.

Cross-Browser support

One of the biggest challenges when testing web applications is the variety of browsers in the market. Pages might render correctly in Google Chrome but be completely broken in Internet Explorer 9. This is because browsers are implemented differently and use different rendering engines. When testing a web application we want to make sure that a certain feature works in the most used browsers so we cover most potential users.

Selenium comes with built-in support only for Mozilla Firefox but this can be extended to other browsers by using specific driver server implementations. For Google Chrome and Internet Explorer, the drivers were created and maintained by the open-source community and now are also supported by their respective browser vendors, Google and Microsoft.

We will now update the “BrowserManager” project to add the capability of also running tests in Google Chrome and Internet Explorer. To do that, we can add a NuGet package with the driver binary for each project:

- Selenium.WebDriver.ChromeDriver
- Selenium.WebDriver.IEDriver

These NuGet packages will add the driver binaries as linked project items and will copy them to bin folder during build:

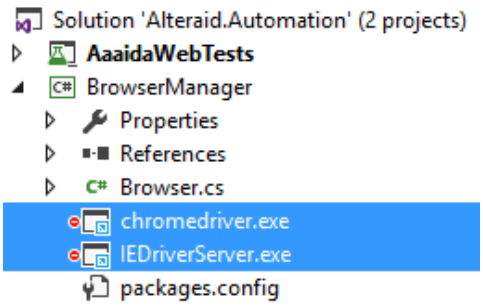

Figure 2.13 Chrome and IE driver binaries in “BrowserManager” project

We now need a mechanism to the tell tests against which browser do we want them to run. As we might want to do that on runtime, to be more flexible, we can add the desired browser in the “App.config” file. So we just need to change an entry on that file rather than updating the code itself:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="nlog" type="NLog.Config.ConfigSectionHandler, NLog" />
  </configSections>
  <nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <targets>
      <target xsi:type="Console"
              name="console"
              layout="${date} | ${uppercase:${level}} | ${logger:shortName=true} | ${message}" />
    </targets>
    <rules>
      <logger name="*" minlevel="Trace" writeTo="console" />
    </rules>
  </nlog>
  <appSettings>
    <add key="BrowserName" value="firefox" />
    <!--Options: firefox / chrome / internetexplorer-->
  </appSettings>
</configuration>
```

Figure 2.14 Section in App.Config to specify browser

From the test class we need to load that value from the App.config file on runtime and pass it to the Browser.Initialize() method that we need to update in order to initialize the specified browser, because currently is only initializing a Firefox instance:

```csharp
static string BrowserName = ConfigurationManager.AppSettings["BrowserName"];

[TestInitialize]
public void Initialize()
{
    Browser.Initiliaze(BrowserName);
}
```

Figure 2.15 Initializing the Browser specified in App.Config file

```csharp
public static void Initialize(string browserName)
{
    switch (browserName)
    {
        case "firefox":
            Logger.Info("Initalizing Firefox browser");
            driver = new FirefoxDriver();
            break;

        case "chrome":
            Logger.Info("Initalizing Chrome browser");
            driver = new ChromeDriver();
            break;

        case "internetexplorer":
            Logger.Info("Initalizing IE browser");
            driver = new InternetExplorerDriver();
            break;
    }
}
```

Figure 2.16 Initialize() method updated to launch different browsers

2.3.3 Page Object Pattern

Test design skills have a huge impact on whether automation pays off right away. Poor practices produce tests that are hard to understand and maintain, and may produce hard-to-interpret results or false failures that take time to research. Good test design practices produce simple, well-designed, continually refactored, maintainable tests.

If we review again the initial implementation of the test in Figure 2.3 we can spot that if one of the element locators changes, for example the one for the “Sign in” button, we will need to update all the references to the old locator in the tests. Additionally, every time we want to Login in our tests we will need to write lots of lines of code. To overcome these two issues we can use the Page Object pattern [35].

Page Object is a design pattern which has become popular in test automation for enhancing test maintenance and reducing code duplication. A page object is an object-oriented class that serves as an interface to a page of your application under test. The tests then use the methods of this page object class whenever they need to interact with that page of the UI. The benefit is that if the UI changes for the page, the tests themselves don’t need to change, only the code within the page object needs to change. Subsequently all changes to support that new UI are located in one place.

We can start by thinking on how we would like our test to look like. Having in mind readability and maintainability one possible test implementation would be like the following:

```
[TestMethod]
public void CanLoginWithValidCredentials()
{
    //Arrange
    Browser.GoTo(AaaidaWeb.LoginUrl);

    //Act
    AaaidaWeb.LoginPage.Login("victor.testing@mailinator.com", "Barcelona1234");

    //Assert
    Assert.IsTrue(AaaidaWeb.HomePage.IsAt(), "Not in Home page");
    Assert.AreEqual("victor.testing", AaaidaWeb.HomePage.LoggedInUser,
                    "Logged in User is not the correct one");
}
```

Figure 2.17 Test for successful Login using PageObjects

Notice the comments to differentiate the 3 different phases that all test should have: Arrange, Act and Assert [36]. It is not always necessary to add the

comments but at least is recommendable to leave a blank line between each phase.

In the Arrange phase we use our “Browser” class, the one that wraps the browser interactions with Selenium, to navigate to the Login page. Rather than passing a hardcoded url of the Login page we get it from a property in the “AaaidaWeb” static class for better maintainability. This class represents our application under test, the Aaaida web portal.

Next phase is the one where we perform the action that we are specifically checking in the test, the successful Login. Notice that we use again the “AaaidaWeb” class to provide the “LoginPage” page object which contains a method to perform the Login with specified email and password.

We can start by creating the “LoginPage” page object that contains that Login() method. For this page we need to define the locators for the different elements: email field, password field and submit button. We will also need to implement the method to perform the Login with given credentials. To keep things tidy we can create a new Class Library project in the solution to host all our PageObjects:

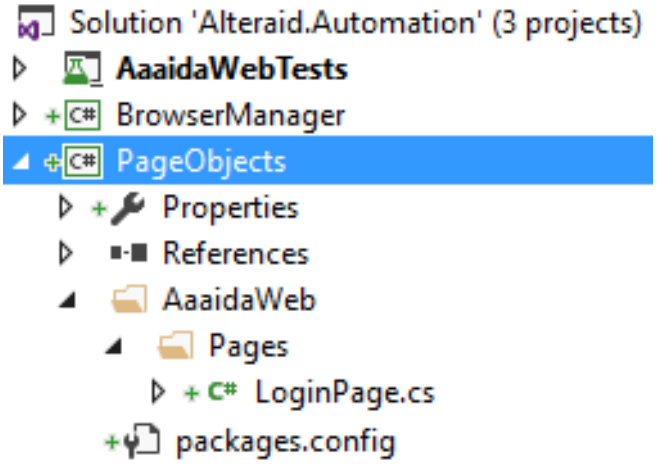

Figure 2.18 “PageObjects” project in the solution

In order to support the PageObject pattern, Selenium WebDriver's support library contains a factory class, using the “PageFactory” and it’s “FindsBy” attributes we can assume that the fields are initialised in a page object. If we don't use the PageFactory, then “NullPointerExceptions” will be thrown if we make the assumption that the fields are already initialised.

```
using NLog;
using OpenQA.Selenium;
using OpenQA.Selenium.Support.PageObjects;

namespace PageObjects.AaaidaWeb.Pages
{
    public class LoginPage
    {
        private static readonly Logger Logger = LogManager.GetCurrentClassLogger();

        //Locators

        [FindsBy(How = How.Name, Using = "email")]
        private IWebElement _emailInput;

        [FindsBy(How = How.Name, Using = "password")]
        private IWebElement _passwordInput;

        [FindsBy(How = How.CssSelector, Using = "#login-form button")]
        private IWebElement _submitButton;

        [FindsBy(How = How.CssSelector, Using = ".col - lg - 8.alert")]
        private IWebElement _errorMessage;

        //Methods

        public void Login(string email, string password)
        {
            PopulateEmail(email);
            PopulatePassword(password);
            SignIn();
        }

        public void PopulateEmail(string email)
        {
            Logger.Info("Populating '{0}' in email field", email);
            _emailInput.SendKeys(email);
        }

        public void PopulatePassword(string password)
        {
            Logger.Info("Populating '{0}' in password field", password);
            _passwordInput.SendKeys(password);
        }

        public void SignIn()
        {
            Logger.Info("Clicking 'Sign in' button");
            _submitButton.Click();
        }
    }
}
```

Figure 2.19 “LoginPage” page object using “PageFactory” attributes for locators

We have grouped all the locators into one section so we can find them easily when working on a page object. Additionally, instead of adding all the logic into the Login() method we just preferred to break down in smaller public methods

that we can reuse for other tests. Notice as well that we have added logging for each key method.

Now we need to implement the “AaaidaWeb” class, it should be a static class, so we do not need to declare it in the tests and for now it will provide us the LoginUrl and the LoginPage:

```
using BrowserManager;
using OpenQA.Selenium.Support.PageObjects;
using PageObjects.AaaidaWeb.Pages;

namespace PageObjects.AaaidaWeb
{
    public static class AaaidaWeb
    {
        //Properties

        public static string LoginUrl
        {
            get { return "http://old.aaaida.com/aaaida/?lang=en"; }
        }

        //Pages

        public static LoginPage LoginPage
        {
            get
            {
                var loginPage = new LoginPage();
                PageFactory.InitElements(Browser.Driver, loginPage);
                return loginPage;
            }
        }
    }
}
```

Figure 2.20 “AaaidaWeb” static class using “PageFactory”

It provides the page objects using the ”PageFactory” class [37], included in the Selenium’s support library, that initializes the elements on the page. The InitElements() method takes the Selenium driver and the page object so we had to expose the Selenium driver into a public property in the “Browser” class:

```
namespace BrowserManager
{
    public static class Browser
    {
        private static IWebDriver _driver;

        public static IWebDriver Driver
        {
            get { return _driver; }
        }
```

Figure 2.21 “Browser” class exposing the Selenium driver

From our refactored test for the successful Login we can now move to the “Assert” phase. As the Login should succeed, we want to assert that we are in the Home page and also that the logged in user is the correct one. To do that, we need to create a “HomePage” page object with a method that checks if we are in the HomePage and also a property with the name of the logged in user shown in UI:

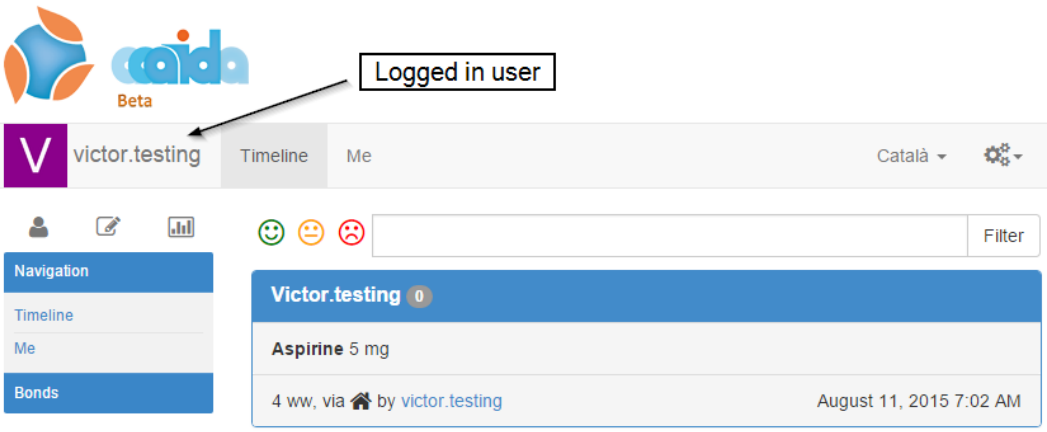

Figure 2.22 Home page for logged in users in Aaaida web site

```
using BrowserManager;
using OpenQA.Selenium;
using OpenQA.Selenium.Support.PageObjects;

namespace PageObjects.AaaidaWeb
{
    public class HomePage
    {
        //Locators

        [FindsBy(How = How.CssSelector, Using = "#header_navbar .navbar-header .navbar-brand")]
        private IWebElement _userNameField;

        //Properties

        public string LoggedInUser
        {
            get
            {
                return _userNameField.Text;
            }
        }

        //Methods
        public bool IsAt()
        {
            return Browser.Url == "http://old.aaaida.com/aaaida/timeline/index";
        }
    }
}
```

Figure 2.23 “HomePage” page object

We also need to modify the “AaaidaWeb” class to expose the Home page url and to provide the “HomePage” page object:

```
using BrowserManager;
using OpenQA.Selenium.Support.PageObjects;
using PageObjects.AaaidaWeb.Pages;

namespace PageObjects.AaaidaWeb
{
    public static class AaaidaWeb
    {
        //Properties

        public static string LoginUrl
        {
            get { return "http://old.aaaida.com/aaaida/?lang=en"; }
        }

        public static string HomeUrl
        {
            get { return "http://old.aaaida.com/aaaida/timeline/index"; }
        }

        //Pages

        public static LoginPage LoginPage
        {
            get
            {
                var loginPage = new LoginPage();
                PageFactory.InitElements(Browser.Driver, loginPage);
                return loginPage;
            }
        }

        public static HomePage HomePage
        {
            get
            {
                var homePage = new HomePage();
                PageFactory.InitElements(Browser.Driver, homePage);
                return homePage;
            }
        }
    }
}
```

Figure 2.24 “AaaidaWeb” class that provides “LoginPage” and “HomePage”

We now need to integrate the Explicit Waits seen in the previous section to the PageObjects architecture. We want to move all the “waiting” logic to the page objects so we do not need to worry about that when writing tests. There are usually two situation where we will need to wait:

- Action to be completed (i.e form to be submitted)
- Page to be loaded

For the first, we can just add the waiting code that we have seen previously in the method that executes the target action. Following example shows the implementation of a method to submit a form, clicking a “Submit” button and waiting for a loading icon to not be displayed, which indicates the form has been submitted:

```
public void SubmitForm()
{
    _submitButton.Click();
    Browser.Wait.Until(x => !_loadingIcon.Displayed);
}
```

Figure 2.25 “AaaidaWeb” class that provides “LoginPage” and “HomePage”

The second is the most common one, waiting for a certain page to load. As we will need that same functionality for every page object we can build it into the architecture of the pages. Every time that we request a page to the “AaaidaWeb” class, we will be waiting for a certain condition to be met, we have to choose a different condition for every page. For example, in the Login page we can wait until the “Submit” button is displayed, and in the Home page we can wait until the user timeline is shown.

We can create an IsLoaded() method in each page object that will return “true” when we consider that the page is loaded. In the following example for the Home page we wait until the “Displayed” property of the timeline container element returns true:

```
public bool IsLoaded()
{
    Logger.Info("Waiting for page to load...");
    var timelineContainer =
        Browser.Wait
                .Until(ExpectedConditions.ElementIsVisible(By.Id("advertisementsList")));

    return timelineContainer.Displayed;
}
```

Figure 2.26 IsLoaded() method for Login page

Now in the page factory, the “AaaidaWeb” class, we need to wait for the IsLoaded() method to be “true” before returning the corresponding page object:

```
public static LoginPage LoginPage
{
    get
    {
        var loginPage = new LoginPage();
        Browser.Wait.Until(x => loginPage.IsLoaded());
        PageFactory.InitElements(Browser.Driver, loginPage);
        return loginPage;
    }
}
```

Figure 2.27 LoginPage property in “AaaidaWeb” class waits for page object to be loaded

Doing it this way we keep tests free of waiting logic and we ensure that page objects are loaded before trying to interact with them. Waiting is one of the most trickiest issues when working with browser automation tools so it is very important to invest some time in building a page object architecture that deals with waits in an easy and understandable way.

2.3.4 Extending Test Suite

With all that architecture in place we will be ready to extend our test suite with tests that look clean and are easy to read. Before extending the suite we can do some refactoring to reduce code duplication.

To simplify the creation of new test classes, we can create an abstract base test class that will have all the basic setup that all the test classes will share. Then we will just need to inherit the test classes from the base test class to get all the setup code.

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using BrowserManager;
using System.IO;
using NLog;
using System.Configuration;

namespace AaaidaWebTests
{
    [TestClass]
    public abstract class BaseAaaidaWebTest
    {
        private static readonly Logger Logger = LogManager.GetCurrentClassLogger();
        private static readonly string BrowserName = ConfigurationManager.AppSettings["BrowserName"];
        public TestContext TestContext { get; set; }

        private string TestIdentifier
        {
            get
            {
                return string.Format("{0}_{1}", TestContext.TestName,
                                     DateTime.UtcNow.ToString("yyyyMMdd_HHmmss"));
            }
        }

        private string ScreenshotsPath
        {
            get
            {
                return Path.Combine(TestContext.ResultsDirectory, "Screenshots");
            }
        }

        [TestInitialize]
        public void Initialize()
        {
            Browser.Initialize(BrowserName);
        }

        [TestCleanup]
        public void Cleanup()
        {
            if (TestContext.CurrentTestOutcome != UnitTestOutcome.Passed)
            {
                Browser.TakeScreenshot(Path.Combine(ScreenshotsPath,
                                                    TestIdentifier +
                                                    "_screenshot.png"));
            }
            Browser.Quit();
        }
    }
}
```

Figure 2.28 Base test class with common setup code

Another improvement we can do before writing more tests is creating a base page object, all page objects will share the IsLoaded() method and the logging capabilities, so in order to avoid code duplication we can create a “BasePageObject” abstract with an IsLoaded() abstract method. Each page object will inherit from the base one so we will need to implement the IsLoaded() for each inherited page object.

```
public abstract class BasePageObject
{
    protected BasePageObject()
    {
        Logger = LogManager.GetLogger(GetType().FullName);
    }

    protected Logger Logger { get; private set; }

    internal abstract bool IsLoaded();
}
```

Figure 2.29 Base page object class

The approach to extending the test suite now will be to write the test code first and then implement the needed parts in the automation framework, this way we are driven by the testing needs and we avoid implementing functionality in the test automation framework that we will not need for the test.

“Login” is a key feature of any application so it would be very useful to have a small subset of tests at the functional level that can catch any regression issue during development. Although, we should have in mind that tests at this level are always slower and less reliable that other tests at lower levels, so when developing this kind of tests we should be always trying to cover the most critical paths and try to cover the rest of the functionality with unit or integration tests when possible.

2.4 API Functional Tests

Nowadays software is composite and layered, 10 years ago most of the applications were talking directly to a database, while now API’s act as a middleware component that is easier to maintain and can be reused for different applications. Aaaida platform consists of the web app portal that we have seen in the previous section, and several mobile apps like Adheptor or Virginia. A key component in the platform is the Aaaida API, which all the apps use to manage all the useful information that the platform provides, like users, bonds, medicines or treatments.

Having a reliable API is essential for a company like Alteraid, a simple error in the API can break all the different mobile and web apps so investing time in testing the API end-to-end is well worth it.

In this section, we will focus on testing the “Medicine” resource in the API. It allows users to manage all CRUD operations for individual medicines in the platform and also allows to get all the medicines with the “Medicines” resource.

We can create a new “AaaidaApiTests” test project in our “Alteraid.Automation” solution. We could create a separate solution for the API tests but it is recommendable to keep it all in the same solution as we will be able reuse the infrastructure of the test automation framework. An example would be when testing the “Login” feature of the Aaaida web app, we could delete a user via the API and then check that we cannot longer login with the credentials of that user via Aaaida web portal.

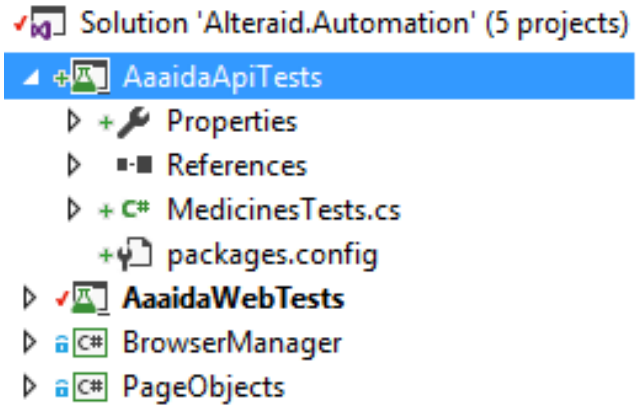

Figure 2.30 “AaaidaApiTests”project in “Alteraid.Automation” solution

2.4.1 API Client

To test the API end-to-end like a mobile or web app does we need to use an API client to send requests and receive responses from the API endpoint. In the .NET framework we could use the “HttpClient” class to perform that operations but we will rather use RestSharp [38] to keep things simple.

RestSharp is an open-source REST and HTTP API Client for .NET, it has helpful features like automatic XML and JSON deserialization that will help us with our implementation. It is actively developed and maintained and we can add it to the project via NuGet.

We will start by writing a simple test to request all Medicines using the RestSharp client. For the API functional tests we will also follow the rules seen in the previous chapter, we will try to differentiate the three phases (Arrange, Act, Assert) and we will try to make our tests as clean and readable as possible, so they serve us as documentation.

```
[TestMethod]
public void CanGetMedicines()
{
    var client = new RestClient { BaseUrl = new Uri("http://147.83.113.216:40000") };
    var request = new RestRequest(Method.GET) { Resource = "Medicines" };

    var response = client.Execute<Medicines>(request);

    Assert.AreEqual(HttpStatusCode.OK, response.StatusCode,
                    "Status code should be 200 (OK)");
    Assert.AreEqual(response.Data.Status.Count, response.Data.Result.Count,
                    "Medicines count does not match");
}
```

Figure 2.31 API test using RestSharp to get Medicines

In the first line we declare the client with the desired API endpoint that is going to hit, in our case one of the test environments at Alteraid. We then create the request specifying the HTTP verb and the resource that we want to get. This will result on the following HTTP request:

- GET http://147.83.113.216:40000/Medicines

Then we can execute that request through the client to get the response back. We can specify the response data model in the Execute method so we get the automatic deserialization feature that RestSharp provides.

```
public class MedicinesResponse
{
    public Status Status { get; set; }
    public List<Medicine> Result { get; set; }
}

public class Status
{
    public int Count { get; set; }
    public int SearchCount { get; set; }
}

public class Medicine
{
    public string Name { get; set; }
    public string Description { get; set; }
    public string Url { get; set; }
    public string _id { get; set; }
}
```

Figure 2.32 Medicines response data model

In the assertion phase we check that the HTTP status code of the response is 200 (OK) and the number of medicines returned and the “Count” value match. As we have provided a data model to the Execute method, the response is deserialized and we can access the different properties in the model like “Status.Count” and “Result”.

Before adding more tests we could do some refactoring to make API functional tests simpler to write and more maintainable:

Setup

As we will need to create the API client for each test we can move that code to the “TestInitialize” method, so it will run before each test in the test class.

```
private RestClient client;

[TestInitialize]
public void Initialize()
{
    client = new RestClient { BaseUrl = new Uri("http://147.83.113.216:40000") };
}
```

Figure 2.33 “TestInitialize” method to create API client

API abstraction layer

In our aim to keep tests simple, we can move all the API client logic to a different project as we have done with the “Browser” logic in the previous section. This will also allow us to reuse the API client for any other purpose in the automation solution, like preparing data for other types of tests.

The new “AaaidaApiClient” project will contain all the logic to use the API, it will also contain all the data models for the different API responses. Following a similar approach as with the web app tests, we will create an “AaaidaApi” static class that will manage the API client and provide the resources and its methods that we can use.

```
using AaaidaApiClient.Resources;
using RestSharp;
using System;

namespace AaaidaApiClient
{
    public static class AaaidaApi
    {
        private static RestClient _client;

        //Resources

        public static MedicinesResource Medicines
        {
            get { return new MedicinesResource(); }
        }

        //Methods

        public static void Initialize(string endpoint)
        {
            _client = new RestClient { BaseUrl = new Uri(endpoint) };
        }

        public static IRestResponse Execute(IRestRequest request)
        {
            return _client.Execute(request);
        }

        public static IRestResponse<T> Execute<T>(IRestRequest request)
            where T : new()
        {
            return _client.Execute<T>(request);
        }
    }
}
```

Figure 2.34 “AaaidaApi” static class to manage API client

The methods are just wrappers for the RestSharp code to initialize the client and execute the API requests. The Initialize() method now takes a parameter which is the API endpoint, this will allow to initialize the API client for different test or production endpoints.

This class will also act as the resources factory, each resource will provide the available methods in the API for the CRUD operations: create, read, update and delete.

```
public class MedicinesResource
{
    public IRestResponse<MedicinesResponse> Get()
    {
        var request = new RestRequest(Method.GET) { Resource = "Medicines" };
        var response = AaaidaApi.Execute<MedicinesResponse>(request);
        return response;
    }
}
```

Figure 2.35 “Medicines” resource with Get() method implementation

With these changes, now the test will look more readable and the API functional tests suite will be easier to extend, even by team members that are not experts on the implementation of the Aaaida API.

```
[TestMethod]
public void CanGetMedicines()
{
    var response = AaaidaApi.Medicines.Get();

    Assert.AreEqual(HttpStatusCode.OK, response.StatusCode,
                    "Status code should be 200 (OK)");
    Assert.AreEqual(response.Data.Status.Count, response.Data.Result.Count,
                    "Medicines count does not match");
}
```

Figure 2.36 Test for getting all Medicines with API abstraction layer

2.4.2 CRUD Tests

With all that architecture in place, we can now extend our suite of tests to add the rest of the CRUD operations for medicines. Actions on individual medicines are done via the “Medicine” resource, so we need to create this resource in the API client. This resource will perform the operations by using the following HTTP verbs:

- GET to Read
- PUT to Create
- POST to Update
- DELETE to Delete

In the Aaaida API, we can get individual medicines by specifying the medicine name as a url segment parameter:

- GET http://147.83.113.216:40000/Medicine/Ibuprofeno

With RestSharp library we can easily add the url segment parameter to the API request:

```
public class MedicineResource
{
    public IRestResponse<Medicine> Get(string medicineName)
    {
        var request = new RestRequest(Method.GET) { Resource = "Medicine/{Name}" };
        request.AddUrlSegment("Name", medicineName);
        var response = AaaidaApi.Execute<Medicine>(request);
        return response;
    }
}
```

Figure 2.37 Method in the “MedicineResource” to get a medicine by name

We can now add a test to the suite to read an individual medicine, it will use the Get() method in the “MedicineResource” that we have just created. We can assert that the status code is OK and also that the medicine returned back by the API is the correct one by checking the medicine name.

```
[TestMethod]
public void CanGetMedicine()
{
    var medicineName = "Ibuprofeno";

    var response = AaaidaApi.Medicine.Get(medicineName);

    Assert.AreEqual(HttpStatusCode.OK, response.StatusCode, "Status code should be 200 (OK)");
    Assert.AreEqual(medicineName, response.Data.Name, "Medicine name does not match");
}
```

Figure 2.38 Method in the “MedicineResource” to get a medicine by name

To create a medicine we need to POST a json object to the API with the medicine minimum details: name, description and url. We can implement a Create() method in the “MedicineResource” for that.

```
public IRestResponse<Medicine> Create(object jsonBody)
{
    var request = new RestRequest(Method.POST) { Resource = "Medicine" };
    request.RequestFormat = DataFormat.Json;
    request.AddBody(jsonBody);
    var response = AaaidaApi.Execute<Medicine>(request);
    return response;
}
```

Figure 2.39 Method to Create a medicine

For updating an existing medicine we have to specify the name of the target medicine via url segment parameter and also PUT a json object with the new medicine details:

```
public IRestResponse<Medicine> Update(string medicineName, object jsonBody)
{
    var request = new RestRequest(Method.PUT) { Resource = "Medicine/{Name}" };
    request.AddUrlSegment("Name", medicineName);
    request.RequestFormat = DataFormat.Json;
    request.AddBody(jsonBody);
    var response = AaaidaApi.Execute<Medicine>(request);
    return response;
}
```

Figure 2.40 Method to Update a medicine

When deleting a medicine we should use the DELETE verb, specifying the name of the medicine to delete in the url. The API will respond with a “No Content” HTTP status code to indicate that the medicine has been correctly deleted.

```
public IRestResponse<Medicine> Delete(string medicineName)
{
    var request = new RestRequest(Method.DELETE) { Resource = "Medicine/{Name}" };
    request.AddUrlSegment("Name", medicineName);
    var response = AaaidaApi.Execute<Medicine>(request);
    return response;
}
```

Figure 2.41 Method to Delete a medicine

Now that we have the four CRUD methods implemented, we can extend the “MedicineTests” test class to include one test for each action. The tests for Create, Update and Delete will modify data, we need to be careful and revert any action that has been done in the test. If not we will end up with a flaky test suite because tests rely on certain data or the tests need to be run in a certain order, which is not recommended.

```
[TestMethod]
public void CanGetMedicine()
{
    var medicineName = "Ibuprofeno";

    var response = AaaidaApi.Medicine.Get(medicineName);

    Assert.AreEqual(HttpStatusCode.OK, response.StatusCode,
                    "Status code should be 200 (OK)");
    Assert.AreEqual(medicineName, response.Data.Name,
                    "Medicine name does not match");
}
```

Figure 2.42 Test for getting a medicine

```
[TestMethod]
public void CanCreateMedicine()
{
    //Arrange
    var medicineJson = new
        {
            name = "TestMedicine",
            description = "A dummy description for the test medicine",
            url = "http://testmedicine.url"
        };

    //Act
    var response = AaaidaApi.Medicine.Create(medicineJson);

    //Assert
    Assert.AreEqual(HttpStatusCode.Created, response.StatusCode,
                    "Status code should be 201 (Created)");
    Assert.AreEqual(medicineJson.name, response.Data.Name,
                    "Medicine name does not match");

    //Revert - Delete the test medicine
    AaaidaApi.Medicine.Delete(medicineJson.name);
}
```

Figure 2.43 Test for creating a medicine

```
[TestMethod]
public void CanUpdateMedicine()
{
    //Arrange - Create test medicine to be updated
    var medicineJson = new
    {
        name = "TestMedicine",
        description = "A dummy description for the test medicine",
        url = "http://testmedicine.url"
    };
    AaaidaApi.Medicine.Create(medicineJson);

    var updatedMedicineJson = new
    {
        name = "TestMedicine",
        description = "An UPDATED dummy description for the test medicine",
        url = "http://testmedicine.url"
    };

    //Act
    var response = AaaidaApi.Medicine.Update(medicineJson.name, updatedMedicineJson);

    //Assert
    Assert.AreEqual(HttpStatusCode.OK, response.StatusCode,
        "Status code should be 200 (OK)");
    //Check that medicine has been correctly updated
    Assert.AreEqual(updatedMedicineJson.description,
                    AaaidaApi.Medicine.Get(updatedMedicineJson.name).Data.Description,
                    "Medication description has not been updated");

    //Revert - Delete the updated test medicine
    AaaidaApi.Medicine.Delete(updatedMedicineJson.name);
}
```

Figure 2.44 Test for updating a medicine

```
[TestMethod]
public void CanDeleteMedicine()
{
    //Arrange - Create test medicine to be deleted
    var medicineJson = new
    {
        name = "TestMedicine",
        description = "A dummy description for the test medicine",
        url = "http://testmedicine.url"
    };
    AaaidaApi.Medicine.Create(medicineJson);

    //Act
    var response = AaaidaApi.Medicine.Delete(medicineJson.name);

    //Assert
    Assert.AreEqual(HttpStatusCode.NoContent, response.StatusCode,
                    "Status code should be 204 (No Content)");
    //Medicine does not longer exist
    Assert.AreEqual(HttpStatusCode.NotFound,
                    AaaidaApi.Medicine.Get(medicineJson.name).StatusCode,
                    "Status code should be 404 (Not Found)");
}
```

Figure 2.45 Test for deleting a medicine

The tests that modify data have a “Revert” phase to take an action to leave the database as it was at the beginning of the test. Doing that we ensure that tests are fully independent from each other and we do not need to run them in any specific order.

When passing the json body to create or update a medicine we are using Anonymous Types, it is a feature in C# that encapsulates a set of read-only properties into a single object without having to explicitly define a type first. The type name is generated by the compiler and is not available at the source code level. The type of each property is inferred by the compiler. This allows us to develop tests in a more flexible way as we do not need to create a model for each json body required by the API methods.

CHAPTER 3. CONTINUOUS INTEGRATION

In this chapter we will introduce another important practice of Agile Testing. We will see the benefits on working on a Continuous Integration [39] workflow and we will also get practical and build a Continuous Integration proof of concept, using the automated functional tests created in the previous chapter and other tools.

3.1 Definition

Continuous Integration (CI) is a development practice that requires developers to integrate their code into a shared repository several times a day. Each commit is then verified by an automated build, this allows the development team to detect problems early. By doing that integration regularly, we can detect errors quickly, and locate them more easily. There is significantly less back-tracking to discover where things went wrong, so team can spend more time building features.

CI has evolved since its conception. Originally, a daily build was the standard. Now, the usual rule is for each team member to submit work on a daily (or more frequent) basis and for a build to be conducted with each significant change. CI doesn't get rid of defects, but it does make them dramatically easier to find and remove. In this respect it's rather like self-testing code. If we introduce a bug and detect it quickly it's far easier to get rid of. Since we have only changed a small bit of the system, we do not have far to look and it's fresh in our memory.

There are several development practices that embrace CI, like maintaining a single source repository, automating the build (including tests), keeping the build fast or making the build state visible to the team. All this practices require a dedicated machine that is usually known as the Continuous Integration server.

3.2 Continuous Integration Server

In order to put CI in practice, developers should commit their code to a source code repository regularly, a Continuous Integration server will monitor the repository and check out changes when they occur. This machine will build the system and run the automated tests (unit, integration, functional) and inform the team about a successful or failed build.

There are a few CI server solutions in the market, both open-source and commercial. We will use Jenkins, which has stablished as the open-source standard one. It is very flexible and highly configurable so it can fit almost any programming language or technology via an extensive collection of plugins. It is used by well-known companies like Facebook, NASA or Netflix [40] and there is a big user base an support community.

Jenkins is an application that monitors executions of repeated jobs, such as building a software project or jobs run by a certain schedule. Among those things, Jenkins focuses on building software projects continuously, making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build.

At Alteraid a single repository is kept for the different products that they develop. The development team uses GitLab as the hosting solution for the source code repository. A new “Alteraid.Automation” repository has been created in GitLab to host the source code from previous chapter.

There is also a Continuous Integration server working for the Aaaida platform, where the Alteraid team integrates the code for the API. For the purpose of this project and to do not interfere with the daily work at Alteraid we will create an alternative CI server that will build and run the Web app and API functional tests developed in this project.

3.2.1 Initial Setup

Jenkins is multiplatform so can be used in Linux, Windows or Mac machines, as the development environment of choice for the WebApp and API functional tests is .NET, we will use a Windows 8.1 virtual machine, provided by the Alteraid team, with Visual Studio installed, in order to build and run the tests.

Using the specific Windows installer for Jenkins it will configure Jenkins as a Windows service and the Jenkins web portal will be reachable at “*http://1.2.3.4:8080*”, where 1.2.3.4 is the IP of the machine.

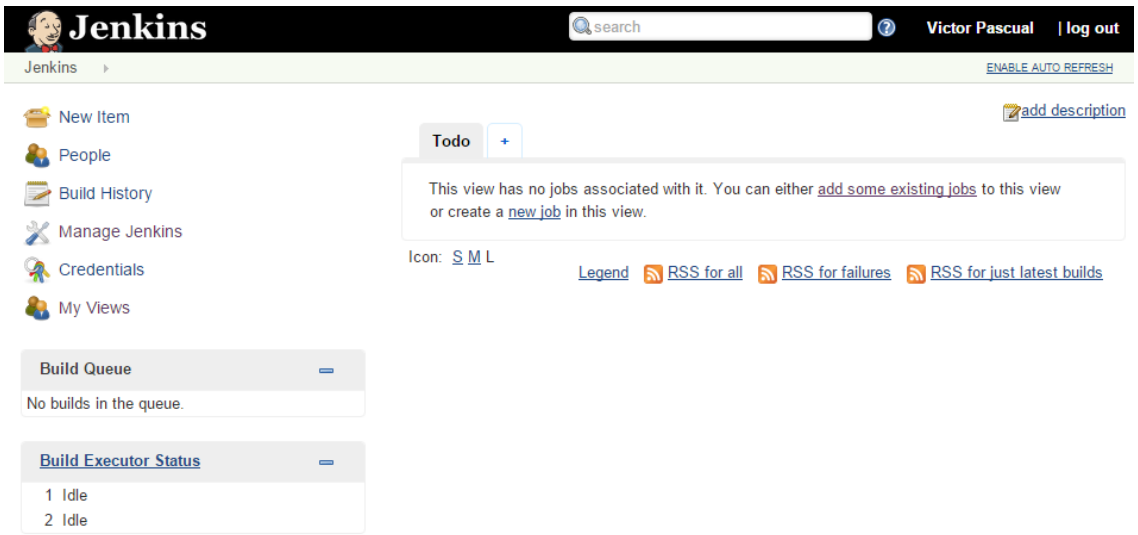

Figure 3.1 Jenkins main page

A fresh installation of Jenkins comes with a minimum setup to only build Java projects, as we are working in .NET, we will install some plugins in order to enable Jenkins to work with this technology. All the Jenkins plugins are available and can be installed from the “Manage Jenkins > Manage Plugins” section.

These are the plugins we will use in order to build and run the functional tests:

- **MSBuild Plugin**: Enables Jenkins to build .NET projects. This plugin uses MSBuild tool, which comes with a Visual Studio installation [41].

- **MSTestRunner plugin**: It uses MSTest command-line tool to run tests from Visual Studio test projects. MSTest it is also available on a Visual Studio installation [42].

- **MSTest plugin**: This plugin converts the MSTest result files into a format the Jenkins can understand. This will allow us to present results in Jenkins after a test run [43].

After installing the plugins there is still some initial setup we need to do before creating the jobs. First we need to specify were are located in disk the tools that we will use to build and run the functional tests, MSBuild and MSTest. We can specify these paths in “Manage Jenkins > Configure System”:

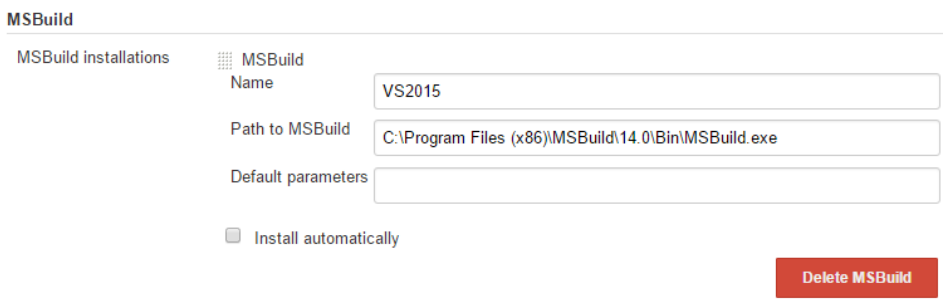

Figure 3.2 MSBuild configuration for Jenkins

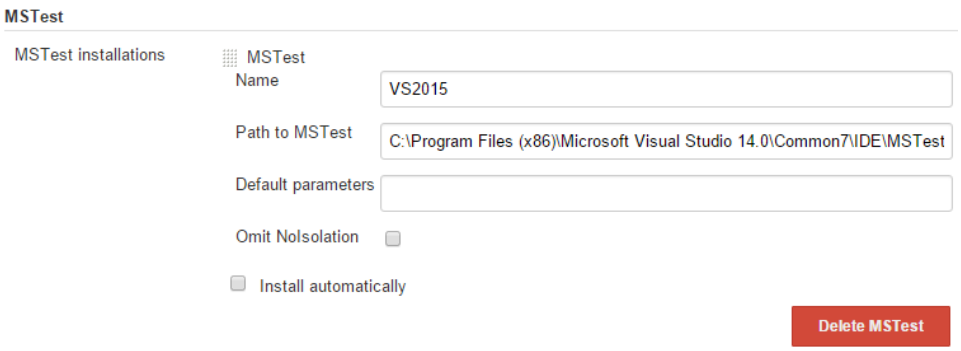

Figure 3.3 MSTest configuration for Jenkins

There is one more plugin we need to install in order to get the code from the GitLab repository. GitLab is based on Git so we will just install the “Git Plugin” for Jenkins [44] so we can get the code from the repository for each job. Once installed we also need to tell Jenkins where it can find the Git executable in the CI server machine:

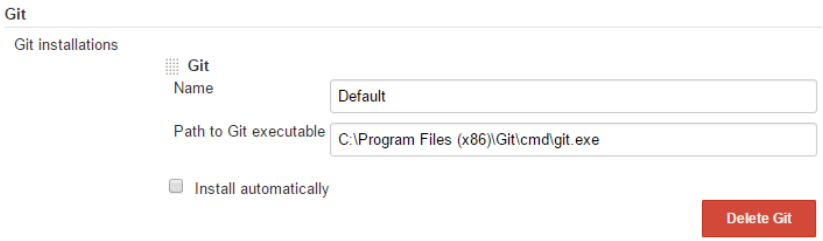

Figure 3.4 Git configuration for Jenkins

In order to access the source code from the jobs, we will need to authenticate to the GitLab server. We can create credentials in Jenkins that we can then use in the jobs, to do that we will go to “Manage Jenkins > Manage Credentials” and create one for the GitLab server:

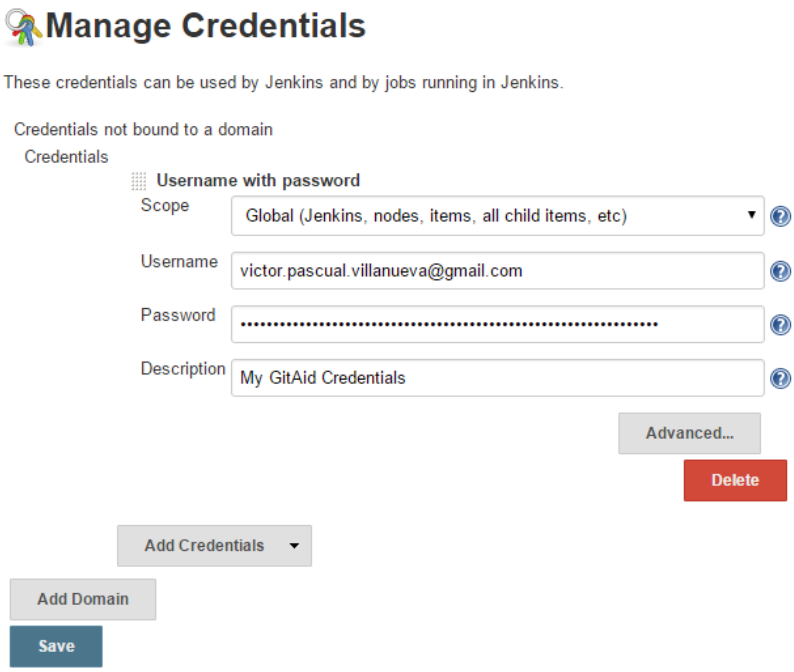

Figure 3.5 Credentials for GitLab server

With all this setup done we are now ready to create the Jenkins jobs that will build and execute the functional tests created in Chapter 2.

3.2.2 Web App Functional Tests job

We can create a new job from the Jenkins portal, this job will get the code from the source code repository, build the Visual Studio solution and run the Web App functional tests.

From Jenkins portal we can select “New Item” from left menu, set a job name and select “Free-style project”:

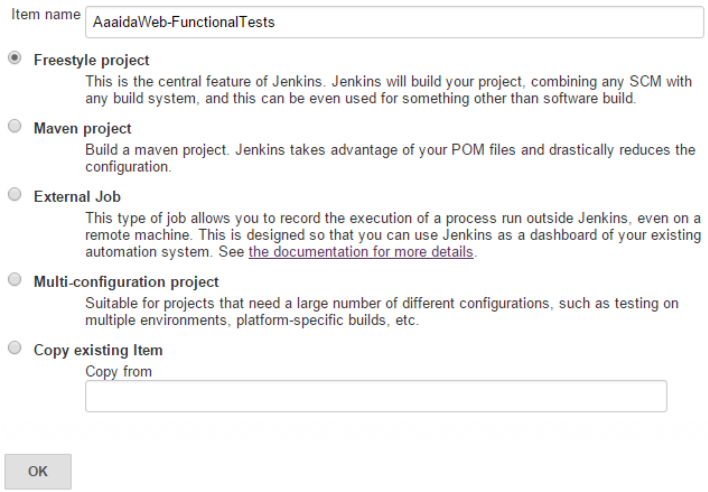

Figure 3.6 Creating new job in Jenkins

First section we need to setup in the job configuration is the “Source Code Management” one, choosing Git as the repository technology and specifying the repository url and credentials in order to access the code:

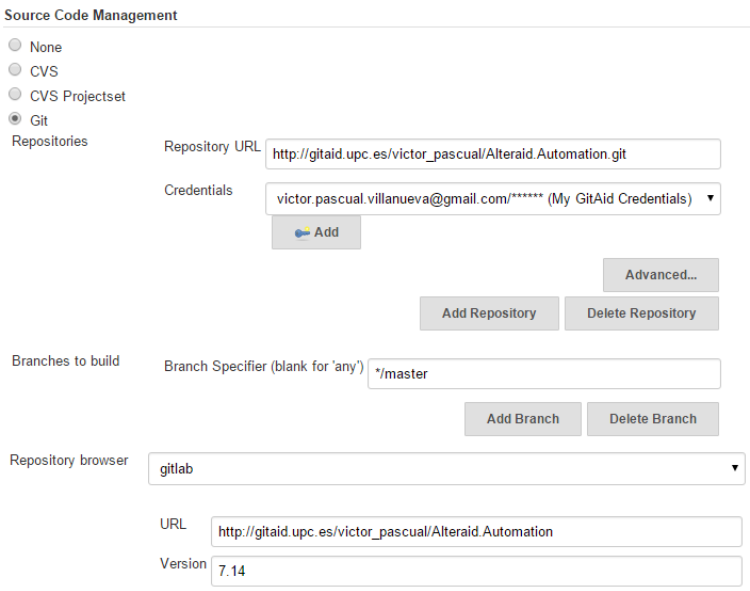

Figure 3.7 Source code configuration in Jenkins

Once the job has downloaded the code from the source repository, it will need to build the “Alteraid.Automation.AaaidaWebTests” project. To do that we will add a build step to “Build a Visual Studio project or solution using MSBuild, specifying the “.sln” solution file that we need to build:

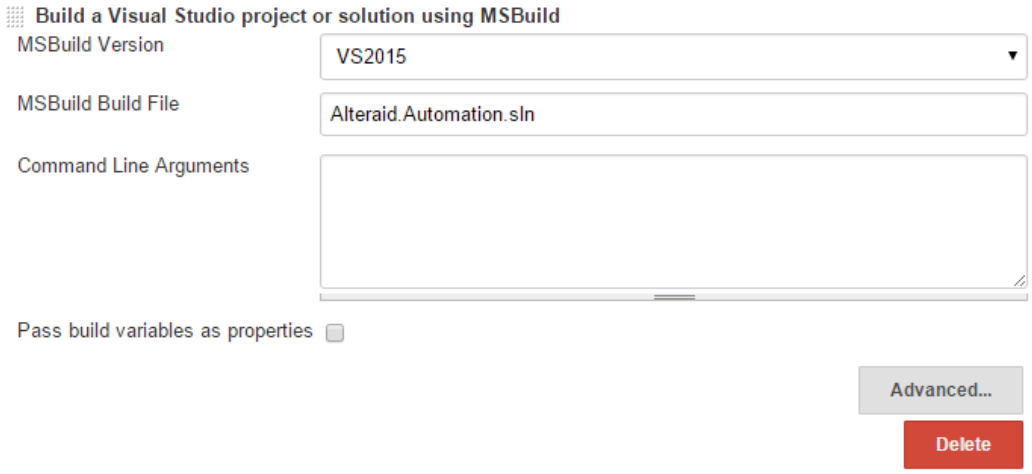

Figure 3.8 Step to build the Solution in Jenkins

Now we are ready to execute the tests, we will add another build step to “Run tests with MSTest”, this will run the tests in the specified .dll using MSTest. We will specify as well the results “.trx” file name so we can later reference to it to show the test results in Jenkins:

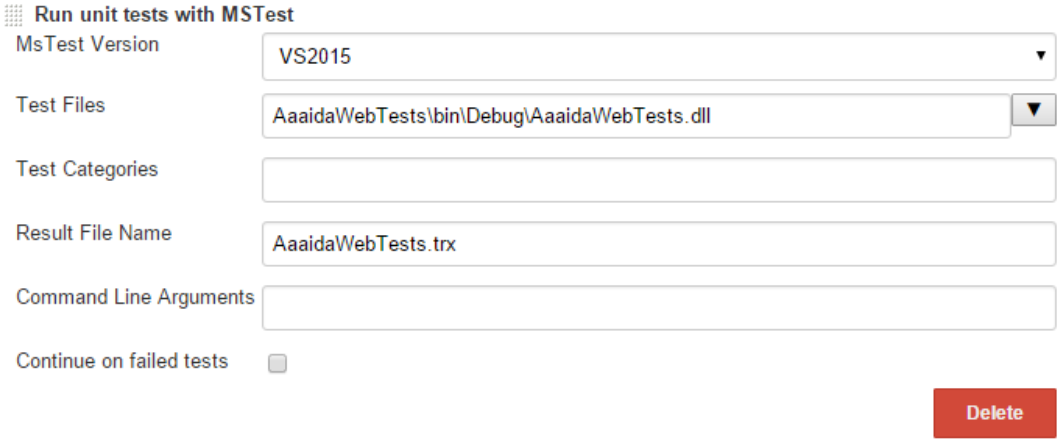

Figure 3.9 Step to run the tests in Jenkins

Jenkins will now get the code from the source repository, build the solution and run the Web App functional tests. We can now decide which files do we want to archive for history purposes. It is always useful to keep the “.trx” test results file and additionally it will be very useful to keep the screenshots that the automation framework, created in previous chapter, generates for each failed test. These

screenshots will be in a “Screenshots” folder in the build workspace. We will use the “Archive the artifacts” post-build action to keep all these files. Notice the pattern used to select every file inside the Screenshots folder:

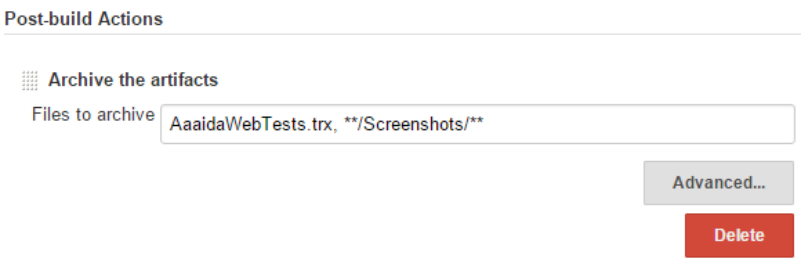

Figure 3.10 Step to archive test build artifacts in Jenkins

To get a user-friendly report of the test results directly in Jenkins we need to add another post-build action that publishes the “.trx” results file in Jenkins using the MSTest plugin that we have installed earlier on the Chapter:

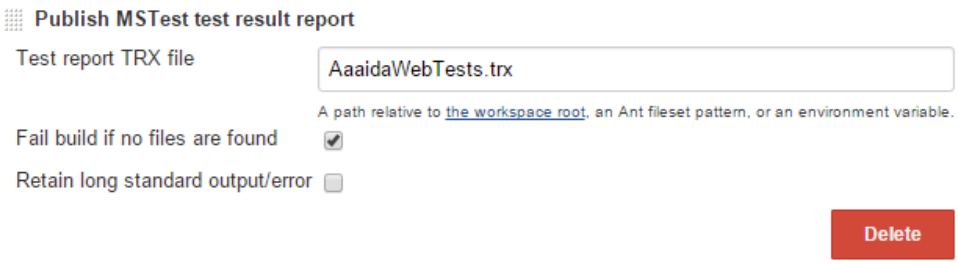

Figure 3.11 Step to publish test results in Jenkins

Also we can make use of the “Image Gallery” Jenkins plugin [45] to show a gallery of the screenshots for failed tests, if any. All this additional information will be very useful when debugging or troubleshooting failed tests in Jenkins:

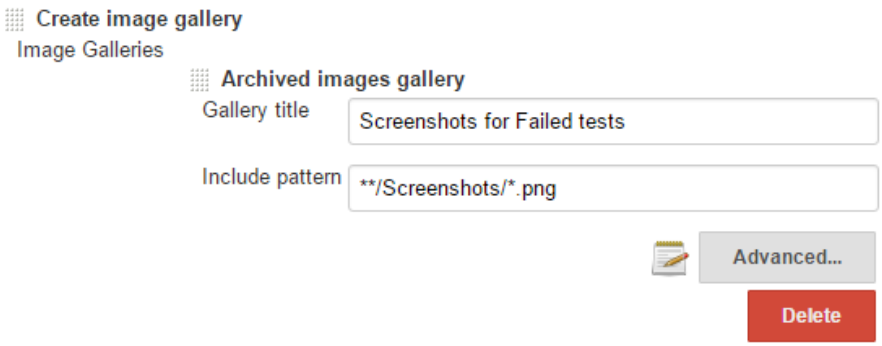

Figure 3.12 Step to publish Screenshots gallery in Jenkins

Job is fully configured now, we can go back to Jenkins portal main page to execute the job. Once the execution is finished, it should have created a test results page. The following figure shows a test run with all Web App functional tests passing:

Test Result : LoginTests

0 failures (±0)

2 tests (±0)
Took 21 sec.
add description

All Tests

Test name	Duration	Status
CanLoginWithValidCredentials	12 sec	Passed
CanNotLoginWithWrongPassword	9,6 sec	Passed

Figure 3.13 LoginTests test result in Jenkins

The following figures show an example on how the test results report will look like if one of the tests fail. Notice that it includes the screenshot of the failing test:

Test Result

1 failures (-1)

2 tests (±0)
Took 32 sec.
add description

All Failed Tests

Test Name Duration Age

AaaidaWebTests.LoginTests.CanNotLoginWithWrongPassword

Error Details

```
Assert.AreEqual failed. Expected:<User or password wrongXXX>. Actual:<User or password wrong>. Error message
incorrect
```

Stack Trace

```
   en AaaidaWebTests.LoginTests.CanNotLoginWithWrongPassword() en C:\Program Files
(x86)\Jenkins\workspace\AaaidaWeb-FunctionalTests\AaaidaWebTests\LoginTests.cs:línea 18
```

Standard Output 25 sec 10

```
2015/09/14 22:41:50.325 | INFO | Browser | Initalizing Chrome browser
2015/09/14 22:42:08.134 | INFO | Browser | Navigating to 'http://old.aaaida.com/aaaida/?lang=en'
2015/09/14 22:42:12.507 | INFO | LoginPage | Waiting for page to load...
2015/09/14 22:42:12.588 | INFO | LoginPage | Populating 'victor.testing@mailinator.com' in email field
2015/09/14 22:42:12.855 | INFO | LoginPage | Populating 'wrongPassword' in password field
2015/09/14 22:42:13.143 | INFO | LoginPage | Clicking 'Sign in' button
2015/09/14 22:42:14.097 | INFO | LoginPage | Waiting for page to load...
2015/09/14 22:42:14.269 | INFO | LoginPage | Waiting for page to load...
2015/09/14 22:42:14.477 | INFO | Browser | Taking Screenshot
2015/09/14 22:42:14.941 | INFO | Browser | Closing driver
```

Figure 3.14 Failing test result in Jenkins

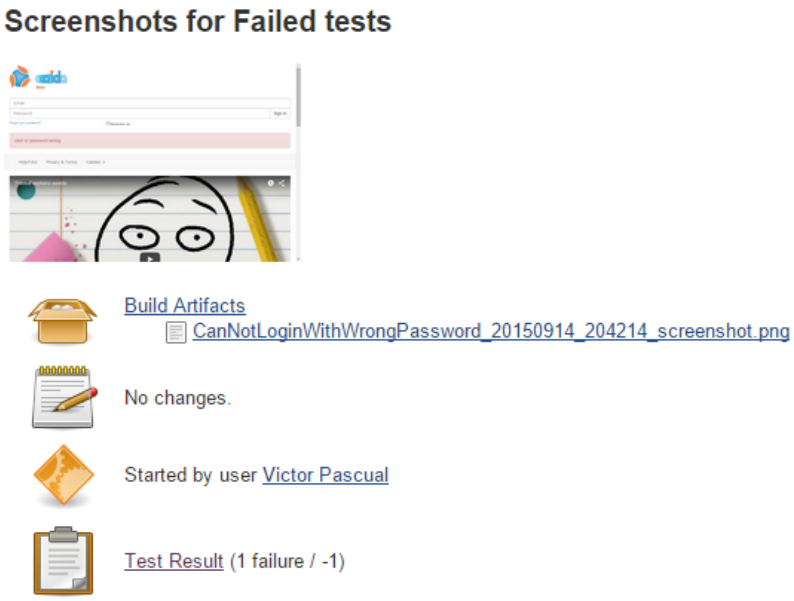

Figure 3.15 Screenshot for failing test in Jenkins

From the previous screenshots we can see how much information do we get when a test fails: we get the error details, stack trace and a detailed list of steps is the standard output. We also get an screenshot when the test failed. That should be enough to troubleshoot failing tests quicker.

3.2.3 API Functional Tests job

The Jenkins job to run the API Functional tests will be very similar to the one created in the previous section, so we'll just show use an special functionality in Jenkins to create a job copying from an existing one:

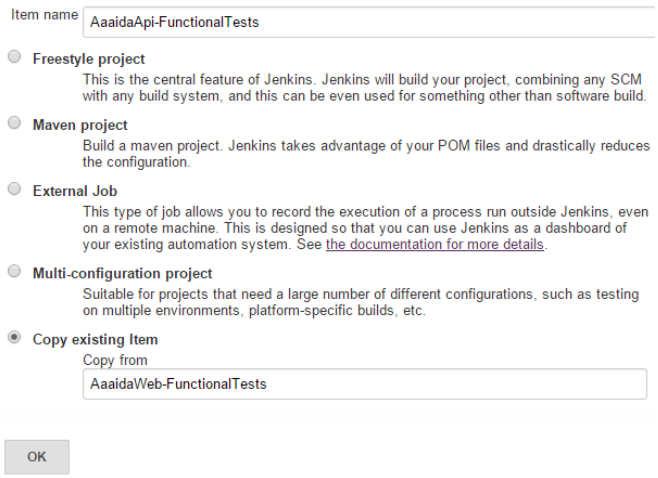

Figure 3.16 Cloning existing job in Jenkins

Now we need to change the job configuration to reference the .dll with the API functional tests instead of the “AaaidaWebTests.dll” one. A part from that, we can remove the section to publish the screenshot gallery as there will be no screenshots for API tests.

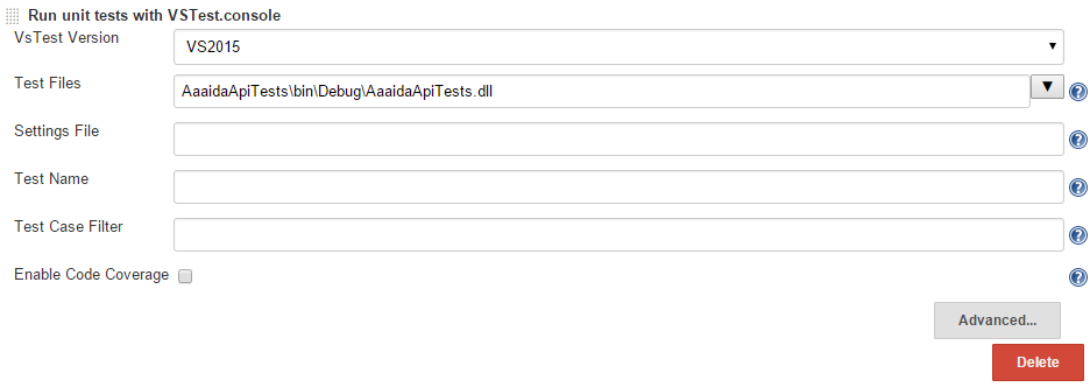

Figure 3.17 Step to run API functional tests in Jenkins

Job should run successfully now and present a report with the test results. Notice that we also get very valuable information on how long took each test to run, this could potentially be used as a performance metric for the API:

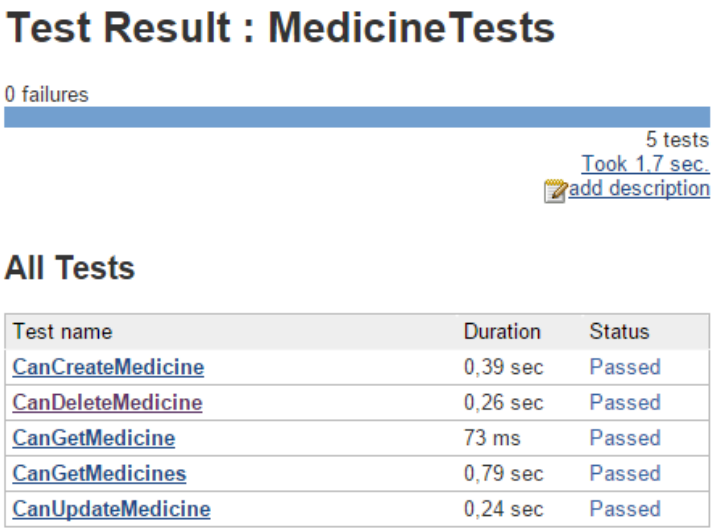

Figure 3.18 MedicineTests test result in Jenkins

3.3 Testing in Continuous Integration

Here we will describe some practices to integrate testing into the Continuous Integration workflow at Alteraid, more specifically for the Aaaida platform. Ideally, every time a developer commits some code, a new job will be automatically triggered in Jenkins to build and run unit tests for the Aaaida Web and Aaaida API, the two main components of the Aaaida platform.

If the build or any unit test fails, the developer who committed the code should take action and fix the build. This task should be highest priority as a broken build will block the rest of the team. A developer is responsible for his code from commit until the build and all tests pass. That's why it is a very recommended practice for the developer to compile the code and run all the unit tests locally before checking in the code.

When the build passes, a second Jenkins job should deploy that code to a test environment, which will be more stable as all unit tests will have passed. Now the tester can use this stable test environment to perform the tests of the related stories. We can also trigger automatically the WebApp and API functional tests that can run against this test environment.

To trigger automatically the Jenkins jobs that execute the Web App and API functional tests we can use a feature in GitLab called "Web Hooks" [46]. In GitLab we can create a "Web Hook" that will be executed when a certain event happens. In our situation we can use it to run the functional tests when some code is checked in against the "Alteraid.Automation" repository:

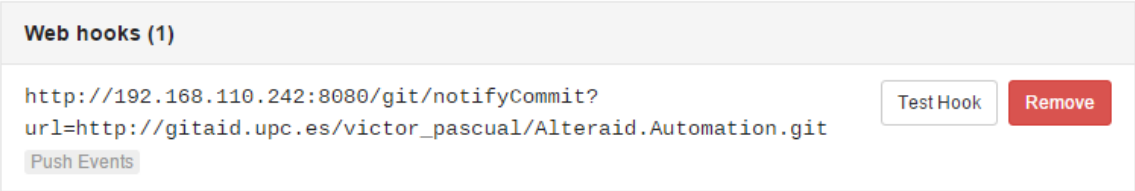

Figure 3.19 Web Hook for Push event in GitLab

This will scan all the jobs that are configured to check out the specified URL, and if they are also configured with polling, it'll immediately trigger the polling (and if that finds a change worth a build, a build will be triggered in turn).

With that configuration we ensure that functional tests still pass if any change is submitted to the test automation framework. If this setup is ever integrated into the existing CI server at Alteraid, the functional test jobs will also be triggered when any change is commited to the main Aaaida API and Aaaida Web repositories, ensuring that every commit is properly covered by automated tests.

CONCLUSIONS

Main objectives in this project have been accomplished: defining a testing methodology for a real world development team and giving an implementation example of some of this methodology practices, Test Automation and Continuous Integration.

Having been able to work with the development team at Alteraid for a few days has been very positive in order to get to know how they work and how their development practices and processes could be improved by implementing Agile Testing. This methodology has some guidelines defined but understanding the context of the company has helped to define a more concrete workflow for the development team.

When introducing these new practices to the Alteraid team, they agreed that it will improve their development process but it will also slow them down. It has to be clear that implementing this kind of practices is always going to slow down a development team in the short-term, but in the mid-term it can help to deliver a higher quality product at a more predictable pace.

End-to-end functional tests can be really slow compared with unit tests, especially the ones that interact with a real browser. This has to be taken into account when implementing them in a real world development cycle as it can be counterproductive to have a really slow suite of tests. This could be mitigated by running tests in parallel in different machines, so test run times get shortened. API functional tests, in the other hand, are still reasonably fast so if a feature can be tested with API tests instead of browser tests, we should favour for the API ones.

Another challenge observed while working on this project is that functional tests can be brittle. This can be caused by random reasons like a certain web page rendering slower than normal or an API call timing out. We can try to mitigate this brittleness but we will never be able to prevent all random failures. That is why from the learnings of this project, I would recommend to always favour for tests at lower levels where test runs are more robust.

While working on the test automation framework I realized that it could be used for other purposes other than functional test automation. A possible continuation of this project could be extending the framework to create a suite of performance tests for the Aaaida API. It could also be used to automate certain actions in the Aaaida web portal, like creating users, that could help to setup data when testing features manually.

Another possible continuation could be targeting some of the mobile apps at Alteraid and extending the framework in order to be able to test them with mobile test automation tools like Appium [47] or Calabash [48]. Mobile testing is still in early stages and could be a great topic to investigate in a future project.

BIBLIOGRAPHY

[1] History of the Agile Manifesto:
- http://www.agilemanifesto.org/history.html

[2] Alteraid news at EETAC-UPC website:
- https://eetac.upc.edu/ca/category/etiquetes/alteraid

[3] What is an Electronic Health Record?
- http://www.healthit.gov/providers-professionals/faqs/what-electronic-health-record-ehr

[4] ISO 13485 Quality Management System for Medical Devices:
- http://www.tuv-sud.com/industry/healthcare-medical-device/quality-management-quality-control-for-medical-devices/iso-13485-quality-management-system-for-medical-devices

[5] Information about Aaaida platform:
- http://www.alteraid.com/en/#page-3

[6] Information about Virginia app:
- http://support.aaaida.com/customer/en/portal/articles/1685802-about-virginia

[7] Information about Adheptor app:
- http://support.aaaida.com/customer/en/portal/articles/1613602-adheptor-help

[8] Information about Dermapac/Dermadoc apps:
- http://www.slideshare.net/alteraid/v3-dermapac-i-dermadoc-a-aaaida-presentacioappssalut20comb

[9] The Joel Test: 12 Steps to Better Code by Joel Spolsky:
- http://www.joelonsoftware.com/articles/fog0000000043.html

[10] Information about GitLab:
- https://gitlab.com/gitlab-org/gitlab-ce/tree/master#README

[11] Information about Subversion:
- https://subversion.apache.org/docs/

[12] Information about Jenkins CI:
- https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins

[13] Information about Pivotal Tracker:
- http://www.pivotaltracker.com/why-tracker/how-it-works

[14] Information about Mantis:
- https://www.mantisbt.org/documentation.php

[15] Information about Trello:
- https://trello.com/tour

[16] Five W's – Wikipedia article:
- https://en.wikipedia.org/wiki/Five_Ws

[17] The traditional Waterfall approach:
- http://www.umsl.edu/~hugheyd/is6840/waterfall.html

[18] Information about Test-Driven Development (TDD):
- http://www.methodsandtools.com/archive/archive.php?id=20

[19] Information about Behaviour-Driven Development (BDD):
- http://guide.agilealliance.org/guide/bdd.html

[20] Information about Agile Testing Quadrants:
- http://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/

[21] Crispin, L., Gregory, J. "*Agile Testing: A Practical Guide for Testers and Agile Teams*", Pearson Education, Indiana, 2009.

[22] Information about Amazon EC2:
- https://aws.amazon.com/ec2/details/

[23] Information about Microsoft Azure:
- https://azure.microsoft.com/en-us/get-started/

[24] Information about Google Compute Engine:
- https://cloud.google.com/compute/docs/

[25] Information about Visual Studio 2015 (Community Edition):
- https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx

[26] Blog post from Martin Fowler about YAGNI:
- http://martinfowler.com/bliki/Yagni.html

[27] MSTest – Wiki article:
- https://en.wikipedia.org/wiki/MSTest

[28] LifeHacker.com article about Mindmup:
- http://lifehacker.com/mindmup-maps-your-brain-in-the-browser-614853279

[29] Information about Selenium WebDriver:
- http://www.seleniumhq.org/projects/webdriver/

[30] Information about NuGet:
- http://docs.nuget.org/consume/overview

[31] How to use Explicit Waits in Selenium:
- http://www.seleniumhq.org/docs/04_webdriver_advanced.jsp

[32] ExpectedConditions class implementation:
- https://github.com/SeleniumHQ/selenium/blob/master/dotnet/src/support/UI/ExpectedConditions.cs

[33] Information about NLog:
- https://github.com/nlog/nlog/wiki

[34] NLog logging levels defined:
- https://github.com/nlog/nlog/wiki/Log-levels

[35] Information about PageObject pattern:
- https://code.google.com/p/selenium/wiki/PageObjects

[36] Information about Arrange, Act, Assert pattern:
- http://c2.com/cgi/wiki?ArrangeActAssert

[37] Information about PageFactory:
- https://code.google.com/p/selenium/wiki/PageFactory

[38] Information about RestSharp:
- https://github.com/restsharp/RestSharp/wiki

[39] Information about Continuous Integration:
- https://www.thoughtworks.com/continuous-integration

[40] Companies using Jenkins:
- https://wiki.jenkins-ci.org/pages/viewpage.action?pageId=58001258

[41] Information about MSBuild plugin for Jenkins:
- https://wiki.jenkins-ci.org/display/JENKINS/MSBuild+Plugin

[42] Information about MSTestRunner plugin for Jenkins:
- https://wiki.jenkins-ci.org/display/JENKINS/MSTestRunner+Plugin

[43] Information about MSTest plugin for Jenkins:
- https://wiki.jenkins-ci.org/display/JENKINS/MSTest+Plugin

[44] Information about Git plugin for Jenkins:
- https://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin

[45] Information about Image Gallery plugin for Jenkins:
- https://wiki.jenkins-ci.org/display/JENKINS/Image+Gallery+Plugin

[46] Information about “Web Hooks” in GitLab:
- https://gitlab.com/gitlab-org/gitlab-ce/blob/master/doc/web_hooks/web_hooks.md

[47] Information about Appium:
- http://appium.io/slate/en/master

[48] Information about Calabash:
- http://calaba.sh/