"Testing is an infinite process of comparing the invisible to the ambiguous in order to avoid the unthinkable happening to the anonymous." – James Bach

"No amount of testing can prove a software right, a single test can prove a software wrong." – Amir Ghahrai

"More than the act of testing, the act of designing tests is one of the best bug preventers known." – Boris Beizer

"The bitterness of poor quality remains long after the sweetness of low price is forgotten" – Benjamin Franklin

Preface

Currently we're completing an Engineer degree in Software development at the National School of Applied Sciences in Kenitra (ENSA - Kenitra). We're on our 5th year and therefore we have to conduct a five months internship.

Industrial training program offered by National School of Applied Sciences, University of Ibn Tofail, is an opportunity for undergraduate students to apply their theoretical knowledge gained during the university academic program into real world industrial based application development and experience professional software development process. The objectives of such programs are to enhance participants' skills, and to enrich their industrial knowledge by keeping them updated with the latest technologies. This opportunity has extremely helped us to expose into an environment where we could think of as a software engineer instead of just as a software developer.

After completing successfully the nine semesters of ENSA's program, we got the opportunity to carry out our industrial training in Media Mobility Company - a company specialized in creating mobile solutions, based in Casablanca, as mobile apps testers' trainees for 24 weeks starting from February the 1st, 2015.

In this internship report we will describe our experiences during our internship period. The internship report contains an overview of the internship company, and the activities, tasks and projects that we have worked on during our internship. Writing this report, we also will describe and reflect our learning objects and personal goals that we have set during our internship period.

Dedication

I would like to dedicate this capstone project to my family and many friends.

To my loving parents, there is no doubt in my mind that without their continued support and counsel I could not have completed this process.

To my brothers and sister, Mourad, Taoufik and Nawal, who taught me that the best kind of knowledge to have is that which is learned for its own sake. I will always appreciate all what they have done for me.

To my best friends, Nada, Hajar, Kaoutar, Zineb and Alae, who have supported me throughout the process.

Manal Jazouli

Dedication

To my Beloved Parents, for all their love, attention, sacrifices and endless patience which has made it possible for me to make it up to this point. They are heaven on earth for me.

To my dear friends, Younes, Khalil, Abdallah, Hamza, who bestowed me with the courage, the commitment and the awareness to follow the best possible route.

Soufiane El Aissaoui

Acknowledgements

Our internship report wouldn't have been possible without contribution of few people, so we're taking this opportunity to express our profound gratitude and deep regards to all of them.

At first we desire to express our deepest sense of gratitude of almighty Allah.

With profound regard we gratefully acknowledge our respected professor Mrs. Oumaira Ilhame, professor of higher education at ENSA Kenitra, Ibn Tofail University, for her generous help, her exemplary guidance, monitoring and constant encouragement in the process of our internship.

Next we would like to show our gratitude towards Mr. Mehdi Alaoui and Mr Khalid Tabyaoui, our onsite supervisors, co-founders of Media Mobility company, first, for granting us the opportunity to make our internship program in this organization, and then for their cordial support, valuable information and guidance, which helped us in completing this task through various stages.

We cannot thank enough to all the personnel of Media Mobility, especially Mr Nabil Sossey Alaoui, Mr Choaib Kidai and Mr Oussama Zaid. They have explained everything we asked for in details. Throughout time they were never impatient. They did not allow us to feel uncomfortable for even a single moment. We are really grateful to all for their supportive and friendly behavior.

Last and not least, we thank our families and friends for their constant encouragement without which this assignment wouldn't have been possible, and we dedicate this work to every person who has ever taught us something valuable for life during the long road to become computer engineers.

Abstract

Nowadays, the world have merged into an era where everybody is talking about their next great mobile app, and in a market with fierce competition, companies need to have the best reputation, and satisfaction rates with their customers. One of the ways that have been proven to guarantee that, is quality assurance of their products, and that means that their apps should go through a quality check phase or more clearly, a testing phase before releasing their products to the public. We can define app testing as the mean that helps to uncover structure, design and coding errors in the app, or any unexpected or unwanted behavior, before allowing it to reach the consumers. It is used to measure the correctness, completeness and quality of apps against a specification.

Moreover, to implement the testing in their industrial process, companies have to hire a lot of testers, and have a standard that allows them to communicate easily with developers, and produce testable code, which could become expensive and time consuming, especially in big companies with multiple complex projects, where the code gets more sophisticated as new features get added, and as a result, testing get harder and more time consuming. To solve this problem, auto-testing got introduced, which means, testers can make bot-like scripts, that handle testing the routine scenarios, leaving them time for more productive tasks, and as a result, this enhances the quality assurance process, reduces costs of testing, and optimizes time consumption during testing.

In our capstone project paper, we try to discuss the importance of the testing phase, we analyze the benefits of going farther to introducing an auto-testing system, and last but not least, we present our suggested solution, that we worked on during our internship, to help the host company assure the quality of their mobile apps, by implementing an auto-testing system.

Keywords: Quality, Auto-Testing, Mobile apps testing.

Résumé

De nos jours, le monde est parti dans une ère ou toutes les entreprises parlent de leur prochaine immense application mobile, et dans un marché à concurrence féroce, ces entreprises ont besoin d'avoir la meilleure réputation et le plus grand taux de satisfaction de leurs clients. L'un des moyens qui a été prouvé de garantir ceci est l'assurance de la qualité des produits, et ceci veut dire que les applications mobiles devront passer par une phase de vérification de qualité, ou encore, une phase de test avant la sortie du produit au public.

On peut définir les tests des applications mobiles comme étant un moyen qui aide à découvrir les erreurs de la structure, du design ou du code de l'application mobile, ainsi que tout comportement inattendu ou indésirable, avant de la permettre d'atteindre le client. Ce moyen est utilisé pour mesurer le bon fonctionnement, la complétude et la qualité de l'application comparé aux spécifications.

En outre, pour implémenter la phase des tests dans le processus industriel, les entreprises devront embaucher plusieurs testeurs, et créer un standard qui leur permettra de communiquer facilement avec les développeurs, et développer un code testable, ce qui pourra s'avérer très cher et trop long, surtout dans les grandes entreprises qui gèrent plusieurs projets complexes, ou le code se procure de plus en plus sophistiqué autant que de nouvelles fonctionnalités sont rajoutées, et par conséquent, les tests manuels deviennent de plus en plus compliqués. Pour résoudre ce problème, on introduit l'auto-testing, c'est-à-dire les testeurs peuvent développer des scripts qui peuvent traiter les scénarios des test routines, et donc libérer le temps des testeurs afin de réaliser des tâches plus productives, ce qui améliore le processus de l'assurance de la qualité, réduit les couts des test, et optimise le temps consommé dans la phase des tests.

Dans notre rapport de projet de fin d'études, nous discutons l'importance de la phase des tests, nous analysons les avantages d'introduire un système d'auto-testing, et finalement, nous présentons notre solution proposée, sur laquelle nous avons travaillé durant notre période de stage, afin d'aider l'entreprise d'accueil à assurer la qualité de ses applications mobiles, en implémentant un système d'auto-testing.

Mots clés: Qualité, Auto-Testing, tests des applications mobiles.

Table of Contents

Table of figures

List of tables

Introduction

Just four years ago, smartphones were viewed as expensive toys for geeks and Apple fan boys. No longer; smartphones are nowadays most widely used in every aspect of our life including personal, social and professional as well. Smartphones have entered the mainstream in developed markets, and are taking a growing proportion of device sales in more cost sensitive markets around the globe.

This ubiquity and popularity of smartphones among end users, and the growth of mobile workforce and the improving capabilities of mobile devices have increasingly drawn enterprises' attention to develop applications that run on these devices.

But developing mobile apps don't always generate good profit. There are obvious parallels between the gold rush of 150 years ago and the era of the mobile app. Just as finding a gold vein took an enormous amount of luck in addition to some skill, producing a hit mobile app is like catching lightning in a bottle. So, while it's great to build the next Instagram or WhatsApp, we see a more lucrative investment thesis in those segments that help enable mobile application development.

In this context, Media Mobility Company, a company for making mobile apps, did not only seek the gold, but decided to build the cave that holds plenty of opportunities for any developer to make any app they want. Media Mobility's team has been working for 3 years now on the ScreenDy project which is a platform for making native apps on iOS and Android, using only Drag & Drop gestures, and without a single line of code.

But as with any new domain, mobile application development has its own set of new challenges, especially challenges with guaranteeing the quality of mobile applications. Developers currently treat the mobile app for each platform separately and manually check that the functionality is preserved across multiple platforms and to test that the requirements and the specifications needed are met.

We had the chance to contribute to the project by documenting some of their plugins while working on an integration project using their platform, but our main task didn't remain in that. Our main task was to introduce a quality assurance phase to their industrial process, and develop an auto-testing system to assure the stability of the platform during different stages of development, and also, to provide the end user with a simple interface to test their app.

Problematic

Testing is an essential part of the software development lifecycle. However, it can cost a lot of time and money to perform. For mobile applications, this problem is further exacerbated by the need to develop apps in a short time-span and for multiple platforms.

In our case, the company developers make frequent changes, upgrades and updates to the project's kernel, whether it's adding or removing lines and/or blocks of code, which increases the risk of affecting the performance or even breaking parts of the kernel, a thing that demands constant testing of the whole project, from stability to view rendering results.

Our mission in the Media Mobility company was to propose a solution to assure a continuous quality during the development of their main project's kernel that is ScreenDy platform.

To get to this we started by creating a full functional App using only ScreenDy, in order to get used to the functionalities offered by the platform and get familiar with its internal working. To put the knowledge gained in this phase into perspective, we documented the plugins used in the App and others, and wrote a tutorial for future users.

We moved then to the auto-testing task, in which we put ourselves in users' and developers' place to extract test cases and test scenarios used by real users, and then work on the ones we can automate. We combined the open source testing platforms found precedently to generate a generic test for a random App, or a specific test for a specific App.

Finally to perfect the auto-testing task, we created a RESTful API, so that we can integrate it seamlessly with the platform, to provide the end-user with an interface to test their Screendy made apps.

Executive Summary

Our internship work was done under the supervision of Mr Khalid Tabyaoui and Mr Mehdi Alaoui externally, and Mrs Ilham Oumaira internally, in a company called Media Mobility, specialized in mobile apps, which is currently working on a mobile app development platform that requires no coding skills, named ScreenDy.

In this report, we're going to present the host company and their project, then we're going to talk briefly about the rise of mobile apps, which will introduce the main problem that our work tried to solve inside Media Mobility, that is quality assurance for mobile apps. We're going to explain how important testing is for any software company, especially mobile apps companies, and then we're going to explain how our work attempted to optimize the testing phase inside the company, by introducing and implementing an auto-testing system, that would handle the routine test scenarios, which will give more time for developers to focus on their tasks.

Development methodology and project management

To conduct our project smoothly, and have a flexible development, we used one of the famous agile development methodologies called scrum, which is based on increments, we can define an increment as a small part of the project. We reported our progress every day to our supervisors, and in the beginning of each week, we held a meeting, to discuss the tasks and parts we accomplished in the previous week, and tell the team about our planning and tasks for the ongoing week. This way, everyone can know what we have done, and the team can help us whenever we're stuck in some challenge, and help us identify the source of the problem. As a result the project was running with no interruptions and smoothly.

Documentation and integration project

Before we can start on our main task, we had to make a whole app using only ScreenDy's drag-and-drop platform, so we can get familiarized with its structure, and understand its architecture, this will enable us later to design and write good test scenarios.

While making the app, we were also writing documentation and a tutorial, to describe different steps of how to make an app using ScreenDy, and how to use its various plugins in the platforms, while describing each plugin's parameters.

After that we tried the app on both Android and iOS, and it worked as expected on both platforms.

Background knowledge:

The use of mobile apps have known an increased percentage nowadays, and the competition over who's going to have the largest share of the market made companies think of ways to attract consumers attention, and that's why quality is essential in this area, which in itself proves the need of the testing phase before any app could reach the public. However, testing can be time consuming and expensive, especially if the testers or developers have to repeat the same routine scenario over and over and every time they add and/or change something, from here comes the need for an auto-testing system that would handle those routine tests, and this has been our main task.

After analyzing the problem, we set our goals, and determined what we can test in the following points:

- Functionality: we would like to test the internal functioning of the app.
- Graphical user interface style: we would like to test if everything renders correctly.
- Performance: we hope to test the performance of the app.
- Security: we also want to test the security level of the app.

But before we could tackle that, we started by getting familiar with the testing job terminology. We talked about two different test development styles, which are:

- Test driven development (**TDD**) essentially means we're going to make our coding evolve around the tests, and the other way around.

- Behavior driven development (**BDD**) falls in the same context as TDD, only it has an added value of having tests designed around behavior, instead of function or features. As a result, project managers and stakeholders can be involved in the progress of the project without having to understand the coding technical terms.

After discussing test development styles, we moved to explaining the three famous testing techniques which are:

- White box: it means we are testing a system, with full knowledge of its architecture, source code details, structure, algorithms used in it, and its logic.
- Black box: it means we are testing a system, without any knowledge of its internal working or logic.
- Gray box: it is basically a combination of the previous two techniques; we test a system, while having some knowledge of its internal working, but not necessarily full access to its source code or documentation.

Implementation and execution

After gaining enough knowledge about the testing job, we moved to execution. In this phase, we needed to find and select the tools that we're going to work with, or use. We also needed to analyze the existing platforms, and see if we can combine different libraries and platforms.

To conduct this phase correctly, we started by making a list of options we need and features we're looking for in a tool, then we started searching for different tools and services, that could provide us with what we need. We started comparing what they offer against the list of features we made, by making comparative table of pros and cons of each service, which we'll attach in the appendices. This helped us in choosing the top four tools and services, which we can try out. Finally we ended up by choosing Appium as an automation tool, and SauceLabs as a service that provides us with multiple mobile devices on the cloud.

After analyzing ScreenDy's architecture and the features and possibilities provided by the testing automation tools present in the market, we devised two solutions:

- We called the first one, the specific test, this test is based on the gray box technique, which means we have an idea about the user's app, and its structure, and we target specific plugins to test them, by feeding the app different inputs, and verifying the outputs, and tapping specific links and buttons, hence the name "specific test". We also designed a pseudo-language, to help the user design his own test cases and test scenarios, by writing simple lines of code, so he can target exactly what he needs to test.
- The second one is the generic test, this test is based on the black box technique; we start testing the user app with no knowledge of its structure or architecture. Basically we built a script, that crawls into the apps structure, blindly, without any knowledge, and tries to click every button, and as it visits different views and windows of the app, we build a tree of the app's pages, or a storyboard, that we show in the end to the user as a report of the test, containing a tree of the app's structure. In every node in the tree, we provide screenshots, with the duration of the loading of each page.

Next, we wrapped up these functionalities inside a RESTful API, which means the requests and results would be communicated through an HTTP canal, in JSON format. This API is going to be used by the front end developer of ScreenDy, to integrate with the platform's studio, so that it enables the end user to test their app from the navigator. We also used web sockets to maintain the connection between our API's server, and the client in the studio, so that the results would be fed in real time, saving the user from the trouble of refreshing the studio every time.

Finally, to conclude, we talked about what we learned in this internship, and the experience it helped us gain, especially that the testing job is a new field in mobile industry, and still has to expand, and has more time to gain maturity. We had some challenges with the asynchronicity aspect of the tools we used, we had to manage that, and the lack of clear documentation made it harder for us, but since the tools were open source, we succeeded in surpassing those challenges, and we completed our tasks successfully.

Résumé détaillé

Notre stage a été effectué sous l'encadrement de M Tabyaoui Khalid et M Alaoui Mehdi extérieurement, et Mme Oumaira Ilham intérieurement, au sein de la société Media Mobility, spécialisée dans le domaine des applications mobiles, qui, actuellement, développe une plateforme, du nom ScreenDy, qui permet la création des applications mobiles sans avoir aucune compétence dans la programmation mobile.

Dans ce rapport, nous allons présenter l'organisation d'accueil et leur produit ScreenDy, puis nous ferons une brève présentation sur l'évolution des applications mobiles, ce qui introduira la problématique que nous avons essayé de résoudre au sein de Media Mobility, et qui est l'assurance du processus de qualité des applications mobiles. Nous expliquerons l'importance de la phase des tests dans n'importe quelle société de logiciels, notamment celles spécialisées dans la création des applications mobiles, puis nous décrirons comment notre travail a pu optimiser la phase des tests dans la société, en introduisant et implémentant un système d'auto-testing, qui pourra assurer les scénarios de test de routine, et donc libérer le temps des développeurs afin de se concentrer sur d'autres tâches.

Méthodologie de développement et gestion de projet

Pour conduire notre projet efficacement, et avoir un développement flexible, nous avons utilisé l'une des méthodes de développement les plus connus du nom Scrum. Scrum est basé sur des incréments, on peut définir un incrément comme une petite part du projet. On reportait notre progrès quotidien aux encadrants, et à chaque début de semaine, on tenait une réunion pour exposer les tâches effectuées la semaine précédente, et puis discuter le planning des tâches de la semaine en cours. De cette façon on arrive à informer toute l'équipe de notre avancement, et donc ils peuvent nous aider si on fait face à un défi, et puis nous aider pour identifier la source du problème. Cette méthode nous a permis d'avoir une progression continue dans notre projet, et sans interruptions.

Documentation et projet d'intégration

Avant d'entamer notre tâche principale, on a commencé par un projet d'intégration qui consistait à la création d'une application mobile entièrement fonctionnelle en utilisant seulement ScreenDy, pour se familiariser avec la plateforme, ainsi que sa structure et son fonctionnement, et comprendre son architecture ce qui nous facilitera la tâche des tests par la suite.

ScreenDy est une plateforme pour la création des applications mobiles très rapide, basé sur le cloud, on y accède en créant un compte, et elle n'utilise que le drag & drop, c'est-à-dire on n'a besoin que de faire glisser les composants pour les instancier et puis les paramétrer pour créer notre propre application mobile personnalisée. Tout ceci est fait trois fois voire dix fois plus rapide que la méthode classique. De plus, le produit final est cross-plateforme, c'est-à-dire on crée l'application une seule fois sur la plateforme et on génère deux versions identiques sur iOS que sur Android.

Dans cette première phase, on avait une tâche secondaire qui était de documenter la plateforme, ses composants et leurs paramétrages, ainsi de faire des tutoriels en anglais pour aider les nouveaux utilisateurs dans leurs premiers pas dans la création d'une application mobile sur ScreenDy.

L'application a été créée en moins d'une semaine, identique au design proposé par le client, et était fonctionnelle sur les deux plateformes iOS et Android, comme attendu.

Connaissances de base du testing

L'utilisation des applications mobiles a connu dernièrement une forte évolution, et la sérieuse compétition entre les entreprises pour gagner la plus grande part du marché les a poussé à penser à des nouvelles méthodes pour attirer l'attention des consommateurs, et c'est pour ce que la qualité des produits est jugé essentiel, ce qui prouve le besoin d'une phase de test avant la sortie d'une application mobile au grand public. Cependant, La phase des tests peut s'avérer très cher en terme de temps et d'argent, notamment si les testeurs et les développeurs devront exécuter les mêmes scénarios de test plusieurs fois, et à chaque fois ils rajoutent et/ou modifient le code, ce qui résulte dans la création d'une nouvelle

itération de test. D'où nait le besoin d'automatiser la phase des tests, et de créer un système qui pourra se charger des scénarios de test de routine, et ceci a été notre tâche principale.

Après avoir analysé le problème, on a défini nos objectifs finaux, et déterminé ce qu'on pourra tester dans les applications mobile dans les points suivants:

- Fonctionnement: En premier lieu on voudrait tester le fonctionnement de l'application, si elle ne bug pas au cours de son exécution.
- Style des interfaces graphiques: On veut ensuite s'assurer que les pages sont bien affichées, les composants sont bien positionnés, et les styles sont appliqués (couleur, image de fond, taille de texte...)
- Performance: Ensuite on voudrait mesurer la consommation de RAM, CPU au cours de l'exécution de l'application mobile.
- Sécurité: Finalement, on voudrait s'assurer du haut niveau de sécurité de l'application et éliminer les lacunes dans ce sens.

Mais avant d'entamer ceci, on a commencé par se familiariser avec le travail des testeurs et son standard. On retrouve deux styles de développement des tests:

- Test driven development (**TDD**): Le développement orienté test est lorsqu'on est guidé par les tests, ce qui veut dire qu'on commence par écrire les scénarios de tests puis on développe les fonctionnalités selon ces tests. Une fois les tests sont réussi, on arrête le développement vu que les cas d'utilisation voulus sont déjà fonctionnels.
- Behavior Driven Development (**BDD**): Le développement orienté comportement est similaire au TDD mais plus raffiné et plus correcte. La différence est que celui-ci est basé plutôt sur les interactions de l'utilisateur avec l'application, c'est-à-dire on se concentre plutôt sur le comportement de l'application et les interactions avec l'utilisateur que sur les fonctionnalités techniques.

Après avoir défini les styles de développement des tests, on est passé par la suite aux techniques des tests les plus connues:

- Boite noir: C'est le cas d'exécuter un test sans connaitre l'architecture interne de l'application, ni son code source, ni sa logique.

- Boite blanche: Dans ce cas, on connaît toutes les informations à propos de l'application, et on conçoit le scenario du test selon ces informations.
- Boite grise: Une combinaison des deux techniques précédentes résulte dans une troisième technique où on teste en utilisant un minimum d'informations à propos de l'application, mais sans avoir d'accès direct au code source.

Implémentation et exécution

Après avoir acquis assez de connaissance à propos du testing, on est passé à l'exécution, et dans cette phase, on avait besoin dans un premier temps de choisir les outils à utiliser comme base pour notre système d'auto-testing, et donc on a analysé les différents plateformes de test existantes en ligne, pour en choisir une ou combiner plusieurs selon les fonctionnalités qu'offre chacune.

La première étape dans cette phase fut lister les options dont on aura besoin dans une plateforme, et puis rechercher les plateformes de tests existantes en ligne qui répondent à nos besoins. Puis on a procédé par un benchmarking de ces plateformes en créant un tableau des avantages et inconvénients pour chacune. Ceci nous a permis de choisir les quatre meilleures solutions pour notre cas. Puis après les avoir essayé, on a fini par choisir deux solutions: Appium qui est un outil d'automatisation des tests open source, et SauceLabs qui est un service payant de testing sur plusieurs gadgets en cloud.

Après avoir analysé l'architecture de ScreenDy et son fonctionnement et les possibilités offertes par les plateformes de test choisies, on a implémenté deux solutions:

- La première fut le test spécifique. Ce test est basé sur la technique de la boite grise; on connait la structure de l'application mobile, et on cible le test des plugins ou composantes ScreenDy, en cliquant des liens et des boutons, alimentant les inputs de l'application et vérifiant l'output et les résultats. On a également créé un pseudo-language simple qui permet à l'utilisateur final d'écrire lui-même son propre scénario de test pour tester son application mobile, et de voir le résultat du test après son exécution. Le résultat du test est sous forme de rapport qui illustre les étapes des tests avec leurs durées, des captures d'écran d'avant et d'après l'exécution de l'étape, ainsi son état, réussite ou échouée.

- La deuxième solution est le test générique. Ce test est basé sur la technique de la boite noire, donc on essaye de tester l'application mobile sans avoir la moindre information sur sa structure ni sur son code source. Concrètement, on a créé un script qui fait le tour de l'application depuis la page d'accueil (Home Page), et qui passe par tous les éléments cliquables existants dans l'application pour visiter toutes ses pages, et génère vers la fin un rapport contenant des captures d'écran de toutes les pages sous forme d'arbre représentant la hiérarchie de l'application, avec le temps de chargement de chaque page. Comme paramètres pour ce test, on peut rajouter des informations d'authentification (dans le cas où l'application contient une page d'authentification) qui seront entrées automatiquement une fois le script du test atteint la page d'authentification, puis continuer la visite des pages restantes.

Ensuite, on a inclus ces deux fonctionnalités dans une API REST, qui est un serveur qui gère des requêtes HTTP arrivants du côté client, exécute le test spécifié, puis retourne la réponse via HTTP sous forme de JSON. Cette API sera intégrée dans la plateforme ScreenDy pour permettre les utilisateurs finaux de tester leurs applications mobiles depuis le navigateur web, en choisissant le device/émulateur sur lequel ils veulent tester, ainsi que la version de l'OS. Le test est donc exécuté sur SauceLabs, plateforme des devices en cloud, puis le résultat est retourné sous forme d'un JSON.

Pour dynamiser l'affichage du résultat côté client, on a fait recours à la technologie des web sockets, afin de garder la connexion retenu entre l'API et le client, et de recevoir les résultats en temps réel sans avoir à rafraichir la page à chaque fois.

Finalement, pour conclure, on a parlé de l'expérience vécue durant ce stage de fin d'études et le savoir-faire accumulé, surtout dans le domaine des tests des applications mobiles, qui est un nouveau domaine dans l'industrie du mobile, et en cours d'évolution et d'expansion. Evidemment on a rencontré des problèmes et des défis au cours de notre recherche et développement, notamment dans le choix des outils à utiliser, ainsi qu'avec l'aspect asynchrone des outils utilisés, en plus du manque de la documentation des différents outils de test, mais vu que les outils étaient open-source, on a pu surpasser ces défis en analysant le code source et déterminant le fonctionnement des outils, et donc compléter nos tâches avec succès.

Chapter 1: Context and objectives

This chapter will be an introduction of the establishment where we were offered our internship and will include information such as the history of the organization, its structure, technical background and its role in IT industry and our society.

1.1. Company identity: Media Mobility

The startup company Media Mobility offers innovative mobile solutions for its different and many consumers, counting telephone operators, agencies and enterprises. Founded in 2007, Media Mobility has mostly been the leader in creating and deploying mobile apps. Based first at Paris, the company holds today up to 20 developers in Dubai, Casablanca, and soon in San Francisco.

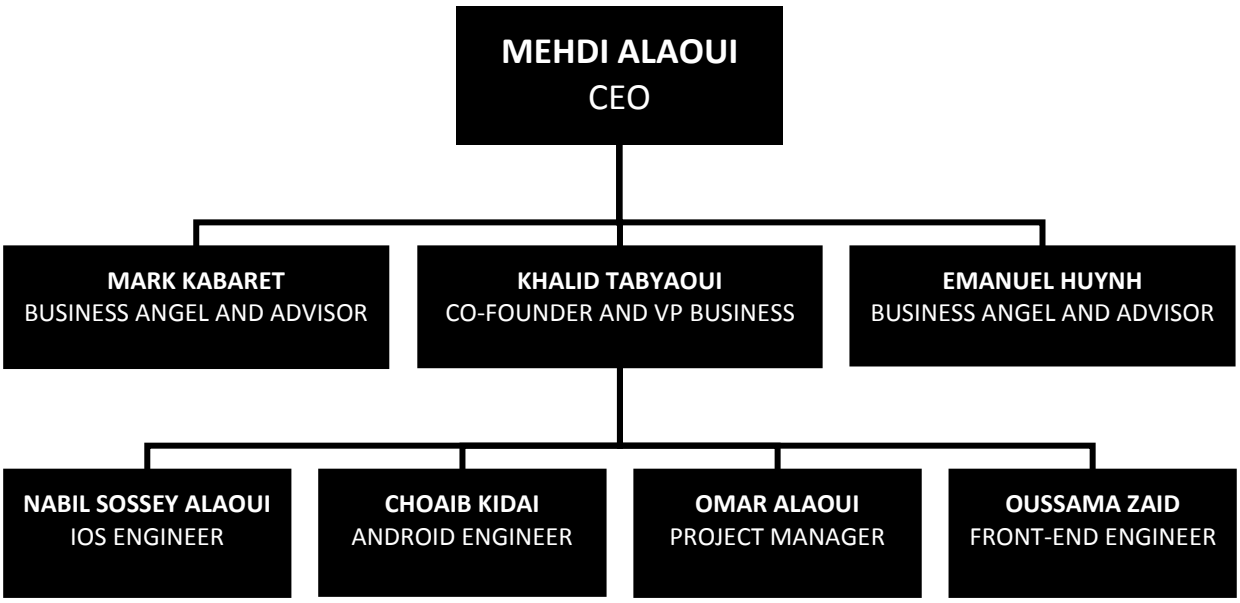


Figure 1 : Media Mobility's organigram

1.2. ScreenDy

As the mobile app market size is growing fast and doubles from one year to another, there are only 3 million mobile developers while there are more than 15 million web developers. The demand for apps is unmet due to the shortage of mobile developers.

This large demand for native apps makes the developers wanting to meet clients' needs faster, which is not always possible because coding takes time. Besides, the growing number of mobile operating systems and devices with different sizes makes the coding even more time and money consuming.

It's pretty predictable that the industry will move towards having development tools where one only creates one code that accounts for 85% of the work and that is then automatically updated and transformed to work cross-platform and across all delivery channels (mobile, tablet, connected TV, connected car, desktop, etc…).

In this context, Media Mobility's team created the most advanced tool to allow any and all developers to quickly and easily build native, custom, advanced and sophisticated cross platform apps: ScreenDy platform.

ScreenDy provides a unique process whereby all developers can build mobile apps even if they aren't mobile developers and do not know mobile coding. Dedicated mostly to web and media agencies, ScreenDy is made so fun and so simple to use that any web developer could redo 80% of all existing apps on the Apple/Play Store within few days, and without having to learn any mobile development language.

The platform is based on WYSIWYG (what you see is what you get) editor, accessible on the cloud, that enables creating apps using a library of components (plugins), with only Drag & Drop gestures. The library of components is supported by two different kernels, Android and iOS, each responsible of generating the right code for each component.

Screendy is a 1 000 000 lines of code today, and holds up to 100 native components in its evolving library, and holds many features such as:

- **Drag and Drop gestures**

Building an app using ScreenDy only uses Drag & Drop gestures. The library provides many components like galleries, forms, RSS lists, reveal menu… that can be dragged to the app and then parameterized.

- **Live testing on devices**

The platform holds a testing tool to preview the app on other devices, test the functioning of the app, as well as estimating the loading time of each page in the app.

- **Live native update (bypassing Apple & google constraints)**

Thanks to its fluent schema, ScreenDy's app can be instantly updated in Apple/Play store, without having to wait for verifications.

- **Pre integration of third parties services**

ScreenDy's components can be static as well as dynamic, and in this case, the component is fed by a third party service that sends the data over internet network

- **Best practice & Pattern design for ux/ui**

ScreenDy guarantees a good user experience, by providing a simple interactive interface, as well as an intuitive configuration.

- **Native performance**

Mobile apps fall broadly into three categories: native, web based, and hybrid. Native applications run on a device's operating system and are required to be adapted for different devices. Web-based apps require a web browser on a mobile device. Hybrid apps are 'native-wrapped' web apps. ScreenDy focuses on native apps, because they can utilize the device's native features (e.g. camera, sensors, accelerometer, geolocation), which makes the app much fluent and stable.

Awards

For its innovative idea, and promising solutions, ScreenDy got selected for many competitions all over the world, and won many awards. Below is a timeline for important competitions won by ScreenDy.

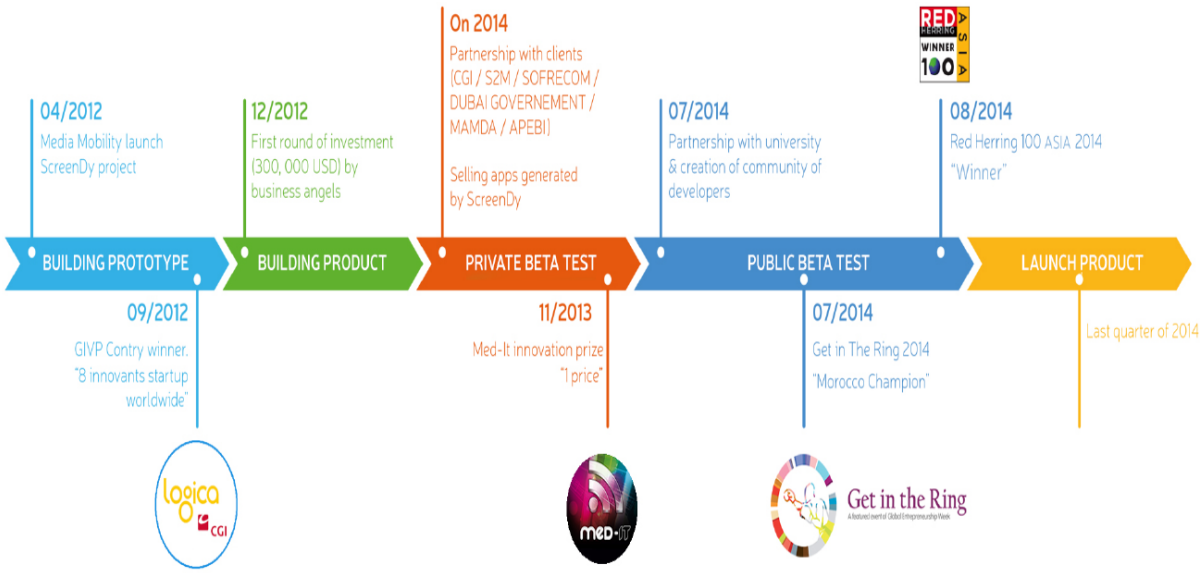


Figure 2 : Awards won by ScreenDy

ScreenDy's main competitors are:

- Phonegap: is more known because they were the first on the market but they do not allow building native apps.
- Titanium: is the most feared competitor. They are however very complex to use! (Selected most complex tool by Research2guidance).
- Xamarin: is very good but can only be used by .Net developers.
- AppMachine: may also become a competitor if they get into the custom-made apps and shift their focus from b2c to b2b.

There are many tools to make either a native, a cross platform or a complex app but no single tool that allows to make all those. That is until ScreenDy arrives.

- Output:
  - Custom, native and sophisticated apps;

- Custom apps based on own client design;
- Extensible apps (adding new plugins is easy).

- Simplicity:
  - Visual tools (drag & drop);
  - Cloud platform (no need to install SDK)
- Skills:
  - Quick learning time (few hours);
  - Few web technology skills needed.
- Cost:
  - Free access to the platform to build & test the app, users only pay to publish an app.
  - ScreenDy is disruptive by its business model: In fact, the user can have access to the cloud where he can find all necessary tools to build and test in real time. He can get his app on 10 devices maximum. This approach enable users to present their apps to their customers. When the user gets his pre-order or wants to monetize he can go to the platform to pay in order to publish the app on the app stores.
  - The cost is between 500dollars to 1500 dollars/app depending on the complexity, after that, if the user wants to update his app, he would pay 100 dollars per update.
  - ScreenDy also allows users to have access to its market place where he can find native components or different add-ons furnished by third parties as advertising or analytics. This approach enables developers to optimize their revenues.
- Services:
  - One dashboard to follow app life cycle;
  - Real time update.

1.3. Development methodology

1.3.1. Case study

A case study is an account of an activity, event or problem that contains a real or hypothetical situation and includes the complexities encountered in the workplace. Case studies are used to help to see how the complexities of real life influence decisions.

Analyzing a case study requires to practice applying the knowledge and the thinking skills to a real situation. To learn from a case study analysis we will be "analysing, applying knowledge, reasoning and drawing conclusions" - *Kardos & Smith 1979.*

There are many types of case studies. Ours required to solve a problem by developing a new tool. This type of case studies is a problem oriented case study.

Overview: Mobile app development is a relatively new phenomenon that is increasing rapidly due to the popularity of smartphones among end-users. ScreenDy is a WYSIWYG platform, recently developed, for making native Apps on iOS and Android, using only drag and drop gestures.

The problem: As the platform is still in development state, changes are frequent in the iOS and Android kernels. It's getting hard to assure the stability of old and new plugins in the platform using only manual testing. Besides, Media Mobility Company wants to upgrade the ScreenDy platform to include a phase of testing both locally and for the end-users.

Objective: The goal of our internship was to develop an auto-testing system that automates the task of testing all plugins, as well as testing separate apps which are made by ScreenDy, and finally wrap up the functionalities inside a RESTful API, so that they can be easily integrated and exploitable by the front-end part of the platform.

The solution: We combined open source tools such as Appium that allows running locally test scripts on user's device, as well as paid services, such as SauceLabs, which is a platform for cloud testing that provides many emulators/simulators on which we can run our test scripts remotely.

The results: The outcome is two types of tests: Generic and specific. The generic test runs on any app, and just goes through all the pages and generates a report containing a storyboard of the app, and the loading time of each page. The specific test is a scenario that the user writes to test his own specific app. Both tests can run locally, or on a cloud testing platform.

1.3.2. Scrum

For our management methodology, we used scrum, which was followed by Media Mobility team.

Scrum is an iterative project management method based on the incremental agile software development. Scrum relies on the project splitting into small parts called “sprint”. Each sprint can last from several hours to one month, but advised duration is two weeks. Each sprint is composed of tasks. Tasks are necessary steps to add a new functionality to the developed software. Sprints begin with the evaluation of tasks duration. During the sprint, each day a meeting is done between development team in order to report work progress and see if everything is right, during these meetings it is possible to reevaluate a task if problems had been encountered during the last day. At the end of the sprint a demonstration is done to the product owner.

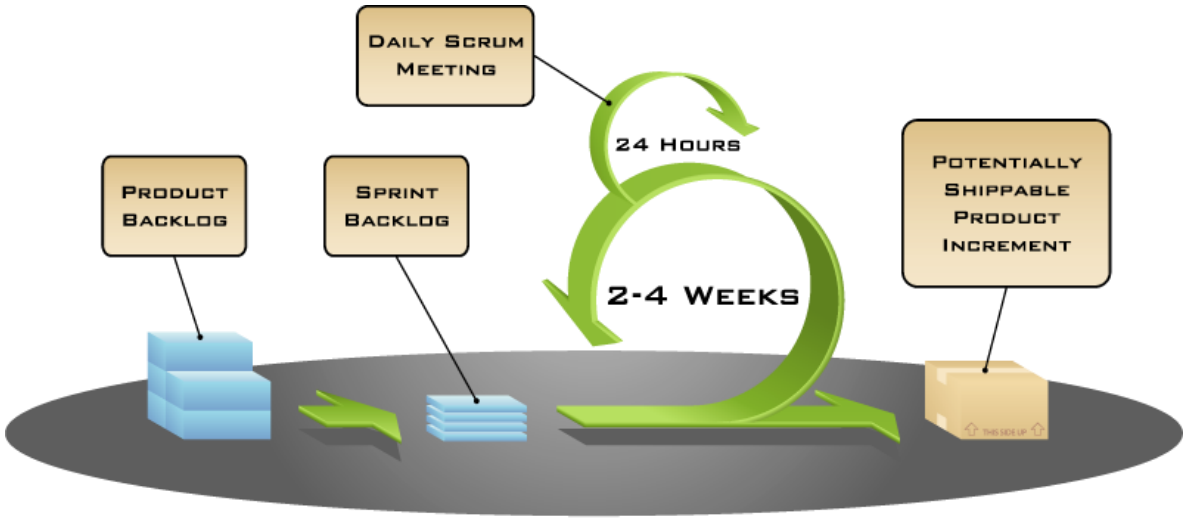


Figure 3 : Scrum's sprint schema

For Media Mobility team, sprints usually took from 1 to 2 weeks, after which we scheduled a meeting to discuss and review the work that was completed in this period and the planned work that was not completed, also we presented demos of the completed work. The completed and the not completed work reflected the limitations in the sprint, which enables the team to identify process improvement actions.

By the end of the meeting, each member of the team makes a list of the upcoming tasks that should be done in the following sprint.

In addition to the meetings by the end of each sprint, we used a tool of the name Trello board, to archive everybody's each day's tasks, and keep everybody updated of the whole team's work. This tool also enabled commenting on each task, to provide ideas and improvements of the task, and also committing other members of the team to some task, and specify the due date, which made the projects' managing a simple organized task.

1.4. Objectives

This report is intended to reflect our achievements, project works and professional growth during the internship period. Besides, anybody looking for a short detail about Mobile apps auto-testing may find this report useful.

This internship is designed to meet specific goals, which are:

- To familiarize with the professional environment which is Media Mobility Company.
- To familiarize with Media Mobility's main project that is ScreenDy, by making a fully working app using only the platform.
- To work on documenting ScreenDy platform and its plugins, and learn its internal working.
- To provide research and benchmarking of existing testing frameworks, both open source and paid ones.
- To automatize the testing task by creating an auto-testing system both for end user and local usage.
- To ease the integration of our auto-testing system into the platform's front-end by making a RESTful API, using REST server and web sockets.

1.5. Planning

The main project to be carried out during the internship was the development of an auto-testing system, to assure the stability of ScreenDy platform. Although, on the course of the internship we were involved in other activities concerning not only programming, but also documenting the use of the platform and other activities described in the following pages.

For that, we chose first to make a global planning that contained the big lines, and then in each step, we made the little tasks that we should work on. The Gantt chart included in the appendices represents the timeline of our dued tasks.

Chapter 2: Integration project

This chapter describes our first task in Media Mobility Company, which was a way of getting familiarized with ScreenDy platform and getting used to its internal working.

2.1. Creation of an App using only ScreenDy

As explained in the earlier sections, ScreenDy is a cloud platform, created by Media Mobility team, to make the task of creating Apps easier and faster, even for non-mobile developers.

As a first task in the company, we were asked to make a full functioning mobile App, using only ScreenDy platform.

The latest version of the studio was still in implementation phase during our first task, so we used the older version, which was simple as well, and intuitive.

The app that we were asked to make was composed of four pages, each one containing variant plugins, and all accessible via a drawer menu.

To start off with our task, we created two accounts in the platform, and then we were given a template for all the pages of the desired app, and some documentation about the working of the platform and its plugins.

The platform's interface is composed mainly of three parts:

- The library that contains all available ScreenDy plugins.
- A side menu that gathers all used plugins in the app (for quick access), including the pages.
- A simulator to which we drag the components, and which shows how the app would look like on the smartphone.

We were able to make a full functioning app looking exactly like the template given by the client in less than a week. The app's pages are attached to the appendices.

How it works

The library offers a variance of components that can be used in an app depending on our needs. A component can be instantiated by dragging it to the simulator view, and then configuring its parameters.

To give the components a nice design, we create styles that can contain many design parameters, such as the width, the height, the color, the background color… and then we assign each style to the right component.

The studio is so simple to use, and intuitive that we were able to make a fully functioning app in less than a week.

ScreenDy platform, as described previously, generates the app for both Android and iOS versions, with an identical result.

2.2. Documenting ScreenDy platform

As the platform is still young and new, its documentation is still poor compared to the many functionalities and features it offers. So one of our tasks in the company was to create a tutorial for creating a simple app using only ScreenDy, and then documenting the basic plugins existing in the platform, to help users get started with ScreenDy.

2.2.1. What is documentation?

Many factors contribute to the success of a software project; documentation included. "Software documentation is an artifact whose purpose is to communicate information about the software system to which it belongs." - *"Software Documentation – Building and Maintaining Artefacts of Communication" by A. Forward, MS thesis. Institute for Computer Science, OttawaCarleton. Canada. 2002.*

Parnas defines a document as "a written description that has an official status or authority and may be used as evidence. In development, a document is usually considered binding, i.e. it restricts what may be created. If deviation is needed, revisions of the document must be approved by the responsible authority." - *Parnas, D.L.: Precise documentation: The key to better software. In: Nanz, S. (ed.) the Future of Software Engineering, 2011, pp. 125–148. Springer, Heidelberg*

2.2.2. Why documenting?

Documentation has been proved to increase the level of confidence of the end deliverable as well as enhance and ensure product's success through its usability, marketability and ease of support.

"The dominant factor between a successful project and an unsuccessful project reduces to the effective dissemination of key information and successful software projects become successful because they give the right level of attention to clearly communicating the key concepts and requirements" - *Elowe, The role of documentation in software development. 2006.*

On one hand, a successful documentation makes it easy to access information, provides a limited number of user entry points, helps new users to get used to the tool quickly, simplifies the use of the product and helps cut support costs.

On the other hand, poor documentation is the first cause for bad user experience with the product, as well as for many errors, which reduce efficiency in many phases from the development of the software to the its use. Documentation is an important activity that should start from the beginning of the software development and continue through all phases in its cycle. It's a tool for better planning and decision making.

2.2.3. Application and benefits of documentation

In a software engineering environment, documentation is used to communicate information to its audience and instill knowledge of the system it describes.

Documentation can also be used for learning a software system, testing a software system, working with a new software system, solving problems when other developers are unavailable to answer questions, looking for big-picture information about a software system, maintaining a software system, answering questions about a system for management or customers, looking for in-depth information about a software system, working with an established software system.

Akin-Laguda lists other uses which include: "facilitates effective communication regarding the system between the technical and the non-technical users, training new users, solve problems like troubleshooting, evaluation process, and quantify the financial

ramifications/footprint of the system" - *F. Akin-Lagida, Documentation in programming. NASSCOM. 2013.*

All the above mentioned uses of documentation, can be simplified and summarized as:

- Documentation is used as a communication medium between members of the development team and probably the clients;
- Used for maintenance;
- Provide information for management to help them plan, budget and schedule the software development process;
- Tell users how to use and administer the system.

2.2.4. Types of Documentation

Sommerville describes two main categories of software documentations in "Software documentation": process and product documents.

Process documentation is used for managing the development process like planning, scheduling and cost tracking, standards among others.

Product documentation describes the main deliverable, which is the software product. It includes: Requirements, Specification, Design documents, Commented Source Code, Test Plans including test cases, Validation and Verification plan and results, List of Known Bugs and user manual.

2.2.5. Software Documentation standards

Standardized documentation can be defined as documents having a consistent appearance, structure and quality. This means it should be easy to read, understand and use.

Standards act as a basis for document quality assurance. "Using a standard means that documentation producers and customers have a consistent accepted reference for the format and content that they will find in the documentation. For example, what does it

mean to say that the documentation is "complete"? Does it have to include every function and screenshot?" - *A. Reilly, Audience-Oriented Standards for Software Documentation from ISO.*

Reilly discusses various ISO Software documentation standards which include:

- ISO/IEC/IEEE 26514:2008, Systems and software engineering-Requirements for designers and developers of user documentation: This standard describes in details both process and product standards.
- ISO/IEC/IEEE 26513:2009, Software and systems engineering-Requirements for testers and reviewers of user documentation: This standard covers activities related to planning and conducting documentation reviews as well as managing the results of the reviews. It also covers the conduction of usability tests for documentation along with tests of accessibility of localized or customized versions.
- ISO/IEC/IEEE 26512, Software and systems engineering Requirements for acquirers and suppliers of user documentation: It defines the processes for acquiring user documentation and managing contractors.
- IEEE Std 1063-2001, IEEE Standard for Software User Documentation: This standard is a revision of IEEE std 1987. This standard covers mainly the software documentation product in all its formats; printed user manuals, online help, user reference documentation, without including the software's development nor managing the software user documentation. It includes the minimum requirements for structure, information content, and user documentation format.

2.2.6. Documenting ScreenDy

For documenting ScreenDy platform, we had to first make a tutorial as we were working on creating an app on the platform. The tutorial was made to help new users in their first steps with ScreenDy. It describes in details how to start off with the platform from creating the project, to making new pages, to dragging new components and styling them, and finally visualizing the app on both iOS and Android versions.

Besides that, we were asked to document the basics plugins in the platform, and explain each component's structure, features and how to use it.

Chapter 3: Introduction and background related to testing

This chapter is an introduction to the mobile apps testing job. It first analyzes the importance of testing and quality assurance in many fields, and then summarizes all the researches we made to define the different testing techniques and styles, and finally set the needs of the host company.

3.1. Introduction

Mobile app testing and monitoring is regarded as a key enabler in the application development process and market growth is expected at an annual rate of 30% (2012-2017) and could reach close to $800 million by the end of 2017, according to ABI Research. About three-fourths of current revenue base comes from tools that enhance manual testing, however in the coming years the market may be driven predominantly by test automation. What makes testing mobile software more demanding than testing computer software is the sheer complexity of the environment, which stems from the diversity of devices, screen sizes, operating systems, and networks. Furthermore, for consumer-facing apps the low barriers to entry and the intense competition over subscribers' attention mean that the first impressions are critical for any application's success. Mobile software testing is critical to that objective.

3.1.1. Growth of mobile apps

Mobile devices have witnessed a phenomenal growth in the past few years. A study conducted by the Yankee Group predicts the generation of $4.2 billion in revenue by 2013 through 7 billion U.S. smartphone app downloads.

This growth in mobile phones and smart phones, came with a new opportunity that is the mobile apps market. Each smartphone manufacturer releases to the public an API for their phone's OS that developers can use to create apps for almost any purpose we can imagine, from simple text editing apps, to complex games and project managing apps. This availability of programming resources made it possible for anybody to create an app, from individuals to big enterprises and companies, which resulted in a big growth of mobiles apps, as the statistics show, as of September 2014, 1.3 million mobile apps were available in the Apple App Store. In terms of money, apple announced that customers spent 10 Billion dollars just on apps.

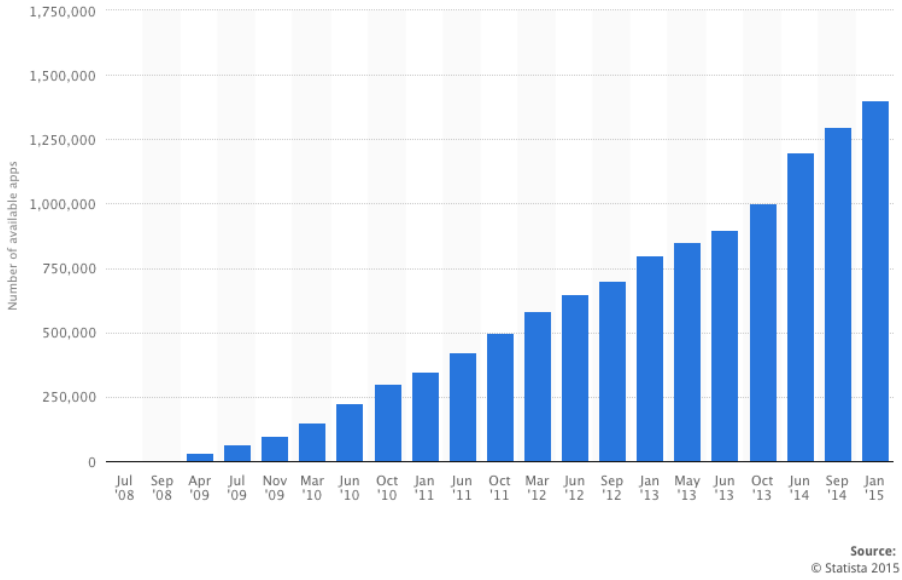


Figure 4 : Graph of number of mobile apps in Apple store - provided by Statista.com

And for android, as of May 2015 the Google Play store has more than 1.5 million mobile apps available for download. These statistics show how much money and effort and time goes into mobile apps industry, and how much it affected people's lives worldwide. It's true that the consumption and demand of mobile apps is high, but so is the offer, and the competition between enterprises and companies has become fierce, and they're all trying to gain the biggest share of the market, and the only way they can do that, is by assuring the quality of their products and maintaining a quality process during the development of their apps.

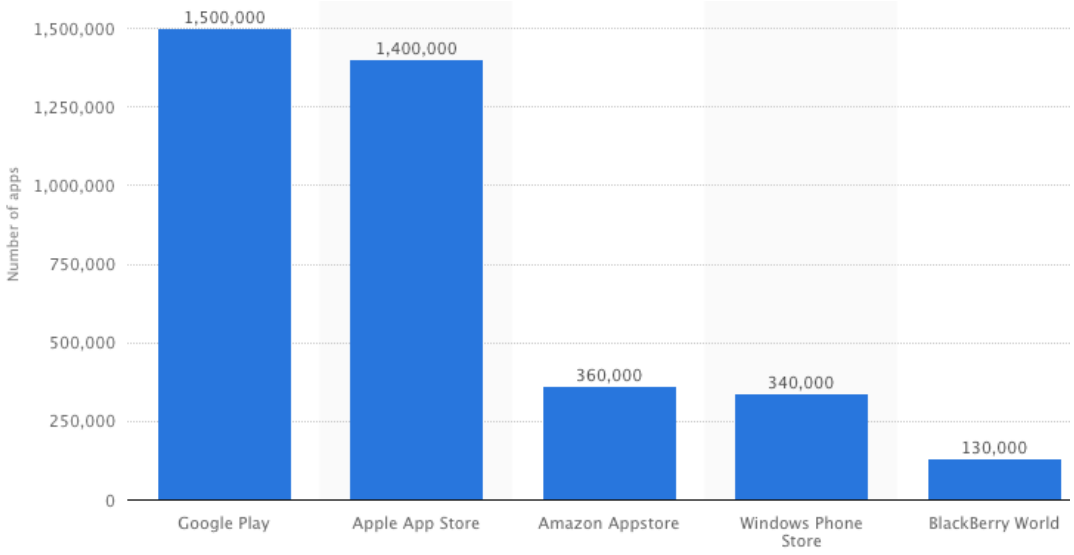


Figure 5 : Number of mobile apps in different platforms

3.1.2. The mobile-testing job

Mobile application testing is a process by which application software developed for handheld mobile devices is tested for its functionality, usability and consistency. Mobile application testing can be automated or manual type of testing. Mobile applications either come pre-installed or can be installed from mobile software distribution platforms.

To qualify an app as a quality product, it should meet some standards which we can summarize mainly in these criteria:

- The app should be stable and handle all errors, unpredicted or unexpected usage (e.g. show error messages when the user tries to do something he's not supposed to do).
- The app should not crash under any circumstances (incoming call, sudden drop of internet connection, interruptions etc.).
- It should use as low as possible of CPU time and memory.
- The graphical user interface should be easy to use and to understand.
- Data transfer should be secured.

The developers should handle all these anomalies before the app can make it to the public, and that's when we realize the importance of the testing phase, and that's where testing engineers come.

It may seem that it's not that important, developers can do that, but it'll consume huge amounts of their productivity time, and that's not good for the company, especially when projects get bigger and more complex.

Why should we have a testing phase?

The main reason why we should have a testing phase is, simply put, the cost of fixing bugs is much cheaper during the development stage, and increases drastically as time goes, until it becomes much expensive during the live use after the release of the product. The earlier the defect is found, the lesser is the cost of defect. For example if an error is found in the

requirements specification stage, then it is somewhat cheap to fix it. The corrections can be done and then it can be re-issued. However, if a defect is not detected until the system has been implemented or even at release time then it will be much more expensive to fix. This is due to the need of reworking the specification and design before changes can be made in construction, because one defect in the requirements may well propagate into several places in the design and code, and because all the testing work done to that point, will need to be repeated in order to reach the confidence level in the software that we require.

	Coding	Integration	Beta-testing	Post-release
Hours to fix	3.2	9.7	12.2	14.8
Cost to fix (USD)	240	728	915	1110

Table 1 : Cost to fix defects in each development stage (Automated testing ROI: fact or fiction? by Paul Grossman)

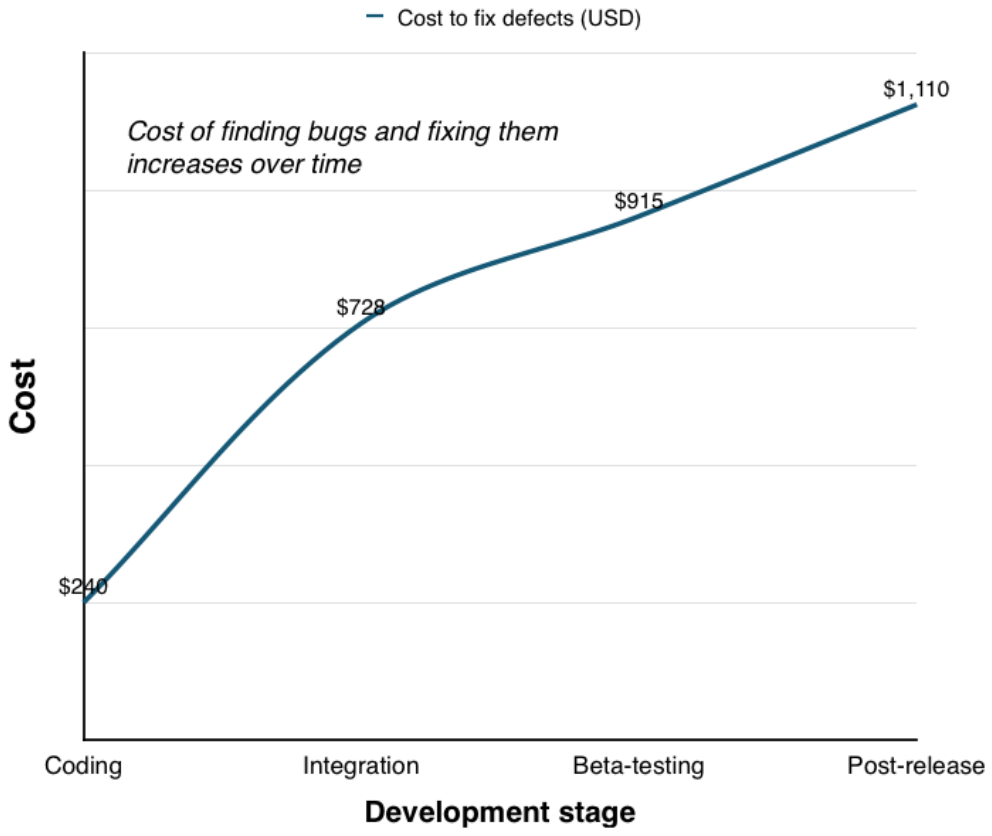


Figure 6 : Graph of cost to fix bugs in each development stage

The second reason that we should not ignore is the reputation of the company and the satisfaction of the consumer.

"The user is willing to accept a mild level of defects, as long as they can easily be worked around and are fixed when reported. We can speculate a little on the relation between customer 'satisfaction' and residual software defects. We can expect, for instance, that if the residual software defect density increases, customer satisfaction decreases. If the number of defects encountered by the customer increases beyond a certain level, we can expect that customer satisfaction will drop so low that the users will avoid using the product. The above assumes that our starting point is complete customer satisfaction, with the target customer unaware of any defects in the product. If the starting point is different, for instance if the target customer assumes a buggy product, based on past experience, it will be harder to re-establish customer satisfaction. A curve that plots customer satisfaction against residual defect density, therefore, will likely exhibit hysteresis (capturing the notion that customers have memory and are affected by past experiences). An attempt to capture these notions is shown in Figure 7. If we start at the point labeled 1 on the upper curve in Figure 7, and slowly move towards point 2, small changes in the residual defect density do not seem to affect customer satisfaction all that much. When we pass 2, though, the effect will become very noticeable. Similarly, is we start on the lower curve at 3 and move towards 5, small changes in residual defect density do not cause easily observable effects, not even if we pass 4, restoring the same residual defect density we had before we started losing users at point 2. Any improvement in defect density will now have to be considerably greater, before it can restore customer confidence that the product is worth using. (It might explain why many companies often decide to abandon a product at this point, rather than attempt to restore lost customer confidence.)" *– From Economics of Software Verification by Gerard J. Holzmann*

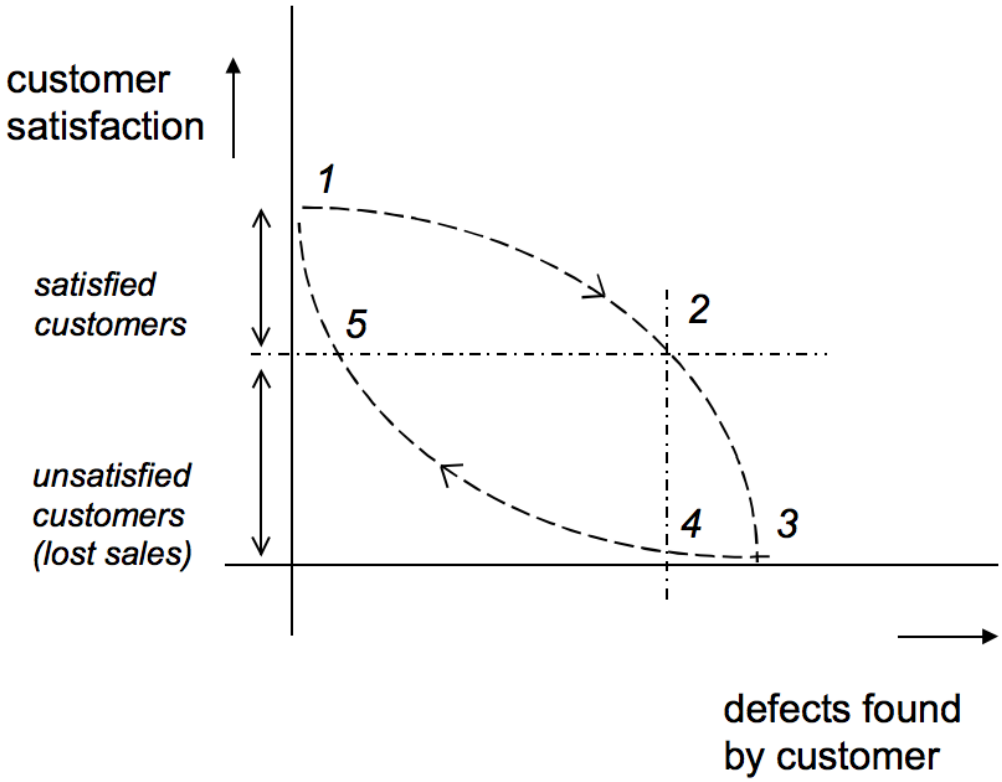


Figure 7 : Graph of correlation of defects found with customer satisfaction

From this we can conclude that bugs that reach the consumers, harm the company's income, both internally, as costs increase, and with loss of consumer's trust, the company could lose potential future sales.

To make it more clear why it is so important to test before release, we're going to give real life examples, to illustrate how dangerous it is to detect a bug only in live use and after release.

The first one is a bug that went undiscovered, and that made the company, Knight Capital, lose **172,222$ a second for 45 minutes**, which totaled to, approximately, 460 million dollars, effectively bankrupting the company. After investigation, it turned out that the bug was originating in a poorly coded block, that haven't been used for 9 years at that time.

The second bug is a rather critical one. In 1983, there was a bug in a Soviet Union's military warning satellite system, that interpreted sun's reflections off the cloud tops, as nuclear missile launch coming from the U.S. Luckily the lieutenant colonel that was on duty that day, decided to report the warning as false alarm, which saved everybody from an event that could have caused the **World War III**.

The third one is about a mobile app bug. This one was discovered in a banking app that enabled anyone that has physical access to the phone, to easily get user's bank account information, which could compromise their account, and get their money **easily stolen** from them.

From all of the above, we can conclude the danger of letting a bug reach the public, for the companies and the consumers, hence, we can affirm that having a testing phase during the beginning of the app development lifecycle, is important if not crucial.

As engineers, we always like to push things further, and optimize processes. For that reason, we're going to talk about test automation next.

3.1.3. Auto-testing systems

By making scripts that run routine tests automatically, we can let the machines test our apps, in parallel, on a number of different devices, and generate for us reports of those tests, with screenshots and even videos of the test to check later, and we can spend that gained time in more productive tasks, like testing unusual complex scenarios to detect more resilient bugs, and developing smarter testing scripts that can detect different types of bugs.

To try to get the importance of the auto-testing, we tried to imagine a hypothetical experiment in the following scenario.

A company X decided to start a new project of a new mobile app. The developers' team planned how they are going to handle the development, and they started working on it. After a while, they got version 0 of the app, before releasing the app to the public, they made sure there are no hidden bugs anywhere, by using unit testing and integration testing with all their phases, and they fixed all the detected bugs, then released the app. Once in public, they started getting feedbacks about more bugs, and more features to add. Of course the team will fix those bugs, implement the new features; only while implementing them, they have to modify the old code and they can't know if the changes broke the old features, so they have to test the new features and the old features.

The company decides to recruit a tester, and the more versions they release, the more features they add, and the tester should always test old features plus new features, which

will be accumulated into huge tasks as time goes by and the need for more testers will grow, to maintain their control on the hidden bugs.

We can summarize the scenario in this approximate table:

Version	N° of developers	N° of testers	Features to code	Features to test
0	2	1	2	2
1	2	2	3	5
2	3	3	3	8
3	4	4	5	13
4	6	6	6	19

Table 2 : Table of testing needs over time

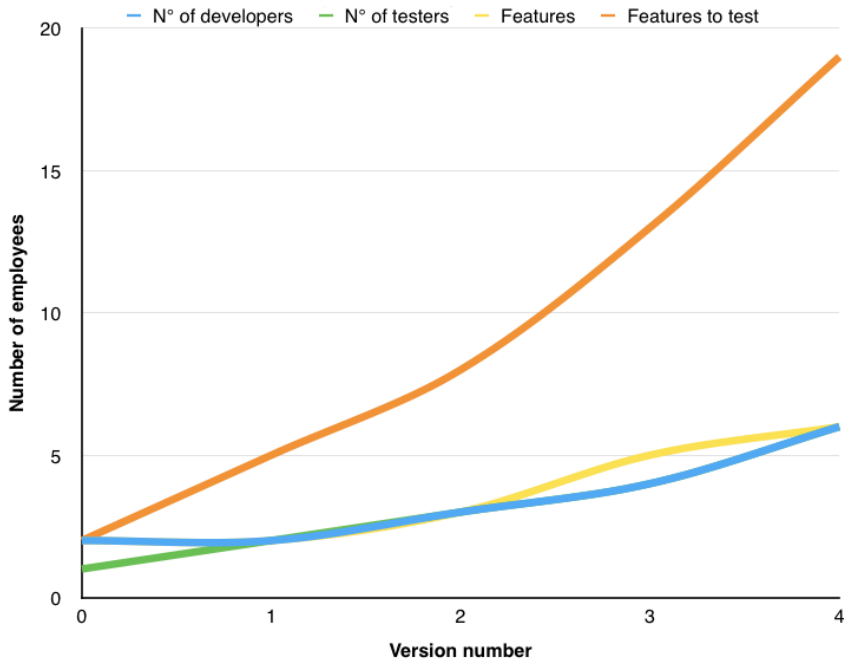


Figure 8 : Graph of testing needs over time

We can clearly see that the testing need is accumulative and grows faster as time goes by. But most of the time, tests have routine scenarios that should always be done to assure the working of an app, like testing the inputs of a form, or the functioning of links and buttons,

and these routine scenarios, make the company waste a huge amount of time, and we have to find a solution for that. That's where the need of automating the testing phases pops up.

If the company spends four weeks testing their app, they can minimize that to a system that does the routine test in a week, giving them the left three weeks, to do stress testing, and sanity checks, which can significantly reduce the post-production incidents.

Conclusion: Hypothetical case study

Finally to conclude everything we said, we propose the following hypothetical case study to see the cost of quality assurance and the value of revenue in three cases: no testing, manual testing and automatic testing. Suppose our company has a new software release every trimester. The average number of bugs in every release, hypothetically, is 1000 bugs that has to be fixed, 250 of them are found by users. Let's say it costs us 10$ to fix bugs discovered by developers, 100$ to fix bugs found by testers, and 1000$ to fix bugs found by customers.

We devised the following table:

Testing	**No testing**	**Manual testing**	**Automated testing**
Staff	0$	60,000$	60,000$
Infrastructure	0$	10,000$	10,000$
Tools	0$	0$	12,500$
Total investment	**0$**	**70,000$**	**82,500$**
Development phase			
N° of bugs found	250	250	250
Fix cost	2,500$	2,500$	2,500$
Testing phase			
N° of bugs found	0	350	500

Fix cost	0$	35,000$	50,000$
After release			
N° of bugs found	750	400	250
Fix cost	750,000$	400,000$	250,000$
Cost of quality			
Conformance	0$	70,000$	85,500$
Non conformance	752,500$	437,500$	302,500$
Total Cost of quality	**752,500$**	**507,500$**	**385,000$**
Return on investment	**N/A**	**350%**	**445%**

Table 3 : Cost of quality assurance and value of revenue in no testing, manual testing and automatic testing cases

*ROI = (gain - cost)/cost

Conformance costs include everything from quality assurance, investments on tools and training employees. Non-conformance costs, are mainly any unexpected failures internally or externally.

We see that there's an improvement in revenue while comparing between no testing, manual testing and automated testing cases, which proves the benefit of integrating an auto-testing phase in the development process.

3.2. What we want to test

Now that we've established the need and importance of auto-testing, it's worth noting that automation is development itself, and it needs scripting and programming skills too. So before we start, we should first get an idea what matters the most in an app, and speak the language of the programmer and the customer too. And to do that, we tried to make a list of the most important aspects that we want to test in an app, and it goes as the following:

3.2.1. Functionality

This is the most obvious thing we should test in an app. Any tester that takes an app, starts by tapping all the buttons, and testing all the pages of the app, that could contain any component from static contents, to interactive ones, like forms and navigation bars.

The app shouldn't crash during any moment of the test, and under any circumstance or complex scenario.

Besides, it shouldn't hog the memory and CPU time while processing for example huge amounts of data. Some apps don't crash but they consume excessively the phone resources (such as CPU, RAM, battery, etc), which forces the phone to kill its process.

Also, the app should react properly to unexpected external events, such as receiving a text message, dropped Internet connection. We don't want the app to freeze the phone while receiving a phone call, which is the main purpose of a phone in the first place.

3.2.2. GUI Style

This part concerns the disposition of the graphical user interface, accessibility of its elements and the right visibility of them. We would like to test if the app shows up as it should do, in the same way described in its code, respecting the colors and sizes of different elements in the interface.

Also, we would like to have the ability to warn the user of unusual practices in the style of their app, for example if there's a some black text in a black background, the app knows the text is there, but there's no way for the user to see the text, if this anomaly makes it to the public.

3.2.3. Performance

This part is interesting too, for the advanced or even beginner users, they would like to have a report of how much RAM the app consumes during different stages of the app, as well as how much CPU usage it takes, how much battery it consumes and the frame drawing rate.

The client can use all these pieces of information to know what procedures and functions consumes resources the most and try to optimize them for a better user experience.

3.2.4. Security

This part concerns mainly interactions with remote databases and services that the app consumes to get data and feeds. It is pointless developing a highly secure app if there are gaping holes in the servers that store and process customer data; conversely, even if the servers are completely secure, an insecure app could allow customer data to be retrieved or redirected to a remote attacker. Mobile applications which send and receive sensitive information are tempting targets for man-in-the-middle (MITM) attacks where a correctly positioned attacker can view and manipulate traffic.

In our study, we focused mainly on the submitted data through forms to discover loopholes in the app's backend service.

3.3. Test development styles

Before we could move on to implementation of an auto testing tool, we had to, first, understand the testing job, what a test engineer does, and gain a knowledge about the testing job terminology, and for that, we're going to talk briefly about two very famous test development styles which are, Test Driven Development (TDD) and Behavior Driven Development (BDD). These styles were proposed and conceived to help developers organize their code, and have better and more reliable testing scripts.

3.3.1. Test Driven Development

In test driven development, the developer is guided by the tests and writes his code around them, in other words, he writes first the tests, makes assertions about what the functions and methods should do or return, and then he builds his code around those assertions.

We can sum up this definition in these steps:

- First the developer writes some tests.
- The developer then runs those tests and they fail because none of those features are actually implemented.
- Then the developer actually implements those features in code.
- If the developer writes his code well, then in the next stage, he will see his tests pass.
- The developer can then refactor his code, add comments, clean it up, as he wishes because the developer knows that if the new code breaks something, then the tests will alert him by failing.

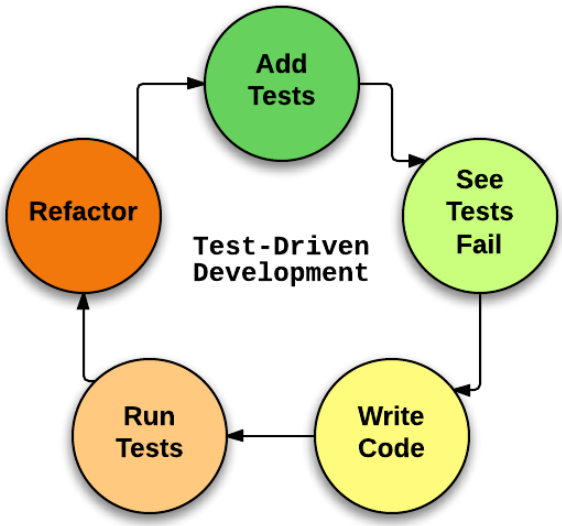


Figure 9 : Test-driven development cycle

This style allows the developer to build a clear, maintainable and testable code, and it allows him to follow the course of development of their app, and by writing tests first, we'll be sure that we'll stop coding the moment our tests pass, which means we'll stop as soon as the features do what they're supposed to do, by that we guarantee that we won't code more than what's actually needed.

It also helps to fix bugs and flaws as soon as our tests fail, so we know exactly our latest changes, and we can easily track what caused the bug, which saves us the trouble of having to recode our features after reaching a moment where maintaining the code is hard.

It's worth mentioning that the tests are accompanied by a verbose and detailed output in the console or in a PDF or HTML page, that describes what features we're testing and what it

does, and whether it failed or passed, that way the developer and project managers can have a clear idea about the progress of their projects.

3.3.2. Behavior Driven Development

It is said that BDD is just TDD done right, but after some deep research, it turns out that it has a fine line of difference with TDD, and it has its own characteristics and advantages, and offers additional value.

The value is largely in the collaborative process for defining requirements and the grammar and structure used for capturing them.

BDD captures requirements as user stories. A story has two parts – a narrative and scenarios. A requirement is not complete until it has both. We write stories in BDD with a rigid structure. We write a narrative using a Role/Benefit/Value grammar and scenarios using a Given/When/Then grammar. Following a rigid structure like this can help adoption of a common vocabulary that leads to unambiguous requirements.

This enables testers and developers to include with them project managers and stakeholders, so that everybody has an idea of the progress of the project.

The crucial value that BDD offers is the process for creating the user stories. User stories are developed collaboratively by the key stakeholders – the Product Manager (or customer or Business Analyst), the QA resource and the developer. This level of collaboration results in:

- Developing features that truly add business value.
- Preventing defects rather than finding defects. Think how much easier it is to write code when you have scenarios that exemplify the positive and negative behaviors and make the intention transparent to the programmer.
- Bring quality assurance involvement to the forefront. This is great for team dynamics because in many agile processes quality assurance personnel feel ignored.
- Define executable, verifiable and unambiguous requirements.

3.4. Testing techniques

After discussing two of the most famous development styles, we're going to explain the different techniques of testing. The three most important techniques that are used for finding errors are: white box, black box, and gray box. We're going to discuss them in details in the next paragraphs, with each one's advantages and disadvantages.

3.4.1. White box testing technique

In white box testing the tester needs to have a look inside the source code and find out which unit of code is behaving inappropriately. In more details, it means we're testing a system with full knowledge and access to all source code and architecture documents, and a clear understanding of the internal working of the system. Having full knowledge about the internal logic and structure of the code, will help in achieving detailed investigations and can reveal bugs and vulnerabilities more quickly than the "trial and error" method of black box testing which we'll discuss later. Additionally, you can be sure to get more complete testing coverage by knowing exactly what to test and how to test it, because when we test while having all the information about the system in question, we can know exactly what to target and how, hence we can write and design more accurate test scenarios, which will reveal errors in hidden code. However, because of the sheer complexity of architectures and volume of source code, white box testing introduces challenges regarding how to best focus the test and analysis efforts, plus it could be expensive and time consuming because we'll need highly skilled testers and time to understand every block of code. The complexity of the architecture could also result in missing or omitting some parts of the code intentionally or unintentionally.

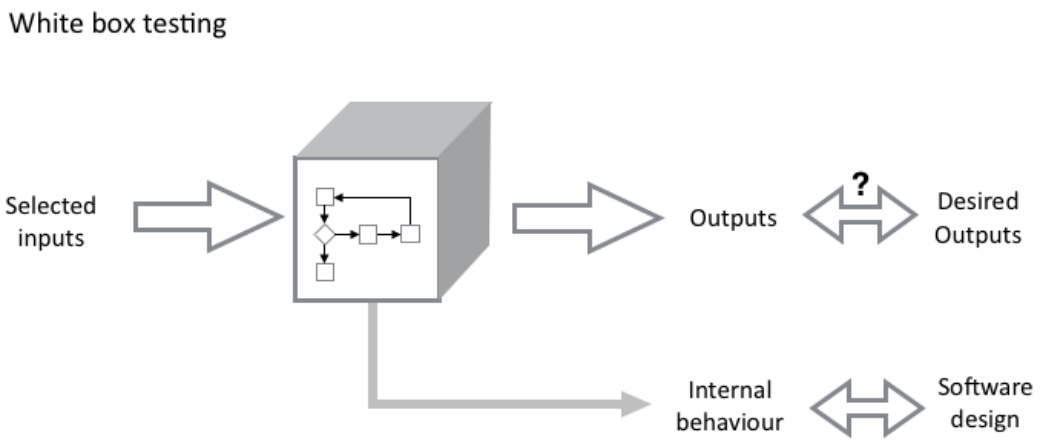


Figure 10 : White box testing logic

3.4.2. Black box testing technique

As its name suggests, Black box testing treats the application as a "Black Box", it is a technique of testing without having any knowledge of the internal working of the application, and it only examines the fundamental aspects of the system. In more details, it refers to testing a system with no specific knowledge to the internal workings of the system, no access to the source code, and no knowledge of the architecture. Essentially, this approach most closely mimics how a user typically approaches your application. It is an efficient technique when it comes to large code segment, and it is less time consuming because the tester's perception is very simple, and we can write test scenarios and develop test cases more quickly. However, due to the lack of internal application knowledge, the uncovering of bugs and/or vulnerabilities can take significantly longer, also, sometimes there's only a limited coverage because only a selected number of test scenarios are actually performed.

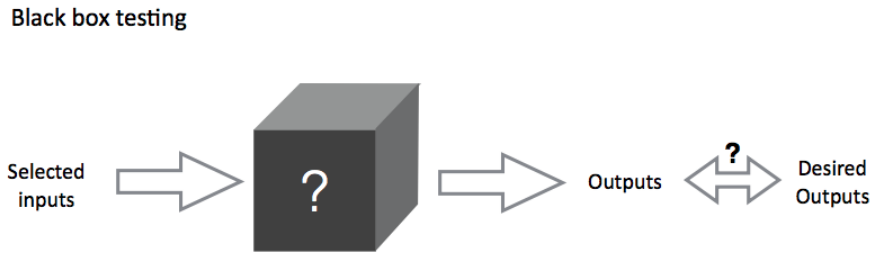


Figure 11 : Black box testing logic

3.4.3. Gray box testing technique

In short terms Gray box is the combination between White box and Black box and tries to leverage the strengths of each, it is a technique to test the application with limited knowledge of the internal working of an application and also has the knowledge of fundamental aspects of the system. This knowledge is usually limited to detailed design documents and architecture diagrams. In gray box testing the codes of modules are studied (white box testing method) for the design of test cases and actual tests are performed in the interfaces exposed (black box testing method). Gray box testing technique will increase the testing coverage by allowing us to focus on all the layers of any complex system through the combination of all existing white box and black box testing. The tester relies on interface definition and functional specification rather than source code, which gives him the possibility to design excellent test scenarios. However, this technique risks the possibility of testing redundancy, if the software designer has already run a test case, also, it is difficult to associate defect identification in distributed applications.

Chapter 4: Implementation of the auto-testing system

This chapter will focus on the experience which we went through during our second task, and also describe the proposed solution from the specification to the integration in the host company's platform.

4.1. Benchmarking of testing platforms

4.1.1. Steps of benchmarking

Using automation tools and testing frameworks yields quantifiable benefits and is recommended. Test automation is done by using software to control the execution of tests and compare actual results to expected results. With test automation we have the possibility to set up test preconditions, to automate test control and test reporting functions, but the question here was, what testing platform can we use?

In this part, we went through the process of benchmarking existing testing platforms to identify which we can take as a base for our automated tests. Dimensions that were measured were of course quality, time and cost.

Benchmarking is a way of discovering what is the best performance being achieved – whether in a particular company, by a competitor or by an entirely different industry. This information can then be used either to identify gaps in an organization's processes in order to achieve a competitive advantage, or to identify similarities and differences between organization's services compared to the cost.

In this process, we studied and compared up to thirty testing platforms that offer approximately similar functionalities that we were targeting, but at different cost, following those steps:

- **Step 1:** Identifying organizational goals and planning. As with any other project, thorough preparation and planning are essential at the outset. In this step, we decided first to take a quick look at the existing testing platforms and the common features that are offered to get an idea about the functionalities that we should be seeking in the testing platform, as well as the bugs that we wanted to test in the mobile applications. Once the needs were established, we moved to the data collection.
- **Step 2:** Data collection. This stage involves first looking for the existing testing platforms online, and gather as much info and reviews about each platform as we could. To better understand each platform features, we tried to contact the

companies by calling and emailing the developers and support teams, so we can make sure of the offered features of each platform as well as the limitations found in the reviews posted online.

- **Step 3:** Data analysis. The key activities here are the validation and normalization of data. The analysis must indicate each platform's strengths and weaknesses based on the needs established in the previous step, to enable comparisons to be made between the platforms.
- **Step 4:** Reporting. The analysis must then be reported in a clear, concise, and easily understood format via an appropriate medium. In this step, we made a little grading system for each platform in which we gave points to both pros and cons of each platform depending on our needs, which would help us choose the suitable testing platform for ScreenDy.
- **Step 5:** Analyzing results and comparing to costs. Data does not equal intelligence. Analyzing the data brings actionable insights and recommendations that can be used to revise future strategy. In this step we were mainly picking the suitable platform for our needs, by comparing the costs of each platform to the features it offers. Once those points have been ascertained, we moved to present the results to the CEO.

After presenting the report and discussing with the CEO, we decided to choose two open source platforms for local testing which are MonkeyTalk and Appium, and two paying platforms for ScreenDy's end users which are Appthwack and SauceLabs.

4.2. Evaluation and tryouts of tools

We started first trying MonkeyTalk with Appthwack. MonkeyTalk is a mobile app testing tool that automates real, functional interactive tests for iOS and Android apps, native, mobile, and hybrid app, real devices or simulators.

We tried MonkeyTalk for a week in which we took a tour on all MT's basic functionalities. We encountered many limitations using this tool, such as problems with taking screenshots especially on iOS, and no commands for getting the App's components tree, which made each MT script specific to one and only one app.

We moved then to try Appthwack. Appthwack is a cloud testing platform that automatically tests Android, iOS, and web apps on 100s of real, non-emulated phones and tablets, providing detailed reports in minutes. Again we faced many limitations with this cloud testing platform such as limited scripting languages (Calabash, MonkeyTalk), and even while using those scripting languages, the results on their devices weren't satisfying. Appthwack has got a feature of "App Explorer". When providing no testing script, Appthwack uses the App Explorer to make a tour on the app and analyze each page, and then return a report containing screenshots of the visited pages as well as some data about performance and errors. Although this feature didn't work as supposed, it gave us an idea for our first testing script.

We decided then to move on to the next platform.

4.3. Selected tools

In this part, we will be introducing the tools and platforms we used to develop the auto-testing system.

4.3.1. Appium

Appium is an open-source tool for automating native, mobile web, and hybrid applications on iOS and Android platforms, that has been around since 2012.

Importantly, Appium is "cross-platform": it allows the user to write tests against multiple platforms (iOS, Android), using the same API. This enables code reuse between iOS and Android test suites.

4.3.1.1. Appium Philosophy

Appium was designed to meet mobile automation needs according to a philosophy outlined by the following four points:

- You shouldn't have to recompile your app or modify it in any way in order to automate it.

- You shouldn't be locked into a specific language or framework to write and run your tests.
- A mobile automation framework shouldn't reinvent the wheel when it comes to automation APIs.
- A mobile automation framework should be open source, in spirit and practice as well as in name.

4.3.1.2. Appium Concepts

- **Client/Server Architecture**

Appium is at its heart a web server that exposes a REST API. It receives connections from a client, listens for commands, executes those commands on a mobile device, and responds with an HTTP response representing the result of the command execution. The fact that it has a client/server architecture opens up a lot of possibilities: we can write our test code in any language that has an http client API, but it is easier to use one of the Appium client libraries. We can put the server on a different machine than our tests are running on. We can write test code and rely on a cloud service like Sauce Labs to receive and interpret the commands.

- **Session**

Automation is always performed in the context of a session. Clients initiate a session with a server in ways specific to each library, but they all end up sending a POST /session request to the server, with a JSON object called the 'desired capabilities' object. At this point the server will start up the automation session and respond with a session ID which is used for sending further commands.

- **Desired Capabilities**

Desired capabilities are a set of keys and values (i.e., a map or hash) sent to the Appium server to tell the server what kind of automation session we're interested in starting up. There are also various capabilities which can modify the behavior of the server during automation. For example, we might set the platformName capability to iOS to tell Appium that we want an iOS session, rather than an Android one. Or we might set the

safariAllowPopups capability to true in order to ensure that, during a Safari automation session, we're allowed to use JavaScript to open up new windows.

- **Appium Server**

Appium is a server written in Node.js. It can be built and installed from source or installed directly from NPM.

- **Appium Clients**

There are client libraries (in Java, Ruby, Python, PHP, JavaScript, and C#) which support Appium's extensions to the WebDriver protocol.

- **Appium.app, Appium.exe**

There exist GUI wrappers around the Appium server that can be downloaded. These come bundled with everything required to run the Appium server. They also come with an Inspector, which enables the user to check out the hierarchy of the app. This can come in handy when writing tests.

4.3.1.3. Test lifecycle on Appium

- **For iOS**

On iOS, Appium proxies command to a UIAutomation script running in Mac Instruments environment. Apple provides this application called ‘instruments’ which is used to do lot activities like profiling, controlling and building iOS apps but it also has an automation component where we can write some commands in javascript which uses UIAutomation APIs to interact with the App UI. Appium utilizes these same libraries to automate iOS Apps.

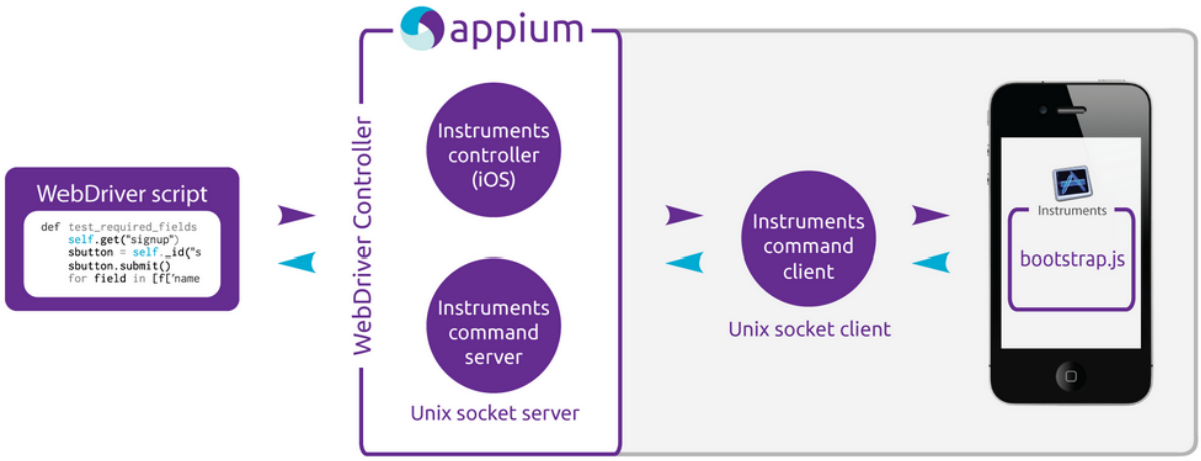


Figure 12 : Test lifecycle on iOS using Appium

In the above figure, we can see the architecture of Appium in context to the iOS automation. If we talk about a command life-cycle, it goes like, Selenium webdriver picks a command from the code like (Element.click) and sends it in form of JSON via HTTP request to the Appium server. Appium server knows the automation context like iOS and Android and sends this command to the Instruments command server which will wait for the Instruments command client (written in node.js) to pick it up and execute it in bootstrap.js within the iOS instruments environment. Once the command is executed the command client sends back the message to the Appium server which logs everything related to the command in its console. This cycle keeps going till the time all the commands gets executed.

- **For Android**

The situation is almost similar in case of Android where Appium proxies commands to a UIAutomator test case running on the device. UIAutomator is Android's native UI automation framework which supports running junit test cases directly into the device from the command line. It uses java as a programming language but Appium will make it run from any of the WebDriver supported languages.

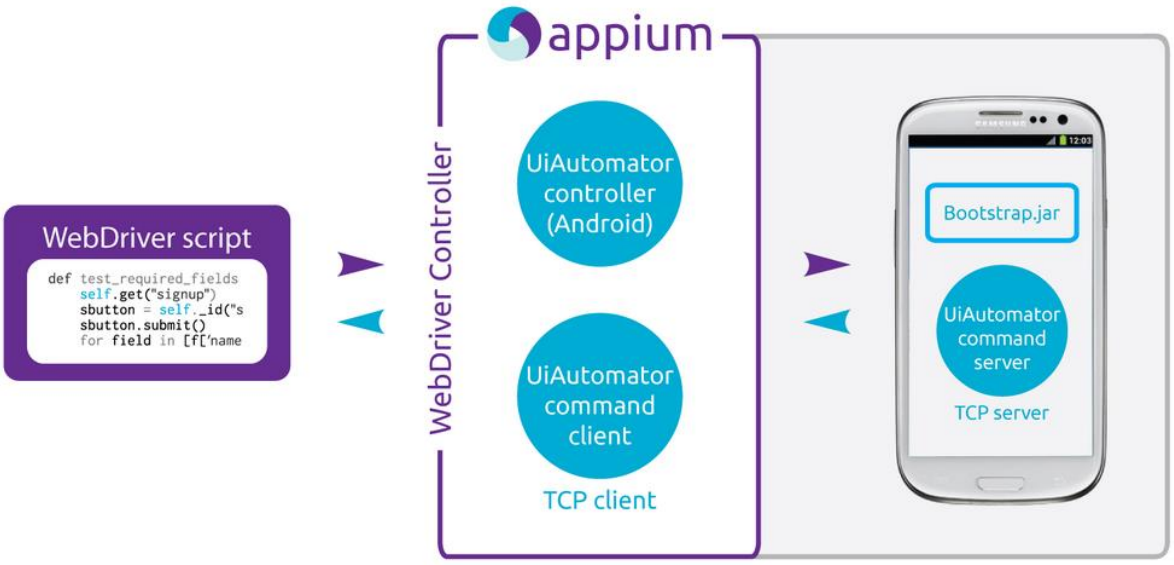


Figure 13 : Test lifecycle on Android using Appium

In the above diagram we can see, here we have Bootstrap.jar in place of bootstrap.js which represents our test case when compiled in java. As soon as it gets launched it spawns a TCP server. Here the TCP server resides inside the device and client is in the Appium process which is just opposite to the way it is in iOS.

If the device has an API level <= 17, Appium installs a Selendroid server that converts the Selenium commands to Android Instrumentation (JUnit) commands.

4.3.2. Sauce Labs

There is a variety of mobile devices with different operating systems on the market, they differ in screen sizes, input methods, hardware capabilities.

We wanted to be able to have a large coverage of those devices as much as we can, so we can test different apps on them, and see how they render on each device and how they behave. Since it'll be so expensive to buy every possible device, we looked for solutions.

Sauce Labs is a cloud-based web and mobile application testing company, based in San Francisco, which helps organizations with Continuous Integration and Delivery, and reduces infrastructure costs for software teams of all sizes.

Founded by Jason Huggins, the creator of Selenium, Sauce Labs lets users run Selenium, Appium and JavaScript unit tests across more than 450 browser and OS platforms at scale without setting up or maintaining dedicated testing infrastructure.

The platform has emerged as one of the major tools in the agile testing ecosystem, growing alongside the adoption of open source standard testing frameworks.

At the core of Sauce Labs is a huge understanding and commitment to security. From the private data center in Mountain View to the “security through purity” architecture of never reusing, sharing or storing data on VMs, Sauce Labs is the most secure, reliable and stable way to run all tests cross-browser in the cloud.

A big supporter of the open source community, Sauce Labs is behind projects such as Appium, Monocle and Selenium. They also offer free accounts to any open source project and provide the testing infrastructure for more than 800 OSS projects, including the Selenium Project, Mozilla, AngularJS, jQuery and many others.

Whether running automated tests for web or mobile apps, JavaScript unit tests, or manual tests, users get test results with screenshots, videos, and log files to help them find bugs faster and release higher quality software more quickly.

4.3.3. NodeJS

Working with Appium included choosing a language for the tests scripting. Appium offers many languages for scripting, as mentioned before, such as Java, Ruby, Python, C# and others. We chose to use NodeJS since ScreenDy users would be more familiar with javascript and web programing than other languages.

Node.js is an open source, cross-platform runtime environment for server-side and networking applications. Node.js applications are written in JavaScript, and can be run within the Node.js runtime on OS X, Microsoft Windows, Linux, FreeBSD, NonStop and IBM i.

Node.js provides an event-driven architecture and a non-blocking I/O API that optimizes an application's throughput and scalability. These technologies are commonly used for real-time web applications.

Node.js allows the creation of web servers and networking tools, using JavaScript and a collection of "modules" that handle various core functionality. Modules handle File system I/O, Networking (HTTP, TCP, UDP, DNS, or TLS/SSL), Binary data (buffers), Cryptography functions, Data streams, and other core functions. Node's modules have a simple and elegant API, reducing the complexity of writing server applications.

Node.js is primarily used to build network programs such as web servers, making it similar to PHP and Python. The biggest difference between Node.js and synchronous languages is that the synchronous languages are a blocking language (commands execute only after the previous command has completed), while Node.js is a non-blocking language (commands execute in parallel, and use callbacks to signal completion).

Node.js brings event-driven programming to web servers, enabling development of fast web servers in JavaScript. Developers can create highly scalable servers without using threading, by using a simplified model of event-driven programming that uses callbacks to signal the completion of a task. Node.js was created because concurrency is difficult in many server-side programming languages, and often leads to poor performance. Node.js connects the ease of a scripting language (JavaScript) with the power of Unix network programming.

Thousands of open-source libraries have been built for Node.js, and can be downloaded for free from the npm website.

4.4. Solution and execution

After deciding the most suitable tools to use for our auto-testing system, we moved then to implement it for both iOS and Android platforms.

We chose to follow the BDD test development style, because we want the tests to express easily the scripts' goals, to be simple and understandable by end users, and to be able to show development progress.

We moved then to analyze ScreenDy's architecture and components, which ended in deciding two types of testing scripts.

4.4.1. Specific test

The specific test is based on the gray box technique; we have little knowledge about the app's structure, and we try to tap links and buttons, type in inputs, scroll to elements… Basically we want to create a test scenario based on the app's structure and the used components to make sure the functionalities are preserved as the development (locally) or app's creation (for the platform's end users) progresses.

To achieve that, we used a javascript testing framework named mocha, which runs on node.js, features browser support, asynchronous testing, test coverage reports, and use of any assertion library. Mocha provides a simple way to structure the test's scenario as steps, and then provides each step's results, on which we based our tests' reports.

We went further than that, and created some test scenarios specific to components (ScreenDy plugins) as modules that can be imported in specific tests.

Finally, and to make the scripting phase simple for end users, we created a pseudo-language for creating test scenarios. The scenario is then compiled and executed, and finally the results are shown in a form of report showing each step's name, duration, screenshots before and after executing the step and its status (failed or succeeded). We attached some screenshots of the specific test report in the appendices.

4.4.2. Generic test

The idea of this test came from Appthwack's AppExplorer feature. The feature didn't work at all on ScreenDy's mobile apps, so we decided to make an app crawler that is generic to all mobile apps made by ScreenDy.

This test is based on the black box technique, which means we have no information about the app, nor about its structure and components, and we try to tap all clickable elements so that we can visit all app's pages from the Home page through all the subpages, and then, as a result, we show the hierarchy of the app as a tree starting from the splash screen, as well as the loading duration for each page.

We chose to use chaining technique instead of using mocha in our generic test, mainly because mocha's logic is to initiate all variables and all steps at the beginning of the test, so for example if we want to loop over an array, the array shouldn't change during the iterations, which is a real limitation for our generic test since we have no knowledge of the app's structure nor its components that we're going to interact with at the very beginning of the test.

In this test, we tried to include as many plugins as we can, which took us a little bit longer than predicted, due to the complexity of some plugins, as well as limitations and problems with the testing platforms.

We also added an option for entering login credentials in case the app contains an authentication page so the test won't stop at this level, but would enter the login and password and continue crawling the app.

This test was made mainly for three reasons:

- First, to give an idea of possible crashes or dysfunctionalities existing in the app. In other words, this test can replace the manual testing of the first use after the app is created. This test can reveal problems in case of some integration problem (forgetting to specify a button's destination page link for example).
- Second, to display the same app on different devices, and without having to write a specific test scenario. We can use this script, with a single click, to visualize all the app's pages in different emulators, and verify if the app displays correctly on various devices and screen sizes.
- Third, to get an idea about the loading time of each page, starting the splashscreen, to the home page to all subpages.

4.4.3. Challenges / Problems / Solutions

During the development of our proposed solution, we faced a lot of challenges and problems, of which we succeeded to get over. Some of them are due to technical issues with

the services, others are due to lack of documentation and the fact that we are new to some technologies. In the following points we're going to talk about some of them.

4.4.3.1. Asynchronicity

During the implementation phase, we discovered that the architecture of Node.JS is asynchronous, which means we had to change our thinking paradigm and get used to the asynchronous behavior.

Contrarily to the synchronous behavior, asynchronous events take action at the same time, and return a result only when they're ready; which means if in the code, action A comes before action B, we might get the result of action B first, in fact we have no guarantee which one's going to return its results first, and this behavior gets more complicated when we have several more events to handle, and it gets more complicated when one action depends on the results of another one.

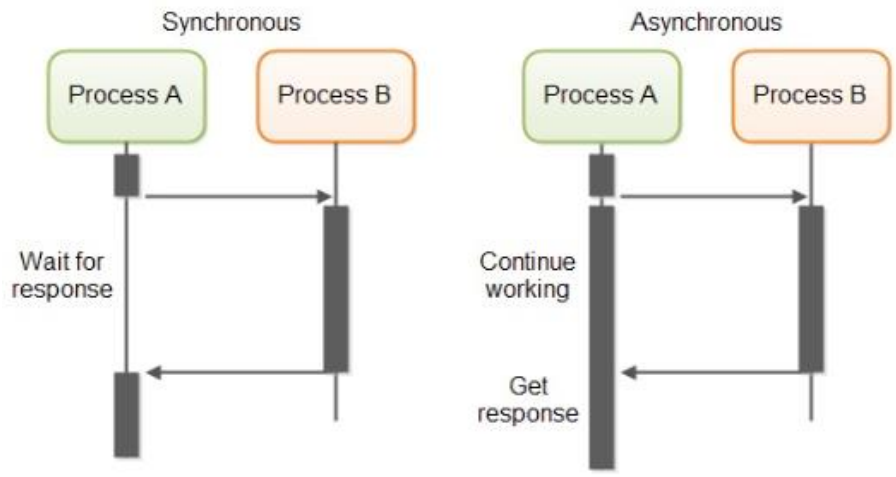


Figure 14 : Difference between synchronous and asynchronous behavior

The challenge we faced here is that, while interacting with different components of any app, we often need to take the next action depending on the result of the previous one, for example we can't decide which button to click, until we investigate the components present in the current page.

The solution here was to use some functions, which are passed as parameters to those actions; these functions are named callbacks, they are called right after the result of the asynchronous action is ready to be consumed. We tried to manage them the right way to accomplish the task.

The other problem we got here is that there are some independent actions, that need to be executed in sequence, and some of them take time, so we need to make the script wait for some actions before executing others; for example, we need to wait for the app to be uploaded before we can start the test, or we need to read the contents of some file to extract some parameters before using them. Because of the asynchronous behavior, this was somewhat hard to accomplish. To remedy this problem, we found two solutions that we used carefully depending on the situation.

- The first solution is by using a NodeJS object called a “promise”; promises enabled us to postpone or defer the execution of some actions until we were ready to.
- The second solution is by using a technique called “chaining”. Chaining events made it possible for us to execute them sequentially while preserving the asynchronous behavior, which simplified for us interactions with the app.

4.4.3.2. Lack of documentation

Documentation and tutorials are the most helpful thing if not important, when starting to learn or use a new technology, so it was a serious setback when we couldn’t find good indexed documentation about the tools we were using, and concepts we were trying to understand. It took us so long to finally get some of it.

The action we took to get over this challenge is by trying to read the source code of some of the tools we were using, and then try to understand it, until we start to get the right keywords to look for, which helped us a lot to find pieces of documentation on the web.

The other thing that held us back is that, even when we find documentation, sometimes it’s not well explained, incomplete or ambiguous. Then the only thing we could do is test the parameters of functions, and improvise according to the results, until we got how everything works.

4.4.3.3. Technical issues with the Cloud service

While working with SauceLabs, we encountered a lot of technical issues that held us back on the development, some of them originated from our scripting structure, some of them were purely technical issues in their service. We usually keep contacting them by email until they fix their problem or help us fix ours.

4.5. Implementation in the ScreenDy platform

As we explained earlier, ScreenDy is a platform that helps people develop mobile apps, in a drag-and-drop interface, using multiple plugins; and our job inside the company, was to develop an auto-testing system, that would run tests on the app, and return to the end-user a report of the functioning of the app with screenshots and durations of loading of each page. Now that we finished developing the first versions of the scripts, we should build an interface in the platform, and the best user experience we can give, is to run everything in just one click, and then expect the report of the test in a seamless abstracted way.

To make that possible, we had to develop a RESTful API in the backend, which would be handled by a server in ScreenDy's VPS, this server will wait for requests, which will come with different parameters:

- The first parameter is whether to test the iOS version or Android version of the App;
- The second parameter precises which user is requesting the test;
- The third parameter indicates what app should we test;
- The fourth parameter tells us on what devices the user wants us to test his app on;
- And the fifth parameter would be the OS version on which we should test the app on.

When the server receives a request, a chain of events gets triggered, and we will explain this chain in the following scenario

- The server will build the requested app to generate the IPA/APK for iPhone/Android testing;
- After that, it'll upload the generated file into the testing platform we chose, which is SauceLabs for now;
- When the upload is done, the server will save a new record in the database holding the id of the test, and the id of the app, with its status indicating that the test is running, and save all the required parameters if there are any;
- The server will start the test, using the desired OS and device that the user requested
- After the test is done, the server will update the database record status into finished, and it will notify the frontend that the test is done, so it'll grab the results from the database.

Obviously the development of the backend goes hand in hand with developing the frontend, and we should put extra attention to it, because that's what the end-user will be seeing and dealing with.

And to put the whole system to work, we developed a page dedicated for the auto-testing in the platform's frontend, and for that we added a single button, that starts the test, and a page in each app's project, that will show the test results.

Before starting the test, the user should precise different parameters for his test, first he should choose whether to run the test in Android, or iOS, or both, then he would choose the device on which we're going to run the test, he'll indicate the device name and the OS version of the desired device, then when he clicks "start the test", our frontend script will prepare a set of requests to send to our server using our RESTful API, which will trigger the chain of events we explained earlier, and then when the testing starts, the front-end will establish HTTP connections to wait for the response. When the tests finish, the server will notify the platform, and then it'll grab the response for parsing, and viewing the results in a user-friendly report.

We devised this system, in this exact scenario, so that we can handle the case when users have unstable internet connections, or if the user refreshes the page for some reason, all we

have to do, is fetch the running tests from the database, and open new connections listening for the results, without having to communicate with the server that starts new sessions of tests. All of this was possible thanks to WebSockets. We attached some screenshots of the API page in the appendices.

General conclusion

The need for quality assurance is a crucial thing for any kind of business, it is how we gain the trust of consumers in the products and how we maintain it. In this report we discussed how testing can help a lot in this matter. We can define app testing as a process that verifies and evaluates the correctness of the behavior of some app, along with its stability. It enables us to detect software defects in an early stage of the development of the app, making it cheaper for us to fix the bug, and prevent it from reaching the public. Generally the earlier a bug is found, the cheaper it is to fix.

But testing alone can be time consuming, especially that there are some intuitive common test scenarios that are repeated in almost every testing task, for this reason, we introduced automated testing which would optimize the time needed for routine testing, as a result, there will be more free time available for more sophisticated and creative test cases, and it will also reduce the costs of quality assurance for the company.

Automating the test phases is practical, however, it should not make us give up on manual testing, and beta testing. No matter how smart a script can be, the human factor in discovering unusual scenarios is still very important in quality assurance, and can cover a bigger surface.

Webography

- **Mobile Apps Are Fueling Huge Growth In Time Spent On Digital Media** by MARK HOELZEL
  http://www.businessinsider.com/mobile-apps-fuel-huge-growth-in-time-spent-on-digital-2014-9
- **Scrum (software development)**
  http://en.wikipedia.org/wiki/Scrum_(software_development)
- **Sorting out black box, white box and gray box software testing methods**
  http://searchsoftwarequality.techtarget.com/answer/Sorting-out-black-box-white-box-and-gray-box-software-testing-methods
- **Introduction to Appium's Philosophy, Design and Concepts**
  http://appium.io/introduction.html
- **Sauce Labs Company details**
  https://www.crunchbase.com/organization/sauce-labs
- **Node.JS**
  http://en.wikipedia.org/wiki/Node.js
- **APPIUM: A CROSS-BROWSER MOBILE AUTOMATION TOOL** by AMIT RAWAT
  http://www.3pillarglobal.com/insights/appium-a-cross-browser-mobile-automation-tool
- **Number of apps available in leading app stores as of May 2015**
  http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/
- **App Store Sales Top $10 Billion in 2013**
  http://www.apple.com/pr/library/2014/01/07App-Store-Sales-Top-10-Billion-in-2013.html

- **Number of available apps in the Apple App Store from July 2008 to January 2015**
  http://www.statista.com/statistics/263795/number-of-available-apps-in-the-apple-app-store/
- **Number of Android applications**
  http://www.appbrain.com/stats/number-of-android-apps
- **Mobile application testing**
  http://en.wikipedia.org/wiki/Mobile_application_testing
- **What is the cost of defects in software testing?**
  http://istqbexamcertification.com/what-is-the-cost-of-defects-in-software-testing/
- **How to lose 172,222$ a second for 45 minutes**
  http://pythonsweetness.tumblr.com/post/64740079543/how-to-lose-172-222-a-second-for-45-minutes
- **The Man Who Saved the World by Doing ... Nothing** by Tony Long
  http://archive.wired.com/science/discoveries/news/2007/09/dayintech_0926

Bibliography

- "Testing Criteria for Android Applications version 1.4: February 2013" **by App Quality Alliance (AQuA).**

  Found at this link : http://www.appqualityalliance.org/files/AQuA_performance_testing_criteria_FINAL_july_7_2014.pdf

- "Mobile Application Development Report 2014" **by AGC Partners.**

  Found at this link : http://agcpartners.com/content/uploads/2014/03/AGC-Mobile-Application-Development-Report.pdf

- "Real Challenges in Mobile App" **by Mona Erfani Joorabchi, Ali Mesbah, Philippe Kruchten, University of British Columbia Vancouver, BC, Canada Development**

  Found at this link : http://www.ece.ubc.ca/~amesbah/docs/mona-esem13.pdf

- "Importance of Software Documentation" **by Noela Jemutai Kipyegen and William P. K. Korir, Department of Computer Science, Egerton University Njoro, Kenya**

  Found at this link : http://ijcsi.org/papers/IJCSI-10-5-1-223-228.pdf

- "A Comparative Study of White Box, Black Box and Grey Box Testing Techniques" **by Mohd. Ehmer Khan, Department of Computer Science, Singhania University, Jhunjhunu, Rajasthan, India, and Farmeena Khan Department of Computer Science, EIILM University, Jorethang, Sikkim, India**

  Found at this link : http://thesai.org/Downloads/Volume3No6/Paper%203-A%20Comparative%20Study%20of%20White%20Box,%20Black%20Box%20and%20Grey%20Box%20Testing%20Techniques.pdf

- "Mobile Application Testing" **by Melissa D. Mulikita, Fachhochschule Köln University of Applied Sciences Cologne 07 Fakultät für Informations-,Medien-und Elektrotechnik Studiengang: Master Technische Informatik**

  Found at this link : http://www.nt.fh-koeln.de/fachgebiete/inf/nissen/sif/Mulikita_Bericht.pdf

- "Economics of Software Verification" **by Gerard J. Holzmann**

  Found at this link : http://spinroot.com/gerard/pdf/paste01.pdf

- "Investing in Software Testing: The Cost of Software Quality" **by Rex Black**

  Found at this link : http://www.compaid.com/caiinternet/ezine/cost_of_quality_1.pdf

- "Automated testing ROI: fact or fiction?" **by Paul Grossman**

  Found at this link : http://www.skyitgroup.com/pub/files/AutomatedtestingROI.pdf

Appendices

1. Benchmarking of existing test platforms

1.1. What we're looking for (Ordered by priority)

- Devices on the cloud
- RESTful API
- Rich scripting language (Try to avoid drag and drop)
- Test based on Native objects properties
- Test based on OCR
- Test based on image comparison
- Detailed reports with screenshots
- Events recorder

1.2. What bugs we want to/can test (Ordered by importance)

- Crashes
- Does the page display?
- If yes, does it display correctly?
- If yes, how long does it take?
- Functioning of objects (Taps, Gestures, Swipes…)
- Little bugs in Style (Black text on black background…)
- Speed of loading (SplashScreen, pages)
- Battery/Memory/CPU usage
- Take a tour in the App and take screenshots (basic)
- Calculate the timing cumulated
- Validations (something that should generate an error message, does it show the message?)
- How does the app behave on interruptions? (A call, a message, low battery...)
- Get a screencast of the whole test – video

1.3. List of platforms and tools we tried

- Appium, eggplant, PerfectoMobile, AppThwack, Appcelerator, TestObject, TestDroid, SauceLabs, Xamarin, SOASta, Bugs Buster, SeeTestAutomation, MonkeyTalk, TestingBot, Telerik, TestComplete, Ranorex, Tellurium, Borland Silk, Parasoft SOATest, Applause, Appurify, Device Connect, Titanium, NativeDriver, Calabash, FrogLogic, GridLastic, CloudBees, Solano, TouchTest by SOASTA…

1.3.1. CloudMonkey

Pros	Cons
• The tool is open source and downloadable for free. • The scripting language is easy and natural, convertible to JavaScript • Can record actions live from the phone and generate a script of the actions. • Generates HTML report with tests passed and tests failed with screenshots for fails • Can verify test success by image comparison. • They provide Extendable JavaScript APIs but in a limited manner (e.g unexploitable printed diagnostics) • Can catch a failed test in an exception • Can get the components tree of a view in JSON format, but only printed in the console, not exploitable in a variable. • Can detect the html objects inside a WebView using xpath	• They don’t have cloud platform with real devices, we have to test on our devices or on simulators. • Image recognition is done pixel by pixel, which is not good, because slight change in the clock or network signal could fail the test (false alarms). • Problems with taking screenshots for real iPhone devices in iOS8 (works only on the simulator, not on a real apple device) • We have to add a library to the source code. • Can’t test on iPhones using Cable, we have to test over Wifi, (no wifi, no testing) which can make it slow. • No videos reporting • Built in functionalities are kind of limited

1.3.2. Experitest

Pros	Cons
• Has Object properties detection • OCR detection • Image recognition • Web objects detection • Works on real devices using cable • Can record actions and convert them into scripts (JUnit, Python, Ruby, NUnit...) • No need to modify or add anything to the code source. • Can have full control on any app and the whole phone.	• Doesn't have real devices on their cloud, you have to set up your own VPN on the company with your devices.

1.3.3. AppThwack

Pros	Cons
• Has over 250 Devices for iOS and android • Can execute tests written with MonkeyTalk and Calabash. • Just upload IPA or APK or APP folder • Provides a restful API • Generates full reports with performance diagnosis and screenshots and passed tests • No apparent image recognition or OCR supported • Can use MonkeyTalk's image comparison	• They don't provide their own interface or tool to script or see tests in real time • Can't find image comparison using Calabash • Problems when uploading monkeytalk scripts

1.3.4. TestDroid

Pros	Cons
• 300+ real devices on their cloud • Supports scripts written in Calabash and Appium • Provides reports on performances and passed tests and screenshots	• They don't provide their own tool or interface for scripting and real time monitoring.

1.3.5. Telerik mobile testing

Pros	Cons
• It gives a variety of devices for testing • We can specify a list of devices we want to run the test on • Uses javascipt for writing tests • Gets screenshots of the test result	• Not for free • The cloud testing is still on beta, and only tests applications made by the platform (http://www.telerik.com/mobile-testing/device-cloud)

1.3.6. Ranorex

Pros	Cons
• Can run tests on multiple devices using wifi or usb • Simple, records the test and then reproduce it, no script writing is needed • We can use the same test record on ios as on Android • Supports image based validation	• Not for free • Run tests only on user's device (no cloud testing included)

1.3.7. TestComplete

Pros	Cons
• Includes image based testing + can compare screenshots of the mobile test screen saved during recording and playback • Generates detailed reports with screenshots by the end of the test • We can choose among many scripting languages to write the test (vbscript, jscript, c++script, c# script and delphiscript) • Can record test scenarios at object level • Can flexibly record a library of multi-touch gestures directly from devices (swipe, pinch, long touch, drag and drop, scroll...)	• Not for free (although gives a trial of 30days) • Doesn't support all versions of ios (only ios 8 included) • .exe file (only used on windows platform) • Cloud testing is on devices that we program (not devices on the end-user that we can use)

1.3.8. SauceLabs

Pros	Cons
• Includes cloud testing • Supports many scripting languages that can be used to write tests (ruby, java, python, php...) • Provides a full report by the end of the test, that includes a video and screenshots • Provides an api that we can use within the platform	• Uses virtual machines for the tests on android • Not for free • Doesn't support image based tests

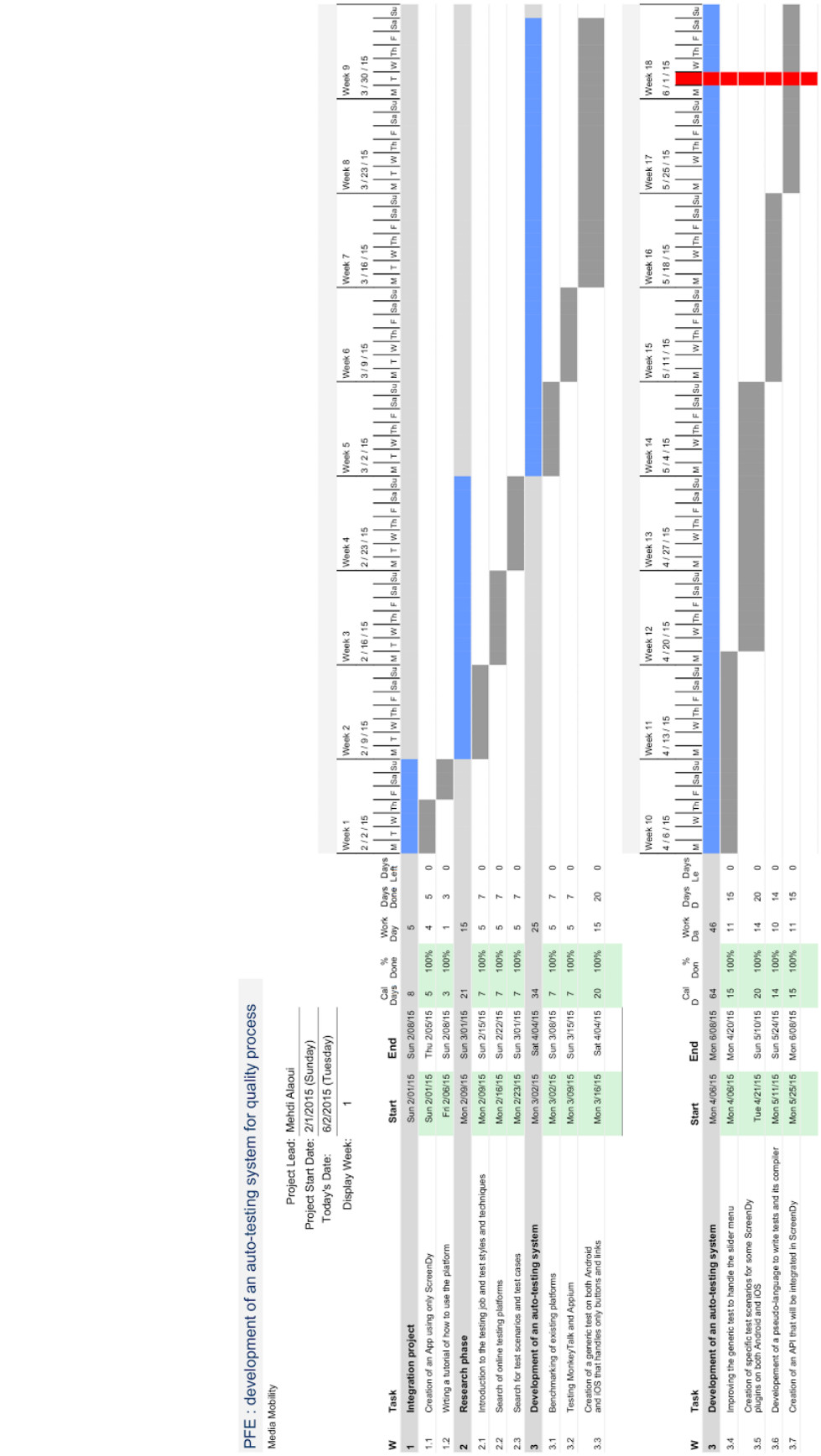
1.3.9. TouchTest

Pros	Cons
• Includes a platform of many devices that we can use in testing (both for Android and ios) • Create a single test for both platforms (not always) • Gives really good report details with battery and memory usage with every single event	• Not for free (unless we want to use our own devices) • No scripting language • Only works on the cloud

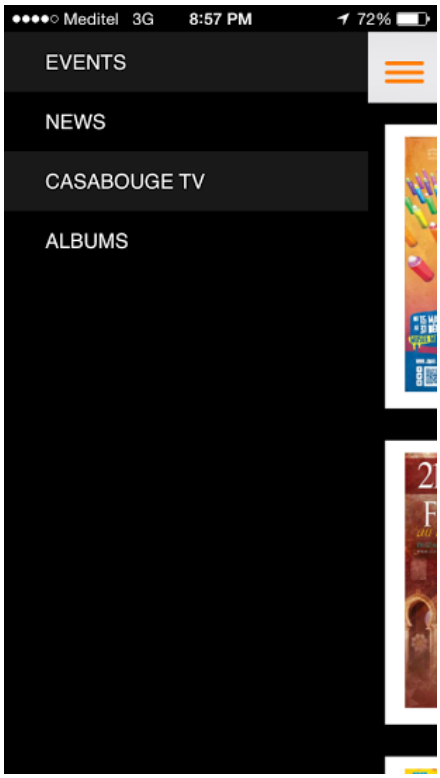
1.3.10. Appium

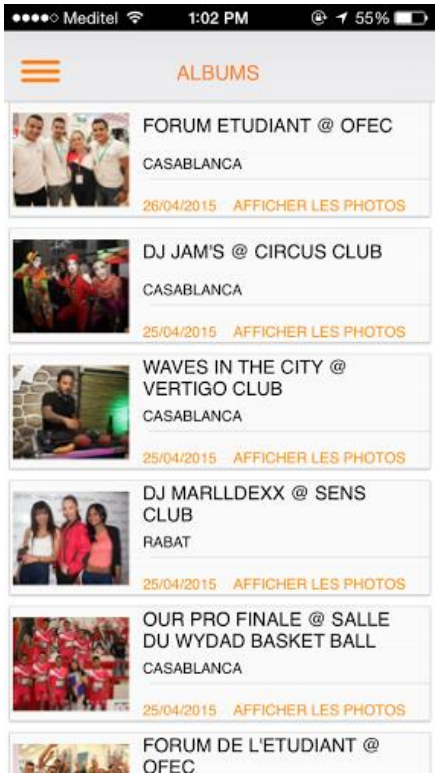
Pros	Cons
• Can control the app without changing the code source or adding any libraries • Can detect native objects easily, and accessing them using xpath • Can connect to cloud services using restful apis • Can record and replay tests, and generate Java, Javascript, Ruby, Python and other languages code • Can extend its functionalities	• Very limited in windows • Can only test one device at a time in the cloud • Can only support android 4.2+

2. Gantt Chart



3. Screenshots of CasaBouge – integration project app




Meditel 1:02 PM 55%
ALBUMS
FORUM ETUDIANT @ OFEC
CASABLANCA
26/04/2015 AFFICHER LES PHOTOS
DJ JAM'S @ CIRCUS CLUB
CASABLANCA
25/04/2015 AFFICHER LES PHOTOS
WAVES IN THE CITY @ VERTIGO CLUB
CASABLANCA
25/04/2015 AFFICHER LES PHOTOS
DJ MARLLDEXX @ SENS CLUB
RABAT
25/04/2015 AFFICHER LES PHOTOS
OUR PRO FINALE @ SALLE DU WYDAD BASKET BALL
CASABLANCA
25/04/2015 AFFICHER LES PHOTOS
FORUM DE L'ETUDIANT @ OFEC


Meditel 1:02 PM 55%
ALBUMS
ESCA
Retour

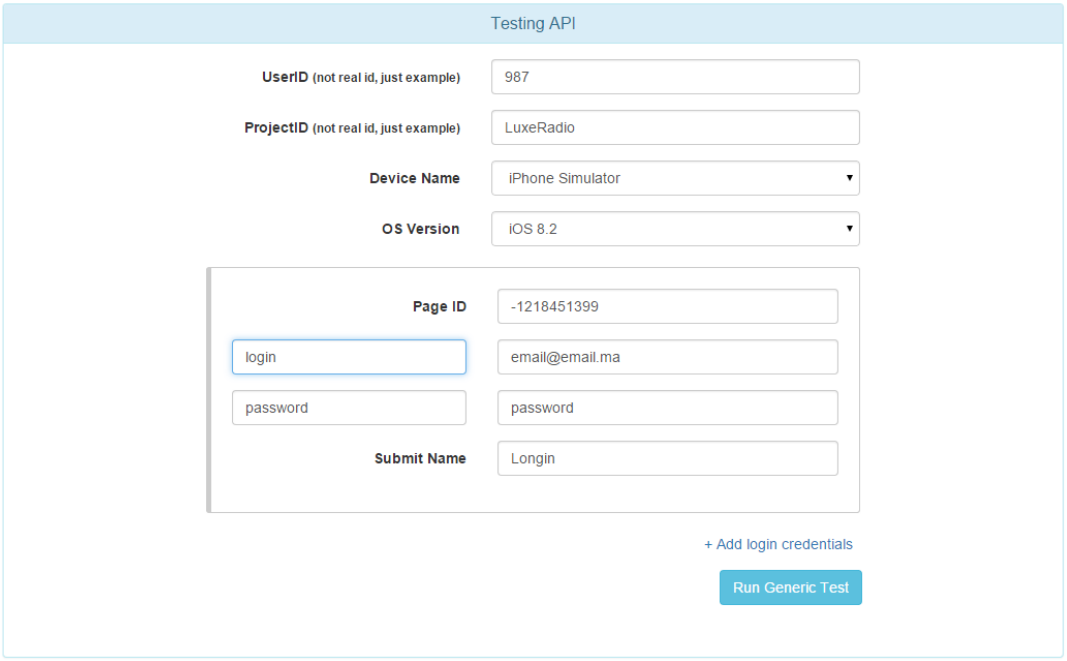

Collège LaSalle
1 / 38

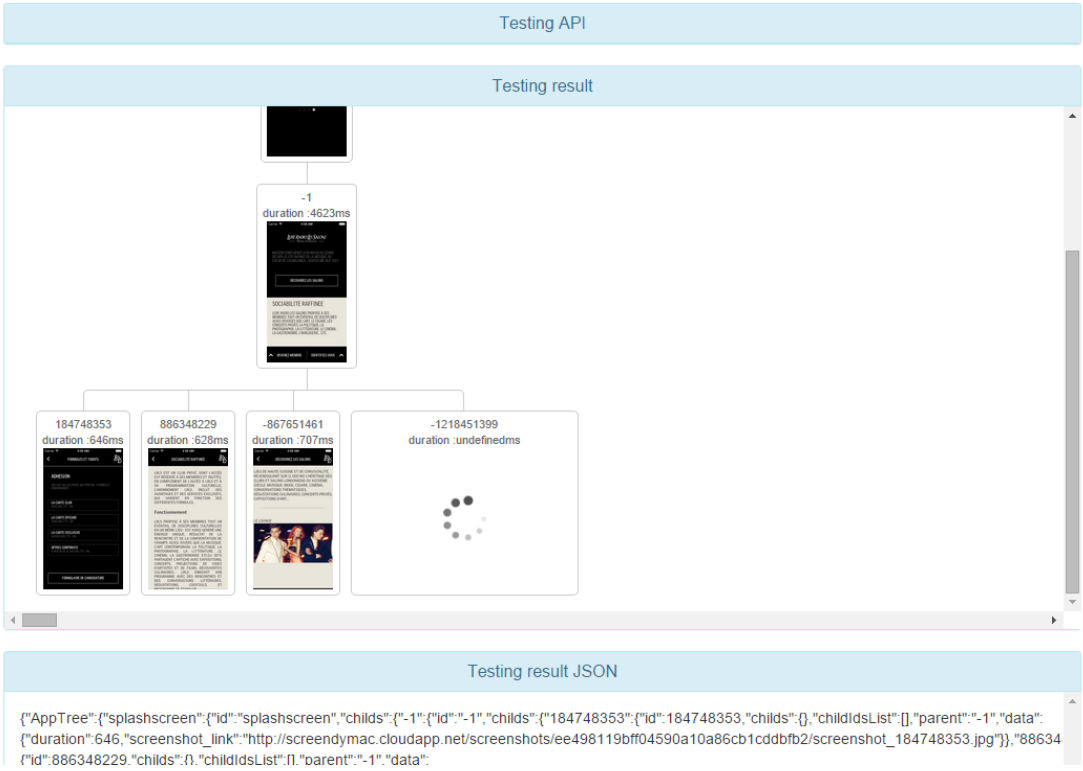

Meditel 1:00 PM 56%
EVENTS
MARRAKECH DU RIRE
MARRAKECH
PALAIS BADII
DU MERCREDI. 10 JUI
AU MERCREDI. 14 JUI
ELIE SEMOUN @ MARRAKECH DU RIRE
MARRAKECH
INSTITUT FRANCAIS
07:30 pm
ELIE SEMOUN
JEUDI 11 JUIN 19H30
JEUDI
11 JUI

4. Screenshots of the testing API

The API's user should specify four parameters in order to run a test:

- The project ID which means the app he wants to test;
- The device name;
- The OS version on this device;
- The logins credentials in case the app contains an authentication page that the user wants to test.


Testing API
UserID (not real id, just example)
987
ProjectID (not real id, just example)
LuxeRadio
Device Name
iPhone Simulator
OS Version
iOS 8.2
Page ID
-1218451399
login
email@email.ma
password
password
Submit Name
Longin
+ Add login credentials
Run Generic Test
Testing result


Testing API
Testing result
-1
duration :4623ms
184748353
duration :646ms
886348229
duration :628ms
-867651461
duration :707ms
-1218451399
duration :undefinedms
Testing result JSON
{"AppTree":{"splashscreen":{"id":"splashscreen","childs":{"-1":{"id":"-1","childs":{"184748353":{"id":184748353,"childs":{},"childIdsList":[],"parent":"-1","data":
{"duration":646,"screenshot_link":"http://screendymac.cloudapp.net/screenshots/ee498119bff04590a10a86cb1cddbfb2/screenshot_184748353.jpg"}},"88634
{"id":886348229,"childs":{},"childIdsList":[],"parent":"-1","data":

5. Example of a specific test

As explained earlier, to run a specific test, the user would have to write his own test scenario using the pseudo-language that we created. An example of a test script would be as the following:

Tap DevenirMembre

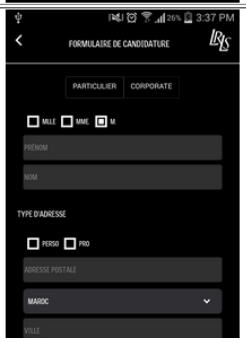
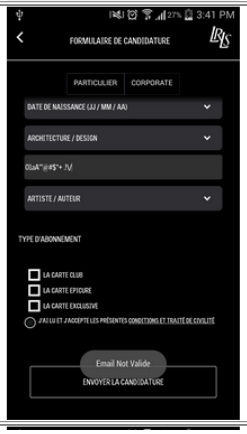
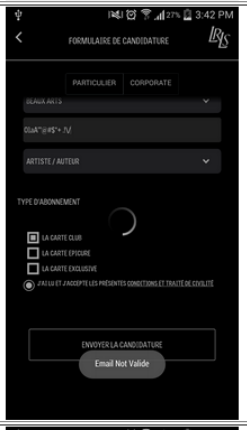
Tap FormulaireCandidature

Test form

This test would start the app, and then once in the home page it's going to look for the button "DevenirMembre" and then tap it, and same thing for the second button. And at last it would run the form test module.

Once the test is finished we get a report describing each step of the scenario in details, as the following:

Report of Testing form Android4.4

	Step	Duration (ms)	Step status	Screenshot before	Screenshot after
✔	**Should tap registration button**	**361ms**	**Passed**		
✔	**Should tap the form button**	**295ms**	**Passed**		
✔	**should put all inputs to a string containing special characters**	**102465ms**	**Passed**		
✔	**should put all inputs to a string containing special characters, and check random values in spinners and radiogroups**	**94278ms**	**Passed**		

Index