Mobile Automation Testing Framework and QA Dashboard

Isaac Troy Robles Atienza

***Abstract*— This project has two parts. The first is on the evaluation and implementation of an automation testing framework for a mobile application, Onsite, and the second is on the design and creation of a dashboard for the Quality Assurance (QA) team at Valocity. Onsite is a mobile application for property valuations in India with a focus on alleviating its complex, inaccurate, or missing property addressing. The introduction of the automation testing framework for Onsite minimizes potential erroneous code leaks to production and reduces the time and cost of pushing to production. The final solution has proven to be sufficient as it can produce the functionalities of the application in the testing scripts efficiently while giving complete control over the scripts to the programmer. However, divergence occurred from the original goal after Onsite faced major User Interface (UI) changes that would invalidate the efforts of automation, resulting in a shift of focus. As for the second part of the project, the QA Dashboard was implemented to provide an overview of the automation status of the different projects and squads within Valocity. The final solution for the dashboard proves to be more than satisfactory as per stakeholder and end-user feedback. Specifically, the managers can utilize the dashboard during stakeholder sessions to show the overall automation status, while the QA team can determine priorities for the upcoming sprints based on areas lacking manual or automated testing.**

***Index Terms*—Automation, Data Visualization, Database System, Design, Mobile Application, Restful API, Software Development.**

I. INTRODUCTION

INDIA has been facing house-addressing issues that make it difficult to distinguish and recognize properties [1, 2, 3, 4]. These house-addressing issues are costly and prohibit inhabitants of households without a legitimate address from accessing bank accounts and receiving mail at home. Without actual addresses, households are forced to be distinguished by their surrounding landmarks and the description of the households' occupants. This makes it difficult to locate and determine the target destination if the weather is suboptimal or if the finder is not previously acquainted with the household occupants. Additionally, this house-addressing issue persists and affects the property valuation process because the valuers are highly unlikely to be acquainted with the owners or previous owners of the property. Therefore, to minimize this issue, Valocity has developed Onsite, a mobile application that adds verbose visualizations and detailing for property valuations.

Onsite is a Flutter-based mobile application that is designed to be utilized in situ. Here, valuers can take snapshots and make detailed descriptions during their valuations to avoid any potential addressing confusion. However, although Onsite has been released for about two years, there has been 0% automation coverage prior to this project. This lack of test automation can exponentially slow down both developers and manual testers with every release and it can introduce potential regression in the application. Therefore, to overcome the uncertainty in releases and streamline the application's development, Onsite's test automation framework project was established, where common frameworks and tools were evaluated, and then implemented.

Goal 10, Reduce Inequality Within and Among Countries [5], is directly tied to Onsite's development and quality assurance. This is because Onsite enables countries that have a lack of house addresses, like India, to perform property valuations similar to other countries without the need for proper addressing to distinguish properties. Furthermore, inequality within the country of India is also minimized as OnSite's valuation process enables property valuations of slum-like properties and that of the higher-end alike, with or without addresses.

The final solution for Onsite's automation testing framework includes the utilization of WebdriverIO, Appium, Android Studio, and BrowserStack. Here, Appium provides a range of device-like functions like shake, rotate, etc, and enables the execution of the test scripts written in the WebdriverIO on a locally emulated device. On the other hand, BrowserStack enables the execution of the test scripts on a multitude of real or simulated devices on the cloud. The frameworks and tools were chosen based on an extensive list of available solutions and were scored based on a list of requirements provided by the industry supervisor. The highlights of the comprehensive comparison where I compared over 100 different frameworks and their results, are discussed in the later Design section. The reliance on an existing testing framework was expected in this project since the creation of a new framework from scratch was out of scope.

The evaluation and comparison of the many different testing frameworks was one of the key deliverables for Onsite. A presentation of the key findings was held to the industry supervisor, where feasibility and adherence to the provided requirements were tested. Therefore, approval of the proposed potential designs was the first metric for the project. Another key deliverable for the automation framework is the setup and implementation of the final testing framework for both simulated and emulated devices from BrowserStack and

This project was supervised by Philip Hu (industry) and Simon McCallum (academic).

Appium + Android Studio respectively. The main goal for this deliverable was on future code maintainability and flexibility and it concluded with a presentation to the QA team at Valocity. Lastly, the final and original goal of 40-50% code coverage has changed, and the focus was instead shifted to the code architecture. This was due to a shift in business priority that would invalidate test scripts written for Onsite had they been written. Therefore, the metrics on code readability, maintainability, and correctness were emphasized, and code architecture had to reach a high likeness with the automation team's web testing frameworks for extensibility purposes.

The previously mentioned shift in business priority not only resulted in the change of goals for the first half of the project, but it also resulted in major changes for the second half of the project. Specifically, the second portion migrated from the continuation of web automation to the complete redesign of the QA dashboard. Although this change was unprecedented, it proved to be manageable given the existing code infrastructure and suggested shortcuts from the previous owner and industry supervisor.

The QA dashboard is a visualization tool for the QA team and its manager. It shows the current automation status of the different projects within the different squads in Valocity. Specifically, this dashboard provides the QA team with a tool to manage their upcoming priorities, and the manager with a quick overview of the overall automation for stakeholder sessions and meetings. It was written using the React framework and it utilizes the Material UI and Chart.js libraries. The dashboard's redesign was proposed since the previous implementation proved to be under-utilized and missing a couple of features that would make it an effective visualizer. The implementation of this dashboard is important as it is becoming increasingly complex to distinguish between features with ample automation versus features that are strictly manual as the company grows. Additionally, the information on regression testing time, pipeline status, and Azure DevOps test plan structure, could further streamline processes within the QA team at Valocity, while also improving transparency within the different squads and projects. The goals for the QA dashboard include successful integration into the QA team, approval from the manager, and numerous performance and correctness metrics. Strategies like caching, efficient database querying, and User Interface (UI) interactivity were implemented to overcome these challenges. The implementation and decisions of these strategies can be seen in the later sections.

II. Related Work

A. Onsite Testing Framework

Due to the nature of this portion of the project befitting a unique use case, background research is limited and oftentimes not applicable. Specifically, throughout this half of the project, there was significant difficulty finding references or other projects with a similar goal online, apart from general suggestions of frameworks from varying websites trying to sell their product. Therefore, the focus for this section is on the adherence to general good practice, and to personal and shared experience relevant to the topic.

Firstly, mobile automated testing has significant friction. Some companies glorifying it, and many others condemning it, especially UI automation. One of the latter's main points is that mobile automation brings unnecessary overhead and work since testing scripts are very fragile when paired with a volatile mobile application. This is especially the case for hybrid mobile applications since it generally means that the application is lightweight and of small scale. In fact, the aforementioned business shift resulted in many of the finalized features of Onsite to change, causing the halt of the code coverage metric for the project, and reinforcing the idea that automated testing should be avoided for lightweight mobile applications.

However, as the application and its users get larger, the cost of automated versus manual testing is justified, and this is reflected in [6]. Moreover, Heusser states that the introduction of automated testing reduces human error from tedious and complicated tasks, therefore explaining Valocity's push for automated testing with their plans to expand Onsite. However, I believe that Onsite is still in its early stages and the shift to avoid extensive automation scripting is more than justified, given the current volatility of the application. The project with its shifted focus on creating the backbone for future testers should therefore be sufficient, and future work on the project could be on the extension of the basic scripts written after Onsite has been further finalized.

Other gripes and potential challenges for mobile automation testing can be seen in [6] and [7], and these should be considered in the future when extending the testing framework. Of these challenges, the most problematic for Onsite would most likely be simulating real-life scenarios since the current solution is purely using emulated and simulated devices. This could be resolved by implementing localized end-user testing in India; however, this is far out of scope for the project. Instead, the solution's inclusion of BrowserStack minimizes this issue, along with many other common challenges listed in [7], since it enables the quick simulation of a multitude of devices with varying types, generations, and operating systems.

B. QA Dashboard

Unlike the first half of the project, this portion of the project allows for significantly more background research and related work. There are many templates, common good practices, frameworks, and libraries that are freely available to avoid having to reinvent the wheel. This significantly eased the implementation of this portion of the project on the QA dashboard and provided a wider pool of possibilities. However, as discussed in the upcoming Design section, several constraints have been placed to avoid additional costs.

Some of the templates used for influence can be found in [8], and the previous and now current iteration of the QA Dashboard utilizes one of these templates as the foundation. This was one of the strict requirements of the QA Dashboard to avoid having to worry about the frontend to focus on the backend. Had the project been made with Bootstrap or something similar instead of Material UI, I believe that the project would still have turned out with a similar result since these front-end frameworks are very extensive and more than sufficient for a lightweight and quick web application, like the QA Dashboard.

The old version of the QA Dashboard was the main point of reference throughout this project, and it acted as the launching pad. Its extension and redesign were fueled by the feedback of the QA team and the original owner, after realizing that it failed to meet its expectations and display the needed information. Therefore, a new side project was created with new requirements and constraints.

There is a stark difference between the old and the new iterations in both the functionality and the UI side. Firstly, the company's colors and fonts were applied to improve likeness and promote familiarity with its users. Secondly, navigation was extended, and information hiding was introduced to minimize the initial clutter and potentially overwhelming for new users. Lastly, additional functionalities were added, including new information on code and branch coverage, dynamic and interactable charts, and a homepage to allow for easy comparison of the different squads at Valocity. These additional features centralize and visualize the information that a member of the QA team might need. However, although minimized by the use of in-memory caching, these additional features add significantly more strain on the backend when a full update is triggered, which could take almost 25 minutes. This is one of the main shortcomings of the QA Dashboard, and it will continue to worsen as the company grows and writes more test cases. The poor performance may be caused by a lack of knowledge on efficient database querying and management since the current solution manually checks whether any record is inconsistent with the data pulled from Azure DevOps in every update.

III. Design

A. Onsite Testing Framework

The evaluation of the potential mobile automation frameworks followed an extensive stepwise approach, and high priority was placed on the evaluation to minimize future costs. The evaluation consisted of three pruning stages and a final applicability test to Onsite itself. These stages had an increasing strictness and an overarching goal to turn the initial findings of over a hundred to a final two. This stepwise pruning method was used to improve presentability and to ensure that granularity is not lost for most prospective candidates. The framework candidates were filtered according to the set of requirements provided by the industry supervisor, and these requirements are listed below and split as original and additional requirements. The additional requirements were proposed after unknown unknowns were encountered throughout the evaluation.

Original Requirements:

- Code language is based on TypeScript, JavaScript, or C#.
- Supports Page Object Models (POM).
- Supports parallel test runs.
- Capable of testing devices of different Operating Systems (OS), brands, and versions.
- Supports asynchronous steps.
- Capable of performance and penetration testing.
- Includes network log tracking.

Additional Requirements:

- Supports visual testing (snapshots or recordings).
- Supports integration/End-to-End (E2E) testing.
- Avoid Behaviour-Driven Development (BDD) and Test-Driven Development (TDD).
- Parallel tests run from regression-level considerations.
- Avoid codeless frameworks.
- Ignore BrowserStack alternatives.
- Avoid frameworks where scripting and device emulation are merged - might result in loss of control and tight coupling.
- Avoid frameworks defining their Domain-Specific Language (DSL). This may result in the loss of all scripts or costly migration/translation to another scripting language.
- Avoid iOS-specific and Android-specific frameworks like Espresso and XCUITest. Overhead for learning Java/Kotlin and Swift/Objective-C far outweighs the performance improvements that they provide.

i. Evaluation Stages Overview

Firstly, the results were filtered according to the requirements listed above and to any outstanding shortcomings of the framework. Secondly, the remaining candidates were compared relative to each other and scored until four to five candidates remained. This second stage is based on the pricing, code and framework maintainability, relevancy/modernity, and community size. Lastly, the third pruning stage further cuts down the candidates to the final two based on their applicability to Onsite and the project industry supervisor's personal preference. Thereafter, the final two were applied to Onsite, to simulate a basic login on an emulated device to find any compatibility or performance issues.

ii. First Stage

The tables for the first stage have been omitted as they proved to be too large and of little relevance. Instead, see the 'Second Stage' for the results of the first stage of pruning. In summary, many of the discovered frameworks were removed for being codeless, not befitting mobile applications, or failing to meet the code language requirement.

To see the extensive table where over a hundred different frameworks are evaluated, refer to the provided document in the appendix.

iii. Second Stage

This stage explored the remaining frameworks that met the requirements. The basis of the second stage of pruning was on pricing, code and framework maintainability, relevancy, modernity, and community size/popularity. Below are the removed frameworks and some of the reasons for removal.

1) **Calabash**
   a) Originally owned by Xamarin, who was later purchased by Microsoft and Calabash was left behind. Currently open-sourced and looking for a maintainer.
   b) Infrequent updates.
   c) Splits iOS and Android.
   d) Many issues dating back to 2015.
2) **TestComplete**
   a) Specific IDE required for tests. However, an unofficial extension exists in VSCode.
   b) Heavily leans towards the use of BitBar (an alternative to BrowserStack).
3) **Eggplant Functional**
   a) Requires and uses gateway connections.
   b) Weird documentation and oftentimes not English.
   c) Requires specific IDE.
4) **Gauge**
   a) Primarily BDD.
   b) The last update was in January last year with a declining trend in frequency.
5) **Sahi**
   a) Uses Sahi Script, a DSL.
   b) Pricey.
6) **OpenTest**
   a) Infrequent updates with two-year gaps.
   b) Primarily in YAML.

iv. Third Stage

This final stage aims to reduce the candidates to the final two candidates by looking into each framework through a finer lens. The major factors that heavily influenced the final decision are listed below.

1) **WebdriverIO**
   a) 8.1k stars on GitHub.
   b) Almost weekly releases.
   c) Supports many integrations.
   d) Can get BDD-style tests with Cucumber in addition to normal scripting.
   e) Many options for reporting/logging.
   f) Used by many major companies like Google, Netflix, Microsoft, Mozilla, etc.
   g) Designed for web testing so there are limitations to its mobile capabilities.
   h) Lacks support for some features provided by Appium.
   i) Auto-waiting.
   j) Requires separate files for iOS and Android.
2) **NightwatchJS**
   a) 11.4k stars on GitHub.
   b) Weekly releases.
   c) Less control in comparison to WebdriverIO, but easier for beginners.
   d) Automatically retries tests after three fails to account for flaky tests.
   e) Requires separate files for iOS and Android.
3) **CodeceptJS**
   a) 3.9k stars on GitHub.
   b) Monthly releases.
   c) Well documented.
   d) Auto-retry.
   e) One file for both iOS and Android scripts.
   f) Build asynchronously automatically (no need to call "awaits" or "async" unless grabbing from the page.
4) **SerenityJS**
   a) 458 stars on GitHub.
   b) Weekly updates.
   c) Utilizes a Screenplay Pattern [9].
   d) Missing mobile documentation.
   e) Large potential in reporting and able to create living documents for reporting.
   f) Uses WebdriverIO under the hood.

v. Conclusion

WebdriverIO and NightwatchJS are the obvious choices with 8.1k stars and 11.4k stars respectively. CodeceptJS is the third choice with its very simple approach to scripting. Its capability for single script testing on both iOS and Android is also very desirable to the industry supervisor. However, CodeceptJS falls short in comparison to Webdriverio and NightwatchJS as it is a much newer framework with significantly less support, a smaller community, and very infrequent releases. These infrequent releases infer that it might not be a good long-term solution for OnSite, placing it third. Lastly, SerenityJS is in last place, with 468 stars and its non-existent mobile documentation. SerenityJS' only redeeming feature is in its reporting, but with WebdriverIO's large plethora of available reporters, this redeeming quality becomes negligible. Therefore, after careful consideration, the test automation frameworks that progressed to the development phase were WebdriverIO and CodeceptJS. CodeceptJS was chosen instead of NightwatchJS since WebdriverIO and NightwatchJS are seen to have slim differences as discussed in [10]. CodeceptJS may also provide novel solutions through its capability of utilizing one file for both iOS and Android.

vi. Practical Compatibility Test

A practical test against an emulated Android device was designed to further determine the final candidate. The goal is to produce a basic testing script on the login for Onsite to reveal any outstanding issues. Additionally, usability, ease of use, future code maintainability, and performance were evaluated. As a result, a simple login script was written through CodeceptJS and WebdriverIO with paling results. The evaluation of the frameworks' feasibility to BrowserStack's real simulated devices was out of scope for this stage of the project and will only occur when the solution is finalized.

As previously hinted, the results from the two frameworks pale in comparison. CodeceptJS was unexpectedly quickly rejected after its failure to comply with the new and upcoming release of Appium 2.0.0, while WebdriverIO produced smooth results. To summarize the found issue, the release of Appium 2.0.0 changed and invalidated many of the required configurations for emulated device connection with the framework. This simulated a major concern with CodeceptJS' less frequent updates where windows of incompatibility may cause significant downtime for the regression tests. Regardless of its potential to reduce work by 50% through its ability to use one file for both iOS and Android, CodeceptJS was quickly disapproved, making WebdriverIO the final and best solution for Onsite. The mitigations or results of the problem were not extensively researched since the issue is likely to persist and repeat in the future. This is implied in CodeceptJS' relatively slow releases and small community in comparison to WebdriverIO. To see the extensive description and issue reproducibility, refer to the section 'CodeceptJS Results' in the attached appendix.

B. QA Dashboard

The design choices for the QA Dashboard were limited in comparison to Onsite's testing framework. This was due to the company having strict preferences for their technology stack. Specifically, moving away from MySQL and the backend language were non-negotiable as they have already been initialized, and a good case was required to justify the extra, repeated work. However, leniency was given to the front-end framework where Angular, as opposed to the React backing of the old implementation, was explored. Migration to D3.js, a similar data visualization library to Chart.js, was also considered, but the latter library proved to be less hands-on making it the better choice and allowing for the required large focus on the backend.

In the end, the decision was to stay with React and this was heavily influenced by personal expertise since learning a new front-end framework was out of scope. Had the project been written on Angular, there would be significantly more work and inferior performance since Angular cannot reuse components and because there is additional overhead with Angular's bidirectional data binding [11, 12]. Additionally, the reuse of components is vital for the dashboard's multi-squad use case where the detailed squad-view holds the same charts as illustrated in [Fig. 1, Fig. 2], making React the obvious choice over Angular regardless of personal expertise.

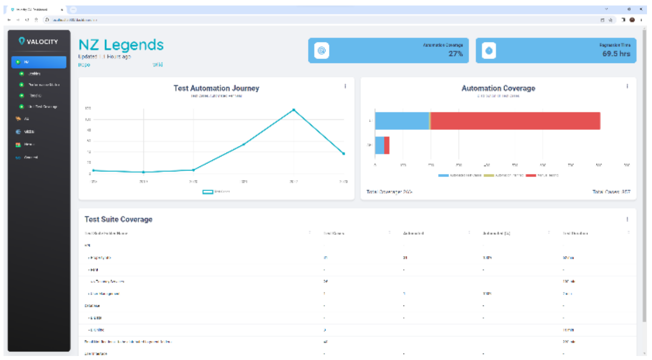


Fig. 1. Detailed NZ squad-view.

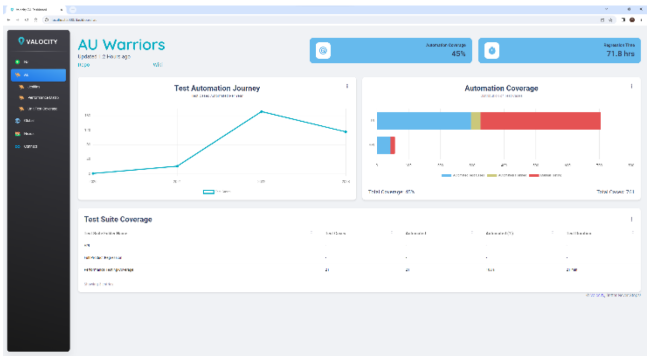


Fig. 2. Detailed AU squad-view.

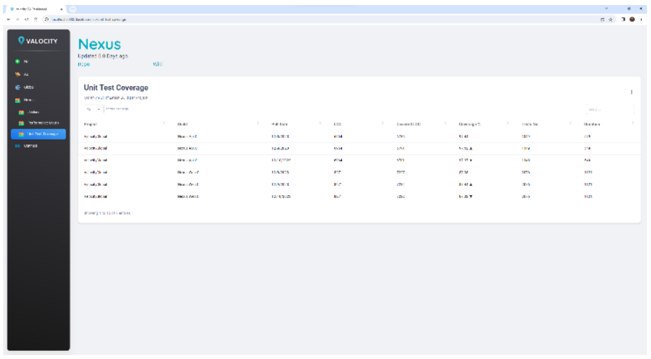


Fig. 3. Unit Test Coverage chart from Nexus squad.

The charts used for this project were influenced and justified by [13]. Moreover, each major iteration of the QA Dashboard included a mock-up and design approval by the main stakeholder. For example, the Test Suite Coverage chart seen in [Fig. 1, Fig. 2] was requested by the QA manager, while the Unit Test Coverage table from [Fig. 3] by the head of technology. These special requests pertained to separate requirements and constraints and are listed below alongside the original requirements for the QA Dashboard. Additionally, these requirements were not provided explicitly, unlike the requirements for the Onsite framework, but were gathered through meetings and informal discussions.

See the diagram on [Fig. 4] for reference on what is meant when referring to test plan, test suite, nested test suite, and test case in the Azure DevOps Test Plans page.

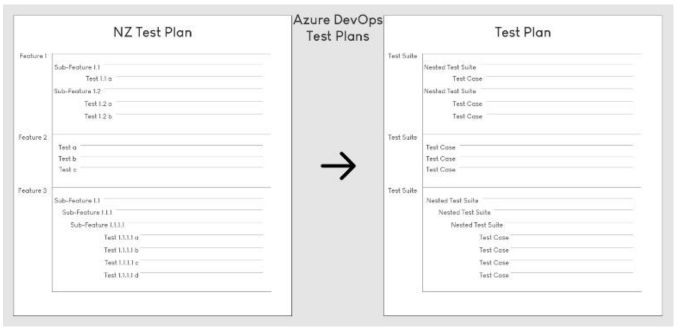


Fig. 4. Azure DevOps Test Plan diagram.

Requirements

- Display total regression time.
- Display automation timeline. i.e., how many automated test cases are made per year.
- Repositories of each squad should be linked to their specific squad-view page.
- Documentation on steps taken and others.
- UI and API testing should be highlighted.
- Updates should not be triggered with every visit to the page.
- Updates should only occur after a set number of hours.
- Display the automation status of each test suite.
- Display overall automation coverage.
- Display the language used to write the automated tests.
- Display Jenkins pipeline status.
- Display the performance of the automation scripts.
- Ensure each squad has their own page.

These were the initial requirements set for the project and were quick to change as the project ensued. Notable changes are discussed in later sections, but the two main additions' requirements are listed below.

Test Suite Coverage Requirements

- Reflects the folder structure in the Azure DevOps Test Plans.
- Dynamic and interactable, where the user can expand and shrink the folders with children.
- Parent folders with no test cases should not be omitted, but instead provide a way to signify that there are no test cases.

The Test Suite Coverage chart is a remodel of the bottom-left chart seen in [Fig. 5], which is the first page in the initial high-fidelity mock-up. The complete mock-up is included in the appendix. The main stakeholder for this chart, the QA manager, predicted that the reflection of the folder structure and its hierarchy from the Azure DevOps Test Plans will provide more value. Additionally, the lack of folder structure could misconstrue the user, especially if subfolders had the same name or if the test plan's hierarchy is heavily extensive. Thus, bringing the remodel to [Fig. 1]. The expandable table proved to be the best choice for this problem since it allowed the user to focus on certain folders by expanding and shrinking others. Additionally, the user's mental model of how to use the chart is improved by mimicking common folder functionality. A major improvement to this chart would be to include totals of each column to the appropriate parent test suite.

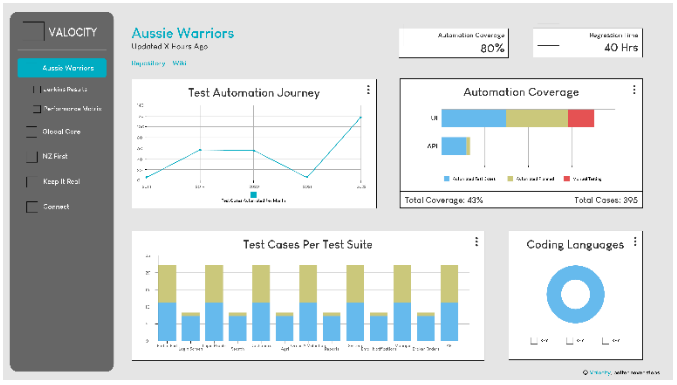
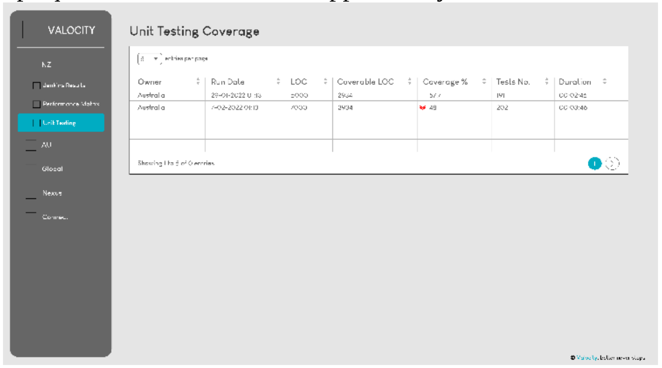


Fig. 5. First high-fidelity mock-up of a squad-view.

Unit Test Coverage Requirements

- Create Unit Test Coverage sub-page within the squads.
- Display coverage %, total lines of code, total number of tests, and test run duration.
- Include the journey of coverage, like the Test Automation Journey chart seen in [Fig. 1].
- Ensure that it fits on one page.
- Update once every month.
- Add new squads, specifically, add Data and Platform since they also have unit testing.

The requirements listed above were the initial requirements gathered with the Head of Technology at Valocity. However, some of these requirements were found to be suboptimal and were edited post hoc. Specifically, the addition of Data and Platform were out of scope and invalid since they lacked automated testing. Moreover, displaying the journey of unit testing within each dedicated subpage provided little value and alternatives, like its migration to the homepage, were proposed. Lastly, the difference in update time for unit tests and automated tests would invalidate the tooltip "Updated X Hours Ago" as can be seen under each title in [Fig. 1]. With these changes in mind, a mock-up is made [Fig. 6], and the proposed unit test journey by the Head of Tech was migrated to the homepage instead as can be seen in [Fig. 7]. These holes within the requirements were quick to be acknowledged and the mock-ups quick to be finalized and approved by the Head of Tech.



Unit Testing Coverage

5 entries per page

Owner	Run Date	LOC	Coverable LOC	Coverage %	Tests No.	Duration
Australia	29-01-2022 01:13	5000	2934	57.7	191	00:02:45
Australia	7-02-2022 01:10	7000	3934	49	202	00:03:46

Showing 1 to 5 of 0 entries

Fig. 6. Unit Test Coverage table.

These changes aim to minimize scope creep and improve comparability between the different squads. The ability to select and disable certain data sets in the line graph meets the coverage journey requirement. Moreover, the proposed model ensures that the user's mental model remains consistent since the charts seen in the subpages were designed to be a detailed view for anyone looking for more information.

Alternatives to this design include the addition of empty parent pages for the Data and Application squad, the duplication of the aforementioned tooltip, and the creation of a dedicated page for the Unit Test Coverage at the bottom of the side navigation bar which would include everything. However, these ideas were quickly dismissed as they would introduce inconsistencies to the dashboard and could cause significant scope creep in the future as more dedicated separate pages are requested.

IV. Implementation

A. Onsite Testing Framework

As discussed in the previous section, WebdriverIO proved to be the best solution through the extensive pruning and filtering of many mobile testing frameworks. In addition to this, Appium and BrowserStack were also finalized for this project. The respective tool and framework were chosen as they were already being utilized within the company and because there is no other competitor.

As outlined in the previous section, correct code architecture and future maintainability are paramount for a successful implementation. Otherwise, the other automation testers picking up the project may find it difficult and confusing since the other automation repositories pertain to a similar structure. To overcome this, large care was taken to learn and understand the automation team's other repositories. This was a major hurdle for this half of the project since previous experience unknowingly gestured towards code reformat regardless of compliance to the expected structure. This was shortly met with a meeting with the industry supervisor heavily advising for a training module designed to reinvigorate and understand the expected structure.

The training module lasted for two weeks, and it consisted of direct involvement with one of the other automation repositories on a different Valocity product. Thereafter, numerous reformats were made to the Onsite testing framework to better match the expected code architecture. Moreover, a better understanding of the reasoning behind the code architecture was contrived and knowledge behind why compliance with the company structure is preferred was gained. The key takeaway for this half of the project is that consistency and readability tend to trump performance and shorter lines of code.

In addition to the previous requirement on code architecture, common general practices for clean, efficient, and performant code were applied to the project. Specifically, [14, 15, 16] were closely followed. These articles applied to the entire setup of the project but were most utilized in the test script written for the login process of Onsite. The language utilized for the testing framework was TypeScript instead of JavaScript for type-safety. The scripting process utilized Appium Inspector to locate elements and Android Studio for an emulated device. With the test cases on the login process passing for the emulated device, the project ensued to the final portion of simulating the scripts on real, cloud devices from BrowserStack. Here, the written login script proved to be working with numerous devices with varying operating systems, versions, and brands. For example, the script was able to run on a Google Pixel 7, a 10th gen iPad, and an iPhone 12.

As part of the requirements, proper documentation from the evaluation to the implementation was written for the project. These documents are written on Valocity's Azure DevOps wiki and include information on the extensive evaluation and research, the Appium emulation steps, and the common onboarding tips for Onsite automation. Additionally, edge cases and minute details about the scripting process were added to the repository's ReadMe.

The implementation of the finalized best solution proved to be minimal in comparison to the evaluation stage of the project. This was especially the case after the code coverage goal was deemed out of scope for the project. However, with the best framework for the use case implemented, and the basic process heavily documented, the next automation tester can easily and confidently pick up the project.

As a final note, the advantages of CodeceptJS over WebdriverIO proved to be negligible after implementing WebdriverIO. This is because WebdriverIO was also able to use one locator for both iOS and Android, given that the "name and "content-desc" keys contained the same value respectively. However, a missing key and value pair would result in having to split the selector for that element. Additionally, this feature seemed to be less valuable than initially considered since a mobile application written natively, rather than hybrid, could result in very different naming conventions and heavy divergence. Therefore, further reinforcing WebdriverIO as the best mobile automation framework for Onsite.

B. QA Dashboard

Unlike the first half of this project, this half of the project consisted of a more extensive implementation than its design and evaluation counterpart. Specifically, major strain was experienced for the backend where caching, database updating, and database querying were required. The backend of the QA Dashboard is written on C# and alternatives to the backend were non-negotiable due to the previous infrastructure. This is similar to the MySQL database used.

The first steps in implementing the dashboard were to pull, parse, and process several Azure DevOps REST APIs. The data received consisted of test plan, test suite, test case, and later unit test data, which was later processed and submitted to the appropriate table in the relational database on MySQL. The creation and design of the database schema was also part of the project and information on the necessary data was gathered during the design and evaluation phase. This updating process is the bottleneck in the program since the database update could take close to 25 minutes initially. This process was later optimized to hit close to 18 minutes but is only subject to increase as the company grows. This was an unavoidable bottleneck as the test data was very large. However, future work could be on the optimization of the database querying and management.

To minimize the updates, a caching mechanism was introduced and set to update when a user opens the dashboard and if it has been three hours since the last update. The caching type is in memory caching from ASP.NET Core and it not only minimizes updates but also greatly improves application performance. It can cut down chart load times from the initial hundreds of milliseconds to tens and even ones of milliseconds after repeated browsing. In-memory caching was used because the dashboard is strictly read-only, meaning that the loss of cached memory in the case of a crash is negligible. Additionally, in memory caching from ASP.NET Core was used since it was the simplest and most documented caching type for C#.

In terms of the frontend, significant effort was placed on the backend to minimize application-layer processing. Specifically, the exposed API endpoints from the backend process the data received from the database prior to sending it to the front-end. Additionally, an endpoint was designed for each chart seen in the dashboard to further minimize processing on the application layer.

The frontend utilizes React, MaterialUI, and ChartJS to avoid the unnecessary reinvention of the wheel. These frameworks and libraries provide interactable charts and common layouts and templates for the dashboard. In fact, the dashboard's frontend is derived from a template from [8], as advised by the industry supervisor to minimize frontend involvement. As explained in the design section, alternatives to both MaterialUI and ChartJS were explored but were soon found to be inferior to the current solution. Additionally, the usage of Angular instead of React was explored, but the final decision came down to personal experience since the self-teaching of a new framework proved to be out of scope for the project. React's component reusability was one of the main features utilized in the project since each squad-view is identical besides a few minor differences between squads. For example, the charts and layouts are identical, but the data, title, caching time, and links associated varied. These minor variations utilized component parameters and were implemented to ensure that the user's mental model stays consistent throughout the different squads.

Many of the unnecessary charts and functionalities from the template were removed, while many additional custom charts were implemented. Specifically, the multi-line and stacked horizontal bar chart from [Fig. 7] were designed and added to meet several of the information display requirements. Moreover, the nested side navigation, horizontal bar chart, vertical bar chart (old Test Suite Coverage chart [Fig. 5]), and interactable table (new Test Suite Coverage chart) from [Fig. 1] were also added. The interactivity of the charts was derived from ChartJS and they utilize numerous React hooks. User interaction was prioritized in the frontend to improve user experience and information retainability. Additionally, the interactive nature of these charts allows the users to modify the information provided by clicking the legends.

In conclusion, the requirements for the QA Dashboard were sufficiently achieved as it displays all the required information, is performant, and provides a quick overview of the automation status within the different squads with a glance. Moreover, the effort placed in minimizing application-layer processing opens avenues for extendibility and scalability in the future. On the other hand, the prioritization of the user's mental model through the application of traits like interactivity and consistency makes the dashboard user-friendly.

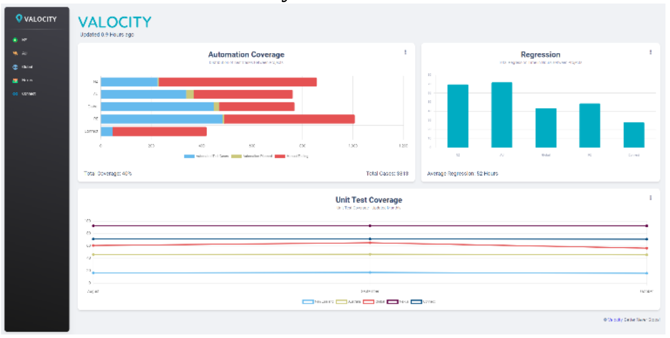


Fig. 7. QA Dashboard homepage.

V. Evaluation

As briefly mentioned in the previous section, the goals and requirements for this project were sufficiently met. This is even more evident with the QA Dashboard since minor hiccups and derailing like the proposed training module for the testing framework did not happen. Although specific metrics were not defined for the project, I believe that the solutions and designs implemented are sufficient through the feedback received from end-user testing and presentations.

As jested by the industry supervisor, the implementation of a test harness for a test harness is ludicrous and the main metrics for Onsite's testing framework should simply be on the framework's ability to perform basic UI tasks. Therefore, to evaluate this, basic scripts were written to check WebdriverIO's ability to navigate, submit input, press buttons, scroll, and perform under varying network conditions with Appium. Specifically, a script that looks for a specific property valuation and a script that switches between Wi-Fi, data, and airplane mode was written. The successful execution of these scripts was sufficient proof to mark this project as a success, but care should be taken for future work on more complicated tasks like zooming, and image validation. Additionally, the positive feedback from the presentation to the QA team showcasing the basic login script running on numerous devices provided further affirmation of the project's successful implementation.

The QA Dashboard is similar to the first half of the project in terms of evaluation as specific metrics were also not provided apart from the list of requirements. However, since adherence to the requirements has been discussed in the previous sections, this section will instead focus on the feedback from the end-users themselves. Firstly, a presentation to the entire QA team of the first iteration of the QA Dashboard netted positive feedback and a lack of questions. The lack of outstanding questions can be taken to mean that the QA Dashboard has proven to be as straightforward as initially intended. Secondly, the second major iteration of the QA Dashboard, with the modification of the Test Suite Coverage chart to a table, proved to also satisfy the expectations and requirements. Specifically, the QA manager can be seen to actively use the new dashboard to present to some of her higher-ups to outline the overall automation status. Moreover, positive feedback was also received on the result regarding the graph and the overall dashboard from the QA Manager. Lastly, the Head of Technology seemed to also be thoroughly impressed with the proposed mock-ups for his unit test coverage module. However, sufficient feedback was not received to guarantee his satisfaction due to his busy schedule. Therefore, future work for the QA Dashboard could be on the assurance that the unit testing additions were satisfactory.

All in all, it seems that the project's implementation was a success, and that the final solution was able to meet the criteria proposed by the numerous stakeholders involved in the project. This seems to especially be the case for the QA Dashboard, but the results could be misconstrued from the fact that only a subset of the QA team, the automation team, reviewed the final Onsite testing framework. Moreover, the senior automation lead and industry supervisor could be the only member within the automation team with high regard for future code maintainability and compliance with the expected code architecture, potentially making the implementation's satisfaction levels higher than the actual level. Future work on this portion of the project could therefore be to define quantifiable metrics and standards that can be measured to better test the testing framework's compatibility with Onsite.

VI. Conclusion and Future Work

In this report, the importance of automation testing and data visualization was discussed. Moreover, the process of determining the best possible automation testing framework for a certain use case and set of requirements was discussed. This process can be used to determine the ideal framework for many different use cases. For Onsite, the optimal testing framework proved to be WebdriverIO, and several other tools and frameworks like Appium, Appium Inspector, Android Studio, and BrowserStack were also utilized in the project. Here, the emulation and simulation of devices were explored with automated scripts executed onto them. The results for this portion of the project proved to be satisfactory and many key learnings were gained from this project. However, the growth of Onsite and its users could result in WebdriverIO's inability to test complicated and lengthy test paths.

Future work on this half of the project could therefore be on the stress testing and edge case testing of Onsite to find WebdriverIO's limitations. The addition of more test scripts to improve code coverage could also be an option for future work that may reveal WebdriverIO's true performance. Moreover, as the application and its users grow, Onsite's developers may choose to opt in for native development rather than hybrid, to access all the device's functionality. Given this case, replication of the study could be another avenue for future work as major divergence in elements and overall application functionality could result in poor performance and doubled work. The native test framework alternatives like XCUITest for iOS and Espresso for Android are found to have better performance since they are "closer to the metal" [17], making a replication study given this situation extremely relevant. Lastly, with CodeceptJS being a close second had it been able to catch up to updates, its re-exploration could prove to be worthwhile with fruitful returns if it offers a different and more efficient way to use one locator for an element.

On the other hand, the design and implementation process for the development of the QA Dashboard was also discussed in this report. The QA Dashboard is the second half of the project, and it aims to visualize automation and testing status. It was brought forward after the old iteration failed to meet its expectations and the QA Dashboard served to replace the original second portion of the project on the continuation of a web testing framework after a large shift in business priority.

Thereafter, requirements, constraints, and expectations were quickly gathered, and mock-ups of potential designs shortly followed suit. The volatile and agile nature of this part of the project meant that features should be easily customizable, and this was made possible with libraries like Material UI and ChartJS. Additionally, the dashboard was written using the React framework to allow for components, which the dashboard used heavily. The dashboard proved to be more than satisfactory from the positive feedback received informally or through presentations. However, the QA Dashboard has one major pitfall, and that's the performance of a full update.

Therefore, the future work for the QA Dashboard could be on the large optimization of the discussed bottleneck on database updates. This is a highly relevant avenue of future work as any improvements made away from the bottleneck are illusory. The cause of this bottleneck could simply be from lack of background knowledge on database management and its good practices. Additionally, there could be better Azure DevOps REST APIs that may avoid redundant querying of unedited test suites or test plans. Several Azure DevOps features may have also been overlooked which may allow for the attachment of listeners or a publish-subscribe model that listens to test suite and test case creations and modifications.

Acknowledgements

I would like to thank Allen Zhou for implementing some of the features of the QA Dashboard. These features were the subpages seen in many of the provided figures. Specifically, they are the Jenkins, Performance Matrix, and Flood IO subpages.

I would also like to thank Valocity for allowing me to take on some of their projects as my final university project. It was a great opportunity with invaluable experience.

References

[1] R. Chandran, "A simple address unlocks new life for Indian slum dwellers," 8 March 2017. [Online]. Available: https://www.reuters.com/article/us-india-landrights-address-idUSKBN16E1SE. [Accessed 13 October 2023].

[2] S. Bhattacharya, S. S. Sathya, K. Rustogi and R. Raskar, "How much do inaccurate addresses cost India? $10 billion to $14 billion a year!," 3 December 2018. [Online]. Available: https://archive.factordaily.com/india-14-bn-problem-with-addresses/. [Accessed 13 October 2023].

[3] S. Bhattacharya, "Automated Systems to Solve India's $10B+ Addressing Problem," 20 January 2019. [Online]. Available: https://santanub.medium.com/automated-systems-to-solve-indias-10b-addressing-problem-9063190921ef. [Accessed 13 October 2023].

[4] S. Chakraborty, "This startup has a simple code to solve India's complex address problem," 14 October 2014. [Online]. Available: https://qz.com/india/280494/this-startup-has-a-simple-code-to-solve-indias-complex-address-problem. [Accessed 13 October 2023].

[5] "United Nations," Department of Economic and Social Affairs, 2015. [Online]. Available: https://sdgs.un.org/goals. [Accessed 30 May 2023].

[6] M. Heusser, "Manual vs. Automated Testing For Mobile Apps: Which Do You Need?," 14 August 2023. [Online]. Available: https://saucelabs.com/resources/blog/mobile-testing-basics-manual-vs-automated-testing. [Accessed 13 October 2023].

[7] BrowserStack, "Challenges in Mobile Testing (with Solutions)," 3 December 2022. [Online]. Available: https://www.browserstack.com/guide/mobile-testing-challenges. [Accessed 13 October 2023].

[8] "Fully built Material UI templates," Material UI, 2023. [Online]. Available: https://mui.com/templates/. [Accessed 14 October 2023].

[9] SerenityJS, "Screenplay Pattern," SerenityJS, 2021. [Online]. Available: https://serenity-js.org/handbook/design/screenplay-pattern/. [Accessed 15 October 2023].

[10] Presidenten, "What is the difference between NightwatchJS and WebdriverIO?," 14 Mar 2016. [Online]. Available: https://stackoverflow.com/questions/35981605/what-is-the-difference-between-nightwatchjs-and-webdriverio. [Accessed 30 May 2023].

[11] H. Dhaduk, "Angular vs React: Which to Choose for Your Front End in 2023?," 28 June 2023. [Online]. Available: https://www.simform.com/blog/angular-vs-react/#:~:text=React%20is%20a%20JavaScript%20library%2C%20whereas%20Angular%20is%20a%20TypeScript,has%20a%20smaller%20bundle%20size.. [Accessed 14 October 2023].

[12] N. Raval, "React vs Angular: Which JS Framework to pick for Front-end Development?," 3 July 2023. [Online]. Available: https://radixweb.com/blog/react-vs-angular#ARDifference. [Accessed 13 October 2023].

[13] A. Ladipo, "Chart Suggestions Guide," salesforce, 28 August 2022. [Online]. Available: https://public.tableau.com/app/profile/adedamola8122/viz/ChartSelectionGuide2/Dashboard. [Accessed 14 October 2023].

[14] labs42io, "clean-code-typescript," GitHub, 8 July 2023. [Online]. Available: https://github.com/labs42io/clean-code-typescript. [Accessed 15 October 2023].

[15] H. Roberts, "Writing efficient CSS selectors," CSS, 17 September 2011. [Online]. Available: https://csswizardry.com/2011/09/writing-efficient-css-selectors/. [Accessed 15 October 2023].

[16] T. Li, "How to Write Clean TypeScript Code," Medium, 19 March 2022. [Online]. Available: https://betterprogramming.pub/how-to-write-clean-typescript-code-eda1716eead1. [Accessed 15 October 2023].

[17] SauceLabs, "Getting Started with XCUITest," SauceLabs, 7 October 2022. [Online]. Available: https://saucelabs.com/resources/blog/getting-started-with-xcuitest. [Accessed 15 October 2023].