University of Waterloo
Faculty of Engineering
Department of Electrical and Computer Engineering

Developing an automated testing strategy for a multi-tier mobile application

Ultimate Software Group Inc.
144 Bloor St. W
Toronto, Ontario, Canada

Prepared by
Arthur Chun-Yin Leung
ID 20601312
userid ac7leung
1B Computer Engineering
8 January 2017

200 University Ave W
Waterloo, Ontario, Canada
N2L 3G1

8 January 2017

Vincent Gaudet
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario
N2L 3G1

Dear Sir:

This report, entitled ‘‘Developing an automated testing strategy for a multi-tier mobile application’’ was prepared as my 1B Work Report for the University of Waterloo. This report is in fulfillment of the course WKRPT 201. The purpose of this report is to propose an optimal strategy for software testers to maximize their effectiveness in their role, using a combination of approaches as well as my own findings on the subject matter.

Ultimate Software Group Inc. establishes itself as a ”Software as a Service” (SaaS) company, mainly specializing in providing web application software solutions for other companies to manage human resources matters; such as payroll, taxes, and employee benefits. The Quality Assurance (QA) group of the mobile platform team that I worked in was managed by Yuvraj Vedvyas, which oversaw the testing of the software application.

I’d like to thank Yuvraj on providing guidance on the design process of the project outlined on this report. I hereby confirm that I have received no further help other than what is mentioned above in writing this report. I also confirm this report has not been previously submitted for academic credit at this or any other academic institution.

Sincerely,

Arthur Chun-Yin Leung
ID 20601312

Contributions

For the duration of this co-op term, I worked in Ultimate Software's mobile platform team, which consisted of around 20 members in the office. The team was further organized into the "Frontend development", "Backend development", and "Quality Assurance (QA)" roles; I was assigned a role in QA, and reported to Yuvraj.

The team's main goal was to develop a JavaScript web application compatible with modern browsers, for users on mobile devices to interface with the company's UltiPro product, and would also provide native applications for both the Android and iOS mobile platforms. It was to be a lightweight client with a responsive design, that communicates with a "backend" web service through Representational State Transfer (REST) APIs. Tasks ranging from feature research and implementation to bug fixing would be reviewed on the team Kanban board during daily stand-up meetings, where team members discuss their progress and any hindrances that occur.

My primary role in the team was to verify that new features produced by the developers did not introduce bugs and undesired behaviour to the application, by utilizing various software tools (Selenium, Postman) to reproduce bugs and automate testing. Additionally, I verified that the feature additions met specification standards, and that they followed the defined User Experience (UX) guidelines. My daily tasks ranged from manual exploratory testing on hardware (iOS and Android smart phones) to designing automated test cases and scenarios using aforementioned tools, then providing feedback to developers according to my findings. I also assisted in setting up meetings with members across other products' teams and UX designers, to ensure that everyone was aware of any major changes or design decisions made by our team's developers. That way, there would be less potential conflicts in the future, regarding the design of a particular component of the software for example.

Occasionally, a live demo of the application would be requested by clients or executives that visit. I would then help the team create and import the necessary test data, and configure each use case or test flow for the feature being demoed. I would also detail the steps I have carried out in documentation, for other team members and future co-op students to reference if there exists the need to replicate the process.

The relationship between this report and my job is a direct one. Though my work less pertains to the development, and more to the testing of the mobile application itself, I was responsible for the design of new unit test cases and maintenance of the existing test collection for the mobile team; with the addition of a new feature, thorough test cases must be crafted covering from the front (User Interface/ UI) to back end (RESTful APIs). Thus, the experience has improved my problem solving abilities as a tester, and collectively reinforced my skills as a

programmer, by increasing my awareness of best practices in development and avoiding bad or bug-prone ones. I have also gained significant insight into software architecture design and the efforts demanded in the upkeep of multi-tier software systems. The effort I have invested in preparing this report has also furthered my communication and analytical skills; skills of which I believe are essential to developing professionalism, and are indispensable in all engineering workplace and academic settings.

In the broader scheme of things, the collection of automated tests I have written for the mobile team will continue to serve its purpose. It will provide the groundwork for maintaining the current product's robustness as features are added, and also be a template for the testing or development of future products in other teams within Ultimate Software.

Summary

The scope of this report is to propose an optimal solution for testing mobile software, which use a multi-tier architecture as is the case with Ultimate Software's mobile platform product. Even though the mobile devices mentioned here mainly refer to handheld smart-phones, the approaches discussed in the analysis can be extended and applied to other mobile device form factors (tablets and wearable smart watches for example) since the concepts of software testing should be device or implementation agnostic.

The major points covered in this report are the different types of testing, benefits and drawbacks of common techniques used, alternative methods, determining when automation is suitable, and how optimization of tests can aid performance.

The major conclusions in this report are that if a software development team is testing their product purely manually, it can be a good idea to consider implementing an automated solution. However, automation should not be treated as a solution to all problems (in other words, a "golden hammer"), but can greatly increase efficiency of the development team when used appropriately. If there team does have an automated solution, then investing time in revisiting test cases to optimize them could be beneficial for overall performance.

The major recommendations in this report are that project managers and development teams should consider different automation technologies in testing if they haven't already, and that tests, automated or not, should be written with maintainability and re-usability in mind. It is also highly recommended that testers should explore means of notifying developers whenever a bug or test failure is found, as to reduce the likelihood that a bug is left unnoticed.

Table of Contents

Contributions . . . iii
Summary . . . v
List of Figures . . . vii
List of Tables . . . viii
1 Introduction . . . 1
1.1 Background . . . 1
1.2 Motivation . . . 1
1.3 Scope and considerations . . . 2
2 Requirements . . . 3
2.1 Summary and Criteria selection . . . 3
3 Design options . . . 4
3.1 Summary of choices available for enhanced testing . . . 4
3.2 Comparison of options for backend testing . . . 6
3.3 Comparison of options for frontend testing . . . 7
3.4 Possible solutions . . . 8
4 Analysis of proposed solution . . . 9
4.1 Setup . . . 10
5 Performance . . . 11
6 Additional optimizations and improvements . . . 12
Conclusions . . . 14
Recommendations . . . 15
Glossary . . . 16
References . . . 17
Appendix A Maven Configuration . . . 18

List of Figures

Figure 1-1 A high level overview of a typical software development build pipeline . 2
Figure 3-1 An overview of the multi-tier architecture of the mobile web application . 4
Figure 4-1 The software development build pipeline with proposed testing process . 11

List of Tables

Table 3-1 Criteria weighing 6
Table 3-2 Qualitative evaluation of backend testing options 7
Table 3-3 Qualitative evaluation of frontend testing options 8
Table 6-1 Time taken to run frontend tests before and after refactoring 13
Table 6-2 Time taken to run backend tests before and after refactoring 13

1 Introduction

1.1 Background

The development industry-level software in the modern day can involve complex infrastructure. Code written by developers usually pass through several processes before becoming part of the final product, and reaches the clients' hands. One of the main reasons for this practice is to allocate time in ensuring the product's quality, to minimize the chance of a customer having a bad experience, as it entails negative reviews. As such, these processes include but are not limited to code reviews, automated and manual testing, and demonstration the pre-release software to clients.

A high level overview of typical software build pipeline is provided in Figure 1-1. First, any code additions or edits written by collaborating developers would be submitted to a repository with support for managing versions and changes, also known as a Version Control System (VCS). Examples of systems include Git [4], SVN [1], and the legacy CVS [3]. Once the code changes are stored, a set of "build agents" would pull the latest source code to compile the application and publish binary artifacts (if there are any, which may be used as components in building other software that require a certain functionality of the code). Finally, the compiled or packaged application can be deployed and run on different environments, from testing to production.

1.2 Motivation

Often times, software testers will only be able to test their product once it has been set up in a testing environment. Apart from errors found at build-time, bugs due to developer oversight that slips past code review and in configuration remain hidden until the deployment stage. Once a bug is identified through exploratory or manual testing, a ticket or report is sent through some form of triage system to notify developers that a fix is in order. Then the fix made by the developer must move through the pipeline until it reaches the tester; a process that could span a few days depending on the severity of the bug. This could be inconvenient especially facing imminent deadlines (such as an upcoming demo), or switching focus to

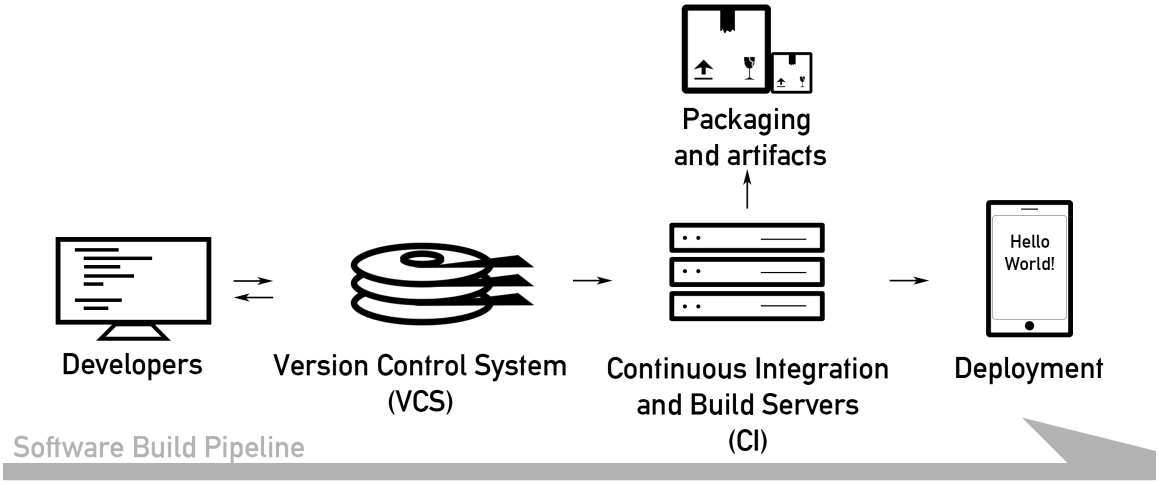

Figure 1-1. A high level overview of a typical software development build pipeline

work on another feature. Ideally, the developer should be notified as early as possible in the build pipeline of any bugs, to save otherwise wasted time in waiting for the feature to be verified.

1.3 Scope and considerations

This report is mainly concerned with strategies and methods to improve the effectiveness of software testers in the software build pipeline, which is shown in Figure 1-1. Any software development project of substantial scope, that uses a multi-tier architecture and also utilizing a similar pipeline process should be able to benefit from the testing strategies outlined in the following sections of this report.

For Ultimate Software Mobile development team in particular, their multi-tiered web application utilizes multiple pipelines in the build process and several deployment environments; one for development, demo, and pre-release (alpha) builds. Integrity of data is imperative wherever multiple tiers or web services communicate with each other, and is where most of the testing should target. The application itself is built with ionic, a hybrid mobile app framework. It produces a JavaScript application for desktop web browsers, and native applications for iOS and Android separately. As the codebase grows, manually testing every functionality in the application would become an increasingly daunting task. The need for

an automated testing system is evident, and the following sections will cover the process from requirements through analysis, to reach a proposed solution.

2 Requirements

2.1 Summary and Criteria selection

The main goal was to build a comprehensive testing strategy that would cover every tier of the mobile application, targeting software regressions as a preventative measure against delivering a buggy product. In addition, the solution was to be automated in the build pipeline to save software testers' time, by enabling them to focus more on exploratory testing as opposed to re-testing old features of the software constantly. Part of the requirement was also to determine which technologies were compatible with the existing software systems, in the case of Ultimate Software's mobile platform.

Additional requirements for the strategy are as follows:

- Should be mostly automated, and any bugs uncovered by this system should be reproducible through manual testing
- Should not cause a significant performance impact to the development pipeline, or have a ”bottleneck” effect
- Should be compatible with existing technologies in the pipeline
- Should provide feedback and alert the developers if a failure occurs

From these requirements we select the following criteria:

- Automation: How well does the system help reduce manual/ repetitive testing? (Level of automation)
- Setup time: How much time is required to get the system running? (Amount of effort required)
- Computational resources: Does the system require great amount of computing power? (Amount of computational power required)
- Coverage: Can the system test multiple tiers and cover multiple processes in the

pipeline? (Amount of coverage provided)

- Maintenance: How much effort is required in the upkeep of this system? (Amount of effort required)
- Re-usability: Could this system be used for other software projects as well? (Level of re-usability)
- Feedback and logging capability: Does this system provide means to notify the developer/ tester of a failed pipeline process or test? (Feedback capability)

3 Design options

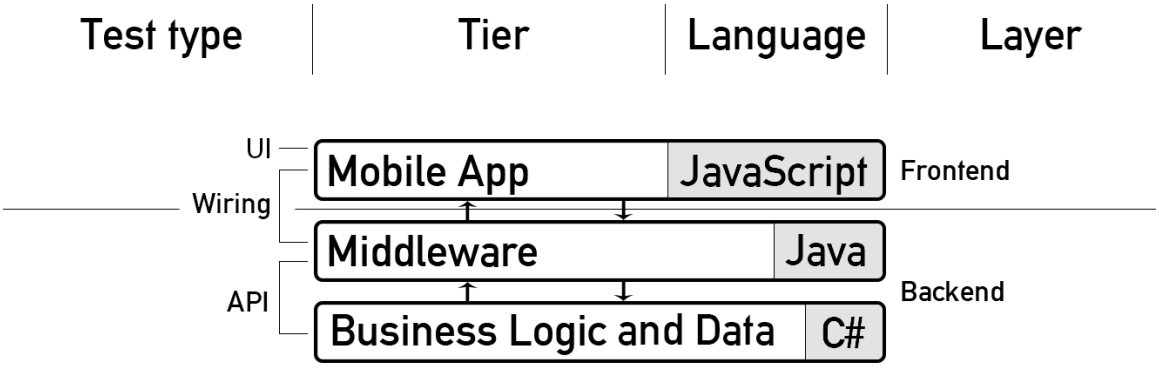

Figure 3-1. An overview of the multi-tier architecture of the mobile web application

3.1 Summary of choices available for enhanced testing

To address the main goal of decreasing time wasted in manual/ repetitive testing in the build pipeline, different technologies were considered based on the features that they offered, compatibility with the build pipeline, and user-friendliness. Figure 3-1 illustrates the multi-tier mobile software system that was to be tested. It consisted of a client facing ”frontend app” implemented in JavaScript, a ”middleware” used to relay data and perform computations written in Java, and the main ”backend” software product written in C#. For simplicity, the logic tier and data tier of the C# application (from a general 3-tier architecture model) are abstracted in the bottommost layer, which is shown as the ”Business Logic and Data tier” in Figure 3-1.

For the backend or serverside components of the mobile application, the choices were:

1. **No automated testing:** All changes by developers would be code reviewed but no testing would be performed at build time. This was intended to shorten build times and deploy to a test environment sooner after a change is made to the software.
2. **Integrated unit tests with mock data:** This was already done for several projects belonging to the serverside tier; the mock data would be derived from sample expected given by the specification, usually a defined JSON object. This mock data would need to be manually updated regularly as new formats expected from new features are implemented.
3. **A separate unit test framework:** This option differs from the one above, as the unit test framework is intended to be flexible enough to test across the multiple service tiers and even products. It would not require mock data, but instead compare the data retrieved from the C# tier to the Java tier, to verify its integrity.
4. **Postman REST client:** An API testing tool with automated features developed and maintained by a startup company, with an intuitive user interface and support for sharing test collections and suites. All test cases would need to be written in JavaScript, for the automated ”runner” feature to execute them.

For the frontend (actual mobile application), the following were considered:

1. **Web browser tests (Selenium):** Since modern browsers such as Mozilla Firefox and Google Chrome provide WebDriver tools for developers, they can be used in simulating use cases of the application.
2. **On physical device:** No additional measures will be added as part of the pipeline, as the application would be pushed onto Android and iOS testing devices at the end of a successful build, through the HockeyApp distribution system [5].
3. **On Android and iOS emulator:** The mobile application will be emulated in place of a physical device, which can be done on most modern desktop machines. For emulating the iOS app however, a Mac computer will be required, since there is currently no official support for Windows computers.

A weighing scheme was chosen according to the importance of each criteria in the con-

Table 3-1. Criteria weighing

Criteria	Auto.	Setup.	Comp.	Cover.	Maint.	Re-use.	Feedback
Weight	25%	12%	13%	15%	5%	10%	15%

siderations of this project, which is shown on Table 3-1. Each criteria of an option was then assigned a numerical score from 0 to 10 based on qualitative analysis, with 0 being the lowest and 10 being the highest score; this is the number to the left of the arrow ($\rightarrow$). Applying the weight to each score by multiplying the percentage weight (ex. score of 5 with a 15% weight would yield 5 * 0.15 = 0.75), then multiplying by 10 gives the normalised score on the right hand side of the arrow (0.75 * 10 = 7.5). The total score for each option was obtained by summing the normalized score of each criteria across the row of the table.

3.2 Comparison of options for backend testing

The main purpose of testing the backend is to ensure that data transferred between the main C# application and Java middleware remains consistent. As previously discussed, the middleware was designed to alleviate the workload done by the mobile app. Such workloads include sorting, formatting, and collating data from different web services abstracted in the main application tier, before it is sent to be displayed on the client or frontend [1]. Since the API supports many functionalities, a unit testing approach is suitable.

The first option of not performing testing was given the lowest score in automation, coverage, and feedback since it would not perform any of these functions. Setup and maintenance received the highest score as no extra effort will be required in simply doing nothing.

The second option of testing against mock data received mid-to-high scores for setup time, maintenance effort, and feedback. It received low scores for Re-usability and Coverage since the mock data would need to be changed accordingly to test different tiers or components of the application.

The third option offered the best in automation and providing feedback, as a comprehensive test framework can incorporate different libraries for these functions. It also has better Re-usability since it's able to interface with multiple tiers of the software system, at the

[1] In most of this report, "backend tests" and "API tests" can be used interchangeably. It is only "frontend tests" that have further distinction between "UI" and "Wiring" tests.

Table 3-2. Qualitative evaluation of backend testing options

	Auto.	Setup.	Comp.	Cover.	Maint.	Re-use.	Feedback	**Score**
1. No tests	$0 \rightarrow 0$	$10 \rightarrow 12$	$10 \rightarrow 13$	$0 \rightarrow 0$	$10 \rightarrow 5$	$5 \rightarrow 5$	$0 \rightarrow 0$	**25**
2. Integrated	$5 \rightarrow 12.5$	$8 \rightarrow 9.6$	$5 \rightarrow 6.5$	$3 \rightarrow 4.5$	$8 \rightarrow 4$	$2 \rightarrow 2$	$9 \rightarrow 13.5$	**52.6**
3. Framework	$10 \rightarrow 25$	$6 \rightarrow 7.2$	$5 \rightarrow 6.5$	$9 \rightarrow 13.5$	$6 \rightarrow 3$	$8 \rightarrow 8$	$10 \rightarrow 15$	**78.2**
4. Postman	$8 \rightarrow 20$	$7 \rightarrow 8.4$	$7 \rightarrow 9.1$	$6 \rightarrow 9$	$8 \rightarrow 4$	$9 \rightarrow 9$	$6 \rightarrow 9$	**68.5**

expense of computational power and effort in maintenance, as well as initial set up of the framework project.

The fourth option scored very well for Re-usability, Automation, and required little effort to Maintain due to Postman's test collection sharing and runner functionality. Its main appeal is how it allows developers and testers to build testing suites collaboratively with its intuitive user interface. Coverage and Feedback were given slightly lower scores since this tool only tested the API functionality, and ran on a single workstation computer.

3.3 Comparison of options for frontend testing

The main purposes of testing the frontend were: Firstly, to ensure that user interface elements in the mobile application were being displayed in the correct location. This would be referred to as the ”UI Tests” in Figure 3-1. Secondly, to ensure that the frontend mobile application communicates with the middleware properly. The scope of this type of testing would include fetching from and sending data to the middleware through HTTP GET and POST requests respectively, with the proper frontend JavaScript implementations. This would be referred to as the ”Wiring Tests” in Figure 3-1.

The first option of testing using a web browser received the best scores in Automation, Coverage, Re-usability, and Feedback. This is so due to the available web drivers and built-in developer tools which are present in most modern browsers (such as the debug console in Chrome, Firefox, Safari). The only drawback would be that WebDrivers usually require more memory and computational power, especially when being run for extended periods of time in automated testing; this point is revisited in Section 6. The second option of testing on physical devices scored the highest in setup and maintenance, as after all, the process of installing the application natively is usually straightforward. Each build or iteration of the app coming from each successful continuous integration build job would be

Table 3-3. Qualitative evaluation of frontend testing options

	Auto.	Setup.	Comp.	Cover.	Maint.	Re-use.	Feedback	**Score**
1. Browser	$10 \rightarrow 25$	$8 \rightarrow 9.6$	$7 \rightarrow 9.1$	$6 \rightarrow 9$	$8 \rightarrow 4$	$9 \rightarrow 9$	$8 \rightarrow 12$	**85.7**
2. Device	$5 \rightarrow 12.5$	$9 \rightarrow 8.4$	$8 \rightarrow 10.4$	$10 \rightarrow 15$	$9 \rightarrow 4.5$	$7 \rightarrow 7$	$6 \rightarrow 9$	**66.8**
3. Emulator	$7 \rightarrow 17.5$	$5 \rightarrow 6$	$3 \rightarrow 3.9$	$8 \rightarrow 12.5$	$7 \rightarrow 3.5$	$5 \rightarrow 5$	$7 \rightarrow 10.5$	**58.9**

pushed onto the device though the HockeyApp distribution system, which could provide crash reports, logging and feedback during manual testing. Full coverage of tests would also be achieved as the end user would ultimately interact with the application through a physical device. However, automation of tests wouldn't be achieved on physical devices as easily[2].

The third option of using an emulator to test scored better than the browser in coverage as the application is tested in the native environment (Android, iOS). However, this option received lower scores for setup time and computational power requirement, since the emulator usually depends on an SDK installation (Android SDK or Xcode for iOS). In addition, certain hardware capabilities present on physical devices wouldn't be available for testing, such as the camera, fingerprint scanner etc...

3.4 Possible solutions

After performing the qualitative analysis for the frontend and backend tests, possible solutions were constructed using one option from each of the frontend and backend layers:

1. **Emulator frontend with Postman backend:**
   Using a local environment on a desktop or laptop, the mobile app would be run within an emulator for testing. This entailed that the corresponding SDK or software development kit would need to be installed on all machines used to test the frontend beforehand, and that Windows operating systems would not have access to the iOS simulator [3]. Testing of the backend API would be accomplished with the Postman REST client's collection runner feature, with separately maintained test scripts.
2. **Device frontend with Test Framework backend:**
   The mobile app would be tested on a physical Android and iOS device, and crash

[2]The device testing here refers to exploratory and/ or stress testing; Appium will be considered together with Selenium for simplicity.
[3]Official support for iOS and Xcode development from Apple is absent on Windows machines, as of the time this report was written

reports could be collected through HockeyApp. This would best represent the end user experience. Backend testing would be run on the continuous integration server as JUnit [7] tests.

3. **Browser frontend with Test Framework backend:**
   The mobile app will be tested using the Selenium webdriver tool in the form of unit tests. This would cover most functionalities of the app with the exception of device specific features, such as multitasking switching or gestures/ swipe-to-go back on iOS. Backend testing would again be run on continuous integration servers. This solution can be implemented using a single Java unit test framework project since Selenium can be imported as a Java library, and the API tests can be separately categorized with JUnit [2].
4. **Browser frontend with Integrated tests backend:**
   The mobile app will be tested using the Selenium WebDriver as per the previous option. Mock data would be generated and used to test the backend layer instead, according to the expected format of data that the middleware and main application returns.

Ultimately, solution 3 was selected as the proposed solution for testing against the mobile application, based on the qualitative analysis score; the following section discusses the benefits and drawbascks of this particular combination.

4 Analysis of proposed solution

As determined from the qualitative decision matrix from section 3, the best option for backend and frontend testing was with a separate test framework. This solution has great maintainability by having one unified collection of unit tests, as opposed to having two independent solutions for frontend and backend testing. Tests could also be categorized by the level of coverage. For example, one may wish to perform ”smoke testing”, where only the essential functions of the application are checked. In the case of Ultimate Software’s mobile application architecture (Figure 3-1), the backend was comprised of a main C# application server, and a Java middleware server; both communicate via transferring JSON data over HTTP requests. An example automated unit test for the API would take following form: Fetch data from C# server, fetch data from Java server, compare the two data objects for discrepancies. Examples of such discrepancies could be precision of decimal values

(12.3400 to 12.3), truncated or untrimmed string literals (”Alice ” to ”Alice”), or differing binary data (such as images).

Frontend UI testing on the other hand was automated using Selenium Webdriver tools, by emulating user flows within the application. For instance, when the user starts the application, the login screen should appear prompting for their username and password in the respective fields, as well as a ”sign in” button in the appropriate location. Verifying this was achieved by searching for the appropriate WebElement and checking if it is visible, either by XPath or tag attribute, which can be found using any modern web browser’s debugging functionality.

Wiring tests are written in a similar style to the UI tests, but concerns more with the data and formatting rather than the positioning of visual elements and any user interface bugs.

Since the middleware is implemented in Java, it is reasonable to reuse classes that define the object model of the resultant JSON in the test framework; details of configuration are discussed in Appendix A and Section 4.1. This reduces the number of duplicate classes and overall increases the maintainability of the test framework.

4.1 Setup

The proposed strategy in figure Figure 4-1 required additional jobs (UI, Wiring, and API testing) to be scheduled on the continuous integration server that would run periodically, most likely daily. This ensured that developers would be notified of any breaking changes at least once per day, and was configured such that any test failures would be reported to the author of the latest change through email. The Maven buildsystem was used to manage the build processes on the continuous integration server. A build agent would fetch the latest source code from version control (Git repository), then download the necessary artifacts for the Java Unit tests to compile and run. These artifacts include the Selenium WebDriver and shared classes from the Java middleware, depending on the type of test being executed, which can be specified by running Maven with command-line parameters [8]. For details on Maven configuration, please refer to Appendix A.

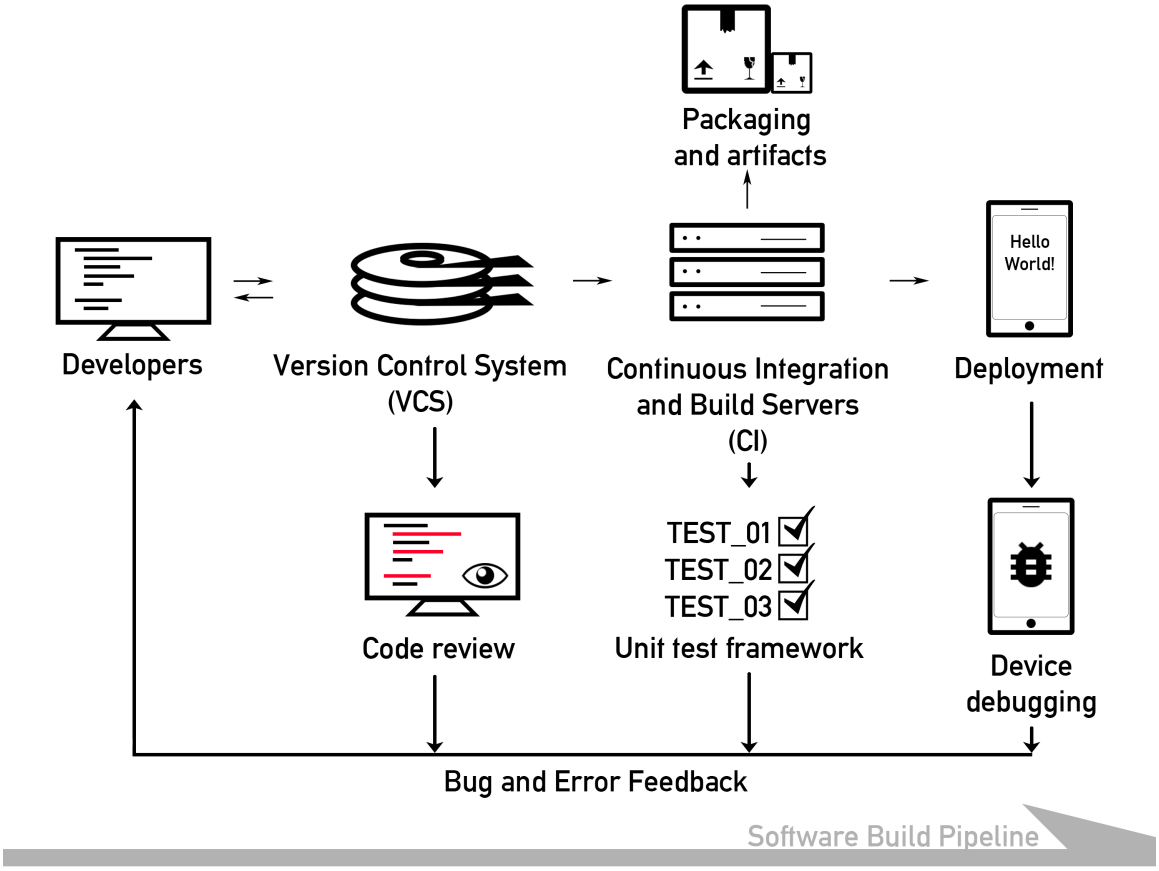

Figure 4-1. The software development build pipeline with proposed testing process

5 Performance

A total of 27 backend test cases and 60 frontend test cases and test flows, including Wiring tests, were written in collaboration with other QA team members initially. 6 build agents were set up as Ubuntu Linux virtual machines, each configured with 4 CPU cores and 16 GB of memory. The backend test suite took around 5 minutes to complete, whereas the frontend tests took nearly 4 hours to finish when it ran on a build agent. At times, several build agents had exhausted the default memory allocated by the JVM, though this issue was remedied by refactoring and is described in the following section.

The proposed solution successfully fulfilled all the requirements described in section 2.1 with through the following characteristics:

Automation: the test suites were set up on the team's continuous integration servers to run periodically, with logging enabled to document any errors and failures in each test case, so that testers may manually reproduce potential bugs.

Performance impact: since the frontend test suite took a relatively long time to complete, it was scheduled to run everyday at 2 AM, when the servers would experience less loads compared to during daytime. The backend test suite was also scheduled daily, but could be run as needed since it took little over 5 minutes.

Compatibility: the test framework project was able to reuse code from the Java middleware project, by specifying project dependencies in the Maven buildsystem configuration file, "pom.xml". Maven is preferred for managing large software projects with many dependencies for this purpose, and is widely adopted in the software industry.

Feedback: whenever a compiler error or test failure occurred, the continuous integration server was configured to automatically send an email to notify the author of the last code change.

6 Additional optimizations and improvements

In order to improve the performance and re-usability of this automated test framework, a series of code refactoring methods were applied to the project as the number of unit tests grew. This ranged from commenting and documenting parts of code to implementing appropriate software design patterns. Accompanying code with comments provides insight into the workings of the existing solution, allowing future collaborators to understand and if necessary, improve upon it. Software design patterns offer a set of guidelines or "best practices" in writing code, and saves developers' time by improving its maintainability and re-usability. Often in software, an easy-to-implement solution is not the most maintainable one in the future. This leads to "Technical Debt" accumulating [12], as developers choose the former over the latter, for reasons such as convenience or in order to meet deadlines.

In the example mentioned in section 5, a memory leak was discovered when running the frontend test suite. Further investigation led to the discovery that every test case created a new instance of HttpClient class, which was found to be inefficient upon review of the

Apache documentation [6]. Therefore, the test framework was refactored accordingly to use the singleton pattern for creating HTTP connections, such that each test case may reuse the same instance of that class. The result was an effective halving of the frontend test suite runtime, from about 4 hours to 2 hours (Table 6-1). The backend tests did not benefit much from this refactoring (Table 6-2) since the test cases used a static class to establish HTTP connections to the backend services, and these connection instances were already being reused.

Table 6-1. Time taken to run frontend tests before and after refactoring

	Frontend tests runtime (Before)	**Frontend tests runtime (After)**
Trial 1	14186 seconds	7511 seconds
Trial 2	13824 seconds	8731 seconds
Trial 3	13596 seconds	8039 seconds
Trial 4	13691 seconds	8129 seconds
Average	13824.25 seconds	8102.5 seconds

Table 6-2. Time taken to run backend tests before and after refactoring

	Backend tests runtime (Before)	**Backend tests runtime (After)**
Trial 1	351 seconds	384 seconds
Trial 2	346 seconds	393 seconds
Trial 3	413 seconds	356 seconds
Trial 4	374 seconds	332 seconds
Average	371 seconds	366.25 seconds

Conclusions

From the analysis conducted on each alternative solution leading up to the selection of the proposed solution, it was concluded that the best option for testing against Ultimate Software's multi-tier mobile application was to use a separately maintained unit test framework. This framework would be a collection of Java unit test cases (JUnit) for verifying each feature of the application, and tests would be categorized by type (API, UI, or Wiring Figure 3-1) with the JUnit categories function. Specifying command line options when running Maven on the continuous integration servers would allow for running each type of test separately, then scheduling each job ensures that each type of test would run at least once per day. This solution successfully satisfied every requirement from Section 2 as determined from the qualitative analysis and performance results, and is also one that can be extended to test other multi-tier software products from Ultimate Software.

Recommendations

Based on the analysis and conclusions in this report, it is recommended that other teams collaborating with the mobile team at Ultimate Software adopt and expand this framework for their own testing purposes, so that it would become a standard method for testing new features in the mobile application. There are a few items that could be extended beyond the scope of this report, either as improvements to performance or to increase the user-friendliness of this solution. Examples of such items to consider would include:

- Investigate or research a more lightweight backend testing solution with Newman [9], which is a command-line compatible interface of the Postman [10] runner.
- Create comprehensive documentation and guidelines for how to use and contribute to the test framework, so that future testers have a proper standard to model their tests after.
- Create a Twitterbot script to post build and test failures from the continuous integration server, to a Twitter feed for the entire development team to see.

Glossary

API: Application Program Interface, a set of defined subroutines of a software application that allows other components or applications to interact with the software..

HTTP: HyperText Transfer Protocol, a protocol definition or rules for transferring and formatting of messages and data between information systems.

JSON: Java Script Object Notation, a standard format for interchanging and generating data by software applications, that is also human-readable.

JVM: Java Virtual Machine, the software that allows computers to execute compiled Java program.

QA: Quality Assurance, a team or department in software development concerned with the testing of software, with the goal of delivering bug-free software to customers.

REST/RESTful: Representational State Transfer, an architecture style in designing networking software for web services to serve web resources (such as JSON and XML data).

UI: User Interface, the means or components of software that the user will use to interact with the application, usually with visual or graphical elements.

WKRPT: Work-term report; the acronym used by the University of Waterloo Undergraduate Calendar.

XML: Extensible Markup Language, a human and machine readable format for encoding data, using enclosing tags.

XPath: Syntax for defining parts of an XML document, using path expressions to navigate the XML document. Expressions can be used to select nodes or node-sets in an XML document, such as web elements in an HTML document.

References

[1] *Apache Subversion Design Spec*. URL: http://svn.apache.org/repos/asf/subversion/trunk/notes/subversion-design.html#goals.

[2] S. Arod. *JUnit4 Categories*. URL: https://github.com/junit-team/junit4/wiki/Categories.

[3] K. Fogel. *Open Source Development with CVS, 3rd Edition*. 2000. URL: http://cvsbook.red-bean.com/cvsbook.html#Introduction.

[4] *Git*. URL: https://github.com/git/git.

[5] *HockeyApp*. Microsoft. URL: https://hockeyapp.net.

[6] *HttpClient Performance Optimization Guide*. Apache Software Foundation. URL: http://hc.apache.org/httpclient-3.x/performance.html.

[7] *JUnit4*. JUnit. URL: http://junit.org/junit4/.

[8] *Maven Tests*. Apache Software Foundation. URL: https://maven.apache.org/surefire/maven-surefire-plugin/examples/single-test.html.

[9] *Newman*. https://github.com/postmanlabs/newman/blob/release/2.x/README.md. 2016.

[10] *Postman*. Postdot Technologies, Inc., 2016.

[11] *Selenium*. SeleniumHQ, 2000. URL: http://www.seleniumhq.org/.

[12] Chris Sterling. *Managing Software Debt: Building for Inevitable Change (Agile Software Development Series) 1st Edition*. Addison-Wesley, 2010, pp. 18, 30.

Appendix A Maven Configuration

Maven:

There were two configuration files that were edited for Maven to source the proper artifacts: pom.xml and settings.xml. Both files resided in the root directory of the Java test framework project, such that the build agent could read the file at build time.

settings.xml:

The highlighted lines represent additions of repositories made to the file, that would allow Maven to source artifacts from the proper hosted servers. Note that there can be multiple repositories under the ¡repositories¿ parent tag, which could serve to separate artifacts by build type (a development build would use snapshots, rather than a release build for example), or provide redundancy in the event that one repository becomes unavailable.

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.1.0" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/SETTINGS
    /1.1.0 http://maven.apache.org/xsd/settings-1.1.0.xsd">
   <profiles>
      <profile>
         <repositories>
            <repository>
               <snapshots>
                  <enabled>false</enabled>
               </snapshots>
               <id>central</id>
               <name>libs-release</name>
               <url>RELEASE-ARTIFACT-REPO-URL</url>
            </repository>
            <repository>
               <snapshots />
               <id>snapshots</id>
               <name>libs-snapshot</name>
               <url>SNAPSHOT-ARTIFACT-REPO-URL</url>
            </repository>
            <repository>
               <snapshots>
                  <enabled>false</enabled>
               </snapshots>
               <id>central2</id>
               <name>Maven Repository Switchboard</name>
               <url>http://repo1.maven.org/maven2</url>
            </repository>
         </repositories>
         <pluginRepositories>
            ...
         </pluginRepositories>
         <id>artifactory</id>
      </profile>
   </profiles>
   <activeProfiles>
      <activeProfile>artifactory</activeProfile>
   </activeProfiles>
</settings>
```

pom.xml:

The highlighted lines show how artifacts published by another software project can be added. In this case, the dependency was a collection of Java classes shared by the middleware. It is ideal to re-use these artifacts whenever possible to reduce duplicated code, which results in better maintainability of the overall test framework. Software libraries such as JUnit [7] and Selenium [11] (and Appium) are also imported this way, as seen in the following snippet:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
    maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.ultimatesoftware.qa-project</groupId>
   <artifactId>qa-project</artifactId>
   <version>0.2.0</version>
   <name>QA test framework project</name>
   <build>
      <plugins>
         <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
               <source>1.8</source>
               <target>1.8</target>
            </configuration>
         </plugin>
      </plugins>
      <testResources>
         <testResource>
            <directory>src/test/resources/Mobile</directory>
         </testResource>
      </testResources>
   </build>
   <description>QA for backend services for the mobile app.</description>
   <dependencies>
        ...
      <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
         <version>4.12</version>
      </dependency>
        ...
      <dependency>
         <groupId>org.seleniumhq.selenium</groupId>
         <artifactId>selenium-java</artifactId>
         <version>2.53.0</version>
      </dependency>
      <dependency>
         <groupId>com.ultimatesoftware.mobile-middleware</groupId>
         <artifactId>mobile-middleware</artifactId>
         <version>0.7.0-SNAPSHOT</version>
      </dependency>
        ...
      <dependency>
         <groupId>io.appium</groupId>
         <artifactId>java-client</artifactId>
         <version>4.0.0</version>
      </dependency>
        ...
   </dependencies>
    ...
</project>
```