LAPPEENRANTA-LAHTI UNIVERSITY OF TECHNOLOGY LUT

School of Engineering Science

Software Engineering

AUTOMATED TESTING OF REACT NATIVE APPLICATIONS

Examiners:	Professor Jari Porras
	Associate Professor Ari Happonen
Supervisors:	Associate Professor Ari Happonen
	M.Sc. (Tech.) Niko Sten

ABSTRACT

Lappeenranta-Lahti University of Technology LUT
School of Engineering Science
Software Engineering
Matias Salohonka

Automated testing of React Native applications
Master's Thesis 2020

93 pages, 15 figures, 2 tables, 29 listings, 6 appendices
Examiners: Professor Jari Porras
Associate Professor Ari Happonen
Keywords: React Native, Azure DevOps, testing, test automation

Testing is an important part of quality assurance. In software engineering, testing can be applied on many levels and by different techniques. Testing is recognized as a key contributor to successful and efficient development of software. Test automation is a practice where the testing effort is shifted from people to software. This allows new types of testing to emerge and raises the efficiency of other types of testing. In this thesis, a design science approach is taken to implement tests for a React Native mobile application and to automate testing in the Azure DevOps cloud environment. The research creates a practical testing solution for an application with a long development and maintenance roadmap. The goal is to identify company needs for testing and automation, gain an understanding on the testing and automation methods and techniques from literature, implement proof of concept tests for every part and conceptual level of the application, and automate these tests in an effective way. The results of this thesis are the code listings, configurations and explanations that account for the theoretical framework, and describe in detail how to implement the testing and automation of a React Native application with Azure DevOps, and explains which techniques are generalizable to other types of development besides mobile applications.

TIIVISTELMÄ

Lappeenrannan-Lahden teknillinen yliopisto LUT
School of Engineering Science
Tietotekniikan koulutusohjelma
Matias Salohonka

React Native sovellusten automatisoitu testaus
Diplomityö 2020

93 sivua, 15 kuvaa, 2 taulukkoa, 29 katkelmaa, 6 liitettä

Työn tarkastajat:	Professori Jari Porras
	Associate Professor Ari Happonen
Hakusanat:	React Native, Azure DevOps, testaus, testausautomaatio
Keywords:	React Native, Azure DevOps, testing, test automation

Testaus on tärkeä osa laadunvarmistusta. Ohjelmistotuotannossa testausta voidaan käyttää monilla eri tasoilla ja tekniikoilla. Testaus on havaittu yhdeksi tärkeäksi onnistuneen ja tehokkaan ohjelmistokehityksen edistäjäksi. Testiautomaatio on käytäntö, jossa testauksen suoritus siirtyy ihmisiltä ohjelmistoille. Tämä mahdollistaa uudenlaiset testaustyypit ja tehostaa muiden testaustyyppien suoritusta. Tässä diplomityössä käytetään design science -menettelyä React Native mobiilisovelluksen testaukseen ja Azure DevOps pilviympäristössä testiautomaation toteuttamiseen. Tutkimus tuottaa käytännöllisen testausratkaisun sovellukselle, jolla on pitkä kehityksen ja ylläpidon etenemissuunnitelma. Tavoite on tunnistaa yrityksen tarpeet testaukselle ja automaatiolle, luoda käsitys testauksen ja automaation menetelmistä ja tekniikoista kirjallisuuden avulla, toteuttaa esimerkkitestit jokaiselle sovelluksen osalle ja käsitteelliselle tasolle, ja automatisoida nämä testit tehokkaasti. Tämän diplomityön tulokset ovat koodilistaukset, asetukset, ja selitykset mitkä kattavat teoreettisen kehyksen ja kuvaavat yksityiskohtaisesti, kuinka toteuttaa testaus ja testiautomaatio React Native sovellukselle Azure DevOpsin kanssa, ja esittää mitkä tekniikat ovat yleistettävissä mobiilisovelluksen ulkopuolelle.

ACKNOWLEDGEMENTS

Many thanks to the people involved in my thesis work. This thesis was an interesting topic and provided useful personal development and a very practical benefit to Valamis.

Pauliina Kulla, Niko Sten, Evgeny Matyukhin, Juho Anttila, and everybody else who answered my questions and provided feedback and insight at Valamis.

Ari Happonen for always being quick with the responses and having very effective and good feedback on multiple aspects of the thesis content and work.

My parents for supporting me and reminding and motivating me about what the thesis process and getting a degree stands for.

Shanice for being the most supportive person imaginable, even when the technical details I explain didn’t make sense, you would congratulate me on the progress, keep me motivated and remind me of how close I am to finish my journey. And of course, scolding me for sleeping in during my study leave.

Thank you.
- *Matias, February 2020, Lappeenranta*

TABLE OF CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
AWS	Amazon Web Services
CD	Continuous Deployment
CI	Continuous Integration
HTTP	Hypertext Transfer Protocol
JDK	Java Development Kit
JNI	Java Native Interface
JS	JavaScript
LXP	Learning Experience Platform
MVP	Minimum Viable Product
OS	Operating System
POC	Proof of Concept
RN	React Native
SDK	Software Development Kit
SQA	Software Quality Assurance
UI	User Interface
UML	Unified Modeling Language
VDOM	Virtual Document Object Model
VSTS	Visual Studio Team Services
YAML	YAML Ain’t Markup Language

1 INTRODUCTION

This is a master’s thesis in software engineering that will describe a subset of software quality assurance: software testing and test automation. The goal is to identify and describe the different types of testing for a given mobile application, investigate how to conduct the types of testing, and finally to implement examples of the tests to facilitate the company to implement full testing of the application. This section will provide an overview of the thesis, with background information to the topic, goals and delimitations, and structure of the contents.

1.1 Background

Software quality has been defined by many people. Tian (2005) suggests, that quality can be observed from many different perspectives and be shaped by varying expectations. For the consumer (or user) of the software, quality is condensed into fit of use and reliability, that is, the software does what is needed and functions correctly with repeated use. Tian expands especially fit for use into a wide variety of factors, such as ease of use. On the other hand, producers have a different understanding of software quality. Customer satisfaction and contractual fulfilment are key factors for service and product managers, while maintenance personnel value maintainability and people involved with service value modifiability. Software quality assurance (SQA) is the field in software engineering dedicated to establishing and maintaining quality in software development. IEEE (2010) defines quality assurance as:

1. *A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements.*
2. *A set of activities designed to evaluate the process by which products are developed and manufactured.*
3. *The planned and systematic activities implemented within the quality system . . . to provide adequate confidence that an entity will fulfil requirements for quality.*

4. *Part of quality management focused on providing confidence that quality requirements will be fulfilled.*

These definitions leave a broad set of activities (especially definition two, which specifically references processes instead of products and artefacts) that are tied to quality, but arguably the most common and integral part of SQA is testing. Testing is defined as “activity in which a system or component is executed under specified conditions, the results are observed or recorder, and an evaluation is made of some aspect of the system or component” (IEEE, 2010). The evaluation part of the definition is crucial: a test must have an expected outcome, which can be compared to the result of the test. This lets the person conducting the test to determine whether the software is defective or not. Testing is prevalent in many industries, but the immaterial and flexible nature of software makes testing very effective at finding and reducing defects. This has led to testing becoming the primary means of detecting and fixing software defects (Tian, 2005).

In the general industrial context, digitalization has driven multiple industries towards “automate or die” decisions, making then to robotize, automate and digitalize their own processes and solutions they offer for their customers (Kortelainen, et al., 2019; Minashkina & Happonen, 2018; Minashkina & Happonen, 2019). As in all other industries, it is the case in software industry, that the automation is one of the key contributors of performance in the software industry. Automation of development life cycle processes leads to better quality software and by extension to higher customer satisfaction and business profitability (Kumar & Mishra, 2016). Organizations engaging in high-performance automation have observed that the balance of speed and stability in development is not a zero-sum game, but rather both are dimensions of quality that automation (which is a key part of DevOps) can enable. Testing is one of these processes and automating it is a proven contributor in the improvement and performance capabilities of software development and delivery, most visibly allowing development time to shift from unplanned work and rework (i.e. fixing bugs and defects) to more new work. (Forsgren, et al., 2018)

Valamis is a company in the software industry, specializing in e-learning software. The company’s namesake product is a learning experience platform (LXP), a cloud-based software targeted towards large organizations and enterprises globally to train and educate their workforce. The product is mostly used via web-application on computers currently. Valamis also conducts service business, but this thesis focuses on product development. Valamis is developing a new mobile application for their product to support a wider demographic with more varied use cases. The mobile application expands the availability of the e-learning content anywhere, anytime and any device. One key motivation is also to differentiate in the market, as not all competitors to Valamis' LXP offer mobile application client software. This led to the decision to have a fresh start to creating a true mobile experience (as compared to using a responsive web-application on a mobile device) for their product in July of 2019. Development effort was focused on creating and publishing a minimum viable product (MVP) as soon as possible. The scope for the MVP application was kept narrow, which indirectly left SQA without the attention necessary. The application has a long roadmap planned for it, so SQA and especially testing is key to ensuring reliability and maintainability for the application in the future.

For an application that interfaces with the LXP and aims to provide a comparable feature-set as the primary web interface, it must implement several distinct functionalities. Important features that were planned for the MVP include viewing embedded presentations, which may include videos, send analytics data from said presentations, track and further user progress in learning content entities, submit assignments, join events, and function offline with downloaded content. The application will be released under the name "Valamis" in both Android and iOS application stores.

Because the project has been in development before meaningful SQA effort was made, some technical decisions are solidified and therefore are constraints for this thesis. React Native (RN) is a mobile application framework that is used to develop multiplatform native mobile applications. RN is explained further in section 2.1. During 2019, Valamis started moving its version control, task management and automation infrastructure from Gerrit and Atlassian Jira to Azure DevOps. Azure DevOps is explained further in section 2.2.

1.2 Goals and delimitations

The goals of this thesis are divided into testing and automation:

1. Testing goals:
   1.1. Find out with the company the level of testing that is feasible for the application,
   1.2. Identify testing techniques that satisfy the agreed upon level of testing,
   1.3. Implement proof-of-concept tests
2. Automation goals:
   2.1. Find out with the company the level of test automation that is feasible for the application,
   2.2. Identify automation techniques that satisfy the agreed upon level of automation,
   2.3. Implement a test automation suite.

The set of goals is reached by exploring with experts from the company the level of testing and automation that is cost-effective for the mobile application development and maintenance. Once the level of reasonably maintainable testing is established, techniques are identified that satisfy the minimum level of benefits agreed with the company. This is achieved by collecting information and requirements on techniques from literature and company expert statements. Once the techniques are identified, test cases are requested from the company and proof of concept (POC) implementations of the tests are made to serve as a guide for future full implementation. The POC implementations are complete, the automation to run the tests is specified. Documentation of Azure DevOps is used to identify the techniques to satisfy the minimum level of automation agreed with the company. The test automation is implemented.

Delimitations and constraints of this thesis are divided into three categories: company-driven, mobile-driven, and SQA-driven. The company delimits the scope of the thesis to the technologies they have chosen to use in the application. Tests, tools and frameworks must be compatible with RN and the automation is implemented with Azure’s specification. The mobile specificity means that techniques inherent to other areas of software development and testing are not covered. Lastly, SQA is a wide field of practice and research, which contains subjects such as process definition, improvement and management, that are omitted. The process-oriented side of SQA for the mobile application development is being improved and executed elsewhere in the company.

1.3 Structure of the thesis

Section 2 describes the technical constraints set for the thesis based on the technologies the company has chosen to use. Section 3 presents the research methodology and the initial set of requirements elicited from the company for the project. Section 4 provides a literature-driven, theoretical investigation on the different types of testing that are relevant for the work. In section 5 test automation is presented on a general level, why, how and when it should be done. Section 6 is the empirical part of the work. It contains the descriptions and listings of the tests and automation that were implemented for the mobile application. Finally, section 7 contains the discussion and conclusions of the thesis based on the work done earlier.

2 PRACTICAL CONSTRAINTS FOR IMPLEMENTATION

This section describes the two major technologies that are fixed. These technologies provide the field of possibilities and limitations that relate to testing and automation. As the impact of these technologies on the selection of tools and frameworks in the implementation is considerable, some basics are explained to help justify the selections.

2.1 React Native

React Native is an open source mobile development framework created by Facebook in 2015 (Facebook, 2020a). It is based on a well-known web-development library React. A goal at Facebook for React Native was to extend the mechanisms of React to native mobile application development, and eventually to all native development platforms, leading a philosophy of “learn once, write everywhere” (Occhino, 2015; Zammetti, 2018). Originally RN was used for Apple’s mobile operating system (OS), iOS, but Android support was added soon after (Danielsson, 2016). The true multiplatform status of RN was solidified in 2016, when Microsoft and Samsung added support for Windows and Tizen platforms respectively (Zammetti, 2018).

Development in RN is different from traditional native development on the iOS and Android platforms. The patterns and structure in RN are fundamentally the same as in React, which is used in web applications; the application is structured into a Virtual Document Object Model (VDOM), that keeps track of the screen elements and their hierarchies. In React, the elements in the VDOM are defined by components. There are several primitive components, like Text and Image, that can be composed into combinations and collections of components. Components can be reused and be given variable properties to customize their content. The process of these components being realized into actual elements (Hypertext Markup Language (HTML) elements in plain React for web, or native elements in RN) is called rendering. The VDOM enables efficient management of the user interface (UI), as small changes in the UI – like changing the value of one field in a form – can be handled by comparing differences in the lightweight VDOMs of before and after the change, and then only updating the relevant, changed parts in the real UI. This prevents having to do complete recalculations of layout and elements every time something changes in the UI. (Akshat & Abshishek, 2019; Zammetti, 2018)

The programming language used is JavaScript (JS) for all platforms and most JS code can be shared for all platforms. In traditional native development, codebases cannot be directly shared between platforms, as the structures and languages used are different, notably in that iOS development uses Objective-C and Swift languages and Android development uses Java and Kotlin languages. (Zammetti, 2018)

The use of native platform languages such as Swift and Kotlin is possible in a RN application. Native languages are separated into their own modules, called native modules. There are cases where some functionality present in the underlying native platform does not have a RN JS implementation, or for example there is a pre-existing module that could be re-utilized. The mechanism to support this in RN is called native bridge. In short, it is an abstraction layer between JS and the native platform. RN runs JS in its own thread (with other threads being used for running the native UI, a threaded queue for layout change calculations, and separate threads for any native modules (Akshat & Abshishek, 2019)). Any JS code that needs to run native actions, such as updating the UI, is evaluated in C by JavaScriptCore, a C/C++ and JS compatibility framework built by Apple. These native actions are stored in a queue to prevent halting the action if the native side cannot immediately process the action, which makes RN asynchronous by nature. Once the action is removed from the queue at the native side, it is processed at the native side in a platform-dependent way. iOS is Objective-C based, which being a superset of C, can handle the actions without any trouble. Android uses Java Native Interface (JNI) to turn the C-based actions into Java compatible. This whole chain of actions that make up the native bridge is presented in Figure 1. (Nivanaho, 2019; Zagallo, 2015; Frachet, 2017)

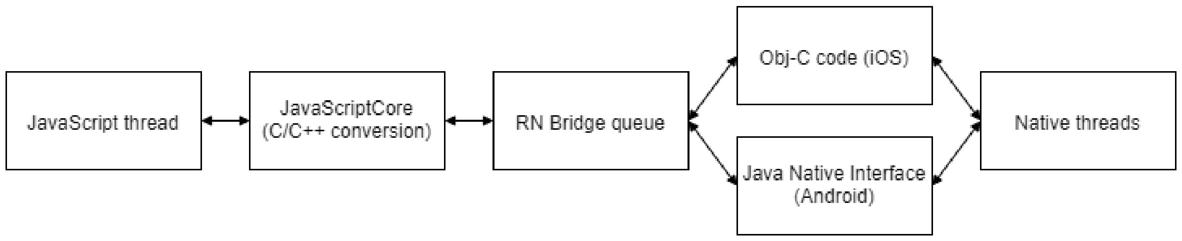

JavaScript thread
JavaScriptCore (C/C++ conversion)
RN Bridge queue
Obj-C code (iOS)
Java Native Interface (Android)
Native threads

Figure 1*: Relation of JavaScript and native threads in React Native, adapted from Frachet (2017)*

Valamis had several reasons to choose RN as their approach to mobile application. A clear goal of targeting multiple platforms was established from the start of the project. This goal had several factors contributing to it: customer coverage and resourcing. While the most urgent customers require exclusively iOS support, which made it the primary platform for development and testing for the MVP release, Android was recognized early on as a major end-user segment that must be covered. The resourcing factors are further divided into two problems: limitations in competences and limitations in availability of manpower. Valamis had next to none in-house experience of developing with native mobile technologies, and this problem would only be amplified by the fact that Android and iOS native development skillsets are separate and expertise in one does not directly translate into the other, which would effectively lead to having two development teams; one for each platform. The limitations in resourcing would spread the hypothetical teams too thin, possibly to even one dedicated developer per platform if done in parallel. The supporting factors to picking RN among the options of multiplatform technologies stemmed from having plentiful React competence inside the company from web development. A fresh start gave the opportunity to choose any technology, and that is what RN excels at. Amazon backed Twitch made remarks that RN is great for new applications, but adapting or integrating existing applications to RN is difficult and in most cases not cost effective (Twitch, 2017).

2.2 Azure DevOps

Azure DevOps is “a cloud service for collaborating on application development” (Rossberg, 2019). It is a collection of services made by Microsoft, and it was introduced in 2018. Previously it was known as Visual Studio Team Services (VSTS), but rebranding was done to move it under Microsoft’s Azure cloud computing services (Rossberg, 2019).

Azure DevOps offers a multitude of tools aimed at developers and managers working on software projects. Planning work and tracking defects and issues in Scrum and Kanban workflows is offered by Azure Boards. Code source control and collaboration on Git is possible with Azure Repos. Automated building and releasing services and facilitation of continuous integration (CI) and continuous deployment (CD) are offered by Azure Pipelines. Testing tools for manual, automated and load testing can be done with Azure Test Plans. Sharing bespoke or internal development packages and libraries can be done with Azure Artifacts. Finally, wiki-pages can be created for documentation and communication for a project, and customizable dashboards and administrative services are available for higher level of control. These features are summarized in Table 1. For the purposes of this thesis, the relevant feature offered by Azure DevOps is Pipelines. (Rossberg, 2019; Microsoft, 2019b; Microsoft, 2019a)

***Table 1**: Summary of Azure DevOps features, adapted from Microsoft (2019a)*

Azure DevOps feature	**VSTS feature**	**Description**
Azure Pipelines	Build & Release	Automation, CI/CD and releasing
Azure Repos	Code	Git repositories
Azure Boards	Work	Work tracking: boards, backlogs and reporting
Azure Test Plans	Test	Planned and exploratory testing
Azure Artifacts	Packages (extension)	Package and library feeds

In early 2019 Valamis started transitioning to Azure DevOps from Gerrit. Gerrit is a Git-based code collaboration tool made by Google. This meant that new projects, where possible, would be managed on DevOps instead of Gerrit. Product development for the LXP, excluding the mobile application, is still hosted on Gerrit, with Jenkins as the automation suite that creates builds and runs tests. Valamis wants to move away from Gerrit and Jenkins because these systems are run locally, meaning dedicated servers must be rented or bought and maintained, and Azure DevOps allows further improvement of the automation tools used in the company. Modern cloud providers offer software-as-a-service solutions that replace the need for having one’s own dedicated servers. This created cost savings to Valamis from no longer having to maintain infrastructure, lessening the manpower and investment in hardware needed. The providers considered were Amazon Web Services (AWS), Microsoft Azure and Google Cloud Platform. At this point, the company already offered their LXP as a cloud-hosted option on Azure, which was recognized as having potential synergy to bring the development work to the same ecosystem. The cost differences between the providers were small enough that they were not a meaningful factor in the decision. Azure’s servers had more optimal geographical locations for customers at the time of decision. Valamis also used analytics tools made by Microsoft, which integrated the best with Azure. Additionally, AWS had compatibility issues with some of the systems Valamis used. Overall this led to the decision to pick Azure’s development platform DevOps as the next step forward. For the mobile application this meant that as a fresh project, it would be started right away on the new platform to avoid migrations.

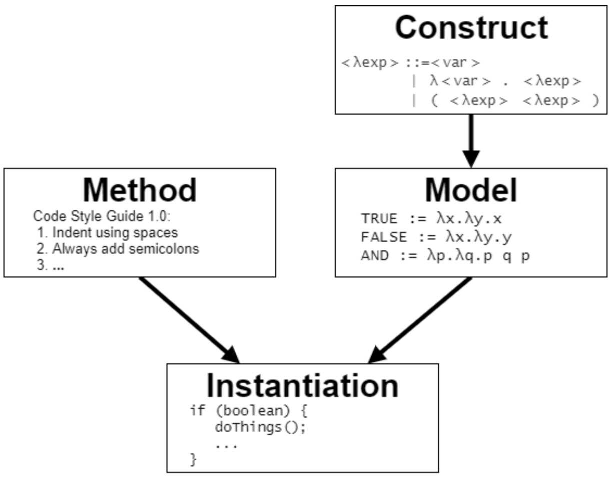
3 RESEARCH METHODOLOGY

This section describes the chosen research methodology and the motivations on the choice and the details of it. The sources of data are explained, and the results of the gathering are presented.

3.1 Design science

In software engineering and information systems research the study is split into behavioural science that predicts and describes the interaction of humans and artefacts (which are usually software and systems), and into design science that solves identified problems by creating and evaluating artefacts (Hevner, et al., 2004; Williamson & Johanson, 2017). Artefacts are classified into constructs, models, methods and instantiations. Constructs are definitions on how to describe problems and concepts, much like languages. They help to formulate and formalize problems, like how to draw a blueprint. Models are the application on constructs: they describe a specific problem or structures of other artefacts, like a blueprint for a house. Methods are processes and guidelines to achieving goals and solving problems. Methods prescribe how to create artefacts. They can be highly defined and formal, like algorithms, or loose and informal, like best practices. Instantiations are concrete systems and software that can be used in practice. Instantiations embed knowledge in them, which could come from a model. The different types of artefacts and their relationships are summarized in Figure 2. (Johannesson & Perjons, 2014)

Design science draws a contrast to traditional empirical research in that design science seeks to change, improve and most importantly create, not only to describe and predict (Johannesson & Perjons, 2014). If creation is seen as the first "half" of design science, then evaluation is the other "half." Evaluation must happen at least once, but often the design science process is iterative, so creation and evaluation take alternating turns (Johannesson & Perjons, 2014). Many sources agree that the artefact itself should be evaluated in at least one of possible criteria – most common being how well did the artefact solve the problem the research set out to do – but Williamson & Johanson (2017) suggest that the creation process itself and the problem definition can be evaluated too. Artefact evaluation can be done in a multitude of methods. Peffers, et al. (2012) categorized the different methods to evaluate an artefact and grouped them by the type of the artefact. They recognized that instantiations are commonly evaluated by prototyping and technical experiments. Prototyping means creating an implementation that demonstrates the utility or fit for use of the artefact. Technical experiment means testing an algorithm using various types of data to evaluate the technical performance.



```
<λexp> ::=<var>
        | λ<var> . <λexp>
        | ( <λexp> <λexp> )
```

Code Style Guide 1.0:
1. Indent using spaces
2. Always add semicolons
3. ...

```
TRUE := λx.λy.x
FALSE := λx.λy.y
AND := λp.λq.p q p
```

```
if (boolean) {
   doThings();
   ...
}
```

***Figure 2**: Types of artefacts produced by design science*

In the context of this thesis, design science was chosen as the research methodology because there is a need to build new things based on real world problems and ideas, and there exists supporting research called kernel theories, that can be leveraged to guide the design of the new artefacts (Johannesson & Perjons, 2014). The artefacts at hand are a collection of test tools and prototype test implementations that satisfy the specifically agreed upon company needs and the test cases provided by the company, and configuration artefacts (in the form of files and settings) to enable automation of the testing process on the company selected platform. To evaluate the artefacts, the technical performance, per se, is not the most valuable attribute to the company, but rather the utility of the artefact, so prototyping is the evaluation method of choice.

3.2 Data gathering

It is assumed that literature provides guidelines, background and best practices on testing and automation techniques. Different kinds of benefits and drawbacks of various types of testing yield the motivation and goals for each type. Opinions and needs from company experts will be gathered during the empiricism via communication. The communication is conducted by posing guiding questions to company developers on technical and practical aspects of the study. These opinions and needs may further limit the concrete choices in tooling and techniques, based on extendibility to outside of this application and integration to other tools outside the scope of this thesis. Both sources provide requirements for testing and automation that are ultimately combined. With the conceptual and technical requirements and limitations uncovered, documentation of testing and automation tools, libraries and frameworks guide the empiricism of the implementation.

3.3 Identified company needs

Valamis recognized that SQA, and more specifically testing and automating testing is important. Kasurinen (2013) supports this by claiming that testing is the most important activity is software engineering from a profitability standpoint, especially during the development phases of a project. As the company has limited experience in developing and maintaining mobile applications, the requirement for mobile application testing in specific was set to investigate the field and provide summaries from each topic, combined with goals applicable to general software development and SQA presented in the following paragraphs.

Parts of test strategy need to be defined for the project. The classic levels of testing were agreed that are necessary, starting with unit tests, with common unit test cases sourced from literature, integration tests that include integration of views, and systems tests in the form of end-to-end testing. Acceptance testing is recognized as an important level of testing, but as it is by definition done by the customer, it will not be covered in this thesis (Singh, 2012). For each level of testing, the specific techniques and tools must be defined, and where possible, tools and techniques already established in the company will be used. Kaner, et al., (2002) define a set of testing strategies, from which *practical* was chosen as the basis for this thesis. This strategy aims to define an extendable set that is not too broad for a small team and the scope of a master's thesis.

Test automation was agreed to be utilized as much as possible, therefore it will be applied to as many levels of testing as is possible with reasonable effort. The application development has a long roadmap, so efficiency is a key driver. Valamis recognizes that automation is necessary to establish and maintain this efficiency in SQA.

4 MOBILE APPLICATION TESTING

Testing in mobile applications draws many similarities to testing in other types of software. Levels of abstraction and purpose in testing can be summarized in the V-model of testing, presented in Figure 3. While the V-model is usually associated with the waterfall lifecycle model of software development (Mathur & Malik, 2010), Haller (2013) points out that the lifecycle phases in modern development has a lot more overlap between testing and development and between different levels of testing in the V-model.

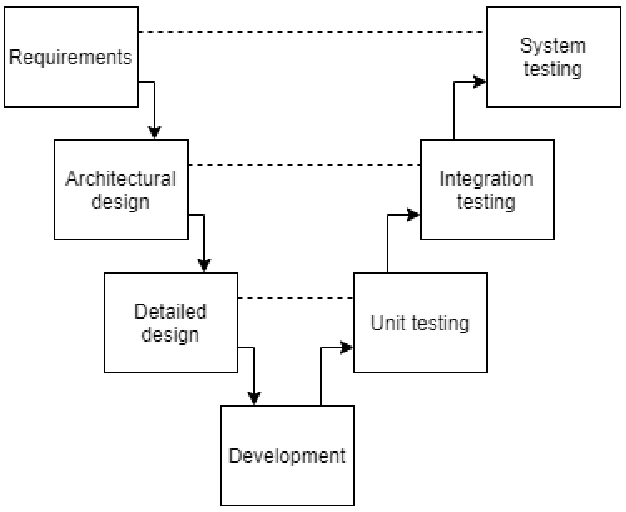


Figure 3*: Levels of testing according to the V-model, adapted from Jorgensen (2008)*

What is also not apparent from the V-model are the mobile-specific test levels, such as device testing and testing in the wild. These tests target different device platforms and versions and running tests in various physical environments and settings that pose challenges, such as low network connectivity. (Haller, 2013)

React Native implements a layer of abstraction between the application and the underlying device and OS. However, the abstraction is still rooted in mobile development, and some RN tools, features and functions must consider platform-related issues. The layer that RN introduces also brings some of its own considerations to multiplatform mobile development and SQA. While developers need not pay as much attention to OS details, a new set of RN details emerges. RN has an arguably quick development cycle, releasing four major versions in 2019 (Facebook, 2020b). This is not counting in the development of React itself, which compounds to RN's development. Components and features get deprecated, new features get added, and 3rd party libraries must keep up with these changes. Luckily, since RN uses JS as its primary language, the tooling for RN development and testing has a large overlap with web development, which also utilizes JS.

This section will describe different levels and types of testing, ascending the right-hand side levels of the V-model. Application specific technologies are briefly explained and the considerations they impose to testing are presented. At the end of each subsection a tool to conduct the type of testing is shortly introduced.

4.1 Unit testing

Unit testing is the most common and used testing method in organizations (Kasurinen, 2013). It tests a singular module, function or object. Unit testing can be divided into two categories: positive testing and negative testing (Olan, 2003). Positive testing verifies that the unit responds correctly to expected inputs. The purpose is to verify that the unit fulfils its functional specification (Singh, 2012). Negative testing verified that the component responds in a controlled way to unexpected or invalid inputs and conditions. A variety of invalid inputs and conditions include incorrect data types, incorrect input ranges, special characters, too short or long inputs, or when communicating with other interfaces fails (Kasurinen, 2013).

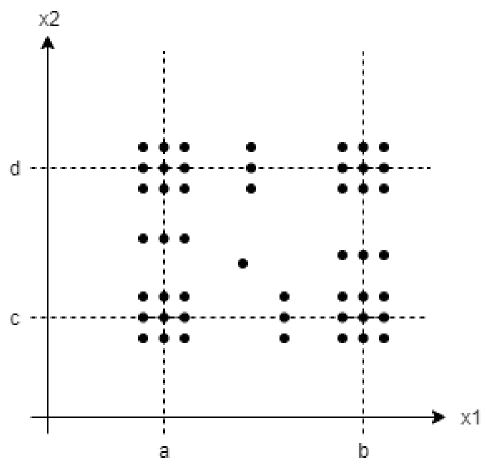


***Figure 4**: Robust worst-case test cases for a function that has two input variables, where x1 and x2 correspond to the input variables, a and b mark the valid input range for x1 and c and d mark the input range for x2 and the black dots mark the test cases, adapted from Jorgensen (2008)*

Especially for input ranges, the concepts of worst case and robustness can help find edge-case errors, as shown in Figure 4 (Jorgensen, 2008; Singh, 2012). Worst case testing checks all the boundary value combinations of all inputs and robustness tests the values slightly above the boundary values (Shahrokni & Feldt, 2013). Kaner, et al. (2002) give a good list of inputs to test the tolerance of an input field. This list can be extended to work with most functions, and the examples include default value, zero value, backslashes and other operating system filename reserved characters, one or more leading + or – signs, and leading and trailing whitespace.

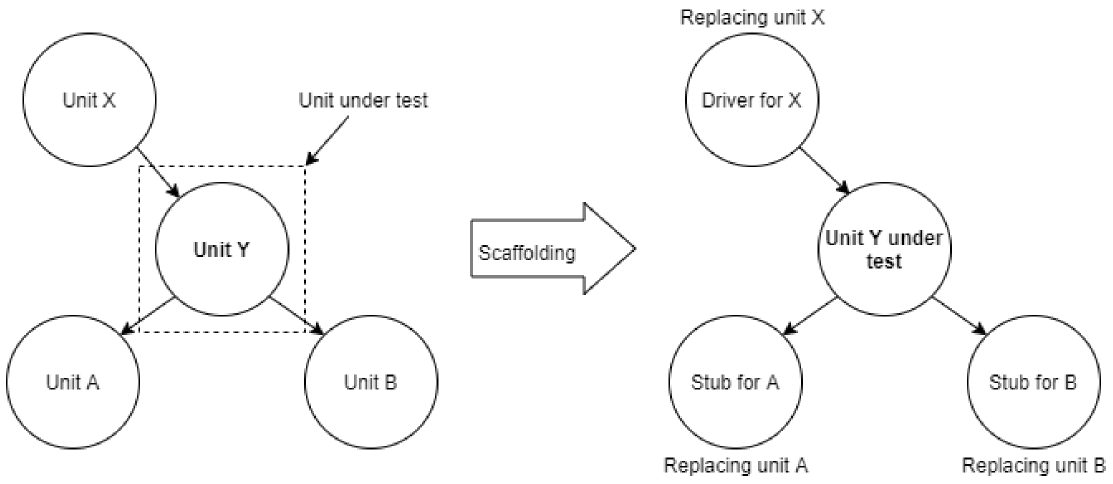


***Figure 5**: Using test scaffolding to replace real units with stubs and drivers to allow testing units that otherwise would be impossible or impractical to test, adapted from Singh (2012)*

Sometimes units have dependencies to other units. This could be the unit calling some other unit, being invoked by one, or being inherited or composed of one. These dependencies can be "faked" for testing by writing stubs and drivers that replace the dependent units. These stubs and drivers are collectively called test scaffolding, as illustrated in Figure 5. (Kasurinen, 2013; Singh, 2012; Shafique & Labiche, 2010)

After writing unit tests and running the once, they may seem pointless to keep since the units have been verified. However, in practice software maintenance, refactoring and dependency updates may change the functionality of already verified units, introducing bugs. Running unit tests continuously will uncover these bugs that didn’t exist before. This type of testing is called regression testing. (Kasurinen, 2013; Singh, 2012; Kaner, et al., 2002)

The tools to conduct unit testing are tied to the programming language used in the software. In the case of RN, that language is JS. Since JavaScript is a widely used language, the testing libraries for it are plentiful (Kleivane, 2011). A starting point for finding a suitable testing library is to see if there is an official or endorsed library for RN. Facebook, the author of React and RN, has their own testing library, Jest. Jest is used by Facebook to test their own React and RN applications (Facebook, 2019a). This relationship gives very good compatibility support, and the library even comes in pre-packaged with the default project template for RN. Jest prides itself as having no configuration on most JS projects and easy mocking (which means building test scaffolding) (Facebook, 2020c). Specific documentation for testing RN exists. Another practical benefit of using Jest for this thesis is that it is already used at Valamis extensively. Jest uses assertion programming, which means a test contains a Boolean predicate that is evaluated to conclude if the test passed or failed (Guo, 2008). A good practice in assertion programming is that tests should always test for a single concept, and preferably contain just one assertion (Martin, 2009).

4.2 View testing

View testing is a middle ground between unit testing and integration testing. Unit tests test individual units and components and combining these components into a view is an integration of units. The React pattern of combining components is composing, in which components are include hierarchically inside one another, as opposed to inheriting properties. This composition creates a nested VDOM of elements (Facebook, 2020d).

Views can be tested the same way as other units, but this runs into a problem of complicated and many assertions because view definitions are usually complicated, and a view may be valid but still have slight differences to the expected outcome defined in an assertion predicate. An alternative technique to test views is called snapshot testing (also known as golden master testing, characterization testing, baseline testing and difference testing). Snapshot testing is based on having a known, valid output (called the golden master or snapshot) and then subsequently comparing the test output to this valid output and failing or passing based on if changes have occurred, and if the changes are allowed. This moves away from assertions, where instead of evaluating explicitly stated predicates that test some detail of the output, the whole output is checked and anything an assertion might have missed is still reported. (Rößler, 2019)

There are two ways to technically achieve snapshot testing for UIs. The obvious one is taking a screenshot of the output on the screen and comparing it using some heuristic, usually pixel-by-pixel, to the snapshot. The other option is the generate the markup structure of the UI, usually in HTML or Extensible Markup Language (XML). The latter option is arguably lighter to execute, but the former option can be utilized to give more confidence in different screen size, platform dependent rendering and to automatically share the screenshots to non-technical people who otherwise couldn't read the component markup. (Rusynyk, 2019)

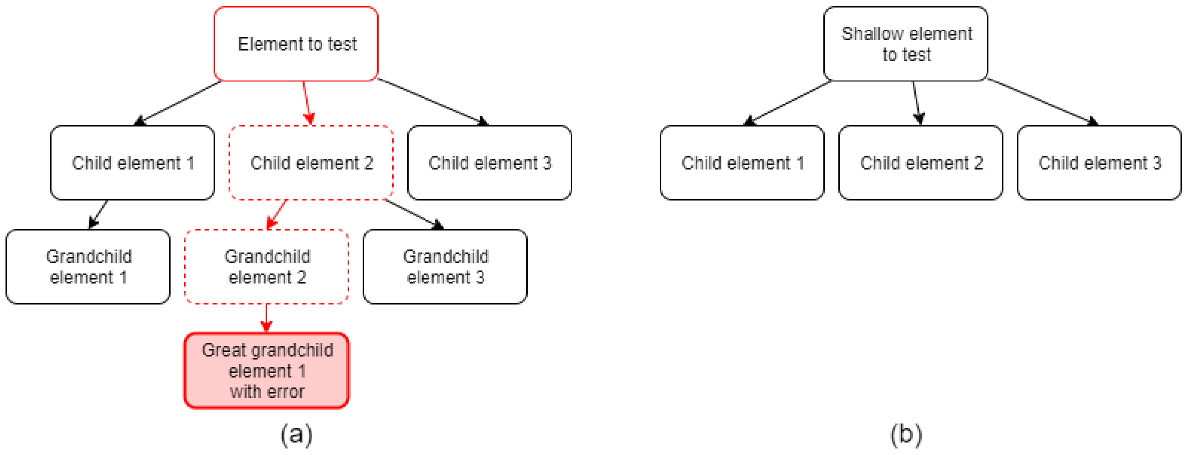


Figure 6*: Rendering strategies for markup-based view testing. In figure a, full depth rendering causes an error in a child element arbitrarily deep in the tree to propagate up to the element originally tested, and fail the test. In figure b, shallow rendering only renders the first layer of children and doesn't consider their contents, preventing error propagation*

Another aspect where there are two ways to technically conduct snapshot testing specifically in React and other composition-based methods is how do you handle the VDOM tree. Test-rendering views or parts of them creates a nested VDOM, but usually the test is focused on just a part of the tree, giving two options for generating the VDOM in the first place. The naïve approach is generating the full tree and testing the full depth of it, including the nested elements. The other option is generating only the element that is intended to be tested and stubs for the first level of nested elements. This is called shallow rendering. It reveals the children that would be created but does not traverse further down the render tree. The benefit of shallow rendering is isolation: if a child has an error, the error will show up in the child’s shallow test, but not in all of the parents and ancestors tests, as illustrated in Figure 6 (Airbnb, 2019).

The two ways to conduct snapshot testing, screenshotting and markdown, require very different tools, as they have very different methods of verifying the output. A popular tool for conducting React component testing and providing shallow test-rendering is react-test-renderer (Garreau & Faurot, 2018). It can be used in conjunction with Jest to create markdown-based snapshot testing. Pixels-catcher is a screenshot-based snapshot testing tool for RN. It can be configured to use a physical device or virtual device and either Android or iOS to capture the screenshots. The tool saves and compares the screen contents as base64 data, which can be converted to an image file.

4.3 State management testing

Applications can contain complicated data structures that are referenced in multiple modules and units. This can sometimes lead to race conditions or inadvertent mutations when two different units access the same data simultaneously. These bugs are difficult to isolate and fix. In React especially it is troublesome to have parts of the application state spread throughout the components of the application. Design patterns of container components and state hoisting help centralize state in the composition structures (Chan, 2020). A problem with these patterns is that sometimes the VDOM trees are very deeply nested and hoisting may have to happen for many levels, leading to boilerplate and complexity. (Kuparinen, 2019; Garreau & Faurot, 2018)

To help manage complicated application states and accesses, the flux pattern was created. A library called Redux for JS implements the flux pattern, centralizing the application state into an immutable data structure that can be accessed anywhere and changed in a controlled way. The mobile application in this thesis uses Redux to manage its state, so to test the management the working of Redux must be understood. The Redux flow, as presented in

Figure 7, starts from a component that has a request to change the application state. This request is called an action in flux/Redux terms. (Garreau & Faurot, 2018)

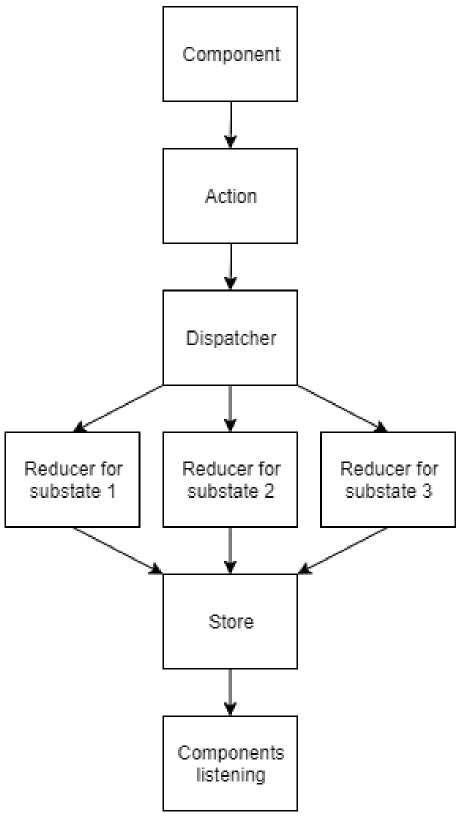


***Figure 7**: Example state management flow with Redux, adapted from Kuparinen (2019)*

The action contains information on how the state needs to be manipulated, and possibly the data that is used for the manipulation. The action is passed to the dispatcher, which is a singleton that processes actions one by one synchronously, preventing race conditions. Ultimately the changes are made to a state object, a tree, called the store. Redux allows splitting the store into multiple subtrees, with each substate having their own pure function called a reducer. A reducer contains definitions for how actions are applied to the state and generates the new state object after the action has been applied. The accessibility of application state is made easy by the fact that actions can be dispatches from anywhere in the application, and the store being practically a global object, can also be read from anywhere, eliminating the need for state hoisting. (Kuparinen, 2019; Garreau & Faurot, 2018)

To test the state management, unit and integration tests are used. As Redux is tied into many components, it makes a critical target for unit testing to verify that the reducers work exactly as expected. Integration testing is used to verify that the connection between components and the store works as expected, for example by dispatching an action from one component and verifying that the change has propagated to another component listening the store. For asynchronous actions the store should be replaced by a stub entirely, requiring Redux-specific testing tools. Redux (2020) themselves recommend the library redux-mock-store to achieve this. (Garreau & Faurot, 2018)

4.4 Integration testing

Integration testing is a natural extension and continuation of unit testing. In the simplest form the idea behind integration testing is to replace test scaffolding in unit tests with real units to see how they function together (Kasurinen, 2013). The combining of units to form an integration test can be done in many ways. The approaches covered in this section are the decomposition approaches, the call graph approaches, path-based approach, and finally the problems of separated subsystem integration testing.

4.4.1 Decomposition testing

In the decomposition approaches, described by Jorgensen (2008), Singh (2012), Kasurinen (2013) and Solheim & Rowland (1993), the program is abstracted into a functional decomposition tree, where the dependencies between units are mapped into a tree structure. The entry point of the program – the main function or module usually – is the root node of the tree. A recommended systematic process of forming the tree is decomposing the units of the system into a table and then forming the tree based on packaging partitions of the units (Jorgensen, 2008). Once the tree has been formed, one of three integration strategies can be used: top-down, bottom-up and sandwich.

Top-down integration is characterized by starting at the root of the tree and integrating the dependent units in a breadth-first manner, as can be seen in Figure 8. Once the first layer of children is integrated, the process moves to the second layer of children, replacing all the stubs in the tests with the real units. This is repeated until every layer of children has been visited.

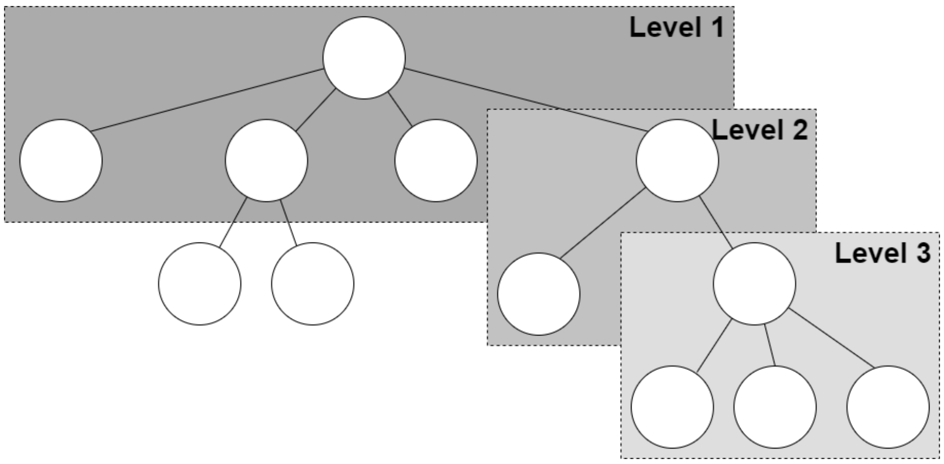


***Figure 8**: The top-down integration strategy, adapted from Singh (2012)*

Bottom-up integration is the reverse of top-down integration. The starting point is the lowest layer in the tree where the starting units are leaf nodes in the decomposition tree, as can be seen in Figure 9. Instead of replacing stubs, the drivers are being replaced since the integration is moving upwards.

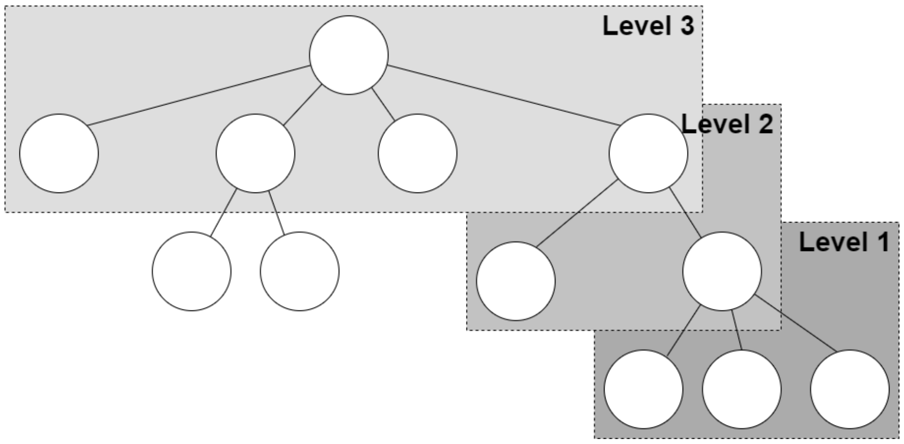


***Figure 9**: The bottom-up integration strategy, adapted from Singh (2012)*

The final decomposition integration strategy is the sandwich strategy. Instead of following the layer order in the tree, logical parts of the tree are selected and integrated at once, as can be seen in Figure 10. The size of the selected subtree is at the discretion of the tester, but larger subtrees require less scaffolding while being harder to isolate and pinpoint test failures.

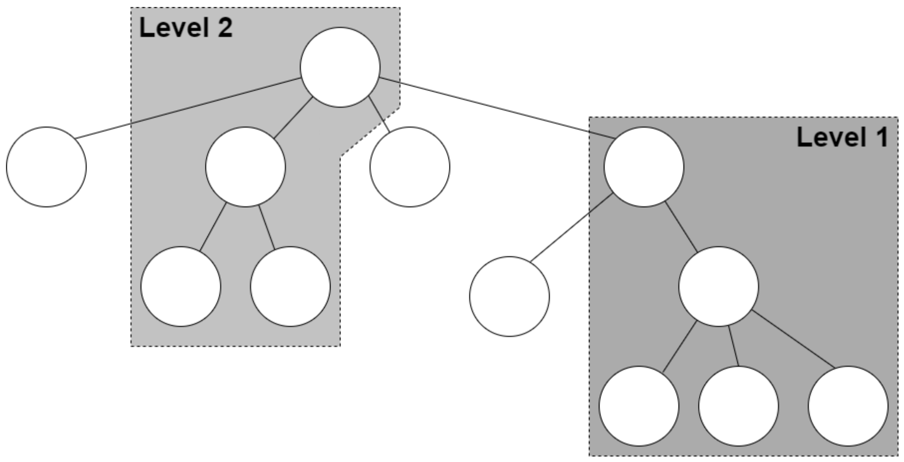


***Figure 10**: The sandwich integration strategy, adapted from Singh (2012)*

The decomposition integration testing approaches have a focus on structural testing, which means mainly the interfaces and static compatibility between units is tested. Solheim & Rowland (1993) found that top-down strategies provide the most reliable systems and bottom-up the least reliable systems. This is contrasted by the fact that sandwich is the most common decomposition integration test strategy in the industry (Singh, 2012). Top-down and bottom-up are breadth-first traversals of the decomposition tree, while sandwich is a combination of breadth and depth traversal on a subsection of the tree (Jorgensen, 2008). The downsides of decomposition approaches is the normally limited depth of the integration when the focus is placed more on the interfaces between units, and often a significant amount test scaffolding is needed if the decomposition trees are deep (Jorgensen, 2008). This means that man-hours must be spent just to enable integration tests.

4.4.2 Call graph testing

Instead of functional decomposition, the source code can be analysed to form a directed graph of all the units and how they access each other (Milanova, et al., 2004). These graphs are called call graphs, and the integration testing methods based on them are collectively called call graph approaches. They take a step from structural testing towards behavioural testing. This also reduces the need for test scaffolding construction, as directly linked units are preferred testing targets. (Jorgensen, 2008)

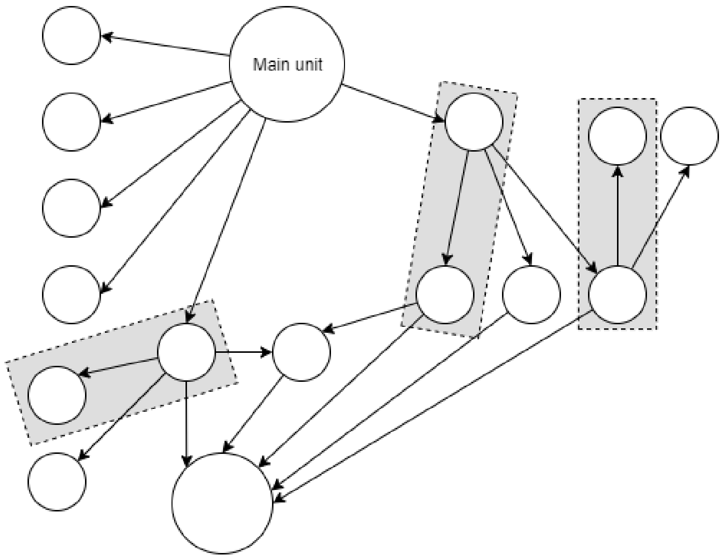


***Figure 11**: Pairwise call graph testing, where a pair of units is selected from the call graph for integration testing, adapted from Jorgensen (2008)*

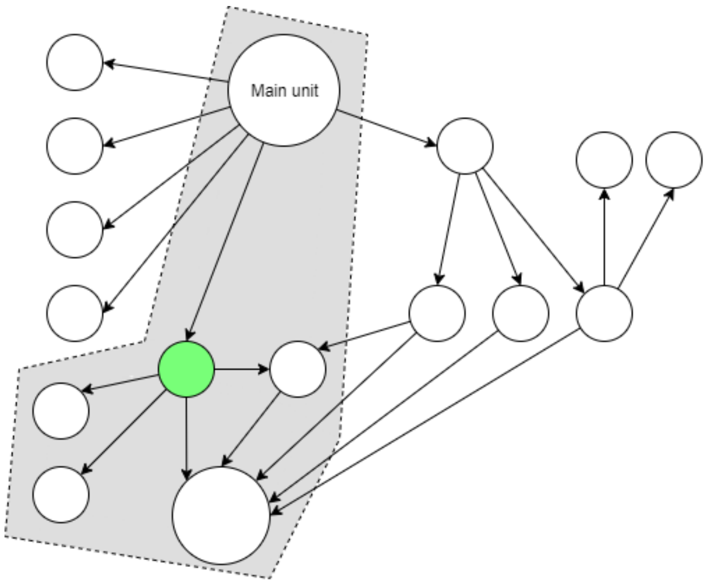


***Figure 12**: Neighbourhood call graph testing, where all the units surrounding the unit in the call graph being tested are included. The unit being tested is highlighted in green. Adapted from Jorgensen (2008)*

Call graph testing can be structured in two ways: pairwise or neighbourhoods. In pairwise call graph testing a pair of units is selected on the graph and tested, eliminating the need for stubs when actual units and their source code can be used. Pairwise selection is presented in Figure 11. Neighbourhood call graph testing takes a more ambitious approach to selecting units on the call graph. A unit is selected to be the centre of the tests, and all units connected to it, units it calls and is invoked by, are tested. The result, as seen in Figure 12, resembles the sandwich decomposition approach, which the neighbourhood selection does share characteristics with: reduced need for scaffolding, but isolation of failures is harder. (Jorgensen, 2008)

The call graphs generated for integration testing can also be used in other areas of SQA. There is a connection between software defects and cohesion and coupling that can be calculated from a call graph (Abandah & Alsmadi, 2013). The problems with call graph integration testing however are that pairwise testing can generate a lot of tests, and depending on the cohesion and coupling between units the neighbourhoods can become very large and multiple neighbourhoods may have significant overlaps, creating redundancy and making isolation even harder.

4.4.3 Path-based testing

To get closer to testing how the system is used, an even more behavioural-driven testing approach than call graph testing can be used. Jorgensen (2008) and Sahu & Mohapatra (2015) describe path-based integration testing. It aims to capture a vertical slice of the software's functionality in what's called a behavioural thread. This concept is getting closer to system testing, essentially being a depth-first approach compared to the hybrid or breadth-first approaches of decomposition and call graphs.

Path-based testing is built around method-message paths (MM-paths) between units. These paths cross between unit boundaries and follow the execution of a behavioural thread. Commonly MM-paths are modelled with Unified Modeling Language (UML) sequence diagrams (Kundu & Samanta, 2016). The downside of path-based testing is that MM-path identification and definition is laborious. While the tests themselves are behaviour-focused, the definition of graphing the path requires structural understanding of the system. The benefits of path-based testing are that it eliminates the need for scaffolding entirely, as the entire chain of stubs and drivers is traversed, and that the step to system testing with multiple behavioural threads is easier when individual threads have already been verified during integration testing. (Jorgensen, 2008; Sahu & Mohapatra, 2015)

4.4.4 Integration testing with remote subsystems

A common case of integration testing is a client-server architecture's interface between the server and the client. Arguably this is often the most critical and biggest integration in a system, especially if the two subsystems are developed by separate and independent teams, which is the situation Valamis is facing. According to Jorgensen's (2008) categorization of client-server functionality separation, this system has a fat server, meaning the application logic and databases are on the server side of the system. A difficulty with fat server systems is keeping the systems in synchronization. The client can be modelled as a complex finite state machine that can engage in multiple simultaneous behavioural threads, implicating there can be multiple points of synchronization (Mesbah, et al., 2012). Synchronization can be problematic in real time systems, but in this case the communication between the client and the server is stateless and asynchronous, so much of the complexity is alleviated. (Jorgensen, 2008)

A possibility to make subsystem integration testing more robust is to have two-way testing. This means that integration both from the client to the server and from the server to the client is tested. This requires joint effort from both teams of the respective subsystems, as a definition of the application programming interface (API) that the client consumes must be formalized, versioned and documented, so that the server-side can have tests that match the client's tests. This however is unfortunately out of scope for this thesis, as it extends outside of client testing.

For the purposes of this thesis, the integration testing effort is focused on the interaction with the server subsystem. The server is mocked with a stub to which the application will connect with. Requests and communications can then be verified to be within the specification of the interface of the server-side subsystem. The server stub can be mocked with practically any Hypertext Transfer Protocol (HTTP) server package. This stub will report the received requests to the test runner, like Jest. The reverse case of what is explored here is also useful in a more general sense; testing that the server responds correctly to client requests by creating a client stub that queries the server, and then verifying that the response and communication (not just the business logic for generating the response) work as expected.

This is especially the case when the same team works on both subsystems and testing for both can be done fluently.

4.5 System testing

After system units are tested individually and integrated together to the point where scaffolding is not involved anymore, the system is tested as a whole. There are different approaches with different focuses and goals for system testing, such as load testing, user testing, exploratory testing, or in the case of mobile applications, device testing (Kasurinen, 2013; Singh, 2012). Compared to integration testing where one input is given to the software and an output is produced, in system testing a sequence of inputs is given to simulate a scenario of normal usage or a specific system-level test case, like handling a certain amount of concurrent users (Jorgensen, 2008; Singh, 2012).

System testing takes on a full-on behavioural approach. The tests are purposed and structured to no longer focus on individual actions, but on the sequence and the outcomes of them. To systematically test behaviour, a finite state machine can be used to identify software states and the inputs that transition the state. Realistically, most of the time a system cannot be exhaustively defined, and the state machine must be modelled to find the system testing cases. Reactive software, such as the mobile client, can be sufficiently modelled with event-based testing. This means the modelling is based on what transitions and subsequent states are possible from the different input events available at a given state. The detail of the modelling is split into five levels of coverage according to Jorgensen (2008):

1. Each input event occurs
2. Common sequences of input events occur
3. Every input event occurs in every relevant context
4. For a given context, all inappropriate input events occur
5. For a given context, all possible input events occur

The first level of coverage means all possible input events are tested in some context or state. This is only useful in stateless and very simple applications, where previous inputs do not affect the outcome of the next input. In stateful applications this is an inadequate level of coverage. The second level of coverage is the most widely applied, but it has the problem of defining what is “common.” The last three levels of coverage are context dependent, which bears a similar problem as the second level: selecting the relevant contexts. In simple systems it is obvious which states or contexts should be tested, but in complex systems with many user groups and use cases it can be difficult to identify the important contexts to cover. (Jorgensen, 2008)

4.5.1 System testing on mobile devices

***Table 2**: Distribution of active Android devices and API levels in May 2019 (Google Developers, 2019)*

Android Version	API level	Distribution of users
2.3.3 – 4.3	10 - 18	3.8%
4.4	19	6.9%
5.0	21	3.0%
5.1	22	11.5%
6.0	23	16.9%
7.0	24	11.4%
7.1	25	7.8%
8.0	26	12.9%
8.1	27	15.4%
9	28	10.4%

System testing on mobile platforms poses a need and an opportunity to conduct system testing on the target devices. Mobile OSs arguably get frequent updates. New devices with entirely new hardware features and specifications enter the market more often than in other fields of software (Haller, 2013). This leads to increasing amounts of different specifications that software must be compatible with. Especially on the Android platform, where updating to the latest OS version is not enforced by manufacturers, the concept of API levels helps developers to deal with compatibility issues. API level is an integer that uniquely maps to an OS interface that can be used in development through a software development kit (SDK) (Brandi, 2019). An application is given an API level definition, and the minimum API level tells the minimum feature set the device must support, or it will not be able to run the application. This leads to developers having to make a trade-off between functionality and wider user base. A lot of consumers run devices with low API levels, so the userbase is very fragmented as can be seen in Table 2.

There are different approaches to running an application on a device for device testing. The most obvious approach is to have a local physical device, a phone or a tablet, that the application is loaded on to and then run and tested. The downside of local device testing is that it’s hard to scale up. Testing various configurations and parallelizing tests requires buying more devices. Due to the nature of the mobile hardware, the fast increments and evolution, warrants periodic renewal of the testing hardware, creating a continuous expense. An extension of local devices is a private device cloud. This means a centralized, private device pool is created, from which a physical device can be selected and connected to, removing the need for testers to deal with physical devices while still maintaining the benefits of local device testing. (Gao, et al., 2014; Haller, 2013; Bordi, 2018)

Since maintaining a first-party solution to device testing is expensive, an option is to outsource the devices themselves for testing purposes. Public device clouds are third-party services that resemble private device clouds in that they provide a pool of physical devices to which the testers can connect to remotely to test. This solution can raise challenges in integrating to the rest of the testing infrastructure, as the providers usually have pre-defined and rigid interfaces. (Haller, 2013; Bordi, 2018)

Instead of outsourcing devices, the whole testing activity can also be outsourced. Crowd-based device testing gets the testing effort from freelancers, contracted actors, or communities. This approach usually has the widest device variance, which might not reflect as much on the cost as the previous approaches. This also grants easy access to in-the-wild testing, as by default the testers are not in a dedicated testing environment or laboratory. The downsides to crowd-based testing are very limited automation and the lowest quality of testing. Especially if the testing is unpaid, there is no guarantee of when the testing will take place, how long it will take, and if the results are of any use. (Gao, et al., 2014; Haller, 2013)

The last approach is to not use physical devices at all. Devices can be simulated on a host platform, a computer, leading to no physical devices ever being needed. This is the cheapest option, but limitations in the configurations that are available, especially on the Android platform, and some cutting-edge device-specific hardware and software form the downsides for emulator-based device testing. (Gao, et al., 2014; Haller, 2013; Bordi, 2018)

4.5.2 Scoping system testing

In addition to selecting how to facilitate system testing, the scope and target of system testing should be defined. System testing can focus on subsystems, and in the case of a client-server architecture of the Valamis LXP and mobile application, the subsystems where the focus could be targeted are the server-side backend and the mobile client frontend. In addition to the subsystem targeting, the test initiation and result observation has an impact on scoping. Some combinations of these scoping options are presented in Figure 13.

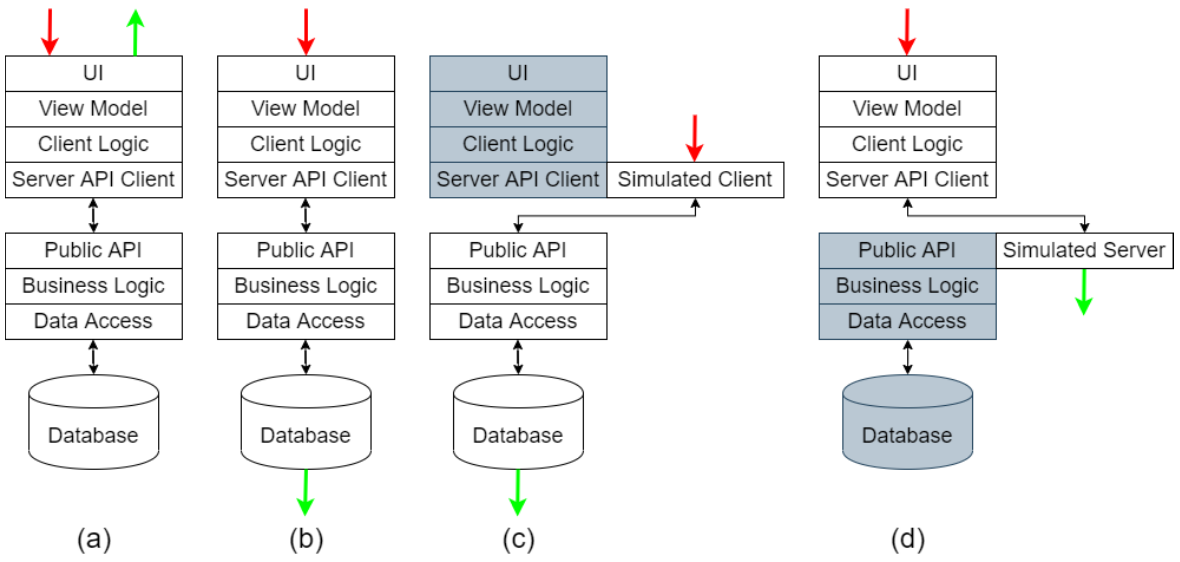


***Figure 13**: Different scopes of system testing. The red arrows signify the test entry point and the green arrows where an assertion is checked to verify the outcome. The parts of the system stack highlighted in grey mean that part of the system is not being tested. Figure a represents a full stack round-trip test, figure b represents a one-way full stack test, figure c represents a one-way server-only test, and figure d represents a one-way client-only test. Adapted from Axelrod (2018)*

The options for targeting subsystems in a client-server architecture are testing the full stack of the system, testing both the client and the server, testing only the server by using the same public interface as the client would use (acting as a driver), or testing only the client by replacing the server where the client is pointed at with a simulated one (acting as a stub). (Axelrod, 2018)

The system test result can be observed and verified at various points in the system. A round-trip end-to-end interaction scope means the test result is observed and verified at the same layer of the system stack as where the test is initiated, so for example if the test is initiated by interacting with the client UI, the test is verified by observing the correct changes in response in the UI. The other option to round-trip is one-way end-to-end interaction scope, where the entry observation layer is different from the initiation layer. Usually in one-way testing the verification layer is placed at the bottom of the system stack, like a database. This means that the test interacts with the UI, but the verification is based on if the expected changes are reflected in the database (or whichever other layer is used for observation). (Axelrod, 2018)

Combinations of round-trip, one-way, server-only, client-only and full-stack testing can be used together. Splitting up a round-trip full-stack test can help problem isolation, automating system testing, saving time by parallelizing, or provide cost savings by not aiming for full test coverage and leaving out some subsystems. (Axelrod, 2018)

4.5.3 Selecting a system testing approach

Device emulation is the most light-weight option and the easiest to automate of the options available. Device testing can be conducted using end-to-end testing automation software, which allows the definition of input and action sequences for creation of behavioural threads.

Bordi (2018) conducted recently an analysis of end-to-end testing tools fit for multiplatform mobile use. This can be used as a reference for selecting a testing tool to conduct system testing for this thesis, as multiplatform capability is a critical factor in both works. Bordi identified 23 potential different tools. Because most of the tools are commercial, academic comparisons and sources on them were limited, so popularity data was used instead to support arguments for practical usability. Discussion metrics on Stack Overflow were used as a data source for this purpose. Of the options available, the potential open-source tools Cavy, Detox, Appium and Calabash were recognized. The final decision in favour of Appium was made based on the offered feature set, perceived risk of use and available support. This rigorous analysis is used as the basis for selecting the same tool for conducting end-to-end system testing for this thesis.

Appium is a tool for automating Android, iOS and Windows applications. The authors of Appium specify that it and the underlying technologies are “not technically ‘test frameworks’ – they are ‘automation libraries’” (JS Foundation, 2019a) that any test runner can use to automate the application, allowing UI and end-to-end testing in graphical user interfaces. Appium provides a unified API for automating all the supported platforms. The API is based on and is a superset of the WebDriver automation API, most well-known for the testing framework Selenium used in testing web applications. A translation layer inside Appium unifies UiAutomator2, the Android framework, and XCUITest, the iOS framework. The core concept of Appium is similar to Selenium; both run a HTTP-server which controls the software being automated, and the WebDriver API sends commands to this server via a REST API. This enables creating implementations of the WebDriver interface in different programming languages, as the server-side interface is language-agnostic. The WebDriver implementation of choice can then be called inside a test runner to create the automated tests. In the case of this thesis, the JS implementation recommended by Appium itself can be used in conjunction with Jest to create end-to-end system tests. (JS Foundation, 2019a; Axelrod, 2018)

The draw conclusions from system testing, there are platform- and device-specific factors that may limit the options to system testing. When conducting behavioural end-to-end testing, conscious decisions about the physical facilitation, scope, and possible remote subsystem integration must be made, as they have significant impact on the tools that can be used. The most popular tool for conducting this type of testing on mobile applications is Appium, which is based on a popular web testing tool Selenium.

5 TEST AUTOMATION

This section describes the purpose, requirements, practices and needs for automating the testing process. The first subsection explores the motivation and trade-offs of automating testing, and the second subsection describes continuous integration and the practicalities of adopting test automation into the software development process.

5.1 Purpose of test automation

With the shift from waterfall-like software development lifecycles to agile practices, testing is no longer an isolated phase among others in a project, but rather an ongoing effort. Having iterative development and multiple releases means verification is also done many times over. Regression testing becomes increasingly important on all levels of testing (unit, integration, system) as complexity is bound to increase and refactoring and re-engineering are done to keep maintainability. This constant change makes automating regression testing very attractive (Kumar & Mishra, 2016). (Axelrod, 2018)

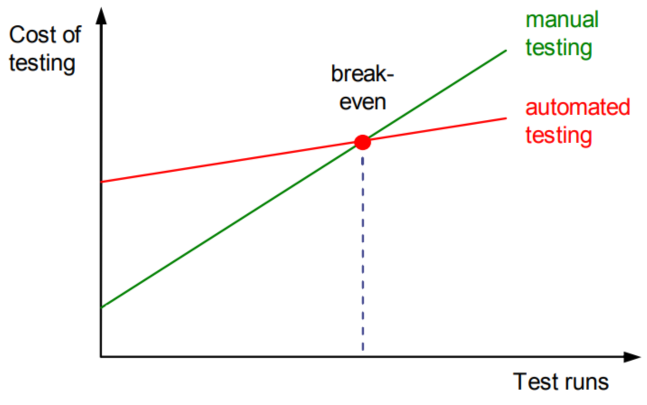


***Figure 14**: Simplified economics of automated testing compared to manual testing as a function of test runs, adapted from Ramler & Wolfmaier (2006)*

Test automation comes with a cost. The initial setup up costs for automated testing are higher than for manual testing in most cases. Especially more complicated testing activities, such as graphical user interface testing, the initial setup may be up to 10 times more expensive than with manual testing (Kaner, et al., 2002). Once the automation is facilitated, the subsequent test runs become very cheap (Polo, et al., 2013). This effect is highlighted in Figure 14, where the slope of the plotted line describes the cost over time. In labour-intensive testing methods this slope is steep for manual testing, but the initial starting point for automated testing is higher. The break-even point is the number of test runs where the total costs of manual testing catches up with automated testing. Some sources place this break-even point at 2-20 runs, Kasurinen (2013) places it at 8. The break-even point is however very variable depending on the testing and automation tools and techniques used. The high initial cost of automated testing poses an opportunity cost that deducts resources from development and manual testing. The implication is that tests that have few runs, are complicated to automate and unlikely to uncover regressions are not optimal to automate. Especially system testing has high initial costs and the testing runs are low, leading to system test automation having bad results in the industry. (Ramler & Wolfmaier, 2006; Berner, et al., 2005)

Software can be used to verify other software. Specialized tools allow the testing of software without human interaction. However, when designing tests and the automation, it must be remembered that not all tests can or should be automated. Automatic testing should always complement manual testing and not replace it. The reasons for this are that manual testing always has randomness in it, which creates slight changes in the tests every time, uncovering even more defects. Manual testing can detect side effects that automated testing would ignore, especially in assertion-based testing. Human interaction always has heuristics built into it; it can pass over false alarms or minor bugs in order to continue the test for the meaningful result. Some testing methods such as exploratory testing and usability testing cannot be meaningfully automated (Prakash, et al., 2012). Kaner, et al. (2002) emphasizes that organizations should ”use automation when it advances the mission of testing. Evaluate your success at automation in terms of the extent to which is has helped you achieve your mission.” It is possible (but rare) that a project won’t benefit from test automation. (Kasurinen, 2013; Kaner, et al., 2002; Forsgren, et al., 2018; Berner, et al., 2005)

Many sources (Ramler & Wolfmaier, 2006; Kaner, et al., 2002; Alpaev, 2017; Axelrod, 2018; Berner, et al., 2005) agree that there are types of testing that are not possible to conduct manually. Most system tests that gauge the non-functional quality attributes such as stability, capacity, performance and configurability can only be tested by leaving the tests running for extended periods, creating large amounts of simultaneous synthetic users, conducting and measuring hundreds of precise timing tests, or smoke testing hundreds of configuration permutations for different hardware and software environments.

The levels of test automation describe how advanced the automation is in terms of the autonomy and ease of creation of the tests. The levels are defined as:

1. Based on recording and playing back actions. May or may not include controlling and reporting failures
2. Modular and data-driven testing, where the test is always a script that can be reused, moved to a different platform, or the test data can be trivially replaced (meaning the data is not hard-coded like in the first level)
3. Keyword-driven testing takes the data-driven automation even further, abstracting the details to concepts which makes the tests application-agnostic and self-sufficient when it comes to for example navigating a UI
4. Behaviour driven development testing frameworks turn creating tests into layperson-understandable language, which allows higher conceptualization and non-technical personnel to understand and create tests

For most organizations the second level is enough, as this is where the most variety of tooling is available and developers and SQA personnel can feasibly cover even a larger project. Speculation about the fifth level of test automation is directed towards artificial intelligence and machine learning, where even the test cases are automatically generated. (Mosley & Posey, 2002; Kasurinen, 2013; Automate The Planet, 2018)

5.2 Facilitating test automation

Test automation needs criteria to run, it can’t exist in a vacuum without a software project. Ongoing software development provides different opportunities and options to running test automation. Tests should be run frequently, as not running them leads to too many changes being accumulated. This leads to poor test coverage as new changes are not being tested, and existing tests will fail due to valid changes and create confusion in SQA. Rapid test running is necessary for trunk-based development where changes are integrated as fast as possible, usually within a day, back to the software mainline in version control. (Forsgren, et al., 2018)

While it is important to decide when to run the tests, it must also be decided where to run the tests. Two options are possible: on the developer's or tester's local computer, or on a centralized server. Running on a centralized server ensures that there are no hidden dependencies or local environment oddities that make or break the software and tests. With modern service providers it is trivial to acquire a dedicated server to host the automation process (Meyer, 2014). The trigger of automation dictates when the tests will run. There are several options to where this trigger is tied to. Plan-driven and waterfall-like lifecycles trigger testing at the end of lifecycle phases. Another option is to run the automation at fixed intervals. This is known as nightly builds and nightly runs, because it is common to run the automation during the night to avoid interrupting development going on during the day. The last, and nowadays the most common option, is to tie the automation to version control check-ins. The automated process that triggers on a check-in that automatically builds (which can be seen as a smoke test) and tests software before accepting it into the version control is called continuous integration (CI). Adopting CI however opens more options on the details: which tests to run and on which version control actions. Common version actions are commits to development branches, integrations to the mainline, and establishing a baseline for a deployment or release. (Axelrod, 2018; Forsgren, et al., 2018; Ståhl & Bosch, 2014)

A typical CI process – also called a pipeline – that runs on a build or automation server is as the following:

1. Get sources from version control
2. Compile/build software
3. Run unit and integration tests
4. Deploy and prepare a test environment or agent
5. Run system tests
6. Generate a report

As discussed in section 4.5.1, device testing can take many different approaches, which means step 4 of deploying the software to a test environment or agent, in this case the mobile device, may be very different based on what device testing approach is selected. In the mobile software field, there are constraints to step 2 as well. Apple limits the compilation and emulation of iOS applications to Mac computers, so if the build server is not running MacOS, it will need to connect to a Mac to compile and conduct and device tests for iOS platform. The last step of generating reports is normally manifested as an integrated outcome of the pipeline, in which any failed tests or steps are specified. (Axelrod, 2018)

A key factor to successful CI is that the process must be kept short in duration. Long runs lead to long wait times for developers, which further lead to developers not wanting to check in code as often. Changes accumulate and become harder to cover, and the CI is more likely to face integration conflicts. To keep the process quick, only the most critical and fastest tests should be run on normal check ins. Depending on the breadth of the unit and integration test suites, the levels may run entirely, but system and UI tests are often too slow. The CI pipeline can be split into stages, where the initially built and tested software can then be deployed to different targets for different tests, for example performance, configuration and load tests that don’t need to run sequentially after the critical set of tests has finished (Mårtensson, et al., 2019). This splitting into stages also allows the pipeline to be parallelized to further speed up the process. (Axelrod, 2018; Meyer, 2014; Ståhl & Bosch, 2014; Fowler & Foemmel, 2006)

In summary, automated testing has compelling benefits, but it mustn’t completely replace manual testing. Different tests can be separated into different types of automation, which allows performance optimization, scheduling and logical sequencing. The benefits of centralized, server-run automation in the form of CI is possible and recommended, even within the company constraints of Azure DevOps.

6 IMPLEMENTATION OF TESTS AND AUTOMATION

The practical part of this thesis is implementing tests and test automation for the Valamis mobile application. The choice of which tests to implement is based on the types of testing presented in section 4. For each type, a corresponding subsection is presented in this section. The individual tests for each type were selected with a team leader and developer from the company, and they had the final say in which tests to include and how to conduct them. Access to the source code of the mobile application project was granted and a separate branch was created in the version control system to isolate the implementation effort from the company's ongoing development.

This section contains the outcomes of the implementation, which are code and configuration listings and steps for each type of testing, which are listed in the respective subsections. The code listings are supported by explanations that detail the contents.

6.1 Unit tests with Jest

For unit test selection, there was one criterion that was agreed upon warranting an example implementation: the use of test scaffolding. This led to the selection of two units to create tests for, one with scaffolding in the form of a stub, and one without any scaffolding. As mentioned in section 4.1, unit tests are implemented using a testing library called Jest. Since the project was initialized using a template, the library is already included and configured in the project. In other cases, the Jest Getting Started documentation describes the installation and configuration process (Facebook, 2019b).

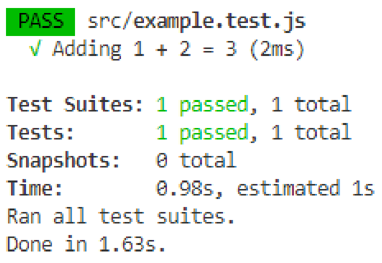
Listing 1*: Jest definition syntax. The call of test expects two inputs: a description string for the test, and a function that contains the assertion. In this listing, the function containing the assertion is formatted as an anonymous function.*

```
const sum = (a, b) => a + b;

test('Adding 1 + 2 = 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

Jest syntax for basic unit tests is straightforward, as shown in Listing 1. Assertions (line 4) are built from two parts, the input that is checked (marked by keyword ‘expect’), and the predicate that contains the expected value (in this case marked by keyword ‘toBe’).

***Listing 2**: Output of 'yarn test' command specifying the results*



```
 PASS  src/example.test.js
  √ Adding 1 + 2 = 3 (2ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.98s, estimated 1s
Ran all test suites.
Done in 1.63s.
```

Tests are recommended to be defined in separate files and named after the file they contain tests for, with an addition of ‘test’ in the filename, such that the tests for ‘functions.js’ would be in the file ‘functions.test.js’. Once Jest is installed and some tests have been created, they can be run with the package manager command ‘yarn test’, which searches the project directories for all files with the name pattern specified earlier and runs them. The results of the tests are displayed in the terminal as shown in Listing 2, specifying all the passed and failed tests. (Facebook, 2019b)

***Listing 3**: Tests with stub scaffolding*

```
test('RequestManager calls course and user actions', () => {
  const mockCourses = jest.fn();
  const mockUser = jest.fn();

  const manager = new RequestManager({
    getMyCourses: mockCourses,
    getMyMobileUser: mockUser,
    login: {
      site: 'site url',
      headers: []
    },
    user: undefined,
    courses: []
  });
  manager.componentDidUpdate();

  expect(mockCourses.mock.calls.length).toBe(1);
  expect(mockCourses.mock.calls[0][0]).toBe('site url');

  expect(mockUser.mock.calls.length).toBe(1);
  expect(mockUser.mock.calls[0][0]).toBe('site url');
});
```

***Listing 4**: Tests for a HTML-stripping function*

```
describe('HTML strip tests', () => {
  test('Strips complete HTML elements', () => {
    expect(stripHTML('This is <div></div> text')).toBe('This is  text');
  });

  test('Strips HTML elements around text', () => {
    expect(stripHTML('This <div>is</div> text')).toBe('This is text');
  });

  test("Doesn't strip partial HTML elements", () => {
    expect(stripHTML('This is <div text')).toBe('This is <div text');
  });

  test('Handles empty strings', () => {
    expect(stripHTML('')).toBe('');
  });

  test('Strips non-breaking space characters', () => {
    expect(stripHTML('This is text')).toBe('This is text');
  });

  test('Returns null on no input', () => {
    expect(stripHTML()).toBeNull();
  });

  test('Returns null on incorrect data types', () => {
    expect(stripHTML(1)).toBeNull();
    expect(stripHTML({})).toBeNull();
    expect(stripHTML([])).toBeNull();
  });
});
```

One criterion leading to two test cases were selected to act as examples for unit tests. The criterion is the use of test scaffolding. The first test case in Listing 4 uses no test scaffolding and the second test case in Listing 3 uses stubs as test scaffolding. The stubs (Listing 3, lines 2 and 3) are built with Jest's mock functions, which can report if they were called and what inputs they were given (Listing 3, lines 17, 18, 20 and 21), without calling the real functions. Tests can also be grouped with the 'describe'-function if multiple test-suites are included in the same file (Listing 1, line 1) (Facebook, 2019c). For the first unit a standalone function that strips HTML elements from descriptions was chosen. This function can receive many kinds of input, so the input tolerance cases mentioned in section 4.1 are utilized. The second unit is a component that will manage network requests. The component will attempt to call Redux actions it has been passed, which are in this case replaced by stubs that can verify the data the component passes them and that the component calls them only once.

6.2 View tests with react-test-renderer

View testing for isolating component rendering had one variable in it: does the component under test have children composed in it. For that reason, two test cases were selected for the view testing implementation; the first case is a leaf-level node with no children, and the second case is a mid-level node with composed children.

To conduct snapshot testing of React components, react-test-renderer combined with Jest with the markdown comparison approach was chosen. The visual comparison with pixels-catcher was not implemented, as the secondary benefits of visual snapshot testing mentioned in section 4.2 were not deemed useful. Jest manages the saving and comparison of the markdown snapshots, while react-test-renderer provides the shallow rendering tools for correct component isolation. On the first run of the test, when no golden master to compare to is present, one is saved and then used for the future tests as the baseline. This means the snapshot is saved as a file in the project and should be included in version control, so that other developers share the same baseline. However, sometimes snapshots intentionally change, for example when a design changes. Existing snapshots can be updated with the command 'jest -u' in the project folder. (Facebook, 2019d)

***Listing 5**: Snapshot test syntax with react-test-renderer and Jest*

```
1.  const Component = () => <Text>Hello world</Text>;
2.
3.  test('Snapshot component', () => {
4.    const renderer = shallow.createRenderer();
5.    renderer.render(<Component />);
6.    const item = renderer.getRenderOutput();
7.    expect(item).toMatchSnapshot();
8.  });
```

The basic syntax of snapshot testing is presented in Listing 5. The test is built by first rendering the component markup, in this case with react-test-renderer's shallow renderer (lines 4 – 6). This markdown is then compared against the snapshot in the test assertion (line 5). The renderer also grants a more complicated VDOM observation and manipulation API, where properties, subcomponents and state can be iterated upon, called functions on, queried or otherwise manipulated. Overall this syntax is very similar to the unit test syntax covered in section 6.1.

Listing 6*: Snapshot test of a component with no children*

```
test(‘Documents Item snapshot’, () => {
  const renderer = shallow.createRenderer();
  renderer.render(
    <DocumentsItem item={{ name: ‘name’, size: ‘size’ }} onPress={jest.fn()} />
  );
  const item = renderer.getRenderOutput();
  expect(item).toMatchSnapshot();
});
```

Listing 7*: Snapshot test of a component with children*

```
test(‘Continue Learning shallow snapshot’, () => {
  const renderer = shallow.createRenderer();
  renderer.render(<ContinueLearningView t={jest.fn()} />);
  const item = renderer.getRenderOutput();
  expect(item).toMatchSnapshot();
});
```

The two tests presented in Listing 6 (snapshot of a component with no children) and Listing 7 (snapshot of a component with children) are very similar, as the purpose of the Jests snapshot and react-test-renderer’s API is to provide easy validation of components. The syntaxes for both tests are identical. However, the saved snapshots for the tests are very different depending on the component, as can be seen in appendices 1 and 2. What is also apparent from the snapshots themselves is that react-test-renderer captures the styling and properties of the component, providing a detailed description of the contents.

6.3 State management tests for Redux

A big logical part of the application structure is the state management. It permeates through the visual components and plays a role in integrating with other subsystems. This makes it an important part of the application to verify, yet the workings of it are dissimilar to how React component rendering or logic works. Action creators and reducers are the key parts of the Redux functionality. Action creators are further divided into synchronous and asynchronous actions, with the asynchronous actions being associated with remote subsystem integration. These areas form the needs of Redux testing. (Garreau & Faurot, 2018)

6.3.1 Testing synchronous action creators

Testing synchronous action creators is straightforward, as action creators usually take an argument and create an object with the action type and passed argument as payload. This lends itself for direct assertion testing. Snapshot testing could be used, but since the action objects are very simple and there is no associated metadata like complicated style objects or dynamic properties, normal assertions with manually defined expected output can be utilized. This means no extra tools besides Redux itself and Jest are needed. (Garreau & Faurot, 2018)

***Listing 8**: Synchronous action creator testing syntax*

```
1.  const actions = {
2.    SAMPLE_ACTION: 'SAMPLE_ACTION'
3.  };
4.  const sendAction = (data) => ({
5.    type: actions.SAMPLE_ACTION,
6.    payload: data
7.  });
8.
9.  test('Action creator for sample action', () => {
10.   const expectedAction = {
11.     type: actions.SAMPLE_ACTION,
12.     payload: { field: 'value' }
13.   };
14.   expect(sendAction({ field: 'value' })).toStrictEqual(expectedAction);
15. });
```

The test syntax displayed in Listing 8 consists of defining the expected action object yielding from an action creator (lines 10 – 13) and asserting on the output of the action creator (line 14). Action creator syntax should be simple, as action creators themselves are not supposed to alter the data they are passed, so the correct action type and the payload should be verified. It is possible to reorganize multiple action creator arguments into single data structure if the standard in the organization or project is to use just one payload field. A noteworthy detail about the assertion is, that the ‘toBe’ assertion used earlier should no longer be used, as it will compare the object instances, i.e. the memory locations of the objects, which will naturally always fail. Instead, the assertion ‘toStrictEqual’ is used, which will recursively check the objects fields, values and types to match (Facebook, 2019e).

***Listing 9**: User action creator tests*

```
1.  describe('User action creators', () => {
2.    test('Save site action', () => {
3.      const expected = {
4.        type: actions.SAVE_SITE,
5.        payload: 'https://www.example.com'
6.      };
7.      expect(saveSite('https://www.example.com')).toStrictEqual(expected);
8.    });
9.
10.   test('Logout action', () => {
11.     const expected = {
12.       type: actions.LOGOUT
13.     };
14.     expect(logout()).toStrictEqual(expected);
15.   });
16. });
```

The test cases implemented, presented in Listing 9, are two synchronous action creators. Both are related to user state. The first action (lines 2 – 8) accepts one argument, which is the address of the LXP instance the user is connecting to. For variety, the second action (lines 10 – 15) takes no arguments, as it signals that the current user has logged out of the application.

6.3.2 Testing asynchronous action creators

Asynchronous action creators are more complicated to test than synchronous action creators. The pattern of asynchronous action creators is that a function starts the asynchronous task, dispatching an action, and when the task ends, another action is dispatched to commit the results to the store. The syntax of this pattern is presented in Listing 10. The difference now compared to the synchronous actions is that calling one function dispatches two actions, usually at different times. To verify that both actions are dispatched, and that the payloads are correct, test scaffolding is needed. The store must be replaced by a stub, which can be queried if the actions were dispatched on it, and the asynchronous action is replaced by a stub function that always returns an expected value. (Garreau & Faurot, 2018)

***Listing 10**: Asynchronous action creator example*

```
const actions = {
  CREATE_TASK: 'CREATE_TASK',
  FINISH_TASK: 'FINISH_TASK'
};

const startTask = () => ({
  type: actions.CREATE_TASK
})

const finishTask = (data) => ({
  type: actions.FINISH_TASK,
  payload: data
});

const createTask = () => {
  return async dispatch => {
    dispatch(startTask());
    const taskResult = await doAsyncThing();
    dispatch(finishTask(taskResult));
  }
};
```

As explored in section 6.1 already, Jest provides tools to mock functions. To create a stub store, a new package is needed. Redux-mock-store allows the definition of stub stores, which can be queried for the actions that are dispatched upon them. Another package that was already used in the application but plays a role in this topic is redux-thunk. Its purpose is to allow asynchronous actions in the first place, by delaying the evaluation of the action until the asynchronous task has been handled, so a promise (JS jargon for an object representing an unfinished task) won't be passed onto a reducer. The use of these packages yields the example syntax presented in Listing 11. The asynchronous task(s) used in the action should be mocked entirely (lines 6 and 9), as often these are calls to external APIs, and the focus is on the actions themselves, not on the functionality of the API. A stub store is created (lines 5 and 18), and it is given the redux-thunk package to enable handling of asynchronous actions. Finally, the assertion itself (lines 20 – 22) is placed in a then-clause (JS jargon to wait for the task to finish), which queries the stub store and expects to find the pre-defined actions to be dispatched. (Garreau & Faurot, 2018; Redux, 2020)

This pattern was applied to an asynchronous task that fetches user information from the backend subsystem. The action itself used the async-await JS syntax, which had to be refactored to use the callback-based then-syntax to allow action to return the task object while it is in progress, so that the test wait for the task to finish and then assert on the result.

The full source code for the testing of an asynchronous task is presented in appendix 3. When dealing with network requests, it is important to remember to include the negative result (lines 30 – 36). A helper function (lines 8 – 15) was used to reduce code duplication.

***Listing 11**: Example pattern of testing an asynchronous task*

```
import MockStore from 'redux-mock-store';
import thunk from 'redux-thunk';
import doAsyncThing from '../asyncThings'; // imagined module

const mockStore = MockStore([thunk]);
jest.mock('../asyncThings');

test('Async action', () => {
  doAsyncThing.mockResolvedValue('Some data');

  const expected = [{
    type: actions.CREATE_TASK
  }, {
    type: actions.FINISH_TASK,
    payload: 'Some data'
  }];

  const store = mockStore();

  return store.dispatch(createTask()).then(() => {
    expect(store.getActions().toStrictEqual(expected));
  });
});
```

6.3.3 Testing reducers

Reducers are the part of Redux that concretely define the new state after a change is requested. Testing reducers is simple, as they are pure functions, which means they always provide the same output for a given input and have no side-effects outside of the function. The input given is a state and an action, and the output is a state with the action applied to it, as can be seen in Listing 12. When defining a reducer, the default state is given to the function as the default option for the first argument. The body of the function consists of identifying the correct action with a switch-statement, and then returning a newly generated state object with the action applied to it. This interaction leads to reducer tests having simple syntax and clear assertions. (Garreau & Faurot, 2018; Redux, 2020)

Listing 12: An example reducer

```
const reducer = (state = { counter: 0 }, action) => {
  switch (action.type) {
    case actions.INCREMENT_BY:
      return { counter: state.counter + action.payload };

    case actions.DECREMENT_BY:
      return { counter: state.counter - action.payload };

    default:
      return state;
  }
};
```

Testing Redux reducers requires no extra packages, because no scaffolding or side-effects are involved. A reducer can be called without being associated with a store. Tests can be written for passing actions, like in Listing 13, but other features of reducers that can be verified are unrecognized actions (wrong action type or no action at all), which should return the passed state without changes, and returning the default state when no state object to be modified is passed. If no state object is passed but an action is, the modification will be done against the default state defined for the reducer.

Listing 13: An example test for the reducer in Listing 12

```
test('Incrementing state', () => {
  const startState = { counter: 15 };
  const expected = { counter: 20 };
  const action = {
    type: actions.INCREMENT_BY,
    payload: 5
  }

  expect(reducer(startState, action)).toStrictEqual(expected);
});
```

Only one reducer was deemed necessary to demonstrate reducer testing, as the pattern of the test itself does not change for different reducers or actions. Two tests shown in Listing 14 were written, one to ensure the default state, and the other to test one action. An action must be passed to the reducer, because the reducer checks the actions type, and due to the ordering of the arguments, a state object must be passed too. The state object and action passed (line 3) are therefore made empty as not to affect the default state. The second test uses JS's spread-notation (line 8) to copy the default state object's contents and apply the expected change (line 9).

***Listing 14**: User reducer tests*

```
describe('User reducer', () => {
  test('Returns default state', () => {
    expect(reducer(undefined, { type: null })).toStrictEqual(initialState);
  });

  test('Saves site', () => {
    const expected = {
      ...initialState,
      site: 'test'
    };
    expect(
      reducer(initialState, { type: actions.SAVE_SITE, payload: 'test' })
    ).toStrictEqual(expected);
  });
});
```

6.4 Integration tests for subsystems and React

Several aspects of integration exist in the application that are worthwhile. As covered in section 4.2, view testing can be extended from unit tests to integration tests by rendering full depth trees. Components also often integrate other modules of code in them, so replacing stubs and drivers from components is effectively integration. A common driver for a component is the state management wrapper that provides the global state store for the component. Arguably the most important integration however is the integration with the remote subsystem, Valamis LXP backend. The code for the interface is automatically generated with a tool called Swagger. The functions that call the endpoints of the API can be tested to verify correct integration with the backend.

To test component composition integration, view testing techniques can be used, but instead of shallow rendering, normal rendering is used to observe the full render tree. To test Redux integration with components, the Redux-wrapped components are used and the use of the state data from Redux in the component can be checked. The backend integration is tested by setting up a stub server that the client will send requests to instead of the remote server, and the requests and dummy responses that arrive at the client can be verified.

6.4.1 Component composition integration testing

While React components may be tested fine in isolation, testing them together is also important. Integrating React components together in a test is as simple as doing a normal test render instead of a shallow render, as mentioned in section 4.2. The composition tree is fully realized, creating a depth-first test like a sandwich decomposition test. The starting point for the tree can be arbitrarily picked, but the tree depth is not limited. Starting at different points in the tree allows the definition of specific properties that are characteristic for the selected root node, for example defining the contents of a list when selecting the list parent component as the root node.

The technique for doing React composition integration testing this way does not change based on the component. The possible data and external drivers for the component should be defined in such a way that the child components exhibit different outputs, covering the result space. For this reason, a list component from the application was chosen, and in the test cases it was given several typical inputs as data: no data, one element, and many elements.

***Listing 15**: An example of full depth rendering and querying*

```
test('List generates children', () => {
  const component = renderer.create(
    <ListComponent
      data={['Harry', 'Richard', 'William']}
      renderItem={({ item }) => <ListItem data={item} />}
    />
  );
  expect(component.getJSON().children).toHaveLength(3);
});
```

The tools to conduct this testing are the same as before, but this time react-test-renderer's full depth rendering is utilized with the help of the querying possibilities as seen in Listing 15, which allows the verification the structure of the composition tree. In shallow rendering, it usually is enough to verify that the component does not throw errors when rendering, but with integration testing there is a larger context to observe. Snapshot testing with Jest's 'toMatchSnapshot'-assertion can be used with full depth rendering, but this method is very sensitive to propagation of changes, as if any of the components in the tree are changed, the snapshot will be outdated.

The full depth rendering should employ some queries as to how the children are rendered, because incomplete or incorrect composition may lead to valid syntax, but the tree is semantically incorrect. This is demonstrated in Listing 16, the test case from the application, that a list is populated with data, and while the list might render correctly, it is queried afterwards that the child elements are rendered in correct amount and of the correct component type. (Facebook, 2019f)

***Listing 16**: Full depth rendering with a query on the child lcomponents*

```
test('Lesson card list renders with children', () => {
  const data = [1, 2, 3].map(i => ({ id: i, title: `item ${i}` }));
  const images = { fallback: null };
  const item = renderer.create(
    <LessonListContainer
      lessons={data}
      navigation={{ navigate: jest.fn() }}
      images={images}
    />
  );

  const children = item.root.findAll(child => child.type.name === 'LessonCard');
  expect(children).toHaveLength(3);
});
```

6.4.2 React and Redux interoperability testing

As mentioned in section 4.3, Redux is useful in managing the complexity of state information in a hierarchical, composition-based application. Accessing a store and dispatching actions from a React component can be done by using the interfaces provided by Redux, but tricky passing of Redux variables is needed to every component that requires access to its functionality. Fortunately, the use of Redux and React together is a common pattern, so interoperability libraries have been developed. Specifically, react-redux is used in this application and widely elsewhere in the industry to bridge the gap.

Four tools offered by react-redux are utilized: the ‘provider’ component, the ‘connect’ component decorator, the ‘mapStateToProps’ store mapping, and the ‘mapDispatchToProps’ dispatcher mapping. The provider is a container-component that should be placed high up in the composition tree, preferably near the root so that it can wrap the whole application, that does its namesake function: provides access to the store for all the child components. The provider is given the Redux store as a property. The connect component decorator is a function that lets a React component access the store that the provider has been passed. Connect returns a new component that contains the original component as a child, a pattern called higher order components in React. The connect function can also be given two optional parameters: ‘mapStateToProps’ and ‘mapDispatchToProps’. These are data structures that contain the mappings of what parts of the store and which actions (defined by action creators) will be made available to the connect-wrapper component. They are used to promote low coupling: not every component should have access to every action and every part of the state.

To test the interoperability of React and Redux, the higher order component should be tested. Drawing a comparison to unit and view testing, which test the components and their logic in isolation, the integration testing of React and Redux aims to verify that the mappings between the independent Redux store and actions to a component that reads and calls them through a higher order component communicating with the provider work as expected. To achieve this, a Redux stubbing is recommended. A stub store with known data and actions is created, passed to a provider, and the connect-decorated component is then observed and manipulated. Redux-mock-store, already mentioned in sections 4.3 and 6.3.2, allows the creation of stub stores that can then be passed to a provider, demonstrated in Listing 17.

Listing 17*: Example usage of a stub store for testing connected components*

```
const stubStore = mockStore()({
  items: ['Alice', 'Mary', 'Jane']
});

test('Connected component', () => {
  const item = renderer.create(
    <Provider store={stubStore}>
      <ListWithConnect />
    </Provider>
  );

  const children = item.root.findAll(
    child => child.type.name === 'NameItem'
  );
  expect(children).toHaveLength(3);
});
```

The above example displays how state mapping can be tested. The mapping of actions can be verified by interacting with the components. For some components this may be difficult, as it may require complicated querying and sequential interactions with the render. This lends itself to be more efficient to test as a part of a behavioural system test, where complicated interactions are happening anyway. For simple components where the actions are tied to elements near the root of the tree and dispatching them does not require long sequences of interaction, action mapping is more available to test. To verify the results of action mapping, the same technique as used in section 6.3.1 (querying actions dispatched to the store) can be used, or by querying the state of the store and calculating the difference between the initial state given to the store.

***Listing 18**: Redux connection testing for state and action mapping*

```
1.  const store = MockStore()({
2.    users: { user: { name: 'testName', id: 0 } },
3.    images: { users: { 0: null } }
4.  });
5.
6.  test('Profile connect', () => {
7.    const nav = { closeDrawer: jest.fn() };
8.    const item = renderer.create(
9.      <Provider store={store}>
10.       <ProfileContainer navigation={nav} />
11.     </Provider>
12.   );
13.
14.   const logout = item.root.findAllByType(TouchableOpacity)[1];
15.   logout.props.onPress();
16.
17.   const name = item.root.findAllByType(Text)[0];
18.
19.   expect(store.getActions()).toContainEqual({ type: 'LOGOUT' });
20.   expect(name.props.children).toBe('testName');
21. });
```

Both types of mappings were implemented into the test case chosen for the application. In Listing 18 the test is initialized by creating the stub store (lines 1 – 4). Some data is fed into the initial state to verify the mapping. The test itself renders the connected component wrapped in a provider with the stub store (lines 8 – 12). An interactable element is queried from the render and a tap is simulated on it (lines 14 – 15). This element is a log-out button, which will dispatch an action on the store. A text component that has the user's name mapped to it from the state is also queried (line 17). Finally, assertions on the dispatched action and state data are made on the queried elements (lines 19 – 20).

6.4.3 Remote subsystem interface testing

When testing with remote subsystems, best practice is to test both ends individually, and then under system testing to cover both at the same time, but as stated in section 4.4.4, the scope and reasonable effort in this thesis is limited to only client testing. As a result, only the compliance to the specification given to the client is verified, but whether the backend provides the same interface or not, which could effectively prevent the system from working, cannot be verified.

The specification of the backend interface in this project is supplied in a standardized format, built with a toolkit called Swagger. This toolkit allows the definition, documentation, deployment and debugging on APIs. The concrete result for the client is that the API definition is built into a markup file, which contains the available endpoints, parameters associated with them, the allowed HTTP-verbs, and expected response data structures. A library called swagger-client can read this markup and automatically generate functions that expect the mandatory parameters, so that the developers do not have to deal with the domain names, endpoint addresses and browsing the definition markup.

When the goal is to verify that the definition markup and the library reading it are working as expected, the assertions should be placed on if the outbound requests are heading to the right endpoint address and if they have the necessary HTTP-request data included, such as headers, query parameters and body. Several technical options exist to creating a stub in the communication flow with a remote subsystem that can identify this information. The first option is to replace the function that swagger-client uses to send a network request with a stub, so that a request is never sent, and the stub can observe the details that swagger-client passes it. The second option is to create a stub server, to which swagger client is pointed at and allowed to send requests. When asked if the company experts have a best practice established in testing one-sided API interaction, the second option was preferred.

Stub server creation is a common pattern in testing. The company experts recommended a JS library meant specifically for this task called nock. Nock is a “HTTP server mocking and expectations library" (Teixeira, 2019). As with other JS libraries, it can be installed through a package manager, in this case Yarn, with the command 'yarn add nock --dev'. The usage and syntax of nock is straightforward. An address to create the stub server for is given, and the endpoints with their HTTP-verbs are defined with a corresponding HTTP response code and the contents of the reply, as can be seen in Listing 19.

***Listing 19**: Example stub server setup with nock (Teixeira, 2019)*

```
const nock = require('nock')

const scope = nock('https://api.github.com')
  .get('/repos/atom/atom/license')
  .reply(200, {
    license: {
      key: 'mit',
      name: 'MIT License',
      spdx_id: 'MIT',
      url: 'https://api.github.com/licenses/mit',
      node_id: 'MDc6TGljZW5zZTEz',
    },
  })
```

***Listing 20**: Test for verifying a swagger endpoint*

```
1.  test('Swagger sends expected course request', () => {
2.    expect.assertions(2);
3.    const server = nock('https://salamis.valamis.io')
4.      .get('/delegate/courses/list/my')
5.      .query(true)
6.      .reply(function() {
7.        expect(this.req.path).toBe(
8.          '/delegate/courses/list/my?withGuestSite=true'
9.        );
10.       return [200, 'mock response'];
11.     });
12.
13.   return getClient()
14.     .then(client =>
15.       client.apis.default.get_courses_list__option_(
16.         { option: 'my', withGuestSite: true },
17.         {
18.           server: 'https://salamis.valamis.io/delegate'
19.         }
20.       )
21.     )
22.     .then(resp => expect(resp.data.toString()).toBe('mock response'))
23.     .finally(() => server.done());
24. });
```

In Listing 20 nock is applied to create a test for an endpoint in the LXP backend to fetch course information. The server stub is created (lines 3 – 11), and one assertion is placed in the reply function, which checks that the request that is being replied to was sent to the exact expected address (lines 7 – 9). After setting up the server, the swagger-client request is sent. A helper function (line 13) forms the client from the specification file, and then uses a generated function to send a request to the stub (lines 15 – 21). After the request gets a response, it is asserted to contain dummy response (line 22) from the stub server to ensure that it didn't query a remote server. Finally, the stub server is shutdown, regardless if the test passed or not (line 23). To note is that since there is multiple asynchronous assertions present in the test, Jest can pass the test only if two valid assertions are made (line 2).

This method of testing extends beyond Swagger to any generated API definition for a client. The very same technique can be used in any JS project, and the pattern can be utilized in any system with remote subsystems, although the syntax and details might vary based on the tooling and language.

6.5 System tests with Appium

This section presents the implementation of the system tests. The first subsection describes the background for the testing: the testing scoping, context and reasoning. The second subsection details the practicalities of facilitating and conducting system tests with Appium.

6.5.1 Device testing on mobile platforms

As mentioned in sections 4.5.3 and 6.4.3, the testing effort can be focused only on the client application, which limits the options of useful system testing. No stress or load testing is necessary as each individual client will be used by one user at a time, while backend subsystem will serve multiple users and would benefit from these non-functional tests. Heavy focus shouldn't be placed on system testing in the scope of all testing, as it is not conducted as often and the benefit over manual testing is the lowest of the types of testing presented in this thesis. The system test type of choice, end-to-end testing, covers a use case. The scoping of the test is a full system stack round-trip test. With Appium, the test can be conducted as device testing. Company testing hardware was used, from both major smartphone OSs': a Samsung Galaxy S9+ (Android version 9) and an iPhone 8 (iOS version 12.4). In addition to the physical device testing, emulators for both major OSs' will be used.

End-to-end testing needs an objective that the system must fulfil. In the behavioural perspective, a functionality or feature that enables behaviour or specific use. The company has made specification for what functionality is in the application in the form of user stories. These stories have defined acceptance criteria, which form the concrete pass or fail conditions for an end-to-end test. Two user stories with acceptance criteria will be covered by the empiricism of this thesis. The test cases will serve as an example of how to build an end-to-end test (and all the necessary pre-requisites) with Appium. The user stories are related to training events that can be joined and viewed in the application. The detailed acceptance criteria are for both to navigate to the event details, for the join-related story a interactable button that joins the event must be present, and for the viewing-related story the information about the training event must be present and readable even without joining the event.

6.5.2 Creating end-to-end tests with Appium

The first step to creating end-to-end tests with Appium is to install Appium itself. This process is two-fold, as Appium has a client-server architecture (covered more in section 4.5.3). The server installation is global, so that it only needs to be installed once no matter how many projects on that specific computer use Appium, but the client must be installed per project basis, as it is usually specific to the language the project uses. The installation process described in this section is from a MacOS perspective, as Valamis uses Macs as their primary computers.

The Appium server can be installed in two ways: via the command line, or with by downloading and installing their desktop application. The command line installation is executed with just one command, ‘yarn global add appium’. This will install the server package and all its dependencies. The other option is downloading the desktop package for the server. This is done from the Appium website by pressing on the button labelled “Download Appium”, and then selecting the Mac package from the list of binaries. (JS Foundation, 2019b)

After installing the server, drivers for the specific platforms, in this case Android and iOS, must be prepared. The driver for iOS, XCUITest, requires the installation of an iOS-specific dependency manager, Carthage. Carthage is installed from the command line with ‘brew install carthage’. If the testing is limited to emulators, the configuration for iOS is ready at this point. To accommodate physical devices, a verified Apple developer team identity is required for code signing and provisioning purposes. This identity is listed on the Apple Developer website. The identity can be provided for the signing and provisioning processes with a configuration file for XCode (which is the iOS development environment) by creating a file named ‘.xcconfig’ on the computer, and filling it as presented in Listing 21. (JS Foundation, 2019c)

Listing 21*: Configuration file contents for physical device testing on iOS. <Team ID> must be replaced with the registered identity visible on Apple’s website. (JS Foundation, 2019c)*

```
DEVELOPMENT_TEAM = <Team ID>
CODE_SIGN_IDENTITY = iPhone Developer
```

The driver for Android, UIAutomator2, requires setting several environment variables, and ensuring that the necessary tools are installed. Java development kit (JDK) must be installed, and Android Studio (which is the Android development environment) must be installed. In Android Studio, it should be ensured that the SDKs for the API levels for the planned tests are installed. The first environment variable is JAVA_HOME, which should be set to the home folder of JDK. The second environment variable is ANDROID_HOME, which should be set to the Android SDK folder. Both variables can be set by typing the adjusted contents of Listing 22 to a shell initialization file. (JS Foundation, 2019d)

Listing 22*: Setting environment variables in MacOS*

```
export <VARIABLE_NAME>="<path>/<to>/<variable>"
```

The client installation for Appium is straightforward. Appium has a list of recommended client libraries, including JS. Out of the options available, WebdriverIO has the most comprehensive documentation, most auxiliary libraries (for example integrating with Appium and a command line interface), and most followers, contributors, releases and commits on GitHub. WebdriverIO can be installed with ‘yarn add @wdio/cli --dev’.

Nominally this installs the command line interface and special test runner for WebdriverIO, but as they depend on the core packages, they will be installed as well. After the package is installed, the configurator is run with ‘./node_modules/.bin/wdio config’. In the configurator the test runner, test definition location, reporting style, synchronicity, backend location and services like Appium are defined. These settings are saved into ‘wdio.config.js’-file in the project folder root. These configurations are by default meant for browser testing, so they must be modified to fit mobile testing with Appium. This is achieved by adapting the desired capabilities the client will request from the Appium server when an automation session is started. The adapted configuration file for emulator testing is included as appendix 4. Multiple capabilities can be defined, meaning many automation suites can be run from a single configuration file. This allows easy batching and separation of session, for example to emulator runs on a build server and physical device runs reserved for on-site testing. (OpenJS Foundation, 2020a; JS Foundation, 2020)

It should be noted that Appium is only one possible backend driver that can be configured for Webdriver.IO. The parts that make the presented end-to-end tests specific for mobile devices in appendix 4 are the capability definitions (lines 8 – 28) and Appium service definitions (lines 45 – 48). If these parts were to be replaced with Selenium-specific details, such as which browser to use, the tests could be converted for web-testing.

When the client and server are ready, the tests themselves can be written. The basic flow of writing Appium tests is identifying elements from the screen, interacting with them, and then asserting on the outcome of the interaction. WebdriverIO allows identifying elements, or in this case RN components, with the use of selectors. The selector function is called with the dollar sign, and it is passed the selector string as a parameter. Some selectors are presented in Listing 23. The first selector is based on accessibility identifiers, which is a way to uniquely label components. The second selector uses the class name for the component’s underlying native implementation. The last selector is called xPath, which is a way to traverse the VDOM tree to a specified node. The xPath selector is not recommended by Appium, as it has poor performance. (OpenJS Foundation, 2020b; JS Foundation, 2019e)

Listing 23: *Selectors available in Appium and WebdriverIO*

```
$('~someTestID');
$('android.widget.DatePicker');
$('//parent/child/grandchild[2]');
```

To assign accessibility identifiers to RN components, two properties are passed: ‘testID’ and ‘accessibilityLabel’ with the desired string value. When elements have been selected, they can be interacted with. Common interactions include clicking (which is simulated by a tap), setting values with the keyboard, and scrolling. Finally, the assertion is done just like in any other language, but in this case instead of using Jest’s assertions (‘expect’), Node.js built-in assertions are used, as WebdriverIO’s test runner is not based on Jest.

When tests are finished, they can be run in four steps. The first step is by starting the Appium server, either by launching the desktop application or with the command line interface. After Appium is started, the RN bundler which will load the application onto the device must be started by executing ‘yarn start’ in the project root. Depending on if the tests are run with emulators or physical devices, they must be prepared too. In the case of emulators, the emulators themselves must be started, and in the case of physical devices, they must be connected to the computer and a development/debugging state that allows incoming connections. Finally, the tests themselves can be launched by using WebdriverIO’s command line tool on the targeted config file, in this case the tests were separated into the physical device configurations and emulator configurations, so the emulator tests are run with ‘./node_modules/.bin/wdio wdio.emulator.conf.js’.

The test cases themselves are identical for iOS and Android because of the API abstraction mentioned in section 4.5.3. If the additional steps covered earlier to facilitate manual testing are done, the test cases can be used with physical devices as well as manual devices without any changes to the test implementations. The implementations of the test cases agreed upon in section 6.5.1 is shown in Listing 24. To abstract and reuse navigational interactions, some of the code was separated to helper functions, listed in Listing 25. A pattern is visible in the helper function interactions: identify and select the element (lines 2, 8, 14), wait for the selected elements to be available to interact with (lines 3, 9, 15), and interact with the elements (lines 4, 10, 16). The test moves through several views and repeats these steps for each to move on to the next view. More detailed assertions could be made based on the content of the fields in Listing 24, but this falls into the area of integration testing on fetching the correct data from the backend and component testing that the passed data is rendered correctly.

***Listing 24**: End-to-end test cases implementations*

```
describe('Event tests', () => {
  it('Join event', () => {
    debugLogin();
    openEventsList();
    openAllEvents();
    openEventCard(0);

    const joinButton = $('~joinEvent');
    joinButton.waitForDisplayed();
    joinButton.click();

    driver.pause(1000);
  });

  it('Event details can be read', () => {
    debugLogin();
    openEventsList();
    openAllEvents();
    openEventCard(0);

    driver.pause(1000);

    assert.ok($('~headerImage'));
    assert.ok($('~titleText'));
    assert.ok($('~placeText'));
    assert.ok($('~participantText'));
    assert.ok($('~calendarText'));
    assert.ok($('~descriptionText'));
  });
});
```

***Listing 25**: Helper functions for end-to-end testing*

```
const openEventsList = () => {
  const calendarButton = $('~calendarButton');
  calendarButton.waitForDisplayed();
  calendarButton.click();
};

const openAllEvents = () => {
  const allEventsButton = $('~allEvents');
  allEventsButton.waitForDisplayed();
  allEventsButton.click();
};

const openEventCard = index => {
  const eventCard = $(`~eventCard-${index}`);
  eventCard.waitForDisplayed();
  eventCard.click();
};
```

After the test is finished, the result is printed in the same console window that started the WebdriverIO's tests. The test results list each test file, the test suites in the file, and the test cases inside the suite. The report also describes the path to the application package that was used in the testing and the device identifier. A sample report is displayed in Listing 26.

***Listing 26**: WebdriverIO's output report*

```
[0-0] RUNNING in /Users/matiassalohonka/mobileapp/android/app/build/outputs/apk/de-
bug/app-debug.apk - /test/specs/test.js
[0-0] PASSED in /Users/matiassalohonka/mobileapp/android/app/build/outputs/apk/de-
bug/app-debug.apk - /test/specs/test.js

 "spec" Reporter:
------------------------------------------------------------------
[emulator-5556 LINUX 10 #0-0] Spec: /Users/matiassalohonka/mo-
bileapp/test/specs/test.js
[emulator-5556 LINUX 10 #0-0] Running: emulator-5556 on LINUX 10 executing /Us-
ers/matiassalohonka/mobileapp/android/app/build/outputs/apk/debug/app-debug.apk
[emulator-5556 LINUX 10 #0-0] Session ID: d8471b45-5633-44a3-81b9-6923e93972fd
[emulator-5556 LINUX 10 #0-0]
[emulator-5556 LINUX 10 #0-0] Event tests
[emulator-5556 LINUX 10 #0-0]    ✓ Join event
[emulator-5556 LINUX 10 #0-0]    ✓ Event details can be read
[emulator-5556 LINUX 10 #0-0]
[emulator-5556 LINUX 10 #0-0] 2 passing (45s)


Spec Files:  1 passed, 1 total (100% completed) in 00:00:53
```

At the time of writing, Appium (version 1.16.0) had a problem with iOS touch interactions, which prevents clicking or tapping motions in the test. This problem was reported on the issue tracker of the tool, and an ongoing investigation as to what causes the issue had not yet yielded solutions. In practice this means that most iOS end-to-end tests are broken with Appium, as the interaction options are severely limited. This is also the reason why iOS capabilities are presented but disabled in the attached emulator configuration file. Android testing with Appium worked normally and the test cases were implemented and verified with Android systems.

6.6 Test automation pipelines in Azure DevOps

This section first defines how Azure DevOps Pipelines work in general and the syntax defining pipelines is explained with the help of a default pipeline. In later subsections pipelines are configured to automatically test the RN application, and the configurations for each pipeline are presented and explained in detail.

6.6.1 Azure Pipelines basics

Test automation, and by extension CI, is managed with the Pipelines feature in Azure DevOps. A Pipeline is tied to a repository which can be hosted in several version control host services, Azure Repos included. The creation of a pipeline is started from the Pipelines menu in Azure DevOps when a project is viewed. First the connected repository host is selected, in this case Azure Repos. After this, the configurator shows the Azure Repos that the user has access to, one of which can be chosen to connect the pipeline to. The mobile application project repository was chosen. Azure Pipelines has template configurations for many common project types, such as Node.js web applications with various frontend frameworks, .NET systems, and native mobile projects. RN does not have its own template, so a generic starter template shown in Listing 27 was chosen. Finally, an editor is shown so the pipeline can be configured, and steps added.

***Listing 27**: Starter Pipeline template*

```
1.  # Starter pipeline
2.  # Start with a minimal pipeline that you can customize to build and de-
    ploy your code.
3.  # Add steps that build, run tests, deploy, and more:
4.  # https://aka.ms/yaml
5.
6.  trigger:
7.  - master
8.
9.  pool:
10.   vmImage: 'ubuntu-latest'
11.
12. steps:
13. - script: echo Hello, world!
14.   displayName: 'Run a one-line script'
15.
16. - script: |
17.     echo Add other tasks to build, test, and deploy your project.
18.     echo See https://aka.ms/yaml
19.   displayName: 'Run a multi-line script'
```

Azure Pipelines uses YAML Ain’t Markup Language (YAML) -files to configure the pipeline. YAML is a markup language, and Microsoft provides extensive documentation on how to write YAML and utilize the features offered by Pipelines. The default template is built from several parts. The first four lines are comments and ignored by the YAML parser. Lines 6 – 7 define when the pipeline will run. The options for triggering the pipeline are the following: push triggers, pull request triggers, and scheduled triggers. Push triggers activate when a commit is made to the version control system and defined criteria, such as selected branches or file path updates, are met. Pull request triggers activate when version control pull requests between branches are made. Similarly, to push triggers, pull request triggers can also be defined to selectively activate on specific branches or file path updates. The final trigger type, scheduled triggers, periodically activate regardless of repository activity. The period is defined in cron-syntax, and branches can be included or excluded from the trigger. Additionally, an option is available to only run the pipeline if changes have been made since last activation. (Microsoft, 2020a)

The third part of the template defines what agent the tasks will run on. An agent is the concrete system that will execute the instructions defined in the configuration. Pool means a collection of agents, which can be private or provided by Microsoft. Microsoft offers several different types of pools, which are categorized by the operating system of the agents: Windows Server, Ubuntu, and macOS. Each part of the pipeline, whether it is a stage or individual parts of a stage, can be configured to run on different pools. If a Microsoft-hosted pool is used, the agent type must be specified with the 'vmImage'-field. (Microsoft, 2020a; Microsoft, 2020b)

The final part of the template defines steps. These define the commands that the agent running the pipeline will execute. Each step runs in its own process, but they share the workspace, meaning file system changes are shared but environment variables are not. Steps must be defined the execution type. The available execution types are Script, Bash, PowerShell, Checkout, Task, and Template. The first three use different command line interfaces to execute commands listed in the step definition, Checkout is used to give custom source code checkout commands, Task is keyword for launching pre-defined tasks that Microsoft has created, such as building and deploying on common platforms, and Template is the keyword to reuse steps defined elsewhere. When writing command line steps, it is often necessary to write multiple commands. This is accomplished by using the pipe symbol after the type of the task, as seen on line 16. (Microsoft, 2020a)

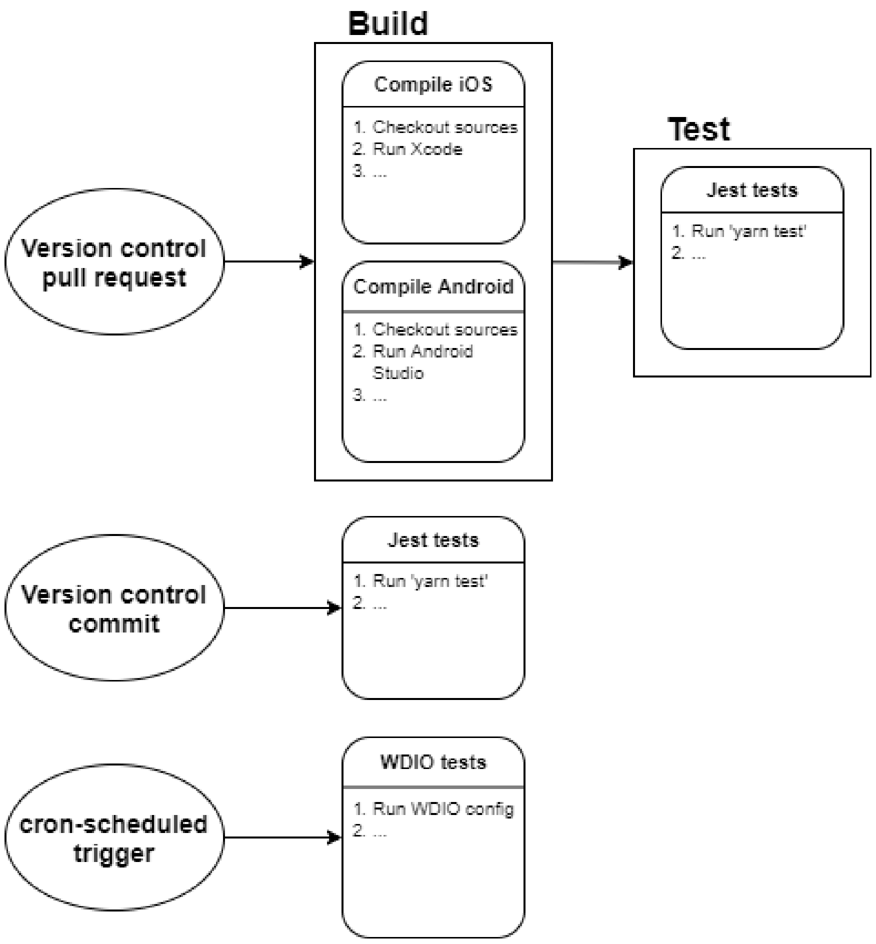


Figure 15*: Model of automation for application project. The leftmost ovals represent triggers for pipelines, the rectangles represent stages, the rounded rectangles represent jobs, and the numbered items inside the jobs represent steps.*

An important part of defining more complex pipelines that is not demonstrated in the template is the use of jobs and stages. Jobs and stages can be used to structure pipelines and form relationships between steps and how they should react to each other. These relationships applied to the RN application project are shown in Figure 15. Jobs are collections of steps that may run conditionally or depend on earlier jobs. Stages are logical collections of jobs. Stages run sequentially. The conceptual benefits and reasoning for splitting a pipeline are explored in section 5.2. In the case of a simple pipeline with only one stage and/or one job, the stage and/or job definitions can be omitted from the configuration. As can be seen from Figure 15, there are three distinct pipelines for the project. Each pipeline must be defined separately, as they use different triggers and need to run independently from each other. The following subsections present the configurations for each pipeline. (Microsoft, 2020a; Microsoft, 2019c)

6.6.2 Commit-triggered pipeline

Listing 28*: YAML configuration file for CI pipeline*

```yaml
1.  trigger:
2.    batch: true
3.    branches:
4.      include:
5.        - '*'
6.
7.  pool:
8.    vmImage: 'ubuntu-latest'
9.
10. steps:
11.   - template: pipeline/cache-yarn.yml
12.   - bash: yarn test
13.     displayName: 'Run Jest'
```

This pipeline, presented in Listing 28, will run on every commit in every branch to execute the basic Jest tests, which are the unit, view, state, and integration tests. These tests are meant to be the fastest to execute and to do basic verification and regression testing. Even with the speed of the pipeline in mind, it is possible that commits can arrive faster than the pipeline can process, so run batching is used alleviate overlapping runs. Caching of JS modules will also speed up the pipeline significantly, as modules are seldom added or removed from the project, but without caching over 800 megabytes of modules would have to be downloaded every time the pipeline runs. The commit-triggered CI pipeline YAML configuration file is presented in (Microsoft, 2020c; Microsoft, 2019d)

The configuration is structured to first define the pipeline to run on every commit in every branch (lines 1 – 5) and to use the default agent, which happens to be the latest long-term support Ubuntu distribution (lines 7 – 8). The caching uses a template (line 11, shown in Listing 29). The final step (lines 12 – 13) run the Jest tests. What is not seen in this configuration that is automatically filled in by Azure Pipelines is the stage and job configurations, as they are not needed in a single-job pipeline. The pipeline also automatically checks out the sources from the branch before the defined steps are executed. After the steps, the cache task saves any changes in the cached folders, so that the next run’s cache task can use them. Listing 29 shows a template that uses a pre-defined task to cache

JS modules. Line 4 defines the key which is used as the identifier for the cached information. The last part, ‘yarn.lock’, calculates a hash from a local file. This hash will reveal if dependencies have changed and the cache is outdated. Lines 5 – 7 identify the keys that will be used to restore data from the cache. Line 8 shows what folder will be saved from and restored to upon caching actions. (Microsoft, 2020d)

***Listing 29**: Pre-defined task for caching JS dependencies*

```
1.  steps:
2.    - task: Cache@2
3.      inputs:
4.        key: 'yarn | "$(Agent.OS)" | yarn.lock'
5.        restoreKeys: |
6.          yarn | "$(Agent.OS)"
7.          yarn
8.        path: $(Pipeline.Workspace)/.yarn
9.      displayName: 'Cache Yarn'
10.
11.   - bash: yarn --frozen-lockfile --cache-folder $(Pipeline.Workspace)/.yarn
12.     displayName: 'Fetch Yarn'
```

6.6.3 Pull request -triggered pipeline

This pipeline’s purpose is to verify that the branch is ready and safe to be merged into the mainline. This means the trigger of the pipeline is tied to pull requests against the main development branch only. In addition to running Jest tests, it will also compile the application for additional verification. The compilation process on iOS requires a Mac, so the default Ubuntu agent must be replaced with a macOS one. The YAML file for this pipeline is included as appendix 5.

For a pull request triggered pipeline, a trigger set in YAML cannot be used. Instead, the pipeline will be set for a specific branch as a branch policy to validate builds. This is done in Azure Repos. The configuring allows the pipeline to be mandatory for a pull request to be mergeable. The pipeline YAML can then be configured to never run (line 1), as the external trigger is outside of the YAML file. (Microsoft, 2020c; Microsoft, 2019e)

The pipeline is structured into two stages, Build and Test (lines 9 – 10 and 51), in which Build has two jobs, android and ios (lines 11 – 12 and 20), which compile the application for the respective platforms. Test contains only one job, test, which runs the Jest tests (lines

54 – 57), similarly to the CI pipeline in section 6.6.2. The android job begins by loading cached JS dependencies, and then navigates to the android folder in the project (line 16) and executes the Gradle wrapper (which is a build tool in the Java Virtual Machine ecosystem) to build the Android application (line 17). Compared to android, ios is much more complicated. ios begins by clearing out any possible artifacts from the previous builds (line 22). It then loads cached JS and iOS-specific dependencies (lines 25 – 26). After loading the dependencies, two iOS-specific preparation tasks, installing certificates and provisioning profiles (lines 28 – 40), take place. The details of these steps are outside the scope of this thesis; however, they use shared variables (lines 31 and 38 – 39) that are configured to hold secret keys. Finally, the application is built with the Xcode -task (lines 42 – 49). The task is passed inputs to use the project configuration files, and to build a debug-configuration. (Microsoft, 2020a; Microsoft, 2019f; Microsoft, 2019g; Microsoft, 2019c)

6.6.4 Scheduled pipeline

This pipeline runs on a schedule to execute the end-to-end system tests. The system tests are the slowest, so this pipeline will run at a quiet time of day to not block other pipelines by reserving agents. The pipeline runs the Appium tests described in section 6.5 on an emulator, as the Microsoft hosted agents do not have access to physical devices. This process necessitates building the application, launching Appium, launching the JS process that serves the application, downloading and launching the correct emulator, running the application on the emulator, and finally running the tests themselves. Of the three pipelines described, the scheduled pipeline arguably has the most complexity in it. The YAML file for this pipeline is included as appendix 6.

The scheduling of the pipeline is defined in cron-syntax. The supported cron syntax is defined in the pipeline triggers documentation. The schedule that was agreed upon with the company is the working days (Monday to Friday) during the week at 06:00. To allow scheduled triggers to work, pull request and CI triggers must be explicitly disabled in the YAML for the pipeline. Scheduling can also be defined in the Azure Pipelines web interface, and in the case that there are two scheduling definitions, the one defined in YAML will be ignored. The trigger definition is written on lines 1 – 9. (Microsoft, 2020c)

As mentioned in section 6.5.2, the iOS testing was not properly functioning at the time, which would lead to the pipeline constantly failing, so iOS tests were left out of the pipeline. The Android process (starting at line 18) being similarly to the pull request pipeline by fetching the dependencies and building the application. After building a debug build, the pipeline starts necessary background processes: Metro Bundler (lines 27 – 28), which serves the emulator the JS code on demand, and Appium (lines 30 – 31). The emulator must also be prepared, which includes three steps: downloading the necessary files for the emulator (lines 33 – 34), creating the virtual device (lines 36 – 37), and starting the virtual device (lines 39 – 40). The step for starting the virtual device is complicated, as it involves connecting to the virtual device with Android Debug Bridge to wait for it to finish starting. Not doing so would start the tests before the device is ready, and the tests would fail. After everything is set up, the tests are executed from the emulator Webdriver.IO configuration file (lines 42 – 44). This step includes a clause to continue the pipeline even if the tests fail, as the last remaining to steps are important for failure diagnostics. Webdriver.IO is configured to take screenshots in the case of a failing test (appendix 4, lines 37 – 41). The last two steps in the pipeline exist to save the screenshots as attachments to observe for the pipeline. This is done by first copying pictures inside the screenshots to a staging directory and then publishing the staging directory content as build artefacts. (Microsoft, 2018; Microsoft, 2019h; Microsoft, 2017; Microsoft, 2019i)

7 DISCUSSION AND CONCLUSION

This section lists the observations made in the research, primarily during the empiricism, and interpreted their significance or impact for the study. Further research topics that surfaced are presented, and the section is finished with the conclusions of the thesis work.

7.1 Observations made during the research

While the system testing approach of full stack end-to-end testing is useful, the benefits of it were not fully realized during the study. The client application was not completely integrated with the backend application because the development is still in progress. This partly changes the full stack nature of the tests to only test the client-side logic and UI. Other types of system testing outside of end-to-end testing through automating the UI could prove useful. But in this case, since the backend subsystem is shared for multiple separate systems, other types of system testing, such as load testing, have been done outside this thesis, but their results were not investigated and presented.

The development of end-to-end tests and specifically automating them in the pipelines faced technical difficulties. The findings and supporting company expert opinions are that WebDriver testing is flaky, in the sense that the tests are very timing-precise and variance in the performance of the test-executing host can cause the tests to sometimes fail, even when the tests have nothing changed between runs that were successful. The automating necessitated very long wait times between finding UI elements, making the tests run slow. A possible solution is to move to a different layer or scope as explored in section 4.5.2, which could result in eliminating emulators along graphical UI interaction from the test stack.

Many of the testing tools publicly available are made by foundations or groups that are non-profit or the tool is a secondary or even tertiary priority for them. Practically this means, that when people are not being paid for the development of the tool, or it isn’t their primary responsibility, urgency and liability in the development can be lost. This was highlighted when Appium tests were being implemented, as the bug that severely limits the iOS testing was reported on the official bug tracker 20 days ago, with no indication that a fix was actively worked on.

The choice of React Native as the application platform proved some challenges, but the technology has had time to mature and gather a developer community, so that other tooling and systems account for its use. Azure provides support for React, Android and iOS, but none for RN specifically. Fortunately, the native mobile documentation can be adapted to work in RN processes. Testing React components has limited options, as React is very popular in web development (and comparatively more popular in the web development space than RN is in the mobile development space), yet RN has key differences that affect component and rendering testing in particular. The biggest difference that was not immediately obvious is the lack of a real DOM and browser variables such as 'document' and 'window' that popular tools like Enzyme rely on to create full-depth renders. Finally, RN's handling of errors by displaying a JS stack trace and halting the application execution but not terminating the process caused Webdriver.IO tests to not detect error states, which lead to the attempt of continuing the tests even when a failure had occurred. Native mobile applications crash completely when errors occur, and Appium and Webdriver.IO are designed around this fact to detect errors, so RN's differing philosophy can prove difficult to debug runtime errors in automated tests in remote environments.

Most of the testing techniques and tools are generalizable to web-development at large outside of mobile application development. The use of JS and its ecosystem means the libraries and tools are mostly directly transferable, barring the some of the incompatibilities mentioned in the previous paragraph. An unexpected transferability is Webdriver.IO and Appium. As Appium is very strongly based on Selenium, the tests presented in this thesis can be used as a guide to develop end-to-end UI tests for web applications with minimal changes, as mentioned in section 6.4.3. The popularity of React and Redux both within Valamis and outside of it means large parts of this study are beneficial even to developers who have not and likely will not develop with RN. The client-server architecture is very common, and the pattern of testing integration with remote subsystems with stubs can be extended to web development and beyond.

Microsoft has created a comprehensive toolkit for automation in Azure Pipelines. When implementing the pipelines, it was noted that the platform offers a great deal of ready-made tasks for common operations in CI and beyond. Common programming languages, packaging systems and orchestration tools such as Maven, Go, Docker, Kubernetes, and Helm have tasks that reduce the workload of defining even complicated pipelines significantly. All the mentioned technologies are in use at Valamis, so while they were not covered in the empiricism of this study, they could arguably be easily adopted for use in the company. The triggers, stages, jobs and other features such as saving files as attachments to pipeline runs are completely independent from mobile development and can be utilized in any software project.

7.2 Further research

As mentioned in the delimitations of the scope for this study, SQA encompasses more than just testing. An area of further research is the wider application of SQA from a mobile (or RN) perspective. During the planning of the study, attention was drawn to the potential of metrics. Data gathered from static analysis during development and analytics and metrics on usage and errors from user testing and live operations have significant implications on development, design and SQA. Research could be directed into how should the data be gathered from an application such as this, what will it be used for and accounted for internally, and how to integrate this data into the automation framework or combine the automation and metrics into a single dashboard. This information could help manage the development process by diagnosing and predicting defects and inefficiencies earlier and more precisely. Static analysis on commit contents and differences could emphasize different tests and gather development practices data (for example the average amount of bugs introduced per type of change committed). This could improve development processes (such as code reviews and commit best practices) and automation (for example optimizing the test execution order). Another area of SQA that is very close to the topic of this thesis but not included in the scope is manual testing: how and when to conduct it, what are the specific test cases, and how is it integrated into the automated testing process. The company showed interest in automatically delivering testable builds to testers. Research could investigate the optimal delivery method and target platforms (which include physical devices, emulators, and device clouds). The manual testing feedback process should also integrate into the whole testing process described in this thesis.

An important topic of further research is extending the automation outside of testing. Continuous delivery and continuous deployment are topics that surfaced in the literature referenced in this thesis, and are claimed to be the natural progression of continuous integration and proven to be the next step in efficiency and competitive advantage (Forsgren, et al., 2018). The next step as for advancing the automation of this mobile application as visioned by the company was automatically delivering staging and production builds to mobile application stores. This ties into the previous idea of providing manual testers with builds to test, and also integrating into customers' delivery methods, such as mobile device management platforms.

A specific type of testing that would be beneficial considering the business case of the application is configuration testing. This type of testing was not relevant to the company at the time of writing but plans for the application point in the direction of large amounts of configurations being needed. The company aims to provide build-time customized applications to different customers, to apply the customers' branding, lock out specific features, and target the customer's backend. The automation and tests could be extended to account for different configurations. Further research could investigate the techniques and details for configuration testing and the relation of the type of customization to what kind of tests are performed.

7.3 Conclusion

The useful types of tests were identified based on the current company needs. The testing types and levels identified were verified to be relevant for the company needs with the company experts. Testing and automation methods, benefits and trade-offs were identified. Literature with arguments for the best methods and types was analysed and the findings presented in sections 4 and 5. The theoretical framework covers every layer of the V-model in testing and introduces testing React- and Redux-specific testing techniques. The system testing focus is placed on functional end-to-end testing, as non-functional testing for the remote subsystem is done elsewhere in the company. Potential in non-functional system testing for the application was recognized, but not deemed mandatory at this point.

POC implementations of tests were implemented. Every feature or module in the application had tests created for it, and the listings and explanations for the tests are presented in sections 6.1 – 6.5. These tests establish a baseline for every level introduced in the theoretical framework for the current project state, serve as useful examples for projects with related technologies (especially for web development), and as patterns for projects that cannot directly apply the examples in the listings. Concrete examples cover all types of modules, and possibilities in more system testing were recognized. The testing covers strictly the client-side subsystem, which is affected by the fact that the application is still under development and full system integration is not complete, but as such the remote sub-system integration testing was completed as far as possible. This work serves as an example of how to conduct full-scale testing of a RN application that utilizes Redux for state management and interacts with a remote subsystem through a pre-defined API.

A complete test automation implementation in the company selected service with the core types of triggers for pipelines and testing activities was made. The pipelines that were agreed upon with the company include a CI pipeline that is triggered on every version control check-in, a pull request pipeline that is triggered when version control branches are brought back into the mainline, and a periodic pipeline that runs every day at a quiet time to execute the slowest system tests. The full configurations and explanations for the automation are presented in section 6.6. These pipelines can easily be adapted to other technology stacks and projects by utilizing ready-made tasks and the trigger definitions that are explained. This provides the company with strong, extendable automation that the theoretical framework recognizes as necessary for a modern software company to conduct effective SQA.

REFERENCES

Abandah, H. & Alsmadi, I., 2013. Call graph based metrics to evaluate software design quality. *International Journal of Software Engineering and Its Applications,* pp. 1-12.

Airbnb, 2019. *Shallow Rendering.* [Online]
Available at: https://airbnb.io/enzyme/docs/api/shallow.html
[Accessed 17 January 2020].

Akshat, P. & Abshishek, N., 2019. *React Native for Mobile Development: Harness the Power of React Native to Create Stunning iOS and Android Applications.* 2nd ed. Berkeley: Apress.

Alpaev, G., 2017. *Software Testing Automation Tips: 50 Things Automation Engineers Should Know.* Berkeley: Apress.

Automate The Planet, 2018. *Generations of Test Automation Frameworks - Past and Future.* [Online]
Available at: https://www.automatetheplanet.com/generations-test-automation-frameworks
[Accessed 30 January 2020].

Axelrod, A., 2018. *Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects.* Berkeley: Apress.

Berner, S., Weber, R. & Keller, R., 2005. *Observations and lessons learned from automated testing.* St. Louis, Association for Computing Machinery, pp. 571-579.

Bordi, K., 2018. *Overview of test automation solutions for native cross-platform mobile applications,* Lappeenranta: LUTPub.

Brandi, P., 2019. *Android API Level, backward and forward compatibility.* [Online]
Available at: https://android.jlelse.eu/android-api-level-backward-and-forward-compatibility-10e6d31cb848
[Accessed 14 January 2020].

Chan, M., 2020. *React Patterns.* [Online]
Available at: https://reactpatterns.com/
[Accessed 27 January 2020].

Danielsson, W., 2016. *React Native application development - A comparison between native Android and React Native,* Uppsala: Uppsala University Library.

Facebook, 2019a. *Testing React Native Apps.* [Online]
Available at: https://jestjs.io/docs/en/tutorial-react-native
[Accessed 15 January 2020].

Facebook, 2019b. *Getting Started · Jest.* [Online]
Available at: https://jestjs.io/docs/en/getting-started.html
[Accessed 10 February 2020].

Facebook, 2019c. *Mock functions · Jest.* [Online]
Available at: https://jestjs.io/docs/en/mock-functions
[Accessed 10 February 2020].

Facebook, 2019d. *Snapshot Testing · Jest.* [Online]
Available at: https://jestjs.io/docs/en/snapshot-testing
[Accessed 11 February 2020].

Facebook, 2019e. *Expect · Jest.* [Online]
Available at: https://jestjs.io/docs/en/expect
[Accessed 12 February 2020].

Facebook, 2019f. *Test Renderer.* [Online]
Available at: https://reactjs.org/docs/test-renderer.html
[Accessed 13 February 2020].

Facebook, 2020a. *React Native - A framework for building native apps using React.* [Online]
Available at: https://facebook.github.io/react-native/
[Accessed 7 January 2020].

Facebook, 2020b. *Releases - facebook/react-native - GitHub.* [Online]
Available at: https://github.com/facebook/react-native/releases
[Accessed 14 January 2020].

Facebook, 2020c. *Jest.* [Online]
Available at: https://jestjs.io/en/
[Accessed 15 January 2020].

Facebook, 2020d. *Composition vs Inheritance.* [Online]
Available at: https://reactjs.org/docs/composition-vs-inheritance.html
[Accessed 16 January 2020].

Forsgren, N., Humble, J. & Kim, G., 2018. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations.* Portland: IT Revolution Press.

Fowler, M. & Foemmel, M., 2006. *Continuous Integration,* s.l.: s.n.

Frachet, M., 2017. *Understanding the React Native bridge concept.* [Online]
Available at: https://hackernoon.com/understanding-react-native-bridge-concept-e9526066ddb8
[Accessed 7 January 2020].

Gao, J., Bai, X., Tsai, W.-T. & Uehara, T., 2014. Mobile Application Testing: A Tutorial. *Computer,* Volume 47, pp. 46-55.

Garreau, M. & Faurot, W., 2018. *Redux in Action.* 1st ed. Greenwich(Connecticut): Manning Publications.

Google Developers, 2019. *Distribution Dashboard.* [Online]
Available at: https://developer.android.com/about/dashboards
[Accessed 14 January 2020].

Guo, P., 2008. *The benefits of programming with assertions (a.k.a. assert statements).* [Online]
Available at: http://www.pgbovine.net/programming-with-asserts.htm
[Accessed 15 January 2020].

Haller, K., 2013. Mobile Testing. *ACM SIGSOFT Software Engineering Notes,* Volume Vol.38(6).

Hevner, A., Salvatore, M., Jinsoo, P. & Sudha, R., 2004. Design Science in Information Systems Research. *MIS Quarterly,* Volume 28, pp. 75-105.

IEEE, 2010. *ISO/IEC/IEEE 24765:2010: Systems and Software Engineering -- Vocabulary.* s.l.:IEEE.

Johannesson, P. & Perjons, E., 2014. *An Introduction to Design Science.* s.l.:Springer International Publishing.

Jorgensen, P., 2008. *Software testing: a craftsman's approach.* 3rd ed. Boca Raton: Auerbach Publications.

JS Foundation, 2019a. *Introduction.* [Online]
Available at: http://appium.io/docs/en/about-appium/intro/
[Accessed 29 January 2020].

JS Foundation, 2019b. *Getting Started - Appium.* [Online]
Available at: http://appium.io/docs/en/about-appium/getting-started
[Accessed 18 February 2020].

JS Foundation, 2019c. *XCUITest Real Devices (iOS) - Appium.* [Online]
Available at: http://appium.io/docs/en/drivers/ios-xcuitest-real-devices
[Accessed 18 February 2020].

JS Foundation, 2019d. *UIAutomator 2 (Android) - Appium.* [Online]
Available at: http://appium.io/docs/en/drivers/android-uiautomator2/index.html
[Accessed 18 February 2020].

JS Foundation, 2019e. *Find Elements - Appium.* [Online]
Available at: http://appium.io/docs/en/commands/element/find-elements/
[Accessed 20 February 2020].

JS Foundation, 2020. *Desired Capabilities - Appium.* [Online]
Available at: http://appium.io/docs/en/writing-running-appium/caps/index.html
[Accessed 18 February 2020].

Kaner, C., Bach, J. & Pettichord, B., 2002. *Lessons learned in software testing: a context-driven approach.* New York: John Wiley & Sons.

Kasurinen, J., 2013. *Ohjelmistotestauksen Käsikirja.* Jyväskylä: Docendo.

Kleivane, T., 2011. *Unit Testing with TDD in JavaScript,* Trondheim: Norwegian Open Research Archives.

Kortelainen, H., Happonen, A. & J, H., 2019. From assert provider to knowledge company - transformation in the digital era. *Lecture Notes in Mechanical Engineering,* pp. 333-341.

Kumar, D. & Mishra, K., 2016. The Impacts of Test Automation on Software's Cost, Quality and Time to Market. *Procedia Computer Science,* Volume 79, pp. 8-15.

Kundu, D. & Samanta, D., 2016. Enumerating message paths for interaction testing of object-oriented systems. *Innovations in Systems and Software Engineering,* 12(4), pp. 279-301.

Kuparinen, E., 2019. *Managing application state and control flow using Redux and Redux-Saga in a web application,* Lappeenranta: LUTPub.

Mårtensson, T., Ståhl, D. & Bosch, J., 2019. Test activities in the continuous integration and delivery pipeline. *Journal of Software: Evolution and Process,* 31(4).

Martin, R., 2009. *Clean Code.* Upper Saddle River: Prentice Hall.

Mathur, S. & Malik, S., 2010. Advancements in the V-Model. *International Journal of Computer Applications,* 1(12), pp. 29-34.

Mesbah, A., van Deursen, A. & Roest, D., 2012. Invariant-Based Automatic Testing of Modern Web Applications. *IEEE Transactions on Software Engineering,* 38(1), pp. 35-53.

Meyer, M., 2014. Continuous Integration and Its Tools. *IEEE Software,* May, 31(3), pp. 14-16.

Microsoft, 2017. *Publish and consume build artifacts in builds - Azure Pipelines and TFS.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/devops/pipelines/artifacts/pipeline-artifacts
[Accessed 25 February 2020].

Microsoft, 2018. *Azure File Copy task - Azure Pipelines.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/devops/pipelines/tasks/deploy/azure-file-copy
[Accessed 25 February 2020].

Microsoft, 2019a. *What is Azure DevOps?.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/devops/user-guide/what-is-azure-devops
[Accessed 8 January 2020].

Microsoft, 2019b. *What features and services do I get with Azure DevOps?.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/devops/user-guide/services
[Accessed 8 January 2020].

Microsoft, 2019c. *Jobs in Azure Pipelines and TFS - Azure Pipelines.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/devops/pipelines/process/phases
[Accessed 21 February 2020].

Microsoft, 2019d. *Pipeline caching - Azure Pipelines.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/devops/pipelines/release/caching
[Accessed 21 February 2020].

Microsoft, 2019e. *Protect your Git branches with policies - Azure Repos.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/devops/repos/git/branch-policies
[Accessed 24 February 2020].

Microsoft, 2019f. *Xcode build and release task - Azure Pipelines.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/devops/pipelines/tasks/build/xcode
[Accessed 24 February 2020].

Microsoft, 2019g. *Build, tests, and deploy Xcode apps - Azure Pipelines.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/devops/pipelines/ecosystems/xcode
[Accessed 24 February 2020].

Microsoft, 2019h. *Publish Pipeline Artifacts task - Azure Pipelines.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/devops/pipelines/tasks/utility/publish-pipeline-artifact
[Accessed 25 February 2020].

Microsoft, 2019i. *Build, test and deploy Android apps - Azure Pipelines.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/devops/pipelines/ecosystems/android
[Accessed 25 February 2020].

Microsoft, 2020a. *YAML schema - Azure Pipelines.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/devops/pipelines/yaml-schema
[Accessed 21 February 2020].

Microsoft, 2020b. *Microsoft-hosted agents for Azure Pipelines - Azure Pipelines.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/hosted
[Accessed 21 February 2020].

Microsoft, 2020c. *Build pipeline triggers - Azure Pipelines.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/devops/pipelines/build/triggers
[Accessed 21 February 2020].

Microsoft, 2020d. *Templates - Azure Pipelines.* [Online]
Available at: https://docs.microsoft.com/en-us/azure/devops/pipelines/process/templates
[Accessed 24 February 2020].

Milanova, A., Rountev, A. & Ryder, B. G., 2004. Precise Call Graphs for C Programs with Function Pointers. *Automated Software Engineering,* , 11(1), pp. 7-26.

Minashkina, D. & Happonen, A., 2018. Operations automatization and digitalization - a research and innovation collaboration in physical warehousing, AS/RS and 3PL logistics context. *LUT Research Reports series report 86,* p. 66.

Minashkina, D. & Happonen, A., 2019. *Decarbonizing warehousing activites through digitalization and automatization with WMS integration for sustainability supporting operations.* Melbourne, E3S web of conferences, p. 8.

Mosley, D. & Posey, B., 2002. *Just enough software test automation.* s.l.:Prentice Hall Professional.

Nivanaho, T., 2019. *Developing a cross-platform mobile application with React Native,* Lappeenranta: LUTPub.

Occhino, T., 2015. *React Native: Bringing modern web techniques to mobile.* [Online]
Available at: https://engineering.fb.com/android/react-native-bringing-modern-web-techniques-to-mobile/
[Accessed 7 January 2020].

Olan, M., 2003. Unit testing: test early, test often. *Journal of Computing Sciences in Colleges,* 19(2), pp. 319-328.

OpenJS Foundation, 2020a. *Getting Started · WebdriverIO.* [Online]
Available at: https://webdriver.io/docs/gettingstarted.html
[Accessed 18 Feburary 2020].

OpenJS Foundation, 2020b. *Selectors · WebdriverIO.* [Online]
Available at: https://webdriver.io/docs/selectors.html
[Accessed 20 February 2020].

Peffers, K., Rothenberger, M., Tuunanen, T. & Vaezi, R., 2012. *Design Science Research Evaluation.* Las Vegas, Scopus (Elsevier B.V), pp. 398-410.

Polo, M., Reales, P., Mario, P. & Ebert, C., 2013. Test Automation. *IEEE Software,* 30(1), pp. 84-89.

Prakash, V., Senthil Anand, N. & Bhavani, R., 2012. Software test automation - the ground realities realized. *Journal of Theoretical and Applied Information Technology,* 43(2), pp. 306-312.

Ramler, R. & Wolfmaier, K., 2006. *Economic perspectives in test automation: balancing automated and manual testing with opportunity cost.* Shanghai, Association for Computing Machinery, pp. 85-91.

Redux, 2020. *Writing Tests.* [Online]
Available at: https://redux.js.org/recipes/writing-tests
[Accessed 3 February 2020].

Rossberg, J., 2019. *Agile Project Management with Azure DevOps: Concepts, Templates, and Metrics.* Berkeley: Apress.

Rößler, J., 2019. *Test Automation without Assertions,* Malmö: s.n.

Rusynyk, M., 2019. *Pixels matter or easy UI screenshot testing in React Native.* [Online]
Available at: https://medium.com/ing-blog/pixels-matter-or-easy-ui-screenshot-testing-in-react-native-6e41ba91b225
[Accessed 17 January 2020].

Sahu, M. & Mohapatra, D. P., 2015. *MM-path Approach for Integration Testing of Aspect-Oriented Programs.* Bhubaneswar, IEEE.

Shafique, M. & Labiche, Y., 2010. *A systematic review of model based testing tool support,* Ottawa: Department of Systems and Computer Engineering.

Shahrokni, A. & Feldt, R., 2013. A systematic review of software robustness. *Information and Software Technology,* 55(1), pp. 1-17.

Singh, Y., 2012. *Software Testing.* New York: Cambridge University Press.

Solheim, J. & Rowland, J., 1993. An Empirical Study of Testing and Integration Strategies Using Artificial Software Systems. *IEEE Transactions on Software Engineering,* pp. 941-949.

Ståhl, D. & Bosch, J., 2014. Modeling continuous integration practice differences in industry software development. *The Journal of Systems & Software,* Volume 87, pp. 48-59.

Teixeira, P., 2019. *nock: HTTP server mocking and expectations library for Node.js.* [Online]
Available at: https://github.com/nock/nock
[Accessed 13 February 2020].

Tian, J., 2005. *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement.* s.l.:John Wiley & Sons.

Twitch, 2017. *Investigating React Native.* [Online]
Available at: https://blog.twitch.tv/en/2017/04/25/investigating-react-native-6032ecced610/
[Accessed 7 January 2020].

Williamson, K. & Johanson, G., 2017. *Research Methods: Information, Systems, and Contexts.* 2nd ed. s.l.:Chandos Publishing.

Zagallo, T., 2015. *Bridging in React Native.* [Online]
Available at: https://tadeuzagallo.com/blog/react-native-bridge/
[Accessed 7 January 2020].

Zammetti, F., 2018. *Practical React Native: Built Two Full Projects and One Full Game using React Native.* Berkeley: Apress.

APPENDIX 1. Snapshot of a shallow render with children

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`Continue Learning shallow snapshot 1`] = `
<View>
  <Text
    style={
      Object {
        “color”: “#16181A”,
        “fontFamily”: “Roboto-Regular”,
        “fontSize”: 20,
        “fontWeight”: “bold”,
        “lineHeight”: 24,
        “marginLeft”: 16,
        “marginTop”: 16,
      }
    }
  />
  <LearningPathSection />
  <LessonSection />
  <AssignmentSection />
</View>
`;
```

APPENDIX 2. Snapshot of a shallow render with no children

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`Documents Item snapshot 1`] = `
<TouchableOpacity
  activeOpacity={0.2}
  onPress={[MockFunction]}
  style={
    Object {
      “alignItems”: “center”,
      “display”: “flex”,
      “flexDirection”: “row”,
    }
  }
>
  <test-file-stub />
  <View
    style={
      Object {
        “display”: “flex”,
        “flexDirection”: “column”,
        “marginLeft”: 12,
      }
    }
  >
    <Text
      style={
        Array [
          Object {
            “color”: “#16181A”,
            “fontFamily”: “Roboto-Regular”,
            “fontSize”: 14,
            “fontWeight”: “bold”,
            “lineHeight”: 16,
          },
          Object {
            “color”: “#55A631”,
          },
        ]
      }
    >
      name
    </Text>
    <Text
      style={
        Object {
          “color”: “#5E6266”,
          “fontFamily”: “Roboto-Regular”,
          “fontSize”: 12,
          “lineHeight”: 16,
        }
      }
    >
      size
    </Text>
  </View>
</TouchableOpacity>
`;
```

APPENDIX 3. Unit tests of an asynchronous Redux action

```
import thunk from 'redux-thunk';
import MockStore from 'redux-mock-store';
import { actions, getMyMobileUser } from './Users/usersActions';
import getClient from '../API/ValamisApi';

const mockStore = MockStore([thunk]);
jest.mock('../API/ValamisApi');
const testForResponse = (expected, mockValue) => {
  getClient.mockResolvedValue(mockValue);
  const store = mockStore();

  return store.dispatch(getMyMobileUser('')).then(() => {
    expect(store.getActions()).toStrictEqual(expected);
  });
};

describe('User action creators', () => {
  test('getMyMobileUser action success', () => {
    return testForResponse([{ type: actions.SET_USER, payload: 'value' }], {
      apis: {
        'Valamis mobile': {
          get_valamis_mobile_user: jest.fn(() => ({
            obj: 'value'
          }))
        }
      }
    });
  });

  test('getMyMobileUser action error', () => {
    const error = new Error('test');
    return testForResponse(
      [{ type: actions.USER_ERROR, payload: error }],
      new Promise((_resolve, reject) => reject(error))
    );
  });
});
```

APPENDIX 4. WebdriverIO Configuration file for emulators

```
exports.config = {
  runner: 'local',
  port: 4723,
  specs: ['./test/specs/**/*.js'],
  exclude: [],

  maxInstances: 10,
  capabilities: [
    // {
    //   maxInstances: 1,
    //   appiumVersion: '1.16.0',
    //   automationName: 'XCUITest',
    //   platformName: 'iOS',
    //   platformVersion: '13.3',
    //   deviceName: 'iPhone 11 Pro Max',
    //   app: `${process.cwd()}/ios/build/ValamisMobile/Build/Products/Debug-iphonesimulator/ValamisMobile.app`,
    //   launchTimeout: 15000
    // },
    {
      maxInstances: 1,
      appiumVersion: '1.16.0',
      automationName: 'UiAutomator2',
      platformName: 'Android',
      platformVersion: '10',
      deviceName: 'pixel_api29',
      app: `${process.cwd()}/android/app/build/outputs/apk/debug/app-debug.apk`
    }
  ],

  logLevel: 'warn',
  bail: 0,
  baseUrl: 'http://localhost',
  waitforTimeout: 10000,
  connectionRetryTimeout: 90000,
  connectionRetryCount: 3,
  afterTest(test) {
    if (!test.passed) {
      driver.saveScreenshot(
        `./test/screenshots/error_${test.title.replace(/ /g, '_')}.png`
      );
    }
    driver.reloadSession();
  },

  services: ['appium'],
  appium: {
    command: 'appium'
  },

  framework: 'jasmine',
  reporters: ['spec'],
  jasmineNodeOpts: {
    defaultTimeoutInterval: 180000
  }
};
```

APPENDIX 5. Pull request -triggered pipeline's configuration YAML

```yaml
trigger: none

variables:
  - group: AutomatedBuildVariables

pool:
  vmImage: 'macOS-latest'

stages:
  - stage: Build
    jobs:
      - job: android
        steps:
          - template: pipeline/cache-yarn.yml
          - bash: |
              cd android
              ./gradlew assembleDebug
            displayName: Gradle compile Android

      - job: ios
        steps:
          - bash: cd ios && rm -rf build
            displayName: Clean iOS folder

          - template: pipeline/cache-yarn.yml
          - template: pipeline/cache-cocapods.yml

          - task: InstallAppleCertificate@2
            inputs:
              certSecureFile: iosDistributionValamisMobile.p12
              certPwd: $(P12pass)
              keychain: 'temp'
              deleteCert: true
            displayName: Install Apple Certificate

          - task: InstallAppleProvisioningProfile@1
            inputs:
              provisioningProfileLocation: 'secureFiles'
              provProfileSecureFile: 'comarcusysvalamismobile_Appstore.mobileprovis
ion'
            displayName: Install Apple Provisioning Profile

          - task: Xcode@5
            inputs:
              actions: clean build
              sdk: iphonesimulator13.2
              configuration: Debug
              scheme: ValamisMobile
              xcWorkspacePath: 'ios/ValamisMobile.xcworkspace'
            displayName: Build iOS

  - stage: Test
    jobs:
      - job: test
        steps:
          - template: pipeline/cache-yarn.yml
          - bash: yarn test
            displayName: 'Run Jest'
```

APPENDIX 6. Scheduled pipeline’s configuration YAML

```yaml
trigger: none
pr: none
schedules:
  - cron: '0 6 * * *'
    displayName: Daily 6:00 (Finnish) build
    branches:
      include:
        - dev
    always: true

variables:
  - group: AutomatedBuildVariables

pool:
  vmImage: 'macOS-latest'

jobs:
  - job: android
    steps:
      - template: pipeline/cache-yarn.yml
      - bash: |
          cd android
          ./gradlew assembleDebug
          cd ..
        displayName: Gradle compile Android

      - bash: npm start &
        displayName: Start Metro Bundler

      - bash: yarn global add appium & appium &
        displayName: Install and start Appium

      - bash: echo y | $ANDROID_HOME/tools/bin/sdkmanager --install 'system-images;android-29;google_apis_playstore;x86'
        displayName: Install AVD files

      - bash: echo no | $ANDROID_HOME/tools/bin/avdmanager -s create avd -n pixel_api29 -k 'system-images;android-29;google_apis_playstore;x86'
        displayName: Create AVD

      - bash: nohup $ANDROID_HOME/emulator/emulator -avd pixel_api29 -no-snapshot -skin 400x700 > /dev/null 2>&1 & $ANDROID_HOME/platform-tools/adb wait-for-device shell 'while [[ -z $(getprop sys.boot_completed | tr -d '\r') ]]; do sleep 1; done; input keyevent 82'
        displayName: Start AVD

      - bash: ./node_modules/.bin/wdio wdio.emulator.conf.js
        displayName: Run WDIO tests
        continueOnError: true

      - task: CopyFiles@2
        inputs:
          SourceFolder: 'test/screenshots'
          Contents: '*.png'
          TargetFolder: '$(Build.ArtifactStagingDirectory)'

      - task: PublishBuildArtifacts@1
        inputs:
          PathtoPublish: '$(Build.ArtifactStagingDirectory)'
          ArtifactName: 'drop'
          publishLocation: 'Container'
```