Huy Trinh

A Practical Approach to JavaScript Testing

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

16 November 2020

Author Title Number of Pages Date	Huy Trinh A practical approach to JavaScript testing 48 pages 16 November 2020
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Mobile Solutions
Instructors	Kari Aaltonen, Principal Lecturer

The purpose of this final year project is to analyze some most common types of testing in JavaScript application, namely Acceptance, System, Integration, and Unit testing. The use-cases and evaluation of each testing level in terms of benefits and costs are also addressed. In addition, different JavaScript testing frameworks are studied with great attention since it has an important effect on application testing.

The use-case of this thesis is a React web application that is implemented in nearly all types of testing described in the thesis. These tests contributed greatly to the project since they allow developers to implement new features without breaking another. Furthermore, the project setup for testing and important test cases are also made available as an open-source for study purpose which can be found at https://github.com/HuyAms/testify

In conclusion, this thesis demonstrates various types of JavaScript application testing and encourages developers to implement them on the software system to confidently deliver professional code and ensure the application is bug-free.

Keywords	JavaScript Testing, Testing Frameworks, Jest, Mocha, Jasmine, React

Contents

List of Abbreviations

API	Application Programming Interface
BDD	Behavior Driven Development
CD	Continuous Development
CI	Continuous Integration
DOM	Document Object Model
IDE	Integrated development environment
MVP	Minimum Viable Product
SDLC	Software Development Lifecycle
TDD	Test Driven Development
TMM	Test maturity model
TPI	Test process improvement

1 Introduction

The number of websites has increased significantly since the last decades. There were more than 1.94 billion websites on the world wide web in January 2019 (Lindsay 2019). In addition, numerous desktop applications have been shifted to web applications to cut down development, maintenance and distribution costs which are very high, especially when it comes to multiplatform applications. This trend leads to an increase in the number of web applications over the last few years (Paul 2002.) See figure 1 below.

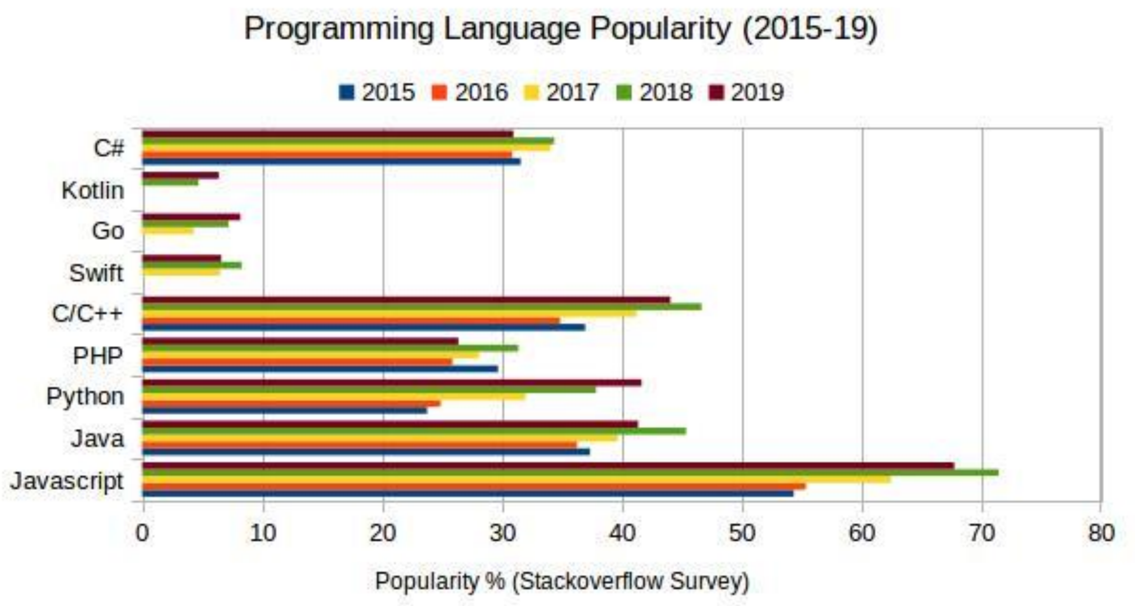


Figure 1. Trend of programming language over the past 4 years (Coding Infinite, 2019)

JavaScript is the most important language used for building a website in today's Web 2.0 world. As illustrated in figure 1, it can be seen that the total usage of JavaScript increased dramatically for the past 5 years, from 2015 to 2019 (Coding Infinite, 2019.) JavaScript is by far the most popular programming language and this is because of the growth of JavaScript frameworks recently such as NodeJS. However, there is a lack of testing in various real-world JavaScript projects which makes a project unreliable to use and difficult to scale. Thus, this thesis focuses on addressing the problem by providing the basic knowledge of JavaScript testing.

Building a website with JavaScript remains a challenging task, especially when the website is scaled frequently. There are various factors that make website development a demanding task. The biggest challenge of web development is to support various browsers with different versions. JavaScript code that runs correctly on Safari may break on Internet Explorer (IE), Google Chrome or Firefox browsers (Hazem Saleh, 2013.) Secondly, developers normally have to produce a low-cost software in a short time while ensuring the software quality. This is because software quality has a huge impact on many factors of our daily lives from personal security, health, safety to national economy (Kshirasagar et al, 2018.)

This thesis aims to demonstrate some common levels of JavaScript testing such as Acceptance testing System testing, Integration testing, and Unit Testing. Moreover, these testing levels are analyzed in terms of its costs and benefit. Three most popular testing frameworks are also studied in this thesis.

The case study of this thesis is a web application built with ReactJS used by a company providing a sustainable heat exchanger solution. The main objective of the project is to design and implement a product configuration and quoting tool for sales and engineers which enables error-free quoting with a minimal amount of manual steps. The main focus of the developer is to build a Minimum Viable Product (MVP) web application that connects to the existing configuration tool. It is worth noting that this application is built based on the company's specific needs, therefore, there is no existing web-app solution like this before.

This configuration tool requires a high level of accuracy for building a physical product, therefore, the application must work correctly as expected. As a result, testing plays an important role in this project as it helps the company to manufacture the product with great confidence. The ultimate goal is to apply the theory to this real-world project, to investigate and understand which important features of the application should be covered by tests since it would be expensive and unrealistic to test all the features. In addition, setting up testing in a JavaScript web-application will also be studied in detail.

2 Theoretical background

This chapter provides the definition of software quality and also emphasizes the importance of testing in software development. Four types of testing including Unit, Integreation, System and Acceptance testing are also addressed in depth.

2.1 Software quality

There are various answers to “What is software quality?” since it is a complex concept which depends highly on the context. According to David Garvin who published the article “What Does Product Quality Really Mean?” in 1984, software quality is defined into four major domains. These domains are economics which focuses on making profits, philosophy focusing on definitions, marketing focusing on customer satisfaction, and management which focuses on the specification and the process of creating a product (Cortney, 2014.) There are Transcendental, User, Manufacturing, Product, and Valued-Based perspective which are described as follows:

- **Transcendental View**: This defines quality as something which can be recognized but really hard to define. In this view, the quality of the product is judged based on subjective opinion and the ability to determine that can only be developed from experience.

- **User View:** According to this approach, user satisfaction is used to measure the quality of a product or service. The essential question to ask when using this approach to evaluate the product quality is “Does the product satisfy user expectations or preference?”

- **Product View**: This is also known as the product-based approach in which the quality is understood as the inner qualities of a product. To give a better understanding of this perspective, Garvin defined eight specs to measure a product quality including conformance, durability, aesthetics, performance, features, reliability, serviceability, and perceived quality. It is worth noting that the dimensions depend on the product inherent characteristics.

- **Manufacturing View:** Under this perspective, the conformance to the product specification decides the quality of a product rather than personal preferences. Thus, any differences from the pre-defined specification of a product reduce the product quality.

- **Value-based View:** Quality, from this view, depends on how much it costs to manufacture a product and how much a user is willing to pay for this product. The quality of a product is determined in terms of its costs and benefits. Therefore, a product with high quality is not the one performing the best (Paul et al, 2014.)

The concept of software quality in terms of measurable scores was first well studied in the mid-1970s. There are two types of quality known as quality criteria and a quality factor which were first well studied by McCall, Richards, and Walters. Quality criteria represent quality factors that are related to software development. Modularity could be seen as a great example. Modularity is a part of software architecture which allows developers to put various small components into one module system. Therefore, it helps improve the maintainability of the software product. On the other hand, a quality factor refers more to system behavior. There are various factors which can be considered to be under this type of quality, for example, reliability, efficiency, testability, and reusability (Kshirasagar et al, 2018.) To give a better understanding of the quality factor, an example of reliability could be addressed. In general, a software system with high reliability must do what they are supposed to do and it must perform the tasks without any problems. In other words, the system must do the right things and do the things right (Jeff Tian, 2005.)

There is a wide range of testing models in the software industry. It is worth mentioning the two well-known models, namely the test process improvement (TPI) model and the test maturity model (TMM). Developers can use these models to measure the current state of the software, find what is the next logical place for improvement and define an action plan for software testing (Kshirasagar et al, 2018.)

2.2 Role of testing

Testing appears in various projects of every company since it plays an important role in software development. Software testing has become an important part of programming because it enables developers to find errors at the very first beginning of a project.

The product should be validated and demonstrated in a laboratory experiment which is similar to a real-world environment before it is released to the public users. In this way, one can ensure that the product or system would work well when the customers use them. In software products, code is executed in such a controlled environment which is broadly known as testing. In software development, tests are normally conducted in a development stage. Code implemented are run through test cases to demonstrate that a service or product works as it should. Thus, the first purpose of testing is to show the proof of quality of a product. On the other hands, in software world, it is quite straightforward to find and fix error compared to physical products which is not flexible to manufacture. Therefore, it is more beneficial if a software product is tested in the development phrase then it can be shipped to end-users as bug-free product. In conclusions, testing has two primary goals including to demonstrate product proper behavior and secondly to detect and fix problems and errors as soon as possible (Jeff, 2005.)

Finally, it is notable that the stakeholders of testing are the programmers, the project managers, and the customers. Each stakeholder would benefit differently from the test process (Kshirasagar et al, 2018.)

For example, the project managers can benefit from cost-effectiveness. A small bug in the program can destroy the whole system and it is not because that the developers are not skillful or careless but it could due to the complexity of the system. If the bug is not detected in the earlier stage of the project, it would be hard to trace back and find it. Thus, it would become expensive to fix the system. Not to mention that in the progress of fixing an error, it is likely that a developer can produce more bugs. That why it is crucial to detect any defects in the beginning stage of the project and testing can help us with that.

On the other hand, users can also benefit from testing. From business perspective, the ultimate purpose of a product is to provide customers with the best satisfaction. One of the reason why testing is crucial to provide the best user experience (Rajkuma, 2019.) From developer point of view, tests are written so developers can be confident that the software will work correctly when the user uses them (Kent, 2019).

2.3 Software development and testing lifecycle

This chapter examines the approaches of software testing and when during the software development cycle testing is typically executed. Moreover, the evolution of software development is also addressed since it affects the testing lifecycle.

2.3.1 Waterfall methodology

Waterfall is known as the classic Software Development Lifecycle (SDLC) which contains of four stages, namely requirements, design, implementation, testing and maintenance. Under waterfall methodology, there is a set of phases and the next phase cannot be started until the previous phase has been completed. There are typically 5 main phases in Waterfall software lifecycle (Brooklin, 2020).

- **Requirements**: At the beginning of a project, problems, and requirements for the system must be clarified. It is supposed that all requirements can be collected at this phrase from customers so that the system can be developed without customers' concerns.

- **Design**: This phase focuses on studying to understand the existing systems thoroughly. Possible solutions are brainstormed and the data model is designed to be used in the system.

- **Implementation**: The implementation phase is where developers study all the requirements from these previous phrases and start to produce the actual code to build the system.

- **Testing and verification**: This phrase is where the QA team works along with developers to test the product and make sure it works as expected. Customers then review the product to see if it meets all the planned requirements.

- **Maintenance**: If a bug is found in the system, it will be fixed during this stage. In addition, the system is also reviewed and enhanced to be a better version.

STLC standing for Software Testing Life Cycle is a series of phases that are carried out by the QA team to verify the quality of software testing. In the other words, STLC is an integration part of Software Development Life Cycle (SDLC) but it corresponds only with the testing phrases, as demonstrated in the figure below (Arnab, 2019.)

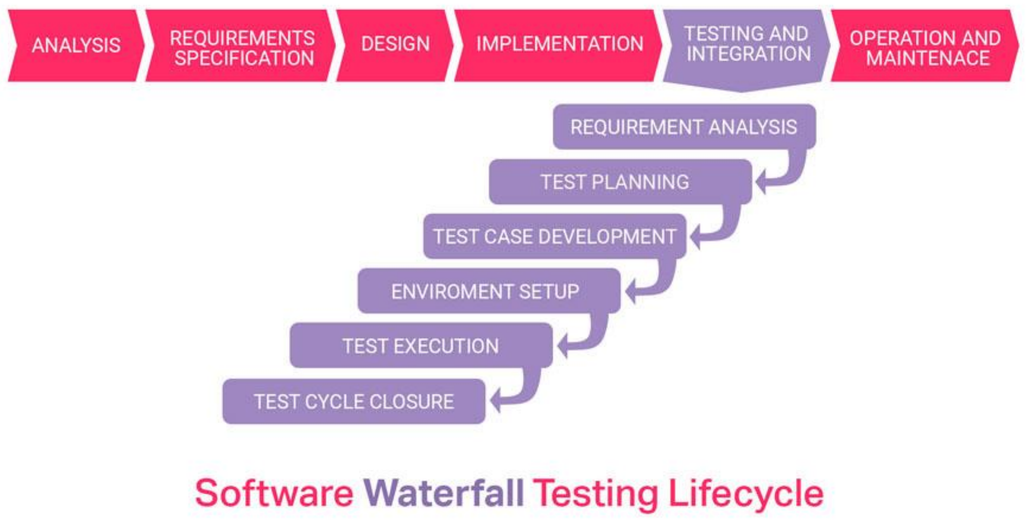


Figure 2. Software Waterfall Testing Lifecycle (Yana, 2020)

Figure 2 illustrates where the testing phrase fits in the Waterfall methodology. It can be seen that testing will be conducted right after the implementation phase is finished. The testing phase consists of 6 sub-steps:

- **Requirement analysis**: At this stage, the team works along with the project stakeholders analyzing the key requirements of the system. Project stakeholders and technical managers discussed together both non-functional including, for example, security and usability and functional purposes of the system.

- **Testing Planning**: This is the most important phase of the testing cycle. Cost is estimated, a strategy of the testing, testing tool, resource planning will be addressed during the testing planning. The outcome of the planning session are a test plan and test effort estimation.

- **Test Case Development:** A set of detailed test cases will be written by testers at this stage. In addition, existing test cases are fixed and updated if there is a new requirement from customers.

- **Setting up a test environment:** The test environment is the condition in which test cases are executed. A staging environment will be set up to test the system in various conditions. The responsibility of the QA team at this stage is just to conduct smoke tests and check if the test environment which is provided by the developers satisfies the requirements.

- **Test Execution:** In this phase, actual test cases are executed as the plan. Testing results are documented in a careful manner. Bugs and errors are also reported for correction and further checks. The testing result should be documented with a careful manner.

- **Closing the tests cycle:** When the testing is done, the result will be examined. Test coverage, time, costs, business objectives are evaluated. The team works together to see where to improve the system and which best practices can be applied to the next project.

Despite the fact that the Waterfall model is quite straightforward, it is not always flexible to apply to the modern IT projects which change along with the market growth and the development process (Yana, 2020.)

2.3.2 Agile methodology

The main difference between waterfall and agile methodology is that agile focuses more on adaptability and customer involvement while waterfall values planning ahead. There are various models of agile development which sharing the same values including Scrum and Kaban. Instead of creating long timeline with planning ahead, Agile breaks the project into small repeated phases called Sprint. Each Sprint typically lasts for one or a few weeks and after it is completed, the result and feedback of the previous sprint is used to plan the next one (Brooklin, 2020.) Under agile methodology, the development of software goes along with customer's requirements, therefore, the testing in agile method is continuous as well. Developers should work closely with QA team to integrate testing during a sprint. See figure 3 below.

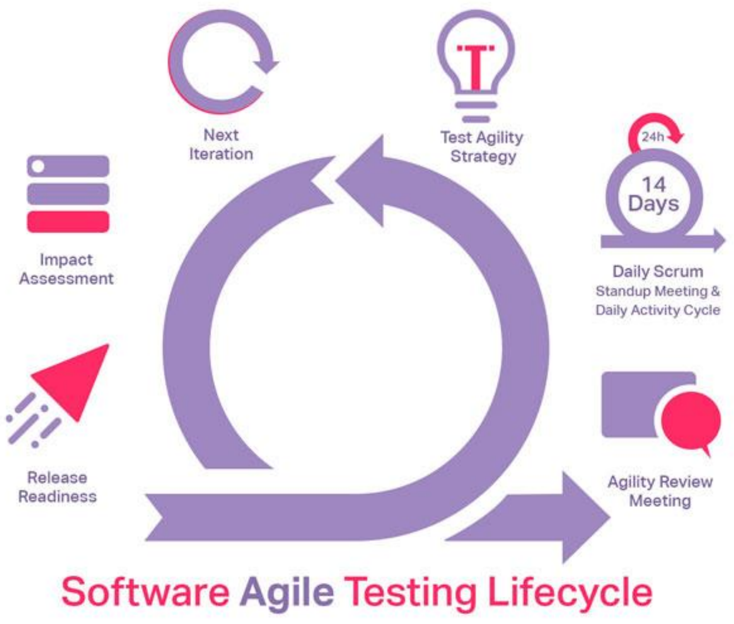


Figure 3. Software Agile Testing Lifecycle (Yana, 2020)

It can be seen in Figure 3 that there are 5 main activities happening in the agile testing.

- **Test Strategy**: The planning session takes place during this stage. The developers, QA team, and stakeholders hold a meeting to define the way of testing including schedule, frequency of testing, and testing strategy.

- **Daily Scrums**: This event happens every working day where the QA team can catch up with the status of developers work to know what has been done and what needs to be tested next.

- **Review meeting**: Typically, there is a meeting after each sprint to evaluate the current status of a project. Project progress will be verified by the QA team and a plan for the next sprint will be discussed.

- **Release readiness**: When the project is about to release. The teamwork together to test the system then decide if it is ready to release to production. If a bug is found, news-print will be needed for fixing all the issues.

- **Impact assessment**: This stage is where the product owner, customers, stakeholders can provide feedback for the team to improve on for the next development cycle (Yana, 2020.)

The agile methodology typically requires fast feedback on code changes, therefore, continuous integration (CI) usually plays an important role. Continuous integration is a practice of merging the new code into a common branch of a repository and testing new features as soon as possible. In an ideal world, developers should integrate their code into the main branch daily or multiple times a day. Without CI, a development team has to manually test a new feature which may take a couple of hours to get the feedback. And at the same time, new code keeps coming which makes the debug process difficult.

There are 2 best practices for integrating the code in a project. Firstly, the project should be rebuilt as soon as a new code is merging. This gives developers instant feedback if the new code breaks the system. The second practice is testing automation which means that new code should be automatically tested before it gets merged into a common branch. CI is exactly the tool for enabling these 2 features. When the new change is pushed, CI rebuilds the code to ensure that the code is compiled correctly. It then automatically runs all the test cases to make sure that the system works as expected. There are various popular CI tools such as Buddy, Jenkins, CircleCI and TravisCI.

2.3.3 Traditional vs Agile Testing

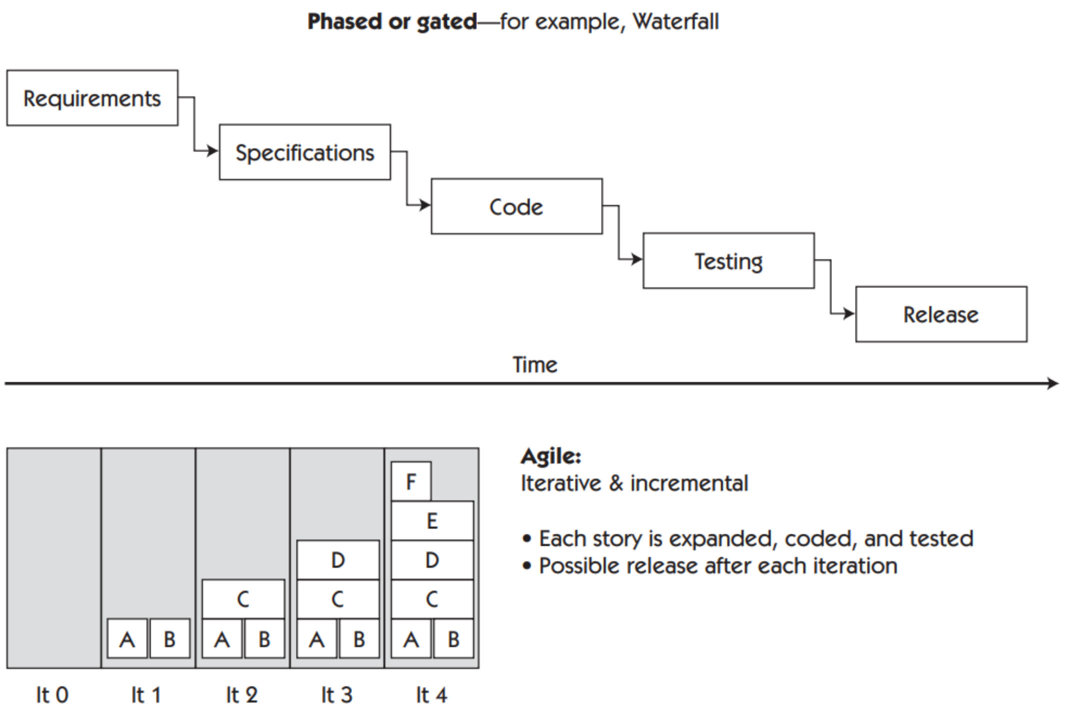

Phased or gated—for example, Waterfall
Requirements
Specifications
Code
Testing
Release
Time
F
E
D
C
A
B
It 0
It 1
It 2
It 3
It 4
Agile:
Iterative & incremental
• Each story is expanded, coded, and tested
• Possible release after each iteration

Figure 4. Waterfall vs Agile testing (Yana, 2020)

Figure 4 demonstrates that in Waterfall approach, testing is only executed at the end before the release phase. This gives the impression that testing phase can take as long as coding phase, however, in a real-world project, coding usually takes longer than the given time. As a result, there is not enough time for testing or the time for project will be much longer than expected.

On the other hand, Agile is incremental and interactive. The QA team can perform the testing as soon as a new feature is released. The team tests the code bit by bit to make sure it works as expected then moves to the next feature. In this way, developers have to work along with testers because the feature is not considered to be done if it has not passed the tests (Lisa and Janet, 2009.)

There are some other good features of agile testing comparing to waterfall model. First feature is quality-oriented, agile avoids ahead planning and focuses on short timeframe. Therefore, QA team has an opportunity to get to know the system once the project starts. In this way, bugs will be found and fixed earlier. The second feature is Test-driven as testing is integrated into the system every sprint which only lasts few months. In addition, acceptance tests are typically conducted by the customers Finally, customer-friendly is a feature to be considered. Agile approach requires customers involvement which results in great opportunities for feedback (Yana, 2020.)

In conclusion, the most important is that Agile provides quick update from testing which pushes the project forward. There are some teams who resist to apply Agile testing for fear of chaos and lack of discipline, however, true agile is all about the efficiency as well as the quality of the product (Lisa and Janet, 2009.)

3 Testing levels

There is a wide range of testing levels in software development, however, this chapter focuses mainly on syntax validation and four levels of testing, namely: unit testing, integration testing, system testing and acceptance testing. Tests are grouped together based on the level of detailing it is implemented or on the stage where they are added in SDLC, as depicted below in Figure 5 (Ulf, 2014).

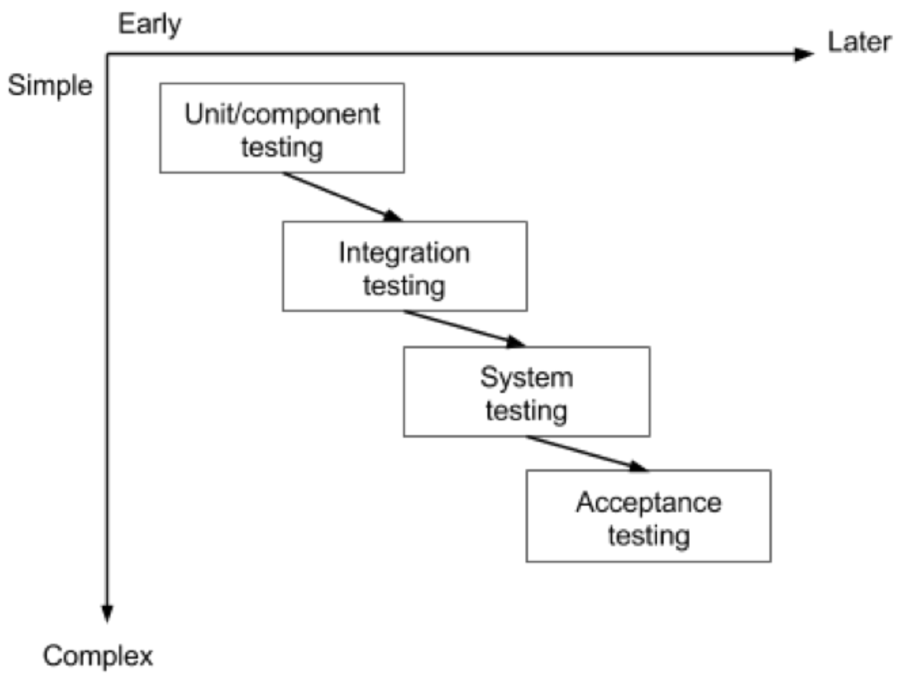


Figure 5. Testing sequence (Ulf, 2014)

Figure 5 illustrates how a software system generally has to go through four stages of testing before it can be deployed. Different stakeholders in the development organization are responsible for performing the first three levels of testing while acceptance level testing is executed by the customers (Kshirasagar, 2008). These four types of testing should be performed in a logical sequence to reduce the risk of bugs before the software can be launched. The importance of each testing phrase should be studied well by any testing team. It is recommended that testers should perform the testing on the simplest components of the system, then continue with the more complex ones. In this way, testers can ensure that they already exam the software thoroughly in the most efficient approach.

The four levels of testing should be seen as a sequence of testing which is implemented during the whole lifecycle of software development rather than as a hierarchy from simple to complex. It is worth mentioning from Figure 5 that later does not mean the acceptance testing should only be done after the software is released. In a more agile approach, such as Scrum, acceptance testing could be performed during every sprint demo which normally lasts from 2 to 3 weeks (Ulf, 2014.)

Cost and speed should also be considered while writing tests. The tests become more costly when we move up to the next testing level. This is because of the actual money which is spent on setting up a CI environment, but also the time it takes developers to maintain and write the test cases. The higher the level of the test, the more chances of failure there are. Thus, it is more possibility that the test will be broken which results in more time taken to analyze and fix the tests. It is also worth mentioning that the higher level of the test, the slower it will run. This is due to the fact that the test in the higher-level runs against more code than the one in the lower level.

This does not mean developers should only focus on the unit testing and ignore other types of test because the more the tests resemble the way software is used, the more benefit we can gain from the tests. Trade-off between benefits and the cost has to be made. As moving up the high-level types of testing, developers can gain much more confident that the system would work as expect but the tests would be really slow and expensive (Kent, 2020.)

3.1 Code Validation

Syntax validation is also known as static testing, however, there is a small difference between validating and testing. Testing does not guarantee valid code but validation often results in better testing. An example could be a web application works well in different browsers because there are a set of tests to make sure that it works as expected. However, the code is broken in terms of formatting or coding style including indentation, well commented, proper spaced. It also does not mean that valid code results in working system, however, valid code often leads to an application which has less errors. In short, code validation is making sure that the syntax and style is up to standard and testing means that the app functionally works as expected.

There are three scenarios to be considered when it comes to code validation. The first situation is code is valid but does not work correctly. This is because of syntax and style validation cannot verify the logical function. Second scenario is that the code is invalid but correct and finally the code can be both invalid and functionally wrong.

There are several benefits of code validation. Firstly, testing would be much simpler if code is valid. This is because legible code makes it much easier to debug because it follows standard styling convention. Validation helps developers follow good practices which includes proper enclosing of tags, good indentation to improve readability (Liang, 2010.)

Practically, static testing or code validation is the cheapest and fastest actions to implement but it is unable to provide developers with the confidence in business logic. However, if one try to use unit testing to validate the types of a variable in dynamic languages such as JavaScript, it could be much better to use static testing with type checking tool such as TypeScript (Kent, 2020.)

3.2 Unit testing

Unit testing divides code into a small, isolated chunks for verifying that it works as expected. A unit is usually seen as a function or a method which acts independently from other units. Therefore, the concept of unit testing is not to test the whole system but to test one function at a time to make sure it works correctly then move on to the next function. (Liang, 2010.)

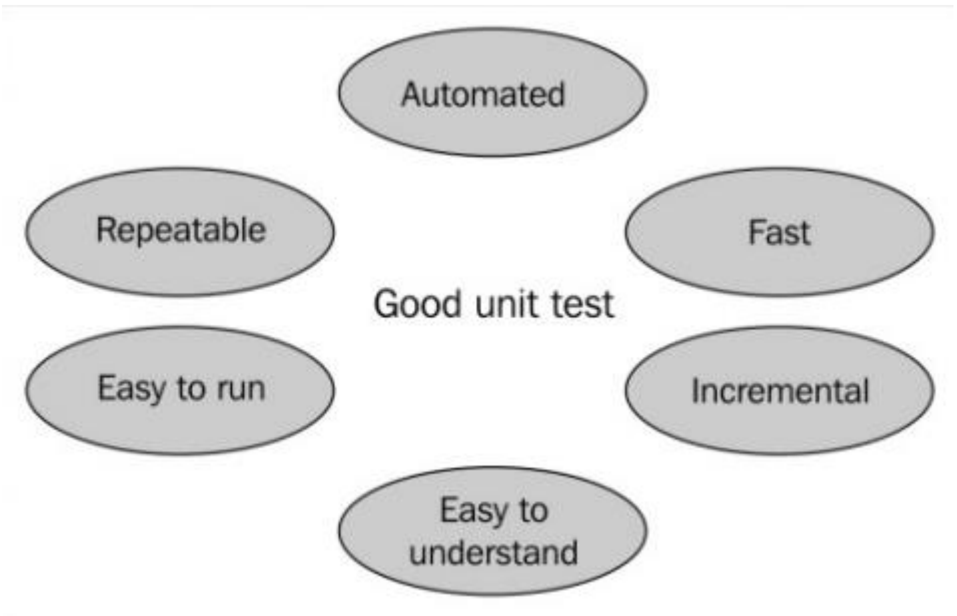


Figure 6. Testing sequence (Ulf, 2014)

As shown in figure 6, there are six elements that make up a good unit test. First, a good unit test should be straightforward to understand which means that existing test cases should be updated easily. Also, more new test cases can be added with ease by peer developers. Secondly, a great set of unit tests should be repeatable and automated. This can be done with the help of CI tools which run all the test cases automatically when the code is changed. Thirdly, the unit test should be incremental which means that it should be updated as soon as a bug is found or a new feature is added in the relevant code. In this way, developers can catch a system defect as soon as possible which helps them more confident while developing the system. Finally, a good test case should be fast to execute. In practice, this boost team's productivity since the unit test should be executed by just running a command or by clicking a button.Integration testing (Ulf, 2014.)

Unit testing is normally the fastest and cheapest type of test to execute because it has no or just little dependencies which are quite easy to be mocked (Kent, 2020.)

3.3 Integration testing

In unit testing, all individual blocks of code will be tested to verify they work correctly. However, the primary goal of integration testing is to make sure that different units of code can also work together. Integration testing is the concept of testing the interaction of all components in a program. As a result, the functional, performance, and requirements of a system can be verified by performing an integration test (Liang, 2010.) Integration testing is quite essential since blocks of code may work individually but not together. Problems which might happen in the interfaces between modules can be caught by integration testing.

There are four main approaches to an integration test which are described as follow:

- **Top-down approach**: In this approach, we start to test from the highest-level modules. In the front-end application, the high-level modules are considered to be closer to the UI level. This approach is used when the lower-level modules are not yet integrated.

- **Bottom-up approach**: In bottom-up testing, each module at the lower level is tested first, followed by higher modules. When the high-level module of the system is not ready, this approach will be used.

- **Critical Part First approach**: When there is a limitation of time, it would not be possible to test the whole application, this approach is used. In this approach, the most important module will be tested first to make sure all the functionality of the system works as expected.

- **Big-bang approach**: This is the simplest approach since all the modules would be combined then verified at once. However, it is quite hard to debug and localize the error when a bug is found. Big bang approach is recommended to be used only when the system is small (Prasad, 2016.)

Integration tests provide a great balance on the trade-offs between benefits and speed/expense. Therefore, it is recommended to spend most of the time on this type of testing (Kent, 2019.)

3.4 System testing

System testing verifies the whole system typically from the customer’s point of view against its functional and non-functional requirements (Prasad, 2016.) The primary goal is that all the systems should work as a whole when real users use it in a production environment. Due to the user's perspective, testers are supposed to test the whole system at high levels of abstraction as illustrated in figure 7 below. Thus, the system will be tested with a black box method which focuses on testing functionalities of a system against its specification. Typically, all system's functions that are exposed directly to the user will be tested first whereas implementation details of components which are already covered by Unit or Integration Testing should be ignored. System testing is generally performed by testers who have extensive knowledge about the system from the user's perspective and the use-case of those features is much more important than the implementation details (Kshirasagar, 2008.)

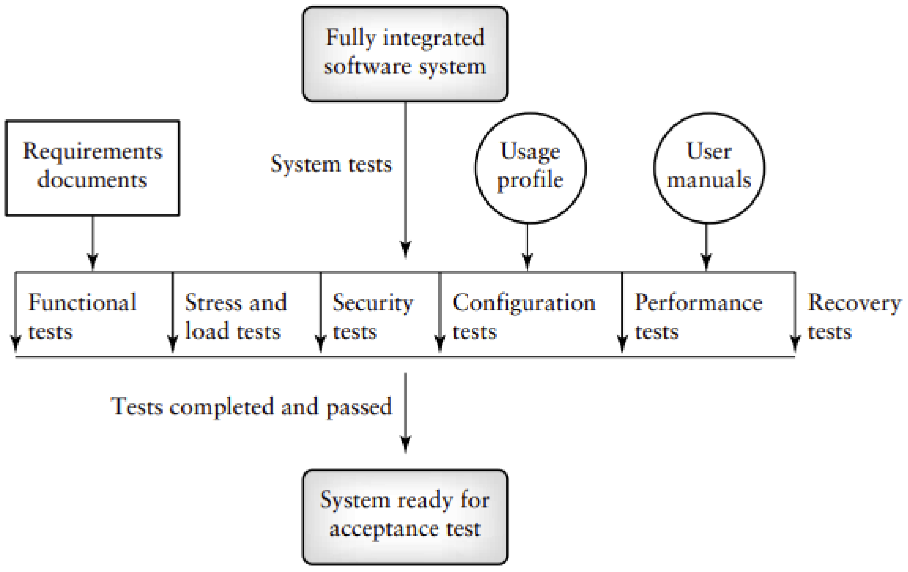


Figure 7. System testing components (Ilene, 2003)

It can be seen in Figure 7 that system testing consists of 6 types of test which are described as follow.

- **Functional testing**: This tests the behavior of the system to see if it follows the requirements specification which is typically described by the users.

- **Performance testing**: The purpose of performance testing is to see if the software meets the performance requirements which include, for example, memory use, delays, and response time.

- **Stress testing**: Under this testing, testers will try to put the system under the circumstance that put the resource of the system to maximum amounts. The ultimate goal of this test is to find the case when they system could be broken.

- **Configuration testing**: If the system works with hardware devices, configuration testing can be carried out. This testing evaluates the availability and performance of the system with different hardware configurations.

- **Security testing**: Security testing is performed to discover the vulnerabilities, risks in a software application and prevent the system from being attacked.

- **Recovery testing**: Under this method, the ability of a system to recover from crashes will be tested.

Both functional behavior and quality requirements including performance, reliability, usability, security, and performance are verified by system testing. Therefore, system testing takes a huge amount of resources (Ilene, 2003.)

3.5 Acceptance testing

Acceptance testing is conducted by real users to confirm that the system works as expected by the customers. This is the final testing before the system is released to production. Under this test, the users have to follow specific procedures to use the system in a staging environment. The test should continue even when the bug is found unless it crashed the system (William, 2005.) After the testing, customers will point out which features do not work as expected and which feature should be added, removed, or modified. If everything is satisfied by the customer, the system will be installed in a production environment (Ilene, 2003.)

There are two types of acceptance testing known as alpha testing and beta testing. Alpha testing is the testing where customers use the system in a controlled environment such as a developer or staging site. On the other hand, beta testing is done in a production environment (Prasad, 2016.)

4 JavaScript testing framework

JavaScript is a language that is most often used for the web browser and it is one of the most important tools for developing interactive components on the web. Unlike server-side language such as Python or PHP, there is no crash if JavaScript fails although the web browsers display some error messages. This feature of JavaScript creates developers some difficult time to debug the application (Liang, 2010.) This chapter introduces 3 most popular frameworks used for JavaScript testing, namely Jest, Mocha, and Jasmine.

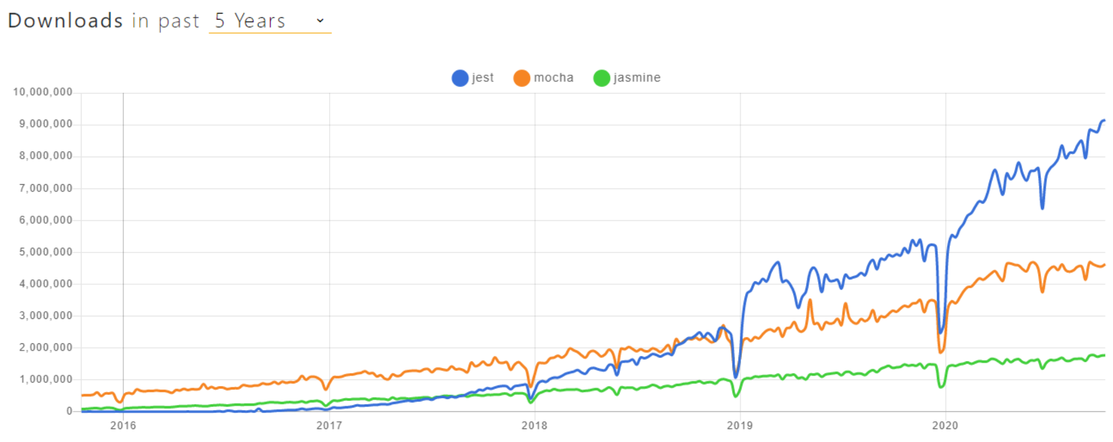


Figure 8. Trend of JavaScript testing frameworks over the past 5 years (NPM trends, 2020)

As illustrated in figure 8, the total downloads of 3 most popular testing frameworks for JavaScript have increased dramatically for the past 5 years, from 2016 to 2020. Jest is by far the most used framework over the past 2 years. The usage of Jest rose from about 1 million downloads in 2019 to 9 million downloads in 2020. On the other hands, Jasmine is the least popular framework which has only 1.7 million usages in 2020.

Overall, Mocha and Jasmine are more suitable for backend testing since they were originally built for Node applications. Thus, they provide more back-end features and documentation than Jest. On the other hands, Jest is stronger for the front-end side because it is created to use with React.

4.1 Zero-config testing framework Jest

Jest is an open-source JavaScript Testing Framework maintained by Facebook Inc which focuses on simplicity. It ensure correctness of any JavaScript projects and allows developers to write test with a simple, familiar API which contains rich features.

There are 4 key features of Jest that make it the most popular JavaScript framework. Firstly, Jest is a zero-config framework which works out of the box. All the work that developers need to do is to install Jest then it is done. Secondly, snapshots testing is promoted by Jest since it can easily keep track of large objects. Snapshots objects can live along within the test files. Snapshot is a great tool for UI testing which takes a screenshot of a UI component and compare it to a reference image stored withing the test. Thirdly, Jest is a fast testing framework which runs all the tests parallelly in their own processes. In addition, Jest executes failed tests first and organizes tests by the running time. In this way, Jest can maximize the performance since all the processes are isolated. Finally, Jest exposes a good Application Programming Interface (API) with well documented instruction (jestjs.io, 2020)

Jest which has nearly zero dependency is the most suitable framework for testing frontend, especially for web applications built with React. Jest is also compatible with other testing libraries built React such as React Testing Library or Ezyme (Michael, 2020.)

```
function sum(a, b) {
  return a + b;
}

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

Listing 1. Test case in Jest (Jest Authors, 2020)

Listing 1 example describes how to write a unit testing for a function that adds two numbers with Jest. It can be seen that Jest comes with a set of assertion functions out of the box.

4.2 Fully-customizable testing framework Mocha

Mocha is a JavaScript framework with rich features which runs on NodeJS and also in the browser environment. Unlike Jest, Mocha does not work out of the box but Mocha is one of the most flexible JavaScript testing framework since it only provides developers with a base framework and allows them to choose which mocking, spy and assertion they want to use. Additional configuration and setup is required in Mocha in exchange for a complete control of the testing framework.

In addition, Mocha requires the most dependencies to enable assertion and mocking feature. Comparing to Jest, Mocha is just a test runner without a built-int mocking and assertion library. Chai is often used as assertion library for Mocha since it is one of the most popular open-source assertion libraries. Chai is a BDD/TDD assertion library which has lots of extensions and plugins. Moreover, Sinon which is a popular mocking and spy library is also recommended to integrate with Mocha (Katia, 2018.)

Due to the fact that, Mocha is built for NodeJS, it is naturally more See listing 2 below suitable for a back-end project. Mocha with its flexible configuration and options for external library can be beneficial more for the large back-end project (Michael, 2020.)

```
var assert = require('assert');
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present', function() {
      assert.equal([1, 2, 3].indexOf(4), -1);
    });
  });
});
```

Listing 2. Test case in Mocha (Mocha Authors, 2020)

Listing 2 illustrates how a unit test written with Mocha is quite similar with the Jest version but Mocha uses Node.Js'built-in assert module since it does not have any built-in assertion library. Notably, any assertion library can be used with Mocha.

4.3 Behaviour-driven development testing framework Jasmine

Jasmine is an open-source JavaScript testing framework which was created in around 2018. Jasmine is pretty similar to Jest regarding to the out of the box feature which means that it attempts to include everything a developer needs for testing. In other word, Jasmine has no external dependencies (David, 2019)

According to the documentation, Jasmine is built as behaviour-driven development testing framework. Assertion and mocking feature is included already in Jasmine which makes it easy to setup. Jasmine is quite popular in Angular world because it is supported by Karma which is a test runner built by AngularJS team to make TDD easier in Angular development (Charith, 2020.) In addition, Angular team uses and recommends Jasmine with Karma for component testing while setting up Jasmine in React is not straightforward (Michael, 2020.)

However, there are a couple of reasons why Jasmine is the least popular one out of the 3 frameworks recently. The first reason is that Jest is built on top of Jasmine and it has all the features of Jasmine plus Karma. Moreover, Jest run twice of triple faster than Karma testing which makes a huge difference when integrating testing using CI/CD tools. Secondly, Jest provides snapshots testing which is not supported out of the box by Jasmine. Thirdly, it takes time to config Jasmine to work with CI tool such as TravisCI since a browser environment needs to be setup while Jest does not require a real browser to run. Finally, Jasmine is not straightforward as Jest when it comes to setup and it does not give developers full control of testing like Mocha either (Charith, 2020.)

```
describe("A suite is just a function", function() {
  var a;

  it("and so is a spec", function() {
    a = true;

    expect(a).toBe(true);
  });
});
```

Listing 3. Test case in Jasmine (Jasmine Authors, 2020)

5 Case study

5.1 Project summary

The case study of this thesis is a web application used to config the heat exchanger product. The company manufactured Shell & Plate exchanger more than 30 years ago and it became the market leader of the industry since then. Previously, the task of setting up a heat exchanger product is done by a desktop application and the process of generating offers is handled manually. The application works without the internet which may cause risk to the business if the data is lost. In addition, the old configuration tool requires installation on the desktop which could be time-consuming. Therefore, taking the tool from the desktop to the web environment is a completely digital transformation for the company.

The goal of this implementation is to bring the whole desktop application to the web platform which enables error-quoting with a minimal amount of manual steps. This is not only a new sales and configuration tool but a complete digital transformation for the company which enables a new kind of collaboration and customer service. The first stage of the project would be an MVP application that covers all core features of the configuration and sale tool such as the ability to add product calculations based on a different configuration, generate an offer that includes multiple calculations, send an offer to the customer for approval. Some of the enhanced features are also put into consideration.

The back-end of the application is written with C# and .NET based technologies. The .NET ecosystem is robust and has extensive tooling for creating web services. The scope of this thesis is to focus on front-end testing that is built with ReactJS. Due to the scope of MVP, static, unit, integration, and acceptance testing will be covered in this project. Jest is used as a JavaScript Testing Framework in this project because it is fast to set up and it can be integrated well with other React testing libraries such as react-testing-library. Furthermore, Typescript which is known as Typed JavaScript will be utilized in the project. TypeScript saves development time by catching errors and providing fixes before the code is run. Other libraries used for static testing are Prettier, Lint-staged, Husky, and Eslint. Generally, the testing tools are successfully set up for this project which results in great confidence of developers while delivering quality code. This project forms a first step to the company's digital transformation in the near future.

For the sake of simplifying the code, an open-source project namely Testify is used to demonstrate the testing implementation. The Testify project has the same structure as the real-world React project that the company uses but it removes all the components which are not related to the thesis topic. Compared to the real project, the Testify form component is much simpler so the important concept of testing can be explained in greater detail. Furthermore, an open-source project would be beneficial for those who are interested in learning from the code.

5.2 Project challenges

The first challenge of building this web-based application is to balance between time and code quality. The MVP project must include nearly all the core features of the original tools which could take a year of work. In this project, the team already discussed which features are actually needed in the MVP version but it is quite challenging to define since the configuration tool is not considered a working version if there are missing core features. On the other hand, the first MVP version should be released within 2 months. The web must be built with high code quality but in a short time. Thus, there is no way to test everything.

The user interface and system design task is a second challenge for this project. The configuration tool requires lots of input and output fields while each input field value has its own logic based on the heat exchanger application and there are a total of 6 different applications in this project. This complex logic poses a threat to the project as it will easily lead to a bug. By contrast, there is no room for bug while configuring a heat exchanger which will become a physical product later on because if an error happens, the company will pay a huge price for the mistake.

Finally, more complex features, such as the ability to share calculations for review or compare different versions of the calculation, will be added to this project in the near future. The developer team needs to make sure that the application can be scaled without breaking the current code.

5.3 Technical solutions

As described in the above section, there are 3 main challenges in this project. The first solution would be to apply Agile methodology instead of the traditional one such as Waterfall. In this way, the team can work at a fast pace while communicating well with others and customers. Any new requirements from customers will be noticed and taken into consideration. The development team can produce new features every sprint which lasts one week while testing the application at the same time. If a bug is found, it is put in the backlog and will be fixed during the next sprint. In this way, the team can control the quality of the code while keeping moving forward.

The second solution is to integrate testing into the application. There is no point in testing every feature, therefore, knowing what to test is the most important task. For the scope of the MVP project, the team decides to utilize static, unit, integration, and acceptance testing in this project. The most low-hanging fruit testing is code validation which is also known as static testing. This type of testing needs only a one-time setup but it brings back values during the whole project. There are various ways that the application can crash. One of the common reasons is related to invalid code such as typos or incorrect types. For example, a developer may accidentally pass a string to a function that receives a number. This is where Typescript, Eslint, Husky, and Lint-staged come into play. Unit and integration testing should be applied only to important features during the MVP project. Moreover, acceptance testing will be carried by the customer at the end of the sprint to find the bug as soon as possible. There is no need for cross-platform testing in the project since the application is mostly used by internal users from the company.

Lastly, as the new feature will be kept added during the development time, CI is used in this project to ensure that a new feature does not break the current features. Since the project is hosted in Azure, the Azure pipeline which enables the developer to continuously build, test, and deploy to the Azure platform will be used in the project. When a new feature is added, it will trigger the pipeline to automatically run all the test cases. Therefore, the CI pipeline will prevent a new feature from developing to the staging or production if it breaks the test.

5.4 Implementation

This chapter focuses on the implementation of JavaScript Testing with the aim of addressing the challenges described above. All the testing tools which are used in the project will be described in the first section. Last but not least, the setup and application of these tools will be also explained in great detail.

5.4.1 Testing tools

Jest – JavaScript Testing Framework

There are several reasons why our developer team chose Jest over Mocha as the JavaScript Testing Framework. Firstly, Jest is straightforward to setup because of it depends on zero dependency as Jest is the test runner, assertion and mocking library itself. Secondly, Jest integrated with react-testing-library which is one of the most popular testing library for React. Moreover, snapshot testing which is useful for testing the React component is supported by Jest. By contrast, Mocha takes more time to setup which is quite time-consuming for the MVP and the customization of the testing tool is not needed in the React project as Mocha is quite robust for the simple React application.

TypeScript – Typed JavaScript

In the beginning, the project was set up with JavaScript by using create-react-app. However, JavaScript is prone to error because it has no type which causes difficulty when validating the data. For example, the developer can easily pass the string value to a function that receives number value. On the other hand, the application is used as a product configuration tool which requires lots of form data validation. Because of this reason, our team decided to re-setup the project with TypeScript.

TypeScript is an open-source language that extends JavaScript by adding static type definitions. TypeScript can prevent developers from making the most common error which is known as type error while using JavaScript. This could because of a failure to understand the function documentations, simple typos or other mistakes. Types are the shape of an object which validates that the object is initialized or set value with the correct type. The purpose of TypeScript is to run type checker before running the code which makes sure that the types of the program are valid. Writing type in TypeScript is not mandatory because the interface feature of TypeScript itself can bring lots of benefits to the JavaScript project. All valid JavaScript code is valid in TypeScript. It is also possible to decide which level of type striction should be used for the project (TypeScript Authors, 2020.) TypeScript is used in the project as a static testing tool.

Prettier – Code Formatter

Prettier is a code formatter which supports various languages including JavaScript and TypeScript. Prettier makes the code styling of the project more consistently by removing the original styling and reprinting the code with new styling based on the configuration rules. These rules could be based on the recommendation styling of a language or configured by the developer.

The biggest benefits of using prettier is to stop argument about coding styles between developers which is time-consuming. Generally, it is best practice that all the developers agree with the common style (Prettier Authors, 2020.) In this way, our team can cut down the time spent on style reviewing and truly focus on the code quality. Prettier cannot fix the error which is related to the logic but it can make the code much more readable which enables developers the ability to spot a common bug.

Eslint – Code Validator

Eslint is a static testing tool which automatically find and fix JavaScript code which prevents the application from bugs. The project code will be verified against a set of rules and most of the problems related to syntax are fixed automatically by Eslint. Therefore, Eslint makes the code more consistent access the team and it also detects the issue in the code patterns which could result in a potential bug. Every single rule can be removed or added by the developer. Furthermore, Eslint is also integrated in most editors and IDE and can be run along with the CI pipeline (Eslint Authors, 2020.)

In our project, React hooks which are introduced since 2018 by Facebook team are utilized. However, there are various mistakes that a developer can make when using hooks, for example, a developer may forget to pass the dependency array into the useEffect hook. All this kinds of issue can be addressed by just installing Eslint.

Husky – Git hook

Husky is an open source library which enables to run a script before every git commit (Chris, 2018). While working on the application, it is important that TypeScript does the type check, Eslint validates the code and all the tests are passing before committing code. Our team can utilize this feature thanks to Husky.

Lint-staged

Lint-staged is a NodeJS script that enables developers to run script against the staged files which prevents invalid code from slipping into the code base. Without lint-staged, every time the code is committed, linters such as Eslint will validate the whole project code which might take lots of time. Therefore, lint-staged is often used with husky to ensure (Andrey, 2016.)

React-testing-library

React-testing-library is a light-weight testing library for React components. The features of react-testing-library are built on top of react-dom/test-utils and react-dom which encourages best practices while testing the app. This library is recommended to be used with Jest although it is not mandatory.

There are several principles that the react-testing-library follows. The primary principle of react-testing-library is that tests should resemble the way the application is used. Secondly, if the tests deal with rendering components, DOM nodes should be used instead of component instances. This helps developers focus on the actual DOM instead of component implementation. Finally, all the APIs that react-testing-library exposes should be flexible and simple (React-testing-library authors, 2020.)

React-testing-library enables our team to write test components that are maintainable which means that the tests would not easily be broken by code refactoring. By using react-testing-library, developers can avoid implementation details of the components while writing tests. In this way, the tests can truly give us confidence when developing the app to production.

Jest-dom

Jest-dom is a library that adds various custom matches which are especially useful for testing the state of the DOM to Jest. In this way, developers can avoid boilerplate and repetitive code while working with DOM testing such as element's text content, CSS classes, and element's attributes assertion.

5.4.2 Static testing implementation

Static testing is all about automatically catching invalid code related to syntax or coding styling. As explained in the technical solutions section, Testify project is utilized TypeScript, Prettier, Eslint, Husky and Lint-staged for static testing. Notably, all of these libraries should be added to the project `package.json` before they can be used.

TypeScript configuration

In order to add TypeScript to a JavaScript project, the minimal step is to add tsconfig.js file to the root folder of the project. This is a special configuration file for TypeScript to declare that the directory is the root of a project. In JavaScript projects, jsconfig.js is used instead for the configuration. In our React project, Webpack and Babel are also utilized for the code transformation.

```
{
    "compilerOptions": {
        "moduleResolution": "node",
        "noEmit": true,
        "jsx": "react",
        "module": "commonjs",
        "target": "esnext",
        "esModuleInterop": true,
        "strict": true,
        "baseUrl": "src"
    },
    "include": ["src/**/*"],
    "exclude": ["dist", "node_modules"]
}
```

Listing 4. TypeScript configuration

The listing 4 demonstrates the `tsconfig` file in a React project. First, the base folder of the project is set to src. This is done by setting the `baseUrl` and include option. TypeScript should only see the src folder and ignore other files which are outside our root folder. The `dist` folder which is produced by Webpack and `node_modules` which contains external libraries should be ignored by TypeScript. In order to use JSX feature in

React, component files should be declared with `.tsx` extension and `jsx` option should be set to react.

It is possible that TypeScript can compile `ts` files to JavaScript. However, in a React project, it is common that TypeScript is only used for checking the types and Webpack and Babel are responsible for the code transformation. In the past, TS Compiler must do compile TypeScript files to JavaScript which is often done by adding `ts-loader` or awesome-typescript-loader to Webpack. The code is then transformed one more time with Babel. However, since Babel 7 is released, TypeScript files can be also compiled with Babel and there is no point in compiling the code twice with two different compilers. Thus, compilation function of TypeScript should be disabled to avoid conflict with Babel and `ts-loader` or awesome-typescript-loader can be removed. It can be done by simple set the noEmit option to true in `.tsconfig` file.

```
module.exports = {
    presets: [
        [
            '@babel/preset-env',
            {
                modules: isTest ? 'common.js' : false,
            },
        ],
        '@babel/preset-react',
        '@babel/preset-typescript',
    ],
    plugins: [
        '@babel/plugin-proposal-object-rest-spread',
        '@babel/plugin-proposal-class-properties',
        '@babel/plugin-syntax-dynamic-import',
        'react-hot-loader/babel',
    ],
    env: {
        production: {
            only: ['src'],
            plugins: [
                'transform-react-remove-prop-types',
                '@babel/plugin-transform-react-inline-elements',
                '@babel/plugin-transform-react-constant-elements',
            ],
        },
    },
}
```

Listing 5. Babel configuration

Finally, Babel should be configured to compile TypeScript code. As seen in listing 5, this is done by adding the preset `@babel/preset-typescript` to `.babelrc.js` file. Our project is now successfully switched from JavaScript to TypeScript which contributes greatly to the code validation.

Prettier and Eslint configuration

As described in the previous section, the main goal of Prettier is to format the code automatically. In this way, our developers can focus on the quality of the code in terms of business logic instead of code styling. There are two steps to config Prettier in the project. First of all, `.prettierrc` file is added to declare all the rules that Prettier should follow to format the code. Secondly, a prettier script is added to `package.json` file to run the code formatting. Notably, these rules can be changed based on the developer's preference. The listing 6 below describes the prettier configuration file.

```
{
    "arrowParens": "avoid",
    "bracketSpacing": false,
    "htmlWhitespaceSensitivity": "css",
    "insertPragma": false,
    "jsxBracketSameLine": false,
    "jsxSingleQuote": false,
    "printWidth": 80,
    "proseWrap": "preserve",
    "requirePragma": false,
    "semi": false,
    "singleQuote": true,
    "tabWidth": 2,
    "trailingComma": "all",
    "useTabs": true
}
```

Listing 6. Prettier configuration

On the other hand, Eslint is utilized to catch and fix issue with JavaScript code automatically. As observed from listing 7 below, setting up the Eslint is quite similar to Prettier. Although Eslint has a set of rules which developers can freely choose to add to the project, it is recommended that the configuration should extend a common set of rules which are approved by a trustworthy organization. In our project, the Eslint configuration extends `eslint:recommended` and `plugin:react/recommended` which is a set of rules recommended by Eslint and React. Furthermore, since there are some rules that are available in both Eslint and Prettier, `eslint-config-prettier` is extended to avoid the conflict between Prettier and Eslint. It is important to note that Eslint does not support

TypeScript by default. For this reason, a special Eslint parser namely `@typescript-eslint/parser` is used to enable Eslint in TypeScript.

```
{
  "parserOptions": {
    "ecmaVersion": 2019,
    "sourceType": "module",
    "ecmaFeatures": {
      "jsx": true
    }
  },
  "extends": [
    "eslint:recommended",
    "plugin:react/recommended",
    "eslint-config-prettier"
  ],
  "plugins": ["react"],
  "env": {
    "browser": true
  },
  "overrides": [
    {
      "files": "src/**/*.+(ts|tsx)",
      "parser": "@typescript-eslint/parser",
      "parserOptions": {
        "project": "./tsconfig.json"
      },
      "plugins": ["@typescript-eslint/eslint-plugin"],
      "extends": [
        "plugin:@typescript-eslint/eslint-recommended",
        "plugin:@typescript-eslint/recommended",
        "eslint-config-prettier/@typescript-eslint"
      ]
    }
  ]
}
```

Listing 7. Eslint configuration

Husky and Lint-staged configuration

Husky and Lint-staged play an important part in Static testing as it enables Eslint, Prettier and TypeScript check to run before the code is committed. Therefore, it is beneficial that developers cannot commit invalid code into the common git repository as they have to pass the styling and syntax validation.

```
{
    "scripts": {
        "start":"webpack-dev-server --config con-
fig/webpack/webpack.dev.js",
        "lint": "eslint --ext .ts,.tsx .",
        "prettier": "prettier --ignore-path ../.gitignore
\"**/*.+(js|jsx|json|yml|yaml|css|scss|ts|tsx|md|graphql|mdx)\"",
        "format": "npm run prettier -- --write",
        "check-format": "npm run prettier -- --list-different",
        "validate": "npm-run-all --parallel check-types check-format
lint",
        "check-types": "tsc"
    }
}
```

Listing 8. Testify application scripts

It can be seen in listing 8, three main scripts are added to the Testify project including script for running lint, formatting code and type checking. Firstly, the `lint` script ensures that Eslint only scans TypeScript files by declaring the extension that Eslint should support. Secondly, `format` script runs prettier on all the files except for the one added to `.gitignore` file. This script will automatically fix the code issue related to styling. Thirdly, types can be check by executing the `check-types` script. Finally, `validate` script is used for executing all the types-check, lint and prettier script parallelly.

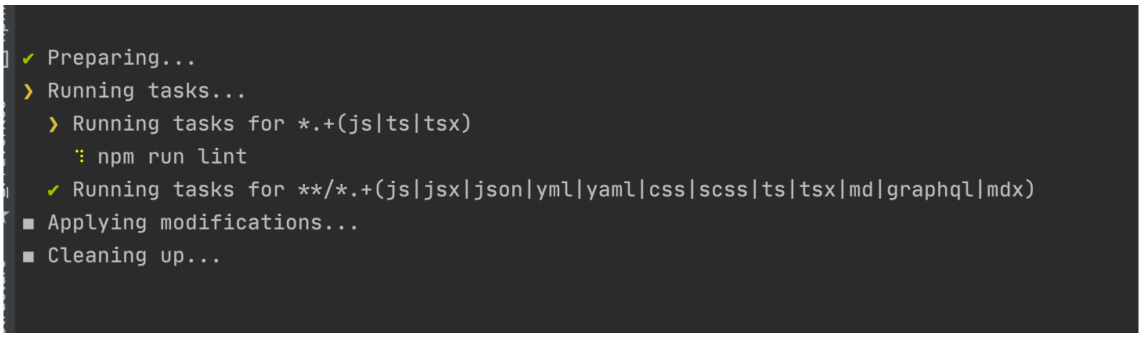
```
{
  "*.+(js|ts|tsx)": [
    "npm run lint"
  ],
  "**/*.+(js|jsx|json|yml|yaml|css|scss|ts|tsx|md|graphql|mdx)": [
    "npm run format",
    "git add"
  ]
}
```

Listing 9. Lint-stage configuration

```
{
  "hooks": {
    "pre-commit": "npm run check-types && lint-staged"
  }
}
```

Listing 10. Husky configuration

It can be observed from Listing 9 and 10, before the code is committed, Husky runs `check-types` script to make sure that our code has correct type. At the same time, the lint-staged script is also executed. Testify `lint-staged` then runs `lint` and `format` script against the staged files. Once the code formatting is done, lint-staged adds the changed code back to the staging area. If the type checking fails or Eslint catches invalid code in the application, Husky will prevent the code from being committed and show the error, otherwise, the code is committed as normal. In this way, developers can have much more confidence while distribute their code to the common repository.



```
✔ Preparing...
❯ Running tasks...
  ❯ Running tasks for *.+(js|ts|tsx)
    ⠹ npm run lint
  ✔ Running tasks for **/*.+(js|jsx|json|yml|yaml|css|scss|ts|tsx|md|graphql|mdx)
◼ Applying modifications...
◼ Cleaning up...
```

Figure 9. Pre-commit tasks

5.4.3 Unit testing implementation

Jest configuration

As explained in JavaScript testing framework theory, Jest comes with zero configuration. However, it is still worth adding Jest configuration file to the project so developers can have a bit control of the framework.

```
module.exports = {
    testEnvironment: 'jest-environment-jsdom',
    collectCoverageFrom: [
        'src/**/*.{js,jsx,ts,tsx}',
        '!src/**/*.test.{js,jsx,ts,tsx}',
        '!src/index.tsx',
        '!src/App.tsx',
    ],
    preset: 'ts-jest',
    // individual test will be reported
    verbose: true,
    // Automatically clear mock calls and instances between every test
    clearMocks: true,
    moduleDirectories: ['node_modules', 'src'],
    moduleFileExtensions: ['ts', 'tsx', 'js', 'jsx'],
    moduleNameMapper: {

'\\.(jpg|jpeg|png|gif|eot|otf|webp|svg|ttf|woff|woff2|mp4|webm|wav|mp3
|m4a|aac|oga)$':
            '<rootDir>/config/testing/__mocks__/fileMock.ts',
        '\\.(css|less|sass|scss)$':
            '<rootDir>/config/testing/__mocks__/styleMock.ts',
    },
    // Where Jest detectes test files
    testMatch: [
        '<rootDir>/src/**/__tests__/**/*.{ts,tsx,js,jsx}',
        '<rootDir>/src/**/?(*.)(spec|test).{ts,tsx,js,jsx}',
    ],
}
```

Listing 11. Jest configuration

It can be seen in the listing 11 that test environment is set to `jest-environment-jsdom.` The reason is that our application is a web app which depends on the browser API. In this configuration file, it is explicit to tell Jest that it should only looks for the JavaScript or TypeScript files. As the Testify project is using SCSS for component styling instead of JavaScript based styling such as styled-component, our tests will false because Jest will try to require the SCSS files as a common JS module. In order to fix that issue, `modu-leNameMap` is used to map module, which is ended with css, less or scss to a mock version of that module so it can be required from the tests.

Snapshot testing

Since our project is related to product configuration, there are various machine specifications that are declared. TypeScript Enum is used to document a set of specifications, therefore, a good strategy should be used to ensure that those specifications are valid and cannot be changed unexpectedly. For example, developers may accidentally modify or delete the value of the specification. Thus, Jest snapshot testing is the best tool to avoid this mistake. Snapshot testing is very useful to make sure that UI components or JavaScript objects do not change unexpectedly.

```
export enum Size {
    _size1 = 'size1',
    _size2 = 'size2',
    _size4 = 'size4',
}

describe('Machine', () => {
    it('should have a correct Size enum', () => {
        expect(Size).toMatchSnapshot()
    })
}.
```

Listing 12. Size Enum snapshot test case

It can be seen in listing 12 that our machine size has only 4 values and it only takes one line of code to test the enum. In our example, the snapshot testing takes a snapshot of the `Size` enum and compares it to a snapshot file that is stored alongside the test. In this way, if the Size value is changed accidentally, the Snapshot test will fail because the new image of the Size enum is different than the previous one. In our React application, there are hundreds of product specifications which means that there are hundreds of enum like the `Size` above. Snapshot testing makes it fast and easier to ensure that all the specifications are not modified by mistake. If the developer intentionally updates the enum, the test will fail and it should be updated to match the new snapshot which makes the developer more aware of what is actually changed.

Component testing

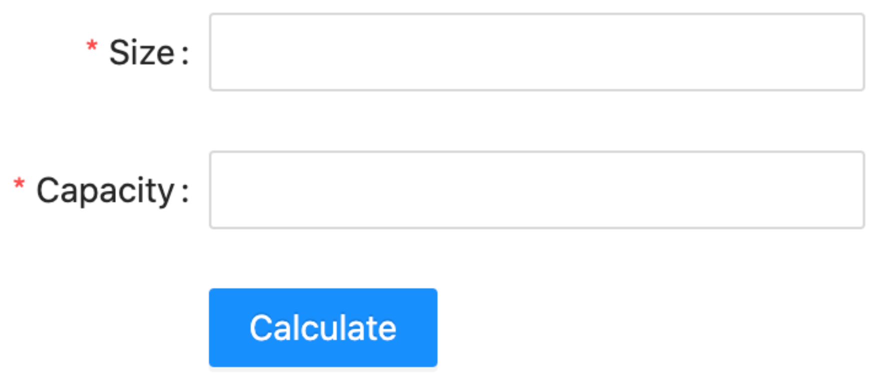


Figure 10. Product configuration form

Testify application have a product configuration form where user can calculate the specification for the machine. In a real-world project, the form would easily have 30 fields with various logic. However, for the sake of simplicity, only 2 inputs will be tested in the product configuration form. As described in figure 9, the form has 2 inputs namely Size and Capacity and a Calculate button. All the inputs are mandatory fields and the Capacity field only accept the number which is below or equal to 10.

It is unrealistic to test everything related to the form since it would take lots of time to write tests. This could easily burn the project budget which should be used for developing other important features of the application. Therefore, in this form, component styling can be ignored from testing because it can be easy to be tested manually when the client does the acceptance test. In addition, even the style of the button does not look exactly like the design, for example it may have a different color, the form would still work fine. Developers should focus on testing the functionality of the form rather than the implementation detail because the test should resemble the way user uses the form.

For this form component, there are 2 features that are important to test. Firstly, the form must be visible to the user. This means that the users should see all 2 fields and a calculate button on their screen. Secondly, the logic of the capacity field which states that the field should only receive a value less or equal to 10 must be correct. If this logic is incorrect then the result of calculation is wrong because the user can input an invalid data. This bug results in a critical error when the physical product is manufactured.

```
import React from 'react'
import './matchMedia.mock'
import user from '@testing-library/user-event'
import {render, waitFor} from '@testing-library/react'
import ConfigurationForm from './ConfigurationForm'

describe('ConfigurationForm', () => {
    it('renders a size and capacity input', () => {
        const {getByLabelText, getByText} = render(<ConfigurationForm
/>)
        getByLabelText(/size/i)
        getByLabelText(/capacity/i)
        getByText(/Calculate/i)
    })

    it('size should be less than 10', () => {
        const {getByLabelText, getByRole} = render(<ConfigurationForm
/>)
        const input = getByLabelText(/capacity/i)
        user.type(input, '11')

        waitFor(() => {
            expect(getByRole('alert')).toHaveTextContent(
                /'capacity' cannot be greater than 10/i,
            )
        })
    })
})
```

Listing 13. Configuration form unit test case

React-testing-library is used for our component testing because it tests a component in a way the user uses that component rather than focusing on the code implementation. As can be seen in listing 13, two test cases are included to test our two critical features accordingly. In the first test case, component visibility on screen is tested. The testing library renders the configuration form and tries to find an input with label `size` and `capacity` and the button which has `Calculation` text in the DOM. The test will fail if it cannot find an input with the correct label or the button with the correct text. This also means that the test would fail if the user cannot see the correct inputs or the calculation buttons on the screen. The second test case is related to the logic of the capacity input. First, the testing library renders our component and grabs the capacity input from the

DOM. Next, it simulates the user input by entering the number 11 which is an invalid number, and then expects the error to be shown on the screen. In other words, the test will break if it cannot find the error text after the user enters the invalid value to the form.

It is worth noting that there should be a test to verify the form works as expected after the user clicks on the calculate button. However, this behavior of the form is related to sending an HTTP request to the API. Therefore, this test case is related to the Integration testing which is covered in the next section.

Utils testing

It is worth testing util functions in our application since they are used in many components. One of the functions called `enumToValues` which is used to convert the enum object to array values. This function is used mostly in form component converting form selection options to values which is then posted to the server.

```
import {enumToValues} from './utils'

describe('Utils', () => {
    it('should convert enum to array value', () => {
        enum Role {
            admin = 'admin',
            user = 'user',
        }

        const values = enumToValues(Role)
        const expectedValues = ['admin', 'user']
        expect(values).toEqual(expectedValues)
    })
})
```

Listing 14. enumToValues util test case

As described in listing 14, an enum called Role which contains 2 values admin and user are declared. The `enumToValues` function receives the `Role` enum and then converts it to an array that contains the enum values. Without knowing about the implementation detail of the enumToValues function, Jest is used to expect the output of the utile function to be equal to an expected value. If this is true then our test case is passed which means that the function is correctly implemented. In this way, unit testing ensures that our testing functions are safe to use across the whole application.

5.4.4 Integration testing implementation

Integration testing is the phase where modules are combined and tested as a group. Unit testing is used to verify that the Configuration form described in the Unit testing section above is rendered correctly on the screen and have valid business logic. However, unit testing cannot ensure that the form would work as expected when it comes to sending value to the server. When the user clicks on the calculation button, the form is sent by the API util which is responsible for handling the HTTP request. Since the unit testing already covered the configuration and the API util function, it can be ensured that the form and the function would work without bug. However, issues can happen when the configuration form and the API util are used together.

```
const ConfigurationForm = () => {
    const onFinish = values => {processForm(values)}

    const processForm = value => {
        value = 0
        sendFormToServer(value)
    }

    return (
        <Form name="basic" onFinish={onFinish}>
            <Form.Item label="Size" name="size"
                rules={[{required: true, message: 'Please input your
size!'}]}>
                <Input />
            </Form.Item>

            <Form.Item label="Capacity" name="capacity"
                rules={[{required: true, max: 10, type: 'number'}]}
            >
                <InputNumber />
            </Form.Item>

            <Form.Item>
                <Button type="primary" htmlType="submit">
                    Calculate
                </Button>
            </Form.Item>
        </Form>
    )
}
```

Listing 15. Configuration form component

As observed from listing 15, when the form is submitted, `processForm` function handles the form value and uses the API function called `sendFormToServer` to send the form to the server. Although the form and the API util function are already verified by the unit test to ensure that they work as expected, however, the component above sent incorrect data to the server. The bug lines under the `processForm` function which accidentally resets the form value to 0. Integration test is used to address this issue since it verifies the interaction between the form and the util function.

```
import {sendFormToServer as mockedSendFormToServer} from
'../utils/apiUtil'

jest.mock('../utils/apiUtil')

describe('ConfigurationForm', () => {

    it('should send form data with correct value to the server', () =>
{
        // Arrange
        ;(mockedSendFormToServer as jest.Mock).mockResolvedValueOnce({
            statusCode: 200,
        })

        const {getByText, getByLabelText} = render(<ConfigurationForm
/>)
        const submitButton = getByText(/Calculate/i)
        const mockedSizeValue = Size._size1
        const mockedCapacityValue = '9'

        // Act
        user.type(getByLabelText(/size/i), mockedSizeValue)
        user.type(getByLabelText(/capacity/i), mockedCapacityValue)
        fireEvent.click(submitButton)

        // Assert

        waitFor(() => {
            expect(mockedSendFormToServer).toHaveBeenCalledWith({
                size: mockedSizeValue,
                capacity: mockedCapacityValue,
            })

            expect(mockedSendFormToServer).toHaveBeenCalledTimes(1)
        })
    })
})
```

Listing 16. Configuration form integration test case

It can be seen from listing 16, `sendFormToServer` function is mocked by Jest. This is because the test should avoid calling the real API which can result in a modification in a real database. In the test case above, the integration test resembles the way the user uses the configuration form. First, the user would fill out the form with the correct value then click on the submit button. The test then expects that the `sendFormToServer` to receive the correct value from the form. If the API function receives the accurate data then it is safe to expect that the form sends the correct data to the server.

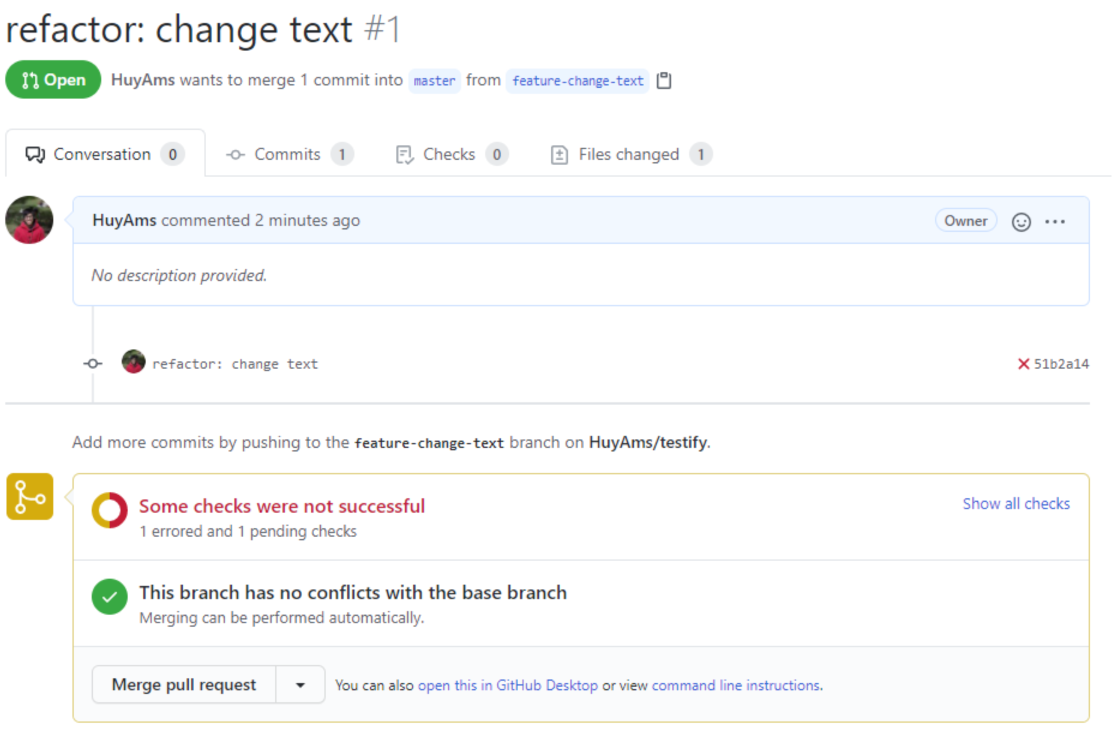
5.4.5 CI configuration

Travis CI is used as a continuous integration tool for the Testify project which automatically builds the application and triggers the testing every time new code is pushed to the common repository. This provides immediate feedback for our team and helps us cut down on manual testing. Travis CI can be easily set up by adding a configuration file namely `.travis.yml` to the root of the project and enabling Travis CI for the remote repo.

```
language: node_js
cache:
  directories:
  - "node_modules"
install:
  - npm install
script:
  - npm run test
```

Listing 17. Travis CI configuration

When the CI is triggered, it clones the Testify project from the GitHub repository into a virtual environment the uses the script in the configuration file to build and test the application. It can be seen from listing 17, Travis CI does 2 main jobs. Firstly, it runs the script to install all the required dependencies by the project. In this way. Travis CI then runs the test script which triggers Jest to run all the test cases including the Unit test and Integration test of our application. The code cannot be merged into the main branch if the testing has not passed. Figure 11 below shows that a pull request cannot be merged if it has not passed the test.


refactor: change text #1

Open HuyAms wants to merge 1 commit into master from feature-change-text

Conversation 0 Commits 1 Checks 0 Files changed 1

HuyAms commented 2 minutes ago Owner

No description provided.

refactor: change text ✗ 51b2a14

Add more commits by pushing to the feature-change-text branch on HuyAms/testify.

Some checks were not successful
1 errored and 1 pending checks
Show all checks

This branch has no conflicts with the base branch
Merging can be performed automatically.

Merge pull request You can also open this in GitHub Desktop or view command line instructions.

Figure 11. Travis CI checks in GitHub

5.4.6 Acceptance testing implementation

In our development flow, acceptance testing is executed at the end of every sprint which lasts for a week. The developer team will conduct a demo to introduce new features or show that bugs have been fixed to the client. The new code is then deployed to the staging environment where the client can interact with the application and find out if everything works as it should. If a new bug is found then it is reported with great detail on how to reproduce the bug on a collaboration platform such as Trello or Jira. The bugs that have been reported will be viewed and fixed by the developer in the next sprint.

5.5 Evaluation

The development team successfully implemented the JavaScript testing on the React application which is deployed to the Azure App Service. Software testing enables our team to find bugs at the very first beginning of our project which cut down on the technical debt greatly. The outcome of the project is a fully-functioning and bug-free React application which met the client requirement.

The application bought a new solution to the challenging issue that the client has suffered, cutting down the amount of manual work. This React application would replace the old desktop-based application that our client is using. The implementation of testing met the ultimate goals which is achieving the balance between code quality and time constraint. Unit testing covers all the important util functions and the form component which is the core feature of the configuration application. Integration testing is written in a way it resembles the way the user would use our application which boosts the developers confidence while deploying the application. Furthermore, continuous integration is implemented to the application and runs the testing every time the code is changed. In this way, it can be guaranteed that invalidate code cannot be pushed into the production environment and a new version of application can be released as rapidly as possible.

6 Conclusion

The main objective of this thesis was to investigate and demonstrate some common testing levels. An open-source project called Testify has been created successfully to demonstrate how to setup and write tests in a real world React application. In addition, three most popular JavaScript frameworks are compared and studied in great detail. This thesis hopefully encourages developers to have a good understanding and implement testing to make the software system more reliable. Moreover, this thesis is currently used as educational material in Columbia Road which is a company I am working at.

In conclusions, there are four main testing levels namely Unit, Integration, System and Unit testing. It is valuable to understand the costs and benefits of each testing level. Unit testing is the cheapest out of 4 levels of testing; however, it cannot ensure that the application will work as expected. On the other hand, Integration and System testing can verify that the application would work correctly but it would come with high cost since it takes lots of resources to implement. Furthermore, it is essential to note that it is unrealistic to cover one hundred percent of an application with tests. Balance between multiple levels of testing should be achieved to ensure the application is bug-free.

Jest, Mocha and Jasmine are the most popular JavaScript testing frameworks. However, a framework needs to be carefully chosen to suit the project. Jest is fast and requires zero setup; by contrast, Mocha would be the best option when flexible configuration is needed.

References

Andrey Okonetchnikov (2016) *Make linting greate again!*. [Online] Available at: https://medium.com/@okonetchnikov/make-linting-great-again-f3890e1ad6b8#.8qepn2b5l (Accessed 22 October 2020).

Arnab Roy Chowdhury (2019). *Software Testing Life Cycle – Everything you Need to Know*. [Online] Available at: https://www.testim.io/blog/software-testing-life-cycle/ (Accessed 17 October 2020).

Brooklin Nash (2020). *Agile vs Waterfall: Learn the Differences in 5 mins*. [Online] Available at: https://bit.ly/2T11jR4 (Accessed 17 October 2020).

Charith Rhettiarachchi (2016) *Why use jest over karma for Angular testing?* [Online] Available at: https://medium.com/@charith.rhettiarachchi/why-use-jest-over-karma-for-angular-testing-b56ffa82f8 (Accessed 19 October 2020).

Chris House (2018) *Pre-commit git hooks with Husky*. [Online] Available at: https://blog.vanila.io/pre-commit-git-hooks-with-husky-b2fce57d0ecd (Accessed 22 October 2020).

Coding Infinite (2019). *What stats & surveys are saying about top programming languages in 2020*. [Online] Available at: https://codinginfinite.com/top-programming-languages-2020-stats-surveys/ (Accessed 6 October 2020)

Cortney Alyssa Wood (2014). *Kitchenham, Pfleeger and Garvin's Five perspectives of Quality Applied to Theater Ticket Managing Software* . pp. 2

Dan Radigan (2020). *Continuous integration*. [Online] Available at: https://www.atlassian.com/agile/software-development/continuous-integration (Accessed 28 October 2020).

David Swersky (2019). *Mocha vs Jasmine, Chai, Sinon & Cucumber in 2019*. [Online] Available at: https://bit.ly/34bb8m7 (Accessed 19 October 2020).

Eslint Authors (2020). *Getting started with Eslint*. [Online] Available at: https://eslint.org/ (Accessed 22 October 2020).

Hazem Saleh (2013). *JavaScript Unit Testing.* UK: Packt Publishing Ltd.

Ilene Burnstein (2003). *Practical Software Testing: a proccess-oriented approach.* New York: Springer-Verlag.

Jeff Tian (2005). *Software quality engineering. Testing, Quality assuarance, and quantifiable improvement*. New Jersey: John Wiley & Sons, Inc.

jestjs.io (2020). *Jest Framework*. [Online] Available at: https://jestjs.io/ (Accessed 18 October 2020).

Katia Wheeler (2018). *Jest vs Mocha: Which Should You Choose?*. [Online] Available at: https://blog.usejournal.com/jest-vs-mocha-whats-the-difference-235df75ffdf3 (Accessed 18 October 2020).

Kent C.Dodds (2018). *How to know what to test*. [Online] Available at: https://kentcdodds.com/blog/how-to-know-what-to-test (Accessed 10 October 2020).

Kent C.Dodds (2019). *Write tests. Not too many. Mostly integration*. [Online] Available at: https://kentcdodds.com/blog/write-tests (Accessed 12 October 2020).

Kent C.Dodds (2020). *Static vs Unit vs Integration vs E2E Testing for Frontend Apps*. [Online] Available at: https://kentcdodds.com/blog/unit-vs-integration-vs-e2e-tests (Accessed 12 October 2020).

Kshirasagar Naik, Priyadarshi Tripathy (2018). *Software testing and quality assurance*. New Jersey: John Wiley & Sons, Inc.

Liang Yuxian Eugene (2010). *JavaScript Testing. Test and debug JavaScript the easy way*. UK: Packt Publishing Ltd.

Lindsay Liedke (2019). *100+ Internet statistics and facts for 2019.* [Online] Available at: https://www.websitehostingrating.com/internet-statistics-facts/ (Accessed 6 October 2020).

Lisa Crispin, Janet Gregory (2009) *Agile Testing: A practical guide for testers and agile teams*. US: Pearson Education, Inc.

Michael Bogan (2020). *Comparing the top 3 JavaScript testing frameworks*. [Online] Available at: https://dev.to/heroku/comparing-the-top-3-javascript-testing-frameworks-2cco (Accessed 18 October 2020).

Npm trend – jest vs mocha vs jasmine (2020) [Online] Available at: https://www.npmtrends.com/jest-vs-mocha-vs-jasmine (Accessed 18 October 2020)

Paul Fields, Daryl Hague, Geoffrey S. Koby, Arle Lommel, Alan Melby (2014). *What is Quality? A Management Discipline and the Translation Industry Get Acquainted*. Vol 404-412 .

Paul Pop (2002). *Comparing Web Application with Desktop Applications: An Empirical Study.*

Prasad Mahajan (2016). *Different Types of Testing in Software Testing.* International Research Journal of Engineering and Technology (IRJET). Vol 03. pp. 1663.

Prettier Authors (2020). *What is Prettier*. [Online] Availabe at: https://prettier.io/docs/en/index.html (Accessed 22 October 2020).

Rajkuma SM (2019). *What is Software Testing – Definition, Types, Methods, Approaches*. [Online] Available at: https://www.softwaretestingmaterial.com/software-testing/ (Accessed 10 October 2020).

React-testing-library authors (2020) *React Testing Library* [Online] Available at: https://testing-library.com/docs/react-testing-library/intro (Accessed 24 October 2020)

TypeScript Authors (2020). *The TypeScript Handbook*. [Online] Availabe at: https://www.typescriptlang.org/docs/handbook/intro.html (Accessed 22 October 2020).

Ulf Eriksson (2014*). Differences Between the Different Levels & Types of Testing*. [Online] Available at: https://reqtest.com/testing-blog/different-levels-of-testing/ (Accessed 11 October 2020).

William E.Lewis (2005) *Software Testing and Continuous Quality Improvement*. Second edition. New York: CRC Press L.C.C.

Yana Andyol (2020). *Software Testing Life Cycle: a Model-Based Explaination*. [Online] Available at: http://www.qamadness.com/software-testing-life-cycle-a-model-based-explanation/ (Accessed 15 October 2020).