Bogdan Moroz

UNIT TEST AUTOMATION OF A REACT-REDUX APPLICATION WITH JEST AND ENZYME

Bachelor’s thesis
Degree programme in Information Technology

2019

South-Eastern Finland
University of Applied Sciences

XAMK
South-Eastern Finland
University of Applied Sciences

Author	Degree	Time
Bogdan Moroz	Bachelor of Engineering	May 2019

Title

Unit Test Automation of a React-Redux Application with Jest and Enzyme

108 pages
12 pages of appendices

Commissioned by

Observis Oy

Supervisor

Timo Hynninen

Abstract

The goal of this bachelor's thesis was to evaluate the benefits of test automation and determine an efficient unit testing strategy for React applications developed at the commissioner company Observis. The thesis aimed at providing examples of units tested according to the strategy and integrating the automated execution of the tests into the deployment pipeline for the case study application – a dashboard application for workforce management called MOWO. MOWO dashboard is written in TypeScript using React and Redux.

The proposed strategy was formulated based on the findings of the literature review on test automation, an overview of the technology stack of the application as well as a summary of four testing strategies used by development teams in other software development companies. The proposed testing strategy aims to improve the quality of React-Redux applications developed by Observis, help developers understand the software under test and reduce the risks of making changes to the existing code. The strategy identifies the four types of units that need to be tested: utility functions, action creators, Redux reducers and React components. The strategy provides instructions on how to structure the tests to make them expressive, readable and maintainable.

In the practical part of the study, detailed examples of tests for each of the four types of units are provided. The automatic execution of tests is achieved by configuring a Git hook for the project as well as creating a project on the Jenkins automation server that executes the tests every time a new merge request is created on GitLab and leaves a comment on GitLab notifying whether the tests passed or failed.

Keywords

Testing, Unit testing, Test automation, Behaviour-driven development, React, Redux, Jest, Enzyme, Typescript, Jenkins

CONTENTS

1 INTRODUCTION

IT organizations strive to provide stable, reliable and secure services to their customers (Kim et al. 2016, xxv). Production ready software requires testing before it is released into production environments. Over the years, approaches to software testing have matured. The fully manual process of testing applications has largely been replaced with writing and executing automated tests. Automated testing enables a fast feedback loop that allows developers to find and fix errors soon after they make them. It also gives developers confidence to refactor the code without breaking existing features. (Vocke 2018.)

However, given the test code is not what the customer is paying for, temptation may arise to abandon testing, when the tests become difficult or expensive to maintain (Meszaros 2007). It is therefore important to study and practice methods and tools that make test automation efficient. The extra cost of writing and maintaining automated tests must provide net benefits by reducing the amount of manual testing and debugging (Meszaros 2007).

In the early days of test automation user interface testing was considered the slowest and most complicated step of software testing. UI tests have been placed at the top of the test pyramid when the concept was introduced by Cohn in 2009. Nowadays, when single-page web applications built with frameworks and libraries like React, Angular and Vue are widespread, this is no longer the case. UI can be unit tested just like other applications. (Vocke 2018.) However, due to the proliferation of available frameworks and tools, there is no one-size-fits-all solution. In the world of front-end development in 2019 there are multiple competing tools and strategies for software testing, and discussions on the right approach can get opinionated and heated.

Observis Oy is committed to continuously improving its software development process. Deployment pipelines are set up for most of the applications developed by the company. However, most of these pipelines include little to no automated tests. Observis is looking for ways to transform the development lifecycle of its products to include thorough automated software testing.

One of the products developed at Observis is a dashboard application for workforce management. It is a web application written in TypeScript using the React and Redux libraries. A deployment pipeline is set up for the application using the Jenkins automation server. Until this point, no tests have been written for the application. The process of testing the application has been manual. The manual approach, even though rigorous, has not been comprehensive. Software bugs were seeping into the production environment as a result.

The goal of this bachelor’s thesis is to evaluate the benefits of test automation for the commissioner company, determine an efficient unit testing strategy for React applications developed at Observis, provide examples of units tested according to the strategy and integrate the automated execution of the tests into the deployment pipeline for the application. The thesis discusses the fundamental concepts of test automation and unit testing and looks at how some of the established industry patterns and techniques can be applied to React-Redux applications. The thesis provides instructions for testing React-Redux applications written in TypeScript using the Jest testing framework with the Enzyme testing utility. The study aims at documenting the setup process of the testing environment, as well as the design and creation of unit tests. The automatic execution of the said tests will be achieved by integrating them into the existing deployment pipeline for the client application described above. Moving forward, this work can be used as a reference by software developers at Observis. Moreover, the testing strategy presented here can be adopted by developers and teams outside the company.

The thesis aims at answering the following research questions:

- How does test automation benefit a company?
- What unit testing strategy can be adopted for React applications?
- How should a testing environment be set up for the specific technology stack of the case study (React, TypeScript, Redux, Jenkins)?
- How to evaluate the efficiency of a testing strategy?

The thesis discusses the key principles of automated testing, including the three commonly used types of automated tests: unit, integration and end-to-end (see e.g. Dodds 2017b; GitLab Documentation 2019b; Pittet 2019b; Wacker 2015).

However, the work focuses on unit tests only. Tools and methods for integration and end-to-end testing are not the target of the study.

The thesis is structured as follows: Chapter 2 provides the theoretical background on automated testing. Chapter 3 introduces the technologies that comprise the SUT and the tools used to test it. Chapter 4 presents the proposed testing strategy for the application. Chapter 5 documents the implementation of the testing strategy. This includes the setup of the testing environment, examples of tests and components under test, code coverage reports and how the tests are integrated into the deployment pipeline. Chapter 6 discusses the results of the study, while Chapter 7 provides the conclusion and suggestions for future research.

2 AUTOMATED TESTING

The research method used for this thesis can be characterized as an illustrative case study as defined by the Center of Innovation in Research and Teaching (2019). The research is closely focused on the case study application MOWO dashboard, including its technology stack, deployment pipeline and the development workflow used by the team working on the application. The study provides a high level of detail on the technologies of the case study application as well as the tools used to test these technologies. In order to discuss a testing strategy for the case study application, the study introduces the fundamental concepts of automated testing and outlines the terminology, techniques and practices used in the industry in the field of software testing. The study uses a definition of testing provided by IEEE/ANSI as the theoretical framework: testing is “a process of operating a system or component under specific conditions, observing or recording the results, and making an evaluation of some aspect of the system or component” (IEEE/ANSI, cited by Kit, 1995).

This chapter introduces the fundamental concepts of automated testing. Motivation for automated testing is established and types of automated tests are listed. The goals of unit test automation are presented. The following section lists various types of code coverage and explains how code coverage reports can be interpreted and used. Then, the behaviour-driven development (BDD) technique and spec-style syntax for writing tests are introduced. Finally, various types of test doubles are discussed.

2.1 Motivation

Traditionally, the process of software testing has largely been manual (Humble 2017a). Code changes made by one developer have been reviewed by another, and the software product itself underwent manual regression testing by quality assurance (QA) teams only after the development process has been completed. Manual testers followed the software documentation step by step to verify that new and existing features work as required. Any issues found by QA were then forwarded back to the developers to fix. When software testing is separated from development and conducted only several times a month or a year, developers often learn about the errors they had made weeks or months after they made them. By that point it is difficult to quickly recollect the logic of the code that has an error, and the error becomes slow and hard to resolve and learn from. (Humble 2017a; Kim et al. 2016, 123.) Aside from the slow feedback for developers, manual testing slows down the pace at which new versions of the software can be released to customers. The reliability of the manual approach can also be questioned, as manual repetitive tasks are prone to human error. Moreover, as the software system under test (SUT) grows and evolves, the documentation used for manual testing requires to be continuously updated as well, costing additional time and effort. (Humble 2017a.)

Automated testing addresses the problems described above. Nowadays, it is a widely adopted practice for developers to write automated tests that fail the build of an application if they discover regressions (Koskela 2013, 4). Running automated tests locally allows developers to notice bugs early and fix them quickly, oftentimes before the errors are checked into version control (Humble 2017c). Automated tests are faster to execute than their manual counterparts and allow for a faster rate of new releases. These tests improve the productivity of software development teams and allow gaining and sustaining a high speed of development. Automated tests require no manual intervention while running.

Good tests are repeatable, meaning they always produce the same result given the same input. This greatly reduces the risk of human errors that manual regression testing is subject to. Automated tests can act as specification and documentation for the software, as it will be demonstrated below. This documentation is a living document updated along with the software and constantly kept up-to-date. (Koskela 2013, 4; Meszaros 2007; North 2012.) According to Gruver (2014, cited by Kim et al. 2016, 123), “without automated testing, the more code we write, the more time and money is required to test our code – in most cases, this is a totally unscalable business model for any technology organization.”

2.2 Types of automated tests

Automated software testing can be performed at different levels, from the simplest building blocks of an application to how the application operates as a whole. The test pyramid, a concept popularized by Cohn, is a way of visualizing a testing strategy that involves multiple layers of varying types of automated tests. The pyramid provides a balance of speed and complexity by placing a large number of small, fast-to-execute tests at the bottom, with increasingly slow and complex tests towards the top. (Fowler 2012; Vocke 2018.) Figure 1 shows a variation of the pyramid as presented by Google at the 2014 Google Test Automation Conference (Wacker 2015). The three types of automated tests mentioned in the pyramid are unit, integration and end-to-end tests. Multiple terminologies exist that define different types of automated tests. The three aforementioned types are commonly found in many of these terminologies (see e.g. Dodds 2017b; GitLab Documentation 2019b; Pittet 2019b; Wacker 2015).

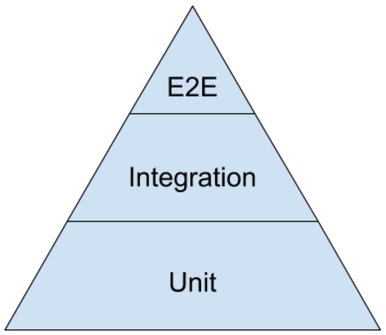


Figure 1. Testing pyramid (Wacker 2015)

A unit test is a term typically defined based on the specific context of the SUT, and there is no single canonical definition (Vocke 2018). According to Johansen (2010, 6), “a unit test is a piece of code that tests a piece of production code. It does so by setting up one or a few more objects in a known state, exercising them (e.g. calling a method), and then inspecting the result, comparing it to the expected outcome”. A broader definition lists a set of common traits shared by all unit tests. First, they are low-level, each focusing on a small part of the codebase. Second, they are written manually by software developers. Third, they are significantly faster to execute than other kinds of tests. According to Fowler (2014), the variation in precise terminology stems from different definitions of a unit by any given software team or an individual developer.

An integration test has a larger scope than a unit test. Once again, varying definitions of the term exist. Essentially, the goal of an integration test is to verify that different parts of an application work together correctly. According to Wacker (2015), “an integration test takes a small group of units, often two units, and tests their behaviour as a whole, verifying that they coherently work together” (Wacker 2015).

An end-to-end test “replicates a user behaviour with the software in a complete application environment. It verifies that various user flows work as expected”. (Pittet 2019b.) The effectiveness of this type of tests is most immediately noticeable. By executing a complete user scenario with the software, it is easy to spot deviations from the required outcome. However, there are strong disadvantages to overly relying on end-to-end tests. First, end-to-end tests can oftentimes be fragile and “flaky”. A flaky test is “a test that exhibits both a passing and a failing result with the same code” (Micco 2016). Flakiness comes as a result of the large scope of the test. Potential causes of test flakiness include dynamic content, caching, concurrency and infrastructure issues (The Code Gang 2017). Meszaros (2007) mentions behaviour, interface, data and context sensitivity as reasons for fragile tests. This doesn’t mean that end-to-end tests are the only tests subject to flakiness. However, due to their large scope they are more vulnerable to this problem when compared to unit and integration tests.

Second, end-to-end tests can be slow to execute. As mentioned earlier, fast feedback is important for software developers in order to reduce the errors that make it into production environments. Idly waiting for slow tests to complete reduces the efficiency of software development teams. Third, determining the root cause of a failing end-to-end test can be a challenging and time-consuming task, as smaller issues can be hiding behind larger ones. (Wacker 2015.)

The purpose of this study is to develop a unit testing strategy. From this point onwards, the terms “test” and “automated test” will be used to refer to unit tests unless otherwise specified.

2.3 Goals of test automation

In order to determine a unit testing strategy for an application, it is important to set the goals that the strategy should help accomplish. Goals of test automation established by Meszaros (2007) fit this purpose well. The first three goals focus on the benefits that the automated tests should provide, while the final three specify the qualities that the automated tests should possess.

1. Tests should help improve quality
2. Tests should help understand the SUT
3. Tests should reduce (and not introduce) risk
4. Tests should be easy to run
5. Tests should be easy to write and maintain
6. Tests should require minimal maintenance as the system evolves around them

What benefits tests should provide

First, tests should improve the quality of the software. The quality of the software can be determined by answering two questions: whether the software was built correctly and whether the right software was built. A practice that helps answering these questions independently from each other is writing tests before writing software. This approach makes it is easier to consider what the software should do separately from how it should do it. (Meszaros 2007.) The test-first approach to writing tests where the tests help design the software is widespread in the industry. By writing tests first it is easy to verify the behaviour of the system before verifying its implementation. (Koskela 2013, 4.)

Test-driven development (TDD) is a software development technique where the development process is guided by writing tests. Every time a new piece of functionality needs to be implemented, a test for that functionality is written first. Then, the feature itself is implemented and passes the test. Finally, both the feature and the test are refactored for structure and clarity. (Fowler 2005.) Writing tests before the software provides a clear definition of success. In this way, tests act as specification for the software. (Meszaros 2007.) Meszaros refers to such tests as the “executable specification”. By thinking through the specifications of the software before starting the development, it is easy to spot and eliminate inconsistencies and ambiguities. Improvement of the specification leads to improved quality of the final product. (Meszaros 2007.) A study conducted by Nagappan et al. (2008, cited by Kim et al. 2016, 135) found that software teams using TDD developed code from 60 to 90% better in terms of defect density compared to non-TDD teams, while spending only 15 to 30% longer on development.

As new tests and features are implemented, all the previous tests are executed repeatedly. Previous tests provide automated regression testing, making sure that any new feature does not introduce harmful side effects. This makes sure most bugs are noticed soon after being introduced, long before they make it to version control or, worse, the production environment. (Meszaros 2007).

Second, automated tests should help understand the SUT. As mentioned above, tests can be written in a way that provides specification of the software. A well planned and clear test suite can act as a form of documentation for the software – it should be clear what the application should do from reading the tests. (Meszaros 2007.) This idea is taken further by a technique called behaviour-driven development discussed in Section 2.5.

Third, tests should only reduce, not introduce risk. Automated tests reduce the risk of breaking existing features by making modifications, giving developers confidence to make changes to legacy code. Old code can be edited and refactored, but if the tests still pass, a developer can be sure their changes did not disrupt the required behaviour of the system. It is important to note that the level of confidence directly depends on how detailed the test suite is, how precisely it verifies the behaviour of the system. On the other hand, poorly constructed tests can provide a false sense of confidence by giving false positives. A test that overly relies on test doubles (a concept discussed in detail in Section 2.6) can replace too much of the SUT. The result is a test that verifies the behaviour of a test double that does not match the behaviour of the SUT. Modifying the SUT in tests is a risk that must be avoided. (Meszaros 2007.)

What qualities tests should possess

This subsection discusses which qualities make an automated test good. Tests should be easy to run. The TDD workflow requires continuously re-running all tests as the software is developed. This is only feasible to do frequently if tests are fast to run, as the point is to provide fast feedback. That said, even though making tests as fast as possible is a good general guideline, its importance may vary depending on the specific context. For certain tests extreme focus can be more important than fast execution. (Koskela 2013, 15–16; Meszaros 2007.)

Executing the tests should require no manual intervention. Each test must also be repeatable, meaning it produces the same results every time it is executed. If that is not the case, each failure requires manual inspection in order to determine whether the failure is caused by the flakiness of the test or by an actual error caused by a recent change to the SUT. Re-running tests to verify that there really is an issue increases the effort of running tests and evaluating their results, which can lead to developer frustration and test results being ignored as unreliable. To achieve repeatability, nondeterministic logic must be avoided in test code. Asynchronous calls and code that depends on current time are two examples of threats to repeatability. Such threats can be mitigated using test doubles, a concept described in Section 2.6. (Kim et al. 2016, 135; Koskela 2013, 24–25; Meszaros 2007.)

Tests should be simple to write and maintain. The test code itself should be simple enough to not require any tests, as testing the test code is a cumbersome task that should be avoided. Studies have shown a correlation between poor code readability and the density of defects in the code (Buse & Weimer 2009, cited by Koskela 2013, 17). Code that is hard to read and understand is also difficult to modify in a meaningful way and therefore difficult to maintain. To achieve readability only one condition should be verified per test as a general guideline. Each feature under test should have a separate test method for each unique combination of state and input before the test. In other words, each test should exercise a single “code path” with the SUT. Moreover, tests should be expressive. Code duplicated in multiple tests (e.g. setup code) should be moved to centralized test utility methods. Using utility methods allows to communicate the intent of the test more clearly. (Koskela 16–18; Meszaros 2007.)

Finally, tests should require minimal maintenance as the SUT evolves around them. Obviously, changes to the SUT are inevitable during the development process. Automated tests should give developers confidence to modify existing code. Making changes to a single method or interface should not result in dozens of failing tests, as that would discourage any modifications to the SUT. Large amounts of failing tests caused by a single change to the SUT is a sign of excessive duplication in the tests. To avoid this, test overlap should be minimized. Reduced test duplication and verifying one condition per test make sure only a few tests will require modification when a change is made to the SUT. (Meszaros 2007; Fowler 2004b.)

2.4 Code coverage

Code coverage is a metric used to express which parts of the software are not exercised by the tests, how much of the source code is tested (Meszaros 2007; Pittet 2019a). The metric can be applied to various types of automated tests including unit testing. The coverage is typically measured in per cent, a relation of items tested to all items found (Pittet 2019a). Various tools exist to provide different types of coverage reports (see e.g. Stackify 2017). Pittet (2019a) defines five common metrics of test coverage based on different criteria:

- Function coverage – How many of the defined functions have been called (Pittet 2019a)?
- Statement coverage – How many of the statements have been executed (Pittet 2019a)?
- Branches coverage – How many branches of control structures (e.g. if blocks) have been executed? In other words, whether the code was driven through a code path where an if block was resolved as true, and a code path where that if block was resolved as false. (Pittet 2019a.)
- Condition coverage – How many of the boolean sub-expressions have been tested for a false and a true value? In other words, for each if block where the condition consists of one or multiple boolean statements, was every combination of statements in a condition resolved as true or false used? (Pittet 2019a.)
- Line coverage – How many lines of source code have been executed (Pittet 2019a)? Line coverage can be very close to statement coverage, as typically developers place each new statement on a separate line.

Even though 100% test coverage may seem like a good goal, in reality it doesn’t guarantee the lack of defects. 100% coverage only means that all of the code under test was executed, not that the system behaves according to specification. (Koskela 2013, 6.) According to Koskela (2013, 6–7), after a certain point in the development process, the more unit tests are written, the less value they provide compared to the effort required to write them. Typically, the first tests that are written cover the crucial high-value high-risk functionality of the software. After the vast majority of the codebase is covered with tests, the remaining untested parts are the least likely to break. (Koskela 2013, 6–7.) Figure 2 illustrates this pattern of diminishing returns.

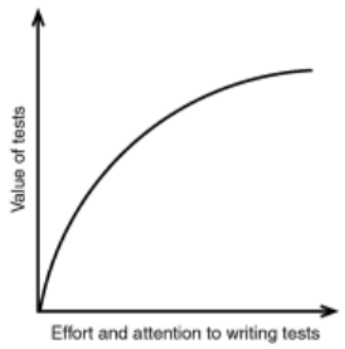


Figure 2. Quality plateau of automated tests (Koskela 2013, 6–7)

Code coverage does not prevent faults of omission – bugs that can only be fixed by adding new code (Marick 1997). Figure 3 provides an example of code under test written in pseudocode.

```
status = perform_operation();  /* do something and assign status code. */
if (status == FATAL_ERROR)
        exit(3);                /* exit with error code 3 */
else
        all is OK; continue on.
```

Figure 3. Pseudocode under test (Marick 1997)

Tests with 100% branch coverage can be written for the code in Figure 3. However, the code can still be wrong. For instance, perform_operation() can return three status codes instead of the two handled by the code: FATAL_ERROR, ALL_OK and RECOVERABLE_ERROR. The code in Figure 3 treats RECOVERABLE_ERROR as ALL_OK, and it is impossible to tell that something is missing by looking at the code coverage. Test coverage tools can only tell the user how much of the existing code has been exercised. The cause of faults of omission is not limited to missing if conditions, but rather includes entire missing if blocks, exception handlers and more advanced cases. (Marick 1997.)

This begs the question of how the coverage reports can be interpreted in a meaningful way. Fowler (2012) and Marick (1997) believe that code coverage is only helpful to find areas of the codebase that are not being tested. Requiring a specific target of code coverage (e.g. 85%) on the management level is discouraged. Strict coverage target incites developers to create low quality tests in order to reach it. The worst example of this is an assertion-free test (also known as the happy path test) which executes the code, but never verifies any of its behaviour with assertions. (Fowler 2004a; Fowler 2012; Koskela 2013, 124; Marick 1997.)

Significantly low code coverage can still hint at problems with the code and be addressed (Fowler 2012). However, having a small number of reliable tests is better than a large number of unreliable ones. Developers should be focused on writing tests that genuinely verify the required behaviour of the software. Testing things that are unlikely to break only to reach the coverage target is a distraction from that goal. (Fowler 2012; Kim et al. 2016, 136.) According to Fowler (2012), the two ways more reliable than coverage reports to tell whether there are enough tests for the software are the rarity of bugs discovered in production environments and the confidence that developers possess to modify existing code without breaking it.

2.5 Behaviour-driven development and spec-style testing

The previous section introduced the concept of TDD. Tests were viewed as an “executable specification” of the software. This perspective characterizes a development approach derived from TDD – behaviour-driven development (BDD). BDD emphasizes the behaviour of the software by stressing the importance of keeping the tests expressive (Marston & Dees 2017). BDD was first introduced by Dan North in 2006 as a means to improve the language of TDD (North 2006; Justice 2018). In particular, the word “test” was replaced with the word “behaviour”. According to North, this change in terminology provided a better way of explaining TDD to beginners, as it gave them guidance on how to name tests and where to start testing from. (North 2006.)

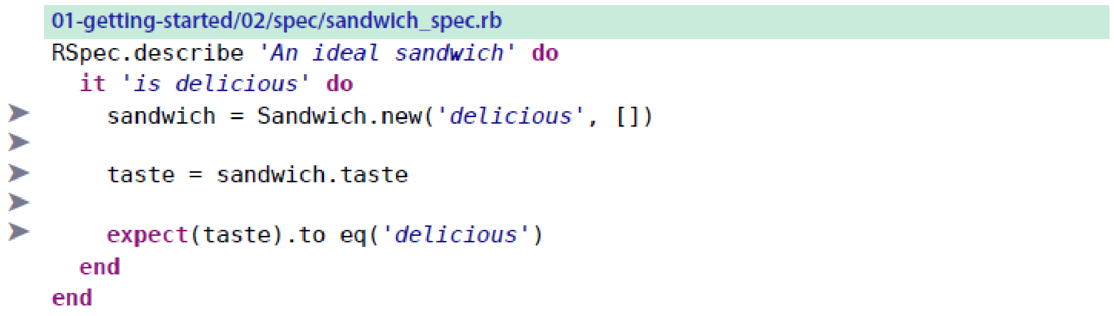
BDD advocates for using a single full sentence for each test method name. Using full sentences allows producing software documentation that makes sense not only to developers, but also to project managers, analysts, testers and other business users. This way BDD can provide living up-to-date documentation for the software. Automated tests are checked into version control together with the code of the application, and developers who want to understand how the system works can read the test suite as an example of how to use the API of the system. Sticking to one sentence per test name makes tests more focused, as only a relatively small piece of behaviour can be described in a sentence. Sentence names are also helpful in cases when tests fail – the desired behaviour is clear immediately. (Kim et al. 2016; North 2006; North 2012.)

According to North (2006), in BDD a failing test means one of the following:

- There is a bug. Test works as expected. The bug should be fixed.
- Specified behaviour is not correct anymore. Requirements for the SUT changed. The test should be deleted.
- Specified behaviour has been moved elsewhere. The test should be moved as well and possibly modified.

Example-driven specification-of-behaviour oriented ideas of North have been widely adopted by the software development community and beyond. The technique is used by business analysts and requirement teams responsible for producing specifications for the software. (Koskela 2013, 13.)

One of the first testing frameworks based on BDD is RSpec for Ruby. RSpec adopted the key BDD concept of driving software design by expressive tests, while adding its own new conventions (Marston & Dees 2017; Meszaros 2007). RSpec uses three terms to refer to tests: test, spec and example. These terms have different emphasis. A test verifies that a piece of code is working correctly. A spec describes the desired behaviour of the piece of code. An example demonstrates how a particular API is intended to be used. (Marston & Dees 2017.) Figure 4 demonstrates a simple test written in RSpec according to BDD.


01-getting-started/02/spec/sandwich_spec.rb

```
RSpec.describe 'An ideal sandwich' do
  it 'is delicious' do
    sandwich = Sandwich.new('delicious', [])

    taste = sandwich.taste

    expect(taste).to eq('delicious')
  end
end
```

Figure 4. Simple RSpec test using BDD and spec-style (Marston & Dees 2017)

RSpec introduced the describe/it convention for naming tests and organizing them into test suites. The commonly used method name “test” is replaced with “it”, forcing developers to think in terms of the behaviour of the SUT – what “it” should do or what “it” should be like. The describe/it convention is often referred to as spec-style. Spec-style has since been adopted by JavaScript testing libraries Mocha, Jasmine and Jest. (Justice 2018.)

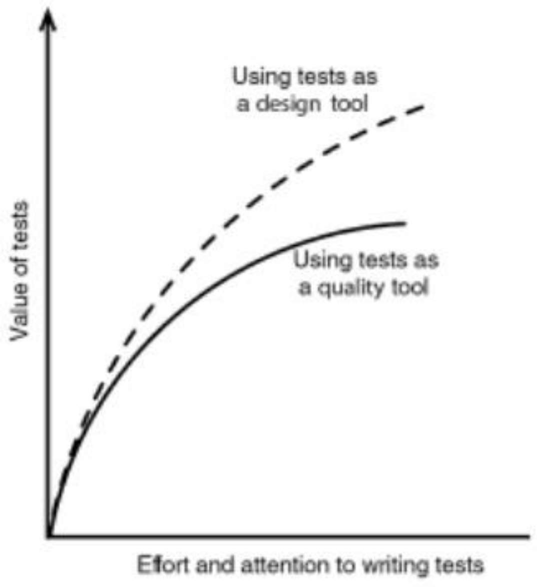


Figure 5. The advantage of tests as a design tool (Koskela 2013, 7)

The single sentence test name convention of BDD and spec-style align well with the goals defined in Section 2.2. TDD and especially BDD allow going beyond using tests as a tool for quality assurance. Tests are instead perceived as a design tool that guides the development process of the software. Tests that are used as a design tool allow reaping greater benefits from test automation. (Koskela 2013, 78.) Figure 5 shows the advantage the tests as design tools have over tests as quality tools. Even though both functions reach a plateau discussed in Section 2.3 (Figure 2), the design tool approach provides an advantage.

2.6 Test doubles

Software under test consists of many units. Each unit has a specified behaviour. Some units are contained within a single class or function. However, most units require interaction with other units. When testing a particular unit, this reliance on other units (referred to as collaborators) can be problematic – either the collaborator is not available, or the collaborator returns results not suitable for the test, or executing the collaborator has unnecessary side effects. In these situations, the collaborators can be replaced in order to gain full control of the environment the unit under test is running in. A test double is a tool used to isolate a unit from its collaborators in order to test it. (Koskela 2013, 28; Meszaros 2007.)

Aside from isolating the code under test, Koskela (2013, 27–33) describes four reasons to use a test double in a test:

- Speed up execution of tests. Test doubles help speed up a test in situations when the test is relying on a slow collaborator. For instance, a call to an asynchronous function or a complex database query can be replaced with a predefined response returned almost immediately.
- Make execution deterministic. A typical example of nondeterministic behaviour is code that relies on current timestamps. Such collaborators would return a different value every time the test is executed, leading to unreliable tests. Test doubles allow replacing such nondeterministic collaborators with controlled values.
- Simulate specific conditions. By simulating specific conditions with a test double, it is possible to test every edge case that would be hard to reproduce with a real collaborator. For example, a test verifying that a unit displays an error when the network connection is lost can be executed using a test double, without having to disconnect from the network to test this condition. As established earlier in Section 2.2, automated tests should require no manual intervention to run.
- Access hidden information. Test doubles allow accessing hidden information, such as private fields of a class or the details of how a certain function was called (how many times, with which parameters).

Koskela (2013, 33–39) and Meszaros (2007) mention five types of test doubles:

- Dummy object – a placeholder passed to the SUT as a parameter or attribute, but never used. Usually only used to fill parameter lists. (Fowler 2006; Meszaros 2007.)
- Stub – a simplest possible implementation of a collaborator. Calls to the stub return predefined values. Typically defined ad hoc inside of a test. (Koskela 2013, 34–35.)
- Fake object – a more elaborate version of a stub. An optimized version of a collaborator that removes undesirable side-effects. Can be reused in different tests. (Koskela 2013, 35.)
- Test spy – a test double that stores information about the way it was called by the SUT (with which parameters, how many times etc.). Created to record what happens to it in order to verify that the unit under test behaves according to the specification. (Koskela 2013, 36–38; Meszaros 2007.)
- Mock object – the most elaborate variation of a test double. A spy that has pre-defined behaviour and has pre-defined expectations of the calls it should receive from the unit under test. (Fowler 2007x; Koskela 2013, 38)

Setup of test doubles is typically the first step in a test. An established pattern for structuring unit tests is arrange-act-assert. First, the objects required by the test are set up. Then, the behaviour under test is executed. Lastly, assertions are made about the result of the execution. The same pattern is also described as a

Four-Phase test. The four steps are setup, exercise, verify and teardown. The only addition to arrange-act-assert is the teardown step where any of the test doubles that may affect later tests are reset to their original state. The teardown step is omitted from arrange-act-assert as it is not always needed, depending on the testing environment or specific test double used. (Fowler 2013x; Koskela 2013, 40–41; Meszaros 2007.)

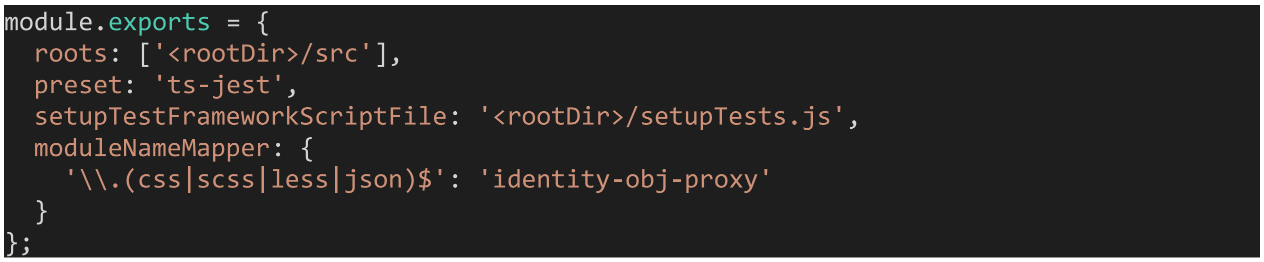
3 TESTING TOOLS AND SUT TECHNOLOGIES

This chapter introduces the technologies comprising the system under test (SUT) and the tools used for testing the SUT. Jest testing framework is introduced, and the essential features of the framework are presented in detail. This is followed by the introduction of the React library. Core concepts and features of React are discussed. Afterwards, the TypeScript language is introduced, and the ways in which TypeScript augments the development of React applications is explained. In the next chapter, the Redux state management library is presented. Section 3.5 discusses the Enzyme testing utility as well as shallow rendering React components with Enzyme and snapshot testing React components with Enzyme and Jest. Finally, Section 3.6 describes the tools and the workflow relevant to the deployment pipeline of the case study application, as well as ways in which automated tests can be integrated into that pipeline.

3.1 Jest

Jest is an open-source JavaScript testing framework created and maintained by Facebook. It is one of many JavaScript testing frameworks, alongside Mocha, Jasmine, Karma and others. In the annual State of JavaScript survey for 2018 where over 20,000 developers were polled, Jest was rated most favourably compared to its competitors in terms of satisfaction ratio over total usage, winning the “Highest Satisfaction” award. 96.5% of the users of the Jest library stated that they would like to use it again. (Greif et al. 2018a; Greif et al. 2018c; Greif et al. 2018d.) The most liked aspects of the framework mentioned in the survey include good documentation, facilitation of elegant programming style and patterns, easy learning curve, fast performance, as well as being a well-established testing option backed by a good developer team/company (Greif et al. 2018e).

Jest is a test runner, an assertion library and a mocking library. Jest also provides a built-in test coverage tool and a feature called snapshot testing. In contrast with some of its predecessors such as Jasmine, Jest does not require a real browser environment to run. Instead, it runs inside of a Node.js process where the browser APIs are emulated with jsdom. A simulated browser environment speeds up the execution of tests and allows to run tests on different systems with the same results. The same tests can be executed on a developer’s machine and on a continuous delivery server. (Burnham 2019.)



```
module.exports = {
  roots: ['<rootDir>/src'],
  preset: 'ts-jest',
  setupTestFrameworkScriptFile: '<rootDir>/setupTests.js',
  moduleNameMapper: {
    '\\.(css|scss|less|json)$': 'identity-obj-proxy'
  }
};
```

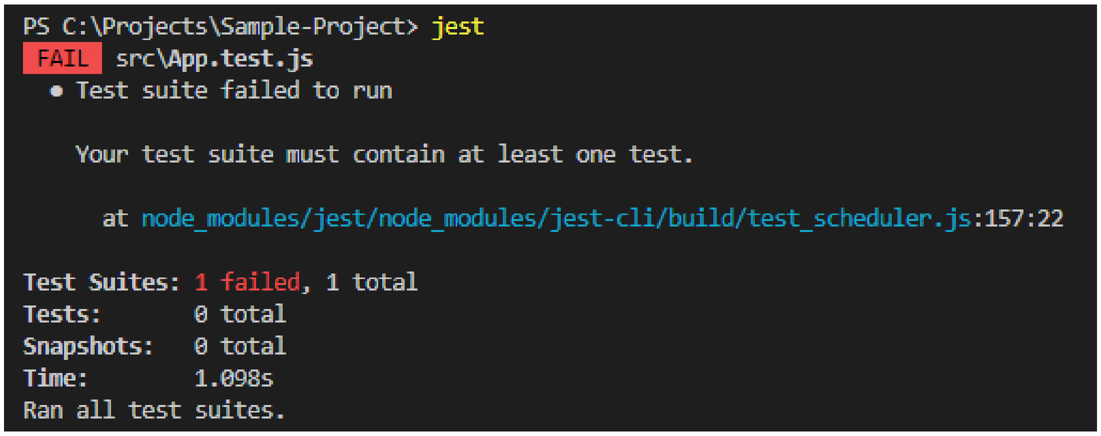
Figure 6. Example of a Jest configuration file

Jest configuration for a project can be provided either inside of the package.json file for that project, or in a separate configuration file titled jest.config.js and located at the root of the project. Figure 6 shows an example of a configuration file for Jest. Jest can be used without a configuration file with default options, but more complex applications require manual fine-tuning.

3.1.1 Test runner

As a test runner, Jest finds and executes tests. More specifically, it looks for test files in a given project and executes the tests inside those files. By default, Jest checks every file in the root directory of the project. A precise list of folders to check can be provided as the `roots` property of the Jest configuration file. By default, Jest interprets as test files every .js, .ts, .jsx and .tsx file inside of folders called __tests__, as well as any file containing a .test or .spec suffix. Files called spec.js and test.js will also be detected automatically. The glob pattern that Jest uses to discover test files can be overridden from the `testMatch` property of the Jest configuration file. (Jest API Reference 2019a.)

Jest CLI package is a command line runner used to execute tests with Jest. The `jest` command is used to execute tests with the options specified in the Jest configuration file, or with the default configuration options if no configuration file is provided. (Jest API Reference 2019c.) Figure 7 shows what the output of this command can look like. In this case Jest found one test suite. Jest failed to execute the test suite because it did not contain any tests.



```
PS C:\Projects\Sample-Project> jest
 FAIL  src\App.test.js
  ● Test suite failed to run

    Your test suite must contain at least one test.

      at node_modules/jest/node_modules/jest-cli/build/test_scheduler.js:157:22

Test Suites: 1 failed, 1 total
Tests:       0 total
Snapshots:   0 total
Time:        1.098s
Ran all test suites.
```

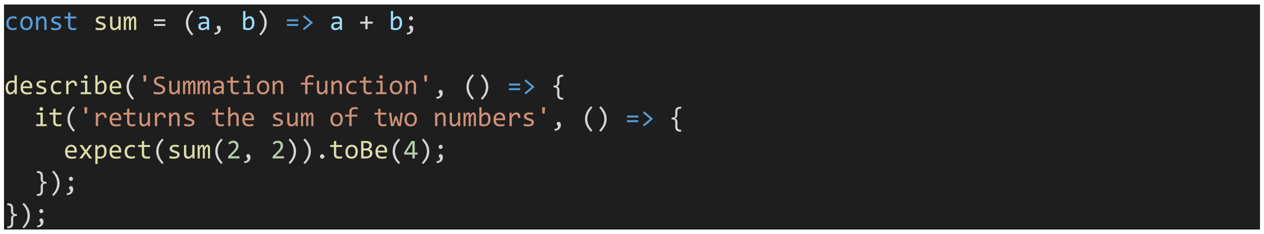
Figure 7. Output of the `jest` command when one empty test suite was found

The `jest` command accepts multiple options that specify how the tests should be executed. These options can be combined for maximum precision. Each command line option can also be specified in the Jest configuration. Command line options can be useful to override or extend the configuration options for a single test run. Jest CLI options include `--watch` and `--watchAll`, used to automatically re-run tests when a file is modified. `--watch` re-runs only the tests related to modified files, while `--watchAll` re-executes all tests every time any file is changed. Running Jest with these options is referred to as running Jest in watch mode. Another CLI option is `--testNamePattern` that can be shortened to `--t`. This option re-runs the tests with names matching a provided pattern. This option can be used to execute a specific test or group of tests. (Jest API Reference 2019c.)

Jest runs each test in a separate process, providing isolation and speeding up the execution. To further speed up developer feedback, Jest executes previously failed tests first and re-organizes the order in which tests are run based on how long each test takes. (Jest 2019.)

3.1.2 Assertion library

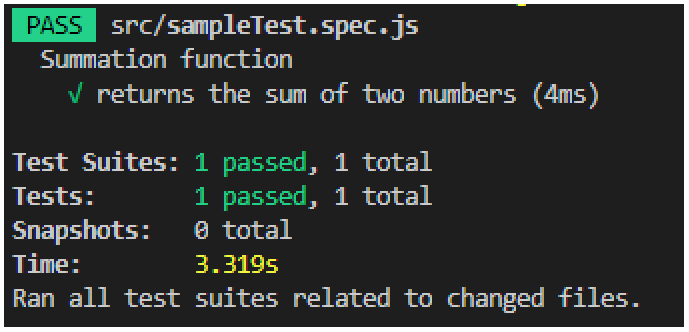
As an assertion library, Jest provides a set of tools to write tests in a simple, human-readable way. These tools are functions called matchers. Matchers allow testing values in different ways. In order to test a specific value, an expectation object needs to be created. Then, a matcher is called on the expectation object. Figure 8 shows an example of a matcher being used. The describe() function defines a test suite, which is a group of tests. The first argument is the name of the describe block, while the second is a function that contains one or multiple tests. The it() function defines a test. The first argument is the name of the test, the second argument is a function containing the code of the test. Finally, the expect() function returns an expectation object. It takes a value as an argument (commonly a value generated by a piece of code under test) and returns an object that exposes a set of matchers. The toBe() matcher verifies strict equality between the argument of expect() and the argument of its own. (Burnham 2019; Jest API Reference 2019b; Jest Documentation 2019d.)



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

describe('Summation function', () => {
  it('returns the sum of two numbers', () => {
    expect(sum(2, 2)).toBe(4);
  });
});
```

Figure 8. Jest test suite with a single test

Notice the way in which the name of the test suite and the name of the test form a full sentence: “Summation function returns the sum of two numbers”. The required behaviour of the function is clear from reading the test suite. Figure 9 displays the output that Jest prints to the console after the test suite from Figure 8 is executed.



```
PASS  src/sampleTest.spec.js
  Summation function
    √ returns the sum of two numbers (4ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        3.319s
Ran all test suites related to changed files.
```

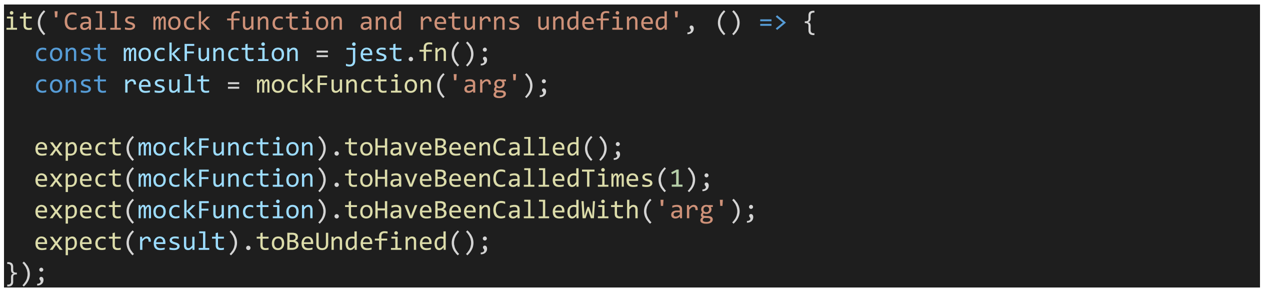
Figure 9. Jest console output

Jest provides multiple matchers, such as toEqual() for deep object comparison, toBeNull(), toBeDefined(), toHaveReturned(), toMatchObject() and many more specified in the documentation (Jest API Reference 2019b.) These matchers allow creating highly customizable tests and will be used extensively throughout this study.

3.1.3 Mocking library

As a mocking library, Jest provides a tool called mock functions to create test doubles. Mock functions allow replacing dependencies inside of a unit test with something that can be controlled and inspected. This includes erasing and/or replacing the actual implementation of a function or a module inside of a test, setting a return value for a function, capturing function calls and parameters passed in those calls. With Jest, mock functions can be created in two ways: either a mock function is created within a test, or a manual mock is set up to substitute a module dependency. (Hanlon 2018; Jest Documentation 2019a.)

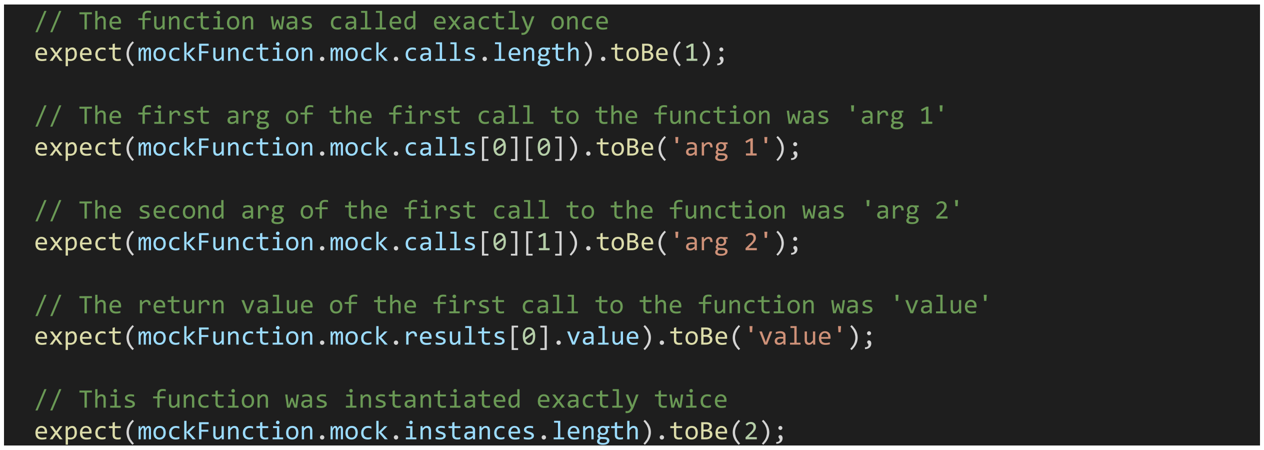
A mock function can be created by calling `jest.fn()`. Figure 10 shows a very simple case of a mock function being used. This test is impractical but allows to introduce the behaviour of a mock function. In the example, a mock function called `mockFunction` is created and called with a string argument `‘arg’`. Three assertions are then made about the mock function and one assertion about the result. Jest captures calls to the function and can verify that the function was called, as well as how many times it was called and what arguments were supplied. The mock function has no implementation, so the returned value `result` is undefined. (Hanlon 2018; Jest Documentation 2019a.)



```
it('Calls mock function and returns undefined', () => {
  const mockFunction = jest.fn();
  const result = mockFunction('arg');

  expect(mockFunction).toHaveBeenCalled();
  expect(mockFunction).toHaveBeenCalledTimes(1);
  expect(mockFunction).toHaveBeenCalledWith('arg');
  expect(result).toBeUndefined();
});
```

Figure 10. Simple usage of a mock function

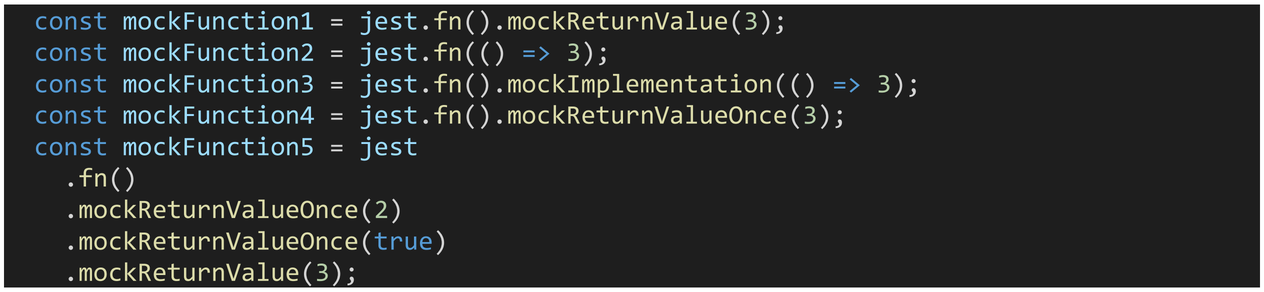
Each mock function has a `mock` property. That property stores the information about how the function was called and what it returned. It also stores the value of the “this” keyword, which is the context in which the function was called. Calls to toHaveBeenCalled() access the mock property of the mock function under the hood, but that property can also be accessed directly. (Jest Documentation 2019a.) Figure 11 shows how the mock function can be inspected in closer detail.



```
  // The function was called exactly once
  expect(mockFunction.mock.calls.length).toBe(1);

  // The first arg of the first call to the function was 'arg 1'
  expect(mockFunction.mock.calls[0][0]).toBe('arg 1');

  // The second arg of the first call to the function was 'arg 2'
  expect(mockFunction.mock.calls[0][1]).toBe('arg 2');

  // The return value of the first call to the function was 'value'
  expect(mockFunction.mock.results[0].value).toBe('value');

  // This function was instantiated exactly twice
  expect(mockFunction.mock.instances.length).toBe(2);
```

Figure 11. Accessing the mock property of the mock function to run assertions (Jest Documentation 2019a)

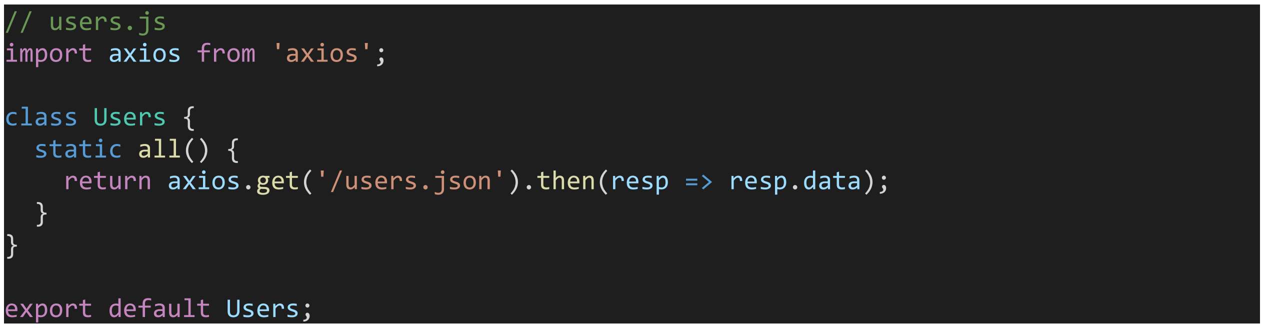
It is possible to mock the implementation of a function or just a returned value of a function. Figure 12 demonstrates how this can be achieved. The function `mockFunction1` mocks only the value returned by the function. The functions `mockFunction2` and `mockFunction3` demonstrate two different ways to mock the implementation of a function. The mock implementation can either be supplied as an argument to `jest.fn()` or to `jest.fn().mockImplementation()`. It is possible to mock a returned value or implementation of a function only once by calling `mockReturnedValueOnce()`. The function `mockFunction4` returns “3” the first time it is called and returns “undefined” after each following call. It is possible to chain the calls to `mockReturnedValueOnce()` so that each following call returns a different value. (Jest Documentation 2019a.) The function `mockFunction5` returns “2” the first time it is called, “true” the second time and “3” after each call that follows. It is also possible to mock the implementation of a function once using `mockImplementationOnce()`. Calls to `mockImplementationOnce()` can be chained the same way.



```
  const mockFunction1 = jest.fn().mockReturnValue(3);
  const mockFunction2 = jest.fn(() => 3);
  const mockFunction3 = jest.fn().mockImplementation(() => 3);
  const mockFunction4 = jest.fn().mockReturnValueOnce(3);
  const mockFunction5 = jest
    .fn()
    .mockReturnValueOnce(2)
    .mockReturnValueOnce(true)
    .mockReturnValue(3);
```

Figure 12. Mocking returned value and implementation of a function

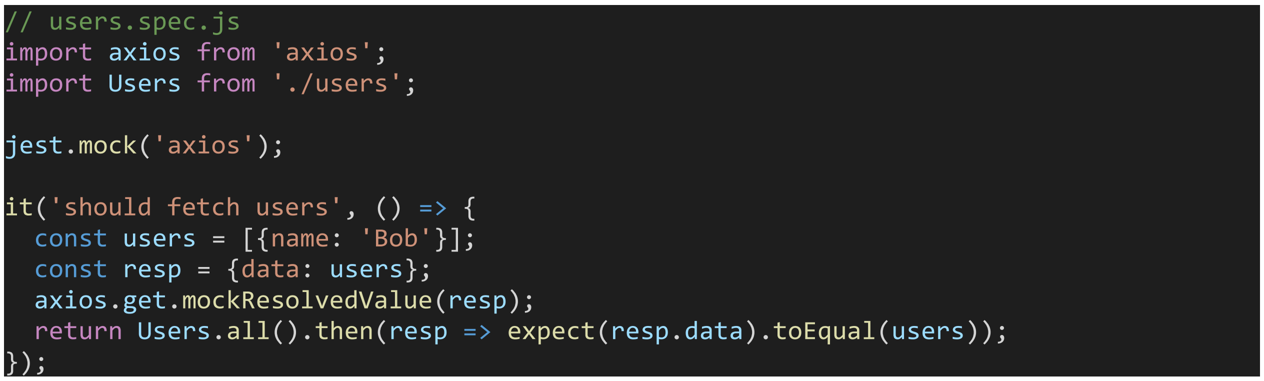
Jest allows to mock entire modules. This can either be done entirely within test code or in a separate file called manual mock. Consider the example of a unit under test in Figure 13. Class Users has a static method all() that makes an asynchronous request via the imported axios module.



```
// users.js
import axios from 'axios';

class Users {
  static all() {
    return axios.get('/users.json').then(resp => resp.data);
  }
}

export default Users;
```

Figure 13. Unit under test using a module to make an asynchronous call (Jest Documentation 2019a)

In order to avoid accessing a real API within the test to speed up test execution and prevent undesirable side effects, the axios module can be replaced with a mock function. Figure 14 displays an example of a test for the class in Figure 13. First, the real axios module is imported. Then, the module is mocked by calling `jest.mock(‘axios’)`. At this point the module is replaced with a mock function the returned value of which can be mocked. Jest allows to mock a resolved value of a promise using the `mockResolvedValue()` function. The `get()` method of axios is replaced with a mock function returning a canned response that does not require making a request to the real API. Just like with any mock function, it is also possible to mock the implementation of the module methods by calling `mockImplementation()`. (Jest Documentation 2019a.)



```
// users.spec.js
import axios from 'axios';
import Users from './users';

jest.mock('axios');

it('should fetch users', () => {
  const users = [{name: 'Bob'}];
  const resp = {data: users};
  axios.get.mockResolvedValue(resp);
  return Users.all().then(resp => expect(resp.data).toEqual(users));
});
```

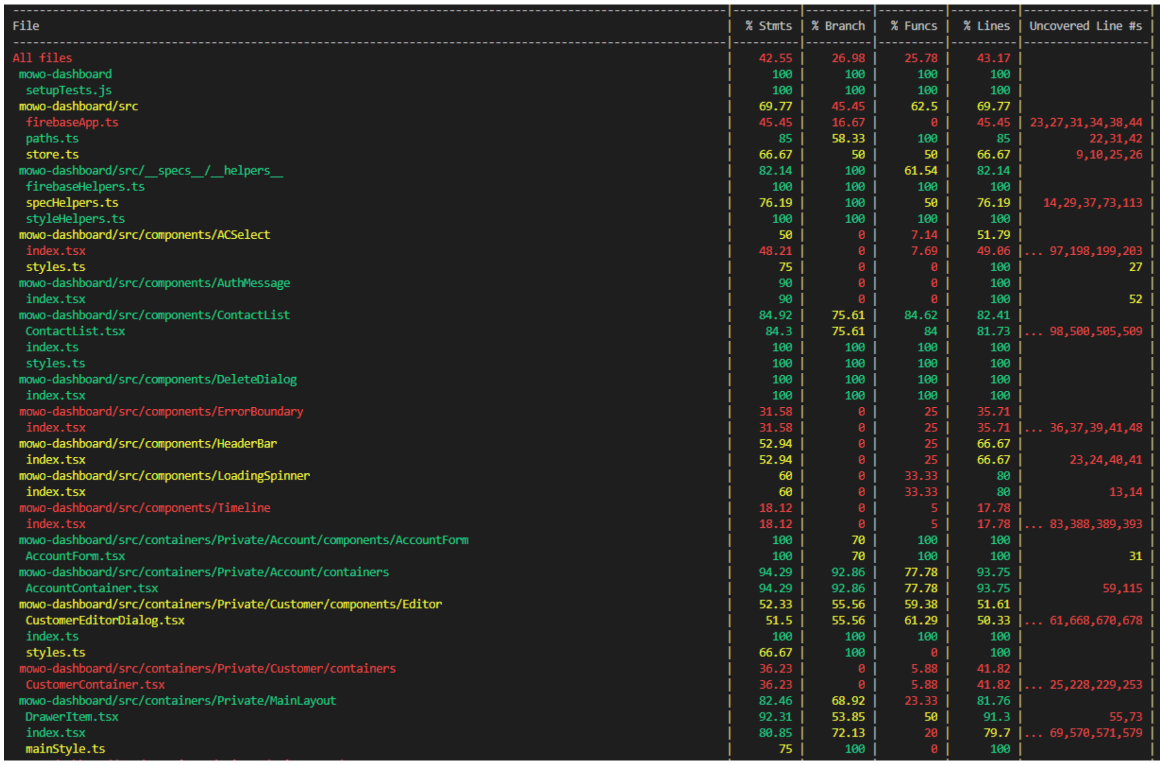
Figure 14. Test using jest.mock() to mock an imported module (Jest Documentation 2019a)

The alternative to creating mock functions directly within test code is manual mocks. Manual mocks are used to replace the functionality of collaborator modules with test doubles. Both custom user modules and Node modules can be manually mocked. In order to mock a user module, a __mocks__ subdirectory must be created in the directory containing the module. A file with the same name as the mocked module must be placed into the __mocks__ subdirectory. To replace the module with the manual mock within the test, the line `jest.mock('./name_of_module')` must be added to the test. In order to mock a Node module, the manual mock must be placed in the __mocks__ directory located in one of the root directories specified by the `roots` property of the Jest configuration. If the roots option is not specified, the __mocks__ folder must be placed in the same directory as the node_modules folder. Explicit call to `jest.mock('./name_of_module')` is not required for mocking Node modules. The Node module will be replaced with the mock automatically. (Jest Documentation 2019a.)

3.1.4 Built-in code coverage

Jest includes the Istanbul library for JavaScript code coverage. Istanbul provides a number of coverage reporters that allow to generate reports in different formats. Reporters can be specified using the `coverageReporters` option of the Jest configuration. The default value of this option is `["json", "lcov", "text", "clover"]`.

- Text option means the coverage report will be printed directly to the console. Figure15 shows an example of such a report. The rest of the options generate files with the code coverage and place it in the coverage directory. The default location of the coverage directory is at the root of the project, but it can be overridden using the `coverageDirectory` configuration option of Jest.
- Clover option generates a report in XML format.
- JSON option generates a report in JSON format.
- LCOV option generates an LCOV coverage file with an associated HTML report. (Jest API Reference 2019a; Istanbul Documentation 2019.)



```
----------------------------------------------------------------|---------|----------|---------|---------|-------------------|
File                                                            | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s |
----------------------------------------------------------------|---------|----------|---------|---------|-------------------|
All files                                                       |   42.55 |    26.98 |   25.78 |   43.17 |                   |
 mowo-dashboard                                                 |     100 |      100 |     100 |     100 |                   |
  setupTests.js                                                 |     100 |      100 |     100 |     100 |                   |
 mowo-dashboard/src                                             |   69.77 |    45.45 |    62.5 |   69.77 |                   |
  firebaseApp.ts                                                |   45.45 |    16.67 |       0 |   45.45 | 23,27,31,34,38,44 |
  paths.ts                                                      |      85 |    58.33 |     100 |      85 |          22,31,42 |
  store.ts                                                      |   66.67 |       50 |      50 |   66.67 |         9,10,25,26 |
 mowo-dashboard/src/__specs__/__helpers__                       |   82.14 |      100 |   61.54 |   82.14 |                   |
  firebaseHelpers.ts                                            |     100 |      100 |     100 |     100 |                   |
  specHelpers.ts                                                |   76.19 |      100 |      50 |   76.19 |  14,29,37,73,113 |
  styleHelpers.ts                                               |     100 |      100 |     100 |     100 |                   |
 mowo-dashboard/src/components/ACSelect                         |      50 |        0 |    7.14 |   51.79 |                   |
  index.tsx                                                     |   48.21 |        0 |    7.69 |   49.06 |... 97,198,199,203 |
  styles.ts                                                     |      75 |        0 |       0 |     100 |                27 |
 mowo-dashboard/src/components/AuthMessage                      |      90 |        0 |       0 |     100 |                   |
  index.tsx                                                     |      90 |        0 |       0 |     100 |                52 |
 mowo-dashboard/src/components/ContactList                      |   84.92 |    75.61 |   84.62 |   82.41 |                   |
  ContactList.tsx                                               |    84.3 |    75.61 |      84 |   81.73 |... 98,500,505,509 |
  index.ts                                                      |     100 |      100 |     100 |     100 |                   |
  styles.ts                                                     |     100 |      100 |     100 |     100 |                   |
 mowo-dashboard/src/components/DeleteDialog                     |     100 |      100 |     100 |     100 |                   |
  index.tsx                                                     |     100 |      100 |     100 |     100 |                   |
 mowo-dashboard/src/components/ErrorBoundary                    |   31.58 |        0 |      25 |   35.71 |                   |
  index.tsx                                                     |   31.58 |        0 |      25 |   35.71 |... 36,37,39,41,48 |
 mowo-dashboard/src/components/HeaderBar                        |   52.94 |        0 |      25 |   66.67 |                   |
  index.tsx                                                     |   52.94 |        0 |      25 |   66.67 |       23,24,40,41 |
 mowo-dashboard/src/components/LoadingSpinner                   |      60 |        0 |   33.33 |      80 |                   |
  index.tsx                                                     |      60 |        0 |   33.33 |      80 |             13,14 |
 mowo-dashboard/src/components/Timeline                         |   18.12 |        0 |       5 |   17.78 |                   |
  index.tsx                                                     |   18.12 |        0 |       5 |   17.78 |... 83,388,389,393 |
 mowo-dashboard/src/containers/Private/Account/components/AccountForm |     100 |       70 |     100 |     100 |                   |
  AccountForm.tsx                                               |     100 |       70 |     100 |     100 |                31 |
 mowo-dashboard/src/containers/Private/Account/containers       |   94.29 |    92.86 |   77.78 |   93.75 |                   |
  AccountContainer.tsx                                          |   94.29 |    92.86 |   77.78 |   93.75 |            59,115 |
 mowo-dashboard/src/containers/Private/Customer/components/Editor |   52.33 |    55.56 |   59.38 |   51.61 |                   |
  CustomerEditorDialog.tsx                                      |    51.5 |    55.56 |   61.29 |   50.33 |... 61,668,670,678 |
  index.ts                                                      |     100 |      100 |     100 |     100 |                   |
  styles.ts                                                     |   66.67 |      100 |       0 |     100 |                   |
 mowo-dashboard/src/containers/Private/Customer/containers      |   36.23 |        0 |    5.88 |   41.82 |                   |
  CustomerContainer.tsx                                         |   36.23 |        0 |    5.88 |   41.82 |... 25,228,229,253 |
 mowo-dashboard/src/containers/Private/MainLayout               |   82.46 |    68.92 |   23.33 |   81.76 |                   |
  DrawerItem.tsx                                                |   92.31 |    53.85 |      50 |    91.3 |             55,73 |
  index.tsx                                                     |   80.85 |    72.13 |      20 |    79.7 |... 69,570,571,579 |
  mainStyle.ts                                                  |      75 |      100 |       0 |     100 |                   |
```

Figure 15. Text coverage report generated with Istanbul and printed to the command line. The percentage in each column represents the fraction of executed code.

Jest produces statement, branches, function and line coverage numbers.

3.2 React

React is a JavaScript library and framework for building user interfaces. As of 2018, React is the most popular framework for front-end development (Greif et al. 2018b). Tyler McGinnis, a prominent educator of React, lists the following biggest benefits of React: its composition, unidirectional data flow, declarative approach and explicit mutations (McGinnis 2017). Dan Abramov, one of the members of the React team at Facebook and creator of Redux, mentions three of these strengths as well (Abramov 2015). These aspects provide a solid overview of what React is.

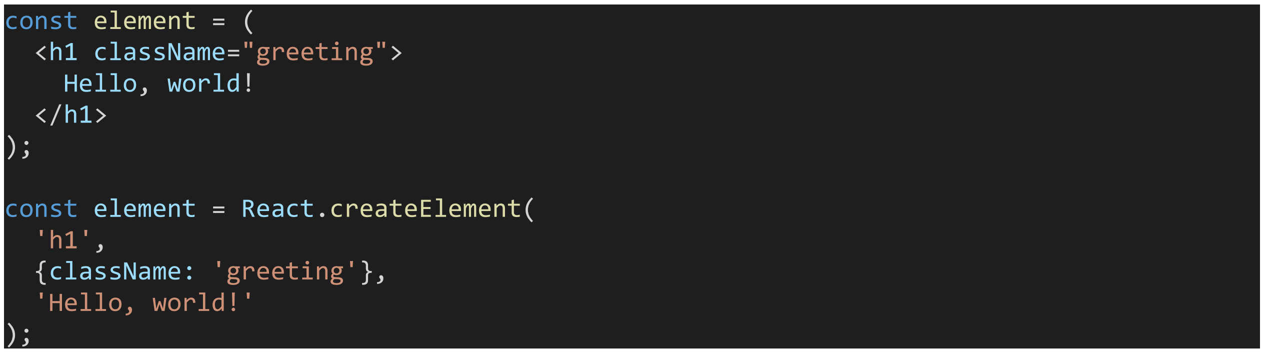
- **Composition.** React applications are built from components. Large complex applications can be constructed from small reusable pieces. (React API Reference 2019a; React Documentation 2019d.) A component is a function or a class that optionally accepts input and returns a React element. A React element is an object representation of a DOM node. (McGinnis 2016.) React elements can be thought of as descriptions of what will be displayed on the screen. React constructs the DOM from these objects and keeps it updated. (React Documentation 2019f.) React resolves what the user interface of an application should look like based on the state of the application. State is a central concept in React. The state consists of component-specific data that might be changed over time. The state is defined by the user and it is a plain JavaScript object. (React API Reference 2019a; React Documentation 2019b.)
- **Unidirectional data flow.** Also referred to as the “top-down” data flow, this is the term for the way state is shared between components in a React application. Any state is always owned by a specific component. This state can only be passed to child components of this parent component. In other words, changes to the state of a component can only affect components “below” that component in the DOM tree. (React Documentation 2019g.) React components share their state with their children via an object called props (short for “properties”). One of the few strict rules in React is that every component must act as a pure function with respect to its props, meaning a component cannot modify the props that it receives. React component state is similar to props, but the state is private and fully controlled by the component. While components cannot modify the props they receive, they do have full control over their state. Some components have no state of their own and only receive props. Such components are called stateless, as opposed to stateful. Parent components have no knowledge of whether their children are stateful or stateless, and vice versa. (React Documentation 2019a; React Documentation 2019g.)
- **Declarative approach.** React minimizes the amount of changes to the real DOM. Instead, it provides an abstraction called the Virtual DOM. The Virtual DOM stores the virtual representation of the DOM in memory and synchronizes the actual DOM to match it (the process referred to as reconciliation). This feature allows to define the user interface of an

application in a declarative way. React receives the state of the application and makes sure the actual DOM displays that state. (React Documentation 2019c.) Instead of describing how the result needs to be achieved step by step (the imperative approach), it is enough to specify what the result needs to look like and let React handle the rest. Applications built with jQuery are a good example of the imperative approach to building user interfaces. With jQuery, attributes and events of the actual DOM are accessed and manipulated directly. Compared to the imperative approach, the declarative method reduces side effects, minimizes the amount of state mutations and results in more readable code with fewer bugs. (McGinnis 2017.)

- **Explicit mutations.** React components provide a setState() API that is used to update component state. When a component calls setState(), setState() puts the specified changes into a queue and lets React know that this component and the children of this component need to be rendered again with the modified state. The changes to state are not applied immediately, and multiple calls to setState() can be grouped together to optimize performance. The explicit calls to setState() make it clear when and how the user interface is updated. (React Documentation 2019g; McGinnis 2017.)

3.2.1 JSX

React provides a syntax extension of JavaScript called JSX, visually similar to a template language such as HTML or XML. JSX expressions are compiled into JavaScript function calls that evaluate JavaScript objects – React elements. (React Documentation 2019e.) Figure 16 shows two identical ways to declare a React element.



```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);

const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

Figure 16. React element declaration with and without JSX (React Documentation 2019e)

The first declaration uses the JSX syntax and is internally compiled into the function call in the second declaration. It is possible to write React applications without JSX, but it is uncommon. JSX acts as visual aid when building user interfaces and allows React to display clearer warning and error messages. (React Documentation 2019e.)

3.2.2 Component lifecycle methods

React provides a number of lifecycle methods that are triggered at different stages during the process of adding a component to the DOM (the process referred to as “mounting”), updating the component in the DOM and removing (“unmounting”) it from the DOM. These methods allow for finer control over the behaviour of components throughout their lifecycle. (React API Reference 2019a; React Documentation 2019g.) Figure 17 shows an overview of the lifecycle of a React component and the most commonly used lifecycle methods for React version 16.4.

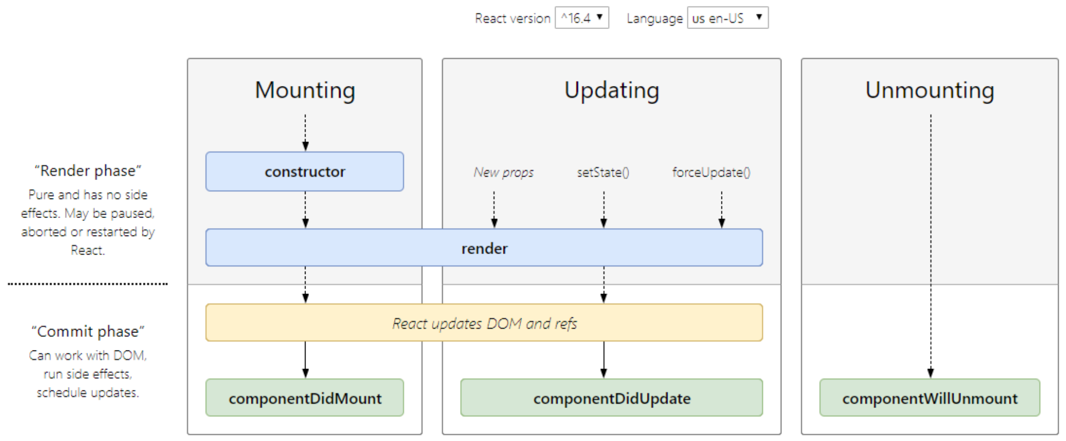


Figure 17. React component lifecycle and common lifecycle methods (React API Reference 2019a)

The following is the summary of the commonly used lifecycle methods:

- The method render() is the only required method when defining a React component as a class. The method is a pure function that returns a React element based on props and state.
- The method constructor() is used for initializing the state controlled by the component and for binding event methods to the instance of the component. The method is called before mounting the component.
- The method componentDidMount() is called immediately after the component is added to the DOM (mounted). Loading data from remote end points and adding subscriptions is typically done within this method.
- The method componentDidUpdate() is called every time state or props of the component are changed. The method receives previous state and previous props as arguments, and it is possible to compare them to the

current state and props and only execute certain logic if a particular piece of state or a prop is different from the previous value.
- Method componentWillUnmount() is called before the component is removed from the DOM. This method is the place to remove network requests or subscriptions set up in componentDidMount(). (React API Reference 2019a.)

It is important to understand the component lifecycle methods in order to create unit tests for React components. The methods componentDidMount() and componentWillUnmount() may require using test doubles in tests, as the subscriptions to remote data sources are typically added and removed in these methods.

3.3 TypeScript

TypeScript is a superset language of JavaScript created by Microsoft that addresses common issues that may arise when building large-scale applications in JavaScript. One of the core issues is the lack of strict types. In JavaScript, types of variables can change with each new assignment. Moreover, types of dynamically typed variables can be changed on the fly while executing an expression. To give an example, when concatenating a string with a number, the number is coerced into the string type. When a string or a number value is used in a logical operation, certain rules determine whether the value is converted into a true or a false boolean value. Such cases are referred to as “type juggling” and can introduce unexpected behaviour. The lack of strict types also hinders the development of software development tools. Strict types allow for improved autocompletion and features like type hinting. (Fenton 2014.)

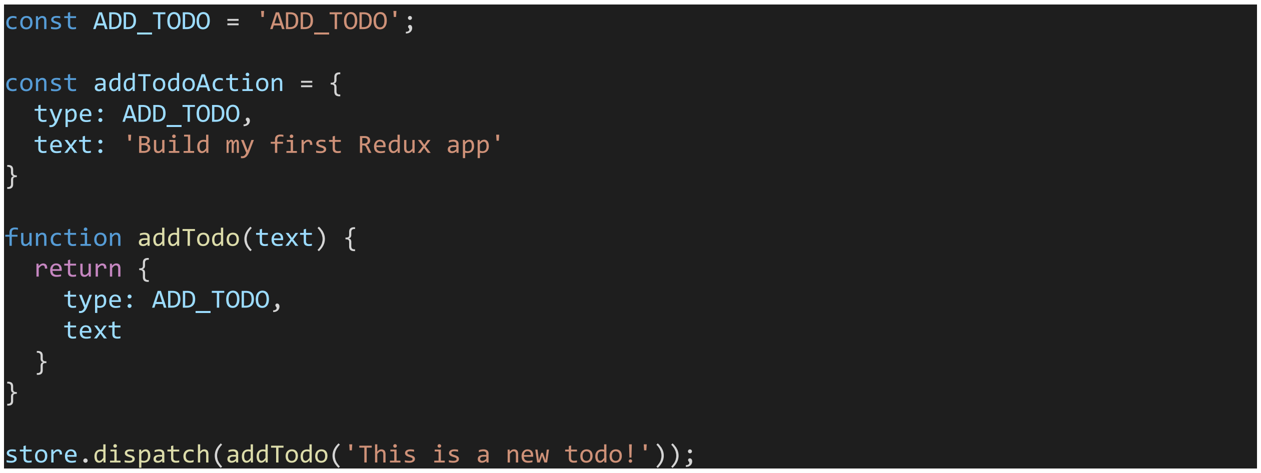
It is important to note that the use of strict types in an application does not eliminate the need for tests. A static type system does not verify that the system behaves according to its specification. It only ensures internal consistency of the source code of the system. A static type system can also strengthen the expressiveness and readability of the code. It eliminates certain errors caused by type ambiguity that can be missed by tests. However, every bit of the required behaviour of the system still has to be covered with tests. (Martin 2017.)

3.4 Redux

This section bases on the official Redux documentation (Abramov et al. 2018). Redux is a predictable state container for JavaScript applications. It is common for React applications to use Redux for state management, as the state is the central concept in both libraries. However, Redux is independent from React and can be used with non-React applications. Redux is a library that provides a centralized way to store the state of an application and propagate that state along the application as well as patterns for modifying the state. In Redux, the entire state of an application is kept as an object tree in a centralized store. The only way to update the state is to dispatch an action – a plain object specifying what kind of change needs to be applied to the state. Pure functions called reducers describe how the state needs to be modified when each action is dispatched.

A Redux action is a plain object that is used to send information to the store. An action is emitted using the `store.dispatch()` command. A common pattern to make new actions is to use action creators. An action creator is a function that optionally accepts input and returns an action. The action can then be dispatched to the store. Figure 18 shows an example of an action created with an action creator and dispatched to the store. The example uses a simple productivity application for maintaining a list of tasks to do.

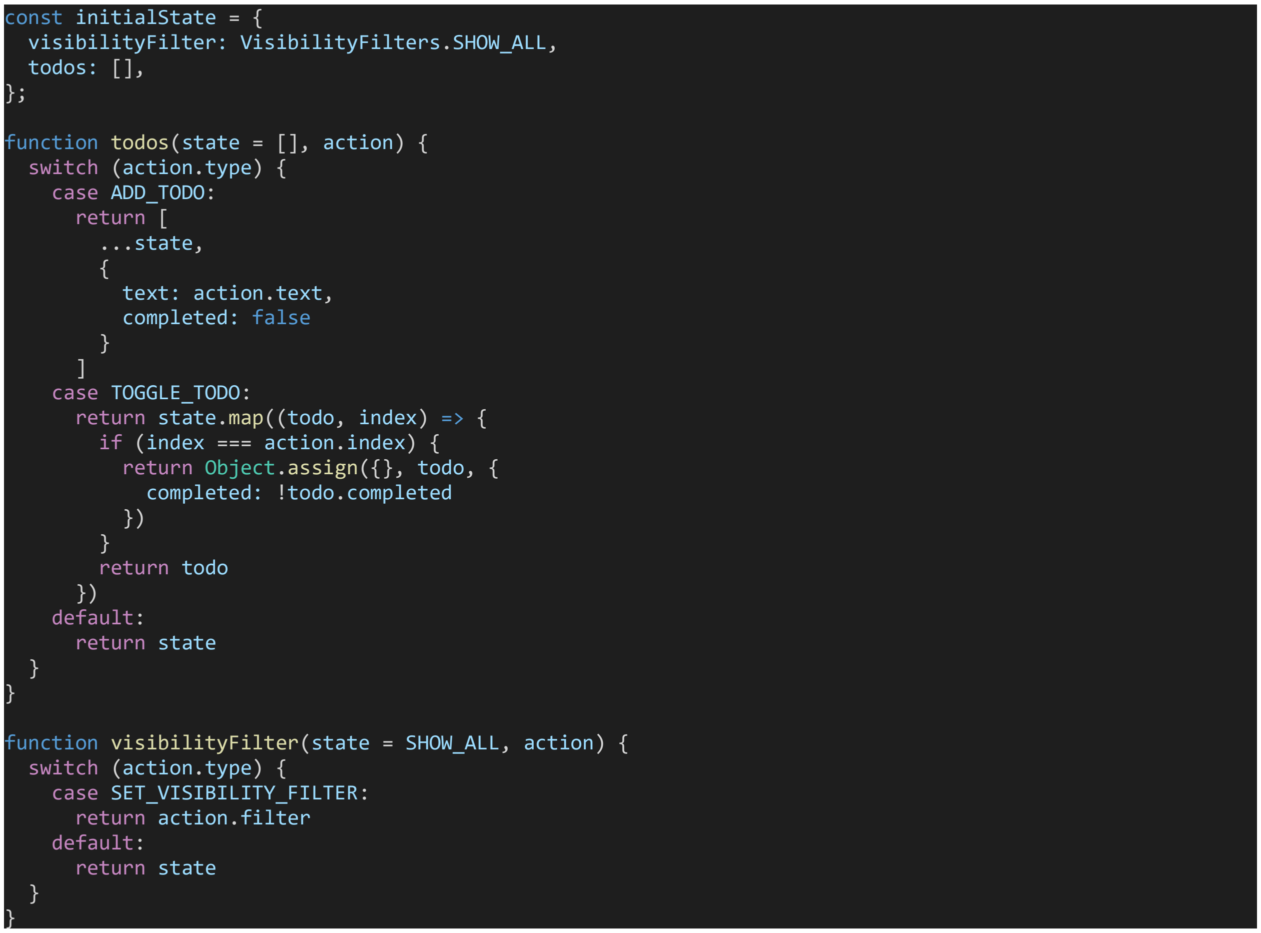
Each action must have a specified type, in the case of the example it is ‘ADD_TODO’. Reducers use the action type to determine how to modify the state. The recommended approach is to keep action types stored as constants, but this is unnecessary for smaller applications. Aside from the type, there are no restrictions on how an action should be structured. The constant addTodoAction is a plain object that contains the action type and the text for the new task as the payload. Function addTodo() is an action creator that accepts the text for a new task as an argument and returns an action with the ADD_TODO type and the supplied text. The action is then dispatched to the store.



```
const ADD_TODO = 'ADD_TODO';

const addTodoAction = {
  type: ADD_TODO,
  text: 'Build my first Redux app'
}

function addTodo(text) {
  return {
    type: ADD_TODO,
    text
  }
}

store.dispatch(addTodo('This is a new todo!'));
```

Figure 18. Action created with an action creator and dispatched to the store (Abramov et al. 2018)

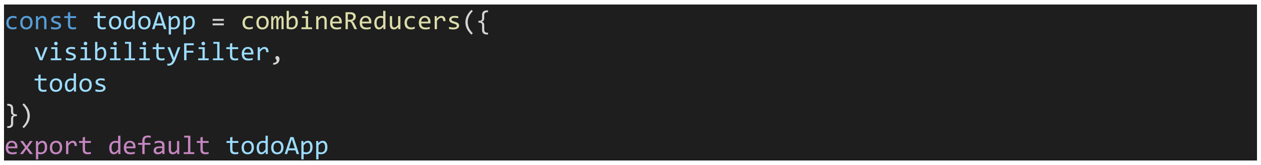
Reducers are pure functions describing how the state should change after receiving each action. Figure 19 shows an example of the default state of a task management application and a reducer responsible for modifying that state. The example uses a Redux pattern called reducer composition – separate reducers manage separate branches of the application state tree. The todos() reducer is responsible for the list of tasks. It can add new tasks to the list and mark existing tasks as completed or not completed. The visibilityFilter() reducer makes certain tasks hidden based on their completed or not completed status. Each reducer accepts a piece of current state and an action as arguments. Each reducer contains a switch statement based on action type. If the type of the supplied action matches any of the cases of the switch, the appropriate transformation is made to the state. If none of the cases are satisfied, the original state is returned. Reducers never modify the original state object supplied as the first argument, but rather take a copy of that object, make modifications to it and return it. The copy is most commonly taken using the object spread syntax (see …`state` in the Figure 19) but can also be made via the Object.assign() method.



```
const initialState = {
  visibilityFilter: VisibilityFilters.SHOW_ALL,
  todos: [],
};

function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case TOGGLE_TODO:
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: !todo.completed
          })
        }
        return todo
      })
    default:
      return state
  }
}

function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter
    default:
      return state
  }
}
```

Figure 19. Two reducers managing separate parts of the state tree (Abramov et al. 2018)

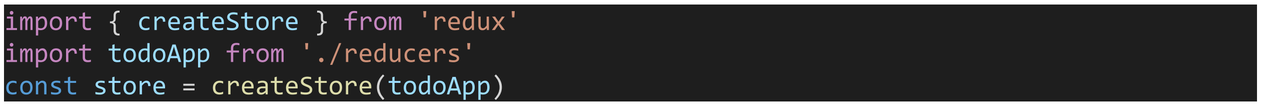
Separate reducers are then combined into a single root reducer using the combineReducers() method of Redux. As applications grow, individual reducers can be placed into separate files and directories, each responsible for a single branch of the state tree. Figure 20 shows the two reducers combined and exported.



```
const todoApp = combineReducers({
  visibilityFilter,
  todos
})
export default todoApp
```

Figure 20. Reducers combined into a root reducer (Abramov et al. 2018)

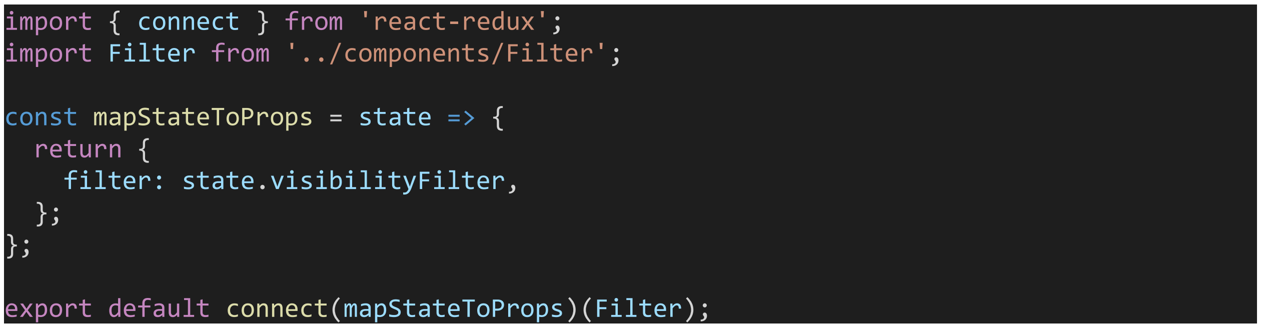
Redux store is an object that contains the full state tree of an application and connects actions to reducers. A Redux application must contain only one store. Figure 21 shows a store created from the root reducer using the createStore() method of Redux. Now, actions can be emitted using the dispatch() method of the store object. It is also possible to get the current state of an application by calling `store.getState()`.



```
import { createStore } from 'redux'
import todoApp from './reducers'
const store = createStore(todoApp)
```

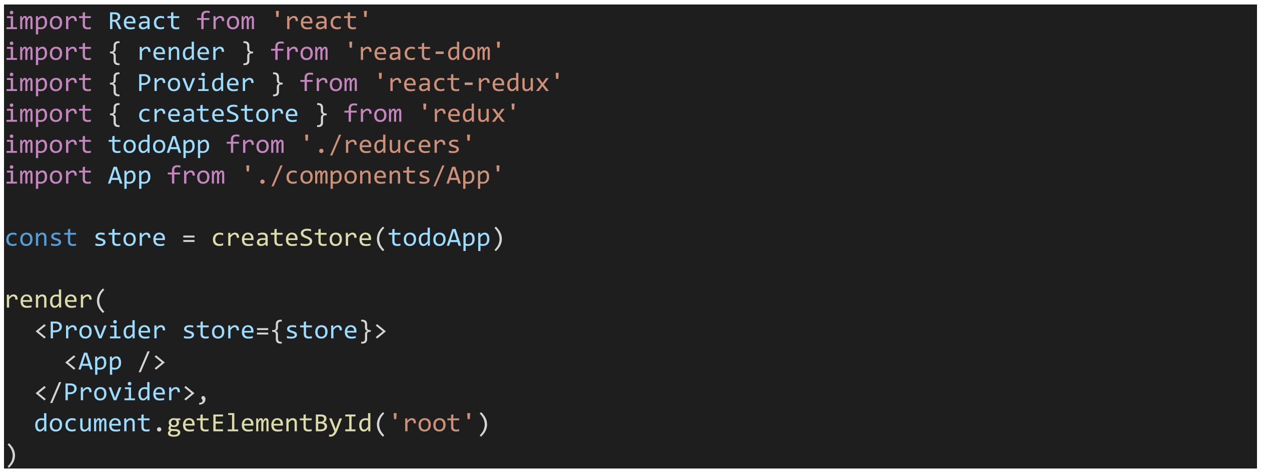
Figure 21. Redux store created from a root reducer (Abramov et al. 2018)

React-Redux is a library used to connect React components to the Redux store. To connect certain props of a component to the values in the global application state a mapStateToProps() function is required. The component is then connected to the store using the connect() function. Figure 22 shows how a Filter component maps the visibilityFilter property of the global state to its own filter prop.



```
import { connect } from 'react-redux';
import Filter from '../components/Filter';

const mapStateToProps = state => {
  return {
    filter: state.visibilityFilter,
  };
};

export default connect(mapStateToProps)(Filter);
```

Figure 22. Connecting a React component to the Redux store



```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import todoApp from './reducers'
import App from './components/App'

const store = createStore(todoApp)

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

Figure 23. Passing the store to the application using the Provider component (Abramov et al. 2018)

Finally, in order for components to be able to subscribe to the store in this way, the root component of the application must be wrapped into a Provider component. The provider component supplies the store to the application. Figure 23 shows the root component wrapped in a Provider.

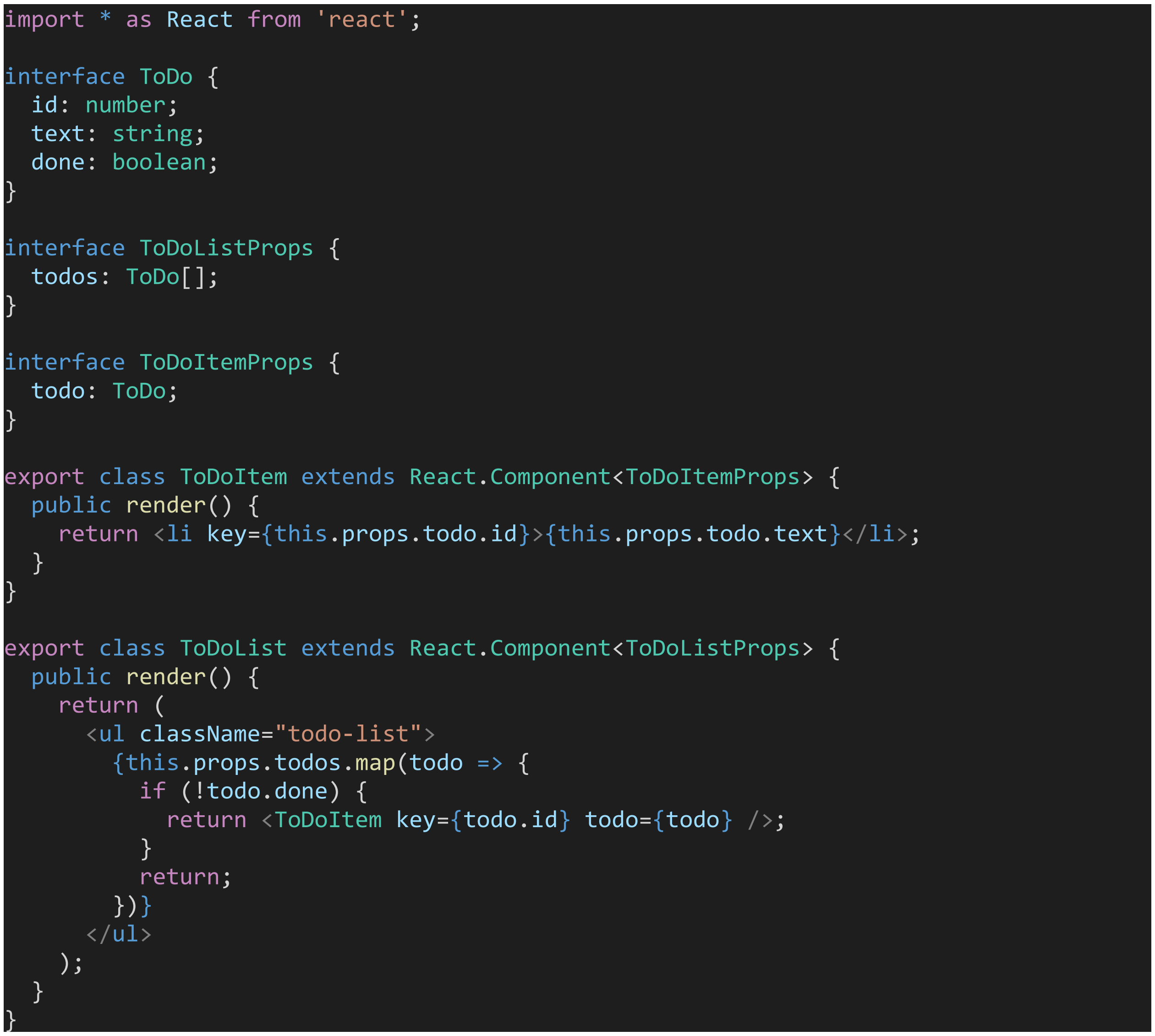
3.5 Shallow rendering with Enzyme

Enzyme is testing utility for React developed by Airbnb. It allows testing the output of the render() method of React components – React elements. Enzyme provides tools to manipulate and traverse the component output and simulate various events. Enzyme API imitates jQuery in the way it allows to query and manipulate elements. (Airbnb 2019.) At the time of this writing, Enzyme is the de facto standard library for testing React components (Burnham 2019).

Shallow rendering is a feature provided by the React development team as part of their react-test-renderer library. With shallow rendering a component can be rendered “one level deep”, and assertions can be made about the output of its render method. A DOM environment is not required to shallow render React components. Enzyme provides an improved higher-level API for shallow rendering, the Shallow Rendering API. (React API Reference 2019b.)

The shallow rendering feature of Enzyme allows to render a single component in isolation, modify the state and props of the component, simulate events and see how the render tree changes. Shallow rendering makes it possible to test each component as an individual unit, making sure no assertions are run on the behaviour of the children of the component. This way changes to the children of the component are less likely to break the tests of the parent. Shallow rendering only allows to see the props that the children receive but doesn’t render the output of the children recursively. (Abramov et al 2018; Airbnb 2019; Burnham 2019.)

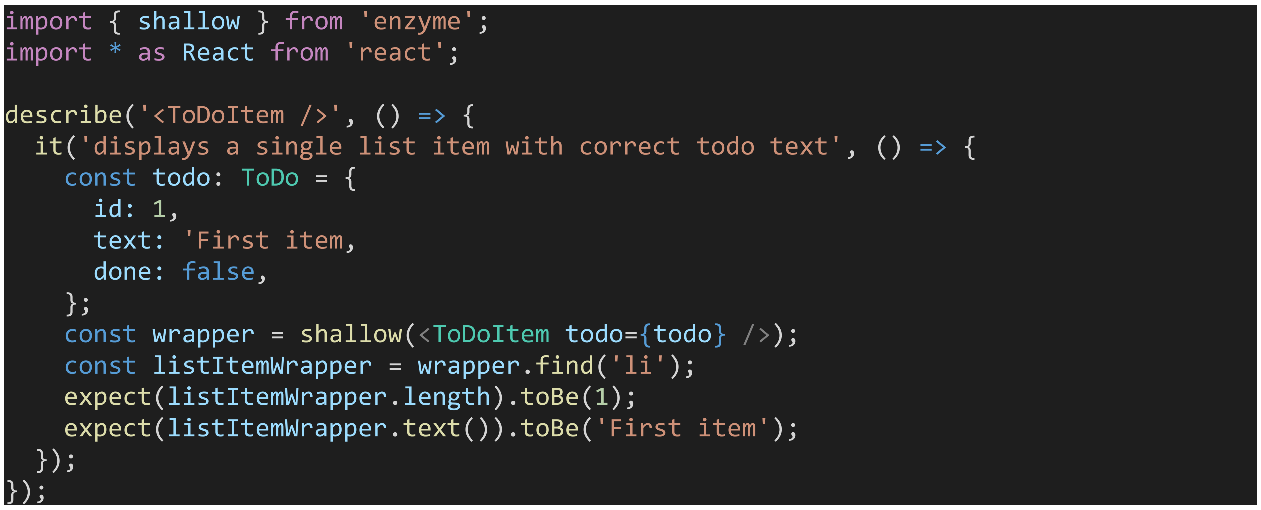
Figure 24 shows an example in TypeScript of a parent ToDoList component that renders a number of ToDoItem children. Normally these components would be declared in separate files, but here they are placed together for the sake of the example. ToDoList only renders the tasks that are not yet done.



```
import * as React from 'react';

interface ToDo {
  id: number;
  text: string;
  done: boolean;
}

interface ToDoListProps {
  todos: ToDo[];
}

interface ToDoItemProps {
  todo: ToDo;
}

export class ToDoItem extends React.Component<ToDoItemProps> {
  public render() {
    return <li key={this.props.todo.id}>{this.props.todo.text}</li>;
  }
}

export class ToDoList extends React.Component<ToDoListProps> {
  public render() {
    return (
      <ul className="todo-list">
        {this.props.todos.map(todo => {
          if (!todo.done) {
            return <ToDoItem key={todo.id} todo={todo} />;
          }
          return;
        })}
      </ul>
    );
  }
}
```

Figure 24. ToDoItem and ToDoList components

Enzyme allows to test ToDoItem and ToDoList components individually. For instance, it is possible to test that a ToDoItem correctly displays a single list item with the text of the task that it received as a prop. To accomplish that, the ToDoItem can be shallow rendered using the `shallow()` method of Enzyme. This method returns an object called shallow wrapper – a wrapper around the root node of the component. It is then possible to use Enzyme's find() method together with Enzyme's selectors in order to find specific nodes within a rendered component and run assertions about those nodes. (Airbnb 2019.) Figure 25 provides an example of such a test for the ToDoItem component.



```
import { shallow } from 'enzyme';
import * as React from 'react';

describe('<ToDoItem />', () => {
  it('displays a single list item with correct todo text', () => {
    const todo: ToDo = {
      id: 1,
      text: 'First item,
      done: false,
    };
    const wrapper = shallow(<ToDoItem todo={todo} />);
    const listItemWrapper = wrapper.find('li');
    expect(listItemWrapper.length).toBe(1);
    expect(listItemWrapper.text()).toBe('First item');
  });
});
```

Figure 25. Test for the ToDoItem component

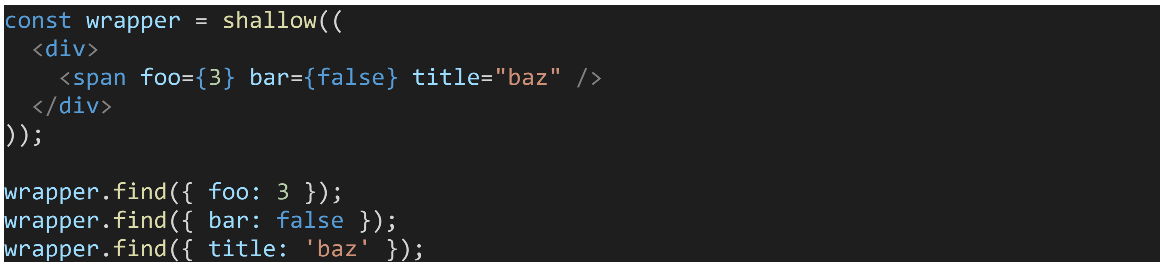
First, a `todo` object is created during the setup (or “arrange”) step of the test. Then, the ToDoItem component is shallow rendered using the `shallow()` method of Enzyme with the `todo` object passed as a prop (the “act” or “execute” step of the test). Finally, assertions are run against the shallow wrapper returned by the call to the `shallow()` method (the “assert” or “verify” step of the test). The list item node of the component’s render tree is found using the `find()` method of Enzyme. This method accepts an Enzyme selector, in this case a valid CSS selector using the `li` element tag name syntax. The `find()` method selects every node in the render tree that matches the selector and returns all of the nodes wrapped in a new shallow wrapper. (Airbnb 2019.)

The test runs two assertions against the render tree of the ToDoItem component. First, it verifies there is only one list item in the tree by running `wrapper.find(‘li’)`. This returns a new shallow wrapper containing every list item that Enzyme could find in the render tree of ToDoItem. The wrapper is saved into a `listItemWrapper` constant, and an assertion is run on the length property of the wrapper. The assertion `expect(listItemWrapper.length).toBe(1)` verifies that the component only renders one list item.

The second assertion verifies the component displays the text of the `todo` that it received is a prop. In order to get the rendered text, the `text()` method of Enzyme is used. The `text()` method returns a string of the rendered text of the current render tree. This method can only be called on a wrapper that contains a single node. The assertion `expect(listItemWrapper.text()).toBe ('First item')` verifies that the rendered text of the list item matches the text property of the todo object passed to ToDoItem as a prop. Another way to test the same thing would be `expect(listItemWrapper.text()).toBe(todo.text)`. (Airbnb 2019.)

The `find()` method of Enzyme accepts four types of Enzyme selectors.

1. A valid CSS selector. This includes class selectors (e.g. .className), id selectors (e.g. #id), element tag name selectors (e.g. input), attribute selectors (e.g. [href="link"]) and universal syntax (*).
2. A React component constructor (e.g. `wrapper.find(ToDoItem)`).
3. The displayName of a React component. React allows to set the displayName property of class components. The property is used in debugging messages. The property is set by default to match the name of a function or a class that defines the component, but can be overridden for class components. For instance, `ToDoItem.displayName = ‘To Do Item’` would set the display name and allow Enzyme to select the root node of the component’s render tree by calling `wrapper.find(‘To Do Item’)`. This selector only works if the displayName is a string that starts with a capital letter.
4. Object property selector allows to select nodes based on parts of their properties. Figure 26 shows an example of such a selector. The span node can be selected by each of its object properties. (Airbnb 2019.)



```
const wrapper = shallow((
  <div>
    <span foo={3} bar={false} title="baz" />
  </div>
));

wrapper.find({ foo: 3 });
wrapper.find({ bar: false });
wrapper.find({ title: 'baz' });
```

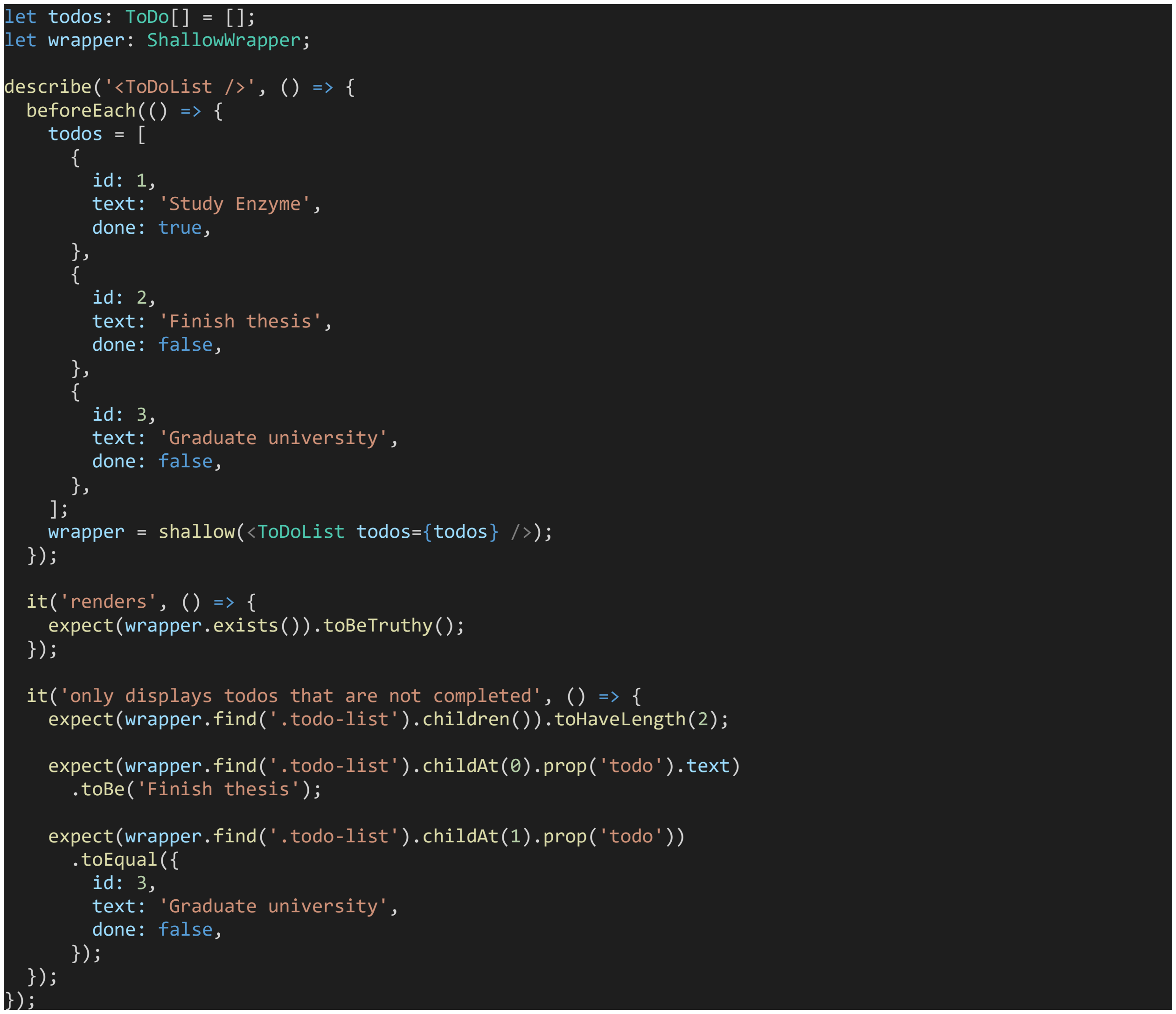
Figure 26. Using object property selector to find nodes in a render tree

The parent component ToDoList can be tested independently from its ToDoItem children. The component is supposed to only display the tasks that are not yet completed. Figure 27 provides an example of a test suite for the ToDoList component – or, in other words, the specification of its behaviour by example. The test suite uses the beforeEach() method of Jest for test setup. The method allows to repeat the same setup process for every test in the test suite (the parent describe block). An array of tasks is created inside beforeEach(), and the ToDoList component is shallow rendered with the array as the todos prop. The wrapper is then used in two tests. The first test verifies that the component renders anything by calling `expect(wrapper.exists()).toBeTruthy()`. The Enzyme method `exists()` checks whether the shallow wrapper contains any nodes. (Airbnb 2019.)

The second test verifies that the component only displays the tasks that are not completed. It contains three assertions. First, it finds the list node using its class name `.todo-list`. Then it uses the `children()` method of Enzyme that returns a new wrapper around the children of the list node. Finally, the test verifies that there are only two child nodes. There should only be two child nodes as the `todos` array passed to ToDoList as a prop contained one already completed task and two incomplete ones. (Airbnb 2019.)

The next assertion verifies that the first child of the list (the first ToDoItem component in the list) receives the task with correct text as its prop. It does so by first selecting the list node by class name and then using the `childAt()` method of Enzyme, which returns a child node at specified number. It then uses a `prop()` method of Enzyme that accepts the prop name as the argument and returns the value of that prop – in this case a ToDo object. The assertion is then run against the text property of the ToDo object – the toBe() matcher is used to compare the value of the prop to a specific string. (Airbnb 2019.)

The last assertion checks the second child of the list node. This time, instead of only asserting the value of one property of the `todo` prop, the entire ToDo object (the value of the `todo` prop) is compared to a specific object using the toEqual() matcher. The toEqual() matcher of Jest conducts a deep object comparison, meaning it compares each property of the objects recursively. (Airbnb 2019; Jest API Reference 2019b.)



```
let todos: ToDo[] = [];
let wrapper: ShallowWrapper;

describe('<ToDoList />', () => {
  beforeEach(() => {
    todos = [
      {
        id: 1,
        text: 'Study Enzyme',
        done: true,
      },
      {
        id: 2,
        text: 'Finish thesis',
        done: false,
      },
      {
        id: 3,
        text: 'Graduate university',
        done: false,
      },
    ];
    wrapper = shallow(<ToDoList todos={todos} />);
  });

  it('renders', () => {
    expect(wrapper.exists()).toBeTruthy();
  });

  it('only displays todos that are not completed', () => {
    expect(wrapper.find('.todo-list').children()).toHaveLength(2);

    expect(wrapper.find('.todo-list').childAt(0).prop('todo').text)
      .toBe('Finish thesis');

    expect(wrapper.find('.todo-list').childAt(1).prop('todo'))
      .toEqual({
        id: 3,
        text: 'Graduate university',
        done: false,
      });
  });
});
```

Figure 27. ToDoList component test suite

The examples in this subsection presented a number of Enzyme methods that can be used when shallow rendering components. The list if far from complete as the API of Enzyme provides a vast variety of methods in its documentation. However, the documentation of Enzyme does not provide strict guidelines for testing React components – it only provides the tools.

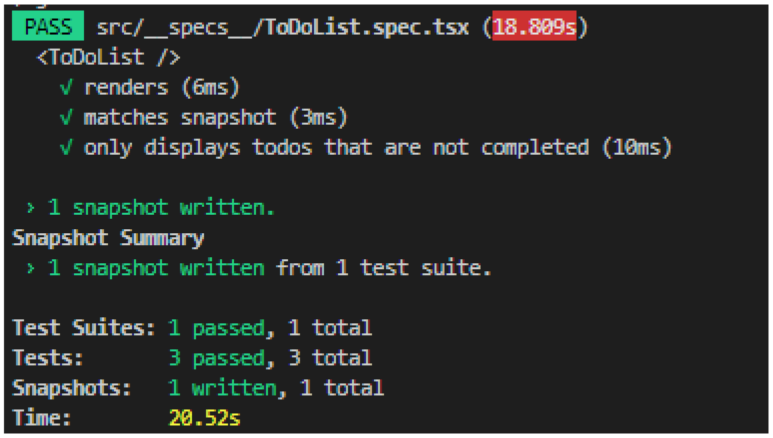
3.5.1 Snapshot testing React components with Jest and Enzyme

Snapshot testing is a Jest tool used to detect unexpected changes to the user interface of an application. The tool was originally introduced by Jest specifically for the purpose of testing React components. Snapshot testing allows to save a rendered React element into a file (“take a snapshot”) and then compare what the component renders to the contents of that file each time the tests are executed. (Jest Guides 2019b; Pojer 2016.) Figure 28 shows an example of a snapshot test for the ToDoList component. The test is added to the test suite in Figure 27 and therefore the setup is handled in the beforeEach() method of that figure. The component is shallow rendered and then compared against a snapshot using the toMatchSnapshot() matcher of Jest. The toJson() helper method from the enzyme-to-json library is used to make the snapshot more readable.

```
  it('matches snapshot', () => {
    expect(toJson(wrapper)).toMatchSnapshot();
  });
```

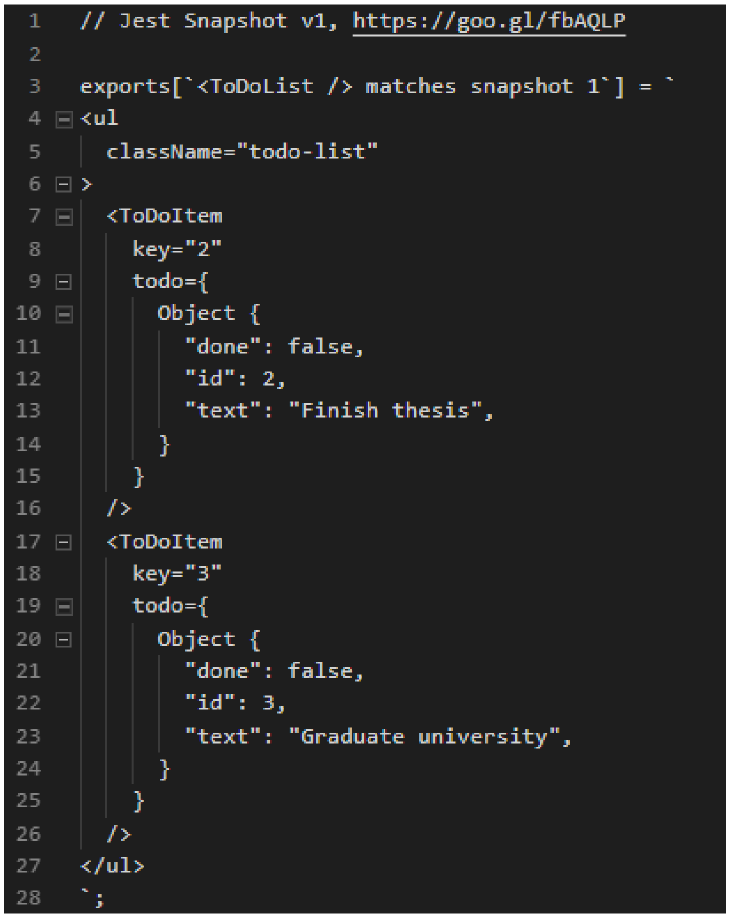
Figure 28. Snapshot test for the ToDoItem component

The first time this test is executed, it creates a new folder called __snapshots__ inside of the __specs__ folder containing the file with the test suite. Inside that folder Jest creates a new file called ToDoList.spec.tsx.snap – the full name of the test suite file with the .snap appended at the end. This file contains a snapshot of the render tree of this component. Figure 29 shows the output Jest prints to the console after the first time the snapshot test is executed and a new snapshot is written, while Figure 30 presents the contents of the snapshot file. Notice the ToDoItem component in the snapshot – its output is not rendered recursively. Shallow rendering only allows to see that a child component was rendered with certain props.



```
 PASS  src/__specs__/ToDoList.spec.tsx (18.809s)
  <ToDoList />
    √ renders (6ms)
    √ matches snapshot (3ms)
    √ only displays todos that are not completed (10ms)

 › 1 snapshot written.
Snapshot Summary
 › 1 snapshot written from 1 test suite.

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   1 written, 1 total
Time:        20.52s
```

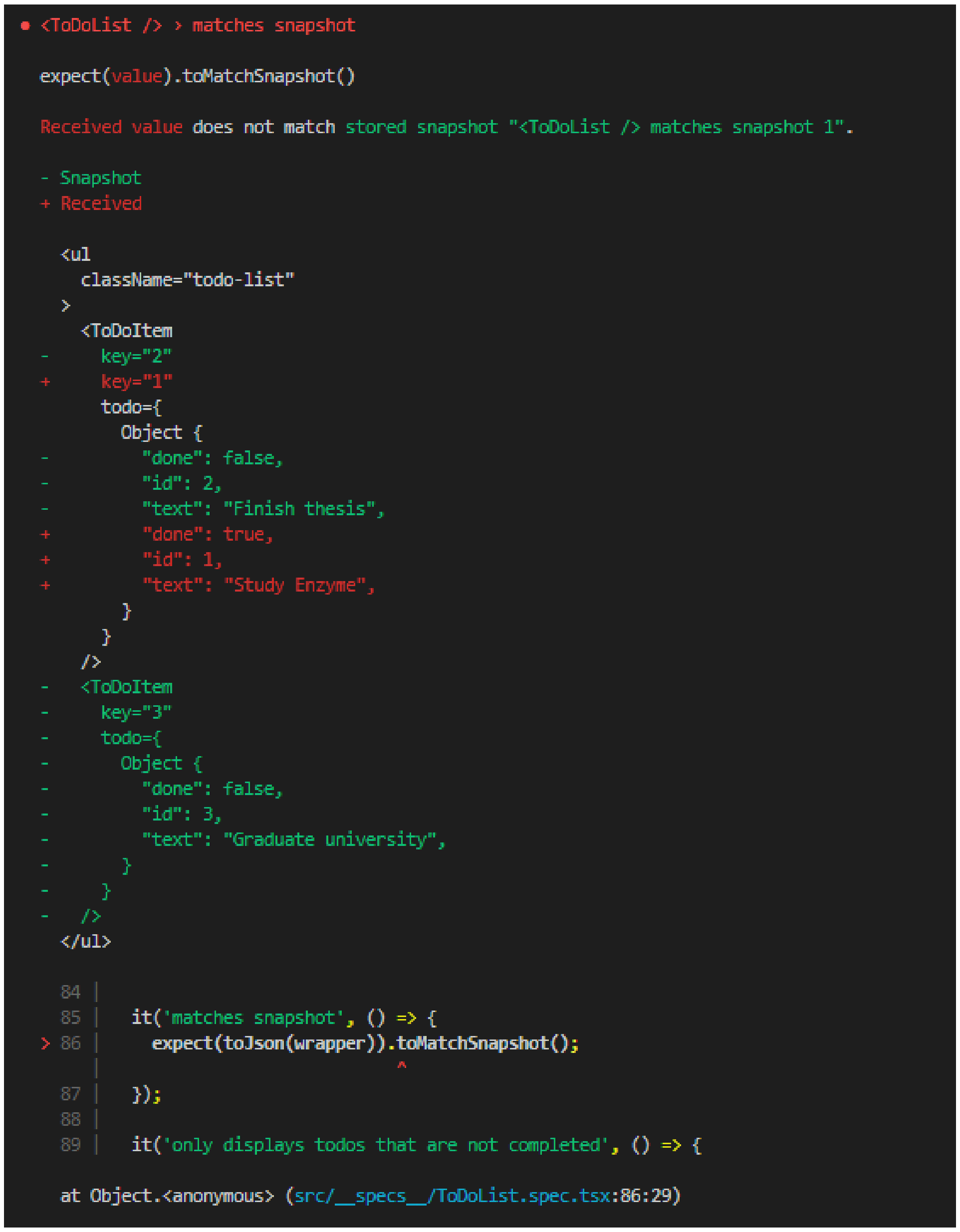
Figure 29. Jest output after executing a snapshot test for the first time



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

exports[`<ToDoList /> matches snapshot 1`] = `
<ul
  className="todo-list"
>
  <ToDoItem
    key="2"
    todo={
      Object {
        "done": false,
        "id": 2,
        "text": "Finish thesis",
      }
    }
  />
  <ToDoItem
    key="3"
    todo={
      Object {
        "done": false,
        "id": 3,
        "text": "Graduate university",
      }
    }
  />
</ul>
`;
```

Figure 30. Snapshot of the ToDoList component – ToDoList.spec.tsx.snap

After the snapshot is saved, the test will fail if the render output of the ToDoList component changes. For instance, a small change can be made to the component so that it renders only completed tasks instead of only the incomplete ones. The component will then only render a ToDoItem for the first task from the `todos` array and will fail the snapshot test. Figure 31 displays the output that Jest prints to the console when a snapshot test fails. Jest marks the differences between expected and received snapshots in green and in red. The expected value stored in the snapshot file is marked in green, while the deviated received value is marked in red. As figure 31 shows, Jest expected a list of two tasks with numbers 2 and 3 but received only task number 1.



```
● <ToDoList /> › matches snapshot

  expect(value).toMatchSnapshot()

  Received value does not match stored snapshot "<ToDoList /> matches snapshot 1".

  - Snapshot
  + Received

    <ul
      className="todo-list"
    >
      <ToDoItem
  -     key="2"
  +     key="1"
        todo={
          Object {
  -         "done": false,
  -         "id": 2,
  -         "text": "Finish thesis",
  +         "done": true,
  +         "id": 1,
  +         "text": "Study Enzyme",
          }
        }
      />
  -   <ToDoItem
  -     key="3"
  -     todo={
  -       Object {
  -         "done": false,
  -         "id": 3,
  -         "text": "Graduate university",
  -       }
  -     }
  -   />
    </ul>

    84 |
    85 |   it('matches snapshot', () => {
  > 86 |     expect(toJson(wrapper)).toMatchSnapshot();
       |                             ^
    87 |   });
    88 |
    89 |   it('only displays todos that are not completed', () => {

    at Object.<anonymous> (src/__specs__/ToDoList.spec.tsx:86:29)
```

Figure 31. Failed snapshot test

A failing snapshot test means one of two things: either the change to the component was accidental and there is a bug that needs to be corrected, or the change is intentional, and it is necessary to update the snapshot. If the change to a component's output is intentional, its snapshot can be updated. This is possible by running the `jest` command with the `--updateSnapshots` CLI option, which can be shortened to `--u`. This will re-run every test and overwrite every failing snapshot. If several snapshots are failing, but only one of them needs to be overwritten, the --u option can be combined with the `--testNamePattern` (`--t`) option mentioned above. When running Jest in watch mode, pressing the `u` key will update every failing snapshot, while pressing the `i` key will enter the so-called Interactive Snapshot Mode, which allows to view each snapshot individually and decide which of them to update. (Jest Guides 2019b.)

The documentation of Jest provides a list of best practices for efficient use of snapshots. First, snapshots should be treated as code and committed into version control. Keeping snapshots in version control helps during the code review process. The snapshots should be relatively short and focused. Second, snapshot tests should be deterministic, meaning they are repeatable and provide the same results every time they are executed. Nondeterministic code such as code dependant on current timestamp or a randomly generated value needs to be made deterministic via test doubles. For instance, a call to `Date.now()` can be mocked to always return a predetermined value in the test. Using nondeterministic code in snapshots can lead to scenarios where developers get used to always updating the snapshots when they fail, without looking into the underlying reasons for the failure – which defeats the purpose of using snapshots altogether. Third, snapshot names should be descriptive. This goes along with the principles of BDD described above. (Jest Guides 2019b.)

In general, however, snapshot testing is not compatible with the principles of TDD and BDD. Snapshots are created after the code that they are supposed to test, and therefore the design of the code cannot be driven by snapshot tests. Even though it is possible to write snapshots manually before writing the code that will match the snapshot (one tutorial from Toptal (Ortega 2019) does exactly that), this is cumbersome and requires learning the internal implementation of snapshot syntax, which takes developers' attention away from the behaviour of the actual SUT. Snapshot tests verify that the output of React components or other units under test has not changed. They can be combined with Enzyme unit tests. (Jest Guides 2019b.)

Snapshot tests make the process of running the tests require manual intervention in cases when snapshots need to be updated. This goes against one of the goals of test automation as established by Meszaros (2007) and already mentioned in

Chapter 2. Jest attempts to mitigate this by disabling the writing of snapshots by default when running tests on a Continuous Integration / Continuous Delivery server. Running tests locally still requires the manual update of snapshots when necessary. (Jest Guides 2019b.)

3.6 Jenkins, GitLab, Gitflow workflow and Git hooks

The final Section of Chapter 3 introduces the tools and the workflow used to enable the continuous delivery of the case study application. Continuous delivery (CD) is a set of principles and practices to reduce the cost, time, and risk of delivering incremental changes to users (Humble 2014). The foundational practices advocated by CD are configuration management, continuous integration and continuous testing (Humble 2017). Software that is developed in accordance with the CD discipline can be deployed into production at any time (Fowler 2013).

The key pattern that enables CD is the deployment pipeline (Humble 2017d). The deployment pipeline splits the build and deployment of an application into multiple steps. Each step supplies an increasing level of confidence, typically at the cost of taking more time as a result of more thorough inspection. The goal is to detect any issues that will result in failures in the production environment. (Fowler 2013a; Fowler 2013b.)

Jenkins is an open-source automation server that can be used to enable the continuous delivery of an application. Jenkins can automate various tasks related to building, testing and delivering software. The server provides a user interface for configuring various projects, but also supports a text-based pipeline syntax. (Jenkins Documentation 2017.)

GitLab is an open-source version control system. Used by over 100,000 organizations, GitLab also includes a variety of tools for CD. It is possible to set up the entire build process of an application using GitLab, but it can also be used for version control only and combined with an automation server such as Jenkins.

(GitLab 2019.) Both GitLab and Jenkins provide tools for integration with each other.

Gitflow is a workflow design for enterprise software projects that have a scheduled release cycle. Gitflow defines a set of branches that a git project should have and the order in which these branches should be created and merged. With Gitflow, a develop branch is created from the master branch at the beginning of the development process. The develop branch becomes the central branch where all new features are integrated. When a new feature needs to be developed, a feature branch is made from the develop branch. When the feature is complete in the feature branch, the feature branch can be merged into develop.

Features are never merged into the master branch directly. Once the develop branch is ready for a release, a new release branch with a specific release number is created from the develop branch. After the release branch is created, it can be tested and perfected with bug fixes and additional documentation. When the release branch is ready, it is merged into the master branch and tagged with the release number, freezing the tagged version in time. The release branch is also merged back into develop at this point, if any commits were made directly into the release branch. (Atlassian Bitbucket. Tutorials. Gitflow Workflow 2019.) Figure 32 provides an illustration of a Gitflow workflow.

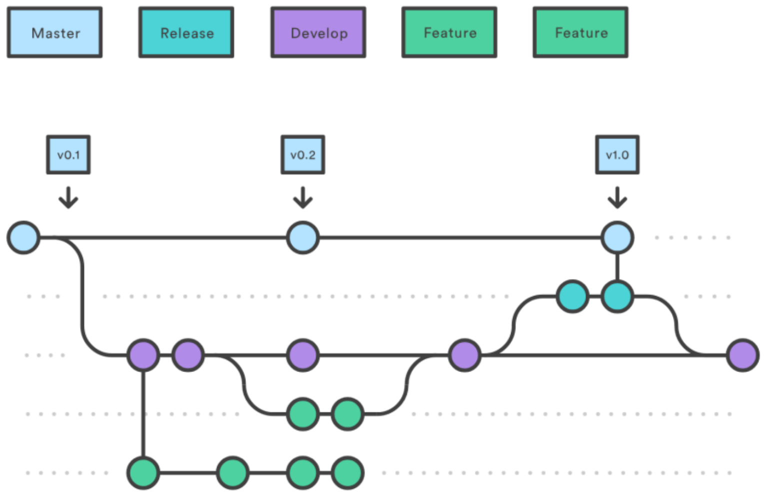


Figure 32. The Gitflow workflow (Atlassian Bitbucket. Tutorials. Gitflow Workflow 2019)

Gitflow provides a number of places to incorporate code review into the development cycle. For instance, the develop branch can be protected from being directly pushed into, and every feature branch can be merged via a merge request. According to the GitLab documentation (2019a), “a merge request is a request to merge one branch into another”. Merge requests allow comparing the differences between the code in the two branches, leaving comments and having a discussion, resolving merge conflicts, as well as assigning the request to be reviewed by a particular developer (GitLab Documentation 2019a). Each feature branch can be manually reviewed by one or multiple developers before being merged into develop. Moreover, automated tests can be executed every time a merge request is opened or modified.

Git hooks are commands or scripts executed by Git after a certain command such as push, commit or branch. Husky is a library that simplifies the addition and configuration of Git hooks for a project. With Husky, the hooks for each Git command can be added to the package.json file of the project. Adding Git hooks to a project can help make sure developers execute tests before submitting a merge request and that all tests pass. (Burnham 2019.)

4 PROPOSED TESTING STRATEGY FOR A REACT-REDUX APPLICATION

In this chapter a testing strategy for React applications at Observis is proposed. The proposal is based on the theoretical background discussed in Chapters 2 and 3, testing instructions provided by the Redux documentation, as well as an overview of four testing strategies used by developer teams at modern software development companies. The four companies were chosen based on the quality of the documentation they provide for their strategies, as well as the accordance with goals of test automation and the BDD technique described in Chapter 2. The four strategies used as a basis for this proposal come from the teams working at Djangostars (see Pysarenko 2019), Selleo (see Bak 2019), Commercetools (see Molinari 2018) and Willowtree (see Fishwick 2018).

Overall goals of the strategy

The strategy aims to conform with the goals of test automation as formulated by Meszaros (2007) and established in Section 2.3. That means the goal of the strategy is to improve the quality of React-Redux applications developed by Observis, help developers understand the software under test and reduce the risks of making changes to the existing code. The tests recommended by this strategy are easy to run, expressive, readable and easy to maintain as the codebase evolves around them. The strategy also aims to follow the principles of the BDD technique where each test acts as specification for a bit of behaviour of the application. For this reason, tests should be referred to as specs, as this shifts the emphasis to describing the desired behaviour of a piece of code. The test suites should act as examples of how a particular API under test is intended to be used. (Marston & Dees 2017.)

How should spec files be named and organized?

Specs should be placed close to the units they are testing. As the application grows, this helps keep specs organized and easy-to-find for each unit. (Bak 2019.) A folder called __specs__ should be created in the directory that contains the unit under test. The file containing the test suite for the unit must be placed in the __specs__ folder. The name of the test suite file must be the same as the name of the unit under test, with the .spec suffix appended. For example, a component called `Timeline` must have a test suite file called `Timeline.spec.tsx`. A utility function file `timeUtils.ts` must have a test suite file called `timeUtils.spec.ts`.

```
components
└───ContactList
│   │   ContactList.tsx
│   └───__specs__
│           ContactList.spec.tsx
└───DeleteDialog
│   │   index.tsx
│   │   DeleteButton.tsx
│   └───__specs__
│           DeleteDialog.spec.tsx
│           DeleteButton.spec.tsx
...
```

Figure 33. Spec file organization in a project

If a directory contains multiple units that must be tested, each unit must have a separate test suite file in the __specs__ folder. Sometimes when the name of the component matches the name of the directory where it is located (e.g `DeleteDialog.tsx` component inside a `DeleteDialog` directory), the component is named `index.tsx` to simplify the relative path when importing this component into other files. In this case, the spec for that component should use the name of the class or a function that declares the component as the basis for the spec name. Figure 33 provides an example of such spec organization.

How should specs be structured?

To keep tests focused and expressive, a single sentence must be used to name each test. Together with the name of the test suite, the test name must form a cohesive sentence that describes an example of how the unit under test should behave. As a whole, each test suite must provide a complete example of how the API of the unit under test is intended to be used. Each test should follow the arrange-act-assert pattern. “Happy path” tests must be avoided, and each test must make at least one assertion. The Commercetools testing strategy (Molinari 2018) emphasizes the usefulness of sticking to a single assertion per test. Using a single assertion in each test makes the test code more readable and focused and allows to quickly identify problems from the Jest output when tests fail. (Molinari 2018.) Meszaros (2007) also recommends verifying only one condition in each test as a general guideline (see Section 2.3). To improve readability and expressiveness of tests, duplication of setup data in multiple tests should be avoided. The two ways to remove duplication are the following:

- Code shared between multiple tests or test suites should be moved to centralized test utility methods. As established in Section 2.3, using utility methods allows to communicate the intent of the test more clearly. (Koskela 2013, 16–18; Meszaros 2007.) The centralized utility methods can be used for commonly used setup code such as creating certain test doubles.
- Tests within one test suite should use a shared, possibly parametrized setup function. A beforeEach() hook of Jest can be used when identical setup is required for every test in the test suite. Bak (2019) stresses the usefulness of using parametrized setup functions. Tests using setup functions are provided in the official Redux documentation (Abramov et al. 2018) as well.

What parts of the application should be tested?
The four types of units that must be tested are React components, Redux action creators, Redux reducers and utility functions.

What parts of the application should not be tested?
The behaviour of third-party libraries should not be tested (Pysarenko 2019). Whenever a unit under test relies on a third-party library, mock functions or manual mocks should be used to replicate the output of the third-party code. This includes the Redux and the React-Redux libraries as well, the behaviour of the libraries should not be tested. The Willowtree strategy does not recommend testing Redux-React methods such as mapStateToProps() (Fishwick 2018).

Creating tests that verify the correctness of data types is not necessary, even though the Djangostars strategy recommends doing so. The internal consistency of the data types in the code of the application is already verified by TypeScript. As mentioned above, according to Martin (2017), types do not specify the behaviour of the system, but rather provide internal consistency to the system's code. As the focus of the strategy is BDD, tests for data types are not necessary.

What tools are used for testing the units?
Jest is used as the test runner for every test written for the application. The test runner provided by Jest makes the tests easy and quick to run. Each test is executed in a separate process, and the tests that have previously failed are prioritized to provide faster feedback. Moreover, the order of test execution is determined by how long each test took during previous executions. (Jest 2019.) The vast assertion library of Jest allows creating readable and expressive tests, while its mock functions and manual mocks allow to configure deterministic test doubles, spy on unit interactions with collaborators and test each unit in isolation.

Enzyme is used for each test for a React component. The shallow rendering feature of Enzyme allows traversing the output of React components, re-render components in various states and see how the render tree changes. Together with the snapshot testing feature of Jest, Enzyme provides a powerful tool to detect unexpected changes in React elements. Moreover, shallow rendering allows to test each component in true isolation, as the children of the shallow rendered component are not rendered recursively. Shallow rendering makes the tests easier to maintain, as changes to child components do not result in failing tests for their parent components.

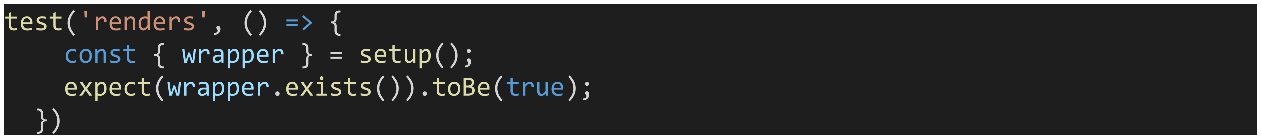
How to test a React component?

A React component should be covered with tests in the following order:

1. Test that the component renders
2. Test each prop of the component
3. Test each state variable of the component (stateful components only)
4. Test event handlers
    a. Simulate each event and expect the appropriate handler to be called
    b. Verify the correct behaviour of each event handler defined within the component
        i. If the event handler updates component state (calls setState())
        ii. If the event handler dispatches an action
        iii. Other behaviour of the event handler (e.g. interactions with Firebase)
5. Test lifecycle methods
6. Test conditions
7. Take a snapshot

According to Bak (2019), a minimal component test verifies that a component renders. This kind of test is referred to as a “smoke test”. A smoke test can be conducted by shallow rendering the component and expecting the shallow wrapper to contain at least one node. (Bak 2019.) Figure 34 shows an example of testing the rendering of a component in this way. The component is shallow rendered in a setup function that returns a wrapper. The `exists()` method of Enzyme is called on the wrapper, and the returned value of this method is expected to be `true`.

An alternative solution for testing component rendering is to take a snapshot. Both Bak (2019) and Pysarenko (2019) suggest taking a snapshot of the shallow wrapper as the first step when testing a component. Although this approach works when covering existing code, this can quickly become frustrating when designing new components through tests. As the component is developed, the snapshot requires to be continually updated. Therefore smoke testing using `exists()` should be preferred over snapshot testing when developing new components, and a snapshot test should be added as the final step, when the component is ready.



```
test('renders', () => {
    const { wrapper } = setup();
    expect(wrapper.exists()).toBe(true);
  })
```

Figure 34. Smoke testing a component

Each prop of the component under test needs to be tested individually. As established earlier, React components act as pure functions with respect to their props. The component under test should be shallow rendered with a specific prop value, and an assertion should be executed on the wrapper to verify that the prop was received by the component. Pysarenko, Bak and Abramov et al. all provide example of such component tests. If the component is stateful, each variable of the component state needs to be tested in the same way as well. The component under test should be shallow rendered with a specific state, and an assertion should be executed on the wrapper to verify that the state variable has been used by the component correctly.

Next, if a component under test includes event handlers, each event handler needs to be tested. Djangostars, Selleo and Willowtree all provide examples of event handler tests.

1. For each event handler, if the event handler is not a private method, create a spy on the event handler, simulate the event and expect the event handler has been called. If the event handler accepts parameters: expect the event handler has been called with correct parameters.
2. Specify the behaviour of each event handler, if this event handler is defined within the component under test and not passed to the component under test as a prop.
    a. If the event handler of a stateful component makes a call to setState(), expect the state to be updated correctly.
    b. If the event handler dispatches an action, expect the correct action to be dispatched.
    c. If the event performs any other operation, test the operation. For example, if the event handler interacts with Firebase, use the

manual mock for Firebase to specify the correct interaction. If an event handler updates a document in a collection, setup a mock collection and expect it to be updated correctly.

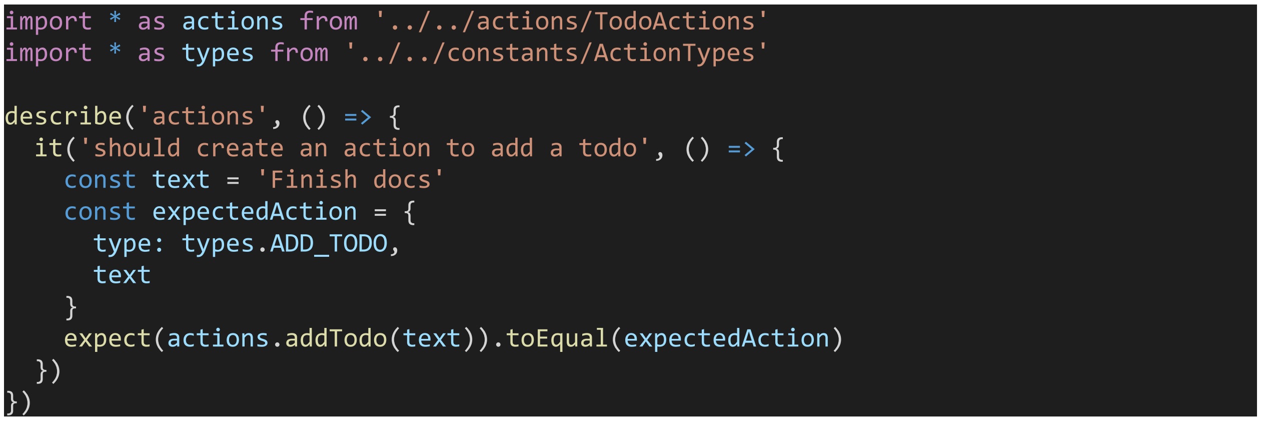
Next, if the component uses lifecycle methods, the lifecycle methods need to be tested. For instance, it is common for components to make subscriptions and asynchronous requests from componentDidMount(). In this case, the asynchronous code must be replaced with a mock function and the correct behaviour of the component upon the successful resolution of the request must be verified.

Next, all possible conditions must be tested. Testing conditions is a step recommended by the Djangostars strategy. If a React component relies on conditional rendering to determine the render output, component must be shallow rendered with different props / in different states and assertions must be run on the wrapper to verify certain parts of the render tree are visible or hidden. It is possible to completely cover this step while testing individual props and state variables in steps 2 and 3, however this is a good place to make sure no condition has been skipped. Viewing the branch code coverage of the component under test can hint at certain conditions being omitted.

Finally, a snapshot test for a component must be added. Both Djangostars and Selleo recommend limited use of snapshot tests. Djangostars suggests only keeping one snapshot per component as a general rule. If a component uses conditional rendering based on certain props or state to hide or show parts of the render tree, a snapshot can only be taken of one of the states. Other conditions are tested during the “testing conditions” step.

How to test action creators?

Action creators should be tested according to the official Redux documentation. As established earlier, an action creator is a function that returns an action – a plain object. In order to test an action creator, the action creator is called with certain parameters. The object it returns is then compared to a correct action.



```
import * as actions from '../../actions/TodoActions'
import * as types from '../../constants/ActionTypes'

describe('actions', () => {
  it('should create an action to add a todo', () => {
    const text = 'Finish docs'
    const expectedAction = {
      type: types.ADD_TODO,
      text
    }
    expect(actions.addTodo(text)).toEqual(expectedAction)
  })
})
```

Figure 35. A test for an action creator of a task management application (Abramov et al. 2018.)

Figure 35 shows an example of a test for an action creator. (Abramov et al. 2018.)

How to test reducers?

Reducers should be tested according to the official Redux documentation. A reducer is a pure function that accepts an action and the state of an application and returns the appropriately updated state. Each reducer should be tested in the following steps:

1. Setup a state object with initial state.
2. Call the reducer with an action of an unsupported type and the initial state and expect the reducer to return the initial state unchanged.
3. Write a separate test for each action type handled by the reducer. In each test, call the reducer with the action of this type and the initial state as parameters.
4. Expect the reducer to return the state updated in the right way. (Abramov et al. 2018.)

How to test a utility function?

Utility functions are typically pure functions that accept parameters and return values. When testing a utility function, a full list of possible behaviours of the function needs to be planned. If the function accepts multiple parameters, different combinations of the parameters must be considered.

1. Test that the function throws exceptions if any of the parameters are undefined or incorrect.
2. Test that the function returns correct values when all parameters are defined. (Pysarenko 2019.)

What metrics should be collected?

The two metrics that should be collected for the application are the test coverage report and the number of issues with the software discovered by the customer in the production environment or during exploratory testing in the staging environment. As mentioned in Section 2.3, the code coverage only shows the percentage of the codebase that is exercised in tests. It does not speak to the quality of the tests, but only illuminates the parts of the codebase that are not covered. Still, it is a useful tool for developers to choose which areas to focus the testing efforts on, and significantly low coverage is a sign that there are issues with the software. Fowler (2012) argues the only two metrics signifying that there are enough tests for the code are the rarity of bugs in production and the confidence that developers have to change existing code. The confidence is difficult to measure by means other than a survey, but the number of discovered bugs is easy to collect and analyse via project management tools such as JIRA. As more tests are written for the software, the number of discovered bugs should decrease.

How are tests integrated into the deployment pipeline?

Developers should be running Jest in watch mode while developing the software. In order to make sure that the software checked into version control passes the tests, a Git hook should be configured to run tests every time the software is pushed. If the tests fail, the push is cancelled. Additionally, a merge request inspector project should be set up on the Jenkins server. The inspector project will be triggered every time a merge request is opened or modified. The project will leave a comment on the GitLab merge request, and if the tests are failing, the code should not be merged into the target branch. Automatic execution of tests for each merge request speeds up the feedback that developers receive on the merge requests that they open. Moreover, it saves time for developers that have to perform manual code review on the merge request – the manual review can only begin when the merge request passes the tests, meaning developers don’t have to spend time reviewing malfunctioning code.

How should incidents with the software be managed?

If a bug is discovered in the application either by the customer in the production environment or during exploratory testing in the staging environment, a bug report must be filed in the project management software (JIRA for the case study). A unit test (or several tests) addressing the bug must be created.

5 PRACTICAL IMPLEMENTATION

This chapter is structured as follows: Section 5.1 documents the setup process of the testing environment. The following sections provide detailed examples of tests for utility functions, action creators, reducers and React components. Next, a Git hook is configured to execute tests locally before each push event. Section 5.7 describes the configuration of a Jenkins project that automatically executes tests for every merge request and leaves a comment on GitLab notifying developers whether the tests passed or failed. Finally, Section 5.8 explains how the HTML code coverage report generated by Jest is displayed on the Jenkins server.

5.1 Configuring the test environment

All the required Node modules need to be installed. Yarn is used as the package manager. Figure 36 shows the yarn command required to install the packages required for testing. Because all of these are development dependencies that are not required in production, the --dev tag is used. The required packages are:

- jest
- enzyme
- enzyme-adapter-react-16. An adapter required by Enzyme to work with a specific version of React. MOWO dashboard uses React 16.
- ts-jest. A TypeScript pre-processor that allows to use Jest together with TypeScript (Ts-Jest 2019).
- identity-object-proxy. A library used for mocking webpack imports. In particular, it can be used to mock styles declared in a separate file and imported into a component. (Identity-obj-proxy 2019.)
- Ts-Mock-Firebase. A library that provides a mock implementation of Google’s Firebase (Ts-Mock-Firebase 2019).
- mockdate. A library to set specific dates within tests.
- Type definitions for several of the libraries.

```
yarn add --dev jest @types/jest enzyme @types/enzyme enzyme-adapter-react-16
@types/enzyme-adapter-react-16 ts-jest identity-obj-proxy ts-mock-firebase mockdate
@types/mockdate
```

Figure 36. Installation of the required packages

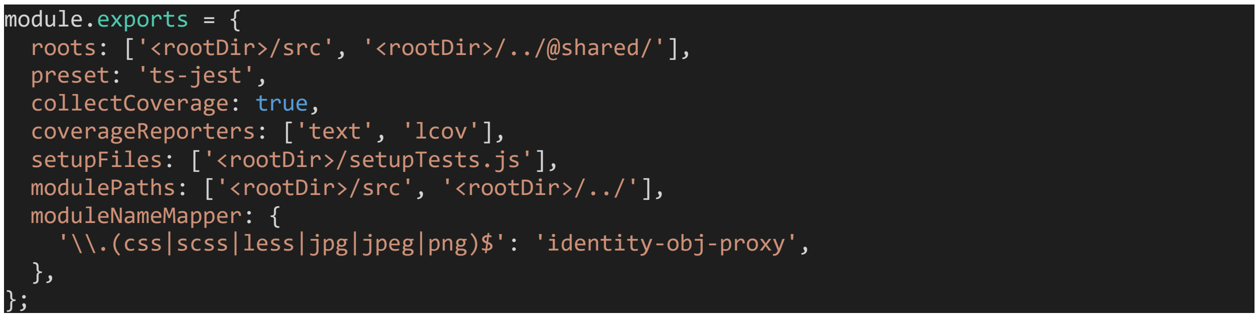
```
{
    "name": "mowo-dashboard",
    "version": "0.1.0",
    "private": true,
    "scripts": {

        "test": "jest",
        "test:watch": "jest --watch --verbose",
        ...
    },
    "dependencies": {
        ...
        "firebase": "5.8.2",
        "moment": "2.22.2",
        "react": "16.6.0",
        ...
        "react-dom": "16.6.0",
        ...
        "react-redux": "5.1.1",
        "react-router-dom": "5.0.0",
        ...
        "ts-date": "2.1.7",
        "ts-form-validation": "1.1.5",
        ...
    },
    "devDependencies": {
        ...
        "@types/enzyme": "3.1.17",
        "@types/enzyme-adapter-react-16": "1.0.3",
        "@types/jest": "23.3.9",
        ...
        "@types/mockdate": "2.0.0",
        "@types/react-dom": "16.0.10",
        ...
        "@types/react-redux": "6.0.10",
        "@types/react-router-dom": "4.3.1",
        "@types/redux": "3.6.0",
        ...
        "awesome-typescript-loader": "5.2.1",
        ...
        "enzyme": "3.8.0",
        "enzyme-adapter-react-16": "1.9.1",
        "enzyme-to-json": "3.3.5",
        ...
        "identity-obj-proxy": "3.0.0",
        "jest": "23.6.0",
        "mockdate": "2.0.2",
        ...
        "ts-jest": "23.10.5",
        "ts-mock-firebase": "^2.1.3",
        "tslint": "5.12.0",
        "typescript": "3.2.2"
    }
}
```

Figure 37. The package.json file of the project (libraries not relevant to the study redacted)

The packages are installed into the devDependencies section of the package.json file. Figure 37 shows the package json file of the project with the versions of the packages used during the study. Packages not relevant to the thesis work are redacted – replaced with ellipses in the figure. Scripts “test” and “test:watch” are configured to run the tests from the command line.

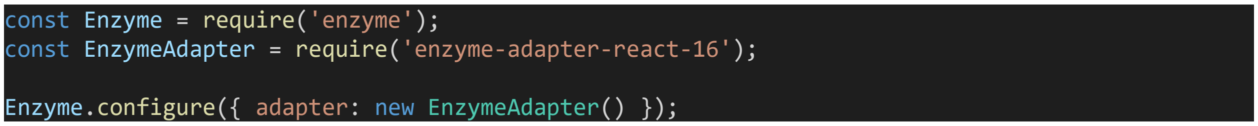
Next, Jest needs to be configured. A configuration file called jest.config.js is created at the root of the project. Figure 38 displays the contents of the configuration file. The following properties were configured:

- roots. An array of paths where Jest will look for test files and source files. The two paths configured are the src folder in the root directory of the MOWO dashboard project and the @shared project located outside of the root directory for MOWO dashboard. The @shared project contains utility functions used both by MOWO dashboard and the MOWO mobile application that is outside of the scope of this study.
- preset. A preset that is used as a basis for the configuration. Configured to point to the ts-jest Node module installed above.
- collectCoverage. If true, Jest collects coverage information and generates coverage reports.
- coverageReporters. An array of coverage reporter names. The text reporter is required to print tests to the console, while the LCOV HTML report will be used to display the coverage report on Jenkins.
- setupFiles. An array of paths to files that will be executed before each test file. This allows to avoid duplication of setup code in every test file. The file specified is setupTests.js in the root directory of the project. Enzyme is configured in this file. Figure 39 shows the contents of the file. Enzyme and the Enzyme adapter are imported in the file, and Enzyme is configured with the adapter.
- modulePaths. An array of paths to locations that Jest will check when resolving modules. The src directory in the root of the project as well as the root of the entire mowo repository are specified.
- moduleNameMapper. A map of regular expressions to module names. Used to stub out resources such as files and styles. A regular expression matching any file with .css, .scss, .less., .jpg, .jpeg or .png extension is configured to replace the imports of files with these extensions with stubs provided by the identity-object-proxy library installed above.



```
module.exports = {
  roots: ['<rootDir>/src', '<rootDir>/../@shared/'],
  preset: 'ts-jest',
  collectCoverage: true,
  coverageReporters: ['text', 'lcov'],
  setupFiles: ['<rootDir>/setupTests.js'],
  modulePaths: ['<rootDir>/src', '<rootDir>/../'],
  moduleNameMapper: {
    '\\.(css|scss|less|jpg|jpeg|png)$': 'identity-obj-proxy',
  },
};
```

Figure 38. Jest configuration for the project



```
const Enzyme = require('enzyme');
const EnzymeAdapter = require('enzyme-adapter-react-16');

Enzyme.configure({ adapter: new EnzymeAdapter() });
```

Figure 39. Configuring Enzyme in the setupTests.js file

Lastly, a manual mock has to be made for Firebase – a cloud database service provided by Google and used by the MOWO dashboard application. NPM package Ts-Mock-Firebase is used for that purpose. The library emulates the functionality of Firebase. As explained in Section 3.1.3, in order to create a manual mock for a Node module a __mocks__ folder needs to be created in one of the roots of the project, and a file with the same name as the Node module needs to be placed into the __mocks__ folder. Figure 40 shows the file called firebase.ts in the __mocks__ folder located in the src folder. The src folder is one of the roots specified in the Jest configuration (Figure 38).

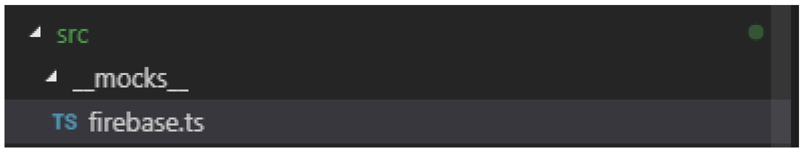
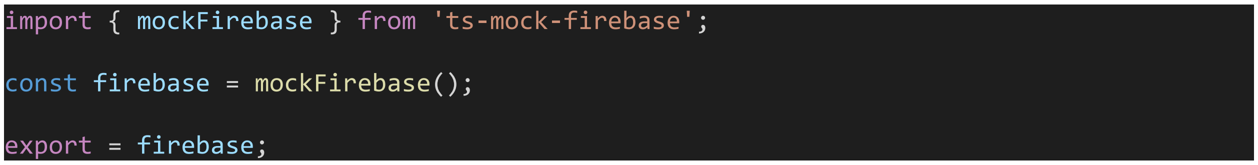


Figure 40. Manual mock file for Firebase

The documentation provided on the GitHub page of Ts-Mock-Firebase provides instructions for configuring the manual mock. It is enough to import the mockFirebase() method from Ts-Mock-Firebase, call it to create a new instance of mock Firebase and export the instance. (Ts-Mock-Firebase 2019.) Figure 41 shows the contents of the manual mock for Firebase. When the Firebase mock is set up, Jest will automatically use the mock instead of the actual library (Jest Guides 2019a; Ts-Mock-Firebase 2019).



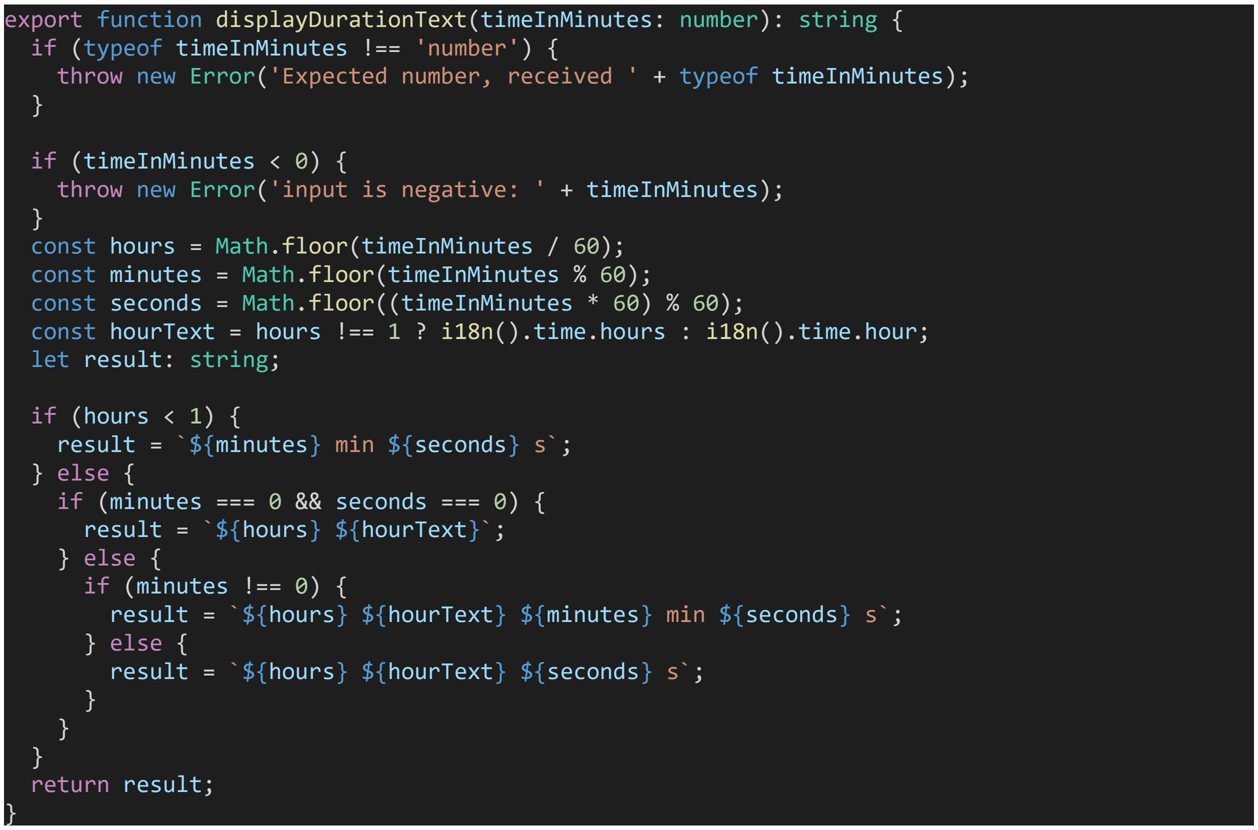
```
import { mockFirebase } from 'ts-mock-firebase';

const firebase = mockFirebase();

export = firebase;
```

Figure 41. Code of the manual mock for Firebase (Ts-Mock-Firebase 2019)

Having established a testing strategy and set up the testing environment, units of the software can now be tested according to the strategy. The benefits of writing tests before the software were emphasized in Chapter 2. However, the core functionality of the MOWO dashboard project has largely been completed by the time the test automation efforts commenced. It is therefore necessary to retroactively cover existing software with tests. Even though the tests in the following examples are written after the SUT, they are structured with the BDD technique in mind and can be used for reference when developing new parts of the software moving forward.

5.2 Example of testing a utility function

The function displayDurationText() is responsible for generating a human-readable string that communicates the duration of a certain task. It accepts a number representing the duration in minutes and returns a string. Figure 42 contains the code of the function. The function determines the correct format for the duration string based on three parameters: hours, minutes and seconds. First, it checks whether the input is a number and whether the input is non-negative and throws errors if either of the conditions fail. Then, it determines the number of hours, minutes and seconds in the provided duration. The function then generates a variation of a human-readable duration string based on whether the provided duration includes a full number of hours, minutes or seconds.



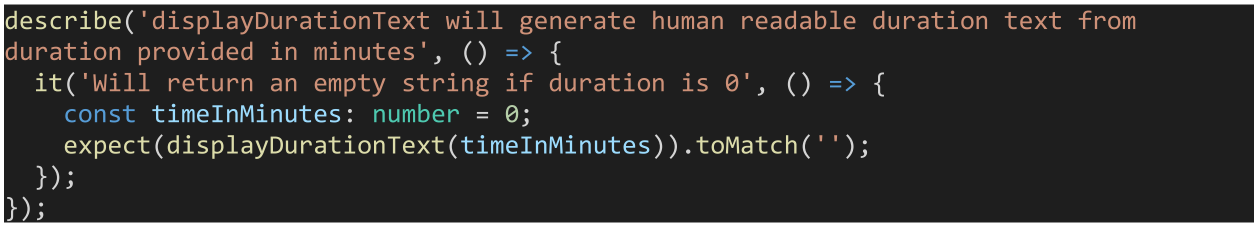
```
export function displayDurationText(timeInMinutes: number): string {
  if (typeof timeInMinutes !== 'number') {
    throw new Error('Expected number, received ' + typeof timeInMinutes);
  }

  if (timeInMinutes < 0) {
    throw new Error('input is negative: ' + timeInMinutes);
  }
  const hours = Math.floor(timeInMinutes / 60);
  const minutes = Math.floor(timeInMinutes % 60);
  const seconds = Math.floor((timeInMinutes * 60) % 60);
  const hourText = hours !== 1 ? i18n().time.hours : i18n().time.hour;
  let result: string;

  if (hours < 1) {
    result = `${minutes} min ${seconds} s`;
  } else {
    if (minutes === 0 && seconds === 0) {
      result = `${hours} ${hourText}`;
    } else {
      if (minutes !== 0) {
        result = `${hours} ${hourText} ${minutes} min ${seconds} s`;
      } else {
        result = `${hours} ${hourText} ${seconds} s`;
      }
    }
  }
  return result;
}
```

Figure 42. Utility function displayDurationText()

It is possible to document the required behaviour by making a table as shown in Table 1. A cross in each column represents whether the duration in minutes contains a full number of hours, minutes or seconds. A test can be written for each line of the table. Figure 43 shows a test suite for the displayDurationText() function with a test for the first line of the table – the case when the input duration is zero.

	Hours	Minutes	Seconds
1			
2			X
3		X	
4	X		
5		X	X
6	X		X
7	X	X	
8	X	X	X

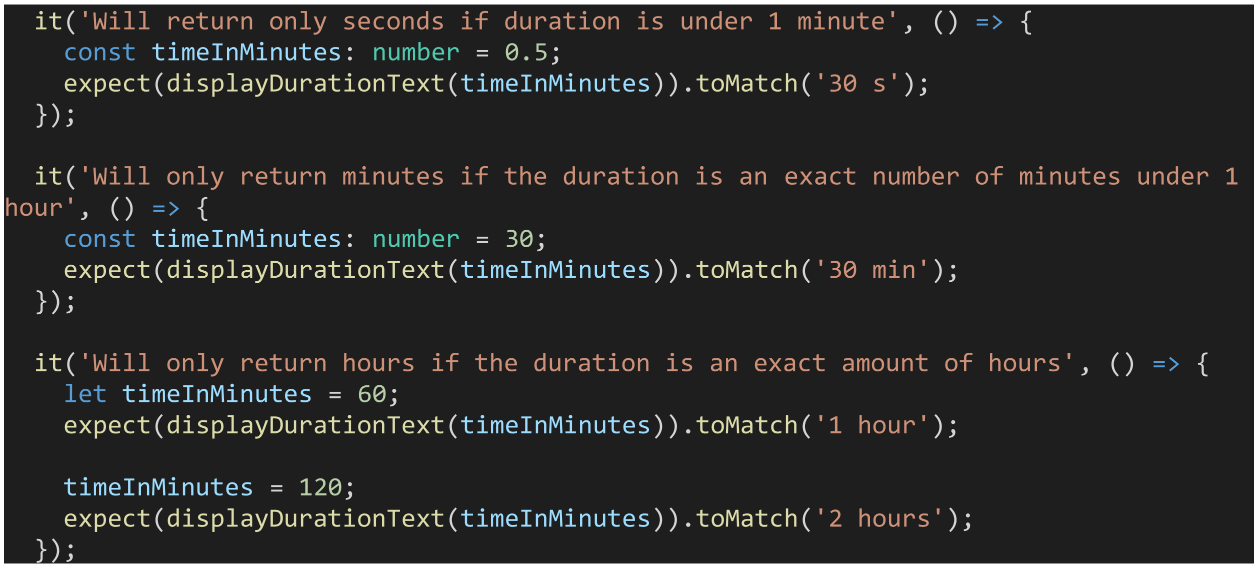
Table 1. Possible variations of the displayDurationText() function input



```
describe('displayDurationText will generate human readable duration text from
duration provided in minutes', () => {
  it('Will return an empty string if duration is 0', () => {
    const timeInMinutes: number = 0;
    expect(displayDurationText(timeInMinutes)).toMatch('');
  });
});
```

Figure 43. The first test for the displayDurationText function()

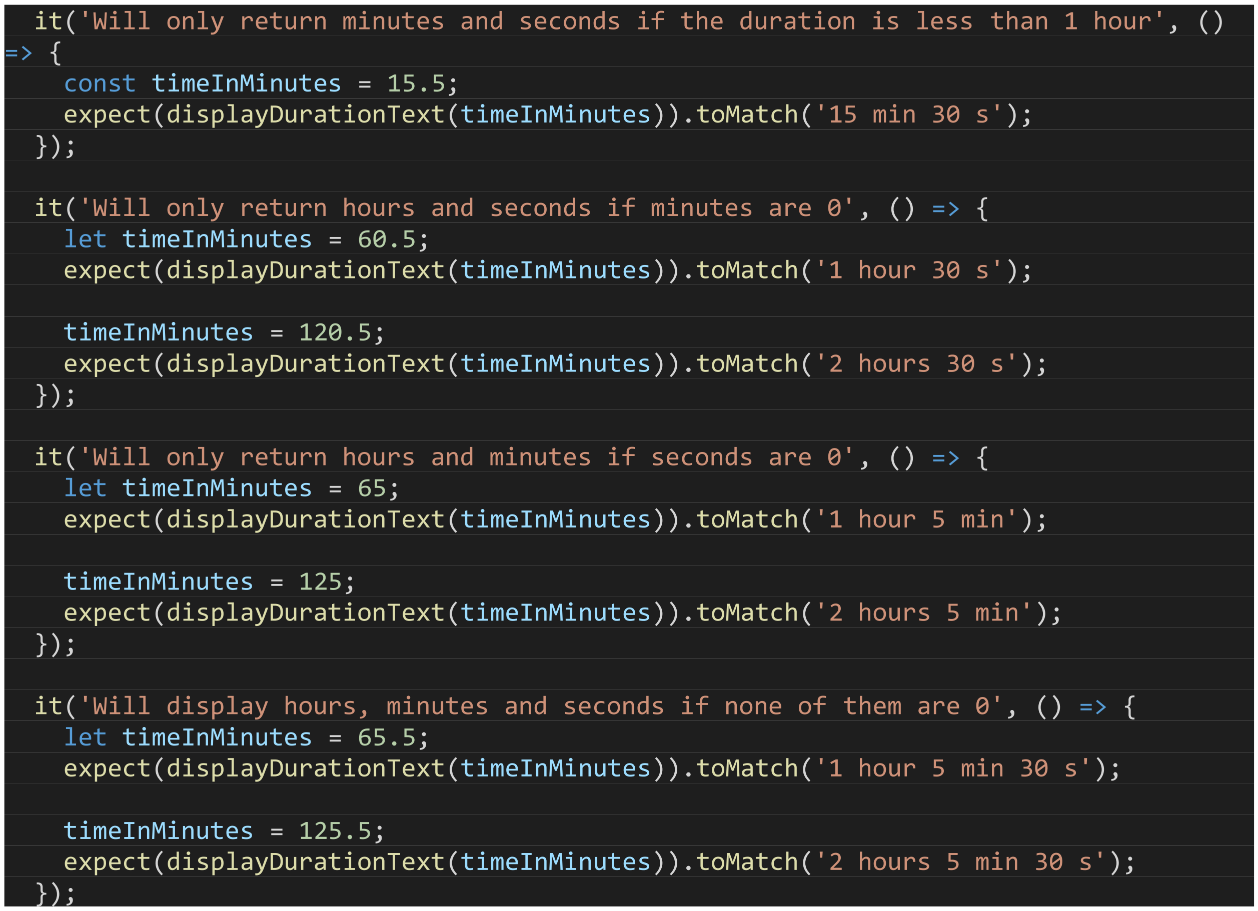
It is now possible to go through the table line by line and write the remaining tests. The tests in Figure 44 cover the cases 2 through 4 where the input duration contains only seconds, only minutes or only hours. The test for the variation 4 specifies the behaviour in two cases: when the duration is less than two hours long and when the duration is more than two hours long. Different strings should be generated in these cases: “hour” and “hours” respectively.



```
  it('Will return only seconds if duration is under 1 minute', () => {
    const timeInMinutes: number = 0.5;
    expect(displayDurationText(timeInMinutes)).toMatch('30 s');
  });

  it('Will only return minutes if the duration is an exact number of minutes under 1
hour', () => {
    const timeInMinutes: number = 30;
    expect(displayDurationText(timeInMinutes)).toMatch('30 min');
  });

  it('Will only return hours if the duration is an exact amount of hours', () => {
    let timeInMinutes = 60;
    expect(displayDurationText(timeInMinutes)).toMatch('1 hour');

    timeInMinutes = 120;
    expect(displayDurationText(timeInMinutes)).toMatch('2 hours');
  });
```

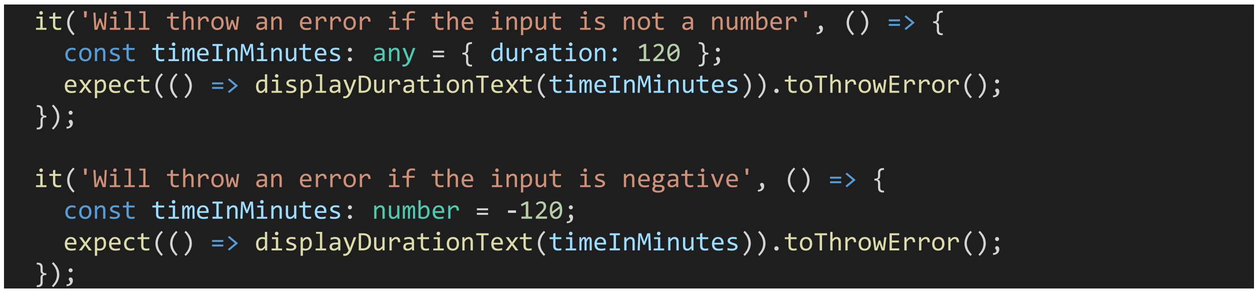
Figure 44. Tests for input variations 2–4

Now the cases 5 through 8 can be specified where the input duration contains only minutes and seconds, only hours and seconds, only hours and minutes, and lastly hours, minutes and seconds. Figure 45 contains these tests. Finally, tests should be written to specify that the function throws errors when the input is negative or not a number, as this is also a part of the function’s behaviour. These tests are presented in Figure 46.



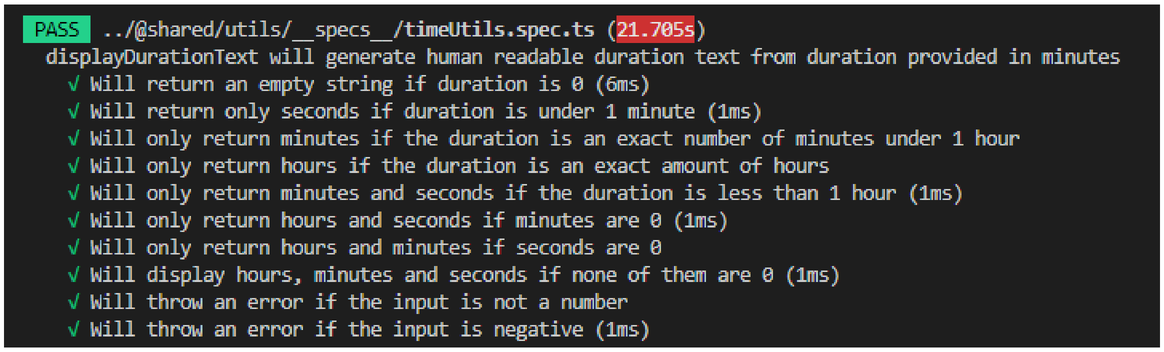
```
  it('Will only return minutes and seconds if the duration is less than 1 hour', ()
=> {
    const timeInMinutes = 15.5;
    expect(displayDurationText(timeInMinutes)).toMatch('15 min 30 s');
  });

  it('Will only return hours and seconds if minutes are 0', () => {
    let timeInMinutes = 60.5;
    expect(displayDurationText(timeInMinutes)).toMatch('1 hour 30 s');

    timeInMinutes = 120.5;
    expect(displayDurationText(timeInMinutes)).toMatch('2 hours 30 s');
  });

  it('Will only return hours and minutes if seconds are 0', () => {
    let timeInMinutes = 65;
    expect(displayDurationText(timeInMinutes)).toMatch('1 hour 5 min');

    timeInMinutes = 125;
    expect(displayDurationText(timeInMinutes)).toMatch('2 hours 5 min');
  });

  it('Will display hours, minutes and seconds if none of them are 0', () => {
    let timeInMinutes = 65.5;
    expect(displayDurationText(timeInMinutes)).toMatch('1 hour 5 min 30 s');

    timeInMinutes = 125.5;
    expect(displayDurationText(timeInMinutes)).toMatch('2 hours 5 min 30 s');
  });
```

Figure 45. Tests for input variations 5–8



```
  it('Will throw an error if the input is not a number', () => {
    const timeInMinutes: any = { duration: 120 };
    expect(() => displayDurationText(timeInMinutes)).toThrowError();
  });

  it('Will throw an error if the input is negative', () => {
    const timeInMinutes: number = -120;
    expect(() => displayDurationText(timeInMinutes)).toThrowError();
  });
```

Figure 46. Specifying the cases where displayDurationText() should throw errors



```
 PASS  ../@shared/utils/__specs__/timeUtils.spec.ts (21.705s)
  displayDurationText will generate human readable duration text from duration provided in minutes
    √ Will return an empty string if duration is 0 (6ms)
    √ Will return only seconds if duration is under 1 minute (1ms)
    √ Will only return minutes if the duration is an exact number of minutes under 1 hour
    √ Will only return hours if the duration is an exact amount of hours
    √ Will only return minutes and seconds if the duration is less than 1 hour (1ms)
    √ Will only return hours and seconds if minutes are 0 (1ms)
    √ Will only return hours and minutes if seconds are 0
    √ Will display hours, minutes and seconds if none of them are 0 (1ms)
    √ Will throw an error if the input is not a number
    √ Will throw an error if the input is negative (1ms)
```

Figure 47. Jest console output when displayDurationText() tests pass

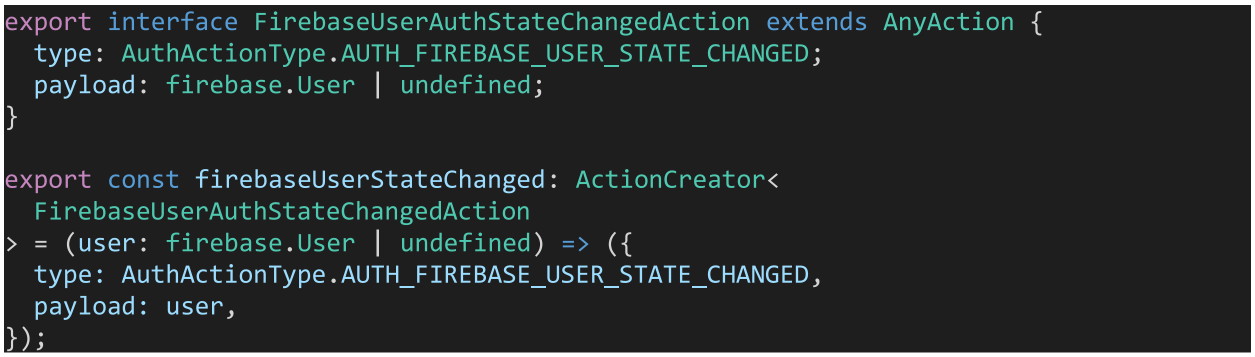
The full test suite for the displayDurationText() function is presented in Appendix 1. Figure 47 shows the output that Jest prints to the console when all the tests for displayDurationText() pass. Notice how making the test names expressive by using full sentences makes it easy to grasp the full behaviour of the function by reading the Jest output.

5.3 Examples of testing action creators

Action creators should be tested according to the official Redux documentation. authActions.ts file contains the action creators responsible for handling application state related to the user authentication status. The five action types it includes are:

1. AUTH_FIREBASE_USER_STATE_CHANGED
2. AUTH_APP_USER_INFO_UPDATED
3. AUTH_APP_USER_ACTIVE_ORG_CHANGED
4. AUTH_FIREBASE_SET_ERROR
5. AUTH_FIREBASE_SET_AUTHENTICATING

As established above, in order to test an action creator, the action creator needs to be called with certain parameters and the output of the action creator has to be compared to the correct action object. Figure 48 presents the code of the action creator 1. The action this creator produces is dispatched when the user logs in or logs out of the system. The creator accepts either undefined or a User object as the payload.

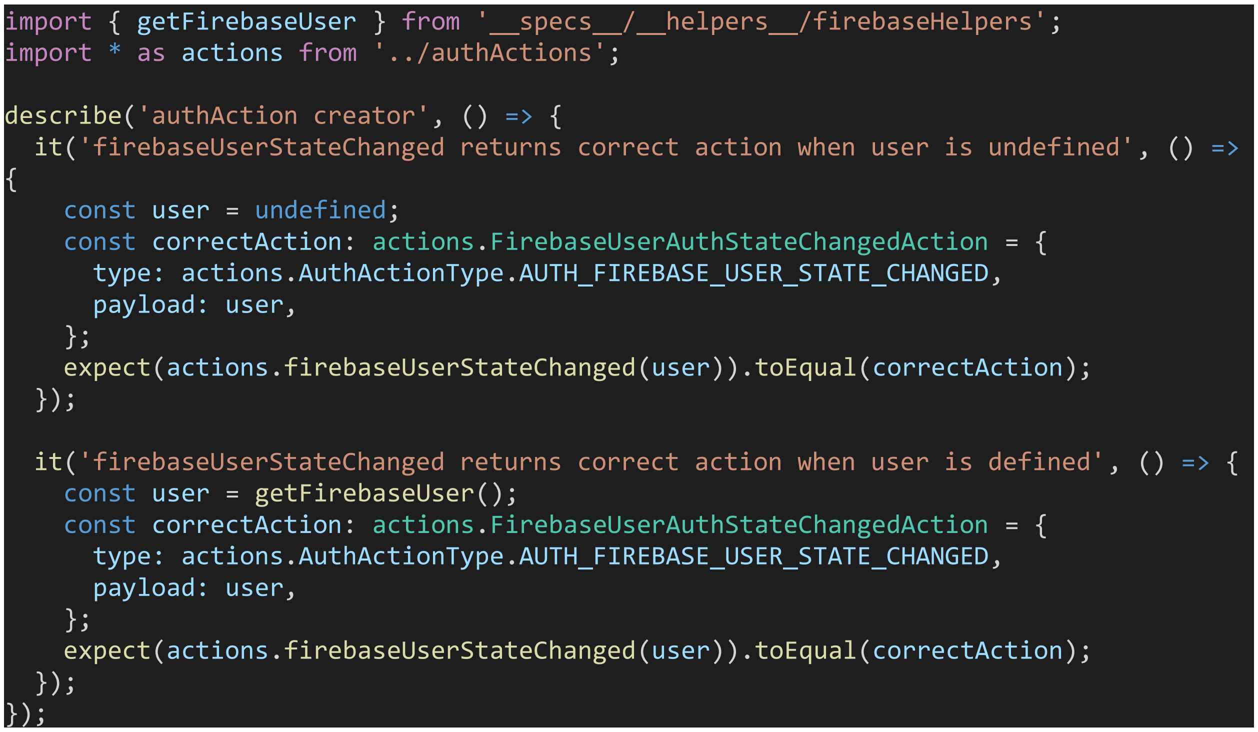


```
export interface FirebaseUserAuthStateChangedAction extends AnyAction {
  type: AuthActionType.AUTH_FIREBASE_USER_STATE_CHANGED;
  payload: firebase.User | undefined;
}

export const firebaseUserStateChanged: ActionCreator<
  FirebaseUserAuthStateChangedAction
> = (user: firebase.User | undefined) => ({
  type: AuthActionType.AUTH_FIREBASE_USER_STATE_CHANGED,
  payload: user,
});
```

Figure 48. firebaseUserStateChanged() action creator

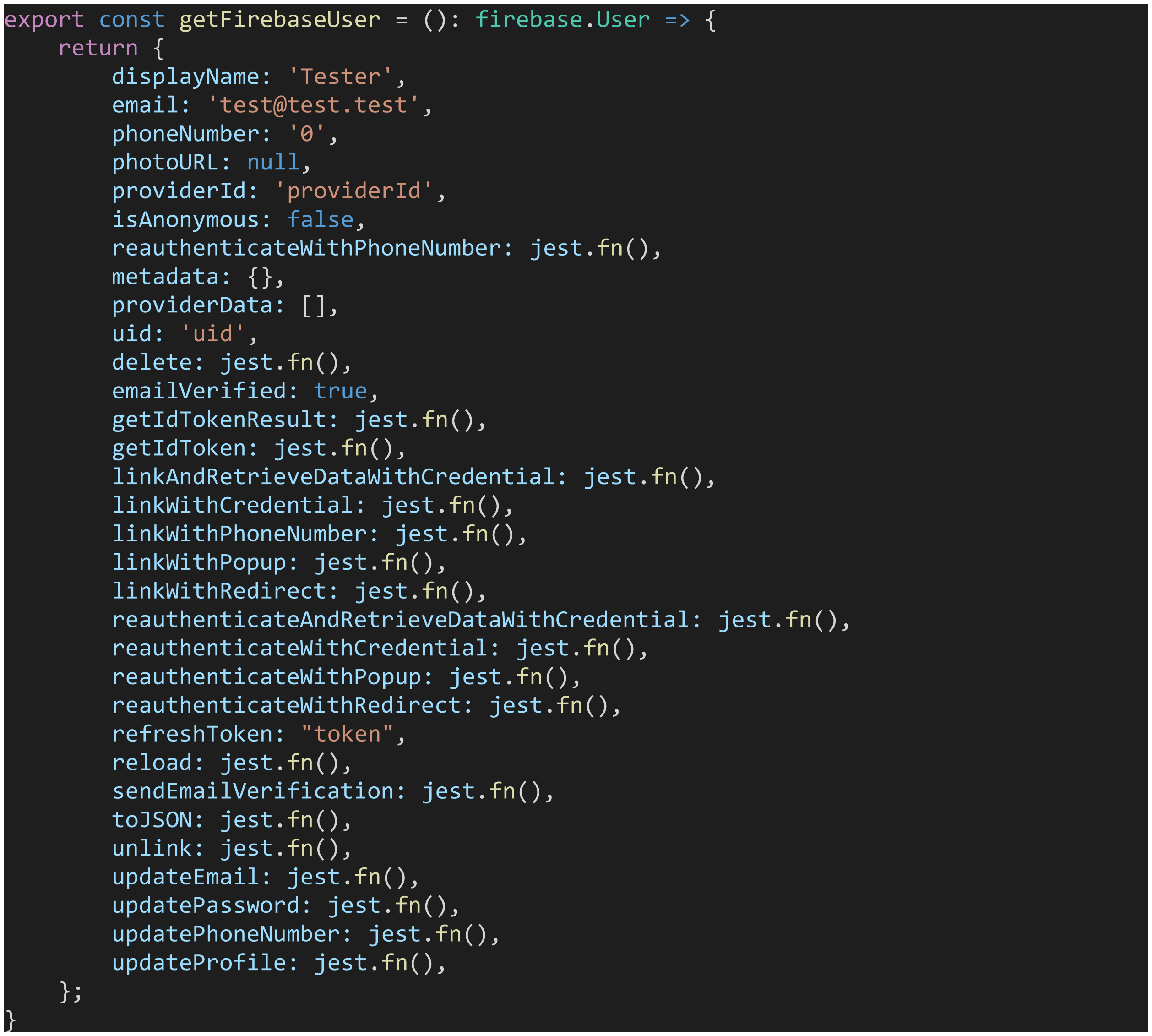
The tests for both types of the payload can be written. Figure 49 contains the test suite for the firebaseUserStateChanged() action creator. The test suite imports the action creators that need to be tested and a getFirebaseUser() centralized utility method. The method returns a test double which is a fake object of type firebase.User. Figure 50 shows the code of the utility method. The firebase.User type required a number of methods that are all replaced with mock functions. Defining this object directly within test code would make the test code longer and take away the focus from the primary goal of the test. Moving the definition of the firebase.User object into a utility method allows to make the test more expressive.

Both tests begin by setting up a user object that will be passed to the action creator and a correct action that is expected from the action creator. Then, both tests call the imported action creator with the user object as argument and compare the returned action to the expected action using the toEqual() matcher of Jest.



```
import { getFirebaseUser } from '__specs__/__helpers__/firebaseHelpers';
import * as actions from '../authActions';

describe('authAction creator', () => {
  it('firebaseUserStateChanged returns correct action when user is undefined', () =>
{
    const user = undefined;
    const correctAction: actions.FirebaseUserAuthStateChangedAction = {
      type: actions.AuthActionType.AUTH_FIREBASE_USER_STATE_CHANGED,
      payload: user,
    };
    expect(actions.firebaseUserStateChanged(user)).toEqual(correctAction);
  });

  it('firebaseUserStateChanged returns correct action when user is defined', () => {
    const user = getFirebaseUser();
    const correctAction: actions.FirebaseUserAuthStateChangedAction = {
      type: actions.AuthActionType.AUTH_FIREBASE_USER_STATE_CHANGED,
      payload: user,
    };
    expect(actions.firebaseUserStateChanged(user)).toEqual(correctAction);
  });
});
```

Figure 49. Test suite for the firebaseUserStateChanged() action creator



```
export const getFirebaseUser = (): firebase.User => {
    return {
        displayName: 'Tester',
        email: 'test@test.test',
        phoneNumber: '0',
        photoURL: null,
        providerId: 'providerId',
        isAnonymous: false,
        reauthenticateWithPhoneNumber: jest.fn(),
        metadata: {},
        providerData: [],
        uid: 'uid',
        delete: jest.fn(),
        emailVerified: true,
        getIdTokenResult: jest.fn(),
        getIdToken: jest.fn(),
        linkAndRetrieveDataWithCredential: jest.fn(),
        linkWithCredential: jest.fn(),
        linkWithPhoneNumber: jest.fn(),
        linkWithPopup: jest.fn(),
        linkWithRedirect: jest.fn(),
        reauthenticateAndRetrieveDataWithCredential: jest.fn(),
        reauthenticateWithCredential: jest.fn(),
        reauthenticateWithPopup: jest.fn(),
        reauthenticateWithRedirect: jest.fn(),
        refreshToken: "token",
        reload: jest.fn(),
        sendEmailVerification: jest.fn(),
        toJSON: jest.fn(),
        unlink: jest.fn(),
        updateEmail: jest.fn(),
        updatePassword: jest.fn(),
        updatePhoneNumber: jest.fn(),
        updateProfile: jest.fn(),
    };
}
```

Figure 50. getFirebaseUser() utility method

The rest of the action creators can be tested in the same way. The full test suite for the authentication action creators is provided in Appendix 2.

5.4 Example of testing a reducer

Reducers should be tested according to the official Redux documentation. The authReducer.ts file contains the reducer responsible for updating the state upon receiving any of the authentication actions. The file also contains the definition of the initial authentication state. Figure 51 shows the initial authentication state, while Figure 52 contains the source code of the authentication reducer.

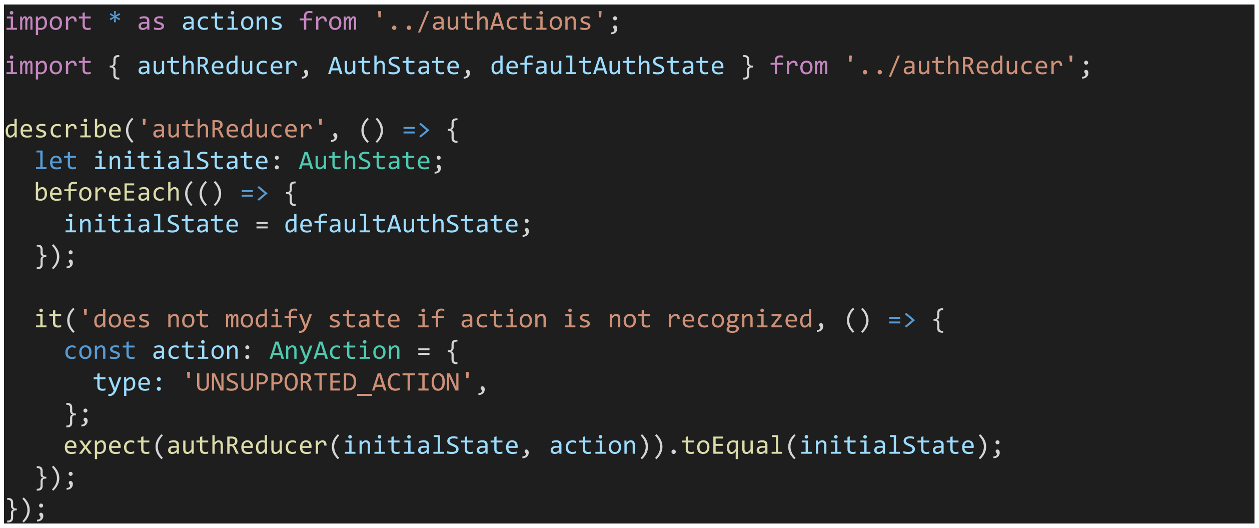
```
export interface AuthState {
  readonly firebaseUser: firebase.User | undefined;
  readonly appUser: User | undefined;
  readonly activeRole: UserRole | undefined;
  readonly firebaseError: firebase.auth.Error | undefined;
  readonly authenticating: boolean;
}

export const defaultAuthState: AuthState = {
  firebaseUser: undefined,
  appUser: undefined,
  activeRole: undefined,
  firebaseError: undefined,
  authenticating: false,
};
```

Figure 51. Initial authentication state

```
export const authReducer: Reducer<AuthState> = (
  state = defaultAuthState,
  action: AuthAction,
) => {
  switch (action.type) {
    case AuthActionType.AUTH_FIREBASE_USER_STATE_CHANGED: {
      return {
        ...state,
        firebaseUser: action.payload,
        firebaseError: undefined,
        authenticating: false,
      };
    }
    case AuthActionType.AUTH_APP_USER_INFO_UPDATED: {
      const appUser = action.payload as User;
      let activeRole;
      if (appUser) {
        if (appUser.systemAdmin) {
          activeRole = UserRole.SYSTEM_ADMIN;
        } else if (
          appUser.companies &&
          appUser.home &&
          appUser.companies[appUser.home] !== undefined
        ) {
          activeRole = appUser.companies[appUser.home];
        }
      }
      return {
        ...state,
        appUser: action.payload,
        activeRole,
      };
    }
    case AuthActionType.AUTH_FIREBASE_SET_ERROR: {
      return {
        firebaseError: action.payload,
        authenticating: false,
        ...state,
      };
    }
    case AuthActionType.AUTH_FIREBASE_SET_AUTHENTICATING: {
      return {
        authenticating: action.payload,
        ...state,
      };
    }
    default:
      return state;
  }
};
```

Figure 52. Authentication reducer

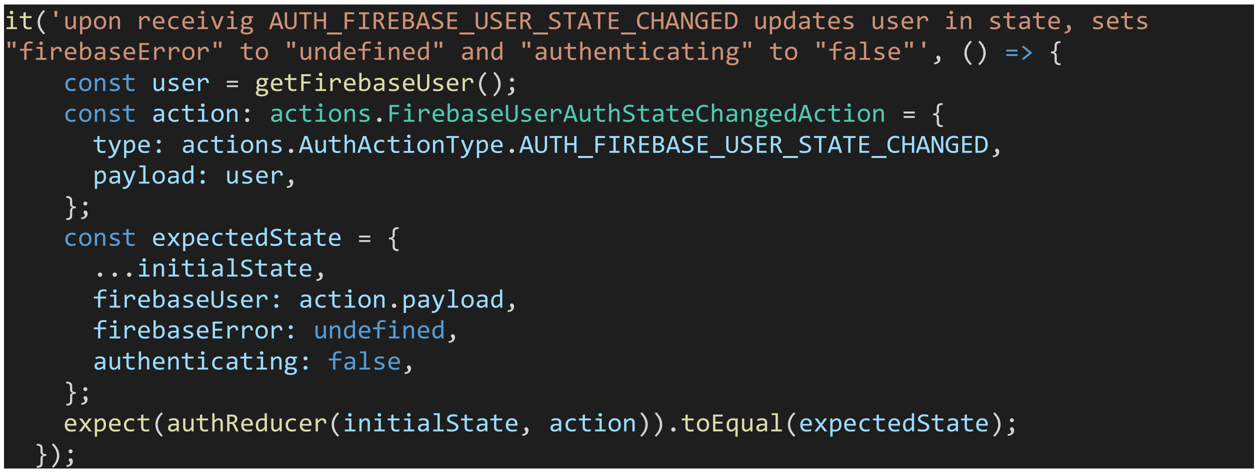
The reducer can now be tested in four steps defined in Chapter 4. First, a state object with the initial state must be set up. Second, a test should be written that verifies the reducer does not modify the state when receiving an action that it does not recognize. Figure 53 provides the setup and the first test for the authentication reducer. The beforeEach() hook of Jest is another spec-style feature originating from RSpec. The setup code inside beforeEach() is executed before each of the tests within the test suite. (Justice 2018.) For this reducer test, beforeEach() sets the initialState object to equal the defaultAuthState object imported from the authReducer.ts file. The first test verifies that the reducer does not modify the state if the supplied action is not recognized. An action object is set up with an ‘UNSUPPORTED_ACTION’ type – a type that is not included in the switch in the code of the reducer. The authReducer is then called with the initial state and the action as parameters and expected to return the original state object.



```
import * as actions from '../authActions';
import { authReducer, AuthState, defaultAuthState } from '../authReducer';

describe('authReducer', () => {
  let initialState: AuthState;
  beforeEach(() => {
    initialState = defaultAuthState;
  });

  it('does not modify state if action is not recognized, () => {
    const action: AnyAction = {
      type: 'UNSUPPORTED_ACTION',
    };
    expect(authReducer(initialState, action)).toEqual(initialState);
  });
});
```

Figure 53. Setup of the initial state and testing the authReducer with an unsupported action

Next, each action supported by the reducer must be tested. Figure 54 shows the test that verifies the reducer updates the state correctly upon receiving the AUTH_FIREBASE_USER_STATE_CHANGED action. The initial state for the test is set up in beforeEach(), while the rest of the setup is done within the code of the test. A user object is set up using the getFirebaseUser() utility method already used in the action creator test. Next, the correct action object and the state expected to be returned by the reducer are set up. Finally, the reducer is called with the initial state and the action as parameters and expected to return the correct state object.



```
it('upon receivig AUTH_FIREBASE_USER_STATE_CHANGED updates user in state, sets
"firebaseError" to "undefined" and "authenticating" to "false"', () => {
    const user = getFirebaseUser();
    const action: actions.FirebaseUserAuthStateChangedAction = {
      type: actions.AuthActionType.AUTH_FIREBASE_USER_STATE_CHANGED,
      payload: user,
    };
    const expectedState = {
      ...initialState,
      firebaseUser: action.payload,
      firebaseError: undefined,
      authenticating: false,
    };
    expect(authReducer(initialState, action)).toEqual(expectedState);
  });
```

Figure 54. Testing the behaviour of the reducer upon receiving a supported action

The rest of the actions supported by the reducer can be tested in the same way. The full test suite for the authentication reducer is provided in Appendix 3.

5.5 Examples of testing React components

This section provides examples of testing React components according to the strategy. It uses two components for that purpose: AccountForm and AccountContainer. The AccountForm component allows logged in users to view their account data and edit certain fields (currently, only the display name is editable). The AccountForm component is wrapped into the AccountContainer component. The AccountForm is a stateless component that receives three props: the account data, a change event handler and a submit event handler. It is responsible for displaying the account data and triggering the event handlers with correct data. Figure 55 shows how the two components are displayed to the user.

The AccountContainer component:

- is responsible for passing account data to AccountForm.
- contains the implementation of the event handlers triggered by AccountForm.
- is connected to Redux to get the global authentication state.
- interacts with Firebase to get the data for the currently logged in account.
- displays the avatar of the account.

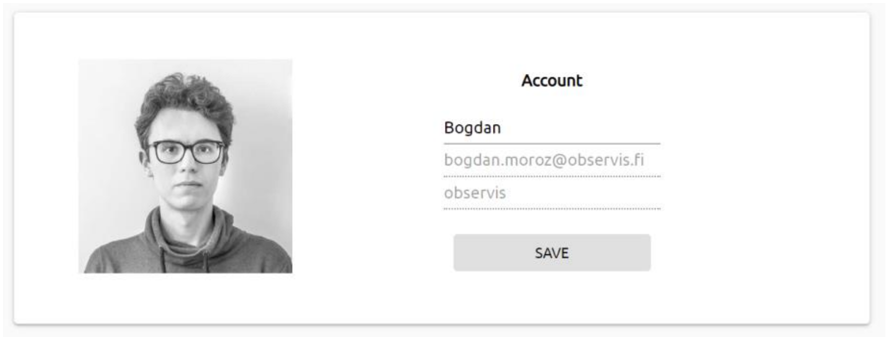


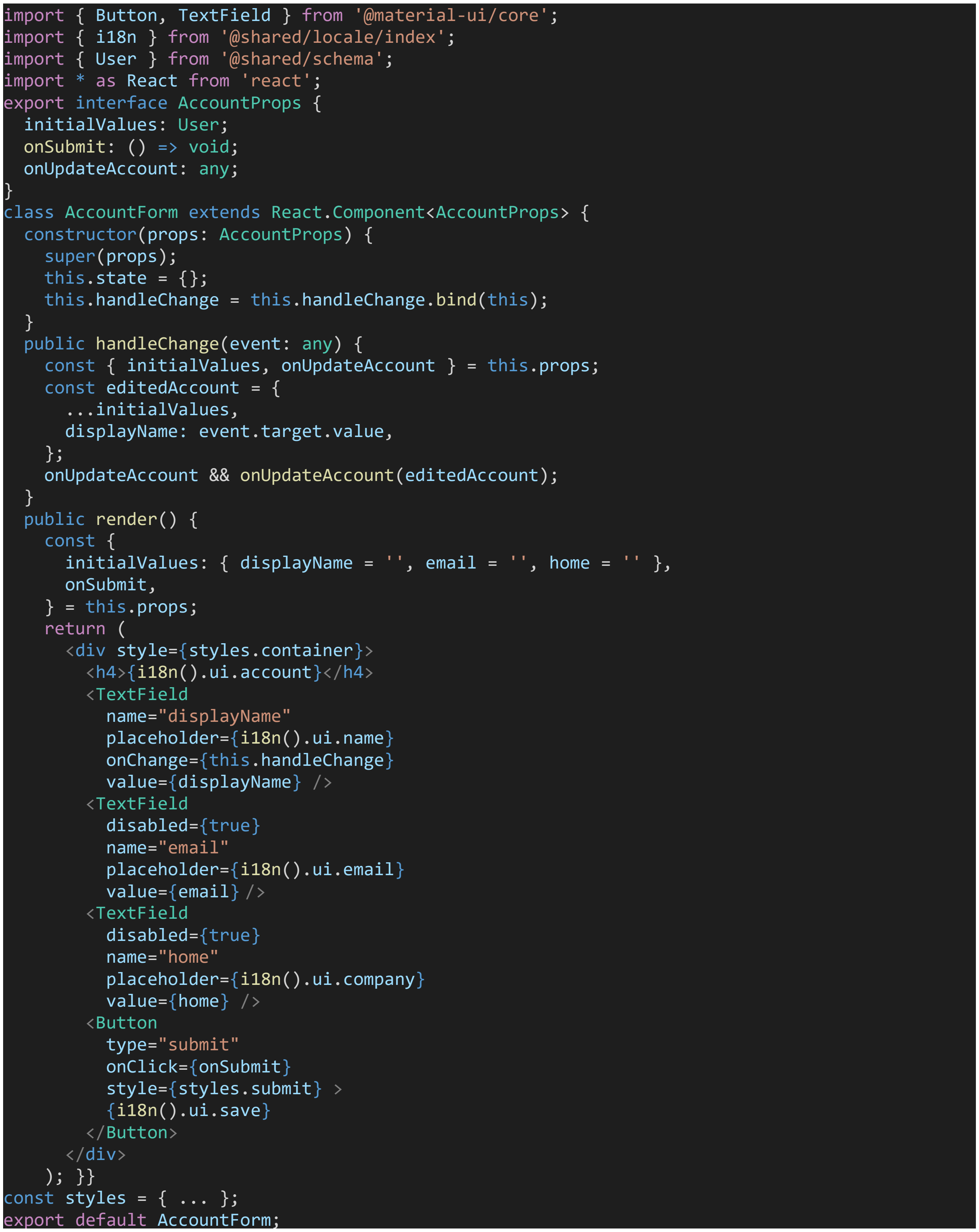
Figure 55. The AccountForm component wrapped into AccountContainer

The components can now be tested according to the strategy.

5.5.1 Testing a stateless component

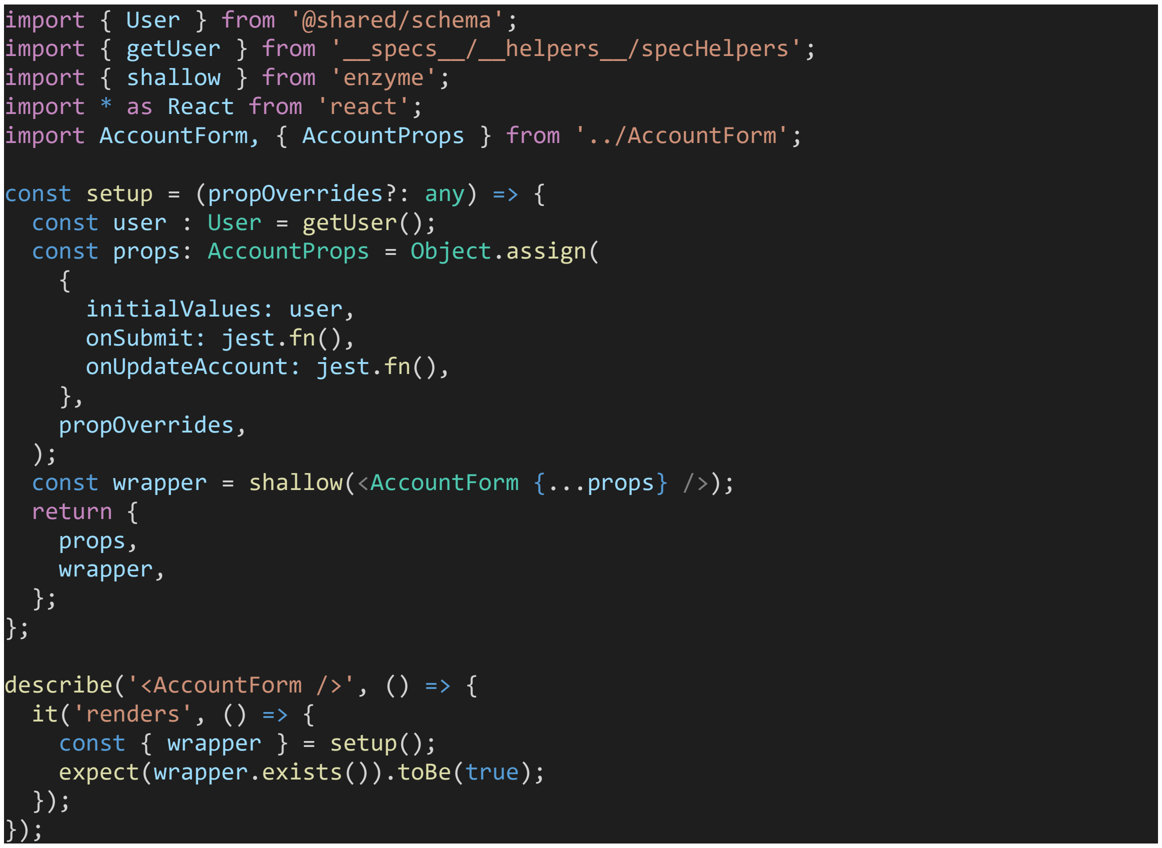
Figure 56 contains the source code of the AccountForm component. The styles object at the bottom is redacted from the figure as the component style is not being tested. The first step of testing the component is to verify that it renders. Figure 57 displays the code of the smoke test as well as the code of the setup function for the test. As established in Chapter 4, the setup function is used to remove duplication within test code and make the tests more expressive. The usefulness of the function will become increasingly apparent as more tests are written. In Figure 57, the setup function accepts an optional parameter propOverrides. The function contains a declaration of the default props object that is overwritten if the propOverrides argument is supplied. The Object.assign() method is used to copy enumerable properties from the propOverrides object into the props object. This way it is possible to only override certain fields of the default props without supplying the entire required props object to the setup function. The setup function uses a utility method getUser() to generate a default object of type User. The user object is set to be the default value of the initialValues prop. The onSubmit and onUpdateAccount props are replaced with mock functions to spy on the way they are called.

Next, the component is shallow rendered with the resolved props. The setup function returns an object containing both the props and the shallow wrapper of the component. The first test then calls the setup function and assigns the wrapper returned by the function to a local constant. It uses the destructuring assignment syntax in order to only take the wrapper from the object returned by the setup function, as the props are not required for the rendering test. The test then verifies that the shallow wrapper contains at least one node by calling `wrapper.exists()` and expecting it to be true.



```
import { Button, TextField } from '@material-ui/core';
import { i18n } from '@shared/locale/index';
import { User } from '@shared/schema';
import * as React from 'react';
export interface AccountProps {
  initialValues: User;
  onSubmit: () => void;
  onUpdateAccount: any;
}
class AccountForm extends React.Component<AccountProps> {
  constructor(props: AccountProps) {
    super(props);
    this.state = {};
    this.handleChange = this.handleChange.bind(this);
  }
  public handleChange(event: any) {
    const { initialValues, onUpdateAccount } = this.props;
    const editedAccount = {
      ...initialValues,
      displayName: event.target.value,
    };
    onUpdateAccount && onUpdateAccount(editedAccount);
  }
  public render() {
    const {
      initialValues: { displayName = '', email = '', home = '' },
      onSubmit,
    } = this.props;
    return (
      <div style={styles.container}>
        <h4>{i18n().ui.account}</h4>
        <TextField
          name="displayName"
          placeholder={i18n().ui.name}
          onChange={this.handleChange}
          value={displayName} />
        <TextField
          disabled={true}
          name="email"
          placeholder={i18n().ui.email}
          value={email} />
        <TextField
          disabled={true}
          name="home"
          placeholder={i18n().ui.company}
          value={home} />
        <Button
          type="submit"
          onClick={onSubmit}
          style={styles.submit} >
          {i18n().ui.save}
        </Button>
      </div>
    ); }}
const styles = { ... };
export default AccountForm;
```

Figure 56. Source code of the AccountForm component before any tests are written

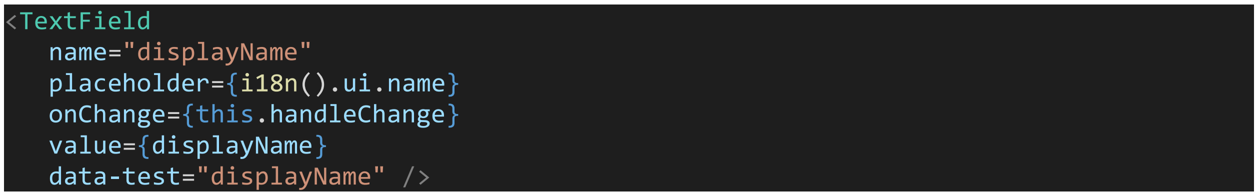


```
import { User } from '@shared/schema';
import { getUser } from '__specs__/__helpers__/specHelpers';
import { shallow } from 'enzyme';
import * as React from 'react';
import AccountForm, { AccountProps } from '../AccountForm';

const setup = (propOverrides?: any) => {
  const user : User = getUser();
  const props: AccountProps = Object.assign(
    {
      initialValues: user,
      onSubmit: jest.fn(),
      onUpdateAccount: jest.fn(),
    },
    propOverrides,
  );
  const wrapper = shallow(<AccountForm {...props} />);
  return {
    props,
    wrapper,
  };
};

describe('<AccountForm />', () => {
  it('renders', () => {
    const { wrapper } = setup();
    expect(wrapper.exists()).toBe(true);
  });
});
```

Figure 57. Setup function and a smoke test for AccountForm

The second step of testing the component is to test each prop. Two of the props that AccountForm receives are event handlers, they will be tested separately during step four. The initialValues prop is an object of type User that contains information about the user account. Three properties of initialValues are displayed in three TextField components. These properties are displayName, email and home. A separate test should be written to verify that each of these properties is displayed in the correct text field. Each test will find the correct TextField node in the component's render tree and verify that the value prop of the TextField matches the right property of initialValues.

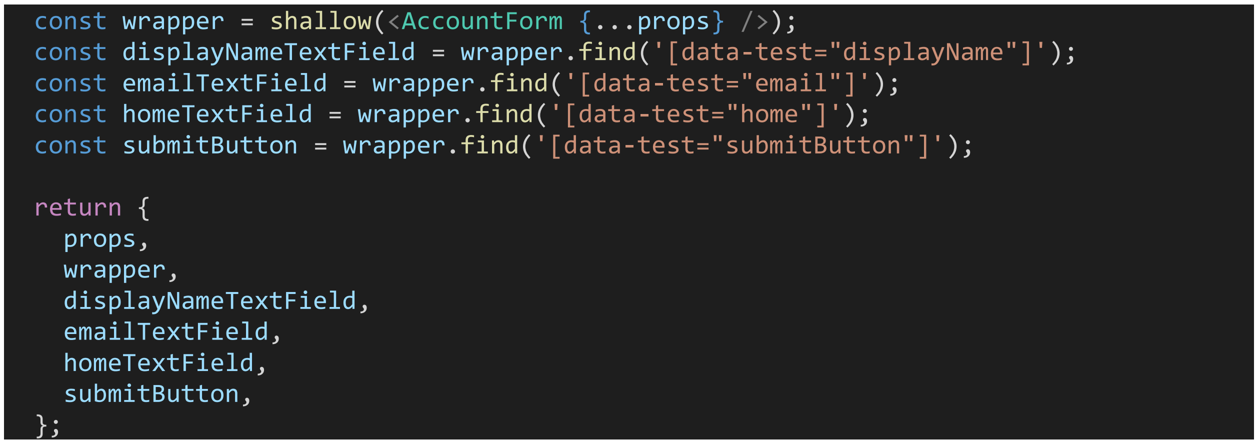
In order to easily find the right TextField nodes with the find() selector of Enzyme, data attributes can be added to every text field. Dodds (2017a) recommends using data attributes to make the tests resilient to future changes. The layout of the component or the class names of the nodes can be modified, but as long as the data attributes remain intact the tests are less likely to break. (Dodds 2017a.) Figure 58 shows the TextField component for displayName with a data-test attribute “displayName” added. Data-test attributes are also added to the email and home TextField components, as well as the submit button.



```
<TextField
   name="displayName"
   placeholder={i18n().ui.name}
   onChange={this.handleChange}
   value={displayName}
   data-test="displayName" />
```

Figure 58. TextField component with a data-test attribute

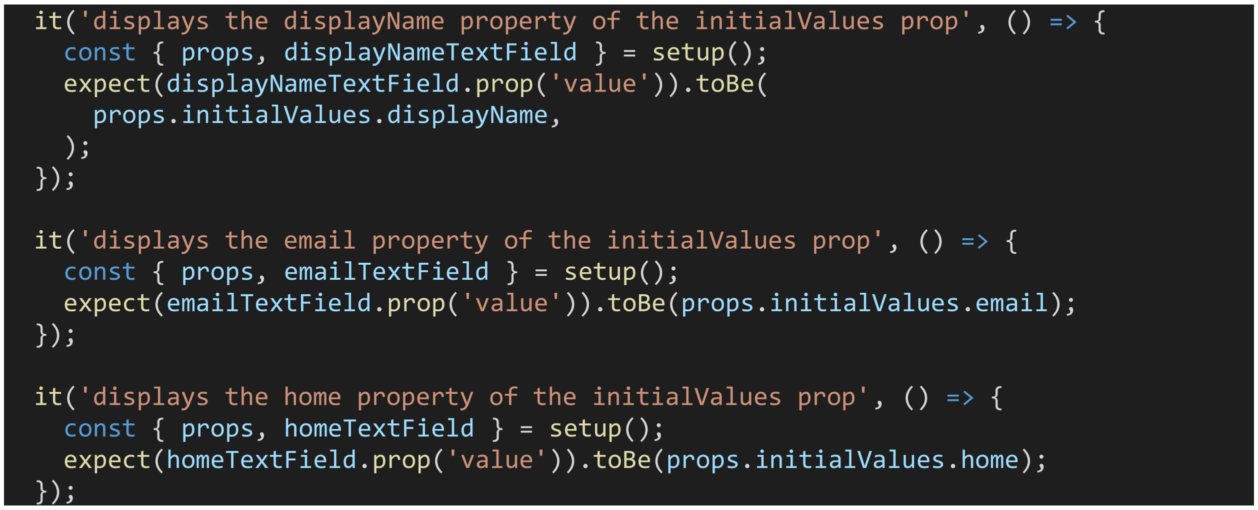
The setup function can now be updated to find and return each of the text fields, as shown in Figure 59. Finding the nodes within the setup function has an advantage over finding the nodes within each test in cases when the same node needs to be used in multiple tests. In the case of the AccountForm test suite, the displayNameTextField node will be used again when testing the change event handler. The email and home nodes are currently only used in one test each. However, if the functionality of the component is expanded in the future and change handlers are added to the email and home text fields, the tests will require no restructuring.



```
  const wrapper = shallow(<AccountForm {...props} />);
  const displayNameTextField = wrapper.find('[data-test="displayName"]');
  const emailTextField = wrapper.find('[data-test="email"]');
  const homeTextField = wrapper.find('[data-test="home"]');
  const submitButton = wrapper.find('[data-test="submitButton"]');

  return {
    props,
    wrapper,
    displayNameTextField,
    emailTextField,
    homeTextField,
    submitButton,
  };
```

Figure 59. Updated setup function finds and returns nodes of the shallow wrapper

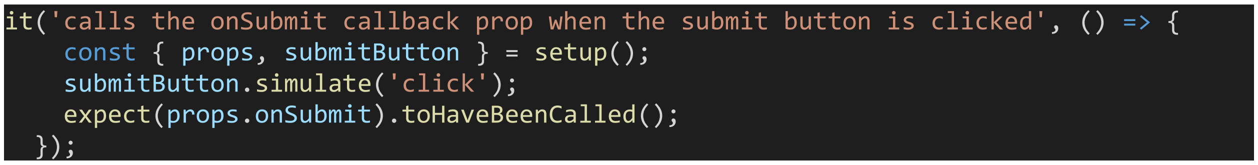
Figure 60 presents the tests for each of the initialValues properties. In each test, the props object and the found TextField node are returned from the setup function. The value of the ‘value’ prop of each TextField is then compared against the corresponding initialValues property.



```
  it('displays the displayName property of the initialValues prop', () => {
    const { props, displayNameTextField } = setup();
    expect(displayNameTextField.prop('value')).toBe(
      props.initialValues.displayName,
    );
  });

  it('displays the email property of the initialValues prop', () => {
    const { props, emailTextField } = setup();
    expect(emailTextField.prop('value')).toBe(props.initialValues.email);
  });

  it('displays the home property of the initialValues prop', () => {
    const { props, homeTextField } = setup();
    expect(homeTextField.prop('value')).toBe(props.initialValues.home);
  });
```

Figure 60. Testing that the initialValues properties are displayed in correct text fields

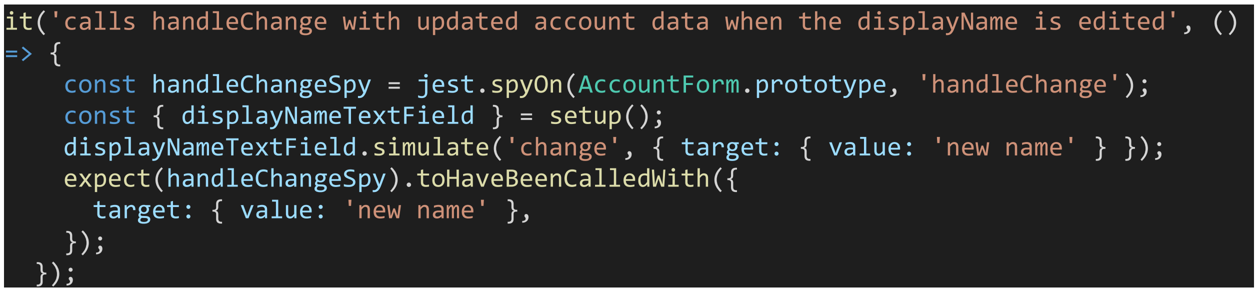
Step 3 is to test each property of the component's state. AccountForm is a stateless component, so this step is not required. Step 4 is to test the event handlers of the component. Figure 61 shows the test of the submit event handler. The component is setup and the props object and the submit button wrapper are returned from the setup function. A click event is simulated on the wrapper using the `simulate('click')` method of Enzyme. The onSubmit prop is then expected to have been called. The behaviour of the submit handler cannot be tested, as the AccountForm component receives the handler as a prop from the parent component AccountContainer. The behaviour of the handler will be tested in section 5.5.2 when specifying the behaviour of AccountContainer.



```
it('calls the onSubmit callback prop when the submit button is clicked', () => {
    const { props, submitButton } = setup();
    submitButton.simulate('click');
    expect(props.onSubmit).toHaveBeenCalled();
  });
```

Figure 61. Testing the submit handler of the AccountForm component

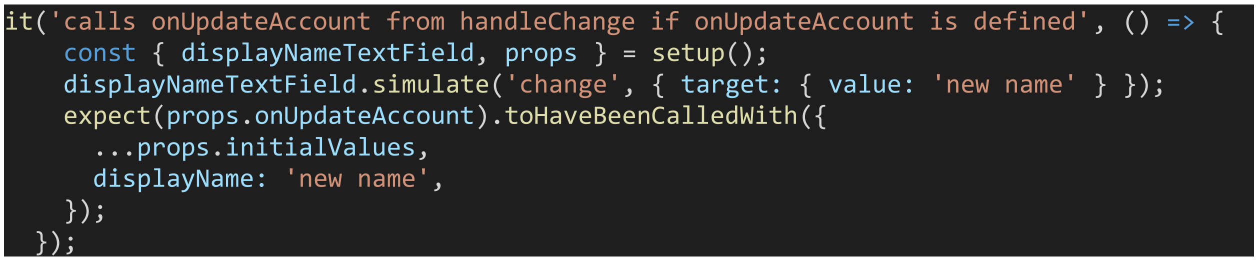
Figure 62 contains the test of the change event handler of the displayName text field. First, a spy is created on the handleChange method of the AccountForm component. Then, the component is setup and the displayName TextField node is returned from the setup function. A change event is then simulated on the node using the `simulate('change')` method of Enzyme. The value of the target of the event is supplied as the second argument to the simulate function. Finally, the spy is expected to have been called with the value of the event target.



```
it('calls handleChange with updated account data when the displayName is edited', ()
=> {
  const handleChangeSpy = jest.spyOn(AccountForm.prototype, 'handleChange');
  const { displayNameTextField } = setup();
  displayNameTextField.simulate('change', { target: { value: 'new name' } });
  expect(handleChangeSpy).toHaveBeenCalledWith({
   target: { value: 'new name' },
  });
 });
```

Figure 62. Testing the change event handler of the displayName text field

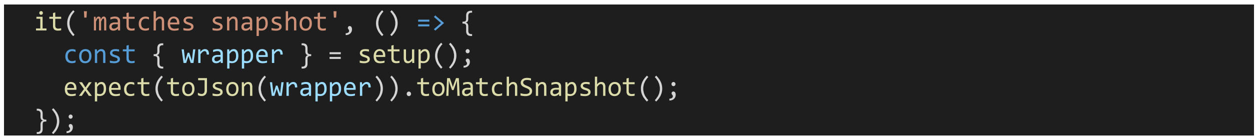
The AccountForm component includes the implementation of the handleChange method. Therefore, it is not enough to test that the method has been called with correct parameters. The behaviour of the handler method needs to be tested as well. The test is presented in figure 63. Once again, a change event is simulated on displayName text field. However, this time the onUpdateAccount prop of AccountForm is expected to have been called with the correctly updated account data.



```
it('calls onUpdateAccount from handleChange if onUpdateAccount is defined', () => {
  const { displayNameTextField, props } = setup();
  displayNameTextField.simulate('change', { target: { value: 'new name' } });
  expect(props.onUpdateAccount).toHaveBeenCalledWith({
   ...props.initialValues,
   displayName: 'new name',
  });
 });
```

Figure 63. Testing the behaviour of the handleChange() method

AccountForm component does not make use of lifecycle methods and does not include conditional rendering, therefore steps 5 and 6 are not needed. The final test required by the strategy is a snapshot test. The snapshot test is presented in Figure 64. The enzyme-to-json library is used to make the snapshot more readable.



```
 it('matches snapshot', () => {
  const { wrapper } = setup();
  expect(toJson(wrapper)).toMatchSnapshot();
 });
```

Figure 64. Snapshot testing AccountForm

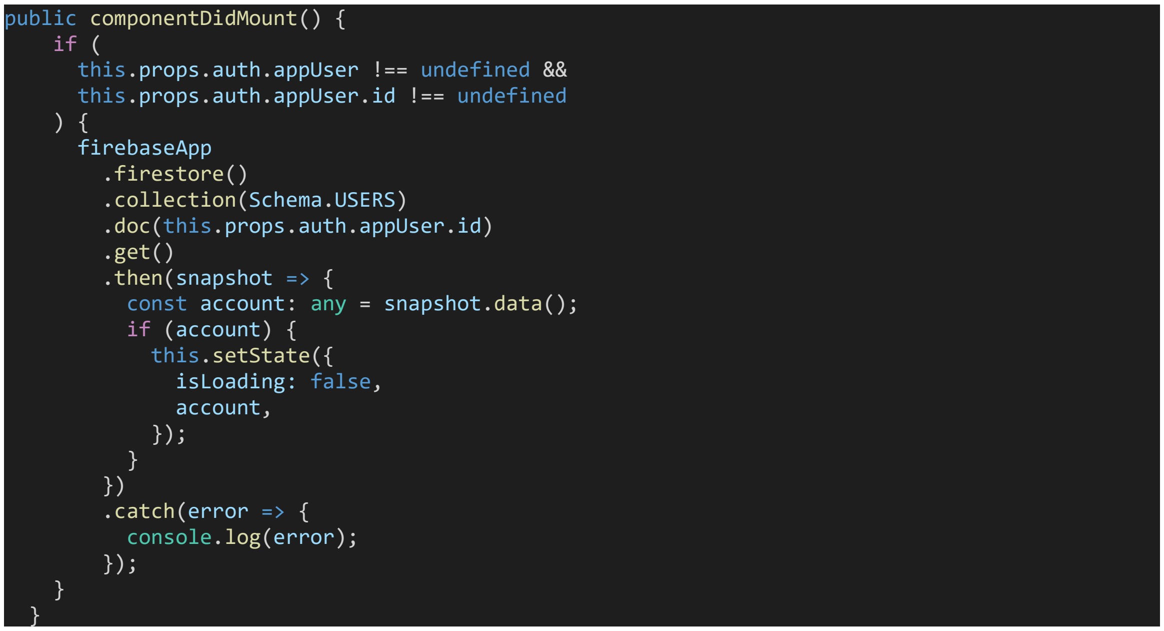
The test suite of the AccountForm component is provided in full in Appendix 4.

5.5.2 Testing a stateful component connected to Redux

Next, the AccountContainer component is tested. Figure 65 contains the source code of the AccountContainer component with the componentDidMount() and render() methods redacted. Those methods are provided in full in figures 66 and 67 respectively. The styles object is also redacted as the styles of the component are not being tested.

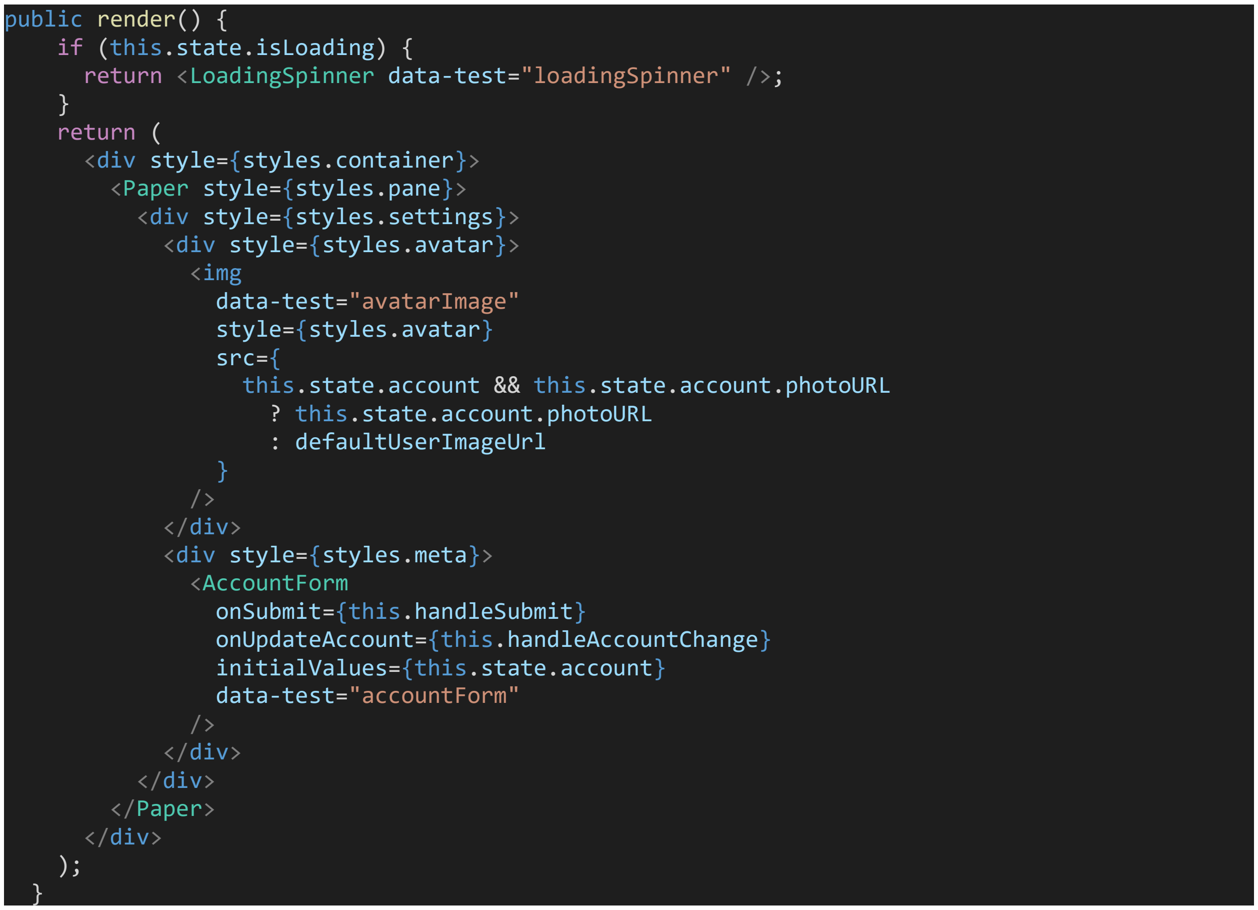
```
export interface AccountContainerProps extends Partial<DispatchProp<any>> {
  auth: any;
}
interface State {
  isLoading: boolean;
  account: User;
}
export class AccountContainer extends React.Component<
  AccountContainerProps,
  State
> {
  constructor(props: AccountContainerProps) {
    super(props);
    this.state = {
      isLoading: true,
      account: {
        id: '',
        displayName: '',
        email: '',
        photoURL: '',
        home: '',
      },
    };
  }
  public componentDidMount() { ... }
  public render() { ... }

  private handleAccountChange = (account: ShortUserInfo) => {
    this.setState({ account });
  };
  private handleSubmit = () => {
    firebaseApp
      .firestore()
      .collection(Schema.USERS)
      .doc(this.state.account.id)
      .update(this.state.account);
  };
}
const mapStateToProps = (
  state: ApplicationState,
  ownProps: Partial<AccountContainerProps>,
) => {
  return {
    ...ownProps,
    auth: state.auth,
  };
};
const styles = { ... };
export default withRouter<any>(connect<any>(mapStateToProps)(AccountContainer));
```

Figure 65. AccountContainer component



```
public componentDidMount() {
    if (
      this.props.auth.appUser !== undefined &&
      this.props.auth.appUser.id !== undefined
    ) {
      firebaseApp
        .firestore()
        .collection(Schema.USERS)
        .doc(this.props.auth.appUser.id)
        .get()
        .then(snapshot => {
          const account: any = snapshot.data();
          if (account) {
            this.setState({
              isLoading: false,
              account,
            });
          }
        })
        .catch(error => {
          console.log(error);
        });
    }
  }
```

Figure 66. The componentDidMount() method of the AccountContainer component



```
public render() {
    if (this.state.isLoading) {
      return <LoadingSpinner data-test="loadingSpinner" />;
    }
    return (
      <div style={styles.container}>
        <Paper style={styles.pane}>
          <div style={styles.settings}>
            <div style={styles.avatar}>
              <img
                data-test="avatarImage"
                style={styles.avatar}
                src={
                  this.state.account && this.state.account.photoURL
                    ? this.state.account.photoURL
                    : defaultUserImageUrl
                }
              />
            </div>
            <div style={styles.meta}>
              <AccountForm
                onSubmit={this.handleSubmit}
                onUpdateAccount={this.handleAccountChange}
                initialValues={this.state.account}
                data-test="accountForm"
              />
            </div>
          </div>
        </Paper>
      </div>
    );
  }
```

Figure 67. The render() method of the AccountContainer component

The AccountContainer component is wrapped in a Redux state provider and a Router. The component uses the mapStateToProps() function of Redux in order to get the auth property of the global application state and maps it to its own auth prop. AccountContainer is a stateful component. It contains two variables in its state: an “isLoading” boolean representing whether the component is loading data from the database, and an “account” object of type User that represents the user account data fetched from the database. The constructor method initializes the component with the default state – “isLoading” is true and the “account” prop has all the fields as empty strings.

The lifecycle method componentDidMount() is called immediately after the component is added to the DOM. AccountContainer fetches the account information from the database from its componentDidMount(), as shown in Figure 66. The component tries to get a document from the “users” collection with the id of the user that is currently logged in. The call to `firebaseApp() .firestore().collection(Shema.USERS).doc(this.props.auth .appUser.id).get()` returns a promise. That promise is resolved when a document snapshot with user data is returned from the database. The returned user data is then set to the state of the component, and the “isLoading” state variable is changed to false.

Finally, the render method in Figure 67 returns a LoadingSpinner component until the account data is loaded from the database. When the account is fetched from the database and “isLoading” is set to “false”, the render method returns a user avatar and the AccountForm component with the account data. The avatar displays the user account avatar if the account data contains a photoURL property and a default image otherwise. The conditions affecting the rendering of AccountContainer are presented in Figure 68.

As this component relies on Firebase database when rendering, the setup function for the tests needs to be adjusted to setup a mock for the Firebase database. The setup function for AccountContainer is displayed in Figure 69. Ts-Mock-Firebase documentation (2019) provides instructions for the setup of a mock Firebase application with a mock database.

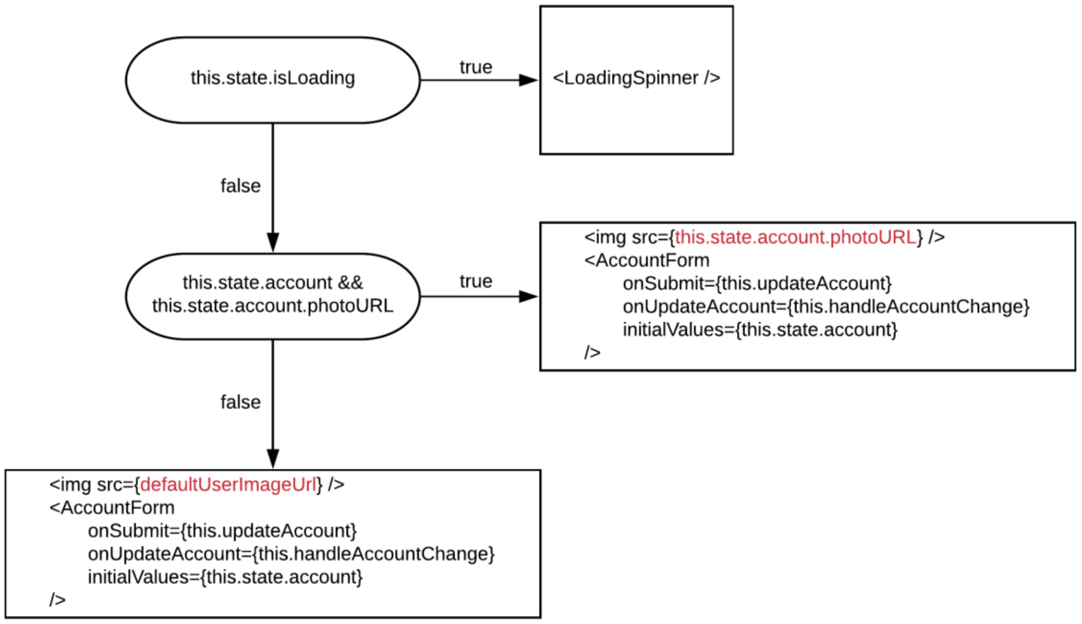
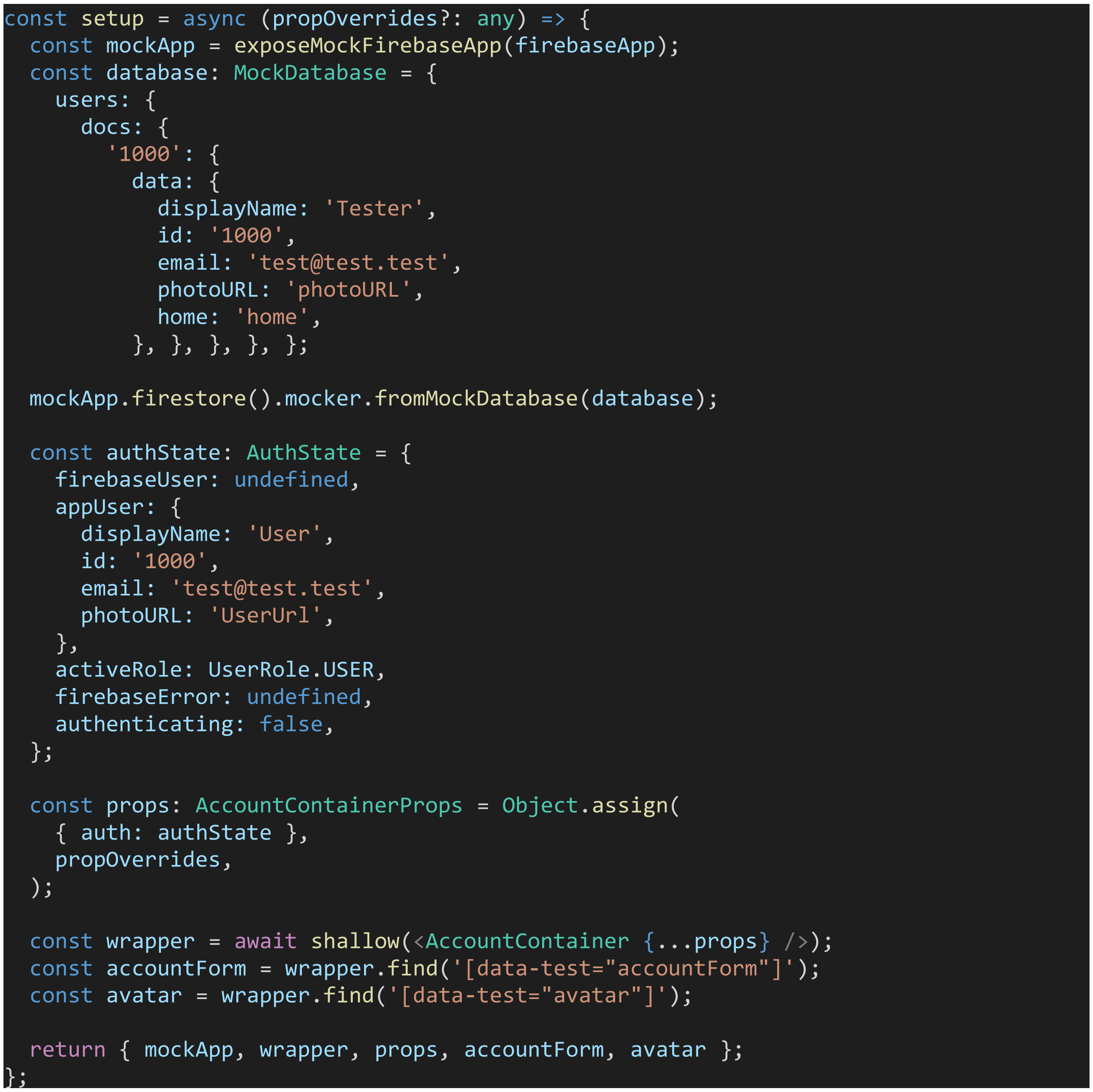

this.state.isLoading
true
<LoadingSpinner />
false
this.state.account &&
this.state.account.photoURL
true
<img src={this.state.account.photoURL} />
<AccountForm
onSubmit={this.updateAccount}
onUpdateAccount={this.handleAccountChange}
initialValues={this.state.account}
/>
false
<img src={defaultUserImageUrl} />
<AccountForm
onSubmit={this.updateAccount}
onUpdateAccount={this.handleAccountChange}
initialValues={this.state.account}
/>

Figure 68. Conditions affecting the rendering of AccountContainer

When Firebase is configured for a project, the entry point for the Firebase SDK is setup using the FirebaseApp class (Google 2019). In MOWO dashboard, the firebase application is setup in a firebaseApp.ts file located at the root of the src directory. The Firebase library itself has already been manually mocked when setting up the environment (Section 5.1), and the firebaseApp already uses the mock library to initialize the app. However, in order to set up the database of firebaseApp with custom data required for tests, the mock interface has to be exposed. This is achieved on the first line of the setup function in Figure 69. The firebaseApp module is imported from the src directory, and the exposeMockFirebaseApp() method of Ts-Mock-Firebase is called with the firebaseApp as the argument. The method returns an instance of firebaseApp with the interface exposed – the constant mockApp. Now, custom data can be placed into the database. (Ts-Mock-Firebase 2019.)

The object "database" of type MockDatabase defines the fake database that will be used in tests. The object contains one collection (users) with one document (one user). The Firebase application is configured to use that fake database by calling `mockApp.firestore().mocker.fromMockDatabase(database).` (Ts-Mock-Firebase 2019.)



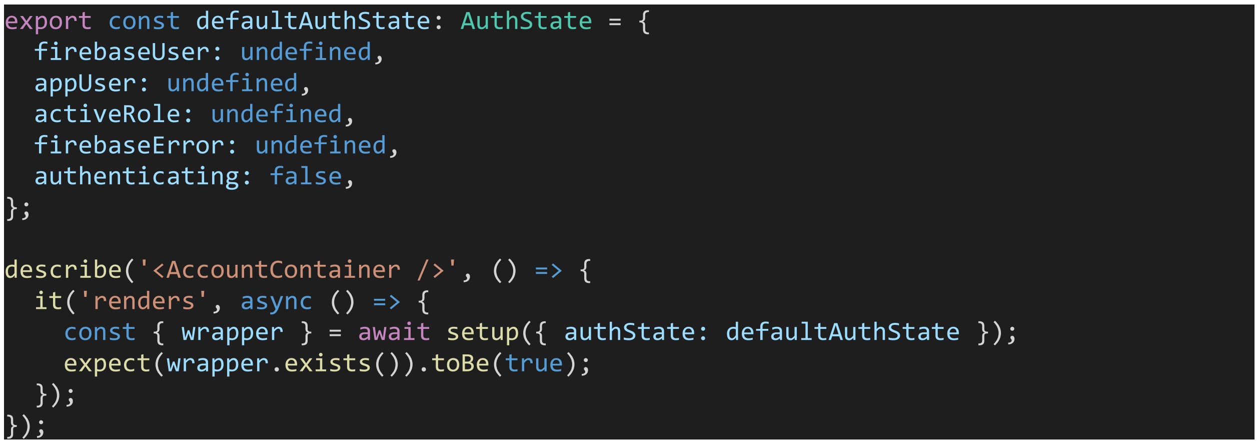
```
const setup = async (propOverrides?: any) => {
  const mockApp = exposeMockFirebaseApp(firebaseApp);
  const database: MockDatabase = {
    users: {
      docs: {
        '1000': {
          data: {
            displayName: 'Tester',
            id: '1000',
            email: 'test@test.test',
            photoURL: 'photoURL',
            home: 'home',
          }, }, }, }, };

  mockApp.firestore().mocker.fromMockDatabase(database);

  const authState: AuthState = {
    firebaseUser: undefined,
    appUser: {
      displayName: 'User',
      id: '1000',
      email: 'test@test.test',
      photoURL: 'UserUrl',
    },
    activeRole: UserRole.USER,
    firebaseError: undefined,
    authenticating: false,
  };

  const props: AccountContainerProps = Object.assign(
    { auth: authState },
    propOverrides,
  );

  const wrapper = await shallow(<AccountContainer {...props} />);
  const accountForm = wrapper.find('[data-test="accountForm"]');
  const avatar = wrapper.find('[data-test="avatar"]');

  return { mockApp, wrapper, props, accountForm, avatar };
};
```

Figure 69. The setup function for AccountContainer tests

Next in the setup function the default props of the component are set up. The same approach is used as with the AccountForm setup function. The setup function defines the default props and accepts prop overrides in cases when the default props need to be modified for any given test. The default value for the auth prop of AccountContainer is configured to equal the authState object, which has the property appUser defined. The id of the appUser is equal to the id of the user that was placed in the mock database. This way, by default when AccountForm is shallow rendered in the setup function it will find the user in the database from componentDidMount() and place it into state.

Finally, the setup function shallow renders the component, finds the AccountForm component and the avatar component using data attributes, and returns the wrapper, the found nodes, props and mockApp. In this case the `shallow()` method is called like an asynchronous function. As mentioned above, when the AccountForm component fetches the user from the database in componentDidMount(), it waits for a promise to resolve in order to set the account data to state. Calling the `shallow()` method with the `await` operator makes the function wait for the promise to resolve. Since `await` is used within the setup function, the setup function is made asynchronous (the `async` operator on the first line of Figure 69).

Now that the setup function is complete, the component can be tested according to the strategy. First, component rendering is tested. The test is presented in Figure 70. The wrapper is set up with defaultAuthState as the prop. This is the default authentication state imported into the test from authReducer. As the test uses the asynchronous setup function, the test must be asynchronous as well – hence the `async` operator is added to the test function.

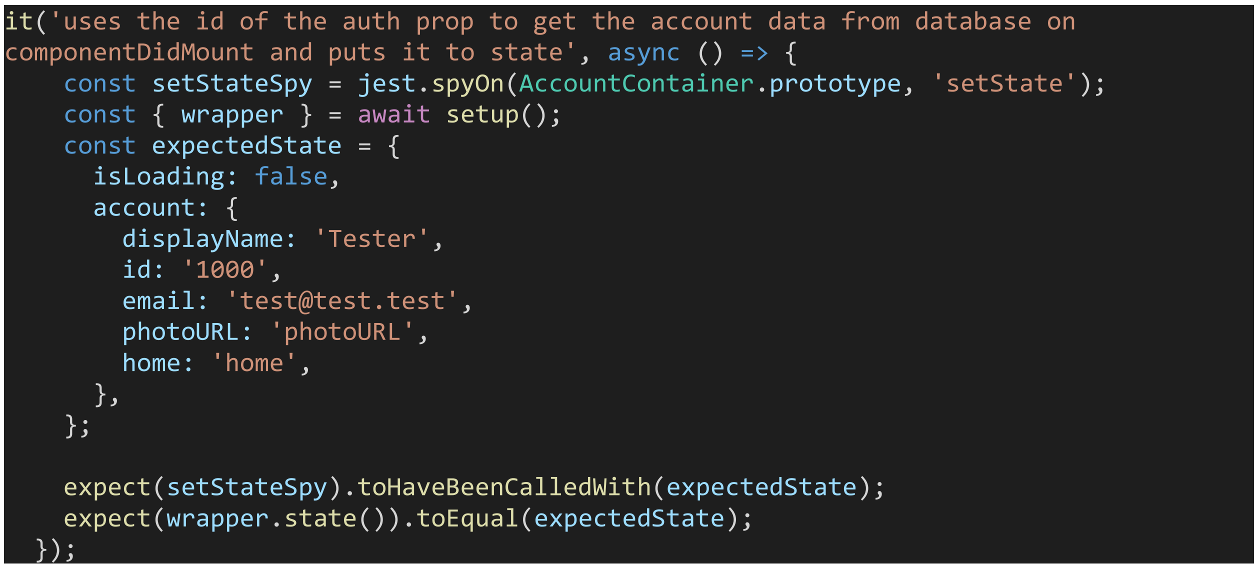


```
export const defaultAuthState: AuthState = {
  firebaseUser: undefined,
  appUser: undefined,
  activeRole: undefined,
  firebaseError: undefined,
  authenticating: false,
};

describe('<AccountContainer />', () => {
  it('renders', async () => {
    const { wrapper } = await setup({ authState: defaultAuthState });
    expect(wrapper.exists()).toBe(true);
  });
});
```

Figure 70. Smoke testing AccountContainer

Step 2 is to test each prop of the component. The AccountContainer only has one prop: auth. The prop contains a copy of the global authentication state. The only place where the prop is used by the component is within componentDidMount(). Because of this, step 2 can be combined with step 5 – testing component lifecycle methods. The first test for the auth prop and the componentDidMount()

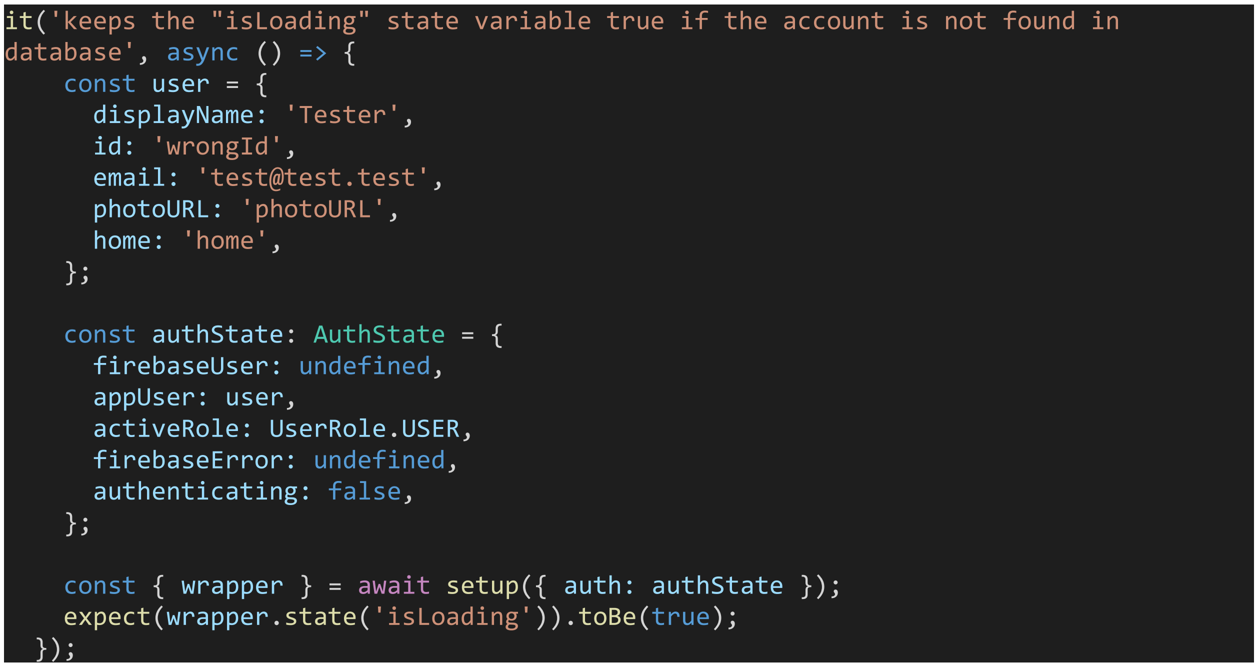
lifecycle method is presented in Figure 71. The test verifies that AccountConainer uses the id of the auth prop to get the account data from the database in componentDidMount() and puts it to state. The test begins by creating a spy on the setState method of AccountContainer. Then, the wrapper is setup with the default props. The state of the wrapper is then expected to equal the expectedState object where “isLoading” is false and “account” is equal to the account stored in the mock database. The spy is also expected to have been called with the expectedState. This test contains two assertions to demonstrate different ways the update of the state can be tested. Only one of these assertions is necessary for the test to function. It would be enough to keep the second assertion and remove the spy and the first assertion.



```
it('uses the id of the auth prop to get the account data from database on
componentDidMount and puts it to state', async () => {
    const setStateSpy = jest.spyOn(AccountContainer.prototype, 'setState');
    const { wrapper } = await setup();
    const expectedState = {
      isLoading: false,
      account: {
        displayName: 'Tester',
        id: '1000',
        email: 'test@test.test',
        photoURL: 'photoURL',
        home: 'home',
      },
    };

    expect(setStateSpy).toHaveBeenCalledWith(expectedState);
    expect(wrapper.state()).toEqual(expectedState);
  });
```

Figure 71. Testing that AccountContainer gets the user from database by auth.id

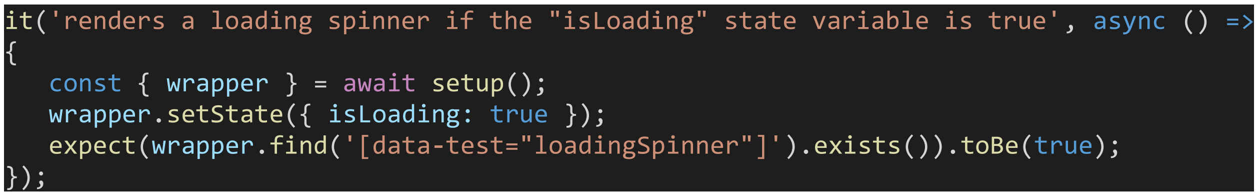
The second test for the auth prop and the componentDidMount() lifecycle method is presented in Figure 72. The test verifies that the component does not update the initial state and keeps isLoading variable “true” if the user account is not found in the database. The test sets up a user object with an id that does not exist in the mock database. The user is then placed into an authState prop that is passed to the setup function to override the default value of the auth prop. The shallow wrapper returned by the setup function is expected to still have the “isLoading” state variable true. This means that the component never stops loading if the user is not found in the database.



```
it('keeps the "isLoading" state variable true if the account is not found in
database', async () => {
    const user = {
      displayName: 'Tester',
      id: 'wrongId',
      email: 'test@test.test',
      photoURL: 'photoURL',
      home: 'home',
    };

    const authState: AuthState = {
      firebaseUser: undefined,
      appUser: user,
      activeRole: UserRole.USER,
      firebaseError: undefined,
      authenticating: false,
    };

    const { wrapper } = await setup({ auth: authState });
    expect(wrapper.state('isLoading')).toBe(true);
  });
```

Figure 72. Testing that AccountContainer does not modify the default value of “isLoading” if the user is not found in the database

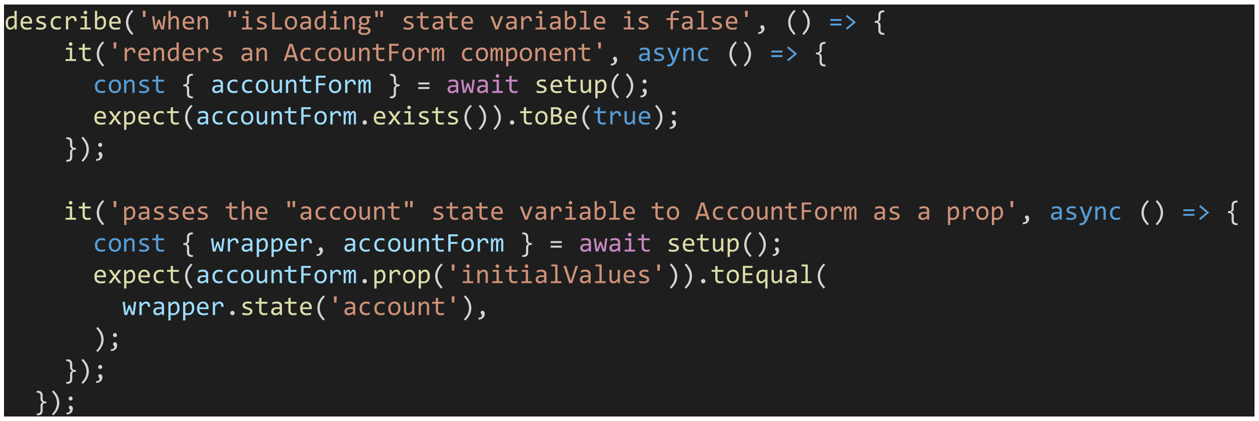
Step 3 is to test each state variable of the component. AccountContainer has two variables in its state: “isLoading” and “account”. The test in Figure 73 verifies that the component renders a LoadingSpinner component if the “isLoading” variable is true. The test sets up the wrapper with the default props, updates the state of the wrapper so that “isLoading” is true, finds the LoadingSpinner component by the data-test attribute and expects it to exist.



```
it('renders a loading spinner if the "isLoading" state variable is true', async () =>
{
   const { wrapper } = await setup();
   wrapper.setState({ isLoading: true });
   expect(wrapper.find('[data-test="loadingSpinner"]').exists()).toBe(true);
});
```

Figure 73. Testing that a LoadingSpinner is displayed when “isLoading” is true in state

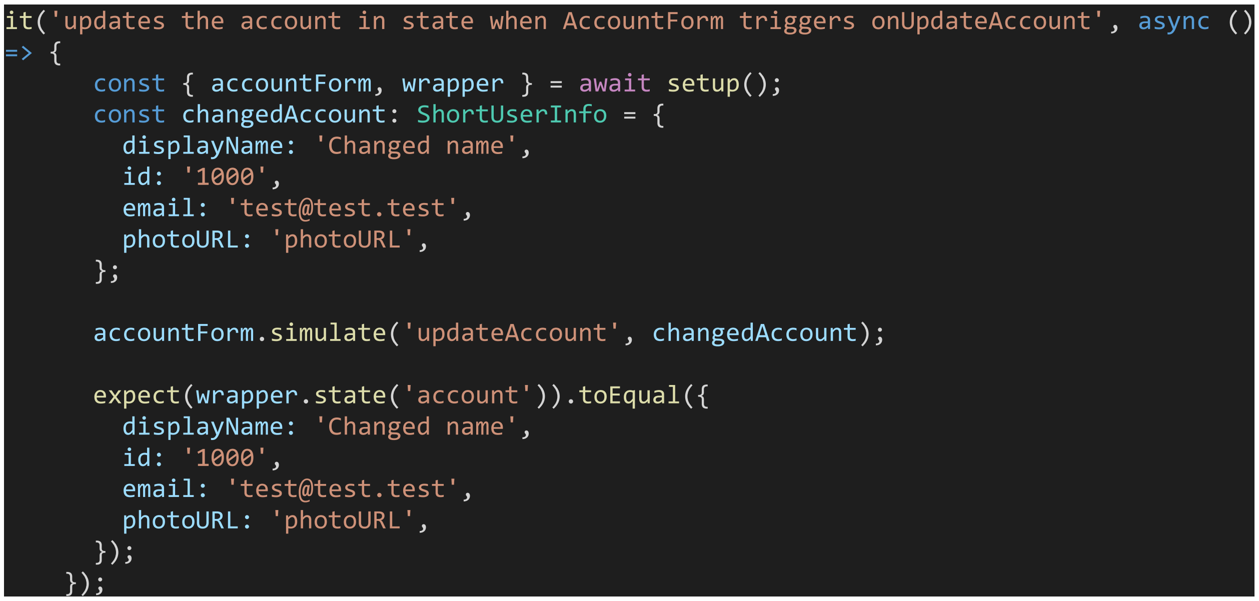
The rest of the tests cover the case when “isLoading” is false. They can be grouped into a single nested describe block to make the structure clearer. The first two tests in the new block (see Figure 74) verify that AccountContainer renders the AccountForm component and passes the “account” state variable as the initialValues prop to the AccountForm component.



```
describe('when "isLoading" state variable is false', () => {
    it('renders an AccountForm component', async () => {
      const { accountForm } = await setup();
      expect(accountForm.exists()).toBe(true);
    });

    it('passes the "account" state variable to AccountForm as a prop', async () => {
      const { wrapper, accountForm } = await setup();
      expect(accountForm.prop('initialValues')).toEqual(
        wrapper.state('account'),
      );
    });
  });
```

Figure 74. Testing AccountForm rendering and its initialValues prop

Step 4 is to test the event handlers. The AccountContainer contains two event handler methods: handleAccountChange() and handleSubmit(). Both event handlers are dispatched from the child AccountForm component. The fact that the handlers are dispatched by AccountForm was verified when testing AccountForm in Section 5.1. Now, the goal is to verify the correct behaviour of each event handler.

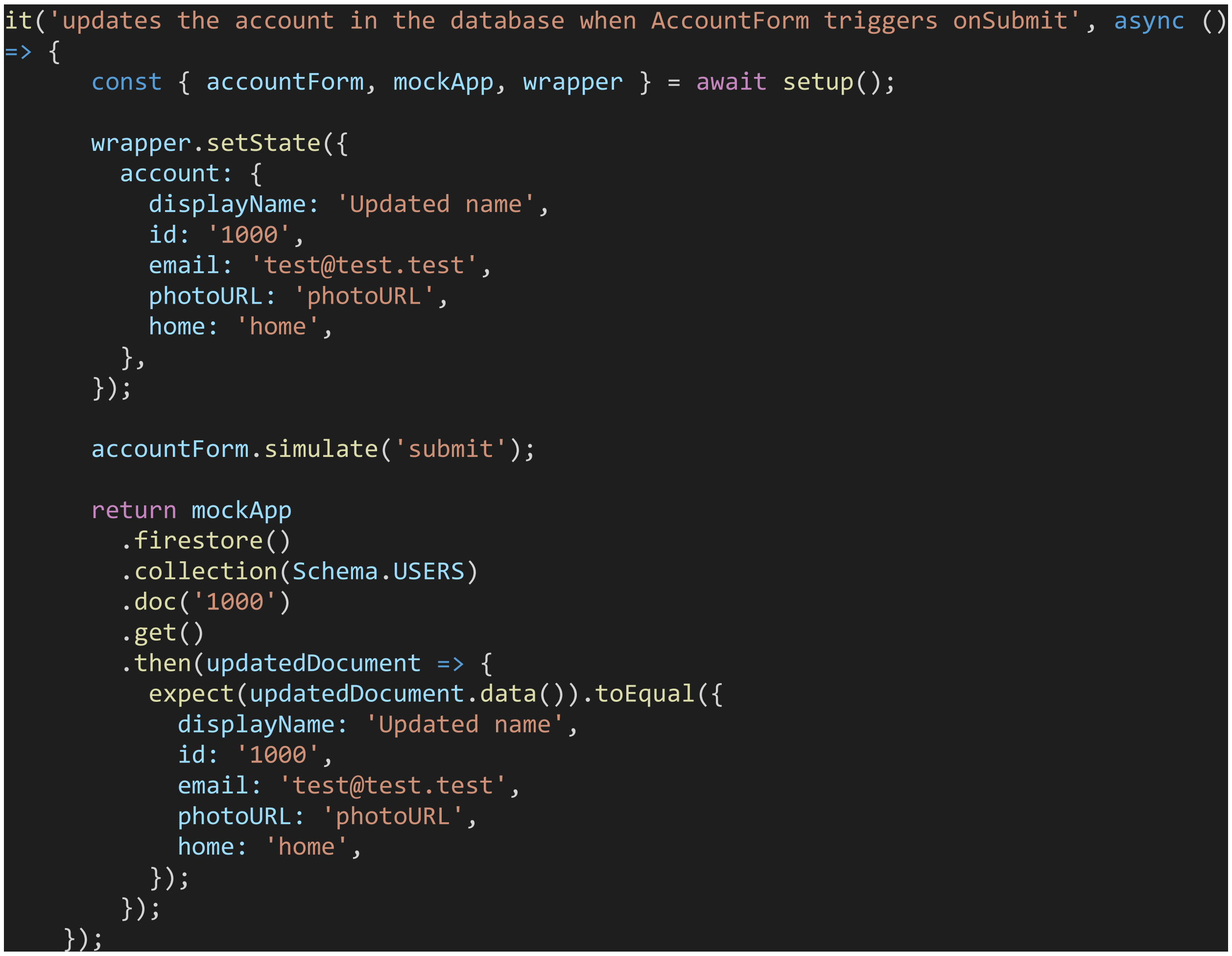


```
it('updates the account in state when AccountForm triggers onUpdateAccount', async ()
=> {
      const { accountForm, wrapper } = await setup();
      const changedAccount: ShortUserInfo = {
        displayName: 'Changed name',
        id: '1000',
        email: 'test@test.test',
        photoURL: 'photoURL',
      };

      accountForm.simulate('updateAccount', changedAccount);

      expect(wrapper.state('account')).toEqual({
        displayName: 'Changed name',
        id: '1000',
        email: 'test@test.test',
        photoURL: 'photoURL',
      });
    });
```

Figure 75. Testing state update when AccountForm triggers onUpdateAccount() event handler

Figure 75 presents the test for the handleAccountChange() handler. The test sets up the wrapper with default props. The AccountContainer wrapper and the AccountForm wrapper are returned by the setup function. The test then sets up a new user object changedAccount with a changed displayName property. Then, an ‘updateAccount’ event is simulated on the accountForm node wrapper. The changedAccount object is provided as the target for the simulated event. Finally, the test expects the “account” state variable of AccountContainer to equal changedAccount, meaning the handleAccountChange() handler method successfully updated the state with the modified account.

The handleSubmit() event handler is triggered on submit event of the AccountForm and updates the user information in the database. This behaviour can be verified using the mock Firebase application mockApp. Figure 76 presents the test for the handleSubmit() event handler.

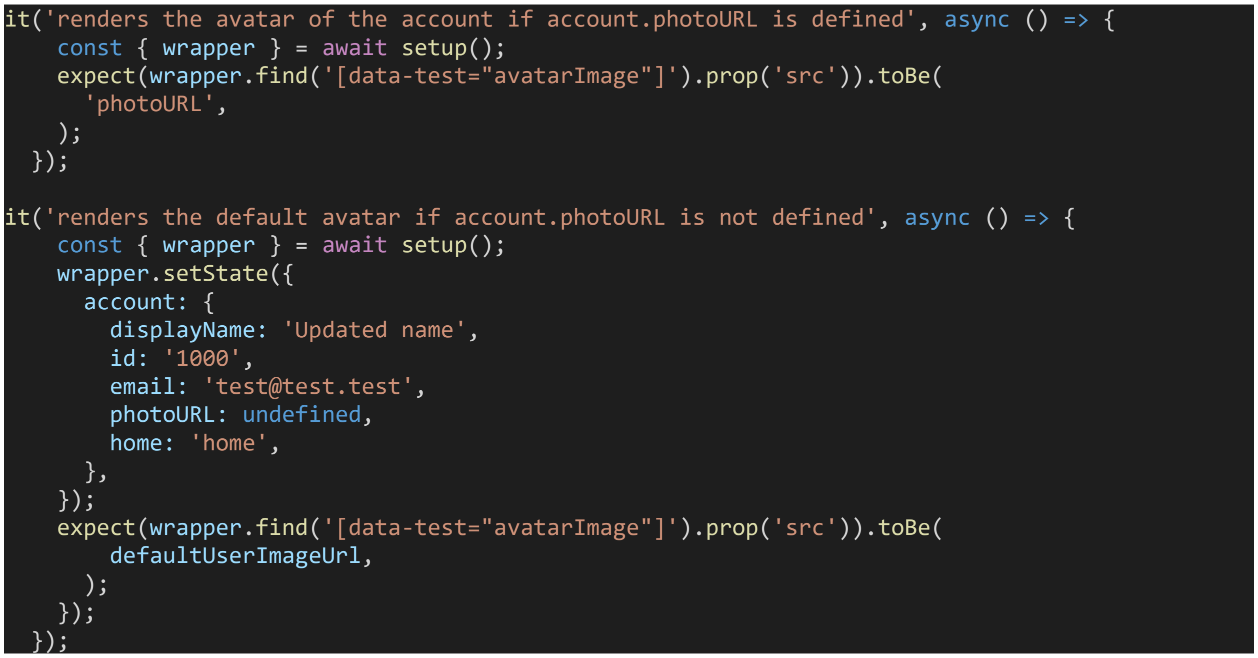


```
it('updates the account in the database when AccountForm triggers onSubmit', async ()
=> {
      const { accountForm, mockApp, wrapper } = await setup();

      wrapper.setState({
        account: {
          displayName: 'Updated name',
          id: '1000',
          email: 'test@test.test',
          photoURL: 'photoURL',
          home: 'home',
        },
      });

      accountForm.simulate('submit');

      return mockApp
        .firestore()
        .collection(Schema.USERS)
        .doc('1000')
        .get()
        .then(updatedDocument => {
          expect(updatedDocument.data()).toEqual({
            displayName: 'Updated name',
            id: '1000',
            email: 'test@test.test',
            photoURL: 'photoURL',
            home: 'home',
          });
        });
    });
```

Figure 76. Testing database update when AccountForm triggers handleSubmit() event handler

The test sets up the wrapper with default props. The setup function returns the AccountContainer wrapper, the AccountForm wrapper and mockApp. The “account” state variable of the AccountContainer is then updated with a different account object. The ‘submit’ event is then simulated on the AccountForm wrapper. Finally, the test fetches the user from the mock database and expects the user in the database to be updated. The call to the mock database is asynchronous, as it involves a promise. In order for Jest to wait for the promise to resolve, the promise has to be returned from the test. (Jest Documentation 2019c.) The test returns `mockApp.firestore().collection(Schema.USERS).doc('1000').get()` promise that makes a request to the mock database to get a user with a specified id. The assertion is added to the callback of the promise.

Step 5 is to test component lifecycle methods, but as mentioned earlier, in the case of AccountContainer this step was performed at the same time as testing props. Step 6 is to test conditions. Multiple conditions have already been covered while testing the state variables. However, looking back at Figure 68 reveals that one condition that affects the rendering of the component has not yet been tested. That condition is the image of the user avatar.



```
it('renders the avatar of the account if account.photoURL is defined', async () => {
    const { wrapper } = await setup();
    expect(wrapper.find('[data-test="avatarImage"]').prop('src')).toBe(
      'photoURL',
    );
  });

it('renders the default avatar if account.photoURL is not defined', async () => {
    const { wrapper } = await setup();
    wrapper.setState({
      account: {
        displayName: 'Updated name',
        id: '1000',
        email: 'test@test.test',
        photoURL: undefined,
        home: 'home',
      },
    });
    expect(wrapper.find('[data-test="avatarImage"]').prop('src')).toBe(
        defaultUserImageUrl,
      );
    });
  });
```

Figure 77. Testing user avatar image rendering

Figure 77 contains the two tests that verify avatar rendering. The first test verifies that if a user has a photoURL defined, the avatar displays an image with that

URL. The second test verifies that if photoURL is not defined, the avatar displays the default image.

```
it('matches snapshot', async () => {
    const { wrapper } = await setup();
    expect(toJson(wrapper)).toMatchSnapshot();
});
```

Figure 78. Snapshot testing AccountContainer

Finally, a snapshot test is made for the component. The component is rendered with default props and a snapshot is taken. Figure 78 contains the snapshot test. The test suite of the AccountContainer component is provided in full in Appendix 5.

5.6 Configuring pre-push test execution with Git hooks

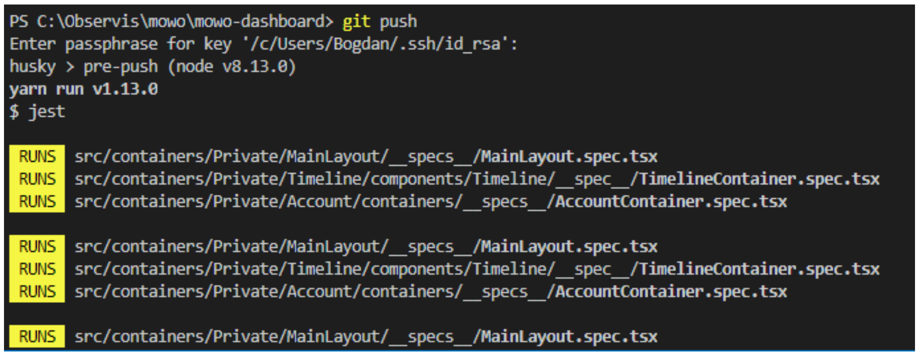
First, the Husky library needs to be installed with yarn (see Figure 79). A hook can now be added to the Git push event that will execute the tests before allowing to push the changes to version control. Figure 80 shows a pre-push Husky hook added in the package.json file of the MOWO dashboard application.

```
yarn add --dev husky
```

Figure 79. Installing Husky

```
"husky": {
    "hooks": {
      "pre-push": "yarn test"
    }
  }
```

Figure 80. Configuring a pre-push command with Husky in package.json



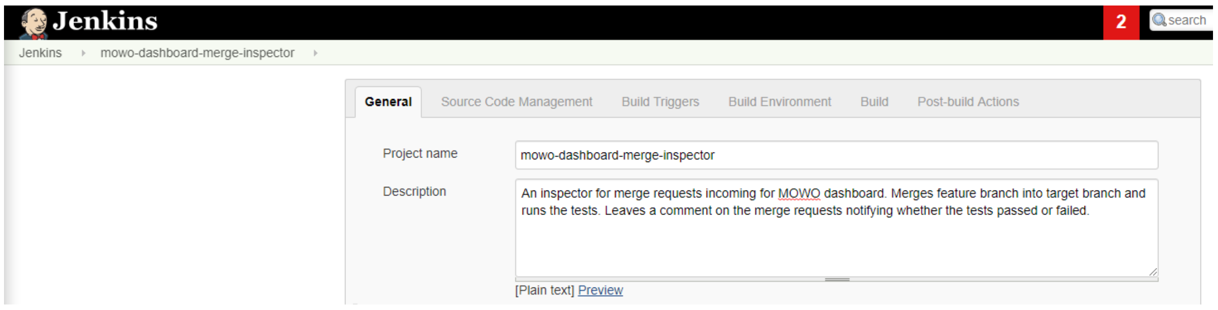
```
PS C:\Observis\mowo\mowo-dashboard> git push
Enter passphrase for key '/c/Users/Bogdan/.ssh/id_rsa':
husky > pre-push (node v8.13.0)
yarn run v1.13.0
$ jest

 RUNS  src/containers/Private/MainLayout/__specs__/MainLayout.spec.tsx
 RUNS  src/containers/Private/Timeline/components/Timeline/__spec__/TimelineContainer.spec.tsx
 RUNS  src/containers/Private/Account/containers/__specs__/AccountContainer.spec.tsx

 RUNS  src/containers/Private/MainLayout/__specs__/MainLayout.spec.tsx
 RUNS  src/containers/Private/Timeline/components/Timeline/__spec__/TimelineContainer.spec.tsx
 RUNS  src/containers/Private/Account/containers/__specs__/AccountContainer.spec.tsx

 RUNS  src/containers/Private/MainLayout/__specs__/MainLayout.spec.tsx
```

Figure 81. Husky executes tests upon push event

The pre-push hook will be executed every time a developer attempts to push the changes into version control. This behaviour is displayed in Figure 81. After a `git push` command is entered in the terminal, Husky automatically runs the `yarn test` command that executes every test for the application. If any of the tests fail, the push command is cancelled.

5.7 Creating a Jenkins project to run tests for each merge request

Jenkins provides a user interface that simplifies the creation of new projects. A new project can be created by going to the main page of the Jenkins server and choosing “New item” in the menu on the left. “Freestyle project” is selected from a number of project presets that the automation server provides. The name chosen for the project is “mowo-dashboard-merge-inspector”. Figure 82 shows the name and the description of the project configured on the Jenkins server.



Project name: mowo-dashboard-merge-inspector

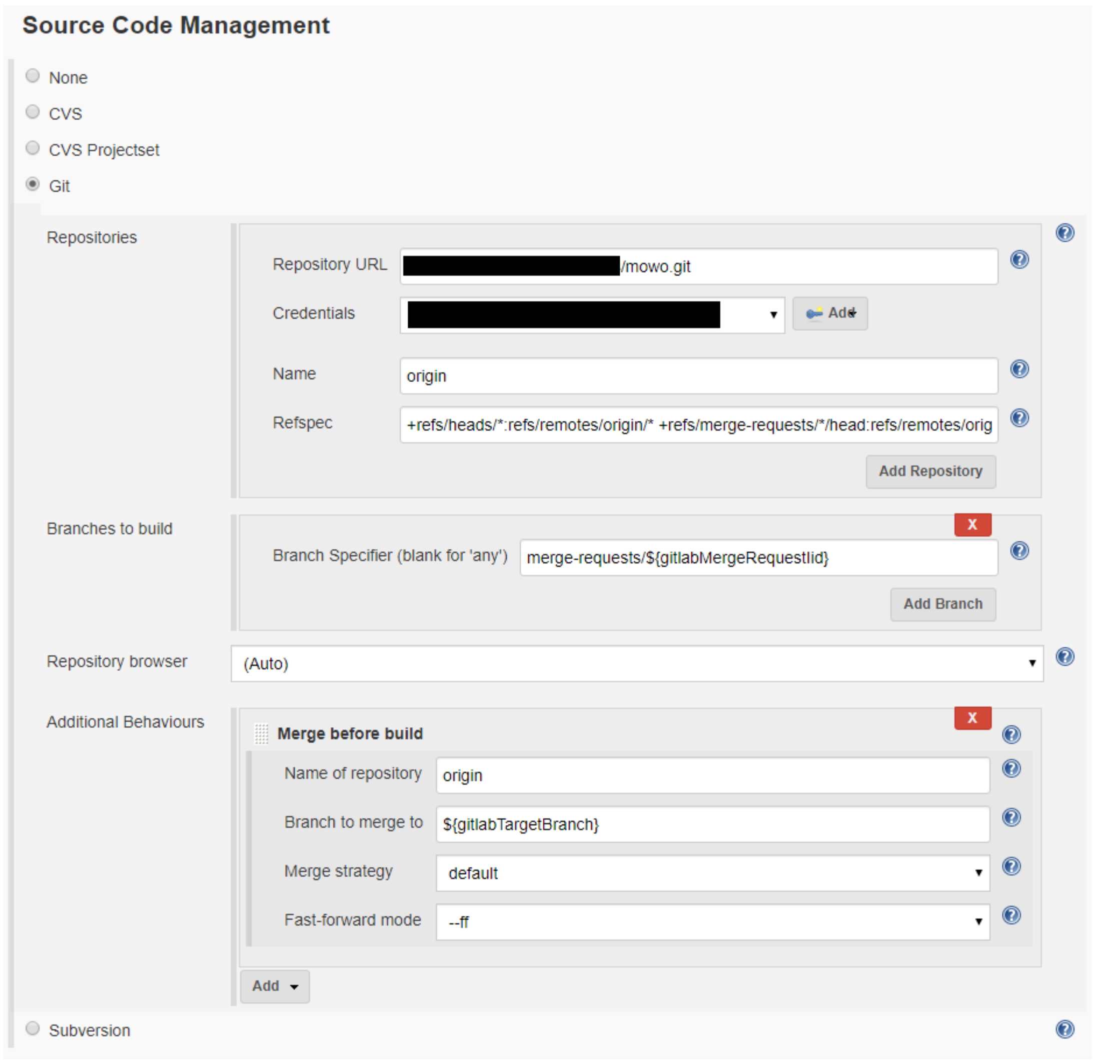
Description: An inspector for merge requests incoming for MOWO dashboard. Merges feature branch into target branch and runs the tests. Leaves a comment on the merge requests notifying whether the tests passed or failed.

Figure 82. Project name and description configured

Next, GitLab integration needs to be configured in the Source Code Management section of the project settings. Jenkins Git Plugin is required for the Git integration options to be available. Figure 83 shows the Source Code Management settings for the mowo-dashboard-merge-inspector project. “Git” should be chosen from the list of the available version control systems. Next, the repository needs to be provided for Jenkins to check out. The repository URL and the correct credentials for GitLab are required. In the advanced options for the repository, a name for the repository can be specified and a refspec provided. The refspec allows to map remote refs in the repository to the local refs on the Jenkins server. The name specified is “origin” and the refspec is

```
+refs/heads/*:refs/remotes/origin/* +refs/merge-
requests/*/head:refs/remotes/origin/merge-requests/* .
```

The next step is to specify which branch the merge inspector should build. The specified branch is the feature branch: merge-requests/${gitlabMergeRequestIid}. Lastly, an additional behaviour “Merge before build” is added. This option makes Jenkins merge the feature branch into the target branch before executing the tests. This way the merge inspector can verify that the tests will still pass after the merge request is accepted and the feature branch is merged into the target branch.


Source Code Management

None [✓ Git]
CVS
CVS Projectset
Git
Subversion

Repositories
- Repository URL: [redacted]/mowo.git
- Credentials: [redacted]
- Name: origin
- Refspec: +refs/heads/*:refs/remotes/origin/* +refs/merge-requests/*/head:refs/remotes/orig

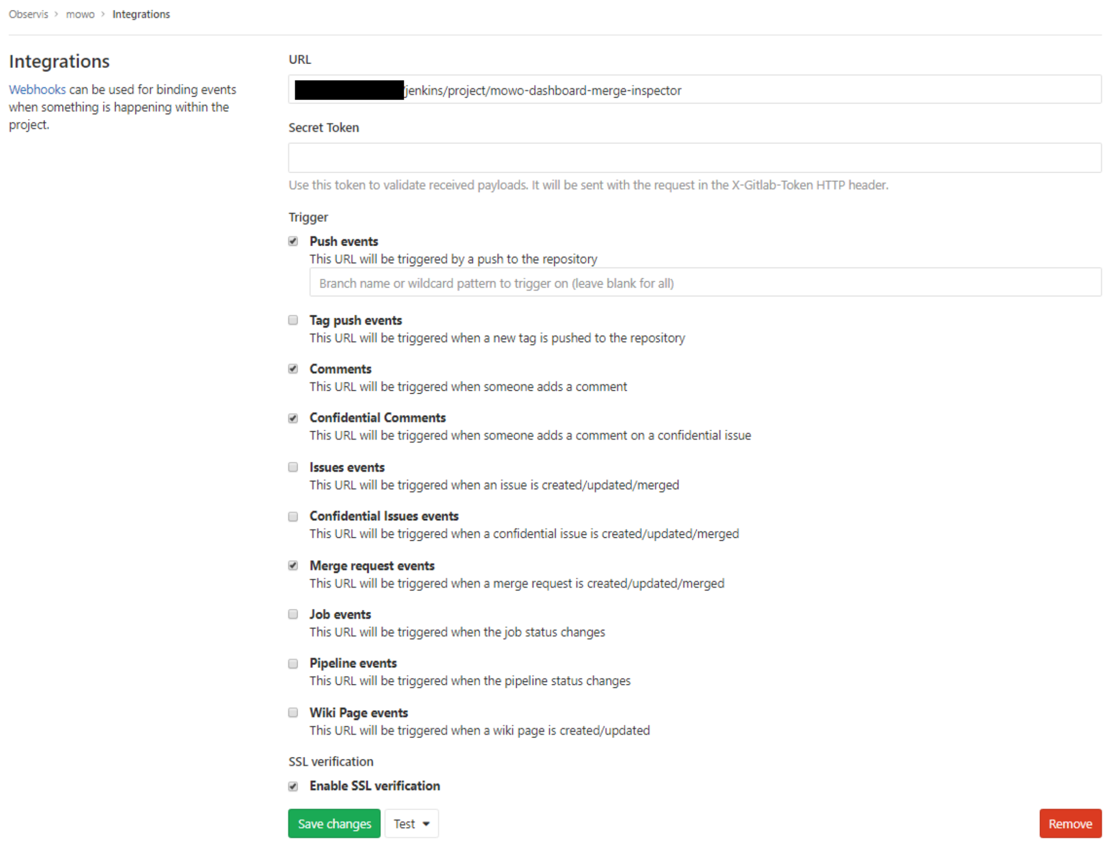
Branches to build
- Branch Specifier (blank for 'any'): merge-requests/${gitlabMergeRequestIid}

Repository browser: (Auto)

Additional Behaviours
- Merge before build
  - Name of repository: origin
  - Branch to merge to: ${gitlabTargetBranch}
  - Merge strategy: default
  - Fast-forward mode: --ff

Figure 83. Git integration configured

Next, Build Triggers are configured for the project. For this step to work, a webhook needs to be added to the repository on GitLab. The webhook can be configured by going to the repository on the GitLab server, and then selecting “Settings/Integrations/Add webhook” from the menu. Figure 84 shows the webhook settings for the MOWO repository. The Jenkins mowo-dashboard-merge-inspector project URL is provided and the types of events that trigger the webhook are specified. The repository is configured to trigger the webhook on every merge request event, comment and push event.



Integrations

Webhooks can be used for binding events when something is happening within the project.

URL

[redacted]jenkins/project/mowo-dashboard-merge-inspector

Secret Token

Use this token to validate received payloads. It will be sent with the request in the X-Gitlab-Token HTTP header.

Trigger

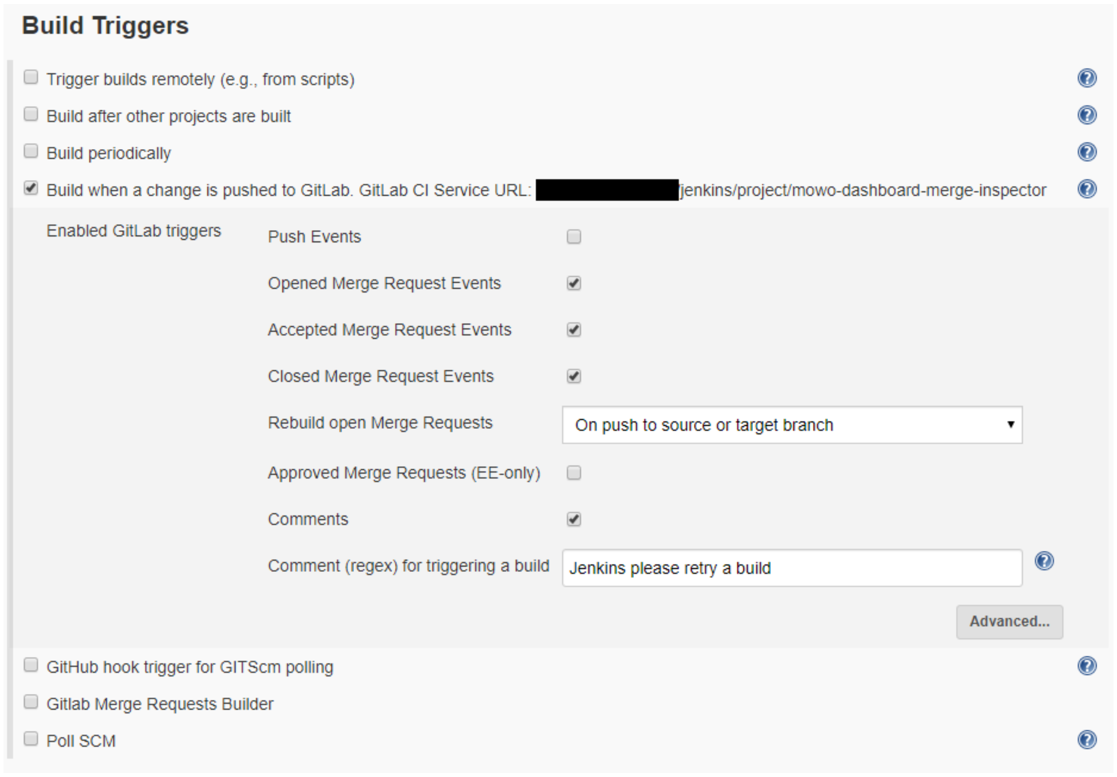
- ✓ **Push events**
  This URL will be triggered by a push to the repository
  Branch name or wildcard pattern to trigger on (leave blank for all)
- ☐ **Tag push events**
  This URL will be triggered when a new tag is pushed to the repository
- ✓ **Comments**
  This URL will be triggered when someone adds a comment
- ✓ **Confidential Comments**
  This URL will be triggered when someone adds a comment on a confidential issue
- ☐ **Issues events**
  This URL will be triggered when an issue is created/updated/merged
- ☐ **Confidential Issues events**
  This URL will be triggered when a confidential issue is created/updated/merged
- ✓ **Merge request events**
  This URL will be triggered when a merge request is created/updated/merged
- ☐ **Job events**
  This URL will be triggered when the job status changes
- ☐ **Pipeline events**
  This URL will be triggered when the pipeline status changes
- ☐ **Wiki Page events**
  This URL will be triggered when a wiki page is created/updated

SSL verification

- ✓ **Enable SSL verification**

Figure 84. Webhook configured on GitLab

Now, the build trigger is also configured on Jenkins as shown in Figure 85. The option “Build when a change is pushed to GitLab” is checked and types of events that trigger the build are specified as well. The events chosen are “Opened Merge Request Events”, “Accepted Merge Request Events”, “Closed Merge Request Events” and “Comments”. The option “Rebuild open Merge Requests On push to source or target branch” makes sure the build will run if either the source or the target branch is updated in the repository. A regex pattern is provided for the comment string that would trigger a build. The string is currently set to the default “Jenkins please retry a build”. Leaving a comment with this text on a merge request on GitLab will trigger the build, but regular comments will not trigger the build.



Build Triggers

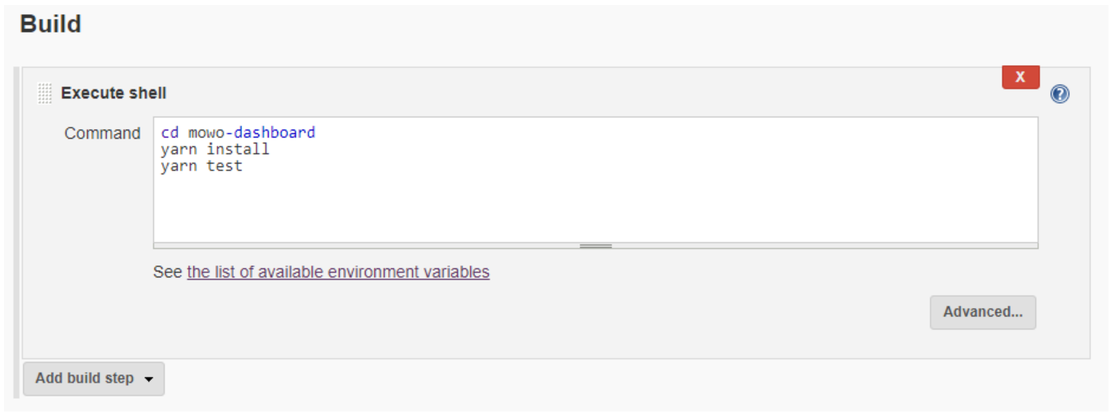
- [ ] Trigger builds remotely (e.g., from scripts)
- [ ] Build after other projects are built
- [ ] Build periodically
- [x] Build when a change is pushed to GitLab. GitLab CI Service URL: [redacted]jenkins/project/mowo-dashboard-merge-inspector

Enabled GitLab triggers

Push Events	☐
Opened Merge Request Events	✓
Accepted Merge Request Events	✓
Closed Merge Request Events	✓
Rebuild open Merge Requests	On push to source or target branch
Approved Merge Requests (EE-only)	☐
Comments	✓
Comment (regex) for triggering a build	Jenkins please retry a build

- [ ] GitHub hook trigger for GITScm polling
- [ ] Gitlab Merge Requests Builder
- [ ] Poll SCM

Figure 85. Build triggers configured



Build

Execute shell

Command

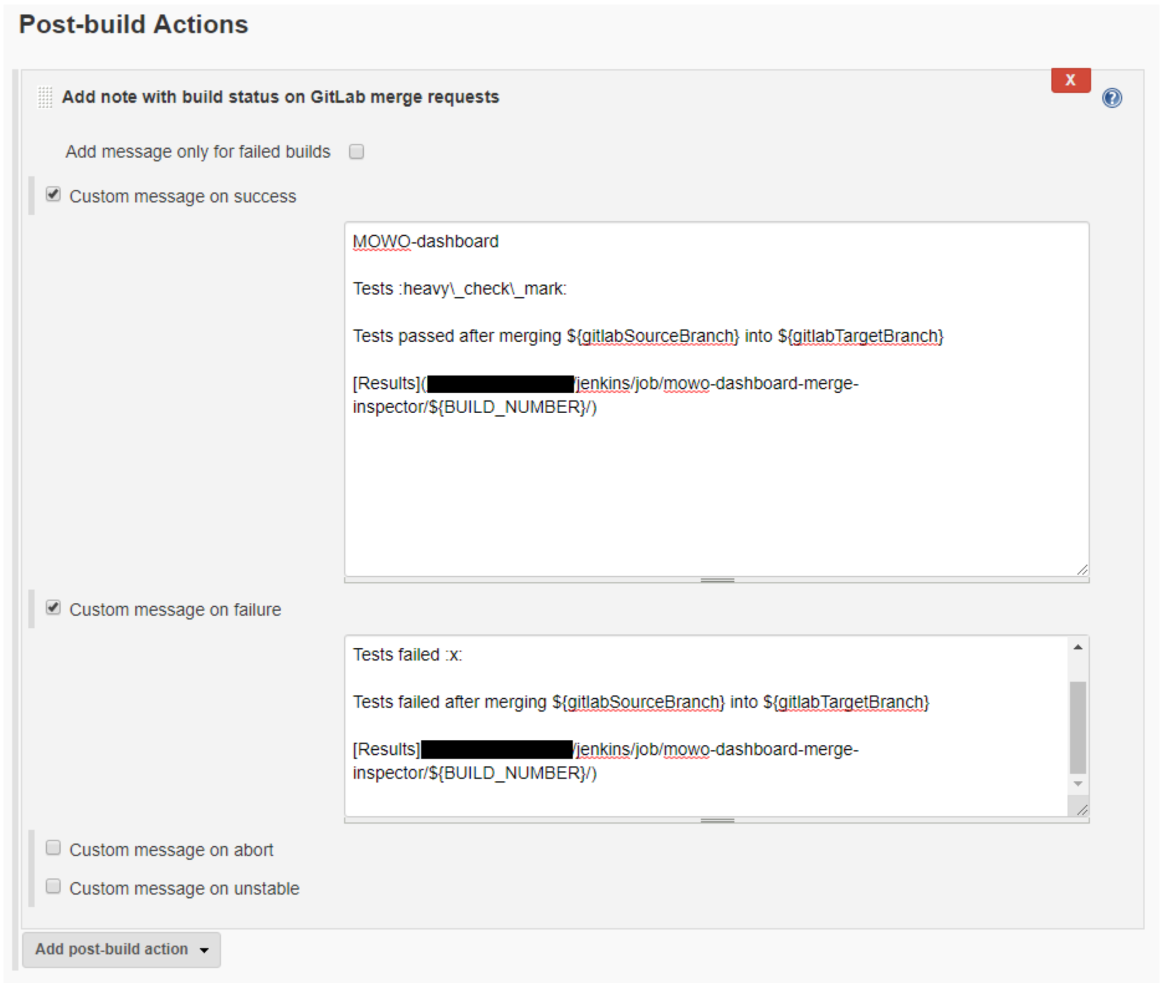
```
cd mowo-dashboard
yarn install
yarn test
```

See the list of available environment variables

Figure 86. Commands executed during the build

Then, the build itself is configured. The build step “Execute shell” is added as shown in Figure 86. The shell script switches from the root directory of the MOWO repository into the mowo-dashboard directory, executes `yarn install` to get the Node modules and finally executes tests by running `yarn test` – the script configured in the package.json of MOWO dashboard in Section 5.1.

Finally, custom messages can be configured for Jenkins to display on merge requests after the build is completed. This is done in the "Post-build Actions" section. Two custom messages are created: one for success and one for failure. The comments use GitLab's built-in emoji syntax to display a green check mark when tests are complete and a red cross when the tests fail. The comments also contain a link to the console output of the Jenkins build that ran the tests. This makes it easier to quickly see why the tests failed. Figure 87 presents the comment configuration.



Add message only for failed builds ☐

☑ Custom message on success

```
MOWO-dashboard

Tests :heavy\_check\_mark:

Tests passed after merging ${gitlabSourceBranch} into ${gitlabTargetBranch}

[Results]([redacted]jenkins/job/mowo-dashboard-merge-
inspector/${BUILD_NUMBER}/)
```

☑ Custom message on failure

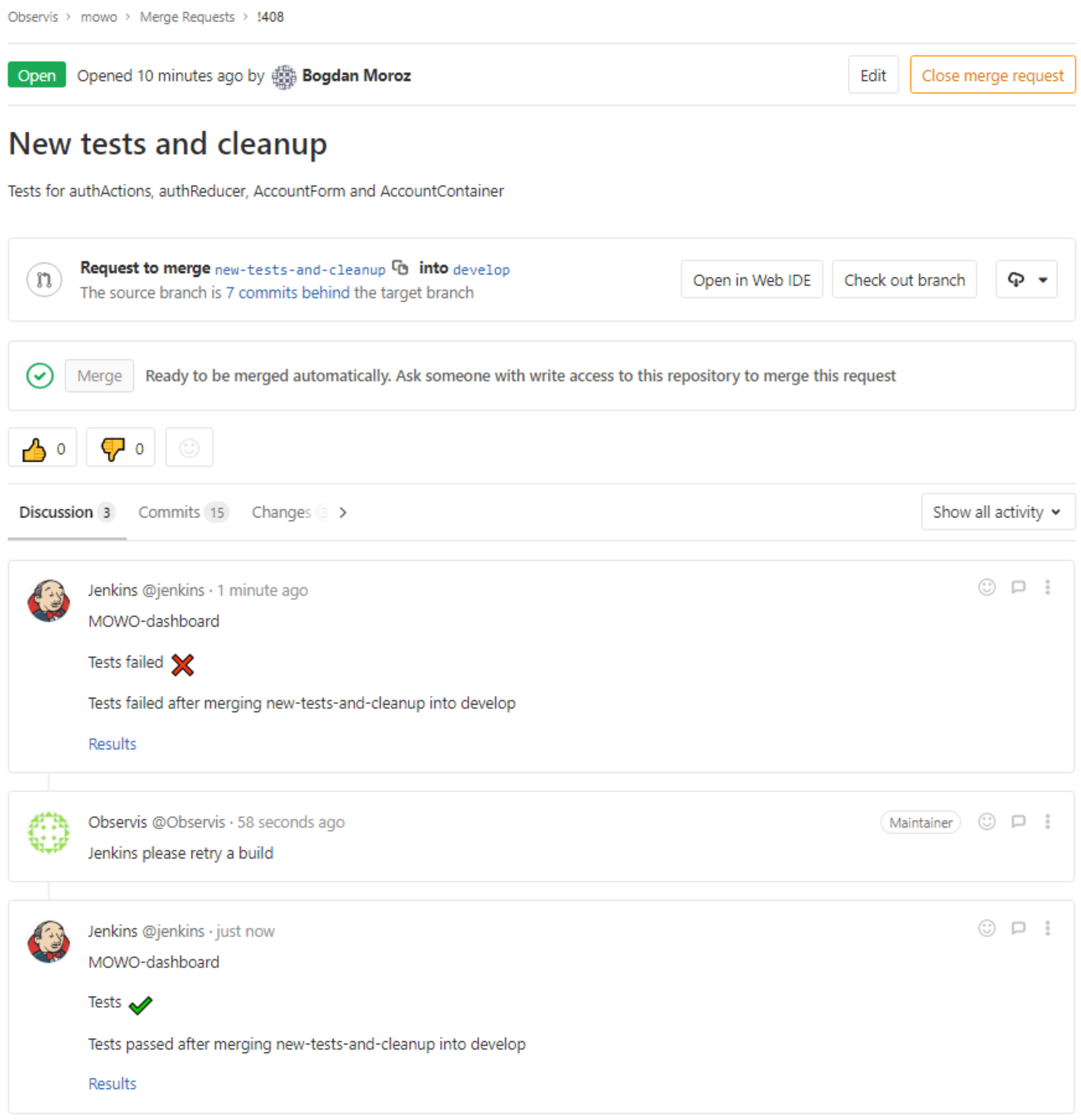
```
Tests failed :x:

Tests failed after merging ${gitlabSourceBranch} into ${gitlabTargetBranch}

[Results][redacted]/jenkins/job/mowo-dashboard-merge-
inspector/${BUILD_NUMBER}/)
```

☐ Custom message on abort

☐ Custom message on unstable

Figure 87. Configuring GitLab comments post-build

Figure 88 displays how Jenkins comments are displayed on a GitLab merge request. When the merge request was first open, the tests for the utility function timeUtils failed. The reason for that was that timeUtils is located outside the mowo-dashboard directory, in a directory called @shared. Node modules were not installed in that directory when the mowo-dashboard-merge-inspector project was executed. In order to fix the issue, the “Build” step of the merge inspector project was updated to run `yarn install` in the @shared directory, then switch to the mowo-dashboard directory and run the tests (the changes are shown in Figure 89).



Opened 10 minutes ago by Bogdan Moroz

New tests and cleanup

Tests for authActions, authReducer, AccountForm and AccountContainer

Request to merge new-tests-and-cleanup into develop
The source branch is 7 commits behind the target branch

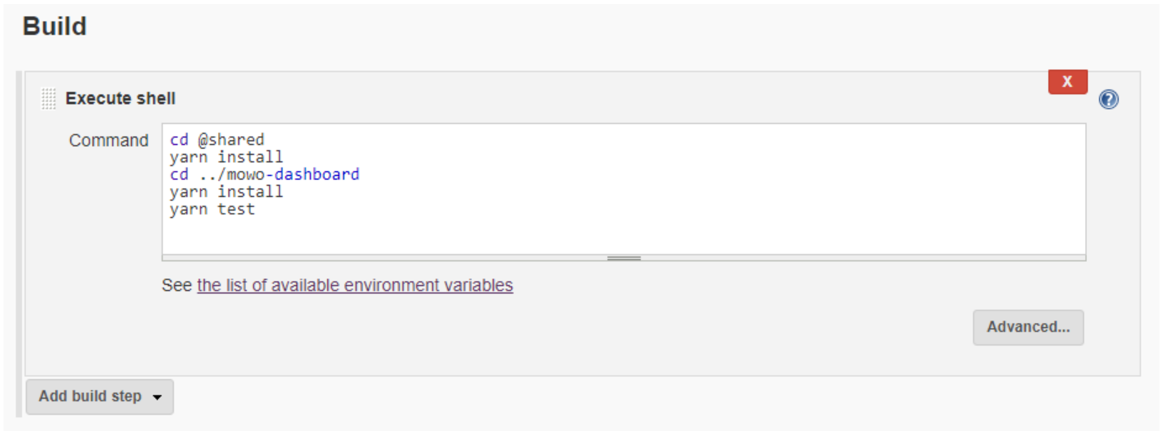
Ready to be merged automatically. Ask someone with write access to this repository to merge this request

Jenkins @jenkins · 1 minute ago
MOWO-dashboard
Tests failed ✗
Tests failed after merging new-tests-and-cleanup into develop
Results

Observis @Observis · 58 seconds ago
Jenkins please retry a build

Jenkins @jenkins · just now
MOWO-dashboard
Tests ✓
Tests passed after merging new-tests-and-cleanup into develop
Results

Figure 88. GitLab merge requests with Jenkins comments after two builds of the merge inspector



```
cd @shared
yarn install
cd ../mowo-dashboard
yarn install
yarn test
```

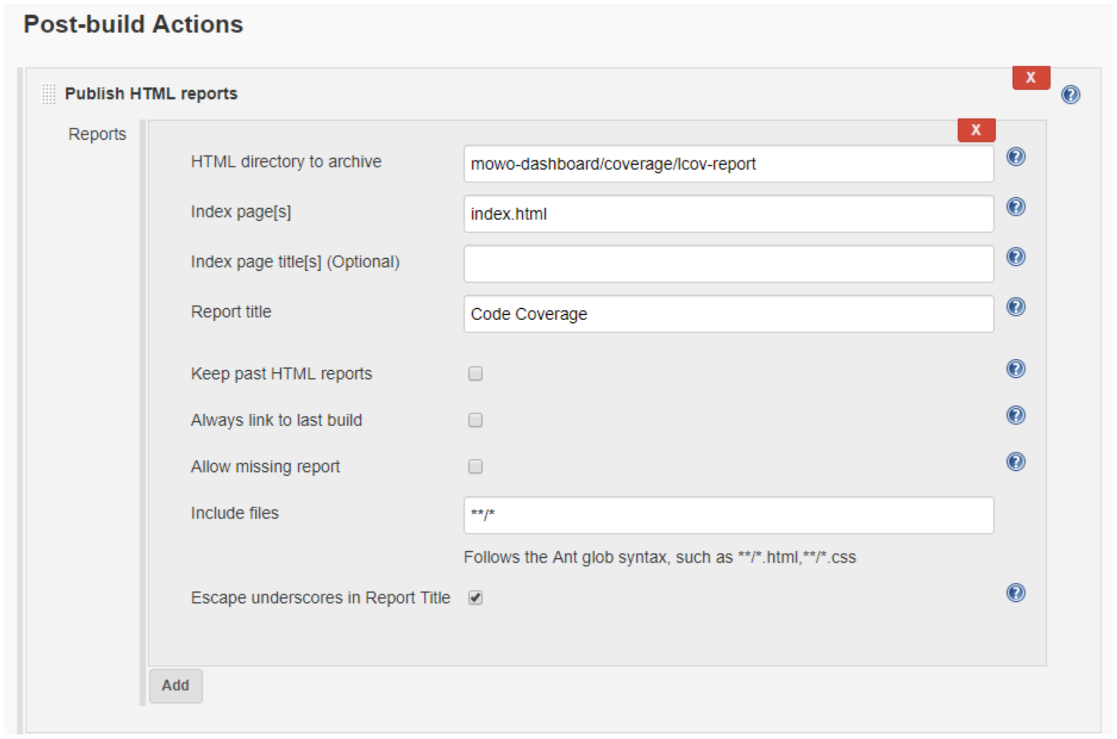
See the list of available environment variables

Figure 89. Updated build step of the merge inspector

After the change was made, a comment with the text “Jenkins please retry a build” was left on the merge request. That comment triggered another build of the merge inspector project. This time the tests succeeded, and the appropriate comment was made on the merge request.

5.8 Displaying a code coverage report on the Jenkins server

Finally, the HTML code coverage report generated by Jest can be displayed on the Jenkins server. Another post-build action needs to be added to the mowo-dashboard-merge-inspector project for that purpose. The configuration of the new post-build action is displayed in Figure 90. The action is added by clicking “Add” and choosing “Publish HTML reports” from the dropdown. The path to the directory where Jest generates the report and the name of the index page is provided. The title for the report is set to Code Coverage.



Post-build Actions

Publish HTML reports

Reports

HTML directory to archive: mowo-dashboard/coverage/lcov-report

Index page[s]: index.html

Index page title[s] (Optional):

Report title: Code Coverage

Keep past HTML reports ☐

Always link to last build ☐

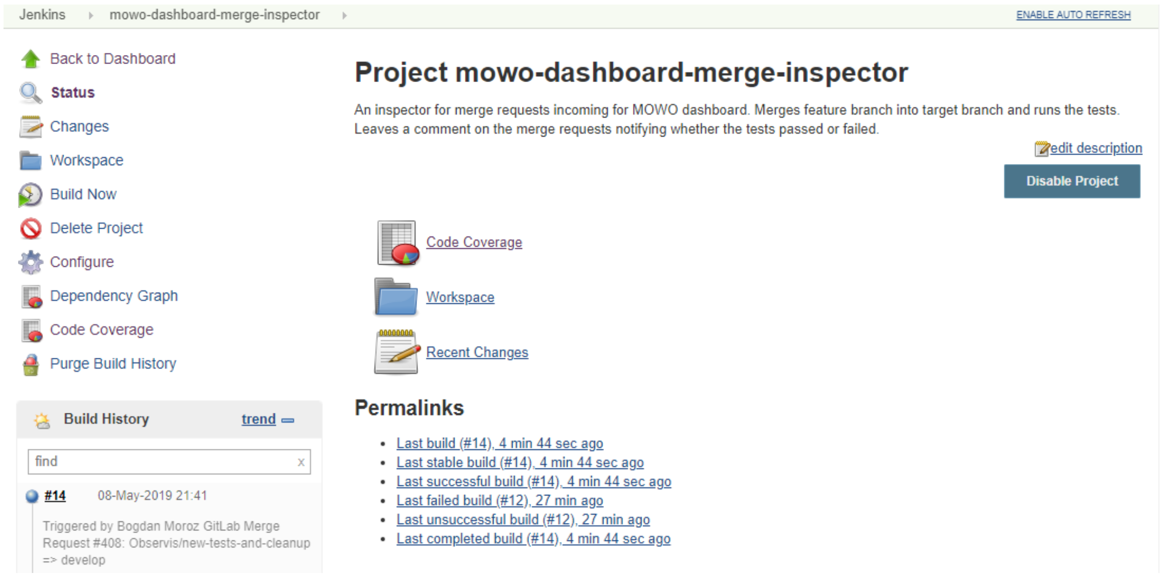
Allow missing report ☐

Include files: **/*

Follows the Ant glob syntax, such as **/*.html,**/*.css

Escape underscores in Report Title ✓

Figure 90. Publish HTML reports post-build action



Jenkins › mowo-dashboard-merge-inspector ›

#14 08-May-2019 21:41

Triggered by Bogdan Moroz GitLab Merge Request #408: Observis/new-tests-and-cleanup => develop

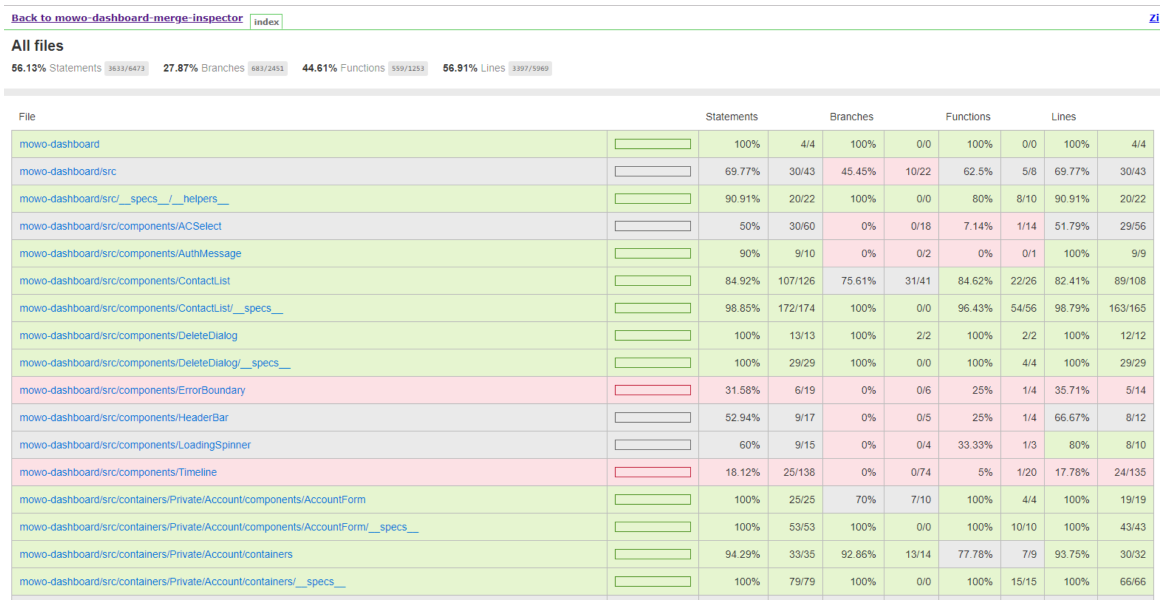
Project mowo-dashboard-merge-inspector

An inspector for merge requests incoming for MOWO dashboard. Merges feature branch into target branch and runs the tests. Leaves a comment on the merge requests notifying whether the tests passed or failed.

Permalinks

- Last build (#14), 4 min 44 sec ago
- Last stable build (#14), 4 min 44 sec ago
- Last successful build (#14), 4 min 44 sec ago
- Last failed build (#12), 27 min ago
- Last unsuccessful build (#12), 27 min ago
- Last completed build (#14), 4 min 44 sec ago

Figure 91. Merge inspector project page with a link to Code Coverage



Back to mowo-dashboard-merge-inspector index Zi

All files

56.13% Statements 3633/6473 27.87% Branches 683/2451 44.61% Functions 559/1253 56.91% Lines 3397/5969

File		Statements		Branches		Functions		Lines	
mowo-dashboard		100%	4/4	100%	0/0	100%	0/0	100%	4/4
mowo-dashboard/src		69.77%	30/43	45.45%	10/22	62.5%	5/8	69.77%	30/43
mowo-dashboard/src/__specs__/__helpers__		90.91%	20/22	100%	0/0	80%	8/10	90.91%	20/22
mowo-dashboard/src/components/ACSelect		50%	30/60	0%	0/18	7.14%	1/14	51.79%	29/56
mowo-dashboard/src/components/AuthMessage		90%	9/10	0%	0/2	0%	0/1	100%	9/9
mowo-dashboard/src/components/ContactList		84.92%	107/126	75.61%	31/41	84.62%	22/26	82.41%	89/108
mowo-dashboard/src/components/ContactList/__specs__		98.85%	172/174	100%	0/0	96.43%	54/56	98.79%	163/165
mowo-dashboard/src/components/DeleteDialog		100%	13/13	100%	2/2	100%	2/2	100%	12/12
mowo-dashboard/src/components/DeleteDialog/__specs__		100%	29/29	100%	0/0	100%	4/4	100%	29/29
mowo-dashboard/src/components/ErrorBoundary		31.58%	6/19	0%	0/6	25%	1/4	35.71%	5/14
mowo-dashboard/src/components/HeaderBar		52.94%	9/17	0%	0/5	25%	1/4	66.67%	8/12
mowo-dashboard/src/components/LoadingSpinner		60%	9/15	0%	0/4	33.33%	1/3	80%	8/10
mowo-dashboard/src/components/Timeline		18.12%	25/138	0%	0/74	5%	1/20	17.78%	24/135
mowo-dashboard/src/containers/Private/Account/components/AccountForm		100%	25/25	70%	7/10	100%	4/4	100%	19/19
mowo-dashboard/src/containers/Private/Account/components/AccountForm/__specs__		100%	53/53	100%	0/0	100%	10/10	100%	43/43
mowo-dashboard/src/containers/Private/Account/containers		94.29%	33/35	92.86%	13/14	77.78%	7/9	93.75%	30/32
mowo-dashboard/src/containers/Private/Account/containers/__specs__		100%	79/79	100%	0/0	100%	15/15	100%	66/66

Figure 92. HTML coverage report generated by Jest displayed on Jenkins

After the build is executed one more time, a link to the code coverage report appears on the page of the mowo-dashboard-merge-inspector project. The page is shown in Figure 91. The HTML report generated by Jest is available by clicking the link. A sample of the contents of the HTML report is presented in Figure 92. The page always displays the code coverage for the latest build.

6 DISCUSSION

What follows is the summary of the work conducted for this thesis.

- The literature on the motivation for and the benefits of test automation was reviewed.
- The literature on the established industry patterns and techniques for writing automated tests was reviewed.
- The technology stack of the case study application was reviewed alongside the tools for testing the applications with this technology stack.
- A testing strategy for the case study application was formulated based on the findings of the literature review as well as an overview of four testing strategies used by software developer teams in modern companies. The strategy prioritized the view of the tests as an executable specification of the system.
- The strategy documented the way the test files should be named and organized and the way the tests should be structured. Four types of units that need to be tested were identified.
- The strategy set limitations on what should be tested and provided the motivation for excluding certain aspects of the application from tests.

- The strategy described how the automatic execution of the tests should be integrated into the deployment pipeline of the application.
- The strategy mentioned how the incidents with the software should be managed and what metrics should be collected to evaluate the efficiency of the strategy.
- The testing environment for the application was set up.
- Detailed examples were provided for testing each of the unit types: namely, utility functions, action creators, reducers and React components. Two examples were provided for testing React components: a stateless component and a stateful component connected to Redux and Firebase.
- A Git hook was configured using Husky to verify the local execution of the automated tests before the changes are pushed into version control.
- A Jenkins project was created to run the automated tests for every merge request opened on GitLab and leave a comment on GitLab notifying whether the tests passed.
- A code coverage report for each merge request was configured to be displayed on the Jenkins server.

The benefits that test automation brings to software development companies were determined. A testing strategy was proposed for applications with the technology stack of the case study. The setup process of a testing environment was documented. As established in Section 2.4, quantifying the effectiveness of the proposed testing strategy cannot be achieved by viewing the code coverage numbers alone. Instead, the development process of the application has to be monitored over a period of time to see whether the number of discovered bugs is decreasing. As the testing strategy was only fully revealed to the development team at the end of this study, the data on the long-term effectiveness of the strategy could not be collected. The strategy requires to be evaluated over time in order to notice a trend in the number of bugs, or lack thereof.

However, the strategy was formulated based on established industry patterns and techniques which have already been proven useful over time. Attention was paid to the expressiveness of the tests, and the BDD technique with spec-style was recommended to make sure tests help understand the SUT, provide an executable specification of the software. Moreover, the use of parametrized setup functions, utility methods and the arrange-act-assert pattern with a single assertion per test as a general guideline was recommended to make sure tests are easy to read, write and maintain as the codebase evolves around them. All in all, the foundation for the strategy to be effective is solid.

Of the four units recommended for testing, React components provided the biggest challenge for determining the testing approach, largely due to the high degree of flexibility in the structure of each component. Observis does not have documented conventions for structuring React components. Therefore, there is a limit to the precision with which the testing process suitable for every component can be described. The proposed strategy aims at targeting every aspect of a component from basic rendering to props, state, event handlers, lifecycle methods and conditional rendering. As shown in the examples of component tests in Sections 5.1 and 5.2, some of these steps can be combined or skipped based on the structure of the component under test.

However, having an established strategy for component tests will influence the development of the components in the future. The strategy argues for the BDD technique where the development of the software is driven by the creation of tests. The test suite examples provided in the practical part of this thesis were written after the SUT. Developing new components test-first and according to the strategy will influence the component structure and provide a higher level of consistency to the codebase. That said, the testing strategy proposed in this study can be augmented by establishing stronger coding conventions for the application.

The strategy proposed in Chapter 4 satisfied the commissioner company and will be adopted for React-Redux applications developed at Observis. The strategy will be extracted into a separate document and shared with the software developers of the company. Moreover, a coaching session will be conducted with the developers to better familiarize them with the strategy.

7 CONCLUSION

The original problem that motivated this thesis was the complete lack of automated unit tests in the development process of React-Redux applications at Observis. The goal of the work was to establish the benefits of automated testing for the commissioner company, determine the tools and the strategy that can be adopted for testing and integrate the automatic execution of the tests into the deployment pipeline of the case study application. At the end of the study, a testing strategy was formulated based on the overview of the established industry patterns and techniques and an examination of the technology stack of the case study. Four types of units that must be tested were identified and detailed examples of tests for each unit type were provided. The automated tests were integrated into the deployment pipeline of the application by configuring a Jenkins project to execute the tests for each merge request.

The testing strategy proposed by the study was based on literature on test automation, unit testing patterns and the behaviour-driven development technique. The writings of Humble (2017a; 2017b; 2017c; 2017d), Fowler (2013a; 2013b; 2014), Kim et al. (2016), Vocke (2018) and several other referenced authors were used to provide an overview of the fundamental concepts of automated testing, explain the benefits of automated testing for a technology organization, as well as reflect on the role of the tests in a deployment pipeline. “xUnit Test Patterns” by Gerard Meszaros (2007) described a number of patterns for test organization and structure as well as a summary of the goals that test automation should accomplish. The goals and patterns outlined by Meszaros were supported by the publications by Fowler (2004b; 2005), Koskela (2013) and Kim et al. (2016).

Together with the aforementioned sources, the work of Koskela (2013), North (2006; 2012), Marston & Dees (2017) and Justice (2018) was used to evaluate the benefits of the behaviour-driven development technique and spec-style for making tests expressive and providing an “executable specification” for the software. Moreover, the proposed strategy was based on a summary of four testing strategies used by development teams in modern companies. The strategies were documented by Pysarenko (2019), Bak (2019), Fishwick (2018) and Molinari (2018). The practical part of the study was largely based on the documentation of the testing tools and the technologies of the case study application.

A single detailed publication that would document a complete testing strategy for a React-Redux application was not found during the research, which adds value to the strategy documented in this study. This thesis combines an overview of the established industry concepts, patterns and techniques with the understanding of the technology stack of the case study application in order to produce a precise testing strategy that can be adopted by software companies as is. That said, the efficiency of the testing strategy is yet to be fully evaluated by collecting the data on the number of bugs discovered with the software over time after the strategy is adopted by the development team.

Next, the original research questions are addressed in turn.

How does test automation benefit a company?

Automated testing significantly speeds up the feedback that developers receive on the software that they are building. Faster feedback allows to notice regressions early and fix them quickly. Automated testing is faster than manual testing and allows for more frequent releases. Automated tests increase the productivity of software developer teams and allow gaining and sustaining a high speed of development. Such tests are less error-prone and more repeatable than the manual tests. Moreover, automated tests can provide a continuously updated specification for the software. Without automated testing, the financial and time investments in development and testing of applications keep increasing as applications grow.

What unit testing strategy can be adopted for React-Redux applications?

Establishing a specific testing strategy for React and Redux applications requires a solid understanding of both technologies, as well as an understanding of the tools available to test them. Moreover, established industry techniques and patterns need to be reviewed in order to justify the choices made in the strategy proposal. A strategy that can be adopted for React-Redux applications was presented in Chapter 4. The strategy is based on the goals of test automation as formulated by Meszaros (2007) and the behaviour-driven development technique. Automated tests created according to the strategy help understand the SUT by being expressive and acting as specifications of unit behaviour. Tests created according to the strategy are easy to read, write and execute and require minimal maintenance as the software grows and evolves around them. The readability and focus of the tests are achieved by using full sentences as test names, reducing duplication of setup code by relying on centralized utility methods and parametrized setup functions and using expressive matchers provided by Jest.

When testing React-Redux applications, four types of units require to be tested. These types are React components, Redux action creators, Redux reducers and various utility functions used by the application under test. A React component can be tested in 7 steps. Some of these steps can be omitted based on the structure of the component. First, basic component rendering must be verified. Second, each prop of the component needs to be tested by shallow rendering the component with a specific value for the prop and making an assertion about the shallow wrapper that verifies that the component received the prop. Third, for stateful components each state variable needs to be tested the same way as props. The next step is to test the event handlers. For each event handler, an event needs to be simulated and the appropriate handler expected to be called. In cases when the event handler is defined within the component under test and not passed down to the component as a prop, the behaviour of the handler needs to be verified as well. For instance, for a handler that updates the state of the component, a test needs to be written that verifies that the state was updated after the event handler was called. Step five is to specify the behaviour of the lifecycle methods of the component, such as data subscriptions and removal of those subscriptions. Step six is to test various conditions that affect the rendering of the component under test and to make a separate test for each condition. Finally, a snapshot needs to be created for the component in order to detect unexpected changes to its output. A single snapshot is recommended for each component, as various rendering results are already verified when testing conditions in step 6.

Redux action creators and reducers need to be tested according to the official Redux documentation. For each action creator, the action creator needs to be called with parameters and a correct action object expected to be returned. For each reducer, a test needs to be written that specifies that the reducer does not update the initial state when it receives an action of an unsupported type. Next, for each action type supported by the reducer, the reducer needs to be called with the initial state of the application and an action of that type and expected to update the state correctly. Lastly, in order to test a utility function, a full suite of function behaviours needs to be planned ahead. Then, the function should be expected to throw exceptions when any of its parameters are undefined or incorrect and to return correct values when all parameters are correct.

Creating tests that specify the behaviour of third-party libraries is not recommended, as the providers of those libraries are expected to test their code. Instead, third-party libraries should be replaced with test doubles within tests using the mocking functionality of Jest. Moreover, tests for the internal consistency of data types are not recommended if the React-Redux application under test uses tools to verify type consistency. The case study application uses TypeScript for that purpose, and other technologies such as PropTypes are available.

The metrics that need to be collected according to the strategy are the code coverage report generated by Jest and the number of bugs discovered with the software over time. The code coverage report illuminates the areas of the codebase that are not yet covered by tests, while the number of bugs serves as the metric for the efficiency of the strategy. For each bug discovered with the software, a new issue needs to be opened in the project management software used by the development team, and a unit test or several tests addressing the bug need to be created.

How should a testing environment be set up for the specific technology stack of the case study (React, TypeScript, Redux, Jenkins)?

The set up of the testing environment for the aforementioned technology stack was documented in the practical implementation part of the study, specifically in Sections 5.1, 5.6, 5.7 and 5.8. The libraries necessary to install are Jest, Enzyme, an adapter for Enzyme to work with a specific version of React and the ts-jest Jest preset that lets Jest work with TypeScript. Several libraries need to be installed to extend the mocking capabilities provided by Jest out of the box. The libraries are identity-obj-proxy for mocking imported files, mockdate to set a custom date within tests and Ts-Mock-Firebase for a precise and well-typed Firebase mock. Lastly, type definitions for the libraries need to be installed.

Jest requires to be set up from its configuration file. Directories for Jest to search the test files in need to be specified, ts-jest preset provided and code coverage collection enabled with text and HTML coverage reporters. A setup file needs to be created where Enzyme is configured to work with the specific version of React for the project using the Enzyme adapter. Jest configuration should point to the setup file in order to execute the setup before each test. Next, module paths need to be configured for Jest to correctly resolve dependencies, and specific file extensions need to be set for identity-obj-proxy to mock the imports with those extensions. If the application uses Firebase as a cloud database, a manual mock for Firebase needs to be configured using the Ts-Mock-Firebase library. The library closely emulates Firebase. The manual mock can be configured as shown in Section 5.1. The configuration is performed according to the official Jest guide for creating manual mocks for Node modules, and the setup specific to Ts-Mock-Firebase is provided in the documentation of the library.

In order to allow spotting and fixing errors before they are checked into version control, a Git hook needs to be configured for every push event. The Husky library can be used for the setup of the hook. Once configured, the hook will execute tests every time a developer attempts to push changes into version control. To integrate the automated execution of tests into the deployment pipeline for the application, a Jenkins job should be configured that verifies that a feature branch can be safely merged into the target branch.

How to evaluate the efficiency of a testing strategy?

The proposed strategy suggests two metrics to be collected for the case study application: a code coverage report and the number of bugs discovered with the application. The study found that code coverage reports cannot verify the efficiency of the strategy, but rather only point out the areas of the codebase that are not yet covered by tests. The only quantifiable metric for strategy efficiency found by the study is the number of bugs discovered with the software either in the production environment or during exploratory testing in the staging environment. In order to evaluate the efficiency of a testing strategy, it needs to be observed over time to see if there is a decreasing trend in the number of bugs.

Further research can be conducted to enhance the test automation efforts at Observis and improve the proposed strategy. The list of topics is as follows:

- Other levels of automated tests can be considered. This study focused on unit tests and did not cover integration and end-to-end tests. React-testing-library is an increasingly popular solution for writing integration tests for React applications, while tools like Cypress and Robot-Framework can be used for end-to-end testing. The analysis of strengths and disadvantages of each of these tools needs to be conducted and a testing strategy proposed.
- Coding conventions for developing React components should be documented. Having a consistent structure and a set of guidelines for component development would allow standardizing the testing process of the components more strictly.
- In addition to the case study application MOWO dashboard, Observis is developing a MOWO mobile application using React Native. A testing strategy for the React Native application needs to be researched and established. It can be evaluated how well the testing strategy proposed in this study is applicable to React Native applications, and what modifications are required for the strategy to support mobile development.

- The proposed testing strategy focused on component behaviour. However, style of the components was not tested. Given that style is an important aspect of a user interface, proper techniques can be established for testing the style of React components. Libraries like Jest-Styled-Components and tools for visual regression testing should be considered.
- Even though the proposed strategy aims to make the tests as simple, readable and expressive as possible, in certain edge cases debugging the tests can be beneficial. The official Jest documentation provides instructions for setting up the environment for debugging and the debugging process for tests. These instructions should be reviewed and debugging set up.

All in all, Jest and Enzyme provide a powerful and flexible set of tools for testing React-Redux applications. Jest and Enzyme allow creating tests that satisfy the goals of test automation. Jest makes the tests easy and fast to execute, with the order of test execution prioritized to maximize the speed. The vast assertion library of Jest allows for readable and expressive tests, while its mocking functionality increases the maintainability of tests by enabling developers to create deterministic test doubles, spy on the way units under test interact with their collaborators and test every unit in isolation. The spec-style used by Jest facilitates thinking of the behaviour of the SUT when creating tests. Enzyme provides a flexible set of features to traverse the output of React components, render components with specific props and simulate events. The shallow rendering functionality of Enzyme allows testing React components in isolation by rendering each component one level deep, without rendering child components recursively. The isolation provided by shallow rendering makes the tests for React components easier to maintain as the codebase grows and evolves, as changes to child components do not affect the tests for their parent components. Together with the snapshot testing feature of Jest, Enzyme helps detect unexpected changes in React elements.

The proposed testing strategy will be applied to all React-Redux applications developed at Observis and can be used by other companies. With frameworks like Electron that allow using web technologies to develop desktop applications, this testing strategy can be adopted for desktop application development as well.

REFERENCES

Abramov, D. 2015. You’re Missing the Point of React. WWW document. Updated 22 January 2015. Available at: https://medium.com/@dan_abramov/youre-missing-the-point-of-react-a20e34a51e1a [Accessed 4 March 2019].

Abramov et al. 2018. Redux Documentation. WWW document. Available at: https://redux.js.org/ [Accessed 19 May 2019].

Airbnb. 2019. Enzyme Documentation. WWW document. Updated 17 February 2019. Available at: https://airbnb.io/enzyme/ [Accessed 24 April 2019].

Atlassian Bitbucket. Tutorials. Gitflow Workflow. WWW document. Available at: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow [Accessed 24 April 2019].

Bak, T. 2019. Testing React Components Best Practices. WWW document. Updated 3 February 2019. Available at: https://medium.com/selleo/testing-react-components-best-practices-2f77ac302d12 [Accessed 24 April 2019].

Burnham, T. 2019. Test-Driven React. Beta. Ebook. Pragmatic Bookshelf. Available at: https://pragprog.com/ [Accessed 26 March 2019].

Center for Innovation in Research and Teaching. 2019. Case Study Method. Available at: https://cirt.gcu.edu/research/developmentresources/research_ready/descriptive/case_study [Accessed 23 May 2019].

Dodds, K. 2017a. Making your UI tests resilient to change. WWW document. Available at: https://kentcdodds.com/blog/making-your-ui-tests-resilient-to-change [Accessed 5 May 2019].

Dodds, K. 2017b. Write tests. Not too many. Mostly integration. WWW document. Available at: https://kentcdodds.com/blog/write-tests [Accessed 24 April 2019].

Fenton, S. 2014. Pro TypeScript: Application-Scale JavaScript Development. Erfurt: Apress.

Fishwick, J. 2018. Best practices for unit testing with a React/Redux approach. WWW document. Available at: https://willowtreeapps.com/ideas/best-practices-for-unit-testing-with-a-react-redux-approach [Accessed 24 April 2019].

Fowler, M. 2004a. AssertionFreeTesting. WWW document. Available at: https://martinfowler.com/bliki/AssertionFreeTesting.html [Accessed 24 April 2019].

Fowler M. 2013a. ContinuousDelivery. WWW document. Available at: https://martinfowler.com/bliki/ContinuousDelivery.html [Accessed 24 April 2019].

Fowler M. 2013b. DeploymentPipeline. WWW document. Available at: https://martinfowler.com/bliki/DeploymentPipeline.html [Accessed 24 April 2019].

Fowler, M. 2013c. ExtremeProgramming. WWW document. Updated 11 July 2013. Available at: https://martinfowler.com/bliki/ExtremeProgramming.html [Accessed 3 March 2019].

Fowler, M. 2004b. Mocks Aren't Stubs. WWW document. Updated 2 January 2007. Available at: https://martinfowler.com/articles/mocksArentStubs.html [Accessed 3 March 2019].

Fowler, M. 2005. TestDrivenDevelopment. WWW document. Available at: https://martinfowler.com/bliki/TestDrivenDevelopment.html [Accessed 24 April 2019].

Fowler, M. 2014. UnitTest. WWW document. Updated 5 May 2014. Available at: https://martinfowler.com/bliki/UnitTest.html [Accessed 20 November 2018].

GitLab. 2019. About Gitlab. WWW document. Available at: https://about.gitlab.com/company/ [Accessed 24 April 2019].

GitLab Documentation. 2019a. Merge requests. WWW document. Available at: https://docs.gitlab.com/ee/user/project/merge_requests/ [Accessed 24 April 2019].

GitLab Documentation. 2019b. Testing levels. WWW document. Available at: https://docs.gitlab.com/ee/development/testing_guide/testing_levels.html [Accessed 24 April 2019].

Google. 2019. Firebase Documentation. FirebaseApp. WWW document. Available at: https://firebase.google.com/docs/reference/android/com/google/firebase/FirebaseApp [Accessed 6 May 2019].

Greif, S., Benitte, R. & Rambeau, M. 2018a. The State of JavaScript 2018: Awards. WWW document. Available at: https://2018.stateofjs.com/awards/ [Accessed 27 January 2019].

Greif, S., Benitte, R. & Rambeau, M. 2018b. The State of JavaScript 2018: Front-End Frameworks - Conclusion. WWW document. Available at: https://2018.stateofjs.com/front-end-frameworks/conclusion/ [Accessed 3 March 2019].

Greif, S., Benitte, R. & Rambeau, M. 2018c. The State of JavaScript 2018: Introduction. WWW document. Available at: https://2018.stateofjs.com/introduction/ [Accessed 27 January 2019].

Greif, S., Benitte, R. & Rambeau, M. 2018d. The State of JavaScript 2018: Testing – Conclusion. WWW document. Available at: https://2018.stateofjs.com/testing/conclusion/ [Accessed 27 January 2019].

Greif, S., Benitte, R. & Rambeau, M. 2018e. The State of JavaScript 2018: Testing – Jest. WWW document. Available at: https://2018.stateofjs.com/testing/jest/ [Accessed 27 January 2019].

Hanlon, R. 2018. Understanding Jest Mocks. WWW document. Updated 8 March 2018. Available at: https://medium.com/@rickhanlonii/understanding-jest-mocks-f0046c68e53c [Accessed 28 February 2019].

Humble, J. 2017a. Continuous Delivery. Continuous Testing. WWW document. Available at: https://continuousdelivery.com/foundations/test-automation/ [Accessed 24 April 2019].

Humble, J. 2017b. Continuous Delivery. Foundations. WWW document. Available at: https://continuousdelivery.com/foundations/ [Accessed 24 April 2019].

Humble, J. 2017c. Continuous Delivery. Principles. WWW document. Available at: https://continuousdelivery.com/principles/ [Accessed 24 April 2019].

Humble, J. 2017d. Continuous Delivery. Implementing. Patterns. WWW document. Available at: https://continuousdelivery.com/implementing/patterns/ [Accessed 19 May 2019].

Identity-obj-proxy. 2019. GitHub. WWW document. Available at: https://github.com/keyz/identity-obj-proxy#readme [Accessed 19 May 2019].

Istanbul Documentation. 2019. WWW document. Available at: http://gotwarlost.github.io/istanbul/public/apidocs/ [Accessed 24 April 2019].

Jenkins Documentation. 2017. WWW document. Available at: https://jenkins.io/doc/ [Accessed 24 April 2019].

Jest. 2019. Home page. WWW document. Available at: https://jestjs.io/en/ [Accessed 19 May 2019].

Jest API Reference. 2019a. Configuring Jest. WWW document. Available at: https://jestjs.io/docs/en/configuration [Accessed 24 April 2019].

Jest API Reference. 2019b. Expect. WWW document. Available at: https://jestjs.io/docs/en/expect [Accessed 19 February 2019].

Jest API Reference. 2019c. Jest CLI Options. WWW document. Available at: https://jestjs.io/docs/en/cli [Accessed 24 April 2019].

Jest API Reference. 2019d. Mock Functions. WWW document. Available at: https://jestjs.io/docs/en/mock-function-api [Accessed 24 April 2019].

Jest Documentation. 2019a. Mock Functions. WWW document. Available at: https://jestjs.io/docs/en/mock-functions.html [Accessed 24 April 2019].

Jest Documentation. 2019b. Setup and Teardown. WWW document. Available at: https://jestjs.io/docs/en/setup-teardown [Accessed 24 April 2019].

Jest Documentation. 2019c. Testing Asynchronous Code. WWW document. Available at: https://jestjs.io/docs/en/asynchronous [Accessed 24 April 2019].

Jest Documentation. 2019d. Using Matchers. WWW document. Available at: https://jestjs.io/docs/en/using-matchers [Accessed 27 January 2019].

Jest Guides. 2019a. Manual Mocks. WWW document. Available at: https://jestjs.io/docs/en/manual-mocks [Accessed 24 April 2019].

Jest Guides. 2019b. Snapshot Testing. WWW document. Available at: https://jestjs.io/docs/en/snapshot-testing [Accessed 24 April 2019].

Johansen, C. 2010. Test-Driven JavaScript Development. 1st edition. Crawfordswille: Pearson Education, Inc.

Justice, J. 2018. Why Do JavaScript Test Frameworks Use describe() and beforeEach()? WWW document. Available at: https://www.bignerdranch.com/blog/why-do-javascript-test-frameworks-use-describe-and-beforeeach/ [Accessed 24 April 2019].

Kim, G., Humble, J., Debois, P. & Willis, J. 2016. The DevOps Handbook. How to create world-class agility, reliability, & security in technology organizations. 1st edition. Portland: IT Revolution Press, LLC.

Kit, E. 1995. Software Testing in the Real World: Improving the Process. Ebook. Addison-Wesley. Available at: https://www.amazon.com/ [Accessed 23 May 2019].

Koskela, L. 2013. Effective Unit Testing. Shelter Island: Manning Publications Co.

Marick, B. 1997. How to Misuse Code Coverage. WWW document. Available at: http://www.exampler.com/testing-com/writings/coverage.pdf [Accessed 24 April 2019].

Marston, M. & Dees, I. 2017. Effective Testing with RSpec 3. Ebook. Pragmatic Bookshelf. Available at: https://pragprog.com/ [Accessed 26 March 2019].

Martin, R. 2017. Types and Tests. WWW document. Available at: https://blog.cleancoder.com/uncle-bob/2017/01/13/TypesAndTests.html [Accessed 24 April 2019].

McGinnis, T. 2016. React Elements vs React Components. WWW document. Updated 15 December 2016. Available at: https://tylermcginnis.com/react-elements-vs-react-components/ [Accessed 3 March 2019].

McGinnis, T. 2017. Why React? An overview of React, Webpack 2, React Router v4, and Babel. Video clip. Available at: https://www.youtube.com/watch?v=ul0tRkeu5CE [Accessed 3 March 2019].

Meszaros, G. 2007. xUnit Test Patterns: Refactoring Test Code. Ebook. Addison-Wesley. Available at: https://www.amazon.com/ [Accessed 26 March 2019].

Micco, J. 2016. Flaky Tests at Google and How We Mitigate Them. WWW document. Available at: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html [Accessed 24 April 2019].

Molinari, N. 2018. Testing in React: best practices, tips and tricks. WWW document. Updated 20 June 2018. Available at: https://techblog.commercetools.com/testing-in-react-best-practices-tips-and-tricks-577bb98845cd [Accessed 24 April 2019].

North, D. 2012. BDD is like TDD if… WWW document. Available at: https://dannorth.net/2012/05/31/bdd-is-like-tdd-if/ [Accessed 23 May 2019].

North, D. 2006. Introducing BDD. WWW document. Available at: https://dannorth.net/introducing-bdd/ [Accessed 23 May 2019].

Ortega, A. 2019. Test-driven React.js Development: React.js Unit Testing with Enzyme and Jest. WWW document. Available at: https://www.toptal.com/react/tdd-react-unit-testing-enzyme-jest [Accessed 24 April 2019].

Pittet, S. 2019a. An introduction to code coverage. WWW document. Available at: https://www.atlassian.com/continuous-delivery/software-testing/code-coverage [Accessed 24 April 2019].

Pittet, S. 2019b. The different types of software testing. WWW document. Available at: https://www.atlassian.com/continuous-delivery/different-types-of-software-testing [Accessed 24 April 2019].

Pojer, C. 2016. Jest 14.0: React Tree Snapshot Testing. WWW document. Available at: https://jestjs.io/blog/2016/07/27/jest-14.html [Accessed 24 April 2019].

Pysarenko, A. 2019. What and How to Test with Jest and Enzyme. Full Instruction on React Components Testing. WWW document. Updated 17 April 2019. Available at: https://djangostars.com/blog/what-and-how-to-test-with-enzyme-and-jest-full-instruction-on-react-component-testing/ [Accessed 24 April 2019].

React API Reference. 2019a. React.Component. WWW document. Available at https://reactjs.org/docs/react-component.html [Accessed 4 March 2019].

React API Reference. 2019b. Shallow Renderer. WWW document. Available at: https://reactjs.org/docs/shallow-renderer.html [Accessed 23 May 2019].

React Documentation 2019a. Components and Props. WWW document. Available at: https://reactjs.org/docs/components-and-props.html [Accessed 19 May 2019]

React Documentation. 2019b. FAQ. Component State. WWW document. Available at: https://reactjs.org/docs/faq-state.html [Accessed 19 May 2019]

React Documentation. 2019c. FAQ. Virtual DOM and Internals. WWW document. Available at: https://reactjs.org/docs/faq-internals.html#what-is-the-virtual-dom [Accessed 4 March 2019].

React Documentation. 2019d. Hello World. WWW document. Available at https://reactjs.org/docs/hello-world.html [Accessed 4 March 2019].

React Documentation. 2019e. Introducing JSX. WWW document. Available at https://reactjs.org/docs/introducing-jsx.html [Accessed 3 April 2019].

React Documentation. 2019f. Rendering Elements. WWW document. Available at https://reactjs.org/docs/rendering-elements.html [Accessed 19 May 2019]

React Documentation. 2019g. State and Lifecycle. WWW document. Available at https://reactjs.org/docs/state-and-lifecycle.html [Accessed 4 March 2019].

Stackify. 2017. The Ultimate List of Code Coverage Tools: 25 Code Coverage Tools for C, C++, Java, .NET, and More. WWW document. Available at: https://stackify.com/code-coverage-tools/ [Accessed 24 April 2019].

The Code Gang. 2019. Flaky Tests - A War that Never Ends. WWW document. Available at: https://hackernoon.com/flaky-tests-a-war-that-never-ends-9aa32fdef359 [Accessed 24 April 2019].

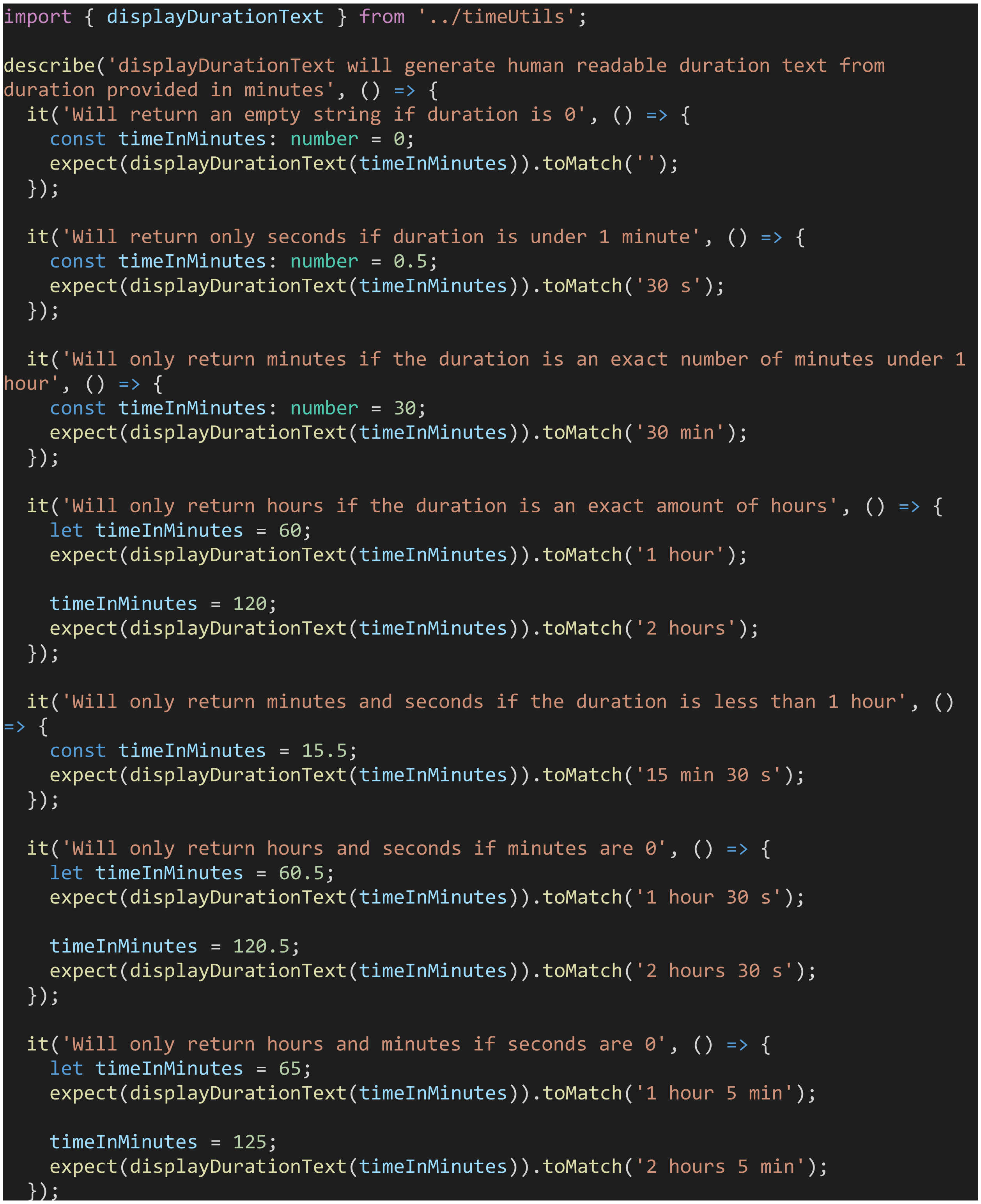
Ts-Mock-Firebase. 2019. GitHub. WWW document. Updated 24 April 2019. Available at: https://github.com/mindhivefi/ts-mock-firebase [Accessed 4 May 2019].

Ts-Jest. 2019. GitHub. WWW document. Available at: https://kulshekhar.github.io/ts-jest/ [Accessed 19 May 2019].

Vocke, H. 2018. The Practical Testing Pyramid. WWW document. Updated 26 February 2018. Available at: https://martinfowler.com/articles/practical-test-pyramid.html [Accessed 25 November 2018].

Wacker, M. 2015. Just Say No to More End-to-End Tests. WWW document. Available at: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html [Accessed 24 April 2019].

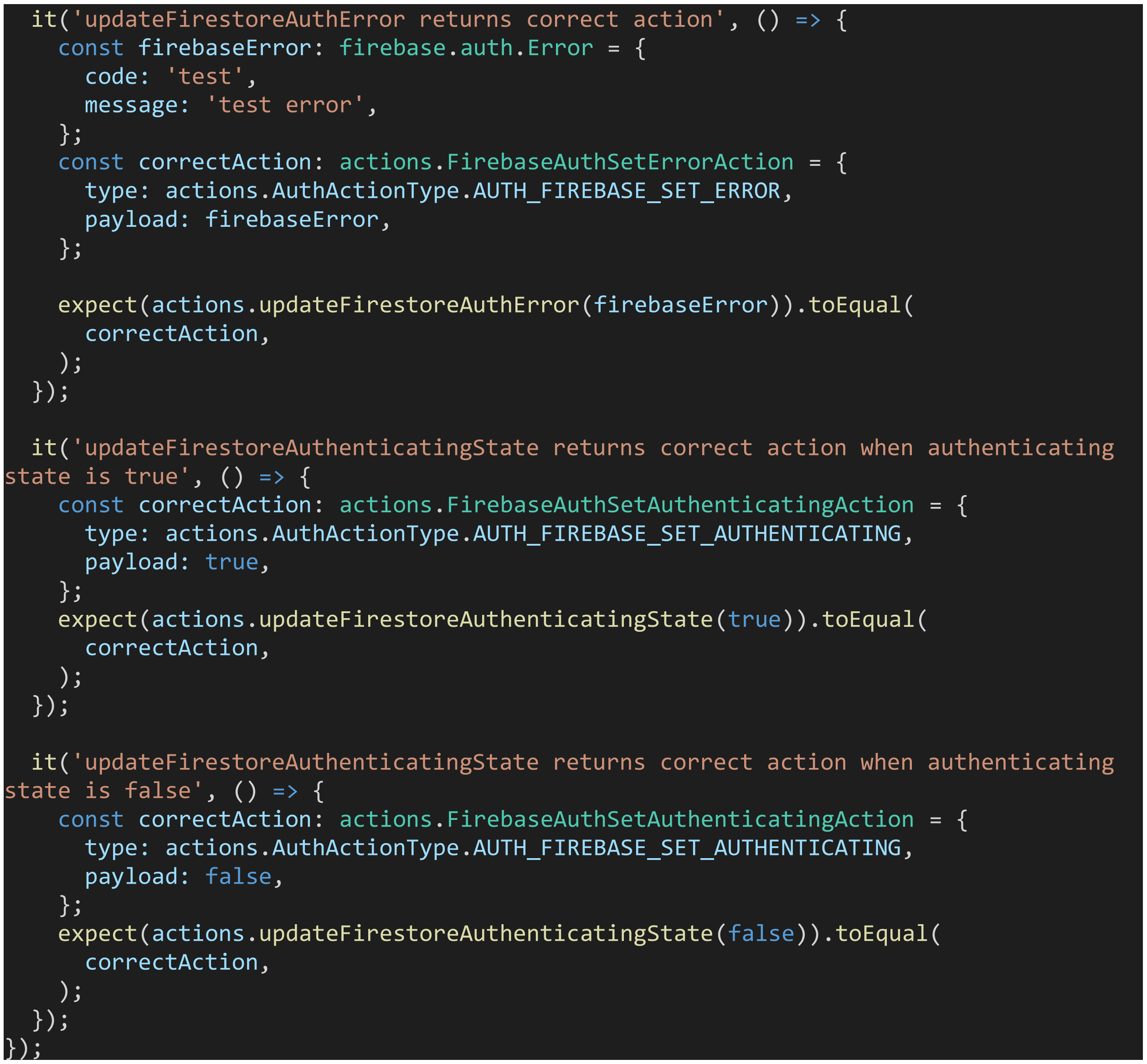
APPENDIX 1. Full test suite of the displayDurationText() function



```
import { displayDurationText } from '../timeUtils';

describe('displayDurationText will generate human readable duration text from
duration provided in minutes', () => {
  it('Will return an empty string if duration is 0', () => {
    const timeInMinutes: number = 0;
    expect(displayDurationText(timeInMinutes)).toMatch('');
  });

  it('Will return only seconds if duration is under 1 minute', () => {
    const timeInMinutes: number = 0.5;
    expect(displayDurationText(timeInMinutes)).toMatch('30 s');
  });

  it('Will only return minutes if the duration is an exact number of minutes under 1
hour', () => {
    const timeInMinutes: number = 30;
    expect(displayDurationText(timeInMinutes)).toMatch('30 min');
  });

  it('Will only return hours if the duration is an exact amount of hours', () => {
    let timeInMinutes = 60;
    expect(displayDurationText(timeInMinutes)).toMatch('1 hour');

    timeInMinutes = 120;
    expect(displayDurationText(timeInMinutes)).toMatch('2 hours');
  });

  it('Will only return minutes and seconds if the duration is less than 1 hour', ()
=> {
    const timeInMinutes = 15.5;
    expect(displayDurationText(timeInMinutes)).toMatch('15 min 30 s');
  });

  it('Will only return hours and seconds if minutes are 0', () => {
    let timeInMinutes = 60.5;
    expect(displayDurationText(timeInMinutes)).toMatch('1 hour 30 s');

    timeInMinutes = 120.5;
    expect(displayDurationText(timeInMinutes)).toMatch('2 hours 30 s');
  });

  it('Will only return hours and minutes if seconds are 0', () => {
    let timeInMinutes = 65;
    expect(displayDurationText(timeInMinutes)).toMatch('1 hour 5 min');

    timeInMinutes = 125;
    expect(displayDurationText(timeInMinutes)).toMatch('2 hours 5 min');
  });
  it('Will display hours, minutes and seconds if none of them are 0', () => {
    let timeInMinutes = 65.5;
    expect(displayDurationText(timeInMinutes)).toMatch('1 hour 5 min 30 s');

    timeInMinutes = 125.5;
    expect(displayDurationText(timeInMinutes)).toMatch('2 hours 5 min 30 s');
  });

  it('Will throw an error if the input is not a number', () => {
    const timeInMinutes: any = { duration: 120 };
    expect(() => displayDurationText(timeInMinutes)).toThrowError();
  });

  it('Will throw an error if the input is negative', () => {
    const timeInMinutes: number = -120;
    expect(() => displayDurationText(timeInMinutes)).toThrowError();
  });
});
```

APPENDIX 2. Full test suite for the authentication action creators

```
import { CompanyId, User } from '@shared/schema';
import { getFirebaseUser } from '__specs__/__helpers__/firebaseHelpers';
import { getUser } from '__specs__/__helpers__/specHelpers';
import * as actions from '../authActions';

describe('authAction creator', () => {
  it('firebaseUserStateChanged returns correct action when user is undefined', () =>
{
    const user = undefined;
    const correctAction: actions.FirebaseUserAuthStateChangedAction = {
      type: actions.AuthActionType.AUTH_FIREBASE_USER_STATE_CHANGED,
      payload: user,
    };
    expect(actions.firebaseUserStateChanged(user)).toEqual(correctAction);
  });

  it('firebaseUserStateChanged returns correct action when user is defined', () => {
    const user = getFirebaseUser();
    const correctAction: actions.FirebaseUserAuthStateChangedAction = {
      type: actions.AuthActionType.AUTH_FIREBASE_USER_STATE_CHANGED,
      payload: user,
    };
    expect(actions.firebaseUserStateChanged(user)).toEqual(correctAction);
  });

  it('appUserStateChanged returns correct action when user is undefined', () => {
    const user = undefined;
    const correctAction: actions.AppUserAuthStateChangedAction = {
      type: actions.AuthActionType.AUTH_APP_USER_INFO_UPDATED,
      payload: user,
    };
    expect(actions.appUserStateChanged(user)).toEqual(correctAction);
  });

  it('appUserStateChanged returns correct action when user is defined', () => {
    const user: User = getUser();
    const correctAction: actions.AppUserAuthStateChangedAction = {
      type: actions.AuthActionType.AUTH_APP_USER_INFO_UPDATED,
      payload: user,
    };
    expect(actions.appUserStateChanged(user)).toEqual(correctAction);
  });

  it('appUserActiveOrganizationChanged returns correct action', () => {
    const companyId: CompanyId = 'TestCompany';
    const correctAction: actions.AppUserActiveOrganizationChangedAction = {
      type: actions.AuthActionType.AUTH_APP_USER_ACTIVE_ORG_CHANGED,
      payload: companyId,
    };
    expect(actions.appUserActiveOrganizationChanged(companyId)).toEqual(
      correctAction,
    );
  });
```



```
  it('updateFirestoreAuthError returns correct action', () => {
    const firebaseError: firebase.auth.Error = {
      code: 'test',
      message: 'test error',
    };
    const correctAction: actions.FirebaseAuthSetErrorAction = {
      type: actions.AuthActionType.AUTH_FIREBASE_SET_ERROR,
      payload: firebaseError,
    };

    expect(actions.updateFirestoreAuthError(firebaseError)).toEqual(
      correctAction,
    );
  });

  it('updateFirestoreAuthenticatingState returns correct action when authenticating
state is true', () => {
    const correctAction: actions.FirebaseAuthSetAuthenticatingAction = {
      type: actions.AuthActionType.AUTH_FIREBASE_SET_AUTHENTICATING,
      payload: true,
    };
    expect(actions.updateFirestoreAuthenticatingState(true)).toEqual(
      correctAction,
    );
  });

  it('updateFirestoreAuthenticatingState returns correct action when authenticating
state is false', () => {
    const correctAction: actions.FirebaseAuthSetAuthenticatingAction = {
      type: actions.AuthActionType.AUTH_FIREBASE_SET_AUTHENTICATING,
      payload: false,
    };
    expect(actions.updateFirestoreAuthenticatingState(false)).toEqual(
      correctAction,
    );
  });
});
```

APPENDIX 3. Full test suite for the authentication reducer

```
import { User, UserRole } from '@shared/schema';
import { getFirebaseUser } from '__specs__/__helpers__/firebaseHelpers';
import { getUser } from '__specs__/__helpers__/specHelpers';
import { AnyAction } from 'redux';
import * as actions from '../authActions';
import { authReducer, AuthState, defaultAuthState } from '../authReducer';

describe('authReducer', () => {
  let initialState: AuthState;
  beforeEach(() => {
    initialState = defaultAuthState;
  });

  it('does not modify state if action is undefined', () => {
    const action: AnyAction = {
      type: 'UNSUPPORTED_ACTION',
    };
    expect(authReducer(initialState, action)).toEqual(initialState);
  });

  it('upon receivig AUTH_FIREBASE_USER_STATE_CHANGED updates user in state, sets
"firebaseError" to "undefined" and "authenticating" to "false"', () => {
    const user = getFirebaseUser();
    const action: actions.FirebaseUserAuthStateChangedAction = {
      type: actions.AuthActionType.AUTH_FIREBASE_USER_STATE_CHANGED,
      payload: user,
    };
    const expectedState = {
      ...initialState,
      firebaseUser: action.payload,
      firebaseError: undefined,
      authenticating: false,
    };
    expect(authReducer(initialState, action)).toEqual(expectedState);
  });

  it('upon receiving AUTH_APP_USER_INFO_UPDATED with admin user payload updates user
and sets activeRole to admin', () => {
    const user = getUser();
    user.systemAdmin = true;
    const action: actions.AppUserAuthStateChangedAction = {
      type: actions.AuthActionType.AUTH_APP_USER_INFO_UPDATED,
      payload: user,
    };
    const expectedState = {
      ...initialState,
      appUser: action.payload,
      activeRole: UserRole.SYSTEM_ADMIN,
    };
    expect(authReducer(initialState, action)).toEqual(expectedState);
  });
  it('upon receiving AUTH_APP_USER_INFO_UPDATED updates user and role for a non-admin
user', () => {
    const user: User = getUser();
    user.companies = {
      home: UserRole.USER,
    };
    const action: actions.AppUserAuthStateChangedAction = {
      type: actions.AuthActionType.AUTH_APP_USER_INFO_UPDATED,
      payload: user,
    };
    const expectedState = {
      ...initialState,
      appUser: action.payload,
      activeRole: UserRole.USER,
    };
    expect(authReducer(initialState, action)).toEqual(expectedState);
  });

  it('upon receiving AUTH_FIREBASE_SET_ERROR sets firebaseError and changes
"authenticating" to false', () => {
    const firebaseError: firebase.auth.Error = {
      code: 'test',
      message: 'test error',
    };
    const action: actions.FirebaseAuthSetErrorAction = {
      type: actions.AuthActionType.AUTH_FIREBASE_SET_ERROR,
      payload: firebaseError,
    };
    const expectedState = {
      firebaseError: action.payload,
      authenticating: false,
      ...initialState,
    };
    expect(authReducer(initialState, action)).toEqual(expectedState);
  });

  it('upon receiving AUTH_FIREBASE_SET_AUTHENTICATING updates "authenticating" to
payload', () => {
    const action: actions.FirebaseAuthSetAuthenticatingAction = {
      type: actions.AuthActionType.AUTH_FIREBASE_SET_AUTHENTICATING,
      payload: true,
    };
    const expectedState = {
      authenticating: true,
      ...initialState,
    };
    expect(authReducer(initialState, action)).toEqual(expectedState);
  });
});
```

APPENDIX 4. Full test suite for the AccountForm component

```
import { User } from '@shared/schema';
import { getUser } from '__specs__/__helpers__/specHelpers';
import { shallow } from 'enzyme';
import toJson from 'enzyme-to-json';
import * as React from 'react';
import AccountForm, { AccountProps } from '../AccountForm';

const setup = (propOverrides?: any) => {
  const user: User = getUser();

  const props: AccountProps = Object.assign(
    {
      initialValues: user,
      onSubmit: jest.fn(),
      onUpdateAccount: jest.fn(),
    },
    propOverrides,
  );

  const wrapper = shallow(<AccountForm {...props} />);
  const displayNameTextField = wrapper.find('[data-test="displayName"]');
  const emailTextField = wrapper.find('[data-test="email"]');
  const homeTextField = wrapper.find('[data-test="home"]');
  const submitButton = wrapper.find('[data-test="submitButton"]');

  return {
    props,
    wrapper,
    displayNameTextField,
    emailTextField,
    homeTextField,
    submitButton,
  };
};

describe('<AccountForm />', () => {
  it('renders', () => {
    const { wrapper } = setup();
    expect(wrapper.exists()).toBe(true);
  });

  it('displays the displayName property of the initialValues prop', () => {
    const { props, displayNameTextField } = setup();
    expect(displayNameTextField.prop('value')).toBe(
      props.initialValues.displayName,
    );
  });

  it('displays the email property of the initialValues prop', () => {
    const { props, emailTextField } = setup();
    expect(emailTextField.prop('value')).toBe(props.initialValues.email);
  });
  it('displays the home property of the initialValues prop', () => {
    const { props, homeTextField } = setup();
    expect(homeTextField.prop('value')).toBe(props.initialValues.home);
  });

  it('calls the onSubmit callback prop when the submit button is clicked', () => {
    const { props, submitButton } = setup();
    submitButton.simulate('click');
    expect(props.onSubmit).toHaveBeenCalled();
  });

  it('calls handleChange with updated account data when the displayName is edited',
() => {
    const handleChangeSpy = jest.spyOn(AccountForm.prototype, 'handleChange');
    const { displayNameTextField } = setup();
    displayNameTextField.simulate('change', { target: { value: 'new name' } });
    expect(handleChangeSpy).toHaveBeenCalledWith({
      target: { value: 'new name' },
    });
  });

  it('calls onUpdateAccount from handleChange if onUpdateAccount is defined', () => {
    const { displayNameTextField, props } = setup();
    displayNameTextField.simulate('change', { target: { value: 'new name' } });
    expect(props.onUpdateAccount).toHaveBeenCalledWith({
      ...props.initialValues,
      displayName: 'new name',
    });
  });

  it('matches snapshot', () => {
    const { wrapper } = setup();
    expect(toJson(wrapper)).toMatchSnapshot();
  });
});
```

APPENDIX 5. Full test suite for the AccountContainer component

```
import { Schema, ShortUserInfo, UserRole } from '@shared/schema';
import { getUser } from '__specs__/__helpers__/specHelpers';
import { shallow } from 'enzyme';
import toJson from 'enzyme-to-json';
import firebaseApp from 'firebaseApp';
import * as React from 'react';
import { AuthState, defaultAuthState } from 'reducers/auth/authReducer';
import defaultUserImageUrl from 'static/User.png';
import { exposeMockFirebaseApp, MockDatabase } from 'ts-mock-firebase';
import { AccountContainer, AccountContainerProps } from '../AccountContainer';

const setup = async (propOverrides?: any) => {
  const mockApp = exposeMockFirebaseApp(firebaseApp);
  const database: MockDatabase = {
    users: {
      docs: {
        '1000': {
          data: {
            displayName: 'Tester',
            id: '1000',
            email: 'test@test.test',
            photoURL: 'photoURL',
            home: 'home',
          },
        },
      },
    },
  };

  mockApp.firestore().mocker.fromMockDatabase(database);

  const authState: AuthState = {
    firebaseUser: undefined,
    appUser: {
      displayName: 'Tester',
      id: '1000',
      email: 'test@test.test',
      photoURL: 'photoURL',
    },
    activeRole: UserRole.USER,
    firebaseError: undefined,
    authenticating: false,
  };

  const props: AccountContainerProps = Object.assign(
    {
      auth: authState,
    },
    propOverrides,
  );

  const wrapper = await shallow(<AccountContainer {...props} />);
  const accountForm = wrapper.find('[data-test="accountForm"]');
  const avatar = wrapper.find('[data-test="avatar"]');
```

```
return {
    mockApp,
    wrapper,
    props,
    accountForm,
    avatar,
  };
};

describe('<AccountContainer />', () => {
  it('renders', async () => {
    const { wrapper } = await setup({ authState: defaultAuthState });
    expect(wrapper.exists()).toBe(true);
  });
it('uses the id of the auth prop to get the account data from database on
componentDidMount and puts it to state', async () => {
    const setStateSpy = jest.spyOn(AccountContainer.prototype, 'setState');

    const { wrapper } = await setup();

    const expectedState = {
      isLoading: false,
      account: {
        displayName: 'Tester',
        id: '1000',
        email: 'test@test.test',
        photoURL: 'photoURL',
        home: 'home',
      },
    };

    expect(setStateSpy).toHaveBeenCalledWith(expectedState);
    expect(wrapper.state()).toEqual(expectedState);
  });

  it('keeps the "isLoading" state variable true if the account is not found in
database', async () => {
    const user = getUser();
    user.id = 'wrongId';

    const authState: AuthState = {
      firebaseUser: undefined,
      appUser: user,
      activeRole: UserRole.USER,
      firebaseError: undefined,
      authenticating: false,
    };

    const { wrapper } = await setup({ auth: authState });
    expect(wrapper.state('isLoading')).toBe(true);
  });

  it('renders a loading spinner if the "isLoading" state variable is true', async ()
=> {
    const { wrapper } = await setup();
    wrapper.setState({ isLoading: true });
    wrapper.update();
    wrapper.instance().forceUpdate();
    expect(wrapper.find('[data-test="loadingSpinner"]').exists()).toBe(true);
  });

  describe('when the "account" state variable is defined', () => {
    it('renders an AccountForm component with account, onSubmit and onUpdateAccount
props', async () => {
      const { accountForm } = await setup();
      expect(accountForm.exists()).toBe(true);
    });

    it('passes the "account" state variable to ApplicationForm as a prop', async ()
=> {
      const { wrapper, accountForm } = await setup();
      expect(accountForm.prop('initialValues')).toEqual(
        wrapper.state('account'),
      );
    });

    it('updates the account in state when AccountForm triggers onUpdateAccount',
async () => {
      const { accountForm, wrapper } = await setup();
      const changedAccount: ShortUserInfo = {
        displayName: 'Changed name',
        id: '1000',
        email: 'test@test.test',
        photoURL: 'photoURL',
      };

      accountForm.simulate('updateAccount', changedAccount);

      expect(wrapper.state('account')).toEqual({
        displayName: 'Changed name',
        id: '1000',
        email: 'test@test.test',
        photoURL: 'photoURL',
      });
    });

    it('updates the account in the database when AccountForm triggers onSubmit',
async () => {
      const { accountForm, mockApp, wrapper } = await setup();

      wrapper.setState({
        account: {
          displayName: 'Updated name',
          id: '1000',
          email: 'test@test.test',
          photoURL: 'photoURL',
          home: 'home',
        },
      });
      accountForm.simulate('submit');

      return mockApp
        .firestore()
        .collection(Schema.USERS)
        .doc('1000')
        .get()
        .then(updatedDocument => {
          expect(updatedDocument.data()).toEqual({
            displayName: 'Updated name',
            id: '1000',
            email: 'test@test.test',
            photoURL: 'photoURL',
            home: 'home',
          });
        });
    });

    it('renders the avatar of the account if account.photoURL is defined', async ()
=> {
      const { wrapper } = await setup();
      expect(wrapper.find('[data-test="avatarImage"]').prop('src')).toBe(
        'photoURL',
      );
    });

    it('renders the default avatar if account.photoURL is not defined', async () => {
      const { wrapper } = await setup();
      wrapper.setState({
        account: {
          displayName: 'Updated name',
          id: '1000',
          email: 'test@test.test',
          photoURL: undefined,
          home: 'home',
        },
      });

      expect(wrapper.find('[data-test="avatarImage"]').prop('src')).toBe(
        defaultUserImageUrl,
      );
    });
  });

  it('matches snapshot', async () => {
    const { wrapper } = await setup();
    expect(toJson(wrapper)).toMatchSnapshot();
  });

});
```