University
of Stavanger

MARIE GRØTTE LARSEN, SYNNE MARIE SÆVIK

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

QuickFeed Frontend Design Improvements and Robustness Testing

Bachelor's Thesis - Computer Science - May 2022


```
func (m *Manager) NewConfiguration(opts ...gorums.ConfigOption) (c *Configuration, err error) {
    if len(opts) < 1 || len(opts) > 2 {
        return nil, fmt.Errorf("wrong number of options: %d", len(opts))
    }
    c = &Configuration{}
    for _, opt := range opts {
        switch v := opt.(type) {
        case gorums.NodeListOption:
            c.Configuration, err = gorums.NewConfiguration(m.Manager, v)
            if err != nil {
                return nil, err
            }
        case QuorumSpec:
            // Must be last since v may match QuorumSpec if it is interface{}
            c.qspec = v
        default:
            return nil, fmt.Errorf("unknown option type: %v", v)
        }
    }
    // return an error if the QuorumSpec interface is not empty and no implementati
    var test interface{} = struct{}{}
    if _, empty := test.(QuorumSpec); !empty && c.qspec == nil {
        return nil, fmt.Errorf("missing required QuorumSpec")
    }
    return c, nil
}
```

I, **Marie Grøtte Larsen, Synne Marie Sævik**, declare that this thesis titled, "QuickFeed Frontend Design Improvements and Robustness Testing" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a bachelor's degree at the University of Stavanger.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

“Programming is a nice break from thinking.”

– Leslie Lamport

Abstract

Background The goal of this project was to assist with frontend tests and design on QuickFeed, a React application. QuickFeed recently had a re-implementation of the frontend. Our task was to further improve the frontend code, and to implement QuickFeeds first frontend tests. Learning about and understanding the importance of tests in the frontend, and not only in the backend is important to develop successful applications. For that reason, we implemented multiple different tests to test both frontend logic and frontend UI.

Result In result we managed to make automatic, useful tests for the frontend, while following good practises and structures.

Acknowledgements

We would like to thank our supervisor Professor Hein Meling for his help with making this thesis. He was a great support with beneficial weekly meetings, and always being available for support or questions when needed. We would also like to thank Jostein Lindhom for sharing his experience with QuickFeed frontend with us, and also always being available for questions or support.

Contents

Abstract iii

Acknowledgements iv

1 Introduction 2
- 1.1 Background . . . 2
  - 1.1.1 React . . . 2
  - 1.1.2 TypeScript . . . 3
- 1.2 Motivation . . . 3
  - 1.2.1 User experience and efficiency with frontend tests . . . 3
  - 1.2.2 Improving the usability, design and layout . . . 4

2 Background 5
- 2.1 The Testing Pyramid . . . 5
  - 2.1.1 Unit Testing . . . 6
  - 2.1.2 Integration Testing . . . 7
  - 2.1.3 End to End Testing . . . 8
- 2.2 Snapshot Testing . . . 9
- 2.3 Usability and User Experience . . . 9
- 2.4 Mocking . . . 9
- 2.5 Continuous Integration and Continuous Delivery of Tests . . . 10

3 Choosing Testing Frameworks and Libraries 12
- 3.1 Evaluating testing frameworks and libraries . . . 12
  - 3.1.1 Criteria when choosing a framework . . . 12
  - 3.1.2 Rainforest . . . 13
  - 3.1.3 Jest . . . 14

3.1.4 Enzyme . . . 15
3.1.5 Mocha . . . 16
3.1.6 Selenium . . . 17
3.2 Choosing a testing framework . . . 18

4 Implementation **20**
4.1 Introduction . . . 20
4.1.1 Requirements . . . 20
4.2 Structure . . . 21
4.2.1 Choosing Operating System . . . 21
4.2.2 Setting up Jest and Enzyme . . . 21
4.2.3 Setting up Selenium . . . 22
4.3 Testing . . . 23
4.3.1 Unit testing . . . 23
4.3.2 Integration testing . . . 26
4.3.3 End to End testing . . . 28
4.3.4 Snapshot testing . . . 34
4.3.5 Responsive web design tests . . . 35
4.4 Design . . . 38
4.4.1 Usability and User Experience . . . 38

5 Discussion **44**
5.1 Results . . . 44
5.2 Choosing what to test . . . 45
5.3 Limitations of testing . . . 45
5.4 Technical credibility . . . 46
5.5 Future work . . . 47
5.5.1 Tests . . . 47
5.5.2 GrpcManager mock . . . 48
5.5.3 Design . . . 48
5.5.4 Test Driven Development (TDD) . . . 50

6 Conclusions **52**

A Instructions to Compile and Run System **57**
A.1 How to run the tests 57
A.1.1 Setup the test environment - Jest 57
A.1.2 Setup the test environment - Selenium 57
A.1.3 Run the jest tests 58
A.1.4 how to run Selenium tests 58

B Attachments **60**

Chapter 1

Introduction

Quickfeed is an automated student feedback application developed at the University of Stavanger. It is very valuable for students, which gets a quick feedback with a score, making learning and hand-ins more efficient. Teachers will also get benefits from using QuickFeed, such as being able to spend time on helping students, instead of grading. The teachers also have the ability to customize their tests according to their own needs.

Our thesis will mainly focus on implementing frontend tests, and to improve the usability of the frontend design. To be able to do this, we will have to look for weaknesses in both the code and the design.

1.1 Background

QuickFeed recently had an re-implementation of the frontend, and the backend is being continuously improved to better its robustness. When QuickFeed had its re-implementation, it was decided that the frontend was going to be made with the TypeScript language, and the React library.

1.1.1 React

React is a open source JavaScript library, actually not a framework, used to build user interfaces mainly for single page applications. It is based upon reusable components which are the applications building block. Because of the components opportunity to be reused, it will save a lot of development time compared

to having to develop it in vanilla JavaScript. React also does not have any specific ways the developer have to do things. You can add multiple different additional libraries and build it your own way after your own needs, which is one of the reasons React is widely used in the frontend world.

1.1.2 TypeScript

TypeScript is a programming language that builds on JavaScript. It is strongly typed and compiles into JavaScript. There are many advantages to using TypeScript over standard JavaScript. Due to TypeScript being compiled, the compiler can give feedback to the developer. Therefore, the developer have the possibility to catch bugs at the compile time, instead of at the runtime.

1.2 Motivation

QuickFeed's new implementation is currently missing frontend tests. These tests varies from tests that check the logic of the code, to tests that checks the user interface (UI). There are multiple issues which can occur when a big application do not have this implemented.
Since QuickFeed recently had an re-implementation we also found some minor issues with the layout we wanted to improve. The remainder of this chapter will contain more about these issues and their reasons.

1.2.1 User experience and efficiency with frontend tests

Users do not directly interact with the backend, which mainly makes their experience based on the frontend. Since we want users to have a good experience with QuickFeed, having frontend tests are crucial to make sure the UI are working as it should. Having implemented frontend tests is also crucial to have an efficient development phase and maintenance phase. When new features or designs are added, we can just run the automated tests, which will return a quick and fail-proof response. This is a lot more practical than having to go through the whole process manually which can be very time consuming.

1.2.2 Improving the usability, design and layout

The frontend design, layout and usability are important for the users experience. One of our goals after manually testing out the current QuickFeed, is to simplify the layout with small tweaks. We want to follow more of the webpage-design norms and standards, to make QuickFeed a more user friendly application. These small changes of the layout are going to both improve the usability and the design.

Chapter 2

Background

In this chapter we will talk more about the different types of tests made for the frontend.

2.1 The Testing Pyramid

When writing tests for an application, we need to know which types of testing is needed for different scenarios. The testing pyramid is a concept of grouping testing into three categories. The testing pyramid was first introduced by Mike Cohn in his book *Succeeding with Agile*(5). The pyramid divides testing into three categories; end-to-end tests, integration tests and unit tests.

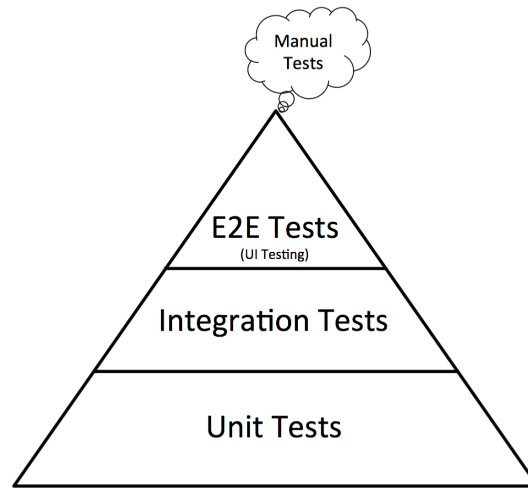


Figure 2.1: The testing pyramid

Though, the names of the tests might vary throughout different sources. The placement in the pyramid indicates the distribution of the amounts of tests that should be produced for each application. The categories closer to the bottom should have the bigger amounts of tests than the categories closer to the top.

At the bottom of the pyramid is unit tests. Simply put unit tests are tests that test small units in isolation. In the middle of the pyramid is integration tests, these test involve interaction between different modules, components etc. At the top of the pyramid are E2E tests, short for End-to-End Tests. E2E are tests that test the functionally of an application, making sure the application runs as flawlessly as possible from end to end.

2.1.1 Unit Testing

Unit testing is testing a isolated piece of code. A so called ”unit”. The unit can be a class, function or simply just a line of code. When making a test combining units, it is not considered a unit test anymore. Given unit tests placement at the bottom of the pyramid, each application should have a bigger amount of unit tests. This is because its scope is small. One test does not cover a lot of code.

Unit tests allow us to test smaller chunks of code in isolation, making it easier

to catch bugs. Having code covered with unit tests will give valuable information if some code were to break. This can save a lot of time, because the time spend debugging or searching for bugs is reduced.

To make the task of testing more manageable, writing tests for isolated parts of the code is a good starting point. Examples of typical unit tests are tests that check the functionally of input fields, validation etc.

2.1.2 Integration Testing

Integration testing is a type of testing where units, modules or components are tested in a group. Integration tests will therefore combine the modules or the components into a single unit. They are mostly useful to test how modules or components work together in an application. In terms of testing a fronted application in react, a integration test may test how different components interact together.

Integration testing consists of different approaches with individual advantages and disadvantages. Some of the most common approaches for frontend testing are the big-bang, top-down, and hybrid testing approaches.

The Big-Bang Approach

The big-bang approach is a testing approach where the whole system is tested at once by integrating all modules together. One of its main advantages is that the whole system can be tested at once. Some of its drawbacks is that it is ineffective for larger systems. When using the big-bang approach for larger systems it is a higher risk of missing issues or bugs, and it can be harder to identify where the issue is originating from.

The Top-Down Approach

The top-down approach is an approach where test starts at the top layers and moves downwards. The top-down approach is especially useful for frontend testing because top layer interface issues will be found at an early stage. It will also be easier to localize where the error is occurring when its localized early. If the

tests makes its way down to the lower layers, and they are not yet integrated we can use stubs. Stubs are a module that can simulate the behaviour of the actual lower level modules.

The Hybrid Approach

The hybrid approach is viewed as three different layers; the main target layer, the layer over the target layer, and the layer under the target layer. The testing is primarily performed on the target layer. When using this approach it is able to make a more high coverage test, which is very useful for making a more time efficient test. The hybrid approach can also utilize stubs, as seen in the top-down approach.

2.1.3 End to End Testing

Sitting at the top of the testing pyramid is End to End(E2E) testing. This is the most involved, and runtime heavy tests. An E2E test is supposed to simulate a user scenario from start to finish. Its goal is to follow a normal user path, and to confirm that there is no breakage along this path. E2E tests are also crucial to test if the multiple layers of the application is working together, such as the database, interfaces and its communication to other systems.

One of the difference between E2E and unit tests are that the unit test is functionally tested through the code. An example could be testing validation for sign in input. The unit tests check the code or function for said validation, while the end-to-end test will check it through button clicks, such as open the web page, click on the sign in button, fill in the credentials and log in. This is in other words how a user would interact with the code.

In terms of value, E2E tests can provide a lot of value. Because they seek to emulate a real user scenario. The drawback is that these tests take more time to run than unit and integration tests. This makes it so that the tests can not be run as often, when for example new commits are pushed to a pull request. They can however be useful tests to run before merging a new pull request or performing a new deployment.

2.2 Snapshot Testing

Snapshot testing is a type of test that takes a snapshot of the system, and saves it. When the test is then ran, it will compare the saved snapshot and a new current snapshot taken. It is mainly used to check for unexpected changes in the UI. One of snapshot testings benefits is that the amount of code lines will be significantly less. This is because we do not have to check for changes in each element individually. It is also easy to update your saved snapshot if your UI have changes made on purpose.

2.3 Usability and User Experience

Usability and user experience are two important terms within front-end programming. Usability is how simply a user can achieve their goal on the application, while user experience is the user's overall experience using the application. When finding new front-end design features to implement, this always had to be in the back of our mind.

2.4 Mocking

Mocking is objects made that simulates the actions of a real object. This is frequently used in testing to save time with having to import impractical objects to the test, and to avoid large test units. In testing, one can use mocks for for example the state since feeding the real state object into the tests can be complex.

While mocking is a very useful technique in testing, it is not without complications. Mocking can make refactoring of the code harder. When code is refactored, added to or changed it can break the already made tests because the mocked script will not fulfill the real objects requirements anymore. Even though mocking can result in some complication, testing components that need actions from an object will often be too difficult without mocking.

2.5 Continuous Integration and Continuous Delivery of Tests

When working with a larger project, the continuous integration and continuous delivery (CI/CD) practice is useful to follow. With continuous integration developers integrate code into the repository frequently. Typically the code added to the shared repository is smaller, making it easier to detect and locate errors early. Some of the key principles of CI is; revision control, automated testing and build automation. With continuous delivery the developers are constantly deploying updates using automated tests. Here we used GitHub Actions, and had a workflow for our tests.

```
name: Jest Test

on:
  push:
    branches: [master]
  pull_request:

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-node@v1
        with:
          node-version: 12

      - name: Install modules
        working-directory: ./dev
        run: npm i

      - name: Run tests
        working-directory: ./dev/src/__tests__

        run: npm test -- --testPathIgnorePatterns="/e2e/|/testHelpers/"
```

Listing 2.1: jest.yml

Every time a pull request is created, the front-end tests will run on that pull request. This can give an indication if the new code gave an unexpected bug in the

frontend code. If the structure of the frontend code has changed or a feature has been changed, it can give an indication that we need to update the tests.

Chapter 3

Choosing Testing Frameworks and Libraries

The main goal of testing QuickFeed's frontend is to make sure that QuickFeed's presentation layer is working as it should, with no bugs or errors. One of the main reasons for a unsatisfied user, is a UI full of bugs and errors. With multiple frontend tests it would be easier to avoid these issues.
QuickFeed's frontend code is as mentioned written in TypeScript and uses the library React. We therefore have to find further frameworks and libraries that will work with both, and fulfill our goals.

3.1 Evaluating testing frameworks and libraries

There are several options when choosing what technology to use for testing. In addition to unit tests that tests the logic of the code, the web application also needs tests that could test the UI elements of the application.

3.1.1 Criteria when choosing a framework

Before choosing a framework we had to set some criteria:

- Framework popularity
  - The more used the framework are, the more information, libraries and assistance can be found online. That being the case, we wanted to use a well known and popular framework.

- Simple implementation with React
  - Choosing one of the many frameworks that is a simple setup with React would save us a lot of setup time, which instead could be used for development.
- E2E testing possibilities
  - A lot of testing frameworks is not able to run E2E-tests, so we have to find at least one that will allow us to run these tests.
- Scalability testing possibilities
  - A lot of testing framework is not able to make scalability testing possible. We also have to make sure to find a framework which could do this.

3.1.2 Rainforest

Rainforest is a software testing application used to easily perform tests on web apps or native apps without having to code. Their goal is to make achieving quality easier, and to deliver meaningful results. The tests are easily made in your browser with a click and drag premade setup as seen in the image below.

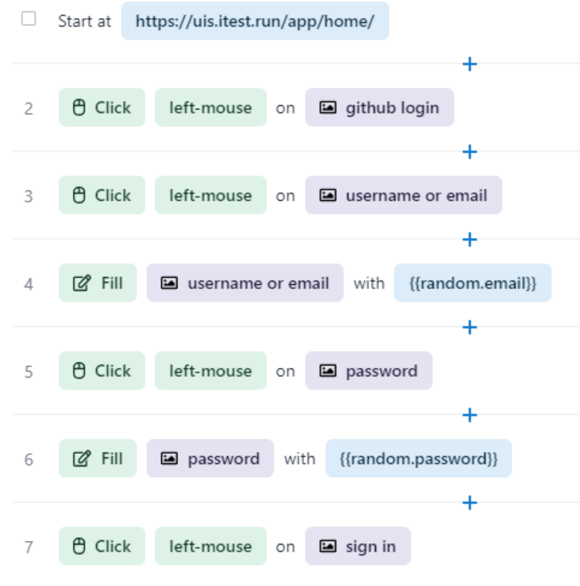


Figure 3.1: Rainforest coding setup

When the tests are made, it is possible to run the tests live on multiple modern browsers.

Rainforest is also unique in the way that it directly interact with the UI. Other testing frameworks (specifically Selenium based testing frameworks) usually only evaluate the DOM. This can cause issues where the tests will overlook problems which can be obvious for a user. If a test is made to check if a specific button is shown, it could pass even though it is hidden for the user. One example is if the login or logout button is hidden under a popup.

3.1.3 Jest

Jest is a JavaScript and Selenium based testing framework used to create, run and structure tests for the frontend. It is a framework which do not need a lot of additions. It is its own test runner, and have its own assertion and mocking library. Jest is currently one of the most popular testing frameworks, especially for React where it is the default testing framework. When using Jest it gives additional context for when tests fails and allows for easy mocking which can be helpful in

a testing development phase. As mentioned, Jest provides you with an assertion library. These libraries are tools to verify that things are correct, so you do not need to use a lot of if-statements. An example of using assertion libraries can be seen in listing 3.1.

```
describe("Testing assertion libraries", () => {
    it("Verify that true equals true", () => {
        x = True
        expect(x).toBe(true)
    }),

    it("Age is over 18 and under 30", () => {
       age = 20
       expect(age).toBeGreaterThanOrEqual(18)
       expect(age).toBeLessThanOrEqual(30)
    });
})
```

Listing 3.1: Assertion library examples

By default, Jests assertion library also gives feedback of what is received and what is expected to be received when a test fails. This can make for a more uncomplicated troubleshooting.

Jest is mostly useful for writing unit tests that tests the logic of the code. This is to make sure that for example a function works the way it should. Because Jest allows the tests to be written in isolation it will be easier to test smaller parts of the code instead of testing the whole application at once.

Jest Snapshot Testing

Jest also deliver a tool for snapshot testing. The snapshot testing tool can be used to prevent unexpected changes in the UI. It works by having a presaved HTML generated fragment, and then comparing them to the current UI. The test will fail if the two snapshot is mismatching.

3.1.4 Enzyme

Enzyme is a testing utility library that creates a simplified interface for writing unit tests, only for React applications. Packages like TestUtils, JSDOM and Chee-

rIO are wrapped by Enzyme. This makes a simplified unit testing interface and intuitive API for unit testing.

It is also important to note that Enzyme is not a test runner, like Jest. This means that it is dependent on Jest, or in other cases other test runners.

One of Enzymes additional feature for Jest, is mounting. Mounting is when react renders and then build the component for the first time. It also renders the full document object model(DOM). Enzymes mount feature can be used to mock a component mounting or unmounting. This is useful for components that interacts with the DOM API.

Shallow rendering is an another additional feature. This is very similar to mounting. The main difference is that shallow test components does not test the child component it renders. Shallow is very useful for small unit tests because of not having to worry about changes in the child components.

Render is the last rendering feature for Enzyme. This renders to static HTML, including child components. This does not have as many functionalities as mounting and shallow, but in return it is less costly.

Enzymes extra features like mounting and traversing component trees will make testing the UI with the Jest testing framework a lot easier.

3.1.5 Mocha

Mocha is an open source test framework originally designed for Node.js used to create and run unit, integration, and end-to-end tests. It is not a independent testing framework. This means that it requires a lot of libraries to be integrated to work correctly. Mocha is mostly suitable for test developers with some testing experience because the developer itself have to specify and choose their own assertion or mocking library. However, it is not strict on what libraries are chosen which makes it a very easy optimized tool if the developer have the knowledge to do so. It also can result in having tests that execute faster.

At the first glance, Mocha looks very simular to Jest in their setup and syntax.

```
describe("Compraing to Jest", () => {
    it("Age is over 18 and under 30", () => {
        x = 20
        expect(x).to.be.lte(30).and.to.be.gte(18)
    });
})
```

Listing 3.2: Mocha setup and syntax

When making this test with Jest, one does not need to add any libraries to have a working test environment. In Mocha we would need to manually add an assertion library, like for example Chai. When Mocha is supported by Chai it would be able to make a shorter and more clean looking test than Jest would, even though they deliver the same results.

3.1.6 Selenium

Selenium is a test automation framework made for web applications. It includes Selenium IDE (Integrated Development Environment), Selenium Grid, and Selenium WebDriver. Selenium IDE is a plugin on the browser, used to easily record and run Selenium tests. It provides the developer with a useful GUI for recording your interactions with the website.

Selenium Grid is a proxy server that helps the developer to do testing parallel on multiple machines and/or browsers. Its two main components are the hub and the nodes. The Hub is the central point of the grid. It is a server which routes JSON test commands to the nodes. The nodes can be found multiple times in a Selenium Grid. Each node is responsible for managing the different browsers, and to execute the commands the hub requests.

The last main feature is the Selenium WebDriver. This is a browser based driver which allows automated cross-browser testing. When executing a test, the test commands are converted into HTTP requests by a mediator. Each browser have their own driver to initialize the server. The browser will then receive requests through its driver, which will launch and navigate the chosen browsers where the tests are executed.

3.2 Choosing a testing framework

After evaluating a lot of different testing frameworks, we had to make a decision of which ones to use. At the first glance Rainforest seemed promising for solving this task. The absence of writing code could speed up the process of testing. The main selling point of Rainforest is its simplicity and ease of use. However it seemed too simplistic for the task. Another drawback of Rainforest is that it needed to be ran on a hosted website. This would require configuring access and security credentials. The application could be deployed with a dummy database, so that we would not have to test with real user information. Given the time it would take to set up, we decided against Rainforest. Additionally, Rainforest is not a free testing tool.

Mocha and Jest were both quite similar. If choosing Mocha, we would have a lot of freedom with choosing our own libraries to suit our tasks. But in regard, we had to familiarize ourselves with all of the possible libraries to use. This would be a challenge because we did not yet have a exact goal of what to test, and what types of tests we were going to produce.

Because QuickFeed needed tests both for the logic and the UI of the application Jest seemed like a good option. Jest with both its assertion tests and snapshot tests was promising for this purpose. After looking further into Jests snapshot tests, Enzyme was a recurrent recommendation. We saw that a lot of developers sworn by combining Jest and Enzyme to make the best testing tool. Unlike Mocha, Jest provides an integrated framework, which was tempting because of our lack of previous experience with configuration. Jest is also the default testing tool for React applications.

We also wanted to look into cross browser testing and testing of QuickFeed's web responsiveness. Jest had the possibility to do this by using the *getBoundingClientRect()*, but it required a lot of work with exact DOM loading. Therefore, we decided to take a look at Selenium. It is often used in responsive web design testing because of its tools which allows testing on different browsers. Also, it does not require any additional downloads to get element coordinates which can be used for making tests to detect overlapping elements on different browsers and screen

sizes. All of this combined, made us choose to use **Jest**, **Enzyme**, and **Selenium** for our frontend tests.

Chapter 4

Implementation

In this chapter we will be focusing on the implementations of the tests, front-end features and design improvements, while explaining our thought process and solutions.

4.1 Introduction

After looking into to various testing frameworks we decided upon writing tests with a combination of Jest, Enzyme, and Selenium, to improve the frontends robustness. This is because both Jest, Enzyme, and Selenium gives us the testing tools we need, and it can easily be implemented into an existing project.

When focusing on frontend code improvements, we realized that we had to get to know the current frontend code. To do this, we manually went through QuickFeed's features and looked at GitHub issues to find things to further improve or implement.

4.1.1 Requirements

Before starting the implementation of the tests and new front-end features, we mapped out some requirements we wanted to meet. They were mostly based off the project description, but we also found more along the way as we got to know the QuickFeed front-end more.

Overall, our main focus when starting the implementation was:

- Write tests for the front-end code
  - Unit tests
  - Snapshot tests
  - Integration tests
  - E2E tests
- Implement various missing front-end features and design improvements
- Improve the usability of the front-end

4.2 Structure

When implementing new changes on a collaborative project, structure is especially important. We tried to keep the structure as simple as possible and to follow good practise to make it easier for a possibly further development by others.

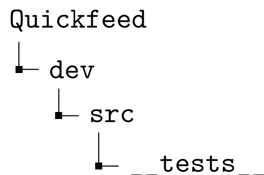
4.2.1 Choosing Operating System

We chose to use Windows when assisting on further development and making tests in QuickFeed. Windows had not been widely used in the development beforehand, which resulted in some problems when first starting out. We had difficulties running **Make** commands in windows. This lead to difficulties in building the **Docker** file. A common issue when developing in windows, is that the file system of windows is different from the file systems of macOS and Linux. This can make functions which depend on paths break on windows. We then thought of the Windows Subsystem for Linux (WSL). This would allow us to run a Linux file system, along with Linux command-line tools which would solve the problems we met. However due to some problems with WSL on our local machines which was too time consuming to solve, we decided upon using Windows.

4.2.2 Setting up Jest and Enzyme

Configuring Jest and Enzyme into a existing React app is a relatively simple task. It requires that you install both Jest and Enzyme locally.

When setting up the testing environment in QuickFeed's code we wanted to follow good practise, and use Jests recommended folder structure. While it is not necessary to follow this structure for the tests to run, it is seen as good practice. When researching Jests folder structure, we found two main structures that repeated themselves: **src/file.test.js** and **src/__tests__/file.test.js**. The first structures positives is that it is close to source files, and it is also whats done in Jests get starting documents. Nevertheless, we chose to use **src/__tests__/file.test.js**. This was simply because when making multiple test.js files, it would look too messy to keep them all directly in **src**. At the same time, we are also keeping the tests close to the source files as done in the Jest starting documents. In the figure below is an illustration of the file structure.

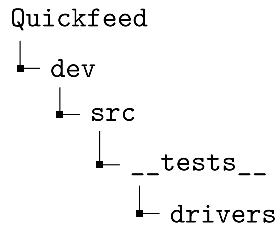


4.2.3 Setting up Selenium

We used *Jest* and *Enzyme's* assertion methods in conjunction with *Selenium* tests. Setting up Selenium takes a few steps. More info about this can be found in appendix A.

For Selenium to work, one needs to install a web driver for a specific browser. Selenium's web driver supports many of the major browsers such as; Firefox, Internet explorer, Edge, Opera, Safari and Chrome. We mostly used Firefox as our browser in development, and therefore we downloaded the web driver for Firefox. In order to run selenium tests, both the web driver and the QucickFeed server, needs to be active.

While Jest had it's recommend folder structure, we could not find a recommend structure for where to place the web driver. To make it easier for our self to locate, we created a folder in the **__tests__** folder. The idea was that this folder would house drivers for different browser, as we assumed not all people working on QuickFeed's development would be using Firefox.



4.3 Testing

In this section we will present our implementations of our different QuickFeed frontend tests.

4.3.1 Unit testing

When first starting out with making tests, we made unit tests. We purposely produced the unit tests first because making different tests for individual units, will later make the more complex test development easier. Having premade unit tests will speed up the process of for example integration and end to end testing. Also, because of us not being experienced test developers, starting with simple unit tests would make for a good learning curve.

One of the first test we produced was the test shown in figure 4.1.

```
1     it("User should be valid", () => {
2         const user = new User().setId(1).setName("Test User").setEmail
    ("mail@mail.com").setStudentid("1234567")
3         const isValidUser = isValid(user)
4         expect(isValidUser).toBe(true)
5     })
```

Listing 4.1: isValid.test.tsx

The purpose of this Jest test is to check if the isValid helper in figure 4.2 is working as it should.

```
1 export const isValid = (elm: User | EnrollmentLink): boolean => {
2     if (elm instanceof User) {
3         return elm.getName().length > 0 && elm.getEmail().length > 0 &&
       elm.getStudentid().length > 0
4     }
5     if (elm instanceof EnrollmentLink) {
6         return elm.getEnrollment()?.getUser() !== undefined && elm.
      getSubmissionsList().length > 0
7     }
8     return true
9 }
```

Listing 4.2: isValid helper

As seen in the isValid **helper**, it should have a name, email and student ID with length longer than zero. In the isValid **test** we therefore mock a user with values that should be valid if the helper is working as it should. Therefore, if a code change unpurposely ruins the helper function, this will be alerted when the test is ran.

All of the unit tests we made are just as simple as this. Unit test does not have a *set* size or degree of difficulty, but they are supposed to be as short and simple as possible. Another example which proves this, is the unit test in listing 4.3.

```
1 describe("Correct permission status should be set", () => {
2     it("If user is not admin, promote to admin", () => {
3         const user = new User().setId(1).setName("Test User").
      setIsadmin(false)
4         const mockedOvermind = createOvermindMock(config, (state) => {
5             state.self = user
6         })
7         window.confirm = jest.fn(() => true)
8         updateAdmin(mockedOvermind, user)
9         expect(user.getIsadmin()).toBe(true)
10    })
```

Listing 4.3: updateAdmin.test.tsx

This unit test checks the functionality of the updateAdmin function.Here we also had to create a overmind(a frictionless state management (15)) mock because updateAdmin requires state as an input. UpdateAdmin collects information about the user from the state. The updateAdmin function 4.4, sets the status to a user to

admin if the user does not have admin privileges. If the user has admin privileges, a prompt will appear to ask if you want to demote the user.

```
1 export const updateAdmin = async ({ state, effects }: Context, user:
    User): Promise<void> => {
2     // Confirm that user really wants to change admin status
3     if (confirm(`Are you sure you want to ${user.getIsadmin() ? "demote
    " : "promote"} ${user.getName()}?`)) {
4         // Copy user object and change admin status
5         const u = json(user)
6         u.setIsadmin(!user.getIsadmin())
7 
8         // Send updated user to server
9         const result = await effects.grpcMan.updateUser(u)
10        if (result.status.getCode() == 0) {
11            // If successful, update user in state with new admin
    status
12            const found = state.allUsers.findIndex(s => s.getId() ==
    user.getId())
13            if (found) {
14                state.allUsers[found] = u
15            }
16        }
17    }
18 }
```

Listing 4.4: actions.tsx

Another test case covers if the user is admin, and is being demoted, as seen in 4.5.

```
1     it("If user is admin, demote user", () => {
2         const user2 = new User().setId(2).setName("Test User2").
    setIsadmin(true)
3         const mockedOvermind2 = createOvermindMock(config, (state) => {
4             state.self = user2
5         })
6         window.confirm = jest.fn(() => true)
7         updateAdmin(mockedOvermind2, user2)
8         expect(user2.getIsadmin()).toBe(false)
9     })
```

Listing 4.5: updateAdmin.test.tsx

4.3.2 Integration testing

Sometimes you can not logically isolate the code or feature that is tested in the system. QuickFeed's frontend is written in React and consists of various components, and often require rendering of both the parent and the child component.

Some of the first integration tests we made was status checks in **UpdateEnrollement.test.tsx**. The first test can be seen in listing 4.6.

```
1    it("If status is teacher, button should display demote", () => {
2        const user = new User().setId(1).setName("Test User").
     setStudentid("6583969706").setEmail("test@gmail.com")
3        const enrollment = new Enrollment().setId(2).setCourseid(1).
     setStatus(3).setUser(user).setSlipdaysremaining(3).
     setLastactivitydate("10 Mar").setTotalapproved(0)
4
5        const mockedOvermind = createOvermindMock(config, (state) => {
6            state.self = user
7            state.activeCourse = 1
8            state.courseEnrollments = { [1]: [enrollment] }
9        })
10       const history = createMemoryHistory()
11       history.push("/course/1/members")
12
13       React.useState = jest.fn().mockReturnValue("True")
14       const wrapped = render(
15           <Provider value={mockedOvermind}>
16               <Router history={history} >
17                   <Route path="/course/:id/members" component={
     Members} />
18               </Router>
19           </Provider>
20       )
21       expect(wrapped.find("i").first().text()).toEqual("Demote")
22   })
```

Listing 4.6: Profile.test.tsx

This test is an integration test which tests a user based scenario. It needs a course, user, enrollment and browser history. As mentioned earlier, integration tests, tests how units of code works together. In this test we are testing how the component *Members*, which is a child component of *TeacherPage* works together with

the state. In order to do this we create a mock of the state as seen in listing 4.7. We then add a logged in user, because this page is only visible to users. Further on, we add an enrollment to the user. The status is also set to 3.

```
1        const user = new User().setId(1).setName("Test User").
     setStudentid("6583969706").setEmail("test@gmail.com")
2        const enrollment = new Enrollment().setId(2).setCourseid(1).
     setStatus(3).setUser(user).set
```

Listing 4.7: Profile.test.tsx

Setting the status to 3, means that the user is a teacher as seen in listing 4.8.

```
1    export enum UserStatus {
2        NONE = 0,
3        PENDING = 1,
4        STUDENT = 2,
5        TEACHER = 3,
6      }
```

Listing 4.8: ag_pb.ts

A teacher status will allow the user to access this page and view the various buttons for demote, promote, accept and reject. It also needs the router history in order to render the members within a given course. As seen in figure 4.9. The logged in user, active course and the mocked course nerollment is added to the mocked state. We also create a router history, so that the test will start at the members page.

```
1        const mockedOvermind = createOvermindMock(config, (state) => {
2            state.self = user
3            state.activeCourse = 1
4            state.courseEnrollments = { [1]: [enrollment] }
5        })
6        const history = createMemoryHistory()
7        history.push("/course/1/members")
```

Listing 4.9: Profile.test.tsx

Members component gets information about which course it needs to render for by the route parameters, as seen in the function *getCourseID*, 4.10. This is why we need to mock the router path.

```
export const getCourseID = (): number => {
    const route = useParams<{ id?: string }>()
    return Number(route.id)
}
```

Listing 4.10: Helpers.ts

All these factors combined is what makes the test a integration test, combining various units such as the state, different components, and testing these combined units as one.

4.3.3 End to End testing

The last types of test we produced were the end to end (E2E) tests. We decided to produce the E2E tests last because they are at the top of the testing pyramid, and should always be executed last. Because end to end tests requires different technology than the earlier integration and unit tests, we also had to implement a new framework, **Selenium**. Selenium offered the option to open browsers for navigation with clicks etc. Nevertheless, we still needed Jest to support Selenium with Jests assertion options, structure and test execution.

Our first E2E test is going to simulate and test the scenario of a user unfavoriting a course. The user path for this task can be seen in figure 4.1.

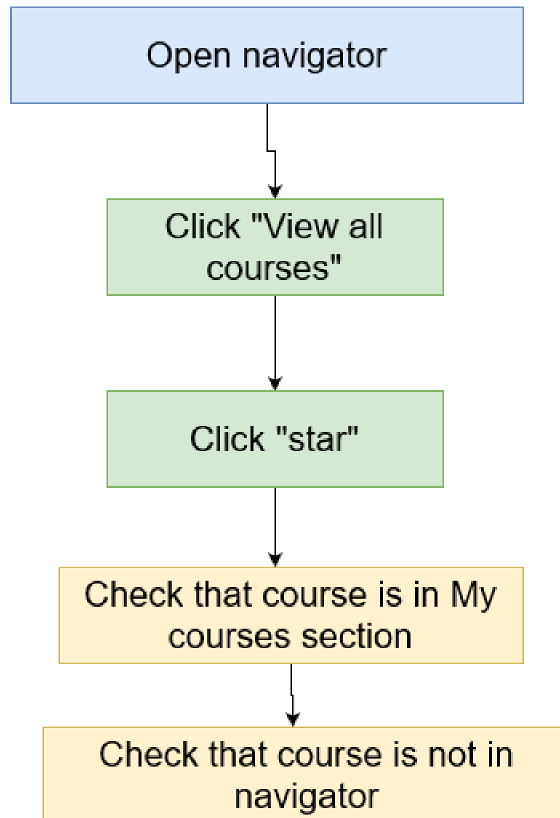


Figure 4.1: E2E flowchart for the test

One of Seleniums drawbacks in this case, is that the test can not be ran automatically on a pull request as mentioned in section *Continuous Integration and Continuous Delivery of Tests*. It has to be manually run.

The get a grasp on what the test is testing, let's go through the user scenario:

The user is logged in, and is on the landing/home page. One way to navigate to the courses page, is through the vertical navigation bar. Firstly they will open the vertical navigation bar by clicking on the hamburger button.

Figure 4.2: Opening the vertical navigation bar

To simulate this action with code we do as seen in listing 4.11.

```
1        const hamburger = await driver.findElement(By.className("
    hamburger"))
2        await driver.wait(until.elementIsVisible(hamburger), 100)
3        hamburger.click()
4        await driver.sleep(1000)
```

Listing 4.11: E2ECourseVisability.test.tsx

These lines of code tells the web driver to locate the hamburger menu by it's class name **hamburger**. When the element is visible, the Selenium web driver will click the hamburger button. When doing this we also need to add a sleep function. Without the sleep functions the DOM will not load properly before moving on to the next steps. This will cause the test to not find the elements it is supposed to find, which will return a test error.

After the vertical navigator bar is opened, the user will navigate to the course page as seen in figure 4.3.

Figure 4.3: Navigate to course page

To do this, we give almost identical instructions to the Selenium web driver as in listing 4.11. It is firstly finding the clickable element, and then clicking it before making the driver sleep to wait for the DOM load.

```
        const goToCourse = await driver.findElement(By.css(".courseLink"))
        await driver.wait(until.elementIsVisible(goToCourse), 100)
        goToCourse.click()
        await driver.sleep(1000)
```

Listing 4.12: E2ECourseVisability.test.tsx

The last piece of user input needed, is for the user to remove the course from favorites. This is done by clicking the yellow star in the top right corner of the course card as seen in figure 4.4.

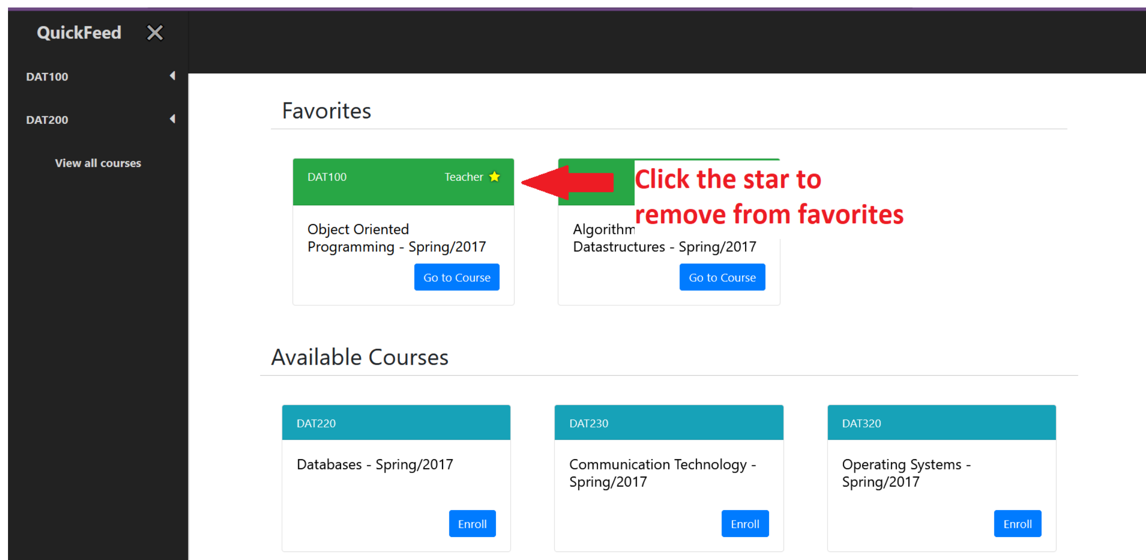


Figure 4.4: Remove course from favorites

The instructions given to selenium web driver are identical to clicking the hamburger bar and navigating to the course page. Locate the element, click it and set a second break.

Lastly, we want to create some assertions for the test. The conditions that needs to be met for the test to pass is as follows;

- The course card should no longer be present in the favorites section.

- The unfavorited subject should not appear in the vertical navigation bar anymore.

In order to check these assertions in listing 4.13, we collect the names of the courses in both favorites and the navigation bar, before removing the course from favorites.

```
1        const card = await driver.findElement(By.css(".card"))
2        const cardCourseCode = await card.findElement(By.css(".card-
    header")).getText()
3        const courseCode = cardCourseCode.split("\n")[0]
4        await driver.sleep(1000)
5        const hamburgerCode = await driver.findElement(By.css("#title")
    ).getText()
```

Listing 4.13: E2ECourseVisability.test.tsx

After course had been removed from favorites, the test will check that the course is no longer in the favorites section. In addition, the test will also check that the course is no longer in the navigation bar.

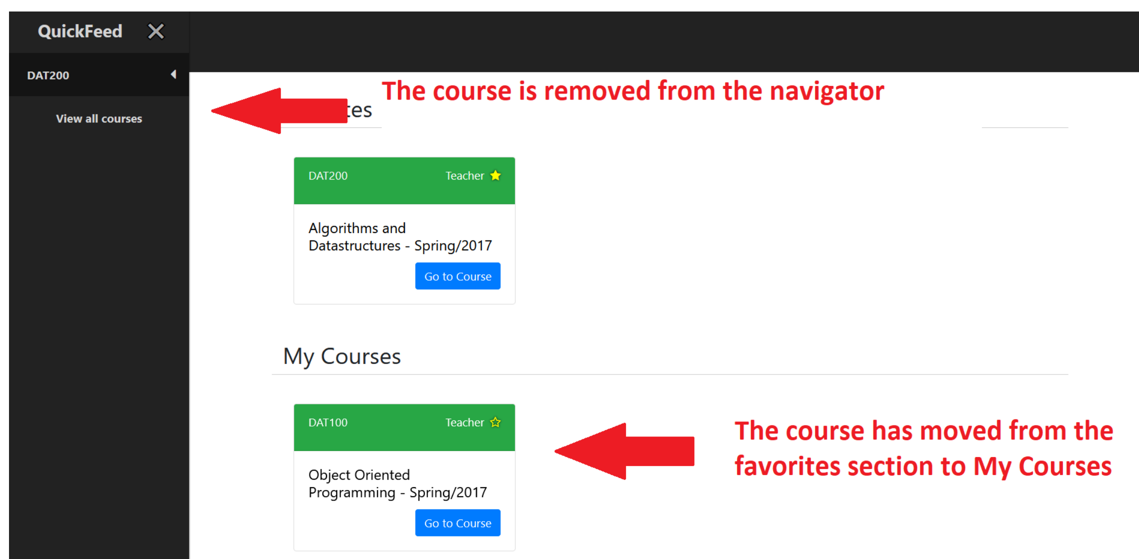


Figure 4.5: Course has been removed from favorites

To check this assertion, in listing 4.14, the test will firstly check if there is a course with the same course code as the unfavorited course in *My Courses* section.

```
1        await driver.sleep(1000)
2        const myCourse = await driver.findElement(By.css(".myCourses"))
3        const myCoursesCard = await myCourse.findElement(By.css(".card-
    header")).getText()
4        const hasMoved = (courseCode === myCoursesCard.split("\n")[0])
```

Listing 4.14: E2ECourseVisability.test.tsx

The next step is to check that the course is no longer present in the navigation bar. To do this, we collect a list of the courses in the navigation bar. Then we'll loop through it to see if a course has a matching course code to the unfavorited course. This can be seen in listing 4.15.

```
1        //Find coursecodes in navigator
2        await driver.sleep(1000)
3        const navigatorCourses = await driver.findElement(By.css(".
    navigator"))
4        await driver.sleep(1000)
5        const courses = await navigatorCourses.findElements(By.css("#
    title"))
6
7        //Check if the course has moved from navigator
8        var isInList = true
9        for (let i = 0; i < courses.length; i++) {
10           if (await courses[i].getText() === hamburgerCode) {
11               isInList = false
```

Listing 4.15: E2ECourseVisability.test.tsx

Lastly, if both **isInList** and **hasMoved** are true, a new boolean value **movedFromFavorites** will be true. This means that the test has passed, and the course has been successfully moved from favorites. Note that this only applies to the frontend code and does not apply to the backend code.

```
1        const movedFromFavorites = (isInList && hasMoved)
2        await driver.close()
3        expect(movedFromFavorites).toBe((true))
```

Listing 4.16: E2ECourseVisability.test.tsx

4.3.4 Snapshot testing

Snapshot testing, also called visual regression testing, does not focus on the code, which makes it an unique test compared to others. Making snapshot tests to en-

sure that the view difference between a logged in user and a visitor is correct, is a smart way to make code line amount efficient tests. We therefore decided to make tests that could give errors if the navigation bar UI does not react correctly to the state changes. When first starting the implementation, we made a simple test that can be seen in listing 4.17 below.

```
1 describe("Visibility when logged in", () => {
2     it("When user is logged in, hamburger menu should appear", () => {
3         const hamburger =  ""
4         expect(wrapped.find(".clickable").text()).toEqual(hamburger)
5     })
```

Listing 4.17: Navbar.test.tsx

This test checks that the hamburger menu in the navigation bar only is visible to users that are logged in. Before the test runs we make a snapshot of the *Navbar* component. This will automatically happen if it does not already have an existing snapshot. It is also possible to manually take a new snapshot if you have changes in your UI you want to cooperate into your tests.

```
1 const history = createMemoryHistory()
2 const mockedOvermind = createOvermindMock(config, (state) => {
3     state.self = new User().setId(1).setName("Test User")
4 })
5 const wrapped = mount(<Provider value={mockedOvermind}>
6     <Router history={history}>
7         <NavBar />
8     </Router>
9 </Provider>
10 )
```

Listing 4.18: Navbar.test.tsx

In listing 4.18 we are adding a mocked state with a logged in user, **Test User**, to the component. This is to ensure that we get a snapshot of the navigation bar in a logged in user view.

4.3.5 Responsive web design tests

Bugs in the functionally of the application, such as validation and various functions can cause a lot of problems. Equally can scalability issues in the UI be infuriating for the user. We created some tests, that tests the UI scalability of QuickFeed, primarily using Selenium.

We first wanted to check if the login and logo of QuickFeed ever would overlap.

Figure 4.6: Sign in button and Logo should not overlap in the navbar

Listing 4.19 is the start of the overlapping check test where the web driver is built. This code need to be modified, depending on the type of browser used.The test runs for various screen resolutions, such as; 960x1080, 1366x768 , 960x1080 and 683x786. They are respectively typical screen resolutions for; desktop,laptop, spilt screen desktop and split screen laptop.

```
1    overlapTests.forEach(test => {
2        it(`Should not overlap on res ${test.width}x${test.height}`,
    async () => {
3            //This test requires you to set
4            const firefox = require('selenium-webdriver/firefox')
5            const service = new firefox.ServiceBuilder('drivers\\
    geckodriver.exe')
6            const driver = new Builder().forBrowser('firefox').
    setFirefoxService(service).withCapabilities(Capabilities.firefox()
7                .set("acceptInsecureCerts", true)).build()
8            await driver.get("https://127.0.0.1/dev")
9            await driver.manage().window().setRect({ width: test.width,
     height: test.height })
```

Listing 4.19: navbarResonsive.test.tsx

Further on, in 4.20 we continued the test. This test is supposed to check if the sign in button and the QuickFeed logo will overlap on different screen resolutions. We decided to use Seleniums **getRect()** function which will give us our chosen elements coordinates. The position is listed as; x, y, height and width values. These coordinates could further on be used to check for unwanted overlaps.

```
        let rect: IRectangle
        let rect2: IRectangle

        const logo = await driver.findElement(By.className("navbar-
    brand"))
        const signIn = await driver.findElement(By.className("signIn"))
        rect = await logo.getRect()
        rect2 = await signIn.getRect()
    })
```

Listing 4.20: navbarResonsive.test.tsx

We made a function that returns a boolean to check for overlaps. This function can be seen in listing 4.21.

```
export const isOverlapping = (rect: IRectangle, rect2: IRectangle) => {
    var overlap = (rect.x < rect2.x + rect2.width &&
        rect.x + rect.width > rect2.x &&
        rect.y < rect2.y + rect2.height &&
        rect.height + rect.y > rect2.y)
    if (overlap) {
        return true //Elements are overlapping
    }
    return false //Elements are not overlapping
}
```

Listing 4.21: testHelpers.ts

After various attempts at writing code that could detect overlapping, we started to look into how it was solved in video games. The code above is called **Axis-Aligned bounding box** (14), which is a simple collision detection algorithm often used in 2D games. The code checks that there is no gap between any of the 4 sides of the rectangles. If there is no gap, the code returns **false**, and there is no collision, if it returns **true** there is a collision, meaning that the elements overlap.

Lastly the test would check the output from **isOverlapping**. As seen in listing 4.22, the test passes if the boolean value from overlap matches test.want. Test.want is in this case *false*, meaning the sign in button and QuickFeed logo should not overlap.

```
            let rect = await logo.getRect()
            let rect2 = await signIn.getRect()

            var overlap = isOverlapping(rect, rect2)

            jest.setTimeout(50000)
            expect(overlap).toBe(test.want)
```

Listing 4.22: navbarResponsive.tsx

4.4 Design

QuickFeed did recently have a full web redesign. The original horizontal navigation bar was switched out with a vertical one, and they added new teacher and student pages. We thought the design looked professional, clean and it gave QuickFeed a good first impression for the user. For that reason, when trying to improve the frontend design and layout, we were foremost looking for small user experience and usability tweaks.

4.4.1 Usability and User Experience

Further on in this section we will describe our process in improving QuickFeed's usability, and to improve users user experience.

Vertical Navigation Bar

Our first idea when looking for usability tweaks was making the vertical navigation bar optional for the user. We did this by adding the hamburger button to open, and the cross to close it. The hamburger button is often used to toggle the navigation bar between displaying or being collapsed on the screen. It is of simple design and its functionality is well known by users all over the world.

Figure 4.7: New design horizontal navigation bar hamburger button

The old navigation bar had a nice design and solution for displaying the various courses users are enrolled in. We did not want to completely scrap this feature. For the sake of usability we wanted to make this feature optional and added it to the new vertical navigation bar.

We moved the code for the old navigation bar to a new component. We did some minor tweaks to the code, such as removing the login/user button but kept it mostly the same. The component is a child component of *NavBar*. The horizontal navigation bar will only be visible if the logged in user has clicked the hamburger button, as seen in listing 4.23.

```
1 {state.isLoggedIn &&
2     <ul className="mr-auto me-auto list-unstyled">
3         <a className="clickable" onClick={() => { onCourseClick() }}
4             style={{ paddingTop: "15px",
5             marginLeft: "10px", fontSize: 25  }}></a>
6     </ul>
7 }
```

Listing 4.23: NavBar.tsx

The code in listing 4.23 is from *NavBar.tsx*. When the user clicks on the hamburger menu symbol, a function will be called. The function will set a boolean value in the state to *true*. If this value is true, the vertical navigation bare will be visible along with the horizontal navigation bar. This can be seen in listing 4.24.

```
1 <div>
2     {state.showFavorites &&
3         <NavFavorites />
4     }
5 </div>
6 }
```

Listing 4.24: NavBar.tsx

Horizontal Navigation Bar

We also added a static horizontal navigation bar. We re-implemented this to have a easier way to access the logout and about button. The old re-design had their button in the bottom of the vertical navigation bar as seen in figure 4.8.

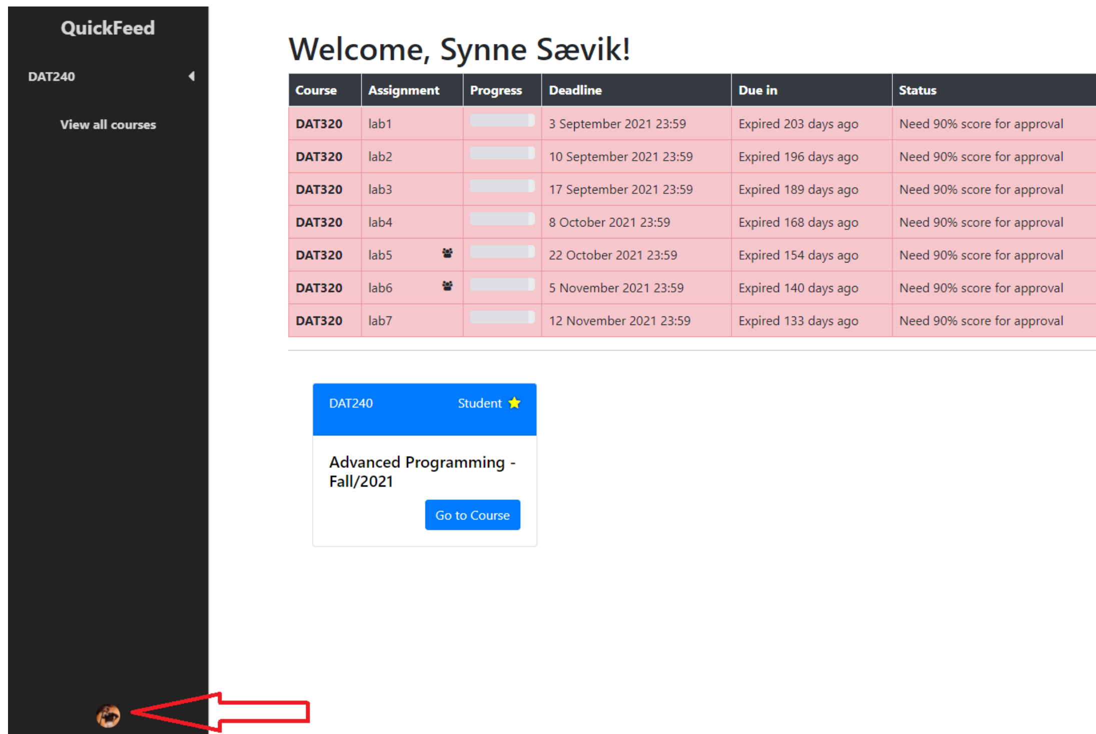


Figure 4.8: Old design vertical navigation bar

We found this button too small, and hard to find. This would especially be the case with a new vertical navigation bar, with the open and close function.

Therefore we moved this button on the static horizontal navigation bar, and increased its size. We were strict at trying to follow common placement patterns to make a good user friendly design. While not necessarily a creative choice, placing it to the right on a horizontal navigation bar will be recognizable to most users as it is the standard placement in todays websites (7).

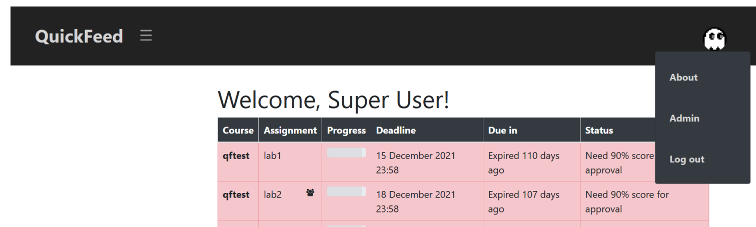


Figure 4.9: Horizontal navbar with user button

We made a completely new react component for the horizontal navigation bar. The design part was made with bootstrap and CSS. The navigation bar component *Navbar* will always be mounted on the page (listing 4.25), just as the old horizontal navigation bar was.

```
1  return (
2      <div>
3          <NavBar />
4          <div className="app wrapper">
5              <div id="content">
6                  {Main()}
7              </div>
8          </div>
9      </div>
10 )
```

Listing 4.25: App.tsx

Responsive Web Design

Responsive web design (RWD) is important to make the user experience good for all users at difference devices. After manually testing QuickFeed's RWD we found multiple problems. There were a couple of scalability issues in the application. Many of them arose when scaling the browser window to a smaller size. Most of these issues could be fixed by some tweaking in the css file or by changing/adding bootstrap. Css's *@media* tag works well for changing placement, layout and position in different screen sizes.

```
@media only screen and (max-width: 980px) {
  .width-resize {
    width: 350px;
    padding-top: 15px;
  }
}
```

Listing 4.26: RWD media tag

In the code in listing 4.26, the **@media** tag changes the width and padding of course utility links when the screen size is less or equal to 980px. In figure 4.10 the screen size is more than 980px wide. When scaling down to under 980px the old QuickFeed would have an overlap of the shown element.

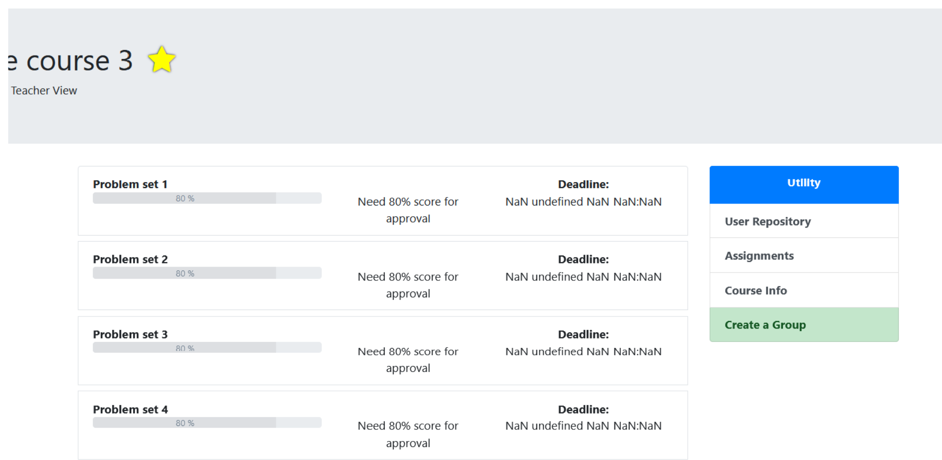


Figure 4.10: Screen size is more than 980px

By using the textbf@media tag we managed to move the utility element below the assignment element, as seen in figure 4.11.

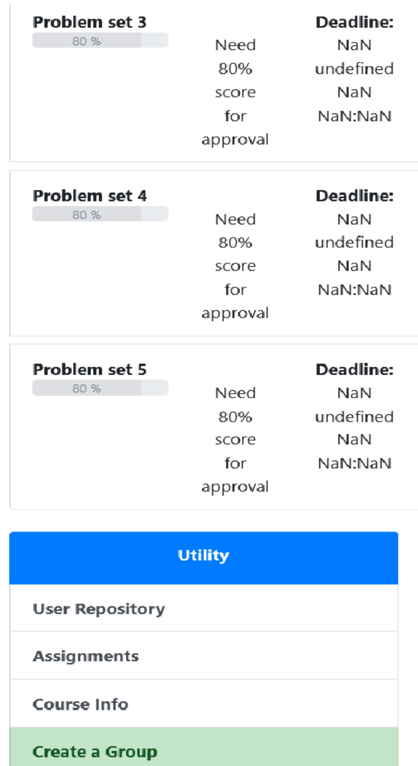


Figure 4.11: Screensize is less than 980px

To give the users good user experience, we needed to consider how users might use QuickFeed. Due to the nature of QuickFeed being an automated student feedback application, one would assume it would mainly be used on desktop or laptop computers. However the screen size may vary widely among these devices, users may also use QuickFeed in minimized browser windows. Therefore we decided to make changes to the fronted to accommodate these needs. We did not focus much on modifying the frontend's scalabilty to work on mobile devices. We made this decision because changing the design to work for mobile devices can be quite time consuming. Having QuickFeed adjusted to fit mobile screens would be a nice feature, but not a very pressing one.

Chapter 5

Discussion

In this chapter we will describe, analyze and interpret our findings and solutions.

5.1 Results

Getting to work on QuickFeed on our last school semester has made us learn a lot. Firstly, we had to learn to work with React and TypeScript. We learned a lot about how React state management works, and how to mock this state after our wants and needs. We also learned a lot about frontend testing, and why it is important for a failproof application. The understanding of how you should build an application upon your tests, instead of building tests upon your application is also a important method learned for future development.

When first starting our work on QuickFeed we had to use a lot of time reading about frontend testing since none of us had ever made frontend tests before. This gave us a basis of understandings for what we were going to develop. After some weeks of reading, we had to start working in our coding environment. We used some time getting QuickFeed setup with our Windows machines, but in the process we learned a lot about how bigger projects setups are employed.

In the end, we managed to meet our main goals for this thesis. We implemented a testing environment inside QuickFeed while following set testing environment standards. With this, we made different types of tests such as unit tests, snapshot tests, end to end tests and integration tests. Additionally, we also did some

frontend features and design improvements, such as changing the navigation bar.

5.2 Choosing what to test

Before beginning on our thesis, we were quite inexperienced with testing. Only having made simple unit tests for various assignments and projects. In the beginning of the thesis we wrote tests, just for the sake of writing tests, in order to learn fronted testing. It became apparent early on that some tests are more valuable than others.

Learning how to write tests is only half of the battle when making frontend tests. We also have to know *what* to test. A good thought to have in mind while starting out is: ”What would I be most upset about if it broke on this app?” (9).

One of the main points to follow when choosing what to test is ”Test use cases, not code” (9). One should never focus on the number of lines covered by tests, but how much of the use cases are covered. Use cases are descriptions on how a user will use the application, measured in steps. Therefore, one should make a list of important features, and then look for these features use cases to choose the most important cases to test.

5.3 Limitations of testing

Tests are incredibly useful and valuable for an application. But nothing is without it's downsides. While frontend tests provide valuable feedback to the code, there are some limitations.

The automated tests we made for QuickFeed are all written in code. Just like any other code, tests can have bugs in them. They can be poorly designed, resulting in tests passing, when in reality this is a false pass. When a test has a bug in it, it can sometimes be harder to catch. The test can end up hiding that there is a bug in the code. While developing our tests we always ran the test on incidents which were supposed to pass, and incidents which were supposed to fail. By using this method we would get a more credible test.

Most of the tests covers code that we did not write ourselves. There is an argument to be made that the developer who made the code is the best equipped to test it. This is because they know how the code is intended to work. However developers who did not write the code, might be able to easier spot bugs and therefore make an unbiased test. When we worked on our tests, we had a lot of communication with one of the frontend developers, Jostein Lindhom, to make sure we fully understood the frontend code intentions.

Tests much like any other code, can become outdated as more code and functionalities is added to the application. Smaller tests such as unit tests are relatively quick to rewrite. Given that they only test code in isolation, they might be able to survive more additions to the code base without becoming outdated. For example, a function that tests the validation of a username, won't necessarily become affected by refactoring or a change in design. A **E2E** frontend test is highly dependent on the UI, layout of the page, other components and units. A change in one these factors can render the test useless.

Tests much like any other code requires maintenance. When code is refactored or a new features is added, it is sometimes required to go back and change old code. This is going to cost in terms of time, or else they fall into obscurity. Tests are important, and the time it takes to maintain them can be a valuable investment. Therefore, when making our frontend tests made sure to make the code as understandable as possible, so further maintenance by others would be a simple task.

5.4 Technical credibility

When creating our tests we needed an Enzyme adapter which would work with React 17.0. Since React never released an official adapter for React 17.0, we found an unofficial one; @wojtekmaj/enzyme-adapter-react-17 (13). Installing and importing unofficial packages into a project could cause security risks. Therefore, we had to do some research on this package before importing it.

When starting looking into package security, we found the website snyk.io (19).

Snyk.io is a security platform which reviews your own code, and in this context open source dependencies. We decided to rely on this website, because of its all over good reviews and usage among other developers online. When looking at @wojtekmaj/enzyme-adapter-react-17 on this website, we firstly saw its popularity. Lately, it have had a total of over 600 000 weekly downloads weekly. Seeing that a lot of other developers also uses this package, while there is not much of dissatisfaction or warnings online is a good sign. This package also have constant maintenance, which is important to improve and retain its security.

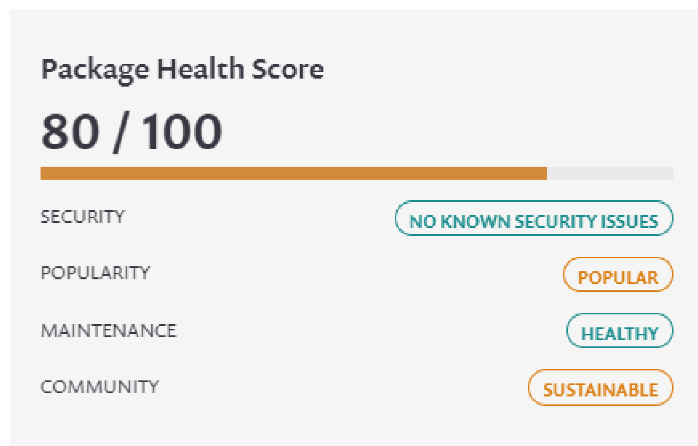


Figure 5.1: Screenshot from snyk.io about @wojtekmaj/enzyme-adapter-react-17

This package got an overall good security score as seen in figure 5.1 (19), and had a good reputation online. In the end, we considered this package as not being a security risk for QuickFeed, and decided to use it.

5.5 Future work

In this section we will cover what we would recommend for future work on QuickFeed testing and design.

5.5.1 Tests

A lot of time was spend setting up tests and learning how different styles of testing works. Through this we made a system for testing that should be easier to pick up for other developers.

There is a lot of the code base that still could benefit from being covered by tests. It is room for more unit tests covering frontend functions, and integration tests that test the **UI**. **E2E** tests were the last implemented tests and there were quite few made. Future work could cover important user paths. Some examples include:

- Admin users creating and changing course
- Accepting and rejecting enrollments
- Approve, revise and rejecting submissions
- Students creating groups

We currently only have an E2E test for a user scenario, so making E2E tests for the remaining roles could be an idea for future work.

5.5.2 GrpcManager mock

Running some of the selenium tests requires setting a mock **GrpcManager** as the default **GrpcManager**. More information about how to run these tests can be found in appendix A. This can be frustrating and time consuming. These tests were some of the last code implemented and therefore we ran out of time to address this issue.

For future work making sure this process is automated would make testing a lot easier. Having a method of automatically setting **GrpcManager** to a mocked manager when running certain tests, would fix this issue.

5.5.3 Design

One feature we would have liked to have implemented was for the application to scale for mobile users.

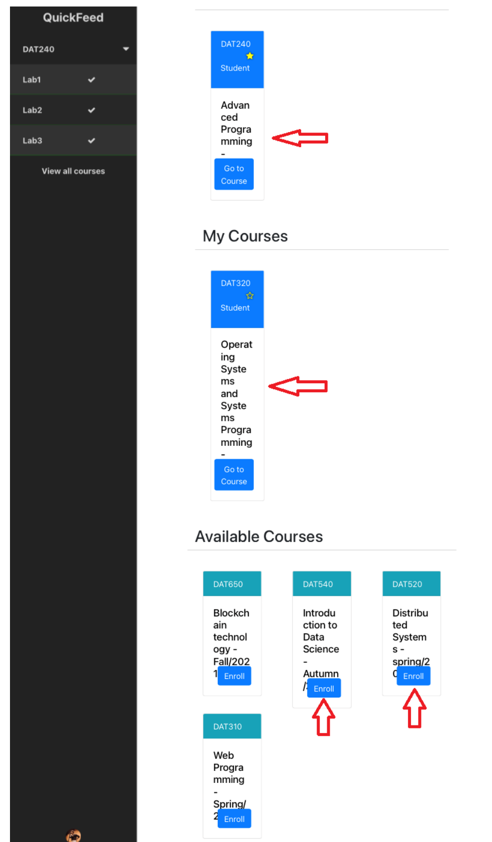


Figure 5.2: Mobile device UI issues

As seen in figure 5.2 QuickFeed have some UI and scaling issues on a mobile platform. Fixing these issues requires some time. QuickFeed's UI is made from a combination of Bootstrap and CSS. Both of these needs to be adjusted to fit this requirement. This was a feature we decided not to pursue. It can be a nice quality of life, but not necessary. QuickFeed was originally designed for laptop/desktop use. Because most students will use QuickFeed to retrieve approvals, feedback etc. on code they've written on a laptop/desktop, it is safe to assume they would access QuickFeed trough a computer. The same goes for teachers, they will most likely use a computer when interacting with QuickFeed.

5.5.4 Test Driven Development (TDD)

Test Driven Development(**TDD**) is a style of programming, in which test cases are being written before the software/web application is developed. Tests are designed and written for every small feature or functionality of the application.

TDD is sometimes refereed to as Test-First Programming(**TF**). **TF** is the practice of writing tests before the code is written. Instead of writing tests after the code is written, which is a common practice. Usually one writes a test, often a unit test before implementing a feature. The test should fail because the code for the feature has not been written. Afterwards, one write the code so that the test passes. Lastly, one refactors the code. When the refactoring is done, the tests are ran again to ensure that the functionally is still there. This cycle is done for every new piece of functionality.

The benefits of **TDD** are cutting down on time spent debugging, by having tests that covers most of the code. It is focusing on making test cases, which allows developers to imagine how the functionally may work from a user perceptive. After writing a test following the **TDD** method, a developer is supposed to write as little and simple code as possible for the test to pass.

There are some downsides to **TDD**. The tests written are often made by the same developer that wrote the production code. This can lead to the developer losing track of what they are testing. They might overlook some parameters that needs to be tested and checked, leading to a test passing, which in turn can lead to a false sense of correctness. Writing and designing a lot of test cases can be time costly. It is recommend that **TDD** is adopted by all the team members. While not necessarily a downside, in our case it is. **TDD** has not earlier been adopted by the QuickFeed's development team.

We do not use **TDD** as a form of software development. Firstly QuickFeed's code had been written and developed long before we started testing the code and adjusting the front-end code. We used the more traditional style of programming, where one writes tests after the code has been written. Most of the tests written were for already implemented code. There is an argument to be made for that we

could write test cases before writing the production code. We deemed this style of programming to be a little excessive, for our purposes. Regardless, if Quick-Feed is ever going to continue developing its frontend features and/or UI, trying to follow the practice of **TDD** would be wise because of all its mentioned benefits.

Chapter 6

Conclusions

Our main goals for this thesis was to make frontend tests, improve the frontend code, and assist with the design on QuickFeed. Firstly, the lack of frontend test made the further development of QuickFeed's state and frontend harder because of not having any analysing tests which could warn about new errors. Our main goal was to fix this. Secondly, since QuickFeed's frontend was recently updated, there were still design and usability improvements that could be implemented. This was not our main goal of this thesis, but something we wanted to improve while working on the frontend.

When implementing the frontend tests we realised QuickFeed could benefit from different types of tests, such as unit, integration and end-to-end tests. Jest, Enzyme and Selenium was used for this task. Unit tests were beneficial for QuickFeed because it is testing smaller part of the code, such as functions and components. Since these single components and functions are widely used all over the application, we implemented multiple of these unit tests. Integration tests were also beneficial since QuickFeed have a lot of components that function together. Lastly, end-to-end tests were also very valuable since QuickFeed have a lot of common user paths. Running tests along these commons paths to check for breakages which can happen when new features are added into the frontend, is beneficial to make sure no important features breaks.

Because Jest has its own test runner, this was used to get all the tests output which will clearly displays test errors. While also making the tests apart of a continu-

ous integration system with a GitHub WorkFlow, failing tests will also display on GitHub pull requests when pushing new changes. Selenium tests were the only exception, given that they need the QuickFeed server to be running locally.

While implementing tests we also looked at the frontend code, design and usability. We made some changes to the design, mostly in the form of the navigation bar. Additionally we made minor adjustments to the design, to allow for better scalability. We also changed the front end code, when other QuickFeed developers posted frontend code issues on GitHub, or when we found a bug ourselves.

We made a lot of different tests, but there could still be developed more tests to cover more of the frontend. For future testing, it will be easier to continue the process because the testing environment is already implemented.

Bibliography

[1] Barnum C. M. (2020) *Usability Testing Essentials: Ready, Set ...Test!*. Morgan Kaufmann.

[2] Bekkhus, S. (2022, May 6). *Getting Started*. JestJS. https://jestjs.io/docs/getting-started

[3] Borys (2021, April 28). Mocha vs. Jest: comparison of two testing tools for Node.js. *Merixstudio*. https://www.merixstudio.com/blog/mocha-vs-jest/

[4] Bose S.(2020, Jan 21). Testing Pyramid : How to jumpstart Test Automation. *BrowserStack*. https://www.browserstack.com/guide/testing-pyramid-for-test-automation.

[5] Cohn M. (2009) *Succeeding with Agile*. Addison-Wesley Educational Publishers Inc.

[6] Costa D.,Fernandes L. (2021) *Testing JavaScript Applications*. Simon and Schuster.

[7] Crestodina A.(2021, Nov). *Web Design Standards vs. Website Best Practices: Our Review of 500 Sites [NEW RESEARCH]*. https://www.orbitmedia.com/blog/web-design-standards/

[8] Digital.ai. *Test driven development*. https://digital.ai/glossary/test-driven-development

[9] Dodds K.C. (2019, April 13). *How to know what to test*. https://kentcdodds.com/blog/how-to-know-what-to-test

[10] Enzymejs.github.io. *Using enzyme with Jest*. https://enzymejs.github.io/enzyme/docs/guides/jest.html#using-enzyme-with-jest

[11] Fowler M.(2014, Feb 26). The Practical Test Pyramid. *martinFowler*. https://martinfowler.com/articles/practical-test-pyramid.html

[12] Madeyski L.(2010).*Test-Driven Development: An Empirical Evaluation of Agile Practice*. Springer Berlin Heidelberg.

[13] Maj W. (2021, Dec 9) *@wojtekmaj/enzyme-adapter-react-17*, https://github.com/wojtekmaj/enzyme-adapter-react-17.

[14] Mdm web docs. *2D collision detectiont*. https://developer.mozilla.org/en-US/docs/Games/Techniques/2D_collision_detection

[15] Overmind. (2021, July) *Overmind, frictionless state management*. https://overmindjs.org/

[16] Quinlivan R.(2019, Nov 27). The Limitations Of Automated Testing And What To Do About It. *Medium*. https://medium.com/swlh/the-limitations-of-automated-testing-and-what-to-do-about-it-c74a34884254

[17] Rainforest. *How Rainforest QA works*. https://www.rainforestqa.com/how-rainforest-works

[18] Selenium (2022, March 16). *The Selenium Browser Automation Project*. https://www.selenium.dev/documentation/

[19] Snyk.io (2022, May 10) *@wojtekmaj/enzyme-adapter-react-17 v0.6.7*, https://snyk.io/advisor/npm-package/@wojtekmaj/enzyme-adapter-react-17

[20] Software Testing Help. (2022, May 5). *Key to Successful Unit Testing – How Developers Test Their Own Code?* https://www.softwaretestinghelp.com/unit-testing/

[21] Software Testing Help. (2022, May 5). *The Differences Between Unit Testing, Integration Testing and Functional Testing*. https://www.softwaretestinghelp.com/the-difference-between-unit-integration-and-functional-testing/

[22] Wattearachchi W. (2019, Jan 14). Testing with Jest and Enzyme in React — Part 4 (shallow vs. mount in Enzyme). *Medium*. https://wasuradananjith.medium.com/testing-with-jest-and-enzyme-in-react-part-4-shallow-vs-mount-in-enzyme-d60cad73f85c

Appendix A

Instructions to Compile and Run System

A.1 How to run the tests

Before trying to run the tests, both Jest and Selenium needs to be set up. Otherwise, tests will return errors.

A.1.1 Setup the test environment - Jest

To run the frontend tests in dev/src/__tests__/, make sure to install the required packages:

```
cd dev
npm ci
```

To run jest from the command line, you will need to install it globally:

```
npm i --global jest
```

For more information on running jest from the command line, please see the getting started documentation (2) or section A.1.3.

A.1.2 Setup the test environment - Selenium

To run Selenium for the command line, you need to install it globally:

```
npm i --global selenium-webdriver
```

For more information on running Selenium from the command line, please see the documentation (18) or section A.1.3.

To install the webdriver:

Go to: `https://www.selenium.dev/documentation/webdriver/getting_started/install_drivers/`

Install the webdriver for your browser. The documentation lists several ways of using the drivers.

A.1.3 Run the jest tests

To run all tests:

```
cd dev/src/__tests__
jest --testPathIgnorePatterns="e2e|testHelpers"
```

To run a specific test:

```
jest <test-filename>
```

A.1.4 how to run Selenium tests

Before running the tests:

1. Boot up QuickFeed locally.
2. Start the geckodriver.
3. run from dev/src/__tests__/

How to run **navbarResponsive**:

```
jest navbarResponsive
```

Note that two of these Selenium tests needs a mocked *grpcManager*.

To run **navbarResponsive** and **E2ECourseVisability**:

1. Go to the **effects.ts** in your IDE, the file can be found in dev/src/overmind/.

2. Comment out **export const grpcMan = new GrpcManager()** and uncomment **export const grpcMan = new MockGrpcManager()**

```
1       import { GrpcManager } from "../GRPCManager"
2       import { MockGrpcManager } from "../MockGRPCManager"
3
4       // Effects should contain all impure functions used to
    manage state.
5       // export const grpcMan = new GrpcManager()
6       export const grpcMan = new MockGrpcManager()
7
```

3. Run the test from your command line

4. To run the **navbarResponsive** test:

   (a) `jest navbarResponsive`

5. To run the **E2ECourseVisability** test:

   (a) `jest E2ECourseVisability`

Appendix B

Attachments

Thesis proposal: `https://github.com/relab/thesis-proposals/blob/master/2022/quickfeed-ui-testing.md`

Further, this appendix contains our pull requests with the code we have written for this thesis:

- `https://github.com/quickfeed/quickfeed/pull/638`
  - A bug fix for members page where the accept button did not change after accepting a student. This pull request includes a fix for this bug and tests for the members page.
- `https://github.com/quickfeed/quickfeed/pull/640`
  - Selenium tests. Introduces selenium to the code base. Includes E2E test and scalability tests.
- `https://github.com/quickfeed/quickfeed/pull/626`
  - Fixing scaling issues, and editing the navigation bar.
- `https://github.com/quickfeed/quickfeed/pull/613`
  - First unit tests in Jest and Enzyme.

University
of Stavanger

4036 Stavanger
Tel: +47 51 83 10 00
E-mail: post@uis.no
www.uis.no

Cover Photo: Hein Meling

© 2022 Marie Grøtte Larsen, Synne Marie Sævik