VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Mai Tran

TESTING REACT APPLICATIONS USING REACT TESTING LIBRARY

School of Technology
2023

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Mai Tran
Title	Testing React Applications Using React Testing Library
Year	2023
Language	English
Pages	80 + 1 Appendix
Name of Supervisor	Anna-Kaisa Saari

Testing has become an essential step in software development. Testing verifies that the application will work as expected and avoids unexpected bugs that might occur when some components change. With React applications, the principle is the same. Developers need to write tests that best replicate the user's interactions with the web application, to find possible defects that will crash the whole application. As the creator of React Testing Library, Kent C.Dodds, stated, "The more your tests resemble how your software is used, the more confidence they can give you."

This bachelor's thesis aims to study the testing library recommended by the official React team for React applications – React Testing Library: why it should be used and what are the best practices. Related concepts were studied as part of the purpose to help understand software testing in React thoroughly. The benefits and drawbacks of the library were analyzed by comparing React Testing Library with Enzyme Testing Library. The Enzyme library has been the most common testing library used for React applications. However, with the new version of React, ReactJS 18, the Enzyme library is no longer supported. For the practical part of the research, an application was built for demonstration purposes. Finally, the best practices for the testing library were reviewed based on development and research experience.

The observations and results from this thesis will help developers understand unit testing in React applications and how to use React Testing Library for it. By understanding the testing library thoroughly, they will be able to structure tests and define test cases better, improving the application quality and reducing the time needed to check for errors and bugs.

Keywords React, React Testing Library, software testing, unit testing, and Jest

TABLE OF CONTENTS

LIST OF FIGURES AND LISTINGS

LIST OF APPENDICES

LIST OF ABBREVIATIONS

GUI	Graphical User Interface
AAA	Arrange, Act, Assert
SUT	System Under Test
MUT	Method Under Test
E2E	End-to-End
UAT	User Acceptance Testing
SPA	Single Page Applications
UI	User Interface
JSX	JavaScript XML
DOM	Document Object Model
API	Application Programming Interface
ES	EcmaScript
RTL	React Testing Library
IDE	Integrated Development Environment

1 INTRODUCTION

With the growth of technologies in recent years, software applications are widely used in many companies serving different purposes. Some companies build software to enable creators to learn and make artistic products, while others use software applications to advertise and sell products. Regardless of which purpose an application serves, building powerful and high-quality software requires time and effort. Before building a software product, the team first needs to analyze its requirements, plan and design the product, then implement and test the code to prepare for deployment. Having well-planned tests is crucial in making sure the software has no unexpected bugs and it satisfies user needs before publishing.

Similar to other types of software products, a web application should also be tested and monitored for its performance and quality before delivery. Nowadays, Single Page Application (SPA) has risen in popularity because of its performance and responsiveness. There are different ways to test single-page applications. One popular method is to focus on testing the user behaviour on the web page, rather than focusing on the implementation details of the page. Focusing on tests that closely resemble how users interact with the web page is also the principle of the DOM testing library, a lightweight solution that provides different APIs for SPA frameworks like React, Angular and Vue.

This thesis focuses on the testing library recommended for React applications by the official React team, React Testing Library, built on top of the DOM testing library. Another popular testing library, Enzyme, will be compared to understand the benefits and drawbacks of both libraries and why RTL should be used for React applications. The testing library has a new approach to testing that does not focus on the code, so to understand the principle, an example application will be tested and explained in detail. It is important to know several common practices when using the library to reduce the execution time and improve the quality of the tests, therefore, they will be discussed in the last chapter of this thesis.

2 SOFTWARE TESTING

Software testing is the process of evaluating and verifying that a software product works as expected without defects before delivering it to the end users. The purposes of testing include preventing bugs and errors, reducing development costs and improving performance [1]. Testing can be performed manually by a person, or automatically by a set of tools.

Manual testing is when the tests are executed by a tester, in many cases, the developer who is working on the feature, or component, to find critical bugs or issues [2]. Automation testing, on the other hand, is a technique that uses special software testing tools to execute test case suites [3]. Even though manual testing is the most primitive testing technique, it is imperative, as there will always exist areas in the application that would need the inspection and intervention of developers to avoid errors. In all cases, a new feature or application should be manually tested first before the tests can be automated and therefore, 100% automation is not possible. Automation testing will help a lot, however, if the component has no new feature and is only needed to be verified before the release of another feature. That would reduce the time cost if the software application has hundreds, or thousands, of components needed to be re-tested.

Software testing is usually classified into three categories: Functional testing, Non-functional testing and Maintenance testing. This study will only focus on functional testing, a type of software testing that validates the application against the functional requirements and specifications. The purpose of the tests is to test each function of the software application, by providing appropriate input and verifying the output against the functional requirements [4].

2.1 Levels of Testing

Test levels are usually defined and grouped by their functionality and specificity in the software development process. Each level of testing has to ensure that every component is tested and verified thoroughly. There are primarily four main levels of testing: Unit testing, Integration testing, System testing and Acceptance testing.

Unit testing is the first level in software testing. It is performed simultaneously with the development of a component to check if it is working properly, compared to the software specifications [5].

After the units are tested individually, the modules that are dependent on each other are grouped and tested for data flow and communication. This is called integration testing and it aims to expose faults in the interaction between integrated units [6].

System testing comes after integration testing. The tests validate the fully integrated software product against the specified requirements to check its compliance with end-to-end use cases [7].

Lastly, acceptance testing, often called alpha or beta testing, is conducted by a group of selected users or project managers. The main purpose of acceptance testing is to check for additional bugs that might have passed other testing levels and determine whether it meets the acceptance criteria and can be published [8].

The order of the four testing levels is not only a hierarchy that extends from simple to complex but also a sequence that spans the whole development process. By testing simple and isolated components earlier before moving on to more complex functionality, the developers can be more confident in the software performance. However, this does not mean that acceptance testing should only be carried out after the whole feature is completed. In agile working approaches, they can be done after each finished sprint, as part of a demo [9]. The lifecycle of the testing process is described in Figure 1.

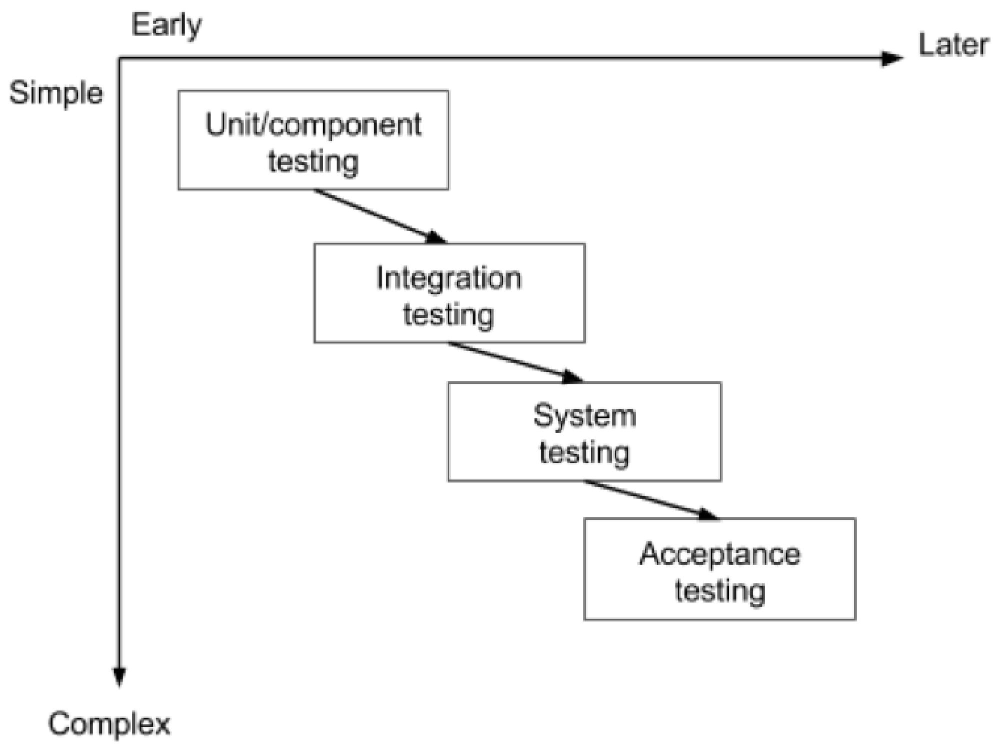


Figure 1. Software testing lifecycle.[9]

The time and number of test cases recommended for each level of testing are represented in the "Test Pyramid" primarily introduced by Mike Cohn in his book "Succeeding with Agile" [27]. The test pyramid is a simpler version of the testing levels, which consists of only three layers but still shares valuable insights on how the testers should establish test cases for each layer. Mike Cohn's test pyramid is shown in Figure 2 and the shape of the pyramid is to emphasize that the more high-level the test, the fewer tests should be written. In practice, this means there should be many small and fast unit tests, some more integration tests and very few system tests [10]. The granularity of the tests should also reflect the pyramid, with more paths covered on the unit testing level than on other test levels.

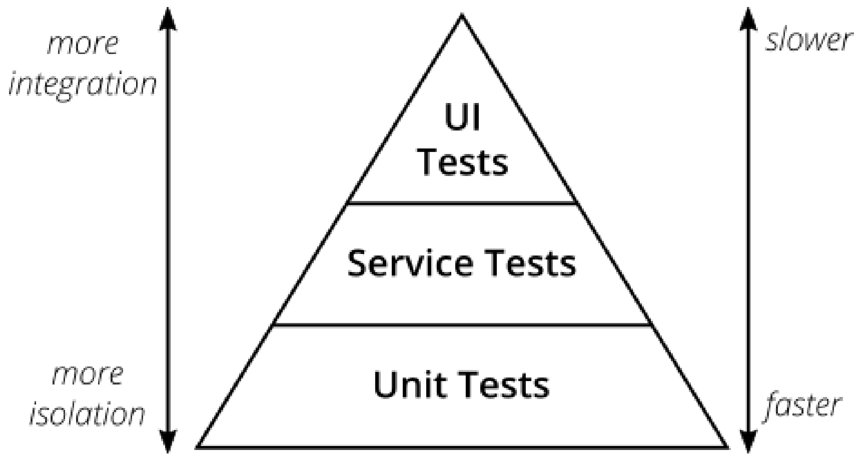


Figure 2. Test pyramid. [10]

2.2 Unit Testing

Unit testing, also called Module testing or Component testing, is the first and most basic level of testing. At this test level, the components are isolated into small sections and the tests are carried out individually without relying on other parts of the software. By testing them in their basic form, developers can find bugs in the early stages of development, allowing better fixes and reducing testing costs [5]. This test level applies the white box testing method, requiring access to the internal design and structure of the component. Because of this and the testing level’s primary goal to catch early bugs, the developers working on the module usually also perform the tests in the developer’s environment.

Writing unit tests for each component takes some time for the developers, but it is compensated by the benefits. Engineers will not need to run the application (in this case, the software GUI) and manually enter some inputs to trigger the desired test cases. Instead, testing can be automated by different sets of information to get the expected results for each test case. Furthermore, if the code contains defects and the test fails, only the latest changes need to be debugged. The result is less cost (time, money, effort, human power) in the overall development of the software. When testing at higher levels, changes made over a long period (days, weeks, months) will need to be scanned [11].

Finally, as the software grows and reaches hundreds of components, with the help of automated unit tests, revisiting old components while checking the new units is faster and more precise. Figure 3 shows the dynamic growth of a typical project without tests. A project without tests has a faster start but eventually, the work hours spent will grow drastically. Overall, if all components introduced also come with unit tests available for them, developers will feel more confident and be able to produce better code.

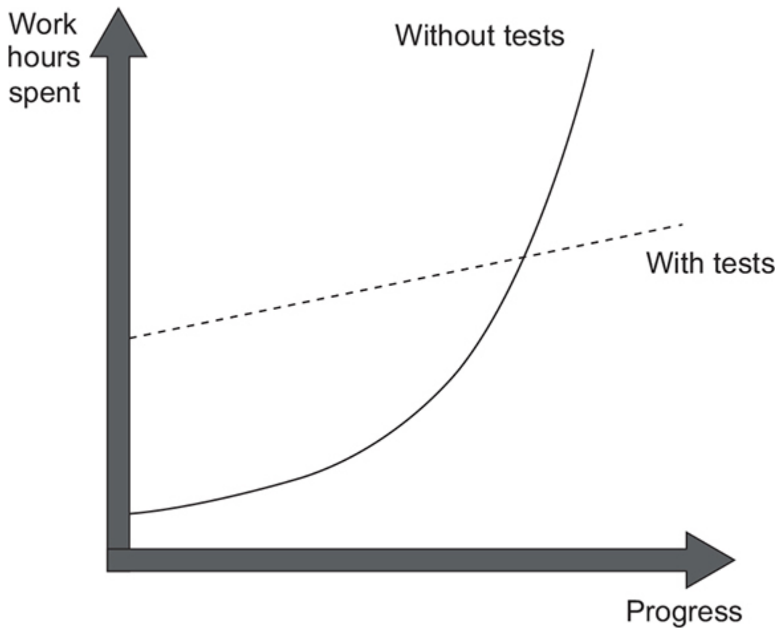


Figure 3. The difference in dynamic growth between projects with and without tests. [12]

2.2.1 Good Unit Tests

Having unit tests for each component will help detect bugs in the early stage of development. However, a test suite containing a large number of test cases is not necessarily a good thing. When writing a test, its value and maintenance cost need to be considered. As Khorikov pointed out in his book, a good unit test should always have the following four attributes as its foundation: protection against regressions, resistance to refactoring, fast feedback and maintainability [12]. On the contrary, bad tests do not help catch regression errors, are difficult to maintain and raise false alarms. All tests will help with the development process, but bad ones only reduce the work hours by a small amount and in a way, will give more work to developers as they have to rewrite the tests every time they refactor, or make changes to the code.

Regression is a software error that happens when a feature stops working as intended after a new functionality is introduced. Unfortunately, the chances of such cases happening increase with the growth of the code base. To prevent the occurrence of regressions happening, it is important to have good unit tests from the beginning. In order to evaluate how well a test will help protect against regressions, a developer should consider the amount of code executed in the test, the complexity of the code and its domain significance [12]. Codes with complex business logic are more important than boilerplate code. In addition, good unit tests should also check the libraries and frameworks used, because those tools also influence the outcome of the software. There are many reasons for a test to be considered bad, but the most common scenario is when a test gives false alarm or passes even if the code is wrong. For example, in Listing 1, there is a function to calculate the sum of two numbers, but the developer mistakenly typed `*` instead of `+`, making this a multiplication function. The test then proceeds to check for this function with test data `2, 2`. In this case, it is a bad test because the functionality of the function is wrong, but the test result is still `true`. The most straightforward way to avoid this is to have multiple sets of test data instead of just one.

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

describe('Sum of two numbers test', () => {
  it('gives the correct result', () => {
    expect(sum(2, 2)).toBe(4);
  });
});
```

Listing 1. A false positive test case.

2.2.2 Structuring unit tests using the AAA pattern

The Arrange-Act-Assert pattern has been a simple yet solid foundation for any unit tests. Following this uniform structure, developers can easily read and understand the tests, which results in reduced maintenance costs. Listing 2 gives an example of a function that calculates the sum of two numbers and a test block that asserts the result of the calculation. The test follows the AAA pattern.

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

describe('Sum of two numbers test', () => {
  it('gives the correct result', () => {
    // Arrange
    const a = 1;
    const b = 2;

    // Act
    const sum = sumOfTwoNumbers(a, b);

    // Assert
    expect(sum).toBe(3);
  });
});
```

Listing 2. Function and test for sum of two numbers.

Each section of the test does what the name implies. In the *arrange* section, the system under test (SUT) and the dependencies are brought to their desired states. Then in the *act* section, the methods on the SUT are called, prepared dependencies are passed and finally, the output is captured. In the *assert* section the output value is verified. The result passed to `expect()` can be the return value or the final state of the SUT and its collaborators. In a test case, the arrange section should be the largest, followed by the assert section, which can compare multiple related outcomes. Finally, the act section should be the smallest, often containing one to two lines of code [12].

In an ideal test block, developers should avoid multiple arrange, act and assert sections. The case is described in Figure 4. The primary reason for this is that, if it performs too many actions in a test case, it can be considered an integration test, rather than a unit test. This can be easily solved by extracting each of the act-assert sections into own test. Tester should also avoid if statements in tests, as a test should be straightforward without additional branches.

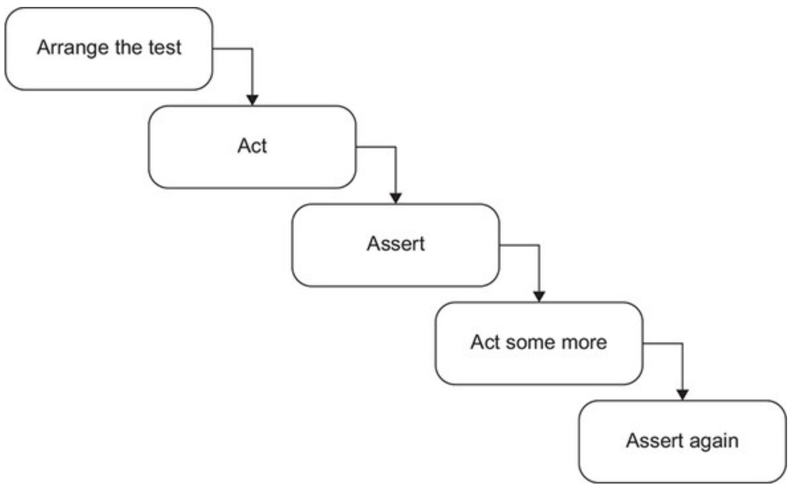


Figure 4. Multiple AAA sections in a test. [12]

2.2.3 Classical and London schools of unit testing

There are two common approaches to unit testing, the *Classical* approach, also known as *Detroit* and the *London* approach. Both approaches isolate the unit that is under test, but in different ways.

In the classical way, it is not the code that needs to be isolated, but the test cases themselves. Each test case should call its dependencies and collaborators in their own desired states. They are then run in parallel with each other, without affecting the other’s states and execution context. This means for example, that the test cases should not touch the shared database or the file system [12].

The other school of unit testing, called London, or *Mockists*, is formed from the use of test doubles in its tests. The London school describes the isolation of code in unit testing as isolating the SUT from its collaborators [12]. This means that all the dependent libraries or components should be replaced with dummies (this process is called mocking the components). A test dummy, however, should still contain the necessary methods needed to execute the test, even if it is only a fake definition of the functions.

Both approaches have their benefits and disadvantages and deciding which way to go depends on the context. The London approach benefits from isolating the dependent components, in which case, developers can focus on the component exclusively. There are, however, cases where the actual dependencies are required and they cannot be mocked, there the Classical approach is used.

2.3 Integration testing

Integration testing is a method for verifying that related components work correctly when combined. Integration tests are usually used to check the communication between services, how the services interact with the database and if the component gives the expected output. Because the integration tests go through a larger amount of code in order to execute the test, it is more costly to maintain integration tests than the unit tests. However, it is also the reason why it protects the code base better against regressions. One good example of integration testing is to check the communication between user inputs and the data server.

Because of the maintenance cost, a good common practice that developers follow is, "check as many of the business scenario's edge cases as possible with unit tests; use integration tests to cover one happy path, as well as any edge cases that cannot be covered by unit tests" [12]. A happy path here refers to a positive outcome of a business scenario, such as entering the proper username and password on the login page. An edge case is when the execution leads to an error, for example, a network error when a user tries to log in. To expand on this principle, the best practice is to select the longest happy path possible to verify all interactions with other dependencies. For edge cases, though, sometimes there is no need to cover everything because not all edge cases provide significant value.

It is important to test also the dependencies when executing integration tests. Khorikov categorizes the dependencies into two types in his book, managed and unmanaged dependencies. In other words, dependencies that one has control over versus dependencies that one does not. How to work with each type is described briefly in Figure 5 below. Managed dependencies are often implementation details, which means they are functions or components created by developers and used in the component under test. Such dependencies should be used as it is in the tests. On the contrary, communications with unmanaged dependencies are part of the system's observable behavior and they should be mocked [12].

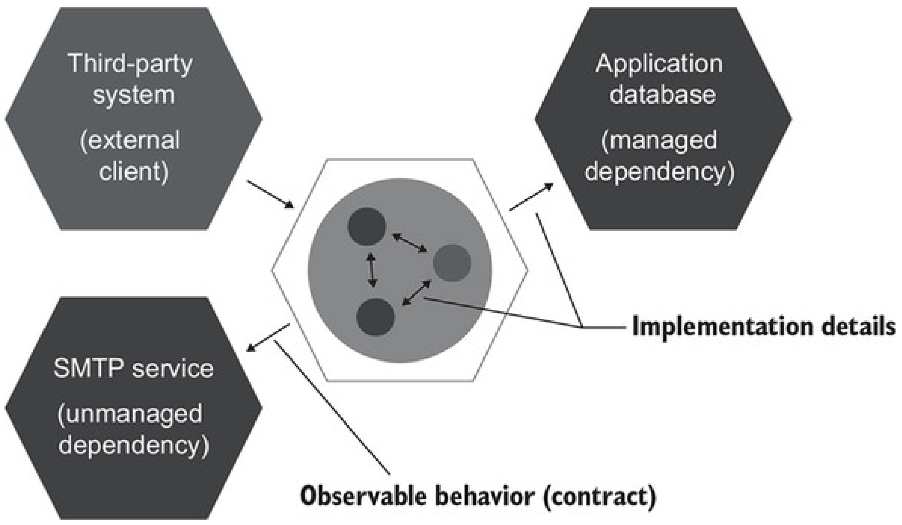


Figure 5. Dependencies in an integration test. [12]

2.4 System testing

A software product is usually only an element of a larger computer system. Therefore, to ensure that the released software will run correctly on different systems, testers need to validate the product before it is published. This level of testing is called system testing and is often performed by both testers and developers.

During the system testing process, testers are required to test the fully integrated application including external peripherals to check how components interact with one another and with the system as a whole [7]. This process is also called End-to-End testing (E2E). The expected outputs of the application should also be verified through different sets of input. Additionally, testers will need to check if the user experience meets the specified requirements.

System testing is performed in different ways, each focusing on a set of requirements that is important to the use of the product. Common types of system testing are: Usability testing, which focuses on the user’s ease to use the application; Regression testing, which includes tests to make sure changes to the software code do not introduce new bugs and old bugs do not re-appear; and Load testing, which ensures that the software will perform reliably in real-life use cases.

2.5 Acceptance testing

User acceptance testing (UAT) is performed manually by a selected group of end users, the client, or product managers to verify the software system against the acceptance criteria before moving it to the production environment [13]. Acceptance testing is the final phase of the testing cycle. Because of the wide range of tester groups, UAT is often categorized into two types, Internal and External.

Internal acceptance testing, or Alpha testing, is performed by the members of the owner organization. The members are usually the product managers, sales, customer support people, or the developers themselves [8]. This testing phase is to ensure the application has met its initial requirements before moving to the second type, called Beta testing.

Beta testing, or external testing, is carried out by the customers of the organization. The customers can be the client that commissioned the application, or a group of volunteer customers [8]. This phase is intended for users to help developers by trying out the software product to provide feedback and to catch possible bugs that might have passed the other testing levels.

3 REACTJS LIBRARY

React is a free and open-source front-end JavaScript library created by Facebook for building user interfaces based on components, released in 2013. Ever since React was released, its popularity has risen significantly. One of the reasons for its popularity is the efficient updates that happen when the data changes. Furthermore, the rise in popularity of Single Page Applications (SPA) also makes the library, which is already popular, become even more well-known. Figure 6 illustrates the number of downloads of ReactJS versus Angular and VueJS in 2022.

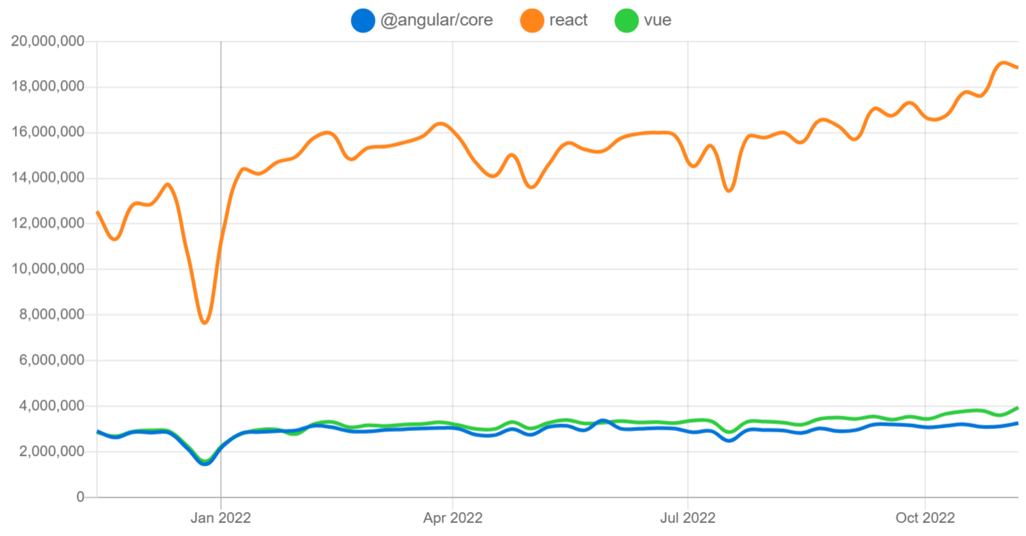


Figure 6. The popularity of React vs Angular vs Vue in 2022. [14]

React is a component-based library, which means it helps developers build and encapsulate components so that they can be reused as many times as needed to make beautiful and complex UIs. In other words, components refer to individual pieces of a broken-down UI. The components render according to the changes in states or props passed to them. Because of this declarative design, React is very fast, reliable and efficient and it allows programmers to create simple views for their web applications.

Being the most popular front-end library, React also has its benefits and drawbacks. The first advantage of using React is that it is easy to learn and use. Having many tutorials, a large community and good documentation, any developer with some knowledge of JavaScript can understand and create basic applications using React in a few days. There are also some handy tools such as React Developer Tools to make development with React better. It allows developers to inspect the hierarchies of the components or the individual piece and its current props and states. React components are also reusable, having their own logic and controllers. Reusability saves developers a lot of time not having to redefine a module many times. Because of this component-based design, React applications are relatively easy to test with the help of native tools. Lastly, React uses Virtual DOM (Document Object Model) to render components to the web page, leading to smoother and faster performance [15].

React also makes creating dynamic web applications easier with the introduction of JavaScript XML extension (JSX). JSX is a particular syntax allowing HTML quotes and HTML tag syntax to render subcomponents. However, this is also one of the drawbacks of the library. Because in JSX, HTML and JavaScript syntax are mixed, which means the view and logic part of the application are combined. If developers are not familiar with the syntax, it can make developing complicated applications confusing and takes a while to learn.

Other drawbacks of React library come from its popularity and continuous evolution. Being updated regularly means the creators are still actively maintaining the library, but it also means that developers using it will have to constantly learn new concepts and move from the old ones. It might be hard for some developers to adopt all the new changes and learn new ways to work with the library. Another disadvantage of being constantly updated is sometimes the documentation will not be changed in time to reflect the new changes. This requires programmers to follow their own instructions with the new releases or ask for help from the community [15]. Luckily, React has a very active community so this is not a huge drawback. An example of the popularity and the community is shown in Figure 7, which illustrates the percentage of Stack overflow questions for each framework.

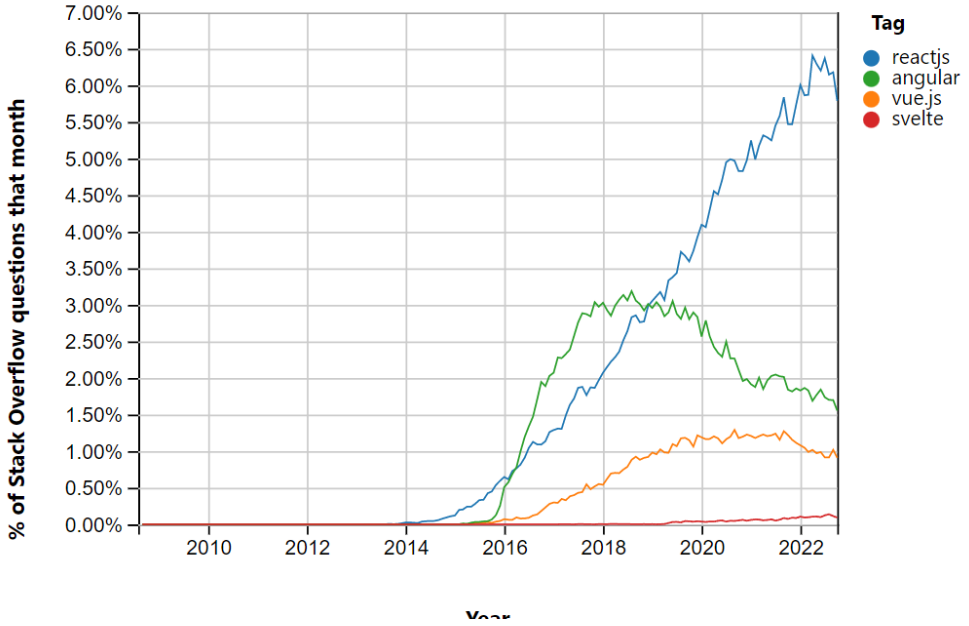


Figure 7. Stack Overflow trends. [16]

3.1 Virtual DOM

In a web application, a DOM (Document Object Model) is a tree data structure, where each element of the HTML web document is presented as a node [17]. Figure 8 shows a simple version of the DOM tree. The elements are accessible through a DOM API, where developers can change, manipulate and store different data. This is usually seen in the JavaScript code, where developers use `getElementById()`, `getElementByClass()` or other methods to modify the content of the DOM. However, as straightforward as it seems, every time a change is made, the browser has to go through all the tree nodes to find the target element. Furthermore, when mutating an element, all of its children are also affected. This proves to be costly and also reduces the speed of the web application. With React, this process is optimized using Virtual DOM.

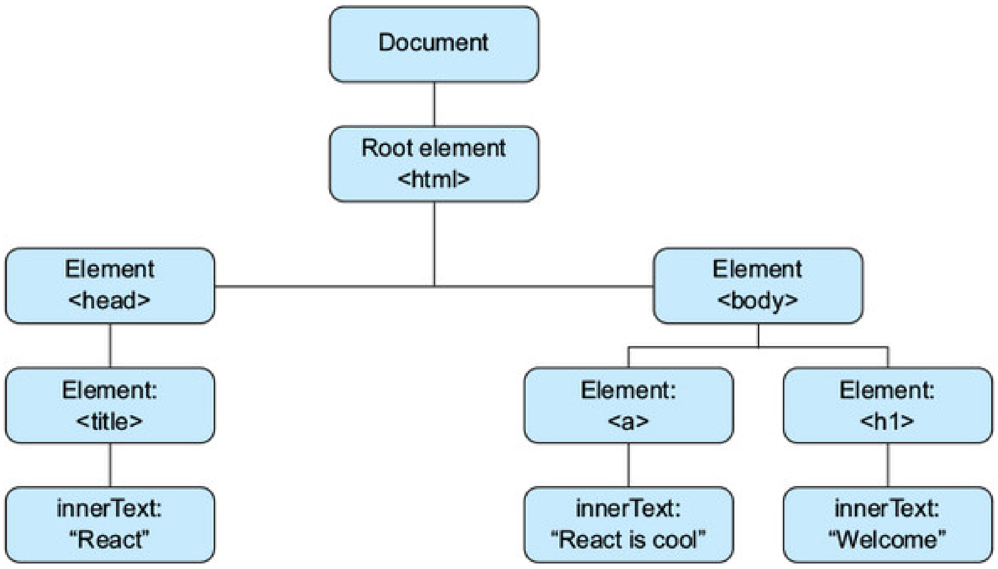


Figure 8. Simple DOM tree structure. [17]

The Virtual DOM in React is a representation of a UI maintained in memory, serving as an additional layer between the application code and the browser DOM [17]. In a React application, whenever anything new is added or some changes are made, a virtual DOM is created. React then compares the new virtual DOM with the most recent one created before the changes are made. This process is called *diffing*. After knowing exactly what needs to be changed, React performs the reconciliation phase, which means communicating with the real DOM to make the necessary update. It might seem ineffective, but reducing the time it takes for the browser to go through the whole tree structure will greatly boost the performance speed.

3.2 JSX

JSX is a syntax extension of JavaScript called JavaScript XML, helping developers write HTML code within JavaScript code. After the code is written, a preprocessor program such as Babel will go through and convert all JSX code into regular JavaScript code. Listing 3 shows an example of an element written in JSX and Listing 4 shows the code after being compiled by Babel.

```
const name = 'Anonymous';

const element = (
  <div className='container'>
    <h1>Hello, {name}</h1>
    <p>Have a nice day!</p>
  </div>
);

ReactDOM.render(element, document.getElementById('root'));
```

Listing 3. Code written in JSX.

```
const name = 'Anonymous';

const element = React.createElement(
  'div',
  {
    className: 'container',
  },
  React.createElement('h1', null, 'Hello, ', name),
  React.createElement('p', null, 'Have a nice day!')
);

ReactDOM.render(element, document.getElementById('root'));
```

Listing 4. JSX code converted into JavaScript code by Babel.

It is important to note that although JSX is similar to HTML, it follows a few different rules. JSX uses the *camelCase* naming convention for HTML attributes, as an example, `tabIndex` instead of `tabindex`. Custom components created in React are also capitalized, to separate the custom and native ones, for example, `Button` and `button`. It is possible to pass a JavaScript expression as an attribute value by using curly braces (`<Button onClick={this.props.doSomething} />`), or to insert the expression inside an element (`<p>{this.props.content}</p>`), as shown in Listing 3. Omitting the value of an attribute (`<Input checked />`) will make JSX treat it as true. So, to treat it as a negative value, pass in an attribute expression (`<Input checked={false} />`) [17].

Although it can be confusing at first, the introduction of JSX has made it easier for developers to write HTML code in React. With the help of JSX, there is no need to write `React.createElement()` for every new element created. Instead, the HTML-like syntax makes nestings of children elements easier to read [17]. JSX is also faster than regular JavaScript, as it performs optimization while doing the conversion.

3.3 Components

Components are the most fundamental units of React. They are pieces of a broken-down interface that can be composed, reused and easily organized. Components are well-encapsulated, which means that each component has its own logic and states to work from. However, they can also be composed together to form new complex and composite components. This is how they are supposed to work in React. Because they are self-contained and isolated, they are portable and reusable throughout the whole application [17]. An example of a useful component is a custom button with a similar style but changeable text. In this case, it would be beneficial for the developers not to be required to define the button every time they want to use it, instead to create a custom button.

There are two types of components in React, Class components and Functional components. Functional components are a more familiar concept and they are simply JavaScript functions. Before the introduction of React Hooks in React 16.8, functional components were mostly referred to as stateless or representational components because they can only accept data as props. With the release of React 16.8, components can now accept, mutate and return their own data, also known as states. Class-based components, following the ES6 class, extend the component class of the React library. They can define and control their own state and also have access to different phases in a React lifecycle (mount, update, unmount) [18].

Although both are components used in React, class-based and functional components have three major differences: the syntax, the state and the lifecycle methods. Using React hooks to create and manage states can be more straightforward than using a class and there is no major difference in performance. The `useEffect()` hook can also help developers implement lifecycle methods that were previously only achievable with class components. In a complex application, the functional components can be easier to understand for some developers, as there is no need for unnecessary method binding and the “this” keyword. The official React team has also stated that it is advisable to use functional components instead, as their documentation will be updated using mostly functional components [18].

3.4 React state and props

State in React is a built-in object that is used to contain data or information about the component at a given instant in time. In a complex and interactive web application, using state is inevitable to control how the application behaves. Without the help of React to help reduce and shield developers from the immensely complex states of modern UI, it can be difficult to create beautiful and fast web applications. There are generally two types of states in React, mutable and immutable states, also known as state and props [17].

Mutable state, or simply state, is the data that can be changed within a component. It is often used to store data of the components that have to be rendered to the view, meaning it is present in the UI. Data stored in state can be changed through event handlers, by using the `useState()` hook (functional component) or the `setState()` function (class component). Listing 5 shows an example usage of state in a React functional component. The state was created using the `useState()` hook in React and is mutable through the `setCounter()` function.

```
const ClickCounter = () => {
  const [counter, setCounter] = useState(0);
  const incrementCounter = () => {
    setCounter((prevCounter) => prevCounter + 1);
  };

  return (
    <div>
      <p>Counter: {counter}</p>
      <button onClick={incrementCounter}>Increase</button>
    </div>
  );
};
```

Listing 5. Example of state in React functional component.

On the contrary, immutable state, also known as props, is the data that a component receives that should not be changed by the component. They are often used by parent components to pass the necessary data down to the child components. Props are, however, not entirely immutable as the parent component can update the props passed down to its children, like passing down the parent's state that can change through time. Props are often required for the child component to work correctly with the parent, so there are a few APIs to help with the development. The most common tool is the PropTypes API, providing a type-checking functionality in which developers can specify what sort of props the component will expect to receive when used [17]. Listing 6 is an example of using props to pass data down to the child component from the parent component.

```
const Counter = (props) => {
  return (
    <div>
      <p>Counter: {props.value}</p>
      <p>Click the button to increase the counter!</p>
    </div>
  );
};
```

```
const CounterContainer = () => {
  const [counter, setCounter] = useState(0);
  const incrementCounter = () => {
    setCounter((prevCounter) => prevCounter + 1);
  };

  return (
    <div>
      <Counter value={counter} />
      <button onClick={incrementCounter}>Increase</button>
    </div>
  );
};
```

Listing 6. Example of props in React functional component.

3.5 React hooks

Introduced in React 16.8 in 2019, React hooks are functions that can be called from functional components and can “hook” into the key functionality of React: state and lifecycle. In other words, they give functional components access to the key features of React, previously only available with class components. The benefits it brings are less code, better code organization, no need for binding methods, a simpler lifecycle model and exportable features that can be reused [19]. Common hooks used are `useState()` to manage the state of the component locally, `useEffect()` to manage the lifecycle of the component, `useReducer()` and `useContext()` often used together to manage the state globally.

Hooks in React applications are used to keep related side-effect logic in one place [19]. React provides many useful hooks, but developers can also define custom hooks that are suitable for their use case. Each custom hook can maintain its own state and the state can be included in its return value. An example of custom hook is a hook that will fetch data from a specified URL when needed, store it locally and return that requested data to the component. This hook will be accessible by other components and help developers keep all the logic in one place.

3.6 Types of testing in React applications

The official React documentation has stated that there are two categories in testing React web applications: Rendering component trees and running a complete application, with the first being carried out before the second [20]. The former category is specific to React, as it renders based on the DOM and the latter is a more familiar concept used in other software development fields. However, both categories are important for the stability and performance of the application, so it is vital to spend some time writing test cases for both categories.

The first category usually contains unit testing and integration testing and the purpose is to test the components in a simplified environment and assert the output. Each test is written to simulate an event and then the output is captured and compared with the asserted output to determine whether the test passes or fails.

The second category, end-to-end testing, runs the application in a realistic browser environment to test its behavior and communication flow. It is usually more costly than the other category, but it helps to ensure the application works as intended on the UI and catches possible bugs before publication.

There are several testing tools that developers can choose from, based on their purposes. The highly recommended tool for testing React applications is React Testing Library, which will be discussed in the next chapter. Other recommended tools are Enzyme and Cypress. Cypress library is designed explicitly for end-to-end testing, but Enzyme and RTL can be used in both testing types for web applications.

4 TESTING IN REACT USING REACT TESTING LIBRARY

4.1 Jest

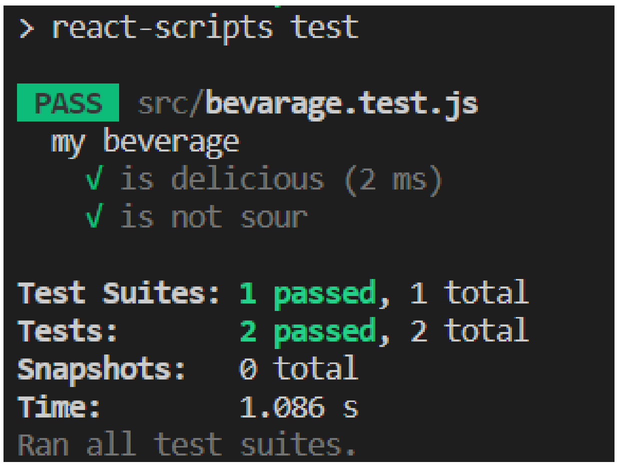
Jest is an open-source JavaScript framework developed by the Facebook team (the same as for React), primarily used to ensure the accuracy of JavaScript applications. The framework acts as a test runner, a mocking library and an assertion library, providing all the necessary toolkits and libraries for programmers to run their tests effortlessly. Furthermore, Jest offers a very elegant and human-readable coding pattern for writing tests, alongside its detailed and well-maintained documentation. This resulted in an easy but effective learning curve, making Jest one of the most popular JavaScript testing frameworks among those currently available, such as Mocha, Jasmine and others [21].

Having a good coding style and documentation is not the only reason for Jest's popularity. The framework is also very easy to install and set up, coming bundled together with the *Create React App* command. Developers can also simply install it using Node Package Manager or Yarn Package Manager. The tests are also run in parallel and in isolation from each other, resulting in a speed, performance and accuracy boost. Another powerful feature of the Jest framework is that it supports snapshot testing, a testing method that lets developers capture the render tree and compare it with the test result, to make sure no UI changes have happened unexpectedly.

In a unit or integration test, it is preferred to run the tests in an environment that imitates the browser in a lightweight manner, as that would help the tests perform fast and securely. This is also the way Jest works. Jest allows developers to access the DOM via jsdom, a lightweight browser implementation that runs inside Nodejs [20]. In most cases, jsdom behaves like a regular browser would, but it does not have features like layout and navigation. However, such features are rarely needed in web-based component tests, and having a simpler DOM environment will make a test run faster.

4.1.1 Test Runner

A test runner is a tool that aids in running the application tests either by an individual selection of test scripts by choice, in groups, or as a whole test suite. Once the test run is completed, the results are also documented and the library will report the success or failure status of the run [22]. By default, Jest finds test files that have the *.test*, or *.spec* suffix, or files in the __*test*__ folders that end with *.js*, *.ts*, *.jsx* and *.tsx* and executes them. This can be modified using the `testMatch` property in Jest's configuration file [23]. After the run is completed, a screen showcasing the test results in a human-readable manner will be displayed. An example of test results shown in Figure 9.



```
> react-scripts test

 PASS  src/bevarage.test.js
  my beverage
    √ is delicious (2 ms)
    √ is not sour

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.086 s
Ran all test suites.
```

Figure 9. Jest test runner result.

In Figure 9, the test file was executed with the command `react-scripts test`. When using the *Create React App* command to create React applications, it will also install and support Jest and the `react-scripts test` script can be used to run tests. Alternatively, tests can also be run using the `jest` script. These two commands can be followed by several options specifying how the tests should be executed. For example, the `--watch` option is to follow files for changes and rerun the tests that contain the modifications, or `--watchAll` to follow files for changes and rerun all tests [23].

4.1.2 Mocking Library

As discussed in Chapter 2, while writing tests, sometimes it is good to focus only on the component and mock other parts that are required for the test. In these cases, a mocking library is needed. The Jest framework has its own mocking library, which provides replacement for dependencies with the function `jest.mock()`, `jest.fn()`, or with other useful functions [23]. Creating test doubles allow developers to spy on the function that is called indirectly by some other code and inspect its behaviors.

Listing 7 shows a simple example of a test using mock functions. The `calculate` function was created before starting the test and in the test, a `mockAdd` function was created using `jest.fn()` which calculates and returns the sum of two arguments. Two assertions were then made to verify that the mock function was called one time and with the arguments of 1 and 2.

```
const calculate = (formula, a, b) => {
  return formula(a, b);
};

it('calculate function calls add', () => {
  const mockAdd = jest.fn((a, b) => a + b);

  const result = calculate(mockAdd, 1, 2);

  expect(mockAdd).toHaveBeenCalledTimes(1);
  expect(mockAdd).toHaveBeenCalledWith(1, 2);
  expect(result).toBe(3);
});
```

Listing 7. Example of function mock in a test. [23]

All mock functions created using `jest.fn()` have a special `.mock` property, which stores information about how the function was called and what the return value was [24]. Testers can use that property to check for the arguments passed and the return values. The example in Listing 7 can be rewritten to use the `.mock` property for the assertion, as shown in Listing 8.

```
it('calculate function calls add', () => {
  const mockAdd = jest.fn((a, b) => a + b);

  calculate(mockAdd, 1, 2);

  expect(mockAdd.mock.calls.length).toBe(1);
  expect(mockAdd.mock.calls[0][0]).toBe(1);
  expect(mockAdd.mock.calls[0][1]).toBe(2);
  expect(mockAdd.mock.results[0].value).toBe(3);
});
```

Listing 8. Example of using mock function's mock property.

Another useful case of Jest's mocking library is the ability to mock existing modules or dependencies in the tests using `jest.mock()`. One example case is using the function to mock API calls so that the tests do not call the actual APIs. It makes the test less fragile and faster. After the dependency is mocked, its implementation and return value can also be mocked using `mockImplementation`, `mockResolvedValue`, or `mockReturnedValue` [24]. An example of the use case of the function can be seen in Listing 9. First, `jest.mock('axios')` was called to mock the actual library. Then `mockResolvedValue` was used to mock the data returned from the asynchronous "GET" API call. Finally, when the method `all()` from the `Users` class was called, it is expected to return the mocked data instead.

```
//users.js
import axios from 'axios';
class Users {
  static all() {
    return axios.get('/users.json').then((resp) => resp.data);
  }
}
export default Users;
```

```
//users.test.js
import axios from 'axios';
import Users from './users';
jest.mock('axios');
it('should fetch users', () => {
  const users = [{ name: 'Mike' }];
  axios.get.mockResolvedValue({ data: users });
  return Users.all().then((data) => expect(data).toEqual(users));
});
```

Listing 9. Example of using jest.mock(). [24]

4.1.3 Assertion Library

In a test, after performing an action with a set of inputs, the result is captured and compared with the expected value. This is usually carried out with the help of the assertion library. Jest also provides built-in assertion methods for developers to test values in different ways. These methods are called “matchers”. There are several different matchers in Jest, for example, `toBe()`, `toHaveBeenCalledWith()`, `toHaveBeenCalledTimes()` and a few more, to help developers test the captured value in different ways [24].

Listing 10 shows an example of a Jest test file. A test file can contain one or many test suites, separated by the keyword `describe()` and a test suite can contain one or more related test cases defined by the keyword `it()` or `test()`. The `expect` function returns expectation objects. These objects are mostly paired with the matchers to assert certain values [24]. In the Listing, `.toBeTruthy()` and `.toBeFalsy()` are matchers. Failing matchers are tracked when Jest runs the tests, so it can print out understandable error messages for testers.

```
const myBeverage = {
  delicious: true,
  sour: false,
};
describe('my beverage', () => {
  it('is delicious', () => {
    expect(myBeverage.delicious).toBeTruthy();
  });
  test('is not sour', () => {
    expect(myBeverage.sour).toBeFalsy();
  });
});
```

Listing 10. Example of a test suite and test case structure. [23]

4.2 React Testing Library

React Testing Library (RTL) is built based on the DOM Testing Library, a lightweight solution for testing web pages by querying and interacting with DOM nodes in a similar way that users find elements on the web. As a result, developers will be confident that the application will work as expected when a real user uses it. A key feature of RTL is that it avoids testing implementation details, such as the internals of a component, though in some cases, it is still possible. Instead, it emphasizes a focus on testing in ways that closely resemble how users will interact with the application, as recommended in its official documentation [25].

Although it is not required, RTL and Jest usually come bundled together as a choice for any testers. RTL will render the React component, search for a DOM node that needs to be tested, perform a user action, then pass the object to Jest for comparison. Jest can test the properties of the queried node, for example, if it exists in the DOM, or if it has a certain length.

4.2.1 Key Features of React Testing Library

To test a component, developers first need to import it to the test file, then render it. RTL provides a very useful `render()` function to render the component.

After that, the testers will search or query for elements on the page. There are different types of queries, `get`, `query` and `find`. Those queries can be accompanied by several options, including `ByRole`, `ByLabelText` and `ByPlaceholderText`. Different queries will result in whether or not the query will throw an error if no element is found. As a rule of thumb, `get` should be used to find existing elements, `query` to assert the absence of an element and `find` for elements that appear asynchronously. The queries are called on a `screen` object provided by the library, for example, `screen.getByText('cancel')`. The `screen` object comes pre-bound to `document.body` – the element that contains the entire content of the document.

After an element is found, testers can use the `user-event` library to perform user interactions that can happen in an actual use case, such as clicking on a button. However, the user event should be set up before the component is rendered, which means calling the `userEvent.setup()` function before the `render()` function. Alternatively, a lightweight `fireEvent` wrapper can be used to dispatch DOM events. These two functions are similar, but `userEvent` simulates full interactions that can fire multiple events and perform additional checks along the way. It is recommended to use `userEvent` as sometimes when performing an action, the browser has to do more than just trigger one event for that interaction [25]. For example, to imitate a user changing the value of an input, using `fireEvent.change` will simply trigger a single change event on the input. On the contrary, using `userEvent.type` will trigger `keyDown`, `keyPress` and `keyUp` events for each character, making it more realistic. User events are performed using functions such as `.click()`, `.dbClick()`, `type()`, `.submit()` and so on.

Sometimes, performing a user action will trigger an asynchronous update to the component, for example, clicking the delete button and waiting for the deleted item to disappear. Such events require developers to wait for them to happen before continuing the tests. Fortunately, RTL provides users with some utilities to deal with asynchronous changes in the DOM, namely `waitFor()` and `waitForElementToBeRemoved()`. `waitFor()` expects a callback as an argument and it will run the callback several times until the timeout is reached, or until the expectation passes. If it is only required to test for the disappearance of an element, developers can use `waitForElementToBeRemoved()` instead and as the name suggests, the function waits for the removal of one or more elements from the DOM. `waitForElementToBeRemoved()` is a small wrapper around the `waitFor()` utility. The default timeout is 1000 milliseconds and the default interval is 50 milliseconds. However, both functions also accept an `options` object as a second argument, specifying the timeout, interval and other properties for them [25].

4.2.2 Benefits and Drawbacks of React Testing Library

One of the biggest benefits of using React Testing Library is in the testing principle of the core library: "The more your tests resemble the way your software is used, the more confidence they can give you." [25]. This means RTL will provide users with functions that can help avoid testing the implementation details of a component and focus on imitating how the user uses the application. In other words, if changes are made to the code base and not the workflow, the tests will not break. However, testing based on the user workflow is also the reason finding bugs might become a little difficult if the tests are not well-implemented.

Furthermore, avoiding implementation details and testing only the output of user behaviour is not always preferred, especially in unit testing. For example, if clicking on a button in one component results in a change in another component. In this case, it is more beneficial to mock and test the function called than monitor the actual changes.

At the moment, RTL only includes one method of rendering, full mounting. Full mount means it will render the component under test fully, including its children into the DOM. This is expected, as the principle of RTL is to test the user behaviour on the component and also very beneficial to test if there are any side effects when executing an action. However, it is also very costly and the execution time will increase. One solution is to group similar test cases into one `it()` block and avoid too many re-renderings. Grouped tests should be relevant to each other.

Whether the traits that define React Testing Library are perceived as a benefit or a drawback depends on each person's use cases. Ultimately, RTL is still a solid choice for a testing library with its growth, the tools it offers and the community support it has.

4.3 Difference between React Testing Library and Enzyme

Before React Testing Library became widely used in 2020, Enzyme was the choice among other testing tools for most developers. Created by Airbnb, the Enzyme library helps render React components in the same way they would be rendered on the browser. The Enzyme library provides the utilities to manipulate and test the state, props and children. However, React Testing Library gives no access to the state and props but provides the ability to reproduce user actions to test the changes happening on the UI. Both are used for testing React components, providing the necessary tools to do so, but with different approaches: The Enzyme library enables writing Test Driven Development (TDD), while React Testing Library focuses on Behavioral Driven Development (BDD).

The Enzyme Testing Library is no longer supported in React version 18, so more and more developers are inevitably migrating to the React Testing Library, as shown in Figure 10. However, it is still worth knowing the difference between the two most popular testing libraries and how they differ from each other, to understand the libraries better.

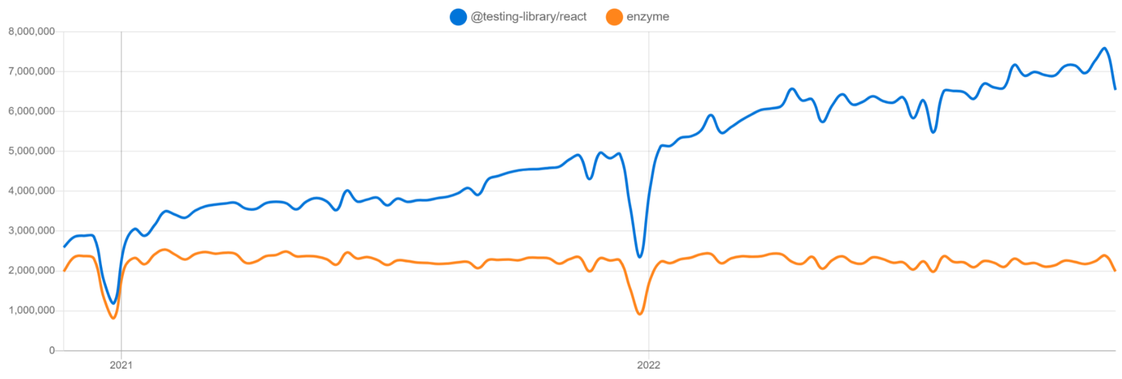


Figure 10. The popularity of Enzyme and RTL. [26]

4.3.1 Concepts

The first difference between the RTL and Enzyme Libraries is the concept of testing. As explained above, Enzyme leans more towards testing the implementation details, allowing testers to access the state and props of the component under test and modify them to match the test case. Tests in Enzyme are more like actual unit testing since Enzyme isolates the component to test its state and prop values, the internals of a component. However, this does not mean that the React Testing Library is not used for unit testing. The React Testing Library just focuses more on user-flow-based testing, often mimicking the user behaviour of the component. These concepts are further demonstrated in Listings 12 and 13.

```
import React from 'react';

const Form = () => {
  const [value, setValue] = React.useState('');

  return (
    <div>
      <input value={value} onChange={(e) => setValue(e.target.value)} />
      <h3>{value}</h3>
    </div>
  );
};
export default Form;
```

Listing 11. Implementation of a form component.

Listing 11 shows an example implementation of a form component. The form has a simple `input` component with the props `value` and `onChange`. The value is captured using the useState hook and displayed in a header component.

```
import React from 'react';
import { shallow } from 'enzyme';
import Form from './Form';

describe('Form', () => {
  const setState = jest.fn();
  const useStateSpy = jest.spyOn(React, 'useState');
  useStateSpy.mockImplementation((initialValue) => [initialValue, setState]);

  it('should update state on input change', () => {
    const wrapper = shallow(<Form />);
    wrapper
      .find('input')
      .props()
      .onChange({ target: { value: 'my input' } });
    expect(setState).toHaveBeenCalledWith('my input');
  });
});
```

Listing 12. Form component test using Enzyme.

Listing 12 is a demonstration of a test performed using the Enzyme library. First, a `setState` mock function was created and the default implementation of the `useState` hook was also overwritten, returning the mocked function instead of the actual one. The test then proceeded with a shallow rendering of the `Form` component (the concept of shallow render and full render will be explained further in the next section). The rendered component, `wrapper`, continued to find the input and modify the `onChange` prop to pass in a target value “my input”. Finally, the test spied on the mocked `setState` function, expecting it to have been called with “my input”. This test heavily depends on the component state and props, as all the arrange, act and assert sections involve them. This can be convenient, but it not useful to an end-user, who only sees the rendered change to the UI.

```
import React from 'react';
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import '@testing-library/jest-dom';
import Form from './Form';

describe('Form', () => {
  it('should update state on input change', () => {
    render(<Form />);
    userEvent.type(screen.getByRole('textbox'), 'my input');
    expect(screen.getByRole('heading')).toHaveTextContent('my input');
  });
});
```

Listing 13. Form component test using RTL.

A similar test was implemented using RTL and shown in Listing 13. In this test, there were no more changes made to the state and props of the component. Instead, the first step was to render the `Form` component. After that, a user event mimicking the typing action was called. This call used the `screen` object to query for the `input` component by its role, “textbox”. The last step was to verify that the text content rendered to the header component `h3` was "my input". The flow of this test mimics a user’s workflow on the component, first seeing the form, typing something to it and seeing the outcome. As a result, the test gives developers more confidence that their applications will work in a real-life scenario.

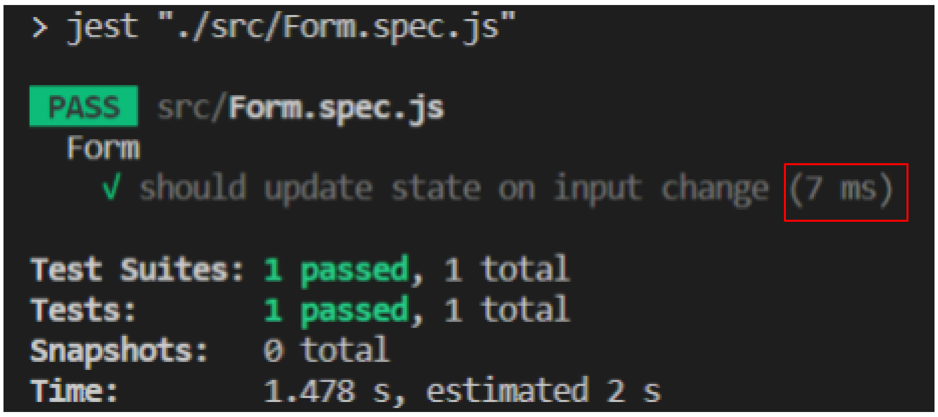
According to the creator of RTL, Kent C. Dodds, “implementation details are things which users of your code will not typically use, see, or even know about” [28]. Those details are the states, props, hooks, or names of the components that users do not usually care about. In Enzyme, several APIs are available that allow developers to find these implementation details, such as `find(‘componentName’)`, `component.props()`, or `component.state()` and assert them to be in the desired state. Using these APIs should be avoided, however, as they can give false positive or false negative results and are hard to maintain when the application grows or when the components need to be refactored, as a single change in the name can cause the whole test to fail. For example, if the `<input />` component was renamed to `<Input />`, the test will fail immediately. But in reality, the component does not have any bugs in it and still functions normally. That is called a false negative result. This can be avoided in RTL, as it does not allow querying the component by name but by role, which is “textbox”, or by the placeholder text of the component. This guideline encourages developers to write better test cases by selecting a DOM element by a few properties such as `placeholder`, `role`, `test-id`, `label`, `text`, or `aria`, which is less likely to get changed than the component names and CSS selectors.

The test case provided in Listings 12 and 13 is quite simple and it is just an example of how different the test approaches of the two libraries are. As the application grows, more and more test cases will be added. Using the Enzyme library will ensure the correct states of the component, but it will not ensure that the results that users see are the desired results. On the other hand, using RTL will not give developers access to the internals of the component, meaning there is no way to verify the state changes happening within the component, but it will ensure all the things users see are correct and intentional.

4.3.2 Rendering Methods

The second big difference between RTL and Enzyme are their rendering methods. As briefly introduced in the last section, there are two ways to render a React component in unit testing, shallow and full rendering. Shallow rendering, or shallow mounting, renders the component in isolation and “one level deep”. This means that only the code defined inside the component is rendered, anything imported elsewhere will not be rendered, including its children. Shallow render returns a JavaScript object and does not require a DOM. Full rendering, or mounting, as opposed to shallow mounting, renders the whole component including its child components into the DOM, using the jsdom library. This render method gives the component full access to the lifecycle methods and DOM APIs, but it is also costlier in the execution time, as it renders the full component tree.

The Enzyme library supports both rendering methods, offering the `shallow()` function for shallow mounting and `mount()` and `render()` for full rendering. Enzyme gives developers the flexibility to choose which method to call, depending on the component. Unfortunately, that is not possible in RTL, as it only provides users with a single `render()` function for full rendering. The child components can still be mocked, however, with Jest’s built-in mock library. Getting the full component tree can be useful when testing higher-order components in React, but because there is no flexibility in choosing the render method, tests written using RTL will generally have a longer run time than tests using Enzyme, as shown in Figures 11 and 12. Figure 11 showcases the runtime of the test using the Enzyme library, taking only 7 milliseconds to run the test case, while Figure 12 reports the runtime of the test using RTL, taking 95 milliseconds. The time difference in a small test file is not major, but with more and more test cases added to a test file, the overall runtime will also increase, resulting in a larger gap.

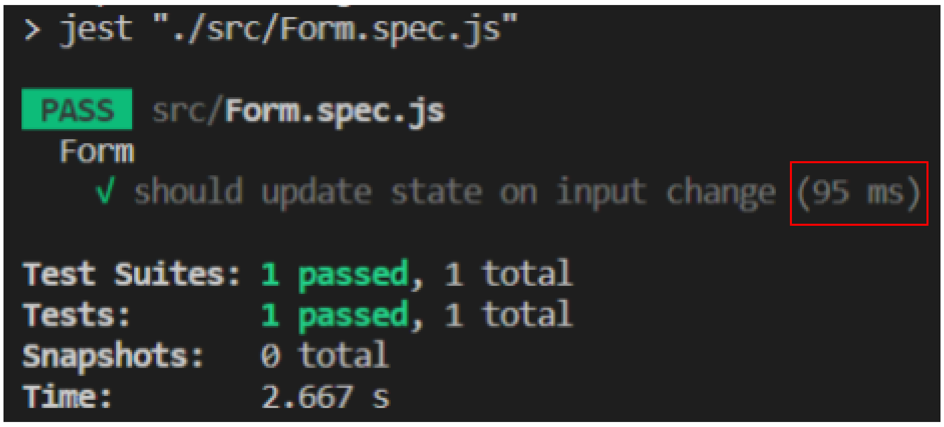


```
> jest "./src/Form.spec.js"

 PASS  src/Form.spec.js
  Form
    √ should update state on input change (7 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.478 s, estimated 2 s
```

Figure 11. Report of the test using the Enzyme library.



```
> jest "./src/Form.spec.js"

 PASS  src/Form.spec.js
  Form
    √ should update state on input change (95 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        2.667 s
```

Figure 12. Report of the test using the RTL.

5 EXAMPLES OF USING REACT TESTING LIBRARY IN REACT APPLICATIONS

In this chapter, the steps needed to set up Jest and React Testing Library for a React application will be discussed. The example application will be a full-stack Expense Tracker web application where users can add and remove their expenses. The built components will be tested and several notable test scenarios will be documented in this study.

5.1 Setting up the Application

In order to run React applications, a runtime environment called NodeJS will need to be installed. The installation of this platform will give us the package manager tool `npm` that helps put the modules of the application in a place where NodeJS can find and manage dependency conflicts. React applications can be created using the command `npx create-react-app app-name`. The command comes with the built-in react scripts for React Testing Library and Jest. The Expense Tracker React application was also created using this command. The application test files can be run either with the built-in `react-scripts test` or the `jest` command and choosing which option to use is up to the developer or the team owning the application. In this study, the built-in command was chosen to run the test files, since using the Jest requires a few additional setups to the jest configuration file.

Firebase and Firestore were selected as backend services for the Expense Tracker Application. Firebase is a toolset with the necessary functions that cover a large portion of the services developers would normally have to build themselves, including authentication, analytics, configuration, database storage and communication between the back-end and front-end [29]. The services are hosted on the cloud, maintained and operated by Google and require little effort from the developers to set it up. The database storage used by Firebase is called

Firestore, a NoSQL document database that has a collection and document data model and each document can have its sub-collections. It is designed to scale automatically depending on user demand, therefore, the database size does not impact query time. Since the purpose of this application is to be used for front-end unit testing, Firebase is a suitable choice with its built-in tools for communication.

To create a new project on the Firebase console page, a Google account is required. After the project has been created, the API key and a few additional pieces of information need to connect the database server with the application can be retrieved from the project settings page. Listing 14 shows an example of the connection in the Expense Tracker web application.

```
import { initializeApp } from 'firebase/app';

const firebaseConfig = {
  apiKey: process.env.REACT_APP_API_KEY,
  authDomain: process.env.REACT_APP_AUTH_DOMAIN,
  projectId: process.env.REACT_APP_PROJECT_ID,
  storageBucket: process.env.REACT_APP_STORAGE_BUCKET,
  messagingSenderId: process.env.REACT_APP_MESSAGING_SENDER_ID,
  appId: process.env.REACT_APP_APP_ID,
};
const app = initializeApp(firebaseConfig);
```

Listing 14. Setup for the Firebase server in the application.

The application consists of three pages: a Login page, a Register page and a Home page. The layout of the application is shown in Figures 13, 14 and 15. Users will first need to sign up for an account using their email, then they can use that account to log in and add their expenses. Forms are handled using the Formik library, the most popular and flexible library in React used to manage form data. Combined with the schema builder Yup, Formik can validate users' input and determine whether it follows the conditions defined. After all the data has passed the validation and the user clicks the submit button, an operation will be performed depending on the functionality of the form. On the Register page, when the user clicks the Sign-up button, a built-in function `createUserWithEmailAndPassword` from Firebase's authentication library will be called to create the new user account. Similarly, when the Sign-in button is clicked on the Login page, the `signInWithEmailAndPassword` function will be called and the user will be logged in, with their information stored in Firebase's `currentUser` property.

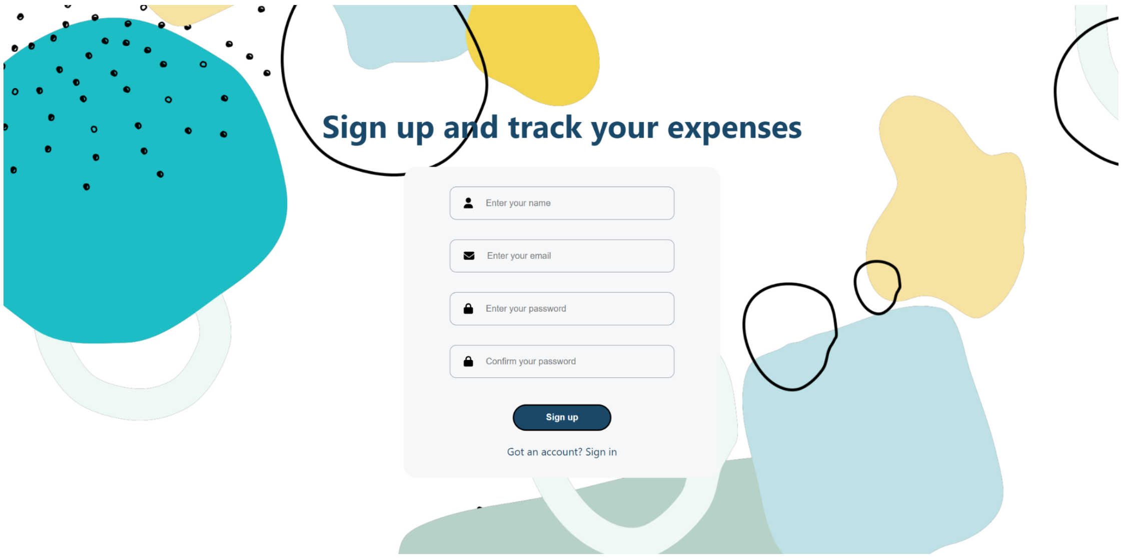

Sign up and track your expenses

Figure 13. Register page.

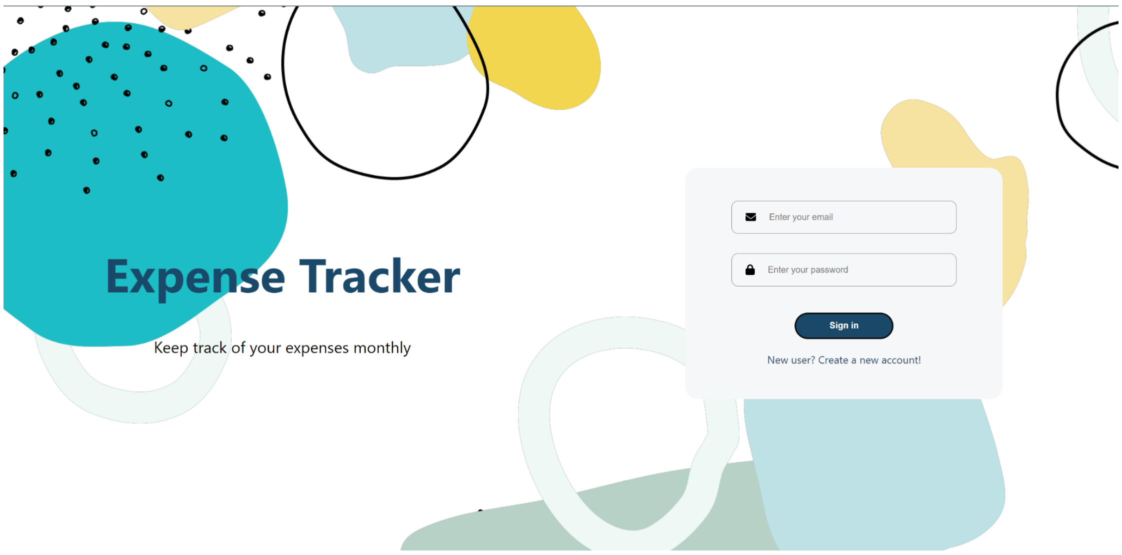
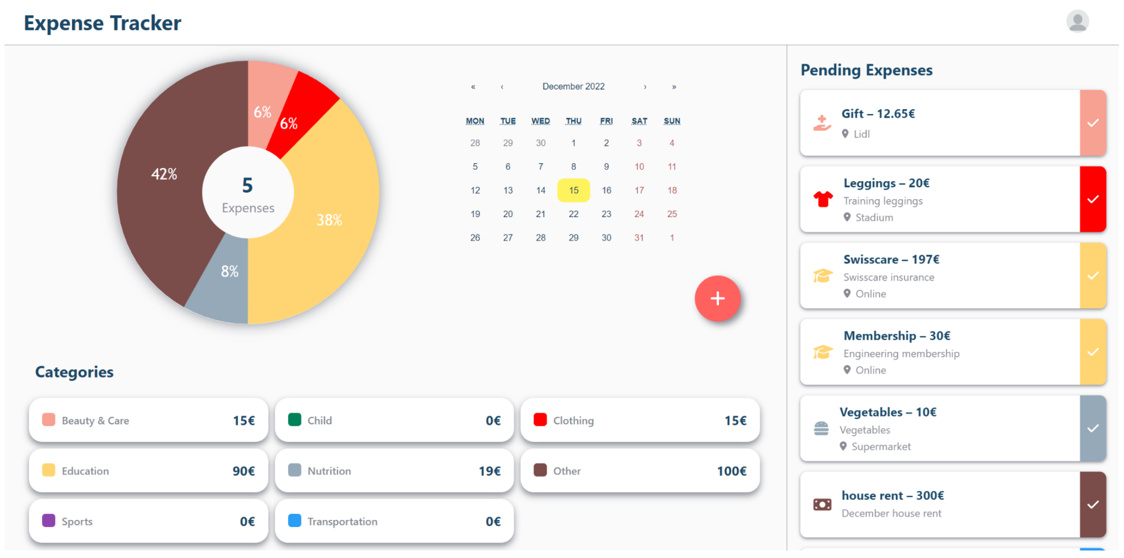

Expense Tracker
Keep track of your expenses monthly

Figure 14. Login page.



Categories

Beauty & Care 15€	Child 0€	Clothing 15€
Education 90€	Nutrition 19€	Other 100€
Sports 0€	Transportation 0€	

Pending Expenses

Gift – 12.65€
Lidl

Leggings – 20€
Training leggings
Stadium

Swisscare – 197€
Swisscare insurance
Online

Membership – 30€
Engineering membership
Online

Vegetables – 10€
Vegetables
Supermarket

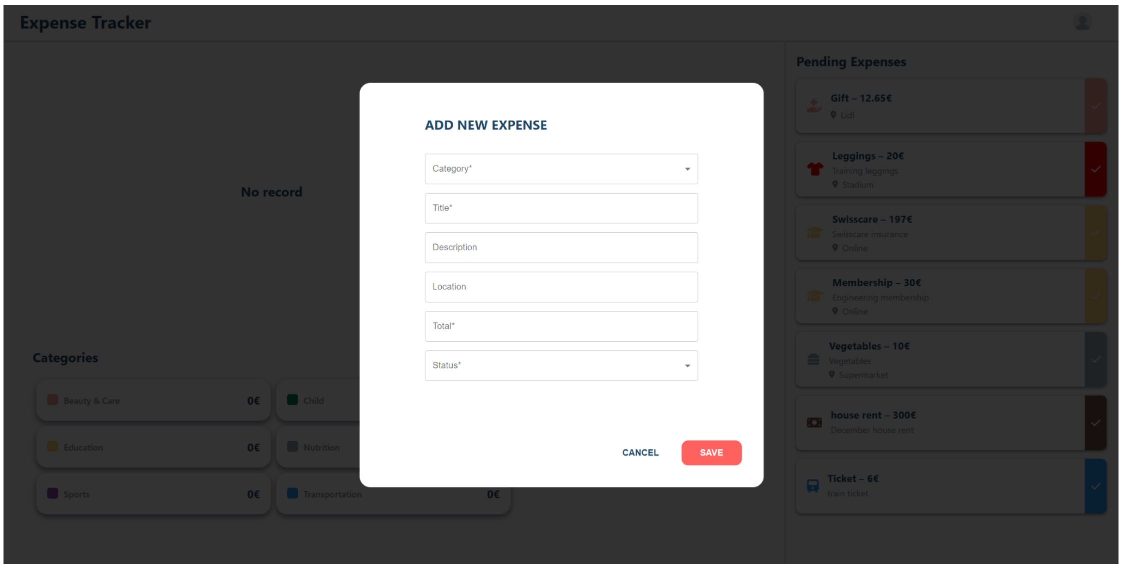
house rent – 300€
December house rent

Figure 15. Home page.

The Home page consists of several different components and it is the main focus of this application. On the page, all communications with the Firestore database, such as retrieving and saving data, are performed through Redux actions and maintained in the Redux store. Expenses saved on the database have two different states: pending and confirmed. Upon logging in, a Redux action will be called to fetch the current payment data of the user from the database. In Figure 15, the right column shows a list of pending costs that users can verify. The left column has a pie chart built using the popular open-source Victory library, which calculates the percentage of the confirmed expenses of each category for the current month and a summary list below that displays their total in number. When the user confirms the pending payment on the right column, it will be added to the left column and the percentage will be re-calculated.

The last essential functionality of the Home page is adding new expenses, which can be done by clicking on the "+" button next to the calendar. After that, a form will appear as in Figure 16, where users can enter information for the new expense. This form was also built using Formik in combination with the schema builder Yup. When filling out the form, category, title, total cost and status are required, while description and location are optional. After the user clicks the

Save button, a Redux action will communicate with Firestore to save the new document to the user's Expenses collection.



Beauty & Care	0€
Child	
Education	0€
Nutrition	
Sports	0€
Transportation	0€

Gift – 12.65€
Lidl

Leggings – 20€
Training leggings
Stadium

Swisscare – 197€
Swisscare insurance
Online

Membership – 30€
Engineering membership
Online

Vegetables – 10€
Vegetables
Supermarket

house rent – 300€
December house rent

Ticket – 6€
train ticket

Figure 16. Add new expenses form.

Each component of this application was tested using Jest and React Testing Library and verified thoroughly through different test cases. In the following sections of this chapter, the most common and essential testing scenarios are documented and discussed with the application's components as examples.

5.2 Defining a Custom Test Render Function for Redux

The example application operates with Redux as the state management tool, so it is required to set up a provider for any components connected to the Redux store. It is also important that each separate test has a separate store instance, so no values will be leaked between them [30]. One simple way to do it is to define a custom render function that creates a new Redux store and wraps the children components in a provider every time it is called, as detailed in Listing 15.

```
import React from 'react';
import { render } from '@testing-library/react';
import { configureStore } from '@reduxjs/toolkit';
import { Provider } from 'react-redux';
import rootReducer from 'redux/reducers';

const renderWithProviders = (
  ui,
  {
    preloadedState = { categoriesData: [] },
    store = configureStore({ reducer: rootReducer, preloadedState
}),
    ...renderOptions
  } = {}
) => {
  const Wrapper = ({ children }) => {
    return <Provider store={store}>{children}</Provider>;
  };

  return { store, ...render(ui, { wrapper: Wrapper, ...renderOptions
}) };
};

export * from '@testing-library/react';

export { renderWithProviders as render };
```

Listing 15. Setup for a custom render function.

In the code snippet, the `renderWithProviders` function accepts the component, named `ui`, as the first argument and an optional object as the second argument, with a `preloadedState` value that can be used for initial data and a `store` value, in case there are any different store configurations. If no values are passed in, the function will automatically create a default store instance with an empty initial state. The component under test is then wrapped in Redux's `<Provider store={store}>` every time it is rendered. The custom render function returns all of the methods of the testing library, so the tester will still have access to the original functionalities and the properties of the store on every render. Listing 16 is an example use case of the function in the test for the form used to add new expenses, `<AddExpenseModal />`. The initial categories data for the Redux store was passed in using the `preloadedState` option.

```
import React from 'react';
import { render } from 'utils/test-utils';
import AddExpenseModal from './AddExpenseModal';

const initialCategoriesData = [
  {
    color: '#FFD573',
    icon: 'education',
    id: '1',
    name: 'Education',
  },
];

//...
render(<AddExpenseModal />, {
  preloadedState: { categoriesData: initialCategoriesData },
});
```

Listing 16. Test component rendering.

5.3 Mocking the Dependencies

For a component to work, sometimes dependency libraries need to be imported and then the functions provided can be used, such as hooks or database mutation functions. Common ways to avoid interfering with the database are to create a new test database or simply mock the functions that communicate with the database. For simplicity, the tests in this thesis will go with the second method.

Listing 17 shows the function to create and save a user to the collection. This function will be called when the user clicks the Sign-up button on the Register page. When writing tests for the page, the helper functions need to be mocked, as demonstrated in Listing 18, so they will not create a new user on the database. Besides the methods seen in Listing 17, the related methods used to set up the communication between the application and the server also require mocking so they will not produce errors when testing, namely the `getAuth()` and `getFirestore()` functions. Furthermore, since their original implementation returns an object, a `null` object must be returned with `jest.fn()`.

```
import { auth, methods, db } from 'utils/firebase';
import { doc, setDoc } from 'firebase/firestore';
//...

const Register = () => {
  const onRegister = async (values) => {
    const { name, email, password } = values;
    try {
      await methods.createUserWithEmailAndPassword(auth, email, password);
      await setDoc(doc(db, 'usersData', auth?.currentUser.uid), {
        name,
        email,
      });
    } catch (err) {
      console.log(err);
    }
  };

  //...
};
```

Listing 17. onRegister function.

```
jest.mock('@firebase/auth', () => ({
  getAuth: jest.fn(() => null),
  createUserWithEmailAndPassword: jest.fn(),
}));
jest.mock('firebase/firestore', () => ({
  getFirestore: jest.fn(() => null),
  doc: jest.fn(),
  setDoc: jest.fn(),
}));
```

Listing 18. Mocks for the Firebase functions.

The second way to mock the Firebase functions is not directly mocking them, but mocking the action function that uses them. The “action function” here can be a utility function or a Redux action. Listing 19 shows an example of a Redux action fetching the categories information from the database and saving them to the

Redux store, which was used on the Home page when first loaded. In the component test, only the `fetchCategories` action needs to be mocked, instead of the Firebase methods that were used, detailed in Listing 20.

```
const fetchCategories = () => {
  return async (dispatch) => {
    const categoriesQuery = query(
      collection(db, 'categoriesData'),
      orderBy('name', 'asc')
    );

    const querySnapshot = await getDocs(categoriesQuery);
    const categoriesData = querySnapshot.docs.map((doc) => {
      const data = doc.data();
      const id = doc.id;
      return { id, ...data };
    });

    dispatch({ type: CATEGORIES_CHANGE, categoriesData });
  };
};
```

Listing 19. fetchCategories action.

```
jest.mock('redux/actions', () => ({
  fetchCategories: () => ({
    type: 'TESTING',
  }),
}));
```

Listing 20. Mocks for the fetchCategories function.

Both methods have benefits and drawbacks of their own. The second method might be more straightforward and cleaner, but it is not always possible to use if there is no access to the function. In such cases, the first method should be used and mock all the Firebase methods that were used in a function. How one chooses to mock the database function should ultimately aid us with the tests so that there will be no data mutation.

5.4 Checking the Contents Rendered

When performing a test for any component, testers should first check for its content rendered on the screen. This way, the developers can make sure all expected information is displayed to the user, with no unexpected additional texts. The contents can be tested with different options. In the official document by RTL, the recommended query options are `ByRole`, `ByLabelText` and `ByPlaceholderText`, followed by other additional ones. Listing 21 shows an example of the content on the Login screen being tested. The `getByRole` query has two arguments: the first one is to specify the role name, such as heading, button, or link, while the second argument is an optional object, where its name, description, or other related properties can be passed in to distinguish the component from other possibly similar ones. A similar test was implemented for every component in the application.

```
describe('render', () => {
  it('shows the correct content', () => {
    expect(
      screen.getByRole('heading', { name: 'Expense Tracker' })
    ).toBeInTheDocument();
    expect(
      screen.getByText('Keep track of your expenses monthly')
    ).toBeInTheDocument();
    expect(
      screen.getByPlaceholderText('Enter your email')
    ).toBeInTheDocument();
    expect(
      screen.getByPlaceholderText('Enter your password')
    ).toBeInTheDocument();
    expect(
      screen.getByRole('button', { name: 'Sign in' })
    ).toBeInTheDocument();
    expect(
      screen.getByRole('link', { name: 'New user? Create a new account!' })
    ).toBeInTheDocument();
  });
});
```

Listing 21. Rendered content test.

When writing the test, the minimum requirement is to ensure all the critical information is present. Additional checks can also be included to confirm that no incorrect information is shown, like in Listing 22, where the "No Record" text is verified it will not be shown if expense data is available. The assertion is performed using `queryByText` instead of `getByText` to check for the absence of an element, as mentioned in the key features of the React Testing Library. Another test case was also added to make sure the expenses are presented correctly and the pie chart includes two slices, specified in the `initialCategoriesData` array.

```
const initialCategoriesData = [
  {
    color: '#F6A192',
    icon: 'healthcare',
    id: '1',
    name: 'Beauty & Care',
    expenses: [
      {
        creation: {
          year: 2022,
          month: date.getMonth() + 1,
          date: 1,
        },
        description: 'Test expense',
        title: 'test',
        status: 'C',
        total: 15,
      },
    ],
  },
  {
    color: '#FFD573',
    icon: 'education',
    id: '2',
    name: 'Education',
    expenses: [
      {
        creation: {
          year: 2022,
          month: date.getMonth() + 1,
          date: 12
        },
        description: 'education expense',
        title: 'Education test',
        status: 'C',
        total: 20,
      },
    ],
  },
];


describe('if there are some expenses data', () => {
  it('shows the correct content', () => {
    render(<ExpenseChart />, {
      preloadedState: { categoriesData: initialCategoriesData },
    });
    expect(screen.queryByText('No record')).not.toBeInTheDocument();
    expect(screen.getByText('2')).toBeInTheDocument();
    expect(screen.getByText('Expenses')).toBeInTheDocument();
  });

  it('has the correct amount of slices', () => {
    render(<ExpenseChart />, {
      preloadedState: { categoriesData: initialCategoriesData },
    });
    expect(screen.getAllByRole('presentation')).toHaveLength(2);
  });
});
```

Listing 22. Additional content tests.

5.5 Testing Input Changes

Another common test case for a web application is testing for the input entered by the user. In React applications, a common way to handle input value changes is to use hooks in combination with the input `onChange` event. However, the test should not care about internal implementations at all, but only check to see if the value rendered on the screen reflects the user's input correctly and if any feedback is shown when the input does not follow the correct format. Fortunately, with the help of the companion `user-event` library, it is not difficult to simulate user interactions in tests.

In the example applications, three components require the user's input, the login form, the register form and the expense adder form. Although the expense adder form uses the Material UI library's `<TextField/>` component, all three components use the `type` method from the `user-event` library to imitate the user typing action. Some snippets of testing the input changes can be seen in Listing 23, where the positive and negative test cases are included together in a `describe()` block.

```
describe('typing expense information into the modal', () => {
  it('shows the correct value', async () => {
    render(<AddExpenseModal />, {
      preloadedState: { categoriesData: initialCategoriesData },
    });

    await user.type(screen.getByRole('textbox', { name: 'Title*' }),
'test');
    await user.type(screen.getByRole('spinbutton', { name: 'Total*' }),
'20');
    //...

    expect(screen.getByRole('textbox', { name: 'Title*' })).toHaveValue(
      'test'
    );
    expect(screen.getByRole('spinbutton', { name: 'Total*' })).toHaveValue(
      20
    );
    //...
  });

  it('shows the correct error if value is not valid', async () => {
    render(<AddExpenseModal />, {
      preloadedState: { categoriesData: initialCategoriesData },
    });


    await user.type(
      screen.getByRole('textbox', { name: 'Title*' }),
      'seddoeiusmodtemporinc'
    );
    //...
    await user.click(screen.getByRole('spinbutton', { name: 'Total*' }));

    expect(
      screen.getByText('Title cannot be longer than 20 words!')
    ).toBeInTheDocument();
    //...
  });
});
```

Listing 23. Input changes test.

When testing for the changes of any input component, the process is similar. Firstly, `user.type()` was used to imitate the user action of typing into the text field. Then, depending on the purpose of the test, testers can continue with the next input, click the submit button, or close the list of options. These actions indicate that the input is no longer focused and will trigger feedback from the form handler. Finally, the result of the inputs can be verified by checking for its content value or checking for the error rendered on the screen.

When doing an action with the `user-event`, it is important to know when to wait for the action to complete before doing anything further. In Listing 23, the `await` keyword was used when typing into the text box to wait for the action to complete before moving to the next one. The keyword was used again in the second test, to wait for the click to complete before checking for the error. This is because the error will only be shown after the input is removed from the focus, but its value is not in the correct format.

5.6 Checking for the Appearance/Disappearance of a Component

Similar to waiting for an action completion when performing it, sometimes it is necessary to wait for an element to appear or disappear before doing the assertion. A common example of this case is when developers implement a feedback banner after a user completes an action, such as adding an item to their cart. In the Expense Tracker application, a banner will appear when the user finishes adding a new expense or confirms a pending expense. Listing 24 demonstrates the test for this case. The click action was first performed on the confirm button with `user.click(screen.getByRole('button'))`, then waiting for the banner to appear using the `await` keyword with React Testing Library's `findByText` query. The `findBy...` query returns a promise which resolves when an element is found. If the element is not found within 1000 milliseconds, it returns an error. The query is suitable for this case as the banner is not present when the component is rendered, but only appears after the user has clicked the confirm button.

```
describe('when clicking on the confirm button', () => {
  it('shows a notification banner', async () => {
    render(<ExpenseList />, {
      preloadedState: { categoriesData: initialCategoriesData },
    });
    user.click(screen.getByRole('button'));
    expect(
      await screen.findByText('Successfully updated!')
    ).toBeInTheDocument();
  });
});
```

Listing 24. Notification banner test.

A different example in the application occurs in the form of adding new expenses. When selecting an option from the list, there are two things to wait for: the appearance of the options list after clicking on the select component and the removal of the options from the DOM tree when an option has been selected. Listing 25 shows the reproduction of these actions. In the Listing, the keyword `await` was used twice, in correspondence to the two actions explained previously. The first usage is similar to Listing 24, where the query `findByRole` accompanies the keyword to wait for the option to appear before clicking it. After an option was chosen, `waitForElementToBeRemoved()` from the testing library was used to wait for the options to be removed and states to be updated. Without this helper function, all the test assertions will fail as the DOM tree will render the list of options on top of the other elements.

```
//...
user.click(screen.getByLabelText('Category*'));
user.click(
  await screen.findByRole('option', {
    name: initialCategoriesData[0].name,
  })
);
await waitForElementToBeRemoved(() => screen.queryAllByRole('option'));
//...
```

Listing 25. Imitation of selecting an option in the test.

5.7 Testing Function Calls

Buttons are a common way to confirm or cancel actions by the user in many applications. In web applications, the `onClick` property is used to define a function that will be executed when the component is clicked. Because of the important functionality of the buttons, it is often essential to include tests to verify the outcome when clicking on them. A familiar method to test the outcome is to create a mock function that will spy on the event, then click the button and confirm whether the spy was called or not. In the example application, this was replicated in the test for the add new expense form. Listing 26 shows how the buttons are rendered in the `AddExpenseModal` component.

```
const AddExpenseModal = (props) => {
  const saveExpense = (values) => {
    props.addExpense(values);
    props.setAdded(true);
    props.setOpen(false);
  };
  //...


  return (
    //...
    <button
      style={styles.cancelButton}
      onClick={() => props.setOpen(false)}
    >
      CANCEL
    </button>
    <button
      type='submit'
      style={styles.saveButton}
      onClick={saveExpense}
    >
      SAVE
    </button>
    //...
  );
};
```

Listing 26. Button component of AddExpenseModal.

The code snippet in Listing 26 consists of two buttons, save and cancel. When clicking on the cancel button, it simply calls a function passed through props with the argument `false`, to close the form. When clicking on the save button, it calls the `saveExpense` function, which will then call three other functions that were passed as props: the Redux action's `AddExpense` function to save the expense, the `setAdded` function to notify the parent component a new expense was added and the `setOpen` function to close the modal. Each button was tested thoroughly in Listing 27 and 28, respectively.

The testing of the save button begins by mocking the related functions, with slightly different approaches. For the Redux action, `jest.mock()` was used and the actual function will not be included in the test. The reason for that, according to the Redux official page, is that "the Redux Toolkit maintainers have already done that" and one should not test the actions by themselves [30]. The other two functions were mocked using `jest.fn()`, so they can be spied on later in the test. The test then renders the component and passes in the necessary functions. Once the arrangement is done, the test proceeds to fill out the form before clicking the save button. After each action to select an option from the dropdown list, the function `waitForElementToBeRemoved()` was called to wait for the state updates to finish. This is because Formik's hooks were used to handle the form data through state. The final steps are clicking the save button and asserting that the spy functions have been called with the correct parameters. In the test, the first assertion was used in combination with the `waitFor()` callback to query for the function call, which only happens after the button was clicked. The second assertion, however, does not need to be wrapped in `waitFor()`, because only one call is needed for the UI to settle to the expected state.

The test for the cancel button is shown in Listing 28. This test is simpler, only the `setOpen` function needs to be mocked and spied to be called with `false`, in other words, the modal will close if cancel is clicked. No mocks are needed for Redux's `addExpense` action, as well as the `setAdded` function, as they will not be used.

```
jest.mock('redux/actions', () => ({
  addExpense: () => ({
    type: 'TESTING',
  }),
}));
//...
describe('clicking the save button', () => {
  it('saves the information correctly', async () => {
    const mockSetOpen = jest.fn();
    const mockSetAdded = jest.fn();
    render(
      <AddExpenseModal setOpen={mockSetOpen} setAdded={mockSetAdded} />,
      {
        preloadedState: { categoriesData: initialCategoriesData },
      }
    );
    await user.type(
      screen.getByRole('textbox', { name: 'Title*' }),
      'test'
    );
    await user.type(
      screen.getByRole('spinbutton', { name: 'Total*' }),
      '20'
    );
    user.click(screen.getByLabelText('Category*'));
    user.click(
      await screen.findByRole('option', {
        name: initialCategoriesData[0].name,
      })
    );
    await waitForElementToBeRemoved(() => screen.queryAllByRole('option'));
    user.click(screen.getByLabelText('Status*'));
    user.click(await screen.findByRole('option', { name: 'Confirm' }));
    await waitForElementToBeRemoved(() => screen.queryAllByRole('option'));
    user.click(screen.getByRole('button', { name: 'SAVE' }));
    await waitFor(() => {
      expect(mockSetAdded).toHaveBeenCalledWith(true);
    });
    expect(mockSetOpen).toHaveBeenCalledWith(false);
    });
  });
});
```

Listing 27. Clicking the save button test.

```
describe('clicking the cancel button', () => {
  it('closes the modal', async () => {
    const mockSetOpen = jest.fn();
    render(<AddExpenseModal setOpen={mockSetOpen} />, {
      preloadedState: { categoriesData: initialCategoriesData },
    });

    user.click(screen.getByRole('button', { name: 'CANCEL' }));

    await waitFor(() => {
      expect(mockSetOpen).toHaveBeenCalledWith(false);
    });
  });
});
```

Listing 28. Clicking the cancel button test.

Even though the principle of the React Testing Library is to test what the user sees in the application, sometimes it is also viable to test what happens internally. For example, there are occasions where a user action in one component will lead to a UI change in another component. In such cases, rather than having to import and monitor the changes of the actual component, a simple way to test that user action is by monitoring the events of the controlled component, as explained in the tests above.

5.8 Testing Expected User Behaviour Involving Several Components

In a React application, smaller components are usually defined separately, then used together as the children of a "container" component. A container component normally represents a web page and it is responsible for the rendering and logic of the web page. Because of the important nature of containers, their tests are composed to make sure everything is rendered correctly and interactions between components are functional. In the example application, the most significant container is the `<Home />` component, which consists of many smaller components. Listing 29 shows how they are rendered, with a focus on how the `<AddExpenseModal />` component is handled.

```
//...
const Home = (props) => {
  const [open, setOpen] = useState(false);
  const [added, setAdded] = useState(false);
  //...

  return (
    <div style={{ position: 'relative' }}>
      //...
      <button
        style={styles.addButton}
        onClick={() => setOpen(true)}
        data-testid='add-button'
      >
        <FontAwesomeIcon icon={faPlus} color={COLORS.white} size='2xl' />
      </button>
      //...

      {open && <AddExpenseModal setOpen={setOpen} setAdded={setAdded} />}
      <Snackbar
        open={added}
        autoHideDuration={4000}
        onClose={() => setAdded(false)}
        anchorOrigin={{ vertical: 'bottom', horizontal: 'right' }}
      >
```

```
        <Alert
          onClose={() => setAdded(false)}
          severity='success'
          sx={{ width: '100%' }}
        >
          Added a new expense!
        </Alert>
      </Snackbar>
    </div>
  );
};
```

Listing 29. Home component rendering.

Listing 29 shows that when the user clicks the "+" button on the home page, the `open` state will be set to `true` and the modal will appear. Furthermore, as shown in the `AddExpenseModal` test in Listing 27, when an expense has been created, the `added` state will also be set to `true`, resulting in the `Alert` banner appearing. This action involves many state updates and UI changes, so it is a minimum requirement to have a test for its "happy path", which is successfully adding an expense, reproduced in Listing 30.

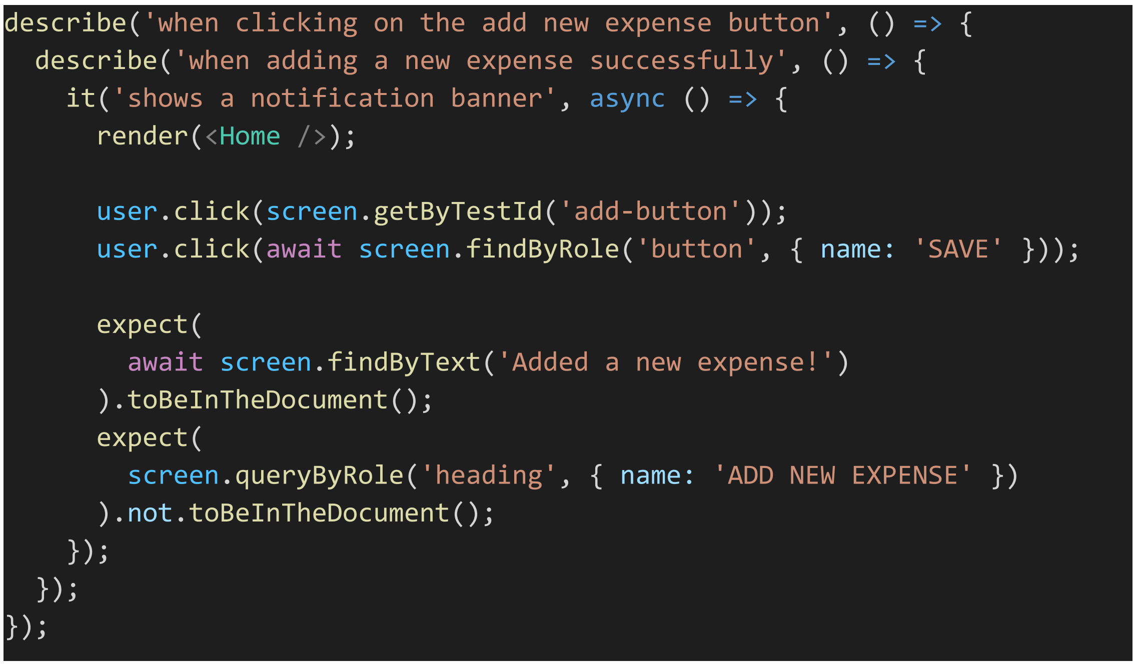
It is important to test out the happy path of the component. However, as this is unit testing, fully rendering the child component is not a requirement. If the full workflow for the expense adder form needs to be tested, it is more suitable to do it in an integration test than in a unit test. Full rendering will be useful whenever the child components need to be tested in combination with the parent. But this is often not realistic, as theoretically, the children should have already been tested separately. As discussed before, RTL only offers full rendering, as opposed to Enzyme which offers both shallow and full rendering. This is a benefit over the latter library, but it is also a drawback as the test runtime will increase. To prevent it from happening, Jest's mock function can be used to turn complex child components into simpler ones, as demonstrated in Listing 27 through the `jest.mock('expense-adder/AddExpenseModal')` function.

Because the purpose is to test the happy path of the Home component, only the save button functions `setAdded` and `setOpen` and the “ADD NEW EXPENSE” header are needed. Other functionalities were removed for simplicity, such as the form and its handler. After mocking the component, the test continues by clicking on the button that has the test id “add-button”, waiting for the form to appear and clicking the save button. Finally, the appearance of the banner is asserted by finding the text “Added a new expense!” and making sure the form is not present anymore by searching for its header, “ADD NEW EXPENSE”. Figure 17 shows that filling out the form is not required anymore and the test passes without a problem.

```
jest.mock('expense-adder/AddExpenseModal', () => ({ setAdded, setOpen })
=> {
  const mockAdd = () => {
    setAdded(true);
    setOpen(false);
  };

  return (
    <div>
      <h2>ADD NEW EXPENSE</h2>
      <button onClick={mockAdd}>SAVE</button>
    </div>
  );
});

//...
```

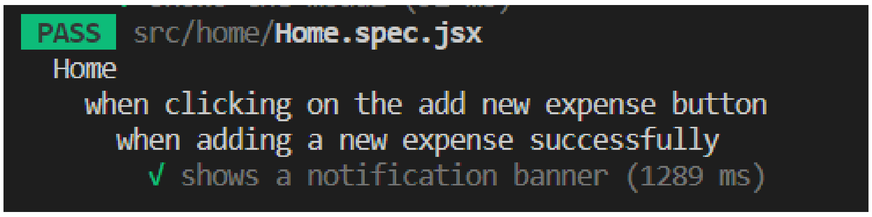


```
describe('when clicking on the add new expense button', () => {
  describe('when adding a new expense successfully', () => {
    it('shows a notification banner', async () => {
      render(<Home />);

      user.click(screen.getByTestId('add-button'));
      user.click(await screen.findByRole('button', { name: 'SAVE' }));

      expect(
        await screen.findByText('Added a new expense!')
      ).toBeInTheDocument();
      expect(
        screen.queryByRole('heading', { name: 'ADD NEW EXPENSE' })
      ).not.toBeInTheDocument();
    });
  });
});
```

Listing 30. Test for adding expenses in the Home component.



```
 PASS  src/home/Home.spec.jsx
  Home
    when clicking on the add new expense button
      when adding a new expense successfully
        √ shows a notification banner (1289 ms)
```

Figure 17. Adding new expense test result.

6 REACT TESTING LIBRARY BEST PRACTICES

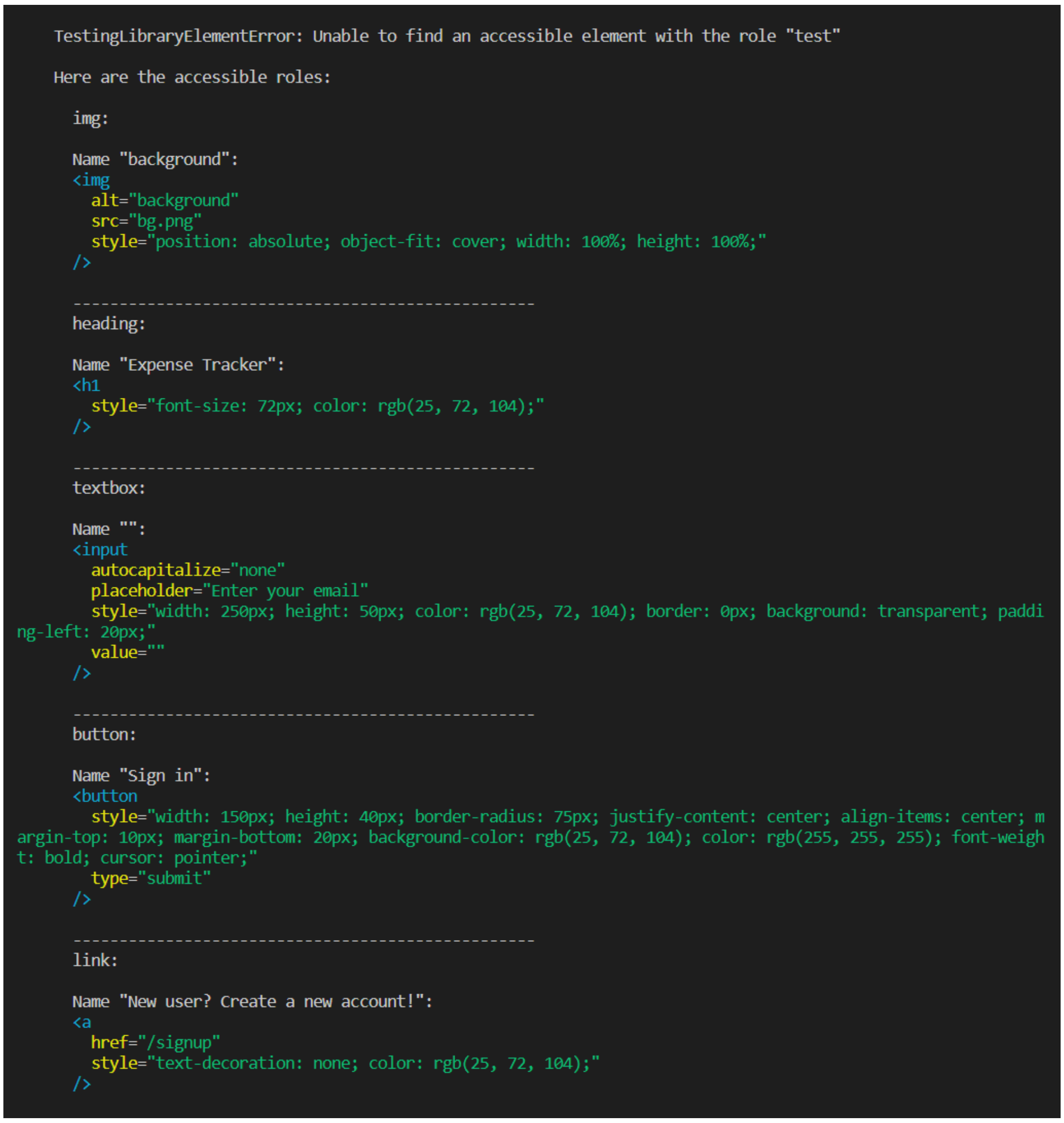
React Testing Library, together with Jest, offers a variety of functionalities that can be utilized in tests. However, assuring that the tests run efficiently and without side effects can be difficult sometimes. Fortunately, common mistakes are avoidable with the Testing Library ESLint plugin that comes built-in when installing applications with `create-react-app`. The plugin is an extension of the base ESLint, a static code analysis tool for identifying problematic patterns found in JavaScript code. It follows a configurable rules list and warns developers when their coding style or code quality does not comply with the specifications, therefore, it will save developers a lot of time not having to monitor their code manually.

As mentioned several times in the previous chapter, when using RTL, every time the `render()` method is called, it will mount the component and all its children into the DOM and that will increase the test run time. This can be handled by grouping related assertions together into one test case to reduce the number of calls for the `render()` method. Furthermore, if a component contains many child components that do not require testing, those child components can be simplified by turning them into simple components using `jest.mock()`. Although having too many re-renders that render everything might not significantly increase the runtime of small components, it will become more noticeable the more the component grows, so having good practices from the start will be valuable eventually.

Before RTL version 6.11.0, developers had to destructure the `render` call to get the queries for their tests. However, version 6.11.0 introduced the `screen` object containing every query that is pre-bound to `document.body`. Developers can now use the `screen` object, followed by the needed query, without having to pay attention to the `render` call. In addition, the `screen.debug()` method can be used to log the document, an element, or an array of elements to the console, which can help to understand the component tree in order to test it. The convenience of using `screen` helps when testing and it is common practice to use it instead of destructuring the queries to use them.

Query priorities are also very important in RTL and knowing which one to use can sometimes be confusing for people new to the library. Different query variants have different purposes in the tests, but the default and first query that should be used is `get`. Other than `get`, the query variant `find` is used when the tested element might not be available immediately, because it will return a promise until the element is found, or until the time runs out and the element is not found. The `query` variant of the queries, on the other hand, is exposed only so testers will have a function that does not throw an error if no element is found. Its only purpose is to verify that an element is not rendered to the page, as it will return `null` if no element is found, while the other two queries will throw a detailed error log together with the whole document for debugging. Therefore, `query` should not be used for anything except checking for non-existence.

Accompanying the different query types is a variety of options, so it is important to understand when and how to use the queries to develop best practices in our tests. As a general rule, the element should be queried by text first before relying on test IDs, class names, or other mechanisms. Moreover, the option, `...ByRole` should be used whenever possible, as outlined in the priority guidelines of the library [25]. The queries come with a list of options that can be used to find the element that needs testing. For example, the `name` option allows us to query the elements by their accessible name, which is what the screen readers will see. In cases where the role of the element under test is unclear, the query can be used with a random option, like `screen.getByRole('test')`, which will make the test fail and log all the available roles. Figure 18 is an example log of all the available roles in the `<Login />` component.



```
TestingLibraryElementError: Unable to find an accessible element with the role "test"

Here are the accessible roles:

  img:

  Name "background":
  <img
    alt="background"
    src="bg.png"
    style="position: absolute; object-fit: cover; width: 100%; height: 100%;"
  />

  --------------------------------------------------
  heading:

  Name "Expense Tracker":
  <h1
    style="font-size: 72px; color: rgb(25, 72, 104);"
  />

  --------------------------------------------------
  textbox:

  Name "":
  <input
    autocapitalize="none"
    placeholder="Enter your email"
    style="width: 250px; height: 50px; color: rgb(25, 72, 104); border: 0px; background: transparent; paddi
ng-left: 20px;"
    value=""
  />

  --------------------------------------------------
  button:

  Name "Sign in":
  <button
    style="width: 150px; height: 40px; border-radius: 75px; justify-content: center; align-items: center; m
argin-top: 10px; margin-bottom: 20px; background-color: rgb(25, 72, 104); color: rgb(255, 255, 255); font-weigh
t: bold; cursor: pointer;"
    type="submit"
  />

  --------------------------------------------------
  link:

  Name "New user? Create a new account!":
  <a
    href="/signup"
    style="text-decoration: none; color: rgb(25, 72, 104);"
  />
```

Figure 18. getByRole result.

The `waitFor()` function is very convenient when performing tests that have asynchronous requests or involve actions through controlled components. The purpose of the function is to wait for a specific thing to happen, so there are a few things to avoid performing inside it. The first thing to avoid is passing an empty callback to it, causing a fragile test that could fail if the logic the refactored. Secondly, having multiple assertions in a `waitFor()` callback can increase the execution time if one of the assertions fails and the test has to wait for the program to run through each of them. Instead, if only one assertion is put inside the `waitFor()` callback and the others are asserted outside of the callback, the test will wait for the UI to change to the state expected and reduce the run time if one of the assertions fails. Performing user actions or side effects in `waitFor()` is also not recommended. Because the function is designed to check for things that will happen after a non-deterministic amount of time, it will be called several times before the assertion passes or fails. Therefore, any user actions inside the function will also be executed; in some cases, this can cause the test to break. One last thing to note is that even though `waitFor()` is intended to wait for a specific result after an amount of time, the appearance of the elements can be checked using the `find` query instead. This is because the query uses `waitFor()` in its implementation, is simpler to use and produces better error messages.

A warning that testers will commonly come across is the "... not wrapped in act()" warning, which usually occurs when there is a change to the state of a component that was not expected. By design, React expects interactions with a component to be wrapped in `act()`, so it will know that some updates are expected from that component. The warning is beneficial as it helps us detect and avoid unintentional updates, however, most of the utilities offered by RTL are wrapped in `act()` automatically, so it is not always necessary to put the actions inside of `act()`. Instead, if the state update was intentional, it can be fixed most of the time by using `waitForElementToBeRemoved()` or `waitFor()` after performing the action that changes the state. This will tell React that the change is expected. Nevertheless, there will be cases where RTL does not have predefined functions and `act` has to be used, such as when testing custom hooks or using Jest's fake timers to test an interval [31]. Because React applications revolve around states, it is possible to encounter the "not wrapped in act" warning when writing our tests, so it is important to understand the source of the problem and be able to fix it.

7 CONCLUSIONS

The React Testing Library is a great package for testing React Apps. Its testing principle encourages minimizing testing implementation details and rather focus on testing the application in a way that represents how users interact with it. To achieve this guideline, the library provides users with the ability to render components into the DOM, utilities that facilitate querying the DOM nodes and access to the `jest-dom` matchers. Verifying that the application will work for the user as intended without worrying about its implementation details can help reduce the time and effort needed to put into writing tests.

This thesis briefly compared two of the most popular testing libraries on the market, Enzyme and React Testing Library. Although Enzyme provides users with more helper functions than RTL, most of the library's utilities facilitate testing implementation details and will not reflect how real users use the application. On the contrary, RTL prioritizes finding and interacting with elements by their visible texts, which is what the users see and interact with. However, the Enzyme library provides a shallow render method that is very useful when performing unit testing in higher-level components, while RTL requires Jest's mock function to help with shallow rendering, which usually results in a longer test execution time.

The examples in this thesis work focused on unit testing without mentioning integration and end-to-end testing. With RTL, writing integration tests is usually the same as unit tests. The only difference expected is the workflow involving several components. For E2E testing, the process is straightforward and there are tools more suitable for it than RTL, like Cypress and Selenium.

Testing the example application demonstrated the versatility that RTL can offer combined with the Jest test runner. There are various ways to test a component or a functionality, but knowing which to choose for the most efficient outcome is not always clear. The thesis outlined some of the best practices that can be used when testing, but it requires time and experience to be familiar with the library.

REFERENCES

[1] What is software testing? Accessed 07.11.2022. https://www.ibm.com/topics/software-testing

[2] Hamilton, T. 2022. Manual Testing. Accessed 07.11.2022. https://www.guru99.com/manual-testing.html

[3] Hamilton, T. 2022. Automation testing. Accessed 07.11.2022. https://www.guru99.com/automation-testing.html

[4] Hamilton, T. 2022. Functional testing. Accessed 07.11.2022. https://www.guru99.com/functional-testing.html

[5] Rajkumar. 2022. Unit Testing Guide. Accessed 08.11.2022. https://www.softwaretestingmaterial.com/unit-testing/

[6] Java T point. Integration testing. Accessed 08.11.2022. https://www.javatpoint.com/integration-testing

[7] Hamilton, T. 2022. What is System Testing? Accessed 08.11.2022. https://www.guru99.com/system-testing.html

[8] Software Testing Fundamentals. Acceptance testing. Accessed 08.11.2022. https://softwaretestingfundamentals.com/acceptance-testing

[9] ReQtest. Differences between the levels & types of testing. Accessed 08.11.2022. https://reqtest.com/testing-blog/different-levels-of-testing/

[10] Vocke, H. 2018. The Practical test pyramid. Accessed 08.11.2022. https://martinfowler.com/articles/practical-test-pyramid.html

[11] Software Testing Fundamentals. Unit testing. Accessed 10.11.2022. https://softwaretestingfundamentals.com/unit-testing/

[12] Khorikov, V. 2020. Unit Testing Principles, Practices, and Patterns. 1st ed. Shelter Island, NY. Manning Publications Co.

[13] Hamilton, T. 2022. What is User Acceptance Testing (UAT)? Accessed 11.11.2022. https://www.guru99.com/user-acceptance-testing.html

[14] Npm trends. @angular/core vs react vs vue. Accessed 14.11.2022. https://npmtrends.com/@angular/core-vs-react-vs-vue

[15] Java T point. Pros and Cons of React JS. Accessed 14.11.2022. https://www.javatpoint.com/pros-and-cons-of-react

[16] Stack overflow. Stack Overflow Trends. Accessed 14.11.2022. https://insights.stackoverflow.com/trends?tags=reactjs%2Cvue.js%2Cangular%2Csvelte

[17] Thomas, M. 2018. React in Action. 1st ed. Shelter Island, NY. Manning Publications Co.

[18] Abiodun, D. A. 2021. React Class Component vs. Functional Component: What's the Difference. Accessed 15.11.2022. https://www.telerik.com/blogs/react-class-component-vs-functional-component-how-choose-whats-difference

[19] Larsen, J. 2021. React Hooks in Action. 1st ed. Shelter Island, NY. Manning Publications Co.

[20] React. Testing overview. Accessed 17.11.2022. https://reactjs.org/docs/testing.html

[21] Java T point. Jest Framework. Accessed 18.11.2022. https://www.javatpoint.com/jest-framework

[22] Begum, G. 2022. A Practical Guide to Testing React Applications [React Testing Tutorial]. Accessed 18.11.2022. https://www.lambdatest.com/blog/react-testing-tutorial/

[23] Jest. Globals.. Accessed 21.11.2022. https://jestjs.io/docs/api

[24] Jest. Getting started. Accessed 22.11.2022. https://jestjs.io/docs/getting-started

[25] React Testing Library. Introduction. Accessed 22.11.2022. https://testing-library.com/docs/

[26] Npm trends. @testing-library/react vs enzyme. Accessed 28.11.2022. https://npmtrends.com/@testing-library/react-vs-enzyme

[27] Mike, C. 2009. Succeeding with Agile: Software Development Using Scrum. 1st ed. Addison-Wesley Professional.

[28] Dodds, Kent C. 2020. Testing Implementation Details. Accessed 29.11.2022. https://kentcdodds.com/blog/testing-implementation-details

[29] Stevenson, D. 2018. What is Firebase? The complete story, abridged. Accessed 16.12.2022. https://medium.com/firebase-developers/what-is-firebase-the-complete-story-abridged-bcc730c5f2c0

[30] Redux. Writing Tests. Accessed 05.01.2023.
https://redux.js.org/usage/writing-tests

[31] Dodds, Kent C. 2020. Fix the "not wrapped in act(...)" warning. Accessed 20.01.2023.
https://kentcdodds.com/blog/fix-the-not-wrapped-in-act-warning

APPENDICES

APPENDIX 1. Expense Tracker application code base

https://github.com/MaiTrn/expense-tracker