VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
INFTITUTE OF COMPUTER SCIENCE
SOFTWARE ENGINEERING STUDY PROGRAMME

Automating the Creation of React Component Tests

React komponentų testų kūrimo automatizavimas

Bachelor's thesis

Author: Tomas Zaicevas (parašas)
Supervisor: lekt. Aurimas Šimkus (parašas)
Reviewer: lekt. Linas Butėnas (parašas)

Vilnius – 2021

Santrauka

Šiame darbe aprašoma kaip buvo kuriamas įrankis, gebantis automatizuoti React komponentų testų kūrimą. Apžvelgtos aktualios technologijos: JavaScript ir React biblioteka, bei įrankiai, naudojami šių technologijų testavimui. Išanalizuoti esami įrankiai, gebantys automatizuoti React komponentų testų kūrimą.

Nuspręsta, kad atviro kodo web aplikacija "Testing Playground" gali būti panaudota tam, kad būtų automatizuoti tam tikri uždaviniai kuriant React komponentų testus. Web aplikacija buvo išplėsta funkcionalumu ir galėtų būti toliau plečiama.

Raktiniai žodžiai: Automatinis testavimas, Programinės įrangos testavimas, Web testavimas, ReactJS bibliotekos testavimas

Summary

This paper describes creation of a tool, capable of automating the creation of React component tests. Relevant technologies were researched and analyzed: JavaScript and React library, together with tools that are being used to write tests for those technologies. Following that, existing tools, which are capable of automating the creation of React component were researched.

It was determined that an open-source web application ”Testing Playground” can be used to automate certain tasks of creating React component tests. The web application was extended with additional functionality and could be extended even further.

Keywords: Automated testing, Software testing, Web testing, ReactJS testing

CONTENTS

INTRODUCTION 4

1 ANALYSIS OF JAVASCRIPT 6

2 REACT'S ARCHITECTURE 9
- 2.1 Components 9
- 2.2 Lifecycle 10
- 2.3 Virtual DOM and reconciliation 12
- 2.4 useReducer hook 13
- 2.5 Summary 14

3 ANALYSIS OF TESTING TOOLS 15
- 3.1 Testing frameworks 15
- 3.2 react-test-renderer 16
- 3.3 React testing utilities 17
- 3.4 react-testing-library 19
- 3.5 Summary 21

4 PROPOSED TOOL'S ARCHITECTURE 22
- 4.1 Automatable tasks 22
- 4.2 Existing tools 22
- 4.3 Technical considerations 23

5 IMPLEMENTATION OF THE TOOL 24
- 5.1 Graphical User Interface 24
- 5.2 Filling the test based on user selections 26
- 5.3 Executing tests 27
- 5.4 End result 29

RESULTS AND CONCLUSIONS 30

REFERENCES 31

APPENDICES 32
- 1 appendix. Extended Testing Playground 33

Introduction

Web pages are becoming more and more interactive and complex. Hence, it is common to use JavaScript libraries and frameworks that not only help with code readability and maintainability, but also simplify the development of smooth user experience.

One of such libraries is ReactJS, or simply - React. React is a "JavaScript library for building user interfaces."[Facd] It is maintained by Facebook, as well as an open-source community. At the time of writing this paper, React is being used in at least 5.3 million Github projects[Faca]. This figure is larger than that of React's alternatives, such as Angular[Goo] and Vue[You], making it one of the most popular JavaScript libraries.

Even when using React, complexity and users' demand not only make the development more difficult, but it also becomes more difficult to ensure that no regressions occurred after a change in the code. There are multiple methods to gain confidence that the shipped software fulfils its' requirements. Those methods include manual testing, performance testing, automated testing.

Manual testing has an obvious downside: it does not scale well with growing codebase and code complexity. For that reason, it is common to rely on automated tests, such as unit, integration, end-to-end, visual tests.

The problem is that writing those tests might be as difficult as writing the web application itself. In fact, software testing might take as much as 80% of the total development cost[Bei90]. Another study shows that "waiting for tests and fixing long-festering regression errors accounted for 10% and 15% of total development time."[SE03] Therefore, it is crucial to make the development experience of writing automated tests as smooth as possible. That could mean how easy it is to write new tests, as well as maintain existing ones.

There are many technologies that can be used to run, debug, write them, make them more readable and user-like. React is not an exception and its' ecosystem provides many different tools, such as $react - test - renderer$, $react - testing - library$, $enzyme$. However, as mentioned before, each tool comes with its' own conventions, rules, difficulties. These rules might be implicit - not abiding them does not throw an error, which can lead to less than ideal tests. Automating such rules with linters or by other means could improve the quality of tests.

Even when using tools and libraries, writing tests takes a certain degree of manual work, such as inspecting and analyzing DOM. Ideally, the manual work could be automated, which would provide two distinct advantages, both of which contribute to development experience and, ultimately, quality of software:

- Make the process of writing tests easier and faster.
- Improve the quality of tests.

The relevance of this paper is based on the fact that while there are tools that automate the creation of tests, such as "Try Playwright" or "Cypress Recorder", most of them are focused on end-to-end tests. This paper focuses on React component tests, also referred to as unit and integration tests. Component tests are capable of rendering a React component, executing DOM-like events, such as clicking on an element and asserting the DOM. Those tests usually run faster and are easier to setup than end-to-end tests.

The goal of this thesis is to create a tool, which automates the creation of React component tests.

To achieve this goal, **the following tasks** will need to be solved:

- Analyze React's architecture.
- Overview tools that are being used to write React component tests.
- Analyze existing tools that are automating the cretion of React component tests.
- Define which parts of test creation, that are not automated by existing tools, could be automated.
- Define and create a tool, which automates tasks, which are unautomated by existing tools.

1 Analysis of JavaScript

The focus of this section is to analyze JavaScript's event loop and DOM event model. It is important, because this paper is centered around React component tests, which implies that they are written with JavaScript. Furthermore, it might be convenient to create the tool using WEB technologies.

JavaScript has a concurrency model, which is based on a single thread[Moza]. This means that actions, which are asynchronous by nature, such as IO, cannot be run in a different thread and thus not block the main thread. It is especially important when JavaScript is being executed in a browser, because UI code is being executed on the main thread. However, JavaScript has a technique, commonly called the event loop, which can be leveraged to make IO and other asynchronous operations not block UI operations.

The event loop means that JavaScript holds a couple of data structures, which store information about which code needs to be executed next. Synchronous operations are prioritized, while asynchronous code, such as callbacks after IO, is executed only after synchronous code. This means that heavy synchronous computations might take a lot of time, so that the asynchronous callbacks are never executed. It is also important in the tests, to make sure the sequence of React code and test code is correct.

JavaScript runtimes hold 3 data structures[Mozc]:

- Call stack.
- Task queue.
- Microtask queue.

Call stack is FILO (First In Last Out) data structure, which holds the prioritized callbacks. Only after the call stack is empty, other data structures (task queue, microtask queue) are accounted for.

Task queue is a FIFO (First In First Out) data structure, which holds asynchronous callbacks. For instance, a callback, scheduled with $setTimeout$, will be put into the task queue and executed when the call stack is empty.

Similarly, microtask queue is also a FIFO data structure, which holds asynchronous callbacks, but has priority over the task queue. Thus, callbacks in the task queue are executed only after the microtask queue is empty.

The listing below illustrates this concept. After executing function foo, 1, 2 and 3 are printed in the console.

```
const foo = () => {
    setTimeout(() => console.log('3'), 0);
    new Promise(resolve => resolve()).then(x => console.log('2'));
    console.log('1');
}
```

Listing 1: Event loop

Figure 1 shows which data structure the callbacks belong to, assuming the call stack grows upwards.

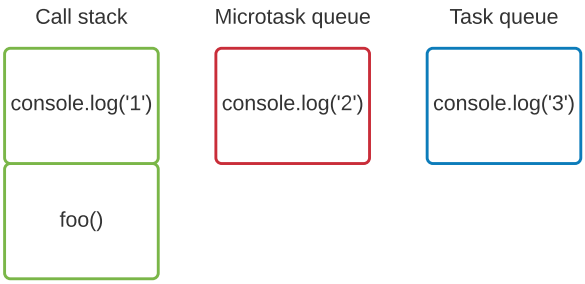

Figure 1. Event loop

It is interesting to note that different browsers might handle the callbacks differently[Arc]. Therefore, in some cases the same code might be executed in different sequences, depending on the browser.

DOM events is another important section of JavaScript programming. It is especially relevant when talking about web tests and, specifically, the automation of creating those tests. That is because those tests should be about user interactions, which implies simulating DOM events that happen in the browser. Ideally, to reduce manual work when creating tests, user could be able to select an element and a suggestion on how to query the element would be handled by a tool.

DOM events are usually fired in response to user's actions, such as scrolling the browser or clicking on an element. Scripts are able to register event handlers and execute arbitrary code when the events are fired. A significant topic of DOM event model is bubbling and capturing[Mozb]. When an event is fired on a DOM element, it is also propagated to element's ancestors. Capturing refers to event handlers being called from the outer-most ancestor all the way down to the target element, on which the event is triggered. While bubbling refers to event handlers being called from the target element up to the outer-most ancestor. This behaviour can be manipulated by methods like $stopPropagation()$ and $stopImmediatePropagation()$. Bubbling and capturing is illustrated in Figure 2. Blue node, which is a child of $< section >$ element, denotes the target element, on which the event was initially fired.

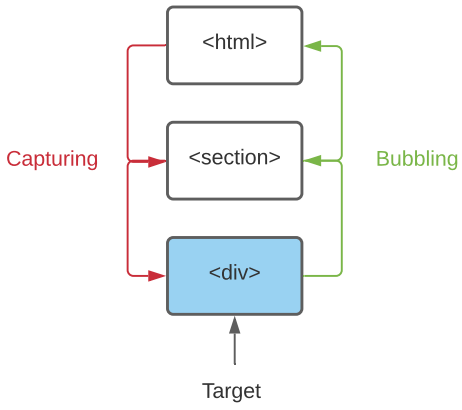

Figure 2. Capturing and bubbling

This model raises questions about how the tool, which will automate the creation of React component tests, will have to be implemented. If the user will be able to select a DOM element, see a suggested query and then execute the test, the event handlers will be executed before the test is run. It might result in a state of DOM that is not desired prior to running the test. Additionally, if the test is going to be executed in the browser, events fired by the executed test, will have to be differentiated from those fired by the user. It is likely that these issues will have to be dealt with, if the tool is going to be implemented as a web application.

With JavaScript's concurrency model and DOM events analyzed, next section will present the analysis of React, which is a JavaScript library for building user interfaces.

2 React's architecture

The purpose of this chapter is to examine parts of React that are important, in order to automate the creation of React component tests. In Section 2.1 the notion of React component will be defined, together with their lifecycle in Section 2.2. Whereas in Section 2.3 the process of updating the DOM is explained in more depth.

As expressed in the introduction, React is a "JavaScript library for building user interfaces."[Facd], meaning it provides an API to help managing DOM changes. A few of React's characteristics are[Facd; Per20]:

- Declarative. React handles all DOM manipulations.
- Component-based. Component is the primary unit of encapsulation.

Therefore, it is important to understand what constitutes a component.

2.1 Components

Components are units of encapsulation that can manage their own state[Facd]. They can then be composed to make complex UIs[Facd].

In JavaScript terms, components can either be defined as classes or functions. The examples, provided in this thesis, will be in functions, because they are recommended by the React community[Facb].

A simple example of a component is shown in Listing 2.

```
const HelloWorld = () => {
  return <div>Hello world!</div>;
}
```

Listing 2: Hello World React component

It is essential to understand that components don't return HTML. In fact, they have to return a **React element**. React element is a JavaScript object, describing the UI. In other words, React elements "are the smallest building blocks of React apps.[Facg]" They can be constructed with $React.createElement()$ function call. What is shown in Listing 2 is a JSX element, which is transformed to $React.createElement()$ function call by a Babel plugin[Bab].

Crucially, components can also have state. For that reason, each component can perform side-effects or change the UI when something happens. For instance, a DOM event, such as $onclick$.

Listing 3 illustrates how state is managed in components. The example corresponds to a component, which renders a number on the screen, starting from zero. Whenever the number is clicked, it is incremented.

```
const Counter = () => {
  const [counter, setCounter] = React.useState(0); // initial value is 0

  return <div onClick={() => setCounter(counter => counter + 1)}>{counter}</div>;
}
```

Listing 3: Stateful Counter component

This component can then be rendered into the DOM, assuming there is a div element with id $root$:

```
ReactDOM.render(<Counter />, document.getElementById('root'));
```

Listing 4: Rendering a component into DOM

As it can be seen, due to how React and components are designed, all of the DOM manipulations are handled by React.

2.2 Lifecycle

With React components defined, it is useful to know how their lifecycle is managed by React.

Components lifecycle could be described in 3 separate events:

1. Mounting.
2. Updating.
3. Unmounting.

The order, as shown in Figure 3, is important.

Figure 3. Component lifecycle events

Mounting refers to the process of rendering component to the DOM for the first time[Fach]. As opposed to **Unmounting**, which is the removal of the DOM nodes, produced by the component[Fach]. **Updating** refers to altering the component's state. For instance, one might want to change the returned React element when a button is clicked.

However, the question is what does React do during each of these events. Clearly, whenever component is mounting or updating, React has to call the component function to get a React element. Otherwise, React would not be aware of what DOM to produce. The lifecycle is depicted in Figure 4, using $Counter$ component as an example.

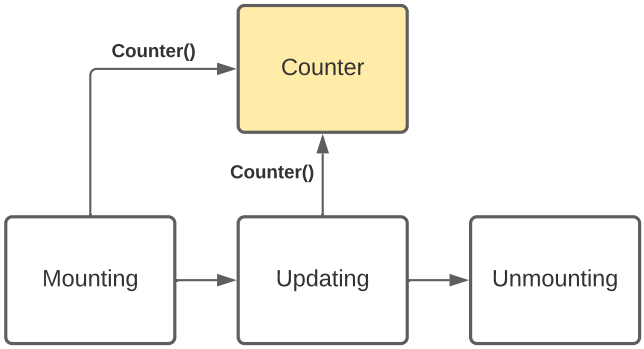

Figure 4. Calling component during mount and update

In order to initiate an update, certain functions have to be called, as portrayed in Listing 3. It is important to outline that React updates the state asynchronously. React is looking for the best timing to update the state, hence it happens nondeterministically. It is a manifestation of React's principles, namely the declarativeness: as a developer, you need to write code, which does not depend on when React updates the state. The exact timing might vary and it is up for React to find the best time to execute the state update. Another important aspect is that React can call the component multiple times for the same update[Facj].

On top of that, React distinguishes 2 phases of producing a DOM representation of a component[Face]:

- Render phase.
- Commit phase.

The purpose of render phase is to calculate the necessary changes to the DOM. Whereas the purpose of commit phase is to perform the DOM changes. Hence, altering DOM will be referred to as **commiting**.

Render phase has to be pure - it cannot have side-effects. However, React has a $useEffect$ hook, which can be used to opt-in into componen's lifecycle and execute side-effects. A common example would be fetching data from a remote API:

```
const Counter = () => {
  const [counter, setCounter] = React.useState(0);

  React.useEffect(() => {
    const fetchData = async () => {
      const response = await fetch(remoteUrl);
      const counterData = await response.json();
      setCounter(counterData.currentNumber);
    };

    fetchData();
  }, [])

  return <div onClick={() => setCounter(counter => counter + 1)}>{counter}</div>;
}
```

Listing 5: Performing side-effects

Listing 5 also illustrates the event loop, analyzed in Section 1.

Render and Commit phases are implemented with a so-called Virtual DOM and a reconciliation algorithm, which will be analyzed in the next subsection.

2.3 Virtual DOM and reconciliation

Virtual DOM (VDOM) is a ”programming concept where an ideal, or “virtual”, representation of a UI is kept in memory and synced with the “real” DOM.”[Facl] Reconciliation is a process of syncing the VDOM with a real DOM[Facl]. The purpose of VDOM and reconciliation is to have as little DOM manipulations as possible.

As described in Section 2.2, every update has a render phase. In this phase, React runs a reconciliation algorithm to compute the minimal necessary part of the DOM that needs to be changed. For instance, when a button component in a complex page becomes disabled, only the underlying $< button >$ DOM node gets changed in the commit phase.

Reconciliation is a O(n) algorithm[Facf], because it is based on a few assumptions:

- Two elements of different types will produce different trees.
- The developer can hint at which child elements may be stable across different renders with a key prop.

Whenever root elements are of different types, React unmounts them and mounts the new tree. This might lead into performance problems, as well as into state management issues, if done carelessly.

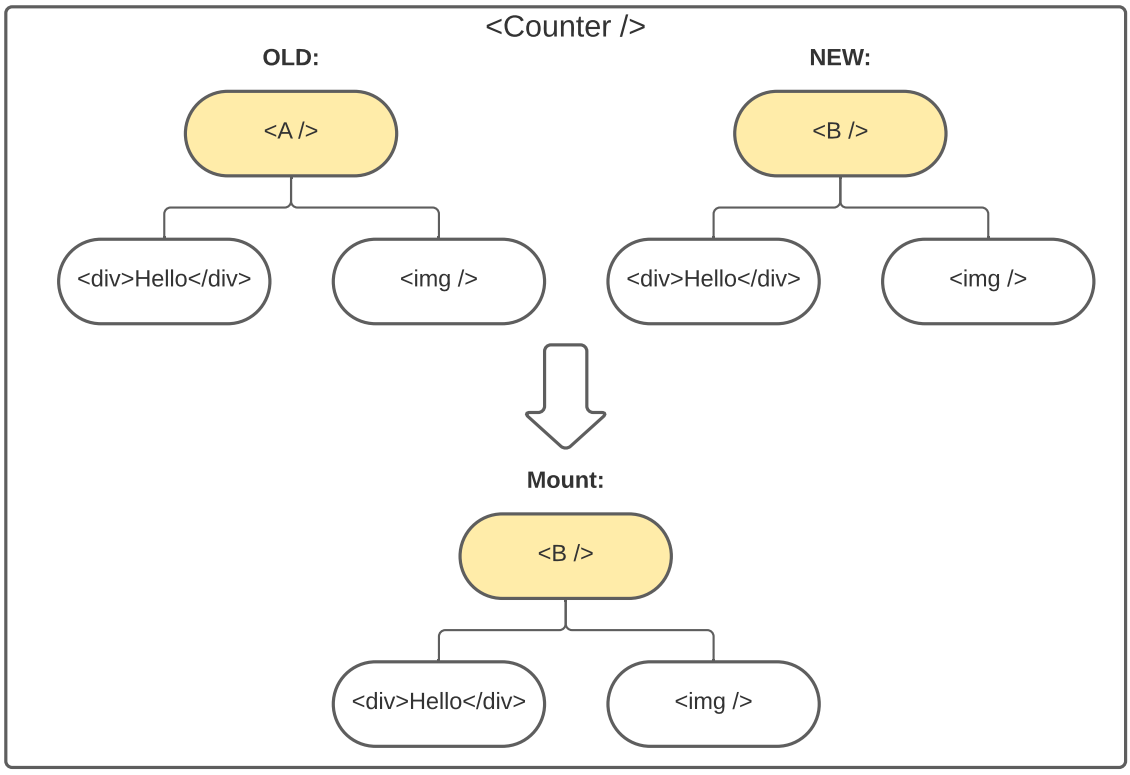

Figure 5. Reconciliation of different root elements

This can be seen in Figure 5. Initially, $Counter$ component renders component A, but then rerenders with root component B. This means that React will unmount component A and mount component B.

Conversely, when React compares DOM elements of the same type, for instance, div with an div, it keeps the underlying DOM, but only updates the changed attributes[Facf]. For React components, the process is repeated until DOM elements are found.

2.4 useReducer hook

Flux is ”the application architecture that Facebook uses for building client-side web applications.”[Facc] React provides a $useReducer$ hook, which is based on the Flux pattern and can be used to manage component's state. For that reason, it is relevant to analyze what is flux and how to use $useReducer$ hook.

The four notions that are central to Flux:

- Action - an object ”containing new data and identifying type property.”[Facc]
- Dispatcher - a mechanism to notify store when an action happens.
- Store - object containing state and logic.
- View - React component.

The data flow is unidirectional, meaning a view sends an action through a dispatcher, which propagates it further to a store, which then updates views. This principle is shown in Figure 6.

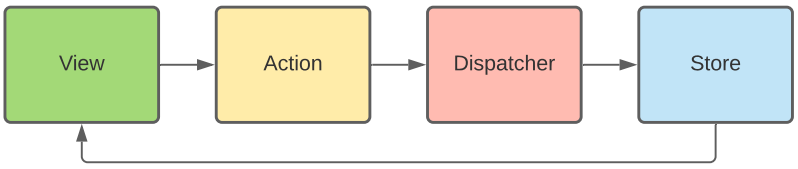

Figure 6. Flux architecture data flow

$useReducer$ adds another step, called a **reducer**. Reducer is a function, which transforms a store, based on a specific action. The modified data flow is shown in Figure 7.

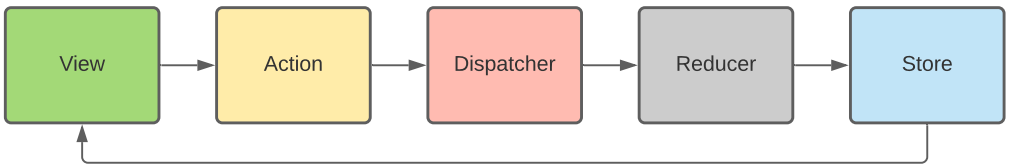

Figure 7. useReducer data flow

A code example is depicted in Listing 6.

```
const initialState = { count: 0 };

const reducer = (state, action) => {
  switch (action.type) {
    case 'add':
      return {count: state.count + 1};
    case 'sub':
      return {count: state.count - 1};
    default:
      state;
  }
}

const Counter = () => {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <>
      <div>{state.count}</div>
      <button onClick={() => dispatch({type: 'add'})}>Add</button>
      <button onClick={() => dispatch({type: 'sub'})}>Sub</button>
    </>
  );
}
```

Listing 6: useReducer counter

2.5 Summary

React is one of the leading libraries, when it comes to building user interfaces with JavaScript. It encourages declarative and component-based style of programming.

Components are units of encapsulation, which compose the UI and have their own lifecycle. The lifecycle is managed by React, but there is a public API to tap into different stages of lifecycle. React notes 2 phases of keeping the DOM up-to-date: render and commit and uses Virtual DOM technique to boost the performance. Additinoally, it supports Flux architecture of state management by exposing $useReducer$ hook.

With React's architecture analyzed, the next section will focus on the analysis of various testing tools, which are used to test React components.

3 Analysis of testing tools

This section will overview tools, which are used to write React component tests.

3.1 Testing frameworks

There are different types of testing software. Although there is no single source of truth when it comes to terminology, tools can usually be somewhat differentiated by their level of abstraction. There are libraries, which operate on the language level. An example could be $react - testing - library$, which gives a set of helper functions, regardless of the environment in which the test is being executed. Then, there are testing frameworks and test runners, which might provide the environment and the fundamental means to write test cases. Those frameworks and test runners can be further expanded into categories, such as continuous test runners, which are capable of integrating with the development environment to rerun tests at real-time[KA14]. On top of that, recent developments are showing progress in testing frameworks that are suited to testing ML systems[Raj21].

There is a number of popular JavaScript testing frameworks: $Mocha$, $Jest$, $Jasmine$. $Jest$ is arguably the most popular JavaScript testing framework. It has close to 12 million weekly downloads, as outlined in the npm registry at the time of writing this section[npm], whereas $Mocha$ has less than 5 million and $Jasmine$ has less than 2 million. A simple $Jest$ test is shown in Listing 7.

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

it('adds two numbers', () => {
  expect(sum(1, 2)).toBe(3);
})
```

Listing 7: Jest test

When testing asynchronous code, $Jest$ provides two solutions:

- The test has to return a promise.
- The test has to call $done()$ function, which comes as an argument.

Both of these variants are represented in Listing 8.

```
// test uses done()
it('adds two numbers', done => {
  const callback = sum => {
    expect(sum).toBe(3)
    done()
  }

  sum(1, 2, callback)
})

// test returns a promise
it('adds two numbers', async () => {
  const sum = await sum(1, 2)
  expect(sum).toBe(3)
})
```

Listing 8: Jest asynchronous test

$Jest$ provides an API to mock a function or a module, as well as spy on it. The example below shows how $Jest$ mocking capabilities can be used to assert that a function has been called.

```
it('calls passed callback', () => {
  const callback = jest.fn()
  doCalculations(callback)
  expect(callback).toHaveBeenCalledTimes(1)
})
```

Listing 9: Jest mocking

$Jest$ uses a $jsdom$ library - a JavaScript implementation of web standards, which can be used with Node.js. Therefore, tests written with $Jest$ don't require a browser to run.

3.2 react-test-renderer

$react - test - renderer$ is an official package, supported by the React community. It can be used to construct JavaScript objects when rendering React components. It also comes with a rich API to query and traverse the constructed tree.

Below is an example of a test, written with $react - test - renderer$ and $Jest$.

```
import TestRenderer from 'react-test-renderer';

const Link = (props) => {
  return <a href={props.page}>{props.children}</a>;
}

it('creates Facebook link', () => {
  const testRenderer = TestRenderer.create(
    <Link page="https://www.facebook.com/">Facebook</Link>
  );

  expect(testRenderer.findByProps({page: "https://www.facebook.com/"}).children).toEqual↩
      (["Facebook"]);
})
```

Listing 10: react-test-renderer with Jest

$react - test - renderer$ encourages white-box testing, because it gives an API to access component's props, children. That is not always desirable, because white-box tests depend on the implementation of a particular React component. It implies that it is harder to maintain such tests, if there are frequent changes in implementation[VKC17]. As was described in introduction, the cost of maintaining and running tests is significant, so it is desirable to make them as maintainable as possible.

Therefore, React documentation recommends using $react-testing-library$, which encourages black-box testing and aims to be closer to how users use React components. Before analyzing $react - testing - library$, it is important to define $act()$ helper, which comes from React test utilities.

3.3 React testing utilities

As described in Section 2, React's state updates are asynchronous. For that reason, executing state updates in tests is not a simple task. As an example, when user clicks on a button, React has to do these steps:

- Call an event handler.
- Run state updates.
- Run $useEffect$.

In the test environment, React has to know when to do those steps. Therefore, it provides $act()$ utility, which hints that the passed callback is supposed to update the state. The first test, depicted in Listing 11, fails, because the rendering part is not wrapped in $act()$ utility, thus React does not know when to run the $useEffect$ lifecycle hook and commit the state update.

```
const Component = () => {
  const [counter, setCounter] = useState(0);
  useEffect(() => {
    setCounter(1);
  }, []);
  return counter;
}

it("this test fails", () => {
  const root = document.createElement("div");

  ReactDOM.render(<Component />, root);

  expect(root.innerHTML).toBe("1");
});

it("this test passes", () => {
  const root = document.createElement("div");

  act(() => {
    ReactDOM.render(<Component />, root);
  });

  expect(root.innerHTML).toBe("1");
});
```

Listing 11: Test with and without act() utility

The reason why the first test fails is illustrated in Figure 8. The state update, which the test expects, happens after the assertion, resulting a failed test. That is because $expect()$ happens before the state is updated.

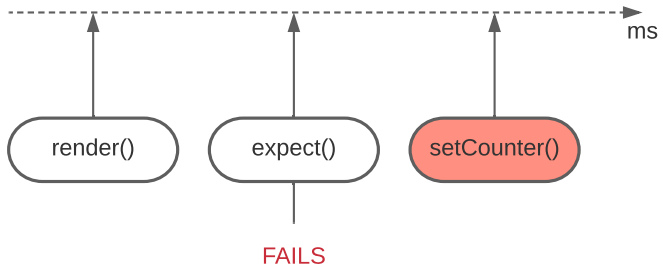

Figure 8. Failing test, due to missing act()

There is also an async version of $act()$ utility. It can be used in cases when state updates are scheduled in a task or microtask queues. For instance, after fetching data from a remote API, as is shown in Listing 5 and Listing 12.

```
it("renders counter after data fetch", () => {
  jest.spyOn(window, "fetch").mockResolvedValue({
    json: async () => ({ currentNumber: 5 }),
  });
  const root = document.createElement("div");

  await act(async () => {
    ReactDOM.render(<Component />, root);
  });

  expect(root.innerHTML).toBe("5");
});
```

Listing 12: Async act()

If a synchronous $act()$ was used, the test would fail, because the assertion would happen before the state update. The assertion would go into the call stack, while the state update would be a part of the microtask queue.

However, one might notice that this involves quite a lot of boilerplate, as every stateful interaction requires it to be wrapped in $act()$, thus cluttering tests. This is one more problem, besides white-box testing, which $react - testing - library$ tries to solve.

3.4 react-testing-library

$React - testing - library$ is a set of utilities, which React documentation recommends using when testing React components[Faci]. It aims to be user-centric and avoid implementation details so that "refactors of your components (changes to implementation but not functionality) don't break your tests."[Dcb] By encouraging user-centric testing, $react - testing - library$ encourages accessibility[Dca].

Additionally, it helps reducing $act()$ calls, as briefly mentioned in Section 3.3. All of the utilities, such as $render()$, are already wrapped in $act()$.

$React - testing - library$ provides 8 types of queries to find an element in the test. The recommended order of priority, based on how much it resembles the way users use the DOM, is the following:

- getByRole,
- getByLabelText,
- getByPlaceholderText,
- getByText,
- getByDisplayValue,
- getByAltText,
- getByTitle,
- getByTestId

A $Jest$ test, written with $react - testing - library$ utilities, is shown in Listing 13.

```
import { render, screen } from "@testing-library/react";

const Component = ({ title }) => {
  return <>{title}</>;
};

it("renders title", () => {
  render(<Component title="Title" />);

  expect(screen.getByText("Title")).toBeInTheDocument();
});
```

Listing 13: react-testing-library test

On top of that, $react-testing-library$ provides async utilities, which can be used to wait for appearance or disappearance. These async utilities fall into three categories:

- findBy queries,
- waitFor,
- waitForElementToBeRemoved

An example with $waitFor$ is depicted in Listing 14.

```
import { useEffect, useState } from "react";
import { render, screen, waitFor } from "@testing-library/react";

const Component = () => {
  const [title, setTitle] = useState("");

  useEffect(() => {
    const fetchData = async () => {
      const response = await fetch(
        "https://jsonplaceholder.typicode.com/posts/1"
      );
      const { title } = await response.json();

      setTitle(title);
    };

    fetchData();
  }, []);

  return <>{title}</>;
};

it("renders fetched title", async () => {
  jest.spyOn(window, "fetch").mockResolvedValue({
    json: async () => ({ title: "Fetched" }),
  });

  render(<Component />);

  await waitFor(() => expect(screen.getByText("Fetched")).toBeInTheDocument());
});
```

Listing 14: react-testing-library async test

In order to simulate user interactions, such as clicking on a button, $react-testing-library$

exposes $userEvent$ and $fireEvent$ modules. For example, $userEvent.click()$ can be used to click on an element, as shown in Listing 15.

```
const Component = () => {
  const [counter, setCounter] = useState(0);
  return <button onClick={() => setCounter(counter => counter + 1)}>{counter}</button>;
}

it("increases counter when clicked", async () => {
  render(<Component />);

  userEvent.click(screen.getByRole("button", {name: "0"}));

  expect(screen.getByRole("button", {name: "1"})).toBeInTheDocument();
});
```

Listing 15: userEvent in react-testing-library test

$react-testing-library$ utilities bring a number of implicit rules, which have to be followed, in order to have the tests reach maximum potential. For instance, depending on the DOM, it might not be possible to use $getByRole$ query, hence one should try to use $getByLabelText$ or another query, based on the priority. Finding the proper query might take a lot of time, even if the developer is an experienced user of $react - testing - library$. At the same time, this could clearly be automated, so that developers do not need to manually search for the prefered way of querrying an element. The same could be done with, async queries, which have to be called with $await$ keyword.

3.5 Summary

To summarize, while there is a number of alternatives, the most popular testing tools in React ecosystem seem to be $Jest$ and $react - testing - library$. React utilities, such as $act()$, are necessary and have to be used as well. Those tools and utilities bring complexity, hence it not only takes a certain amount of manual work to write tests, but it is also difficult to write tests, which adhere to the principles of those testing tools. For instance, when writing $react-testing-library$ tests, one has to manually inspect DOM and search for a prefered query. For those reasons, it seems that the amount of manual work could be reduced by automation and, thus, improve the development experience of using those tools. Those tasks, as well as how they could be automated, will be the focus of Section 4.

4 Proposed tool's architecture

The goal of this section is to define the scope of the proposed tool, as well as how it can be implemented.

4.1 Automatable tasks

From analysis in Section 3, it is clear that there is a certain number of tasks that could be automated, in order to improve development experience of writing tests, as well was suggest how to write better tests.

The following tasks could be automated:

1. Create an it block based on user typed test.
2. Automatically pick a prefered query when hovering over an element, so that the user does not have to manually inspect the DOM.
3. Let the user pick an interaction and automatically convert it into $userEvent$ or $fireEvent$ utility.
4. Let the user pick an assertion, such as text to appear in the document.
5. Build a test case out of interactions and assertions.
6. Run the test in a browser, so that the user can visually see it pass or fail.

If these tasks are automated, developers would have to spend less time writing test code manually. This list is not an attempt to identify all of the tasks that, if automated, would result in developers spending no time writing manual test code. That is because it is difficult to define the needs of all of the systems and all of the developers. However, this list is comprehensive enough, in order to build user-facing test cases and run them without the need for manually writing test code.

4.2 Existing tools

This subsection will explain existing tools, which might be of help, when automating the aforementioned tasks. Only one tool: Testing Playground, was found to be trying to achieve the goal of automating the creation of React component tests. There are somewhat similar tools for end-to-end frameworks, for instance, https://try.playwright.tech/ or "Cypress Recorder"[Kab] chrome extension, but end-to-end frameworks are not the target of this thesis.

Testing Playground is an open-source website, as well as a Chrome, Firefox extension, which helps choosing a prefered testing-library query to find an element. It is accessible at http://testing-playground.com. To use it, one has to paste the DOM and hover over an element. Although this tool does little more than showing the prefered query, it could be expanded to support more features. Apart from displaying the prefered query to find an element, it is capable of parsing JavaScript scripts, meaning that it is possible to include scripts alongside the DOM output. Test code, such as $userEvent$ calls, can also be executed.

On the other hand, React code cannot be parsed. This means that it might be difficult to take code, written with React, and run it within the tool. Especially because React is based on client-side rendering. This implies that the DOM is rendered by a JavaScript, hence the DOM is not available

until React has rendered the component. The Testing Playground is written with JavaScript, React, so it should be relatively easy to extend with the knowledge, explained in previous sections.

Overall, the Testing Playground already automates "Automatically pick a prefered query when overing an element, so that the user does not have to manually inspect the DOM" task. It also has some capabilities to implement "Run the test in a browser, so that the user can visually see it pass or fail", but it does not automate any of the other tasks.

4.3 Technical considerations

The most convenient platform for the proposed tool seem to be a web application, because developers are already using a browser when developing a web app and writing tests for it. Browser extension would not be a bad choice either, but extension APIs are browser-specific, which means it is more difficult to have a coherent solution that works for all browsers. It is also clear that **extending the open-source Testing Playground tool is the best way to move forward** with the proposed tool, because it is capable of suggesting a prefered query when hovering over an element and execute JavaScript code.

The most technically difficult tasks seem to be:

- Making the test visually run in the browser and yield pass/fail result.
- Run test cases, which are context specific. For instance, tests with mocked 3rd party libraries.

The first task is already partially solved, because the Testing Playground is able to run JavaScript code. $userEvent$ calls can be executed on a real browser. However, it is not clear whether $Jest$ can run in the browser. There are tools to use it in conjunction with end-to-end frameworks, such as puppeteer[Fack]. $Mocha$ is an alternative testing framework, which can be used in a real browser. Syntactically, it is somewhat similar to $Jest$. Therefore, it might be possible to convert $Jest$ tests to $Mocha$ tests and then run them in the browser.

Additionally, Testing-library comes with $jest-dom$ library, which is a set of "Custom jest matchers to test the state of the DOM."[Dcc] Therefore, not only $Jest$ assertions and other helpers are necessary, but the $expect$ object, which comes from $Jest$, should ideally be extended with $jest-dom$ matchers. Whether it is possible to run $Jest$ assertions, together with $jest-dom$ matchers in the browser, will be experimented with and described in the next section.

5 Implementation of the tool

This section describes how the tool was implemented and the reasoning behind taken decisions. As it was written in previous section, the Testing Playgroundtool, which is a web application, will be extended with additional functionality, in order to further automate the creation of React component tests. In fact, while extending the Testing Playground tool, it was discovered that a GitHub issue, proposing such features, already exists[pet], but it did not get much attention. The end result of the extended Testing Playground is publicly available in a GitHub fork at Appendix nr. 1.

The Testing Playground heavily relies on Flux architecture and $useReducer$ hook, which was analyzed in Section 2.4. Although it does not directly use $useReducer$, it uses $useEffectReducer$[dav], which is a custom hook, based on $useReducer$ and $useEffect$. For that reason, it was quite difficult to make changes without affecting existing functionality, as a lot of state is global.

5.1 Graphical User Interface

In the header, a ”build test” button was added, which opens a menu with a ”New” entry. The ”New” button could be skipped, relying solely on ”build test” button, but it was decided to have both, because there might be a need to expand this menu in the future.

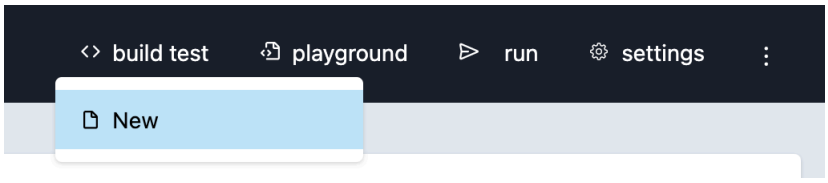

Figure 9. Header

After clicking ”New” user is able to type the test name and proceed with building the test:

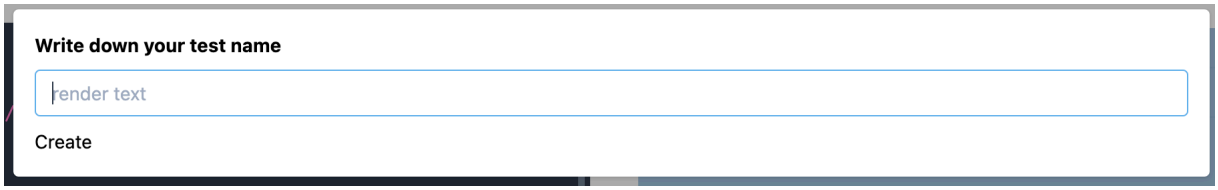

Figure 10. Create test

Testing Playground already has an input element, accessible under ”Query” tab. A new tab, called ”Test”, was added, in which the user is able to build the test case and run it. Adding a ”Test” tab seemed like the easiest and most intuitive solution to implement the building of a test case. Ability to delete the test case was added with a button at the bottom right corner. This functionality is shown in Figure 11.

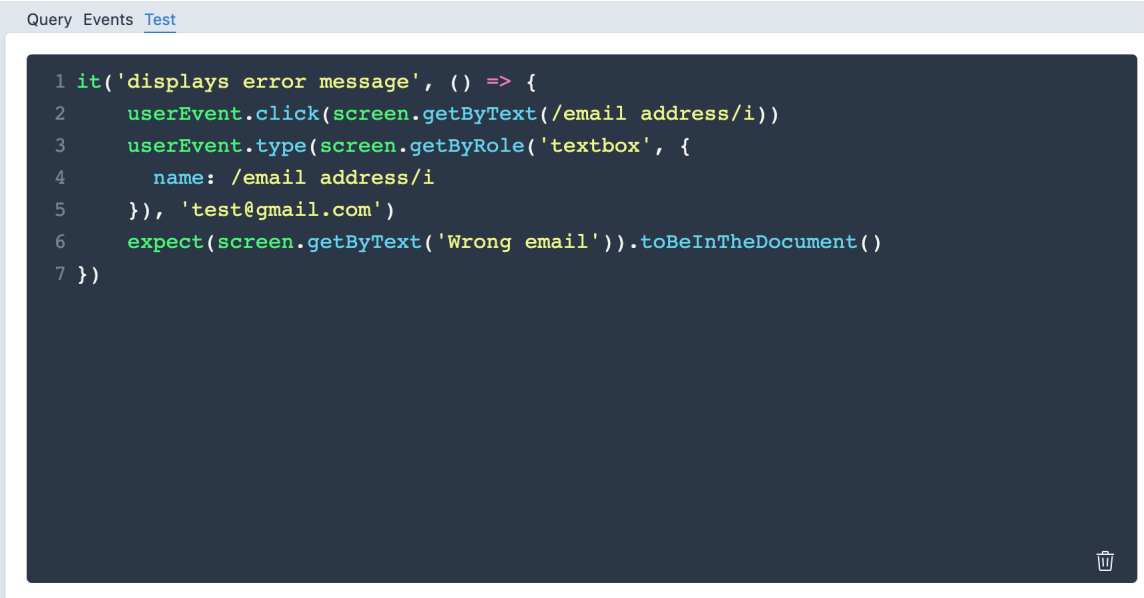

Figure 11. New test tab

The test can be built by clicking on an element. After the click, the tool suggests the prefered query and displays a list of $userEvent$ interactions and $Jest$ assertions. An example of prefered query and $userEvent$ interactions is shown in Figure 12.

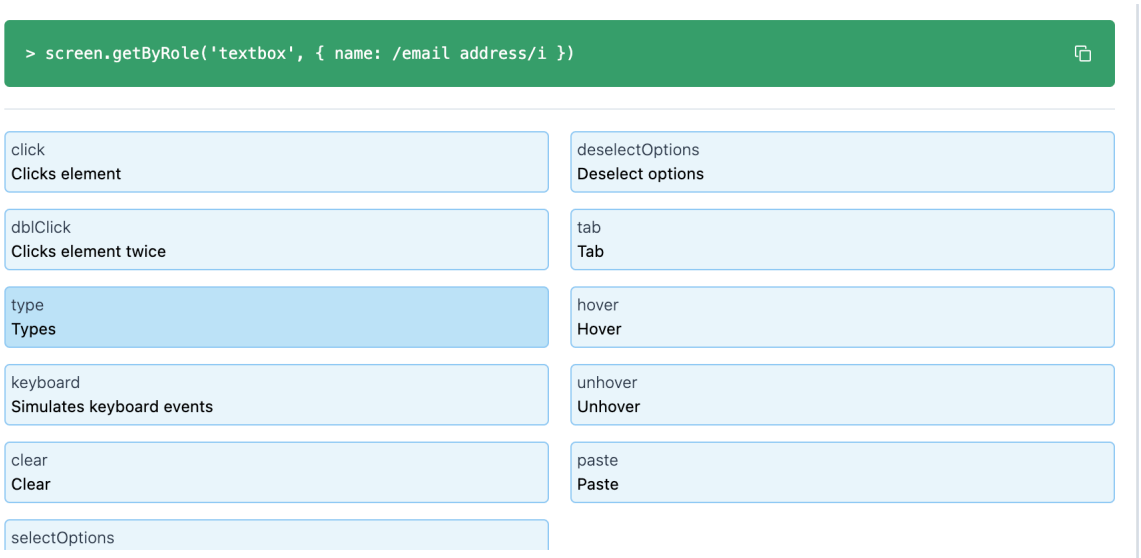

Figure 12. Suggested query and events

The list is scrollable and includes two assertions, as illustrated in Figure 13. The one at the left is used to ensure that the selected element is in the DOM, while the one at the right lets user type a text fragment, which has to be in the DOM. It was decided to only include $userEvent$ events and two assertions, because this amount of interactions/assertions is enough to show tool's posibilities.

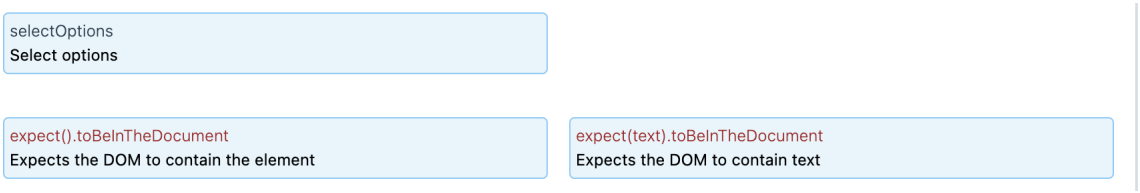

Figure 13. Available assertions

Thereby, the main flow from user's perspective, is as depicted in Figure 14.

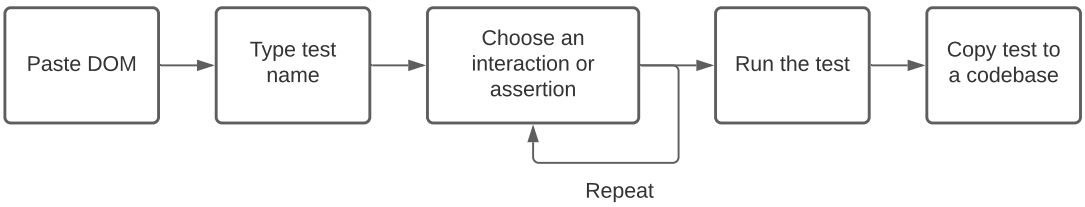

Figure 14. User flow

5.2 Filling the test based on user selections

The fundamental part of the proposed tool is to enable users to build the test case by selecting interactions and assertions. As shown in Section 5.1, it was done by typing the test name, clicking on DOM elements and selecting $userEvent$ calls and $expect$ assertions.

From the technical standpoint, just as other features, it was implemented consistently with the existing codebase. SET_TEST_EDITOR and SET_TEST Redux actions were added. The former updates the state when user selects the "Test" tab, while the latter fires when users create a new test case. An example of how the SET_TEST_EDITOR action was implemented is shown in Listing 16. When the action is fired, an it block is inserted into the "Test" tab input.

```
case 'SET_TEST': {
  exec({ type: 'UPDATE_EDITOR', editor: 'test' });

  const test = `it('${action.test}', () => {
})`;

  return {
    ...state,
    dirty: true,
    test,
  };
}
```

Listing 16: SET_TEST action

When user clicks on an DOM element, a prefered query is shown and $userEvent$ call or $expect$ assertion can be selected. Possible $userEvent$ calls and assertions were implemented by having a reusable data structure, which holds information about the suggestion or assertion. The $code$ is a string, which defines the code that will be put into the "Test" tab. : $query$ and : $input$ are

parsed when user selects the option, based on what is typed and the selected DOM element.

```
{
  name: 'type',
  desc: 'Types',
  code: "userEvent.type(:query, ':input')",
  withInput: true,
}
```

Listing 17: Interaction and assertion data structure

Thereby, it is relatively easy to add new options or modify existing ones.

Initially, when the user clicked on a DOM element, not only the prefered query was shown alongside possible interactions/assertions, but DOM events were triggered as well. Therefore, if a script was typed into the DOM output and it had $click$ or other listeners, they were fired. This was not ideal, because it prevents one from running a test with a $userEvent.click()$. For instance, the test asserts that after the click on a button, the text is updated. This test would not be executed, because when the user clicks on the button to select it in the playground, the text is updated even before the user runs the test. The chosen solution was to catch the event in the capturing phase and stop event's propagation, if the event was invoked by the user's click on the element.

5.3 Executing tests

One of the difficulties was to implement the execution of tests and show the result. Testing Playground already had a mechanism to execute arbitrary JavaScript code from a "Query" tab. However, it was being executed on every input change, which is not desired. The actions and global state had to be adjusted so that the test is executed and the result is shown only when user clicks on a "Run" button.

Initially, it was not clear how to run tests in a browser. From initial analysis, it seemed that $Jest$ tests cannot run in the browser. Therefore, instead of running the whole test case, it was decided to try running only interactions and assertions. Experiments showed that the $expect$ function, which comes from $Jest$, can be successfully executed in the browser. To make $expect$ available for execution of tests that the user can build, it had to be added to $eval$ execution context.

The testing experience would be pretty shallow if the tool did not allow executing $jest-dom$ assertions, because they come from testing library. Otherwise, assertions, such as $toBeInTheDocument$ would not be executed. Adding these assertions was not straightforward due to how $jest-dom$ is implemented. By examining $jest-dom$ source code it was found that a global $expect.extend()$ function is being called:

```
// extend-expect.js
import * as extensions from './matchers'

expect.extend(extensions)

// App.js
import expect from 'jest-matchers';
window.expect = expect;
import '@testing-library/jest-dom';
```

Listing 18: extend-expect.js

This means that $window.expect$ has to be available before $jest-dom$ is imported. However, ES6 modules do not allow dynamic import for 3rd party modules and the code above was not executed sequentially. The issue could be solved by patching $jest-dom$ and export a function, which could be called on demand:

```
// extend-expect.js
import * as extensions from './matchers'

export default (expect) => expect.extend(extensions)

// App.js
import expect from 'jest-matchers';
import extend from '@testing-library/jest-dom';

extend(expect);
```

Listing 19: Extending expect by patching jest-dom

However, by analyzing how the code was bundled, it was found that the code was being compiled with *babel*. By using *babelrepl* (http://babeljs.io/repl) it was found that the code in Listing 18 was being compiled into the following code:

```
"use strict";

var _jestMatchers = _interopRequireDefault(require("jest-matchers"));

require("@testing-library/jest-dom");

function _interopRequireDefault(obj) { return obj && obj.__esModule ? obj : { default: ↩
    obj }; }

window.expect = _jestMatchers.default;
```

Listing 20: Compiled ES6 code

It shows that the $jest-dom$ import was being hoisted and for that reason $window.expect$ was not available by the time $jest-dom$ code was being executed. When experimenting with $CommonJS$ module system, it became apparent that when importing $jest-dom$ with $require$ call, as opposed to ES6 $import$ statement, the $require$ call was not hoisted and it successfully solved the issue of extending $expect$ with $jest-dom$ assertions, as shown in Listing 21.

```
import expect from 'jest-matchers';
window.expect = expect;
require('@testing-library/jest-dom');
```

Listing 21: ES6 with require statement

Other approaches on how to run tests were investigated as well. Instead of running tests using $Jest$, they could be transformed into $Mocha$ tests. $Mocha$ tests are designed to be executed in the browser, but this solution adds a bigger set of complexity, such as transforming assertions and, furthermore, $jest - dom$ would not be available. Not having $jest - dom$, as already explained, could lead to a poor user experience.

5.4 End result

At the end, Testing Playground was expanded with the following features:

1. ”Test” tab was added, which has a writeable input.
2. User is able to create an it block by typing the test name. The it block is inserted into the input in the ”Test” tab.
3. User can click on an element and select a $userEvent$ call or an $toBeInTheDocument()$ assertion, which is inserted into the input in the ”Test” tab.
4. Run the test by clicking ”Test” button. Test result is then displayed.

The Testing Playground tool shows potential to extend its functionality even further.

Results and conclusions

In this thesis, the following **results** were achieved:

1. Analyzed JavaScript and its concurrency, DOM event model.
2. Analyzed React component-based programming model, component lifecycle and $useReducer$ hook, which is based on Flux state management architecture.
3. Identified and overviewed recommended and the most popular React testing tools, which were targeted when automating the creation of tests: $Jest$, $react - testing - library$.
4. Identified 6 tasks that can be automated when creating React component tests with $Jest$ and $react - testing - library$.
5. An open-source web application Testing Playground, which is written with JavaScript and React, was extended. The result is publicly available at a GitHub repository and can be found in Appendix nr. 1. It was chosen to extend this tool because it already implements selection of a prefered query and partially implements running tests in a browser. The tool was expanded with automation of the following tasks:
    1. Create an it block, based on user typed test name.
    2. Let the user pick an interaction and automatically convert it into $userEvent$ call.
    3. Let the user pick an assertion and automatically convert it into $expect$ call.
    4. Build a test case out of selected interactions and assertions.
    5. Run the test in a browser, so that the user can visually see it pass or fail.

In conclusion, the open-source web application Testing Playground was successfully extended to automate the creation of React component tests and can be extended even further.

The created tool has a number of **potential improvements**:

1. Support typing React code as opposed to DOM with vanilla JavaScript
2. Support more interactions and assertions:
    1. $fireEvent$ utilities
    2. $jest - dom$ assertions, besides $toBeInTheDocument()$
3. Add asynchronous queries to expand the area of tests that can be written using the tool
4. Add mocking, spying and other $Jest$ capabilities to expand the area of tests that can be written using the tool

References

[Arc] Jake Archibald. Tasks, microtasks, queues and schedules. https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules. [Visited May 15, 2021].

[Bab] Babel. Babel/plugin-transform-react-jsx | babel. https://babeljs.io/docs/en/babel-plugin-transform-react-jsx. [Visited May 15, 2021].

[Bei90] Boris Beizer. *Software Testing Techniques (2nd Ed.)* Van Nostrand Reinhold Co., USA, 1990. ISBN: 0442206720.

[dav] davidkpiano. Useeffectreducer github repository. https://github.com/davidkpiano/useEffectReducer. [Visited May 26, 2021].

[Dca] Kent C. Dodds and contributors. Accessibility | testing library. https://testing-library.com/docs/dom-testing-library/api-accessibility/. [Visited May 17, 2021].

[Dcb] Kent C. Dodds and contributors. Introduction | testing library. https://testing-library.com/docs/. [Visited May 17, 2021].

[Dcc] Kent C. Dodds and contributors. Testing-library/jest-dom github repository. https://github.com/testing-library/jest-dom. [Visited May 20, 2021].

[Faca] Facebook. Facebook/react: a declarative, efficient, and flexible javascript library for building user interfaces. https://github.com/facebook/react. [Visited May 14, 2021].

[Facb] Facebook. Hooks faq - react. https://reactjs.org/docs/hooks-faq.html. [Visited May 14, 2021].

[Facc] Facebook. In-depth overview | flux. https://facebook.github.io/flux/docs/in-depth-overview. [Visited May 20, 2021].

[Facd] Facebook. React - a javascript library for building user interfaces. https://reactjs.org/. [Visited May 14, 2021].

[Face] Facebook. React.component - react. https://reactjs.org/docs/react-component.html. [Visited May 14, 2021].

[Facf] Facebook. Reconciliation - react. https://reactjs.org/docs/reconciliation.html. [Visited May 21, 2021].

[Facg] Facebook. Rendering elements - react. https://reactjs.org/docs/rendering-elements.html. [Visited May 14, 2021].

[Fach] Facebook. State and lifecycle - react. https://reactjs.org/docs/state-and-lifecycle.html. [Visited May 14, 2021].

[Faci] Facebook. Testing overview - react. https://reactjs.org/docs/testing.html. [Visited May 17, 2021].

[Facj] Facebook. Unexpected render | facebook/react. https://github.com/facebook/react/issues/18098. [Visited May 18, 2021].

[Fack] Facebook. Using with puppeteer. https://jestjs.io/docs/puppeteer. [Visited May 18, 2021].

[Facl] Facebook. Virtual dom and internals. https://reactjs.org/docs/faq-internals.html. [Visited May 21, 2021].

[Goo] Google. Angular/angular: one framework. mobile & desktop. https://github.com/angular/angular. [Visited May 14, 2021].

[KA14] Mohammed Khoudary and Wesam Ashour. Design and implementation of a selective continuous test runner. *International Journal of Artificial Intelligence and Applications for Smart Devices*, 2:23–38, 2014-11. DOI: 10.14257/ijaiasd.2014.2.2.03.

[Kab] KabaLabs. Cypress recorder - chrome web store. https://chrome.google.com/webstore/detail/cypress-recorder/glcapdcacdfkokcmicllhcjigeodacab. [Visited May 26, 2021].

[Moza] Mozilla. Concurrency model and the event loop. https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop. [Visited May 13, 2021].

[Mozb] Mozilla. Introduction to events. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events. [Visited May 26, 2021].

[Mozc] Mozilla. Using microtasks in javascript with queuemicrotask(). https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API/Microtask_guide. [Visited May 15, 2021].

[npm] npm. Jest - npm. https://www.npmjs.com/package/jest. [Visited May 16, 2021].

[Per20] M. Persson. Javascript dom manipulation performance: comparing vanilla javascript and leading javascript front-end frameworks. In 2020.

[pet] petrkrejcik. Events recorder with test code output. https://github.com/testing-library/testing-playground/issues/294. [Visited May 26, 2021].

[Raj21] Raju. Mutation testing framework for machine learning, 2021. arXiv: 2102.10961 [cs.SE].

[SE03] David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *ISSRE 2003: Fourteenth International Symposium on Software Reliability Engineering*, pp. 281–292, Denver, CO, 2003-11.

[VKC17] Akanksha Verma, Amita Khatana, and Sarika Chaudhary. A comparative study of black box testing and white box testing. *International Journal of Computer Sciences and Engineering*, 5:301–304, 2017-12. DOI: 10.26438/ijcse/v5i12.301304.

[You] Evan You. Vuejs/vue: vue.js is a progressive, incrementally-adoptable javascript framework for building ui on the web. https://github.com/vuejs/vue. [Visited May 14, 2021].

Appendix nr. 1
Extended Testing Playground

A result of this thesis: a Testing Playground fork with more capabilities, allowing users to build test cases and execute them. The fork is publicly available at https://github.com/zaicevas/testing-playground/tree/test-builder