An Nguyen Van

# Label printing web application library for Finnish paint company

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

3 May 2022

## Abstract

| | |
|---|---|
| Author: | An Nguyen Van |
| Title: | Label printing web application library for Finnish paint company |
| Number of Pages: | 53 pages |
| Date: | 8 May 2022 |
| | |
| Degree: | Bachelor of Engineering |
| Degree Programme: | Information Technology |
| Professional Major: | Software Engineering |
| Supervisors: | Janne Salonen (Principal Lecturer) |

The purpose of the project was to plan and construct a standalone label customizing and printing library, which would be used in the point-of-sale application for selling paint for client on the hardware store using web technologies. This is part of the company's vision to migrate from multiple old school desktop applications using different technologies into a universal web application. The decision is not only providing a streamlined development experience but also enhancing the scalability of the whole application in the future.

The project was the combination of observing web technologies frameworks and researching existed methodologies. The team concluded that the label printing application would be an isolated web application at first but still utilizing the same technologies as others so it could be integrated with ease later. Furthermore, the frontend development process is also supported by a workflow to ensure the collaboration between developers, avoid time consuming repetitive tasks.

In conclusions, the project was a great success. The label web printing fulfils the technical needs from the development team and the company's vision. It could function well both as a standalone and as a library of other web application. However, there is still several enhancements that could be done such as developing tests for the library to allow shipping the product with confidence, updating dependency packages for new features from other libraries. However, the migrating tasks are addressed first, improvements will be assigned after the new point-of-sale is released.

Keywords: HTML, CSS, JavaScript, React, Redux, Front-end development, GitHub actions, CI/CD

## Contents

## List of Abbreviations

| | |
|---|---|
| HTML: | Hypertext Markup Language. |
| CSS: | Cascading Style Sheet. |
| JS: | JavaScript. |
| UI: | User Interface. |
| OS: | Operating System. |
| API: | Application Programming Interface. |
| CI/CD: | Continuous Integration and Continuous Delivery. |
| QA: | Quality Assurance. |
| DevOps: | Application development and operations. |
| ES6: | ECMAScript 6. |

# 1 Introduction

In the digital age, when the needs for rapidly information transfer rise and individuals are managed to access to technologies with ease, so does the surge of the website functionality. However, as the website evolves into web application, scalability and solid architecture system are needed during the developing period. This thesis presents the approach of Company C for implementing a reusable web application utilizing the new front-end technologies. During 2019, Company C decided to migrate from their monolith desktop-based application into multiple micro web applications.

Nevertheless, the core of front-end web development still consists of Hypertext Markup Language (HTML) for structuring document, Cascading Style Sheet (CSS) for styling and JavaScript (JS) for interactive functionality. The web application nowadays is frequently modified to adopt with the latest design, but several practices and structures must be followed in a team of developers to allow collaboration and scalability.

The thesis describes the entire process of developing and bundling the label customizing and printing library with custom structure and solution. Furthermore, to fulfil Company C’s demands for a scalable, maintainable product and could be integrated with future projects with ease.

# 2 Theoretical Background

## 2.1 Client company

### 2.1.1 Client decision

As the current paint desktop application requires various vendors to maintain and in need of update for future expansion of the company. Company C, being one of the global innovators in producing paint pigment used in agriculture, construction and other industries. The company has multiple subsidiaries across Europe, America, China and Sound East Asia, they provided the paint product accompanied by the paint dispensing software for the client.

Company C has been utilized its own custom desktop software since the foundation. Nevertheless, the current application is only compatible on a certain type of operating system, which lead to several restrictions with potential client.

Based on the feedbacks from client, the company made the decision to migrate from its monolith desktop application to several reusable web applications. The great migration started during the Summer of 2019 and is still in progress, the customized label printing application is part of the requirements.

### 2.1.2 Client requirements

Traditionally, new requests form the client must be audited by both the designer and developer teams to ensure the quality of the features. However, to fulfil the business plans for upcoming clients from across the globe in the shortest amount of time as possible, Company C’s designers decided to utilize the prototype of the current desktop application to design the new user interface.

The development team presented a custom solution allows them to continuously develop new features, ensure the compatibility between projects, a workflow that automatically run the test and immediately deploy to the hosting service to shorten the reviewing process. With this system, each team in the software development department would be able to collaborate with each other confidently while being independently in logic.

By using the new approach to the design and development process, the teams met the requirements in terms of transforming a monolith software into multiple applications.

## 2.2 Web technologies

### 2.2.1 Introduction to web technologies

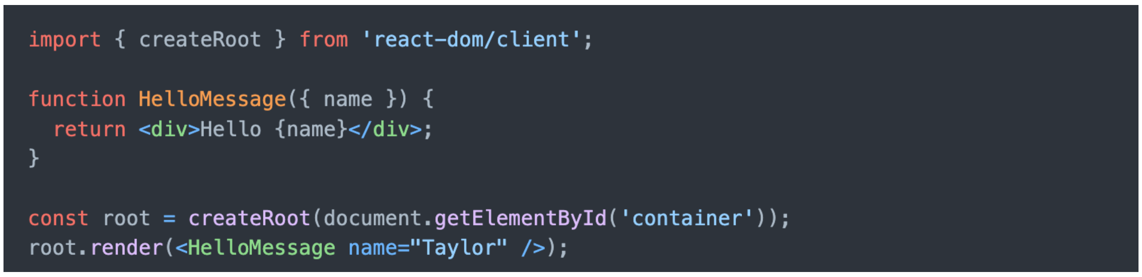
The basic technologies functioned as the core for the majority of website are HTML, CSS and JS. HTML is the markup language which constructs the layout of web content in a meaningful context, it utilized “markup” text to product “elements” that responsible for displaying the content in a web browser [1]. Originally, HTML is the only essential piece for the website, but as website evolves into web application, it often includes CSS and JS. CSS is the web language that describes the web’s appearance [2], the language provided the web page not only the enhancements in terms of presentation but also the scalability for multiple different devices. Lastly, JS, a lightweight programming language, famous for being the scripting language for web pages and many other environments [3]. JS is the “brain” behind the web page that allows it to connect to resources from sever, handle complex logic resulted in an interactive and dynamic experience for users.

Nowadays, most of the modern web pages or applications used the combination of those technologies. As the demands for web pages grow, those projects need to be architected in a concise manner for reducing the time consumption in both developing new features and fixing bugs. This issue leads to the birth of various web frameworks, whose provide out-of-the-box basic logic handling for several functionalities. Hence, the projects could be easily bootstrapped as the developer teams only need to concentrate on business logic.

### 2.2.2 React & Create-React-App

During the process of researching for the suitable web frameworks to migrate the desktop application, there are several options which can be adopted are Angular React and Vanilla JavaScript. After having discussions regarding the technical specifications, developer’s interests and plans for the web application, the software department has chosen React to be the core web framework for front-end development due to being a scalable lightweight framework and easy to adopt.

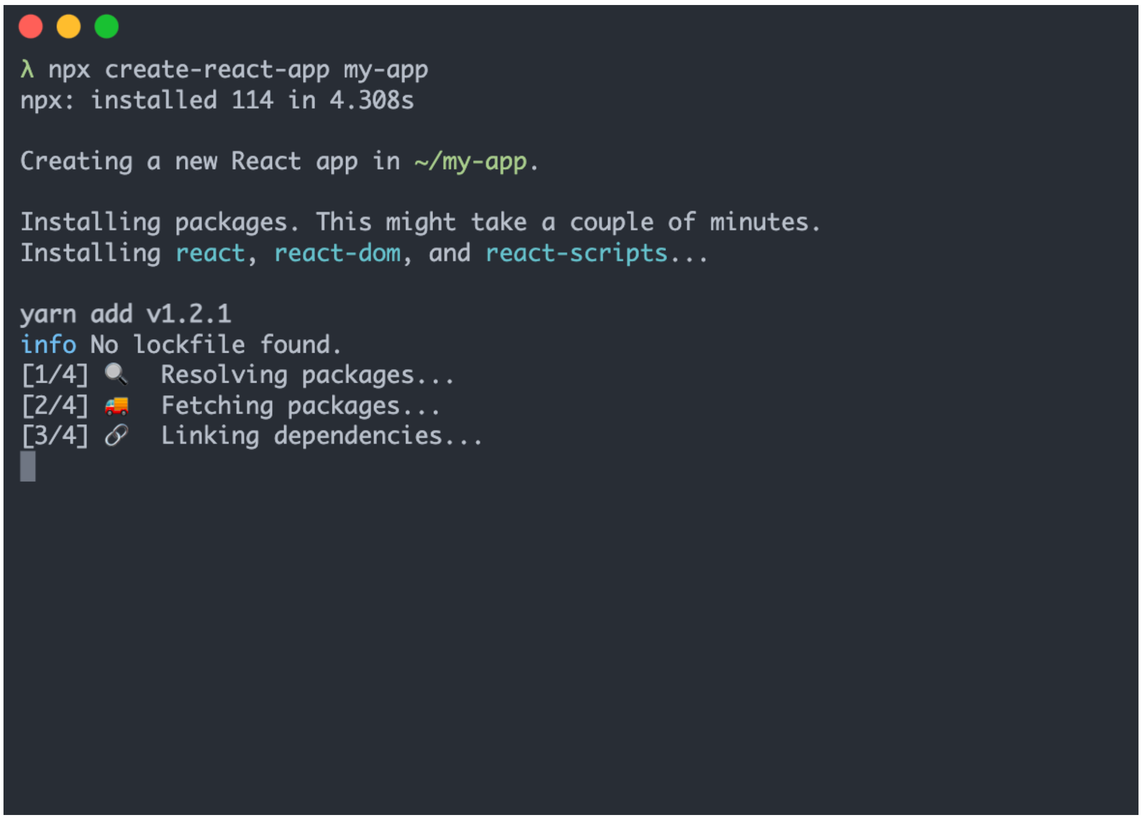
React is a JavaScript library developed by Meta Company, formerly known as Facebook, with component-based approach for developing user interfaces. Firstly, component-based approach is the building of isolated web sections with their own state, hence the name component. The components could be later composed together to form an interactive UI, this plays an essential role in rapid development since encapsulated components support the declarative programming paradigm. Furthermore, the library allows the usage of external packages and plugins in the React application, which drastically shorten the development process since the team can use the well tested solutions from the community in the projects [4].



```
import { createRoot } from 'react-dom/client';

function HelloMessage({ name }) {
  return <div>Hello {name}</div>;
}

const root = createRoot(document.getElementById('container'));
root.render(<HelloMessage name="Taylor" />);
```

Figure 1: Component based development [4].

Due to the shortage of time and the size of boilerplate generated code for bundling the React application, the team decided to utilize Create React App, a proven solution developed by the engineers' team from Meta Company. Create React App is the combination of multiple tools under one easy to maintain package, it offers out-of-the-box modern setup for building React application, handling styling import, etc [5]. Provided with these technologies, Company C's label customization application can start on shipping the business focused product to the client without consuming time on setting up the needed environment for development.



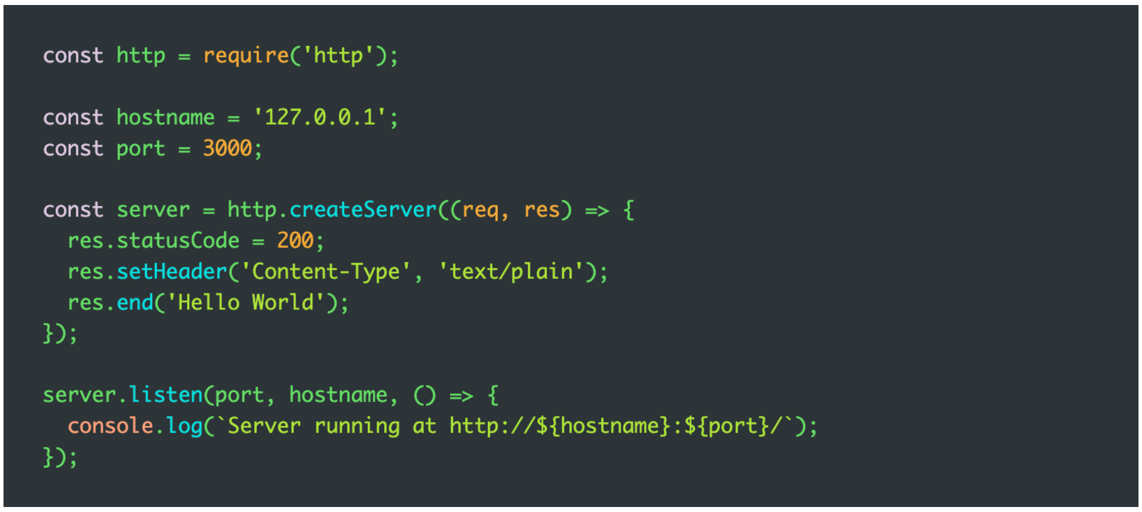
```
λ npx create-react-app my-app
npx: installed 114 in 4.308s

Creating a new React app in ~/my-app.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts...

yarn add v1.2.1
info No lockfile found.
[1/4] 🔍  Resolving packages...
[2/4] 🚚  Fetching packages...
[3/4] 🔗  Linking dependencies...
```

Figure 2: Setting up React project with Create React App with one bash command [5]

### 2.2.3 Node.js & NPM

React library requires an environment to run on local development, tool to manage dependencies, compiler to convert the modern JavaScript used in project into version that can be run by other JavaScript engines and browsers, etc. By using Create React App, those tools are automatically installed in the project, Node.js and NPM are included in the packages.

“Node.js is a JavaScript runtime built on Chrome’s V8 JavaScript engine” [6]. Traditionally, JavaScript is only run-on browser, but nowadays with the creation of Node.js, the new run time permits JavaScript to interact with HTTP request, access operating system, build scalable server for production and use various useful functionalities that previously are not possible.



```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Figure 3: Node.js enable server creation with JavaScript [7].

NPM is an essential piece of technology that allow developer to develop and share packages of encapsulated code [8]. Furthermore, it also stands for Node Package Manager, as the name suggested, NPM automates the installation, updating the project related packages.

The usage of Node.js and NPM increase the productivity of the software department team by integrating the library with multiple quality external packages and unlocking potentials for shipping as a standalone component to the point-of-sale application.

### 2.2.4 Rollup.js

The application is developed under local development provided by Node.js, nevertheless, Create React App is designed to build the code into production not a library. The team recognize the need for a custom-made bundler which the developers can use to build the React code into a library instead of a production application.

Firstly, team will use the boilerplate code created by Create React App not only as part of a local development, but also a testing application integrated with the library since the point-of-sale application is also built using Create React App. Secondly, implementing the label printing application using custom module bundler built on Rollup.js. Finally, when the library is bundled and confirmed on the ability to be used as a component in the testing environment, the team will deploy the code from the testing React application to hosting service for feedbacks.

Rollup.js is a JavaScript bundler that composed the project modular code base into a library. It has simple API and various built-in functionalities such as tree shaking, support multiple formats and using packages from other libraries. [9]

### 2.2.5 Redux

Thanks to the usage of React, the application development has progressed smoothly, however, the team is currently facing the issue of having to pass down the data from parent components to their child for them to access the state to the web application state. After researching, Redux is chosen to be the main state manager for its flexibility and easy to debug.

Redux is a consistent state container library that centralized entire the application’s state and logic in a single source [10]. Redux is created with the three fundamental principles. Firstly, the entire state of the application is encapsulated in a single source of truth called the “store”. A single source of data store provides persistent for the application’s state hence faster development and simpler debugging progress. Secondly, the state tree is immutable through any other effects in the application and can only be modified by dispatching an action. An action is a plain object contained the instruction and data which the state needs for the transformation. Finally, after receiving the package from the action object, the state tree uses reducers to execute the modification. Reducer is basically a pure function, which has the state of the application and an action as parameters. It performs the state transformation based on the instruction and data of the action then returning the new the state instead of directly modifying the previous state [11].

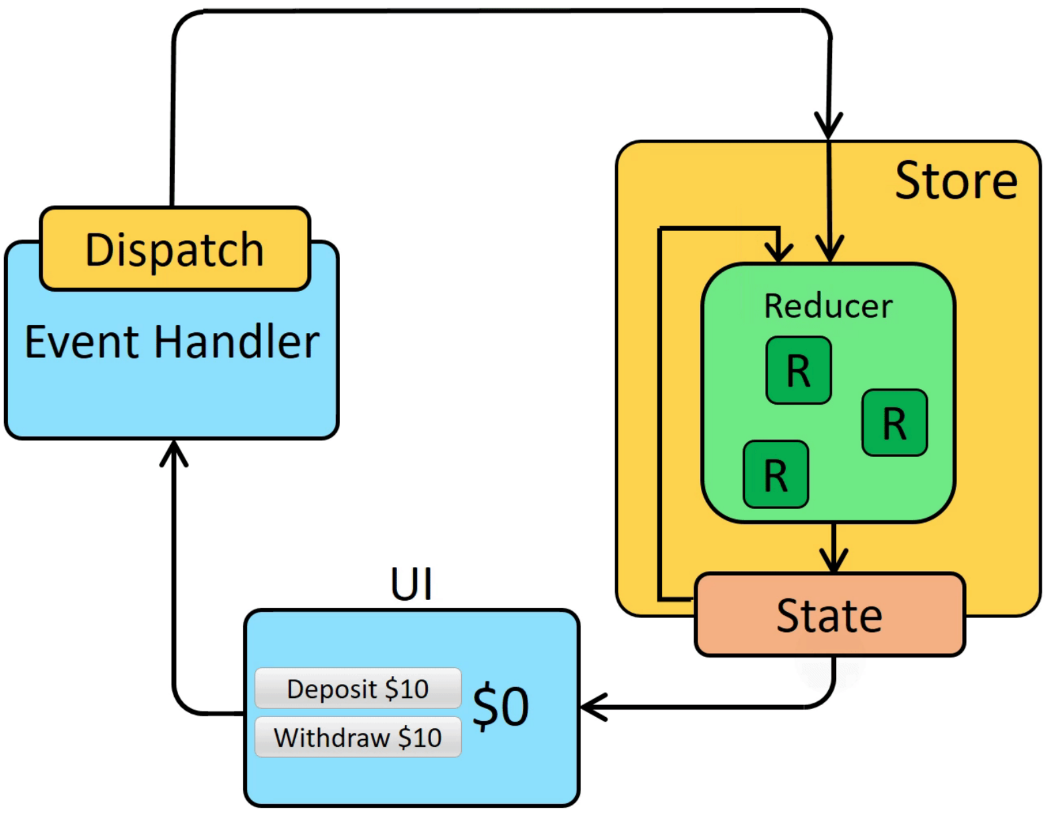


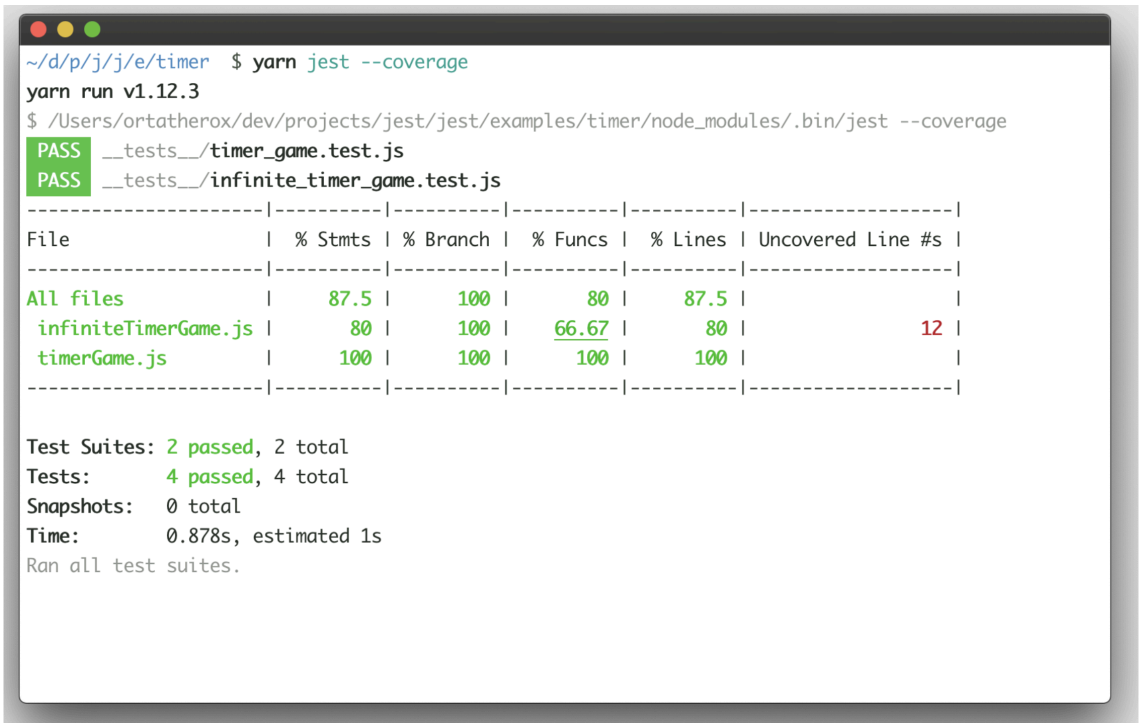
Figure 4: Simple application workflow used Redux as state manager [12].

With the addition of Redux, the printing application's components have access to the state tree's data and performing safe changes to it when needed.

### 2.2.6 Jest

When it comes to software, the process of developing the application usually accompanied by the creation of bugs. The development team would want to ship the library with confidence and help the QA department in the testing period. The development team chooses Jest as the tool for testing isolated functionalities for the application.

Jest is a JavaScript testing framework, created by Meta, that allows developers to implement tests in a simple and fast way. Jest provides the team with feature-rich APIs to use such as mocking capability, running tests in parallel and display the code coverage of the entire application [13].



```
~/d/p/j/j/e/timer  $ yarn jest --coverage
yarn run v1.12.3
$ /Users/ortatherox/dev/projects/jest/jest/examples/timer/node_modules/.bin/jest --coverage
 PASS  __tests__/timer_game.test.js
 PASS  __tests__/infinite_timer_game.test.js
----------------------|----------|----------|----------|----------|-------------------|
File                  |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
----------------------|----------|----------|----------|----------|-------------------|
All files             |     87.5 |      100 |       80 |     87.5 |                   |
 infiniteTimerGame.js |       80 |      100 |    66.67 |       80 |                12 |
 timerGame.js         |      100 |      100 |      100 |      100 |                   |
----------------------|----------|----------|----------|----------|-------------------|

Test Suites: 2 passed, 2 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        0.878s, estimated 1s
Ran all test suites.
```

Figure 5: Testing coverage with Jest [13].

The team original plans on writing only unit test the library due to the fact the reusability is handled by the Rollup.js bundler and Create React App testing environment.

### 2.2.7 GitHub Actions

Version control is an essential tool for software development to ensure the collaboration between developer as easy as possible, and in case of changes that break the production, the ability to roll back to a previous version is crucial. The software development teams in Company C make use of Git for version control and GitHub for hosting the repositories on daily work.

The label printing team realizes that developers a considerable amount of time to build the software and deploy the application to the hosting service manually for feedbacks. GitHub provides users with a continuous integration and continuous delivery (CI/CD) system to perform the building, testing and deploying tasks automatically called GitHub Actions. GitHub Actions is not only allowing DevOps related operation, but also providing possibility for the pipeline to interact with the repository when other events occur [14].

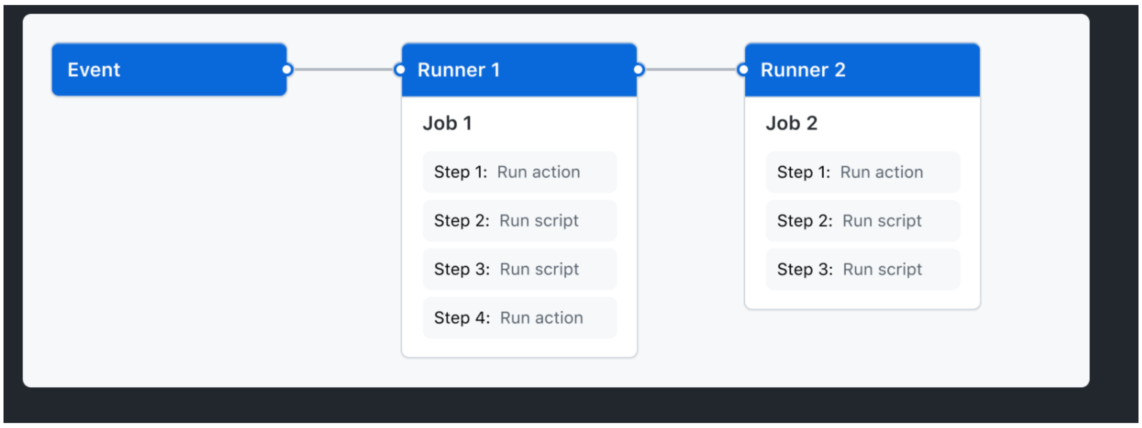
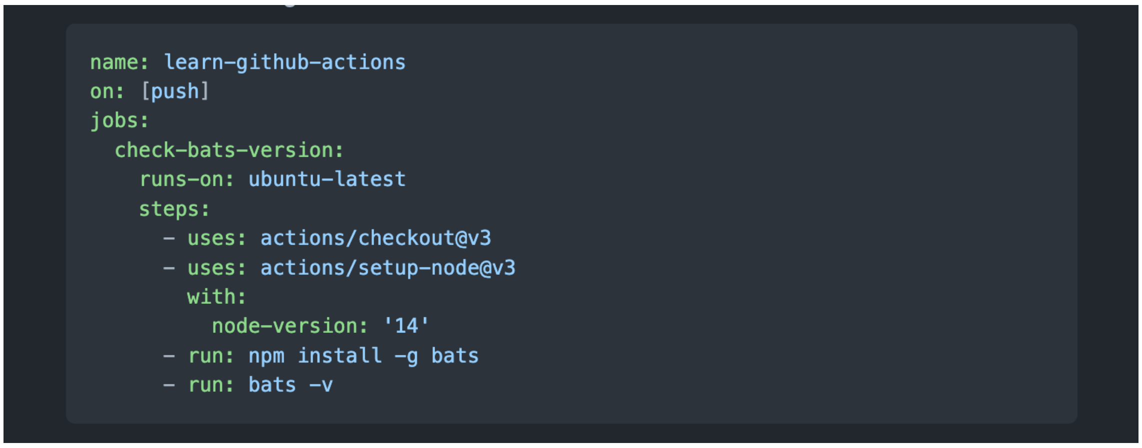


Figure 6: Event triggers workflow to run [14].

GitHub Actions pipeline will run when workflows attached to events in the repository are triggered. Workflow is a customizable process in a YAML file that contained one or several jobs, which will run when triggered by any related events. Event is a certain activity that triggers one or more workflows to execute. For example, pushing a commit to the remote repository or merging a pull request into a branch, etc. A job normally consists of variable steps that run on a specific virtual machine called runner [14].



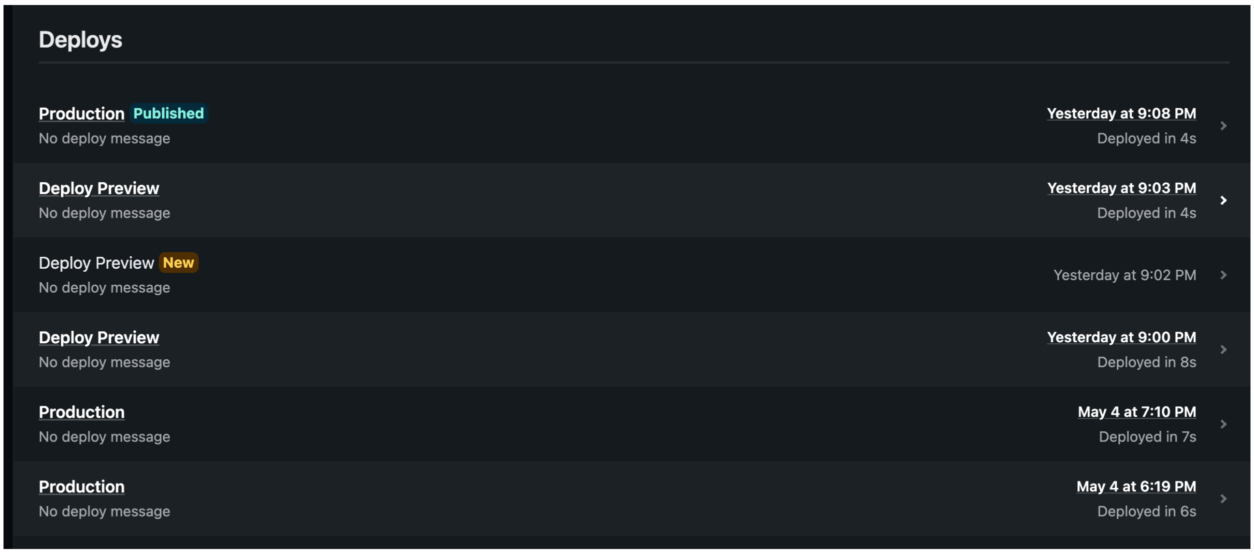
```
name: learn-github-actions
on: [push]
jobs:
  check-bats-version:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '14'
      - run: npm install -g bats
      - run: bats -v
```

Figure 7: An example workflow in GitHub Actions [14].

Utilizing the GitHub Actions, the team has increased the productivity significantly due to time spent on manual repetitive task can be used on developing core features of the application.

### 2.2.8 Netlify

Netlify is the chosen PaaS cloud service for hosting the application due to it fast to set up and easy to use CLI. Furthermore, Netlify has the potential to scale in the future such as replacing GitHub Actions pipeline in continuous integration, deployment pipeline, serverless function, user authentication for server if there is a need for its backend services [15]. For web application, Netlify also manages the domain and DNS out-of-the-box, the ability to quickly build the application as the preview or production helps shorten the feedback loops during the development. The QA team can utilize multiple preview build versions of web app to test with different issues and features effortlessly before publishing to the production.


Deploys

| | |
|---|---|
| Production Published<br>No deploy message | Yesterday at 9:08 PM<br>Deployed in 4s |
| Deploy Preview<br>No deploy message | Yesterday at 9:03 PM<br>Deployed in 4s |
| Deploy Preview New<br>No deploy message | Yesterday at 9:02 PM |
| Deploy Preview<br>No deploy message | Yesterday at 9:00 PM<br>Deployed in 8s |
| Production<br>No deploy message | May 4 at 7:10 PM<br>Deployed in 7s |
| Production<br>No deploy message | May 4 at 6:19 PM<br>Deployed in 6s |

Figure 8: Multiple build versions ready for testing.

With the combination of GitHub Actions and Netlify, the developers in label printing team can collaborate effectively with the designers, quality assurance team. Moreover, these technologies are highly scalable, the label software team can help other teams in the department to set up their workflow to bootstrap the productivity in every projects.

# 3 Implementation

As mentioned from the introduction, the main objective of this thesis is to migrating the label printing a web library standalone application with compatibility and scalability. This section contains four subsections each with its own merits:

- **Methodology**: the methods used in the application development cycles.
- **Implementing User Interface with React & Redux**: structuring and creating the project front-end with different type of components such as presentational components for displaying the user interface, container components for handling logic and connecting with the data store created by Redux.
- **Bundling library**: demonstrating the usage of Rollup.js to allow bundling, handling styling and exporting the library.
- **Implementing workflows with GitHub Actions**: utilizing the GitHub Actions workflows to create an automation system for the testing, building and deploying tasks of the application.

## 3.1 Methodology

### 3.1.1 React Concepts

As React is used as the core tool for developing user interface in the project, there are several essential keys knowledge that a developer must deeply understand to implement a good performance and well-structured application.

Firstly, instead of using the real DOM to interact with the page, React provides the programmer with Virtual DOM. DOM, stands for Document Object Model, represents an interactive API for web documents. It creates tree-like structure for HTML documents and allow developer to manipulate the nodes in the various ways [16].

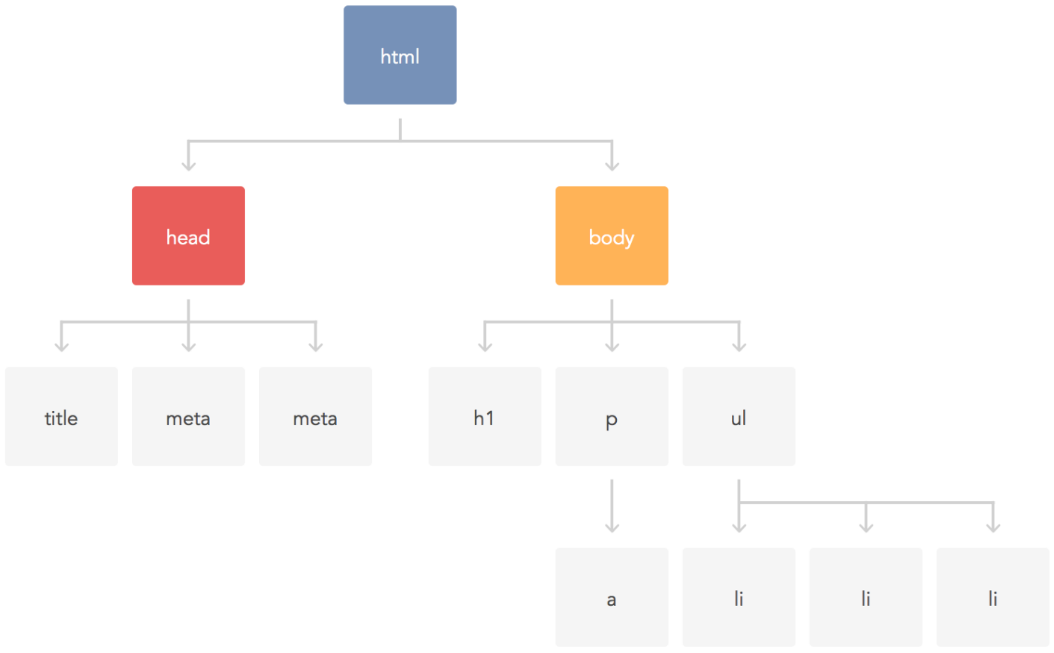


Figure 9: DOM tree [17].

Nevertheless, as website grow more complex and dynamic, the DOM tree will also grow and contain a huge of number of nested nodes. The DOM's traversing process and selecting the desired node to trigger an update for the user interface based on the state of the application is slow, time consuming and error prone.

Virtual DOM is a copy version stored in memory of the real DOM [18]. It can be described as a lightweight, more abstract version of the DOM, any operations that modifies the user interface occur in Virtual DOM is simpler and faster. When the changes are made in the Virtual DOM, React determines the update strategy for the real DOM using advanced algorithm named reconciliation.

The algorithm uses element type and key to justify the update cycle. If the roots in virtual DOM and real DOM have the same types and attributes, the process recurses on the children nodes. React triggers re-render when the traversing reaches a difference, the tree will be rebuilt if it is the root element while only update the change for a specific element if it is a children node. React also introduces key attribute that supports conversion tree node to be faster and more efficient [19].

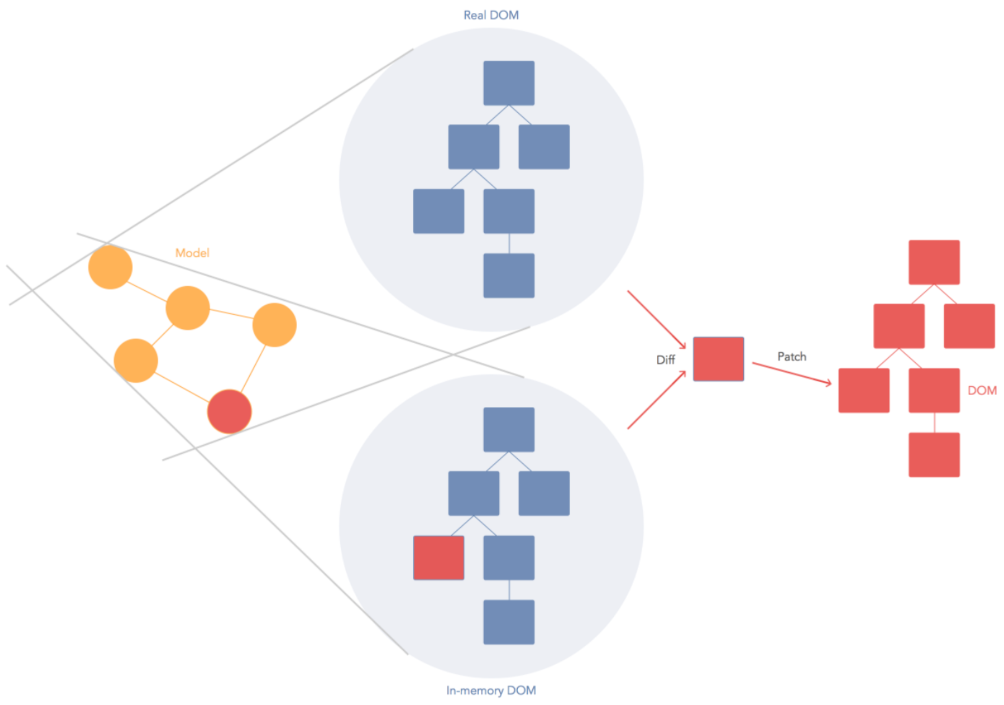
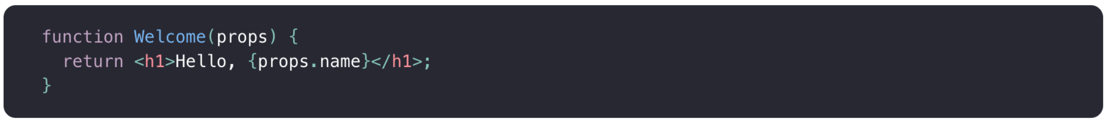


Figure 10: Comparing and updating DOM [17].

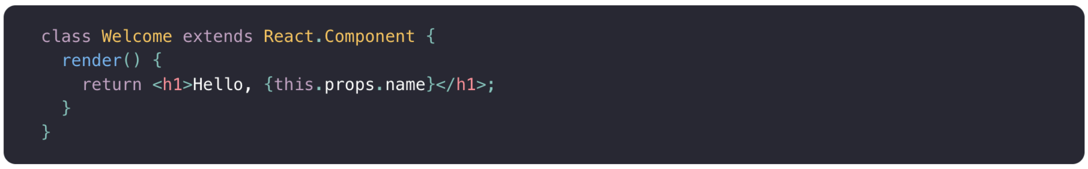
Secondly, React proposes components based approach to the development of user interface, so what is a component? Component is theoretically the same as JavaScript function. Normal function can have different type of inputs and return the outputs, like function, a component accepts arbitrary inputs called “props” (an object contains data) and outputting the React elements to the screen [20]. React element is a plain JavaScript object, much more lightweight and cheaper to instantiate comparing to DOM node [21].

There are two ways to produce a React components. The first way is the write a JavaScript function with “props” as the parameters and returning a React element, normally this is called “function components”. Another way is to use new ES6 class [20].



```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Figure 11: Function component [20].



```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Figure 12: Class component [20].

These two components are displaying the same user interface in the web application. However, the Class component allows the programmer to add local state and life cycle to the block of code.

Finally, the idea of isolated component development without affecting other part of the application is supported by state and life cycles. State in class component is normally an empty object but in contrast to props, it is exclusive and controllable by their own component [22].

A component is equipped with various life cycle methods. A life cycle method can be known as a special function that runs during a certain life cycle of the component [23]. Traditionally, a component has three main life phases:

- **Mounting**: when the component is instantiated and attached to the DOM.
- **Updating**: when an event triggers the changes in either props or state, the component enters the updating phase.
- **Unmounting**: when the component is removed from the DOM.

The essential methods which are needed when developing a component:

- **Constructor:** the constructor method is called before the component reach the mounting phase. This is where the instantiation of local state and event handler bindings to the component happened.
- **ComponentDidMount:** this method is executed immediately the insertion of the component into the DOM. Performing data fetching, handling side effects for the first time is normally done is this method.
- **ComponentDidUpdate:** the function is invoked after the update events occur. A network request or modification to the element

state can be accomplished in this method if they depend on the newly changed state or props.

- **ComponentWillUnmount:** the method is invoked when the component is destroyed from the DOM if the needs for cleaning up any side effects that might occur even the component is unmounted.

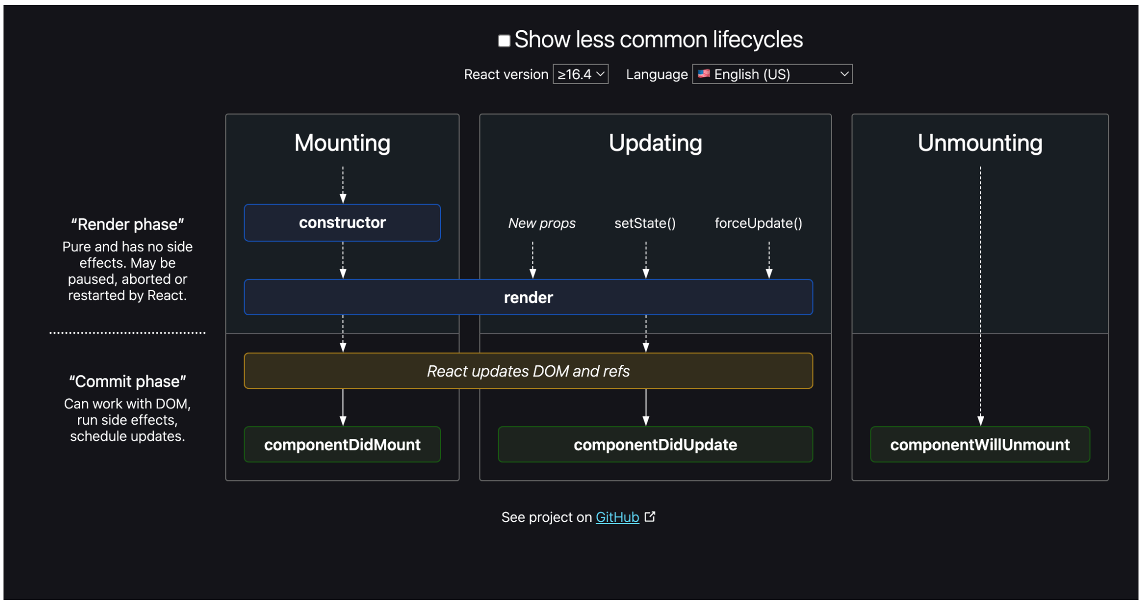
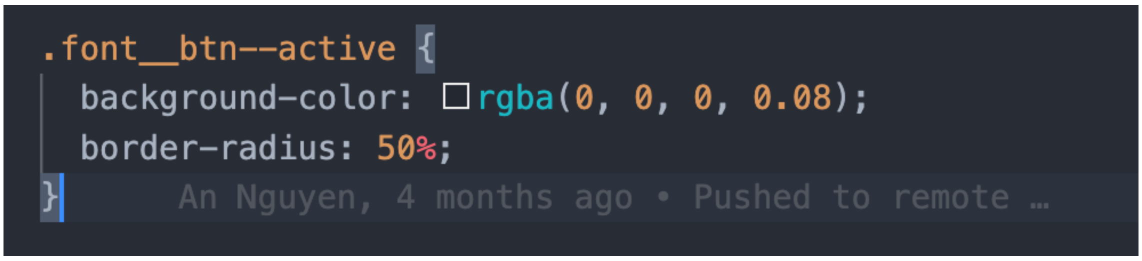


Figure 13: Lifecycles and methods of a component [24].

### 3.1.2 BEM Naming Convention

How to structure the styling for component plays a crucial role in the scalability of the application especially when the project becomes more complicated. With well-organized styling code, it shortens the development cycle by reducing the amount of code written and loading time for the browser. BEM methodology, stands for Block, Element, Modifier, is a well-known naming convention for CSS as not only is it organizing the styling code in a simple to understand way but also creating a relationship between the component and CSS [25].

The way the BEM Naming Convention works is firstly, a parent block is positioned at the top-level. Secondly, nested child or elements are followed by adding two underscores and the name of the block after the parent. Finally, when there is a need for a specific style of the same element without altering the other styling modules, a modifier can be inserted at the end of the name block by writing two hyphens and a text indicates the change [25].



```css
.font__btn--active {
  background-color: □rgba(0, 0, 0, 0.08);
  border-radius: 50%;
}       An Nguyen, 4 months ago • Pushed to remote …
```

Figure 14: Example of using BEM in CSS.

### 3.1.3 CI/CD

CI/CD can be also known as continuous integrations and continuous delivery [26]. The methodology implies the automation tasks focusing on recurring and well-tested software delivery operation in real time.

The workflow starts with continuous integrations, after developers perform an action which alters the source code of a branch in a CI integrated system, it automatically tests and builds the code which allows for frequent quality code contribution to the repository.

After the code passes the test and is built from CI, continuous delivery and continuous deployment are the next steps on the DevOps pipeline. The code is then putted into an environment where the team can deploy manually or auto triggering a deployment script to push the code directly to the production.

With the combination of CI and CD, the programmer can deliver code quickly with confidence either to testing area for testing and feedbacks or to production for releasing.

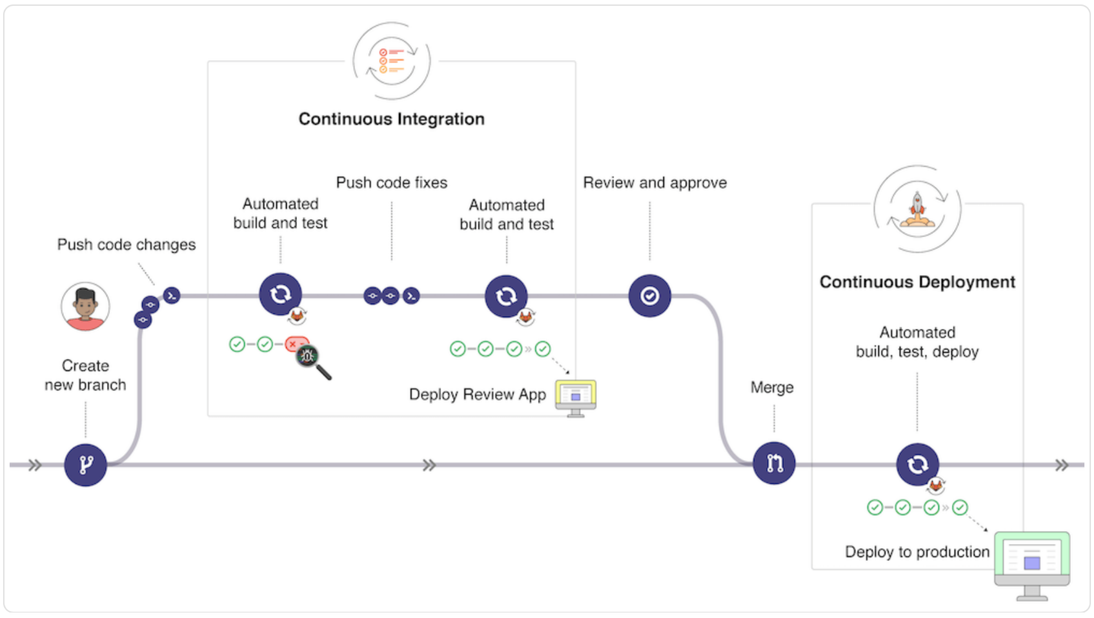


Figure 15: Example of CI/CD workflow [27].

## 3.2 Implementing User Interface with React & Redux

### 3.2.1 Project structure

The project use BEM for structuring the CSS, but even if the relationship between the component and its styling is clarify, there is still a need for organize the components in a way so that it would be easy to identify which is stateless or presentational and which is stateful or container.

Presentational component can function in isolation with the rest of the application, meaning it is not dependent on mutual state with other component, its purpose is mainly for displaying the styled element on the screen. Furthermore, it does not perform any operations related to modifying data on its own, but rather receiving data and the data-altering call backs only through props passed down from parent or from specific state manager [28].

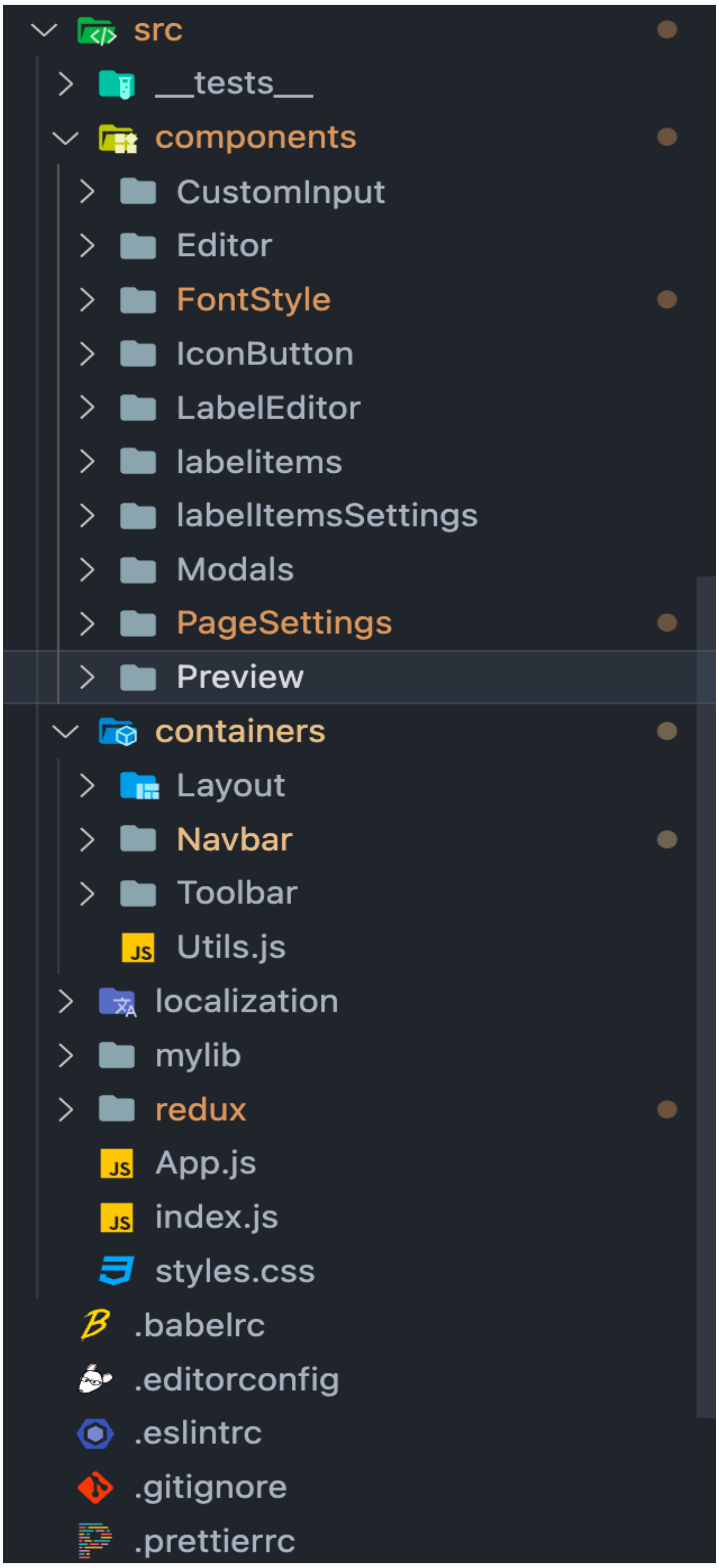
In contrast to the stateless component, the container component main responsibility is providing the data to other components, so it is usually a stateful class component or connecting with the mutual state of the application.

It may not have its own element but being the combination of other presentational and container components [28].

Thanks to the separation of concerns regarding the functionality of components, programmer can develop better reusable and adaptive component and only use container for when state is needed [28]. For example, when the new design requires changes related to several stateless components, they can be modified with ease and does not affect the logic of the entire application.

The project implements the structure as follow:

- **__tests__:** storing the unit test for component.
- **components:** including the implementation of all reusable components.
- **containers:** containing the stateful, Redux store connected container components.
- **mylib:** containing utility functions.
- **redux:** including the redux reducers and actions implementation.



```
src
  __tests__
  components
    CustomInput
    Editor
    FontStyle
    IconButton
    LabelEditor
    labelitems
    labelItemsSettings
    Modals
    PageSettings
    Preview
  containers
    Layout
    Navbar
    Toolbar
    Utils.js
  localization
  mylib
  redux
  App.js
  index.js
  styles.css
.babelrc
.editorconfig
.eslintrc
.gitignore
.prettierrc
```

Figure 16: Project structure with separation of components and containers.

### 3.2.2 Presentational Components

Here is the implementation of a state components in the application:

```
const IconButton = (props) => {
  const {
    icon,
    iconSize,
    text,
    inNavbar,
    onClick,
    color,
    active,
    draggable,
    id,
    onDragStart,
    spin,
  } = props;
  const buttonStyle = classnames({
    btn__container: true,
    'btn__container--navbar': inNavbar,
    'btn__container--toolbar': !inNavbar,
  });
  const iconStyle = classnames({
    icon: true,
    'icon--toolbar': !inNavbar,
  });
  return (
    <Button
      type="button"
      color={color}
      className={buttonStyle}
      size="md"
      onClick={onClick}
      active={active}
      draggable={draggable}
      id={id}
      onDragStart={onDragStart}
    >
      <span className={iconStyle}>
        {icon && (
          <FontAwesomeIcon icon={`${icon}`} size={iconSize} spin={spin} />
        )}
      </span>
      {text}
    </Button>
```

```
 );
}
```

The IconButton component, by utilizing the new syntax of ES6, the component is declared with arrow function with props as the parameter and only returning the React element using the data passed from the parent and not causing any side effects. The button styling is following BEM methodology for addressing which style modifier needs to be used:

```
'btn__container--navbar': inNavbar,
'btn__container--toolbar': !inNavbar,
```

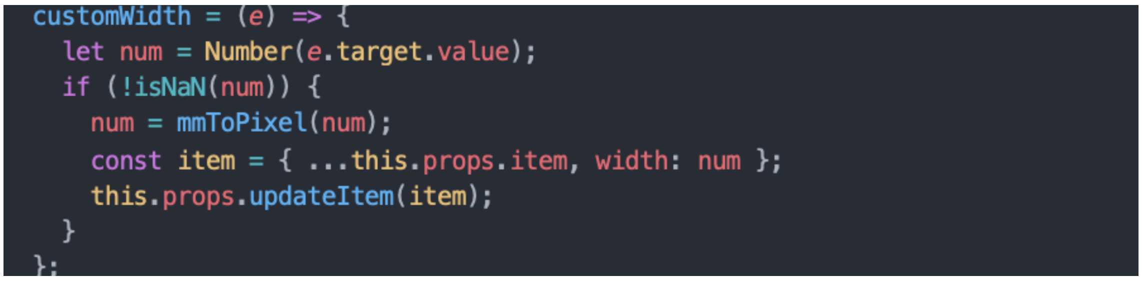
The customized button component is highly reusable due to its being a pure function, which is when provided with the same input, it always returns the same expected output.

Here is a different approach when implementing presentational component when life cycle is needed:

```
class labelItemsSettings extends Component {
 shouldComponentUpdate(nextProps) {
  if (
    this.props.item !== nextProps.item ||
    this.props.layout.marginTop !== nextProps.layout.marginTop ||
    this.props.layout.marginBottom !== nextProps.layout.marginBottom ||
    this.props.layout.marginLeft !== nextProps.layout.marginLeft ||
    this.props.layout.marginRight !== nextProps.layout.marginRight
  ) {
    return true;
  }
  return false;
 }
 customWidth = (e) => {
  let num = Number(e.target.value);
  if (!isNaN(num)) {
    num = mmToPixel(num);
    const item = { ...this.props.item, width: num };
    this.props.updateItem(item);
  }
 };
 customHeight = (e) => {
  let num = Number(e.target.value);
  if (!isNaN(num)) {
    num = mmToPixel(num);
    const item = { ...this.props.item, height: num };
    this.props.updateItem(item);
  }
};
customPositionLeft = (e) => {
  let num = Number(e.target.value);
  if (!isNaN(num)) {
    num = mmToPixel(num);
    const item = { ...this.props.item, x: num };
    this.props.updateItem(item);
  }
};
customPositionTop = (e) => {
  let num = Number(e.target.value);
  if (!isNaN(num)) {
    num = mmToPixel(num);
    const item = { ...this.props.item, y: num };
    this.props.updateItem(item);
  }
};

render() {
  const { t, item, variableGroup, barcodeVariables, codeToInfo } = this.props;
  const sizeInputs = classnames('labelItem__inputs', {
    'labelItem__inputs--line': false,
  });
  const itemSettings = (
    <React.Fragment>
      <div className="labelItem__position">
        <p className="labelItem__title margin-0">
          {t('lbl.positionFrom', 'Position from')}
        </p>
        <div className="labelItem__inputs">
          <CustomInput
            type="number"
            disabled={!item}
            onChange={this.customPositionLeft}
            value={
              item
                ? pixelToMm(Math.round(item.x)) === 0
                  ? ''
                  : pixelToMm(Math.round(item.x))
                : ''
            }
            inputSize="md"
            variant="vertical"
            labelText={t('lbl.leftMargin', 'Left')}
          />
          <CustomInput
            type="number"
            disabled={!item}
            onChange={this.customPositionTop}
            value={
              item
                ? pixelToMm(Math.round(item.y)) === 0
                  ? ''
                  : pixelToMm(Math.round(item.y))
                : ''
            }
            inputSize="md"
            variant="vertical"
            labelText={t('lbl.topMargin', 'Top')}
          />
        </div>
      </div>
      <div className="labelItem__size">
        <p className="labelItem__title margin-0">
          {t('lbl.size_itemOnLabel', 'Size')}
        </p>
        <div className={sizeInputs}>
          <CustomInput
            type="number"
            disabled={!item}
            onChange={this.customWidth}
            value={
              item
                ? pixelToMm(Math.round(item.width)) === 0
                  ? ''
                  : pixelToMm(Math.round(item.width))
                : ''
            }
            inputSize="md"
            variant="vertical"
            labelText={t('lbl.width', 'Width')}
          />
          <CustomInput
            type="number"
            disabled={!item || (item && item.itemType === 'line')}
            onChange={this.customHeight}
            value={
              item
                ? pixelToMm(Math.round(item.height)) === 0
                  ? ''
                  : pixelToMm(Math.round(item.height))
                : ''
            }
            inputSize="md"
```

```
          variant="vertical"
          labelText={t('lbl.height', 'Height')}
        />
      </div>
    </div>
  </React.Fragment>
);

return <React.Fragment>{itemSettings}</React.Fragment>;
}
}
```

This component is initialized using ES6 class syntax. Traditionally, the class component life cycle method starts with constructor but since it does not implement state or bind methods manually, it is not necessary to invoke the constructor method. The component styling is also using BEM, there are several local methods written in arrow function to convert the value before invoking the parent call backs.



```
customWidth = (e) => {
  let num = Number(e.target.value);
  if (!isNaN(num)) {
    num = mmToPixel(num);
    const item = { ...this.props.item, width: num };
    this.props.updateItem(item);
  }
};
```

Figure 17: Local event handler for changing width.

The life cycle used in the component is **shouldComponentUpdate**:

```
shouldComponentUpdate(nextProps) {
  if (
    this.props.item !== nextProps.item ||
    this.props.layout.marginTop !== nextProps.layout.marginTop ||
    this.props.layout.marginBottom !== nextProps.layout.marginBottom ||
    this.props.layout.marginLeft !== nextProps.layout.marginLeft ||
    this.props.layout.marginRight !== nextProps.layout.marginRight
  ) {
    return true;
  }
  return false;
}
```

Since the component performs side effects through its local method by implementing call backs from the parent container. There is a need to optimize the performance to avoid redundant re-renders, **shouldComponentUpdate** method determines whether the component's re-render progress is necessary when the component received new props or state [23]. **shouldComponentUpdate** default returning value is **true** which means if the data is changed, the component will always render a new version even if that data is not related to it. In the example above, the component checks for multiple needed props to decide whether it is crucial to build a new element or not.

All the components are created and ready to be implemented in the container for logic handling.

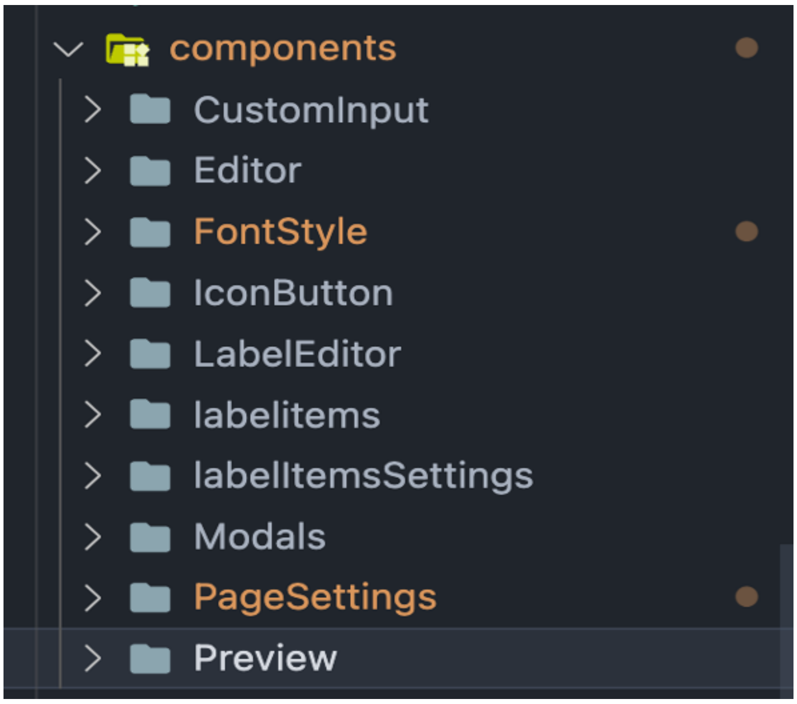
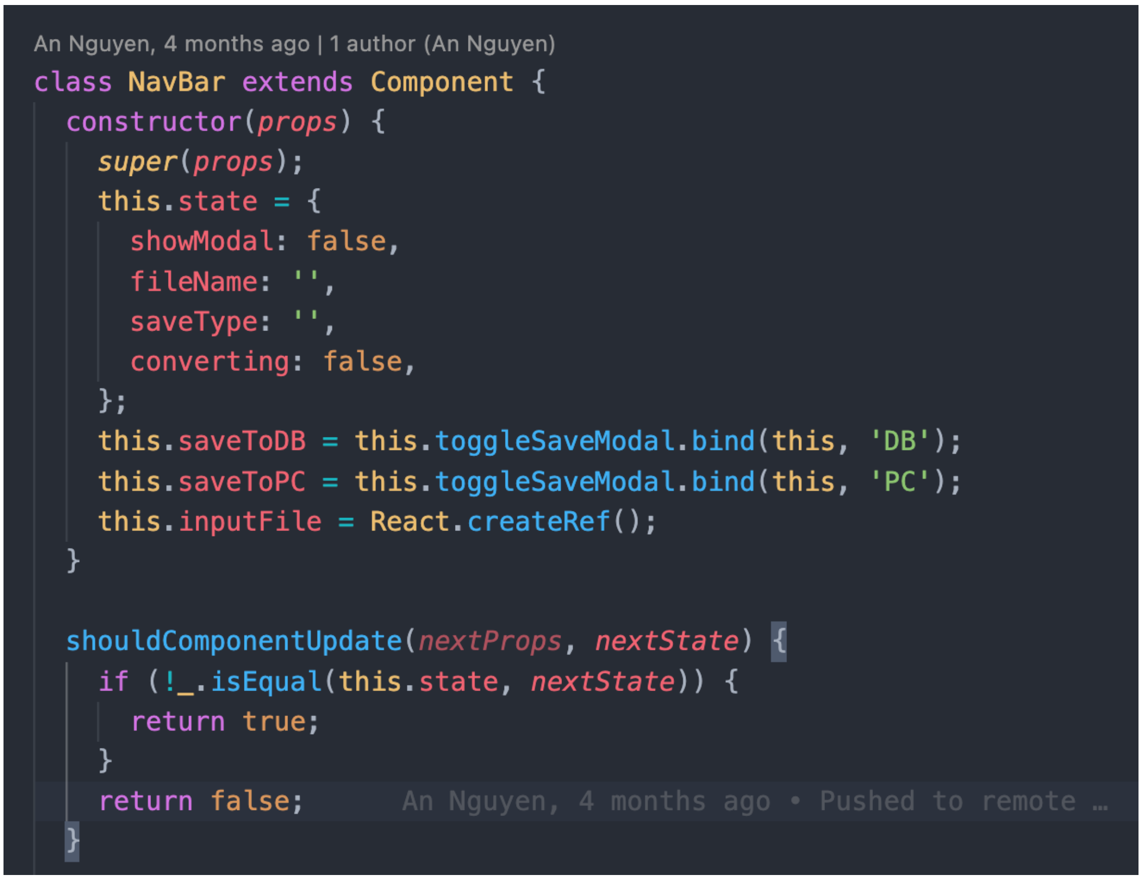


Figure 18: Structured reusable components.

### 3.2.3 Container Components

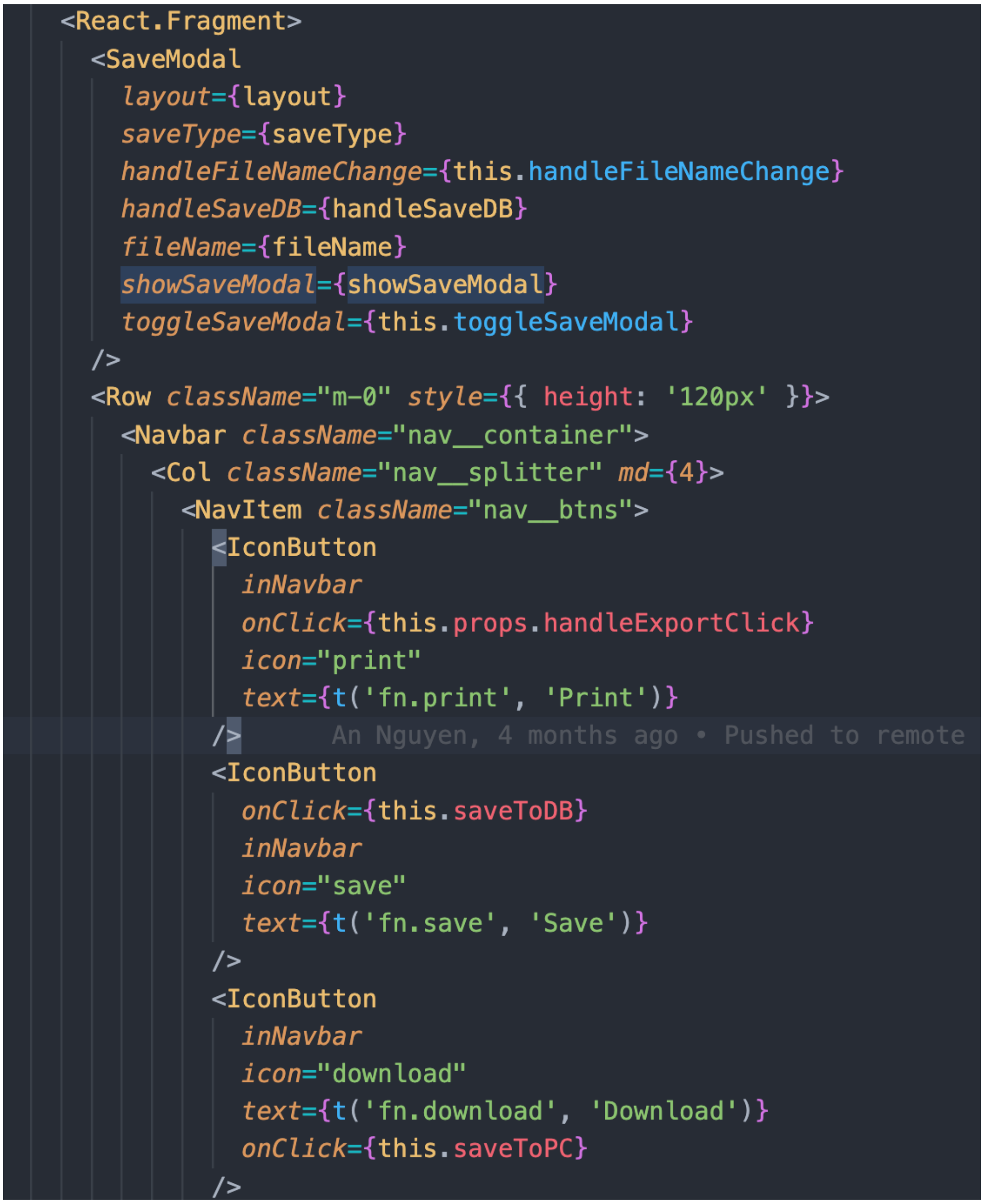
The implementation of **Navbar** container component:



```
An Nguyen, 4 months ago | 1 author (An Nguyen)
class NavBar extends Component {
  constructor(props) {
    super(props);
    this.state = {
      showModal: false,
      fileName: '',
      saveType: '',
      converting: false,
    };
    this.saveToDB = this.toggleSaveModal.bind(this, 'DB');
    this.saveToPC = this.toggleSaveModal.bind(this, 'PC');
    this.inputFile = React.createRef();
  }

  shouldComponentUpdate(nextProps, nextState) {
    if (!_.isEqual(this.state, nextState)) {
      return true;
    }
    return false;      An Nguyen, 4 months ago • Pushed to remote …
  }
```

Figure 19: Setting up state and life cycle method.

As described above, the container is created with class component and has its props data and own individual state object initializing under the **Constructor**. Moreover, the constructor is also the life cycle method where programmer binds the local methods of the container. This component also takes performance optimization into account by using the **shouldComponentUpdate** by a utility function from a library that deeply comparing the data of the previous and upcoming state to trigger the re-render.



```
<React.Fragment>
  <SaveModal
    layout={layout}
    saveType={saveType}
    handleFileNameChange={this.handleFileNameChange}
    handleSaveDB={handleSaveDB}
    fileName={fileName}
    showSaveModal={showSaveModal}
    toggleSaveModal={this.toggleSaveModal}
  />
  <Row className="m-0" style={{ height: '120px' }}>
    <Navbar className="nav__container">
      <Col className="nav__splitter" md={4}>
        <NavItem className="nav__btns">
          <IconButton
            inNavbar
            onClick={this.props.handleExportClick}
            icon="print"
            text={t('fn.print', 'Print')}
          />      An Nguyen, 4 months ago • Pushed to remote
          <IconButton
            onClick={this.saveToDB}
            inNavbar
            icon="save"
            text={t('fn.save', 'Save')}
          />
          <IconButton
            inNavbar
            icon="download"
            text={t('fn.download', 'Download')}
            onClick={this.saveToPC}
          />
```

Figure 20: Composition of other presentational components in the project.

The container is designed with composition, it does not have its own native React element but contain various other components. Containers, like **Navbar,** are mainly responsible for handling logic operations with the data of either one section that they are named after or the entire application through a centralized state manager. The state management system used in this project is the Redux.

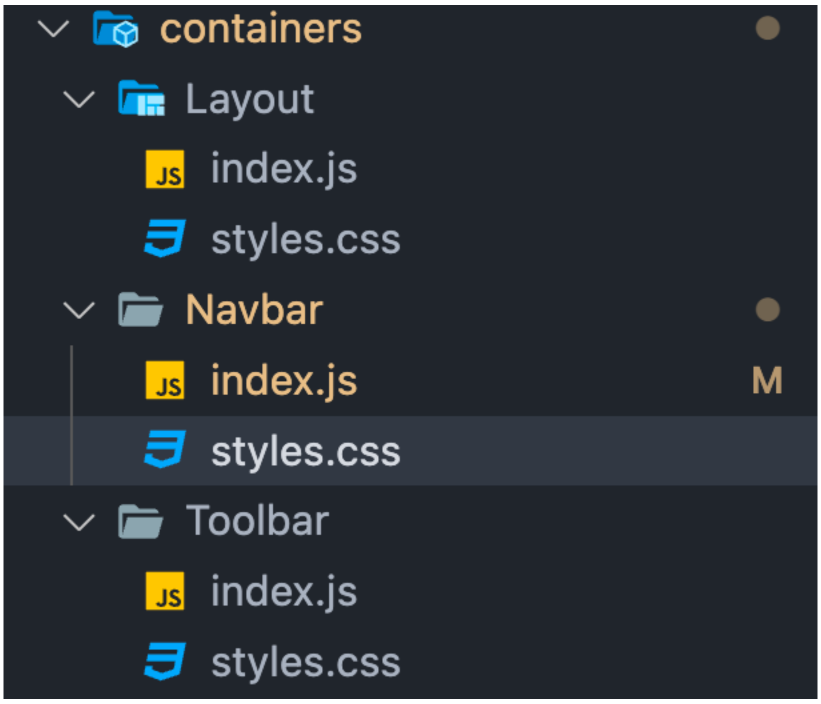


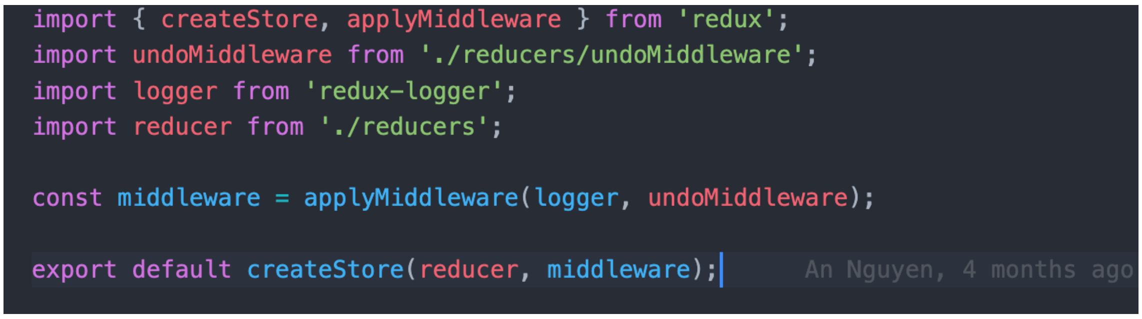
Figure 21: Container components with state.

### 3.2.4 Implementing Redux

As mentioned before, Redux consists of three main principles when implementing:

- Store: the only immutable central data source.
- Action: an object normally contains two properties, an instruction and data body.
- Reducer: a pure function which gets the partial or entire store for initial state, an action object as arguments and returns the expected state after performing mutation operations.

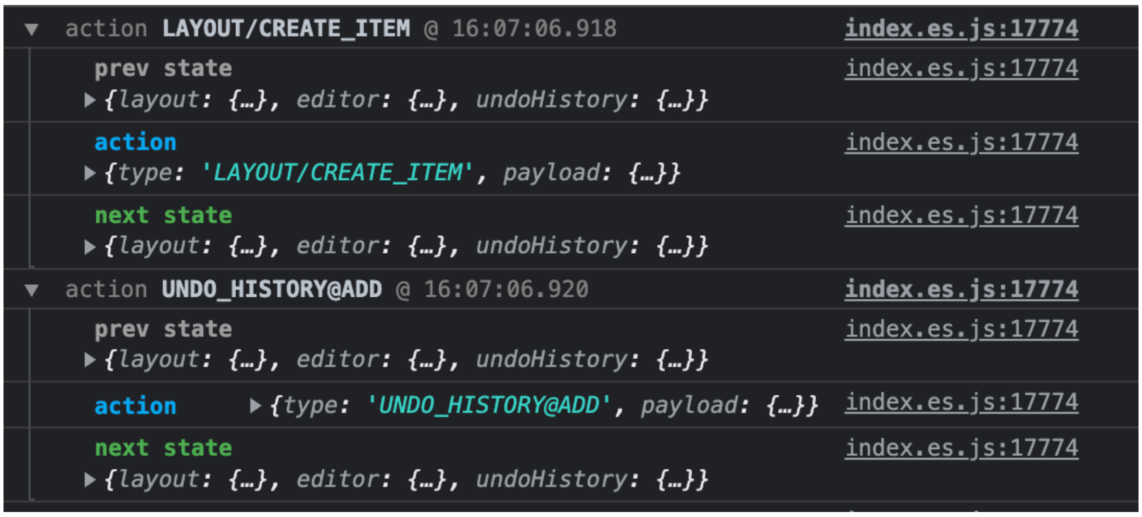
Firstly, the store creation:



```
import { createStore, applyMiddleware } from 'redux';
import undoMiddleware from './reducers/undoMiddleware';
import logger from 'redux-logger';
import reducer from './reducers';

const middleware = applyMiddleware(logger, undoMiddleware);

export default createStore(reducer, middleware);        An Nguyen, 4 months ago
```

Figure 22: Create Redux Store.

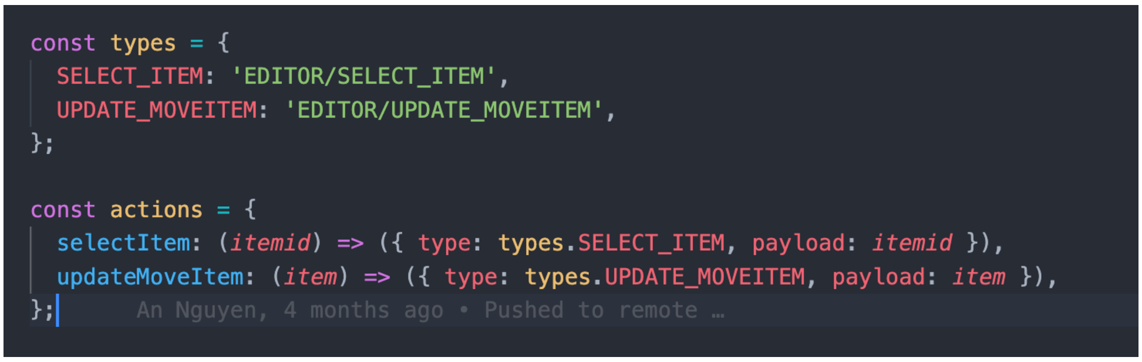
Redux library provides with several utility function to set up the project such as **createStore** and **applyMiddleware**. **createStore** establishes a state based on the provided reducer function and **StoreEnhancer** by combining multiple middleware using **applyMiddleware** [29]. What is middleware in Redux? Middleware functions as a point which programmer can perform more complex logic handling when the action is dispatched but not executed by the reducer [30]. The store has two middleware called **logger** and **undoMiddleware**, which is responsible for logging all side effects changes to the console and allowing undo dispatched action.



```
▼ action LAYOUT/CREATE_ITEM @ 16:07:06.918                index.es.js:17774
    prev state                                            index.es.js:17774
   ▶{layout: {…}, editor: {…}, undoHistory: {…}}
    action                                                index.es.js:17774
   ▶{type: 'LAYOUT/CREATE_ITEM', payload: {…}}
    next state                                            index.es.js:17774
   ▶{layout: {…}, editor: {…}, undoHistory: {…}}
▼ action UNDO_HISTORY@ADD @ 16:07:06.920                  index.es.js:17774
    prev state                                            index.es.js:17774
   ▶{layout: {…}, editor: {…}, undoHistory: {…}}
    action       ▶{type: 'UNDO_HISTORY@ADD', payload: {…}} index.es.js:17774
    next state                                            index.es.js:17774
   ▶{layout: {…}, editor: {…}, undoHistory: {…}}
```

Figure 23: Logging side effects and adding action to undo thanks to middleware.

Secondly, store’s actions implementation:

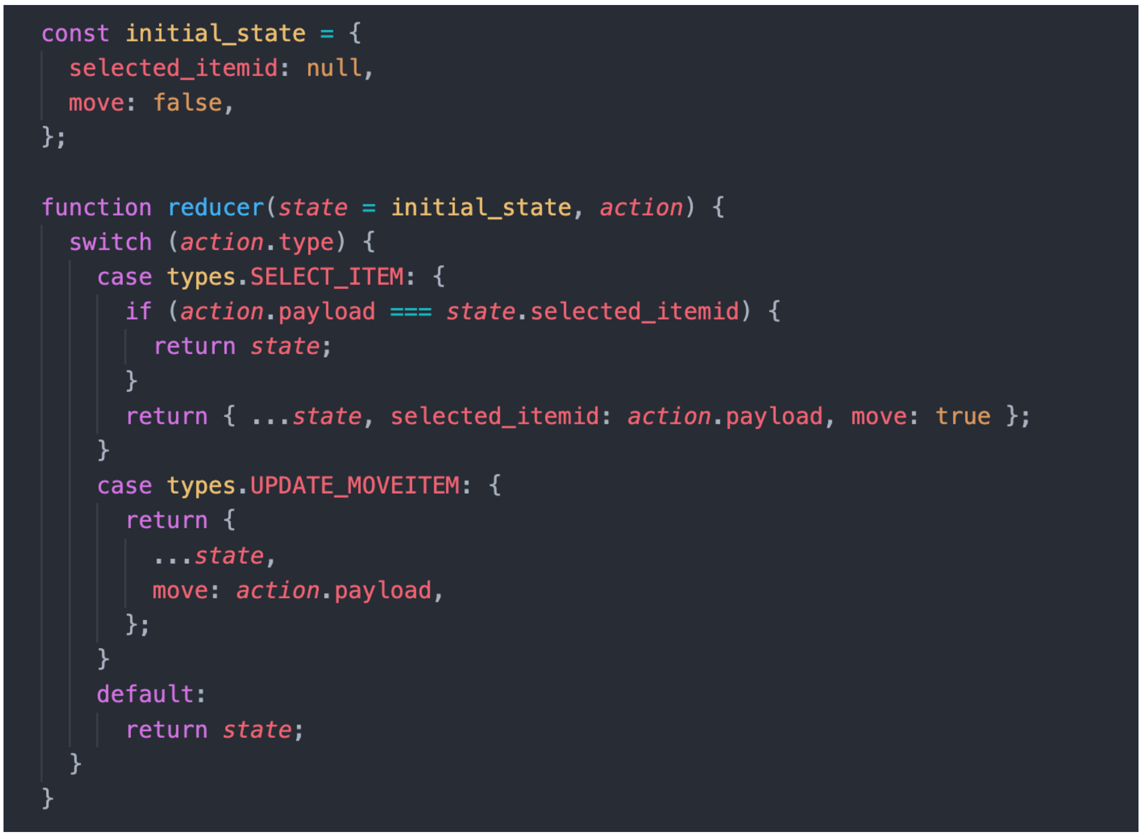


```
const types = {
  SELECT_ITEM: 'EDITOR/SELECT_ITEM',
  UPDATE_MOVEITEM: 'EDITOR/UPDATE_MOVEITEM',
};

const actions = {
  selectItem: (itemid) => ({ type: types.SELECT_ITEM, payload: itemid }),
  updateMoveItem: (item) => ({ type: types.UPDATE_MOVEITEM, payload: item }),
};      An Nguyen, 4 months ago • Pushed to remote …
```

Figure 24: Types and Actions in Editor reducer.

The instructions or types is a separated object of strings, and all the actions related to the reducer is put in the same file to allow scalability and avoid hard to debug typos.

Finally, the reducers initialization:

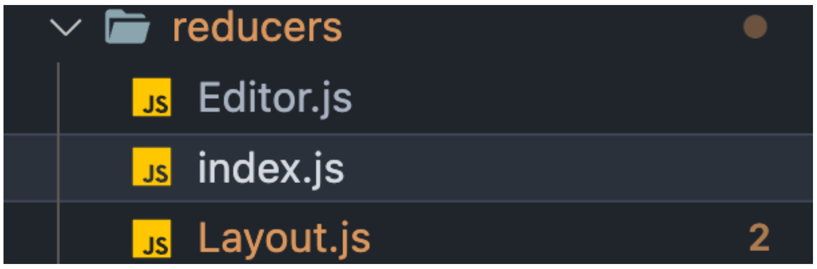


```
const initial_state = {
  selected_itemid: null,
  move: false,
};

function reducer(state = initial_state, action) {
  switch (action.type) {
    case types.SELECT_ITEM: {
      if (action.payload === state.selected_itemid) {
        return state;
      }
      return { ...state, selected_itemid: action.payload, move: true };
    }
    case types.UPDATE_MOVEITEM: {
      return {
        ...state,
        move: action.payload,
      };
    }
    default:
      return state;
  }
}
```

Figure 25: Editor Reducer.

After dispatching an action from the user interface, the reducer receives the action package along with the current state of the application as inputs then using switch case to handle the mutation based on the action's type.

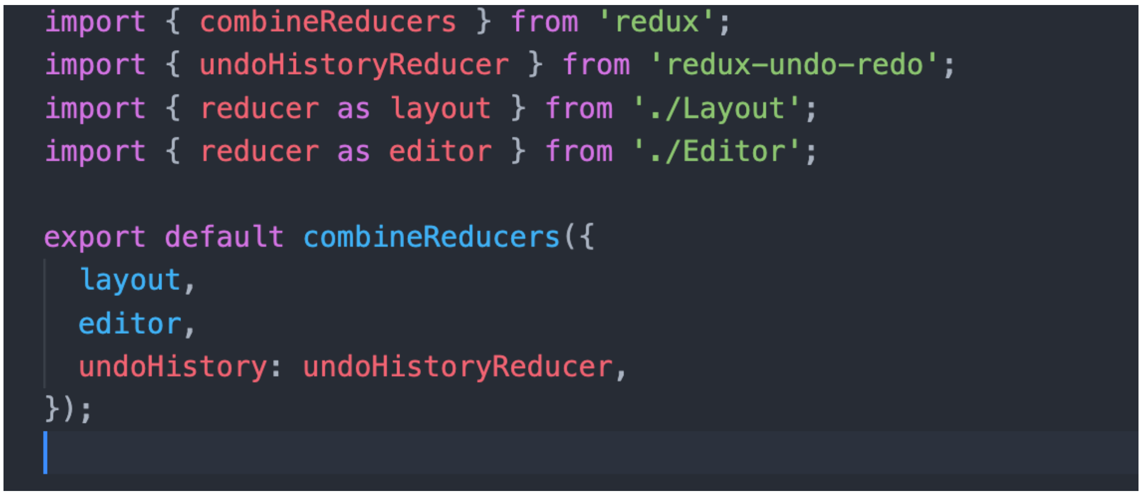
As the application grows, the reducer must handle many cases, a good solution addressing the issue is splitting one reducer into multiple reducers and then combining them before passing to the store as argument.



```
reducers
    Editor.js
    index.js
    Layout.js    2
```

Figure 26: Splitting one reducer into Editor and Layout for scalability development.

The combination happens in the **index.js** in the reducer folder.



```
import { combineReducers } from 'redux';
import { undoHistoryReducer } from 'redux-undo-redo';
import { reducer as layout } from './Layout';
import { reducer as editor } from './Editor';

export default combineReducers({
  layout,
  editor,
  undoHistory: undoHistoryReducer,
});
```

Figure 27: Combining reducers into one.

### 3.2.5 Higher Order Component

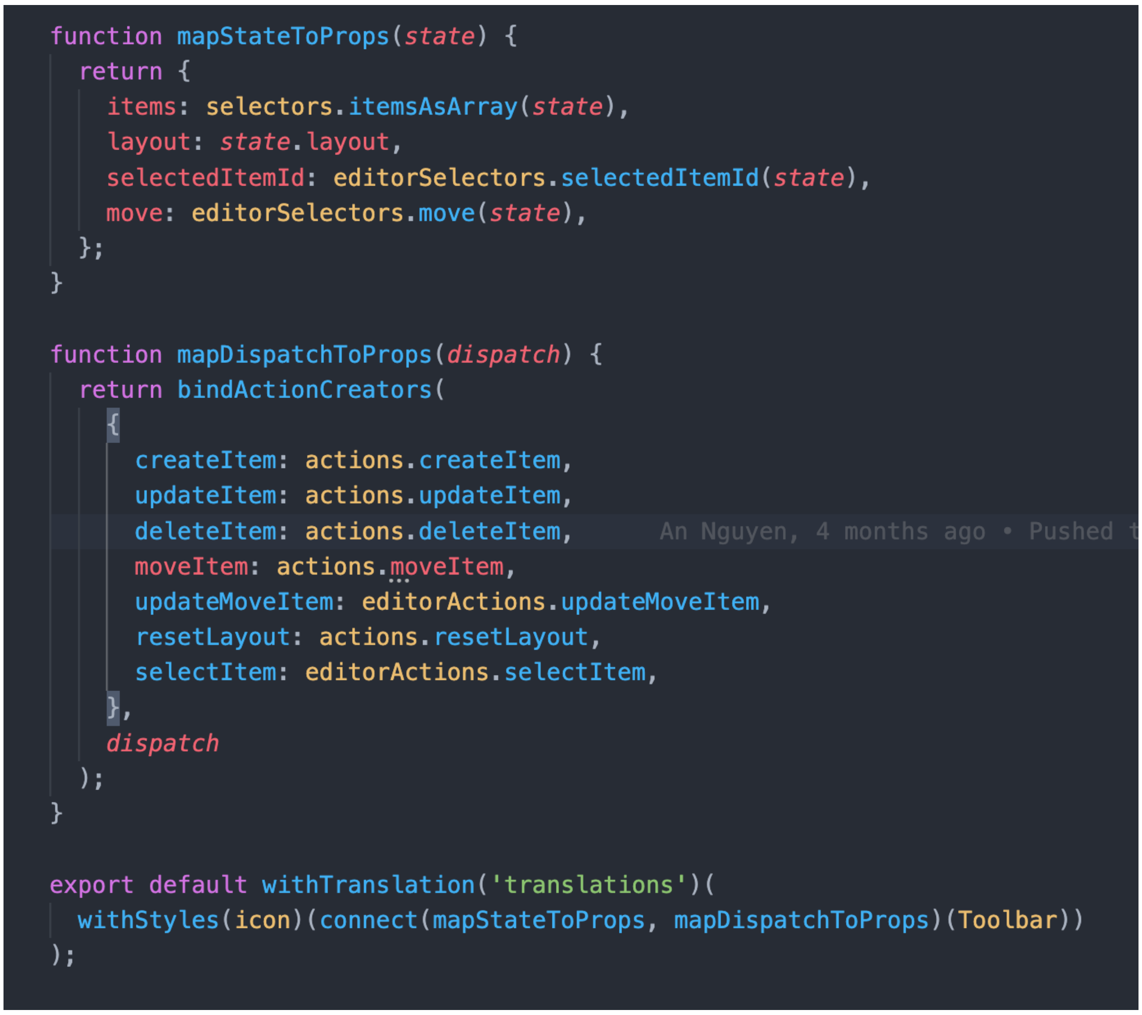
So far, the project has implemented all the components needed for displaying user interface, using composition to form block of related elements, handling logic and passing data from parent to child element. In the previous section, the Redux store, actions and reducers are up and running. The next step is connecting React components and Redux state together using higher order components provided by **React Redux** library.

What is higher order component? Higher Order Component (HOC) is a function which has a component as a required input and returning a new data and methods injected version of the component used as argument for the function [31].

To pass data from Redux store to a React component, the project uses the **connect()** from React Redux package. As the name suggested, the function provides the connection from component to store data and action dispatchers in Redux [32]. **mapStateToProps()** and **mapDispatchToProps()** are the main inputs of the **connect()** function:

- **mapStateToProps():** the function has the store data as the first parameter and return an object contains the reference of the Redux state in component props [32].
- **mapDispatchToProps():** the function with the connection to all actions dispatch method by default and hence the name implies, mapping the Redux dispatch functionality to the component props [32].

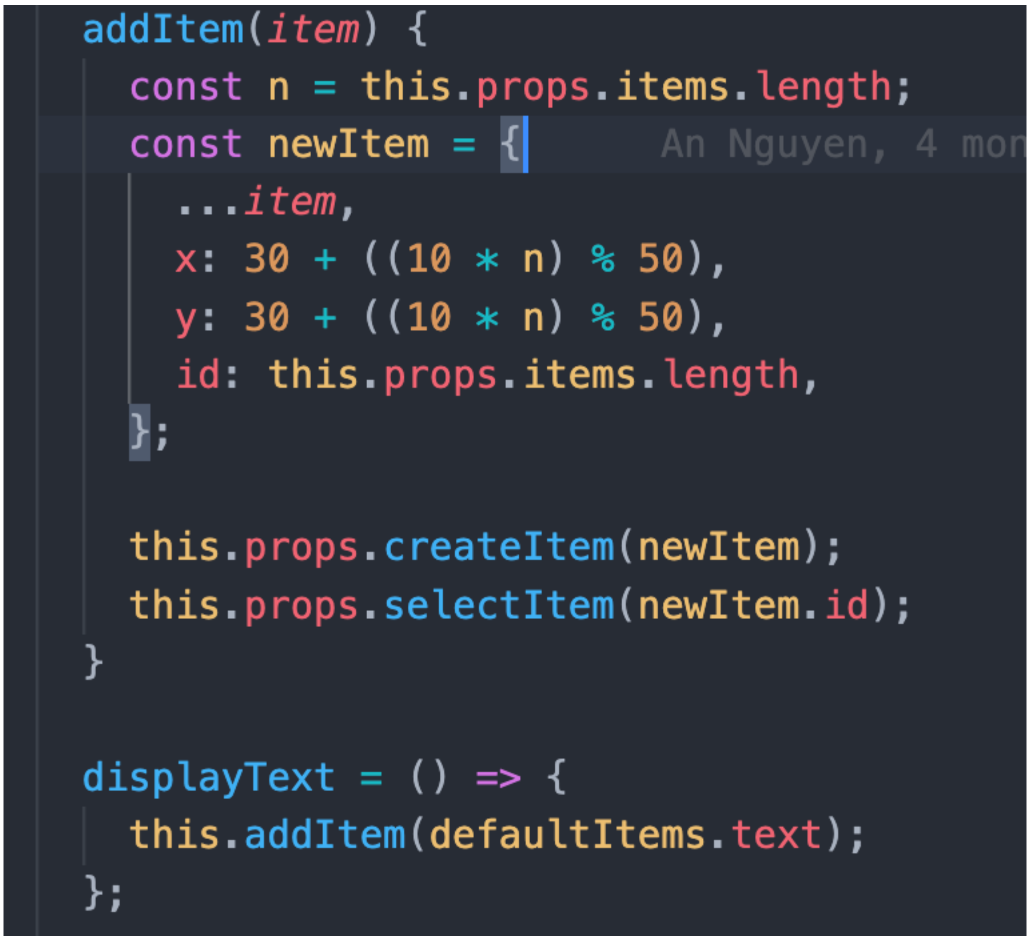
The connect function returns an HOC, a wrapper component, that passes the data and dispatch needed from the input component.



```
function mapStateToProps(state) {
  return {
    items: selectors.itemsAsArray(state),
    layout: state.layout,
    selectedItemId: editorSelectors.selectedItemId(state),
    move: editorSelectors.move(state),
  };
}

function mapDispatchToProps(dispatch) {
  return bindActionCreators(
    {
      createItem: actions.createItem,
      updateItem: actions.updateItem,
      deleteItem: actions.deleteItem,      An Nguyen, 4 months ago • Pushed t
      moveItem: actions.moveItem,
      updateMoveItem: editorActions.updateMoveItem,
      resetLayout: actions.resetLayout,
      selectItem: editorActions.selectItem,
    },
    dispatch
  );
}

export default withTranslation('translations')(
  withStyles(icon)(connect(mapStateToProps, mapDispatchToProps)(Toolbar))
);
```

Figure 28: Connecting Toolbar component with Redux.

The toolbar can access to the Redux store data and perform logical actions through its own props due to the mapping from the connect HOC and the mapping functions.

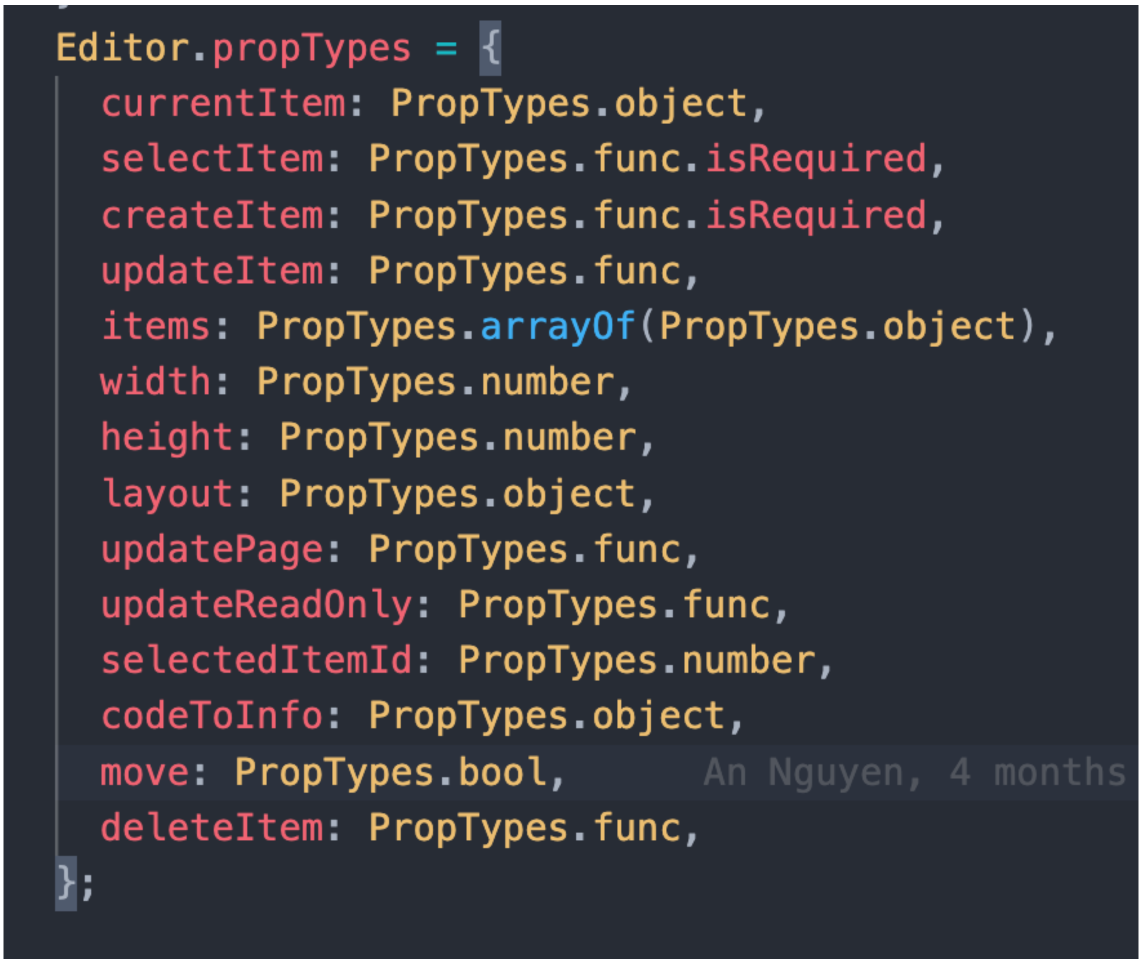


```
addItem(item) {
  const n = this.props.items.length;
  const newItem = {        An Nguyen, 4 mon
    ...item,
    x: 30 + ((10 * n) % 50),
    y: 30 + ((10 * n) % 50),
    id: this.props.items.length,
  };

  this.props.createItem(newItem);
  this.props.selectItem(newItem.id);
}

displayText = () => {
  this.addItem(defaultItems.text);
};
```

Figure 29: Accessing items and createItem method through props.

### 3.2.6 Prop Types

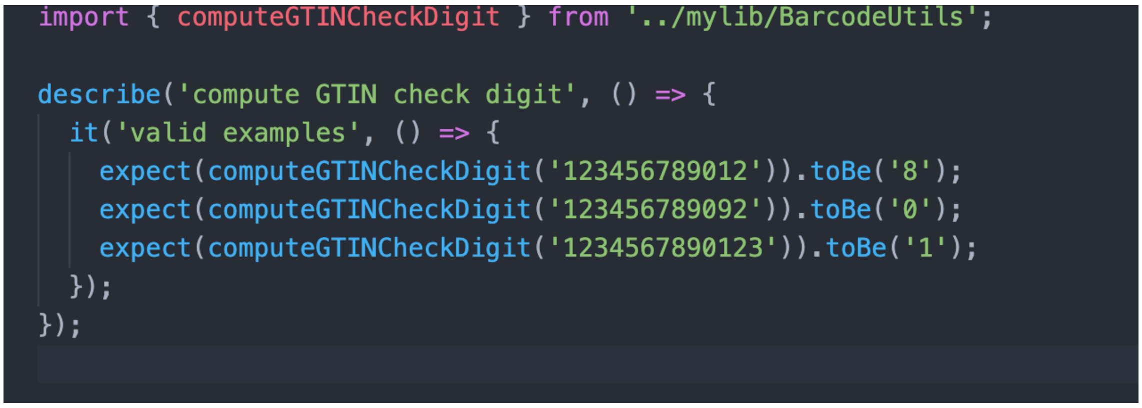
As the project gets more complex and with JavaScript being a typed dynamic language has its advantages and disadvantages [33]. Since the developers in the project uses Python daily for backend development, the dynamic variable can be utilized which means more time spending on developing the application. Nevertheless, the process of coding is always accompanied by creating bugs and without some sort of type checking. Consequently, it affects the scalability and causing unexpected bugs in production, that results in the team decides to use **PropTypes** for typing the props each component is expected to receive. This way, the programmer can still use the dynamic functionality to an extent while providing a basic safety measure when developing the component.



```
Editor.propTypes = {
  currentItem: PropTypes.object,
  selectItem: PropTypes.func.isRequired,
  createItem: PropTypes.func.isRequired,
  updateItem: PropTypes.func,
  items: PropTypes.arrayOf(PropTypes.object),
  width: PropTypes.number,
  height: PropTypes.number,
  layout: PropTypes.object,
  updatePage: PropTypes.func,
  updateReadOnly: PropTypes.func,
  selectedItemId: PropTypes.number,
  codeToInfo: PropTypes.object,
  move: PropTypes.bool,       An Nguyen, 4 months
  deleteItem: PropTypes.func,
};
```

Figure 30: PropTypes for Editor component.

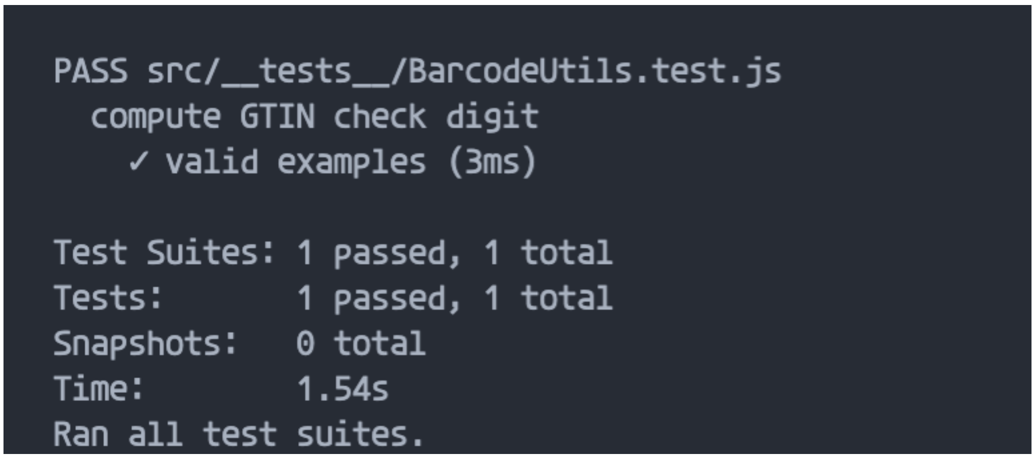
Project create with Create React App is automatically installed **Jest** as test runner. Tests are located under **__tests__** folder which runs with bash command.



```
import { computeGTINCheckDigit } from '../mylib/BarcodeUtils';

describe('compute GTIN check digit', () => {
  it('valid examples', () => {
    expect(computeGTINCheckDigit('123456789012')).toBe('8');
    expect(computeGTINCheckDigit('123456789092')).toBe('0');
    expect(computeGTINCheckDigit('1234567890123')).toBe('1');
  });
});
```

Figure 31: Test barcode with Jest.

The example above shows how to structure the test. Firstly, **describe()** is the outmost block, it groups all the related tests together. Secondly, each test starts with **it()**, the function contains the actual implementation of the tests using utility function and Jest API to assert the outcomes.



```
PASS src/__tests__/BarcodeUtils.test.js
  compute GTIN check digit
    ✓ valid examples (3ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.54s
Ran all test suites.
```

Figure 32: Test results.

At the end of this chapter, the application’s user interface and logic have been successfully structured and implemented in a scalable way. However, the project can only be built into a production application, the goal is to create a library that could also be used in other React application.

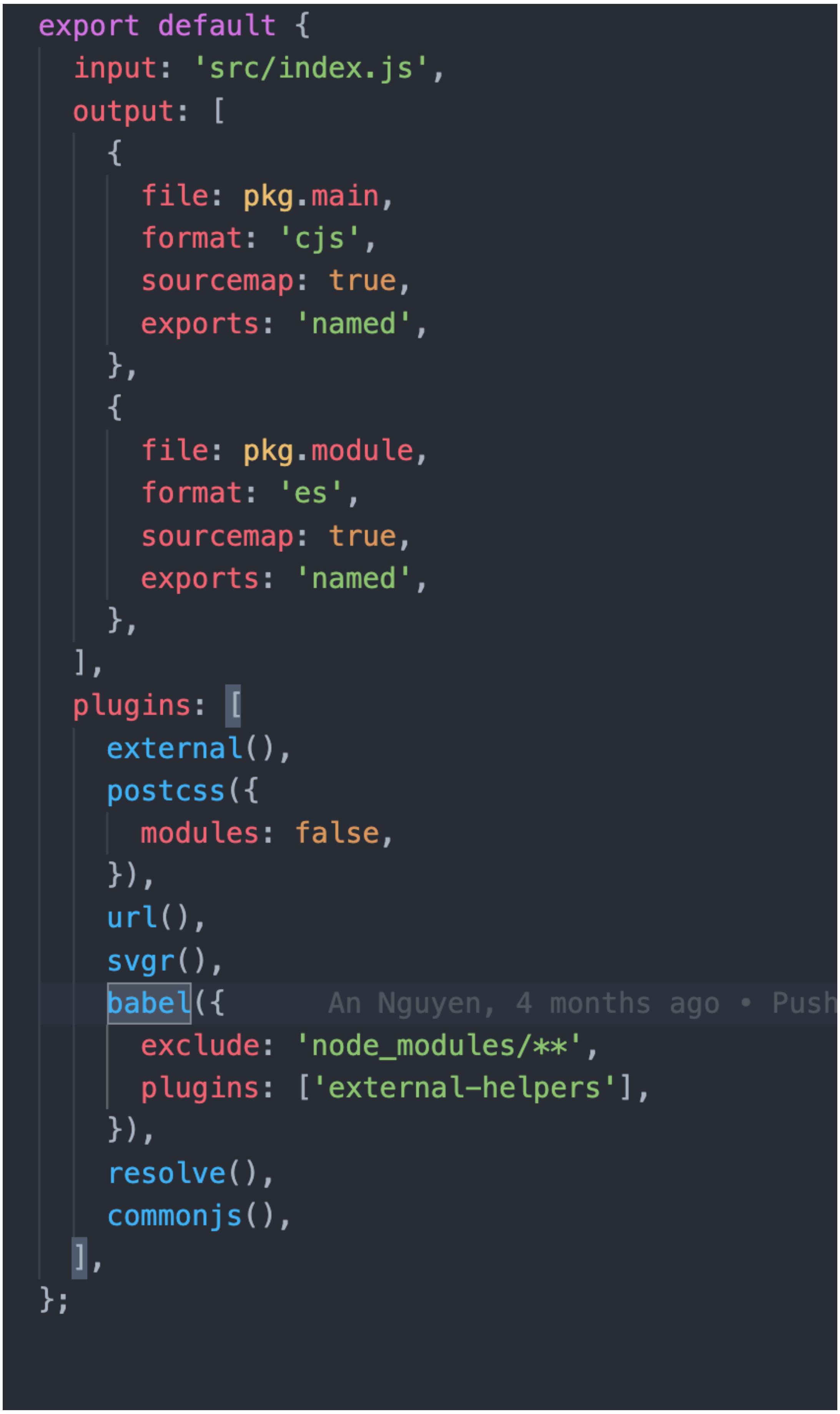
## 3.3 Bundling library

### 3.3.1 Setting up environment & scripts

To generate a reusable library, the project code must move to an environment where the compiling and bundling processes are handle by **Rollup.js**.

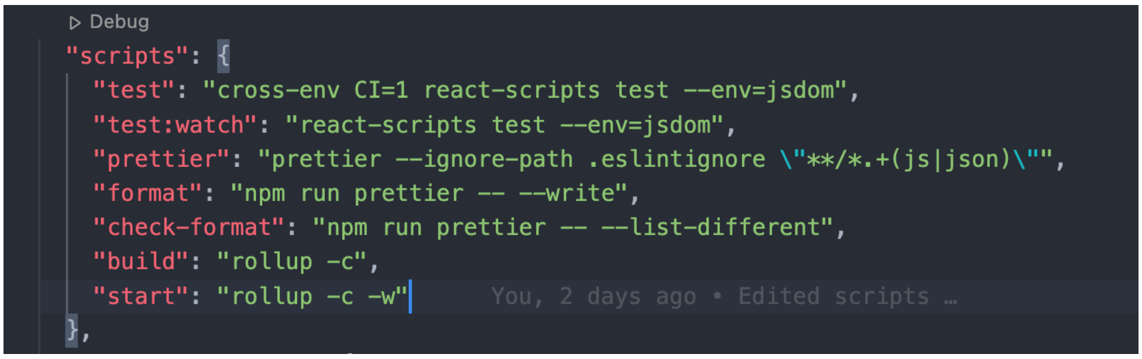
Changing the directory to outside of the React project then setting a new project by following these steps:

- Installing Rollup.js in the local project with command line.
- Creating a new folder called **src** and new file in that folder called **index.js**
- Creating a new **package.json** file and add the build script.
- Adding a customized Rollup config file.
- Installing Rollup necessary plugins such as babel for transpiling code, postcss for import styling files, etc.
- Modifying the Rollup config file accordingly.
- Moving the React code to the newly created **src** folder.



```
export default {
  input: 'src/index.js',
  output: [
    {
      file: pkg.main,
      format: 'cjs',
      sourcemap: true,
      exports: 'named',
    },
    {
      file: pkg.module,
      format: 'es',
      sourcemap: true,
      exports: 'named',
    },
  ],
  plugins: [
    external(),
    postcss({
      modules: false,
    }),
    url(),
    svgr(),
    babel({       An Nguyen, 4 months ago • Push
      exclude: 'node_modules/**',
      plugins: ['external-helpers'],
    }),
    resolve(),
    commonjs(),
  ],
};
```

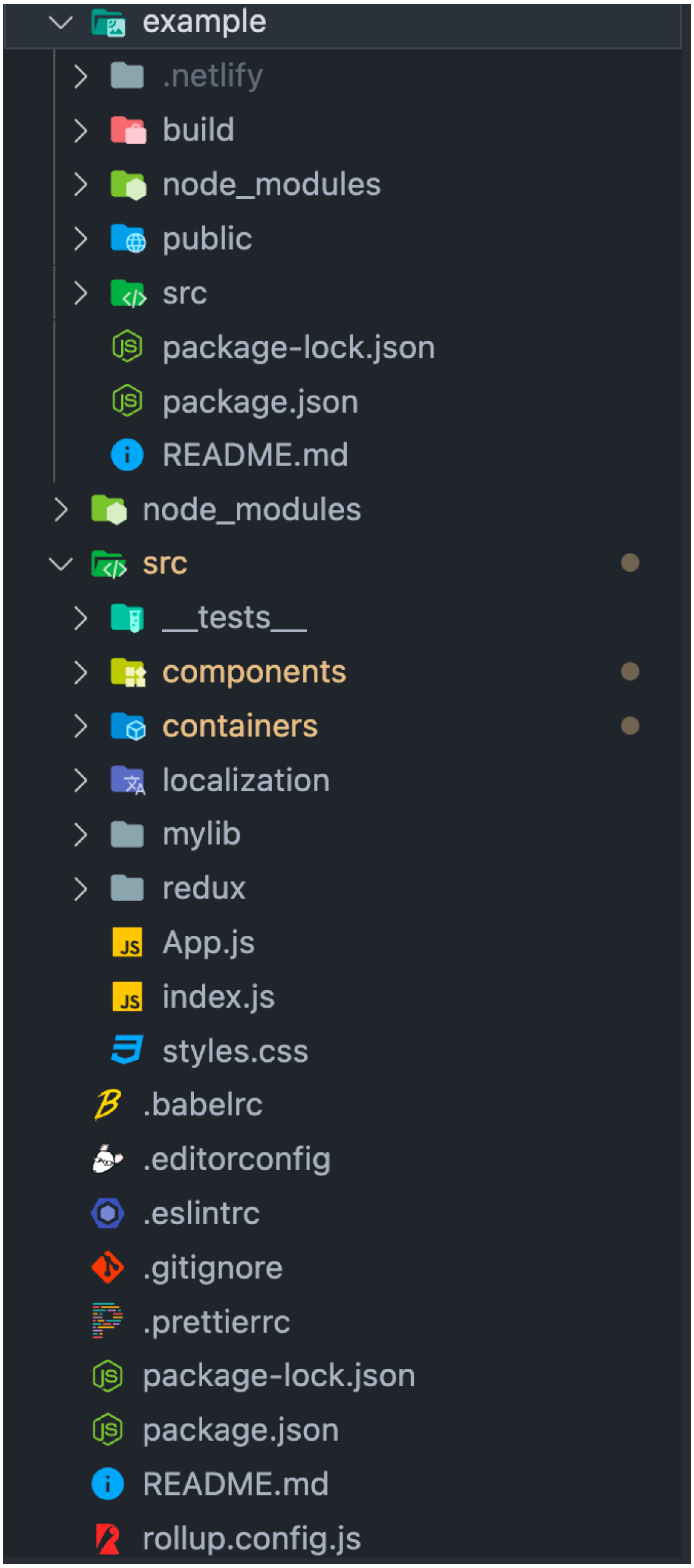
Figure 33: Rollup.config.js



```
▷ Debug
"scripts": {
  "test": "cross-env CI=1 react-scripts test --env=jsdom",
  "test:watch": "react-scripts test --env=jsdom",
  "prettier": "prettier --ignore-path .eslintignore \"**/*.+(js|json)\"",
  "format": "npm run prettier -- --write",
  "check-format": "npm run prettier -- --list-different",
  "build": "rollup -c",
  "start": "rollup -c -w"      You, 2 days ago • Edited scripts …
},
```

Figure 34: Package.json scripts.

The folder tree after the implementation of the new bundler where the contents inside the example folder are from Create React App, this place functions as a testing React application importing the built library:



```
example
  .netlify
  build
  node_modules
  public
  src
  package-lock.json
  package.json
  README.md
node_modules
src
  __tests__
  components
  containers
  localization
  mylib
  redux
  App.js
  index.js
  styles.css
.babelrc
.editorconfig
.eslintrc
.gitignore
.prettierrc
package-lock.json
package.json
README.md
rollup.config.js
```

Figure 35: New folder structure.

Since the library is used with internal React application, the essential dependencies such as React, Redux are installed in the parent project rather than in the library. The dependencies needed to develop the library is moved to **peerDependencies** in the **package.json** file since the other application will provides those.

### 3.3.2 Testing integration with React application

With the addition of new bundler, now, whenever in root folder run

```
Npm start
```

The library builds the application from the file contents of **index.js** file to **dist** folder, then it enters watch mode, which means any new changes will trigger the rebuilt of the library.

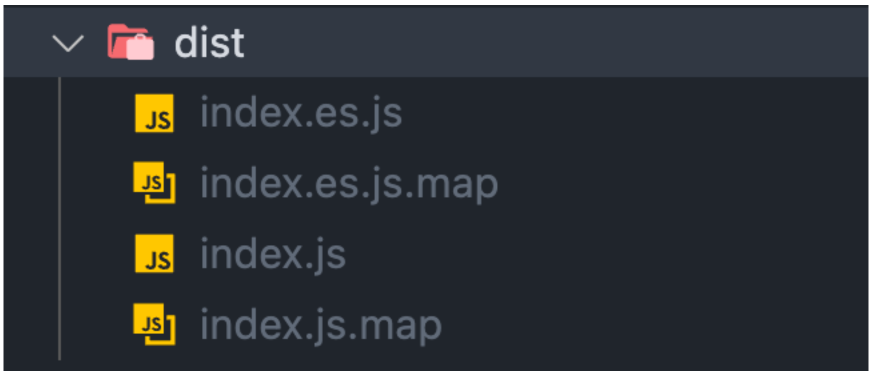
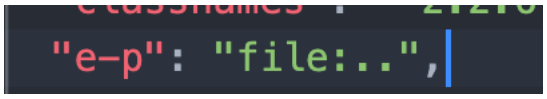

dist
index.es.js
index.es.js.map
index.js
index.js.map

Figure 36: Dist folder contains builds of the library.

Since the project is for internal usage and does not need to be publish. To use the library, navigating to the **package.json** file in the **example** folder, adding the dependency name with the path to the **dist** folder.



```
"e-p": "file:..",
```

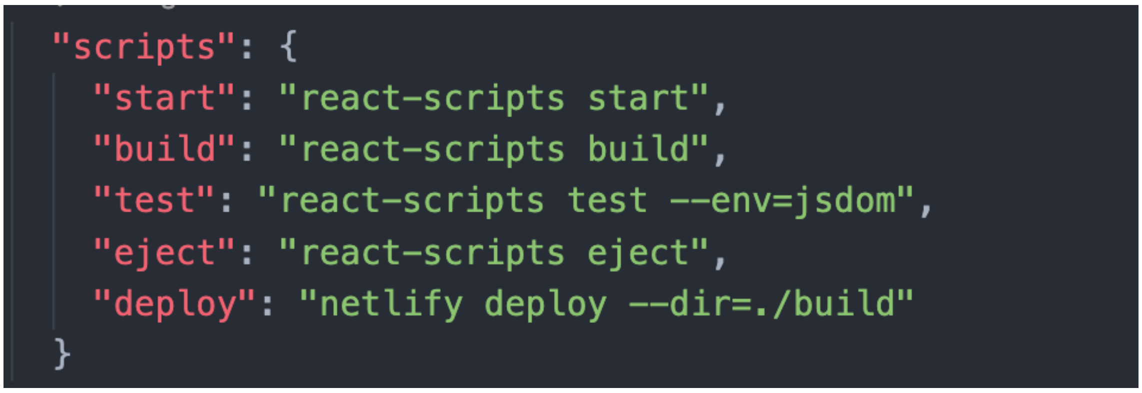
Figure 37: Add local library to React app.

Importing the package into the React application and start the local development. The application is running on the local server and serving the user interface in the browser.

## 3.4 Implementing workflows with GitHub Actions

Before setting up the workflow, there is a need to integrate the site from hosting service with the project repository.

- Install Netlify CLI through terminal.
- Log in to Netlify and generate personal access token for the project on the website.
- Create new site and copy the site ID.
- Use Netlify CLI to set up the connection between the site and the project.
- Add a script for deploy scripts in the package.json file in React application folder, this allows for manual deployment to the preview-built site with unique URL and can be accessed in Netlify's site.
- Navigate to the project repository on GitHub, under Settings, fill the site ID and personal access token to the Secrets section.



```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test --env=jsdom",
  "eject": "react-scripts eject",
  "deploy": "netlify deploy --dir=./build"
}
```

Figure 38: Adding deploy script to React application.

To create a GitHub Actions workflow, the project must have a folder called **.github** and inside that folder, a **workflow** folder also needs to be generated.

Adding the project workflow with a **YML** file extension, the developer writes customized for automating the build, test and deploy process. The workflow runs the job called **build-test-deploy** when there is a **push** event occurs in the **main** branch of the repository. Firstly, the chosen OS to operate the steps is **ubuntu-latest**, then it executes the steps as design. Secondly, it performs a checkout on the repository, sets up node version for the machine. Finally, it dispatches the library build and test actions, builds the React testing application and then deploys to Netlify.

```yaml
name: CI & CD
on:
  push:
    branches: [main]

jobs:
  build-test-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set-up Node
        uses: actions/setup-node@v1
        with:
          node-version: '16.x'
      - name: Build library
        run: |
          npm install --legacy-peer-deps
          npm run test
          npm run build
      - name: Build React App
        run: |
          cd example
          npm install --legacy-peer-deps
          npm run build
      - name: Deploy
        run: |
          ./node_modules/.bin/netlify deploy --prod --dir=./example/build --
site ${{secrets.netlify_site_id}} --auth
${{secrets.netlify_personal_access_token}}
```

# 4 Conclusion

The main objective of the thesis was to show the development and bundling process of the label customizing and printing library with comprehensive and automated workflow for a Finnish paint company. Both the main requests for a scalability library and working smoothly with the main application are met with satisfaction from Company C. The project’s code base is easy to read, following good methodologies and has auto build and deploy system. The challenges rooting from the previous application is written for desktop usage with complex logic for label creation so there is a need for researching thoroughly through the technology stack and user interface organization in web application. The point-of-sale development team can now integrate the label printing library into their application with ease and later, connecting to the label printer to generate well structure and diverse labels for their paint product.

However, due to the majority time spending on migrating desktop features into web application, the testing coverage of the project is not as expected from the development team. Although, the workflow produces a unique preview built of the library when any features are developed and bugs are fixed, the lack of tests not only allowing unexpected bug to sometimes appear in production but also making the onboard process of new developer becomes more time consuming.

To be able to ship the library with confidence, more test cases are needed as well as enforcing type checking more strictly. Since the library will be implemented as part of a bigger application, further development should take accessibility and the design of the other application into account.

## References

1 MDN Web Docs. HTML [internet]. 2022 May 2 [cited 2022 May 4]. Available from: https://developer.mozilla.org/en-US/docs/Web/HTML.

2 World Wide Web Consortium. HTML & CSS [internet]. 2016 [cited 2022 May 4]. Available from: https://www.w3.org/standards/webdesign/htmlcss#whathtml.

3 MDN Web Docs. JavaScript [internet]. 2022 April 2022 [cited 2022 May 4]. Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

4 Meta Platforms, Inc. React [internet]. 2022 [cited 2022 May 4]. Available from: https://github.com/facebook/react.

5 Facebook, Inc. Create React App [internet]. 2022 [cited 2022 May 4]. Available from: https://create-react-app.dev/.

6 OpenJS Foundation. Node.js [internet]. 2022 [cited 2022 May 4]. Available from: https://nodejs.org/en/.

7 OpenJS Foundation. About Node.js [internet]. 2022 [cited 2022 May 4]. Available from: https://nodejs.org/en/about/.

8 Npm, Inc. About npm [internet]. 2022 [cited 2022 May 4]. Available from: https://www.npmjs.com/about

9 Rollup.js Team. Rollup.js [internet]. 2022 [cited 2022 May 4]. Available from: https://github.com/rollup/rollup.

10 Abramov D and the Redux documentation authors. Redux [internet]. 2022 [cited 2022 May 4]. Available from: https://redux.js.org/.

11 Abramov D and the Redux documentation authors. Three Principles [internet]. 2021 June 25 [cited 2022 May 4]. Available from: https://redux.js.org/understanding/thinking-in-redux/three-principles.

12 Abramov D and the Redux documentation authors. Redux essentials, Part 1: Redux Overview and Concepts [internet]. 2022 February 19 [cited 2022 May 4]. Available from: https://redux.js.org/tutorials/essentials/part-1-overview-concepts.

13 Facebook Inc. Jest [internet]. 2022 [cited 2022 May 4]. Available from: https://jestjs.io/.

14 GitHub Inc. Understanding GitHub Actions [internet]. 2022 [cited 2022 May 4]. Available from: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions.

15 Netlify. One workflow. One platform [internet]. 2022 [cited 2022 May 5]. Available from: https://www.netlify.com/products/.

16 World Wide Web Consortium. What is the Document Object Model? [internet]. 2022 [cited 2022 May 5]. Available from: https://www.w3.org/TR/WD-DOM/introduction.html.

17 Peyrott S. React Virtual DOM vs Incremental DOM vs Ember's Glimmer: Fight. 2015 November 20 [cited 2022 May 5]. Available from: https://auth0.com/blog/face-off-virtual-dom-vs-incremental-dom-vs-glimmer/.

18 Meta Platforms, Inc. Virtual DOM and Internals. 2022 [cited 2022 May 5]. Available from: https://reactjs.org/docs/faq-internals.html.

19 Meta Platforms, Inc. Reconciliation. 2022 [cited 2022 May 5]. Available from: https://reactjs.org/docs/reconciliation.html.

20 Meta Platforms, Inc. Components and Props. 2022 [cited 2022 May 5]. Available from: https://reactjs.org/docs/components-and-props.html.

21 Meta Platforms, Inc. Rendering Elements. 2022 [cited 2022 May 5]. Available from: https://reactjs.org/docs/rendering-elements.html.

22 Meta Platforms, Inc. State and Lifecycle. 2022 [cited 2022 May 5]. Available from: https://reactjs.org/docs/state-and-lifecycle.html.

23 Meta Platforms, Inc. React.Component. 2022 [cited 2022 May 5]. Available from: https://reactjs.org/docs/react-component.html.

24 Maj W. React Lifecycle Methods diagram. 2022 [cited 2022 May 5]. Available from: https://github.com/wojtekmaj/react-lifecycle-methods-diagram.

25 Rendle R. BEM 101. 2015 April 2 [cited 2022 May 5]. Available from: https://css-tricks.com/bem-101/.

26 IBM Cloud Education. What Are CI/CD and the CI/CD Pipeline. 2021 September 27 [cited 2022 May 6]. Available from: https://www.ibm.com/cloud/blog/ci-cd-pipeline.

27 GitLab, Inc. CI/CD concepts. 2022 [cited 2022 May 6]. Available from: https://docs.gitlab.com/ee/ci/introduction/.

28 Abramov D. Presentational and Container Components. 2015 March 23 [cited 2022 May 6]. Available from: https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0.

29 Abramov D and the Redux documentation authors. createStore. 2021 June 26 [cited 2022 May 7]. Available from: https://redux.js.org/api/createstore.

30 Abramov D and the Redux documentation authors. Middleware 2021 June 25 [cited 2022 May 7]. Available from: https://redux.js.org/understanding/history-and-design/middleware.

31 Meta Platforms, Inc. Higher-Order Components. 2022 [cited 2022 May 7]. Available from: https://reactjs.org/docs/higher-order-components.html.

32 Abramov D and the Redux documentation authors. connect(). 2022 April 4 [cited 2022 May 7]. Available from: https://react-redux.js.org/api/connect.

33 Meta Platforms, Inc. Typechecking with PropTypes. 2022 [cited 2022 May 8]. Available from: https://reactjs.org/docs/typechecking-with-proptypes.html.