Sami Sikkilä

Data Driven User Interfaces with React

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Bachelor's Thesis

15 November 2021

Abstract

Author:	Sami Sikkilä
Title:	Data Driven User Interfaces with React
Number of Pages:	43 pages + 0 appendices
Date:	15 November 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Software Engineering
Supervisors:	Matti Oosi, Principal Lecturer

The main purpose of the research was to study JavaScript and React. React is one of the most popular and fastest growing front-end programming libraries.

React is developed by Facebook, which recently became to be known as Meta, originally to be used in their internal front-end developing. Later it was made into open source, so everybody gained access and its popularity started growing rapidly. It also made it possible for more people to take part in developing it further and produce more features to it. React is currently considered by many to be the go-to library when creating a user interface that needs to deal with a lot of data and users.

This thesis investigates many of the fundamental aspects of technologies such as React, Redux and JavaScript. After going through these features there is an example application introduced that is meant to be making use of most if not all these fundamental aspects.

The example application has a typical React-application structure with different components meant to be working together to display the intended result in the browser. The application is a shopping list that takes in the items that are to be put on the list as input and then shows it in the browser.

Keywords: ReactJS, Redux, JSX, JavaScript, HTML, Front-end, DOM, Framework, Library, Component, Action, Reducer

Tiivistelmä

Tekijä:	Sami Sikkilä
Otsikko:	Data driven user interfaces using React
Sivumäärä:	43 sivua + 0 liitettä
Aika:	15.11.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Lehtori Matti Oosi

Tämän opinnäytetyön pääasiallisena tarkoituksena oli opiskella Reactia, JavaScriptiä sekä Reduxia. React on yksi suosituimmista ja nopeimmin kasvavista käyttöliittymien ohjelmoimiseen käytettävistä kirjastoista.

React on Facebookin, joka vaihtoin juuri nimensä Metaksi, kehittämä käyttöliittymien ohjelmointiin kehittämä kirjasto. Alunperin React kehitettiin Facebookin omaan sisäiseen käyttöön, mutta myöhemmin se muutettiin avoimeksi lähdekoodiksi, jolloin siihen pääsi käsiksi kaikki halukkaat. Muutoksen jälkeen Reactin kehittämiseen on osallistunut Facebookin lisäksi useita muita, jopa itsenäisiä, kehittäjiä ja toiminnallisuuksien määrä on kasvanut. Useat ohjelmoijat ja asiantuntijat pitävät Reactia yhtenä parhaista kirjastoista, jos kehitettävän käyttöliittymän tarvitsee käsitellä suuria määriä tietoa ja käyttäjiä kerrallaan.

Tässä opinnäytettyössä käydään läpi monia tärkeitä osa-alueita teknologioista kuten React, Redux ja JavaScript. Näiden läpikäynnin jälkeen opinnäytettyön lopussa on yksinkertainen esimerkki applikaatio, jossa pyritään käyttämään näitä ominaisuuksia.

Esimerkki applikaatiossa on perineinen React-applikaation rakenne, jossa luodaan erilaisia komponentteja joiden tulee toimia yhdessä ja esittää käyttäjän sille syöttämät asiat selaimessa. Applikaation ideana on toimia ostoslistana.

Avainsanat: ReactJS, Redux, JSX,JavaScript,HTML ,DOM,Framework, Library, Component, Action, Reducer

Contents

List of Abbreviations

API:	Application Program Interface
DOM:	Document Object Model
HTML:	Hypertext Markup Language
JS:	JavaScript
JSX:	JavaScript XML
URL:	Uniform Resource Locator
HTTP:	Hypertext Transfer Protocol

1 Introduction

JavaScript has become one of the mainly used programming languages in computer science and especially in web development. The syntax of JavaScript is easily readable and understandable.

React is a growingly popular front-end JavaScript library and it is developed by Facebook, and first released in 2013. React was developed to provide a way for developers to create interactive user interfaces in an easier fashion. Interactive user interface means that when a state of the application changes it will render the application again and so user input can be seen immediately affecting the display.

This thesis talks about different elements and components of React and the goal is to provide a decent set of information to start working with React. This thesis is to be used as a guideline on which the actual, more specified guide to React or Redux could be written. The thesis covers the most essential elements of React.

The thesis first discusses theory of React and other related technologies and in the end, there is a simple example application introduced where the technologies were taken into use.

2 React Fundamentals

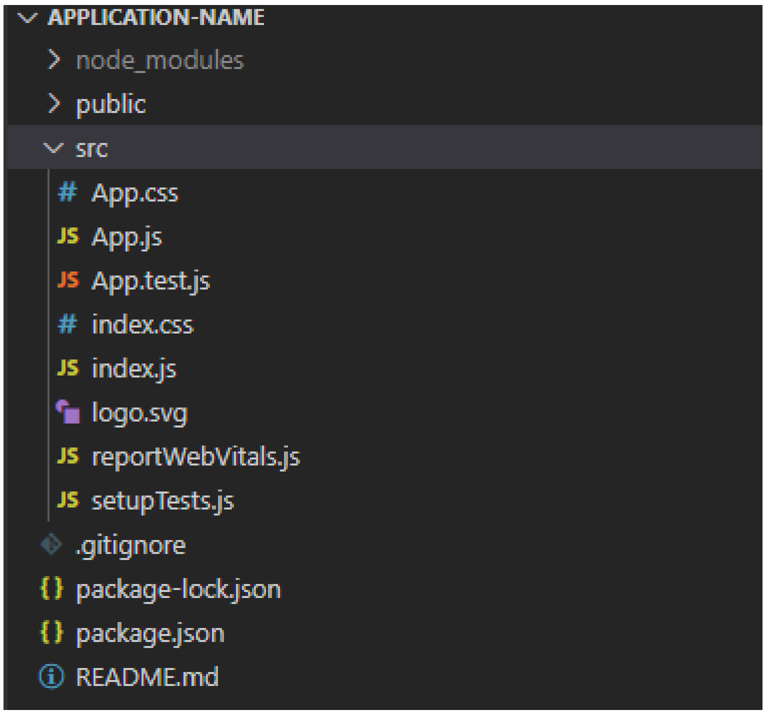
This chapter introduces the fundamental aspects of React. The following chapters provide examples of the fundamental ideas in code snippets. These fundamentals are later used as basic building blocks in the example application and other smaller examples. [1]

2.1 Setting up React Project

The most basic React app is created with the following command:

```
npx create-react-app application-name
```

Before using this command Node and npm will need to be installed on the system. The before-mentioned command will give a sort of React application template upon which one can start building the application further. The command will automatically create a project with a structure pictured in Figure 1. [1]



```
APPLICATION-NAME
  node_modules
  public
  src
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
    reportWebVitals.js
    setupTests.js
  .gitignore
  package-lock.json
  package.json
  README.md
```

Figure 1 React application created with create-react-app

This a complete application in that it is functional and can be started as is. So, all the configurations and setting up was done automatically by the create-react-app command.

Here is a breakdown of what the folders and files are: [2]

- README.md: The extension .md means that the file in question is a markdown file. Markdown is a markup language with plain text formatting syntax. Most if not all projects have this file, and it is used to give important or useful information about the project. The readme file is usually displayed in the git page of the project. In this case readme file was automatically generated to mimic the one in the official create-react-app repository
- node_modules/: node_modules folder contains all the node packages that has been installed via npm. With create-react-app some packages were installed automatically and are included in this folder. This folder does not need to be updated manually since most packages are installed or uninstalled with npm command and the content of node_modules would be updated automatically.
- package.json: package.json is a sort of manifest of the project that describes the metadata that is relevant to the project. This file is used to give information to npm so it is able to identify the dependencies that the project has. It may also have the metadata that describes the name or version of the project for example. There is no set of rules what must be in this file, but it must be in the json format. If json format is not applied the application will not be able to read it.
- package-lock.json: This file generated automatically for all operations where npm alters either the node_modules or package.json. Package-lock.json describes the tree exactly as it was generated. This allows subsequent install to generate identical trees.
- .gitignore: This file describes what files or folders that should not be included in git repository. These files are then not going to go to any remote repository and any changes made locally are for your use only.
- Public/: Public folder will have the development files. For example, index.html that is displayed on the browser when the application is started.

When creating an application with the create-react-app command the base project is created. With this also the required dependencies are installed. Upon installing dependencies package-lock.json is generated. This file is updated automatically and therefore is rarely manually altered.

2.2 Components

With React components can be defined as either classes or functions. Class and function components have slightly different ways of functioning, but most of the essential logic can be done in both. [2]

The first component here is the one created with the create-react-app command used in setting up the project. This component is shown in Figure 2.

```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

Figure 2 React component created with create-react-app

Figure 2 above describes a component and more specifically a functional component and we will delve more into component types in a later chapter. Default App component does not take in any parameters. In React these parameters are referred as props and those too will be covered later. The most important thing to notice is that this component returns something that looks like HTML. This is in fact called JSX, which is an acronym for JavaScript Syntax Extension.

2.2.1 Stateless Functional Components

Many applications have smart components that can hold state but render components that simply take in props and return HTML in the form of JSX. Stateless functional components are easier to reuse, and they usually have a positive impact to the performance of your application. [3]

Stateless components have two main characteristics:

1. They receive an object with the props passed to it
2. They return JSX that is to be rendered

Figure 3 displays the classic Hello World! application that is done using stateless function.

```
import React from "react";

const HelloWorld = () => {
    const sayHelloWorld = () => {
        alert('Hello World!');
    };

    return (
        <div>
            <a href = '#'
            onClick={sayHelloWorld}>Hello!</a>
        </div>
    );
};

export default HelloWorld;
```

Figure 3 Hello World as functional component

The HelloWorld in Figure 3 is a simple functional component that does not take in any parameters and its only task is to display a popup alert when the button is clicked.

2.2.2 Stateful Components

Unlike in the stateless component described in the previous chapter, stateful components have a state object that can be updated. The updating of these states can be updated with the setState method, and the state must be initialized before it can be set. Figure 4 describes a component that is stateful.

```
import React, { Component } from "react";

class StatefulComponent extends Component  {
    constructor(props) {
        super(props);

        this.state = {
            toggle: true
        };

        this.onClick = this.onClick.bind(this);
    }

    onClick() {
        this.setState((prevState) => ({
            toogle: !prevState.toggle
        }));
    }

    render() {
        return (
            <div onClick={this.onClick}>
                Toggle is: {this.state.toggle}
            </div>
        )
    }
}
```

Figure 4 Example of a stateful component

In Figure 4 the StatefulComponent is prefixed with the word ‘class’, making it a class component. The task that this component is given is to toggle a boolean between true and false upon button click.

3 JavaScript for React

JavaScript was first released in 1995 and has gone through multiple iterations after that. Initially JavaScript was used to add interactive elements to a web page. For example, clicking a button or validating a form and such elements were done with JavaScript. Nowadays Node.js, JavaScript has become a software language that is used to build full-stack applications. [3]

3.1 Variables

Variables in programming are where data can be saved to. JavaScript being weakly typed programming language the variables are not prefixed with the type of the variable, for example String if the variable is to store textual value. With JavaScript all the variables can be declared with same keywords. What keyword is used depends on the context where it needed.

3.1.1 Const

Const keyword are used when creating a constant. That means that it is a variable whose value cannot be changed after it is created. There are many variables in JavaScript that one usually does not want to overwrite so const is going to be used a lot. Constants were introduced into JavaScript with ES6 in 2015. Before the program could only declare variables that could be overwritten. Figure 5 demonstrates the usage of const keyword.

```
let exampleVar = '0';
exampleVar = '1';
console.log(exampleVar);

const exampleConst = true;
exampleConst = false;
```

Figure 5 Declaring variable and constant

In Figure 5 the first part is that a normal variable is created, its value changed and the logged in to the console. This is going to work without errors and ‘1’ will be logged correctly. The second part where the constant exampleConst is created and tried to be assinged it a new value it will give an error. The error will be logged into console even without the console.log type of line and the error will read “Uncaught TypeError: Assignment to constant variable.”. This will make sure that one will not even accidentally change the value of a variable that is meant to be unalterable.

3.1.2 Let and Var

In JavaScript there is the let keyword. Let is used when declaring a variable that is meant to be scoped inside a block. On the other hand, var is used for variables that are treated as normal variables. JavaScript knows three ways to declare a variable: var, let and const. Let keyword is used like any other variable keyword, as shown in Figure 6.

```
let exampleVar = '0';
exampleVar = '1';
console.log(exampleVar);
```

Figure 6 Variable with let

The distinction between the var and let keyword is that even though they are both used to declare variables the scope is different. Var is called function scoped variable and let is block scoped. That means that var is visible to all within the function and let is visible only in a block like for example in if/else statement block.

3.2 Functions

Repeatable tasks are usually best placed inside a function. From there it can be called as many times as needed and it is accessible from anywhere within the application. This chapter provides a look at the syntax used when creating a function. [3]

3.2.1 Declaring Function

In JavaScript a function is declared with function keyword. After the function keyword there is the name of the function and then the block where the functionality or tasks that the function should do is located. In Figure 7 there is an example of declaring a function.

```
function writeInLog() {
    console.log('This is written in the log');
}

writeInLog();
```

Figure 7 Declaring a function

In Figure 7 above there is a function called writeInLog. The only functionality this function has is that it writes ‘This is written in the log’ into the browser console. Below the function there is this line that reads writeInLog();. That is a function call. Every time a function is called in that fashion another line gets written into the console.

There is alternative way of creating or declaring functions. That is by using something that is called function expressions. Basically, this means that the function is technically created as a variable as shown in Figure 8.

```
const writeInLog = () => function() {
    console.log('This is written in the log');
}
```

Figure 8 Function expression

In figure 8 the same function as before has been created but this time as a function expression. The result of the function is the same in both functions and the message will be logged in the console in both cases.

The only difference that these functions have is that function declarations can be called from anywhere in the code, even before it is declared whereas function expression can only be called afterwards. Figure 9 describes a situation that would end in error.

```
writeInLog();

const writeInLog = function() {
    console.log('This is written in the log')
}

function writeInLog() {
    console.log('This is written in the log');
}
```

Figure 9 Trying to call function expression too early

The result of the code presented in Figure 9 would be an error. Function that is not declared with function keyword can only be called after its declaration.

3.2.2 Parameters

The functions displayed in the previous chapter currently do not take in any parameters. In some cases, this is not wanted, and instead one would want to pass arguments to the function and dynamically alter what the function does.

In JavaScript, like many other programming languages, the parameters that a function takes in is described in parentheses after the function name. If for example, we wanted to alter the function described in the previous chapter to take in the first name of a person and print a greeting one could do what is described in Figure 10.

```
function writeInLog(firstName) {
    console.log('Hello, ' + firstName)
}

writeInLog('Sami');
```

Figure 10 Passing first name to the function

The function writeInLog in Figure 10 would then write the message “Hello, Sami” into the console. Functions can also have default values set for the parameters it takes. For example, one could change the function to what is displayed in Figure 11.

```
function writeInLog(firstName = 'Sami') {
    console.log('Hello, ' + firstName);
}
```

Figure 11 Function with default value for firstName

In this scenario, the function can be called without giving any parameters and it would still write the same message as previously into the console. If called with a parameter, it would replace the default value with that value and print that instead.

This can be taken a bit further as well. If for example the function is meant to take in something else than a string for the first name but actually a variable that has both first and last name like in Figure 12.

```
const examplePerson = {
    name: {
        firstName: 'Sami',
        lastName: 'Sikkilä'
    }
};

function writeInLog(person = examplePerson) {
    console.log('Hello, ' + person.name.firstName +
                    ' ' + person.name.lastName);
}
```

Figure 12 Function taking in a person with default value

That function would work the same as the previous one in that by default it is going to print the message with the values from the default person. If no parameters are given with the call the message would then be ‘Hello, Sami Sikkilä’.

3.2.3 Arrow Functions

With ES6 JavaScript got a new feature called arrow functions. Arrow functions allow to create the functions without needing to use the function keyword and quite often one can also leave out return keyword. For example, the function shown in Figure 13 could be turned into an arrow function as described in the figure

```
const writeInLog = firstName => `Hello, ${firstName}`;
```

Figure 13 writeInLog with as arrow function

This feature makes the syntax much simpler, and the function can be fitted into a single line. Also, as might be noticed the other way of using the parameters. To do this one would need to use `-character and not the single quote.

The reason this form of doing a function does not require a return statement is that the arrow is pointing to what the function should be returning. As can be seen there is no parentheses and that is because there is only one parameter. If there would be more than one, the parentheses would again be needed.

The arrow function in the previous example is concise and fits in one line. This, however, does not need to be the case. Arrow function can be multiple lines as well as illustrated in Figure 14.

```
const writeInLog = (firstName, lastName) => {
    if (!firstName || !lastName) {
        throw new Error('You should give the full name!');
    }

    return `Hello, ${firstName} ${lastName}!`;
};
```

Figure 14 writeInLog with checking names are given

In the example shown in Figure 14 the arrow function is multiple lines, but it works just as well. There is now an if block that checks that both names were given to the function. In case that they were not both present an error message will be shown and if they both are found the message will be logged into console normally.

3.3 Classes

When JavaScript was first introduced, there were no classes. In 2015 classes were introduced into JavaScript and the syntax is much like that in for example Java. At first classes were used a lot to build user interfaces, but slowly programmers and libraries have begun to use them less. Nowadays functions are used more in creating the components, but classes are still very much in use at least in the older React code base and therefor might be useful to recognise and understand. Following the example set by previous examples, an example of a JavaScript class is shown in Figure 15. [4]

```
class Person {
    constructor(firstName, lastName) {
        this.firstName = firstName
        this.lastName = lastName
    }

    print() {
        console.log(`Hello, ${this.firstName} ${this.lastName}`);
    }
}

const person = new Person('Sami', 'Sikkilä');

person.print();
```

Figure 15 Example of a JavaScript class

In the example above (Figure 15) there is a class called Person. This class has two variables: firstName and lastName. Very much like in Java for example the ‘new’ keyword is used to create a new person. Application can create as many of these persons as is wanted and they can then be passed to other components like any other variable in JavaScript.

4 ReactJS Properties

This chapter introduces some of the most fundamental properties in ReactJS, such as JSX and lists. [4]

4.1 ReactJS

ReactJS is a JavaScript library created by Facebook. It was created in 2013 and has since become one of the most used frameworks for front-end development. React makes creating user interfaces easier by splitting each page into smaller pieces. These pieces in React are called components.

React-component is a piece of code that depicts a certain part of the website. Each of the components are JavaScript functions that returns code that is later used to render something into the user interface.

React uses JSX syntax but does not require it. One can utilize React with JavaScript but most commonly JSX is used as it conceived to be useful.

4.2 JSX

JSX is a syntax extension to JavaScript. React embraces the fact that rendering logic is inherently coupled with other UI logic on how to handle events, changes to the state over time and how the data is prepared for displaying.

JSX combines JavaScript with the X from XML. JSX is an extension to JavaScript, and it allows defining React elements with tag-based syntax directly from our JavaScript code. JSX is sometimes confused for HTML because they look alike. JSX is alternative way to create React elements aiming to make it simpler and less prone to errors in syntax such as missing commas. [2]

4.2.1 React Elements as JSX

JSX was released with React with the intention of offering an easy and readable syntax for creating complicated DOM trees with attributes. With JSX the types of element are specified with tags. Attributes of the tag describe properties and the children of the element can be added between the opening and closing tags.

In an unordered list, children may be added to it with JSX tags. This structure looks very much like HTML, as can be seen in Figure 16.

```
<ul>
    <li>Chassis</li>
    <li>Engine</li>
    <li>Transmission</li>
    <li>Steering wheel</li>
</ul>
```

Figure 16 HTML list

JSX can be used with components by defining the component using the name of the class. Figure 17 shows an example on how to create component with JSX.

```
React.createElement(partsList, {list:['Chassis, Engine']});

<PartsList list={['Chassis', 'Engine']} />
```

Figure 17 Creating PartsList with JSX

JSX element is a way to call React.createElement(). With JSX that method should typically not be called directly. The two lines displayed in Figure 17 above are essentially the same or at least the outcome is.

Passing the array of parts or whatever the list is to contain, it must surround it with curly braces. By doing this, a JavaScript expression is created, and such syntax must be used when passing values as properties to components. Properties of components can either strings or JavaScript expressions. JavaScript expressions may be arrays, objects or even functions.

5 React States

React components need data to truly function. User interface created with React needs some data to be passed into the components for them to do what you want. That data does not have to be in any specific format and not all the data have to be present all the time. React components can detect what parameters are present when and act accordingly when executed.

States and properties are in a relationship with each other. With React applications the components are tied together based on that relationship. When changing the state of one component in the tree the properties change too, and the newly changed data will flow to other parts of the tree triggering components and properties down the line to render new content to the user interface.

With states application can be rendered any time something some relevant information or data within the application changes. This is one of the key features of React. [2]

5.1 State in Functional Component

As default, functional components are stateless so the way to make a functional component stateful would be by using a hook that React has for it. In addition, functional components need other helpers or hooks to use states. These hooks are covered in a later chapter.

The build-in way to control state in functional component is to use the useState hook. useState hook is already in included and available in the react package and can be imported with the import shown in Figure 18. States in function components are used in a React feature called a Hook. These hooks are code that can be reused and are separate from the component tree of the application.

```
import React, { useState } from 'react';
```

Figure 18 Importing useState hook

UseState hooks are initialized as shown in Figure 19. In this example a state that is boolean value, that is it can have two values. It is either true or false.

```
const [hidden, setHidden] = useState(true);
```

Figure 19 Initializing useState

useState hook takes two parameters. The first one is the variable that is used to access the current value that the state has. The second one is the setter method that is used to change the state. The useState does not have to be boolean, it can be anything you might want to track.

5.2 State in Class Component

To create a class component, a class file must be created and then have it extend React’s Component class. This can be done in a couple of ways, and one of them can be seen in Figure 20.

```
import React, { Component } from 'react';

export default class Product extends Component {

}
```

Figure 20 Class that extends React.Component

By extending Component the class gains all the functionality that a React class component would have. One important part of these functionalities is the states. Managing states in class components are a little bit easier when compared to functional components. This is because class components are stateful by default which means that one does not have to use separate hooks to manage them.

5.3 Ref

Typically, in the dataflow of React props are the only way for components to interact with its children components. In order to alter the child, it would need to be rendered again with the new props. There might be cases where child component needed to be altered outside of the typical dataflow. The child component that needs to be modified could be an instance of a React component or it could be a DOM element. For these few cases, React provides an easier and better way. [4]

Refs are created with React.createRef() and and attached to React elements via ref attribute. Most commonly refs are assigned to a variable on creation of the component. This way the ref can be referenced from anywhere in the component. An example of class that has a ref is given in Figure 21.

```
class MyComponent extends React.Component {
    constructor(props) {
        super(props);
        this.myRef = React.createRef();
    }

    render() {
        return <div ref={this.myRef} />
    }
}
```

Figure 21 Class that has a ref [4]

Refs are often used to store data inside the component that needs to survive refreshing the page but does not need to be accessed in some other component. If a data needs access from more locations a state would provide better solution.

6 Props in React

Props can be thought of to be the parameters in programming. It is a way to pass information or data from one component to another where it can be used to create the desired elements. Props in React are immutable and allows one to create reusable components where props do not have to be declared separately. A basic example of the usage of props can be seen in Figure 22.

```
import React, { Component } from 'react';

class Parent extends Component {

    render() {
        return (
            <div>
                <Child
                    title='TitleExample'
                    text='Lorem Ipsum'
                />
            </div>
        )
    }
}

class Child extends Component {
    render() {
        return (
            <div>
                <h1>{this.props.title}</h1>
                <h2>{this.props.text}</h2>
            </div>
        )
    }
}
```

Figure 22 Example giving props from component to another

In Figure 22 there are to class components. The parent component will render another component, in the example named to be Child. As can be seen, the Child components in the return section of the Parent component have two props: title and text. These props will be passed to Child component where they can be accessed with this.props.title and this.props.text. This way the Child component can be called with whatever props given in these two fields, making it reusable. It is not needed to create a completely new component when wanting to change these values.

7 Redux

As the development of the application goes forward and becomes more complex and the number of components become larger, managing the data in the application is going to become a problem that will have to be solved somehow. [5]

Redux was created as a solution for this problem. It is currently one of the most used libraries in front-end development. Redux is an open-source and it was created by Dan Abramov and Andrew Clark in 2015. According to them, Redux will help with data-flow management of the applications and make it more efficient. In Figure 23 one can see the data flow in React/Redux application.

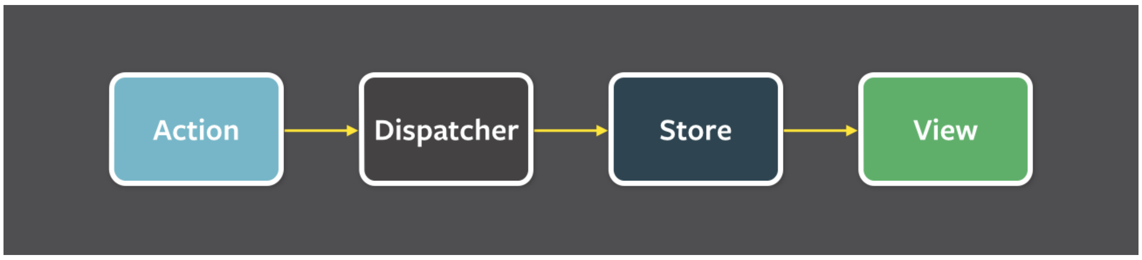


Figure 23 Data flow in React/Redux application [5]

Figure 23 describes the basic flow with Redux. The three main factors of Redux include: Action, reducer and store. Redux follows unidirectional data flow which means that the data of the application will follow in one-way binding data flow. When the application becomes too big and complex it will no longer be easy to reproduce issues or add new features if there is no control over the state of your application. Redux data flow would go in the following steps:

1. An action is dispatched by the user interacting with the application
2. The root reducer function is called with the current state and the dispatched action. The root reducer may divide the task among smaller reducer functions, which will return a new state.
3. Store notifies the view by executing their callback functions.
4. The view can retrieve the updated state and render the user interface again

These four principals are the essence of Redux and are very important to understand when working with Redux. The only way to update data in the Redux store is by using reducers which requires an action.

7.1 Actions

Redux is used to reduce the complexity of the code. This is done by enforcing a restriction on how and when the state updates are allowed to happen. This makes managing and updating states easier.

An action is a JavaScript object, and it has a field called type. Action can and should be thought of event that describes something happening in the application. Type field is usually a string that gives the action a name that is descriptive enough to tell what the action does. There is also another field that provides more information about the action that happens. Usually, this field is named as payload. Below in Figure 24 there is an example of action.

```
const addToDoAction = {
    type: 'todos/todoAdded',
    payload: 'Buy milk'
}
```

Figure 24 Example of an action

Action addToDoAction in Figure 24 has the type todos/todoAdded and payload Buy milk. When action is invoked, these parameters will be passed to reducers so it knows how the state of the application should be updated.

7.2 Reducers

A reducer is function that receives current state and an action and then decides if the state should be updated. If the state is to be updated the reducer will return the new state. Reducers are sort of event listener that handle events based according to the action object that was passed to it.

There are some rules that the reducer must follow every time: [6]

- The new value of the state should only be calculated based on the state and action it was passed

- Reducers cannot modify the existing state, so they have to copy the value of the state and make changes to the copied value
- Reducers cannot use asynchronous logic, calculate random values or cause any other unintended side effects

Reducers can have any logic inside of them and below in Figure 25 there is an example of what reducer might look like

```
function counterReducer(state = initialState, action) {
    if (action.type === 'counter/incremented') {
        return {
            ...state,
            value: state.value + 1
        }
    }
    return state
}
```

Figure 25 Example of a reducer

In the example above (Figure 25) if the type of the action that is passed to the reducer is ‘counter/incremented’ the state is copied and then the value of the copied state gets incremented by one.

7.3 Store

The current state in Redux application is stored in an object called store. The store takes in a reducer as parameter when created. Store has a method getState that will return the current value of the state.

Stores also have a method called dispatch. The state can only be updated by calling this dispatch method and give an action as an argument. The store will then run the reducer method and save the new value of the state where it can be then retrieved by calling getState.

8 Data

With React applications data is absolutely what makes it work. Without data React components would not display anything. Simply put, the components discussed earlier are only vessels or containers for this data. [4]

8.1 Requesting Data

Hyper Text Transfer Protocol, HTTP for short, provides the way to communicate with internet within React. For example, when opening Google search engine, the browser is sending a request to Google servers asking for the search page. Upon receiving the request, the server will handle it internally and find the files that were requested and then send them back to the browser again via HTTP.

In JavaScript the most used way of making HTTP requests is using fetch. If it is wanted to get information from GitHub about the user Moon Highway, application would try to send the requested shown in Figure 26.

```
fetch(`https://api.github.com/users/moonhighway`)
    .then(response => response.json())
    .then(console.log)
    .then(console.error);
```

Figure 26 Request to GitHub to get information. [2]

The fetch function always returns a promise. In the example described in Figure 12 there is an asynchronous request made to the URL: https://api.github.com/users/moonhighway. Making requests will always take some time move through the internet and respond with the requested information. Therefore, the callback function is used then. This way one waits for the request to process and the response to arrive back. Then the information received in the response is given to call back function where one can start processing it. In this case, the response is JSON but inside HTTP response body so one needs to call response.json() to extract the data and parse it as JSON.

There is an alternate way to work with promises. That is by using async and await. As discussed, fetch returns a promise, so await can be used inside an async function as seen in Figure 27.

```
async function requestGithubUser(githubLogin) {
    try {
        const response = await fetch(
            `https://api.github.com/users/${githubLogin}`
        );
        const userData = await response.json();
        console.log(userData);
    } catch (error) {
        console.log(error);
    }
}
```

Figure 27 async/await function [2]

The examples given in Figures 26 and 27 achieve the same results. If application uses await it will wait for the promise to be resolved before going forward.

8.2 Sending Data

Requests often require data to be uploaded within the request. For example, if a new user is created for the application and the request will be sent to server, one would need to send the username and password with the request.

Usually when creating new data, the application would use POST request and when modifying data that already exists would use PUT. Another information given with fetch is the argument that allows to pass an object of options that are then used by fetch when creating the HTTP request. An example of POST can be seen in Figure 28.

```
fetch("create/user", {
    method: "POST",
    body: JSON.stringify({username, password})
});
```

Figure 28 POST request when creating new user

In the example above (Figure 28) the method that is used is POST to create a new user. With the request in the body, it is passing the new login information of the new user. This information will need to be extracted from the user interface.

8.3 Uploading Files

In order to upload a file, one needs a new kind of HTTP request. This request would a multipart-formdata. Multipart-formdata request will tell the server that one or more files is included in the request body. In order to make the request in JavaScipt one needs to pass a formData object with the request as shown in Figure 29.

```
const formData = new FormData();
formData.append('firstName', 'Sami');
formData.append('lastName', 'Sikkilä');
formData.append('profilePicture', profilePic);

fetch('/register/user', {
    method: "POST",
    body: formData
});
```

Figure 29 formData being passed with fetch

In this example the form passes the first and last name as well as the image the user wants to use as the profile picture. Requests create a new FormData object to which the necessary values are added with key and value pairs. These will then be sent with the request where they can be extracted from wherever the logic is located and processed. This example is a bit flawed in the way that the values are hardcoded, and this is most likely never the case. Usually, they are going to be read from some sort of form and then passed on.

8.4 Saving Data

If the application wants to save some data locally, it can use Web Storage API. With this API application can save data locally into the browser of the user. There are two options for this: localStorage and sessionStorage. There are some differences between the two, but the main idea is that the data will be saved into the browser where it is kept for later access. The data saved in local storage will persist until it is removed and data in session storage will only persist until the tab, or the browser is either closed or restarted. In Figure 30 there is an example of first saving the first name of a person into the session storage and then retrieved right after. An example of how to save and fetch items from storage can be seen in Figure 30.

```
window.sessionStorage.setItem('firstName', 'Sami');
window.sessionStorage.getItem('firstName');
```

Figure 30 Example of using session storage

Saving data into the browser storage is vulnerable to several sorts of attacks so it is not recommended to use these when storing any type of sensitive data.

9 Making Example Application

In this chapter the creation of a simple example application with React and using Redux to save the applications state is introduced. This component was done on Windows computer, but it should work the same on any other operating systems as well. This application requires React and react-redux packages to be installed via npm. [1]

9.1 Setting up

As described in Chapter 2.1 a new project can be set up with the command npx create-react-app. For example, a new React app called is created as shown in Figure 31.

```
npx create-react-app
```

Figure 31 Example application [7]

This produces the template upon which the application can be built. The application is runnable already and can be started with the command npm start. After entering the command and hitting enter the application can be accessed in localhost:3000. The webpage is shown in Figure 32.

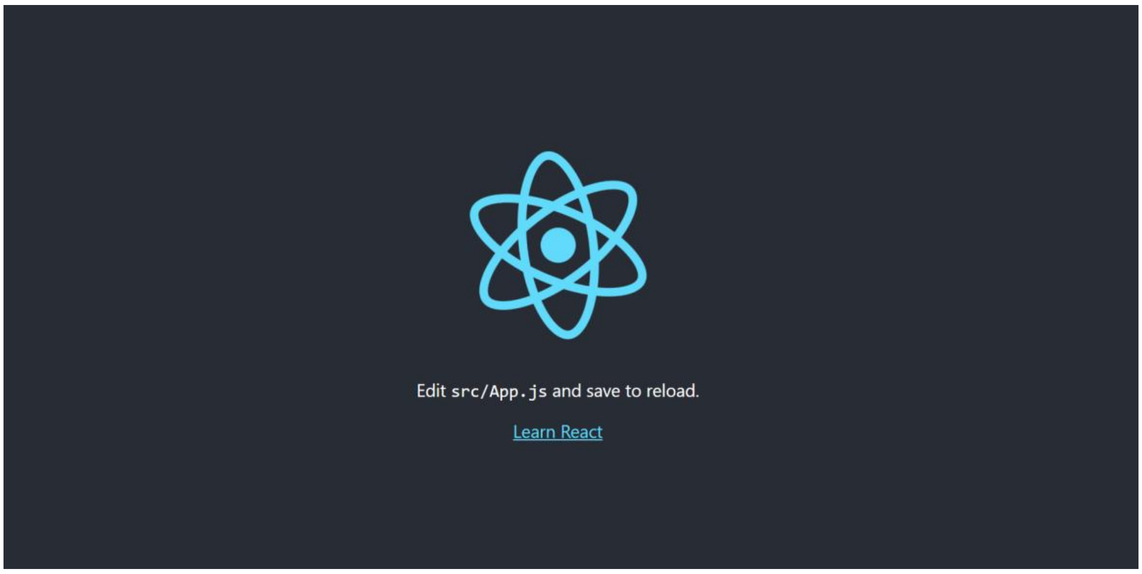


Figure 32 The default page after creating the app

The way React works is that it will check the file called package.json for the endpoint which to render on the page. In this case package.json does not specify a file to use for this so the app will automatically go forward and look for index.js and render that instead. With the default React app, the following index.js file shown in Figure 33 is created:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Figure 33 index.js created with create-react-app

As can be seen from the example above (Figure 33) the application that is to be rendered on the screen is nested inside ReactDOM.render call. StrictMode in React is a tool that is used for highlighting any problems that exists in the application. Strict mode itself does not render anything on the screen so it is not visible to the user. It is still a very helpful step to add especially when creating user interfaces for the first time to give insight to what might go wrong.

Inside there is one line that reads <App />. That is a call to a component named App which is imported from file called App.js according to the import statement. The usual convention with components is such that a component and all the parts that are not reusable somewhere else should be place in the same file. Parts that are reusable with some other component should then be placed elsewhere where it can be accessed by multiple files. App.js looks by default looks like the following (see Figure 34).

```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

Figure 34 App.js created by default

The elements inside App function are what renders to the browser. Everything that gets returned by the App function or any other function must be encapsulated within one container. In this case, the container used is a div but could also be any other container present. Having multiple parallel containers would result in an error.

9.2 Creating Component

React is most useful and understandable when it is separated to clear parts, therefore a folder that houses the components should be created under src folder and can be named components. This is a useful practice because then the often reusable and similar elements of the application are in the same location.

This example application is going to include a shopping list, so a new file called ShoppingList.js is created as illustrated in Figure 35.

```
import React from 'react';

function ShoppingList() {
    return (
        <li className="shoppingList">
            <div>
                <label>
                    Milk
                </label>
            </div>
            <div>
                <button type="button">
                    Buy
                </button>
                <button type="button">
                    Remove
                </button>
            </div>
        </li>
    );
}

export default ShoppingList;
```

Figure 35 ShoppingList component

In the newly created ShoppingList component a div a list is returned. Currently in the list there is a div where it reads the item that is placed in the shopping list. Below that there are two buttons, one to set the item as bought. This is going to be set to remove from the list after the Buy button is clicked. The second button is to remove in case the user decides not to buy the item after all and just wants to remove it from the list.

Now with the new custom component the contents of the App file and App function can be changed accordingly, as illustrated in Figure 36.

```
import './App.css';
import ShoppingList from '../src/components/ShoppingList';

function App() {
  return (
    <div>
      <ShoppingList />
    </div>
  );
}

export default App;
```

Figure 36 App that uses new component

The ShoppingList component that was created can now be imported into this file and used in rendering the application. The code that formerly resided here can now be simply replaced with the call to the component. With the styling that is described in Chapter 9.3 the application looks like this (Figure 37)

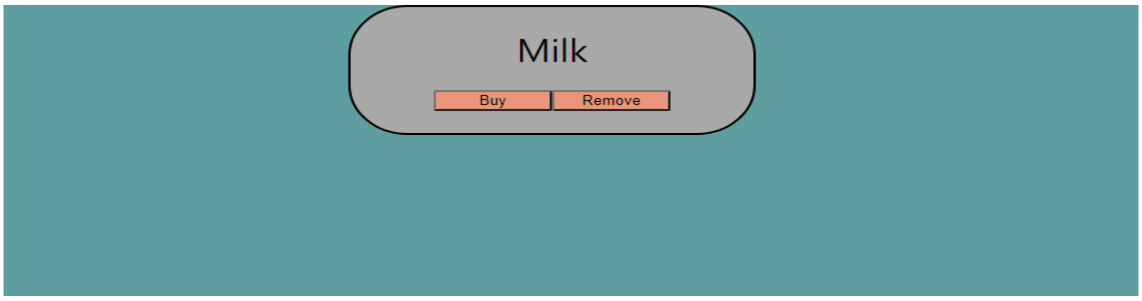


Figure 37 ShoppingList component with styling

More complex styling is usually done with bootstraps, but they are not included in the scope of the present study, so the styling was done with plain CSS.

9.3 Styling

Like with any front-end programming styling is important. Without styling the webpage would not look good. There are two several different methods of styling components, more popular and reusable way is to create a CSS file to house the styling to the elements as described in Figure 38. The CSS file that was generated automatically can be found in src/index.css. [8]

```
body {
  margin: 0;
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', 'Roboto',
'Oxygen',
    'Ubuntu', 'Cantarell', 'Fira Sans', 'Droid Sans', 'Helvetica Neue',
    sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  background-color: cadetblue;
  padding-left: 40%;
}

code {
  font-family: source-code-pro, Menlo, Monaco, Consolas, 'Courier New',
    monospace;
}

ul {
  list-style: none;
}

.shoppingList {
  font-size:xx-large;
  width: 300px;
  padding: 20px;
  border-width: 2px;
  list-style-type: none;
  border-style: solid;
  text-align: center;
  border-radius: 50px;
  background-color:darkgray
}

button {
  background-color:darksalmon;
  width: 100px;
}

.visually-hidden {
  display: none;
}
```

Figure 38 Styling

Without styling the page will be displayed without any order. With CSS the application can be given a visually pleasing structure.

9.4 Multiple Items

More ShoppingList components can be added to be returned with App and they can be as many as wanted. If for in this case two are added the following window will be rendered, cf. Figure 39:

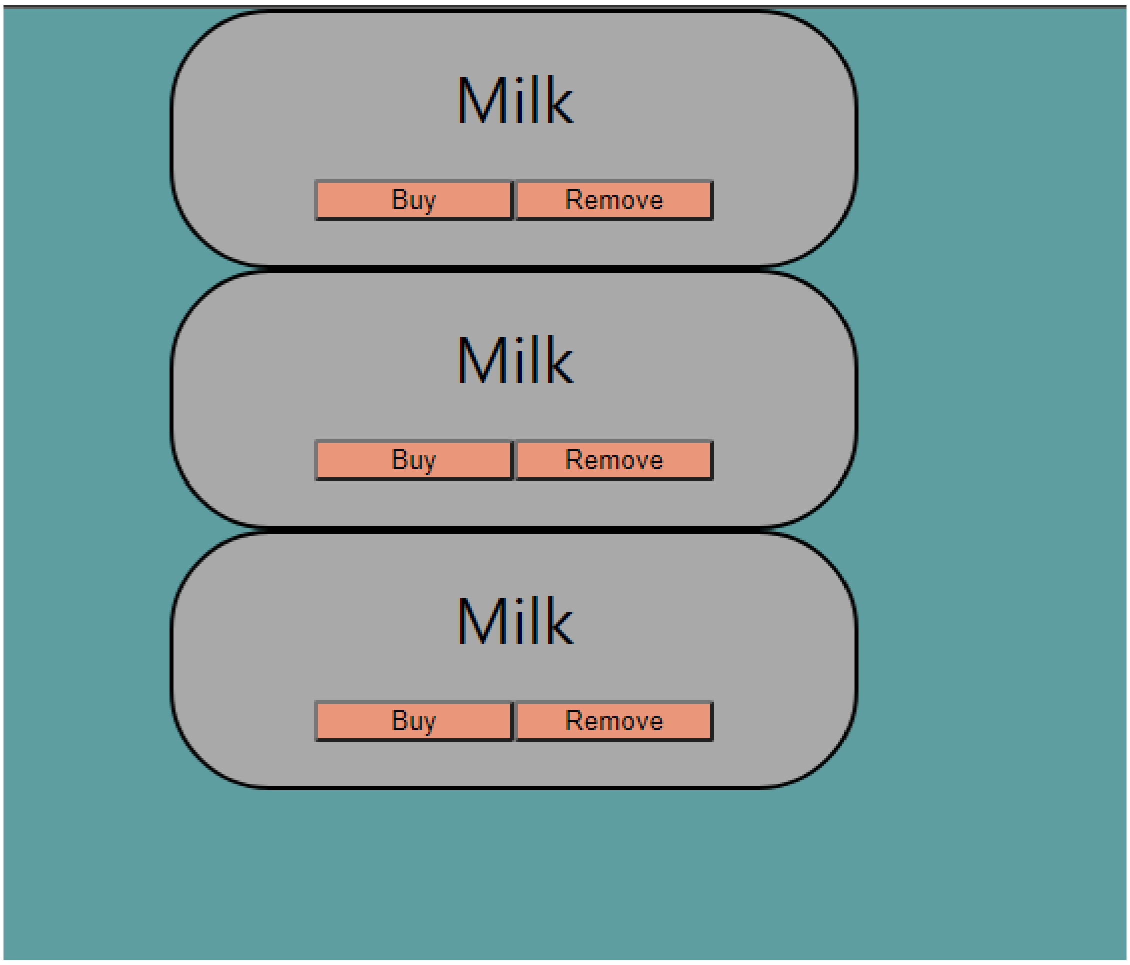


Figure 39 Page with multiple components

As can be seen, the if the components are simply added to the app, they will all render the same with the default data that is set in the component itself. That is for obvious reasons not very practical and a way to pass data to each instance of the component is needed. One way that can be achieved is specifying a parameter with the creation the way illustrated in Figure 40.

```
import '/App.css';
import ShoppingList from '../src/components/ShoppingList';

function App() {
  return (
    <div>
      <ShoppingList productName = 'Milk' />
      <ShoppingList productName = 'Butter' />
      <ShoppingList productName = 'Coffee' />
    </div>
  );
}
```

Figure 40 Creating components with parameters

In the way illustrated above (Figure 40) every component will render different if take that value that is passed in consideration in the component as follows (Figure 41):

```
import React from 'react';

function ShoppingList(props) {
    return (
        <li className="shoppingList">
            <div>
                <label>
                    {props.productName}
                </label>
            </div>
            <div>
                <button type="button">
                    Buy
                </button>
                <button type="button">
                    Remove
                </button>
            </div>
        </li>
    );
}

export default ShoppingList;
```

Figure 41 Shopping List that takes in the product name

In Figure 41 above there is now “props” included in the function declaration. With this the properties passed by the call can be accessed. Every parameter is included in props can be accessed like with dot operator like in the previous image. This will then render the following page (Figure 42):

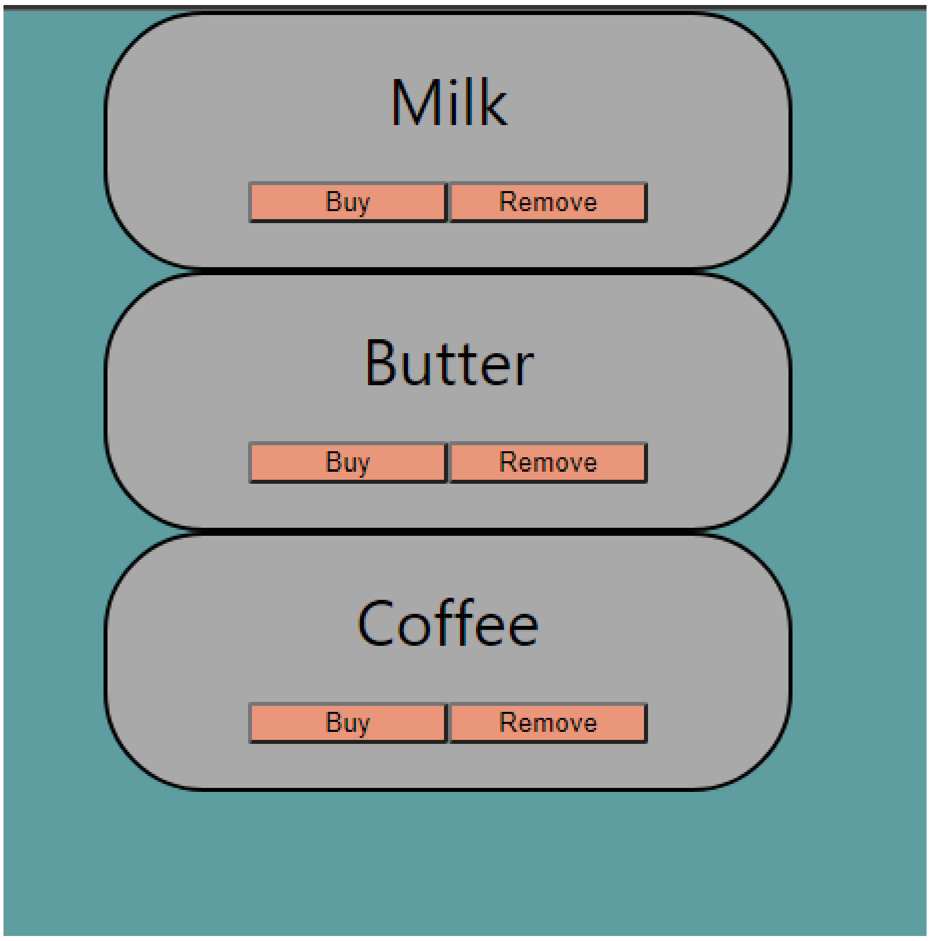


Figure 42 Components each with different product

This view is now better but still not optimal. There should be a way for the user of the application to add the components and have them rendered dynamically.

9.5 Redux and User Input

Redux is often used alongside with React to store the data for later use and make the data accessible and updateable from various locations inside the application. Redux might feel confusing at first but it is quite a nice way to store the data. [5]

9.5.1 Creating Store

First, Redux will need a store. In Redux the store is where all the states of the application are stored so it is a very essential part. Creating store is simple as can be seen in Figure 43.

```
import { createStore } from "redux";
import rootReducer from "../reducers/index";

const store = createStore(rootReducer);

export default store;
```

Figure 43 Creating a store

Creating a Redux store only takes one parameter: Reducer. Creating a reducer will be covered later. Redux comes with a function called createStore which can be used. This method takes in a reducer as a parameter and its creation is explained in the next chapter.

9.5.2 Creating Reducer

Reducer is a function that takes two parameters: action and the current state. Action will be created in the manner described in the next chapters. A simple reducer is to be created that allows the user to add items to the state that holds the items that have been added to the shopping list. A simple way to do this is described in Figure 44.

```
import { ADD_ITEM } from "../constants/actionTypes";

const initialState = {
  items: []
};

function rootReducer(state = initialState, action) {
  if (action.type === ADD_ITEM) {
    return Object.assign({}, state, {
      items: state.items.concat(action.payload)
    });
  }
  return state;
}

export default rootReducer;
```

Figure 44 Creating a reducer

Reducer defines how the state of the application should be changed. Usually there is if blocks that are executed based on the type parameter that is passed from with the action. Reducers also define the initialState that is used that the updates are applied to.

9.5.3 Creating Action

In Redux, the only way to change a state is by passing it to the store. In order to do that an action object is needed. Actions will normally have two properties: type and payload. Type dictates how the state change should be handled and payload describes what is to change. Type is always required but payload is not if there is nothing new to save to the store. Figure 45 illustrates creating an action that is going to be used for adding items to a list.

```
import { ADD_ITEM } from "../constants/actionTypes";

export function addItem(payload) {
    return { type: ADD_ITEM, payload }
  };
```

Figure 45 Creating an action

In the case of this example application the type parameter of the first action gets the value ADD_ITEM and then there is the payload. This function will then be used when the state must be changed. Payload will be the input that is received from the user via the user interface. Both the type and input will be passed to the reducer.

9.5.4 Getting User Input

At this point everything should be set and waiting for the user to start giving names for the products that need to be added. In Order to do this a form will be added to the application. The code for the form can be seen in Figure 49.

The form might be the most complex looking component in the application, and it just might be too. Form is essentially where it all is combined. The first function there is called mapDispatchToProps. This function connects the Redux actions to the React props.

The form is set up to have a text input field where the user can enter the name of the product that should be added to shopping list and upon submitting the form the dispatch in mapDispatchToProps will be triggered. This will then perform the action which will then in turn go to reducer to check what the type is and then update the state that the action type in question should control.

9.5.5 Rendering Items in List

Now that the items should be set in the state, the only thing left is to render them to the web page. App.js shown in Figure 46 is now in the form shown below.

```
const App = () => (
  <>
  <div>
      <h2>Add new item to shopping list</h2>
      <Form />
      <ShoppingList />
  </div>
  </>
);

export default App;
```

Figure 46 App.js

The component in Figure 46 describes that what will be rendered on the page is firstly a header to tell the user what this is, then the form where the user can enter the name of the product and lastly ShoppingList component. This component will render the list and it looks as follows (Figure 47):

```
import React from "react";
import { connect } from "react-redux";

const mapStateToProps = state => {
  return { items: state.items };
};

const ShoppingList = ({ items }) => (
  <ul>
    {items.map(el => (
      <li key={el.id}><ShoppingListItems productName={el.item} /></li>
    ))}
  </ul>
);

export default connect(mapStateToProps)(ShoppingList);


function ShoppingListItems(props) {
    return (
        <li className="shoppingList">
            <div>
                <label>
                    {props.productName}
                </label>
            </div>
        </li>
    );
}

export { ShoppingListItems };
```

Figure 47 ShoppingList.js

Here again mapStateToProps is used to get the items list that is stored in Redux. ShoppingList function will iterate through the list and create a ShoppingListItems component for each of the entries. The parameter el.item holds the name of the product and is passed as argument in the same way as before.

The finished application looks as illustrated in Figure 48.

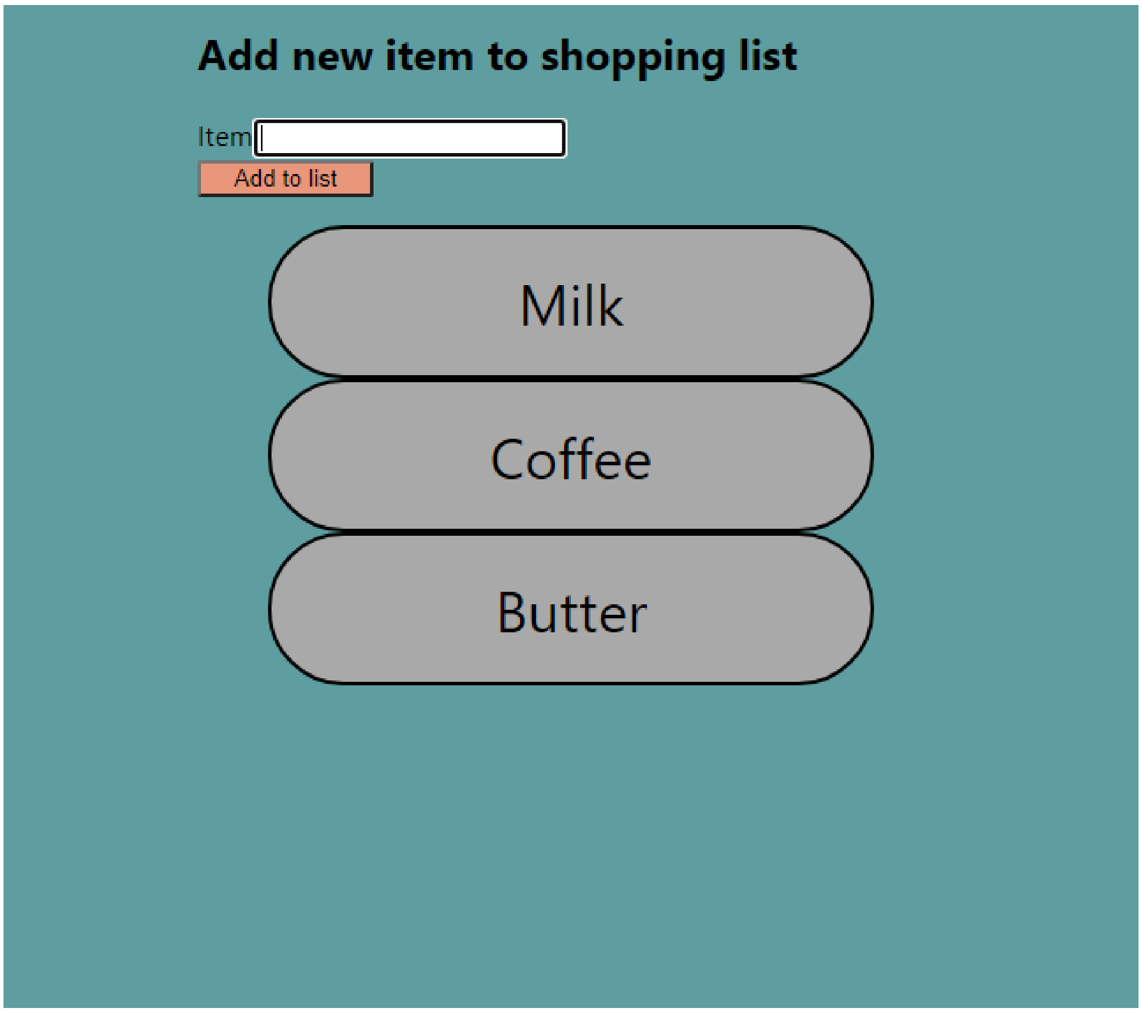


Figure 48 Shopping list application

The user interface in Figure 48 allows user to write the name of the item into the text field and add it to the list by clicking the button. Upon clicking the button, the item will be added to the shopping list and displayed at the end of the list. The Figure 49 in the next page describes what the ShoppingList component looks like.

```
import React, { Component } from "react";
import { connect } from "react-redux";
import { addItem } from "../actions/index";

function mapDispatchToProps(dispatch) {
  return {
    addItem: item => dispatch(addItem(item))
  };
}

class ShoppingList extends Component {
  constructor(props) {
    super(props);
    this.state = {
        item: ""
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({ [event.target.id]: event.target.value });
  }

  handleSubmit(event) {
    event.preventDefault();
    const { item } = this.state;
    this.props.addItem({ item });
    this.setState({ item: "" });
  }
  render() {
    const { item } = this.state;
    return (
      <form onSubmit={this.handleSubmit}>
        <div>
          <label htmlFor="item">Item</label>
          <input
            type="text"
            id="item"
            value={item}
            onChange={this.handleChange}
          />
        </div>
        <button type="submit">Add to list</button>
      </form>
    );
  }
}

const Form = connect(
  null,
  mapDispatchToProps
)(ShoppingList);

export default Form;
```

Figure 49 Form to get user inputs

Figure 49 is the form that the user can add items to the list with. The form will invoke handleSubmit method when the button is clicked. From there the addItem action will be called and passed to parameter where it is saved into the Redux store for later access.

10 Conclusion

The goal of the study was to learn about React and create a guideline based on which more detailed guides could be created. The author did not know much about React before starting the study and a lot of new things were learned on the way. The idea initially came from the commissioner where there is a need for a guide on how to start working with React and create user interfaces that deal with a lot of data.

React and Redux both have some complex terminology and subjects that take a while to get used to. When studying these concepts further, they turned out to be powerful tools that give developers excellent features to deploy. The advantage of React is that developers can program reusable components that can be customized to function in any way needed.

Redux has become one of the most used libraries with React because it allows the application to share its data between multiple components. With Redux storing and accessing data becomes logical and it works in the same way all the time because everything is done by using the same actions and reducers.

Due to the time being limited, the example application is not very complex, and will likely be either advanced further or a completely new application will be created. The application does not for example get to use fetches to get data from an API. React was found to be a great choice for creating user interfaces and very enjoyable to explore.

The outcomes of the present study will prove useful when creating a more specified guide that caters more to the needs of the commissioner.

References

1 About npm. Npm documentation. Npmjs. <https://docs.npmjs.com/about-npm>. Read 25.10.2021.

2 Banks, Alex & Porcella Eve. 2020. Learning React, 2nd Edition. O'Reilly.

3 Wieruch, Robin. 2018. The Road to React.

4 React Documentation. Online document. ReactJS. <https://react.js.org>. Read 1.10.2021.

5 Redux Documentation. Online document. Redux. <https://redux.js.org/usage/configuring-your-store>. Read 1.10.2021.

6 Fullstack Open. Online Course. Helsinki University. <https://fullstackopen.com/en/part6/flux_architecture_and_redux> Read 15.10.2021.

7 A JavaScript Library for Building User Interfaces. Online document.

<https://opensource.fb.com/>. Read 29.9.2021.

8 CSS tutorial. Online documentation. W3schools. <https://www.w3schools.com/css/>. Read 25.11.2021.