VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Kim Tran

STATE MANAGEMENT IN REACT

School of Technology

2022

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Kim Tran
Title	State Management in React
Year	2023
Language	English
Pages	92 pages
Name of Supervisor	Harri Lehtinen

The primary purpose of this thesis is to study different state management tools in a React application thoroughly. React has been one of the most popular front-end libraries for developing single-page applications, and state management is a fundamental aspect when building a React app. Within React, "state" is an important JavaScript object that can change the behavior of a component, resulting in application re-rendering, based on the user's actions. As developers, it is essential to know different kinds of approaches and patterns, as well as practical use cases and trade-offs of many state management tools.

The thesis will briefly introduce React and why state management is so important so that readers new to the library can learn the basics and those who are already familiar with the technology can sharpen their knowledge. As we understand the significance of state management in React, several libraries will be studied and compared through an example application to help developers consider when each of the tools best serves their purposes.

As this is a broad topic, the libraries that will be discussed in the study are not comprehensive and definitive. Rather, it shows how different tools approach the same problem in their own unique way. Understanding how each tool works allows developers to be able to find the most appropriate state management library depending on their project and personal preferences.

Keywords React, state management, web development, and front-end development

CONTENTS

LIST OF FIGURES AND TABLES

LIST OF ABBREVIATIONS

UI	User Interface
UX	User Experience
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
API	Application Programming Interface
SPA	Single-Page Application
SEO	Search Engine Optimization
DOM	Document Object Model
JSX	JavaScript XML
XML	Extensible Markup Language
ES6	ECMAScript 2015 / ECMAScript 6
URL	Uniform Resource Locator
REST	Representational State Transfer
SWR	Stale While Re-validate
VS Code	Visual Studio Code
MacOS	Mac Operating System
npm	Node Package Manager
npx	Node Package Executor
CLI	Command Line Interface
Href	Hypertext REFerence
ID	IDentification

1. INTRODUCTION

With the rapid advancement of technology, the internet has long become an essential part of human lives. Nowadays, it would be hard to find a person without access to the internet, be it for personal development or to keep up with the latest global news. Because the internet is so important, demands for businesses to be available online increase drastically. Many services that used to only be available physically can now be done virtually. However, customers not only expect these online services to be functional but also secure and responsive.

To build an engaging, unique, and seamless web application, a single-page application (SPA) is the new preference instead of the old multi-page application. A single-page application (SPA) is a web application or website that interacts with the user by dynamically rewriting the current web page with new data from the web server, instead of the default method of a web browser loading entire new pages.[1] The goal is faster transitions that make the website feel more like a native app.[1] This will result in a quick load time and enjoyable user experience (UX), but there are some tradeoff disadvantages, such as search engine optimization (SEO) and browser resource consumption.[2] Among many of the frameworks created for SPA development, some of the more favored ones are AngularJS, ReactJS, VueJS, EmberJS, BackboneJS, and MeteorJS. Though each has its own set of features and functions, this research will focus solely on ReactJS as it is still one of the most popular and widely used front-end frameworks.

The purpose of this thesis is to carry out in-depth research on ReactJS and one of its core concepts – state management. It is recommended that readers have basic knowledge of coding and JavaScript. As a developer, the ability to fluently manage states using different approaches is very important when building a React application. The thesis will cover many well-known state management tools and libraries. Knowing the strengths and weaknesses of each library will help one decide which tool to use depending on the complexity of their application.

2. REACTJS AS A FRONT-END FRAMEWORK

2.1. Front-end and Back-end in Web Development

For web development, the front-end and the back-end are two complementary aspects that work together to create a complete web application. In order to become a good web developer, one must understand these two most popular concepts thoroughly. To put it briefly, the front-end provides the graphical interface of the web application, while the back-end provides the server-side functionality and data management. Developers need to ensure that both the front-end side and back-end side are able to communicate with one another effectively and seamlessly to improve the functionality of the web page and the user experience.

Front-end development involves creating the user interface and user experience (UI/UX) of a website, which includes everything that the users see and interact with directly: the layout, visual design, responsiveness, and interactivity of the website. In order to build the front-end of a web application, HTML, CSS, and JavaScript are used. The goal is to create an attractive and user-friendly interface that provides a smooth experience for the users.

Back-end development involves managing various internal functionality and sever-side databases to ensure that the client-side of the web page works correctly. The primary responsibility of back-end developers is to build and maintain data storages, ensuring that the process of fetching and retrieving data from front-end to back-end is carried out effortlessly. Programming languages, such as PHP, Python, and Node.js are used to build the back-end functionalities, while databases like MySQL and MongoDB are used to store data.

The front-end and back-end communicate with each other through APIs. The front-end sends requests to the back-end for data, and the back-end sends data and responses back to the front-end to be displayed. Therefore, both the front-end and the back-end are equally important for the creation of a successful web application, and comprehensive understanding of both front-end and back-end development will undoubtedly be beneficial for a web developer. This thesis will focus on the front-end aspect of web development and one of its most widely used frameworks – ReactJS.

2.2. What is ReactJS

ReactJS, also known as React.js and React, is a popular open-source front-end JavaScript framework developed by Meta and the community. React is component-based, which means a React application is made up of multiple components, and each component contains a small piece of reusable HTML code. Combining these independent components together allows developers to build any type of application, from a small counter application to more complex ones such as Facebook and Twitter.

Initially developed in 2011 and later publicly released in 2013, React quickly became popular among developers, even after nine years. According to 2022’s Stack Overflow Developer Survey, React is the most wanted web technology and the most common front-end technology used by developers, alongside Node.js for back-end technology.[3] Figure 1 shows the survey for most wanted web frameworks and technologies, with React accounting for 22.54%; while figure 2 shows React.js at 42.62% and Node.js at 47.12% as the most commonly used front-end and back-end frameworks, respectively.[3]

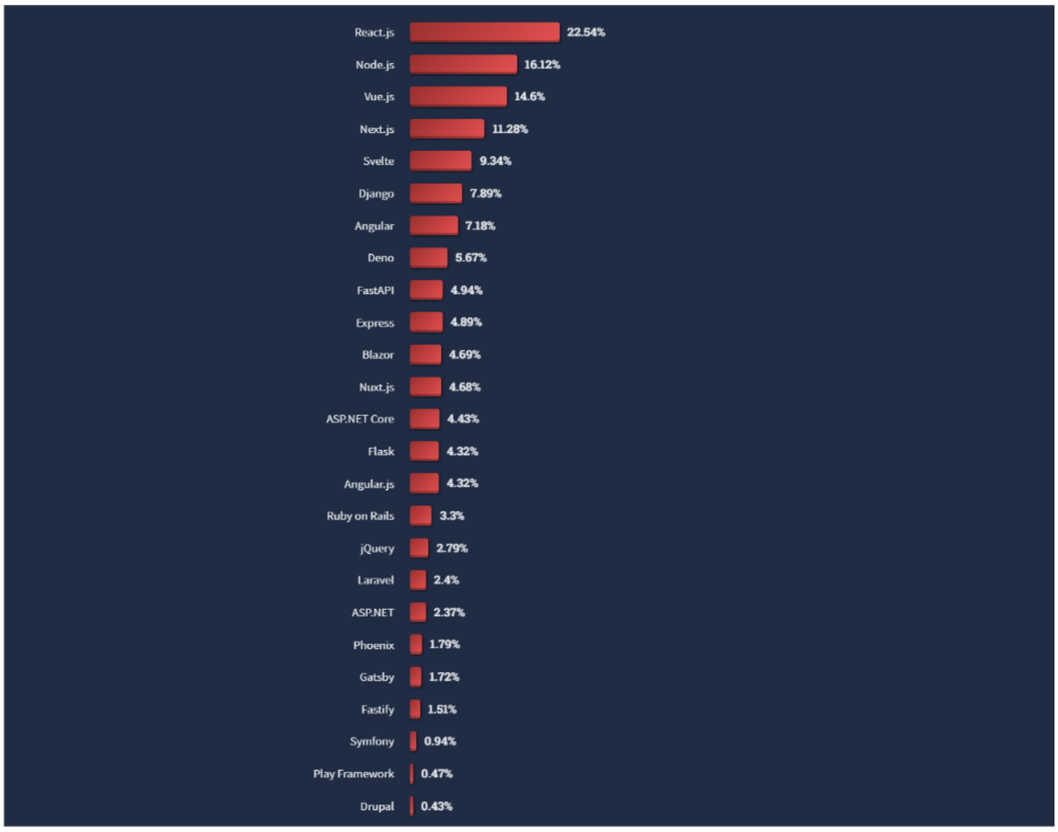


Figure 1. Most wanted web frameworks and technologies

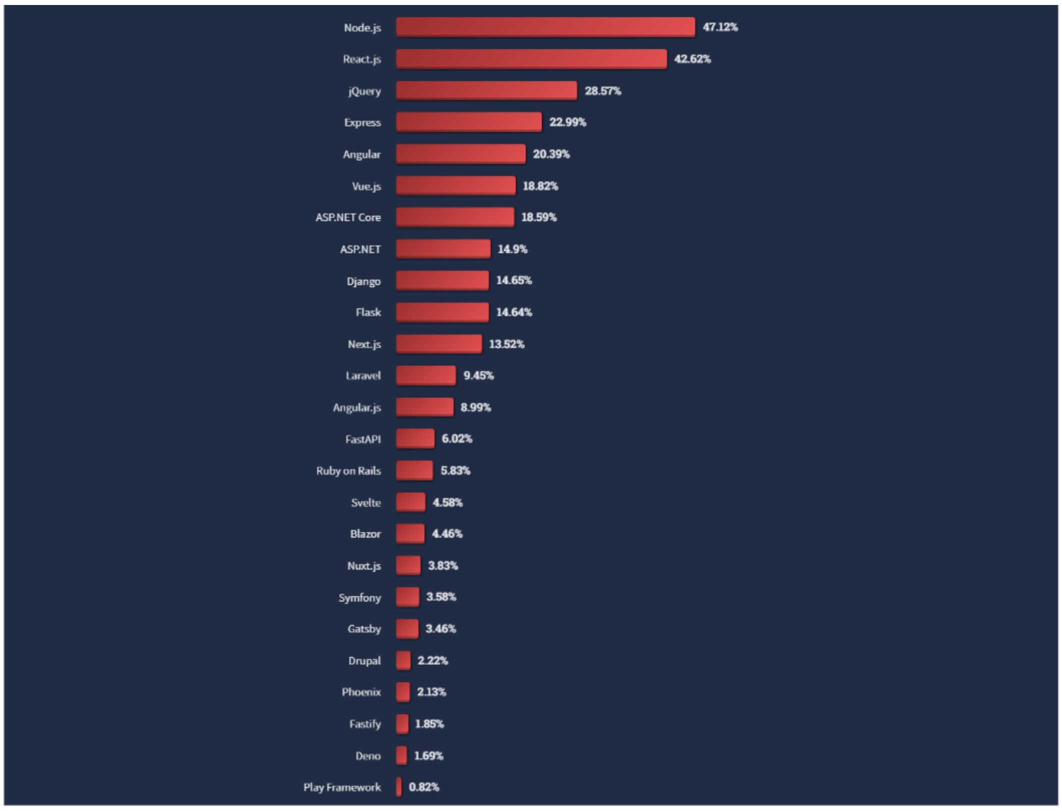


Figure 2. Most commonly used web frameworks and technologies

There are many reasons why React is widely discussed and used among developers and those learning to code. Most of them are because of React’s unique features: component-based architecture, the Virtual DOM, JSX, and one-way data binding. More information regarding these terms will be explained further in the thesis, though it can be concluded that React’s ability to simplify the process of creating and managing the user interface (UI) without sacrificing the application performance was what made React so popular. An article on Noble Desktop stated that another part of React’s success was due to Meta’s (formerly known as Facebook) status as the company that owns the largest social media platform in the world – an interesting yet equitable reason.[4] Nevertheless, developers have discovered that React is easier to learn and implement compared to other alternatives. Companies, such as Airbnb, Netflix, Shopify, Salesforce, Walmart, and Amazon are using React to create the UI on their apps, which not only improves the user experience in their business but also helps boost the framework’s popularity.[4]

2.3. React Components and Props

React allows developers to build complex UI by combining many isolated snippets of code known as components. Components are the heart of a React application, so in order to understand React, one needs to start by understanding React components. On the original documentation website, React defines components as “conceptually like JavaScript functions”.[5] Components allow splitting the UI into independent, reusable pieces, and thinking about each piece in isolation.[5] It accepts arbitrary inputs called “props” and returns React elements describing what should appear on the screen.[5] In other words, one might think of components as simple JavaScript functions that accept inputs and render outputs to the interface. As functions are reusable, components can also be reused if necessary. Merging components, therefore, creates a complex UI for any web application.

Though many documentations, including React’s official document, relates components to JavaScript functions, there are actually two types of React components: class components and functional components. Before React 16.8, class components were the only way to track the state and lifecycle of a React component.[6] Functional components were considered stateless.[6] Now, with the addition of hooks, functional components are almost equivalent to class components.[6] The scope of this thesis will not cover how to use each type of component, but only describes how to declare them:

- A class component can be created using a class that is extended from React component. The syntax for class component declaration is `class MyComponent extends React.Component {...}`
- A functional component can be created using traditional or ES6's function declaration. It means a component can be created by either using `function MyComponent() {...}` or `const MyComponent = () => {...}`

Functional components are preferred due to their simplicity compared to class components, hence they will be used mainly in this thesis. Figure 3 describes how to break down a simple welcome page into components based on its functionality.

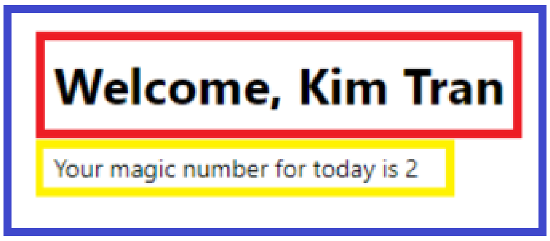
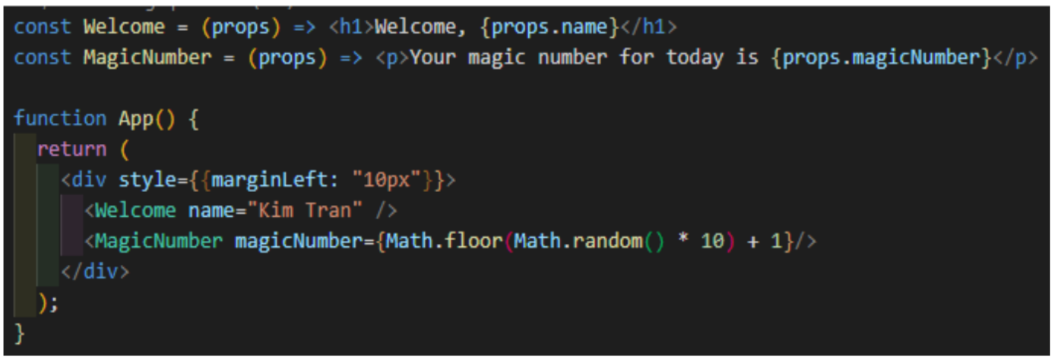


Figure 3. Components of a welcome page

Once identified, clear names should be given to distinguish each component. It is advisable for the name to be as relevant to the component functionality as possible. For instance:

- `Welcome` (red): Page title.
- `MagicNumber` (yellow): A random magic number.
- `App` (blue): The container of all child components.

In a React application, there will always be a component that acts as the container to ensure that all other child components are kept within a structure for easier management. This component is commonly called parent component, or container component. After the components have been identified and created, React will then send them to the virtual DOM for processing, and finally render the updated UI to the real DOM, which is shown on the web application. Because React mainly uses its own virtual DOM to build a web application, more information regarding its workflow will be introduced later in the thesis to avoid confusion. For now, it is sufficient to understand this brief explanation of how React works. The implementation details for these components are shown in Figure 4.

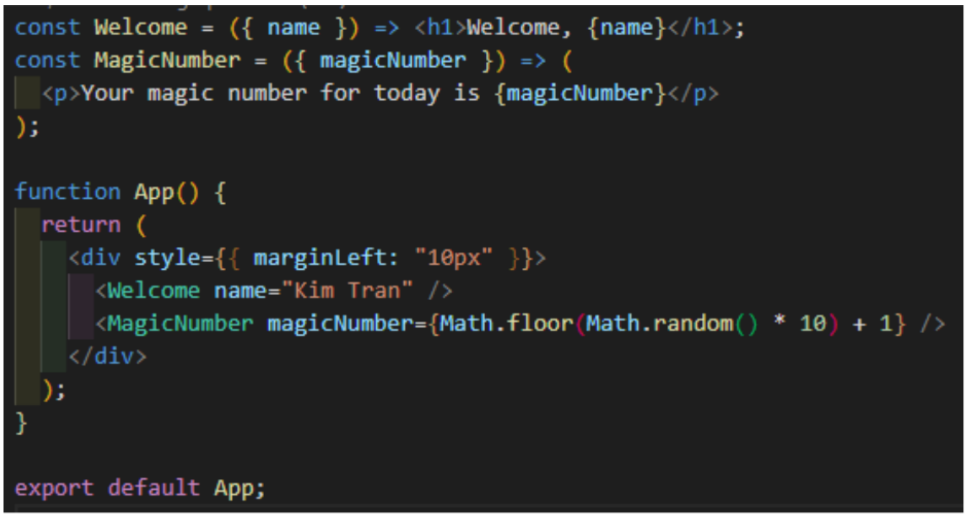


```
const Welcome = (props) => <h1>Welcome, {props.name}</h1>
const MagicNumber = (props) => <p>Your magic number for today is {props.magicNumber}</p>

function App() {
  return (
    <div style={{marginLeft: "10px"}}>
      <Welcome name="Kim Tran" />
      <MagicNumber magicNumber={Math.floor(Math.random() * 10) + 1}/>
    </div>
  );
}
```

Figure 4. Implementation of components

Note that the example uses both traditional function declaration and ES6's arrow function. To understand this simple application, it is essential to know what props are. "Props", which stands for properties, is an object argument with data and returns a React element.[5] First, the `Welcome` component accepts a single `prop` with the data `{name: 'Kim Tran'}` and therefore renders a `<h1>Welcome, Kim Tran</h1>` as a result. The same principles apply to the `MagicNumber` component, with every render returning a different randomized number from 1 to 10.

There are two ways to use props in a component: one without destructuring and one with destructuring. In this implementation, props were declared using the first method, which is by simply defining properties as attributes and passing props as an argument to the component, then proceeding to call them like how one would call an object. The second method, however, is slightly different. Instead of passing props as an argument, the object is destructured and passed in as individual variables. Figure 5 describes this in more detail, note how the props used inside their respective components also change.



```
const Welcome = ({ name }) => <h1>Welcome, {name}</h1>;
const MagicNumber = ({ magicNumber }) => (
  <p>Your magic number for today is {magicNumber}</p>
);

function App() {
  return (
    <div style={{ marginLeft: "10px" }}>
      <Welcome name="Kim Tran" />
      <MagicNumber magicNumber={Math.floor(Math.random() * 10) + 1} />
    </div>
  );
}

export default App;
```

Figure 5. Example with destructured props

Props in React are read-only, thus a component should never attempt to modify its own props. React's documentation stated that "All React components must act like pure functions with respect to their props".[5] Pure functions mean the return value is determined by its input values only and the return value is always the same for the same inputs.[7]

Because this example application is pretty straightforward, the components are small and do not need a lot of work. However, as applications grow, it is essential to know how to extract components efficiently. Components should not be too large that it is difficult to maintain, but they should also be modular enough to cover many cases with the same piece of code. This might seem like a daunting task at first, but having reusable components is one of React's best practices as a web developer.

2.4. The Concept of Virtual DOM

In order to understand what virtual DOM is, it might be beneficial to revisit the definition of the original DOM. Document object model (DOM) is a programming interface for HTML and XML documents.[8] It defines the logical structure of documents and the way a document is accessed and manipulated.[8] An example DOM tree is shown in figure 6, with each node holding both the structure and the behavior of an object.

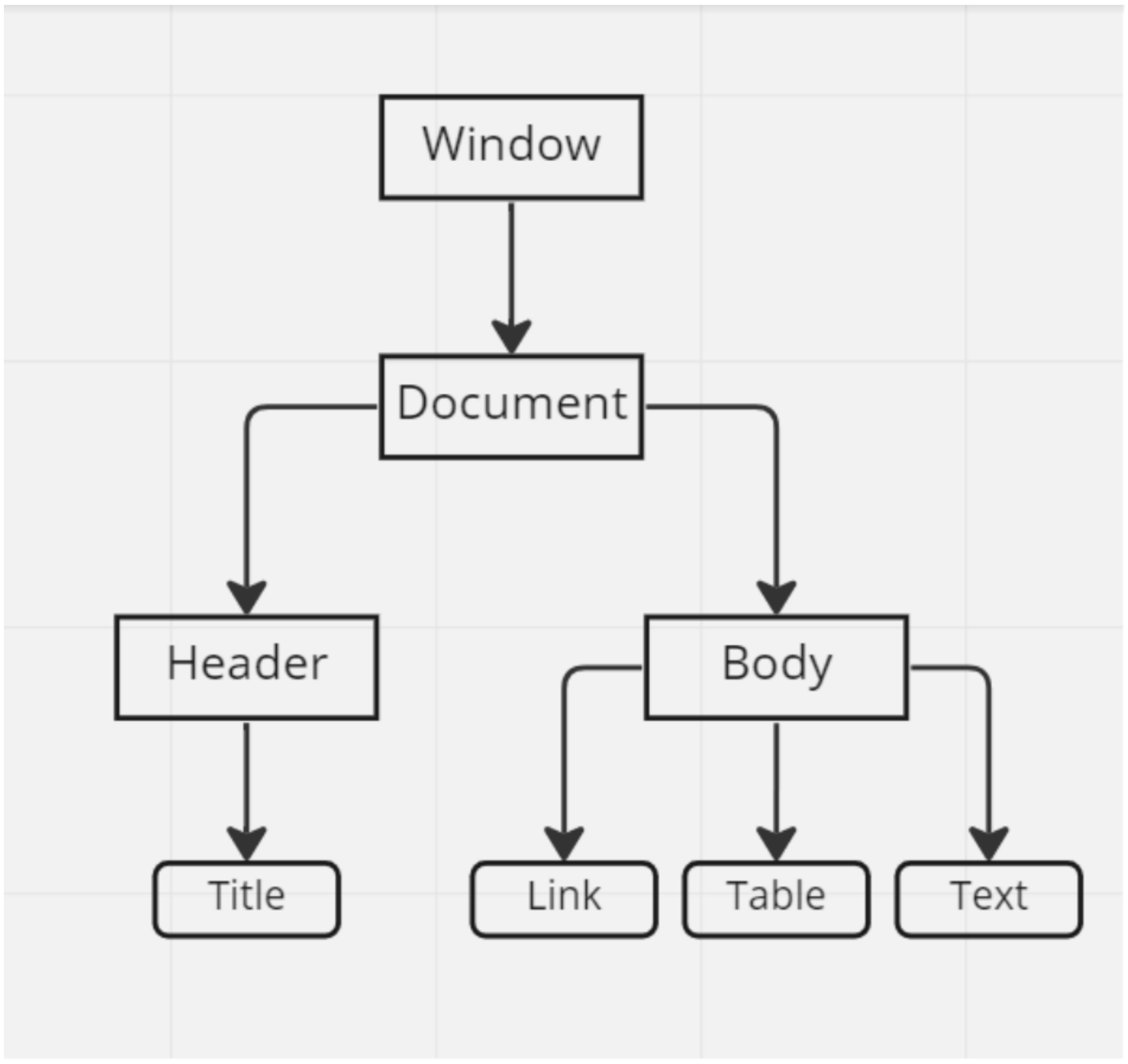


Figure 6. Representation of the DOM

The problem with DOM is that every time a small part of the DOM changes, the browser would have to recalculate the styling, rerun the scripts, and repaint the whole web page. For single-page applications, this process would happen much more frequently due to Javascript having to update the DOM much more. This is why React uses its own virtual DOM to minimize the process of DOM manipulation.

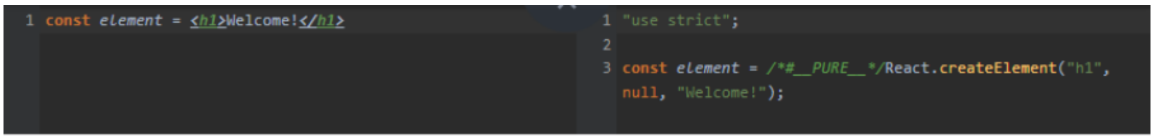
Virtual DOM, as the name implies, is a virtual implementation of the real DOM. It uses a strategy that ensures the DOM only receives the necessary data to repaint the UI. By using this method, the DOM is updated without having to redraw all the web page elements. Virtual DOM is faster than the real DOM because nothing gets drawn onto the screen.

The way it works is whenever an element is added to the UI, a virtual DOM is created. So if the state of this element changes, React creates another new virtual DOM. These two versions then go through a “diffing” process, which is to compare and detect which object has changed. Finally, React updates only that specific object on the real DOM. This is where the optimization happens because without having to re-render the whole web page, the performance cost drops significantly. As stated above, the virtual DOM is one of the many reasons why React became so popular among developers.

2.5. Introducing JSX

Javascript XML, widely known as JSX, is an XML-like syntax extension to Javascript. JSX defines how the UI will look like in a React application. It is powerful because it allows components to contain both markup (HTML) and logic (Javascript) instead of putting them in separate files.[9] Though not required, React developers recommend everyone to use JSX because it makes developing the UI for a React application much easier and more intuitive.

After declaration, JSX will be translated into plain Javascript at runtime. How a JSX syntax is written and compiled is described in figure 7.



```
const element = <h1>Welcome!</h1>
```

```
"use strict";

const element = /*#__PURE__*/React.createElement("h1",
null, "Welcome!");
```

Figure 7. JSX declared and compiled into plain Javascript

Here, the left side is JSX syntax while on the right side it is converted into Javascript. It is worth first noting that JSX is neither a string nor HTML. The reason why JSX needs to be translated into Javascript is because there is no direct implementation that helps browser engines read and understand JSX directly. So React relies on compilers like Babel or Typescript to transform JSX code into regular Javascript, resulting in code snippets that are less human-friendly but will run in browsers.[10] Looking at the result, one can easily conclude that using JSX is much easier to describe the UI for a React application compared to the traditional way.

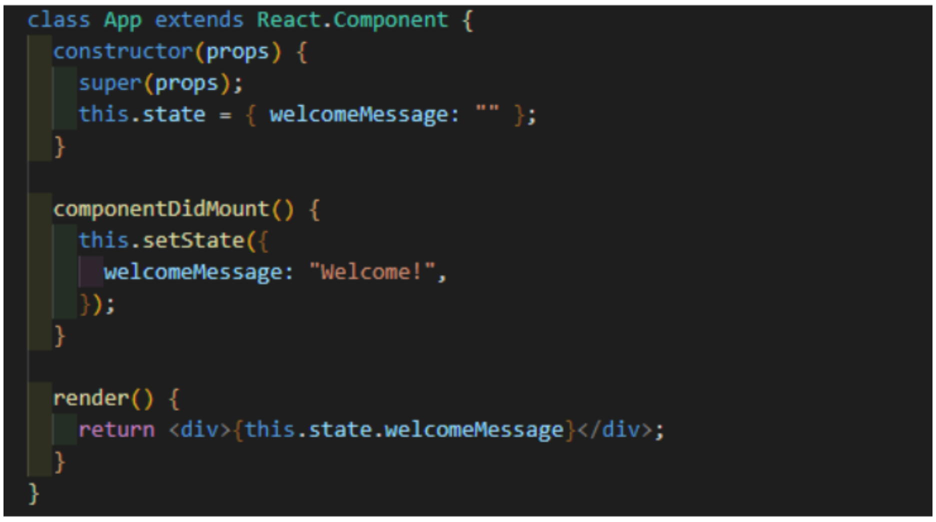
JSX, with its own unique features, is a close resemblance to Javascript. This means in JSX, any valid Javascript expression can be used by putting them inside curly braces, "{ }". JSX itself is an expression as well. It can be assigned to variables, used in conditional statements and loops, accepted as arguments, and returned from functions.[9] Another very important rule of JSX is that when specifying element attributes and event handlers, camelCase property naming convention is used. Examples include using className instead of class, and tabIndex instead of tabindex.[9]

Though the scope of this thesis does not cover JSX in depth, it is undeniable that JSX has made developing React applications effortless. In the beginning, it might be hard to adapt to this new React-exclusive syntax, but with time, understanding JSX can be easier than plain HTML and Javascript.

2.6. States and Lifecycle

The state is a built-in React object where data about the component is stored.[11] React state is closely related to components re-rendering. Whenever the state object changes, the component re-renders.[11] Unlike props, states are mutable, but they are local and specific to the component that owns it. That means no other components can access a component's state data unless it chooses to pass states down as props. To "pass states down as props" means to share a component's internal state as read-only props with other components. Even then, the child component does not know where it received the states from, hence it cannot update the states unless referenced.

Modifying the state is not done directly by reassigning, and should never be done as such. Instead, `this.setState` is used to change the value in the state object. `setState` enqueues all of the updates made to the component state and instructs React to re-render the component and its children with the updated state.[12] It performs a shallow merge between the new and previous state.[12] Figure 8 shows how a state is implemented and updated using `setState`.



```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { welcomeMessage: "" };
  }

  componentDidMount() {
    this.setState({
      welcomeMessage: "Welcome!",
    });
  }

  render() {
    return <div>{this.state.welcomeMessage}</div>;
  }
}
```

Figure 8. State declaration and update using setState()

A class constructor is added to assign the initial state. Note that a class component should always call the base constructor with props passed in. After the component is mounted, meaning it is rendered for the first time, the state is changed for the `welcomeMessage` string. React understands this process through its built-in function `componentDidMount`. `setState` will ensure the component knows it has been updated and calls the `render` method again.

Each component in React has its own lifecycle with three main phases: “Mounting”, “Updating”, and “Unmounting”. For a class component, all three phases have many built-in methods that get called at a certain point, allowing developers to intervene and customize their behavior. Figure 9 is the order of methods called when mounting, updating, and unmounting a component.

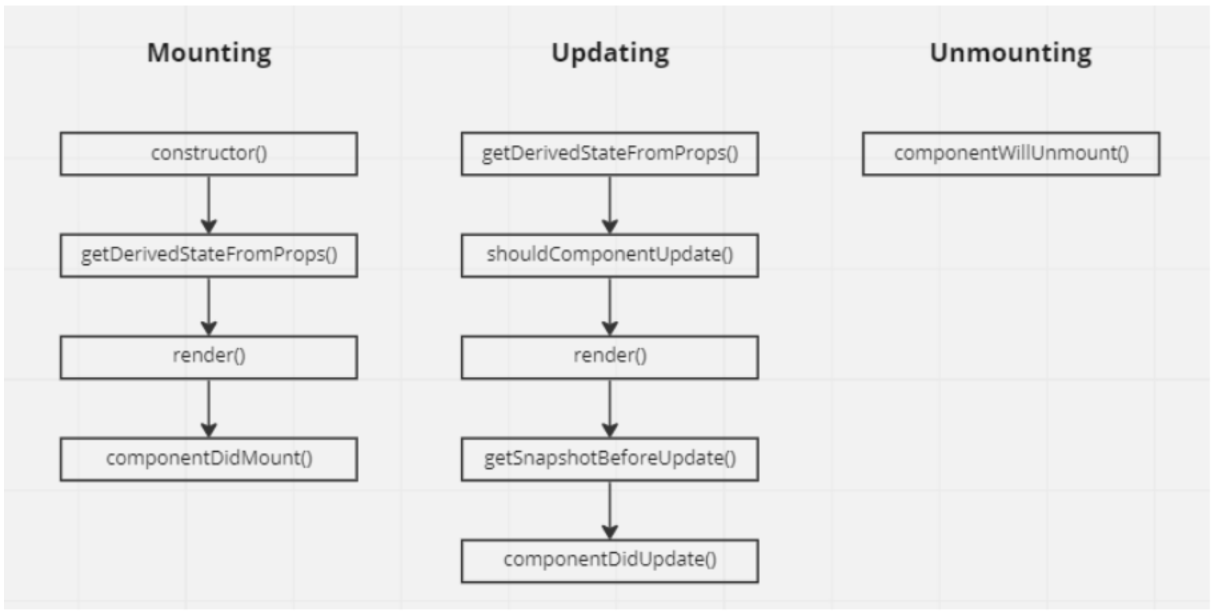


Figure 9. Built-in methods run order for each phase

Mounting is when the component puts elements on the DOM and renders on the web page. It is the first stage of the React component lifecycle. The four methods classified into this stage are as follows:

- `constructor` is called before the component is mounted. It is used to initialize local states and bind event handlers, which is why it needs to be implemented at the top of the component structure.
- `getDerivedStateFromProps` is where the state is set based on the initial props.
- `render` contains JSX code that will be rendered to the DOM when the component is mounted. This method is required as React will call this method automatically independent of the user when props or state changes.

- `componentDidMount`, as the name suggested, is invoked after the component is mounted and rendered on the DOM. This is where statements that require the component to already be placed in the DOM get called.

The next phase in the lifecycle, the Updating phase, is where the component gets updated. The component will get updated when there is a re-render triggered by changes either to its state or props. Updating consists of five methods that will be called:

- `getDerivedStateFromProps` is also called in the update phase. It is the first method to be called and its function is similar to the explanation above.
- `shouldComponentUpdate` returns a boolean value determining if the component should update or not. Conditions can be set here to eliminate unnecessary re-renders. The default return value is true.
- `render` will re-render the DOM, updating the web page with new changes. It is also required, with the same functionality as in the mounting stage.
- `getSnapshotBeforeUpdate` simply gives access to the props and state values before the update. It is important to know that if this method is used, `componentDidUpdate` should also be included.
- `componentDidUpdate` is called after HTML elements have finished loading and updated.

Unmounting is the last stage of a React component's lifecycle. It is when a component is removed, or unmounted, from the DOM. It contains only one built-in method:

- `componentWillUnmount` is invoked right before the component prepares for its destruction. This is where cleanups for anything that was called in `componentDidMount` happen.

It can be seen that the built-in methods have self-explanatory names. These methods control the lifecycle of a React component; though with the introduction of React hooks for functional components, they have decreased in popularity. More about React hooks and how it controls the React component lifecycle will be discussed later.

In the DOM tree, React components can communicate with each other, either by receiving, modifying, or passing data. They can either be the child or the parent of another component. This is how React allows the creation of flexible yet complicated applications. By controlling the components' lifecycle through the usage of built-in methods or hooks, developers can control the flow of the whole application. Learning how to manage states and understanding how a component's lifecycle transpires is an essential part of being a React developer.

2.7. One-way Data Binding

Another reason that made React popular is because of its unique "one-way data binding". It is one of the core architectures of React. One-way data binding, also known as the uni-directional flow of data, is when the data consumer (in this case the View, or UI) gets automatically updated when the data provider (the Model, or component) changes, but not the other way around.[13] It is different from AngularJS or EmberJS's two-way data binding. Two-way data binding is where changes from the data consumer or data provider update the other.[13]

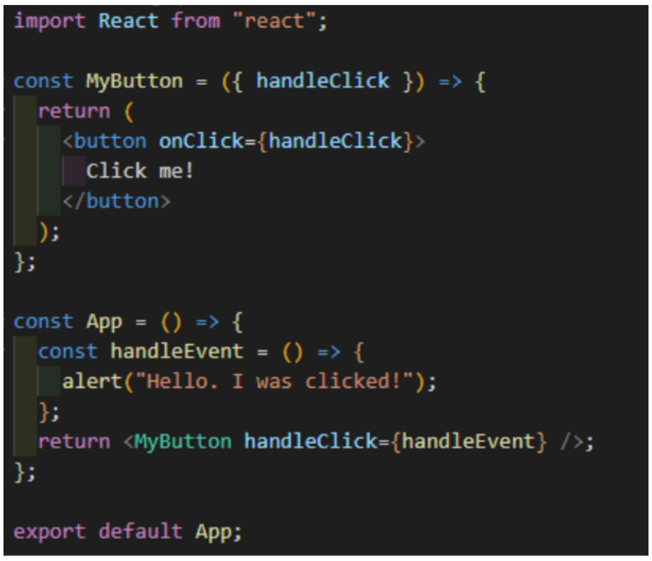
In order to update the Model from the View in React, event listeners need to be defined. This is where the component knows it needs to respond to changes made from the UI. The typical methods used are `onChange` for inputs or `onClick` for buttons.

One-way data binding might be easier to understand and follow for beginners compared to two-way data binding. It offers more control and helps avoid side effects by ensuring the data flows throughout the application in a single direction. Knowing how the data flows can help predict the result or debug the problem should anything go wrong.

2.8. Handling Events in React

Event handlers in React determine what actions should be taken whenever an event occurs. React has the same events as HTML, when the user clicks a button, changes an input field, or hovers over a banner. Though handling events in React is similar to handling DOM events, there are a few exceptions. Firstly, React events are written in camelCase instead of flatcase like in DOM events. So for a click event, it should be written `onClick` instead of `onclick`. Secondly, React event handlers should be put inside curly braces, so it is `onClick={handleEvent}` instead of `onclick=”handleEvent()”`. Notice the lack of parentheses for React’s function call. This is because including parentheses will cause the function to run automatically, rather than only after a click event. If arguments need to be passed to event handlers, an arrow function needs to be used. So `onClick={handleEvent}` will become `onClick={() => handleEvent(arg)}`. With only a few differences, one should have no difficulties learning how to handle events in React if they are familiar with how events work in HTML and Javascript.

Figure 10 is an example of event handling in React. In this code snippet, a prop named `handleClick` is passed to the `<MyButton />` component, which then gets passed as the `onClick` event handler for `<button />`.



```
import React from "react";

const MyButton = ({ handleClick }) => {
  return (
    <button onClick={handleClick}>
      Click me!
    </button>
  );
};

const App = () => {
  const handleEvent = () => {
    alert("Hello. I was clicked!");
  };
  return <MyButton handleClick={handleEvent} />;
};

export default App;
```

Figure 10. Component with event handler

2.9. React Hooks

Hooks were first introduced in React 16.8, which was released in 2019. They allow the usage of state and other features of React without writing a class.[14] React hooks are Javascript functions that allow you to “hook into” React state and lifecycle features from functional components.[14] There are two rules when using hooks:

- Hooks should only be called from the top level. This means they should never be called inside loops, conditional statements, callbacks, or nested functions.
- Hooks should only be called from React functional components. This means they do not work inside regular Javascript functions, or more importantly, class components.

Following these rules ensures hooks will always be apparent in the source code and preserves the order of hooks called on every render. This helps React associate and return the right data from each hook on every render.

Ever since its release, many built-in hooks have been introduced by React. Some of them are less popular while some are more commonly used, though they are all very useful to learn. A list of all available hooks and their usage can be viewed on React’s official “Hooks API Reference” documentation page. These are the few hooks that are seen more often:

- `useState` is used to manage states. This hook when called will return a stateful value and an updater function similar to `setState`.
- `useEffect` is used to manage effects like changing the DOM, fetching data, setting timers, mutations and subscriptions. It is the equivalent of `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` – the three methods from React class’ lifecycle combined into one single API.

- `useContext` is used to return the current value for a context. A context in React allows passing down and consuming data in any component needed without using props.
- `useReducer` is an alternative to `useState` which helps manage more complex states. Instead of returning an updater function, it returns a dispatch method.
- `useCallback` returns a memoized callback that only changes if one of the dependencies passed in has changed. This prevents re-rendering unnecessarily.
- `useMemo` returns a memoized value that will only be recomputed when one of the dependencies has changed. The only difference between `useCallback` and `useMemo` is the thing that is memoized.
- `useRef` returns a mutable ref object with an initialized `.current` property. It is useful to store values that are not needed for re-rendering as `useRef` does not trigger a re-render when updated.

It can easily be seen from the list above that each of these hook's name starts with "use". This is standard practice for React hooks. Because React gives developers the ability to create their own custom hooks, it will be easier to identify a hook in the codebase using this rule. A custom hook is just a Javascript function whose name starts with "use" and may call other hooks.[15] Building a custom hook allows sharing a variety of reusable logic between components, a flexibility that wasn't possible with class components.

By introducing hooks, React has offered a simpler way to work with states and lifecycle methods. Although they have not covered all use cases for classes, like the more uncommon lifecycle methods, they are undeniably making React even more appealing for developers who are new and want to learn React.

3. BASIC CONCEPTS OF STATE MANAGEMENT

3.1. Why State Management Is Important in React

State has long been one of the most difficult parts when working with React. Understanding how state flows inside the application structure can be daunting, especially for larger applications. As the codebase becomes more and more complex, it gets even more complicated to manage states shared across multiple components. Using React alone is insufficient to tackle this problem, instead, the community has created a different solution – state management libraries. As soon as they came out, state management libraries quickly caught developers' attention and have been growing rapidly ever since.

First, let us understand what state management is. State management is simply a way to control the flow of states and communication between components, yet it is an important and unavoidable feature of a React application. While the concept itself is not hard to understand, managing states can get confusing and chaotic as the application grows bigger. For example, if a newly created component needs data from the states of a component at the same level, the states will have to be lifted to the closest parent component that is common to both child components that need them, then get passed down. This process is called "lifting the state up", which newer developers might find overwhelming and difficult to maintain. As a result, redundant or duplicate states might appear as a common source of bugs. This is why it is essential to use state management tools, such as Redux, Recoil, and many others.

State management libraries help the process of monitoring data easier. It serves as the single source of truth so that state values are not scattered throughout the whole application but instead well organized, thus making the development process much faster and less painful. Given its numerous advantages, it is not hard to understand why the React community has started creating state management libraries. The number of libraries available becomes enormous as React continues to grow. Each tool, even though serving the same purpose, has its own uniqueness and drawbacks. As developers, it is significantly important to know which library to choose so that it is efficient and suitable for the development of a particular project.

3.2. Timeline

Before getting to know React state management libraries and their development, it is important to first understand some terms that are commonly used. Note that these are not official names, and other variations of naming might be seen, but the underlying meaning stays the same. The terms included are:

- UI State, or Local State – the state of the interactive part of the application, for example, dark mode toggles, modals, and settings.
- Form State – the state of a form in the application, for example, loading, submitting, validating, and disabling.
- Server Cache State – the state from the server side which is cached for easy access on the client side, for example, API calls.
- URL State – the state of the application managed by the browser, for example. Filtering and querying.

Over the years, many specialized tools have been created to manage each of these specific states. The invention of these tools has made the process of working with states effortless by automating things that used to be done manually. Knowing the available possibilities will undoubtedly make the development process easier whether working with a particular state or with all of them.

3.2.1. UI State

Most UI states are triggered by user actions, and normally reset on page reload. Because UI states can change from being very simple to very complicated easily, there have been ongoing releases of many UI state management libraries. The following list, though not comprehensive, is a rough timeline of when popular UI state management libraries were invented:

- 2015: Redux.
- 2016: MobX.
- 2018: Context.
- 2019: React Hooks, Zustand, XState.
- 2020: Jotai, Recoil, Valtio.

Currently, none of these libraries hold the most favorite spot within the React community. Each has its own set of use cases, so a developer is not required to learn all of them. Instead, understanding how they work and when to use them is sufficient.

3.2.2. Form State

For simple forms managing them is easy and can be done by the developers themselves. But because a form has many trigger events and validations, it might help to reach for its own libraries, especially for complex cases. Three of the most used form handling libraries are:

- 2015: Redux Form.
- 2017: Formik.
- 2019: React Hook Form.

Figure 11 describes the number of downloads in the past year for these three libraries, taken from npmtrends.[16]

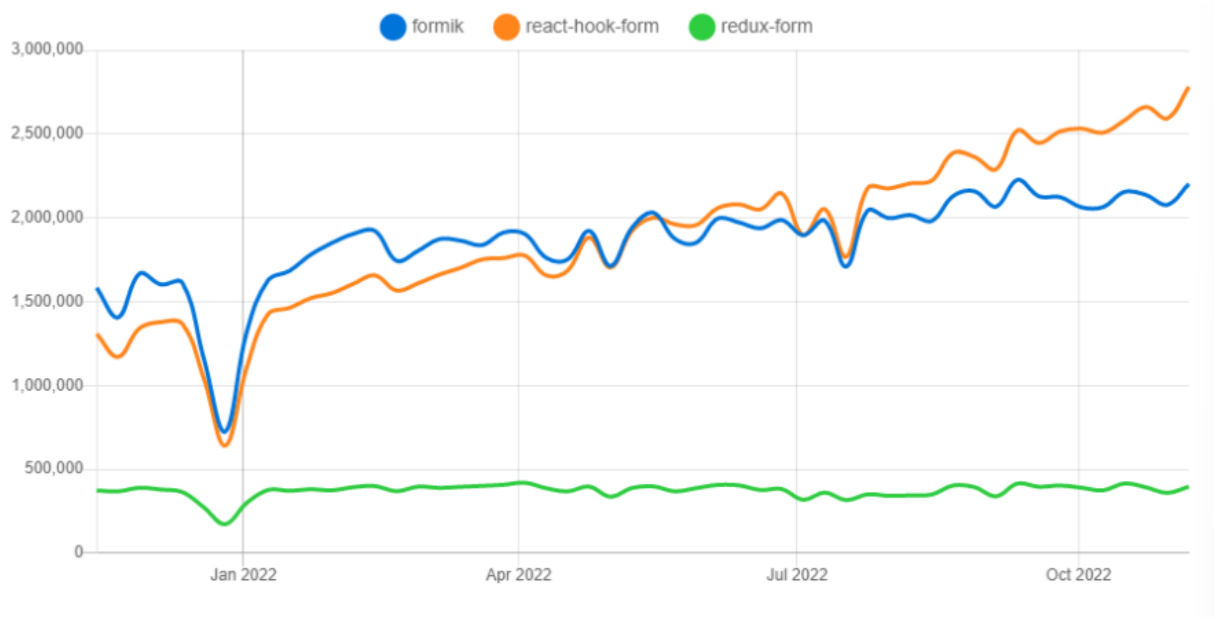


Figure 11. Form management libraries download numbers in 2022

It can be seen that Redux Form is not as popular as React Hook Form or Formik. Even so, React Hook Form has quickly taken over the chart due to its exceptional performance with fewer re-renders and quicker mount time, as compared to Formik.

3.2.3. Server Cache State

Server state data is stored on the server and the process of fetching them is asynchronous. This means developers do not have full control over the data and the time it takes to fetch the data is not pre-determined. Therefore, handling server data might be difficult and tricky if done without specialized state management libraries. Fortunately, there are many libraries that help automate the process, drastically reducing the amount of manual work that needs to be done. Here is the timeline of the currently most popular libraries to handle RESTful server state:

- 2016: Relay Runtime.
- 2019: Apollo Client, SWR (Stale While Re-validate), and Redux Toolkit.
- 2020: React Query (now known as TanStack Query).

For server state management libraries, there is a clear favorite: React Query is currently the most popular library to handle REST APIs. It is a sophisticated library, yet it makes fetching, caching, synchronizing, and updating data easy by providing hooks that handle most of the issues one might face.

3.2.4. URL State

URL path, filtering, and query parameters all play a part in how a web page is displayed. Without helper functions provided by dedicated routing libraries, accessing URL parameters can be hard. Some of the more popular ones are:

- 2015: React Router
- 2021: React Location

Though React Location is becoming a worthy alternative as developers start to recognize its benefits, React Router is still dominating the URL state management libraries due to its longer lifespan and more extensive documentation.

3.3. Conclusion

State management in React has and will continue to evolve in the coming years. It is the most difficult part of building large web applications with React. Understanding different types of states and their management system, as well as their advantages and trade-offs, is crucial for making a reasonable decision based on the objective of the application.

There are clear winners for Form, Server, and URL state management libraries, though the same cannot be said about UI state management. The decision is left to the developer to decide. Besides introducing the best tools for other states, this thesis will provide in-depth knowledge of some of the most popular UI state management tools so that one can determine which is the best according to their needs and preferences.

4. CREATING AN EXAMPLE REACT APPLICATION

4.1. Installation

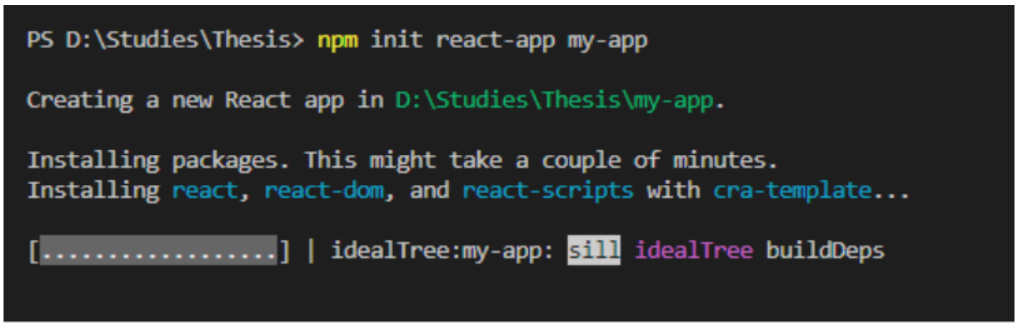
Depending on the requirements and preferences, there are many source-code editors to choose from. Some things to consider while choosing are: free or paid usage, multiple coding language support, syntax highlighting, auto-completion, built-in compiler and debugger, as well as many other quality-of-life features. It is up to the developer themselves to decide which editor program works best, and there is no right or wrong when it comes to personal choices.

Currently, the most used source-code editor program is Visual Studio Code. In summary, Visual Studio Code (commonly referred to as VS Code) is a fast, free, and extensible editor that supports hundreds of software development languages. It is available on MacOS, Linux, and Windows. It helps make the development process seamless by providing error correction, syntax highlighting, code autocompletion and formatting, a built-in debugger, and Git commands. Users can customize Visual Studio Code with extensions like color themes or support for programming languages. Due to its popularity and ease of usage, Visual Studio Code will be used to edit code snippets in this thesis.

After Visual Studio Code is installed, Nodejs version 16 is needed to shorten the process of initializing a React application. Downloading and installing Nodejs will introduce two additional CLI tools that are commonly used: a package manager called npm, and a package executor called npx. These two CLI tools, with many useful commands, will help install and run dependency packages in the terminal.

4.2. Basic setups

After all the necessary installations are made, let us now start by creating a React application. First, the command `npm install create-react-app` was run in the built-in terminal of Visual Studio Code to offer a modern and straightforward build setup with no configuration for React applications. Once the “create-react-app” command has been installed, it introduces two methods of creating an application, either by typing `npx create-react-app my-app` or `npm init react-app my-app`. Figure 12 shows the application being installed.



```
PS D:\Studies\Thesis> npm init react-app my-app

Creating a new React app in D:\Studies\Thesis\my-app.

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

[..................] | idealTree:my-app: sill idealTree buildDeps
```

Figure 12. React app being installed

Once this is done, there will be a few commands suggested, with the last two being how to start the app on the local server. Running the two suggested commands one by one will result in an application similar to Figure 13 being shown on the browser.

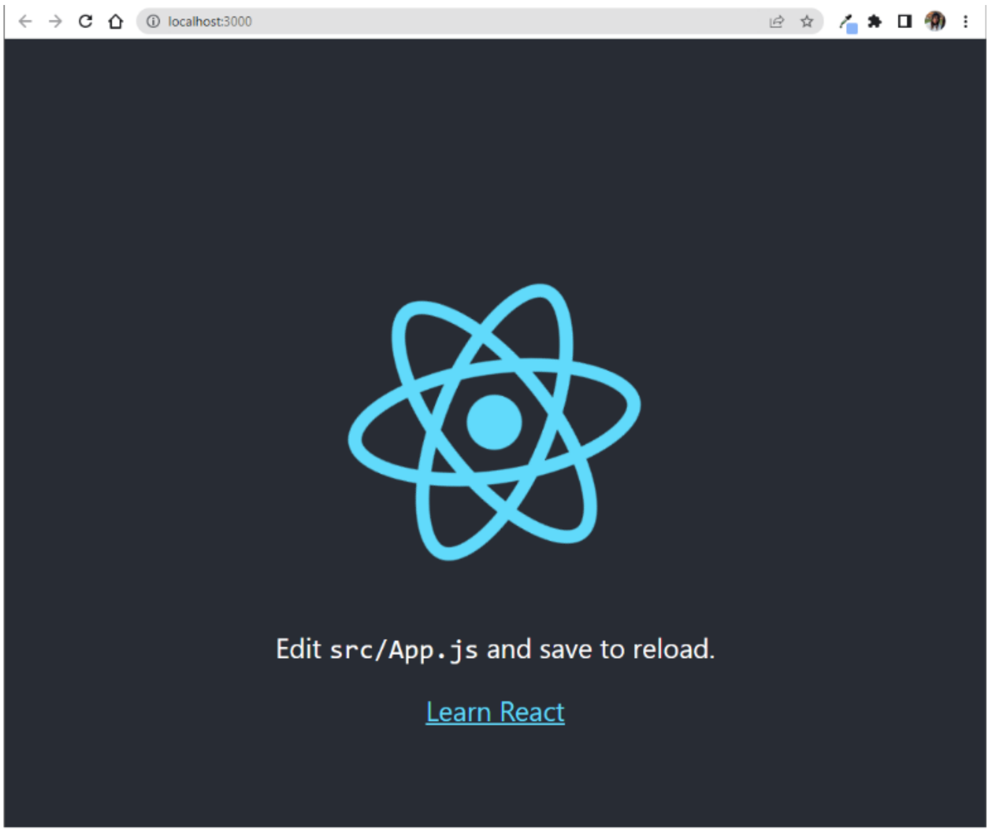
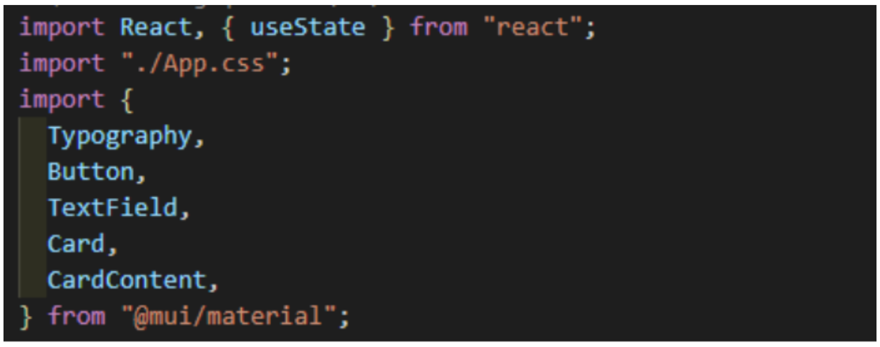


Figure 13. The initial state of a React application

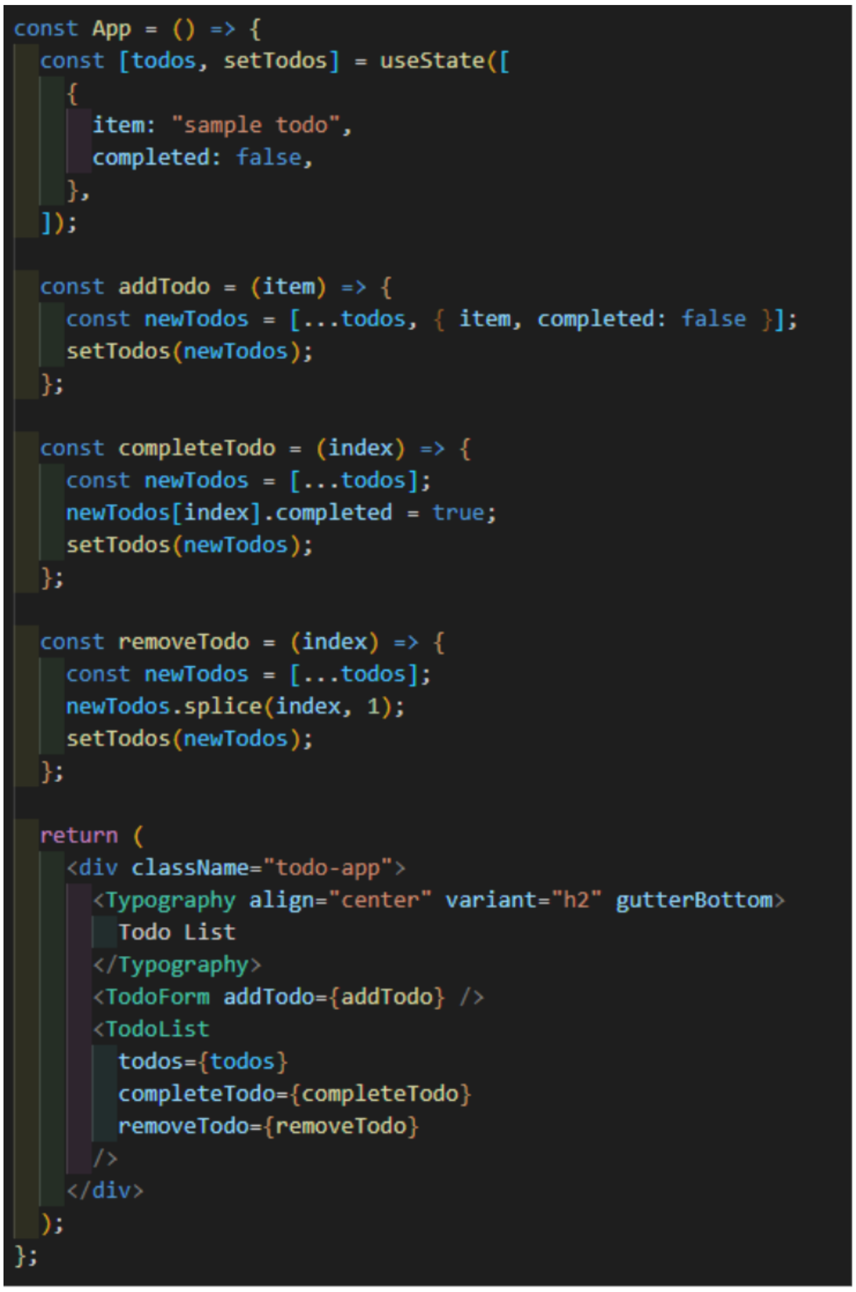
Note that the application is run on the local server. A to-do list application will be built for the purpose of this thesis. In order to achieve a pleasant UI without spending too much time styling the application, Material UI's core packages called "@mui/material", "@emotion/react", and "@emotion/styled" will be installed in the terminal, then the imports will be added to the "App.js" file, as demonstrated in Figure 14 .



```
import React, { useState } from "react";
import "./App.css";
import {
  Typography,
  Button,
  TextField,
  Card,
  CardContent,
} from "@mui/material";
```

Figure 14. App.js file imports

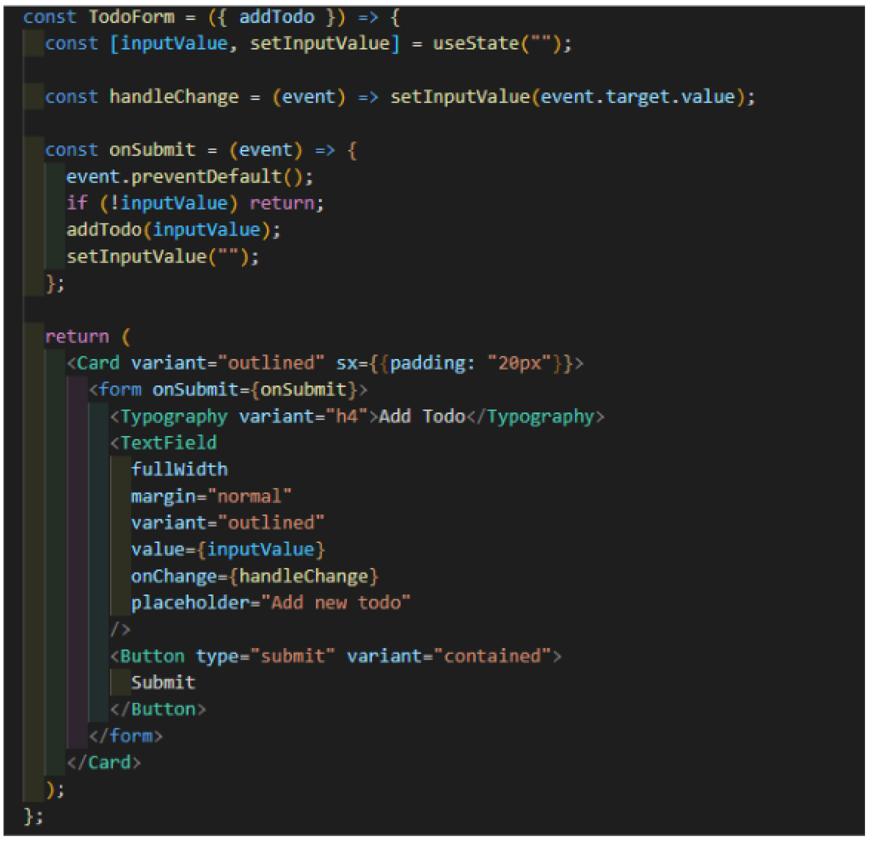
Material UI offers a wide variety of UI tools which will help building an application faster with less effort required. Within the same file, an `App` component will be created to store a list of to-do items, event handler functions, as well as what will be rendered to the screen. The list of to-do items will be an array of objects initialized using React hooks' `useState`, and all the event handler functions will use the returned `setTodos` function to add, update, or delete an item in the array accordingly. Figure 15 shows the completed implementation of the `App` component.



```
const App = () => {
  const [todos, setTodos] = useState([
    {
      item: "sample todo",
      completed: false,
    },
  ]);

  const addTodo = (item) => {
    const newTodos = [...todos, { item, completed: false }];
    setTodos(newTodos);
  };

  const completeTodo = (index) => {
    const newTodos = [...todos];
    newTodos[index].completed = true;
    setTodos(newTodos);
  };

  const removeTodo = (index) => {
    const newTodos = [...todos];
    newTodos.splice(index, 1);
    setTodos(newTodos);
  };

  return (
    <div className="todo-app">
      <Typography align="center" variant="h2" gutterBottom>
        Todo List
      </Typography>
      <TodoForm addTodo={addTodo} />
      <TodoList
        todos={todos}
        completeTodo={completeTodo}
        removeTodo={removeTodo}
      />
    </div>
  );
};
```

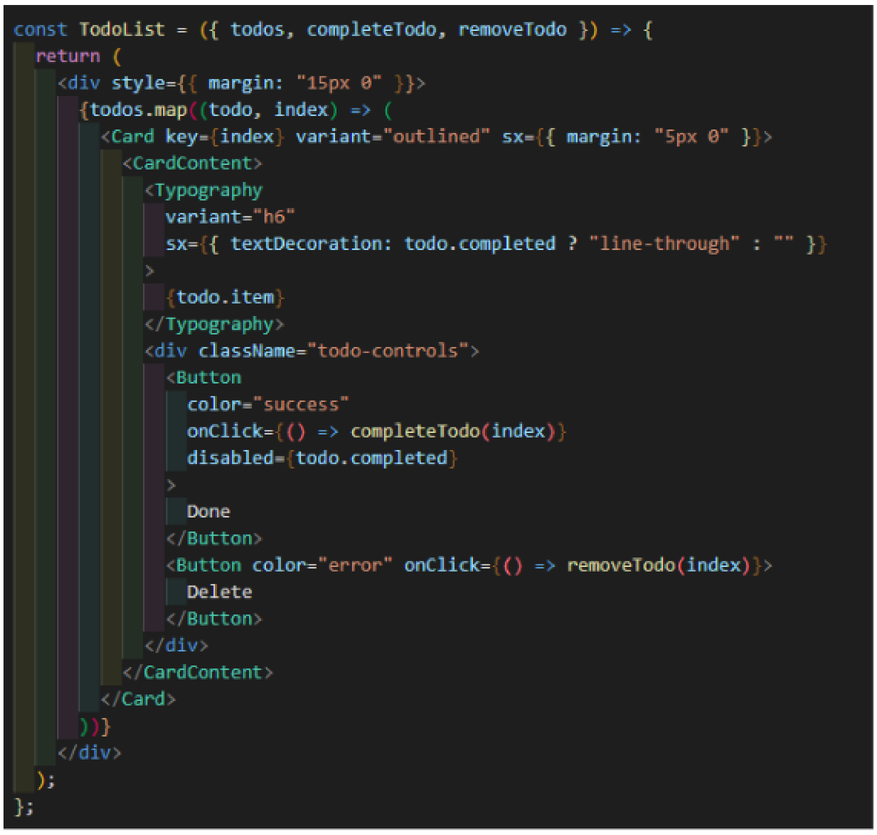
Figure 15. Full implementation of App component

In this implementation, there are two other components that need to be defined: `TodoForm` and `TodoList`. `TodoForm` is a form that adds an item to the list when the submit button is clicked, whereas `TodoList` displays all items in the list, with each item including two buttons for marking the to-do as completed and deleting the to-do. Figures 16 and 17 implement `TodoForm` and `TodoList` respectively.



```
const TodoForm = ({ addTodo }) => {
  const [inputValue, setInputValue] = useState("");

  const handleChange = (event) => setInputValue(event.target.value);

  const onSubmit = (event) => {
    event.preventDefault();
    if (!inputValue) return;
    addTodo(inputValue);
    setInputValue("");
  };

  return (
    <Card variant="outlined" sx={{padding: "20px"}}>
      <form onSubmit={onSubmit}>
        <Typography variant="h4">Add Todo</Typography>
        <TextField
          fullWidth
          margin="normal"
          variant="outlined"
          value={inputValue}
          onChange={handleChange}
          placeholder="Add new todo"
        />
        <Button type="submit" variant="contained">
          Submit
        </Button>
      </form>
    </Card>
  );
};
```

Figure 16. TodoForm implementation



```
const TodoList = ({ todos, completeTodo, removeTodo }) => {
  return (
    <div style={{ margin: "15px 0" }}>
      {todos.map((todo, index) => (
        <Card key={index} variant="outlined" sx={{ margin: "5px 0" }}>
          <CardContent>
            <Typography
              variant="h6"
              sx={{ textDecoration: todo.completed ? "line-through" : "" }}
            >
              {todo.item}
            </Typography>
            <div className="todo-controls">
              <Button
                color="success"
                onClick={() => completeTodo(index)}
                disabled={todo.completed}
              >
                Done
              </Button>
              <Button color="error" onClick={() => removeTodo(index)}>
                Delete
              </Button>
            </div>
          </CardContent>
        </Card>
      ))}
    </div>
  );
};
```

Figure 17. TodoList implementation

Now that the application is running normally, it can be used to experiment with the different React state management tools to see how they perform. Note that this will be used as the starting point for every state management library. In other words, no library will be built on top of each other. Form state, server state, and URL state management tools will be demonstrated first in that particular order. As there is a variety of choices for UI state, it will be introduced last and will be the main focus of this thesis.

5. REACT STATE MANAGEMENT TOOLS

5.1. Form State

5.1.1. React Hook Form

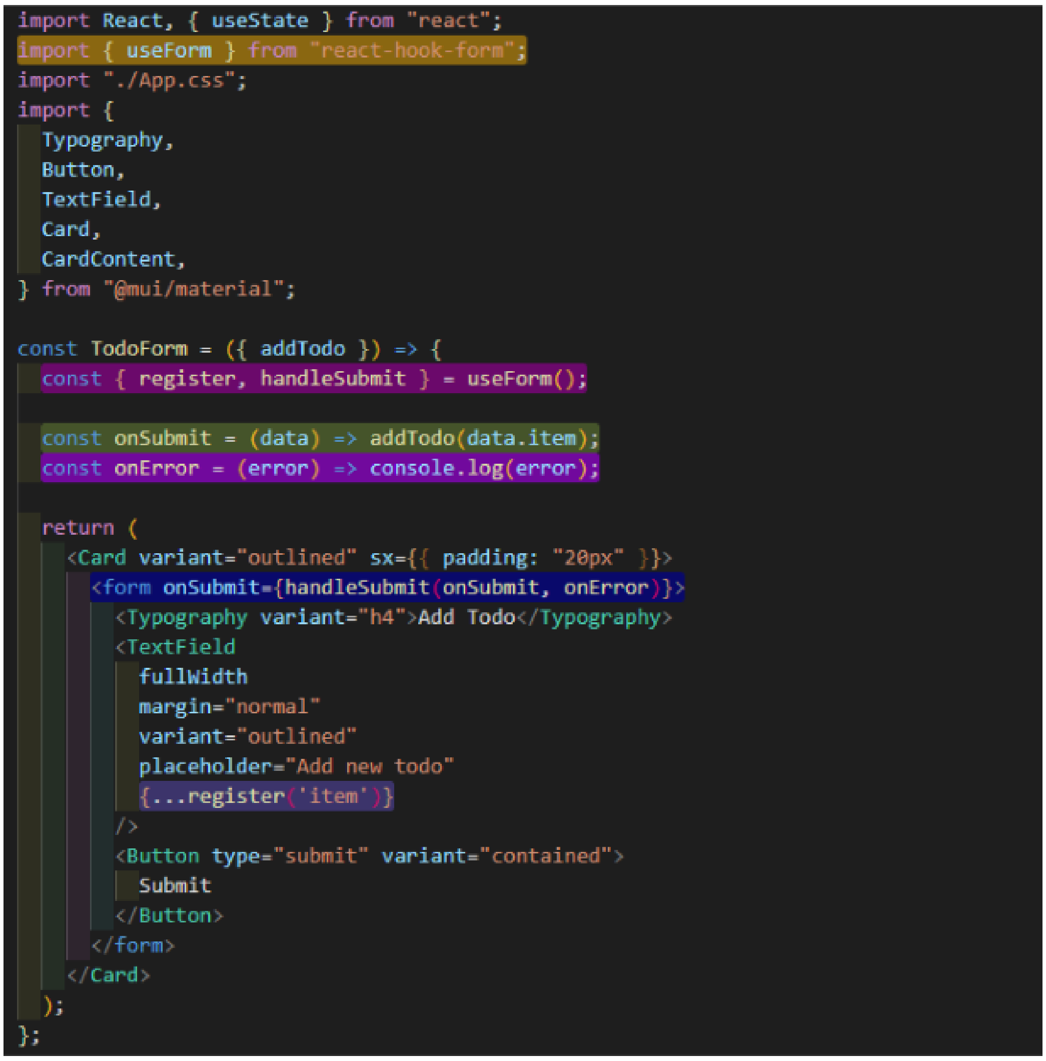
Form is an essential part of how users interact with web applications. There is no limitation to what kind of data a user might be required to put into the form, so validating form data is crucial in preventing security risks; though it may involve many dependencies and complicated authenticating functions. Fortunately, React Hook Form provides seamless form control and validation, allowing developers to have the same functionality but with fewer lines of code in their application.

React Hook Form is a lightweight library that requires no other dependencies. It adopts the usage of `ref` for form inputs rather than depending on the component state such as other libraries. This way, the amount of re-rendering that occurs when the user changes the form data reduces significantly, resulting in a faster mounting process. The library also provides helper functions through its hook, `useForm`, so developers have less code to maintain as they do not need to create custom event handlers. In summary, React Hook Form is more efficient than other libraries because it simplifies the handling of form events.[17]

To demonstrate the usage of this library, `npm install react-hook-form` is run in the terminal to install the package. After the installation is completed, the `useForm` hook from the “react-hook-form” package is imported. The hook returns one object containing a total of fifteenth useful methods and accepts one object with ten configurable properties as an optional argument when called. React Hook Form’s official documentation offers a detailed explanation of all these arguments. For now, there are two methods – `register` and `handleSubmit` – that will be used in the example application. They are obtained from the hook by adding `const { register, handleSubmit } = useForm();` to the `TodoForm` component. The `register` method, as the name suggests, helps to register input field data so that it can be tracked upon changes and validated on request. Invoking the function will return the following properties:

- `onChange` subscribing to the change event of the input.[18]
- `onBlur` subscribing to the blur event of the input.[18]
- `ref` is the input reference for the form hook to register.[18]
- `name` is the input name for the form hook to register.[18]

This method will be passed into the `TextField` field of the `TodoForm` component, replacing the `value` and `onChange` attributes. The other method which will be used, `handleSubmit`, manages form submission by accepting two arguments: one for a successful callback, and one for an error callback. The error callback is an optional argument, but it is advisable to try and catch any error that might occur. Figure 18 shows the `TodoForm` component rewrote with React Hook Form library, with the new and modified lines highlighted.



```
import React, { useState } from "react";
import { useForm } from "react-hook-form";
import "./App.css";
import {
  Typography,
  Button,
  TextField,
  Card,
  CardContent,
} from "@mui/material";

const TodoForm = ({ addTodo }) => {
  const { register, handleSubmit } = useForm();

  const onSubmit = (data) => addTodo(data.item);
  const onError = (error) => console.log(error);

  return (
    <Card variant="outlined" sx={{ padding: "20px" }}>
      <form onSubmit={handleSubmit(onSubmit, onError)}>
        <Typography variant="h4">Add Todo</Typography>
        <TextField
          fullWidth
          margin="normal"
          variant="outlined"
          placeholder="Add new todo"
          {...register('item')}
        />
        <Button type="submit" variant="contained">
          Submit
        </Button>
      </form>
    </Card>
  );
};
```

Figure 18. TodoForm rewrote with React Hook Form

It can be seen that the number of coding lines has been significantly lowered. For instance, the custom `onSubmit` function that used to contain four lines of code now only contains one, and the logic is very much simplified as well. Other than that, no additional state needs to be defined to control the input value.

Though this example shows only the very basics of React Hook Form, the library is mostly popular for its form validation. Validation properties can be passed into the register function as the second argument, and it covers basically everything: from being required, minimum and maximum input length, to validating patterns and conditions. In addition to this, other validation schemas like Yup, Zod, Joi, or Superstruct are also supported, allowing developers to customize their application forms effortlessly. When the validation process fails, error messages can be shown with `useForm` hook's returned method property `formState.errors`.

With almost everything needed to handle web forms built into its hooks, it is no longer a surprise when the React Hook Form library surpasses Formik in popularity. It is easier to use, has less code to maintain, and no additional libraries need to be installed despite its small package size. The choice is certainly up to the developers to decide which one best suits their application's demands, but it is undeniable that React Hook Form – a library that focuses more on developer experiences – has proven itself to be successful.

5.2. Server Cache State

5.2.1. React Query

When developing an application, there will be times when user information needs to be kept for future use. Data that is stored on the client side will be lost when the web page is closed, which is why server-side databases are created. These server-side data are organized remotely, which means the front-end does not have direct control over them. Therefore, communications between the back-end side and the front-end side might be challenging without the use of state management libraries. Since its release, React Query has been chosen by many developers as their favorite tool for this purpose.

React Query, or Tanstack Query, has become the most popular solution for server state management in React applications. It provides everything from fetching and updating to caching and synchronizing data under its own custom hooks. This helps to completely remove the need for hooks and global state when operating with remote data, resulting in less code to write and maintain.

Just like React Hook Form, React Query has no dependencies. React Query also provides many custom hooks and utility functions, as well as its higher-order components. All of this can be found on their API reference page. This thesis will cover the three core concepts of React Query: queries, mutations, and query invalidation.

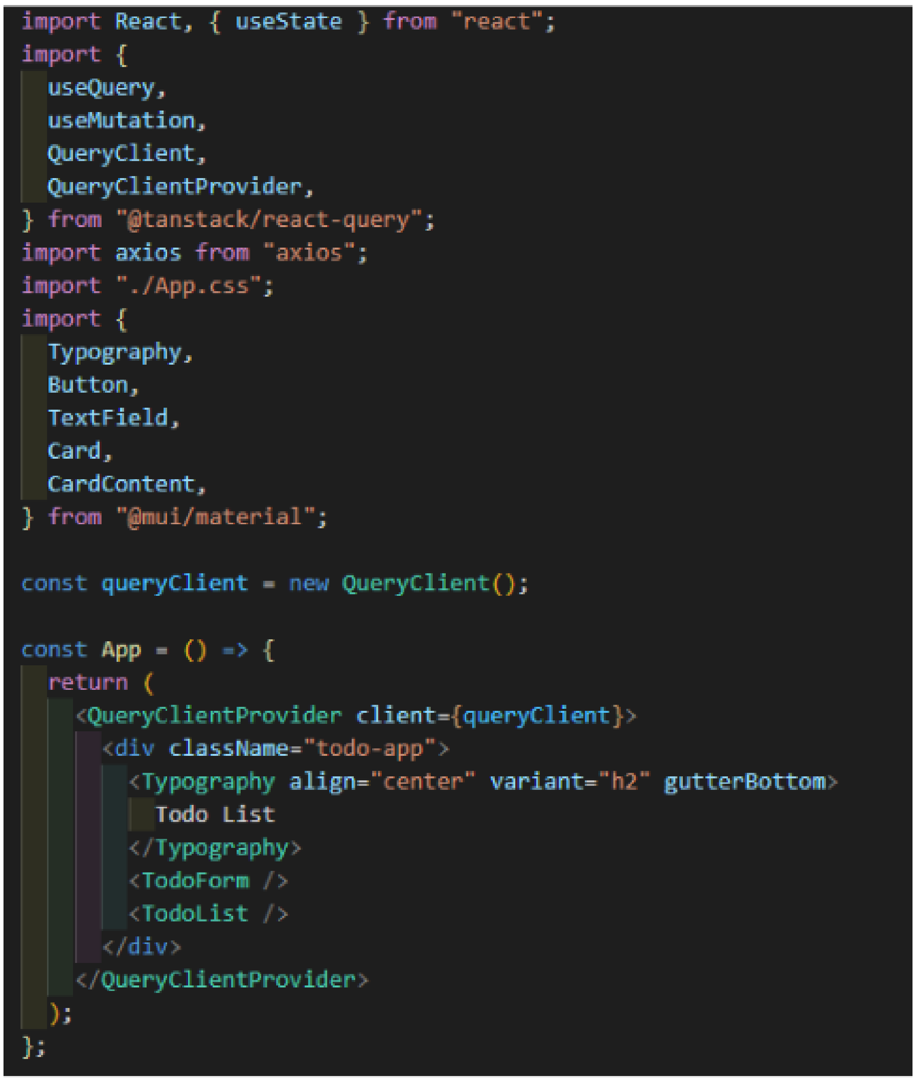
A query is a declarative dependency on an asynchronous source of data that is tied to a unique key.[19] A query can be used with any Promise-based method (including GET and POST methods) to fetch data from a server.[19] To subscribe to a query, the `useQuery` hook is called with at least a unique key referring to the query, and a function that returns a promise that resolves the data or throws an error. The hook then returns an object providing all the information needed to perform any logic implementations. For now, it is sufficient to focus on some of the more important properties:

- `data` contains the available data if the query is successful.
- `error` contains the error if the query is unsuccessful.
- `isLoading` returns true if the query has no data yet.
- `isError` returns true if the query encountered an error.
- `isSuccess` returns true if the query is successful and there is data.

If the application needs to modify data on the server, mutations are used instead. Unlike queries, mutations are typically used to create, update, and delete data or perform server side-effects.[20] For this purpose, the `useMutation` hook is created. This hook requires an asynchronous function to update data (commonly between POST, PUT OR DELETE requests) and returns states similar to the `useQuery` hook, as well as a `mutate` function that can be called to trigger the mutation.[21] Besides the asynchronous function, other optional functions that will only be fired when certain events happen can be passed into the hook as well, such as `onSuccess`, `onError`, `onSettled`, and `onMutation`. These helpers allow easy control of side-effects during the mutation lifecycle, and they become convenient when invalidating and re-fetching queries after mutations.

After modifying the data, it is not very practical to wait for queries to become stale before fetching them again, which is why it is recommended to invalidate the queries right after the mutation successfully happened. This can be achieved by calling the `invalidateQueries` method inside the `onSuccess` helper function of the `useMutation` hook. `invalidateQueries` is a method provided by the `QueryClient` – an object used to interact with caches. The method can be used to invalidate and re-fetch single or multiple queries in the cache based on their query keys or any other functionally accessible property/state of the query.[22] By default, all matching queries are immediately marked as invalid and active queries are re-fetched in the background.[22]

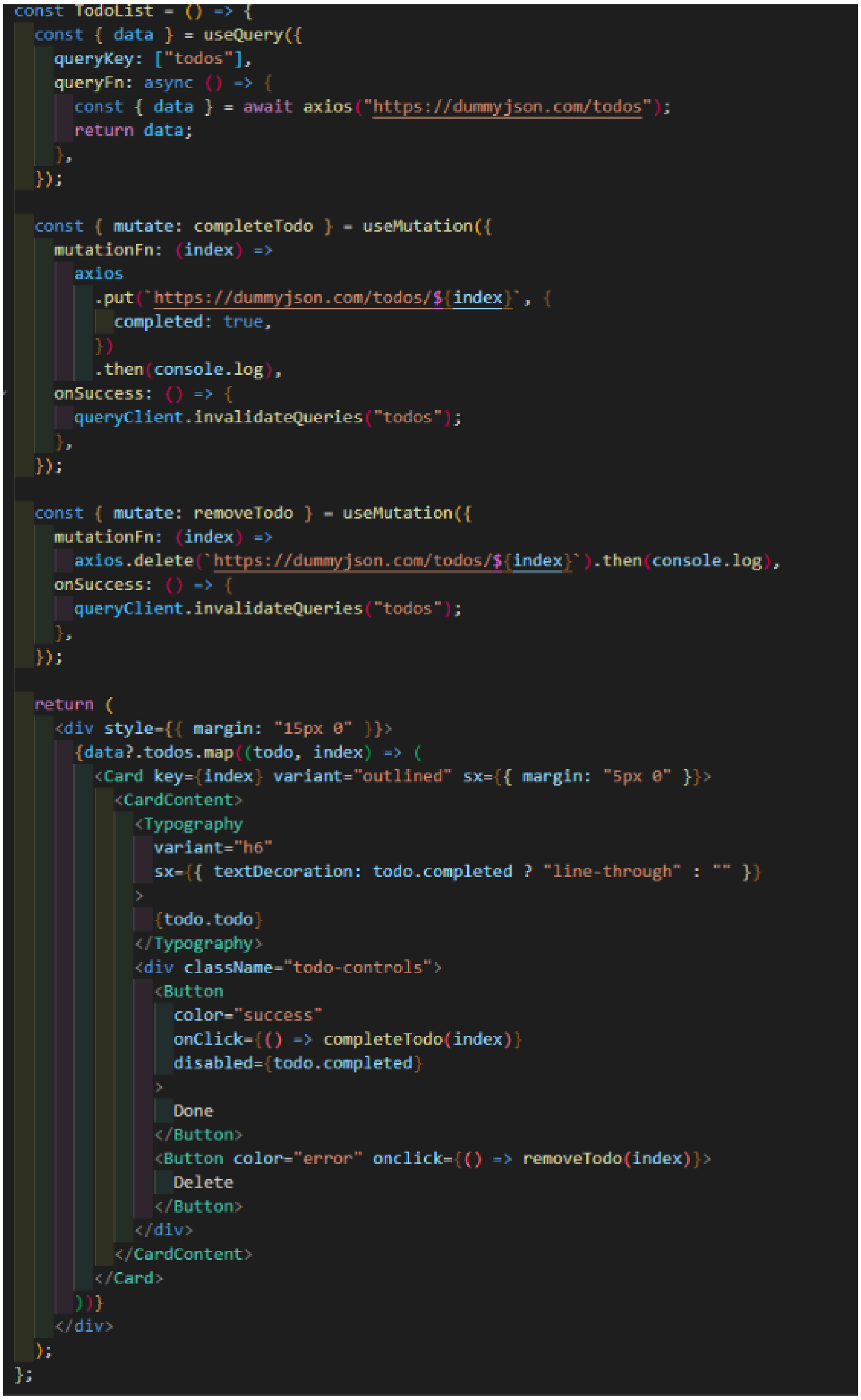
Now that the three most basic concepts of React Query have been explained, the necessary libraries are installed by running `npm install @tanstack/react-query axios` in the terminal. Note that Axios was also installed to communicate with the back-end. Then, a `QueryClient` is initiated and provided to the application by using `QueryClientProvider`. The `QueryClientProvider` component will act as a connector and allow the client to be used anywhere within the application. Figure 19 shows the updated imports and `App` component.



```
import React, { useState } from "react";
import {
  useQuery,
  useMutation,
  QueryClient,
  QueryClientProvider,
} from "@tanstack/react-query";
import axios from "axios";
import "./App.css";
import {
  Typography,
  Button,
  TextField,
  Card,
  CardContent,
} from "@mui/material";

const queryClient = new QueryClient();

const App = () => {
  return (
    <QueryClientProvider client={queryClient}>
      <div className="todo-app">
        <Typography align="center" variant="h2" gutterBottom>
          Todo List
        </Typography>
        <TodoForm />
        <TodoList />
      </div>
    </QueryClientProvider>
  );
};
```

Figure 19. Imports and App component with React Query

As shown in the figure, all the event handler functions and the state used to handle the to-do list have been removed. The App component is now as simple as a container component without any logic written here. That is because the to-do list data will be stored on a back-end server instead of a global state, and React Query together with Axios provides the ability to perform any actions on remote data with ease. All of the actions for fetching and handling to-do list items will be implemented inside the TodoForm and TodoList components. Let us first take a look at the TodoList component in Figure 20.

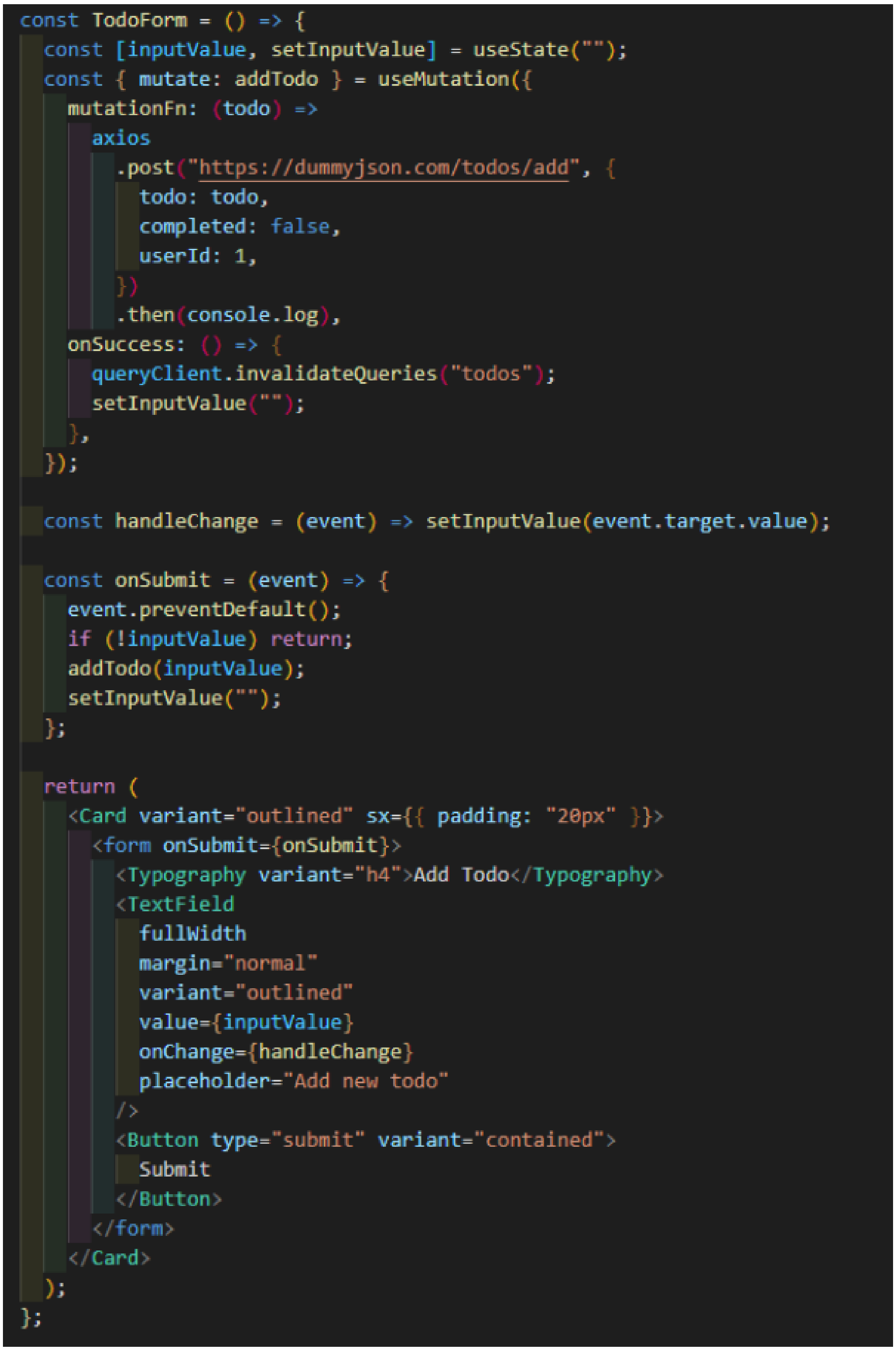


```
const TodoList = () => {
  const { data } = useQuery({
    queryKey: ["todos"],
    queryFn: async () => {
      const { data } = await axios("https://dummyjson.com/todos");
      return data;
    },
  });

  const { mutate: completeTodo } = useMutation({
    mutationFn: (index) =>
      axios
        .put(`https://dummyjson.com/todos/${index}`, {
          completed: true,
        })
        .then(console.log),
    onSuccess: () => {
      queryClient.invalidateQueries("todos");
    },
  });

  const { mutate: removeTodo } = useMutation({
    mutationFn: (index) =>
      axios.delete(`https://dummyjson.com/todos/${index}`).then(console.log),
    onSuccess: () => {
      queryClient.invalidateQueries("todos");
    },
  });

  return (
    <div style={{ margin: "15px 0" }}>
      {data?.todos.map((todo, index) => (
        <Card key={index} variant="outlined" sx={{ margin: "5px 0" }}>
          <CardContent>
            <Typography
              variant="h6"
              sx={{ textDecoration: todo.completed ? "line-through" : "" }}
            >
              {todo.todo}
            </Typography>
            <div className="todo-controls">
              <Button
                color="success"
                onClick={() => completeTodo(index)}
                disabled={todo.completed}
              >
                Done
              </Button>
              <Button color="error" onclick={() => removeTodo(index)}>
                Delete
              </Button>
            </div>
          </CardContent>
        </Card>
      ))}
    </div>
  );
};
```

Figure 20. TodoList rewrote with React Query

The component is slightly longer than its previous implementation because all the event handlers have been moved here. First, the to-do list data is fetched using the `useQuery` hook. As seen from the figure, an array of unique query keys and an asynchronous query function used to fetch the data are passed into the hook. With `useQuery`, no additional `useState` or `useEffect` hook needs to be defined, reducing the number of hooks required to operate with remote data. The returned data is then destructured and rendered to the screen with no visible changes. Next, two `useMutation` hooks are called: one for completing the to-do item, and one for removing the to-do item; hence the changed names of the `mutate` functions. To complete a to-do, a PUT request is called to update the `completed` property of the item matching the specified index. To remove a to-do, a DELETE request is called instead, and the item with the specified index will be deleted. Notice that in both of these mutation calls, an `onSuccess` function is given, invalidating the queries after successfully modifying the data. This is needed for React Query to know and re-fetch the to-do list, which sequentially will re-render the UI of the web application.

The final adjustment needed is to change the way a user adds a new item to the to-do list. This can be done by using the `mutate` function of the `useMutation` hook to add new data to the API through a POST request. An `onSuccess` function will also be defined to invalidate queries and clear the input. As in the `TodoList`'s mutation functions, this means after data creation (POST request) is successful, the query will be invalidated to notify React Query to re-fetch data from the API. Figure 21 shows how to implement the new `addTodo` event handler into the `TodoForm` component.



```
const TodoForm = () => {
  const [inputValue, setInputValue] = useState("");
  const { mutate: addTodo } = useMutation({
    mutationFn: (todo) =>
      axios
        .post("https://dummyjson.com/todos/add", {
          todo: todo,
          completed: false,
          userId: 1,
        })
        .then(console.log),
    onSuccess: () => {
      queryClient.invalidateQueries("todos");
      setInputValue("");
    },
  });

  const handleChange = (event) => setInputValue(event.target.value);

  const onSubmit = (event) => {
    event.preventDefault();
    if (!inputValue) return;
    addTodo(inputValue);
    setInputValue("");
  };

  return (
    <Card variant="outlined" sx={{ padding: "20px" }}>
      <form onSubmit={onSubmit}>
        <Typography variant="h4">Add Todo</Typography>
        <TextField
          fullWidth
          margin="normal"
          variant="outlined"
          value={inputValue}
          onChange={handleChange}
          placeholder="Add new todo"
        />
        <Button type="submit" variant="contained">
          Submit
        </Button>
      </form>
    </Card>
  );
};
```

Figure 21. TodoForm rewrote with React Query

Because the API used here is for demonstration purposes, no real POST, PUT, or DELETE requests will be dispatched to the server. Rather, it will only simulate those requests and return the necessary data upon being called. This is why, to see the results, a `console.log` statement is needed. Even though an actual back-end server was not used, the important thing to consider is that React Query is a great library for managing remote data requests. It eases the process of making server calls for fetching data, caching them, and invalidating them when needed. It also removes the need for additional hooks when working with remote data, replacing them with React Query's own logic. Without a doubt, React Query will definitely help keep the application maintainable, responsive, and fast when working with remote data.

5.3. URL State

5.3.1. React Router

When building a single-page application, navigating when to show a specific resource based on the current URL state of the web page is an important aspect. With such an essential element not supported natively by React, the community has provided many third-party libraries to make handling routing easier. Among all the options, React Location and React Router have become two of the most popular choices, though React Router is still preferred by many developers because of its extensive support by the community.

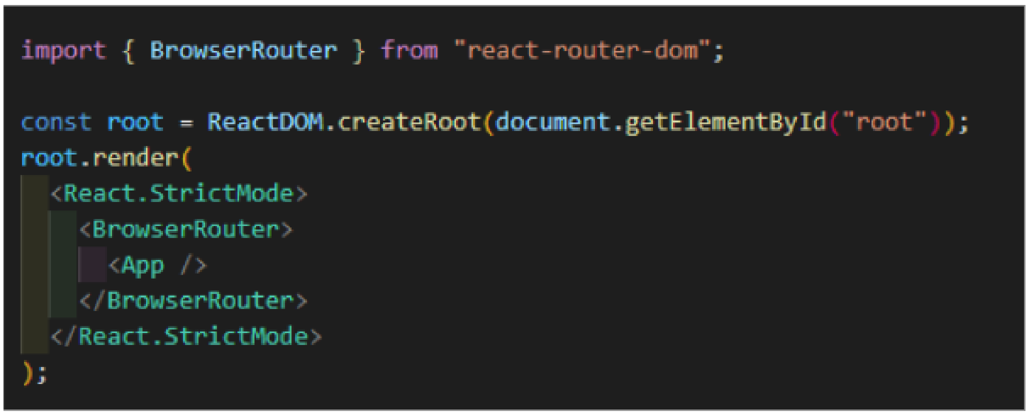
One of React Router's greatest successes is in its client-side routing principle. This means that whenever the user clicks a link in the application, it allows fetching and rendering new UI elements directly instead of re-starting the whole process of requesting documents from the server, evaluating CSS and JavaScript, and rendering HTML contents; resulting in faster and more dynamic user experience.[23] In addition, the library is also famous for its other features, which include:

- Nested routing, allowing developers to combine segments of the URL to component hierarchy and data.[23] In other words, nested routes can be created by defining one route as the child of another.

- Dynamic routing, allowing segments of the URL to be dynamic placeholders that are then parsed and provided to other components. Dynamic segments are defined with the colon punctuation mark, “:”, followed by the segment name.[23] For example, a dynamic route can be "projects/:projectId", with “:projectId” as the dynamic segment.
- Ranked route matching, allowing React Router to rank the routes and pick the most specific match.[23]

Other than these top four features, there are still many great components, utility hooks, and helper functions provided by React Router. The example application will go through some of the most basic but commonly used components: `BrowserRouter`, `Routes`, `Route`, and `Link`.

To illustrate how React Router works, the package is installed in the terminal with `npm install react-router-dom`. After navigating to “index.js” file and the `BrowserRouter` component is imported from the newly installed package, then it is wrapped around the `App` component in the `render` function. Figure 22 demonstrates all of the above.

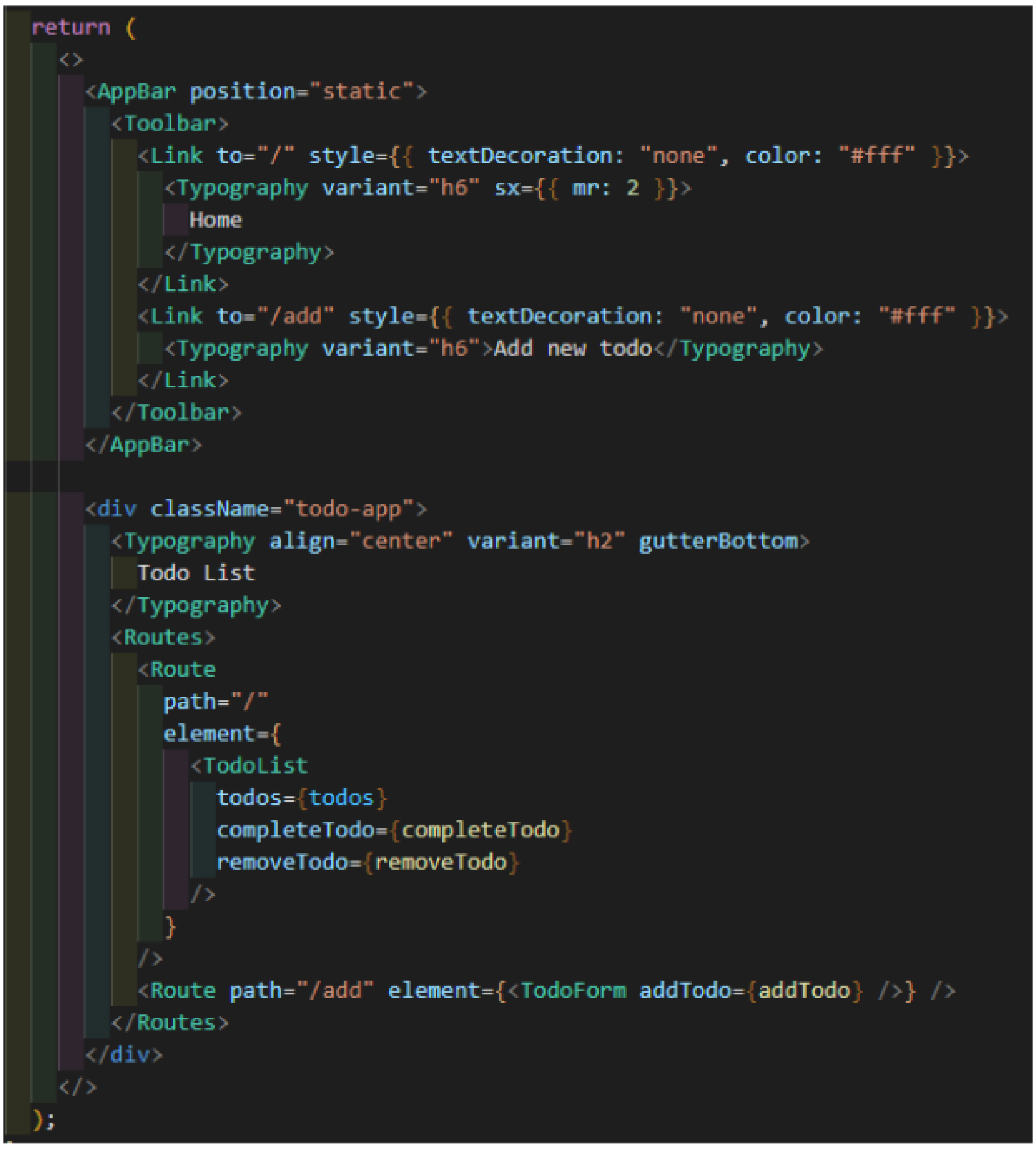


```
import { BrowserRouter } from "react-router-dom";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <React.StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>
);
```

Figure 22. BrowserRouter imported and used in index.js

A `BrowserRouter` is a parent component that stores the current location in the address bar of the browser using clean URLs and navigates using the built-in history stack of the browser.[24] In other words, `BrowserRouter` helps the library keep track of the navigation history of the application, therefore making it available whenever needed. For other environments that are not the browser, React Router provides other components like `HashRouter`, `MemoryRouter`, `NativeRouter`, or `StaticRouter`.

After enabling React Router through its wrapper component, it is necessary to move back to the “App.js” file to create routes and navigation links for them. `Routes`, `Route`, and `Link` are imported from the package to the file, then the changes shown in Figure 23 are made to the `return` function of the `App` component. Note that there are also two new Material UI components imported, `AppBar` and `Toolbar`.



```
return (
  <>
    <AppBar position="static">
      <Toolbar>
        <Link to="/" style={{ textDecoration: "none", color: "#fff" }}>
          <Typography variant="h6" sx={{ mr: 2 }}>
            Home
          </Typography>
        </Link>
        <Link to="/add" style={{ textDecoration: "none", color: "#fff" }}>
          <Typography variant="h6">Add new todo</Typography>
        </Link>
      </Toolbar>
    </AppBar>

    <div className="todo-app">
      <Typography align="center" variant="h2" gutterBottom>
        Todo List
      </Typography>
      <Routes>
        <Route
          path="/"
          element={
            <TodoList
              todos={todos}
              completeTodo={completeTodo}
              removeTodo={removeTodo}
            />
          }
        />
        <Route path="/add" element={<TodoForm addTodo={addTodo} />} />
      </Routes>
    </div>
  </>
);
```

Figure 23. New return function of the App component

Let us first go through the `Route` component – the most important part of React Router. `Route` couples URL segments to components, data loading, and data mutations.[25] Put simply, it allows mapping to different React components based on the application's location. For that to happen, pass a `path` and an `element` prop into the `Route` component, with the `path` being the pattern to match against the URL and the `element` being the component to render when the route matches. There will be multiple `Route` components in an application, and when the current web page location matches a `path`, it will render the corresponding `element`.

The problem with `Route` is that it is possible for multiple routes to match against a single URL, which should be prevented most of the time. Fortunately, there is another component – `Routes` – that will help React Router render only the route that best matches the specified path. `Routes` is a wrapper for the `Route` components whose job is to look through all its child routes to find the best match and renders that branch of the UI.[26] Though it is not clearly shown in this simple application, once more complex routings are added, `Routes` will help prevent possible issues by enabling intelligence rendering.

The final additional component of the application is the `Link` component. `Link` is an element that allows the user navigate to another page by clicking on it.[27] In the DOM, a `Link` renders an accessible HTML anchor element with a real `href` (hypertext reference) attribute that points to the resource it is linking to.[27] To communicate which path to take when the user clicks the link, pass a `to` prop that will be holding a URL path to the component, as shown in figure 23.

The components introduced above are expected to appear in any React application that uses React Router. React Router allows keeping the root components clear, maintainable, and readable. Despite not establishing all of its benefits that help to navigate an application easier, it is undeniable that React Router makes displaying multiple views in a single-page application effortless.

5.4. UI State

5.4.1. Local component state and props

The simplest way to manage the UI state is to use local component states and props with the `useState` hook, as shown in the initial version of the example application. In this version, a to-do list, as well as all the related event handlers, are stored in the wrapping `App` component. They are then passed down as props in the component tree to the elements that use them. To revise, states are used to store data, whereas props are used to pass data. A React component uses states to control data, then communicates with its child components through props. For beginners, the two terms can be easily mistaken as they not only seem similar to each other, but are also closely connected. One easy way to distinguish them is that props cannot be changed by the component that receives them, while states are used to contain data that can be modified over time. Though it seems hard to remember their functions and differences, the clarity between states and props will improve as one gets more experienced with React.

The approach of using states and props together might seem the easiest and most straightforward approach. However, in larger applications, the process of passing down props will happen repeatedly until the props reach the component where the data is needed. This is known as prop drilling, and it will only get worse as the application grows. The most basic solution to this is to identify where the states should be defined. In other words, it means identifying which components should own the states. They should not be owned by a component that is too high up in the component tree, yet not by one that is on the same level as all the required components either. Even after applying this rule, there will be times when the application has become too complex that managing the states in various components turns into a daunting task. Fortunately, to address this issue, the React development community has created several global state management libraries. These libraries ensure data is managed globally and can be accessed from anywhere in the application, therefore help avoid prop drilling. Each of them provides its own syntax, possible benefits, and tradeoffs when used in certain contexts. Learning how to obtain the most out of the tools is an essential yet interesting part of being a web developer.

5.4.2. Redux

First coming out in 2015, Redux is still widely used and extremely popular today. Redux is a predictable state container for JavaScript apps.[28] It helps write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.[28] In simple terms, Redux helps manage states in the application. It acts as a centralized "store", containing states needed across the application in one single object and only allows them to be updated using events called "actions" while ensuring that all state updates follow a certain "reducer" pattern. This means the three main building blocks of Redux are: the store, reducers, and actions.

Before going into the main concepts of Redux, there are a few patterns to be remembered when building an application with Redux. The first pattern that Redux adopts is called "single source of truth", which means there can be only one centralized storage holding the single state object – the application state – for the whole application. The second one is called "immutability", meaning that the state should not be changed directly but instead through actions and reducers. How this works will be explained later on. One final important rule to understand is that despite having Redux as a global store for the state, other components can have their own internal state as well. Data that is used only in a single component should not be stored in the application state.

The "store" is the center of every Redux application, storing its state globally. Although many have mistaken it as a class, the store is actually a JavaScript object that also holds other functions and objects besides the global state. The concept of the store is very easy to understand, yet it is the most important. Another thing to note is that even though theoretically, it is possible to create multiple stores in an application, it is against the "single source of truth" pattern that Redux follows.

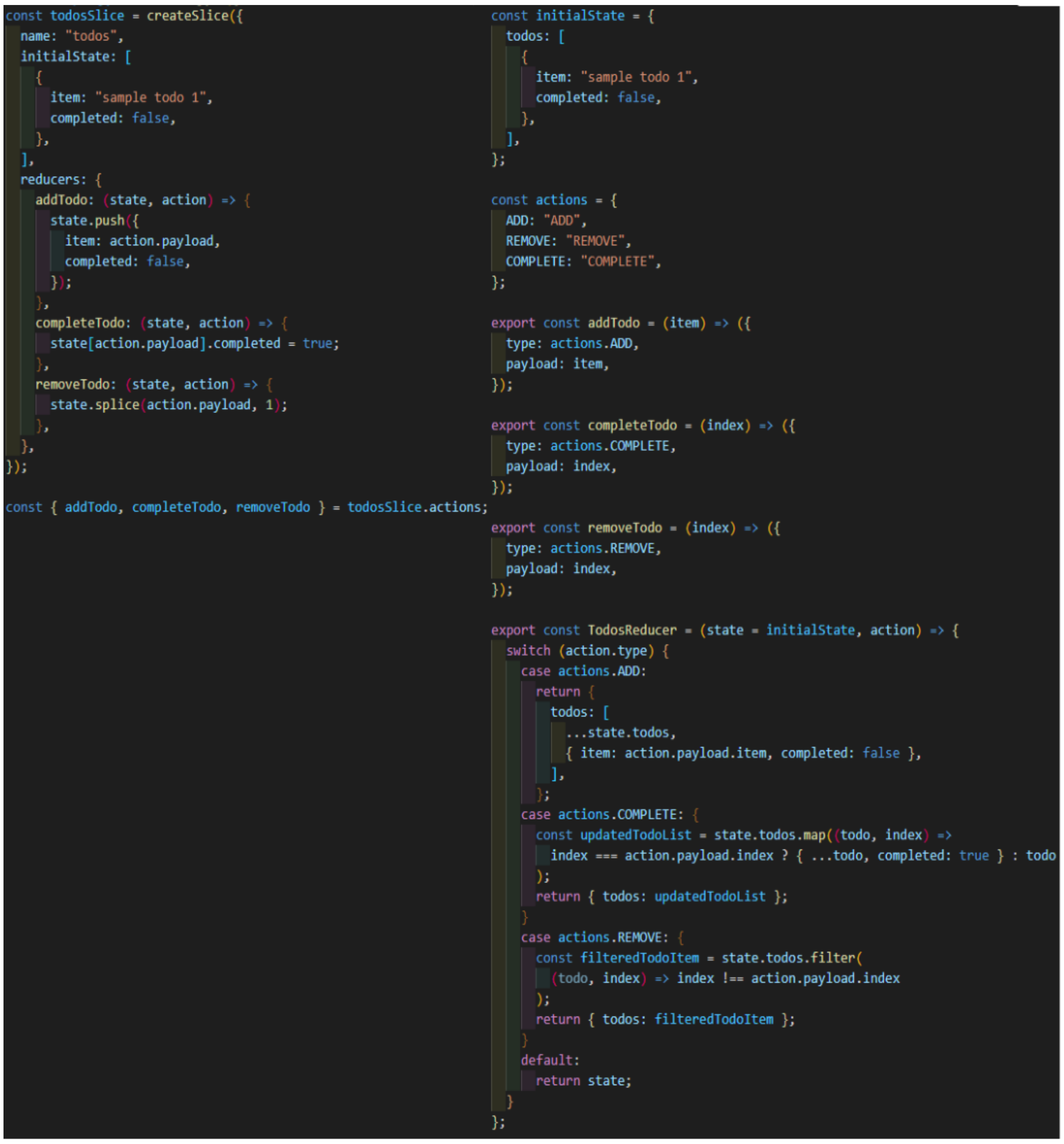
“Actions” are JavaScript objects that plainly describe what event happened, but not how to handle the event. They have a required `type` field that communicates what kind of event happened, therefore should be given a descriptive name. To update the state of the application based on an action, a `dispatch` method has to be called. In other words, dispatching an action is the same as triggering an event in the application. What happens afterward is handled by reducers, which act like event listeners.

If actions describe what happened, then “reducers” decide how the application state changes based on the event. A “reducer” is a pure function that receives the current `state` and an `action` object, then return a new state after calculations. This means that whenever an action is dispatched to the store, the store then passes that action to the reducer, which will return the updated state.

The flow of data through a Redux application might sound complicated at first, but it is actually fairly straightforward. To understand how it really works internally, let us think about an event where the user clicks a button. When the click happens, an action will be dispatched to the store with the `dispatch` method in response. The store then runs the reducer function to calculate a new state, and finally, the new state will be rendered to the UI.

Now that the basic concepts of Redux have been explained, a more in-depth application regarding this can be practiced. Start by running `npm install react-redux @reduxjs/toolkit` in the terminal to install the necessary packages. The application will be using Redux Toolkit, a package created by the Redux developers themselves, to eliminate repetitive code and prevent common mistakes, such as complicated store configuration, too many dependencies, and too much boilerplate code.[29] After installation, `configureStore` and `createSlice` APIs are imported from Redux Toolkit; as well as `useSelector`, `useDispatch`, and `Provider` from React Redux. Each of these APIs and hooks will be explained as they are implemented into the application so that in the end, the whole process will be understood thoroughly.

The first thing that needs to be done is to create a slice reducer for the to-do list. This can be done with the `createSlice` API. `createSlice` accepts an object of reducer functions, a slice name, and an initial state value, and automatically generates corresponding action creator functions and action type strings internally.[30] This means that the code written will be drastically simpler compared to legacy hand-written Redux. The comparison between the two approaches is shown in Figure 24. Note that the approach on the left is what will be used in the example application.



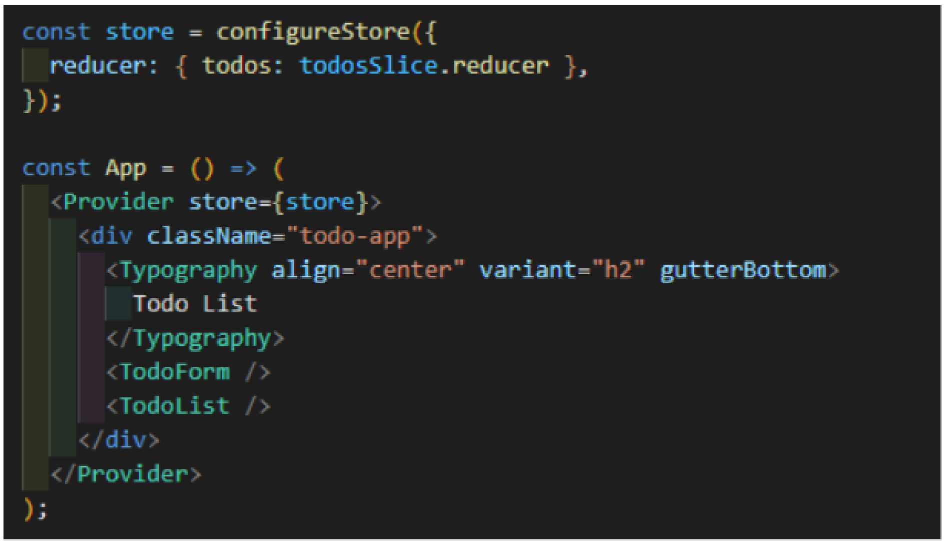
```
const todosSlice = createSlice({
  name: "todos",
  initialState: [
    {
      item: "sample todo 1",
      completed: false,
    },
  ],
  reducers: {
    addTodo: (state, action) => {
      state.push({
        item: action.payload,
        completed: false,
      });
    },
    completeTodo: (state, action) => {
      state[action.payload].completed = true;
    },
    removeTodo: (state, action) => {
      state.splice(action.payload, 1);
    },
  },
});

const { addTodo, completeTodo, removeTodo } = todosSlice.actions;
```

```
const initialState = {
  todos: [
    {
      item: "sample todo 1",
      completed: false,
    },
  ],
};

const actions = {
  ADD: "ADD",
  REMOVE: "REMOVE",
  COMPLETE: "COMPLETE",
};

export const addTodo = (item) => ({
  type: actions.ADD,
  payload: item,
});

export const completeTodo = (index) => ({
  type: actions.COMPLETE,
  payload: index,
});

export const removeTodo = (index) => ({
  type: actions.REMOVE,
  payload: index,
});

export const TodosReducer = (state = initialState, action) => {
  switch (action.type) {
    case actions.ADD:
      return {
        todos: [
          ...state.todos,
          { item: action.payload.item, completed: false },
        ],
      };
    case actions.COMPLETE: {
      const updatedTodoList = state.todos.map((todo, index) =>
        index === action.payload.index ? { ...todo, completed: true } : todo
      );
      return { todos: updatedTodoList };
    }
    case actions.REMOVE: {
      const filteredTodoItem = state.todos.filter(
        (todo, index) => index !== action.payload.index
      );
      return { todos: filteredTodoItem };
    }
    default:
      return state;
  }
};
```

Figure 24. Comparison on how to create reducer between Redux Toolkit (left) and legacy Redux (right)

In the legacy Redux version, the `addTodo`, `completeTodo`, and `removeTodo` functions are called action creators, and they simply return an action object. After one of these actions is dispatched, Redux will use its action type string to try and find the corresponding handler function in the reducer, which is one of the case statements in the switch. Redux Toolkit's `createSlice` performs this same functionality, but it reduces the number of lines written in half. The API looks at all the handler functions defined in the `reducers` field and for each case automatically generates an action creator that uses the key of the function as its action type string. Notice that now the application state can be modified directly as well, and Redux Toolkit will take care of everything under the hood. Once created, the slice will return an object containing several fields: `name`, `reducer`, `actions`, `caseReducers`, and `getInitialState`; with the functions previously defined in the `reducers` argument returned through the `actions` field.

To continue, a "store" needs to be created and connected to the application. This can be done through the `configureStore` and `Provider` APIs, as implemented in Figure 25.



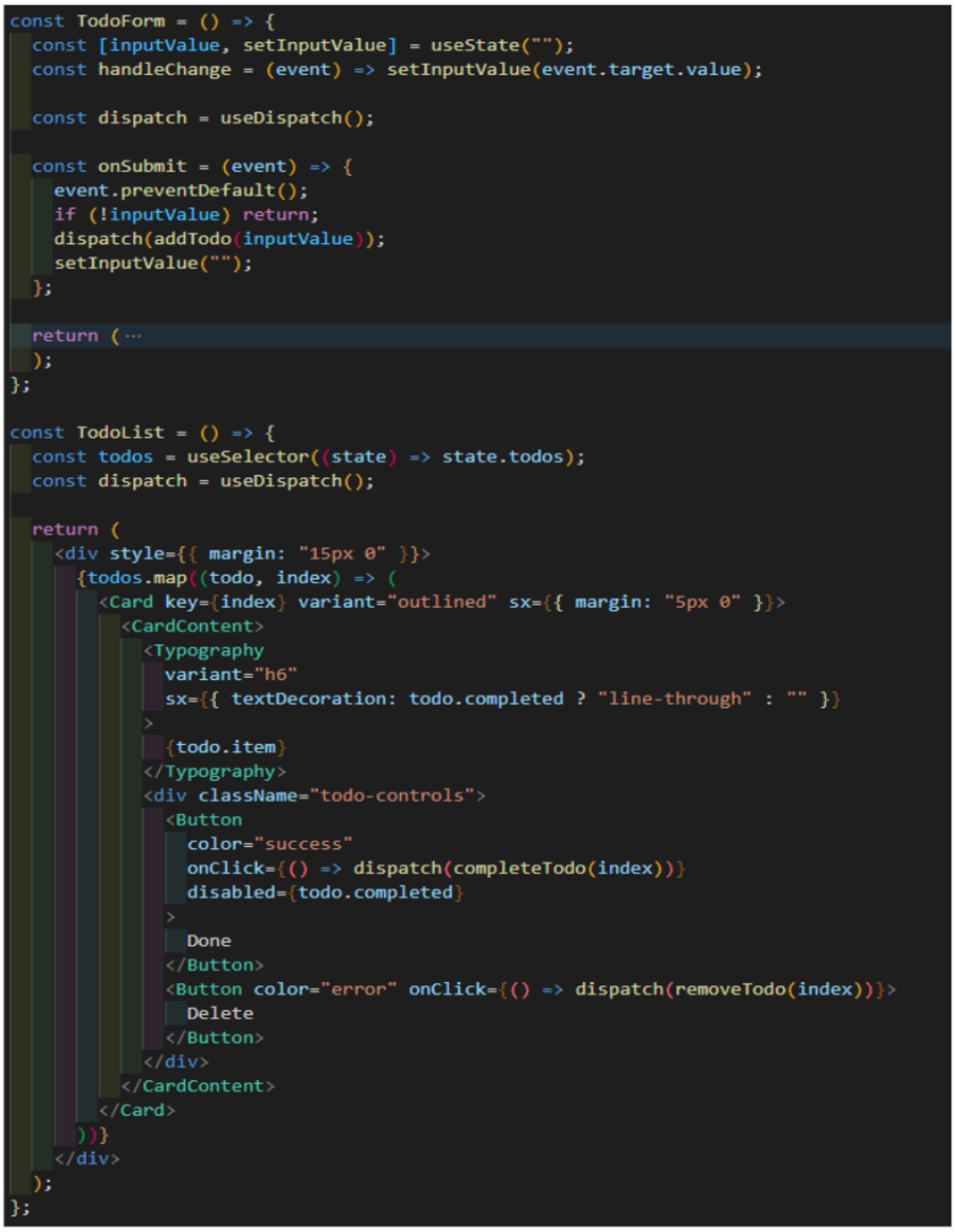
```
const store = configureStore({
  reducer: { todos: todosSlice.reducer },
});

const App = () => (
  <Provider store={store}>
    <div className="todo-app">
      <Typography align="center" variant="h2" gutterBottom>
        Todo List
      </Typography>
      <TodoForm />
      <TodoList />
    </div>
  </Provider>
);
```

Figure 25. Creating a Redux store with configureStore() and Provider APIs

Through a single function call, `configureStore` calls all the necessary functions within itself and provides a well-configured Redux store by default. It accepts a single configuration object with one required and four optional parameters. The required field, named `reducer`, can be seen in Figure 25. It accepts two types of declaration: if it is a single function, it will be used as a root reducer for the store; while if it is an object of slice reducers, these slice reducers will be combined to create a root reducer.[30] The reducer passed in can be the returned `reducer` function from the `createSlice` API.

The `Provider` API presents a simple wrapper component that makes the store available to any nested components that need to access the Redux store.[31] Usually, the `Provider` component will be rendered at the top level, with the entire application component tree enclosed inside of it. For any kind of application, it is enough to pass a `store` prop to the `Provider`.

Now that the actions are created and the store is connected to the application, the final step is to make use of the `useSelector` and `useDispatch` hooks to read the state and dispatch the action created in the slice. The `useSelector` hook allows extracting data from the Redux store state using a selector function.[32] A selector function should be a pure function that accepts the Redux store state and returns necessary data based on that state. The `useDispatch` hook, on the other hand, returns a reference to the `dispatch` function from the Redux store which can be used to dispatch functions as needed.[32] The implementation of these hooks can be seen in Figure 26.



```
const TodoForm = () => {
  const [inputValue, setInputValue] = useState("");
  const handleChange = (event) => setInputValue(event.target.value);

  const dispatch = useDispatch();

  const onSubmit = (event) => {
    event.preventDefault();
    if (!inputValue) return;
    dispatch(addTodo(inputValue));
    setInputValue("");
  };

  return (…
  );
};

const TodoList = () => {
  const todos = useSelector((state) => state.todos);
  const dispatch = useDispatch();

  return (
    <div style={{ margin: "15px 0" }}>
      {todos.map((todo, index) => (
        <Card key={index} variant="outlined" sx={{ margin: "5px 0" }}>
          <CardContent>
            <Typography
              variant="h6"
              sx={{ textDecoration: todo.completed ? "line-through" : "" }}
            >
              {todo.item}
            </Typography>
            <div className="todo-controls">
              <Button
                color="success"
                onClick={() => dispatch(completeTodo(index))}
                disabled={todo.completed}
              >
                Done
              </Button>
              <Button color="error" onClick={() => dispatch(removeTodo(index))}>
                Delete
              </Button>
            </div>
          </CardContent>
        </Card>
      ))}
    </div>
  );
};
```

Figure 26. useSelector() and useDispatch() implemented in TodoForm and TodoList component

The `return` function of the `TodoForm` component has been hidden for better visibility because nothing inside has changed. By using hooks, Redux makes sure that the UI will be kept up-to-date by re-rendering whenever a state changes or an action is dispatched. As can be seen, both the state and event handlers that used to be defined in the `App` component have been moved to Redux, reducing the complexity of this wrapper component. This is one of the advantages when using Redux – it helps make managing global states and application logic more predictable and maintainable.

Though Redux acts as an organizer that ensures an application will work as expected, it has its tradeoffs. Compared to other UI state management libraries, Redux has more concepts to learn, more code to be written, and more rules to follow. This is why it is worth considering before choosing Redux over other tools, despite all the long-term benefits it provides. Redux is known to be most useful in a number of situations, according to its official documentation. The first case is when the application consists of a large number of global states that are needed in many places.[33] Secondly, when the application state frequently updates with complex logic.[33] The last case to consider using Redux is when the application has a medium or large-sized codebase and is worked on by many people.[33] In summary, not all applications need Redux. It takes time to think about the scale of the application being built and decide if Redux is the best tool to solve the possible use cases in the most efficient way.

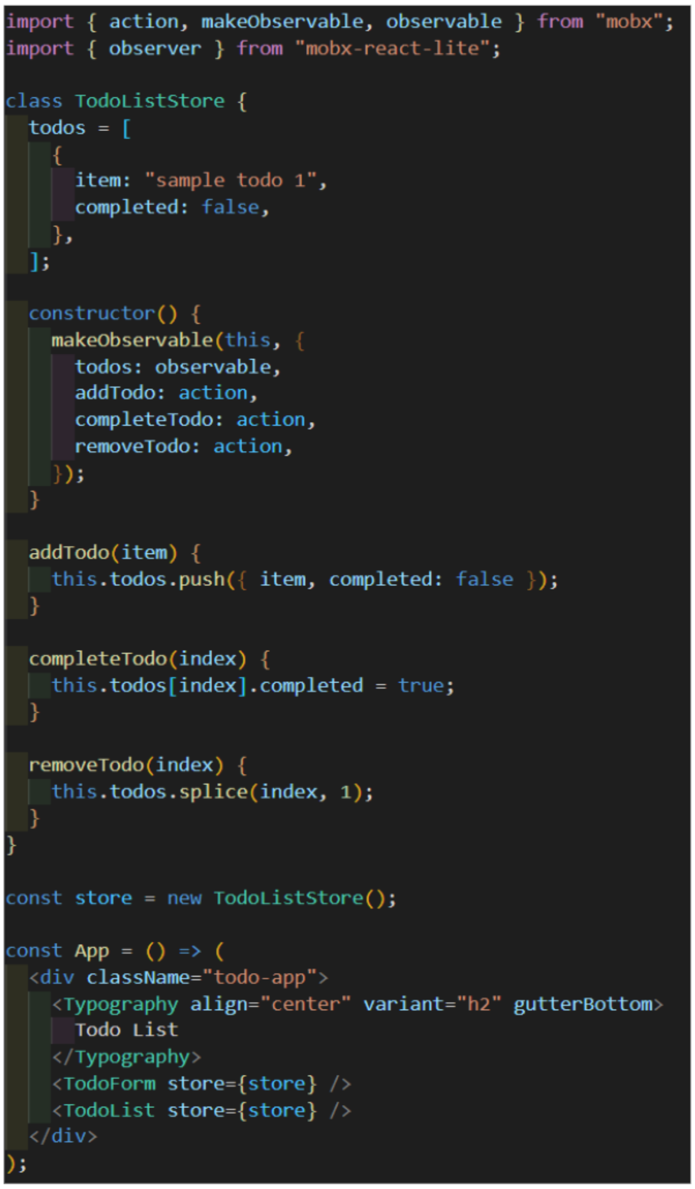
5.4.3. MobX

MobX is a state management tool that highly follows the principles of object-oriented programming, and can be used with any popular front-end framework. It was first introduced in 2016 and is considered a great alternative to Redux. Similar to Redux, it introduces three new concepts: state, actions, and derivations. These concepts within MobX are pretty straightforward:

- "State" is the data that determines how the application works and any properties that get changed over time should be marked as observable so MobX can track them.[34]
- An "action" is any piece of code that changes the state.[34] In other words, they are the event handler functions.
- A "derivation" is anything that can be derived from the state without any further interaction.[34] The two types of derivation are "reaction", which is the side effects that occur automatically when the state changes, and

"computed value", which always returns a value derived from the current state.[34]

These new terms can be confusing without proper demonstration. To better understand how MobX works, `npm install mobx mobx-react-lite` is run in the terminal window to install its packages. Afterward, the `action`, `makeObservable`, and `observable` APIs are imported from the "mobx" package, as well as the `observer` API from the "mobx-react-lite" package. Before explaining how they each contribute to the way MobX works, let us first view the implementation details in Figure 27.



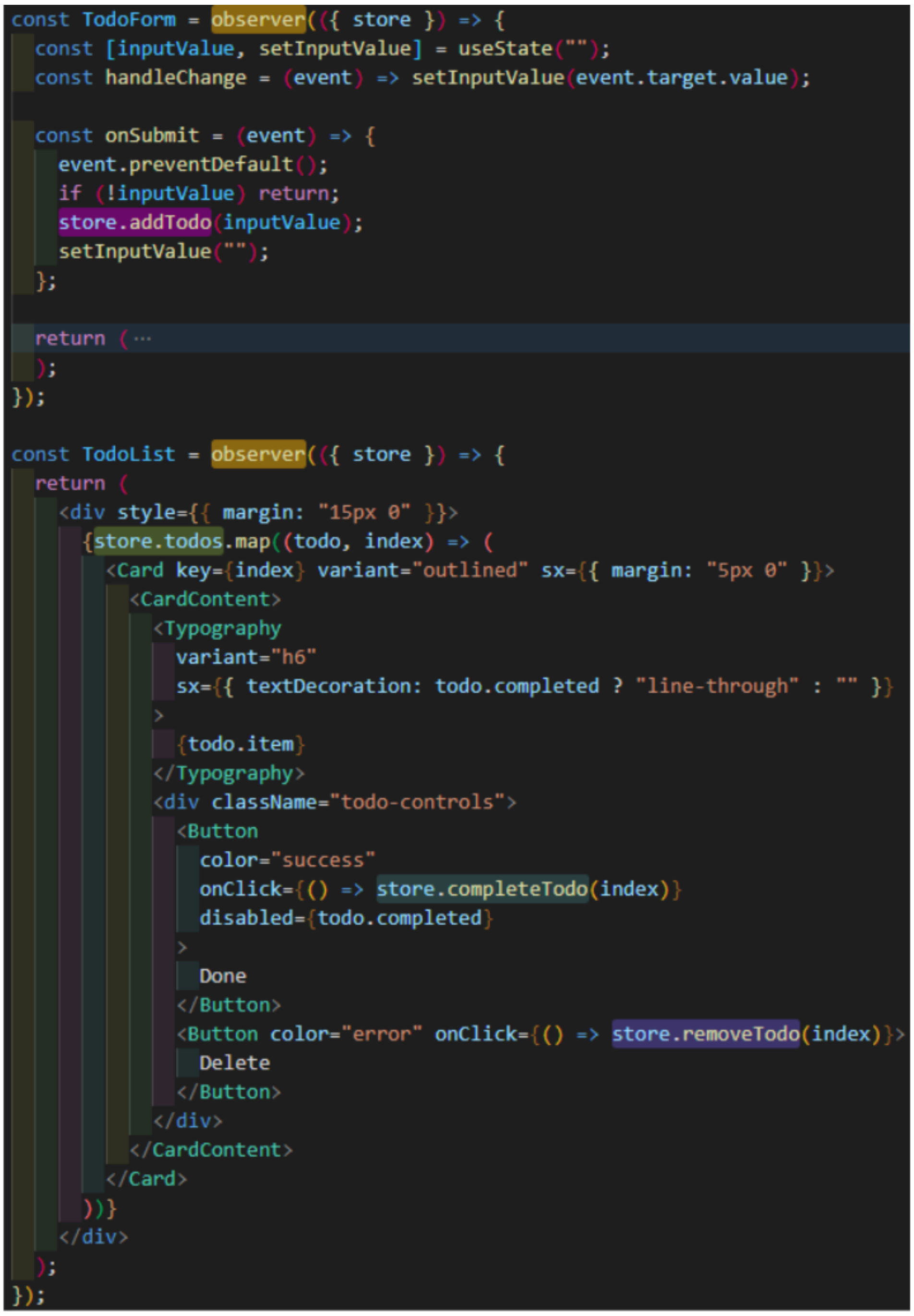
```
import { action, makeObservable, observable } from "mobx";
import { observer } from "mobx-react-lite";

class TodoListStore {
  todos = [
    {
      item: "sample todo 1",
      completed: false,
    },
  ];

  constructor() {
    makeObservable(this, {
      todos: observable,
      addTodo: action,
      completeTodo: action,
      removeTodo: action,
    });
  }

  addTodo(item) {
    this.todos.push({ item, completed: false });
  }

  completeTodo(index) {
    this.todos[index].completed = true;
  }

  removeTodo(index) {
    this.todos.splice(index, 1);
  }
}

const store = new TodoListStore();

const App = () => (
  <div className="todo-app">
    <Typography align="center" variant="h2" gutterBottom>
      Todo List
    </Typography>
    <TodoForm store={store} />
    <TodoList store={store} />
  </div>
);
```

Figure 27. Creating an application store using MobX

The class named `TodoListStore` is a representation of the application store. It contains a `todos` property initialized as an array of one object – an equivalent to the state of the initial version – and all the functions used to handle adding, completing, and removing a to-do. Notice how the "actions" here are fairly similar to those from Redux. In order for MobX to make this class reactive, and not just like any ordinary JavaScript class, the `makeObservable` API was introduced. The function makes any existing properties of a JavaScript object, including class instances, observable. By observing these properties, MobX will refresh and update the UI according to the changes that happened. The use of the API should be started by passing into its function two parameters: one as a reference to a class instance, and one as an object configuration of the class instance methods and fields. There are a few rules regarding the object configuration options:

- `observable` defines a trackable field that stores the state.[35] In other words, any field value of the class that holds an array, an object, or any type of value should be configured as an `observable`.
- `action` marks a method as an action that will modify the state.[35] Usually, event handler functions will be configured as `action`.
- `computed` marks a getter that will derive new facts from the state and cache its output.[35] This means any functions that return a value derived from the current state of the store should be configured as `computed`.

Additionally, reactions – the functions that run when the state changes but do not return a value – should use the `autorun` function to let MobX know what to trigger every time it observes any changes. Once all the necessary setups are made, the store needs to be initialized to be put into action. The store is initialized in the same way as one would with a regular JavaScript class, as shown in Figure 27. After everything is ready, the store is passed as a prop into the desired components. In order for these components to communicate with the store, their implementations are wrapped with the `observer` function from "mobx-react-lite". By wrapping the components with an observer, they will automatically become aware of any changes made in the store, therefore re-rendering the UI accordingly. This is the final step to creating a MobX application, and now both the state and the actions can be used similarly to those of Redux, but without the need for hooks. Figure 28 depicts how this process should be done, with the newly added code highlighted.



```
const TodoForm = observer(({ store }) => {
  const [inputValue, setInputValue] = useState("");
  const handleChange = (event) => setInputValue(event.target.value);

  const onSubmit = (event) => {
    event.preventDefault();
    if (!inputValue) return;
    store.addTodo(inputValue);
    setInputValue("");
  };

  return (⋯
  );
});

const TodoList = observer(({ store }) => {
  return (
    <div style={{ margin: "15px 0" }}>
      {store.todos.map((todo, index) => (
        <Card key={index} variant="outlined" sx={{ margin: "5px 0" }}>
          <CardContent>
            <Typography
              variant="h6"
              sx={{ textDecoration: todo.completed ? "line-through" : "" }}
            >
              {todo.item}
            </Typography>
            <div className="todo-controls">
              <Button
                color="success"
                onClick={() => store.completeTodo(index)}
                disabled={todo.completed}
              >
                Done
              </Button>
              <Button color="error" onClick={() => store.removeTodo(index)}>
                Delete
              </Button>
            </div>
          </CardContent>
        </Card>
      ))}
    </div>
  );
});
```

Figure 28. Using the store's state and actions in TodoForm and TodoList components

Looking at this example to-do list application, one can say that the simplicity of MobX provides a lot of freedom in terms of implementation and usability. MobX allows for the creation of multiple stores as opposed to just one, making it easily scalable when the situation requires another store. Additionally, MobX also requires less boilerplate code compared to other state management libraries, such as legacy Redux. Because there was no Redux Toolkit when MobX and Redux were first introduced, many developers preferred using MobX since Redux requires a significant amount of boilerplate code.

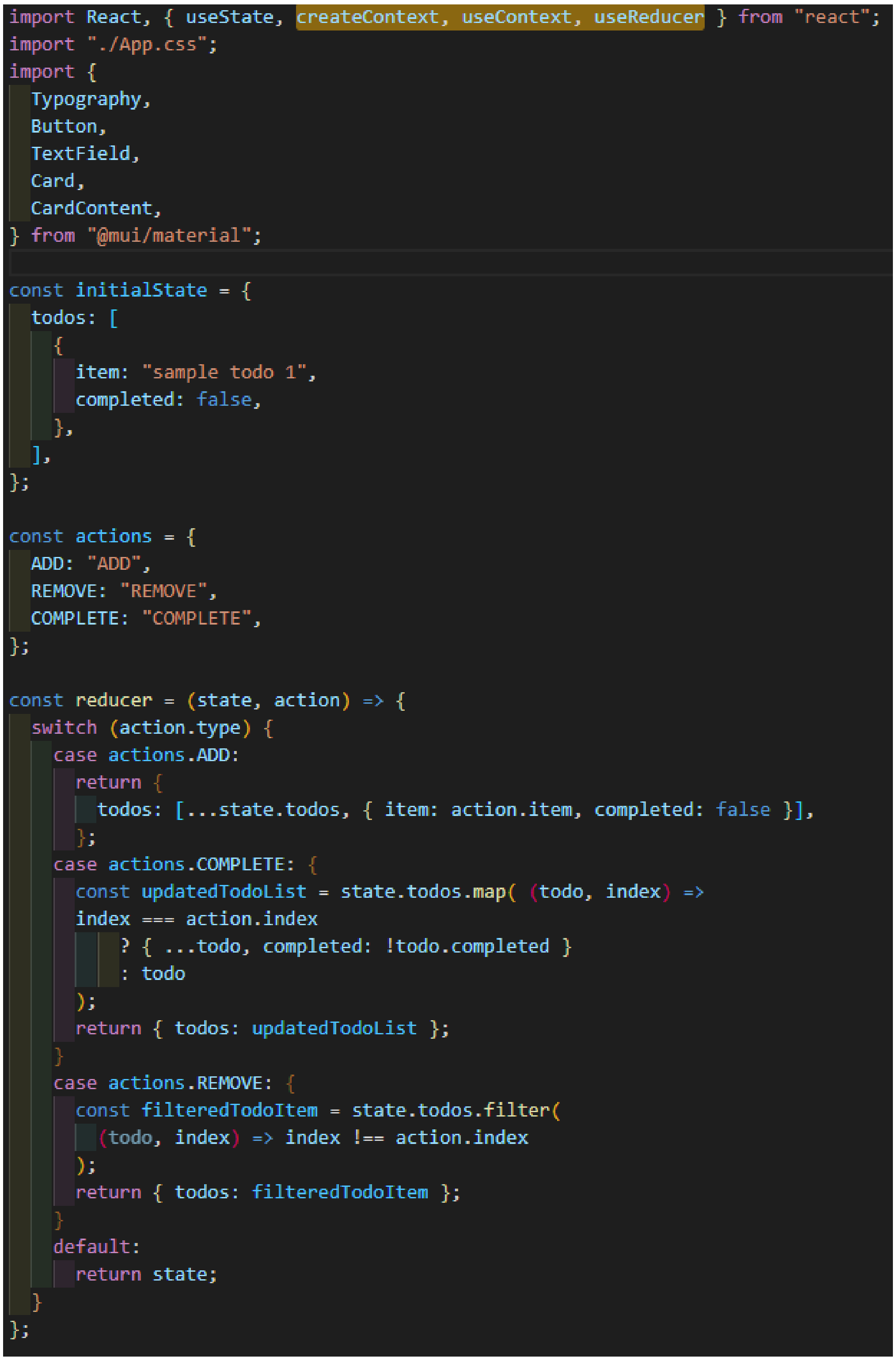
All of the advantages offered by MobX can sound intriguing to consider it the best UI state management tool, but sometimes the lack of proper rules and restrictions can be problematic when trying to scale and maintain the application. Creating many application stores means that there is no single source of truth, therefore the state can be mutated internally without the developers' awareness. Another drawback for MobX is that its states are mutable, meaning they can be easily updated and overwritten. This makes testing and debugging harder because of the unpredictable source of bugs.

As with any other UI state management tool, MobX has its own advantages and disadvantages. When choosing a library, consider if it will affect the application scalability and testability, as well as the purpose for creating the application. In most cases, MobX is chosen for proof-of-concept applications or if the development time is limited, though this does not mean that MobX is not useful in other situations.

5.4.4. Context API with useContext and useReducer

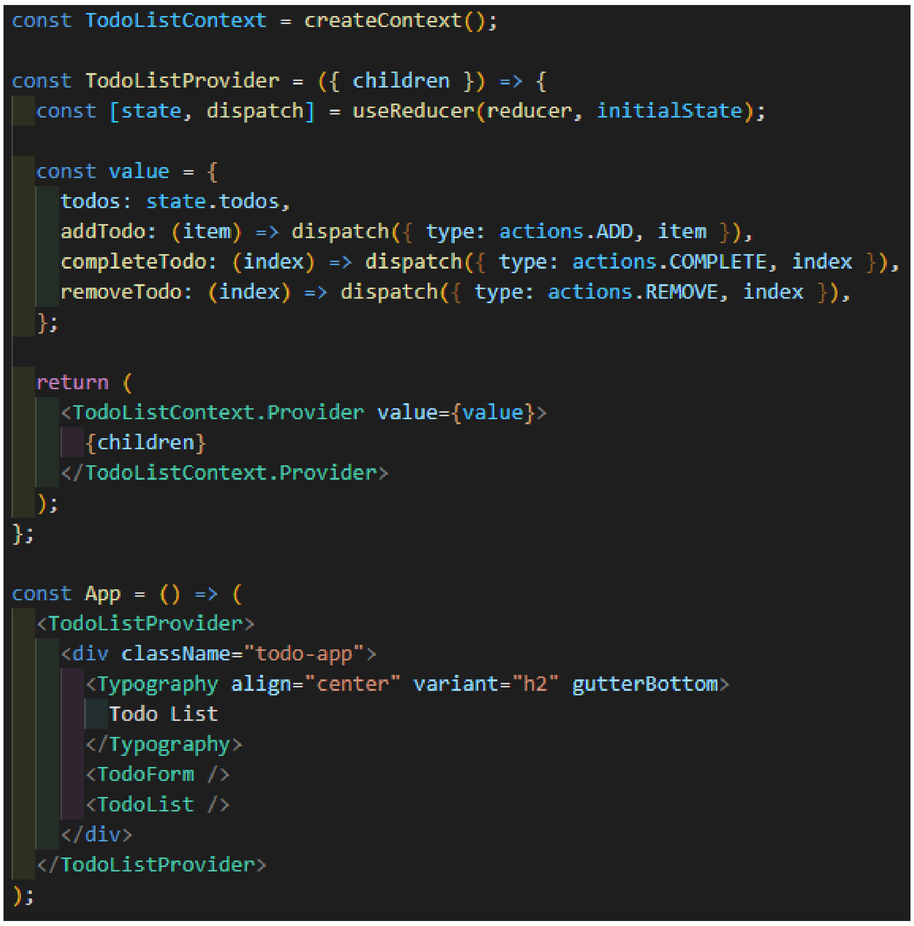
React Context provides a way to pass data through the component tree without having to pass props down manually at every level.[36] This helps avoid prop-drilling when props are required by many child components. Though its purpose is the same as every other state management library, what makes React Context stand out is that it is a native API, which means no external packages or dependencies need to be installed in order to use it. In larger applications where developers would want to minimize their bundle size as much as possible, this might be worth considering as the ideal solution. It is important to mention, however, that the Context API alone is not a UI state management tool, but a type of Dependency Injection. In fact, for the Context API to achieve its state managing functionality, it needs to be used together with two other hooks from React: `useContext` and `useReducer`. To revise, `useContext` is used to read and subscribe to globally available context data from any component in the application, while `useReducer` is an alternative to `useState` which returns a `dispatch` method together with the current state instead of an updater function. Except for the fact that it is more lightweight, the way React Context works is similar to legacy Redux, with the same immutability principles. This means the state, actions, and reducers are initialized highly identical to that of Redux. Let us examine how this state management tool works by coming back to the example application.

To start, `createContext`, `useContext`, and `useReducer` are added in addition to the existing imports from the “react” package. Then, as shown in Figure 29, three constants are defined: one for the initial state of the application, one for actions related to to-do list events, and one for reducers to handle those actions. Similarly, Figure 30 shows the context and the provider being created after implementing the necessary configurations, as well as how to use them in the example application.



```
import React, { useState, createContext, useContext, useReducer } from "react";
import "./App.css";
import {
  Typography,
  Button,
  TextField,
  Card,
  CardContent,
} from "@mui/material";

const initialState = {
  todos: [
    {
      item: "sample todo 1",
      completed: false,
    },
  ],
};

const actions = {
  ADD: "ADD",
  REMOVE: "REMOVE",
  COMPLETE: "COMPLETE",
};

const reducer = (state, action) => {
  switch (action.type) {
    case actions.ADD:
      return {
        todos: [...state.todos, { item: action.item, completed: false }],
      };
    case actions.COMPLETE: {
      const updatedTodoList = state.todos.map( (todo, index) =>
      index === action.index
          ? { ...todo, completed: !todo.completed }
          : todo
      );
      return { todos: updatedTodoList };
    }
    case actions.REMOVE: {
      const filteredTodoItem = state.todos.filter(
        (todo, index) => index !== action.index
      );
      return { todos: filteredTodoItem };
    }
    default:
      return state;
  }
};
```

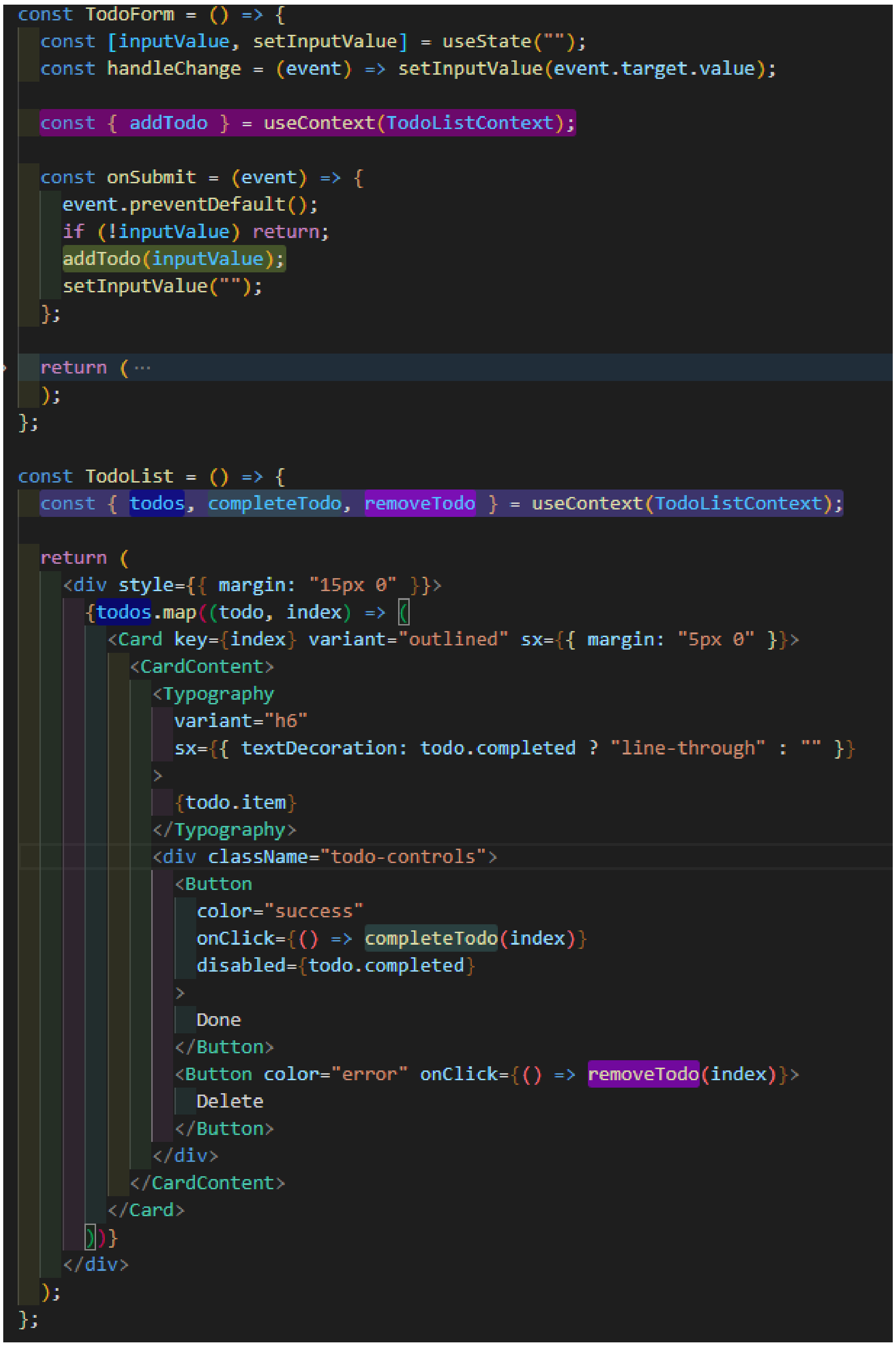
Figure 29. Initialize the state, actions, and reducers in a React Context application



```
const TodoListContext = createContext();

const TodoListProvider = ({ children }) => {
  const [state, dispatch] = useReducer(reducer, initialState);

  const value = {
    todos: state.todos,
    addTodo: (item) => dispatch({ type: actions.ADD, item }),
    completeTodo: (index) => dispatch({ type: actions.COMPLETE, index }),
    removeTodo: (index) => dispatch({ type: actions.REMOVE, index }),
  };

  return (
    <TodoListContext.Provider value={value}>
      {children}
    </TodoListContext.Provider>
  );
};

const App = () => (
  <TodoListProvider>
    <div className="todo-app">
      <Typography align="center" variant="h2" gutterBottom>
        Todo List
      </Typography>
      <TodoForm />
      <TodoList />
    </div>
  </TodoListProvider>
);
```

Figure 30. Creating a context and a provider

A context is created using the `createContext` method. Inside the provider component, the `reducer` function and the `initialState` previously defined are passed to the `useReducer` hook, returning an array with the current state value and a `dispatch` method. The `value` object, which is the prop passed into the built-in `Provider` component of the context, is the object that holds the to-do list state, and the three functions `addTodo`, `completeTodo`, and `removeTodo` which will use the `dispatch` method to trigger the “ADD”, “COMPLETE”, and “REMOVE” actions to the child components, respectively. After creating the `TodoListProvider` component, it is wrapped around the children of the `App` component, which will allow them – the consumers – to subscribe to context changes. All the properties defined in the `value` object can now be accessed in other parts of the application using the `useContext` hook. Finally, in the `TodoList` and `TodoForm` components, subscribe to the `TodoListContext` with the `useContext` hook, and extract the required properties returned from the hook depending on each objective of each component, as highlighted in Figure 31.



```
const TodoForm = () => {
  const [inputValue, setInputValue] = useState("");
  const handleChange = (event) => setInputValue(event.target.value);

  const { addTodo } = useContext(TodoListContext);

  const onSubmit = (event) => {
    event.preventDefault();
    if (!inputValue) return;
    addTodo(inputValue);
    setInputValue("");
  };

  return (⋯
  );
};

const TodoList = () => {
  const { todos, completeTodo, removeTodo } = useContext(TodoListContext);

  return (
    <div style={{ margin: "15px 0" }}>
      {todos.map((todo, index) => (
        <Card key={index} variant="outlined" sx={{ margin: "5px 0" }}>
          <CardContent>
            <Typography
              variant="h6"
              sx={{ textDecoration: todo.completed ? "line-through" : "" }}
            >
              {todo.item}
            </Typography>
            <div className="todo-controls">
              <Button
                color="success"
                onClick={() => completeTodo(index)}
                disabled={todo.completed}
              >
                Done
              </Button>
              <Button color="error" onClick={() => removeTodo(index)}>
                Delete
              </Button>
            </div>
          </CardContent>
        </Card>
      ))}
    </div>
  );
};
```

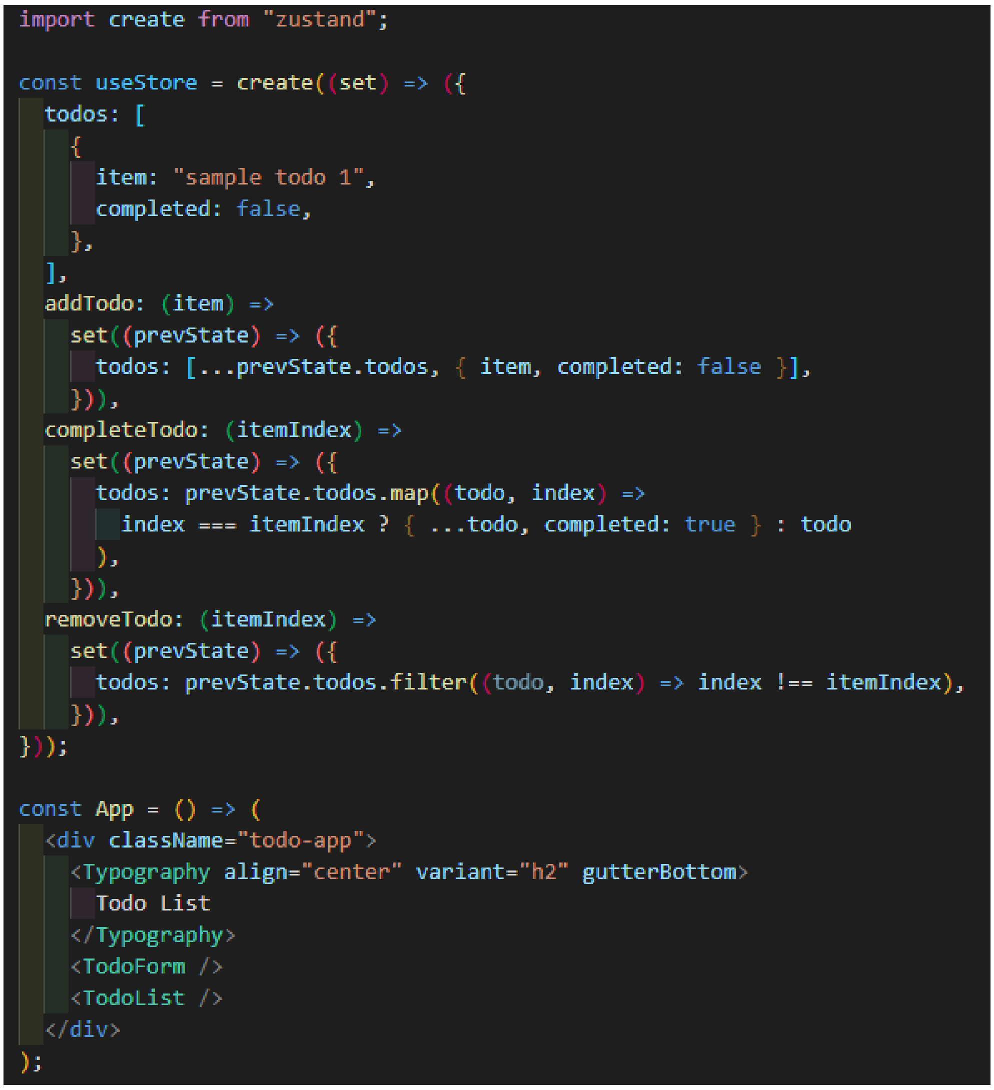
Figure 31. TodoForm and TodoList using Context API and React hooks

Though React Context is natively built into React and no additional third-party packages need to be installed, there are certain situations where using it might not be the best option. One of the worst problems concerning React Context is that the consumers of a context always re-render if the state it subscribed to changes, regardless of whether the component used that updated state. For this reason, it is advisable to have separated contexts to prevent re-rendering unnecessarily. Additionally, React Context itself is not a state management tool without the `useContext` and `useReducer` hooks, despite popular belief, therefore using it may be confusing for inexperienced developers. In conclusion, using the Context API may cause performance problems if not done carefully, but the benefits it brings to the application are undeniable.

5.4.5. Zustand

Best known among the front-end developer community for its simplicity, Zustand is a small, fast, and scalable state management solution using simplified flux principles.[37] Despite being only 1.5kB in size, it resolves many common pitfalls encountered when developing an application with React. Zustand is mostly similar to Redux Toolkit in its syntax and usage, but even more straightforward and easy to use without any of the complicated concepts that Redux introduces. This is the reason why in smaller applications, Zustand is preferred over other libraries such as Redux and React Context.

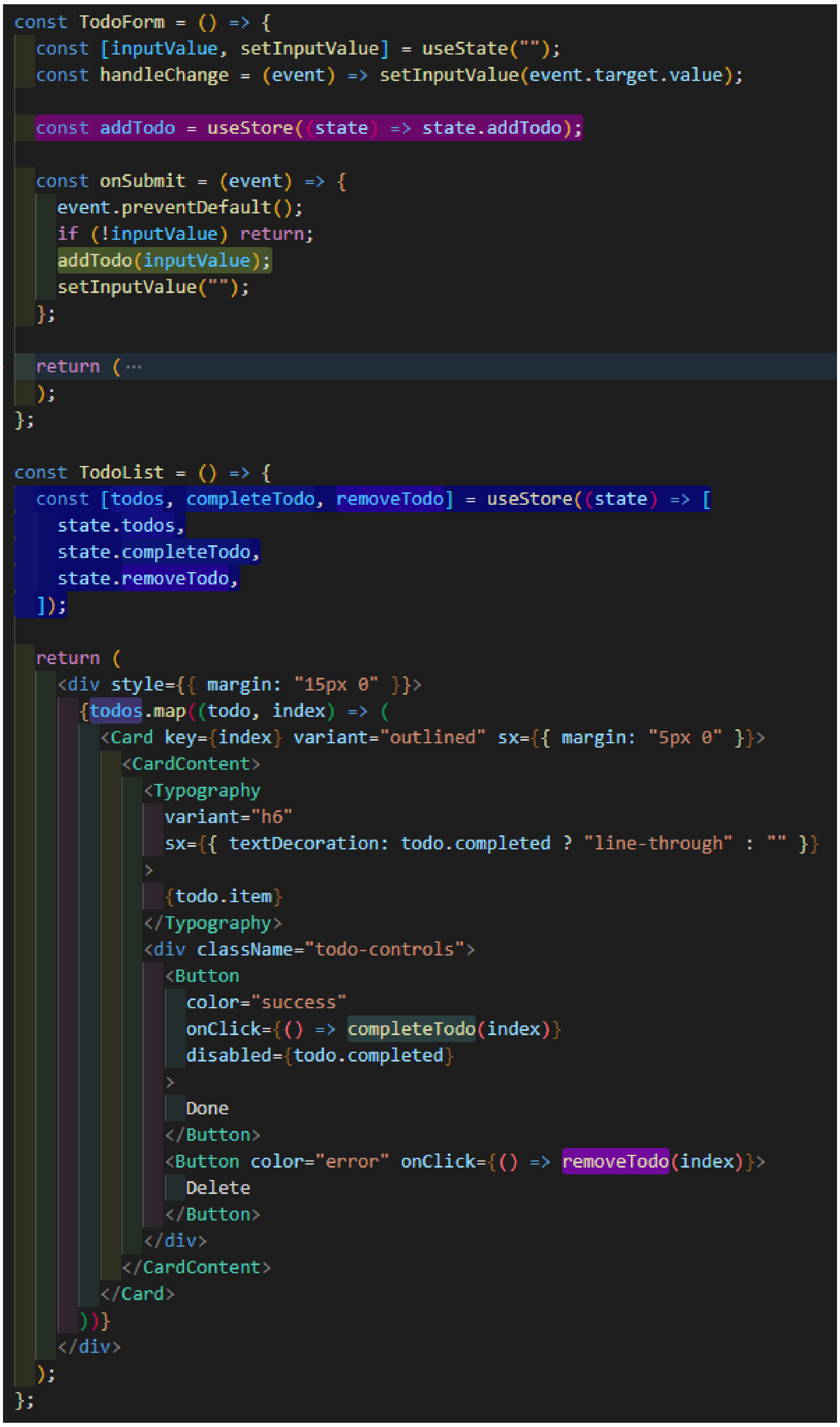
The implementation of Zustand into the example to-do list application is started by installing its package with `npm install zustand`. After the installation is completed, the `create` function is imported from the package. This function will act as the application store and accepts a callback function as its argument, which in turn accepts a `set` function that will help update the immutable state, similar to React's `useState` hook. Let us see how this is implemented in Figure 32 before further explanation.



```
import create from "zustand";

const useStore = create((set) => ({
  todos: [
    {
      item: "sample todo 1",
      completed: false,
    },
  ],
  addTodo: (item) =>
    set((prevState) => ({
      todos: [...prevState.todos, { item, completed: false }],
    })),
  completeTodo: (itemIndex) =>
    set((prevState) => ({
      todos: prevState.todos.map((todo, index) =>
        index === itemIndex ? { ...todo, completed: true } : todo
      ),
    })),
  removeTodo: (itemIndex) =>
    set((prevState) => ({
      todos: prevState.todos.filter((todo, index) => index !== itemIndex),
    })),
}));

const App = () => (
  <div className="todo-app">
    <Typography align="center" variant="h2" gutterBottom>
      Todo List
    </Typography>
    <TodoForm />
    <TodoList />
  </div>
);
```

Figure 32. Creating an application store with Zustand

Compared to Redux Toolkit, it can be said that Zustand provides a remarkably better and simplified way to create a store with the same purpose. The `create` function after being called will return a hook, which in this case is called `useStore`. This hook can now be used anywhere in the application without having to wrap the `App` component with a provider or pass props down to child components. Through the actions derived from the hook, these components will be re-rendered whenever the state changes. The highlighted lines in Figure 33 demonstrate the usage of the `useStore` hook within the `TodoForm` and `TodoList` components.



```
const TodoForm = () => {
  const [inputValue, setInputValue] = useState("");
  const handleChange = (event) => setInputValue(event.target.value);

  const addTodo = useStore((state) => state.addTodo);

  const onSubmit = (event) => {
    event.preventDefault();
    if (!inputValue) return;
    addTodo(inputValue);
    setInputValue("");
  };

  return (⋯
  );
};

const TodoList = () => {
  const [todos, completeTodo, removeTodo] = useStore((state) => [
    state.todos,
    state.completeTodo,
    state.removeTodo,
  ]);

  return (
    <div style={{ margin: "15px 0" }}>
      {todos.map((todo, index) => (
        <Card key={index} variant="outlined" sx={{ margin: "5px 0" }}>
          <CardContent>
            <Typography
              variant="h6"
              sx={{ textDecoration: todo.completed ? "line-through" : "" }}
            >
              {todo.item}
            </Typography>
            <div className="todo-controls">
              <Button
                color="success"
                onClick={() => completeTodo(index)}
                disabled={todo.completed}
              >
                Done
              </Button>
              <Button color="error" onClick={() => removeTodo(index)}>
                Delete
              </Button>
            </div>
          </CardContent>
        </Card>
      ))}
    </div>
  );
};
```

Figure 33. TodoForm and TodoList implemented with Zustand

By using hooks as the primary way to consume and modify state, Zustand ensures that the application will only re-render components affected by the changes. Though this approach is similar to Redux Toolkit, it is even better as no providers are needed. Zustand is a great state management tool that is not only exclusive to React but for other frameworks and applications as well. This means that for companies that use micro front-ends, state management can be integrated using one central store. Another reason for Zustand's popularity is that it is flexible, therefore, it does not require following much boilerplate like React Context. In the world of state management, Zustand is one of the examples where simplicity is better than over-engineering.

Though Zustand is a remarkable UI state management library, it has its flaws as any other tool. Many parts of its documentation are still labeled as "under construction", implying that the document is incomplete and does not cover all the use cases available. Even for completed items, the writing could be improved to avoid confusion. For example, beginners might be overwhelmed trying to understand how creating a store and updating the state are done due to the library having too many built-in functions. This can also be a problem for more complex applications requiring complicated state manipulations.

5.4.6. XState

XState is a library for creating, interpreting, and executing finite state machines and state charts, as well as managing invocations of those machines as actors.[38] In other words, XState provides a new way of managing the application state with state machines, unlike previously introduced UI state management libraries. This is why, to thoroughly understand and maximize the potential of XState, several fundamental concepts of computer science will be presented first.

A finite state machine is a mathematical model of computation that describes the behavior of a system that can be in only one state at any given time.[38] It illustrates how the state of a process transitions to another state when an event occurs.[39] With state machines, the number of possible states is limited and the transition of the states is controlled, therefore resulting in a predictable and reliable application management system.

State charts, like other types of diagrams, are a visual representation of state machines. It modelizes the different states an object can have during its lifecycle. In comparison to state machines, one undeniable benefit of using statecharts is that they make it easier to ensure that all the states have been explored. However, they also introduce a foreign way of coding which in some cases is simply not necessary. Figure 34, referenced from the XState's documentation, is the basic guideline for creating a state chart.[39]

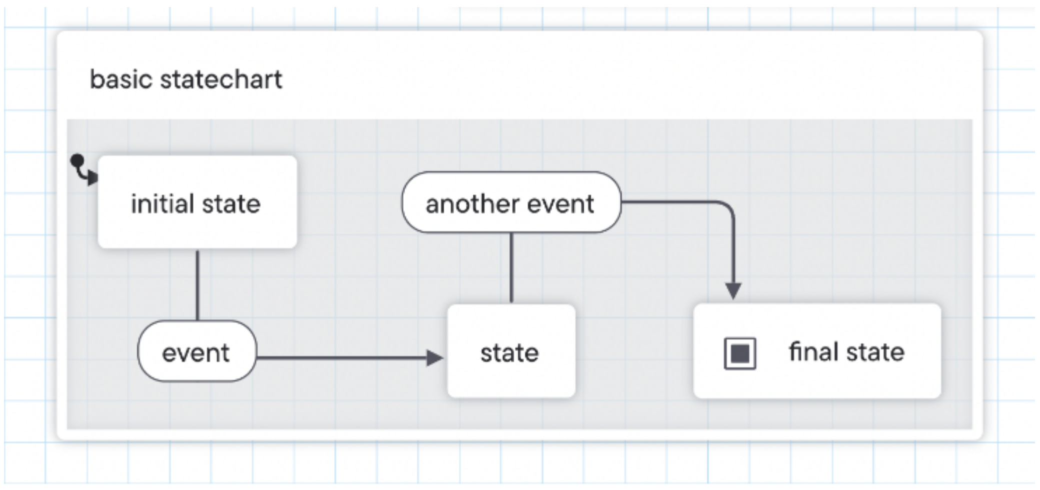


Figure 34. Basic state charts guideline

The actor model is a mathematical model of message-based computation that simplifies how multiple "actors" communicate with each other through events called "messages".[40] An actor can send and receive messages, as well as decide its action in response to the received message: changing its own local state, sending messages to other actors, or spawning new actors.[40] When a state machine transitions its state due to an event, the next state contains the new context value and the action to be executed, which can be used as the actor's new local state and behavior. This is why state machines and state charts are usually used to demonstrate actor models.

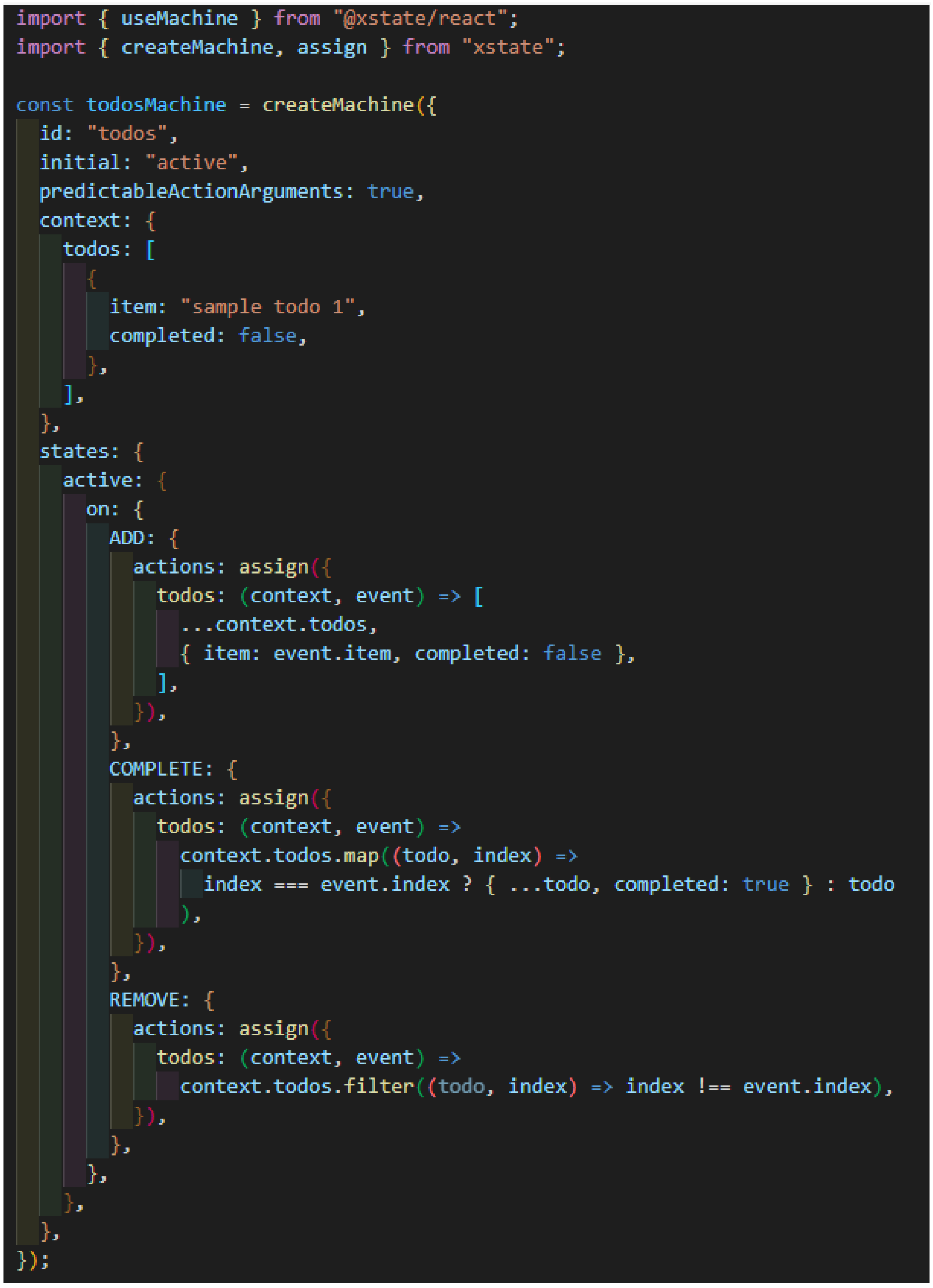
With all the fundamental concepts explained, let us now implement XState into the example application. First, `npm install xstate @xstate/react` is run in the terminal to install its core library and React hooks. To create a state machine, XState provides a built-in function called `createMachine` which accepts an object of state configuration as a requirement, as well as an optional context object representing extended state values. The state configuration object has over twenty optional properties, from which some of the more important ones are:

- `id` - The unique ID of the state node, which can be referenced as a transition target.[41]
- `initial` - The initial state node key.[41]
- `context` - The initial context.[41]
- `states` - The mapping of state node keys to their state node configurations.[41]

Although the optional context argument will not be used in the application, it is worth briefly going through its four optional properties:

- `actions` - the mapping of action names to their implementation.[42]
- `delays` - the mapping of delay names to their implementation.[42]
- `guards` - the mapping of transition guard names to their implementation.[42]
- `services` - the mapping of invoked service names to their implementation.[42]

The options can be overwhelming at first without reading the documentation carefully. Before further explanation, refer to figure 35 to understand how the `createMachine` function is implemented into the to-do list application.



```
import { useMachine } from "@xstate/react";
import { createMachine, assign } from "xstate";

const todosMachine = createMachine({
  id: "todos",
  initial: "active",
  predictableActionArguments: true,
  context: {
    todos: [
      {
        item: "sample todo 1",
        completed: false,
      },
    ],
  },
  states: {
    active: {
      on: {
        ADD: {
          actions: assign({
            todos: (context, event) => [
              ...context.todos,
              { item: event.item, completed: false },
            ],
          }),
        },
        COMPLETE: {
          actions: assign({
            todos: (context, event) =>
              context.todos.map((todo, index) =>
                index === event.index ? { ...todo, completed: true } : todo
              ),
          }),
        },
        REMOVE: {
          actions: assign({
            todos: (context, event) =>
              context.todos.filter((todo, index) => index !== event.index),
          }),
        },
      },
    },
  },
});
```

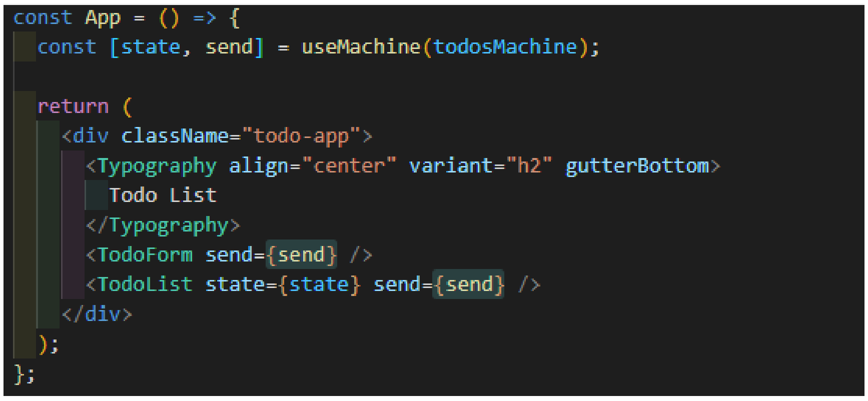
Figure 35. Creating a state machine with XState's createMachine function

Let us now go through each of the options respectively. The `todosMachine` is initialized in an “active” state, and because this is a simple example application, the machine’s state will not be changed. The next configuration option, `predictableActionArguments`, has been set to “true” as advised by XState. Doing this will allow XState to always call an action with the event directly responsible for the related transition.[43] Nevertheless, it is not a requirement for beginners to understand this option thoroughly. One of the more important properties is the `context` option, which is a reference to the application states to distinguish them from machine states. Anything specified in the `context` property will be the initial context value of the application. Lastly, the machine's `states` property specifies state configurations. This configuration object describes all the states the machine can be in, as well as the actions to take when a certain event happens. As seen in figure 35, the assign function is used to update the machine's context through an object. This object is called an "assigner", and it can change any context value through two arguments: the `context`, which is the current context of the application, and the `event`, which is the event that triggered this action. The handling logic inside each action is similar to other state management libraries discussed above.

After creating a state machine, the `useMachine` hook is imported from the "@xstate/react" package to help start the service in the to-do list application. This hook accepts an argument referring to a XState machine and returns a tuple of three values:

- `state` - The current state of the machine as an XState `State` object.[44]
- `send` - A function that sends events to the running service.[44]
- `service` - The created service.[44]

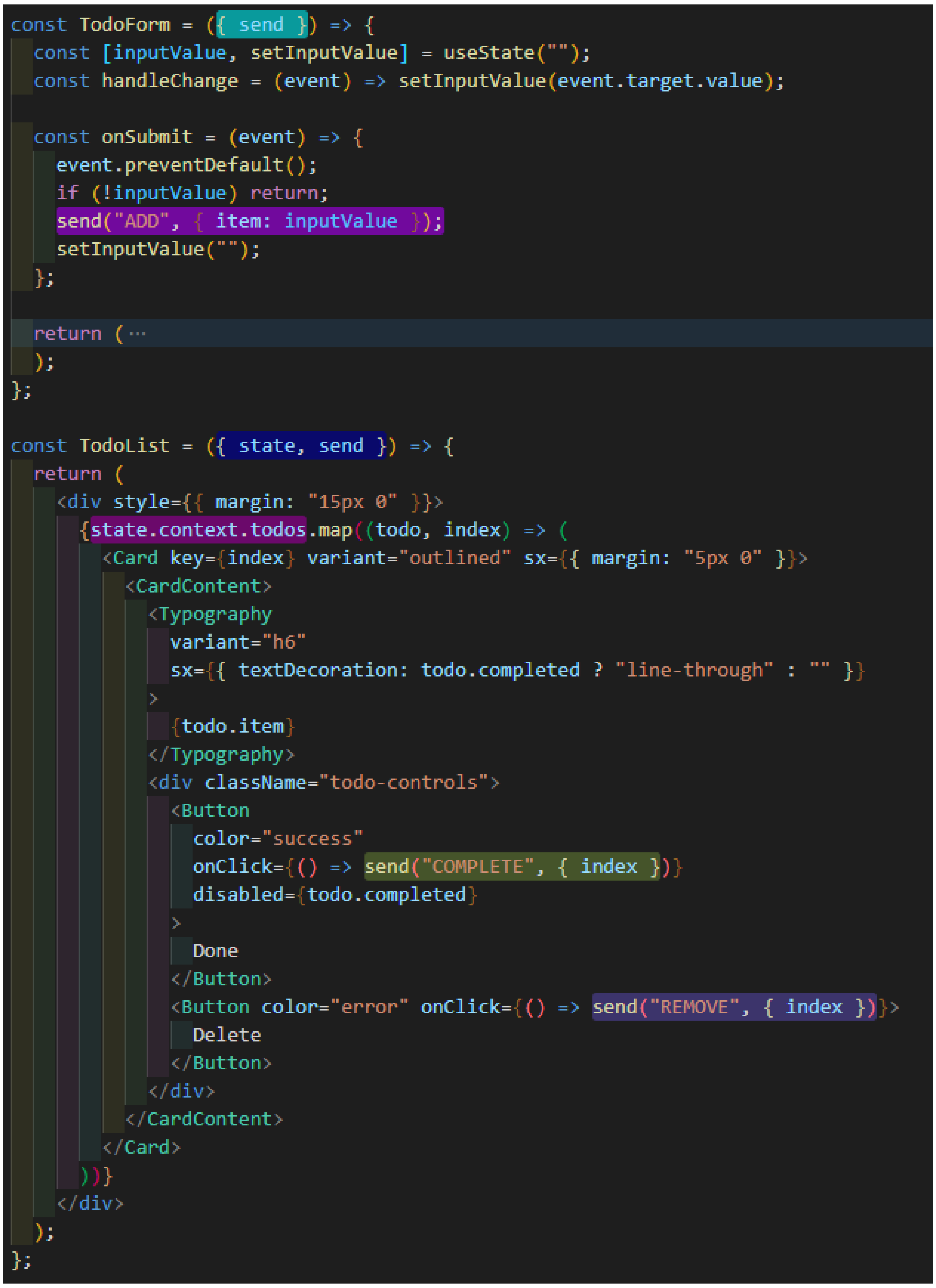
In this application, only the first two values will be used. Refer to figure 36 to understand how the state machine is implemented into the application.



```
const App = () => {
  const [state, send] = useMachine(todosMachine);

  return (
    <div className="todo-app">
      <Typography align="center" variant="h2" gutterBottom>
        Todo List
      </Typography>
      <TodoForm send={send} />
      <TodoList state={state} send={send} />
    </div>
  );
};
```

Figure 36. Using XState's state machine in a React application through the useMachine hook

The `state` and `send` values will then be passed down to the `TodoForm` and `TodoList` components to be used as needed. Figure 37 shows how the syntax regarding these values is performed inside the child components.



```
const TodoForm = ({ send }) => {
  const [inputValue, setInputValue] = useState("");
  const handleChange = (event) => setInputValue(event.target.value);

  const onSubmit = (event) => {
    event.preventDefault();
    if (!inputValue) return;
    send("ADD", { item: inputValue });
    setInputValue("");
  };

  return (···
  );
};

const TodoList = ({ state, send }) => {
  return (
    <div style={{ margin: "15px 0" }}>
      {state.context.todos.map((todo, index) => (
        <Card key={index} variant="outlined" sx={{ margin: "5px 0" }}>
          <CardContent>
            <Typography
              variant="h6"
              sx={{ textDecoration: todo.completed ? "line-through" : "" }}
            >
              {todo.item}
            </Typography>
            <div className="todo-controls">
              <Button
                color="success"
                onClick={() => send("COMPLETE", { index })}
                disabled={todo.completed}
              >
                Done
              </Button>
              <Button color="error" onClick={() => send("REMOVE", { index })}>
                Delete
              </Button>
            </div>
          </CardContent>
        </Card>
      ))}
    </div>
  );
};
```

Figure 37. Returned useMachine values implemented in TodoForm and TodoList components

Everything implemented in the example application is barely the surface of state machines, especially XState as a state management tool. Due to its complexity and steep learning curve, XState is less popular compared to other UI state management libraries. Nevertheless, state machines approach complex application states differently by defining all the possible and available states as well as their transition events, therefore keeping everything in control and preventing situations where the application is in an unknown state. In conclusion, state machines like XState offer a better way to organize complex states in React applications and consequently help scale the applications easier, but also much more challenging to adapt compared to normal state management tools.

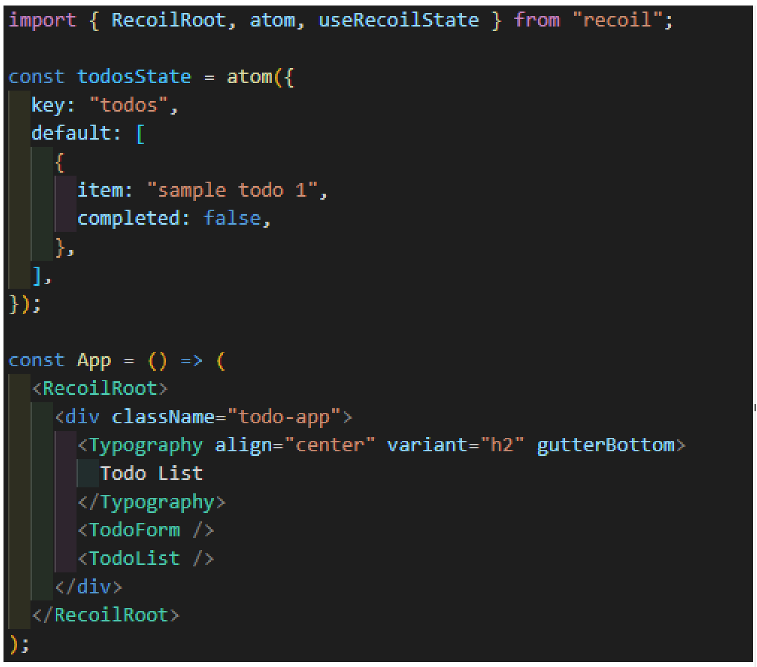
5.4.7. Recoil

Recoil was developed by Meta, formerly known as Facebook, in 2020 as its own state management library. Compared to other tools, Recoil is still relatively new and experimental. However, it has already gained the attention of many web developers as it solves the complex boilerplate problem of Redux and unnecessary re-renders of React Context. Recoil is strongly built based on hooks and introduces two new concepts – atoms and selectors – which are used to store the state and calculate derived data, respectively.

Atoms are units of state which are updatable and subscribable.[45] Whenever an atom is updated, each subscribed component is re-rendered with the new value.[45] They can be created with the built-in `atom` function, which accepts a unique key and a default value. Using Recoil’s `useRecoilState` hook, which accepts an atom as its argument and returns the current state along with an updater function, these values can then be read and modified. This hook is similar to React's `useState` except that any component can access the state regardless of its location in the component tree.

Selectors are pure functions that accept atoms or other selectors as their input and will be re-evaluated whenever the input changes.[45] They are used to calculate derived data that is based on the application state, which is then called derived state.[45] This is a powerful concept as it makes creating dynamic states based on other states possible. Similar to atoms, components that subscribe to selectors will be re-rendered when the selectors change. To create a selector, use Recoil's built-in `selector` function and provide a unique key as well as a `get` property as the function that is to be computed. The function passed into the `get` property will have access to other atoms and selectors through the `get` argument passed to it.[45] Components can then read selectors using the `useRecoilValue` hook, which will return a value corresponding to the atom or selector that is its input argument.

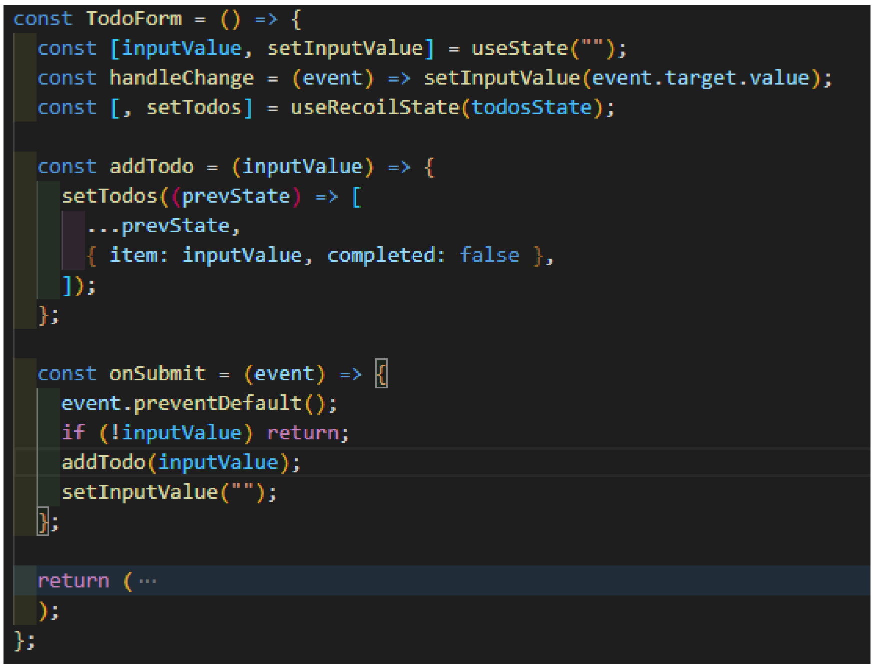
Now that the core concepts have been familiarized, `npm install recoil` is run in the terminal to install the library's package. Then, `RecoilRoot`, `atom`, and `useRecoilState` are imported from the package. `RecoilRoot` will provide context values of atoms and selectors to child components, therefore it is necessary for `RecoilRoot` to be the top-level parent to any components that use Recoil. The `atom` function, as previously explained, will be used to create an atom – a Recoil's state object – and `useRecoilState` will be used to perform actions on the given state. Refer to figure 38 to see how a Recoil application is initialized, as well as how an atom was created with the `atom` utility function.



```
import { RecoilRoot, atom, useRecoilState } from "recoil";

const todosState = atom({
  key: "todos",
  default: [
    {
      item: "sample todo 1",
      completed: false,
    },
  ],
});

const App = () => (
  <RecoilRoot>
    <div className="todo-app">
      <Typography align="center" variant="h2" gutterBottom>
        Todo List
      </Typography>
      <TodoForm />
      <TodoList />
    </div>
  </RecoilRoot>
);
```

Figure 38. Initializing a Recoil application and creating an atom

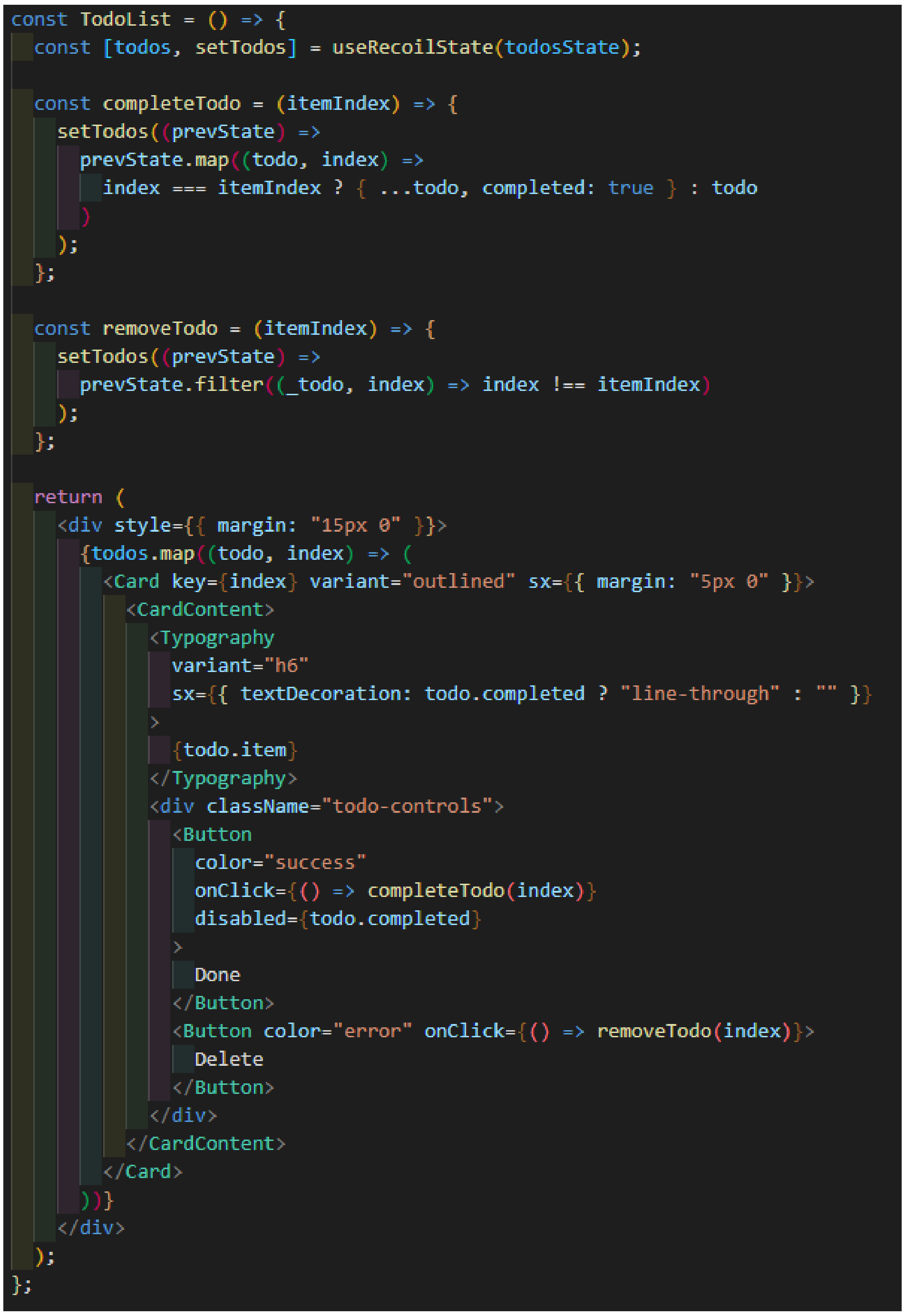
Notice that a unique key and default value were given to the todosState atom. The atom can now be used in TodoForm and TodoList components as needed using the useRecoilState hook. Figure 39 shows TodoForm's implementation using Recoil's hook and a function handling adding to-do.



```
const TodoForm = () => {
  const [inputValue, setInputValue] = useState("");
  const handleChange = (event) => setInputValue(event.target.value);
  const [, setTodos] = useRecoilState(todosState);

  const addTodo = (inputValue) => {
    setTodos((prevState) => [
      ...prevState,
      { item: inputValue, completed: false },
    ]);
  };

  const onSubmit = (event) => {
    event.preventDefault();
    if (!inputValue) return;
    addTodo(inputValue);
    setInputValue("");
  };

  return (⋯
  );
};
```

Figure 39. TodoForm implemented with Recoil

Since the todos state is not necessary for this component, it was not derived from the returned values of the hook. Other than the new useRecoilState hook introduced by Recoil, other implementation logic is rather similar to using React's local component state and hooks. Another way to achieve the same objective is to use useSetRecoilState hook, also provided by Recoil's API. The hook takes an atom or writable selector and simply returns the setter function for updating the value of atoms. In fact, this is the recommended hook to use when a component intends to write to the state without reading it, as using useRecoilState will require the component to subscribe to state updates and cause unnecessary re-renders.[46] Nevertheless, for the simplicity of the example application, no further changes will be made. Instead, let us look at TodoList's implementation, shown in figure 40.



```
const TodoList = () => {
  const [todos, setTodos] = useRecoilState(todosState);

  const completeTodo = (itemIndex) => {
    setTodos((prevState) =>
      prevState.map((todo, index) =>
        index === itemIndex ? { ...todo, completed: true } : todo
      )
    );
  };

  const removeTodo = (itemIndex) => {
    setTodos((prevState) =>
      prevState.filter((_todo, index) => index !== itemIndex)
    );
  };

  return (
    <div style={{ margin: "15px 0" }}>
      {todos.map((todo, index) => (
        <Card key={index} variant="outlined" sx={{ margin: "5px 0" }}>
          <CardContent>
            <Typography
              variant="h6"
              sx={{ textDecoration: todo.completed ? "line-through" : "" }}
            >
              {todo.item}
            </Typography>
            <div className="todo-controls">
              <Button
                color="success"
                onClick={() => completeTodo(index)}
                disabled={todo.completed}
              >
                Done
              </Button>
              <Button color="error" onClick={() => removeTodo(index)}>
                Delete
              </Button>
            </div>
          </CardContent>
        </Card>
      ))}
    </div>
  );
};
```

Figure 40. TodoList implemented with Recoil

The execution of using Recoil in the component is simple and straightforward, with little needs to be discussed. All the event handler functions as well as the hook are familiar and self-explanatory, as the library closely resembles React's syntax and architecture. This is the reason why using Recoil feels remarkably similar to using React itself, with the addition of many useful hooks available for utilization.

Recoil is widely adopted by the React community because it feels much more simple, more elegant, and more intuitive compared to Redux and React Context. By using Recoil, unnecessary re-rendering problems faced by the older libraries are eliminated, making it easier to build an application with less concern for side effects. It also closely resembles React, which reduces the number of new terms and concepts that need to be learned. Developers simply need to become familiar with the use cases for the hooks provided by the Recoil library as it is based on React's well-known "hook" concept. However, it is important to remember that other state management libraries have had many years of development, while Recoil is fairly new and still subject to changes. While small applications might benefit greatly from using this library, larger-scale applications need to consider carefully to avoid needless complexity and confusion as the codebase continues to grow.

5.4.8. Jotai

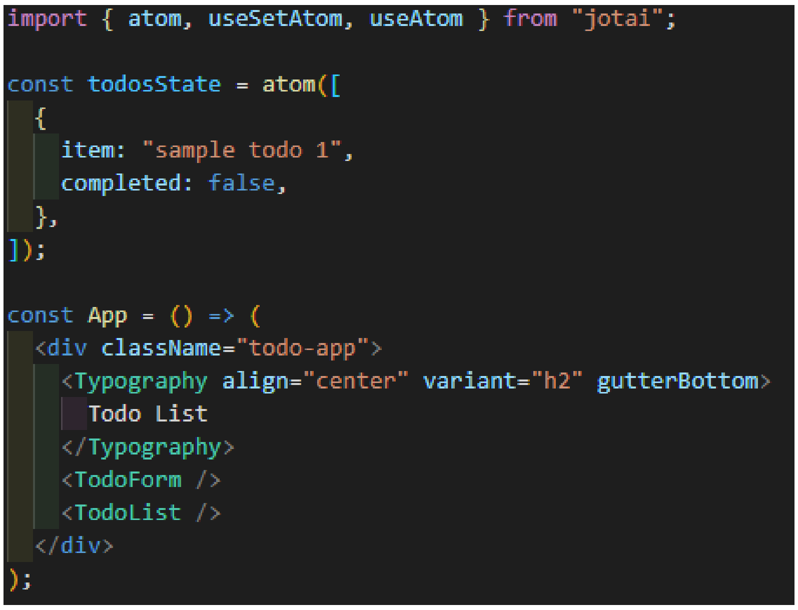
Jotai is a state management library that is built upon the React basics to keep everything as simple as possible.[47] Similar to Recoil, Jotai was created to solve extra re-rendering issues in React, which is why it is built using the same atomic model. The two libraries share many similarities in their concepts and principles, with the only difference being that Jotai’s atom object, which does not require a unique key, is the combination of both Recoil’s atoms and selectors.[48] This makes Jotai even quicker and easier to learn compared to Recoil. Additionally, Jotai has a very big amount of integrations with other libraries like React Query, XState, Redux, Zustand, and many others, allowing simple state management with extra functionalities.[48] Though this thesis will not introduce any of these integrations, it might be worth reading through them on Jotai’s official documentation web page for future reference.

The library revolves around two basic APIs: atoms and the `useAtom` hook. In Jotai, atoms can take the form of anything: a string, an array, another atom, or even a derived state. Because of this, the `atom` function that is used to create atoms accepts anything as its argument and will return an atom configuration object.

This object can then be passed into the provided `useAtom` hook, which will then return a tuple of the current atom value as well as its updater function, similar to React's `useState` and Recoil's `useRecoilState`. Other additional features of Jotai includes:

- `Provider` component – This component should be used if the application needs initial values for server-side rendering.[49]
- `useSetAtom` hook – This hook is useful in cases where the component needs to update the atom value without reading it.[49]
- `useAtomValue` hook – This hook allows reading without modifying the atom value.[49]

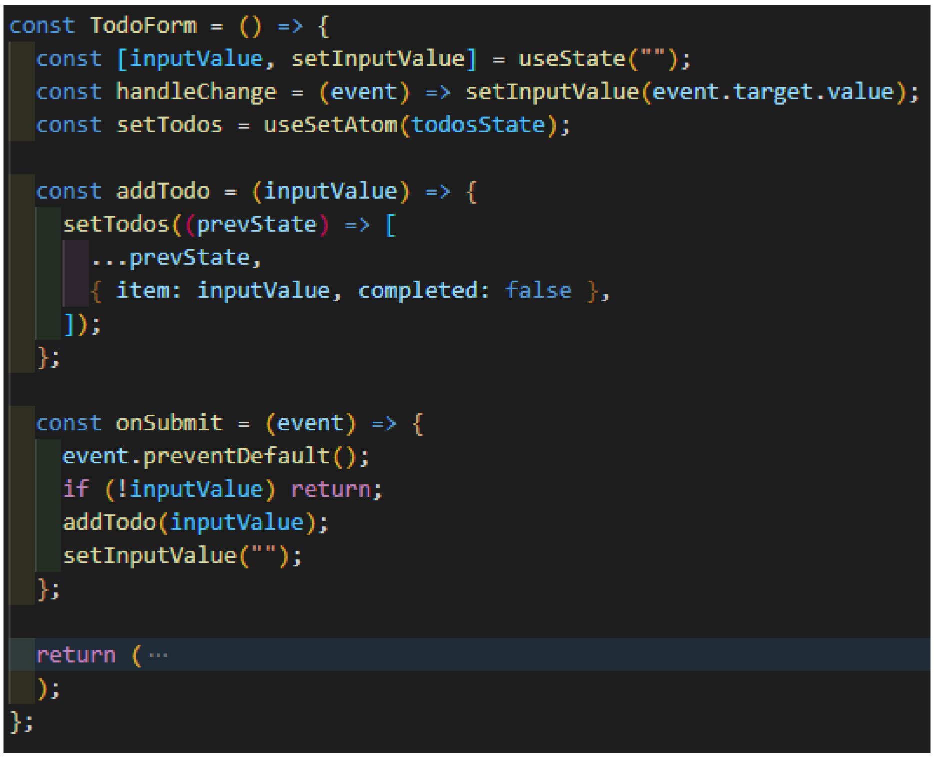
To start the implementation, `npm install jotai` is run in the terminal. After installation is completed, the `atom`, `useAtom`, and `useSetAtom` APIs are imported from the package. These three APIs will help set up a fully functional Jotai application. Historically, a `Provider` component was also needed in order for child components to be able to read atom values, but a provider-less mode has been introduced in version 0.15.0, which is why the `App` component will be simplified without any wrapper component. Figure 41 shows the syntax of creating an atom object and the implementation of the `App` component.



```
import { atom, useSetAtom, useAtom } from "jotai";

const todosState = atom([
  {
    item: "sample todo 1",
    completed: false,
  },
]);

const App = () => (
  <div className="todo-app">
    <Typography align="center" variant="h2" gutterBottom>
      Todo List
    </Typography>
    <TodoForm />
    <TodoList />
  </div>
);
```

Figure 41. Creating atom object and implementing App component with Jotai

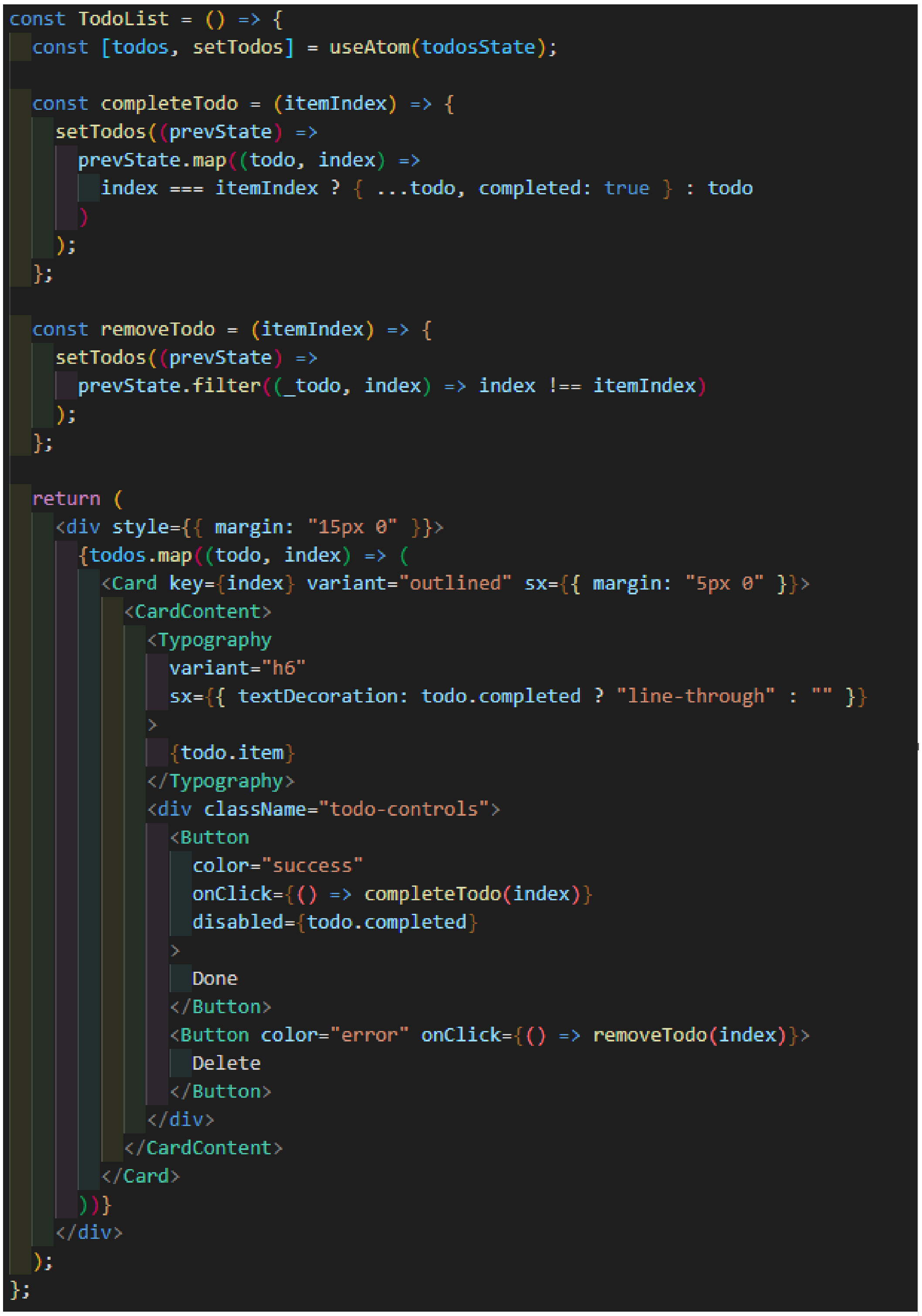
As shown in the figure, although the atom creation is similar to that of Recoil, it is more minimalistic without any string keys. In their official documentation, Jotai stated that the atom configuration itself can be used as a key.[50] The `App` component, as stated before, is also very clean with just a few lines of code, and does not contain any programming logic. Now that the building blocks of the to-do list application have been set up, the two imported hooks will be used to achieve specific goals in the child components. Figure 42 shows the `useSetAtom` hook used in the `TodoForm` component to handle adding new to-do items, as the component does not require reading the atom value.



```
const TodoForm = () => {
  const [inputValue, setInputValue] = useState("");
  const handleChange = (event) => setInputValue(event.target.value);
  const setTodos = useSetAtom(todosState);

  const addTodo = (inputValue) => {
    setTodos((prevState) => [
      ...prevState,
      { item: inputValue, completed: false },
    ]);
  };

  const onSubmit = (event) => {
    event.preventDefault();
    if (!inputValue) return;
    addTodo(inputValue);
    setInputValue("");
  };

  return (…
  );
};
```

Figure 42. TodoForm implemented with Jotai

The hook for “updating atom value without reading it” is used here to demonstrate its syntax, contrary to Recoil where it was not used for simplicity reasons. Other than this, the remaining implementation logic is identical to Recoil’s version of the `TodoForm` component. The last component in the application – `TodoList` – is shown using the `useAtom` hook to handle displaying to-dos, completing and removing to-dos in figure 43.



```
const TodoList = () => {
  const [todos, setTodos] = useAtom(todosState);

  const completeTodo = (itemIndex) => {
    setTodos((prevState) =>
      prevState.map((todo, index) =>
        index === itemIndex ? { ...todo, completed: true } : todo
      )
    );
  };

  const removeTodo = (itemIndex) => {
    setTodos((prevState) =>
      prevState.filter((_todo, index) => index !== itemIndex)
    );
  };

  return (
    <div style={{ margin: "15px 0" }}>
      {todos.map((todo, index) => (
        <Card key={index} variant="outlined" sx={{ margin: "5px 0" }}>
          <CardContent>
            <Typography
              variant="h6"
              sx={{ textDecoration: todo.completed ? "line-through" : "" }}
            >
              {todo.item}
            </Typography>
            <div className="todo-controls">
              <Button
                color="success"
                onClick={() => completeTodo(index)}
                disabled={todo.completed}
              >
                Done
              </Button>
              <Button color="error" onClick={() => removeTodo(index)}>
                Delete
              </Button>
            </div>
          </CardContent>
        </Card>
      ))}
    </div>
  );
};
```

Figure 43. TodoList implemented with Jotai

Like the `TodoForm` component, `TodoList`'s implementation using Jotai is also identical to its Recoil counterpart. A tuple of atom state value and its setter function is returned from the `useAtom` hook, then used in handling functions to acquire their particular objectives. In conclusion, there is not much to discuss about the syntax and functionality of the `TodoList` component when implemented with Jotai, because they remain similar to how they were done using Recoil.

As Jotai is built highly inspired by Recoil, the two libraries are usually seen compared to each other. Many developers believe Jotai is better than Recoil due to the fact that it has a noticeably smaller bundle size, less boilerplate code, and simpler core API while serving the same purpose. Jotai solves the issues that Recoil tried to solve, and additionally the problems that Recoil faced – the performance speed and memory leakage. Considering these reasons, it is probably better to use Jotai in smaller projects where using Recoil might be too much work. However, as the projects grow, debugging and snapshot testing can become an obstacle in Jotai that will not happen with Recoil. Furthermore, although both libraries are new and experimental, Recoil will no doubt receive more attention as it is developed by Meta. Choosing which state management libraries is heavily dependent on the developer’s preferences and the project’s scalability, though learning both will undoubtedly be beneficial as the atomic state is expected to become the future of state management solutions.

6. SELECTION OF UI STATE MANAGEMENT TOOL IN A REACT APPLICATION

UI state management is crucial when working with React. Understanding how to choose a practical and reasonable state management library will undoubtedly enhance the application performance and the users' experience. Every one of the state management libraries outlined previously offers a different and unique approach for managing shared data across the entire application, though the goal of solving problems related to application states remains consistent. When deciding which state management tool to use, it is beneficial to take the scale of the project as well as the developer's personal preferences into consideration. For instance, using React's `useState` hook will be better than any external third-party libraries when building small personal projects. This means the most complicated tools are not always the best ones, as they might bring unnecessary confusion and over-complicate the whole React application.

When experimenting with any new state management library, one should think about how much one wishes to know about the particular tool. While small examples are useful for faster learning, they often make a library appear too complicated. Larger examples can help understand how to put something into practice, but they can be too overwhelming to use as an introduction. One common approach of online tutorial resources and self-studied developers is to start with small projects first, then gradually get into more advanced and realistic ones. This is great as it prevents absorbing too much information at once, which can be overwhelming even for the simplest tool.

Among all the UI state management libraries, Redux has inarguably held the most-favorite spot for the longest time. With the introduction of the Redux Toolkit that eliminates the excessive boilerplate problem Redux used to have, using it has become even more concise and straightforward. As a result, understanding Redux thoroughly is really beneficial for anyone aiming to be a good React web developer. However, while Redux used to dominate the state management libraries, several other tools have proven to be strong competitors. This is due to the fact that each of the other libraries addresses a unique set of problems that Redux may have overlooked. Particularly, Recoil and Jotai, the latest addition to the UI state management libraries, have been receiving a lot of attention from many developers. They both take an atomic, React-inspired approach to state management, significantly reducing their learning curve while maintaining fully functional and feature-rich APIs. Nevertheless, they are still open to change, which causes some developers, particularly those working for large companies, to choose a more reliable and matured approach.

There is not one definite answer to the question “Which state management tool is best for UI state in React?”. Throughout this thesis, it has been made clear that although their complexity may vary, each state management tool solves its own unique set of problems, and each is favorable in different circumstances. For new React developers, learning Redux is generally the most recommended as even though not as popular as before, it is still used worldwide. Many programming tutorials might instead encourage using React Context with `useContext` and `useReducer` hooks because it is the native approach to state management, reducing any external libraries needed. However, it is important to understand that React Context itself is not a state management tool, so using it without proper research might be misleading. Additionally, learning Recoil and Jotai can be beneficial, as they are the most recent state management libraries with a very low learning curve that are progressively being adopted by many developers within the community. In conclusion, there are no options that are better than the others; the choice of which UI state management tool to utilize is up to the developers and their project to decide.

7. CONCLUSIONS

State management in React is the process of maintaining a piece of data across multiple components to form a complete data flow that allows the application to understand the state of the data at any given moment. It is one of the most important concepts when building an application with React. The different types of states each have many of their own approaches and libraries, and the tools discussed in this thesis are neither comprehensive nor definitive. It is an extensive topic, and rather than learning about every state management library available, the goal should be to understand how each solves a specific problem in a particular way.

The adoption of state management libraries has made the implementation of stateful applications straightforward and seamless. These libraries allow React developers to control the application by ensuring that the data flow between the front-end and back-end works as expected. Therefore, it is crucial to carefully consider all available options and select a state management library that is most suited for the application. This guarantees that preserving and updating the application will not be difficult for both current and future requirements.

The future of state management in React will continue to grow as more libraries are introduced and older ones are renovated. Without a doubt, all of them will continue to help reduce the amount of work developers need to handle manually. Ultimately, the most important thing to know about state management in React is to be aware of the many possibilities, consider their advantages and disadvantages, and decide on the option that best suits the developing use cases for a project.

REFERENCES

[1] Wikipedia. Single-page Application. Accessed 21.11.2022. https://en.wikipedia.org/wiki/Single-page_application

[2] Arshpreet, K. 2021. What is Single Page Application (SPA)? Pros and Cons with Examples. Accessed 21.11.2022. https://www.netsolutions.com/insights/single-page-application/

[3] Stack Overflow. 2022. 2022 Developer Survey. Accessed 22.11.2022. https://survey.stackoverflow.co/2022/

[4] Noble Desktop. 2022. What is the React Framework & Why Is It So Popular?. Accessed 22.11.2022. https://www.nobledesktop.com/classes-near-me/blog/what-is-react

[5] React Docs. Components and Props. Accessed 22.11.2022. https://reactjs.org/docs/components-and-props.html

[6] W3Docs. React Class Components. Accessed 23.11.2022. https://www.w3schools.com/react/react_class.asp

[7] Geeks for geeks. 20221. What is a pure functional component in ReactJS?. Accessed 24.11.2022. https://www.geeksforgeeks.org/what-is-a-pure-functional-component-in-reactjs/

[8] Robie, Jonathan.. What is the Document Object Model?. Accessed 24.11.2022. https://www.w3.org/TR/WD-DOM/introduction.html

[9] React Docs. Introducing JSX. Accessed 25.11.2022. https://reactjs.org/docs/introducing-jsx.html

[10] Luna Ruan. 2020. Introducing the New JSX Transform. Accessed 25.11.2022. https://reactjs.org/blog/2020/09/22/introducing-the-new-jsx-transform.html

[11] W3Schools. React State. Accessed 25.11.2022. https://www.w3schools.com/react/react_state.asp

[12] Taha Sufiyan. 2022. ReactJS State: SetState, Props and State Explained. Accessed 25.11.2022.
https://www.simplilearn.com/tutorials/reactjs-tutorial/reactjs-state

[13] Imoh, Chinedu . 2022. How to Bind Any Component to Data in React: One-Way Binding. Accessed 28.11.2022.
https://www.telerik.com/blogs/how-to-bind-any-component-data-react-one-way-binding

[14] React Docs. Hooks at a Glance. Accessed 29.11.2022.
https://reactjs.org/docs/hooks-overview.html

[15] React Docs. Building Your Own Hooks. Accessed 29.11.2022.
https://reactjs.org/docs/hooks-custom.html

[16] Npmtrends. 2022. Formik vs react-hook-form vs redux-form. Accessed 30.11.2022. https://npmtrends.com/formik-vs-react-hook-form-vs-redux-form

[17] Bigscal Technologies. 2022. Why is React Hook Form better than other forms? Accessed 7.12.2022.
https://medium.com/@Bigscal-Technologies/why-is-react-hook-form-better-than-other-forms-497054a6b2fe

[18] React Hook Form. register. Accessed 8.12.2022.
https://react-hook-form.com/api/useform/register

[19] Tanstack Query. Queries. Accessed 13.12.2022.
https://tanstack.com/query/v4/docs/guides/queries

[20] Tanstack Query. Mutations. Accessed 13.12.2022.
https://tanstack.com/query/v4/docs/guides/mutations

[21] Sebhastian, Nathan . 2020. How and Why You Should Use React Query. Accessed 13.12.2022.
https://blog.bitsrc.io/how-to-start-using-react-query-4869e3d5680d

[22] Tanstack Query. Query Client. Accessed 13.12.2022.
https://tanstack.com/query/v4/docs/reference/QueryClient

[23] React Router. Feature Overview. Accessed 14.12.2022.
https://reactrouter.com/en/main/start/overview

[24] React Router. BrowserRouter. Accessed 16.12.2022.
https://reactrouter.com/en/main/router-components/browser-router

[25] React Router. Route. Accessed 16.12.2022.
https://reactrouter.com/en/main/router-components/browser-router

[26] React Router. Routes. Accessed 16.12.2022.
https://reactrouter.com/en/main/components/routes

[27] React Router. Link. Accessed 16.12.2022.
https://reactrouter.com/en/main/components/link

[28] Redux. Getting Started with Redux. Accessed 19.12.2022.
https://redux.js.org/introduction/getting-started

[29] Redux Toolkit. Getting Started with Redux Toolkit. Accessed 20.12.2022.
https://redux-toolkit.js.org/introduction/getting-started

[30] Redux Toolkit. ConfigureStore. Accessed 21.12.2022.
https://redux-toolkit.js.org/api/configureStore

[31] React Redux. Provider. Accessed 21.12.2022.
https://react-redux.js.org/api/provider

[32] React Redux. Hooks. Accessed 21.12.2022.
https://react-redux.js.org/api/hooks

[33] Redux. Redux Essentials, Part 1: Redux Overview and Concepts. Accessed 22.12.2022. https://redux.js.org/tutorials/essentials/part-1-overview-concepts

[34] MobX. The gist of MobX. Accessed 26.12.2022.
https://mobx.js.org/the-gist-of-mobx.html

[35] MobX. Creating observable state. Accessed 27.12.2022.
https://mobx.js.org/observable-state.html

[36] React Docs. Context. Accessed 29.12.2022.
https://reactjs.org/docs/context.html

[37] Pmndrs.docs. Introduction. Accessed 02.01.2023.
https://docs.pmnd.rs/zustand/getting-started/introduction

[38] XState. Concepts. Accessed 05.01.2023.
https://xstate.js.org/docs/about/concepts.html

[39] XState. Glossary. Accessed 05.01.2023.
https://xstate.js.org/docs/about/glossary.html

[40] XState. Actors. Accessed 05.01.2023.
https://xstate.js.org/docs/guides/actors.html

[41] XState. Interpreter. Accessed 06.01.2023.
https://paka.dev/npm/xstate@4.35.2/api

[42] XState. Machines. Accessed 06.01.2023.
https://xstate.js.org/docs/guides/machines.html

[43] XState. Actions. Accessed 09.01.2023.
https://xstate.js.org/docs/guides/actions.html

[44] XState. @xstate.react. Accessed 09.01.2023.
https://xstate.js.org/docs/packages/xstate-react

[45] Recoil. Core Concepts. Accessed 10.01.2023.
https://recoiljs.org/docs/introduction/core-concepts

[46] Recoil. useSetRecoilState(state). Accessed 11.01.2023.
https://recoiljs.org/docs/api-reference/core/useSetRecoilState

[47] Jotai. Concepts. Accessed 11.01.2023.
https://jotai.org/docs/basics/concepts

[48] Tyszkiewicz, Michał. 2022. Jotai: React state management made easy.
Accessed 11.01.2023.
https://aexol.com/posts/jotai-react-state-management-made-easy

[49] Jotai. Core. Accessed 11.01.2023. https://jotai.org/docs/api/core

[50] Jotai. Atoms in atom. Accessed 12.01.2023.
https://jotai.org/docs/guides/atoms-in-atom