

Comparing State Management Between Redux And Zustand In React

LAB University of Applied Sciences

Bachelor of Business Administration, Business Information Technology

Autumn 2023

Thai Le

Abstract

Author(s)	Publication type	Completion year
Thai Le	Bachelor's thesis	2023
	Number of pages	
	47	
Title of the thesis		
Comparing State Management Between Redux And Zustand In React		
Degree, Field of Study		
Bachelor's Business Administration, Business Information Technology		
Abstract		
<p>In modern web development, the creation of reliable and effective apps in contemporary web development depends critically on maintaining the application state. React gives programmers a variety of alternatives for managing the state. Redux and Zustand are two well-known state management frameworks that provide several methods for handling application state among these choices.</p> <p>Firstly, the thesis examines the conceptual differences between Redux and Zustand, exploring the core principles and mechanisms behind each library's state management approach.</p> <p>Secondly, how Redux and Zustand work are presented and then the comparison is conducted based on a "To Do List" application to show the result</p> <p>Ultimately, this thesis contributes to the understanding of Redux and Zustand, assisting developers in making informed decisions regarding state management in JavaScript applications. The research offers potential directions for future improvements and developments in both libraries, further advancing the field of state management</p>		
Keywords		
Redux, Zustand, State management		

Contents

1	Introduction.....	1
1.1	Introduction	1
1.2	Objectives	2
1.3	Research Question	2
1.4	Limitations.....	2
2	Research Methodology and Data Collection	4
3	Theoretical Framework.....	5
3.1	State Management.....	5
3.2	Redux.....	6
3.3	Zustand	7
4	Development Environment	8
4.1	Figma	8
4.2	Visual Studio Code	8
4.3	Chrome Dev Tool.....	8
4.4	Redux DevTool	9
4.5	Zookeeper DevTool	10
5	Implementation of test project	11
5.1	Project idea and discussion	11
5.2	Redux implementation	12
5.2.1	Create Redux Store	12
5.2.2	Wrap the rootComponent.....	13
5.2.3	Build up actions in reducer.....	14
5.2.4	Import useSelector and useDispatch to the child components	15
5.2.5	Use the value of state at consumer components.....	16
5.3	Zustand implementation.....	20
5.3.1	Create a store file.....	20
5.3.2	Build up actions in store file	21
5.3.3	Use the value of state at consumer components.....	24
6	Differences between Redux and Zustand	30
6.1	Implementation	30
6.2	Tracking the state changes.....	31
6.2.1	Redux.....	31
6.2.2	Zustand	35
6.3	Processing speed	37

6.4	Scalability	40
6.5	Project size and code length.....	40
6.5.1	Project size	40
6.5.2	Code Length	41
6.6	Result.....	42
7	Conclusion.....	44
	References	45

1 Introduction

1.1 Introduction

Many web technologies have been developed in the contemporary period to make it easier for web developers to create user interfaces, with React being the most popular JavaScript library (Stephen 2021).

In web development, state management is an important aspect that directly affects the user experience and overall application performance. Since web applications have become more intricate, it takes more work to maintain a state with well-organised and more efficient. In order to solve this problem, many state management libraries have emerged, each offering its own unique approach and feature set. Among these libraries, Redux and Zustand have gained considerable popularity in the JavaScript ecosystem.

Redux is a Predictable State Management tool for applications running in the JavaScript programming language. Redux was born to create a layer that manages the overall state of the application. Redux helps the applications you write to work consistently and can work in different environments such as client, server, and native. In addition, using the Redux library for storage also simplifies the testing and testing process (Redux 2023).

On the other hand, Zustand is also appreciated by developers for its ease of use and simplicity. Zustand works by using hooks that are tree connections to state and functional programming principles to provide a lightweight state management solution.

The goal of this thesis is to conduct a comprehensive comparison analysis of state management between Redux and Zustand. In addition, the basic principles of each library are also described as well as how each works.

1.2 Objectives

This thesis delves into a comprehensive comparison between Redux and Zustand, focusing on their implementation in the context of a "To-Do List" application. To achieve this comparison, it is imperative to first gain a deep understanding of these libraries, encompassing their core concepts and fundamental aspects related to state management, including actions, reducers, stores, and hooks. Subsequently, a developer-centric assessment is conducted, the goal is to identify the strengths and weaknesses of each library in terms of usability and efficiency. Furthermore, the research endeavours to analyse and quantify the performance characteristics of Redux and Zustand, employing benchmarks to compare factors such as memory usage, re-render frequency, and overall runtime performance, providing valuable insights into their practical implications.

1.3 Research Question

The primary research question is: What are the differences between Redux and Zustand based on comparison findings?

Based on the research question, the study is going to show which is better and in what situations it should be used, helping developers make better decisions.

1.4 Limitations

This study is limited by:

Even though Redux and Zustand are well-known options for JavaScript state management, it's crucial to recognize that not all of the solutions in the JavaScript ecosystem are covered in this study. Moreover, the assessment is predicated on a solitary, rather uncomplicated sample implementation, which can produce disparate outcomes when employed in more intricate and expansive undertakings.

Furthermore, it's critical to understand that performance characteristics can have a wide range of effects based on the complexity and size of the program as well as the particular usage patterns that are used. Because of this, the research's conclusions might not apply to every possible situation, which emphasizes the importance of carefully evaluating the environment in which these libraries are used.

Another limitation is related to the dynamic nature of the technology. The experiment was conducted using Redux version 8.1.2 and Zustand version 4.4.1. Both Redux and Zustand are actively maintained and updated, which means that new features, optimizations, or improvements may have been introduced since this study was completed. As a result, the comparison may not reflect the latest advancements in either library.

Despite these limitations, this thesis provides valuable insights into the comparison of state management between Redux and Zustand. Future research can build upon these findings, considering additional libraries and exploring the evolving landscape of state management in applications using JavaScript language.

2 Research Methodology and Data Collection

In this study, the research method used is the comparative method. This method is going to compare the differences and similarities between Redux and Zustand such as performance, tracking state changes, implementation, scalability, and software engineer's point of view.

Data collection is taken from the support of the Chrome Dev tool through application performance that has been implemented by Redux and Zustand

3 Theoretical Framework

3.1 State Management

Any data that characterizes an application's current behavior is referred to as its state. State management involves overseeing changes in an application's state over time, encompassing tasks like storing the initial value, reading the current value, and updating it. (Common Tools & Practices). In React, managing the state is crucial for controlling an application's evolving data. The intricate process of passing data through the component tree entails governing the application's state, which comprises information about both its intended behavior and existing state. This encompasses diverse data like user information, application settings, and component details.

According to Teimur (2022), to evaluate a well-managed library should meet the following criteria:

- **Usability:** one of the most important factors that affects developer output, code quality, and project success. Since it enhances the entire development experience, streamlines the procedure, reduces the likelihood of errors, and improves the development experience overall, it is essential to choose the appropriate state management system for a project.
- **Maintainability:** A usable library makes it easier to maintain code over time. The ability to review and edit code is made simpler for developers, which is essential as applications develop and expand.
- **Performance:** an essential component of state management libraries that influences the user experience, resource efficiency, and long-term survival of applications
- **Testability:** That assists in ensuring the reliability and strength of the state management logic.
- **Scalability:** Both small and large applications should be handled well in terms of state management, making sure that it is able to expand stably as the application gets bigger.
- **Modifiability:** A must-have feature in projects, especially large projects, it helps developers easily find errors without wasting too much time.
- **Reusability:** By utilising pre-existing technologies, developers may create solid, scalable apps with less time and effort.
- **Ecosystem:** It increases the functionality of the library and offers developers a wealth of resources and tools.
- **Community:** a supportive group that can aid with queries and problem-solving.

3.2 Redux

Redux serves as a foreseeable state container designed for JavaScript applications, particularly because it is able to be used with React specifically and with other libraries more broadly (Soham 2022). Redux is a predictable state container for JavaScript applications and it can be used with React in particular and other libraries in general. If applications are developed with the help of Redux, it will make it easier for developers to detect errors and make the application run smoothly in different environments.

Redux's operating model includes 3 stages: Action, Reducers, and Store

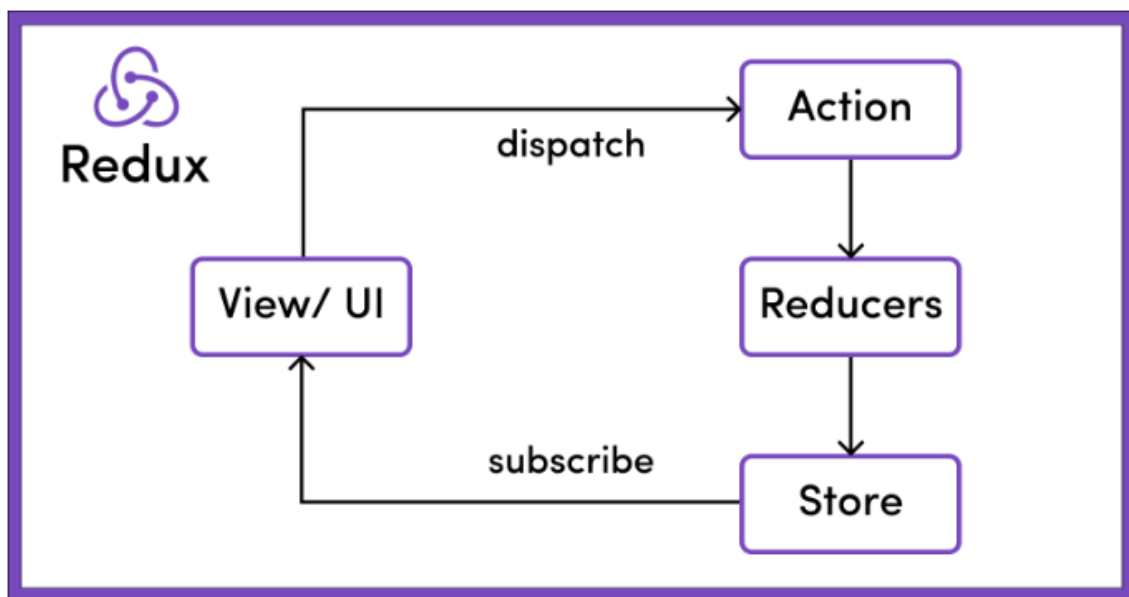


Figure 1. Redux's workflow (Nikhil 2023)

Action dispatch: At the beginning, after the user interacts with the application, actions are dispatched by using the “dispatch()” function. These actions are called JavaScript objects that describe events or interactions (Soham 2022).

Reducer handling: After receiving the dispatched actions, the reducer is responsible for changing the state in the application based on the initial or most recent state, and the received actions are treated as a pure function.

Store: When the reducer completes handling the action, the new state is returned from the reducer and becomes the updated state in the store.

3.3 Zustand

A complicated user interface is what most developers focus on. Component trees have several levels, their roots and leaves are separated by a great distance, which causes bloated code and makes extending their interfaces more challenging. Numerous state management libraries, notably Zustand, were created as a result of this issue.

Zustand has a very flexible and basic framework. According to its intended usage, a store could contain not only the state but also functions, constants, and objects. When components call a specific store, they need to set the appropriate state through Hooks. A special feature of Zustand is that it is still able to provide the state to components without requiring them to be wrapped in a Provider.(Abhishek 2023)

As Uma (2022) points out, with hooks like `useState` and `useStore` used to handle state instead of boilerplate code, Zustand is renowned for its simplicity. Developers will find Zustand to be a helpful solution because it offers simple access and status updates.

4 Development Environment

4.1 Figma

Figma is a popular interface design application today, application or website designers are strongly supported by the necessary tools and the included plugins. Besides that, implementing code becomes easier and faster as developers use Figma's test mode to CSS style and measure from design (Andrei 2020). In the process of designing the To Do List application on Figma, the UI objects are grouped by the author into components to build more profitable React components.

4.2 Visual Studio Code

Visual Studio Code, developed by Microsoft, is a robust and versatile source code editor. It seamlessly combines the functionality of an Integrated Development Environment (IDE) with that of a code editor. This software is available for free and is compatible with macOS, Windows, and Linux operating systems. Visual Studio Code offers an array of powerful features tailored to enhance the programming experience. Some of its notable features include debugging tools, Git integration, syntax highlighting, automated syntax completion, code snippets, a variety of themes, and customizable keyboard shortcuts. (Visual Studio Code)

One of the key strengths of Visual Studio Code is its support for multiple programming languages. This wide language support makes it a preferred choice for developers working on diverse projects. Therefore, this code editor had been used for writing code for the Todo list app.

4.3 Chrome Dev Tool

Google Chrome is a free web browser developed and operated by Google (Elise 2022). The Google Chrome web browser provides an indispensable set of utilities for web developers known collectively as Chrome DevTools that play an important role in web and application development such as debugging layout errors, locale, identify JavaScript errors, monitor various metadata, etc. The To Do List application was built based on using the following tools:

- Element tab: This is a powerful tool for inspecting and manipulating the HTML and CSS of a web page.
- Console tab: is a versatile tool for JavaScript development, offering capabilities for interactive coding, debugging, error reporting, logging, and performance profiling. It is an essential component for web developers, enabling them to inspect and

manipulate the JavaScript environment of a web page for troubleshooting and development tasks.

- Performance tab: Website or application performance is analysed as CPU usage, rendering time, memory consumption, etc, by recording the process of the application

4.4 Redux DevTool

Redux DevTools is a collection of browser extensions that allow developers to check the status of the state and its changes based on every single time.

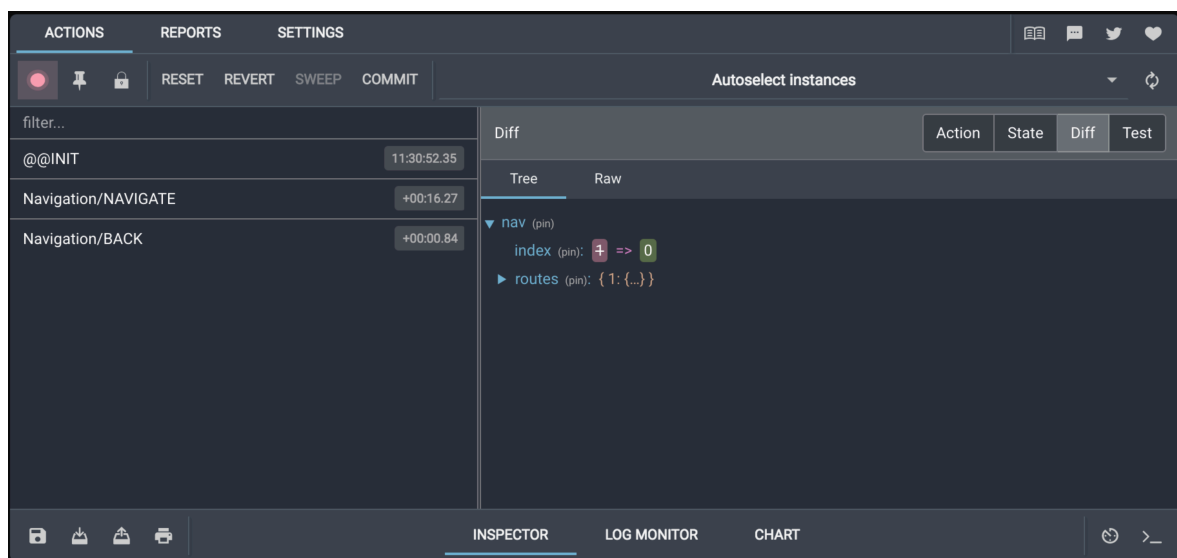


Figure 2. Redux DevTool workplace (github)

The working area of redux devtool is divided into 2 main areas. The left panel shows a list of actions used by users interacting with the application. However, the right panel shows the results of the actions in detail by tabs such as action, state, diff, trace, and test.

- Action tab: displays a log of all actions dispatched in the app. It includes detailed information about each action, such as action type and payload.
- State tab: shows the current state of the Redux store. In addition, it provides a snapshot of the complete state for exploring the data.
- Diff tab: shows the difference between two consecutive states, which helps developers monitor how the state evolves over time.
- Trace tab: is used to provide tracking information where the action has been called.

- Test tab: creates a test template in some pre-provided testing framework. The previous state is taken and a written test is provided on what the current state should be given the dispatched action

4.5 Zukeeper DevTool

Zukeeper is a set of extensions for Zustand. With a simple design, Zukeeper has two main panels that provide essential features for developers. The list of actions is placed on the left side, but the right panel contains the state tab, dif tab, and tree tab to monitor changes in state in each period.



Figure 3. Zukeeper DevTool workplace (Zukeeper)

5 Implementation of test project

5.1 Project idea

In order to compare two state management between Zustand and Redux, the To Do List application was conducted that can apply the main features of state which are adding, editing, and deleting.

Briefly talking about the idea of the project, the To Do List application has only one page and it is divided into 2 separate main components. The left component is a form for users to enter values and buttons to perform the actions they desire. The element on the right is used to represent a to-do list.

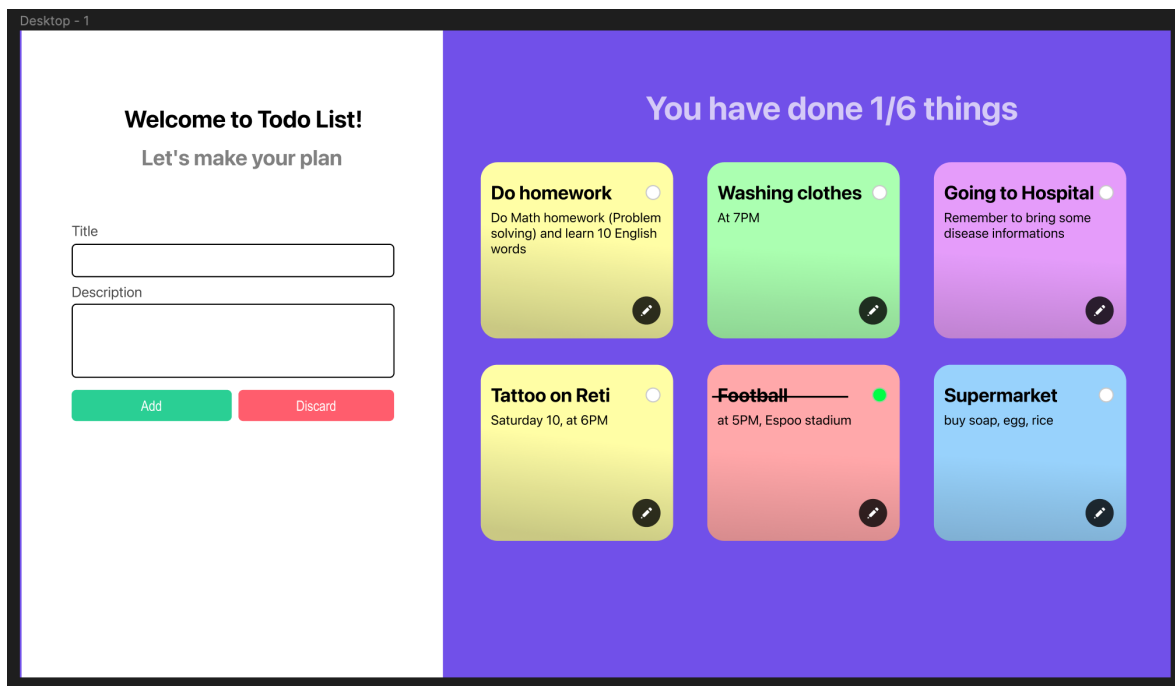


Figure 4. The To Do List application is designed in Figma

Based on the To Do List application design, there are 4 issues that need to be solved.

First of all, When the user enters values in the Title input and Description input and then clicks the Add button, the store on the state has to recognize a newly added object and also display those values in the todo cart in the right area of the application.

Moving on to the second issue, if a specific todo card needs to be edited, the user selects the edit button in the below right corner of each card, and then the values of that card need

to be displayed on the left form. At this point, the Add and Discard buttons are changed to the new Update and Delete features

For the third issue, after getting the object that needs to be edited on the Store, if the user changes the content and clicks update, the selected todo card needs to be updated with new content. However, if the user clicks delete, that card will be removed from the list.

Finally, the number of Todo cards and the number of completed Todo cards need to be shown in the title.

5.2 Redux implementation

The implementation process using Redux Hooks includes 5 steps.

5.2.1 Create Redux Store

This stage involves calling the createStore function.

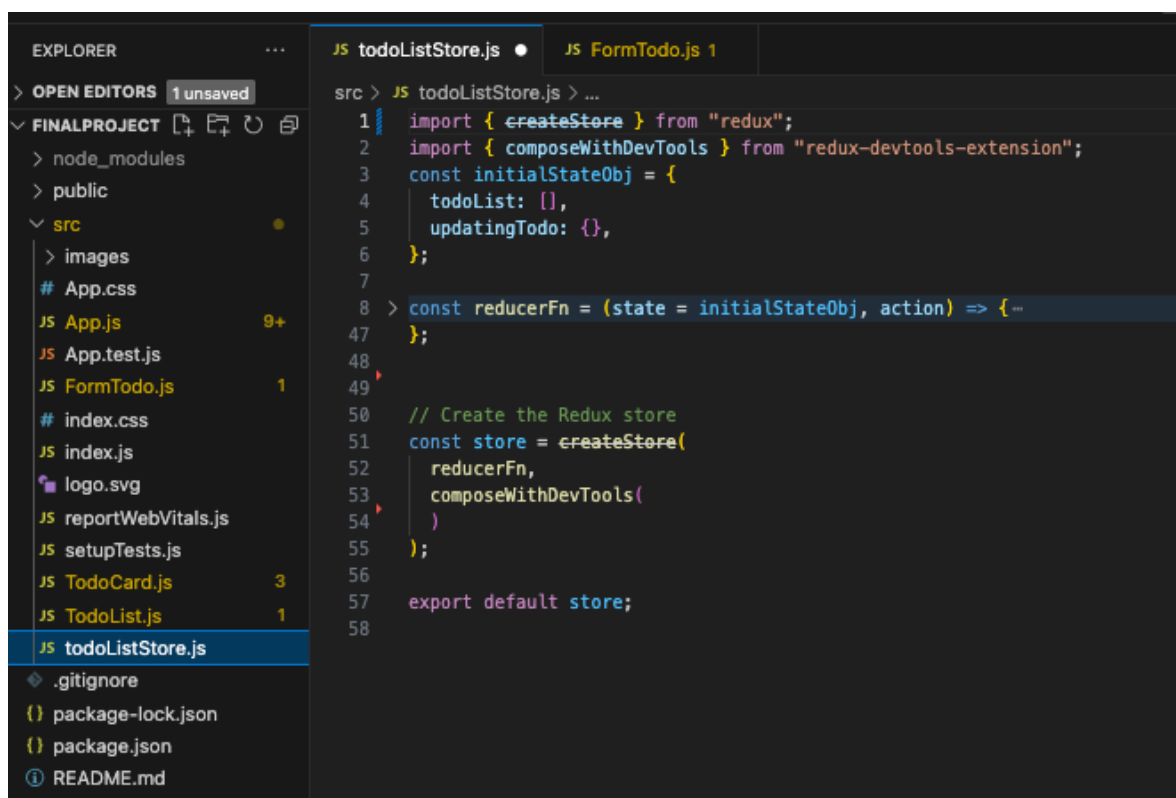


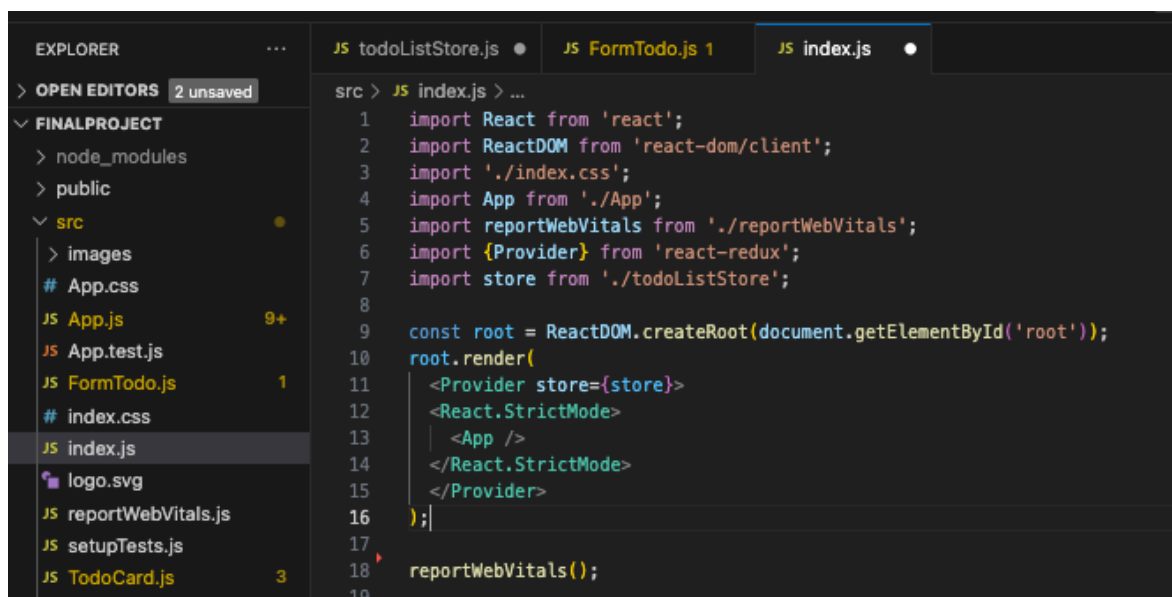
Figure 5. Redux store in the To Do List application

In the To Do List app, the store is set up through a few steps as shown in Figure 5.

- Declare the initial state variable: it is essential to establish the initial state of the application before any actions are dispatched. In picture 5, there are 2 prototypes in the `initialStateObj` object: `todoList[]` and `updatingTodo{}`
- Create Reducer: as mentioned above, the main responsibility of the reducer is to manage and update the state of the application in response to dispatched actions. In the To Do List app, the reducer needs two input values: `initialStateObj` and `action`
- Create Store: It serves as a key component of the state management of the Redux-based application. As shown in Figure 5, two inputs are given to store: `reducerFn` and `composeWithDevTools`. With `composeWithDevTools`, it allows integration with the Redux DevTools extension, this function is commonly used in the development environment.

5.2.2 Wrap the rootComponent

It's crucial to create a connection between a React app and the Redux store once the Redux setup is complete, as seen in figure 6



```

src > JS index.js > ...
1  import React from 'react';
2  import ReactDOM from 'react-dom/client';
3  import './index.css';
4  import App from './App';
5  import reportWebVitals from './reportWebVitals';
6  import {Provider} from 'react-redux';
7  import store from './todoListStore';
8
9  const root = ReactDOM.createRoot(document.getElementById('root'));
10 root.render(
11   <Provider store={store}>
12     <React.StrictMode>
13       <App />
14     </React.StrictMode>
15   </Provider>
16 );
17
18 reportWebVitals();
19

```

Figure 6. Wrap the Redux store into a React app in the To Do List application

As shown in the illustration, in order to use the Redux store from a React project, the `Provider` component from the `react-redux` library and the store from the Redux store must both be imported.

Currently, every child component of the app can call any action to update the store or access the store to get the most recent state information.

5.2.3 Build up actions in reducer

For the reducer function, it needs two arguments: the current state and action. Inside a reducer function in Redux, how the state should change in response to different actions. The switch statement in the reducer function normally evaluates a type property of the action and modifies the state accordingly.

```

8  const reducerFn = (state = initialStateObj, action) => {
9    switch (action.type) {
10     case "ADD_TODO":
11       const newTodo = action.payload;
12       const newTodoList = [...state.todoList, newTodo];
13       return { ...state, todoList: newTodoList };
14
15     case "UPDATE_TODO":
16       const updateTodo = state.todoList.map((todo) => {
17         if (todo.id === action.payload.id) {
18           return { ...action.payload, completed: todo.completed };
19         }
20         return todo;
21       });
22       return { ...state, todoList: updateTodo, updatingTodo: {} };
23
24     case "SET_UPDATING_TODO":
25       const todo = state.todoList.find((todo) => todo.id === action.payload);
26       return { ...state, updatingTodo: todo };
27
28     case "UPDATE_TODO_COMPLETED":
29       const updateTodoCompleted = state.todoList.map((todo) => {
30         if (todo.id === action.payload.id) {
31           return { ...todo, completed: action.payload.value };
32         }
33         return todo;
34       });
35       return { ...state, todoList: updateTodoCompleted };
36
37     case "REMOVE_TODO":
38       const updatedTodos = state.todoList.filter(
39         (todo) => todo.id !== action.payload
40       );
41       return { ...state, todoList: updatedTodos, updatingTodo: {} };
42
43     default:
44       break;
45   }
46   return state;
47 };

```

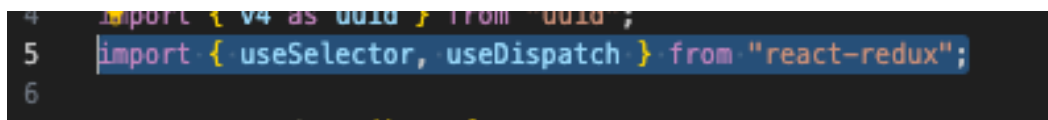
Figure 7. Build reducers to handle Redux actions in the To Do List application

As described in figure 7, there are 5 actions as following set up for the app:

- In the "ADD_TODO" action, to add a todo card to the todoList array with the title and description values that the user just entered. It will create a todo card with these 2 values and also add an ID to it. Then, this new Todo card will be dropped into the current array to create a new todoList array.
- In the "SET_UPDATING_TODO" action, to get a todo object that the user selects in the todoList array to attach to UpdateTodo in the store based on filtering out the corresponding ID in the todoList array through the map() function and then putting it into UpdateTodo.
- In the "UPDATE_TODO" action, that updates the todo that the user wants to edit: title and description. It checks the ID of each todo card in the todoList array to compare it with the ID of the selected todo card. If the IDs match, the todo card values in the todoList array are assigned new values
- In the "UPDATE_TODO_COMPLETED" action, it is similar to the "UPDATE_TODO" action, however, only the "completed" value is altered. With the value "completed", it will have two values "true" and "false" representing whether the todo card was completed or not.
- In the "REMOVE_TODO" action, it deletes selected todo cards based on their respective IDs. The filter() function is used to filter out unselected todo cards.

5.2.4 Import useSelector and useDispatch to the child components

To be able to interact with the Redux store, components need to have the command as figure 8 shown: `import { useSelector, useDispatch } from "react-redux";`.



```

4 import { v4 as uuid } from 'uuid';
5 import { useSelector, useDispatch } from 'react-redux';
6

```

Figure 8. Import useSelector and usDispatch in the To Do List application

- The "useSelector" hook allows components to select from and access data in the Redux store. Using the useSelector function, a certain state object can be retrieved from the Redux store and made accessible inside a component. The argument can be a selector function that specifies the area of the state to retrieve.

- The “useDispatch” hook enables elements to send actions to the Redux store. The Redux store's state may be changed via actions. The dispatch function is accessed and utilised to launch events inside components by using useDispatch.

In the To Do List app, there are three files that use this command: FormTodo, TodoCard, and TodoList.

5.2.5 Use the value of state at consumer components

After completing the import of useSelector and useDispatch into all components, this step will delve into how the functions in each component are handled.

Pertaining to the FormTodo component, the users will interact with this component for three purposes: add, edit, and delete.

```

7  const FormTodo = () => {
8    const updatingTodo = useSelector((state) => state.updatingTodo);
9
10   const dispatch = useDispatch();
11
12   const [title, setTitle] = useState("");
13   const [description, setDescription] = useState("");
14
15   const handleSubmit = (e) => {
16     e.preventDefault();
17
18     if (title === "" && description === "") {
19       alert("PLEASE ENTER TITLE AND DESCRIPTION");
20     } else {
21       if (updatingTodo.id) {
22         dispatch({
23           type: "UPDATE_TODO",
24           payload: { id: updatingTodo.id, title, description },
25         });
26       } else {
27         const id = uuid();
28         const todoObj = { id, title, description };
29         dispatch({ type: "ADD_TODO", payload: todoObj });
30       }
31       setTitle("");
32       setDescription("");
33     }
34   };
35
36   const handleDiscard = () => {
37     if (updatingTodo.id) {
38       dispatch({ type: "REMOVE_TODO", payload: updatingTodo.id });
39     }
40     setTitle("");
41     setDescription("");
42   };
43   console.log("updating todo", updatingTodo);
44   useEffect(() => {
45     if (updatingTodo.id) {
46       setTitle(updatingTodo.title);
47       setDescription(updatingTodo.description);
48     }
49     [updatingTodo.id];
50   }, [updatingTodo.id]);

```

Figure 9. Create functions in the FormTodo component in the To Do List application

As shown in Figure 9, first connecting to the Redux store, the React components need to use the "useSelector" and "useDispatch" hooks.

- `useSelector`: the `updatingTodo` property is taken out of the Redux store's state.
- `useDispatch`: Actions are sent to the Redux store using this dispatch function, which causes the application's state to alter as a result.

In the `FormTodo` component, it needs to initialise the two variables `title` and `description` with an empty initial value.

```
const [title, setTitle] = useState("");
const [description, setDescription] = useState("");
```

Where `"title"` and `"description"` represent the current value of the state variable and `"setTitle"` and `"setDescription"` are functions used to update the value.

There are two handling functions in the `FormTodo` component: `handleSubmit` and `handleDiscard`. (Figure 11)

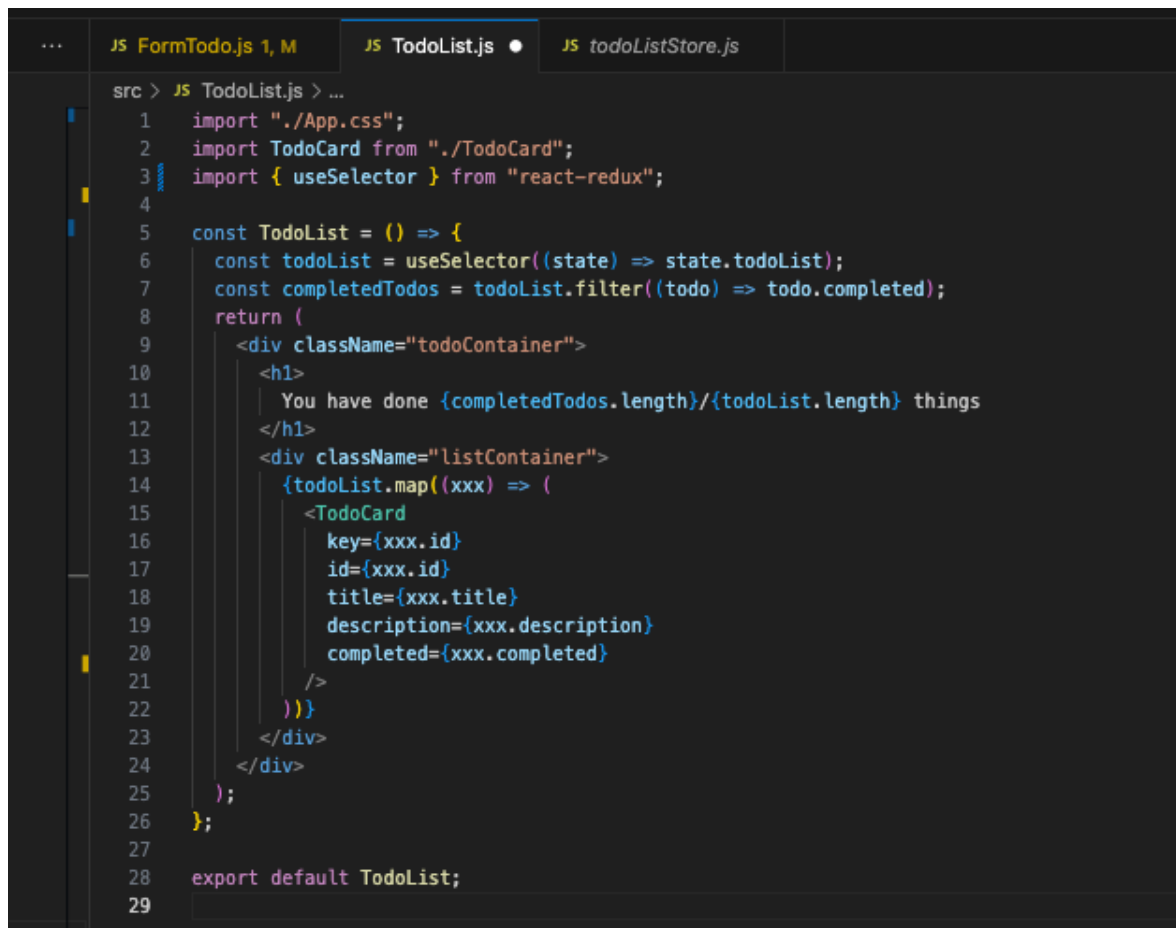
Regarding the `handleSubmit()` function, it has two cases:

- `"if (title === "" && description === "") { ... }"`: This condition checks whether both `title` and `description` are empty strings. If they are, it displays an alert asking the user to enter a title and description.
- `"if (updatingTodo.id) { ... } else { ... }"`: This condition checks whether `updatingTodo.id` exists or not. If it does, it means that the user is in an update mode (editing an existing todo), and it dispatches a `"UPDATE_TODO"` action with the properties in the payload such as `id`, `title`, and `description`. On the other hand, if `updatingTodo.id` doesn't exist, it means the user is adding a new todo, and it dispatches an `"ADD_TODO"` action with a new todo object.

After dispatching the action and handling the form submission, `"setTitle("")"` and `"setDescription("")"` need to reset the state variables to the empty strings. The purpose is to clean up the form fields for the next input.

Secondly, the `handleDiscard()` function dispatches an action with the type `"REMOVE_TODO"` and a payload containing the `id` of the `updatingTodo` to indicate that a todo item should be removed if there is an `updatingTodo.id` existing. Otherwise, the `setTitle` and `setDescription` are set to empty.

Moving forward to the TodoList component, this component completely retrieves data from the store to represent the todo cards as well as the number of completed todo cards as shown in the below picture.



```

src > JS TodoList.js > ...
1  import "../App.css";
2  import TodoCard from "../TodoCard";
3  import { useSelector } from "react-redux";
4
5  const TodoList = () => {
6    const todoList = useSelector((state) => state.todoList);
7    const completedTodos = todoList.filter((todo) => todo.completed);
8    return (
9      <div className="todoContainer">
10        <h1>
11          You have done {completedTodos.length}/{todoList.length} things
12        </h1>
13        <div className="listContainer">
14          {todoList.map((xxx) => (
15            <TodoCard
16              key={xxx.id}
17              id={xxx.id}
18              title={xxx.title}
19              description={xxx.description}
20              completed={xxx.completed}
21            />
22          ))}
23        </div>
24      </div>
25    );
26  };
27
28  export default TodoList;
29

```

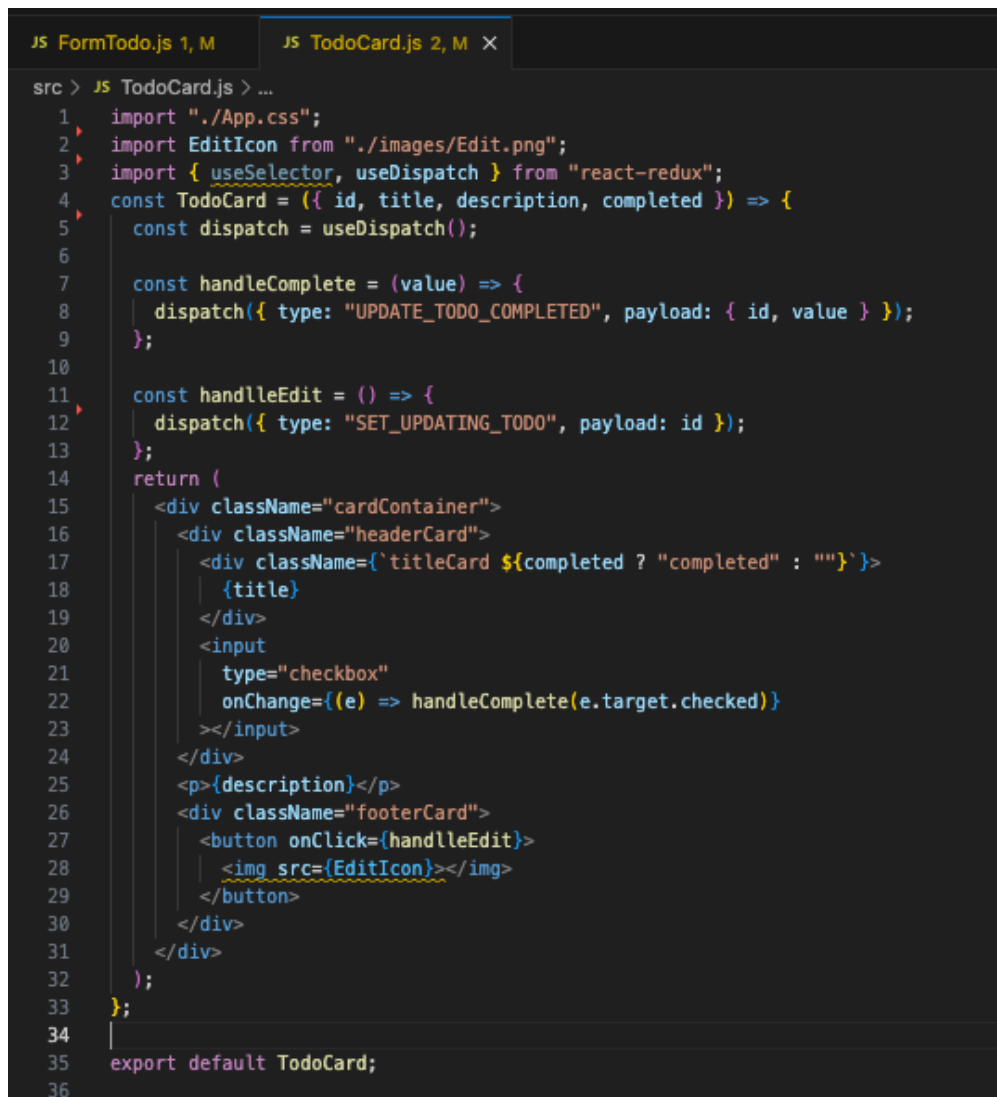
Figure 10. TodoList Component in the To Do List application

As shown in Figure 10, normally, when connecting to a Redux store, useSelector and useDispatch will come together in the import statement. However, in this instance, this component only obtains data for display rather than doing any dispatch actions. There are two main tasks in this component: showing the number of todo cards and completed todo cards and rendering each todo card from the todoList array.

- The todoList likely represents an array of the todo cards and it filters todo cards that have a "completed" property into a new array called "completedTodos". After determining the todoList and completedTodos arrays, rendering the number of elements in these two arrays in HTML code is done by attaching the ".length" statement as shown in Figure 10.

- The `map()` function is used in HTML code to retrieve all of the todo cards in the `todoList` array in a loop while also retrieving the property values for each todo card.

Next to the last component in the application, the `TodoCard` component has two tasks to perform: attach input values to the HTML code area and dispatch 2 actions for handling completed and edit.



```

src > JS TodoCard.js > ...
1  import "../App.css";
2  import EditIcon from "../images/Edit.png";
3  import { useSelector, useDispatch } from "react-redux";
4  const TodoCard = ({ id, title, description, completed }) => {
5      const dispatch = useDispatch();
6
7      const handleComplete = (value) => {
8          dispatch({ type: "UPDATE_TODO_COMPLETED", payload: { id, value } });
9      };
10
11     const handleEdit = () => {
12         dispatch({ type: "SET_UPDATING_TODO", payload: id });
13     };
14     return (
15         <div className="cardContainer">
16             <div className="headerCard">
17                 <div className={`titleCard ${completed ? "completed" : ""}`}>
18                     {title}
19                 </div>
20                 <input
21                     type="checkbox"
22                     onChange={(e) => handleComplete(e.target.checked)}
23                 ></input>
24             </div>
25             <p>{description}</p>
26             <div className="footerCard">
27                 <button onClick={handleEdit}>
28                     <img src={EditIcon}></img>
29                 </button>
30             </div>
31         </div>
32     );
33 };
34
35 export default TodoCard;
36

```

Figure 11. `TodoCard` Component in the To Do List application

As shown in Figure 11, the input values are passed directly to the HTML code such as `title` and `description`. In this component, there are two handling functions: `handleComplete` and `handleEdit`.

- `handleComplete`: It only accepts one input, "value", which is required to be a boolean value indicating whether or not a to-do card has been completed. It sends a "UPDATE_TODO_COMPLETED" action with a "id" and "value" object payload. In addition, the unique identity of the to-do item is probably represented by its ID, and its value shows whether it has been finished (true) or not (false).
- `handleEdit`: it doesn't take any arguments. In addition, a "SET_UPDATING_TODO" action is delivered with the id in its payload. The `updatingTodo` property, which frequently indicates that the user is in edit mode for a particular todo card, is set in the Redux store using this action. The todo card that is being modified is identified by its ID.

5.3 Zustand implementation

There are 3 steps to use Zustand in the project.

5.3.1 Create a store file

In the project directory, create a new JavaScript file named `store.js`. In the Todo list application, the store file was named with `todoListStore.js`.

Based on figure 12, the basic structure of a file store includes three things that need to be done:

- `import {create} from "zustand"`: is to import the create function from the Zustand library. This function is an important part of Zustand and is used to create a store for managing application state.
- `const useTodos = create((set) => {})`: is to contain actions that modify the state such as `addTodo`, `setUpdatingTodo`, `updateTodo`, `updateTodoCompleted`, and `removeTodo`.
- `export default useTodos`: is to make the "useTodos" store available for use in other parts of the application

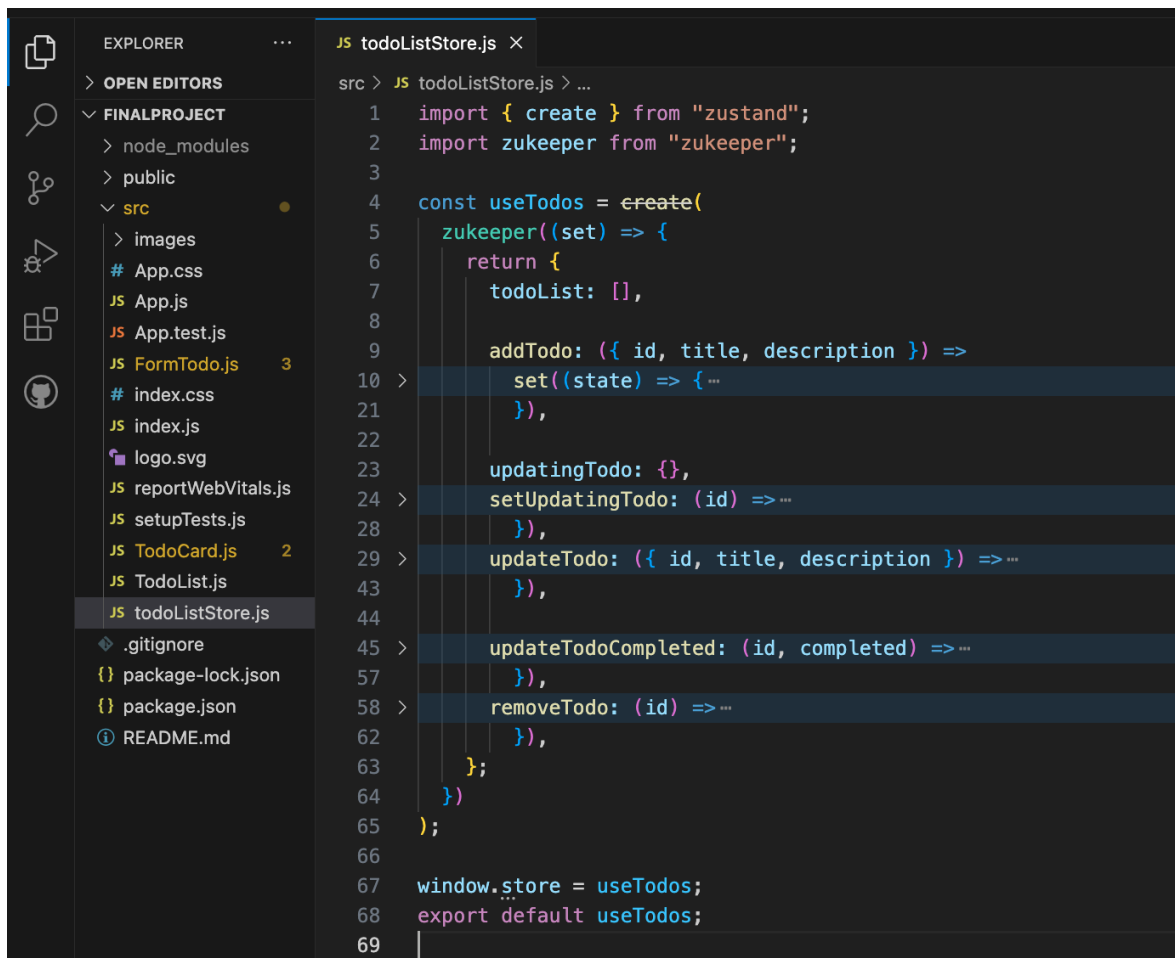


Figure 12. create store for application in the To Do List application

5.3.2 Build up actions in store file

When creating an action in store, the basic structure of that action is as follows:

```

property: ({input value}) =>

    set((state) => {});

```

In the Todo list application, first, an array was created named todoList that is responsible for storing todo lists. Then addTodo action was executed and it was passed with three properties id, title, and description as figure 13 shows.

```

7      todoList: [],
8
9      addTodo: ({ id, title, description }) =>
10         set((state) => {
11             const newTodo = {
12                 id: id,
13                 title: title,
14                 description: description,
15                 completed: false,
16             };
17
18             const newTodoList = [...state.todoList, newTodo];
19
20             return { ...state, todoList: newTodoList };
21         }),

```

Figure 13. addTodo action in the To Do List application

There are three stages to perform based on figure 13:

- Create a new todo object with the provided ID, title, and description.
- Create a new todo list by copying the existing list and adding the new todo.
- Return a new state object that includes the updated todo list.

Moving on to the next actions, if the user wants to edit the information of any todo card, that todo card needs to be identified based on its ID.

```

21      ...
22      ...
23      updatingTodo: {},
24      setUpdatingTodo: (id) =>
25         set((state) => {
26             const todo = state.todoList.find((todo) => todo.id === id);
27             return { ...state, updatingTodo: todo };
28         }),
29      updateTodo: ({ id, title, description }) =>
30         set((state) => {
31             const updatedTodos = state.todoList.map((todo) => {
32                 if (todo.id === id) {
33                     return {
34                         id,
35                         title,
36                         description,
37                         completed: todo.completed,
38                     };
39                 }
40                 return todo;
41             });
42             return { ...state, todoList: updatedTodos, updatingTodo: {} };
43         }),
44      ...

```

Figure 14. Creating actions for updating feature in the To Do List application

To perform the update feature for the application, there are main steps required (Figure 14):

1. Create an empty object named `updatingTodo`, that is used as a place to hold which todo card needs to be identified for update.
2. Create a “`setUpdatingTodo`” action to filter out todo cards with corresponding IDs in `todoList` array and return this todo card to the `updatingTodo` object
3. Create an `updateTodo` action with ID, title, and description as input values. In this action, it will go into the `todoList` array to find which todo card corresponds to the input ID before returning the new values.

Transitioning to creating `updateTodoCompleted` action, similar to `updateTodo` action, it also needs two input values: `id` and `completed`. In this case, instead of listing the properties that need to remain the same, they can be abbreviated with `...todo` and just write the property whose value needs to be changed as figure 15.

```

46      updateTodoCompleted: (id, completed) =>
47      set((state) => {
48        const updatedTodos = state.todoList.map((todo) => {
49          if (todo.id === id) {
50            return {
51              ...todo,
52              completed,
53            };
54          }
55          return todo;
56        });
57        return { ...state, todoList: updatedTodos };
58      })

```

Figure 15. Creating `updateTodoCompleted` actions in the To Do List application

Finally, when the user wants to delete any of todo card, `removeTodo` action conducts going to todo list to find the ID of the corresponding todo card. To delete an element in an array, the `filter()` function was used to filter out elements that match the condition (Figure 16).

```

59     removeTodo: (id) =>
60     {
61       set((state) => {
62         const updatedTodos = state.todoList.filter((todo) => todo.id !== id);
63         return { ...state, todoList: updatedTodos, updatingTodo: {} };
64       });
65     }
66   };

```

Figure 16. Creating removeTodo action in the To Do List application

5.3.3 Use the value of state at consumer components

There are three consumer component files linked with the store: FormTodo, TodoCard, and TodoList.

Regarding FormTodo file, as mentioned in step 2, to interact with the store, the store needs to be exported and other files also need to be imported as links by the command `import useTodos` from `\"./todoListStore\"`;

```

src > JS FormTodo.js > ...
1  import \"./App.css\";
2  import useTodos from \"./todoListStore\";
3  import React, { useEffect, useState } from \"react\";
4  import { v4 as uuid } from \"uuid\";
5

```

Figure 17. Importing Store in the To Do List application

The command in Figure 17 is called a custom hook to allow programmers to access and manage the state in the application.

Inside the component, there are several variables that need to be initialised that contain references to functions from the store. Besides that, there are local variables used to get values before pushing them to the state as Figure 18 shows below.

```

5
6  const FormTodo = () => {
7
8      //Use the custom hook to get the functions from store
9      const addTodo = useTodos((state) => state.addTodo);
10     const removeTodo = useTodos((state) => state.removeTodo);
11     const updateTodo = useTodos((state) => state.updateTodo);
12     const updatingTodo = useTodos((state) => state.updatingTodo);
13
14     //// Create a local state
15     const [title, setTitle] = useState("");
16     const [description, setDescription] = useState("");
17

```

Figure 18. Create variables for using the custom hook and local state in the To Do List application

After initialising the variables, the handleSubmit and handleDiscard functions are created. For the handleSubmit function, there are two main cases: the user submits after filling in the title and description and the user submits without filling anything. (Figure 19)

```

14  const handleSubmit = (e) => {
15      e.preventDefault();
16
17      if(title=="" && description=="")
18      {
19          alert("PLEASE ENTER TITLE AND DESCRIPTION")
20      }
21      else{
22          if(updatingTodo.id)
23          {
24              updateTodo({id:updatingTodo.id, title, description})
25          }
26          else{
27              const id = uuid()
28              addTodo({id, title, description})
29          }
30          setTitle('');
31          setDescription('');
32      }
33  }

```

Figure 19. handleSubmit function in the To Do List application

- The user submits without filling: the `alert()` function is used to notify the user that they have not entered values in the title and description

- The user submits after filling in the title and description: At this point, there are two cases: add and update. If the handleSubmit function recognizes that updatingTodo has changed, it will know that the user wants to update, otherwise, it performs the addTodo action

Moving forward to the handleDiscard function, the task of this function is to delete the value in both title and description when the user wants to quickly delete it. However, this function will delete a todoCard in the todoList if updatingTodo receives a todoCard id that matches the id of any element in the todoList array. (Figure 20 below)

```

37
38     const handleDiscard = () => {
39         if (updatingTodo.id) {
40             removeTodo(updatingTodo.id)
41         }
42         setTitle('');
43         setDescription('');
44     };
45     console.log("updating todo", updatingTodo);

```

Figure 20. handleDiscard function in the To Do List application

Also, the useEffect() function was created to automatically populate the title and description input fields in components with the values from the updatingTodo object when the updatingTodo.id property changes as figure 21.

```

38     console.log("updating todo", updatingTodo);
39     useEffect(() => {
40         if (updatingTodo.id) {
41             setTitle(updatingTodo.title);
42             setDescription(updatingTodo.description);
43         }
44     }, [updatingTodo.id]);
45
46     return (

```

Figure 21. useEffect function in the To Do List application

The return statement contains the HTML code for rendering the form, including input fields, buttons, and labels shown in figure 22.

```

46   return (
47     <div className="formContainer">
48       <h1>Welcome to Todo list!</h1>
49       <h2>Let's make your plan</h2>
50       <form onSubmit={handleSubmit} className="todoForm">
51         <label>Title</label>
52         <br />
53         <input
54           type="text"
55           name="title"
56           className="todo_title"
57           value={title}
58           onChange={(e) => setTitle(e.target.value)}
59         ></input>
60         <br />
61         <label>Description</label>
62         <br />
63         <textarea
64           rows="8"
65           className="todo_description"
66           value={description}
67           onChange={(e) => setDescription(e.target.value)}
68         ></textarea>
69         <br />
70         <button type="submit" className="addButton">
71           {updatingTodo.id ? "update" : "Add"}
72         </button>
73         <button type="button" onClick={handleDiscard} className="discardButton">
74           {updatingTodo.id ? "Delete" : "Discard"}
75         </button>
76       </form>
77     </div>
78   );

```

Figure 22. HTML code in FormTodo component in the To Do List application

Going on to the TodoCard file, there are 4 values passed into this component: id, title, description, completed for the purpose of attaching the value of title and description to the HTML code as shown in figure 23

```

6  const TodoCard = ({id, title, description, completed}) => {
7
8      const setUpdatingTodo = useTodos((state) => state.setUpdatingTodo)
9      const updateTodoCompletedStore = useTodos((state) => state.updateTodoCompleted)
10
11     const handleComplete = (value) =>{
12         console.log(value)
13         updateTodoCompletedStore(id, value)
14     }
15
16     const handleEdit = () => {
17         setUpdatingTodo(id)
18     }
19     return(
20     <div className='cardContainer'>
21
22         <div className='headerCard'>
23             <div className={`titleCard ${completed ? 'completed': ''}`}>{title}</div>
24             <input type='checkbox' onChange={(e) => handleComplete(e.target.checked)}></input>
25         </div>
26         <p>
27             {description}
28         </p>
29         <div className='footerCard'>
30             <button onClick={handleEdit}><img src={EditIcon}></img></button>
31         </div>
32     </div>
33 )}
34
35 export default TodoCard;

```

Figure 23. TodoCard component in the To Do List application

As shown in figure 23, in the todoCard component there are two issues that need to be resolved: click edit and mark complete. Once the user clicks on edit a todoCard in the todo list, the id of that todoCard is passed to the `setUpdatingTodo` function as an input value and then the `setUpdatingTodo` function in store is invoked and performed as mentioned in Figure 14. On the other hand, to capture the value of the checkbox, the `onChange` attribute was used first in the input card that specifies an event handler function, then it calls the `handleComplete` function before the `handleComplete` on the store conducts its own job.

Finally, in the `TodoList` component, the number of `todoCards` is determined by accessing the `todoList` array on the store using the `map()` function as shown in Figure 24.


```
5  const TodoList = () => {  
6    const todoList = useTodos((state) => state.todoList);  
7    const completedTodos = todoList.filter((todo) => todo.completed);  
8    return (  
9      <div className="todoContainer">  
10        <h1>  
11          You have done {completedTodos.length}/{todoList.length} things  
12        </h1>  
13        <div className="listContainer">  
14          {todoList.map((xxx) => (  
15            <TodoCard  
16              key={xxx.id}  
17              id={xxx.id}  
18              title={xxx.title}  
19              description={xxx.description}  
20              completed={xxx.completed}  
21            />  
22          ))}  
23        </div>  
24      </div>  
25    );  
26  };  
27  
28  export default TodoList;  
29
```

Figure 24. TodoList component in the To Do List application

6 Differences between Redux and Zustand

This chapter will compare Zustand with Redux based on the following criteria: usage, tracking changes, processing speed, and scalability.

6.1 Implementation

In terms of settings, data processing, and code readability, Zustand and Redux Hooks implementations differ from one another. The following table analyses these differences.

Redux	Zustand
Setting up in whole application	
<p>As mentioned above, Redux has 5 steps for installation as below:</p> <ol style="list-style-type: none"> 1. Create Redux store 2. Wrap the root component with the Provider 3. Build up actions in reducer 4. Import the useSelector and useDispatch to the child components <p>Use the values of the global state or invoke actions at consumer components</p>	<p>There are 3 simple steps when setting up Zustand in the application:</p> <ol style="list-style-type: none"> 1. Create store 2. Build up actions in store file <p>Use the values of the global state or invoke actions at consumer components</p>
Readability	
<p>In bigger systems where predictability and maintainability are critical, Redux's organized approach can result in extremely legible code. Projects with intricate needs for global state management are especially well suited for it.</p>	<p>The absence of a predetermined framework for managing state brought on by Zustand's simplicity might make it more difficult to organize and comprehend the state management logic in bigger, more sophisticated applications. The component-centric strategy used by Zustand could result in a more decentralized state management paradigm as an application evolves. As a result, keeping a clear, centralized view of the application's state may</p>

	become more difficult.
--	------------------------

Table 1. Comparison of Zustand and Redux in implementation

6.2 Tracking the state changes

During the course of running an application and meticulously scrutinising alterations in the global state to monitor state modifications becomes an indispensable asset. This capability not only expedites the debugging process but also proves particularly advantageous in extensive and intricate project environments, contributing to smoother troubleshooting and more efficient maintenance efforts

6.2.1 Redux

Along with Redux technology is Redux devtool, a powerful tool for developers. It supports features such as action, state, diff, trace, and test for developers to clearly understand state changes.

In the action tab as shown in figure, it presents two values: type and payload. With the type, it represents the name of the corresponding action. Payload is the component of the action object that contains the information required to specify what needs to change in the state.

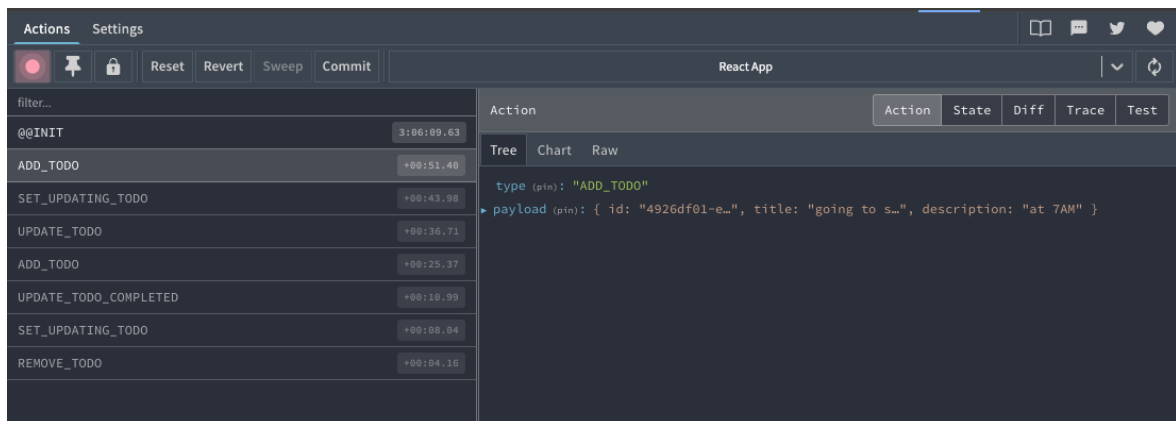


Figure 25. Action tab in Redux dev tool in the To Do List application

As shown in Figure 25, after the user clicks the add button, the ADD_TODO action is executed with the payload being the newly entered title and description values, ID and completed value. By doing this, the developer can make sure that the action object includes the

right information before passing it to the appropriate reducer to process and create the new global state.

With state tab as displayed in figure 26, it represents the current data of the properties in the state at a specific action

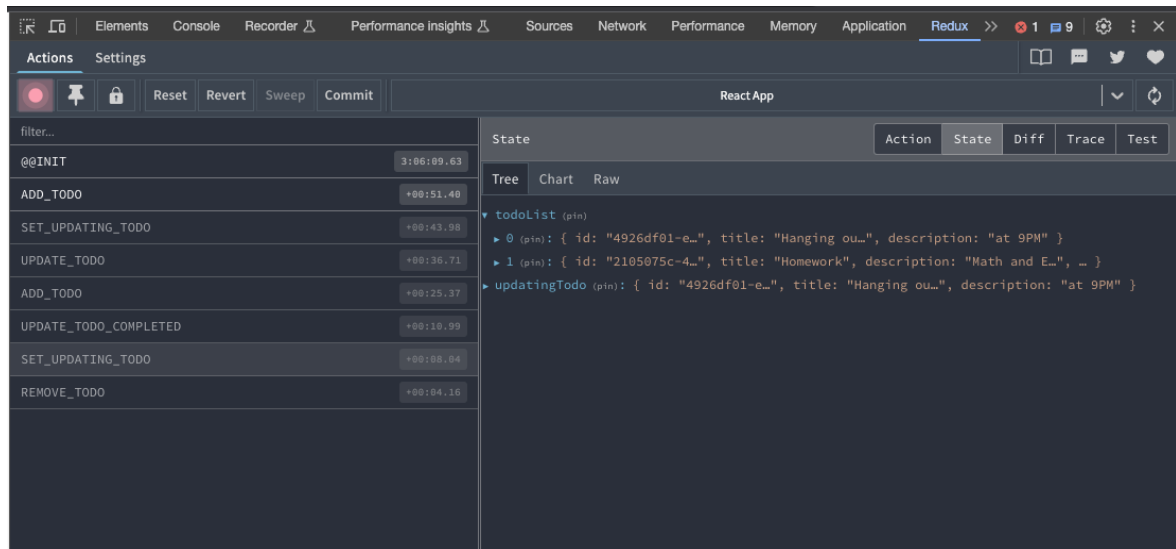


Figure 26. State tab in Redux dev tool in the To Do List application

There are 2 properties on the state of the To Do List application: `todoList` and `updatingTodo`. In which the `todoList` property is an array containing todo cards and the `updatingTodo` property contains information of a todo card that needs to be updated (Figure 26).

To see the changed values in the state, the diff tab presents a visualisation that is easy for programmers to understand.

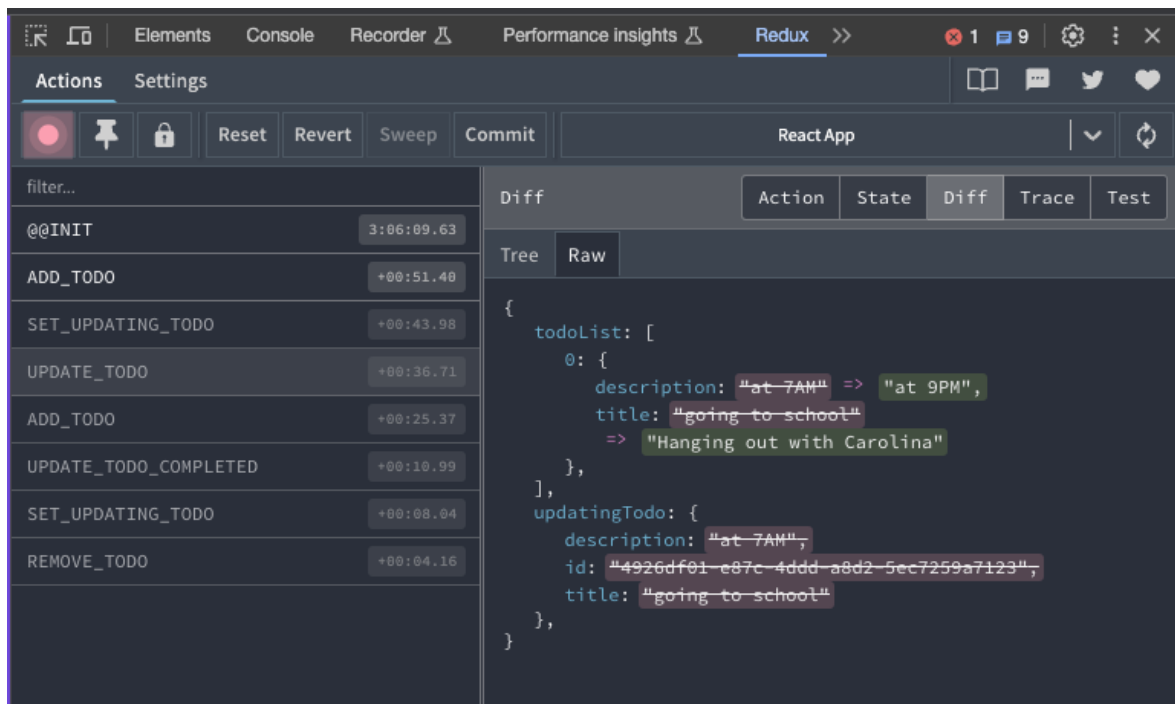


Figure 27. Diff tab in Redux dev tool in the To Do List application

In figure 27, the UPDATE_TODO action changes the title and description values of a todo card in the todoList array. This tab improves the following ability of state changes for developers.

Next, Redux dev tool provides Trace tab that can help the developers easier finding issues on small project or big project. With this powerful feature, it is incredibly beneficial for a developer to reduce tracking time down and fix any problems.

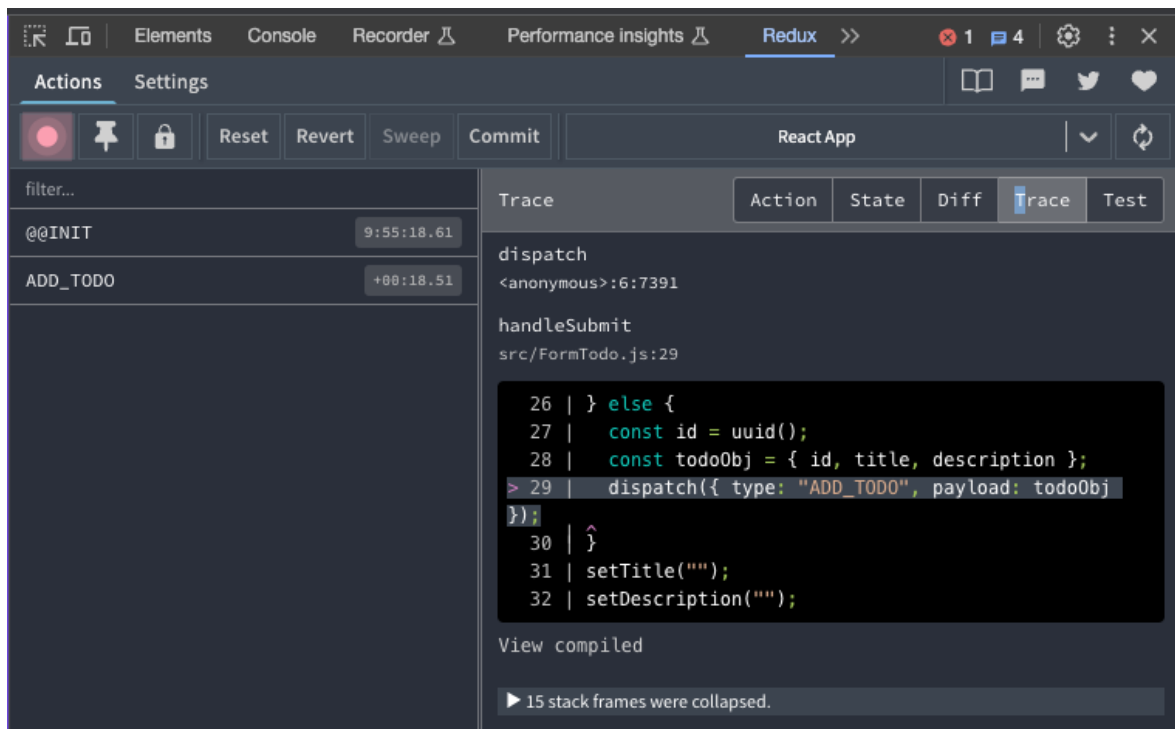


Figure 28. Trace tab in Redux dev tool in the To Do List application

In large projects, finding bugs is a very time-consuming task for programmers. As shown in figure 28, The trace tab indicates that the ADD_TODO action was executed in line 29 from "src/FormTodo.js" in the To Do List application.

Finally, the Test tab creates a test template in a number of testing frameworks that are already included, including Jest, Mocha, Tape, and Ava. It uses the root state and provides a written test to establish what the proper contents of the end state should be. (Figure 29)

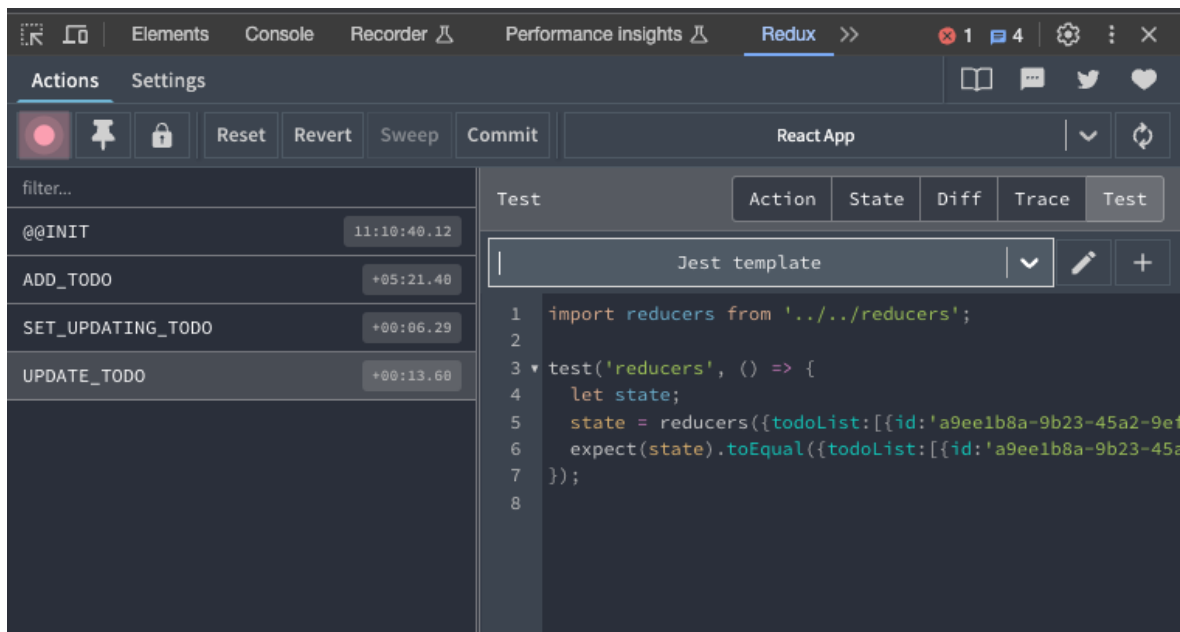


Figure 29. Test tab in Redux dev tool in the To Do List application

In general, the use of Redux Devtools generally emphasizes its superiority since it provides a wealth of sophisticated tracking options that provide React developers complete control over state monitoring. This essentially implies that React developers have the ability to carefully monitor and control state changes, giving them a potent tool to improve the development and debugging of their apps.

6.2.2 Zustand

Basically, Zustand does not have a tool to support state change tracking, but instead, developers are supposed to use a third-party state tracking tool, typically Zukeeper.

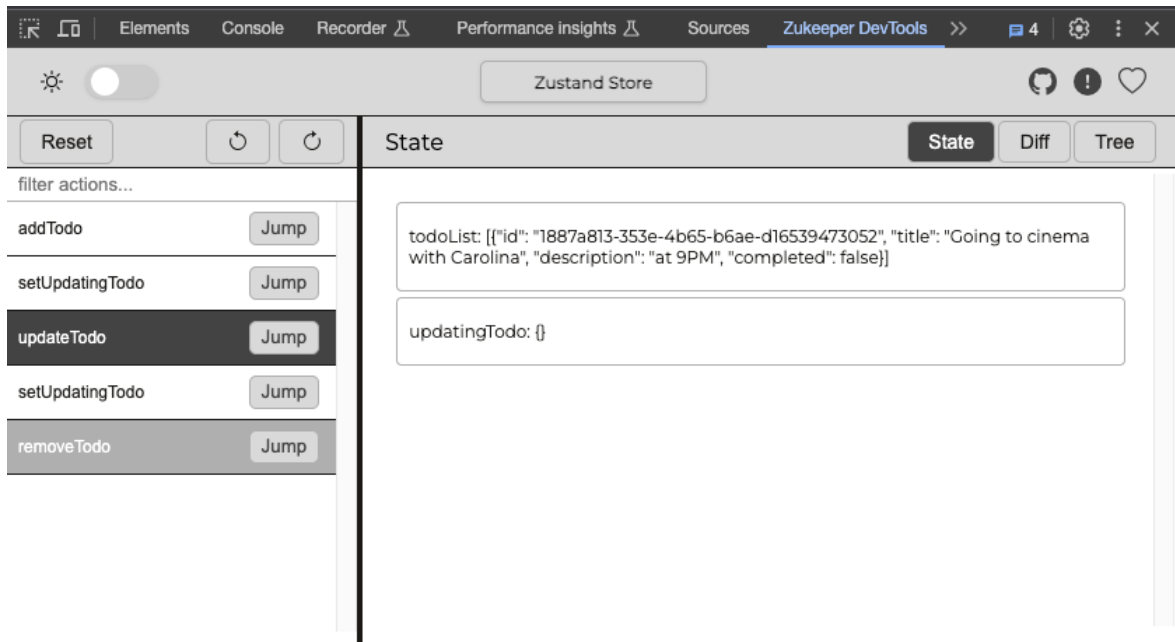


Figure 30. Tracking state changing by Zukeeper in the To Do List application

As shown in Figure 30, the status tab on the right side essentially displays the values of properties based on specific actions after being executed on the left side. Therefore, developers can easily track data through each action that users interact with and create new global states.

In addition, Zukeeper also supports the Diff feature for changed values as shown in Figure 31 below.

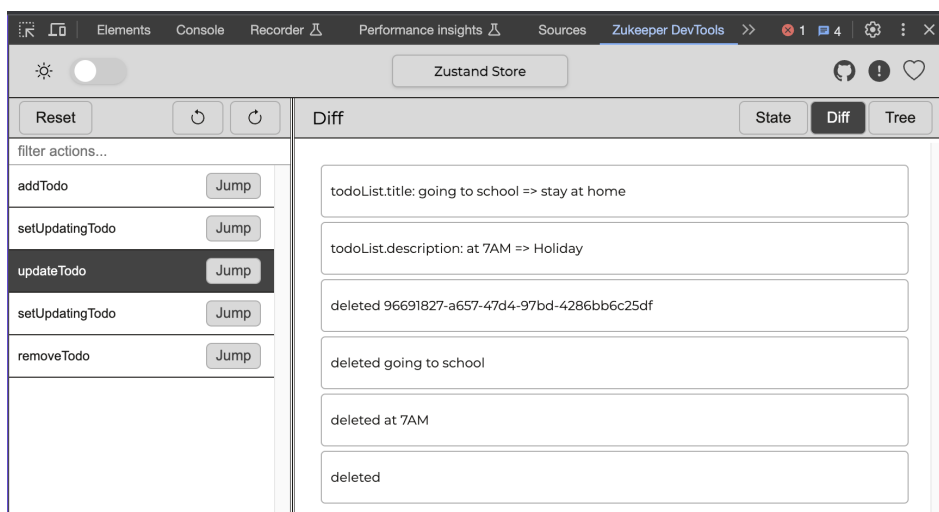


Figure 31. Diff tab in Zukeeper devtool in the To Do List application

Figure 31 is an example of when a user performs an update action. At updateTodo action, Zukeeper shows before and after changes to properties to increase tracking capabilities for developers.

In case of multiple states in the store, the tree tab presents the state values and their inheritance as the Tree Visualization feature as shown in Figure 32 below

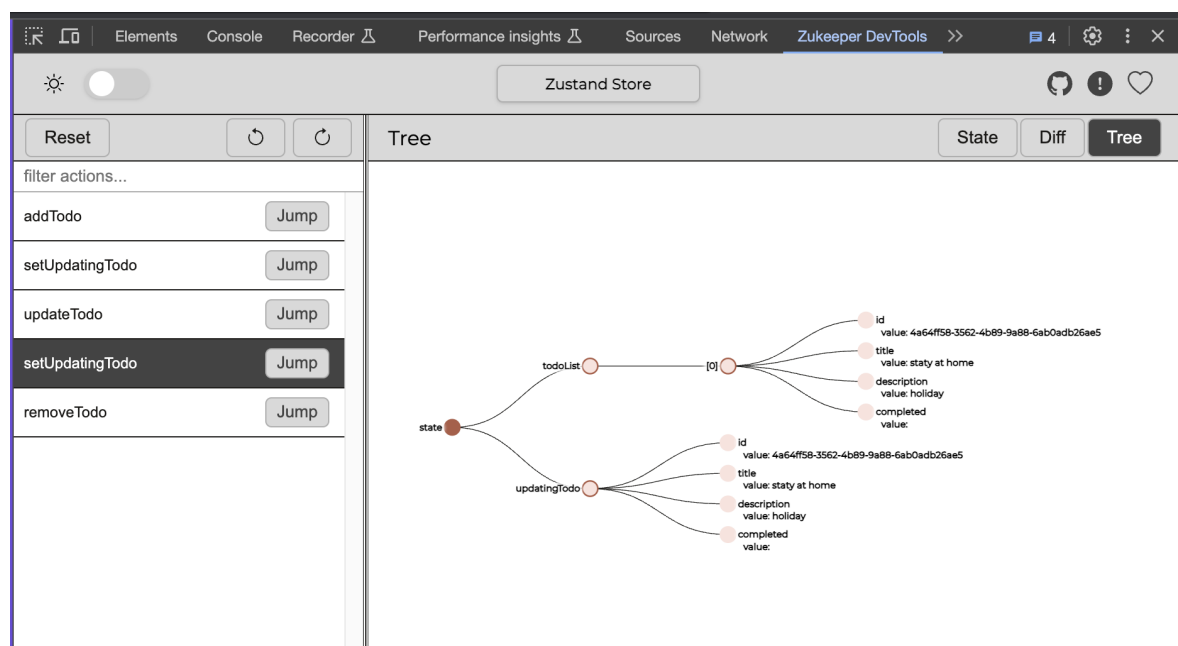


Figure 32. Tree tab in Zukeeper devtool in the To Do List application

It can be easily seen that the updatingTodo property has the values of the chosen todo card and the todoList property contains a todo card with the values of ID, title, description, and completed respectively (in figure 32).

Although Zukeeper has state management features as shown in figure 32, most developers still choose to install the redux dev tool in Zustand projects for management.

6.3 Processing speed

The rate at which events are processed is a key factor in determining how well Zustand and Redux Hook technologies perform. The evaluation was conducted using the Google

Chrome web browser, which is widely used in web application development. The purpose of the evaluation was to measure the time it takes, in milliseconds, for each technology to complete an identical task. This involved performing the task under various conditions and scenarios to ensure a comprehensive assessment.

To ensure the reliability of the results, the test was repeated 15 times for each technology. The consistent findings from these multiple measurements indicated that the browser's performance was stable, and it produced comparable results across runs. As a consequence of this reliability, the obtained result has been opted to use in the Google Chrome browser as the representative outcome in their evaluation. This choice reflects the browser's ability to provide consistent and reliable results, making it a suitable environment for the assessment of Zustand and Redux Hook technologies.

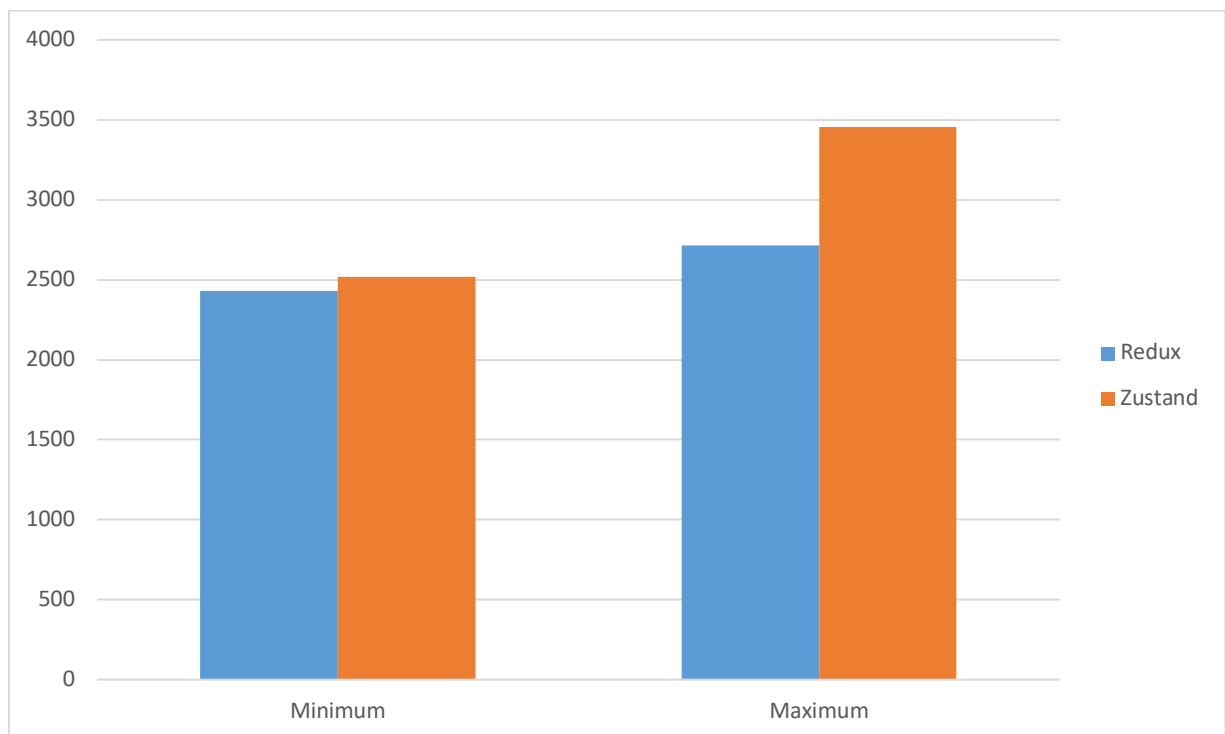


Figure 33. Minimum and maximum processing time of Redux (left) and Zustand (right) in the To Do List application

Based on measured results from Google Chrome as shown in figure 33, Redux processing all actions in the To Do List app takes time from 2430ms to a little more than 2700ms. However, when using Zustand, the processing time is quite clearly longer with the least time being about 2510ms and the maximum time being nearly 3500ms.

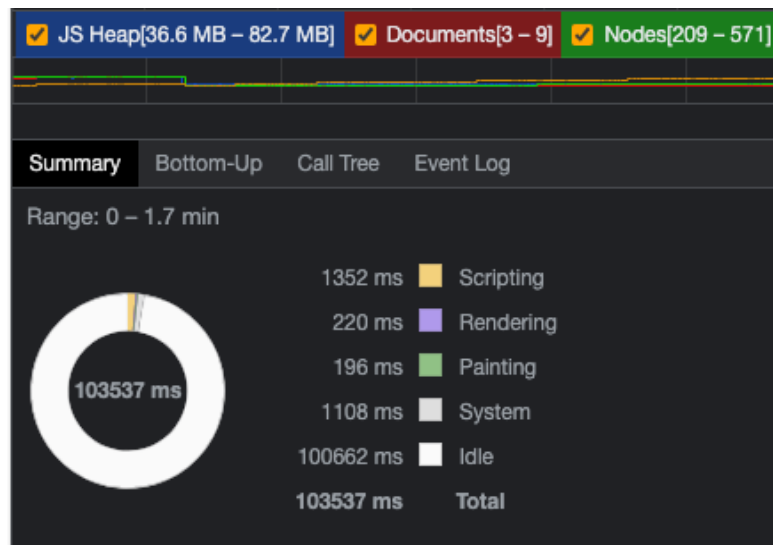


Figure 34. Performance of using Zustand in the To Do List application

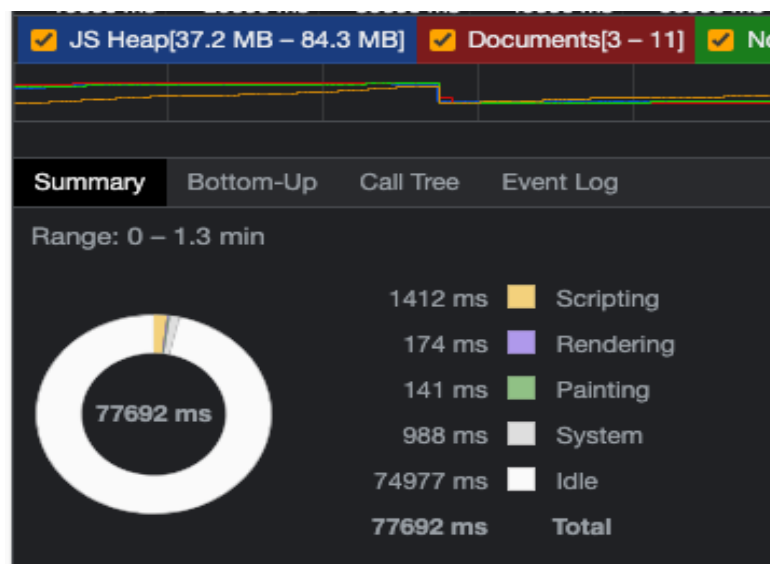


Figure 35. Performance of using Redux in the To Do List application

The following formula is used to determine the real consumption time to update state:

The actual consumption time = total consumption time - idle time

Based on Figures 34, 35, and the formula above, when the To Do List application uses Zustand, the total rendering time of all application operations takes 220 milliseconds while using Redux it takes only about 174 milliseconds.

Overall, Redux hooks provide higher performance than Zustand after many tests. It's highly likely that Redux offers tools for building memorization selectors, which may significantly cut down on pointless re-rendering.

6.4 Scalability

Redux has scalability enablers such as application size, state complexity, middleware support, performance optimization, and ecosystem.

Redux is well-known as the go-to state management for large and complex projects because its structure and centralization support state management and make maintenance easier as the application grows. One of the powerful and important points of Redux is its middleware support, which aims to handle asynchronous operations when the application interacts with cache management or involves complex business logic. By avoiding pointless re-rendering, Redux choose and remember capabilities can improve performance in large-scale applications. Building scalable applications is facilitated by Redux's established ecosystem, which includes a broad selection of middleware, development tools, and community-contributed packages.

On the other hand, since Zustand was created with simplicity in mind, its scalability is somewhat constrained, making it appropriate for smaller projects or applications that don't call for complexity like Redux. Because state management needs are simpler and Zustand primarily focuses on reducing reactivity and local updates, Zustand's performance is fairly good for lower-size applications. However, Zustand's straightforward API makes it easier for developers to quickly create smaller applications.

Therefore, Redux has a variety of capabilities for maintaining state and handling scalability, making it ideal for big, complicated applications. While Zustand is a lighter alternative that could work well for smaller applications.

6.5 Project size and code length

6.5.1 Project size

When using Zustand and Redux for state management, developers may evaluate the difference in project size by utilizing the Chrome DevTools, which provide a useful function of showcasing the sizes of the corresponding "bundle.js" files. JavaScript bundles, often known as "bundle.js" files, are files that contain the built and minified code that make up an application.

Name	Status	Type	Initiator	Size
localhost	304	document	Other	299 B
bundle.js	304	script	(index):27	302 B
css2?family=Poppins:wght@400;700&family=Roboto:wght@100&displa...	200	stylesheet	about:client:1	(memory cache)

Figure 36. The size of the bundle.js file when using Zustand and Redux in the To Do List application

After checking, the results of project size when deployed with Zustand and Redux are the same 302B as Figure 36. The high possibility for this result is that because the project was implemented simply, noticeable results were not obtained. Nevertheless, compared to Zustand, Redux often has a bigger bundle size because it is a more feature-rich framework. Redux provides with a collection of tools like Redux DevTools and middleware like Redux Thunk or Redux Saga, which might increase the bundle size. In addition, Zustand offers only the essential state management functionality without the additional features and dependencies of Redux, which might lead to a smaller bundle size.

6.5.2 Code length

The number of lines of code is one of the important factors on the comparison table, it not only saves time but also affects the ability to maintain or develop the project.

	Redux	Zustand
todoListStore.js	56 lines	66 lines
FormTodo.js	49 lines	44 lines
TodoCard.js	13 lines	17 lines
Total	118 lines	127 lines

Table 2. Code length of Redux and Zustand in implementation

As shown in table 2, there are three files considered in terms of number of lines of code: todoListStore.js, FormTodo.js and TodoCard.js file. In which todoListStore.js is the state management store, and FormTodo and TodoCard.js files are the places that work with the store. In general, the number of lines of code when using Redux is less than Zustand because the code structure of Redux is built to be intuitive and clear.

6.6 Result

Through the differences between Redux and Zustand mentioned above, table 3 evaluates the criteria of good state management based on Teimur's opinion (2022) to see which is superior between Redux and Zustand.

	Redux	Zustand
Usability	In the To Do List app, Redux's approach is structured and predictable	Zustand has shown to be very helpful through the implementation of the To Do List app due to its simplicity and low setup requirements.
Maintainability	Thanks to Redux's clear organization and support tools for developers, maintenance is easy	Zustand has minimal boilerplate code and simplicity that saves maintenance time
Performance	Based on the measured results from implementing To Do List application using Redux and Zustand, it shows that Redux has better performance than Zustand	
Testability	Redux has Redux devtool which provides all the necessary information	Zustand requires the use of a third-party tracking tool such as Zukeeper, but it also provides basic information.
Scalability	Redux is a reputable option for intricate and sizable projects.	In situations where simplicity and performance are important factors, Zustand may be more suited for scaling.
Modifiability	Redux can display powerful modifiability thanks to its structured and predictable approach. As can be clearly seen in the To Do List app, the actions and reducers are clearly separated which helps	Zustand's ability to modify makes it easy to change, which is a benefit. However, Zustand finds that while working on big projects, integrating middleware might be difficult.

	to identify the state association logic easily	
Reusability	Redux has very strong reusability because of its structured architecture and ability to create reusable actions and reducers.	Zustand provides a simpler, lighter solution that required less setup and it is very reusable.
Ecosystem	Large ecosystem, it provides many libraries, middleware, etc.	The ecosystem is limited
Community	The community using Redux is extremely large	The Zustand user community is less prominent

Table 3. Comparing criteria between Redux and Zustand

7 Conclusion

In the ever-evolving landscape of web development, the choice of a state management solution for React applications has become a critical decision for developers and teams alike. The objective of this thesis is to examine variances in state management approaches between Zustand and Redux Hooks. Additionally, this thesis presents comparative findings derived from real-world projects for each case. Ultimately aiding developers in making informed decisions when selecting the most suitable state management solution for their React projects.

In comparison, in terms of setup, Redux requires more steps to install than Zustand. Redux's presentation is said to make it easy to read and control the code for installers while Zustand's presentation lacks clarity which can lead to difficult application scaling. Regarding the ability to track state changes, while Redux has its own extremely powerful Redux devtool, Zustand needs the help of a third party such as Zukeeper devtool but can only provide basic information, however, Zustand also uses Redux devtool for status tracking. Redux's processing speed is quite superior to Zustand's thanks to the memoised selector. Therefore, Redux's ability to scale across projects is easier than Zustand's.

To summarise, the decision between Redux and Zustand should be guided by the unique project requirements, team size, and the trade-offs desired between complexity and productivity. Zustand is a preferable option for smaller projects or limited component scope, whereas Redux is better suited for projects dealing with intricate state data processing.

References

Andrei, M. 2020. How to use Figma's Inspect Panel. Retrieved on 14 August 2023. Available at <https://webdesign.tutsplus.com/how-to-use-figmas-inspect-panel--cms-36323t>

Abhishek, K. 2023. Harness State Management Using Zustand. Retrieved on 23 October 2023. Available at <https://betterprogramming.pub/harness-state-management-using-zustand-5f2ee597d1c1>

Common Tools & Practices. State Management: Overview. Retrieved on 10 September 2023. Available at <https://react-community-tools-practices-cheatsheet.netlify.app/state-management/overview/>

Elise, M. 2022. What is the Google Chrome Browser? Retrieved on 14 August 2023. Available at <https://www.lifewire.com/what-is-google-chrome-4687647>

Nikhil, S, S. 2023. A Comprehensive Guide to Understanding Redux: Simplifying State Management. Retrieved on 22 September 2023. Available at <https://nikhilsomansahu.medium.com/a-comprehensive-guide-to-understanding-redux-simplifying-state-management-521dd0e49c99>

Redux. 2023. Getting Started with Redux. Retrieved on 22 September 2023. Available at <https://redux.js.org/introduction/getting-started>

React. Managing State. Retrieved on 10 September 2023. Available at <https://react.dev/learn/managing-state>

Stephen, A. 2021. ReactJS: A Brief History. Retrieved on 10 August 2023. Available at <https://medium.com/@sjarancio/reactjs-a-brief-history-3c1e969a477f>

Soham, D, R. 2022. What is Redux? Store, Actions, and Reducers Explained for Beginners. Retrieved on 12 August 2023. Available at <https://www.freecodecamp.org/news/what-is-redux-store-actions-reducers-explained/>

Teimur, G. 2022. The Best React State Management Tools Enterprise Applications. Retrieved on 23 October 2023. Available at <https://www.toptal.com/react/react-state-management-tools-enterprise>

Uma, V. 2022. Zustand: simple, model state management for react. Retrieved on 12 August 2023. Available at <https://medium.com/stackanatomy/zustand-simple-modern-state-management-for-react-242bc5ab13db>

Visual Studio Code. Getting started. Retrieved on 14 August 2023. Available at <https://code.visualstudio.com/docs>