Thanh Le

Comparison of State Management Solutions between Context API and Redux Hook in ReactJS

Metropolia University of Applied Sciences

Bachelor of Engineering

Mobile Solution

Bachelor’s Thesis

31 March 2021

Abstract

Author:	Thanh Le
Title:	Comparison of State Management Solutions between Context API and Redux Hook in ReactJS.
Number of Pages:	40 pages + 6 appendices
Date:	31 March 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Mobile Solution
Instructors:	Ilkka Kylmäniemi, Principal Lecturer

Comparison between Context API and Redux is to expand knowledge of the state management methods behind React. The report will highlight the differences between Context API and Redux. One of the targets is to point out the specific cases of each compared state management method.

The methodology chosen for this report was to present how Context API or Redux operate, do the comparison based on criteria, and make conclusions. As a result, the study obtains a sufficient amount of data to analyse and verify the outcome that a developer should have when choosing the state management method for the highest application performance.

The result from the comparison illustrated that Context API seems to be suitable to store simple data that are operating in a small scope while Redux performs better in bigger applications. Besides that, Redux supplies a powerful tool to tracing states, which empowers the developer to debug and control the action traffic.

However, this scope of the report does not confirm that the archived conclusions are always correct in all cases. To reach more solid conclusions, more research would need to be conducted and compared.

Finally, the thesis was successful in comparing and drawing conclusions of Context API and Redux.

Keywords:	React, Redux, Hooks, Context API, state managements

Contents

List of Abbreviations

1 Introduction 6

2 React and State Managements 7

2.1 React Overview 7
2.2 State management in React 8
2.3 Context API 8
2.4 Redux 9
2.4.1 Actions 10
2.4.2 Reducer 11
2.4.3 Store 11

3 Development Environment 12

3.1 Developer tools 12
3.1.1 Figma 12
3.1.2 VScode 12
3.1.3 GIT 13
3.1.4 Chrome Dev tool 13
3.1.5 React DevTool 14
3.1.6 Redux DevTool 14
3.2 Technologies 16

4 Project discussion and implementation 17

4.1 Project idea and discussion 17
4.2 State Management Solution 18
4.2.1 Context API 18
4.2.2 Redux Hooks 22

5 Comparison 28

5.1 Implementation 28

5.2 Tracking the state changes 30
5.2.1 Context API 30
5.2.2 Redux store 31
5.3 Additional package installations 34
5.4 Code Complexity 35
5.5 Resources consumption 37
5.6 Processing speed 39
5.7 Scalability 40
6 Conclusion 41
References 42
Appendices
Appendix 1: Context API in Me_Portfolio project
Appendix 2: Redux Hook in Me_Portfolio project

List of Abbreviations

JSX:	JavaScript XML
DOM:	Document Object Model
MPV:	Model-View-Presenter
API:	Application Programming Interface
UI:	User Interface
HTML:	HyperText Markup Language
CSS:	Cascading Style Sheets
IDE:	Integrated Development Environment
AJAX:	Asynchronous JavaScript And XML
XML:	Extensible Markup Language
MIT:	Massachusetts Institute of Technology
JS:	JavaScript

1 Introduction

Nowadays, there are many web technologies assisting web developers to build up a user interface easily and React is known as the most used Javascript Library.

React was first introduced in 2011 by Jordan Walke who was a software engineer at Facebook [1]. As the technology used in the world's largest social network, React was quickly known by the programming community around the world. Moreover, a variety of React's powerful tools and features facilitates new React programmers to learn and enhance their experience of programming React applications. Until now, the concepts of virtual DOM, reusable components, function component, MVP, one-way data flow, Hooks, JSX are no longer unfamiliar to website developers.

The huge success of React is powered by advanced state management methods such as Context API and Redux. These methods have innovated the way that components communicate and share states through the component tree. This fact promotes the developers to create maintainable and scalable websites by separating different parts of logic and states that belong to a specific component.

However, it is doubted that the latest React update, including Context API first launch, will be a predicted finish of the Redux library. This report’s purpose is to clarify Context API and Redux as the most popular data management methods and compare the differences between them.

2 React and State Managements

2.1 React Overview

ReactJS is a JavaScript library and is generated for the purpose of building reusable UI components. The components receive some inputs as props, which will decide how the components are rendered. In React, a component could be nested inside other components until forming a complete web, which builds up a tree of components called the virtual DOM. Loads of web developers are using React as the View in MVC model and run their apps on the Node server. Finally, React is built on Flex as the application architecture of one way data flow (also known as the unidirectional data flow) that means the data has only one way to transfer from a component to another component of the React app. [2.]

React Features

- JSX is a JavaScript syntax extension that allows React developers to write HTML code and Javascript and in the same file [2] as App.jsx given below:

```
function App () {
              const greeting = 'Hello Function Component!';
         return <h1>{greeting}</h1>;
}
export default App;
```

- Components: to build up an application with React, developers have to brainstorm how to break the complex user interface into simpler components. Components in React could be nested and reusable, which facilitates the developer team to test, maintain and expand the codebase while building up the project [2]. At the moment, there are 2 types of components: class component and function component. To not extend from React as in class components, in the practical project, the author was only using function components to accept props and return a React component.

- Hooks: to be introduced in the React 16.8 version. It allows programmers to use state and other React features to write function components instead of class components. Hooks are only innovated to manage states and lifecycle features for function components only. [3.]

React Advantages

- The virtual DOM enhances the app performance since it is faster than the regular DOM. [2.]
- React library achieves high compatibility to run on client side, server side or with other frameworks. [2.]

React Limitations

- Since React is a library to render the view layer of the app, the developers may search for other technologies to fulfil an architecture tooling set for development. [2.]
- JSX is underestimated by the developer community due to its complexity and consequent hard learning curve. [2.]

2.2 State management in React

It can be said that passing data through the component tree in React is quite complicated. In order to receive data in a low-level component, the data has to be transferred as props through many middle-level components unnecessarily, which results in writing loads of extra code and giving the middle-level components unused properties. To solve this problem, there are many state management libraries, typically Context API - built into React version 16.8 and Redux, providing the global state solutions that all components in the virtual DOM are able to access. [4.]

2.3 Context API

Compared to previous versions of React, the developers are more successful based on a pattern of storing the state in a location at a tree root. All React developers need to learn how to pass the states of the component tree down and up through properties. However, this is no longer necessary and appropriate as React evolved and the component tree became larger. Maintaining state in a position at the root of the component tree for a complex application is not easy for many developers. There are many bug-ridden devices that appear when the developers pass the state of the tree both down and up through numerous components. [5.]

Most of the developers work on complex user interfaces. The component trees have many layers, the root of the tree and leaves are far apart. Therefore, the data layers being spaced apart and all components must receive the props that the upper layer only passes to its dependent. As a result, the code will be bloated and UI harder to scale. [5.]

State data passed through every component as props, which will finish when props reach the needing components. This is similar to traveling by train, the passenger will pass through every state but they only leave the train until they reach their destination. [5.]

In React, developers create the context provider to put context. The context provider is a component of React that the developer can wrap a part or entire component tree. A context provider is the starting point of data to different points. Each destination is a component of React, which pulls data from context. [5.] When developers use context, state data is stored in a location that passed through the tree without having to pass props down unnecessary components [5, 6].

2.4 Redux

Facebook's Flux and functional programming language Elm is the inspiration for Dan Abramov to create Redux around June 2015 [7]. Redux is working as a predictable state container and usually used in JavaScript applications. Redux helps the developers generate applications that can run in different environments, operate consistently, and be easily checked. Developers can also achieve a great experience such as time travel debugging combined with live code editing when using redux. Moreover, Redux can be used at the same time with React or any other view library. Redux has a large ecosystem of add-ons, even though it is very small. [8.]

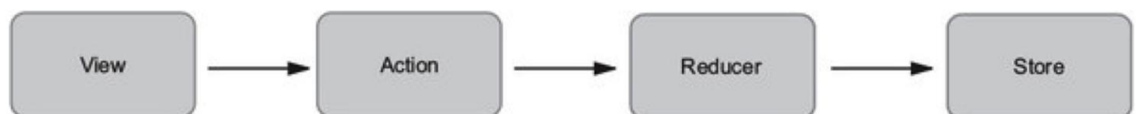


Figure 1. Data flows through a React/Redux application [9].

As shown in Figure 1, Redux is created based on three main factors including action, reducer and store. Redux operation follows the unidirectional data flow so that data in the app will follow in one-way binding data flow [10]. To be more clear, Redux data flow includes 4 following steps [10]:

- User triggers an action by interacting with the application.
- The root reducer function is called with the current state and the dispatched action. The root reducer may divide the task among smaller reducer functions, which ultimately returns a new state.
- The store notifies the view by executing their callback functions.
- The view can retrieve updated state and re-render again.

2.4.1 Actions

Everything happens in the app including the data to complete the transaction is stored in a historical record by Action's inspection. This makes it easier for developers to maintain a grasp of complex applications. [9.]

- The payload of information that sends data from the developer's application to the store called actions. The store has only one source of information is actions. The structure of Action depends on the Developer. [11.]
- Action must have type property. Type should be formatted as a string to display what kind of action being performed. Developers capitalize and use underscores as separators when using this property. [9, 11.]

2.4.2 Reducer

A Reducer is considered as a worker in a factory who will receive an action as a guideline of what to do (see Figure 2).

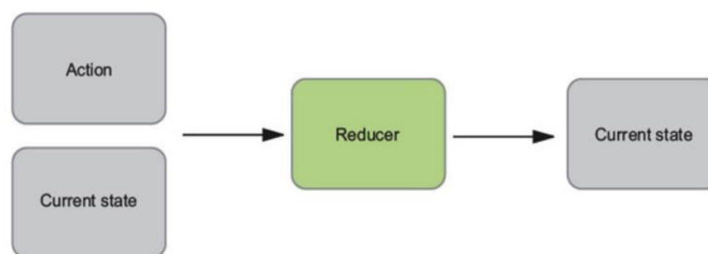


Figure 2. An abstract representation of a reducer's function signature [9].

In Figure 2, Reducers means to point out states that should be updated after a specific action executed. However, it is noticed that the actions only describe what should happen - not how the state changes. Being pure functions, the reducers receive both the previous state and an action to return the new upgraded states. The same input and action has to always yield the same value return. This helps developers to easily create tests for them. [9.]

2.4.3 Store

Reducers will be in charge of how to update state in response to an action, but they are not able to do it directly and that belongs to the store's responsibility. In Redux, all states will be kept in a single place called store and therefore any components are able to access directly to the store and archive the data. There are some principles of store, which are:

- Keep application state.
- Allow components to archive the state.
- Provide a way to specify updates to state by dispatching an action.
- Allow any components to subscribe to changes of state.

After the reducer processes the action and computes the next state, it is time for updating states in the store and broadcasting the new state to all related components. [5.]

3 Development Environment

3.1 Developer tools

3.1.1 Figma

As an interface design application, Figma provides all the tools needed for the design phase of any application interfaces. In addition, there are plenty of UI resources available on Figma library so that it boosts up significantly the process of designing for designers. [12.] While working on Figma to create the design of Me_Portfolio project, the author tends to group UI objects to be components, which smooths the path of building up React components later on. The step of design app is truly important as the final design or prototypes will be prerequisite for front end developers to structure the app as well as to style up the user interface with CSS code.

3.1.2 Visual Studio Code

Visual Studio Code is a powerful source code editor that was launched by Microsoft based on the combination of IDE and Code Editor. Free to use and compatible in macOS, Windows or Linux, Visual Studio Code supplies plenty of features for optimizing programming such as debugging, git, syntax highlighting, syntax auto completing, snippets, themes, shortcuts, etc. [13]. As Visual Studio Code supports multiple programming languages, the author had chosen this code editor to write code for the Me_Portfolio app.

3.1.3 GIT

Git is an open source distributed version control system or content tracker that is because Git is used to store the content. Once the code in Git is updated, for example, the codebase is modified or added, Git is responsible for maintaining a history of the changes, which means when the new feature code is committed, Git will add the commit into the historical commit tree. In addition, developers are able to create new branches to develop new features or merge the current code branch into the main branch to protect the current app features and avoid generating the codebase conflicts [14]. In the Me_Portfolio project, the author utilised Git to implement Context API and Redux Hooks on different branches. The comparisons of these state management solutions will be conducted later in the chapter 5.

3.1.4 Chrome Dev tool

Google Chrome browser provides a collection of web developer tools, also known as Chrome Dev-Tools. With the powerful support of JavaScript console, these tools are useful and handy to search for the problems in layouts, debug the JavaScript errors or track other meta information related to the web application [15]

- Element tab: since HTML and CSS code generate the layout and styles of the web application, any UI issue could be seen and changed in the

element tab of Dev Tools. However, any fix on the element tab must be copied to the codebase prior to refreshing or closing the website. [15.]

- Console tab: all the Javascript errors will be shown on the Javascript console in detail from the console window, developers can access any variables or functions defined in the codebase. [15.]

- Network tab: the meta data of the web application can be found on the Network panel. Network tab records all of the network requests and displays the information about the requests and responses. [15.]

- Performance tab: how the website performs will be analysed in the performance panel such as response, animation, idle phases. [15.]

3.1.5 React DevTool

React Developer Tools is an extension on Chrome and implemented in the open-source React JavaScript library to inspect the React component hierarchies. There are 2 tabs on the Chrome DevTools: Components and Profiler. [16.]

- The Components tab is to show all React components that were rendered on the page in the tree format. By selecting one component, developers are able to inspect or edit its current props and state in the panel on the right. [16.]

- The Profiler tab is to record performance information while running the application or executing an UI event. [16.]

3.1.6 Redux DevTool

Redux Devtools enable developers to perform time-travel debugging and live editing on Redux app [17]. The figure 3 below is a screenshot of Redux Devtools that is showing the inspection of the modifying state after the action of “[PROJECT] SET_SELECTED_PROJECT” was executed.

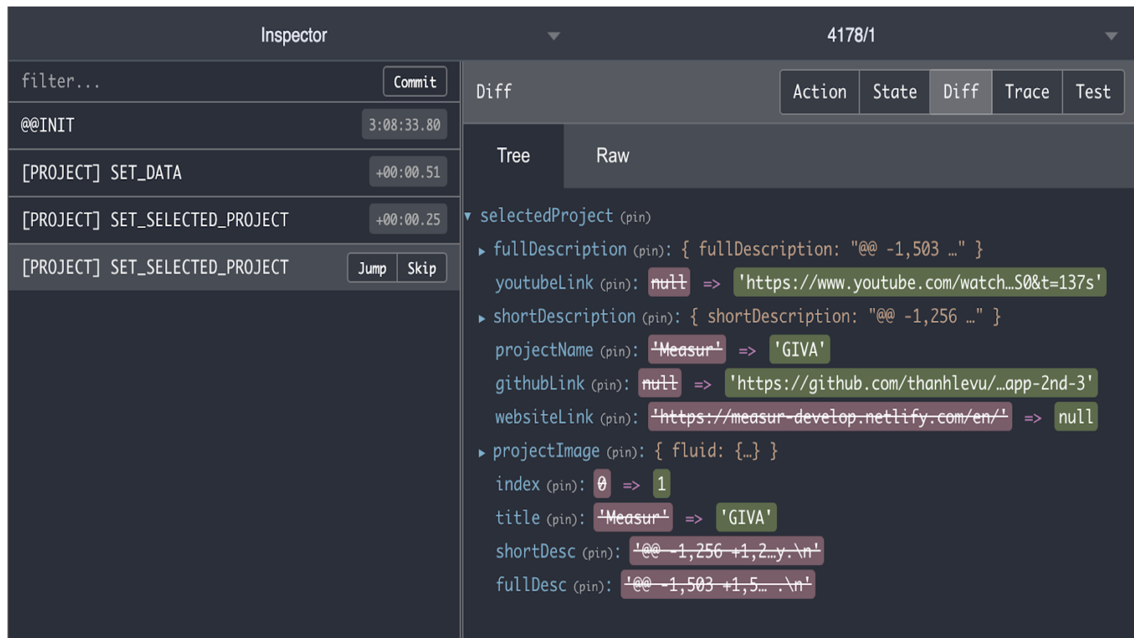


Figure 3. Redux DevTool panels

As seen in Figure 3 above, there are 2 panels of Redux Devtool. While the left panel lists the actions throughout action types, there are more details of the action presented on the right-side panel with many tabs: action, state, diff, trace and test.

- Action tab: to view action type and payload/data of an action
- State tab: to present the state after the update
- Diff tab: to watch the changes from the last state to the current state.
- Trace tab: to inspect where the action has been called.
- Test tab: to run tests for the corresponding reducers.

Moreover, developers have two options of “Jump” and “Skip” once hovering any action.

- Skip button is to skip a particular action. The app will assume that the selected action did not happen and recalculate the state. [17.]
- Jump button is to return to the state when the selected action happened. This feature is helpful when debugging and finding errors in the codebase. [17.]

3.2 Technologies

In the practical project, there are multiple technologies used in different roles. Those are shown in the table 1.

Table 1. The technologies applied into The Me_Portfolio app

Technologies	Version	Roles
React	16.11.0	React, a JavaScript library, is used to build user interfaces for the thesis practical project. [18.]
Typescript	3.7.2	Typescript is a programming language and built on Javascript by adding static type definitions. [19.]
Redux	4.0.5	To store, manage and update state in the application by using events called " actions".[20.]
Redux Devtools	2.13.8	To inspect the application’s state changes. The action and the changes of state are illustrated on the Redux devtool window. [21.]
Thunk	2.3.0	As Redux store does not support async logic and only handles synchronously dispatch actions and update the state by executing the root reduce function.Therefore, Thunk, a Redux middleware, is applied to enable writing async functions inside Redux stores. [22.]

Gatsby	2.17.11	Gatsby is a React-based open-source framework that combines Webpack and GraphQL to build websites following the latest web standard and enhanced speed and security. [23.]
Contentful	2.1.57	Contentful is content infrastructure that is to create, manage and distribute content to the website.[24.]
Netlify		Netiflify is a platform to host the website infrastructure, continuous integration and deploy pipeline with a single workflow. [25.]
Node	14.15.4	Node js an open source server environment or an synchronous event-driven JavaScript runtime, used to run dynamic page content. [26.]

As in the above table, there is a list of technologies, libraries as well as their roles and versions in the practical project.

4 Project discussion and implementation

4.1 Project idea and discussion

To serve the purpose of comparison between Redux and Context API, it is necessary to create a simple project (Figure 4) that uses either Redux or Context API on 2 different branches. The project selected in this thesis is about building up a portfolio that illustrates developer's information such as overview, education, technologies, projects, etc. The Portfolio project will be kept as at a simple level as possible to highlight the differences of state management performance on a specific feature.

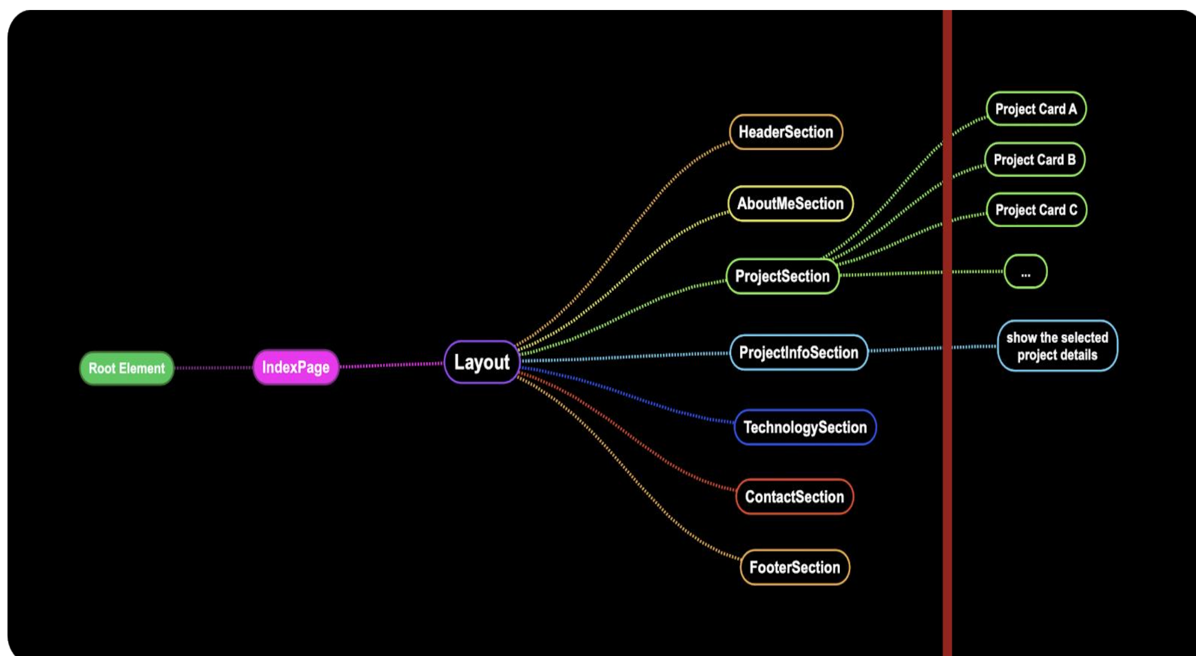


Figure 4. Schema of component architecture

For the portfolio website, there is only one page named IndexPage basically including multiple child components such as Layout, header, about me, projects, project info, technology, contact and footer, which can be seen in Figure 4.

The first issue is, ProjectSection and ProjectInfoSection would like to access data in IndexPage without prop drilling. The difference of the ProjectSection and ProjectInfoSection is that while the ProjectSection presents a brief introduction of projects in a list of cards, the ProjectInfoSection illustrates descriptions of the project in detail.

For the second issue, once the user clicks on any project card in ProjectSection, there will be an update at ProjectInfoSection with the selected project.

In order to obtain data at any child component or listen to an event from ProjectCard as a child component to update content inside ProjectSection,

there will be a demand for a global state management that wraps around those components and makes them operate in a consistent way.

4.2 State Management Solution

In the React world, there are two common solutions applied as Context API and Redux Hook. This section will lead through each of them clearly, so that the comparison will be executed in the next chapters.

4.2.1 Context API

“Context provides a way to pass data through the component tree without having to pass props down manually at every level.”
Context provides APIs that are functions to create Context objects and update sharing states inside themselves.
First of all, to make the codebase transparent and clean, the Context should be placed in a separate folder as in the picture below.

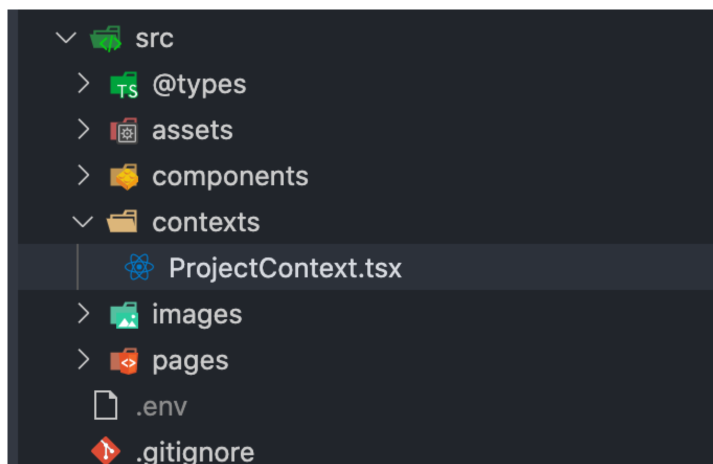


Figure 5. Portfolio App Architecture with Context

Splitting Context settings into their own files and stored in “contexts” folder assists the developer team to manage Context sharing states in a more effective direction, especially when it comes to the spec of scaling up the project in the future (see Figure 5).

To create a context object, the `createContext` method should be called and passed in an input as its default value as illustrated in the following figure.

Figure 6. Create a context object for sharing states.

As in the code example above, a context - ProjectContext is created with a default value that is an empty object.

This ProjectContext method returns the Provider component that makes the states available to all nested-level components inside itself. The Provider component has a `value` prop that receives an object of states and functions to update value of states in Figure 7.

Figure 7. ContextProvider component wraps the IndexPage component.

In Figure 7 is the IndexPage component. It has all data of the Portfolio website that wishes to share to child components. The data, projectList, setProjectList,

selectedProjectIndex and setSelectedProjectIndex are now shared to all child components covered by ProjectContext. Provider components.

To connect to a context, the child component should access the context by calling useContext that needs a parameter to identify which context to connect to. This is displayed in the Figure 8 below:

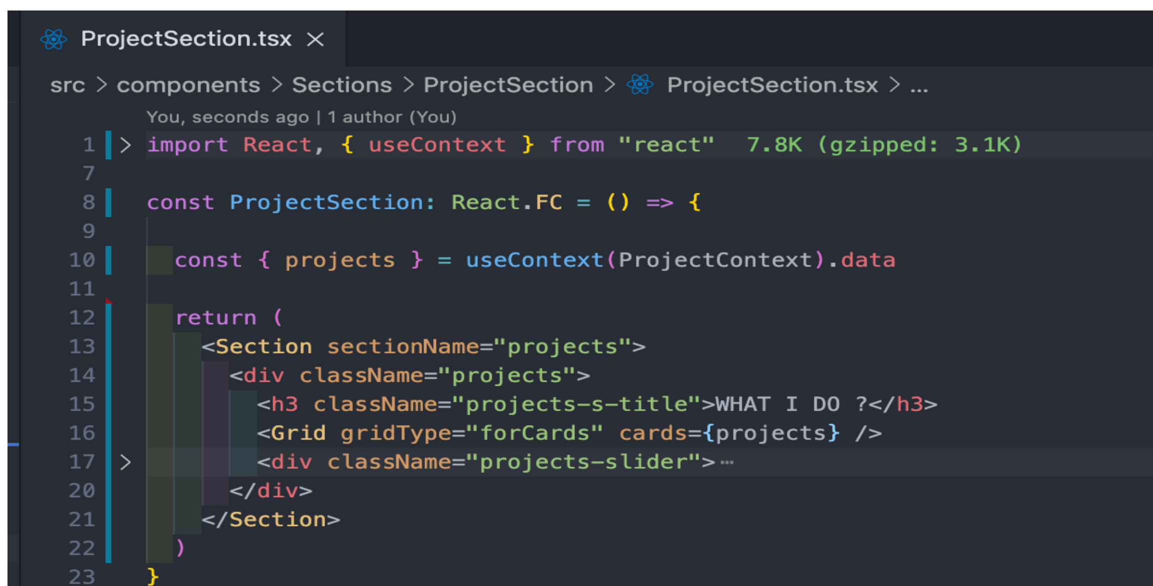


Figure 8. Access to ProjectContext from ProjectInfoSection.

As mentioned in picture 8, the function component - ProjectSection obtains “projects” data for its rendering by connecting to ProjectContext. The “Grid” component receives the data to render a list of “Card” components used to update “selectedProjectIndex” state when click on it (see Figure 9).

```
Card.tsx ●
src > components > Card > Card.tsx > ...
You, seconds ago | 1 author (You)
1  > import React, { useContext } from "react"  7.8K (gzipped: 3.1K)
6
7    const Card: React.FC<CardProps> = ({ title, shortDesc, index }) => {
8
9      const { setSelectedProjectIndex } = useContext(ProjectContext)
10
11     return (
12       <div
13         className={`card card-background-${index + 1}`}
14         onClick={() => {setSelectedProjectIndex(index)}}
15       >
16         <h3 className="card--title">{title}</h3>
17         <p className="card--shortDesc">{shortDesc}</p>
18       </div>
19     )
20   }
```

Figure 9. Card component to update the state.

As shown in figure 9, the Card component gets the setSelectedProjectIndex function from

Overall, there are 3 steps to set up a global state management using Context:

- Create Context
- Create states and Context Provider
- Call useContext to get state from child components

4.2.2 Redux Hooks

The following are steps of implementation of Redux into the Me_Portfolio app.

1. Create action types:

The very first step to work with Redux is creating action types that describe the action implemented and displayed on the Redux console as the given example 10 underneath.

```
TS actionTypes.ts ×
src > store > indexPage > TS actionTypes.ts > [⊘] SET_SELECTED_PROJECT
      You, seconds ago | 1 author (You)
  1   export const SET_DATA = "[PROJECT] SET DATA"
  2   export const SET_SELECTED_PROJECT = "[PROJECT] SET SELECTED PROJECT"
```

Figure 10. Redux Action types in Me_Portfolio App.

The action's names are capitalized and made understandable in Figure 10 because of its string values. The description of an action should start with a prefix such as state name, for example [PROJECT], which facilitates the project able to scale up in the future without the problem of duplicating action expression.

In the Me_Portfolio app, there are 2 actions to update the state:

- SET_DATA : to set all project data into a state in Redux's store
- SET_SELECTED_PROJECT: to update the selected project that the user would like to view more information on it.

2. Create Redux store

At this step, to create Redux store, there is a call of createStore function that requires 3 inputs including:

- Reducer: update the current state in store based on action invoked [27].
- Initial state: as known as the default value to state.
- Middleware: dispatch the result of other action creators even though those are synchronous or asynchronous.

```
index.ts
src > store > index.ts > store
You, seconds ago | 1 author (You)
1  import { createStore, applyMiddleware, Store, Dispatch } from "redux"  5.4K (gzipped: 2.1K)
2  import reducer from "./indexPage/reducer"
3  import { composeWithDevTools } from 'redux-devtools-extension';  800 (gzipped: 429)
4  import reduxThunk from 'redux-thunk';  559 (gzipped: 328)
5
6  const initialState = {
7    projects: null,
8    selectedProject: null
9  }
10 const middleware = [reduxThunk]
11
12 export const store: Store<any, any> & {
13   dispatch: Dispatch
14 } = createStore(reducer, initialState, composeWithDevTools(applyMiddleware(...middleware)))
```

Figure 11. Redux store in Me_Portfolio App.

The picture 11 shows how the store is set up with reducer, initialState and reduxThunk (middleware).

- The reducers: to handle the state update.
- initialState is an object that has 2 properties: “projects” to hold all projects in the app and “selectedProject” to demonstrate what project is chosen to show in the ProjectInfor section.
- The middleware in the project is “redux-thunk”: to write async logic that interacts with states in the store

3. Design actions:

As mentioned in the chapter 2.4.1, to create a Redux action, the developer has to create an object having two properties: type and payload.

```
TS index.ts    TS actionCreators.ts ×
src > store > indexPage > TS actionCreators.ts > ...
You, 3 days ago | 1 author (You)
1  import { store } from ".."      You, 3 days ago • draft
2  import * as actionTypes from "./actionTypes"
3
4  const setData = (data: any) => ({
5      type: actionTypes.SET_DATA,
6      data,
7  })
8
9  export const boundSetData = (data: any) => store.dispatch(setData(data))
10
11 const setSelectedProject = (projectIndex: number) => ({
12     type: actionTypes.SET_SELECTED_PROJECT,
13     projectIndex,
14 })
15
16 export const boundSetSelectedProject = (projectIndex: number) => store.dispatch(setSelectedProject(projectIndex))
```

Figure 12. Design Redux actions in Me_Portfolio App.

As described from the above picture, the app has 2 action set up as followings:

- The "setData" action has the type of "SET_DATA" and the payload of data containing "projects" and "selectedProject".

- The "setSelectedProject" action has the type of "SET_SELECTED_PROJECT" and the payload of data containing the selected project information only.

Besides designing actions, the author creates `boundSetData` and `boundSetSelectedProject` functions with the purpose of dispatching the actions immediately at the using places without invoking `store.dispatch` again.

4. Create reducers:

It is required that each action needs a specific reducer to update state. However, before all, the developer has to design the state structure with `initialState` that is to set the default value for state at the starting point.

Because a reducer is a Javascript function that takes two arguments (current state and action) in order to return a new state, it would reach a higher optimizing level of codebase if using Javascript switch statement to classify cases depending on action type rather than regenerating multiple reducers.

index.ts | actionTypes.ts | reducer.ts | actionCreators.ts

src > store > indexPage > reducer.ts > ...

You, 3 days ago | 1 author (You)

```
import * as actionTypes from "./actionTypes"

const initialState = {
    projects: [],
    selectedProject: {}
}

const reducer = (
    state: any = initialState,
    action: any
) => {
    switch (action.type) {
        case actionTypes.SET_DATA:
            return action.data

        case actionTypes.SET_SELECTED_PROJECT:
            return { ...state, selectedProject: state.projects[action.projectIndex] }

        default:
            return state
    }
}

export default reducer

```

Figure 13. Creating Reducers to handle Redux actions

As seen in figure 13, the format of global states in the app includes “projects” and “selectedProject” which are an empty array and an empty object correspondingly at the beginning.

In the reducer, there are 2 cases to handle 2 different actions of SET_DATA and SET_SELECTED_DATA. Specifically, they are set up as the followings:

- SET_DATA: since action.data is an object that has new “projects” and new “selectedProject”, the reducer of SET_DATA will return “action.data” as a new global state.
- SET_SELECTED_PROJECT: the payload of action in this case is “projectIndex” that is the new index of the project that the user selected to read more information on the UI. Hence, the new global states will be formed based on the same “projects” property by destructing the current global state. In the meanwhile, the “selectedProject” property is identified by the selected project index in the “state.projects” array.

5. Wrap around the rootComponent:

Once finishing Redux setup, it is an important step to generate the connection between React app and Redux store as the following figure 14.

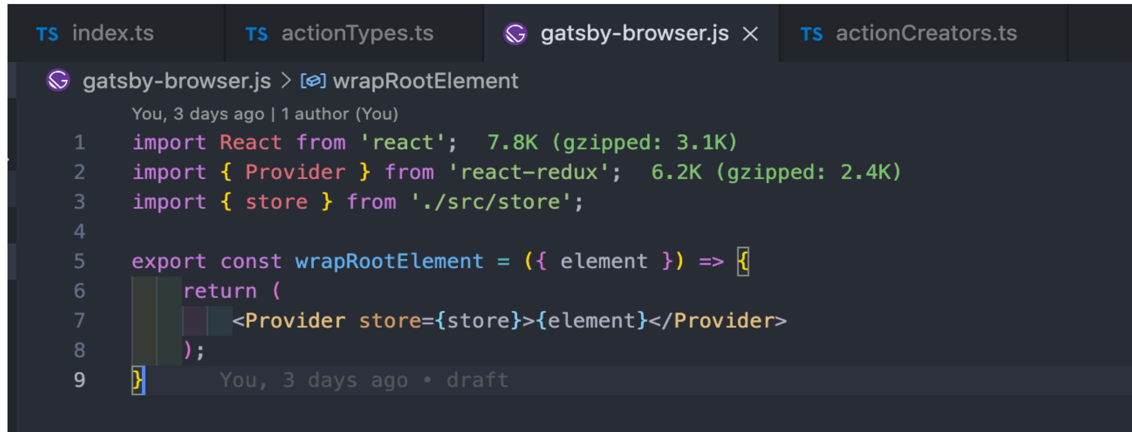


Figure 14. Injecting Redux store into React app.

As described in the picture, to access the Redux store from React app, it is necessary to import the Provider component from react-redux library and store from Redux store.

This could be done in 2 following steps:

- The Provider component will cover the “element” as RootComponent inside wrapRootElement component.
- After that, the Redux store created by the developer will be passed into the store prop of the Provider component.

At this time, all child components in the app are able to access the store to get the states or call any actions to update the store.

6. Get selectedProject from store

To extract data from Redux store state, the useSelector Hook will be called along with the clear description of the state to return the state correctly.

```
31
32  const ProjectInfoSection: React.FC = () => {
33
34      const selectedProject: Project = useSelector(
35          (state: any) => state.selectedProject,
36          shallowEqual
37      )
```

Figure 15. Accessing a state in Redux store.

A Redux store is an object and contains states as the properties. In order to gain any property like “selectedProject ", in the picture 15, the developer accesses the store and gets state with useSelectore hook function, then extracts the state to achieve selectedProject value.

7. Invoke an action of boundSetSelectedProject.

While building up the Card component, the developer creates a function of handleCardSelected in order to handle onClick events on the project card.

```
11
12  const Card: React.FC<CardProps> = ({ title, shortDesc, index }) => {
13      const handleCardSelected = React.useCallback(
14          (index) => boundSetSelectedProject(index),
15          [boundSetSelectedProject]
16      )
17
```

Figure 16. Obtaining actions in Redux at a component.

The handleCardSelected function will get an input as index of the project in the project list data so that the developer could set correct project data into selectedProject state (figure 16).

In conclusion, Redux implementation contains 7 steps from settings to usage.

5 Comparison

To clarify the difference between Context and Redux, this chapter will make comparison via criteria: usages, tracking changes, packages installations, complexity, resource consumptions.

5.1 Implementation

The implementations of Context API and Redux Hooks are different regarding their settings, data processing and code readability, which are analysed in the following table.

Table 2. Comparison of Context API and Redux Hooks in implementation.

<table>
<tr><th>Context API</th><th>Redux Hooks</th></tr>
<tr><td colspan="2">To set up and apply at the child components</td></tr>
<tr><td>As explained in the chapter above, it is obvious that Context's setup is simple through 3 steps:
1. Create context
2. Wrap the parent component with context's provider which contains global states
3. Invoke global states to use at any child components.</td><td>Redux requires more details for its own setup with 5 steps as below:
1. Create store
2. Create actions
3. Create reducers with state's default values
4. Wrap the parent component with Redux's Provider
5. Invoke global states or actions to use at child components.</td></tr>
</table>

To handle complex data in the global state	
In the chapter 3, to update the state in the context, there are many functions created and passed into Provider such as setProjectList or setSelectedProjectIndex. For big projects, this would be a significant disadvantage if the global state has dozens of actions because it results in a huge and complicated component.	Redux allows users to split states, actions, reducers and store settings into their own separate files so that developers could manage or review the data flow in an easier way.
To code readability	
Even though the global states (data or projectList) in the Context are created as states inside component IndexPage only, those global states actually aim to update IndexPage's child components further than the IndexPage component itself. This means that when a developer wants to investigate global states, they will probably have to go to a parent component that doesn't seem to be related to them. For example, in the Me_Portfolio app, the Provider component should be placed inside the Layout component but it is not because IndexPage is a place to archive data from the server.	The great benefit of Redux is that Redux settings stay away from React components and that each action or reducer is separately on their own file. Hence, it brings higher transparency to inspect any Redux changes throughout it's actions and reducers.

As discussed above, the Context API is superior in terms of use, but it has many obstacles in extending the project and handling complex data. In other words, Context API is more suitable for small and medium projects rather than big projects.

5.2 Tracking the state changes

In the process of running the app or inspecting the changes of the global state, the ability of tracking state changes plays a critical role as it facilitates debugging tasks faster and more effectively, especially in the large projects.

5.2.1 Context API

In order to track changes with ContextAPI, the Chrome extension of React Developer Tool is used popularly even though the tool only shows the state after updating. However, in most cases, the front end developers would like to earn more information on the update such as which components or events triggered the update and how the states are changed from the last time.

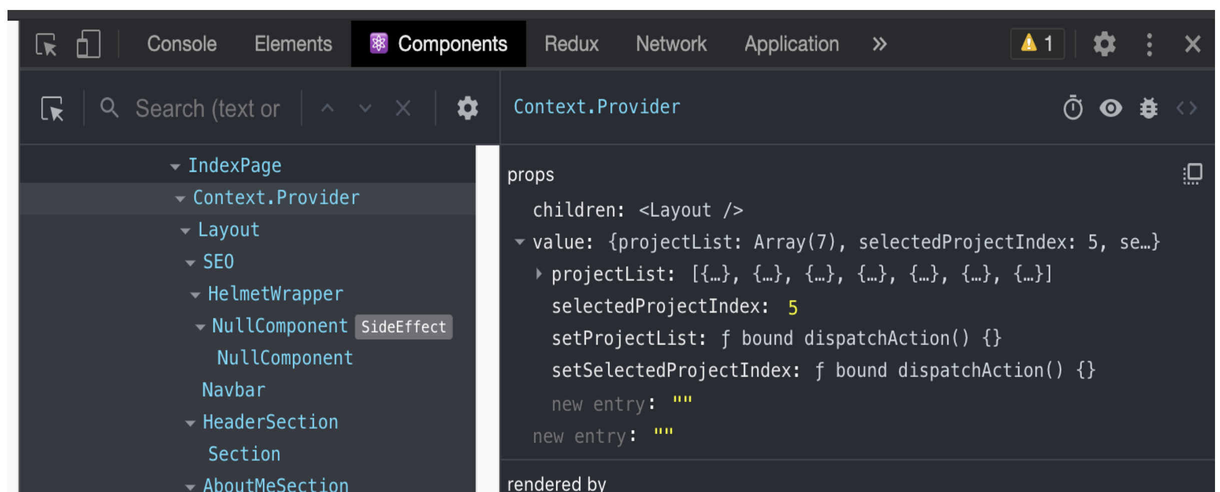


Figure 17. Tracking state changing in Context API

As presented in figure 17, the global state could be viewed at value props of Context.Provider components. To compare the versions of global state during the update, a function of console.log should be injected into the update-affected components. However, in a huge React App, this solution seems to be impossible as the update could be triggered from hundreds of places.

5.2.2 Redux store

Regarding Redux technology, there is a Redux tool which is powerful and developed along with the React-Redux. It allows developers to dive into every single update to review the changes or even access to where the event is called.

As in the Figure 18 below, the action tab shows the selected action object with its properties: “type” and “projectIndex”.

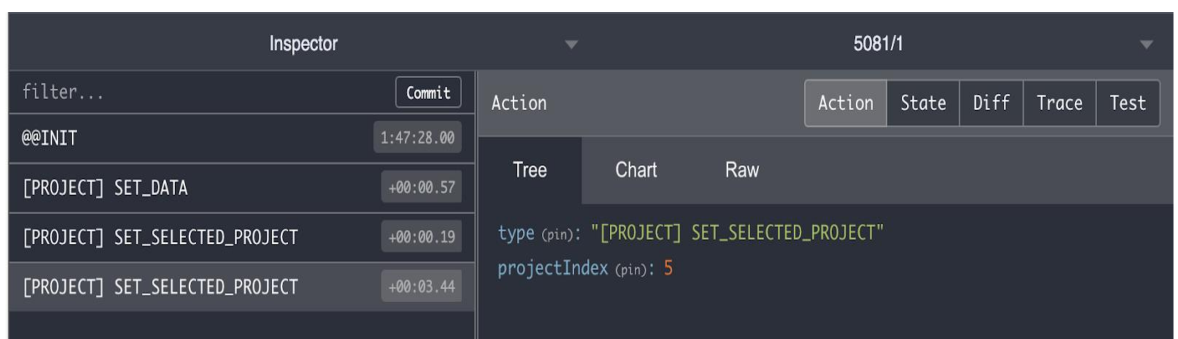


Figure 18. Action tab in the Redux console.

Figure 18 is an example of when a user clicks on a project card on the website. After the click event, the action of “[PROJECT] SET_SELECTED_PROJECT” is executed with the payload is the new index of the selected project. Thanks for this, the developer would ensure that the action object contains correct data that will be transferred to the corresponding reduce to process and generate the new global state.

Secondly, in order to review the value of state in the meanwhile, there is a state tab in Figure 19 showing a clone of the current global state in three formats of tree, chart or raw.

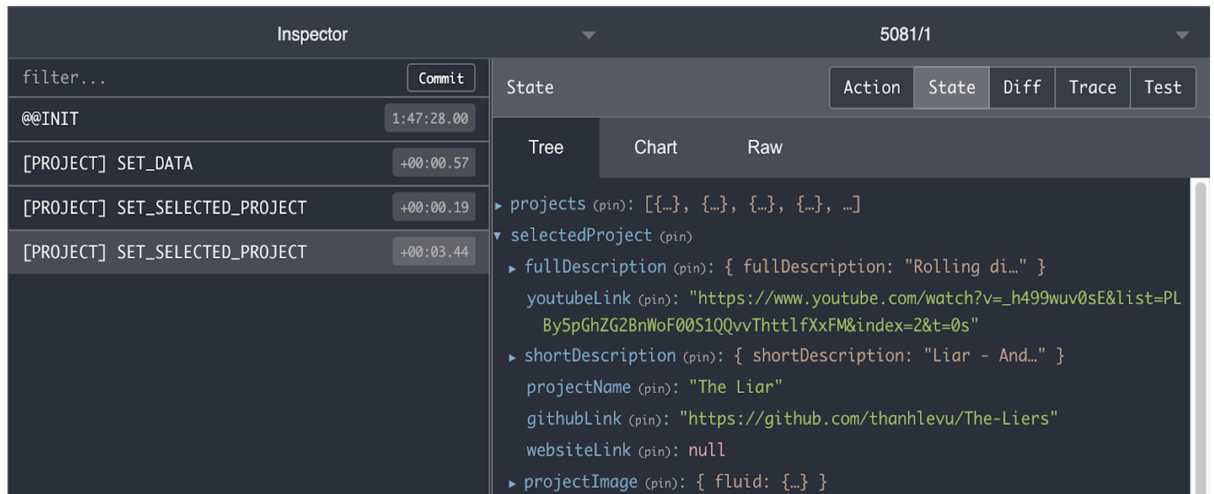


Figure 19. State tab in the Redux console.

With the state tab, the developers are able to explore all values belonging to the global state as in the Figure 19. This option is the same on the tab of Components in React Developer Tools when selecting “Provider” component. By seeing this, the developers are confident to know the data in the global state always in the control before firing new changes.

Thirdly, after executing an action, the diff tab in the figure 20 will show only what that action changed in the state tree.

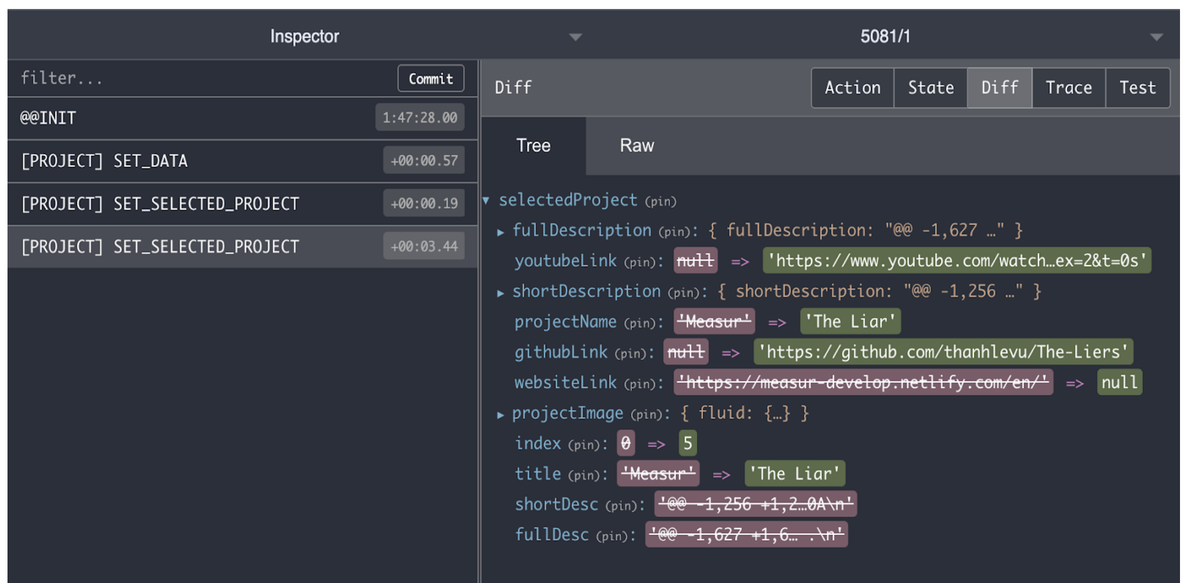


Figure 20. Diff tab in the Redux console.

The action of “[PROJECT] SET_SELECTED_PROJECT” selected in Figure 20 is created to only make a change at “selectedProject” of the global state so in the diff panel, there are values of “selectedProject” state before and after performing the action. In general, this tab strengthens the ability of tracking changes of state to developers by illustrating what exactly occurs during an action.

In such a big project where an action could be triggered in many places, it is difficult to know the correct place running the action. To solve this problem and help programmers reduce debugging time, the trace tab lists out files that action invoked and previous happenings. This could be found in the Figure 21 below.

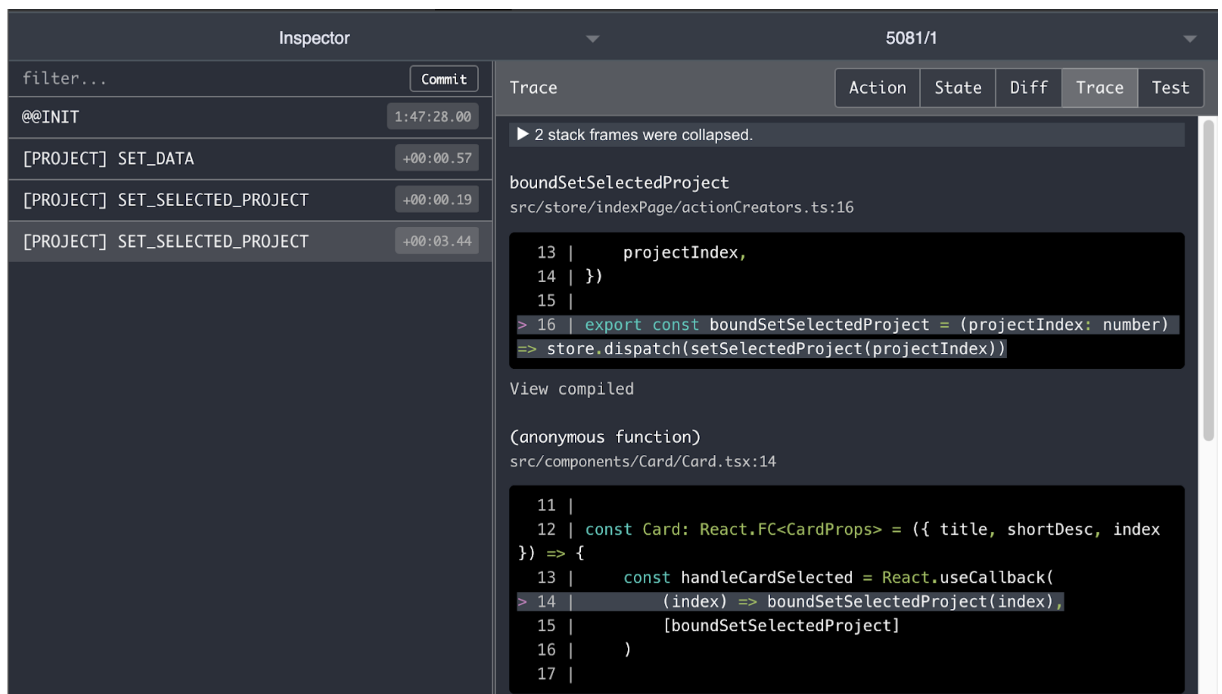


Figure 21. Trace tab in the Redux console.

As can be seen in the figure 21, the trace panel demonstrates the action of “[PROJECT] SET_SELECTED_PROJECT” was called in “boundSetSelectedProject” at the line 16 in the path of “src/store/indexPage/actionCreators.ts”. Moreover, it also points out the flow of all events happening from click event until the state completing the update.

Finally, it is the test tab that is in charge of displaying tests associated with the action.

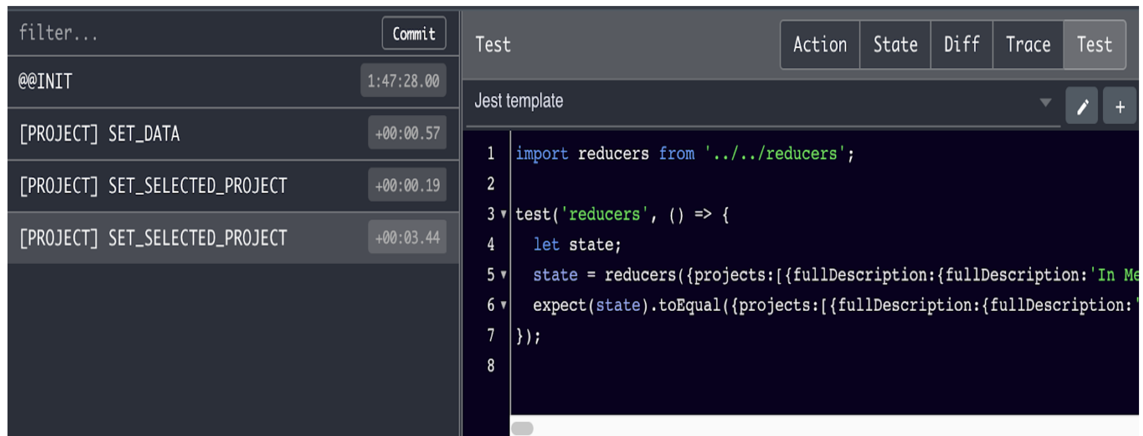


Figure 22. Test tab in the Redux console.

In Figure 22, the test panel is showing the test of reducer handling the related action. Besides that, the programmers could add more tests of actions and reducers into the test tab by using testing libraries such as React testing library, JEST or Enzyme.

Overall, with the support of Redux Devtool, Redux has identified its superiority against Context API by providing many extremely tracking features for React programmers. In other words, React developers own full control to monitor the state variations.

5.3 Additional package installations

The table below is presenting the installation packages for each state management solution.

Table 3. Additional packages going along with Context and Redux.

Context API	Redux
• From the React 16.8 version, context API is built up into React • React Dev tools	• Redux library • Redux Dev tool • Middleware Thunk

As presented in the table 3, from the release of React version 16.8, Context API was built up into React as a solution to handle the complication of prop drilling which assists React developers to approach or build up a state management without needing to install any external package.

However Redux is known as a state management technology independent of React, it needs to be installed into React apps as an external library which definitely occupies a small amount of space from the development environment memory.

In addition, as mentioned before in chapter 3.2, Redux is only able to process simple synchronous updates throughout dispatching an action. Hence, Thunk middleware is installed in the practical project to enable developers to write async functions of accessing the store and handle AJAX requests. [28.]

In the last chapter, both Context API and Redux require browser extensions to monitor the changes of state. While React Dev tool is built for Context, it is Redux dev tool for Redux. These two extensions are available and easy to install on Chrome, FireFox and other browsers.

5.4 Codebase Complexity

To reach a higher level of convincing in the comparison of codebase complexity, the thesis takes into account practical projects of Context and Redux thanks to Advanced Search Tool on GitHub [29]. The selected projects are reliable replied on search conditions:

- Later than Jan 1st 2018
- MIT license
- More than 20 stars

The table 4 given below is built based on the measure of Context or Redux related codebase into each project. After calculating, the values are rounded to the nearest ten.

Table 4. Complexity of Context API and Redux Hooks in different React projects.

State Management Solution	Project Link	Project size (in KB)	Total code size of state management (in KB)	Complexity percentage %
Context API	Movie List	4053	3	0.064
	Budget App	646	2	0.31
	Search Github User	1443	2.6	0.18
	Me Portfolio	998 874	0.5	0.00005
Redux	React Social Network	791	10	1.26
	Shopping Cart	394	4	1
	Me Portfolio	998 964	1.4	0.00014

The table 4 illustrates the code complexity of context and Redux through various practical projects in different sizes.
The complexity percentage in the table 4 is calculated based on the following formula:

Complexity percentage = $\frac{Project\ size\ KB}{State\ management\ size\ in\ KB} \times 100\%$

Which are:

- Project size is the total project code base in KB.
- State management size is the state management codebase inside the project in KB.
- Complexity percentage is how much state management codebase occupies in the total project codebase in percentage.

It is obvious that the amount of Context code in projects is less than Redux, which is because Context consumes less code setup while Redux structure needs more code lines and files.

Regarding the portfolio website of Me_Portfolio that uses either Context or Redux, the amount of Redux volume is double than Context's in general. Moreover, when reviewing the total code numbers, the size of the project with Redux is slightly higher than one of Context.

Overall, even though Redux required more codebase than Context, the consumptions of Context and Redux are negligible.

5.5 Resources consumption

These columns of General Memory and Javascript Memory below illustrate how much the Me_Porfolio app is using memory on multiple device simulators under a state management application of Context API or Redux Hook:

- The General Memory column is about native memory which stores DOM nodes. If the value of General Memory is growing up, there are more DOM nodes created on the DOM tree. [30.]
- The numbers in the JavaScript Memory column are representing the JS heap which is how much memory the reachable objects on the app are utilizing. These numbers are increasing once either more objects are being created, or the existing objects are growing. [30.]

The table 5 describes how Context API and Redux Hooks use Chrome browser memories. The data is collected from Chrome developer tools while running the Me_Portfolio project only.

Table 5. Context API and Redux Hook on Chrome Browser resources.

		General Memory MB	**Javascript Memory KB**
Context API	Desktop	71 ~ 95.9	~ 12,972
	Ipad	133 - 189	13,924 ~ 14,948
	Iphone	113 - 120	11,156
Redux Hook	Desktop	74 ~ 102	14,645 ~ 15,655
	Ipad	133 - 273	15,215 ~ 15728
	Iphone	120 - 128	15,472

From data in the table 5, these 2 state management solutions of Context API and Redux HOOK obtain fast processing speeds. However, data also points out

that Redux spends more memory than Context API regarding General Memory and Javascript Memory in the Me_Portfolio project scope.

5.6 Processing speed

Regarding the performance of Context API and Redux Hook technologies, the speed of processing events is an important criterion. The measure was conducted on 3 popular browsers such as Chrome, Firefox and Microsoft Edge and the results show how much time it takes to process the same job in milliseconds. All browsers return similar results after 20 measurements on each browser. Therefore, the author selects the result from the chrome browser as the representative result. See Figure 23 below.
The actual consumption time to update state is calculated by using the following formula:

The actual consumption time = total consumption time - idle time

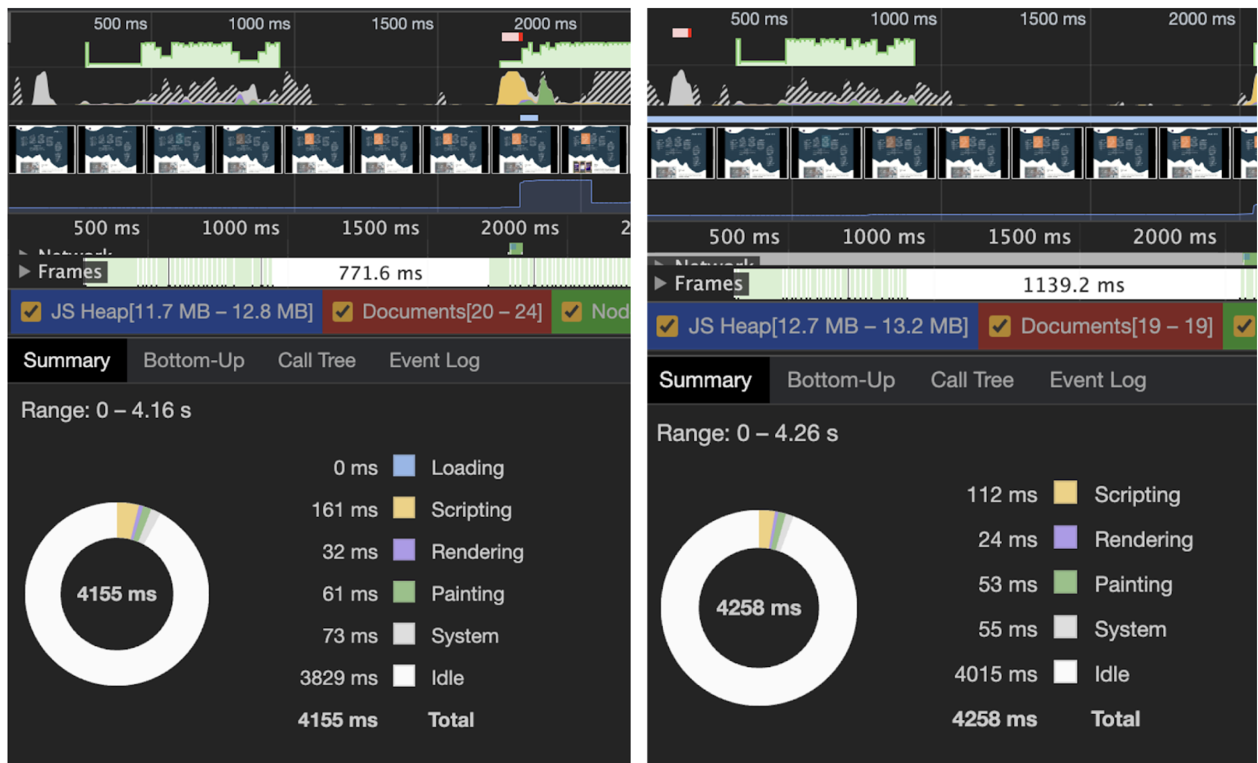


Figure 23. Performances of Context API (left) and Redux Hook (right) on updating the state.

For the Me_Portfolio app using Context API, it takes 326 milliseconds to set a newly selected project into global state while Redux Hook consumes less to 243 milliseconds, only 75% compared to Context API.

After many tests, the results are the same as Redux Hook brings higher performance than Context API. The proper reason for this fact is that once any state in context is modified then all context-related components will re-render again. In a better way, Components using Redux states only update when that specific state is changed.

5.7 Scalability

Context API prompts a re-render on each update of the state and re-renders all components regardless. This is not always a good idea because the rendering should be only triggered at components with state updated in order to reduce unnecessary work for browsers. Redux seems to be smarter since it only re-renders the updated-state components. This can be monitored on the Redux's console, as there is a log in each state update, which is quite helpful when solving problems in a big and complicated project.

The truth, that states in Context API have to be created inside components by useState, will scale up the components with unrelated update-state functions. Since to update a state in specific cases, a developer needs to declare a corresponding function for each of them. The difference that Redux is considered as more successful is all actions to update state will be stored in a separate file, which enhances significantly the ability of management for the development team.

Overall, the Context API tends to serve better in a small scope of sharing states while Redux assists every state update clearer and easier to monitor, especially in huge projects

6 Conclusion

The target of this thesis is to analyse the differences of the state management solutions between Context API and Redux Hooks. Moreover, the thesis also points out the usage cases for each solution in the real practical projects. As a result, the study reached the expectation of comparing those state management solutions throughout 7 criteria: implementation, tracking changes, additional package installations, codebase complexity, resources consumption, processing speed and scalability. Besides creating the Me_Portfolio app to conduct the internal comparisons, the thesis also references many reliable outside applications to obtain the highest objectivity and persuasion in other comparisons.

During the comparisons, the context proves that it is more comfortable and flexible than Redux. Context API is a robust feature that performs nicely in maintenance and data flow understanding at a simple level inside small React projects. Besides that, it is an obstacle to monitor the changes of the global state while Redux gets strong support to solve the problem thanks to the Redux dev tools. However, it takes time to explore and practice Redux as well as its additional packages since Redux setup splits store, actions, actiontypes and reducers into their own files. This also explains the fact that the size of Redux codebase is always higher than Context's and Redux solutions consump more memory than Context's. Despite the issues, Redux technology performance is better than Context API that is because all components using state from Context API will be rerendered when the global states are modified. Therefore, the projects of Redux are more feasible to scale up rather than Context API.

In conclusion, the thesis proved that Context API will be a better choice for small projects or small scope of component tree while Redux is more matching to the projects of processing complex state data.

Even though the comparison gained quite good results in the scope of the thesis, it obviously needed more research in order that the outcome would be more precise and conniving

References

1 ReactJS History. Online. Education Ecosystem. <https://www.education-ecosystem.com/guides/programming/react-js/history>. Accessed 1 April 2021.

2 ReactJS - Overview. Online. Tutorialspoint. <https://www.tutorialspoint.com/reactjs/reactjs_overview.htm#:~:text=React%20is%20a%20library%20for,data%20that%20changes%20over%20time.&text=React%20can%20also%20render%20on,native%20apps%20using%20React%20Native.>. Accessed 1 April 2021.

3 React Hooks. Online. Javatpoint. <https://www.javatpoint.com/react-hooks#:~:text=Hooks%20are%20the%20new%20feature,lifecycle%20features%20from%20function%20components.&text=Also%2C%20it%20does%20not%20replace%20your%20knowledge%20of%20React%20concepts.>. Accessed 1 April 2021.

4 Fernando, Shalini. 2020. Using redux and Context API. Online. Codehouse. <https://www.codehousegroup.com/insight-and-inspiration/tech-stream/using-redux-and-context-api#:~:text=Context%20API%20prompts%20a%20re,a%20log%20in%20each%20component.>. Accessed 1 April 2021.

5 Banks, Alex & Porcello, Eve. 2020. Learning React. 2nd ed. Electronic book. O'Reilly Media, Inc.

6 Context. Online. Facebook.<https://reactjs.org/docs/context.html>. Accessed 1 April 2021.

7 Bachuk, Alex. 2016. Redux - An introduction. Online. Smashing Magazine.<https://www.smashingmagazine.com/2016/06/an-introduction-to-redux/>. Accessed 1 April 2021.

8 Getting started with redux. Online. Dan Abramov and the Redux documentation authors. <https://redux.js.org/introduction/getting-started>. Accessed 1 April 2021.

9 Garreau, Marc & Faurot, Will. 2018. Redux in Action. Electronic book. Manning Publications.

10 Redux-Data flow. Online. Tutorials Point. <https://www.tutorialspoint.com/redux/redux_data_flow.htm>. Accessed 1 April 2021.

11 Redux Fundamentals, Part 2: Concepts and Data Flow. Online. Reduxjs. <https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow>. Accessed 1 April 2021.

12 What Is Figma? a 101 Intro. Online. Compact Creative.<https://designshack.net/articles/software/what-is-figma-intro/>. Accessed 1 April 2021.

13 Visual Studio Code Overview. Online. Microsoft.<https://code.visualstudio.com/docs>. Accessed 1 April 2021.

14 Sridhar, Aditya. 2018. An introduction to Git: what it is, and how to use it. Online. freeCodeCam. <https://www.freecodecamp.org/news/what-is-git-and-how-to-use-it-c341b049ae61/>. Accessed 1 April 2021

15 Chrome DevTools Overview. 2016. Online. Google Developer. <https://developer.chrome.com/docs/devtools/overview/>. Accessed 1 April 2021.

16 React Developer Tools. Online. Facebook. <https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en#:~:text=React%20Developer%20Tools%20is%20a,in%20the%20Chrome%20Developer%20Tools.&text=This%20extension%20requires%20permissions%20to,not%20transmit%20any%20data%20remotely.>. Accessed 1 April 2021.

17 Redux - Devtools. Online. Tutorials Point. <https://www.tutorialspoint.com/redux/redux_devtools.htm>. Accessed 1 April 2021.

18 Getting Started. Online. Facebok. <https://reactjs.org/docs/getting-started.html>. Accessed 1 April 2021.

19 What is TypeScript. Online. TypeScript.<https://www.typescriptlang.org/>. Accessed 1 April 2021.

20 Redux Essentials, Part 1: Redux Overview and Concepts. Online. Dan Abramov and the Redux documentation authors. <https://redux.js.org/tutorials/essentials/part-1-overview-concepts>. Accessed 1 April 2021.

21 Redux DevTools. Online. Remotedevio. <https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd?hl=en#:~:text=Overview,It%27s%20an%20opensource%20project.>. Accessed 1 April 2021.

22 Redux Fundamentals, Part 6: Async Logic and Data Fetching. Online. Dan Abramov and the Redux documentation authors. <https://redux.js.org/tutorials/fundamentals/part-6-async-logic>. Accessed 1 April 2021.

23 Welcome to the Gatsby Way of Building. Online. Gatsbyjs.<https://www.gatsbyjs.com/docs/>. Accessed 1 April 2021.

24 FAQ / About Contentful. Online. Contentful. <https://www.contentful.com/faq/about-contentful/#what-is-contentful>. Accessed 1 April 2021.

25 Welcome to Netlify. Online. Netlify.<https://docs.netlify.com/>. Accessed 1 April 2021.

26 About Node.js®. Online. OpenJS Foundation.<https://nodejs.org/en/about/>. Accessed 1 April 2021.

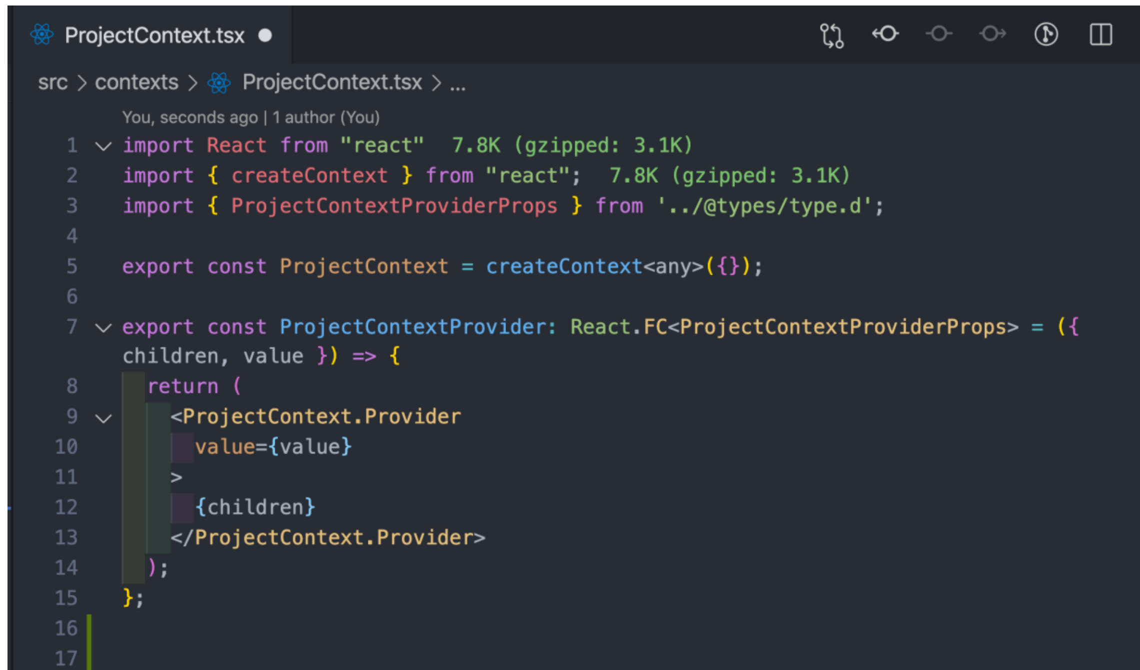
27 Redux Fundamentals, Part 3: State, Actions, and Reducers. Online. Dan Abramov and the Redux documentation authors.<https://redux.js.org/tutorials/fundamentals/part-3-state-actions-reducers>. Accessed 1 April 2021.

28 Redux Thunk. Online. GitHub. <https://github.com/reduxjs/redux-thunk#why-do-I-need-this>. Accesses 1 April 2021.

29 Advanced Search Tool. Online. Github.<https://github.com/search/advanced>. Accessed 1 April 2021.

30 Basques, Kayce. 2015. Fix memory problems. Online. Google developer.<https://developer.chrome.com/docs/devtools/memory-problems/>. Accesses 1 April 2021.

Appendix: Context API in Me_Portfolio project



ProjectContext.tsx

index.tsx

src > pages > index.tsx > IndexPage

You, seconds ago | 1 author (You)

```
import React from "react"  7.8K (gzipped: 3.1K)
import { useState } from "react"  7.8K (gzipped: 3.1K)
import Layout from "../components/Layout/Layout"
import "./HomePage.scss"
import { graphql } from "gatsby"
import { ProjectContextProvider } from "../contexts/ProjectContext"

const IndexPage: React.FC<any> = ({ data }) => {
  const {
    projects,
  } = data.indexPage

  projects.map((project: any, i: number) => {
    project.index = i
    project.title = project.projectName
    project.shortDesc = project.shortDescription.shortDescription
    project.fullDesc = project.fullDescription.fullDescription
    return project
  })

  const [projectList, setProjectList] = useState(projects)
  const [selectedProjectIndex, setSelectedProjectIndex] = useState(0)

  return (
    <ProjectContextProvider value={{
      data: data.indexPage,
      projectList, setProjectList,
      selectedProjectIndex, setSelectedProjectIndex
    }}>
      <Layout />
    </ProjectContextProvider>
  )
}

export default IndexPage

```

/pages/index.tsx

Appendix: Redux Hook in Me_Portfolio project

```
TS actionTypes.ts ●
src > store > indexPage > TS actionTypes.ts > ...
You, seconds ago | 1 author (You)
1  export const SET_DATA = "[PROJECT] SET_DATA"
2  export const SET_SELECTED_PROJECT = "[PROJECT] SET_SELECTED_PROJECT"
3
```

actionTypes.ts

```
TS actionCreators.ts ●
src > store > indexPage > TS actionCreators.ts > ...
You, seconds ago | 1 author (You)
1  import { store } from ".."
2  import * as actionTypes from "./actionTypes"
3
4  const setData = (data: any) => ({
5      type: actionTypes.SET_DATA,
6      data,
7  })
8
9  export const boundSetData = (data: any) => store.dispatch(setData(data))
10
11 const setSelectedProject = (projectIndex: number) => ({
12     type: actionTypes.SET_SELECTED_PROJECT,
13     projectIndex,
14 })
15
16 export const boundSetSelectedProject = (projectIndex: number) => store.dispatch
   (setSelectedProject(projectIndex))
17
```

actionCreators.ts

TS reducer.ts

src > store > indexPage > TS reducer.ts > default

You, seconds ago | 1 author (You)

```
import * as actionTypes from "./actionTypes"

const initialState = {
  projects: [],
  selectedProject: {}
}

const reducer = (
  state: any = initialState,
  action: any
) => {
  switch (action.type) {
    case actionTypes.SET_DATA:
      return action.data

    case actionTypes.SET_SELECTED_PROJECT:
      return { ...state, selectedProject: state.projects[action.projectIndex] }

    default:
      return state
  }
}

export default reducer

```

reducer.ts

```
TS index.ts ●
src > store > TS index.ts > ...
You, seconds ago | 1 author (You)
1   import { createStore, applyMiddleware, Store, Dispatch } from "redux"  5.4K (gzipp
2   import reducer from "./indexPage/reducer"
3   import { composeWithDevTools } from 'redux-devtools-extension';  800 (gzipped: 429
4   import reduxThunk from 'redux-thunk';  559 (gzipped: 328)
5
6   const composeEnhancers = composeWithDevTools({
7     trace: true,
8     traceLimit: 25
9   });
10
11  const initialState = {}
12  const middleware = [reduxThunk]
13
14  export const store: Store<any, any> & {
15    dispatch: Dispatch
16  } = createStore(reducer, initialState, composeEnhancers(applyMiddleware(...
    middleware)))
17
18
```

/store/index.js

```
gatsby-browser.js ×
gatsby-browser.js > ...
You, seconds ago | 1 author (You)
1   import React from 'react';  7.8K (gzipped: 3.1K)
2   import { Provider } from 'react-redux';  6.2K (gzipped: 2.4K)
3   import { store } from './src/store';
4
5   export const wrapRootElement = ({ element }) => {
6     return (
7       <Provider store={store}>{element}</Provider>
8     );
9   }
10
```

gatsby-browser.js

```
Card.tsx ●
src > components > Card > Card.tsx > ...
You, seconds ago | 1 author (You)
1   import React from "react"  7.8K (gzipped: 3.1K)
2   import "./Card.scss"
3   import { boundSetSelectedProject } from "../../store/indexPage/actionCreators"
4   import { smoothScroll } from "../scrolling"
5
You, 2 months ago | 1 author (You)
6   export interface CardProps {
7       title: string
8       shortDesc: string
9       index: number
10  }
11
12  const Card: React.FC<CardProps> = ({ title, shortDesc, index }) => {
13      const handleCardSelected = React.useCallback(
14          (index) => boundSetSelectedProject(index),
15          [boundSetSelectedProject]
16      )
17
18      return (
19          <div
20              className={`card card-background-${index + 1}`}
21              onClick={() => {
22                  handleCardSelected(index)
23                  document.getElementById("project--title")!.offsetTop <
24                      document.getElementById("project--img")!.offsetTop
25                      ? smoothScroll("project--title")
26                      : smoothScroll("project--img")
27              }}
28          >
29              <h3 className="card--title">{title}</h3>
30              <p className="card--shortDesc">{shortDesc}</p>
31          </div>
32      )
33  }
34
35  export default Card
36
37
```

Card.tsx