前端網站開發人員證書課程
(二) 進階網絡程式設計--專業React.js應用

8. Introduction to Redux

Presented by Krystal Institute

erb 僱員再培訓局 Employees Retraining Board
SCOPE School of Continuing and Professional Education 專業進修學院 香港城市大學 City University of Hong Kong
KRYSTAL INSTITUTE DIGITAL ECONOMY CORE TECHNOLOGY

Lesson Outline

- Introduce Redux and teach how to use it the right way
- Learn to make Redux applications using the tools and patterns learned before

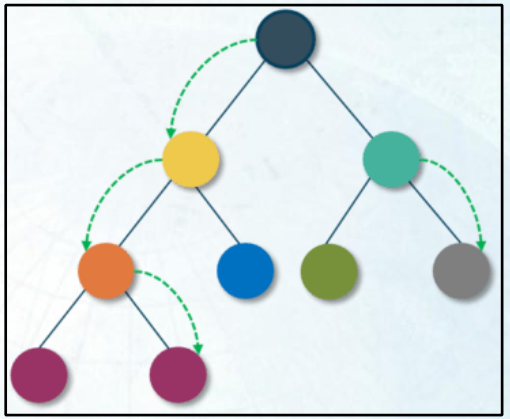
8.1 Why Redux with React?

8.1.1 Why Redux with React?

- React is one of the most popular JavaScript libraries which is used for front-end development.
- It has made our application development easier and faster by providing a component-based approach.
- As you might know, it's not the complete framework but the view part of the MVC (Model-View-Controller) framework.
- So, how do you keep track of the data and handle the events and state(data) in the applications developed using React? Well, this is where Redux comes as a savior and controls the data flow of the application from the backend.

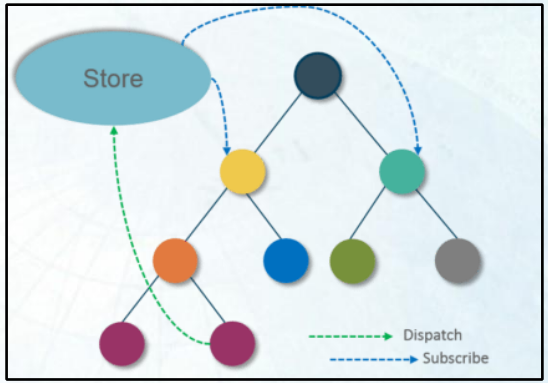
8.1.2 Parent to Child Communication

- React follows the component-based approach, where the data flows through the components.
- The data in React always flows from parent to child components which makes it unidirectional.
- Unidirectional data flow keeps our data organized and helps us in controlling the application better. Because of this, the application's state is contained in specific stores and as a result, the rest of the components remain loosely coupled.
- Unidirectional data flow makes our application more flexible leading to increased efficiency. That's why communication is convenient between parent and child components.



8.1.3 Non-parent Communication

- What happens when we try to communicate from a non-parent component?
- A child component can never pass data back up to the parent component but can update the parent component's state by using an event handler from parent to child.
- React does not provide any way for direct component-to-component communication.
- Redux provides a global "store" as a way for two non-parent components to pass data to each other.



8.1.3 Non-parent Communication

- A store is a place where you can store all your application state together.
- The components can “dispatch” state changes to the store and not directly to the other components.
- Then the components that need the updates about the state changes can “subscribe” to the store.
- Thus, with Redux, it becomes clear where the components get their state from as well as where should they send their states to.
- The component initiating the change does not have to worry about the list of components needing the state change and can simply dispatch the change to the store. This is how Redux makes the data flow easier.

8.1.4 What is Redux?

- Redux is also a library that is used widely for front-end development.
- Redux is a pattern and library for managing and updating application state, using events called "actions".
- Redux is basically a library for managing both data-state and UI-state in JavaScript applications.
- Redux separates the application data and business logic into its own container in order to let React manage just the view.
- Rather than a traditional library or a framework, redux is an application data-flow architecture.
- It is most compatible with **Single Page Applications (SPAs)** where the management of the states over time can get complex.
- Redux serves as a **centralized** store for a state that needs to be used across your entire application, with rules ensuring that the state can only be updated in a predictable fashion.

8.1.5 Why and When to Use Redux?

- Redux helps you manage the "global" state - state that is needed across many parts of your application.
- The patterns and tools provided by Redux make it easier to understand when, where, why, and how the state in your application is being updated, and how your application logic will behave when those changes occur.
- Redux guides you toward writing code that is predictable and testable, which helps give you confidence that your application will work as expected.
- You have large amounts of application state that are needed in many places in the app
- The app state is updated frequently over time
- The logic to update that state may be complex
- The app has a medium or large-sized codebase and might be worked on by many people

8.2 Redux Libraries and Tools

8.2 Redux Libraries and Tools

Redux is a small standalone JS library. However, it is commonly used with several other packages:

- React-Redux:
  - Redux can integrate with any UI framework and is most frequently used with React.
  - React-Redux is the official package that lets react components interact with the redux store by reading pieces of state and dispatching actions to update the store.

Redux-Saga:

- Redux Saga is a middleware library used to allow a Redux store to interact with resources outside of itself asynchronously.
- Redux Saga includes making HTTP requests to external services, accessing browser storage, and executing I/O operations. These operations are also known as side effects.

8.2 Redux Libraries and Tools

Redux Toolkit:

- Redux Toolkit is recommended approach for writing Redux logic.
- Redux contains packages and functions that we think are essential for building a Redux app.
- Redux Toolkit builds in our suggested best practices, simplifies most Redux tasks, prevents common mistakes, and makes it easier to write Redux applications.

Redux DevTool Extension:

- The Redux DevTool Extension shows a history of the changes to the state in your Redux store over time.
- The Redux DevTool Extension allows you to debug your applications effectively, including using powerful techniques like “time-travel debugging”

8.3 Immutability

8.3 Immutability

- Mutable" means "changeable". If something is "immutable", it can never be changed.
- JavaScript objects and arrays are all mutable by default.
- If I create an object, I can change the contents of its fields. If I create an array, I can change the contents as well:

```
const obj = { a: 1, b: 2 }
// still the same object outside, but the contents have changed
obj.b = 3

const arr = ['a', 'b']
// In the same way, we can change the contents of this array
arr.push('c')
arr[1] = 'd'
```

8.3 Immutability

- This is called *mutating* the object or array. It's the same object or array reference in memory, but now the contents inside the object have changed.
- In order to update values immutably, your code must make copies of existing objects/arrays, and then modify the copies.
- We can do this by hand using JavaScript's array/object spread operators, as well as array methods that return new copies of the array instead of mutating the original array:
- Redux expects that all state updates are done *immutably*. We'll look at where and how this is important a bit later, as well as some easier ways to write immutable update logic.

```
const obj = {
  a: {
    // To safely update obj.a.c, we have to copy each piece
    c: 3
  },
  b: 2
}

const obj2 = {
  // copy obj
  ...obj,
  // overwrite a
  a: {
    // copy obj.a
    ...obj.a,
    // overwrite c
    c: 42
  }
}

const arr = ['a', 'b']
// Create a new copy of arr, with "c" appended to the end
const arr2 = arr.concat('c')

// or, we can make a copy of the original array:
const arr3 = arr.slice()
// and mutate the copy:
arr3.push('c')
```

8.4 Important Terminologies

8.4 Important Terminologies

There are some important Redux terms that you'll need to be familiar with before getting started with the app.

Actions:

- An action is a plain JavaScript object that has a type field.
- Actions are objects that have a **type** property and any other data that it needs to describe the action.
- You can think of an action as an event that describes something that happened in the application.
- The type field should be a string that gives this action a descriptive name.
- We usually write that type string like "*domain/eventName*", where the first part is the feature or category that this action belongs to, and the second part is the specific thing that happened.
- An action object can have other fields with additional information about what happened. By convention, we put that information in a field called *payload*.

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

8.4 Important Terminologies

Action Creators:

- An action creator is a function that creates and returns an action object.
- We typically use these so we don't have to write the action object by hand every time:

```
const addTodo = text => {
  return {
    type: 'todos/todoAdded',
    payload: text
  }
}
```

8.4 Important Terminologies

Reducers:

- A reducer is a function that receives the current *state* and an *action* object, decides how to update the state if necessary, and returns the new state: (*state, action*) => *newState*.
- You can think of a reducer as an event listener which handles events based on the received action (event) type.
- "Reducer" functions get their name because they're similar to the kind of callback function you pass to the Array.reduce() method.
- Reducer should only calculate the new state value based on the *state* and *action* arguments.
    - state: A state is an object that describes aspects of your application that can change over time. You should make your state the minimal amount of data that is required to describe the state of the application.
    - action: Actions are objects that have a type property and any other data that it needs to describe the action.

8.4 Important Terminologies

Reducers:

- Reducers are not allowed to modify the existing state. Instead, they must make immutable updates, by copying the existing state and making changes to the copied values.
- Reducer must not do any asynchronous logic, calculate random values, or cause other "side effects".

```
const initialState = { value: 0 }

function counterReducer(state = initialState, action) {
  // Check to see if the reducer cares about this action
  if (action.type === 'counter/increment') {
    // If so, make a copy of `state`
    return {
      ...state,
      // and update the copy with the new value
      value: state.value + 1
    }
  }
  // otherwise return the existing state unchanged
  return state
}
```

8.4 Important Terminologies

Store:

- The Store is the center of every Redux application.
- A "store" is a container that holds your application's global state.
- A store is a JavaScript object with a few special functions and abilities that make it different than a plain global object.

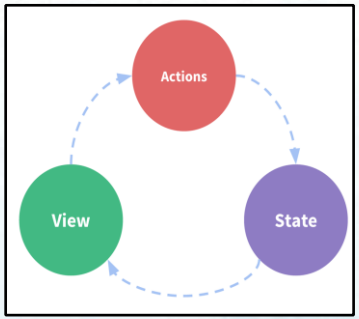
Specific functions and abilities of Store:

- You must never directly modify or change the state that is kept inside the Redux store
- Instead, the only way to cause an update to the state is to create a plain **action** object that describes "something that happened in the application", and then **dispatch** the action to the store to tell it what happened.
- When an action is dispatched, the store runs the root **reducer** function and lets it calculate the new state based on the old state and the action.

8.4 Important Terminologies

- Finally, the store notifies **subscribers** that the state has been updated so the UI can be updated with the new data.

```
import { configureStore } from '@reduxjs/toolkit'

const store = configureStore({ reducer: counterReducer })

console.log(store.getState())
// {value: 0}
```



8.4 Important Terminologies

Dispatch:

- The Redux store has a method called dispatch.
- The only way to update the state is to call the *store.dispatch()* and pass in an action object.
- The store will run its reducer function and save the new state value inside.

```
store.dispatch({ type: 'counter/increment' })

console.log(store.getState())
// {value: 1}
```

```
const increment = () => {
  return {
    type: 'counter/increment'
  }
}

store.dispatch(increment())

console.log(store.getState())
// {value: 2}
```

8.4 Important Terminologies

Selectors:

- Selectors are functions that know how to extract specific pieces of information from a store state value.
- As an application grows bigger, this can help avoid repeating logic as different parts of the app need to read the same data.

```
const selectCounterValue = state => state.value

const currentValue = selectCounterValue(store.getState())
console.log(currentValue)
// 2
```

8.5 Redux Application Data Flow

8.5 Redux Application Data Flow

Data Flow

- Sequence of steps to update the react app:
  - The **state** describes the condition of the app at a specific point in time
  - The UI is rendered based on that state.
  - When something happens (such as a user clicking a button), the state is updated based on what occurred.
  - The UI re-renders based on the new state.
- For Redux specifically, we can break these steps into more detail:
- Initial setup:
  - A Redux store is created using a root reducer function
  - The store calls the root reducer once and saves the return value as its initial state
  - When the UI is first rendered, UI components access the current state of the Redux store, and use that data to decide what to render. They also subscribe to any future store updates so they can know if the state has changed.

8.5 Redux Application Data Flow

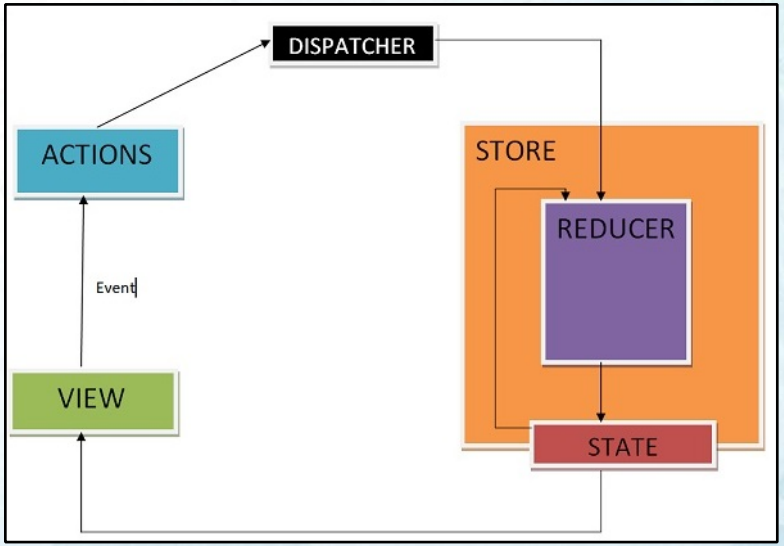
Data Flow

- For Redux specifically, we can break these steps into more detail:
- Updates:
  - Something happens in the app, such as a user clicking a button.
  - The app code dispatches an action to the Redux store.
  - The store runs the reducer function again with the previous state and the current action and saves the return value as the new state.
  - The store notifies all parts of the UI that are subscribed that the store has been updated.

8.5 Redux Application Data Flow

Data Flow

- Each UI component that needs data from the store checks to see if the parts of the state they need have changed.
- Each component that sees its data has changed forces a re-render with the new data, so it can update what's shown on the screen.



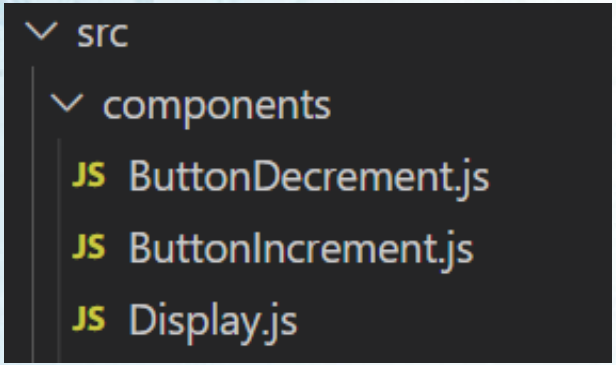
8.6 Redux Application

8.6 Redux Application

Initial Setup

Let's get started with the application. For this tutorial we have picked the Counter app that is done as an assignment in previous class.

- We have three components in this app which is rendered in the App component:



- Open the app in the Visual studio code editor.

8.6 Redux Application

Initial Setup

- Get into the app folder and install redux, react-redux, redux-thunk, and redux toolkit.

```
C:\workspace\react tutorial\create react app>cd redux-tutorial

C:\workspace\react tutorial\create react app\redux-tutorial>npm i redux react-redux redux-thunk
```

```
C:\workspace\react tutorial\create react app\redux-tutorial>npm i @reduxjs/toolkit
[..................] - idealTree:redux-tutorial: sill idealTree buildDeps
```

8.6 Redux Application

Initial Setup

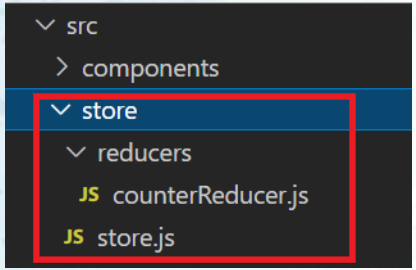
- Open package.json and view the dependencies after the installation of libraries.

```
"name": "redux-tutorial",
"version": "0.1.0",
"private": true,
"dependencies": {
  "@reduxjs/toolkit": "^1.8.3",
  "@testing-library/jest-dom": "^5.16.4",
  "@testing-library/react": "^13.3.0",
  "@testing-library/user-event": "^13.5.0",
  "react": "^18.2.0",
  "react-dom": "^18.2.0",
  "react-redux": "^8.0.2",
  "react-scripts": "5.0.1",
  "redux": "^4.2.0",
  "redux-thunk": "^2.4.1",
  "web-vitals": "^2.1.4"
},
```

8.6 Redux Application

Write Reducer

- Reducers are functions that take the current state and action as arguments, and return new state result. (state, action) => newState.
- Create a folder structure and create a new reducer file “counterReducer.js”.



- Create an initial state for the display value.

```
const initialState = {
    value: 0
};
```

8.6 Redux Application

Write Reducers

- Create a “reducer” function that determines what the new state should be when the increment and decrement button is clicked depending on the action type.

```
function counterReducer(state = initialState, action) {
    switch (action.type) {
        case "counter/incremented":
            return { ...state, value: state.value + 1 };
        case "counter/decremented":
            return { ...state, value: state.value - 1 };
        default:
            return state;
    }
}
```

- Export the default function.

```
export default counterReducer;
```

8.6 Redux Application

counterReducer.js

src > store > reducers > JS counterReducer.js > ...

```
1  const initialState = {
2      value: 0
3  };
4
5  function counterReducer(state = initialState, action) {
6      switch (action.type) {
7          case "counter/incremented":
8              return { ...state, value: state.value + 1 };
9          case "counter/decremented":
10             return { ...state, value: state.value - 1 };
11         default:
12             return state;
13     }
14 }
15
16 export default counterReducer;
```

8.7 ConfigureStore

8.7 ConfigureStore

- A friendly abstraction over the standard Redux **createStore** function that adds good defaults to the store setup for a better development experience.
- **configureStore()** method that simplifies the store setup process. **configureStore()** wraps around the Redux library's **createStore()** method and the **combineReducers()** method, and handles most of the store setup for us automatically.
    - **createStore():** Creates a Redux store that holds the complete state tree of your app.
    - **combineReducers():** This function helps you organize your reducers to manage their own slices of state, similar to how you would have different Flux Stores to manage different state. With Redux, there is just one store, but combineReducers helps you keep the same logical division between reducers.

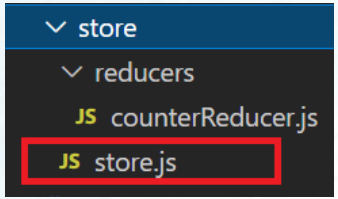
8.7 ConfigureStore

- Parameters: **configureStore** accepts a single configuration object parameter, with the following options:
  - reducer.
  - middleware.
  - devTools.
  - trace.
  - preloadedState.
  - enhancers.
- reducer:
  - If this is a single function, it will be directly used as the root reducer for the store.
  - If it is an object of slice reducers, like **{users : usersReducer, posts : postsReducer}**, configureStore will automatically create the root reducer by passing this object to the Redux **combineReducers** utility.

8.7 ConfigureStore

Write Store

- Switch to store.js and import configureStore from redux Toolkit.




```
import { configureStore } from '@reduxjs/toolkit';
```

- Import counterReducer.

```
import counterReducer from './reducers/counterReducer';
```

- Create a store, where all the reducers in the app should be included as an object.

```
const store = configureStore({
    reducer: {
        counter: counterReducer
    }
})
```

- Export the store.

```
export default store;
```

8.7 ConfigureStore

Store.js

src > store > JS store.js > ...

```
1  import { configureStore } from '@reduxjs/toolkit';
2
3  import counterReducer from './reducers/counterReducer';
4
5  const store = configureStore({
6      reducer: {
7          counter: counterReducer
8      }
9  })
10
11 export default store;
```

8.8 Provider

8.8 Provider

- The <Provider> component makes the Redux store available to any nested components that need to access the Redux store.
- Since any React component in a React Redux app can be connected to the store, most applications will render a <Provider> at the top level, with the entire app's component tree inside of it.
- React uses provider pattern in Context API to share data across the tree descendant nodes.
- Provider is a component given to us to use from the react-redux node package.
- We use Provider in order to pass the store as an attribute.
- By passing the store as an attribute in the Provider component (<Provider store={store}>), we are avoiding having to store the store as props.

```
<Provider store={store}>
  <App />
</Provider>
```

8.8 Provider

Provide the store to the counter Components

- Import provider from react-redux library.

```
import { Provider } from 'react-redux';
```

- Import store.

```
import store from './store/store';
```

- Remove the useState hooks and increment and decrement functions.
- Wrap the counter components with provider by passing the store <Provider store={store}>.

```
<Provider store={store} >
  <ButtonIncrement />
  <Display />
  <ButtonDecrement />
</Provider>
```

8.8 Provider

App.js

src > JS App.js > ...

```
1  import { Provider } from 'react-redux';
2
3  import store from './store/store';
4
5  import Display from './components/Display';
6  import ButtonIncrement from './components/ButtonIncrement';
7  import ButtonDecrement from './components/ButtonDecrement';
8
9  import './App.css';
10
11 function App() {
12   return (
13     <Provider store={store} >
14       <ButtonIncrement />
15       <Display />
16       <ButtonDecrement />
17     </Provider>
18   );
19 }
20
21 export default App;
```

8.9 useSelector and useDispatch Hooks

8.9 useSelector and useDispatch Hooks

useSelector

- Allows you to extract data from the Redux store state, using a selector function.

```
const result: any = useSelector(selector: Function, equalityFn?: Function)
```

- The selector will be called with the entire Redux store state as its only argument.
- The selector will be run whenever the function component renders (unless its reference hasn't changed since a previous render of the component so that a cached result can be returned by the hook without re-running the selector).
- *useSelector()* will also subscribe to the Redux store, and run your selector whenever an action is dispatched.

8.9 useSelector and useDispatch Hooks

useDispatch

- *useDispatch()* returns a reference to the dispatch function from the Redux store. You may use it to dispatch actions as needed.

```
const dispatch = useDispatch()
```

- When passing a callback using dispatch to a child component, you may sometimes want to memoize it with useCallback.
- If the child component is trying to optimize render behavior using React.memo() or similar, this avoids unnecessary rendering of child components due to the changed callback reference.
- The **dispatch** function reference will be stable as long as the same store instance is being passed to the <Provider>. Normally, that store instance never changes in an application.

8.9 useSelector and useDispatch Hooks

Updating the Display component

- Import useSelector hook from react-redux.

```
import { useSelector } from 'react-redux';
```

- Use the selector to access the store value.

```
const counter = useSelector(state => state.counter.value) ?? 0;
```

state.counter.value → value in the initialState in counterReducer.js

```
const initialState = {
    value: 0
};
```

8.9 useSelector and useDispatch Hooks

Display.js

src > components > JS Display.js > ...

```
1   import { useSelector } from 'react-redux';
2
3   const Display = () => {
4       const counter = useSelector(state => state.counter.value) ?? 0;
5
6       return (<label style={{ marginLeft: '.5rem' }} >{counter}</label>)
7   }
8
9   export default Display;
```

8.9 useSelector and useDispatch Hooks

Updating Increment and Decrement components

- Import useDispatch from react-redux.

```
import { useDispatch } from 'react-redux';
```

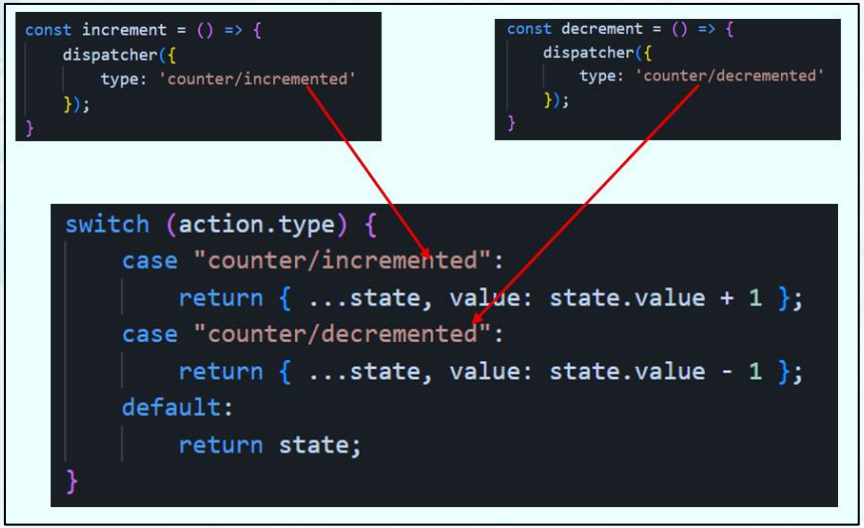
- Create an instance for useDispatch hook.

```
const dispatcher = useDispatch();
```

- Create a function to handle the increment/decrement, where the action type should be passed.
- The action type should match the type passed in the switch case of the counterReducer.

8.9 useSelector and useDispatch Hooks

Updating Increment and Decrement components



- Invoke the increment/decrement handle function on the click event of button

```
return (<button style={{ marginLeft: '.5rem' }} onClick={decrement} >
```

8.9 useSelector and useDispatch Hooks

ButtonIncrement.js

src > components > JS ButtonIncrement.js > ...

```
1   import { useDispatch } from 'react-redux';
2
3   const ButtonIncrement = () => {
4
5       const dispatcher = useDispatch();
6
7       const increment = () => {
8           dispatcher({
9               type: 'counter/incremented'
10          });
11      }
12
13      return (
14          <button style={{ marginLeft: '.5rem' }} onClick={increment} >
15              +1
16          </button>
17      )
18
19  }
20
21  export default ButtonIncrement;
```

8.9 useSelector and useDispatch Hooks

ButtonDecrement.js

src > components > JS ButtonDecrement.js > ...

```
1   import { useDispatch } from 'react-redux';
2
3   const ButtonDecrement = () => {
4
5       const dispatcher = useDispatch();
6
7       const decrement = () => {
8           dispatcher({
9               type: 'counter/decremented'
10          });
11      }
12
13      return (<button style={{ marginLeft: '.5rem' }} onClick={decrement} >
14          -1
15      </button>)
16  }
17
18  export default ButtonDecrement;
```

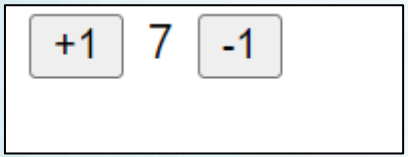
8.9 useSelector and useDispatch Hooks

Run the App

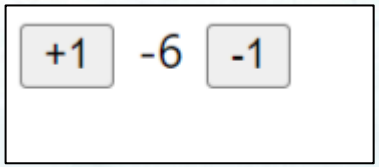
- Run the app and view the output.

```
C:\workspace\react tutorial\create react app\redux-tutorial>npm start

> redux-tutorial@0.1.0 start
> react-scripts start

(node:11640) [DEP_WEBPACK_DEV_SERVER_ON_AFTER_SETUP_MIDDLEWARE] DeprecationWar
(Use `node --trace-deprecation ...` to show where the warning was created)
(node:11640) [DEP_WEBPACK_DEV_SERVER_ON_BEFORE_SETUP_MIDDLEWARE] DeprecationWa
n.
Starting the development server...
Compiled successfully!
```

When incremented



When Decrement



8.10 Assignment

8.10 Assignment

Outline:

To enhance your counter app by adding two special functionality “Increment If Even” and “Decrement If Odd”.