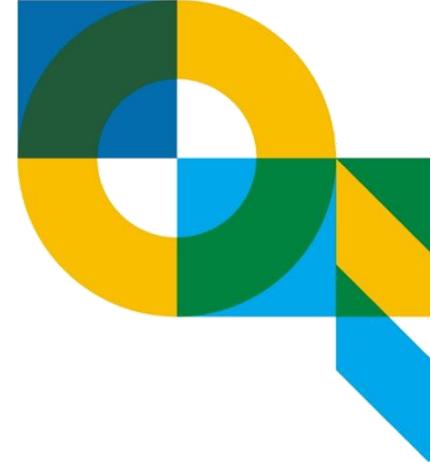




國立高雄科技大學
National Kaohsiung University of Science and Technology

智慧商務系
Department of Intelligent Commerce



11. 狀態管理 Redux

State Management - Redux

廖奕雯

yiwen923@nkust.edu.tw

Redux



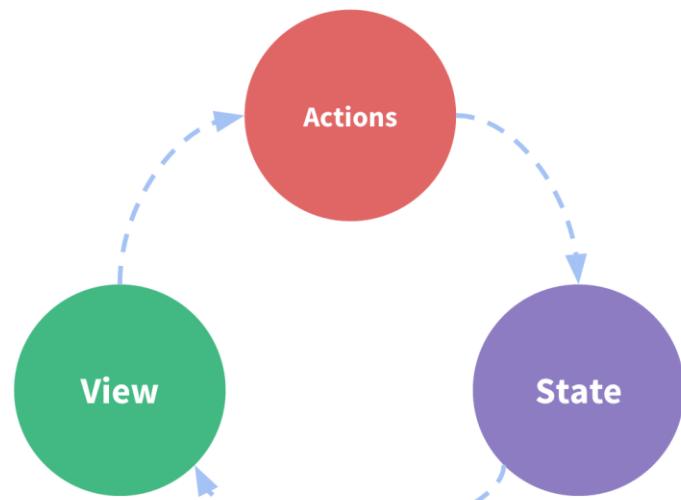
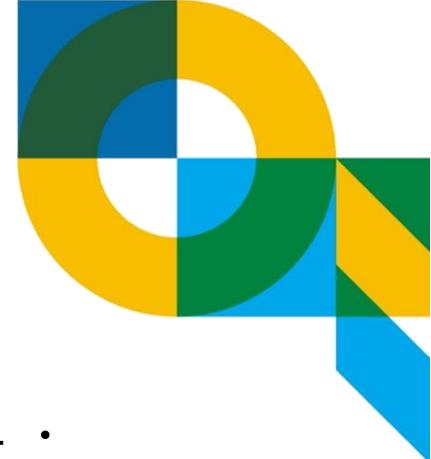
- Redux 是 JavaScript 應用的 state container，提供可預測的 state 管理。
- Redux 是使用 “**actions**” 的事件去管理和更新應用 state 的模式和工具庫。以**集中式 Store** (centralized store) 的方式對整個應用中使用的 state 進行集中管理，其規則確保狀態只能以**可預測**的方式更新。
- Redux 協助管理 **global state** - 應用程式中的許多元件都需要的使用到的 State。

Redux 基礎



- 所有 Redux 應用的中心都是 **store** 。
- “store” 是保存應用程式的 全域(global) state 的容器(container) 。
- store 是一個 JavaScript 物件，具有一些特殊的功能，使其與普通的全域物件不同：
 - 切勿直接修改 (modify) 或更改 (change) 保存在 Redux 存儲中的 state
 - 相反，導致 state 更新的唯一方法是創建一個 statement “應用程式中發生的某些事情”的普通 action 物件，然後將該 action dispatch 到 store 以告訴它發生了什麼或是要做什麼。
 - 當一個 action 被 dispatch 後，store 會呼叫 root reducer 方法，讓其根據 action 和舊 state 計算出新 state
 - 最後，store 會通知 訂閱者(subscribers) state 已更新，以便可以使用新資料更新 UI 。

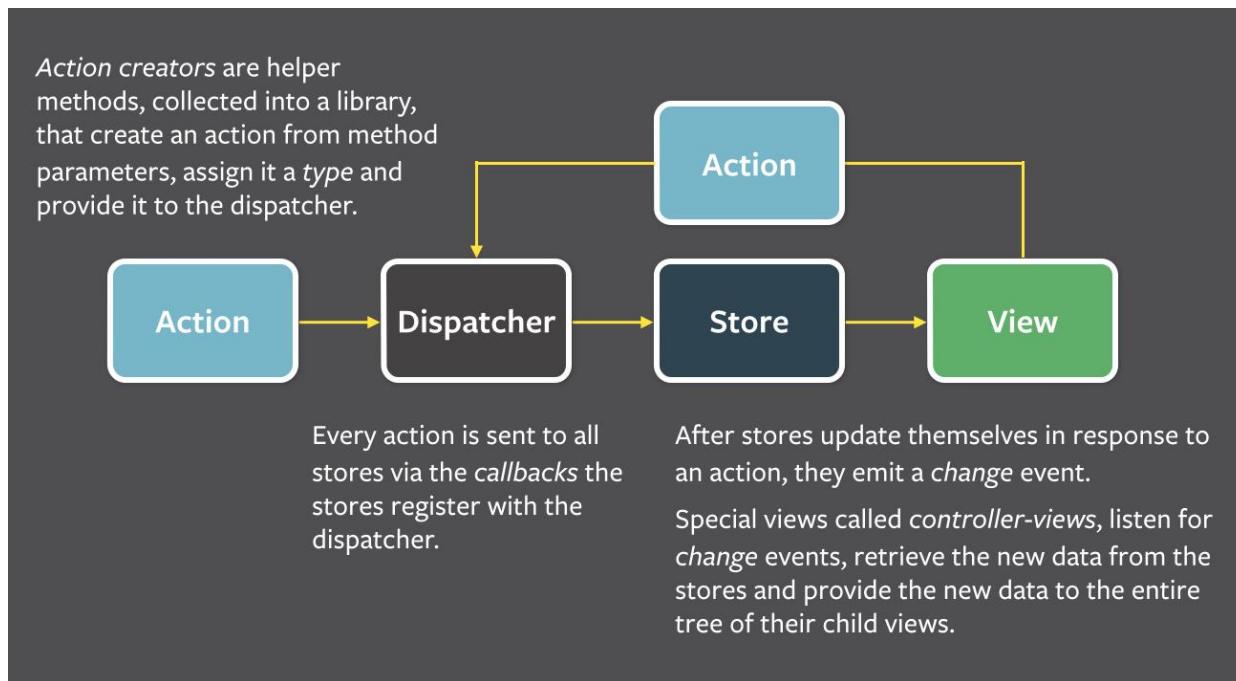
Unidirectional data flow



- “Unidirectional data flow”的例子：
 1. State 紀錄應用程式在特定時間點的狀況
 2. 基於 state 來渲染 UI
 3. 發生某些事件時（例如使用者按一下按鈕），state 會根據發生的事件進行更新
 4. 基於新的 state 重新渲染 UI
- 然而，當我們有多個元件需要**共用和使用相同 state**時，可能會變得很複雜，尤其是當這些元件位於應用程式的不同部分時。



Unidirectional data flow



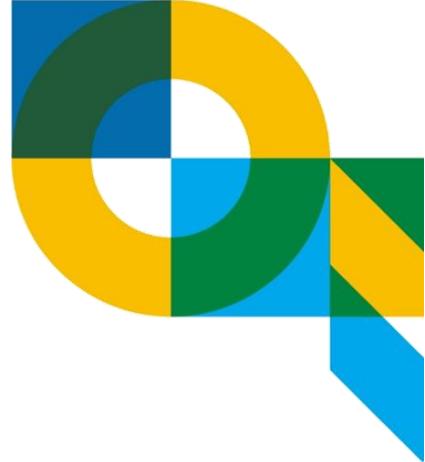
- 整個流程如下：
 1. 首先要有 action，通過定義一些 action creator 方法根據需要創建 Action 提供給 dispatcher
 2. View 層通過用戶交互（比如 onClick）會觸發 Action
 3. Dispatcher 會分發觸發的 Action 給所有註冊的 Store 的 callbacks function
 4. Store callbacks function 根據接收的 Action 更新自身資料之後會觸發一個 change 事件通知 View 資料更改了
 5. View 會監聽這個 change 事件，拿到對應的新資料並調用 setState 更新組件 UI
- 所有的狀態都由 Store 來維護，通過 Action 傳遞資料，構成了如上所述的單向資料流程迴圈，所以應用中的各部分分工就相當明確，高度解耦了。

什麼時候應該使用 Redux ?



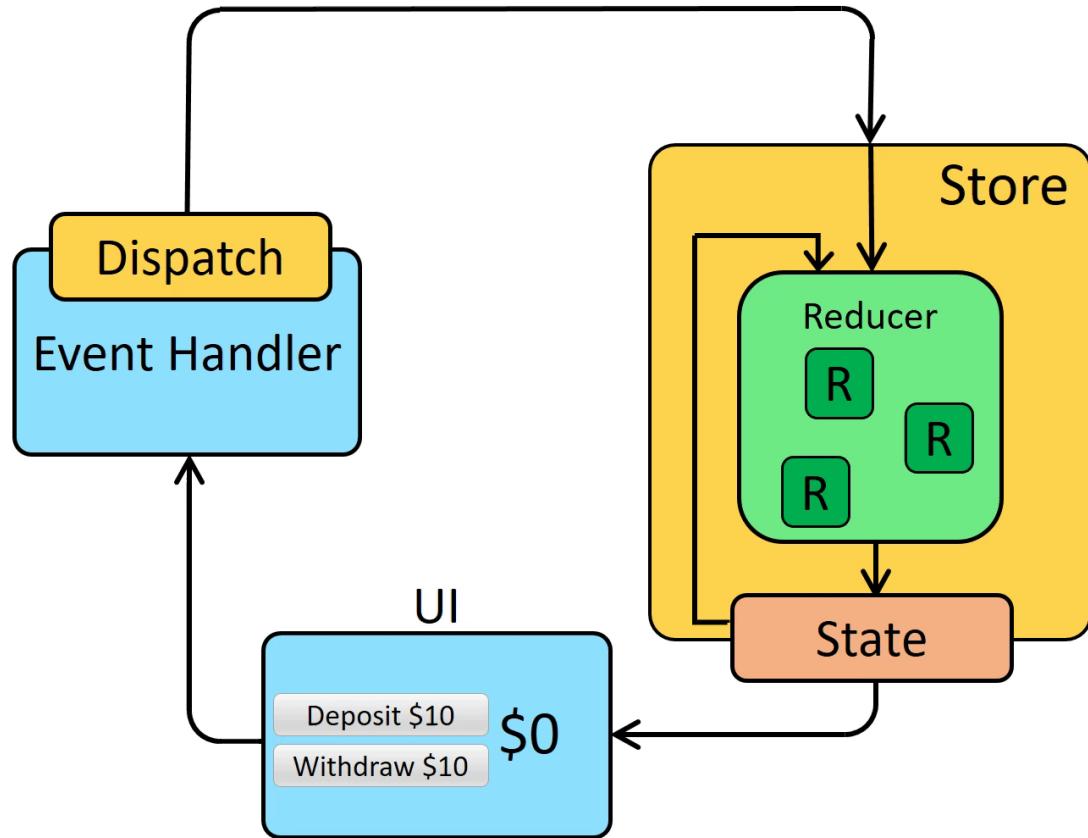
- Redux 可以處理管理與共用 state ，但與任何工具一樣，它也有權衡。
- 代價包含: 有更多的概念需要學習，還有更多的代碼需要編寫。它還為你的代碼添加了一些額外代碼，並要求你遵循某些限制。這是短期和長期生產力之間的權衡。
- Redux 在以下情況下更有用：
 - 在 App 的大量地方，都存在大量的state
 - State 會隨著時間的推移而頻繁更新
 - 更新 state 的邏輯可能很複雜
 - 中型和大型代碼量的應用，很多人協同開發
- 並非所有應用程式都需要 Redux 。花一些時間思考你正在構建的應用程式類型，並決定哪些工具最能幫助解決你正在處理的問題。

Redux component



- Action: 描述應用程式中發生了什麼事件。
- Reducers: 事件監聽器，根據接收到的 action (事件) 類型處理事件。
- Store: Redux 應用的 state 存在於一個名為 store 的對象中。store 是通過傳入一個 reducer 來創建的，並且有一個名為 getState 的方法，它返回當前 state 值。
- Dispatch: 更新 state 的唯一方法是呼叫 store.dispatch() 並傳入一個 action 對象。
- Selectors: Selector 函數可以從 store 狀態樹中提取指定的片段。隨著應用變得越來越大，會遇到應用程式的不同元件需要讀取相同的資料，selector 可以避免重複這樣的讀取邏輯。

Data Flow Example



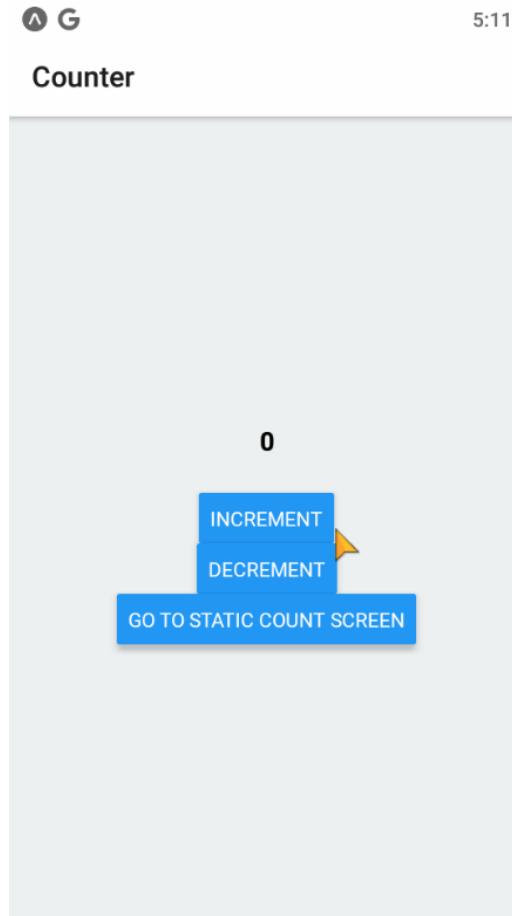
1. 初始啟動：

- 使用最頂層的 root reducer 函式建立 Redux store
- store 呼叫一次 root reducer，並將返回值保存為它的初始 state
- 當 view 首次 rendering 時，view 元件訪問 Redux store 的當前 state，使用該資料來決定要呈現的內容。同時監聽 store 的更新，以便他們可以知道 state 是否已更改。

2. 更新環節：

- 應用程式中發生了某些事件，例如使用者按一下按鈕
- dispatch 一個 action 到 Redux store，例如 `dispatch({type: 'counter/increment'})`
- store 用之前的 state 和當前的 action 再次運行 reducer 函數，並將返回值保存為新的 state
- store 通知所有訂閱過的view，通知它們 store 發生更新
- 每個訂閱過 store 資料的 view 元件都會檢查它們需要的 state 部分是否被更新。
- 發現 state 被更新的每個元件都強制使用新資料重新 rendering，緊接著更新畫面

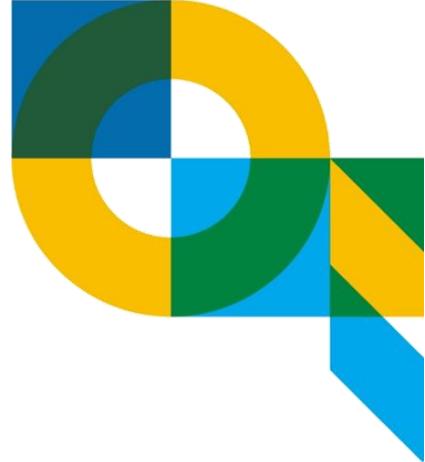
整合 React Navigation 與 Redux



範例目標:

1. 整合 React Navigation 與 Redux
2. 不同頁面讀取相同的 state 值

環境安裝

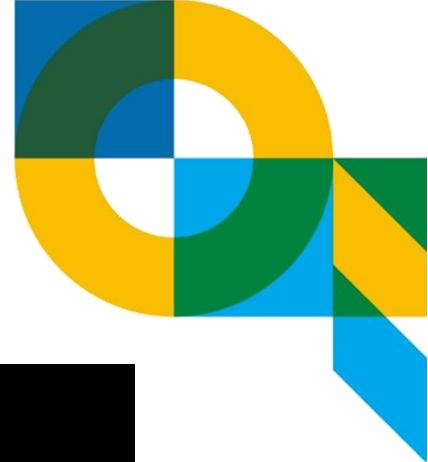


- 安裝套件

- npm install redux
- npm install react-redux
- npm install redux-devtools
- **npm install redux-devtools --force**

- npm install redux-thunk
- **npm install redux-thunk --force**

修改 app() 使所有的 view 共用 state



```
01: import { Provider } from 'react-redux';
02: import { NavigationContainer } from '@react-navigation/native';
03:
04: // Render the app container component with the provider around it
05: export default function App() {
06:   return (
07:     <Provider store={store}>
08:       <NavigationContainer>
09:         {/* Screen configuration */}
10:       </NavigationContainer>
11:     </Provider>
12:   );
13: }
```

- 將 App() 的 render 改寫成這樣。
- 所有的 view 需要包含在 Provider裡面，才能夠共用 state

建立 view 與 state 的關聯



一個普通的 view 的 component :

```
1: function Counter({ value }) {  
2:   return <Text>Count: {value}</Text>;  
3:  
4:  
5: const CounterContainer = connect(state => ({ value: state.count }))(Counter);
```

建立 view 與 state 的關聯

其中會使用到 state

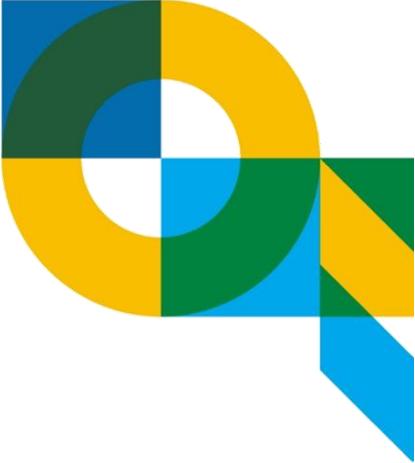
第一個參數是要關聯的 state

使用 connect function 來建立關聯。

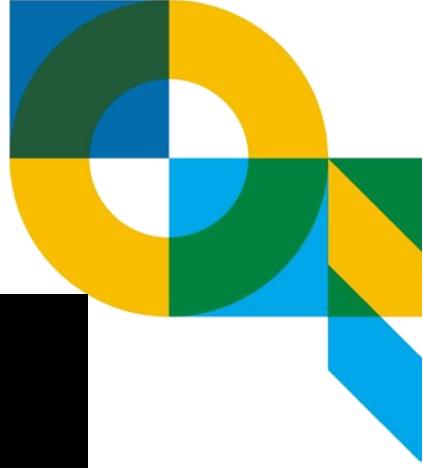
第二個參數是要關聯的 view

App()

```
File: App.js
63: // Create our stack navigator
64: let RootStack = createNativeStackNavigator();
65:
66: // Render the app container component with the provider around it
67: export default function App() {
68:   return (
69:     <Provider store={store}>
70:       <NavigationContainer>
71:         <RootStack.Navigator>
72:           <RootStack.Screen name="Counter" component={CounterContainer} />
73:           <RootStack.Screen
74:             name="StaticCounter"
75:             component={StaticCounterContainer}
76:           />
77:         </RootStack.Navigator>
78:       </NavigationContainer>
79:     </Provider>
80:   );
81: }
```



建立 store 和 reducer

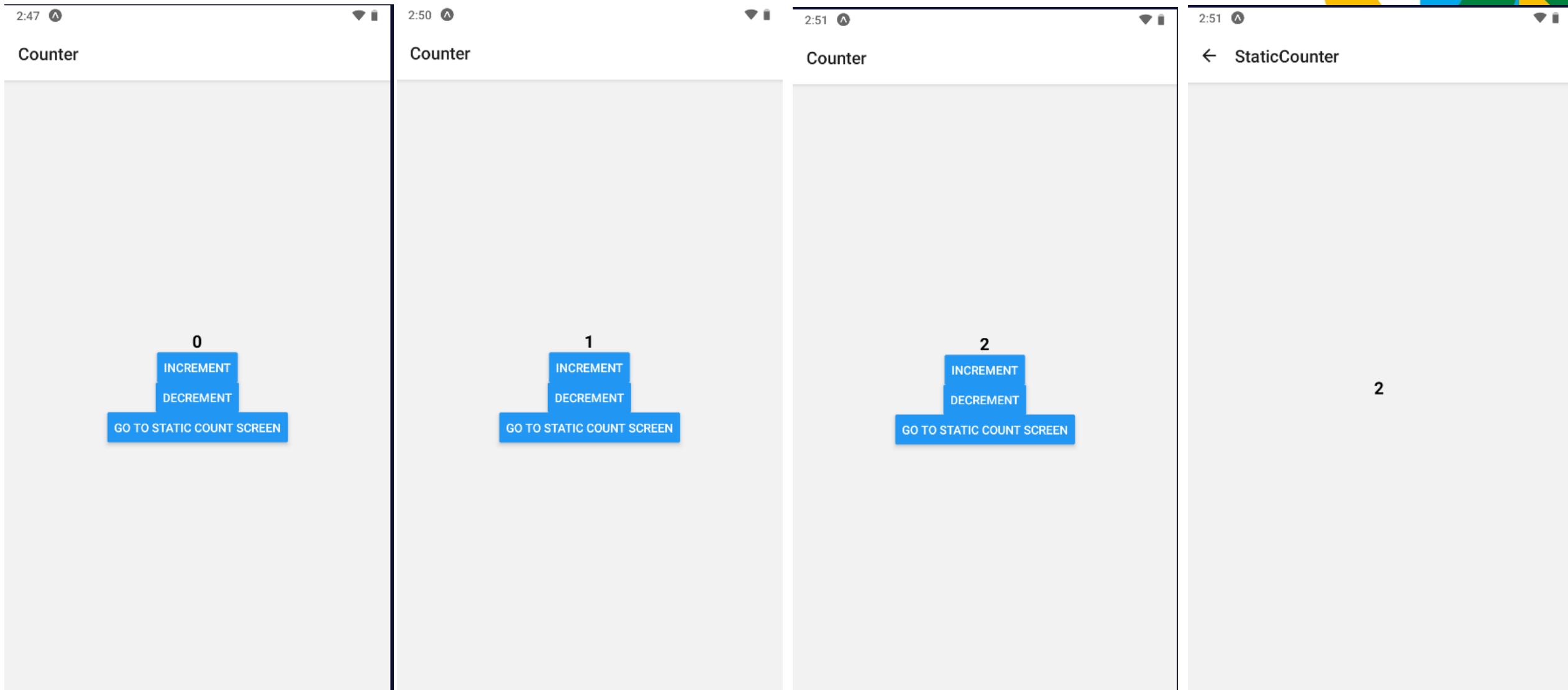


```
File: App.js
08: // A very simple reducer
09: function counter(state, action) {
10:   if (typeof state === 'undefined') {
11:     return 0;
12:   }
13:
14:   switch (action.type) {
15:     case 'INCREMENT':
16:       return state + 1;
17:     case 'DECREMENT':
18:       return state - 1;
19:     default:
20:       return state;
21:   }
22: }
23:
24: // A very simple store
25: let store = createStore(combineReducers({ count: counter }));
```

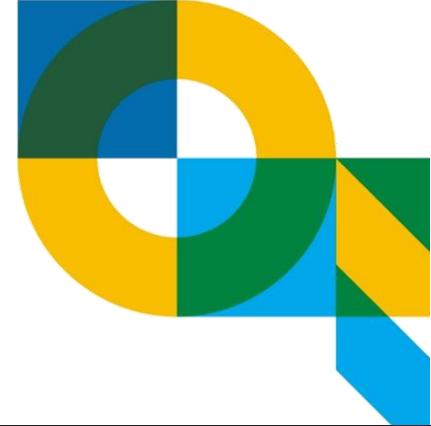
File: App.js

```
27: // A screen!
28: function Counter({ count, dispatch, navigation }) {
29:   return (
30:     <View style={styles.container}>
31:       <Text style={styles.paragraph}>{count}</Text>
32:       <Button
33:         title="Increment"
34:         onPress={() => dispatch({ type: 'INCREMENT' })}
35:       />
36:       <Button
37:         title="Decrement"
38:         onPress={() => dispatch({ type: 'DECREMENT' })}
39:       />
40:       <Button
41:         title="Go to static count screen"
42:         onPress={() => navigation.navigate('StaticCounter')}
43:       />
44:     </View>
45:   );
46: }
47:
48: // Another screen!
49: function StaticCounter({ count, dispatch }) {
50:   return (
51:     <View style={styles.container}>
52:       <Text style={styles.paragraph}>{count}</Text>
53:     </View>
54:   );
55: }
56:
57: // Connect the screens to Redux
58: let CounterContainer = connect((state) => ({ count: state.count }))(Counter);
59: let StaticCounterContainer = connect((state) => ({ count: state.count }))(StaticCounter)
60: );
61: );
```

App_Navigation2



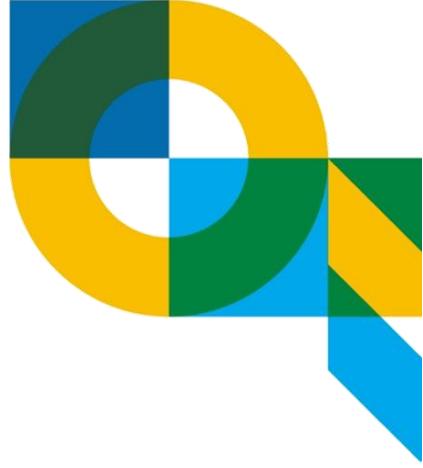
App_Navigation2



JS App.js > Counter

```
1 import React from 'react';
2 import { View, Text, Button, StyleSheet } from 'react-native';
3 import { createSlice, configureStore } from '@reduxjs/toolkit';
4 import { Provider, connect } from 'react-redux';
5 import { NavigationContainer } from '@react-navigation/native';
6 import { createStackNavigator } from '@react-navigation/stack';
7
8 // A very simple slice
9 const counterSlice = createSlice({
10   name: 'counter',
11   initialState: 0,
12   reducers: {
13     increment: (state) => state + 1,
14     decrement: (state) => state - 1,
15   },
16 });
17
18 const { actions, reducer } = counterSlice;
19
20 // A very simple store using Redux Toolkit's configureStore
21 const store = configureStore({
22   reducer: { count: reducer },
23 });

25 // A screen!
26 function Counter({ count, dispatch, navigation }) {
27   return (
28     <View style={styles.container}>
29       <Text style={styles.paragraph}>{count}</Text>
30       <Button
31         title="Increment"
32         onPress={() => dispatch(actions.increment())}
33       />
34       <Button
35         title="Decrement"
36         onPress={() => dispatch(actions.decrement())}
37       />
38       <Button
39         title="Go to static count screen"
40         onPress={() => navigation.navigate('StaticCounter')}
41       />
42     </View>
43   );
44 }
```



```
// Another screen!
function StaticCounter({ count }) {
  return (
    <View style={styles.container}>
      <Text style={styles.paragraph}>{count}</Text>
    </View>
  );
}

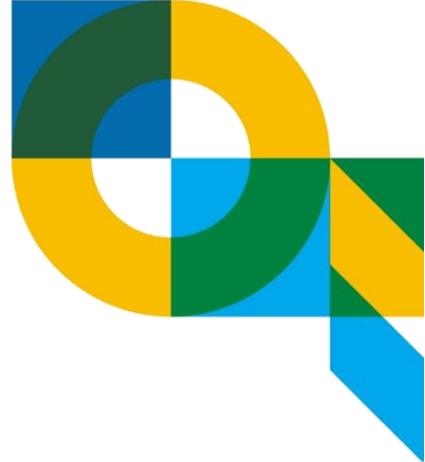
// Connect the screens to Redux
let CounterContainer = connect((state) => ({ count: state.count }))(Counter);
let StaticCounterContainer = connect((state) => ({ count: state.count }))(StaticCounter);

// Setup the stack navigator
const RootStack = createStackNavigator();

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  paragraph: {
    fontSize: 20,
    fontWeight: 'bold',
  },
});
```

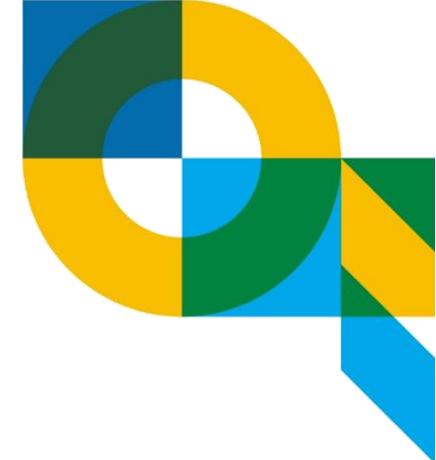
```
  // Render the app container component with the provider around it
  export default function App() {
    return (
      <Provider store={store}>
        <NavigationContainer>
          <RootStack.Navigator>
            <RootStack.Screen name="Counter" component={CounterContainer} />
            <RootStack.Screen
              name="StaticCounter"
              component={StaticCounterContainer}
            />
          </RootStack.Navigator>
        </NavigationContainer>
      </Provider>
    );
  }
}
```

練習一下



- 請嘗試在前面的範例中
 1. 新增一個 store
 2. 在多個 view 中顯示相同的 store

Reference



- <https://redux.js.org/>
- <https://react-redux.js.org/>
- <https://redux-toolkit.js.org/>
- <https://reactnavigation.org/docs/redux-integration>