06016323 Mobile Device Programming

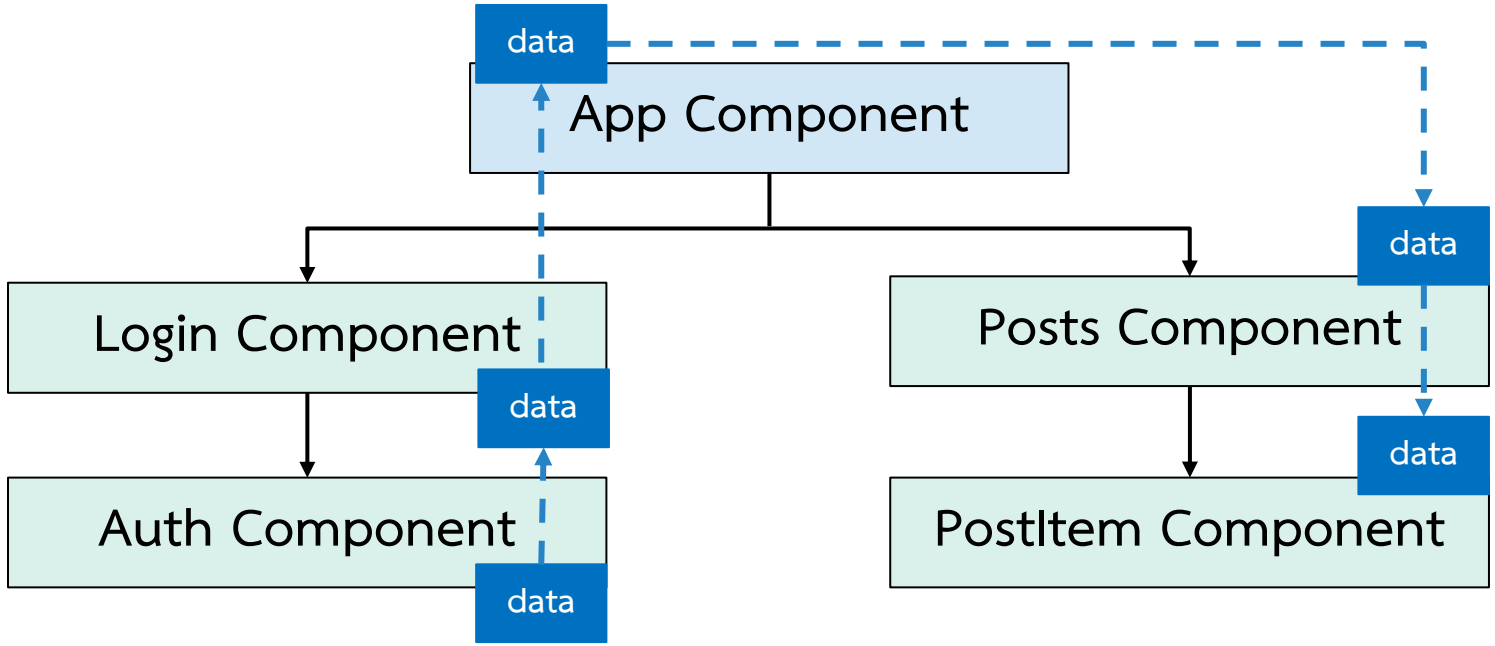
CHAPTER 11 : STATE MANAGEMENT

State Management

- เราสามารถจัดการสเตทต่างๆ ได้ภายในคอมโพเนนต์ต่างๆ และหากต้องการส่งค่าสเตทเหล่านั้นให้กับคอมโพเนนต์อื่น ก็สามารถส่งให้ได้ด้วยการใช้ props
- แต่ในกรณีที่แอปพลิเคชันมีความซับซ้อน และมีการอ้างอิงข้อมูลร่วมกันเป็นจำนวนมาก การส่งข้อมูลให้กันผ่าน props อาจจะไม่เหมาะสมนัก

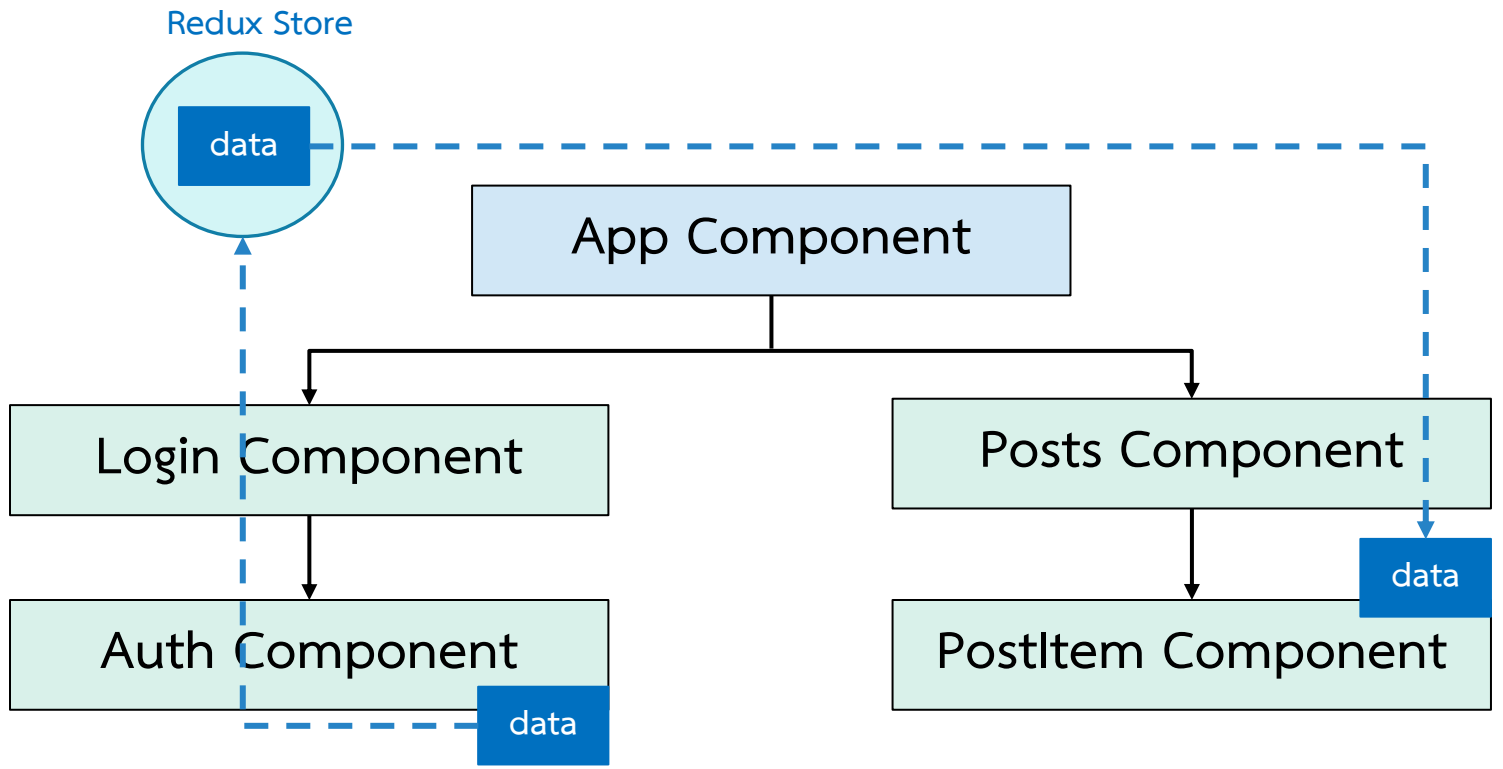
ตัวอย่างการอ้างอิงข้อมูลที่สัมพันธ์กันในหลายคอมโพเนนต์

- ตัวอย่างเช่น การล็อกอินเข้าใช้งานแอปพลิเคชัน โดยข้อมูลการล็อกอินนี้จะถูกแชร์ไปยังคอมโพเนนต์ที่เกี่ยวข้องต่างๆ ซึ่งข้อมูลการล็อกอินนี้ อาจมีผลต่อการแสดงผลของคอมโพเนนต์แต่ละส่วนได้



Redux

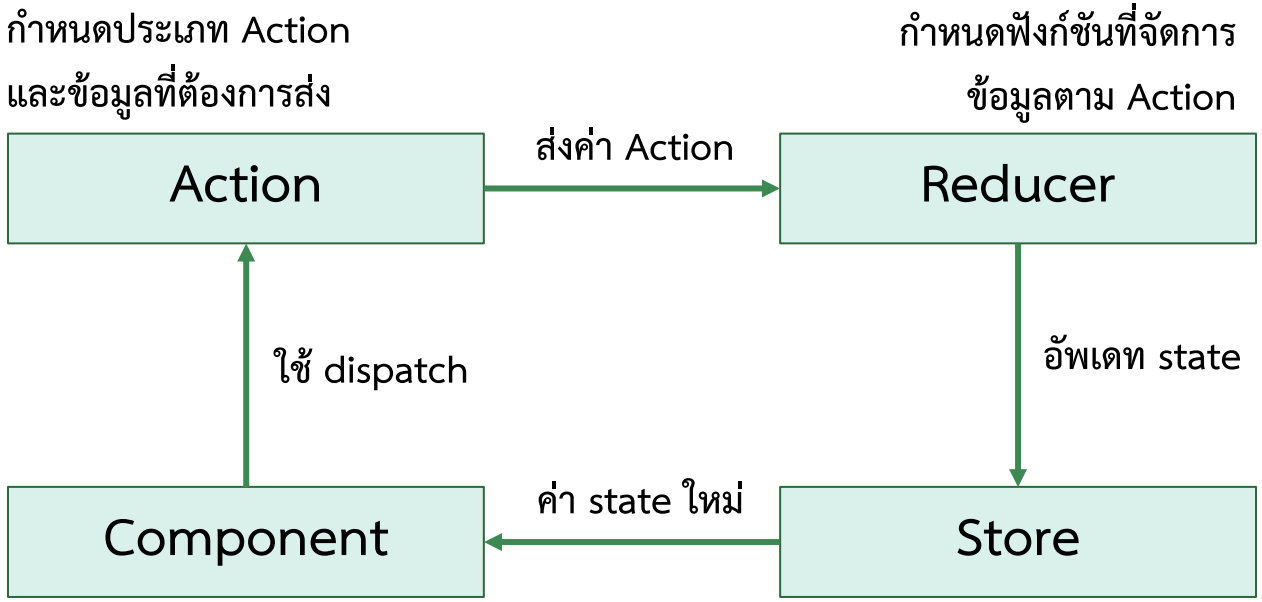
- เป็นไลบรารีที่ช่วยบริหารจัดการสเตทสำหรับแอปพลิเคชันที่มีความซับซ้อน
- ช่วยให้เราสามารถแชร์ข้อมูลและแชร์ฟังก์ชันระหว่างคอมโพเนนต์ได้ง่ายขึ้น



ติดตั้ง Redux

- ติดตั้ง redux และ react-redux
  - expo install redux react-redux หรือ
  - npm install --save redux react-redux

ส่วนประกอบพื้นฐานของ Redux



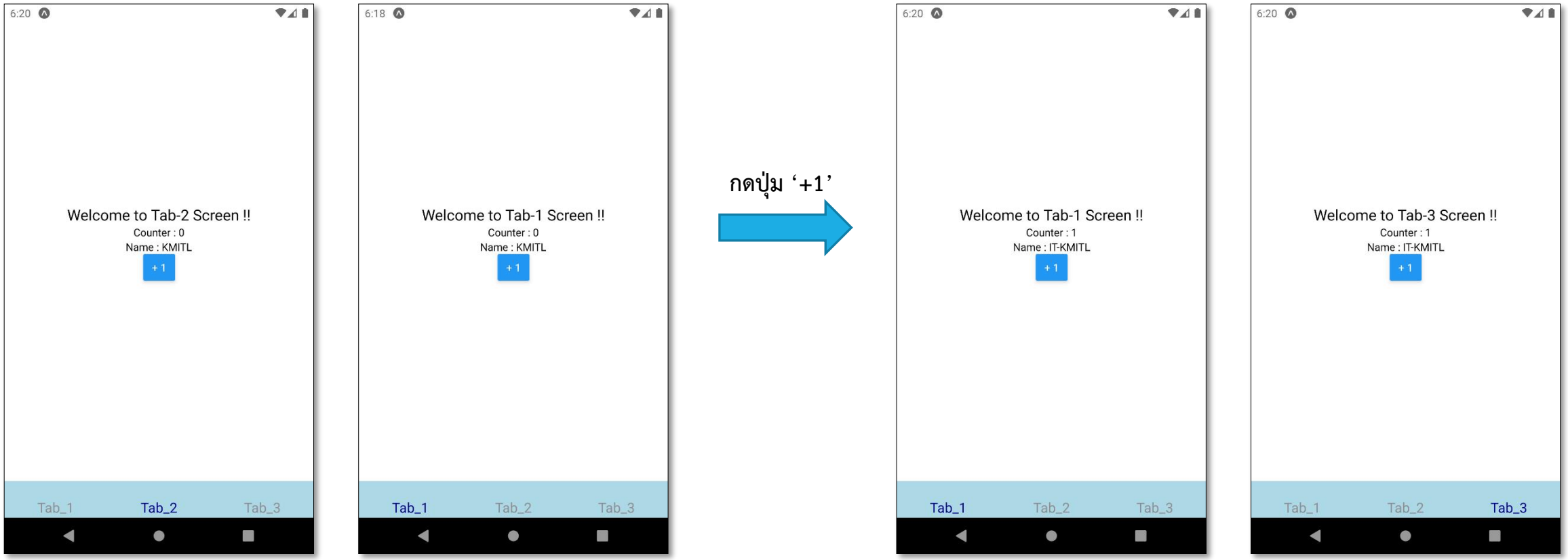
ส่วนประกอบพื้นฐานของ Redux

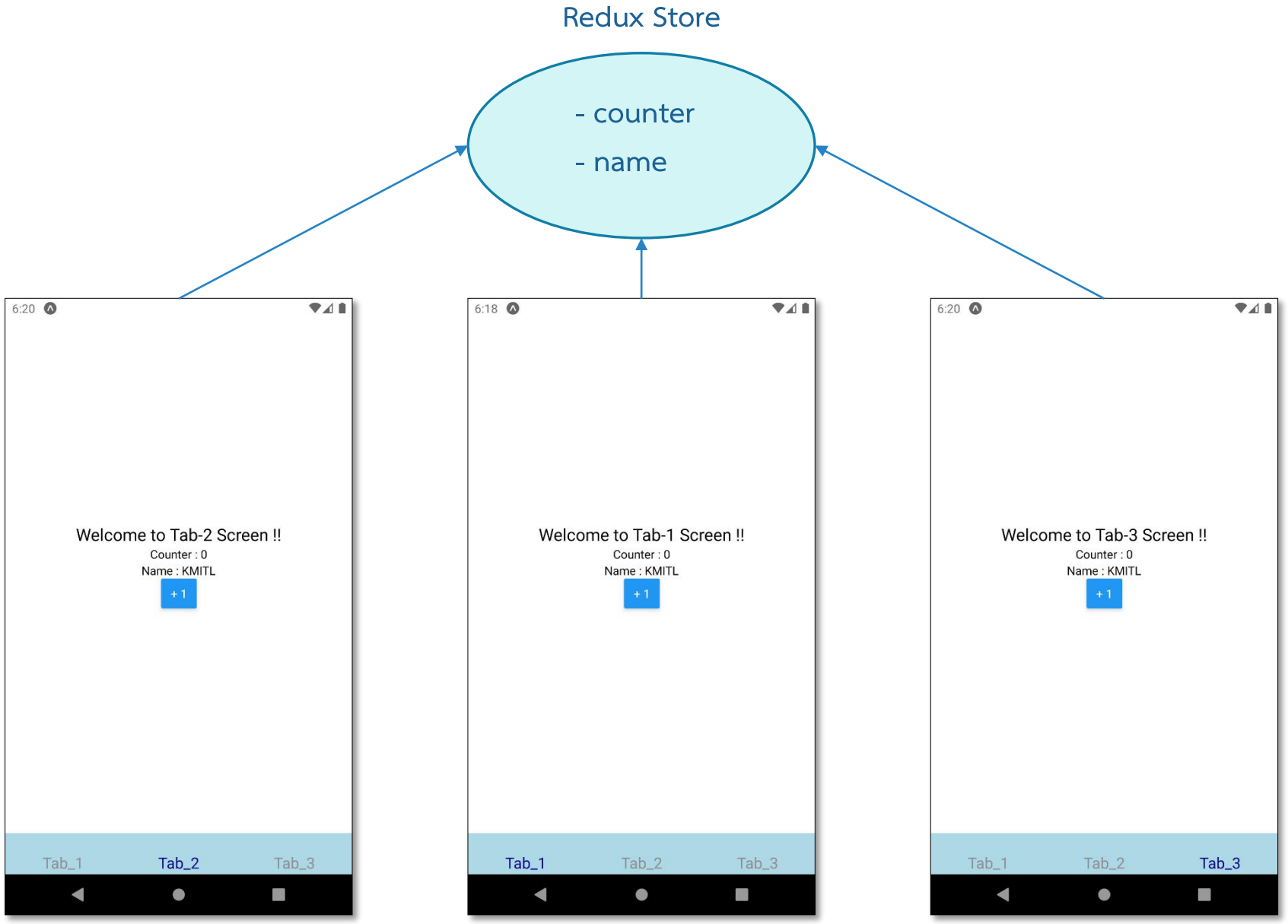
- Store
  - ส่วนเก็บสเตทที่ส่วนกลาง หากมีคอมโพเนนต์ต้องการจัดการสเตทนั้น ก็สามารถส่ง Action เพื่อบอกว่าต้องการจัดการอะไร
- Reducer
  - ส่วนฟังก์ชันที่ใช้จัดการข้อมูลใน store เช่น เพิ่มค่า แก้ไขค่า ของสเตท
  - ฟังก์ชันที่จัดการข้อมูลใน store จะมีอาร์กิวเมนต์ 2 ตัว คือ state และ action

ส่วนประกอบพื้นฐานของ Redux

- Action
  - เป็นอ๊อบเจ็กต์ที่ใช้บอก reducer ว่าต้องการจัดการข้อมูลสเตทใน store อย่างไร
  - ตัวอย่างเช่น {type: ‘UPDATE_DATA’, data: mydata} เป็นอ๊อบเจ็กต์ Action ประเภท UPDATE_DATA พร้อมกับส่งข้อมูล data ไปด้วย
    - เพื่อแจ้งกับ reducer ว่าต้องการอัพเดทข้อมูลใน store
- Component
  - เมื่อต้องการจัดการสเตทใน store คอมโพเนนต์จะส่ง Action ที่ต้องการ ผ่านการเรียก dispatch
  - คอมโพเนนต์สามารถอ้างถึงข้อมูลใน store ที่อัพเดทได้

ตัวอย่างโปรแกรม

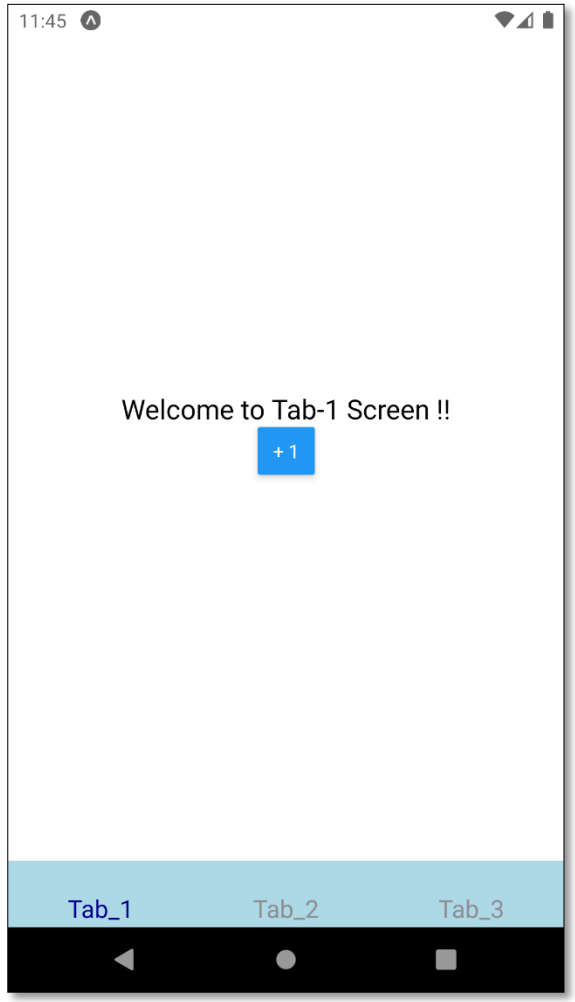



Redux Store
- counter
- name
6:20
Welcome to Tab-2 Screen !!
Counter : 0
Name : KMITL
+ 1
Tab_1
Tab_2
Tab_3
6:18
Welcome to Tab-1 Screen !!
Counter : 0
Name : KMITL
+ 1
Tab_1
Tab_2
Tab_3
6:20
Welcome to Tab-3 Screen !!
Counter : 0
Name : KMITL
+ 1
Tab_1
Tab_2
Tab_3

Tab1.js

```
import React from "react";
import { View, Text, Button } from "react-native";

const Tab1 = (props) => {
return (
    <View>
      <Text>Welcome to Tab-1 Screen !!</Text>
      <Button title=" + 1 " onPress={() => {}} />
    </View>
  );
};
```



MyNavigator.js

```
import { createBottomTabNavigator } from "react-navigation-tabs";
import { createAppContainer } from "react-navigation";

import ... (Tab1,Tab2, Tab3)

const MyTabNavigator = createBottomTabNavigator(
  {
    Tab_1: { screen: Tab1 },
    Tab_2: { screen: Tab2 },
    Tab_3: { screen: Tab3 },
  },
  {
    tabBarOptions: { ... },
  }
);

export default createAppContainer(MyTabNavigator);
```

การใช้งาน Redux – กำหนด Reducer

- กำหนด Reducer เพื่อใช้จัดการ state ที่อยู่ใน store
  - กำหนด state เริ่มต้นที่จะเก็บใน store (initialState) อาจเก็บเป็นอ๊อบเจ็กต์ที่ประกอบด้วยข้อมูลต่างๆ เช่น อ๊อบเจ็กต์ที่เก็บข้อมูล counter และ name เป็นต้น
  - สร้าง Reducer ซึ่งเป็นฟังก์ชันจัดการ state มีอาร์กิวเมนต์เป็น state และ action
    - สามารถกำหนดให้ state เริ่มต้นเป็น initialState
  - กำหนดการจัดการ state ตามประเภทของ action (action.type)
    - สามารถใช้ switch-case statement แยกเคสของ action.type
  - รีเทิร์น state ที่ถูกแก้ไขออกไป

การใช้งาน Redux – กำหนด Action

- กำหนด Action เพื่อใช้บอก Reducer ว่าจะต้องจัดการ state อย่างไร
  - อาจกำหนดฟังก์ชันเพื่อรีเทิร์นอ๊อบเจ็กต์ Action ได้
  - Action ประกอบด้วย ประเภทของ Action (type) และข้อมูลที่ต้องการส่งไปด้วย
  - ตัวอย่างเช่น
    - export const increaseCounter = (name) => {

      return { type: INC_COUNTER, myname: name };

      };
    - ฟังก์ชัน increaseCounter รับค่า name และทำการรีเทิร์น Action ประเภท INC_COUNTER และส่งพารามิเตอร์ myname ที่เก็บค่า name ที่รับมาด้วย

กำหนด Reducer และ Action (testReducer.js, testAction.js)

```
// testReducer.js
import { INC_COUNTER } from "../actions/testAction";
const initialState = {
  counter: 0,
  name: "KMITL",
};

const testReducer = (state = initialState, action) => {
  switch (action.type) {
    case INC_COUNTER :
      return  { counter: state.counter + 1, name: action.myname };
    default:
      return state;
  } };

export default testReducer;
```

```
// testAction.js
export const INC_COUNTER = "INC_COUNTER";

export const increaseCounter = (name) => {
   return { type: INC_COUNTER, myname: name };
};
```

การใช้งาน Redux – สร้าง Store และกำหนดให้ Component

- import สิ่งที่จำเป็นสำหรับสร้าง Store
  - import { createStore, combineReducers } from "redux";
  - import { Provider } from "react-redux";
- กรณีที่แอปพลิเคชันมีความซับซ้อน อาจมี Reducer หลายตัว ที่ใช้จัดการข้อมูลในส่วนต่างๆ ใน Store ได้
  - ในกรณีนี้ เราสามารถสร้าง Reducer หลัก (rootReducer) ที่รวมเอา Reducer หลายๆ ตัวเข้าด้วยกันด้วย combineReducers
  - const rootReducer = combineReducers({

    test: testReducer, });
  - สามารถอ้างอิงถึง testReducer ได้ด้วย id ชื่อ test
  - ตัวอย่างนี้ มี Reducer เพียงตัวเดียว

การใช้งาน Redux – สร้าง Store และกำหนดให้ Component

- สร้าง Store ด้วยการเรียก createStore()
  - const store = createStore(rootReducer);
- ทำการกำหนดให้ store สามารถใช้งานกับคอมโพเนนต์ต่างๆ ได้
  - กำหนดแท็ก <Provider></Provider> ครอบส่วนที่จะแสดงในคอมโพเนนต์ App รวมถึงกำหนด store ที่จะใช้งาน
  - export default function App() {

    return (

    <Provider store={store}> <MyNavigator /> </Provider>

    ); }

App.js

```
import { createStore, combineReducers } from "redux";
import { Provider } from "react-redux";
import MyNavigator from "./navigation/MyNavigator";
import testReducer from "./store/reducers/testReducer";

const rootReducer = combineReducers({
  test: testReducer,
});
const store = createStore(rootReducer);

export default function App() {
  return (
    <Provider store={store}> <MyNavigator /> </Provider>
  ); }
```

การใช้งาน Redux – Component เรียกใช้ข้อมูลจาก Store

- import สิ่งที่จำเป็นสำหรับการเรียกใช้ข้อมูลใน Store
  - import { useSelector, useDispatch } from "react-redux";
- กรณีที่ต้องการดึงข้อมูล state จาก store สามารถทำได้ด้วย useSelector
  - const counter = useSelector((state) => state.test.counter);
  - เป็นการดึงข้อมูลจาก reducer id ชื่อ test (ในที่นี้คือ testReducer) และอ้างอิงถึงข้อมูล counter
- เมื่อต้องการจัดการอื่นๆ กับข้อมูลใน store คอมโพเนนต์จะส่ง Action ที่ต้องการ ผ่านการเรียก dispatch
  - const dispatch = useDispatch();
  - dispatch(increaseCounter("IT-KMITL")); // increaseCounter(name) เป็นฟังก์ชันที่กำหนดใน testAction.js

Tab1.js

```
import { useSelector, useDispatch } from "react-redux";
import { increaseCounter } from "../store/actions/testAction";


const Tab1 = (props) => {
  const counter = useSelector( (state) => state.test.counter );
  const name = useSelector( (state) => state.test.name );


  const dispatch = useDispatch();


  const increaseCounterHandler = () => {
    dispatch( increaseCounter( "IT-KMITL" ) );
  };
// ... ต่อด้านขวา ...
```

myname ในอ๊อบเจ็คต์ Action

```
  return (
    <View>
      <Text>Welcome to Tab-1 Screen !!</Text>
      <Text> Counter : { counter } </Text>
      <Text> Name : { name } </Text>
      <Button title=" + 1 " onPress={ increaseCounterHandler } />
    </View>
  );
};
```

Hook

- Hook เป็นฟังก์ชันที่ช่วยให้เราสามารถจัดการ react feature ต่างๆ ได้จากฟังก์ชันคอมโพเนนต์
- ตัวอย่างของ Hook ที่เคยใช้คือ useState ซึ่งช่วยในการจัดการ state ในคอมโพเนนต์ได้
- รูปแบบการใช้ Hook
  - สามารถเรียก Hook ได้ที่ระดับบน (Top level) ของฟังก์ชันเท่านั้น
  - สามารถเรียกใช้ Hook ได้จากฟังก์ชันคอมโพเนนต์เท่านั้น

useEffect()

- เป็น Hook ที่ช่วยจัดการ side effects ที่เกิดจากฟังก์ชันคอมโพเนนต์
- ซึ่งจะถูกเรียกให้ทำงานหลังจากทำ render() แล้ว
- useEffect() อนุญาตให้เราสามารถลงทะเบียนฟังก์ชัน ซึ่งจะถูกประมวลผลหลังจากกระบวนการ render ได้ ตัวอย่างเช่น
  - useEffect(() => {

    props.navigation.setParams({ param: paramVal });

    }, [dependency]);
  - กำหนดให้ทำการกำหนดพารามิเตอร์ให้กับ navigation property ด้วยคำสั่ง setParams() ซึ่งจะทำหลังจากการ render
  - นอกจากนี้ ยังสามารถกำหนด dependency ในอาร์กิวเมนต์ที่ 2 เพื่อบอกว่า **ฟังก์ชันที่กำหนดใน useEffect() จะถูกประมวลผลเมื่อ dependency มีการเปลี่ยนแปลงเท่านั้น** ไม่จำเป็นต้องทำทุกครั้งหลัง render()

useCallback()

- เป็น Hook ที่มักจะใช้ร่วมกับ useEffect() เนื่องจากจะช่วยป้องกันการสร้างฟังก์ชันซ้ำๆ ได้
- กรณีที่มีการกำหนดฟังก์ชันซ้อนฟังก์ชัน
  - ฟังก์ชันที่อยู่ภายในจะถูกสร้างใหม่ทุกครั้งที่ฟังก์ชันภายนอกทำงาน
  - นั่นคือ กรณีที่เราเขียนฟังก์ชันคอมโพเนนต์ จะทำให้ฟังก์ชันที่ถูกกำหนดภายในคอมโพเนนต์นั้นจะถูกสร้างใหม่ทุกครั้งที่คอมโพเนนต์ถูกสร้างขึ้น
  - ซึ่งอาจทำให้เกิดปัญหาได้ ในกรณีที่ฟังก์ชันภายในนั้น ถูกกำหนดเป็น dependencies ของ useEffect()

useCallback()

```
const MyComponent = props => {
  const innerFunction = () => {
    // โค้ดโปรแกรม
  };

  useEffect(() => {
    innerFunction();
    // กำหนด innerFunction เป็น dependency เพื่อให้อัพเดทตลอดเวลา
       เมื่อ innerFunction มีการเปลี่ยนแปลง เช่น ข้อมูลที่ใช้เปลี่ยน
  }, [innerFunction]);
};
```

- มีปัญหาในกรณีที่ ถ้า innerFunction ทำบางอย่างที่ทำให้ MyComponent ต้องถูกสร้างใหม่ อาจทำให้เกิด infinite loop ได้
- ป้องกันโดย ใช้ useCallback() ครอบการกำหนดฟังก์ชันนั้น และให้กำหนด dependency ของฟังก์ชันนั้นด้วย เพื่อให้มั่นใจว่า ฟังก์ชันนั้นจะถูกสร้างใหม่ เมื่อ dependency เปลี่ยนเท่านั้น
- ทำให้ฟังก์ชันจะไม่ถูกสร้างใหม่ในทุกครั้งที่ render() อีก

useCallback() เพื่อป้องกัน infinite loop

```
const MyComponent = props => {
   const innerFunction = useCallback ( () => {
       // โค้ดโปรแกรม
   }, [dependency] )

   useEffect(() => {
       innerFunction();
       // กำหนด innerFunction เป็น dependency เพื่อให้อัพเดทตลอดเวลา
          เมื่อ innerFunction มีการเปลี่ยนแปลง เช่น ข้อมูลที่ใช้เปลี่ยน
   }, [innerFunction]);
};
```

JavaScript: Array Methods

- splice() : เมธอดที่สามารถเพิ่มหรือลบอิลิเมนต์ออกจากอะเรย์ได้
- concat() : เมธอดที่ทำการสร้างอะเรย์ใหม่ โดยเกิดจากการรวมอะเรย์ที่มีอยู่กับอะเรย์หรือข้อมูลอื่นๆ
- map() : เมธอดที่สร้างอะเรย์ขึ้นใหม่ ด้วยการทำฟังก์ชันกับอิลิเมนต์แต่ละตัวในอะเรย์
- filter() : เมธอดที่สร้างอะเรย์ใหม่ให้กับอิลิเมนต์ของอะเรย์ที่ผ่านการตรวจสอบตามเงื่อนไข
- every() : เมธอดที่ตรวจสอบว่าอิลิเมนต์ในอะเรย์ผ่านการตรวจสอบตามเงื่อนไขทั้งหมดหรือไม่
- some() : เมธอดที่ตรวจสอบว่ามีอิลิเมนต์บางตัวในอะเรย์ที่ผ่านการตรวจสอบเงื่อนไขหรือไม่
- indexOf() : เมธอดที่ค้นหาอิลิเมนต์ที่ต้องการในอะเรย์ และคืนค่าตำแหน่งของอิลิเมนต์นั้นออกมา
- find() : เมธอดที่คืนค่าอิลิเมนต์ตัวแรกในอะเรย์ที่ผ่านการตรวจสอบเงื่อนไขที่กำหนด
- findIndex() : เมธอดที่คืนค่าตำแหน่งของอิลิเมนต์ตัวแรกในอะเรย์ที่ผ่านการตรวจสอบเงื่อนไขที่กำหนด
- length : property เก็บขนาดของอะเรย์