[2022-11-29] 2반 실습 파일

학습 목표

- React의 상태 관리 패턴인 Flux 패턴에 대해 배우고, useContext를 통해 전역 상태관리를 구현합니다.
- useReducer과 같은 추가 Hook도 학습합니다.
- 전역 상태 관리의 필요성을 인지하고 React에서의 전역 상태 관리법에 대해 학습합니다.
- useContext, useReducer 등의 훅을 학습하고 활용해 봅니다.

▼ 이전 질문! useFetch 훅에서 useEffect 대신 useCallback 사용 가능한가요?

https://codesandbox.io/s/usefetch-custom-hook-x6bdwl?file=/src/App.js:0-1870

아주아주 간단하게만 언급하고 넘어가겠습니다!

useCallback은 자주 쓰는 함수가 재생성되는게 싫어서 만들어진 함수 자체를 메모이징 해놓고 쓸 때 쓰는 훅인데,

질문의 배경을 보면 useFetch처럼 별도의 훅으로 만들어 놓은 것처럼 useCallback함수로 그 기능들을 메모리에 저장해두고 쓰고 싶을 때마다 꺼내쓰자는 의미로 useEffect대신 useCallback함수를 써도 되냐?라고 여쭤보신 것 같은데,

용도가 달라요! 우선 useEffect의 용도는 fetch함수를 딱 한번만 실행하기 위함 | useCallback의 용도는 함수를 메모리에 저장해놓고 재생성하지 않고 필요할 때 쓰기 위함입니다.

그래서 만약 useCallback을 사용한다고 해도 useEffect는 여전히 필요하게 됩니다!

그~래~도~ 만약에! 여전히! useCallback을 사용하고 싶다고 하면 아래와 같은 방식으로 사용할 수는 있습니만…

사실 이런 상황 useCallback이 잘 쓰이는 상황이 있기는 한데, 그건 이따가 알아보도록 하고!!

우선 쓸 수 있으나 이 경우에는 잘 쓰지 않을 것 같다라고만 생각하고 넘어갑시다!

```
import React, { useState, useEffect, useCallback } from "react";

const useFetch = (url) => {
```

```
  const [result, setResult] = useState(null); // 훅 내에서의 상태값 변화 또한 훅이 사용된 컴포넌트
 (호스트 컴포넌트)에 대해 리렌더링을 발생합니다
  useEffect(() => {
    fetch(url)
      .then((res) => res.json())
      .then((res) => setResult(res.data))
      .catch((err) => console.log(err));
  }, [url]);
  return result;
};

// useCallback을 사용한 useFetch?!
const useFetch_cb = (url) => {
  const [result, setResult] = useState(null);
  const createFetchFn = useCallback(() => {
    // useCallback: 함수가 재생성되는게 싫어 한번 생성해두고 메모리에 저장해놓고 씀
    fetch(url)
      .then((res) => res.json())
      .then((res) => setResult(res.data))
      .catch((err) => console.log(err));
  }, [url]);
  useEffect(() => {
    createFetchFn();
  }, [createFetchFn]);
  return result;
};

function App() {
  const url = "https://reqres.in/api/users/2";

  const result = useFetch(url);

  return (
    <div>
      <h4>React Axios로 HTTP GET 요청하기</h4>
      <div>
        Name: {result?.first_name + " " + result?.last_name}
        Email: {result?.email}
      </div>
    </div>
  );
}

export default App;
```

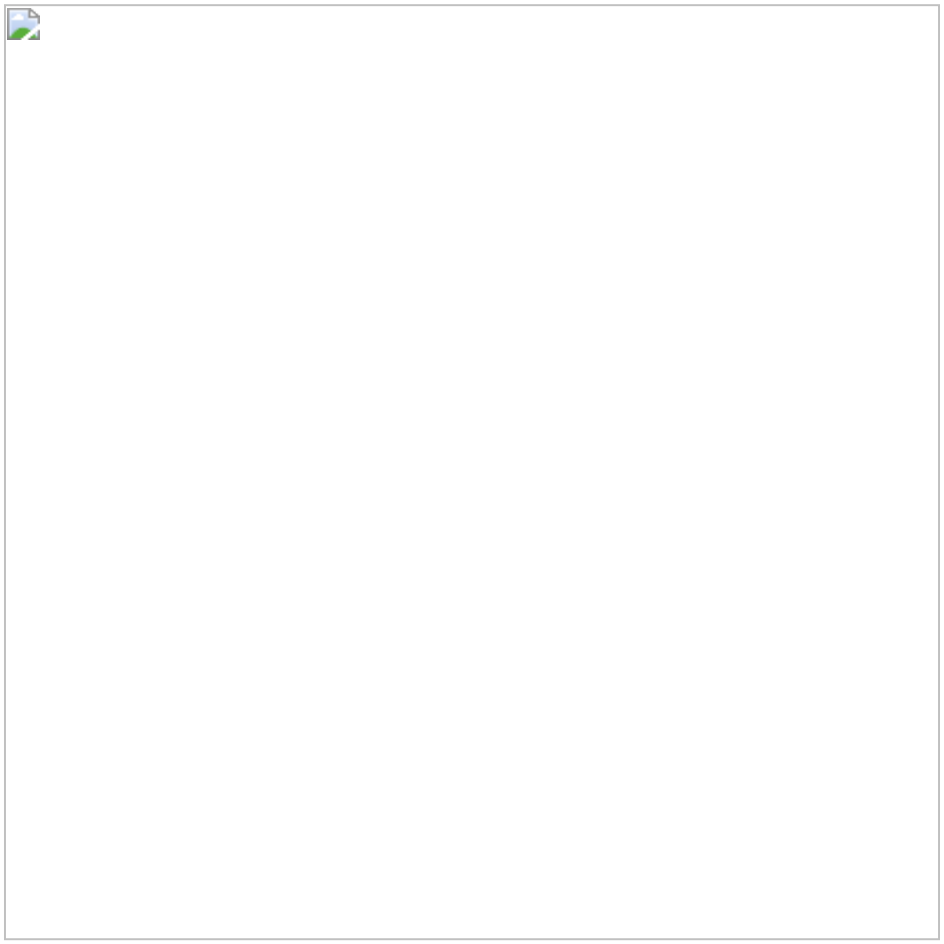
문제 풀이

1. React useReducer로 복잡한 상태 관리하기

사전 지식

Flux 패턴

기존 MVC 패턴과 Flux 패턴을 큼직한 차이와 등장 배경을 보고 알아봅시다. (나중에 인터뷰에도 나올 수 있기 때문에 한번 다뤄보시죠).



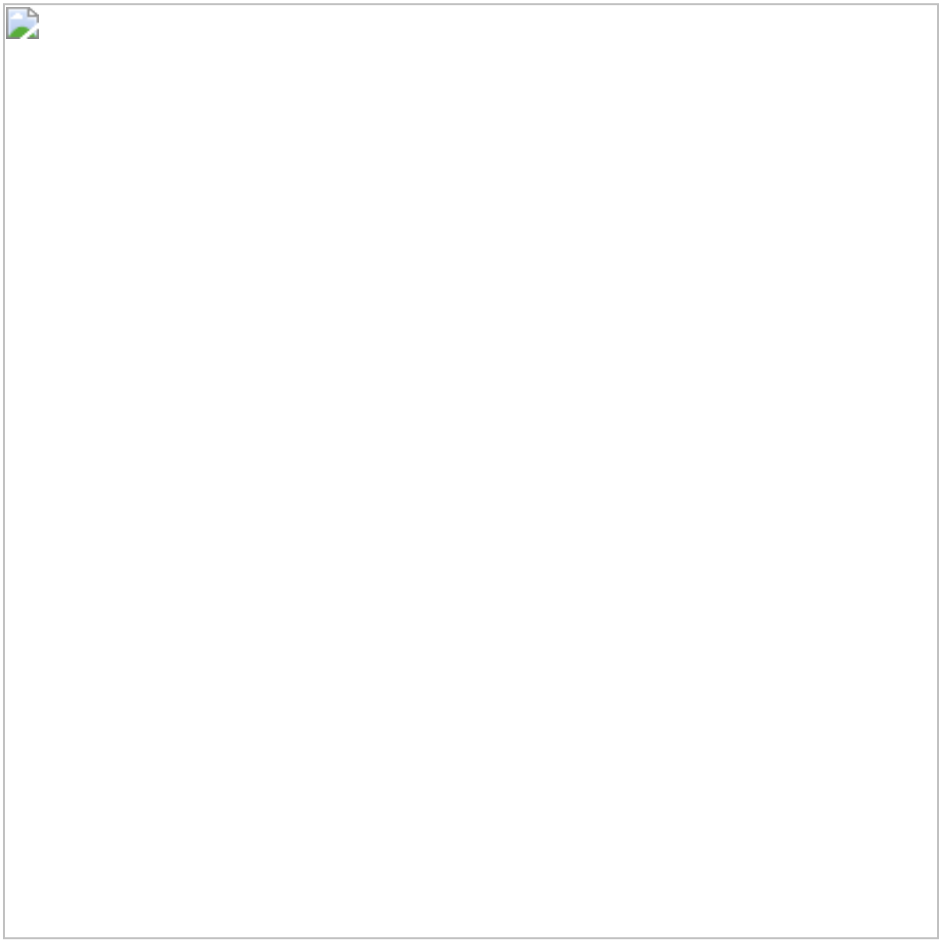
MVC 패턴입니다. (model: 데이터 저장, controller: 유저요청 + 데이터 CRUD 관리, View: 사용자에게 보여주는 부분)

1. model에서 데이터 변형 → view에서 보여짐
2. view를 통해 데이터 입력 → 역시나 model 업데이트 가능 (양방향 데이터 흐름)

규모가 커지고 모델/뷰가 많아지고… 어떤 뷰는 어떤 모델들을 보여주고, 어떤 모델을 여러 뷰에서 연쇄적으로 업데이트가 보여지고… 등. 이 양방향 데이터 흐름이 플로우를 복~잡하고 추~적하기 어렵게 만들어.

그래서 등장한 패턴 두둥 Flux 패턴입니다. 양방향의 데이터 흐름을 한 방향으로 해결해주면 될 것이라 생각한 것이죠.

이렇게 데이터를 업데이트? 해줄 때 시그널(action)을 보냅니다. 그 시그널을 받은 대행자(dispatch)가 대신 store라는 외부 저장소에 가서 업데이트해주고, 그 업데이트를 view에 보여줍니다. 오늘 배울 useReducer, context api를 사용하면 이게 방식이 구현 가능해진답니다!



개인적으로 엄청나게 다른 패턴이라기보단 약간의 업그레이드 버전?으로 보이는데요

좀 더 직관적이고 자세한 명령을 대신 수행해주는 대행자를 둠으로써 controller의 복잡도 줄였다? 같아요. 하지만 관점에 따라 다를 수 있으니 ^-^

그래서 MVC를 쓰면 데이터가 양방향으로 흘러 커지면 복잡해지겠구나~ 그래서 데이터를 단일 방향으로 흐르게 하는 Flux 패턴이 생겼구나~ 까지만 이해하고 넘어가면 될 것 같아요~

useReducer?

언제 쓰냐요? 관리해야할 state값이 많을 때! 그 state들을 업데이트하는 함수들이 많을 때! 한 곳에 모아서 쉽게! 관리하고 싶을 때 사용합니다.

좋은 코딩 스타일? 고민해보면 복잡한 건 숨기고 사용은 편하게? 이런 관습? 같은게 있는데

일반 웹사이트도 유저 인터페이스에 조작은 쉽데 그 화면 뒤에는 엄청나게 복잡한 로직이 숨겨진 것처럼…

이론 강의 때 언급한 그 응잡화/중앙화에 맞는 스타일입니다

▼ useReducer 구조

1. 초기 state들을 담은 오브젝트 생성
2. reducer 함수 만들기

기존 상태값을 업데이트하던 함수를 switch 안에 정의한다고 생각하면 될 것 같습니다.

```
// reducer 생성

const initialState = {
  ...
}


const reducer = (state, action) => {
  switch (action.type) {
    case "<<액션1>>": {
      return {
        ... // non-mutate
      };
    }
    ... // 뭐시기 여러 case
    default:
      throw new Error(); // 아무런 case에 속하지 않는 경우
  }
};
```

+ 컴포넌트 내 함수의 재생성을 막으려면 useCallback을 사용해야하는데 여기서는 dispatch를 통해 만들어진 함수들이 전달 가능함 (나름의 장점)

```
// reducer 사용

import { useReducer } from "react";

const App = () => {
  const [상태값, dispatch] = useReducer(reducer, initialState)
// useReducer는 모든 상태값 + 그걸 업데이트 시킬 때 쓰일 dispatch 함수 반환합니다
// (아직 초기값을 특정 로직 처리 후 설정하고 싶을 때  세번째 파라미터로 init함수 사용)
  ...
  return (
    <div>
      {<<상태값 꺼내기>>}
      <button onClick={()=>{dispatch({ type: "<<action 타입>>", payload: "<<참고하고 싶은 값
>>" });}}></button> // dispatch 내부에는 객체 형태로 전달 (그 객체가 reducer에서 action으로 읽어옵
니다)
    </div>
)
}
```

실제 예시를 봐봅시다!

▼ useReducer 사용 전

```
// useReducer 사용 전
import "./styles.css";
import { useState } from "react";

export default function App() {
  const [count, setCount] = useState(0);
  const [message, setMessage] = useState("인삿말");
  const [idealType, setIdealType] = useState("송강");

  const incrementCount = () => {
    setCount((count) => count + 1);
  };

  const decrementCount = () => {
    setCount((count) => count - 1);
  };

  const changeToEng = () => {
    setMessage("하이");
  };
  const changeToKR = () => {
    setMessage("안뇽");
  };
  const changeToThai = () => {
```

```
    setMessage("사와디카");
  };
  const changeIdealType = (e) => {
    setIdealType(e.target.value);
  };

  return (
    <div className="App">
      <div>숫자: {count}</div>
      <button onClick={incrementCount}>+</button>
      <button onClick={decrementCount}>-</button>
      <br></br>
      <div>인삿말: {message}</div>
      <button onClick={changeToEng}>English</button>
      <button onClick={changeToKR}>한국어</button>
      <button onClick={changeToThai}>Thai</button>
      <br></br>
      <div>이상형: {idealType}</div>
      <input value={idealType} onChange={changeIdealType} />
    </div>
  );
}
```

▼ useReducer 사용 후 https://codesandbox.io/s/usereducer-practice-3x7n9k?file=/src/App.js

```
// useReducer 사용 후

// reducer.js

export const initialState = {
  count: 0,
  message: "인삿말",
  idealType: "송강"
}; // 1. 초기 state를 담은 오브젝트 생성

export const Actions = { // 하드코딩, 타이포 예방
  incrementCount: "incrementCount",
  decrementCount: "decrementCount",
  changeToEng: "changeToEng",
  changeToKR: "changeToKR",
  changeToThai: "changeToThai",
  changeIdealType: "changeIdealType"
};

// 2. 함수 만들기
// (대게 별개 reducer.js 파일 별개로 만듦)
export const reducer = (state, action) => {
  switch (action.type) {
    case Actions.incrementCount: {
      return {
        ...state,
```

```
        count: state.count + 1
      };
    }
    case Actions.decrementCount: {
      return {
        ...state,
        count: state.count - 1
      };
    }
    case Actions.changeToEng: {
      return {
        ...state,
        message: "하이"
      };
    }
    case Actions.changeToKR: {
      return {
        ...state,
        message: "안뇽"
      };
    }
    case Actions.changeToThai: {
      return {
        ...state,
        message: "사와디카"
      };
    }
    case Actions.changeIdealType: {
      return {
        ...state,
        idealType: action.payload
      };
    }
    default:
      throw new Error();
  }
};


// App.js

import "./styles.css";
import { useReducer } from "react";
import { Actions, initialState, reducer } from "./reducer";

export default function App() {
  const [state, dispatch] = useReducer(reducer, initialState);
  const { count, message, idealType } = state;

  return (
    <div className="App">
      <div>Count: {count}</div>
      <button
        onClick={() => {
          dispatch({ type: Actions.incrementCount });
```

```
        }}
      >
        +
      </button>
      <button
        onClick={() => {
          dispatch({ type: Actions.decrementCount });
        }}
      >
        -
      </button>
      <br></br>
      <div>인삿말: {message}</div>
      <button
        onClick={() => {
          dispatch({ type: Actions.changeToEng });
        }}
      >
        English
      </button>
      <button
        onClick={() => {
          dispatch({ type: Actions.changeToThai });
        }}
      >
        Thai
      </button>
      <button
        onClick={() => {
          dispatch({ type: Actions.changeToKR });
        }}
      >
        한국어
      </button>
      <br></br>
      <div>이상형: {idealType}</div>
      <input
        onChange={(e) => {
          dispatch({ type: Actions.changeIdealType, payload: e.target.value });
        }}
      />
    </div>
  );
}
```

이렇게 useReducer를 활용하면 많은 상태값과 복잡한 업데이트함수를 쉽게 관리할 수 있다~

useContext?

언제 쓰나요? props를 전달하려고 계단을 너무 많이 내려가야할 때~ (=== prop drilling이 싫을 때)

▼ useContext 구조

```
// useContext 뼈대

import { createContext, useContext } from "react";

const MyContext = createContext();

export default function App() {
  return (
    <MyContext.Provider value={<<넘기고싶은값넘겨주기>>}>
      <div className="App">
        <Welcome/>
      </div>;
    </MyContext.Provider>
  );
}


const Welcome = () => {
  const {<<넘겨받은값>>} = useContext(MyContext);
  return <div>{<<넘겨받은값>>}</div>
}
```

실제 예시를 봐봅시다!

▼ useContext 사용 전

```
// App.js

import "./styles.css";
import {
  createContext,
  useContext,
  useEffect,
  useState,
  memo,
  useCallback
} from "react";

export default function App() {
  const [message, setMessage] = useState("배달중");

  console.log("꼭대기 rerendered");
  return (
    <div className="App">
      꼭대기층
      <Level3 message={message} setMessage={setMessage} />
    </div>
  );
}

const Level3 = ({ message, setMessage }) => {
  console.log("3층 rerendered");
  return (
    <div>
      3층
      <Level2 message={message} setMessage={setMessage} />
    </div>
  );
};

const Level2 = ({ message, setMessage }) => {
  console.log("2층 rerendered");
  return (
    <div>
      2층
      <Level1 message={message} setMessage={setMessage} />
    </div>
  );
};

const Level1 = ({ message, setMessage }) => {
  console.log("1층 rerendered");
  useEffect(() => {
    setMessage("배달 완료");
  }, []);

  return (
```

```
    <div>
      1층 도착!
      <br></br>
      {message}
    </div>
  );
};
```

▼ useContext 사용 후 https://codesandbox.io/s/usecontext-practice-quphxt?file=/src/App.js:0-858

자 그런데 useContext를 사용할 때 조금 조심할 필요가 있습니다. 탑-레벨 컴포넌트의 state가 바뀌면 하위 컴포넌트들이 전부 다 제렌더링 되기 때문에 “반드시 필요한 렌더링인건가?”를 잘 생각하고 사용해야합니다.

대표적인 최적화 방법! 여기선 React.memo 와 useCallback만 살펴보겠습니다. (아직 이해못하셔도 괜찮습니다. 이번주는 useReducer + useContext만 배우는게 목표! 다음주에 최적화 방법론 다시 배울 것이라 오늘은 맛보기~)

```
// App.js

import "./styles.css";
import {
  createContext,
  useContext,
  useEffect,
  useState,
  memo,
  useCallback
} from "react";

const Context = createContext();

export default function App() {
  // 업데이트 시키고 싶다면 state값으로 만들어줘야해
  const [message, setMessage] = useState("배달중");
  // 사용하고 싶은 탑-레벨 컴포넌트에서 Context.Provider로 감싸줘
  console.log("꼭대기 rerendered");

  const sayHello = useCallback(() => {
    console.log("안뇽하시요");
  }, []); // 왜냐? 자바스크립트는 함수, 오브젝트, 리스트 선언된 것들을 저장할 때
  // 주소값을 저장하는데,
  // 컴포넌트가 재렌더링될 때 내부 함수도 다시 선언되면서 새로운 주소값을 가져
  // 전달된 prop도 업데이트 되었다고 판단하여 memo가 되어있어도 리렌더링됨

  return (
    <div className="App">
```

```
      <Context.Provider value={{ message, setMessage }}>
        꼭대기층
        <Level3 sayHello={sayHello} />
      </Context.Provider>
    </div>
  );
}
const Level3 = memo(() => {
// 만약에 업데이트가 일어난 컴포넌트를 제외하고는 업데이트가 반영될 필요가 없다면 memo 사용할 수 ㅇ
// 컴포넌트를 메모리징. 이 내용을 그대로 재사용하여 렌더링시 dom에서 달라진 부분을 확인하지 않아 성능상 이점이
있을 수 있음
// memo는 고차컴포넌트?라고도 부르는데 (Hoc) 그냥 컴포넌트를 받아서 어떠어떤 처리를 해서 컴포넌트를 반환
// export 할 때 export memo(Level3) 이런식으로
// 최적화...고민!! 메모리가 차면 느려지고.. 리렌더링이 많이 발생하면 서버,CPU 무리
// trade-off..
  console.log("3층 rerendered");
  return (
    <div>
      3층
      <Level2 />
    </div>
  );
});
const Level2 = memo(() => {
  console.log("2층 rerendered");
  return (
    <div>
      2층
      <Level1 />
    </div>
  );
});
const Level1 = () => {
  const { message, setMessage } = useContext(Context);
  console.log("1층 rerendered");
  useEffect(() => {
    setMessage("배달 완료");
  }, []); // 여기서 state가 변경되며 App 내 모든 컴포넌트가 리렌더링이 일어나는 걸 볼 수 있음
  // 만약에 업데이트가 일어난 부분을 제외하고는 업데이트가 반영될 필요가 없다면

  return (
    <div>
      1층 도착!
      <br></br>
      {message}
    </div>
  );
};
```

요!약!

useContext로 어떠한 값을 ***컴포넌트 레벨에 상관없이*** 사용할 수 있겠다 ~

useReducer로 ***많은 상태값 + 그 업데이트 함수***들을 관리할 수 있겠다~

두개 합쳐서 쓴다면??!!

⇒ 컴포넌트 레벨에 상관없이~ 여러 상태값+함수들을 관리할 수 있겠구나~ 라는 생각까지 오면 아주 굿굿

▼ 이 둘 useReducer + useContext를 합친 구조는…?

요로코롬 될 수 있겠죠?

```
const initialState = {
  ...
}

const reducer = (state, action) => {
  switch (action.type) {
    case "<<액션1>>": {
    }
    ...
  }
};

const MyContext = createContext();

export default function App() {
  const {상태값, dispatch} = useReducer(reducer, initialState)
  return (
    <MyContext.Provider value={{상태값, dispatch}}>
      <div className="App">
        <Welcome/>
      </div>;
    </MyContext.Provider>
  );
}

const Welcome = () => {
  const {상태값, dispatch} = useContext(MyContext);
  return (
    <div>
      <div>{<<상태값 꺼내기>>}</div>
      <button onClick={()=>{dispatch({ type: "<<action 타입>>", payload: "<<참고하고 싶은 값
>>" });}}></button>
    </div>
)
}
```

▼ 문제 정답

```
// App.js

import React, { createContext, useReducer } from 'react';
import ContainerA from './components/A';
import './App.css';

export const MyContext = createContext(null);

/*
  다음 초기값을 완성해주세요.
*/
const initial = {
  top: 'apple',
  middle: 'banana',
  bottom: 'coconut',
};

/*
  상태 변경 설명서라 할 수 있는 reducer를 지시사항에 맞게 작성해주세요.
*/
function reducer(state, action) {
  switch (action.type) {
    case 'top-middle-change': //기존값을 참조해 switch 해주면 되겠구나
      return { ...state, top: state.middle, middle: state.top };
    case 'middle-bottom-change':
      return { ...state, middle: state.bottom, bottom: state.middle };
    case 'top-bottom-change':
      return { ...state, top: state.bottom, bottom: state.top };
    case 'top-update': // 입력한 값을 payload로 받아오면 되겠구나
      return { ...state, top: action.payload };
    case 'middle-update':
      return { ...state, middle: action.payload };
    case 'bottom-update':
      return { ...state, bottom: action.payload };
    default:
      throw new Error();
  }
}

export default function App() {
  // useReducer를 사용해 state와 dispatch를 value 넣어서 전달해주세요.
  const [state, dispatch] = useReducer(reducer, initial);
  const value = { state, dispatch };

  return (
    <div className="layout">
```

```
      <MyContext.Provider value={value}>
        <ContainerA />
      </MyContext.Provider>
    </div>
  );
}
```

```
// B.jsx

import ContainerC from './C';
import ContainerD from './D';
import ContainerE from './E';
import { useContext, useRef } from 'react';
import { MyContext } from '../App';

export default function ContainerB() {
  // Context로 부터 dispatch를 받습니다.
  const { state, dispatch } = useContext(MyContext);

  const inputRef = useRef();
  //   console.log('inputRef', inputRef.current?.value); // dom selector 역할 console로 확
인해보기 (다른 역할에도 쓰이는데 여기서는 dom selector 역할을 하는구나~ 생각하고 넘어가기)
  return (
    <div>
      <div className="container b">
        <div className="title">Component B</div>
        <div className="content">
          <ContainerC />
          <ContainerD />
          <ContainerE />
          <div className="button-group">
            <button
              onClick={() => {
                // 올바른 액션 타입을 지정해주세요.
                dispatch({ type: 'top-middle-change' });
              }}
            >
              Top Middle Change
            </button>
            <button
              onClick={() => {
                // 올바른 액션 타입을 지정해주세요.
                dispatch({ type: 'middle-bottom-change' });
              }}
            >
              Middle Bottom Change
            </button>
            <button
              onClick={() => {
                // 올바른 액션 타입을 지정해주세요.
                dispatch({ type: 'top-bottom-change' });
              }}
```

```
            >
              Top Bottom Change
            </button>
          </div>
          <div className="button-group">
            <input type="text" data-testid="input" ref={inputRef} />
            <button
              onClick={() => {
                // 올바른 액션 타입과 페이로드를 전달해주세요.
                dispatch({
                  type: 'top-update',
                  payload: inputRef.current.value,
                });
              }}
            >
              Top Update
            </button>
            <button
              onClick={() => {
                // 올바른 액션 타입과 페이로드를 전달해주세요.
                dispatch({
                  type: 'middle-update',
                  payload: inputRef.current.value,
                });
              }}
            >
              Middle Update
            </button>
            <button
              onClick={() => {
                // 올바른 액션 타입과 페이로드를 전달해주세요.
                dispatch({
                  type: 'bottom-update',
                  payload: inputRef.current.value,
                });
              }}
            >
              Bottom Update
            </button>
          </div>
        </div>
      </div>
    </div>
  );
}
```

2. 체크아웃 폼 만들기

▼ 문제 정답

```
// App.js

import React, { useReducer, useState } from 'react';
import styled from 'styled-components';
import { placeOrder } from './api'; // 얘...지금은 디비에 추가한다던가 그런 코드가 없지만...

const initialState = {
  success: '', // API 요청 성공시 성공 메시지를 저장합니다.
  error: '', // API 요청 실패시 에러 메시지를 저장합니다.
  loading: false, // API 로딩 상태를 저장합니다.
};

// request -  success, error, loading 상태를 초기화합니다.
// success - success message를 저장합니다.
// error - error message를 저장합니다.
const reducer = (state, action) => {
  switch (action.type) {
    case 'request': { // 요청할 때는 success, error 메시지가 비어있고, loading=true면 좋겠다는 생각
      return {
        ...state,
        success: '',
        error: '',
        loading: true,
      };
    }
    case 'success': { // 상태값 success에 넣어줄 message를 payload로 받아와야겠다~
      return {
        ...state,
        success: action.payload.message,
        loading: false,
      };
    }
    case 'error': {
      return {
        ...state,
        error: action.payload.message,
        loading: false,
      };
    }
  }
};

// 이 함수를 활용하여 폼의 상태를 체크하세요.
const validateForm = (email, address) => {
  // 이메일 포맷 체크... 정규표현식 쓸 수 있는데 시간이 없어 나중에 jest 배울 때 하는 걸로
  // 폼의 데이터를 체크해주세요.
  // 성공시 빈 문자열을 리턴하고, 에러가 있을 시 에러 메시지를 리턴합니다.
  if (!email.length) return 'email을 입력해주세요.';
  if (!address.length) return 'address를 입력해주세요.';
  return '';
};
```

```
export default function Checkout() {
  // return 코드 먼저 보고 파악
  const [state, dispatch] = useReducer(reducer, initialState);
  const { error, success, loading } = state;

  const [email, setEmail] = useState('');
  const [address, setAddress] = useState('');
  const [formError, setFormError] = useState('');

  const handleSubmit = e => {
    e.preventDefault();
    // 먼저 formError를 체크하세요.
    const formError = validateForm(email, address);
    setFormError(formError);
    // 에러가 있을 경우, 요청을 보내지 않도록 합니다.
    if (formError) return;
    // placeOrder API를 이용해 요청을 보내세요.
    // API 요청, 성공, 실패의 경우 dispatch를 이용해 상태를 처리하세요.

    dispatch({ type: 'request' });

    placeOrder({ email, address })
      .then(() =>
        dispatch({
          type: 'success',
          payload: { message: '주문에 성공하였습니다. 내일 만나요!' },
        })
      )
      .catch(err =>
        dispatch({ type: 'error', payload: { message: err.message } })
      );
  };

  return (
    <Container>
      <form onSubmit={handleSubmit}>
        <FormGroup>
          <label htmlFor="email">Email</label>
          <Input
            id="email"
            type="text"
            name="email"
            value={email}
            onChange={e => setEmail(e.target.value)}
            autoComplete="off"
          />
        </FormGroup>

        <FormGroup>
          <label htmlFor="address">Address</label>
          <Input
            id="address"
            type="text"
            name="address"
```

```
            value={address}
            onChange={e => setAddress(e.target.value)}
            autoComplete="off"
          />
        </FormGroup>

        <Button disabled={loading}>주문하기</Button>
      </form>

      <Message>{formError || error || success}</Message>
      // 아~ error, success 상태값 (메시지)가 여기서 보여지는 구나~
    </Container>
  );
}

// 추가 스타일링을 추가해도 좋습니다.
const Container = styled.div`
  border: 5px solid #f1f3f5;
  border-radius: 4px;

  width: 350px;
  height: 350px;

  display: flex;
  align-items: center;
  justify-content: center;
  flex-direction: column;
`;

const Input = styled.input`
  width: 100%;
`;

const FormGroup = styled.div`
  width: 100%;
  & + & {
    margin-top: 4px;
  }
`;

const Button = styled.button`
  margin-top: 12px;
  width: 100%;
  padding: 4px;
`;

const Message = styled.div`
  word-break: break-all;
  font-size: 14px;
  margin-top: 8px;
`;
```

```
// api.js

const API_REQUEST_TIME = 500;

export const placeOrder = (formData) => // promise 선물보따리 반환
  new Promise((resolve, reject) => {
    const { email, address } = formData;

    setTimeout(() => {
      if (!email || !address || email.length < 1 || address.length < 1) {
        return reject(new Error("email, address 정보를 정확히 입력해주세요."));
      } // 여기서 실제 db에 주문을 생성하고 그게 성공하면 resolve 함수 호출하고  이런 코드를 만들 순 없으
니까 단순 구현

      resolve({ email, address });
    }, API_REQUEST_TIME);
  });
```

---

3. 주문 내역 추가하기

▼ 문제 정답

```
// App.js

import React, { useEffect, useState, useCallback, useReducer } from 'react';
import { Link, Route, BrowserRouter } from 'react-router-dom';
import styled from 'styled-components';
import { getAllOrders, placeOrder } from './api';

const initialState = {
  success: '',
  error: '',
  loading: false,
};

const reducer = (state, action) => {
  switch (action.type) {
    case 'request': {
      return {
        ...state,
        success: '',
        error: '',
        loading: true,
      };
```

```
    }

    case 'success': {
      return {
        ...state,
        loading: false,
        success: action.payload.message,
      };
    }

    case 'error': {
      return {
        ...state,
        loading: false,
        error: action.payload.message,
      };
    }

    default:
      return state;
  }
};

const validateForm = (email, address) => {
  if (email.length === 0) {
    return 'email을 입력해주세요.';
  } else if (address.length === 0) {
    return 'address를 입력해주세요.';
  }
  return '';
};

export default function App() {
  return (
    <BrowserRouter>
      <Route exact path="/"> // exact를 쓰지 않으면 /checkout에도 적용되어 orderlist, checkout
컴포넌트 모두 렌더링되어요
        <OrderList />
      </Route>
      <Route path="/checkout">
        <Checkout />
      </Route>
    </BrowserRouter>
  );
}

// "/"
function OrderList() {
  const [orders, setOrders] = useState(undefined);

  useEffect(() => {
    getAllOrders().then(res => setOrders(res));
  }, []);

  return (
```

```
    <PageLayout>
      <Container>
        <Header>
          <h3>주문 내역</h3>
          <StyledLink id="checkout" to="/checkout">주문하러가기</StyledLink>
          // id 안넣어줘도 되는데 테스팅코드짤 때 필요하여 넣어준 것 같아요
        </Header>
      </Container>

      {!orders ||
        (orders.length === 0 && <Message>주문내역이 없습니다.</Message>)}
      {orders && (
        <StyledOrderList>
          {orders.map(({ email, address, date }, idx) => (
            <OrderItem key={idx}>
              <div>이메일 - {email}</div>
              <div>주소 - {address}</div>
              <div>주문 시간 - {date}</div>
            </OrderItem>
          ))}
        </StyledOrderList>
      )}
    </PageLayout>
  );
}

// OrderList 페이지를 추가하세요.
// getAllOrders API 함수로 데이터를 보여주세요.
// Checkout 페이지로 가는 네비게이션을 추가합니다.

// /checkout
function Checkout() {
  const [state, dispatch] = useReducer(reducer, initialState);
  const { error, success, loading } = state;

  const [email, setEmail] = useState('');
  const [address, setAddress] = useState('');
  const [formError, setFormError] = useState('');

  const handleSubmit =
    e => {
      e.preventDefault();
      const formError = validateForm(email, address);
      setFormError(formError);
      if (formError) return;
      dispatch({ type: 'request' });
      placeOrder({ email, address })
        .then(() =>
          dispatch({
            type: 'success',
            payload: { message: '주문에 성공하였습니다. 내일 만나요!' },
          })
        )
        .catch(e =>
          dispatch({ type: 'error', payload: { message: e.message } })
```

```
        );
    };

  return (
    <Container>
      <form onSubmit={handleSubmit}>
        <FormGroup>
          <label htmlFor="email">Email</label>
          <Input
            id="email"
            type="text"
            name="email"
            value={email}
            onChange={e => setEmail(e.target.value)}
            autoComplete="off"
          />
        </FormGroup>

        <FormGroup>
          <label htmlFor="address">Address</label>
          <Input
            id="address"
            type="text"
            name="address"
            value={address}
            onChange={e => setAddress(e.target.value)}
            autoComplete="off"
          />
        </FormGroup>

        <Button disabled={loading}>주문하기</Button>
      </form>

      <Message>{formError || error || success}</Message>

      {success && (
        <StyledLink id="home" style={{ margin: '12px auto' }} to="/">
          목록으로 돌아가기
        </StyledLink>
      )}
    </Container>
  );
}

const Container = styled.div`
  width: 90%;
  max-width: 90%;
`;

const PageLayout = styled.div`
  border: 5px solid #f1f3f5;
  border-radius: 4px;

  width: 350px;
  max-height: 600px;
```

```
  display: flex;
  align-items: center;
  justify-content: center;
  flex-direction: column;
`;

const Input = styled.input`
  width: 100%;
`;

const FormGroup = styled.div`
  width: 100%;
  & + & {
    margin-top: 4px;
  }
`;

const Button = styled.button`
  margin-top: 12px;
  width: 100%;
  padding: 4px;
`;

const Message = styled.div`
  word-break: break-all;
  font-size: 14px;
  margin-top: 8px;
`;

const StyledLink = styled(
  Link
)` // 저저번 강의 styled-component 상속 개념처럼 컴포넌트 확장 가능!
  display: inline-block;

  text-decoration: none;
  color: #343a40;
  background: #f1f3f5;
  padding: 12px;

  :hover {
    background: #e9ecef;
  }
`;

const Header = styled.header`
  display: flex;
  justify-content: space-between;
  align-items: center;
`;

const Image = styled.img`
  width: 100px;
`;
```

```
const OrderItem = styled.li`
  list-style: none;

  & + & {
    padding-top: 4px;
    margin-top: 4px;
    border-top: 3px solid #f1f3f5;
  }
`;

const StyledOrderList = styled.ul`
  padding: 0;
`;
```