React

React?

일반적인 JavaScript만을 이용해서도 충분히 웹앱을 만들 수 있지만 귀찮은 DOM관리와 상태값 업데이트 관리를 최소화하고, 오직 기능 개발, 그리고 UI를 구현하는 것에 집중할 수 있도록 하기 위해 Angular, Vue, Backbone, React 등의 여러 라이브러리 혹은 프레임워크가 만들어졌습니다.

그 중에서도 React는 가장 대표적인 JavaScript 라이브러리로 컴포넌트 라는 개념에 집중되어 있는 라이브러리입니다. 페이스북에 의해 탄생했으며, 많은 개발자들이 사용하기 때문에 생태계 또한 매우 넓다는 것 또한 큰 장점이 됩니다.

Library vs Framework

- Library
  - 소프트웨어를 개발하기 쉽게 어떤 기능을 제공하는 도구들
  - 어떤 기능을 쓰던 안쓰던 개인의 자유
  - `React` `jQuery` 등
- Framework
  - 특정 프로그램을 개발하기 위한 여러 요소들과 메뉴얼인 룰을 제공하는 프로그램
  - 꼭 지켜야 하는 규칙이 있다.
  - `Angular` `Spring` `Django` `Ruby on Rails`

페이스북은 왜 리액트를 만들게 됐을까?

React가 존재하기 이전에도 이미 Angular, Ember, Backbone 등의 프레임워크가 존재했으며, 해당 프레임워크들은 데이터 단을 담당하는 모델(Model), 사용자의 화면에서 보여지게 되는 뷰(View), 사용자가 발생시키는 이벤트를 처리해주는 컨트롤러(Controller)로 이루어진 MVC 패턴, 그리고 여기서 파생된 MVVM, MVW 등의 패턴으로 이루어져 있었습니다.

여기서 공통점이 모델인데, 이전 프레임워크들은 대부분 양방향 데이터 바인딩을 통해 모델에 있는 값이 변하면, 뷰에서도 이를 변화시켜줍니다. 여기서 핵심이 **변화시켜준다** 인데 첫 화면을 보여주고, 변화에 따라 필요한 곳을 바꿔주는 것입니다.

하지만 이런 변화(Mutation)은 상당히 복잡했습니다. 특정 이벤트가 발생했을 때, 모델에 변화를 일으키고, 변화를 일으킴에 따라 어떤 DOM 을 가져와 어떤 방식으로 뷰를 업데이트 시켜줄지 로직을 정해줘야 했는데, 페이스북에서는 이런 발상을 했습니다.

> Mutation을 하지 말고, 대신 데이터가 바뀌면 뷰를 날려버리고 새로 만들어버리자.

그런데 무슨 새로 만들어지는게 뚝딱 되는 것도 아니고 성능적으로 반드시 문제가 있을 것이었습니다. 그리고 그래서 사용하게 된 것이 Virtual DOM 입니다.

Virtual DOM

Virtual DOM은 가상의 DOM 입니다. 변화가 일어나면 실제로 브라우저의 DOM에 새로운 것을 넣는 것이 아니라, Virtual DOM에 한 번 렌더링을 해주고, 기존의 DOM과 비교한 이후 변화가 필요한 곳만 업데이트를 해줍니다. 이 Virtual DOM을 사용함으로서, 데이터가 바뀌었을 때 어떻게 업데이트 할 지를 고려하는게 아니라, 그냥 일단 바뀐 데이터로 Virtual DOM에 그린 다음 비교를 통해 바뀐 부분만 찾아서 바꾸게 되었습니다.

조금 더 자세하게 보겠습니다. 분명히 위에서 DOM 조작이 비효율적이고 느리기 때문에 Virtual DOM이 사용됐다고 했습니다. 하지만 JavaScript 엔진은 분명 좋아지고 있습니다. 그렇다면 대체 뭐가 불편해서 쓰게 됐을까요?

정확히 말하자면 DOM 조작이 전체 동작을 비효율적으로 만드는 것이 아니라, 그 이후에 일어나는 일 때문에 작업이 더뎌지는 것입니다.

브라우저는 HTML을 전달받으면, 브라우저의 렌더 엔진이 이를 파싱하고 DOM 트리를 만들게 됩니다. 이후, CSS 파일을 파싱해 DOM 트리에 새로운 트리인 렌더 트리를 만들게 됩니다. 렌더 트리를 만드는 과정에서, 각 요소들의 스타일이 계산되고, 이 계산 과정에서 다른 요소들의 스타일 속성들을 참조하게 됩니다. 당연히 이리저리 얽혀있으니 그럴 수 밖에 없습니다. 그 이후, 레이아웃 과정을 거쳐 각 노드들의 위치가 정해지고, 페인팅 작업을 통해 색이 입혀지면서 스크린에 원하는 화면이 나오게 됩니다.

간단하게 하자면, DOM 에 변화가 생기면 렌더 트리를 재생성하면서 모든 요소들의 스타일이 다시 계산되고, 레이아웃을 만들고, 페인팅을 하는 과정을 반복하는 것입니다. 그리고 복잡한 SPA에서는 DOM 조작이 시도때도 없이 발생합니다. 그 뜻은 브라우저의 연산 횟수가 늘어난다는 것이고, 전체적인 프로세스가 비효율적이게 된다는 뜻입니다.

하지만 Virtual DOM을 사용하게 된다면 뷰에 변화가 생길 시, 그 변화는 Virtual DOM에 먼저 적용이 되고, 최종적인 결과를 실제 브라우저의 DOM에 전달하기 때문에 브라우저 내에서의 연산이 줄어든다는 것입니다.

Virtual DOM은 실제 브라우저에 있는 DOM이 아니기 때문에 적용이 간단합니다. 실제 렌더링이 일어나지 않기 때문에 각 요소들의 스타일이 다시 계산될 일이 없어 연산 비용이 적습니다. 그리고 그 최종적인 결과물을 실제 DOM에 딱 1번만 던져주기 때문에 실제 DOM에서의 레이아웃 계산과 리렌더링 규모는 커지겠지만 딱 1번만 하는 것이기 때문에 실제 연산의 횟수는 압도적으로 줄어듭니다. 일반 DOM만을 사용해 조작 1번당 리렌더링, 레이아웃, 페인팅 과정을 전체 노드에 작업하는 것과는 비교도 되지 않습니다. 그리고 이러한 장점들이 Virtual DOM을 쓰는 이유입니다.

하지만 물론 리액트만 Virtual DOM을 쓰는 것은 아닙니다. Vue, Marko 등등 여러 라이브러리나 프레임워크에서 사용하고 있죠.

리액트가 특별한 이유

여러 곳에서 Virtual DOM을 쓰면 리액트를 고집할 필요가 없습니다. 하지만 분명히 장점이 있기에 리액트를 쓸 텐데요. 리액트가 특별해질 수 있었던 이유는 기능적인 이유 보다는 가장 사랑을 많이 받았기 때문입니다.

기술적으로 특별할 내용은 많이 없습니다. 오히려 타 프레임워크에 비해 불편하다는 의견도 있습니다. 하지만 가장 많은 사람들이 사용하는 도구가 되었기 때문에 생태계가 가장 넓으며, 사용하는 회사도 많습니다. 생태계가 넓다는 것은 수많은 사람이 리액트를 더 잘 사용하기 위한 도구를 만들고 있으며, 정보를 찾기도 쉽다는 뜻입니다. 그리고 사용하는 곳이 많다는 것은 리액트를 사용하는 프로젝트가 많다는 것이고, 저희가 해당 프로젝트에 참여하기 위해서는 반드시 리액트를 익혀야 된다는 뜻입니다.

함수형 컴포넌트, 클래스형 컴포넌트

리액트는 컴포넌트 기반 라이브러리라고 이전에 설명드렸습니다. 그리고 리액트에는 이 컴포넌트를 2가지 방식으로 만들 수 있습니다.

리액트 초기에는 클래스형 컴포넌트 사용이 압도적으로 많았습니다. 가장 큰 이유는 LifeCycle API, State 의 사용 가능 유무였습니다. 함수형 컴포넌트에서는 해당 내용들을 사용할 수 없었고, 클래스형 컴포넌트에서만 사용이 가능했기 때문에 함수형 컴포넌트를 사용할 이유가 없었습니다.

하지만 리액트 16.8 버전부터 Hook의 개념이 등장했고, Hook의 등장으로 함수형 컴포넌트에서 LifeCycle API, state의 사용이 가능해져 현재 대부분의 리액트 프로젝트는 함수형 컴포넌트 기반으로 동작하고 있습니다.

State, LifeCycle API

state는 동적인 변수를 의미합니다. 예를 들어 변하지 않을 변수라면 굳이 state로 사용할 필요 없이 let, const 등을 이용해 일반적인 변수 선언을 해도 상관없지만 동적인 데이터라면 반드시 state를 통해 변수를 선언해야 합니다. 본래 클래스형 컴포넌트에서만 사용 가능했지만 useState라는 Hook의 등장으로 인해 함수형 컴포넌트에서도 사용이 가능해졌습니다.

LifeCycle API는 렌더링 과정 중 사용할 수 있는 함수들을 의미합니다. 클래스형 컴포넌트에서 사용이 가능하고 componentDidMount, componentWillMount 등등 여러 함수들이 있지만 대부분 함수형 컴포넌트로 넘어오면서 해당 함수들을 외울 필요는 없어졌습니다.(사용할 일이 웬만하면 없습니다.)

대신 Hook의 등장으로 LifeCycle API 와 유사하게 사용할 수 있는 함수들이 몇몇 생겼습니다. 그리고 그 중 가장 대표격은 역시 useEffect 입니다.

useEffect

useEffect를 이용하면 컴포넌트가 마운트 됐을 때(처음 나타났을 때), 언마운트 됐을 때(사라질 때), 업데이트 될 때(특정 데이터가 바뀔 때) 특정 작업을 처리할 수 있습니다.

```
// 마운트/언마운트 일 때
// 보통 마운트 될 때 해야 할 작업은
// 1. props로 받은 값을 로컬 상태로 저장
// 2. 외부 API dycjd
// 3. 라이브러리 사용 등

// 언마운트 될 때 해야 할 작업
// 1. setInterval, setTimeout 을 사용한 작업을 clear
// 2. 라이브러리 인스턴스 제거 등
// 이 있습니다.
useEffect(() => {
  console.log("마운트~")

  // 이렇게 useEffect 안에서 반환되는 함수를 cleanup 함수라고 하고
  // useEffect에 대한 마무리 작업을 해준다고 하면 됩니다.
  // deps가 없을 경우 컴포넌트가 사라질 때 실행됩니다.
  return () => {
    console.log("언마운트~")
  }
}, []) // 이 배열 안에 들어갈 내용을 deps라고 하며 deps가 없을 경우 useEffect는 초기 실행 시점에서 실행됩니다.

// 데이터가 바뀔 때
useEffect(() => {
  console.log("user 값이 바뀜~")
}, [user]) // deps에 특정 값이 들어가면 해당 값이 바뀔 때마다 useEffect 안 내용을 실행해 줍니다.

// 아예 deps를 생략했을 때
useEffect(() => {
  console.log("리렌더링 될 때마다 실행")
}) // deps를 아예 생략한 경우 컴포넌트가 리렌더링 될 때마다 실행됩니다.
```

useMemo

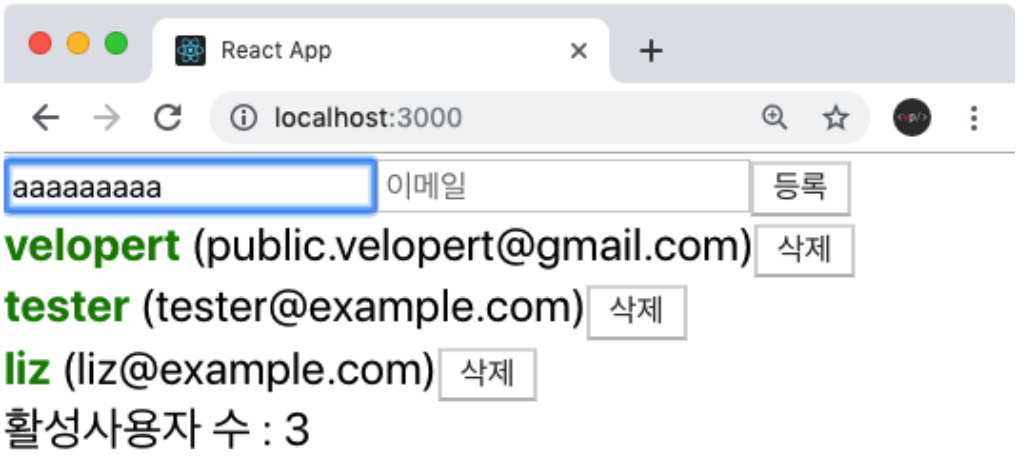
useMemo에서 Memo는 memoized를 의미하며, 이는 이전에 계산 한 값을 재사용한다는 의미를 갖고 있습니다.

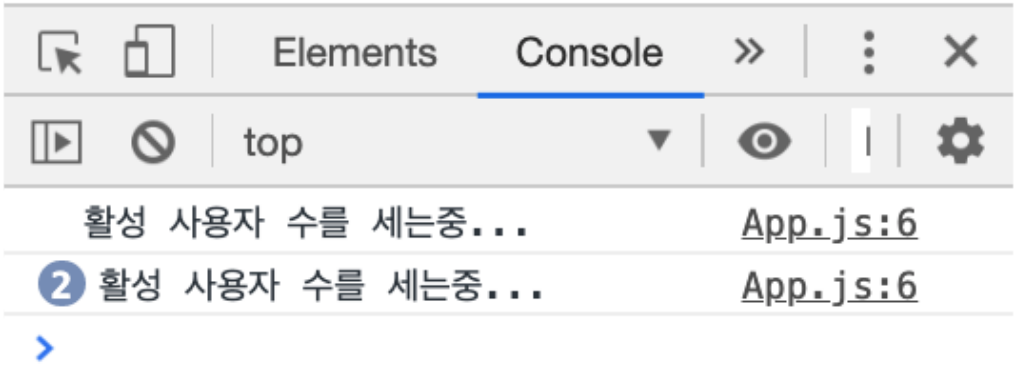
```
useMemo(() => { 작업 }, [deps]);
```

useMemo의 첫 번째 파라미터에는 어떻게 연산할지 정의하는 함수를 넣어주고, 두 번째 파라미터에는 deps 배열을 넣어주면 되는데, deps의 값이 바뀌면, 우리가 등록한 함수를 호출해서 값을 연산하고, 만약에 내용이 바뀌지 않았다면 이전에 연산한 값을 재사용하게 됩니다. 예시는 다음과 같이 쓸 수 있습니다.

```
import React, { useRef, useState, useMemo } from 'react';
import UserList from './UserList';
import CreateUser from './CreateUser';

function countActiveUsers(users) {
  console.log('활성 사용자 수를 세는중...');
  return users.filter(user => user.active).length;
}

function App() {
  const [inputs, setInputs] = useState({
    username: '',
    email: ''
  });
  const { username, email } = inputs;
  const onChange = e => {
    const { name, value } = e.target;
    setInputs({
      ...inputs,
      [name]: value
    });
  };
  const [users, setUsers] = useState([
    {
      id: 1,
      username: 'velopert',
```

```
      email: 'public.velopert@gmail.com',
      active: true
    },
    {
      id: 2,
      username: 'tester',
      email: 'tester@example.com',
      active: false
    },
    {
      id: 3,
      username: 'liz',
      email: 'liz@example.com',
      active: false
    }
  ]);

  const nextId = useRef(4);
  const onCreate = () => {
    const user = {
      id: nextId.current,
      username,
      email
    };
    setUsers(users.concat(user));

    setInputs({
      username: '',
      email: ''
    });
    nextId.current += 1;
  };

  const onRemove = id => {
    // user.id 가 파라미터로 일치하지 않는 원소만 추출해서 새로운 배열을 만듬
    // = user.id 가 id 인 것을 제거함
    setUsers(users.filter(user => user.id !== id));
  };
  const onToggle = id => {
    setUsers(
      users.map(user =>
        user.id === id ? { ...user, active: !user.active } : user
      )
    );
  };
  const count = useMemo(() => countActiveUsers(users), [users]);
  return (
    <>
      <CreateUser
        username={username}
        email={email}
        onChange={onChange}
        onCreate={onCreate}
      />
      <UserList users={users} onRemove={onRemove} onToggle={onToggle} />
      <div>활성사용자 수 : {count}</div>
    </>
  );
}

export default App;
```



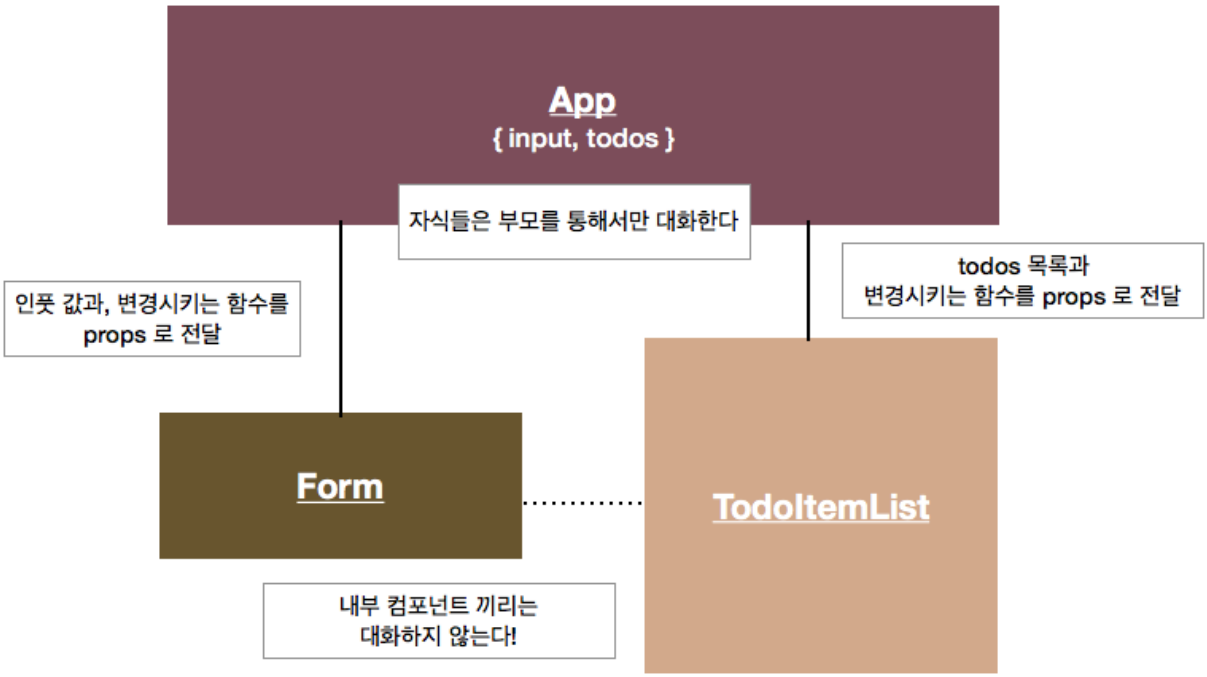


유저를 클릭하면 활성 사용자 수가 카운트 되는 컴포넌트입니다. 이때, useMemo를 사용하고 deps에 users를 넣어줌으로써, users가 바뀌어야만 count를 업데이트 해줍니다. 만약 해당 내용을 useMemo 없이 그냥 `count = countActiveUsers(users)` 라고 했다면 countActiveUsers 함수가 input에 뭘 입력만 해도 계속 실행 될 것이고, 이는 엄청난 비효율을 초래하게 됩니다.

Redux

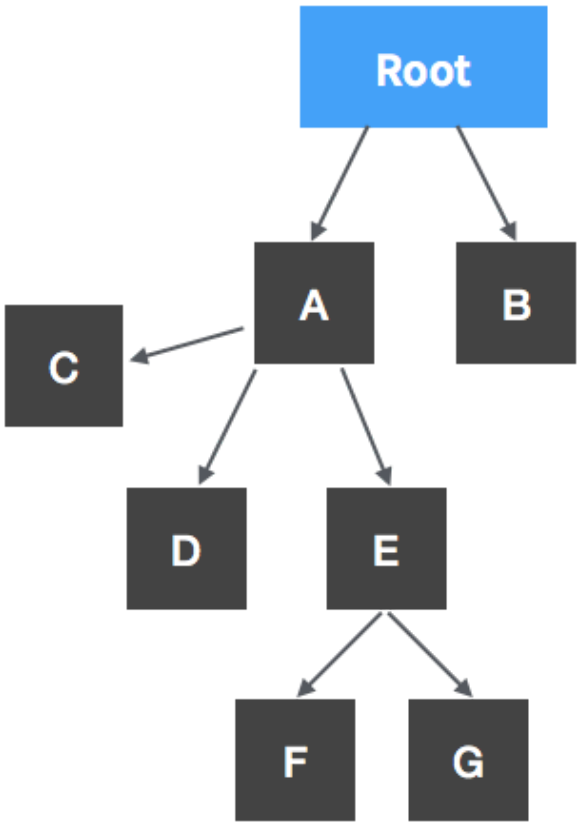
React의 상태 관리

Redux를 알기 전에 React의 상태 관리 방법에 대해 알아보겠습니다.

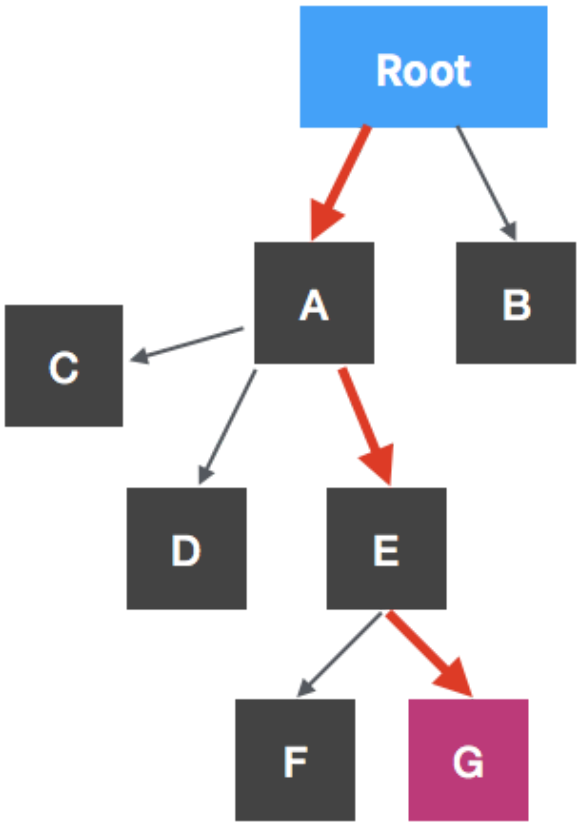


리액트 프로젝트에서는 대부분의 작업을 할 때 부모 컴포넌트가 중간자 역할을 합니다. 컴포넌트 끼리 직접 소통하는 방법도 있지만 그렇게 하면 코드가 굉장히 꼬이기 때문에 절대 권장되지 않습니다. 예를 들어 TodoList 를 만들었다고 생각해보겠습니다. 부모 컴포넌트인 App 컴포넌트에서 input, todos 라는 변수와 input 값을 변경시킬 onChange, todo 아이템을 만들어 낼 onCreate 함수를 생성해 각 자식들에게 주는 방식으로 투두리스트가 만들어질 것입니다.

이러한 구조는 부모 컴포넌트에서 모든걸 관리하기 때문에 매우 직관적이며, 오히려 규모가 작으면 작을수록 관리하는 것도 꽤 편하게 느껴집니다. 하지만 규모가 커진다면 App 컴포넌트의 코드가 넘칠 정도로 길어질 수밖에 없고 이에 따라 유지보수도 매우 힘들어 집니다. 예를 들어 아래와 같은 프로젝트 구조가 있다고 생각해봅시다.



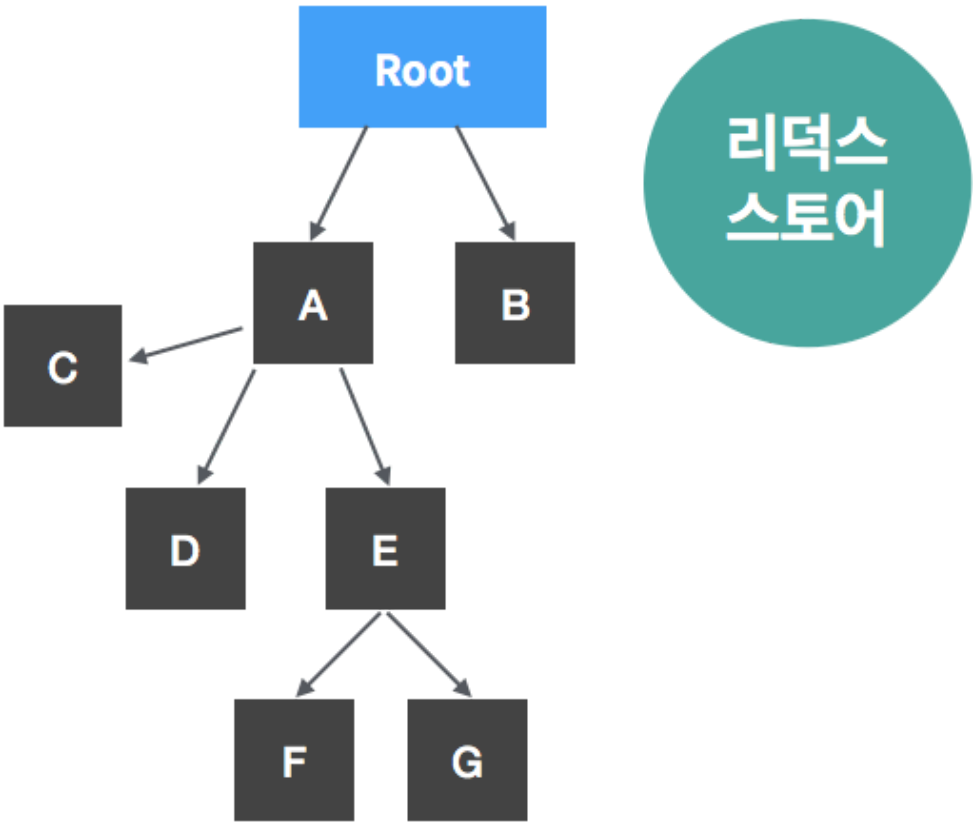
Root 컴포넌트에서 G 컴포넌트에게 어떠한 값을 전달해야 한다면 아래와 같이 이를 잇는 모든 컴포넌트를 거쳐야 합니다.



Redux!!

그럼 이제 리덕스로 넘어와 보겠습니다. 위에서 봤다시피 리액트의 state 관리는 상당히 힘든 구조를 가지고 있습니다. 하지만 Redux를 이용해 전역으로 상태를 관리한다면 정말 편해집니다.

리덕스에서 가장 중요한 개념은 **Store** 라고 생각합니다. 리덕스에서 스토어는 어느 컴포넌트에도 종속되지 않고 독립적인 구조를 가지고 있습니다. 아래 사진을 보면 바로 이해가 되실겁니다.



각 컴포넌트들은 스토어를 구독(subscribe)하고 있습니다. 유튜브 구독이랑 똑같습니다. 만약 G 컴포넌트에서 스토어의 count 라는 변수를 사용하고 있다면 다른 컴포넌트에서 count 변수를 업데이트 해도 G 컴포넌트에서 그대로 업데이트 된 값을 사용할 수 있게 됩니다.

그리고 여기서 스토어의 state를 업데이트 하는 것을 **dispatch** 라고 합니다. dispatch 시에는 반드시 action 이라는 객체를 스토어에게 던져주는데 이 객체 안에는 필수 값으로 **type**이 들어갑니다. 즉, 아래와 같은 구조를 띄게 되는 것입니다.

```
let action = { type: 'INCREMENT', ... }
dispatch(action)
```

type을 제외하고는 선택값이며 보통 변수값이 들어가고, 이 선택값을 payload 라고 부릅니다.

Reducer를 통해 상태 변화시키기

dispatch를 했으면 리덕스는 Reducer를 통해 상태를 업데이트 합니다. action 안에는 반드시 type 값이 있어야 한다고 했습니다. 그렇기 때문에 Reducer는 type에 따라 어떻게 상태를 업데이트 할 지 정의하고, 또한 상태를 업데이트 하는 함수입니다.

예를 들어 action: { type: 'INCREMENT', payload: 2 } 라고 하면 특정 변수에 2를 더해주는 작업을 수행하게 되겠죠. 이렇게 업데이트 로직을 정해주는 함수를 리듀서라고 합니다.

Reducer는 **state, action** 이라는 2가지 파라미터를 받습니다. action은 위에서 설명드린 action 그대로고, state는 **현재 상태**를 의미합니다. 즉, action에 따라 현재 state를 변경하면 되는거죠. 그리고 그렇게 특정 변수가 업데이트 되면 해당 변수를 사용하고 있던 컴포넌트는 자동으로 리렌더링을 하게 되는 것입니다.

Redux의 3가지 규칙

1. 하나의 어플리케이션 안에는 하나의 스토어가 있습니다.
2. 상태는 읽기전용 입니다.
    - 리액트에서 state를 업데이트 할 때는 setState를 사용하고, 배열을 업데이트 할 때는 배열 자체에 push를 하지 않고, concat 같은 함수를 사용해야 합니다. 리덕스 또한 마찬가지입니다. 기존의 상태를 건들이지 않고 새로운 상태를 생성하여 업데이트를 해주는 방식으로 해야합니다.
    - 리덕스에서 불변성을 유지해야 하는 이유는 내부적으로 데이터가 변경 되는 것을 감지하여 shallow equality 검사를 하기 때문입니다. 리덕스는 내부적으로 shallow equality 검사를 계속 진행하는데 이를 사용해 컴포넌트가 올바르게 업데이트 되려면 불변성이 필수입니다. 실제로 리덕스 공식 홈페이지에서 아래와 같은 문구를 확인할 수 있습니다.

    > Redux's use of shallow equality checking requires immutability if any connected components are to be updated correctly.

3. 변화를 일으키는 함수, 리듀서는 순수한 함수여야 합니다.

단점

이렇게 좋아보이는데… 요즘 추세는 Redux를 지양하는 쪽으로 가고 있습니다. 물론 여전히 대부분의 기업에서 Redux를 쓰고 있고 있지만 유명한 기업들에서 Redux 보단 다른 상태관리 라이브러리들을 사용하기 시작했죠.

Redux는 위에서 설명드린 대로 상태 관리를 위한 라이브러리 입니다. 그리고 여기서 맹점은 **상태 관리만을 위한 라이브러리** 라는 것입니다. 상태 관리를 위해서만 만들어졌으며 프레임워크가 아닌 라이브러리이기 때문에 필요한 다른 기능들이 있을 경우 또 다른 라이브러리들을 가져와 사용해야 합니다. 예를 들어, 서버 데이터를 사용하기 위해서는 Redux-saga, Redux-thunk, Redux-toolkit 등과 같은 또 다른 미들웨어를 필수적으로 사용해야 하죠. 그리고 이렇게 미들웨어가 많아질수록 코드의 길이가 길어지게 됩니다.

즉, Redux를 이용해 서버 데이터를 관리하게 되면 **서버 데이터를 위한 로직이 너무 커지고, 그로 인해 Redux를 활용하기 위한 보일러 플레이트가 비대해 진다는 것입니다.**

그리고 이런 문제를 해결하기 위해 대체 기술들을 여러 기업들이 찾아보았고, 한때는 swr 이 React-Query 보다 앞서가기 시작했으나 현재는 뒤집힌 상황이 되었고 앞으로도 격차가 커지지 않을까 됩니다.