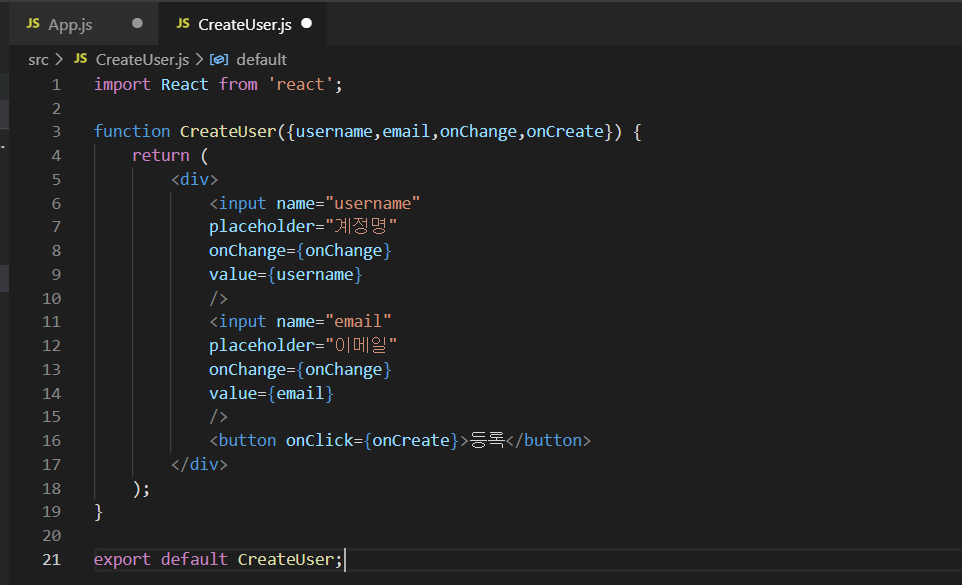
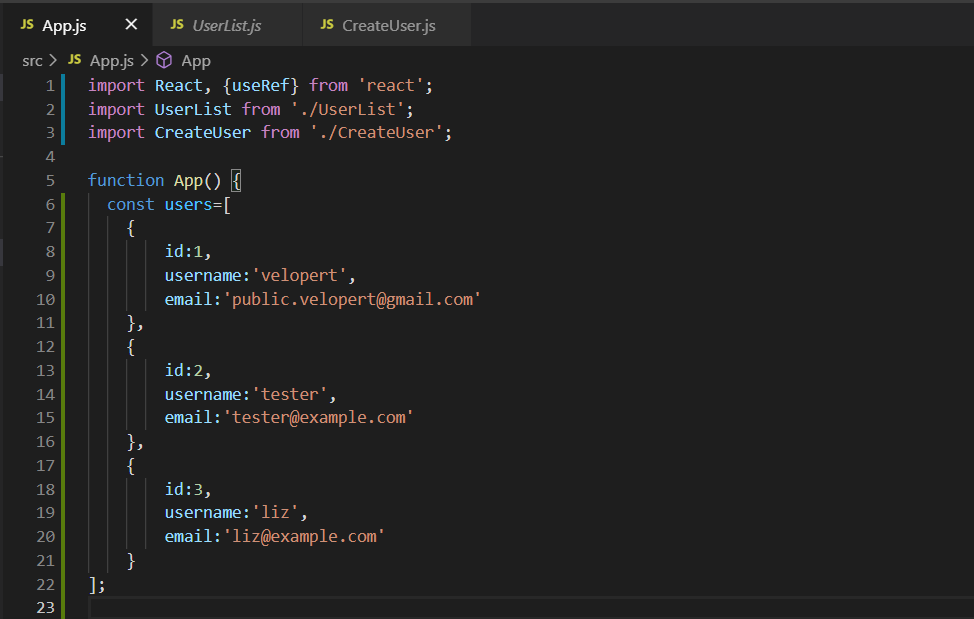
배열에 항목 추가하기

input 두 개와 button 하나로 이루어진 CreateUser.js라는 컴포넌트를 src 디렉토리에 만들어보자

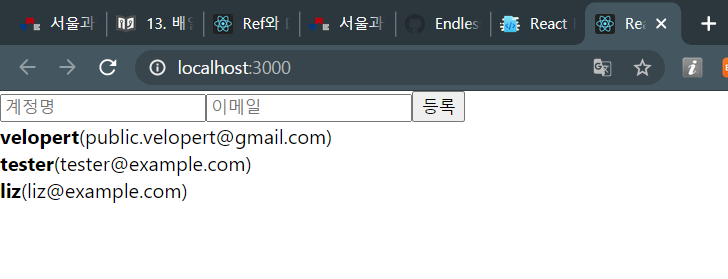


이번 컴포넌트에서는 상태 관리를 App에서 하고, input의 값 및 이벤트로 등록할 함수들을 props로 넘겨받아서 사용해보자. 이 컴포넌트를 App에서 UserList위에 렌더링해보자,





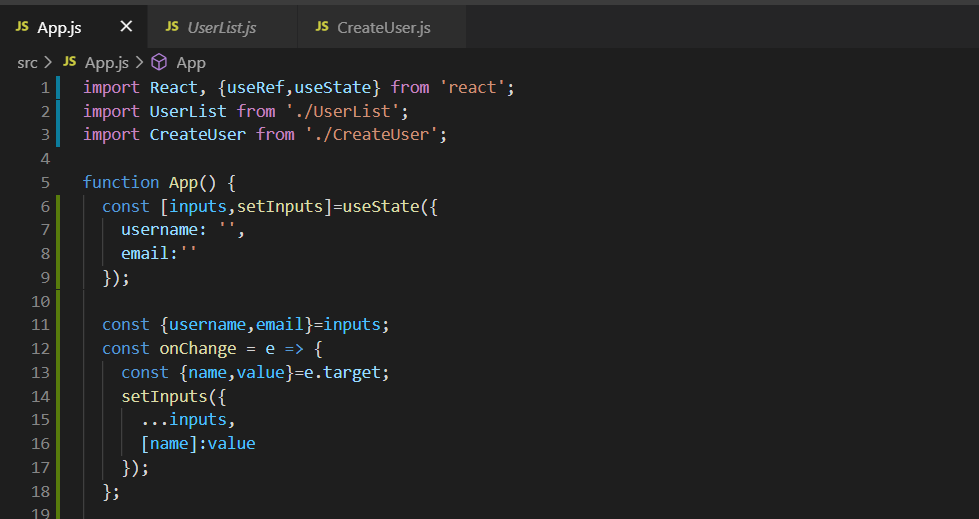
다음과 같이 작성하면 App.js를 작성하면 아래와 같은 결과를 얻을 수 있다. 아직 등록 버튼은 작동되지 않는 상태이다.



배열의 불변성을 지키면서 배열에 새 항목을 추가하는 방법은 두가지가 있다.

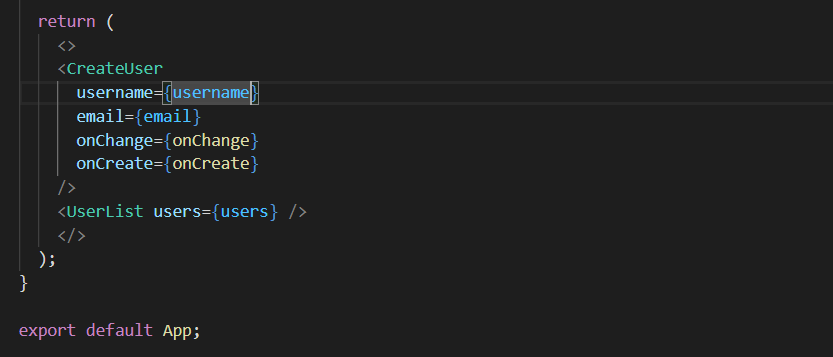
1. **spread 연산자(…)를 사용하는 것**

spread라는 단어가 가지고 있는 의미는 펼치다. 퍼뜨리다 이다. 이 문법을 사용하면, 객체 혹은 배열을 펼칠 수 있다.

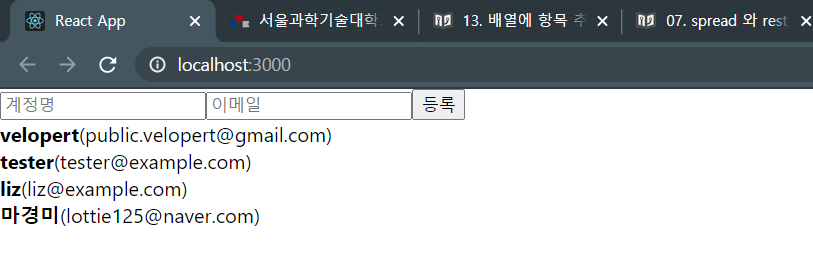




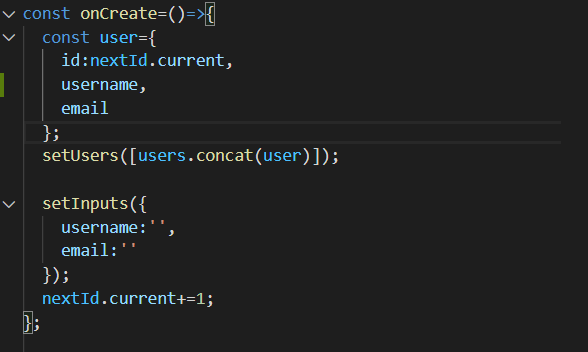




다음과 같이 작성하면 등록 버튼을 눌렀을 때 배열에 새로운 항목이 삽입되었음을 알 수 있다.



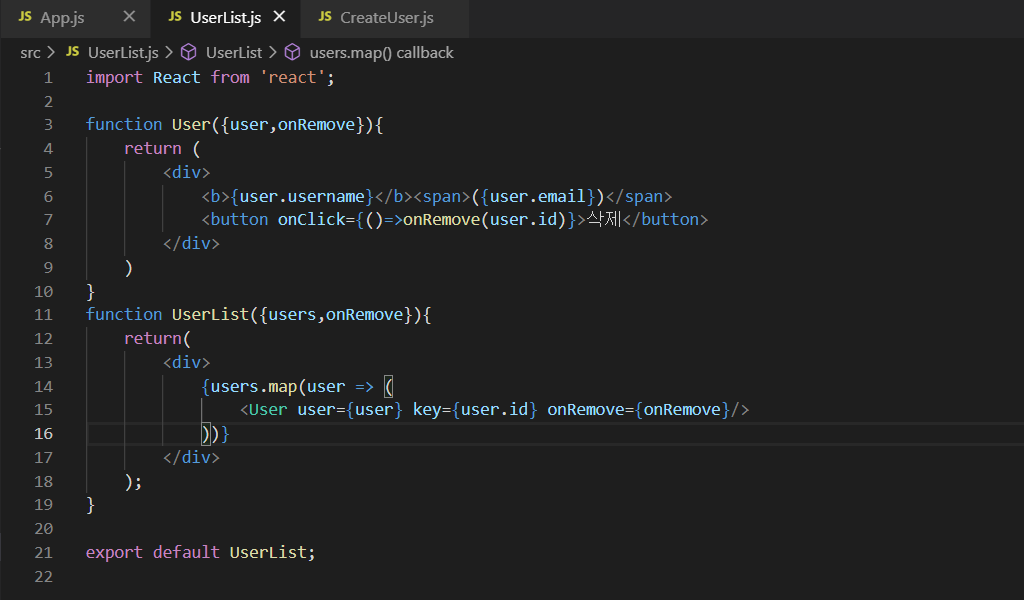
1. **concat 함수를 사용하여** 기존의 배열을 수정하지 않고, 새로운 원소가 추가된 새로운 배열을 만들 수 있다.



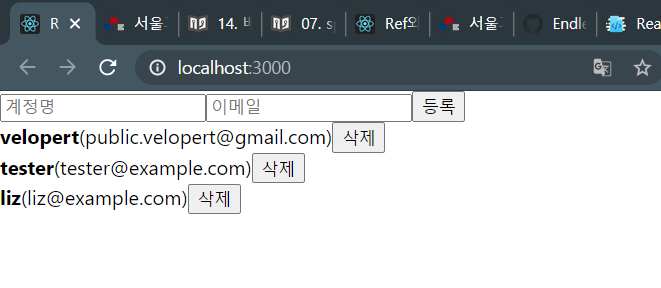
setUsers()부분을 다음과 같이 concat을 사용하여 정의한다

배열에 항목 제거하기

이번에는 배열에 항목을 제거하는 방법을 알아보자 UserList를 다음과 같이 수정한다



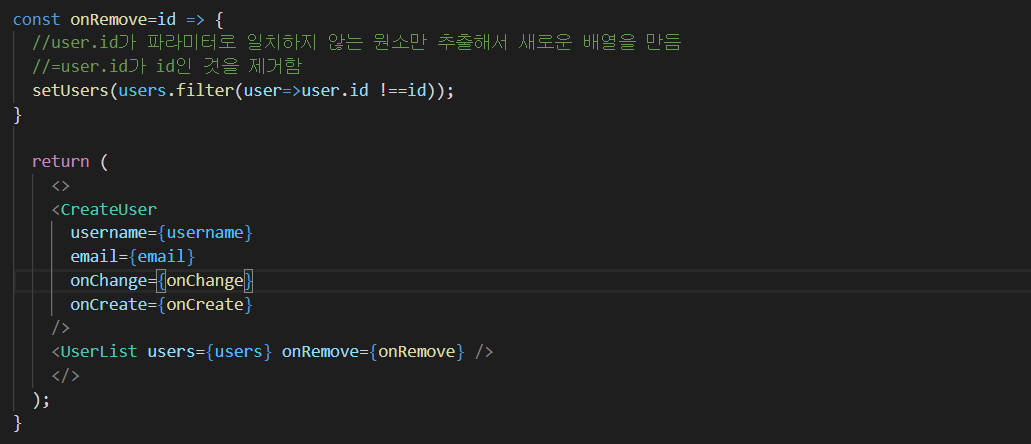
아래와 같이 삭제 버튼을 만들고 아직 삭제는 행동되지 않는다.



User컴포넌트의 삭제 버튼이 클릭 될 때에 user.id 값을 앞으로 props로 받아올 onRemove 함수의 파라미터로 넣어서 호출해줘야 한다. 이 때 onRemove의 역할은 id가 \_인 객체를 삭제하라 이다. 이 onRemove 함수는 UserList에서도 전달 받을 것이며, 이를 그대로 User 컴포넌트에게 전달해 줄 것이다.

이제 onRemove 함수를 불변성을 지키면서 구현해보자. **불변성을 지키면서 특정 원소를 배열에서 제거하기 위해선는 filter 배열 내장 함수를 사용하는 것이 가장 편하다.** 이 함수는 배열에서 특정 조건이 만족하는 원소들만 추출하여 새로운 배열을 만들어준다.

다음과 같이 App 컴포넌트에 onRemove 함수를 구현하자.



이와 같이 구현하면 삭제 버튼이 잘 작동됨을 알 수 있다.

배열 항목 수정하기

User 컴포넌트에 계정명을 클릭했을 때 색상이 초록색으로 바뀌고, 다시 누르면 검정색으로 바뀌도록 구현해보자.

App컴포넌트의 users 배열 안의 객체 안에 **active라는 속성을 추가한다.**

active 값에 따라 글자의 색상을 바꿀 수 있게 구현하고, 추가적으로 cursor 필드를 설정하여 마우스를 올렸을 때 커서가 손가락 모양으로 변하게 구현해 보자

이 때 App.js에서 onToggle 함수를 구현한다. 배열의 불변성을 유지하면서 배열을 업데이트 할 때에도 map 함수를 사용할 수 있다. id 값을 비교해서 id가 다르다면 그대로 두고, 같다면 active 값을 반전시키도록 구현을 한다. onToggle 함수를 만들어서 UserList 컴포넌트에게 전달해주자

UserList컴포넌트에서 onToggle을 받아와서 User에게 전달해주고, onRemove를 구현했던 것처럼 onToggle에 id를 넣어서 호출한다.

App.js

import React, {useRef,useState} from 'react';

import UserList from './UserList';

import CreateUser from './CreateUser';

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,user]);

  setInputs({

    username:'',

    email:''

  });

  nextId.current+=1;

};

const onRemove = id => {

  //user.id가 파라미터로 일치하지 않는 원소만 추출해서 새로운 배열을 만듬

  //=user.id가 id인 것을 제거함

  setUsers(users.filter(user=>user.id !==id));

}

const onToggle=id=>{

  setUsers(

    users.map(user=>

      user.id===id ? {...user,active: !user.active} :user

      )

  );

};

  return (

    <>

    <CreateUser

      username={username}

      email={email}

      onChange={onChange}

      onCreate={onCreate}/>

    <UserList users={users} onRemove={onRemove} onToggle={onToggle} />

    </>

  );

}

export default App;

UserList.js

import React from 'react';

function User({user,onRemove,onToggle}){

    return (

        <div>

            <b style={{

                cursor:'pointer',

                color:user.active ?'green':'black'

            }}

            onClick={()=>onToggle(user.id)}

            >

                {user.username}

                </b>&nbsp;

                <span>({user.email})</span>

            <button onClick={()=>onRemove(user.id)}>삭제</button>

        </div>

    );

}

function UserList({users,onRemove,onToggle}){

    return(

        <div>

            {users.map(user => (

                <User user={user}

                key={user.id}

                onRemove={onRemove}

                onToggle={onToggle}/>

            ))}

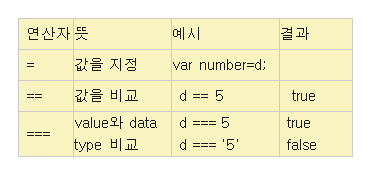
        </div>

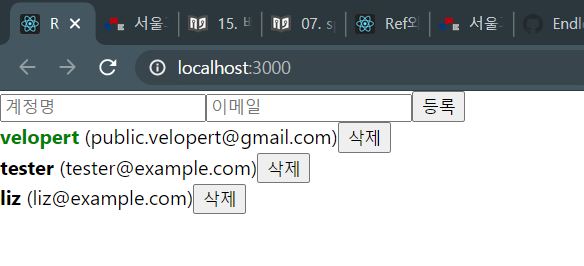
    );

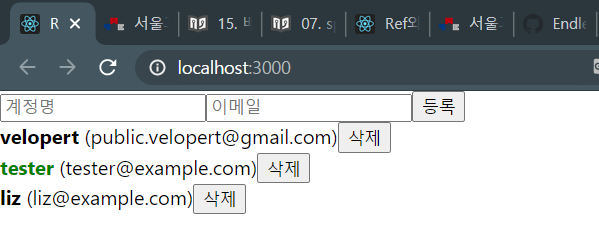
}

export default UserList;

다음과 같이 작성하면 아래와 같이 계정명을 누르면 초록색을 변하고 다시 눌렀을 때 검정색으로 변함을 볼 수 있다. 또한 캡쳐에는 나와있지 않지만 cursor 모양도 변한다.

이때 ===연산자는  value와 data type까지 비교하는 연산자이다.





UseEffect를 사용하여 마운트/언마운트/업데이트 시 할 작업 설정하기

useEffect라는 Hook을 사용하여 컴포넌트가 마운트 됐을 때 (처음 나타났을 때), 언마운트 됐을 때 (사라질 때), 그리고 업데이트 될 때 ( 특정 props가 바뀔 때) 특정 작업을 처리하는 방법에 대해서 알아보자.

마운트 = DOM에 노드 추가, 언마운트=DOM에서 제거, 업데이트= 이미 DOM에 있는 노드 변경

이라고 생각하면 될 것 같다

**마운트/ 언마운트 관리**

useEffect를 사용할 때에는 첫번째 파라미터에는 함수, 두번째 파라미터에는 의존값이 들어있는 배열(deps)을 넣습니다. 만약에 deps 배열을 비우게 된다면, 컴포넌트가 처음 나타날때에만 useEffect에 등록한 함수가 호출됩니다.

useEffect,에서는 함수를 반환 할 수 있는데 이를 cleanup함수라고 부릅니다. cleanup 함수는 useEffect에 대한 뒷정리를 해준다고 보면 된다, deps가 비어있는 경우에는 컴포넌트가 사라질 때 cleanup 함수가 호출된다.

UserList.js를 다음과 같이 수정하고 항목을 추가/삭제를 하면서 콘솔을 확인해보자.

import React,{useEffect} from 'react';

function User({user,onRemove,onToggle}){

    useEffect(()=>{

        console.log('컴포넌트가 화면에 나타남');

        return()=>{

            console.log('컴포넌트가 화면에서 사라짐');

        };

    },[]);

    return (

        <div>

            <b style={{

                cursor:'pointer',

                color:user.active ?'green':'black'

            }}

            onClick={()=>onToggle(user.id)}

            >

                {user.username}

                </b>&nbsp;

                <span>({user.email})</span>

            <button onClick={()=>onRemove(user.id)}>삭제</button>

        </div>

    );

}

function UserList({users,onRemove,onToggle}){

    return(

        <div>

            {users.map(user => (

                <User user={user}

                key={user.id}

                onRemove={onRemove}

                onToggle={onToggle}/>

            ))}

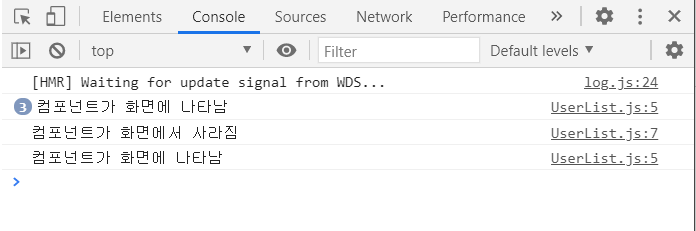
        </div>

    );

}

export default UserList;

다음과 같이 console에 뜸을 볼 수 있다.



마운트 시에 하는 작업들

* props로 받은 값을 컴포넌트의 로컬 상태로 설정
* 외부 API 요청(REST API 등)
* 라이브러리 사용(D3,Video.js등)
* setInterval을 통한 반복작업 혹은 setTimeout을 통한 작업 예약

언마운트 시에 하는 작업들

* setInterval,setTimeout을 사용하여 등록한 작업들 clear하기
* 라이브러리 인스턴스 제거

리액트 컴포넌트에는 일반적으로 두 종류의 side effects가 있다. 정리(clean up)이 필요한 것과 그렇지 않은 것으로 나뉜다

**clean up을 이용하지 않는 effects**

리액트가 DOM을 업데이트한 뒤 추가로 코드를 실행해야 하는 경우가 있다. 네트워크 리퀘스트, DOM 수동 조작, 로깅 등은 clean up이 필요 없는 경우이다.

useEffect Hook을 이용하여 우리는 리액트에게 컴포넌트가 렌더링 이후에 어떤 일을 수행해야 하는 지를 말한다. 리액트는 우리가 넘긴 함수를 기억했다가 (이 함수를 effect라 한다) DOM 업데이트를 수행한 이후에 불러낼 것이다. 데이터를 가져오거나 다른 명령형 API를 불러내는 일을 할 수 있다.

**clean up을 이용하는 effects**

외부 데이터에 구독(subscription)을 설정해야 하는 경우 메모리 누수가 발생하지 않도록 clean up 하는 것이 매우 중요하다.

cleanup의 실행을 위해 별개의 effect가 필요하다고 생각할 수도 있지만 구독(subscription)이 추가와 제거를 위한 코드는 결합도가 높기 때문에 useEffect는 이를 함께 다루도록 고안되었다. effect가 함수를 반환하면 리액트는 그 함수를 정리가 필요한 때에 실행시킨다.

effect에서 함수를 return하는 이유는 추가적인 cleanup메커니즘이다. 모든 effect는 정리를 위한 함수를 반환할 수 있다. 이 점이 subscription의 추가와 제거를 위한 로직을 가까이 묶어둘 수 있다. 구독의 추가와 제거가 모두 하나의 effectㄹ르 구성하는 것이다.

**deps에 특정 값 넣기**

deps에 특정 값을 넣어보자. 컴포넌트가 처음 마운트 될 때에도 호출이 되고, 지정한 값이 바뀔 때에도 호출이 된다. 그리고 deps 안에 특정 값이 있다면 언마운트시에도 호출이 되고, 값이 바뀌기 직전에도 호출이 된다.

UserList.js의 useEffect 부분을 다음과 같이 수정해보자

 useEffect(()=>{

        console.log('user 값이 설정됨');

        console.log(user);

        return()=>{

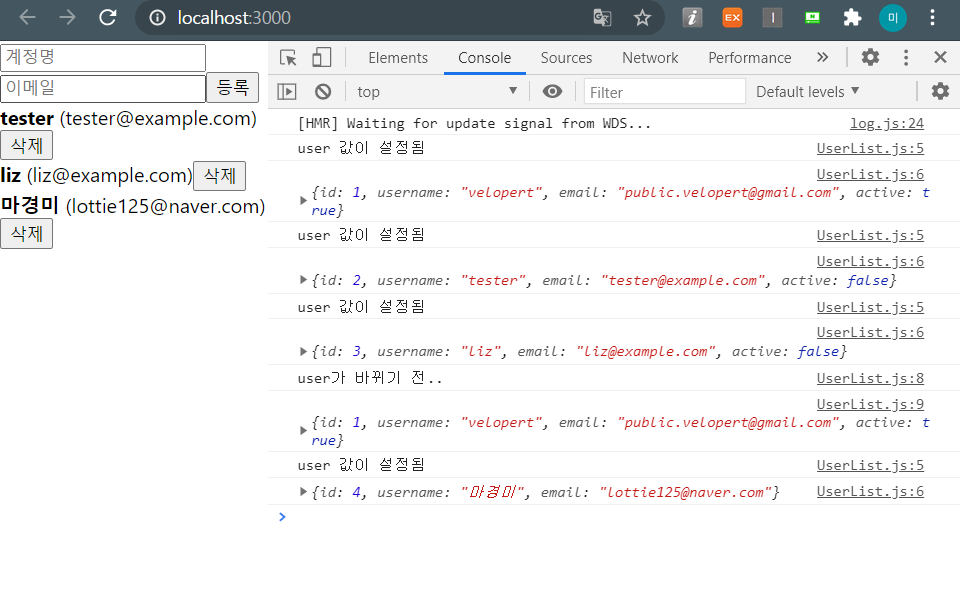
            console.log('user가 바뀌기 전..');

            console.log(user);

        };

    },[user]);

그러면 console에 다음과 같이 뜬다.



useEffect안에서 사용하는 상태나, props가 있다면, useEfffect의 deps에 넣어주어야 한다.

만약 useEffect안에서 사용하는 상태나 props를 deps에 넎지 않게 된다면 useEffect에 등록한 함수가 실행될 때 최신 props/상태를 가리키지 않게 된다.

**deps 파라미터를 생략하기**

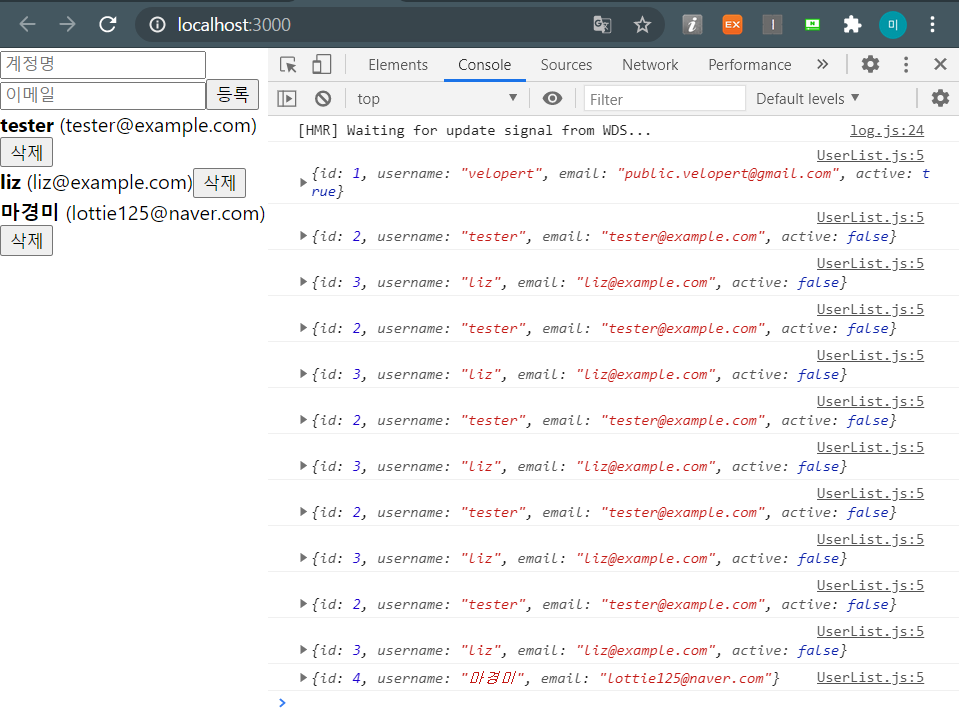
deps 파라미터를 생략한다면, 컴포넌트가 리렌더링 될 때마다 호출이 된다.

다음과 같이 UserList.js의 useEffect 부분을 수정하자.

useEffect(()=>{

        console.log(user);

    });



이와 같이 deps 파라미터를 사용했을 때 보다 더 많이 호출된 것을 알 수 있다.

리액트 컴포넌트는 기본적으로 부모 컴포넌트가 리렌더링 되면 바뀐 내용이 없다 할지라도 자식 컴포넌트 또한 리렌더링 된다. 실제 DOM에 변화가 반영되는 것은 바뀐 내용이 있는 컴포넌트에만 해당한다. 하지만 virtual DOM에는 모든걸 다 렌더링하고 있다는 것이다. 나중에는 컴포넌트를 최적화 하는 과정에서 기존의 내용을 그대로 사용하면서 virtual DOM에 렌더링 하는 리소스를 아낄 수 도 있다.