# 18. useCallback 을 사용하여 함수 재사용하기

useCallback 은 우리가 지난 시간에 배웠던 useMemo 와 비슷한 Hook 입니다. useMemo 는 특정 결과값을 재사용 할 때 사용하는 반면, useCallback 은 특정 함수를 새로 만들지 않고 재사용하고 싶을때 사용합니다.
이전에 App.js 에서 구현했었던 onCreate, onRemove, onToggle 함수를 확인해봅시다.

```
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
    )
  );
};
```

이 함수들은 컴포넌트가 리렌더링 될 때 마다 새로 만들어집니다. 함수를 선언하는 것 자체는 사실 메모리도, CPU 도 리소스를 많이 차지 하는 작업은 아니기 때문에 함수를 새로 선언한다고 해서 그 자체 만으로 큰 부하가 생길일은 없지만, 한번 만든 함수를 필요할때만 새로 만들고 재사용하는 것은 여전히 중요합니다.

그 이유는, 우리가 나중에 컴포넌트에서 props 가 바뀌지 않았으면 Virtual DOM 에 새로 렌더링하는 것 조차 하지 않고 컴포넌트의 결과물을 재사용 하는 최적화 작업을 할건데요, 이 작업을 하려면, 함수를 재사용하는것이 필수입니다.
useCallback 은 이런식으로 사용합니다.

## App.js

```
import React, { useRef, useState, useMemo, useCallback } 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 = useCallback(
    e => {
      const { name, value } = e.target;
      setInputs({
        ...inputs,
        [name]: value
      });
    },
    [inputs]
  );
  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 = useCallback(() => {
    const user = {
      id: nextId.current,
      username,
      email
    };
    setUsers(users.concat(user));

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

```
  const onRemove = useCallback(
    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));
    },
    [users]
  );
  const onToggle = useCallback(
    id => {
      setUsers(
        users.map(user =>
          user.id === id ? { ...user, active: !user.active } : user
        )
      );
    },
    [users]
  );
  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;
```

주의 하실 점은, 함수 안에서 사용하는 상태 혹은 props 가 있다면 꼭, `deps` 배열안에 포함시켜야 된다는 것 입니다. 만약에 `deps` 배열 안에 함수에서 사용하는 값을 넣지 않게 된다면, 함수 내에서 해당 값들을 참조할때 가장 최신 값을 참조 할 것이라고 보장 할 수 없습니다. props 로 받아온 함수가 있다면, 이 또한 `deps` 에 넣어주어야 해요.

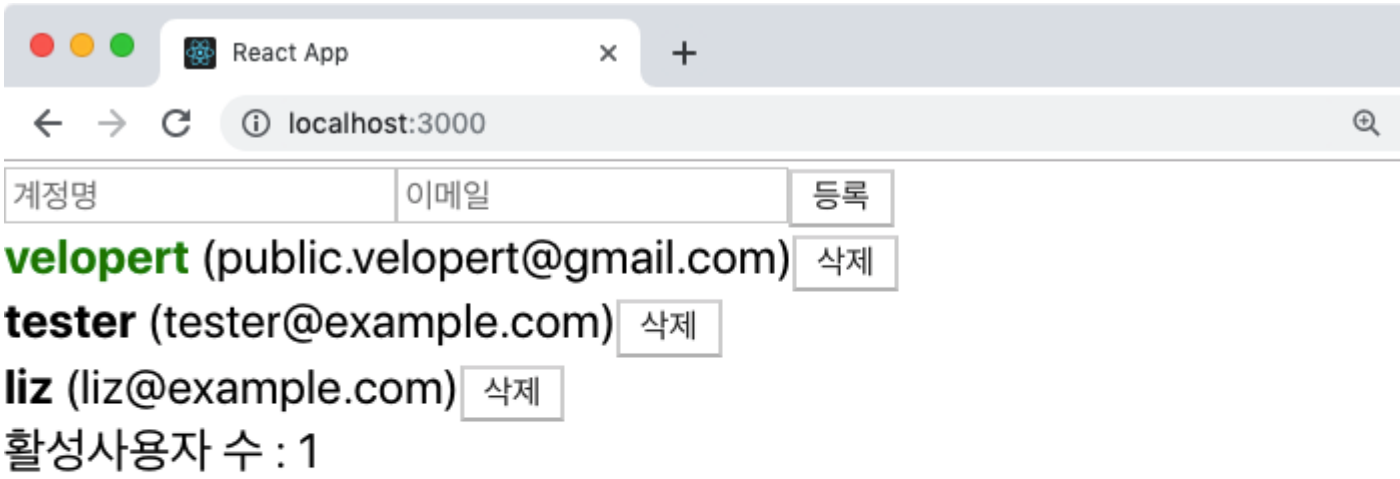
사실, `useCallback` 은 `useMemo` 를 기반으로 만들어졌습니다. 다만, 함수를 위해서 사용 할 때 더욱 편하게 해준 것 뿐이지요. 이런식으로도 표현 할 수 있습니다.

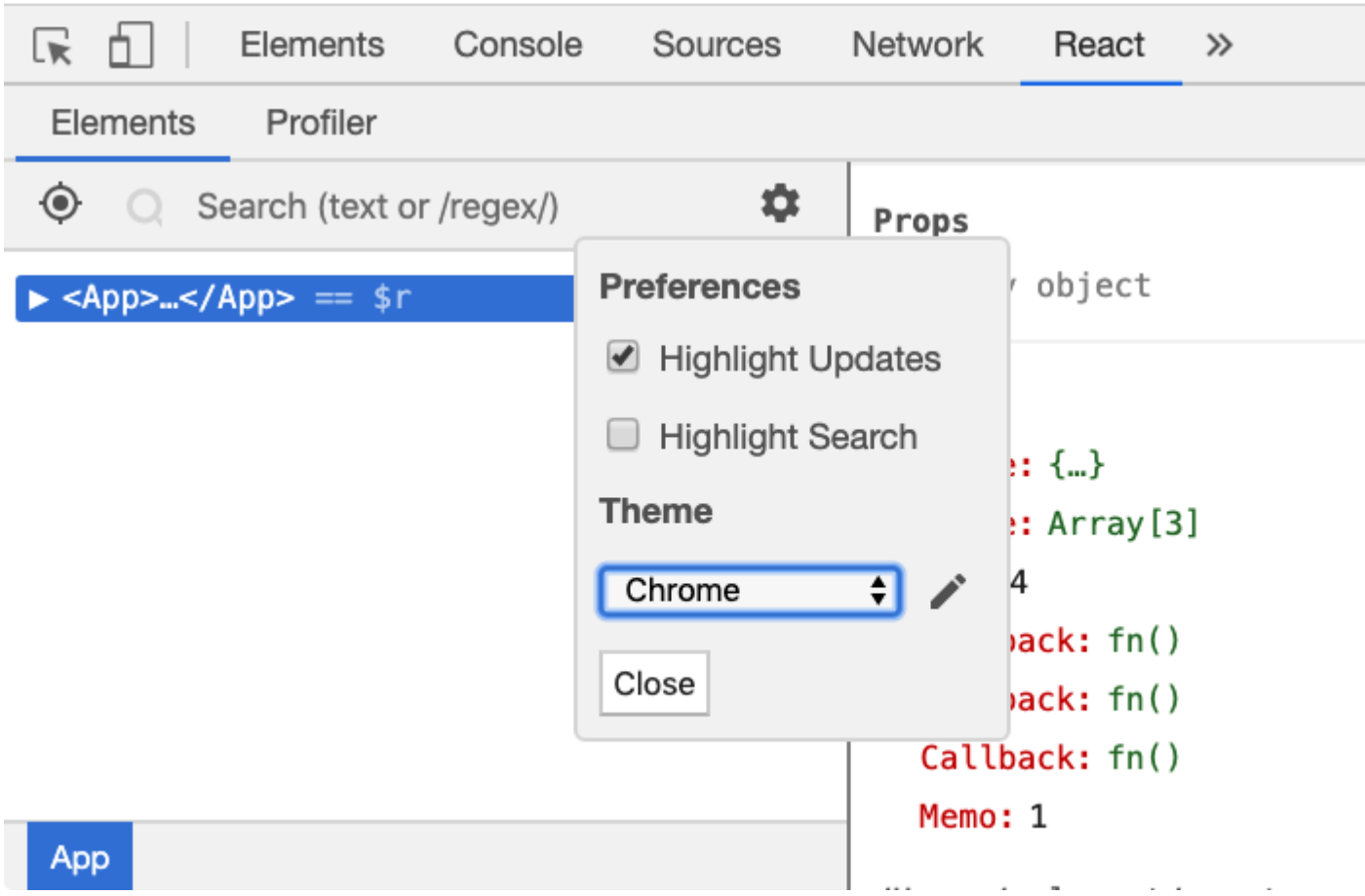
```
const onToggle = useMemo(
  () => () => {
    /* ... */
  },
  [users]
);
```

`useCallback` 을 사용 함으로써, 바로 이뤄낼수 있는 눈에 띄는 최적화는 없습니다. 다음 영상에서, 컴포넌트 렌더링 최적화 작업을 해주어야만 성능이 최적화되는데요, 그 전에, 어떤 컴포넌트가 렌더링되고 있는지 확인하기 위해서 React DevTools 라는 것을 소개드리겠습니다.

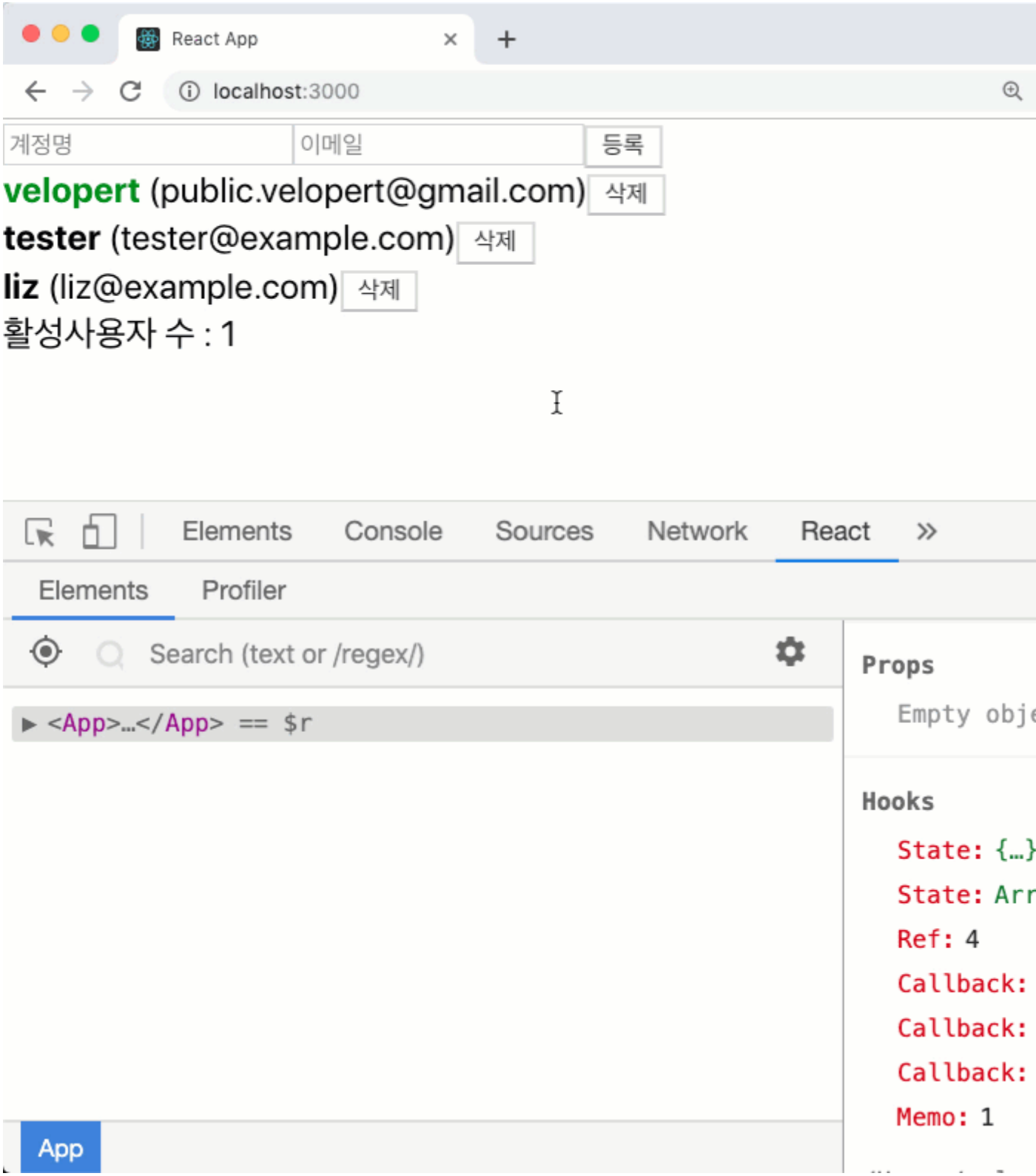
우선, 구글에 React DevTools 를 검색해서 크롬 웹스토어에 들어간뒤, 크롬 확장 프로그램을 설치해주세요. 링크

설치를 하고 나면 다음과 같이 React 탭이 개발자 도구에 뜹니다. 톱니바퀴 아이콘을 누르고, 'Highlight Updates' 를 체크해주세요.



이 속성을 키면 다음과 같이 리렌더링 되는 컴포넌트에 사각형 형태로 하이라이트되어 보여지게 됩니다.




지금 보면, input 이 바뀔 때에도 UserList 컴포넌트가 리렌더링이 되고 있지요?