19. React.memo 를 사용한 컴포넌트 리렌더링 방지

이번에는, 컴포넌트의 props 가 바뀌지 않았다면, 리렌더링을 방지하여 컴포넌트의 리렌더링 성능 최적화를 해줄 수 있는 React.memo 라는 함수에 대해서 알아보겠습니다.

이 함수를 사용한다면, 컴포넌트에서 리렌더링이 필요한 상황에서만 리렌더링을 하도록 설정해줄수있어요.

사용법은 굉장히 쉽습니다.

그냥, 감싸주시면 돼요.

우선 CreateUser 부터 적용을 해주겠습니다.

CreateUser.js

```
import React from 'react';

const CreateUser = ({ username, email, onChange, onCreate }) => {
  return (
    <div>
      <input
        name="username"
        placeholder="계정명"
        onChange={onChange}
        value={username}
      />
      <input
        name="email"
        placeholder="이메일"
        onChange={onChange}
        value={email}
      />
      <button onClick={onCreate}>등록</button>
    </div>
  );
};

export default React.memo(CreateUser);
```

참 쉽죠?

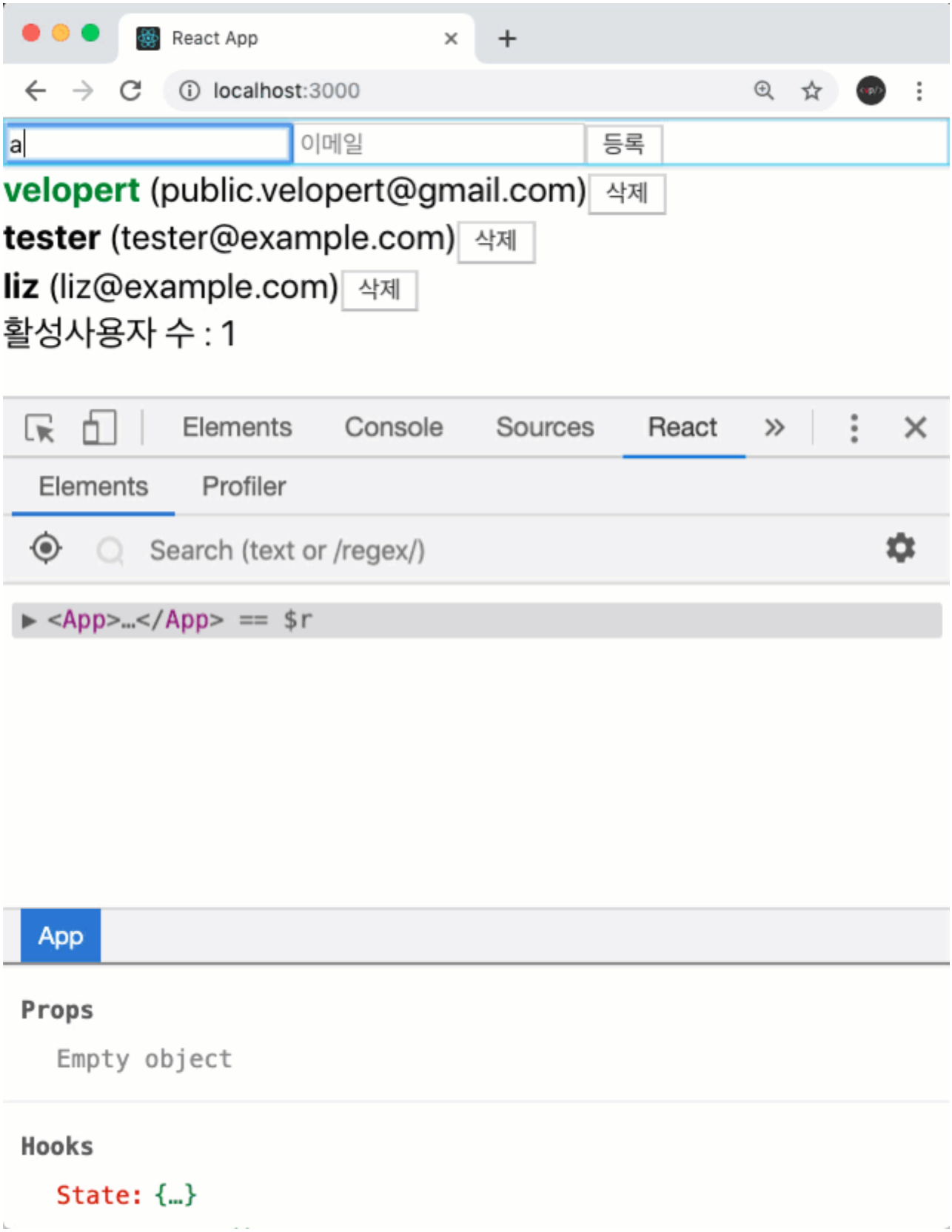
UserList 와 User 컴포넌트도 적용을 해줄게요.

UserList.js

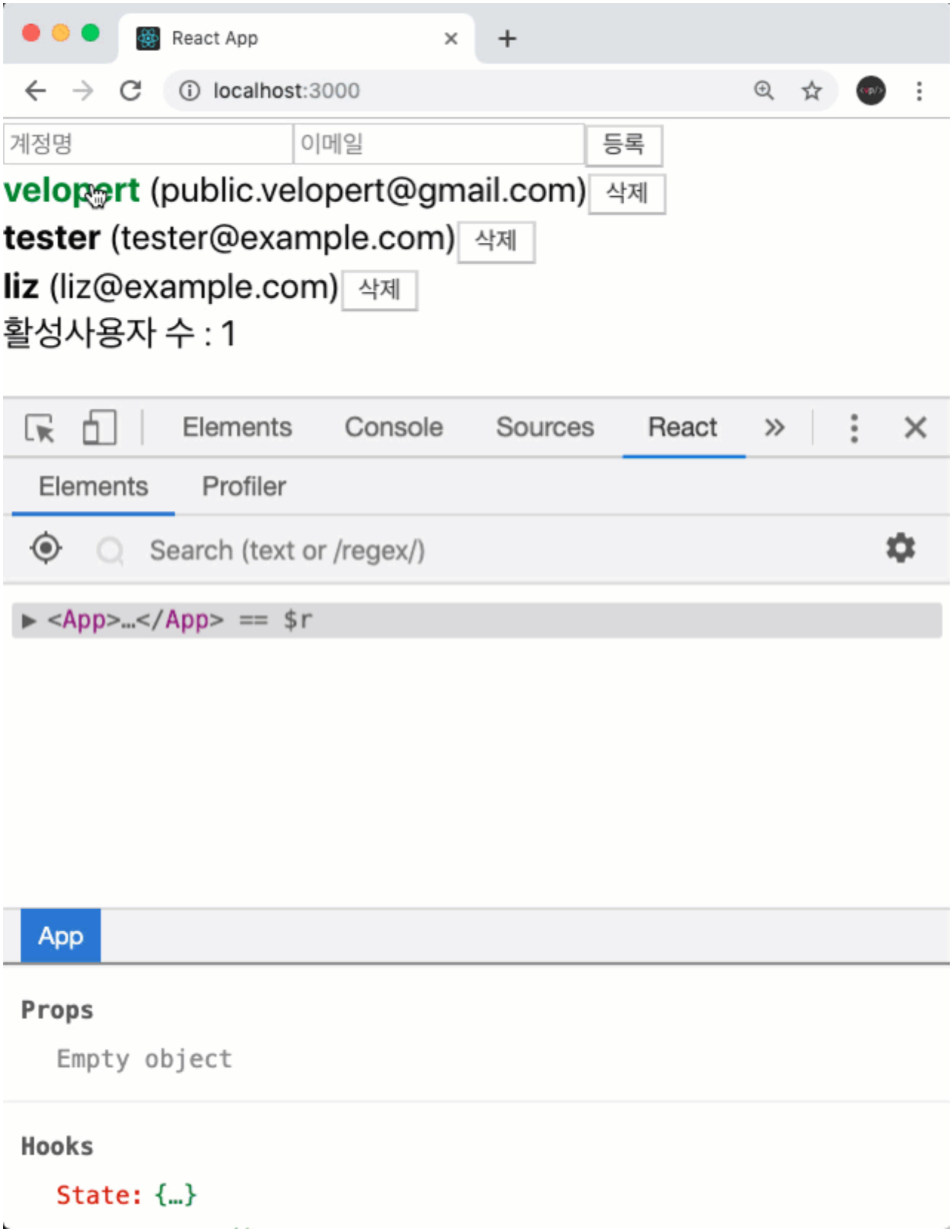
```
import React from 'react';

const User = React.memo(function User({ user, onRemove, onToggle }) {
  return (
```

```
    <div>
      <b
        style={{
          cursor: 'pointer',
          color: user.active ? 'green' : 'black'
        }}
        onClick={() => onToggle(user.id)}
      >
        {user.username}
      </b>
       
      <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 React.memo(UserList);
```

적용을 다 하고 나서, input 을 수정 할 때 하단의 UserList 가 리렌더링이 되지 않는것을 확인해주세요.



그런데, User 중 하나라도 수정하면 모든 User 들이 리렌더링되고, CreateUser 도 리렌더링이 됩니다.



왜 그런걸까요? 이유는 간단합니다. users 배열이 바뀔때마다 onCreate 도 새로 만들어지고, onToggle,onRemove 도 새로 만들어지기 때문입니다.

```
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]
);
```

`deps` 에 `users` 가 들어있기 때문에 배열이 바뀔때마다 함수가 새로 만들어지는건, 당연합니다.
그렇다면! 이걸 최적화하고 싶다면 어떻게해야 할까요?

바로 deps 에서 `users` 를 지우고, 함수들에서 현재 `useState` 로 관리하는 `users` 를 참조하지 않게 하는것입니다. 그건 또 어떻게 할까요? 힌트는, `useState` 를 배울때 다뤘던 내용이에요.
정답은 바로, 함수형 업데이트입니다.

함수형 업데이트를 하게 되면, `setUsers` 에 등록하는 콜백함수의 파라미터에서 최신 `users` 를 참조 할 수 있기 때문에 `deps` 에 `users` 를 넣지 않아도 된답니다.
그럼 각 함수들을 업데이트 해주세요 (onChange 의 경우엔 함수형 업데이트를 해도 영향은 가지 않지만, 연습삼아 해주겠습니다).

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

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

```
  const onToggle = useCallback(id => {
    setUsers(users =>
      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;
```

이렇게 해주면, 특정 항목을 수정하게 될 때, 해당 항목만 리렌더링 될거예요.

리액트 개발자 도구의 버그인지, CreateUser 도 렌더링 되는것처럼 보이는데 실제로 console.log 찍어보시면 렌더링이 안되고 있는 것을 확인 할 수 있습니다.

그럼 최적화가 끝난겁니다!

리액트 개발을 하실 때, `useCallback`, `useMemo`, `React.memo` 는 컴포넌트의 성능을 실제로 개선할수있는 상황에서만 하세요.
예를 들어서, User 컴포넌트에 `b` 와 `button` 에 `onClick` 으로 설정해준 함수들은, 해당 함수들을 `useCallback` 으로 재사용한다고 해서 리렌더링을 막을 수 있는것은 아니므로, 굳이 그렇게 할 필요 없습니다.
추가적으로, 렌더링 최적화 하지 않을 컴포넌트에 React.memo 를 사용하는것은, 불필요한 props 비교만 하는 것이기 때문에 실제로 렌더링을 방지할수있는 상황이 있는 경우에만 사용하시길바랍니다.

추가적으로, React.memo 에서 두번째 파라미터에 `propsAreEqual` 이라는 함수를 사용하여 특정 값들만 비교를 하는 것도 가능합니다.

```
export default React.memo(
  UserList,
  (prevProps, nextProps) => prevProps.users === nextProps.users
);
```

하지만, 이걸 잘못사용한다면 오히려 의도치 않은 버그들이 발생하기 쉽습니다.
예를 들어서, 함수형 업데이트로 전환을 안했는데 이렇게 users 만 비교를 하게 된다면, onToggle 과 onRemove 에서 최신 users 배열을 참조하지 않으므로 심각한 오류가 발생 할 수 있습니다.

지금까지 우리가 구현한 내용은 다음 CodeSandbox 에서 확인 할 수 있습니다.