React

◆ Hooks - 2

정수아

Contents

01 useReducer()

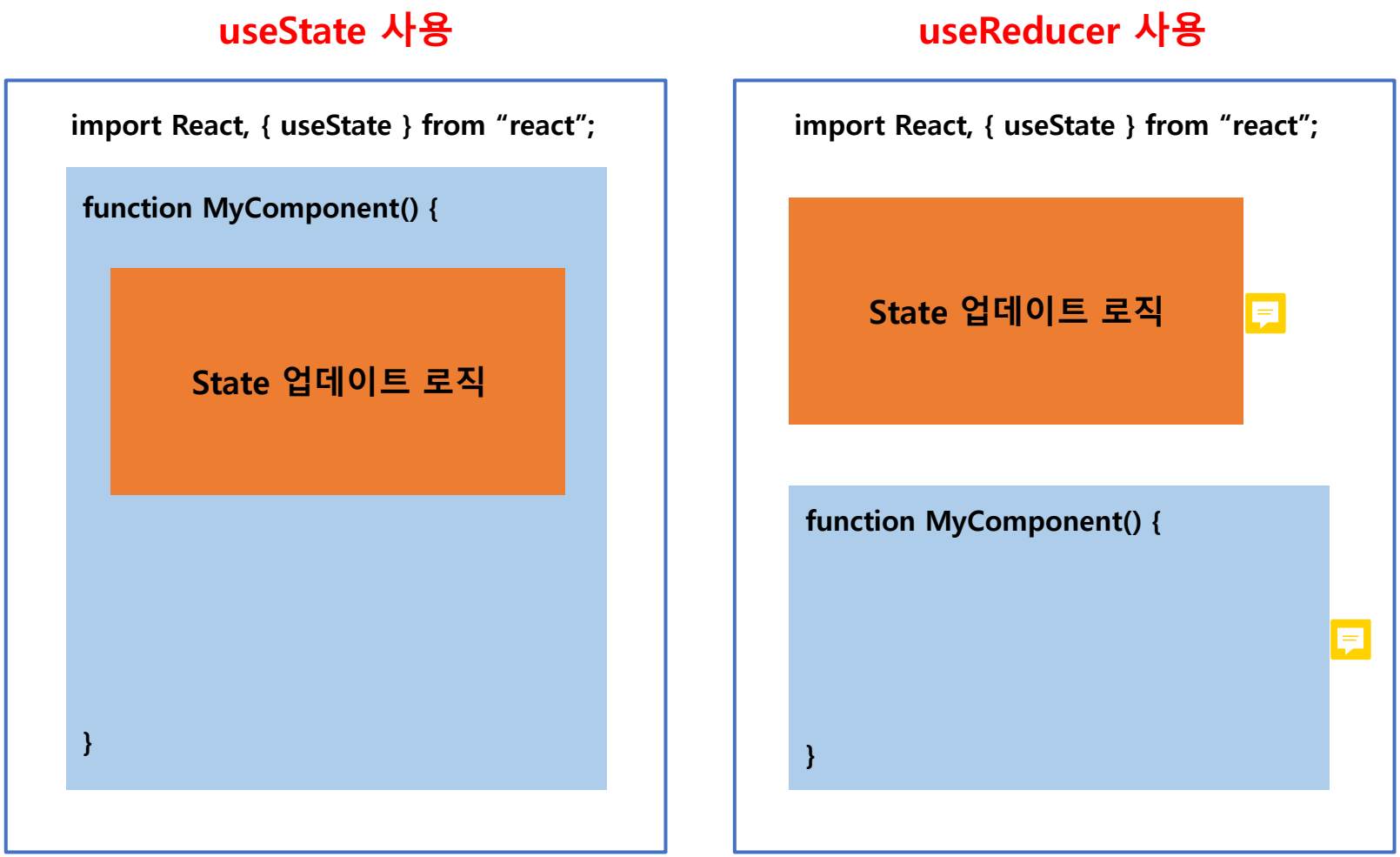
useReducer()

❖ **useReducer()**

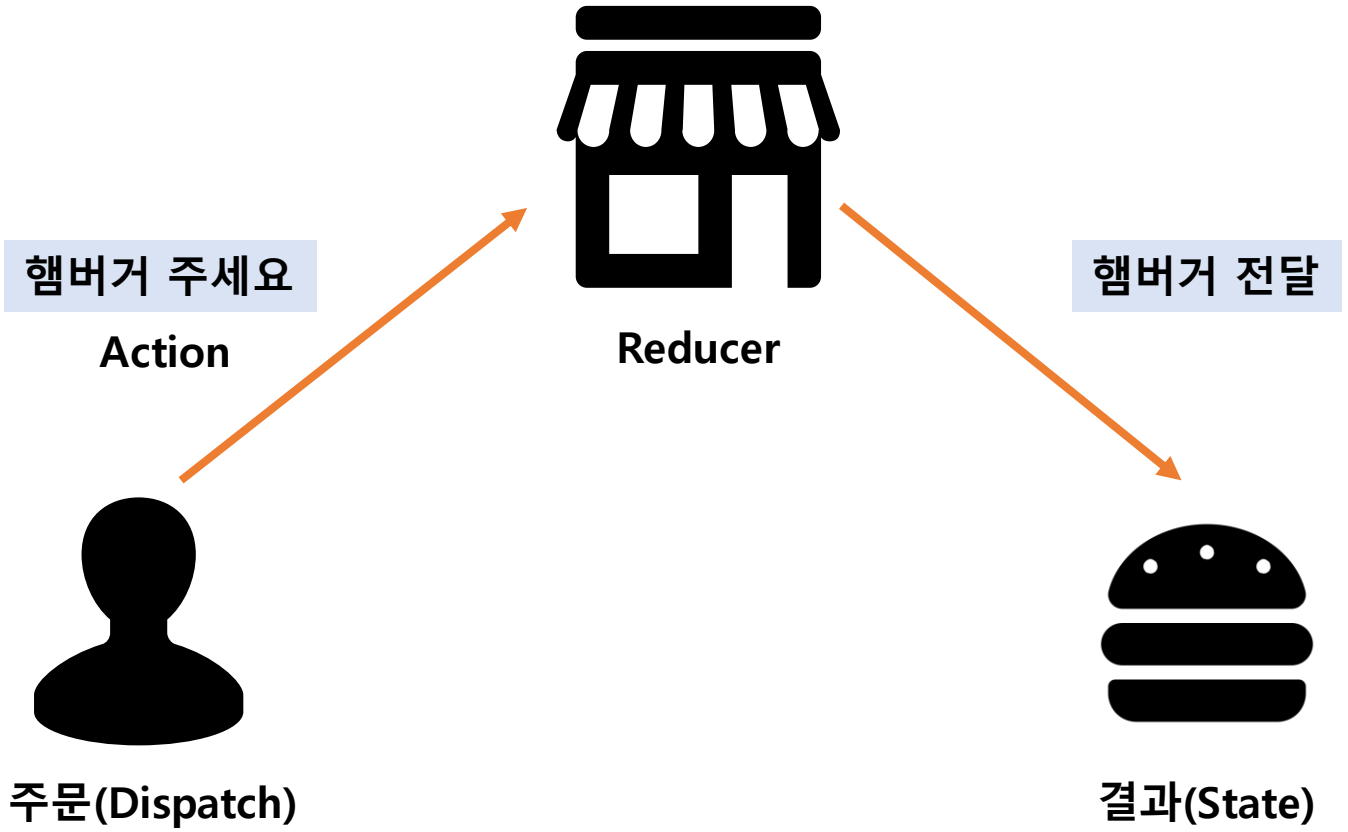
- 컴포넌트의 상태를 관리할 때 사용하는 Hooks

- 특징
  - 컴포넌트 상태 업데이트 로직을 컴포넌트에서 분리 가능
  - 다른 컴포넌트에서도 해당 로직을 재사용 가능

useReducer()

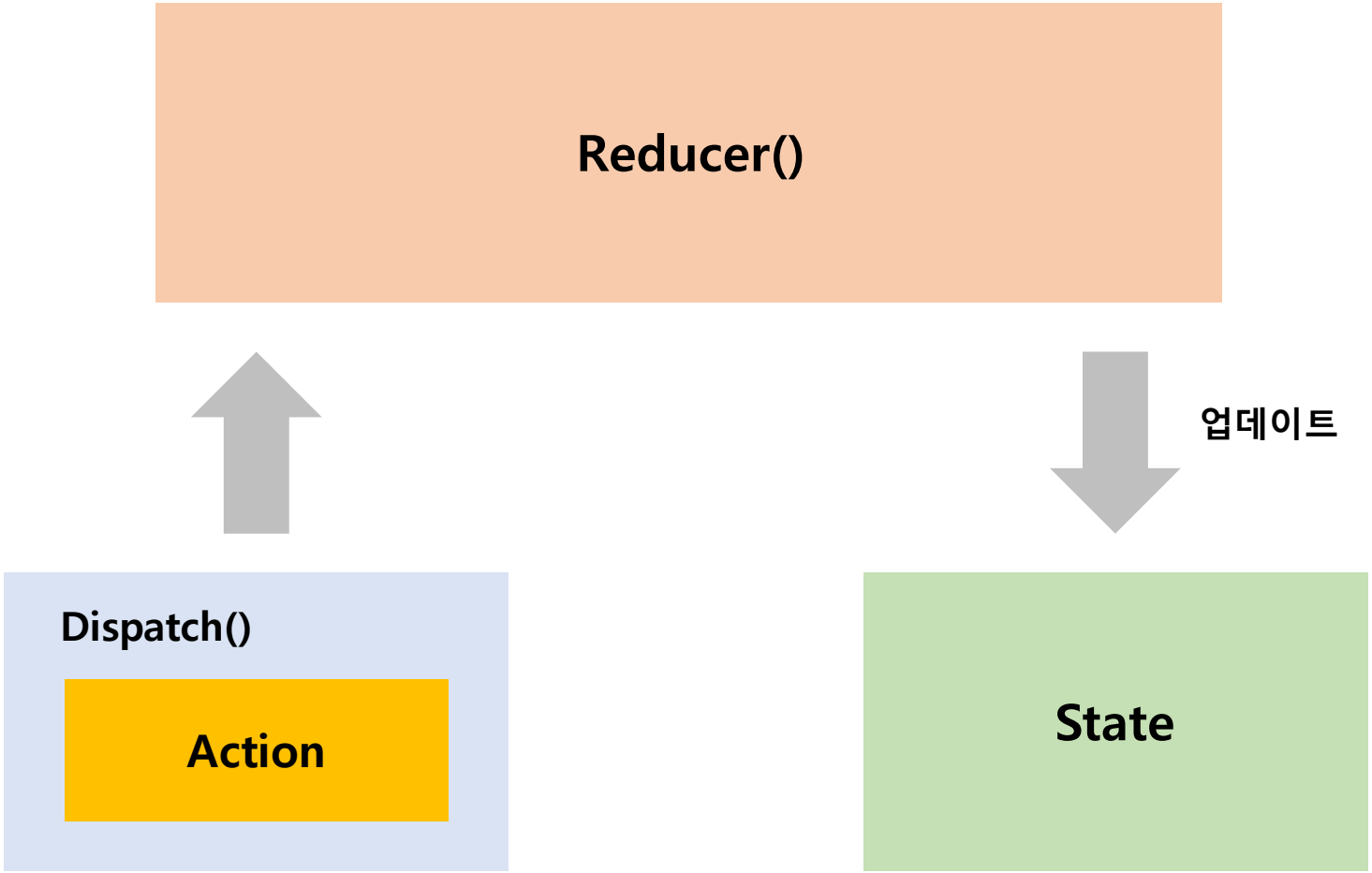

useState 사용
import React, { useState } from “react”;
function MyComponent() {
State 업데이트 로직
}
MyComponent.js
useReducer 사용
import React, { useState } from “react”;
State 업데이트 로직
function MyComponent() {
}
MyComponent.js

useReducer()



useReducer()

❖ 동작 과정



useReducer()

❖ useReducer() 사용 방법

```
const [state, dispatch] = useReducer(reducer, initialState);
```

값	설명
state	컴포넌트에서 사용할 상태 값
dispatch	액션을 발생시키는 함수
reducer	reducer 함수
initialState	초기 상태 값

❖ dispatch() 사용 방법

```
dispatch({ key : value })
```

useReducer()

❖ **reducer()**

- 현재 상태(state)와 action 객체(업데이트를 위한 정보)를 인자로 받아와서 새로운 상태를 반환해주는 함수

```
function reducer(state, action) {
   // 새로운 상태를 만드는 로직
   return 새로운 상태;
}
```

[실습] Counter Reducer 생성하기

❖ **countReducer.js**

```
function countReducer(state, action) {
   switch (action.type) {
      case 'INCREMENT':
         return state + 1;
      case 'DECREMENT':
         return state - 1;
      default:
         return state;
  }
}

export default countReducer;
```

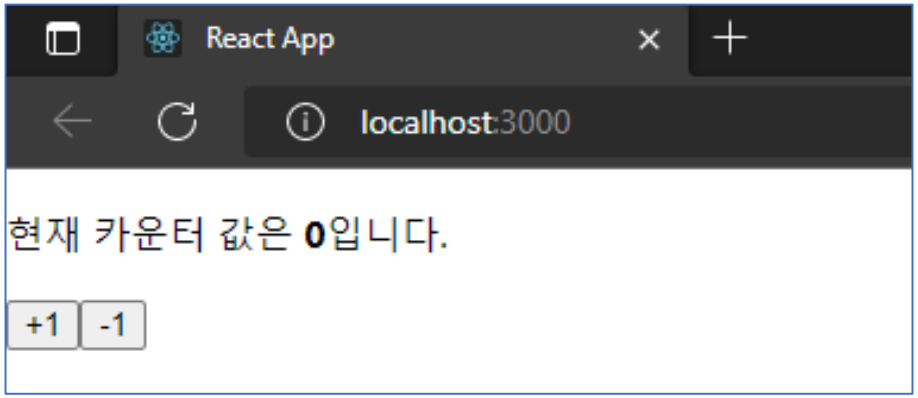
[실습] Counter Reducer 생성하기

❖ Counter.js

```
import React, { useReducer } from "react";
import countReducer from "./countReducer";

const Counter = () => {
  const [state, dispatch] = useReducer(countReducer, 0);

  function numUp() {
     dispatch({ type: "INCREMENT" });
  }

  function numDown() {
     dispatch({ type: "DECREMENT" });
  }

  return (
    <div>
      <p>
        현재 카운터 값은 <b>{state}</b>입니다.
      </p>
      <button onClick={numUp}>+1</button>
      <button onClick={numDown}>-1</button>
    </div>
  );
};
export default Counter;
```

[실습] Counter Reducer 생성하기

❖ 실행 결과



[실습] dispatch()에 값 여러 개 넣기

❖ Counter.js

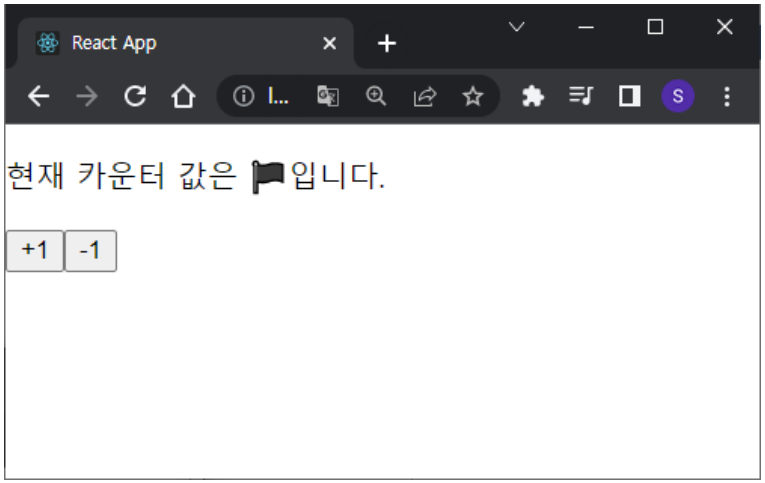
```
function numUp() {
     dispatch({ type: "INCREMENT", icon : "🚩" });
}

function numDown() {
    setValue({ type: "DECREMENT", icon: "🏴‍☠️" });
}
```

❖ countReducer.js

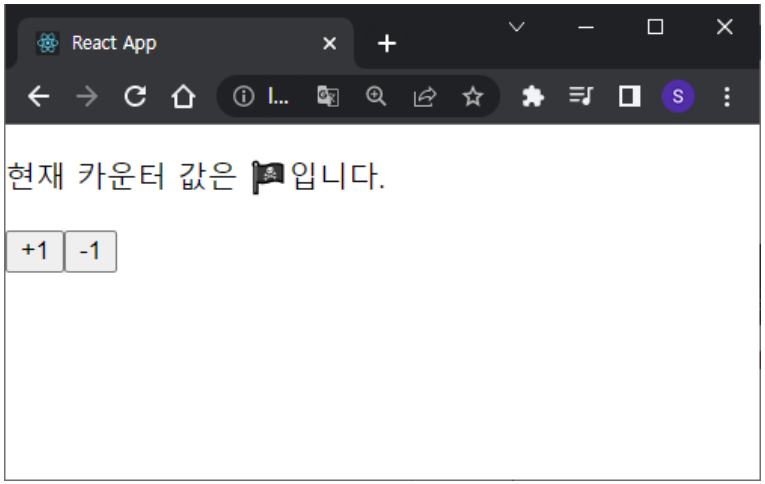
```
function countReducer(state, action) {
   switch (action.type) {
      case 'INCREMENT':
         return action.icon;
      case 'DECREMENT':
         return action.icon;
      default:
         return state;
  }
}
```

[실습] dispatch()에 값 여러 개 넣기

❖ 실행 결과



+1 버튼 클릭



-1 버튼 클릭

useReducer() vs useState()

❖ **useState()**
- 컴포넌트에서 관리하는 값이 한 개
- 값이 단순한 숫자, 문자열, 불리언 등의 값인 경우

❖ **useReducer()**
- 컴포넌트에서 관리하는 값이 여러 개
- 구조가 복잡한 경우

useReducer() vs useState()

❖ 다음과 같은 데이터는 어떤 상태관리 함수가 좋을까?

```
{
    name: "sooa",
    age: 20,
    job: "teacher",
    skill: [
              { HTML: "상", JavaScript: "중" },
              { Python: "상", Java: "하" }
    ]
};
```

02

useMemo

useMemo()란?

❖ **useMemo()**

- 컴포넌트 최적화를 위해 사용되는 Hook
- 동일한 계산을 하는 함수를 포함하고 있는 컴포넌트가 반복적으로 렌더링이 될 때, 해당 함수의 값을 메모리에 저장해 놓고 재사용할 수 있게 함

❖ **메모이제이션(Memoization)**

- 동일한 계산을 반복해야 할 때, 이전에 계산한 값을 메모리에 저장함으로써 동일한 계산의 반복 수행을 제거
- 프로그램 실행 속도를 빠르게 하는 기술

useMemo()는 왜 사용하는가?

Component

함수() { return 결과 값 }

return 함수 결과 값

useMemo()는 왜 사용하는가?

```
function MyComponent({ a, b }) {
   const result = compute(a, b);

   return <div>{result}</div>;
}
```

- 만약 compute() 함수가 매우 복잡한 연산을 수행한다면?
- 따라서 결과 값을 리턴하는데 몇 초 이상 걸린다면?
  - 컴포넌트의 재렌더링이 필요할 때마다 compute() 함수가 호출
  - 사용자는 지속적으로 UI에서 지연이 발생하는 것을 경험

useMemo()

❖ useMemo() 구조

```
const memoizedValue = useMemo(
    () => {
            // 연산량이 많은 작업을 수행
            return compute(a, b)
        }, [a, b]);
```

- 첫 번째 매개변수
  - 콜백함수
  - 메모이제이션 할 값을 계산해서 반환해주는 함수

- 두 번째 매개변수
  - 의존성 배열
  - 배열 안의 값이 업데이트 될 때만 콜백함수를 재호출

useMemo()

❖ 의존성 배열이 빈 배열인 경우

```
useMemo(() => {
    // 연산량이 많은 작업을 수행
   return compute(a, b)
}, []);
```

- 컴포넌트가 마운트되었을 때만 콜백함수 호출
- 이후에는 항상 같은 값을 가져와서 사용

❖ 의존성 배열이 없는 경우

```
useMemo(() => {
    // 연산량이 많은 작업을 수행
   return compute(a, b)
});
```

- 컴포넌트가 렌더링 될 때마다 콜백함수 호출
- useMemo()를 사용하는 의미가 없음

useMemo()

❖ **useMemo()를 적용하여 컴포넌트 수정**

```
function MyComponent({ a, b }) {
    const result = useMemo(() => compute(a, b), [a, b]);
    return <div>{result}</div>;
}
```

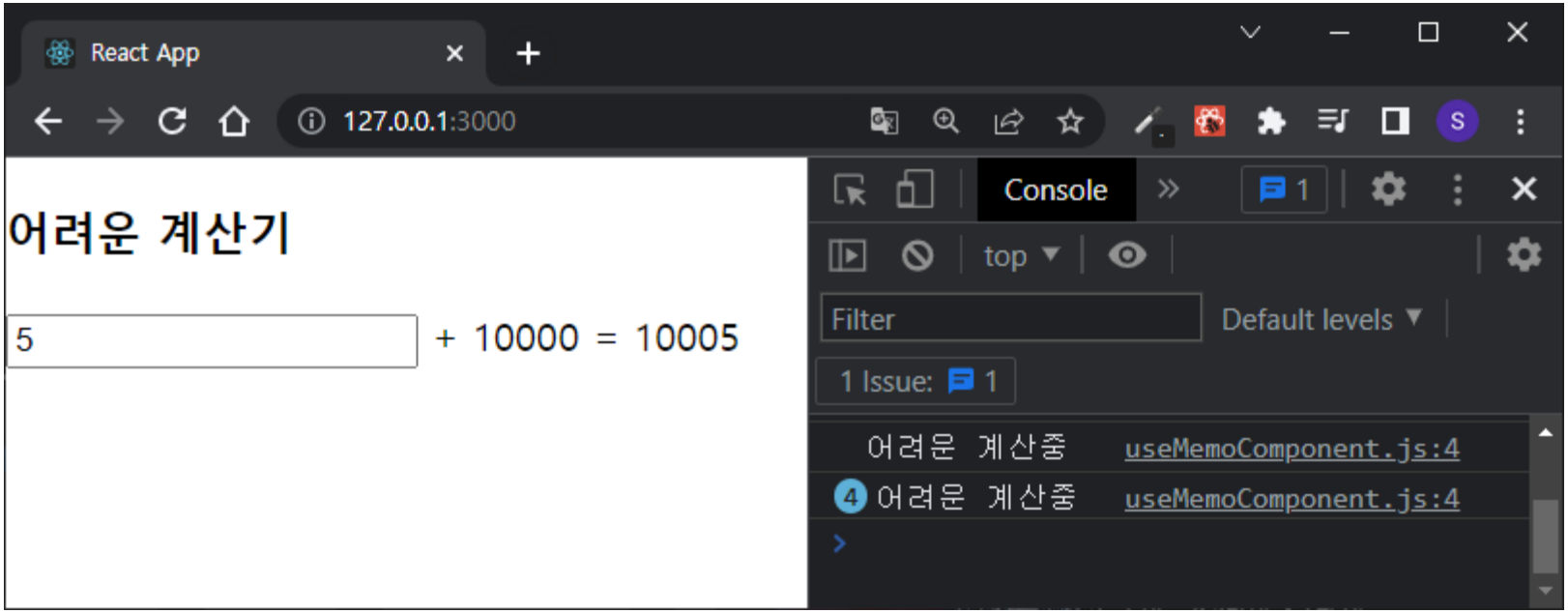
[실습] useMemo()를 사용한 계산

❖ **useMemoComponent.js**

```
function hardCalculate(number) {
   console.log("어려운 계산중");
   for (let i = 0; i < 100000000; i++) {}

   return number + 10000;
}

const useMemoComponent = () => {
  const [hardNumber, setHardNumber] = useState(1);

  const hardSum = hardCalculate(hardNumber);

  return (
    <div>
      <h3>어려운 계산기</h3>
      <input type="number" value={hardNumber}
             onChange={(e) => setHardNumber(parseInt(e.target.value))}
      <span> + 10000 = {hardSum} </span>
    </div>
  );
};
```

[실습] useMemo()를 사용한 계산

❖ 실행 결과



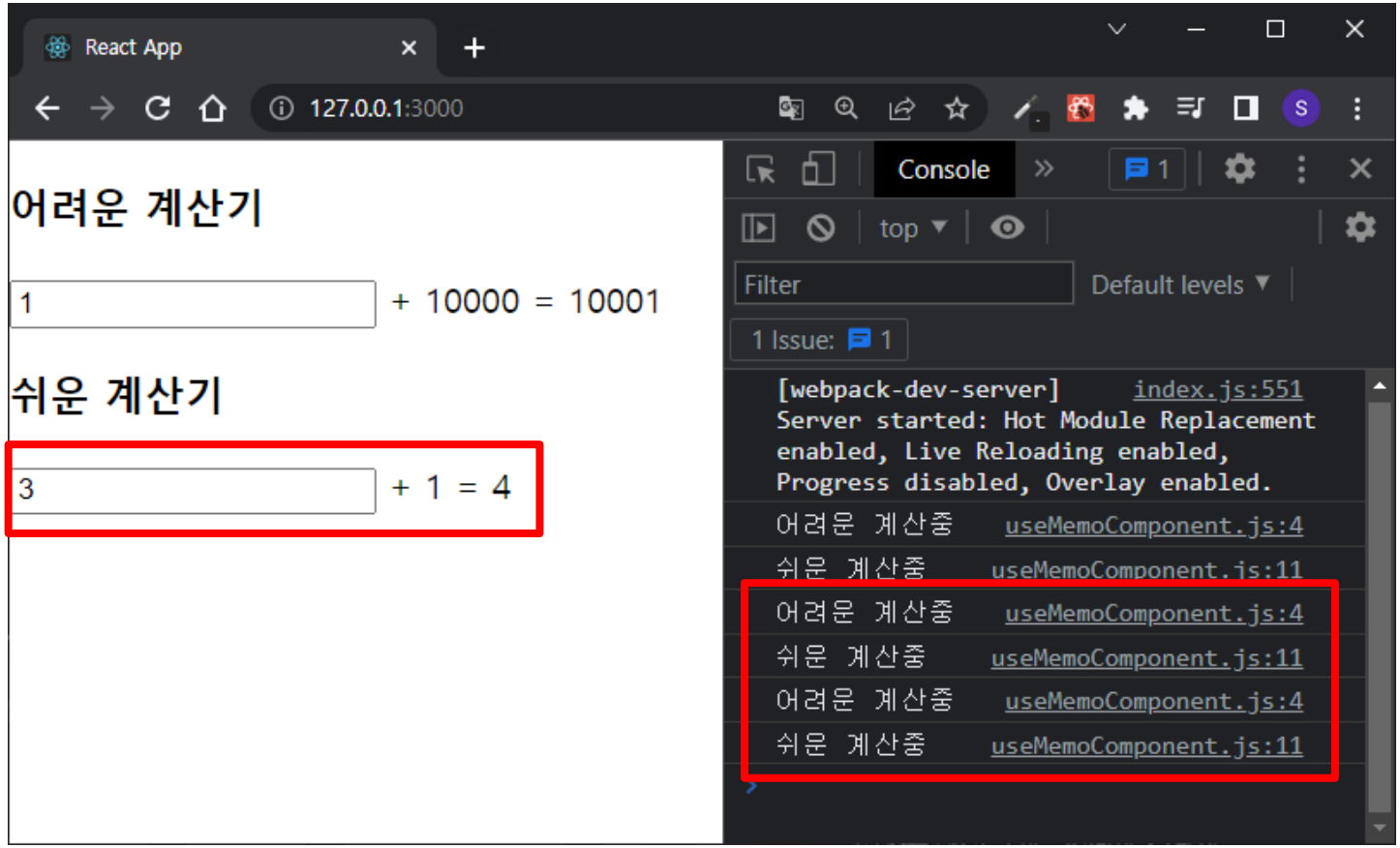
[실습] useMemo()를 사용한 계산

❖ **useMemoComponent.js**

```
function hardCalculate(number) {  // 생략  }

function easyCalculate(number) {
  console.log("쉬운 계산중");
  return number + 1;
}

const useMemoComponent = () => {
  const [hardNumber, setHardNumber] = useState(1);
  const [easyNumber, setEasyNumber] = useState(1);

  const hardSum = hardCalculate(hardNumber);
  const easySum = easyCalculate(easyNumber);
  return (
    <div>
      <h3>어려운 계산기</h3>
      // 생략
      <h3>쉬운 계산기</h3>
      <input type="number" value={easyNumber}
             onChange={(e) => setEasyNumber(parseInt(e.target.value))}/>
      <span> + 1 = {easySum} </span>
    </div>
  );
};
```

[실습] useMemo()를 사용한 계산

❖ 실행 결과



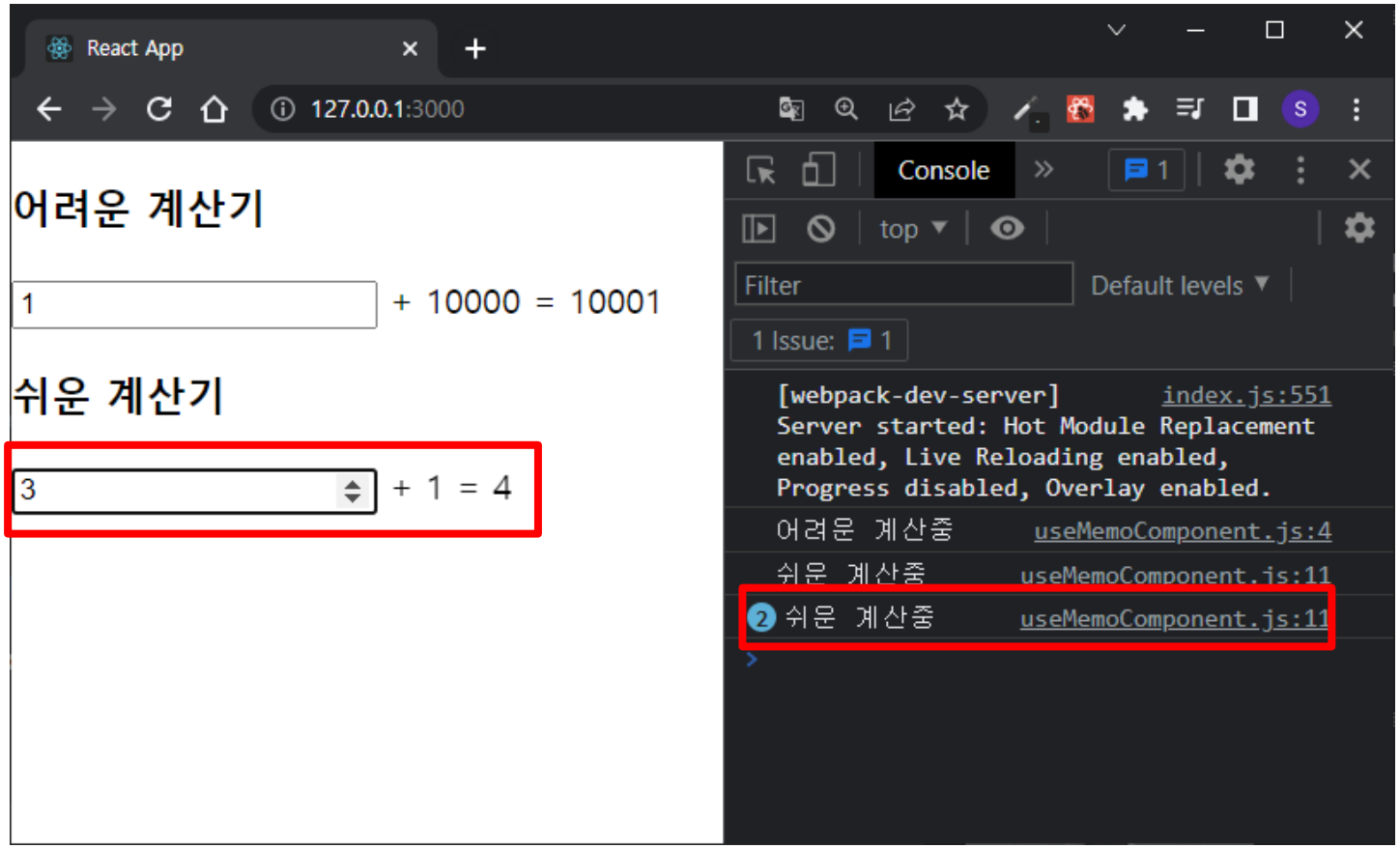
[실습] useMemo()를 사용한 계산

❖ **useMemo()를 사용하여 수정**

```
const hardSum = useMemo(() => hardCalculate(hardNumber),
                        [hardNumber]
                       );
```

[실습] useMemo()를 사용한 계산

❖ 실행 결과



useMemo()

❖ **useMemo 사용 시 주의점**

- 목적
  - 값을 재사용하기 위해 별도의 메모리를 할당하여 값을 저장
- 불필요한 값까지 메모이제이션하면 메모리 용량이 늘어나 성능 저하 발생

03 useCallback

useCallBack()

❖ useCallBack()

- 이미 생성해 놓은 함수를 재사용할 수 있게 해주는 Hook
- useMemo()와 유사한 Hook

- 컴포넌트에 useCallBack()을 사용하지 않고 함수를 정의한 경우
  - 렌더링이 발생할 때마다 함수가 새로 정의됨

```
const MyComponent = () => {
  function myFunction() {
    console.log("함수 생성 완료")
  }
}
```

useCallBack()

❖ useCallBack() 구조

```
const memoizedCallback = useCallBack(
                                () => doSomething(a, b),
                                [a, b]
                            );
```

- 첫 번째 매개변수
  - 콜백함수

- 두 번째 매개변수
  - 의존성 배열
  - 배열 안의 값이 변경되면 함수를 새로 생성

useCallBack()

❖ 의존성 배열이 빈 배열인 경우

```
useCallBack(() => {
                    return doSomething(a, b)
            }, []);
```

- 컴포넌트가 마운트되었을 때만 함수를 새로 생성
- 이후에는 항상 같은 함수를 재사용

❖ 의존성 배열이 없는 경우

```
useCallBack(() => {
                    return doSomething(a, b)
            });
```

- 컴포넌트가 렌더링 될 때마다 함수를 새로 생성
- useCallBack()를 사용하는 의미가 없음

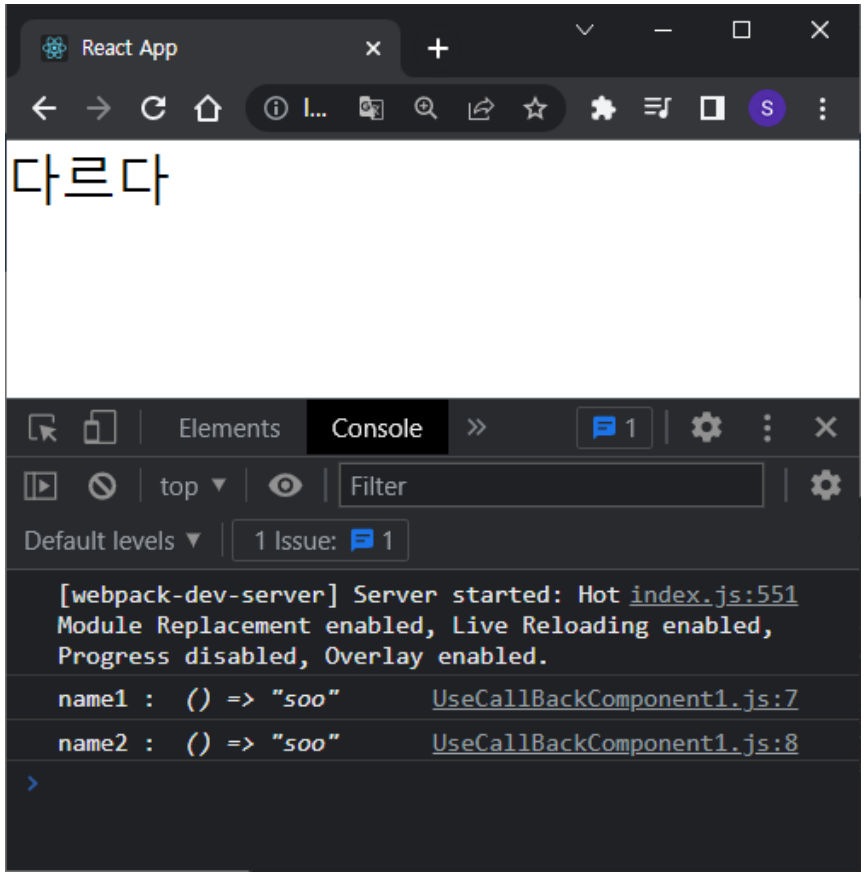
[실습] 자바스크립트 함수 동등성

❖ UseCallBackComponent1.js

```
const UseCallBackComponent1 = () => {
  const name1 = () => "soo";
  const name2 = () => "soo";

  console.log("name1 : ", name1);
  console.log("name2 : ", name2);

  return <div>{name1 === name2 ? "같다" : "다르다"}</div>;
};

export default UseCallBackComponent1;
```

[실습] 자바스크립트 함수 동등성

❖ 실행 결과



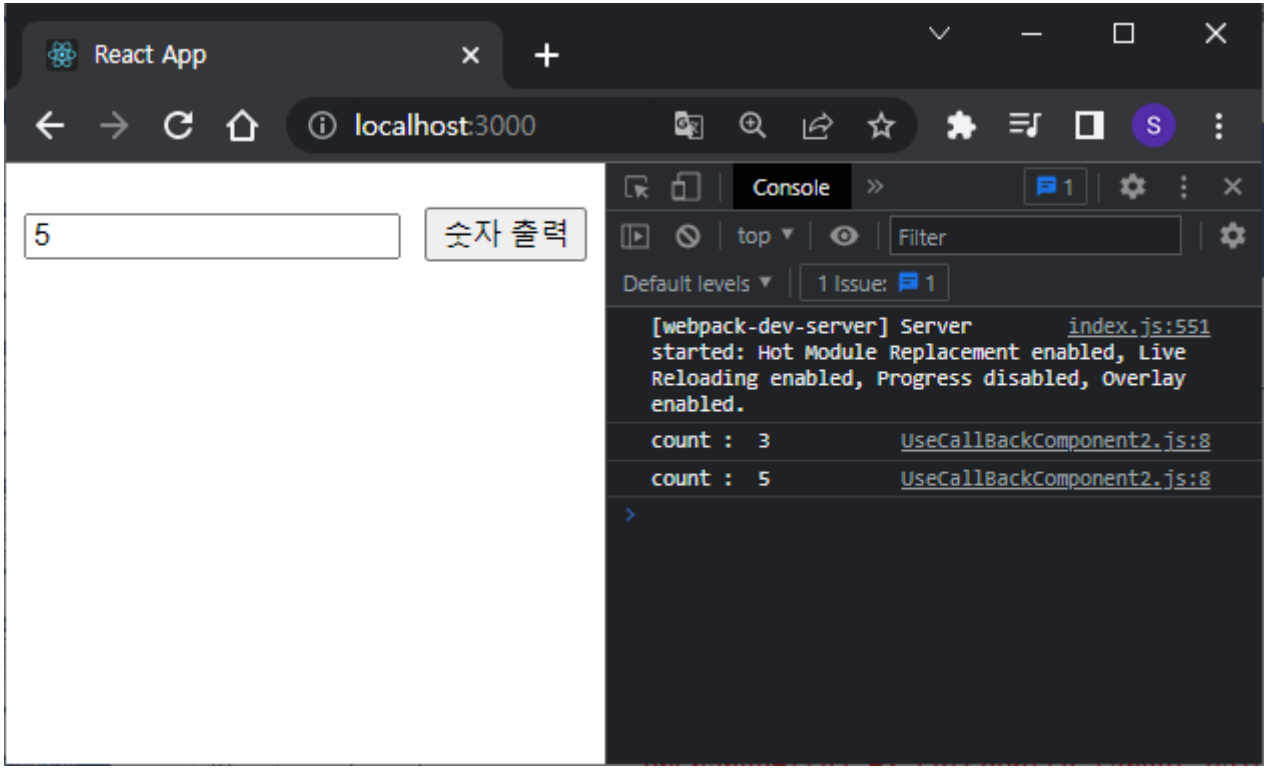
[실습] useCallback()

❖ UseCallBackComponent2.js

```
const UseCallBackComponent2 = () => {
  const [count, setCount] = useState(0);

  const clickHandler = () => {
     console.log("count : ", count);
  };

  return (
    <div>
      <input type="number" value={count}
             onChange={(e) => setCount(e.target.value)} />
      <button onClick={clickHandler}>숫자 출력</button>
    </div>
  );
};

export default UseCallBackComponent2;
```

[실습] useCallback()

❖ 실행 결과



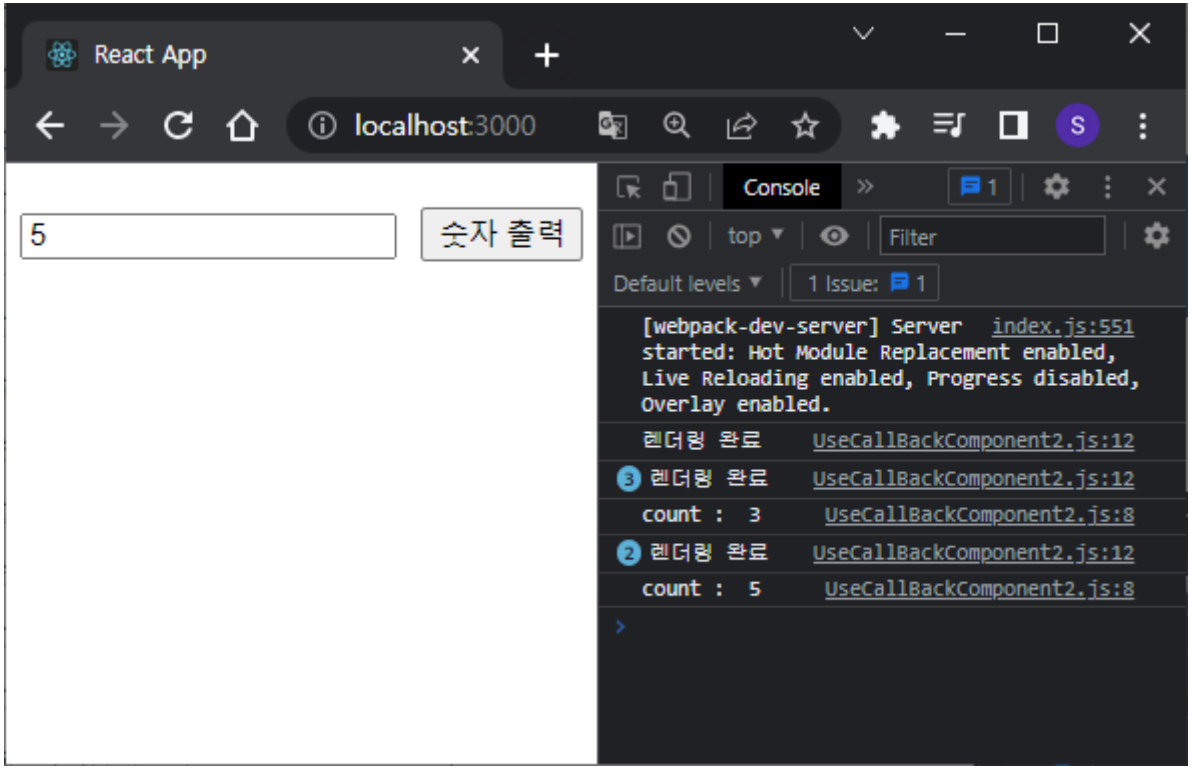
[실습] useCallback() - useEffect() 추가

❖ **UseCallBackComponent2.js**

```
const UseCallBackComponent2 = () => {
  const [count, setCount] = useState(0);

  const clickHandler = () => {
     console.log("count : ", count);
  };

  useEffect(() => {
     console.log("렌더링 완료");
  });

  return (
    <div>
      <input type="number" value={count}
             onChange={(e) => setCount(e.target.value)} />
      <button onClick={clickHandler}>숫자 출력</button>
    </div>
  );
};

export default UseCallBackComponent2;
```

[실습] useCallback() - useEffect() 추가

❖ 실행 결과



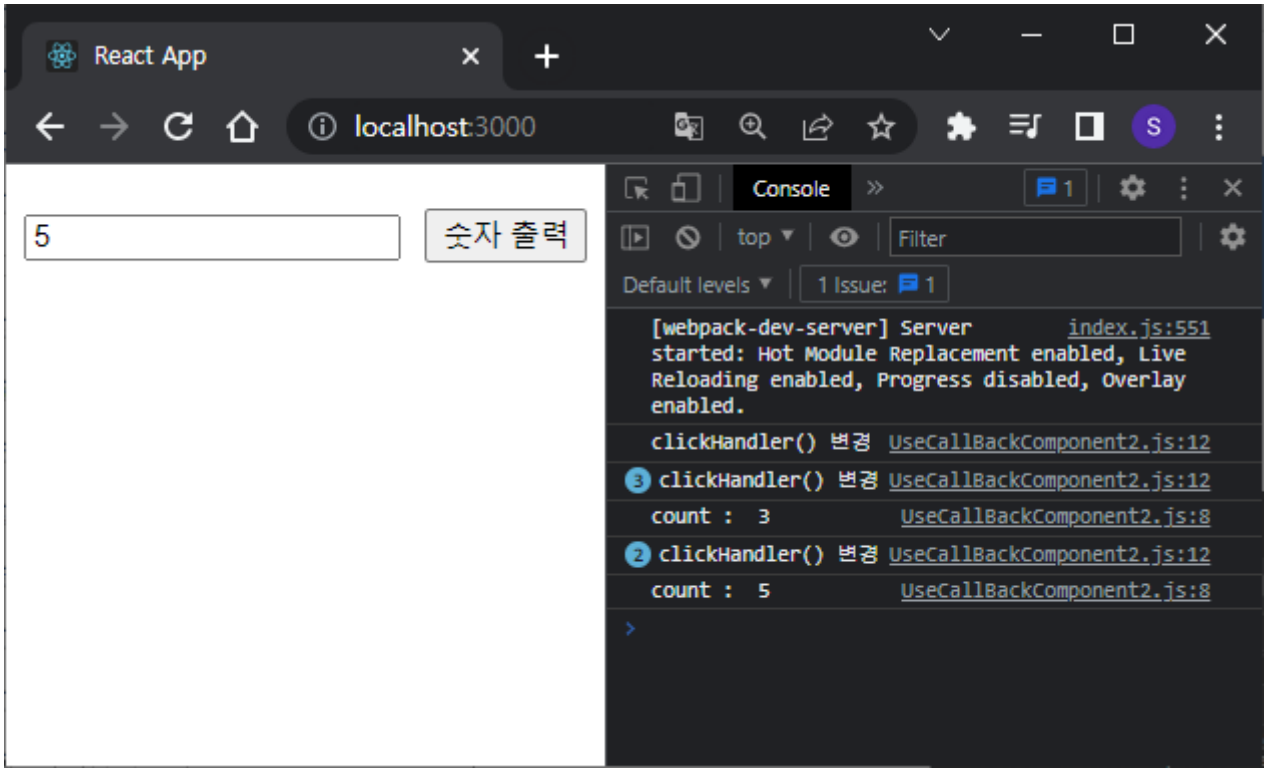
[실습] useCallback() - useEffect() 추가

❖ **UseCallBackComponent2.js**

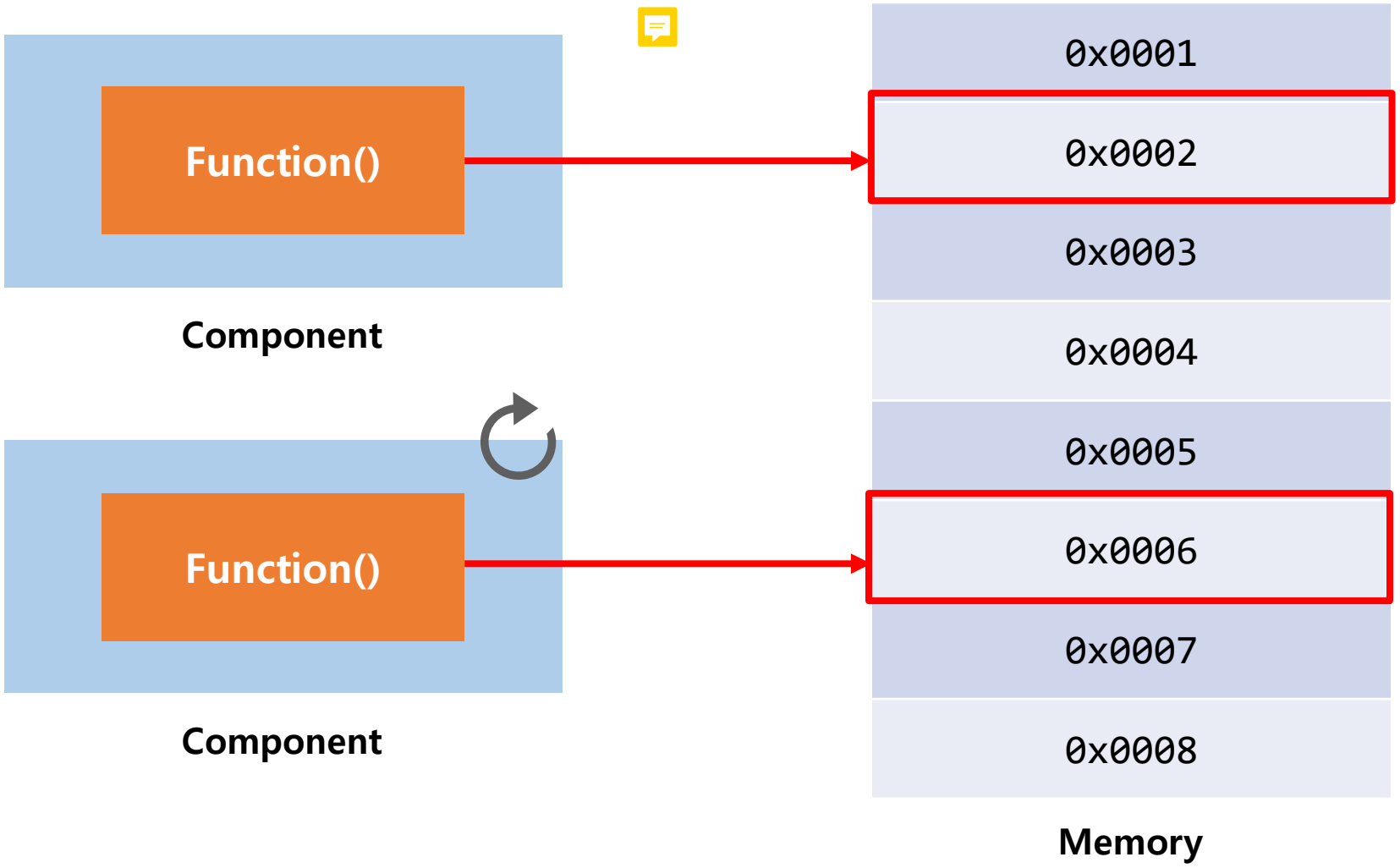
```
const UseCallBackComponent2 = () => {
  const [count, setCount] = useState(0);

  const clickHandler = () => {
     console.log("count : ", count);
  };

  useEffect(() => {
     console.log("clickHandler() 변경");
  }, [clickHandler]);

  return (
    <div>
      <input type="number" value={count}
             onChange={(e) => setCount(e.target.value)} />
      <button onClick={clickHandler}>숫자 출력</button>
    </div>
  );
};

export default UseCallBackComponent2;
```

[실습] useCallback() - useEffect() 추가

❖ 실행 결과



[실습] useCallback() - useEffect() 추가



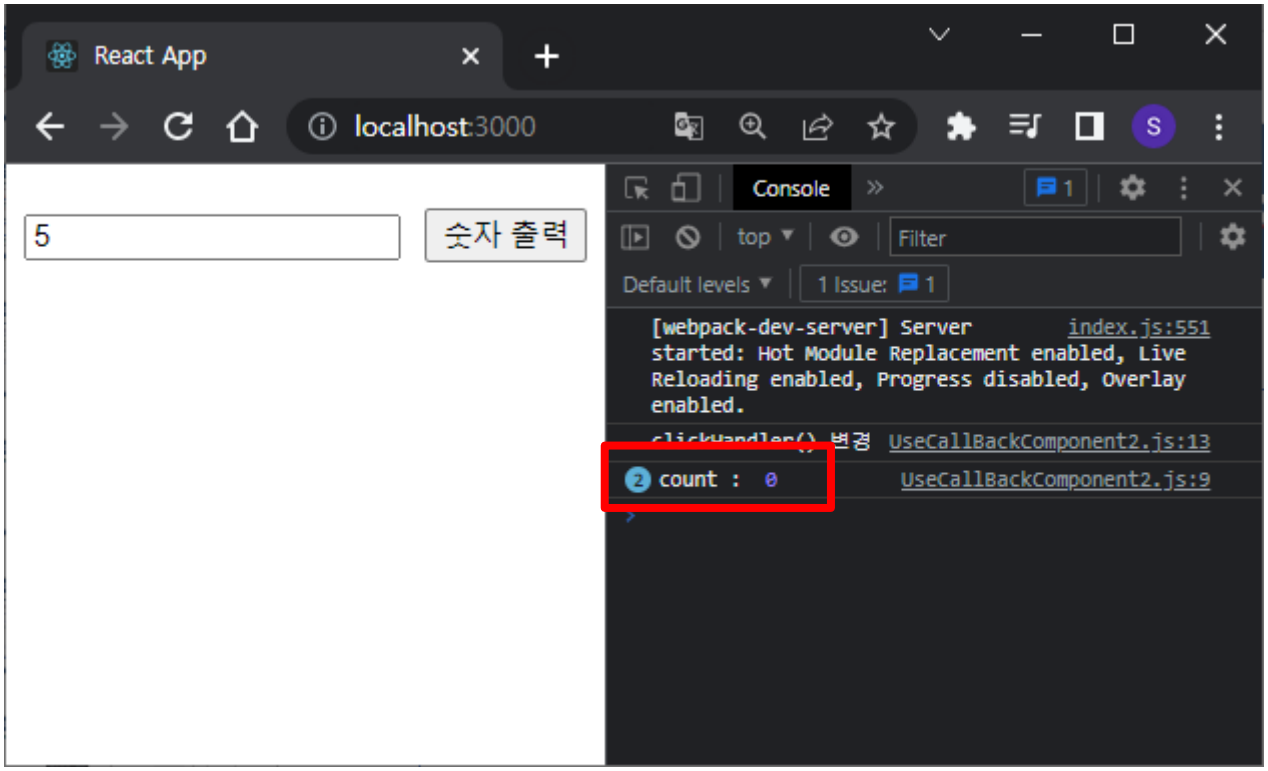
[실습] useCallback() - useCallback() 추가

❖ UseCallBackComponent2.js

```
const UseCallBackComponent2 = () => {
  const [count, setCount] = useState(0);

  const clickHandler = useCallback(() => {
    console.log("count : ", count)
  }, []);

  useEffect(() => {
     console.log("clickHandler() 변경");
  }, [clickHandler]);

  // 생략
};

export default UseCallBackComponent2;
```

[실습] useCallback() - useCallback() 추가

❖ 실행 결과



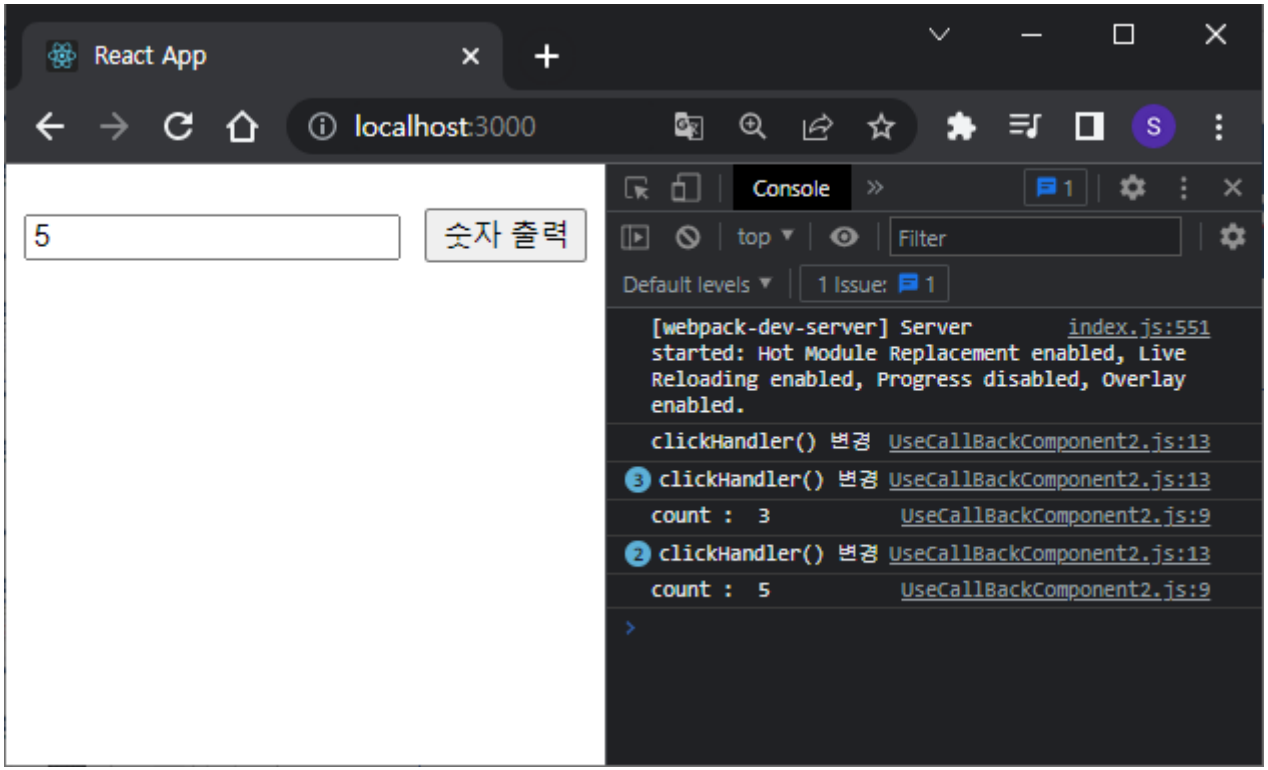
[실습] useCallback() - useCallback() 최종

❖ UseCallBackComponent2.js

```
const UseCallBackComponent2 = () => {
  const [count, setCount] = useState(0);

  const clickHandler = useCallback(() => {
    console.log("count : ", count)
  }, [count]);

  useEffect(() => {
     console.log("clickHandler() 변경");
  }, [clickHandler]);

  // 생략
};

export default UseCallBackComponent2;
```

[실습] useCallback() - useCallback() 최종

❖ 실행 결과



[실습] useCallback() + React.memo()

❖ App.js

```
function App() {
  const [count, setCount] = useState(0);

  const updateHandler = () => {
     console.log("update");
  };

  return (
    <div>
      <input type="number"
             onChange={(e) => setCount(e.target.value)} />

      <ChildComponent update={updateHandler} />
    </div>
  );
}

export default App;
```

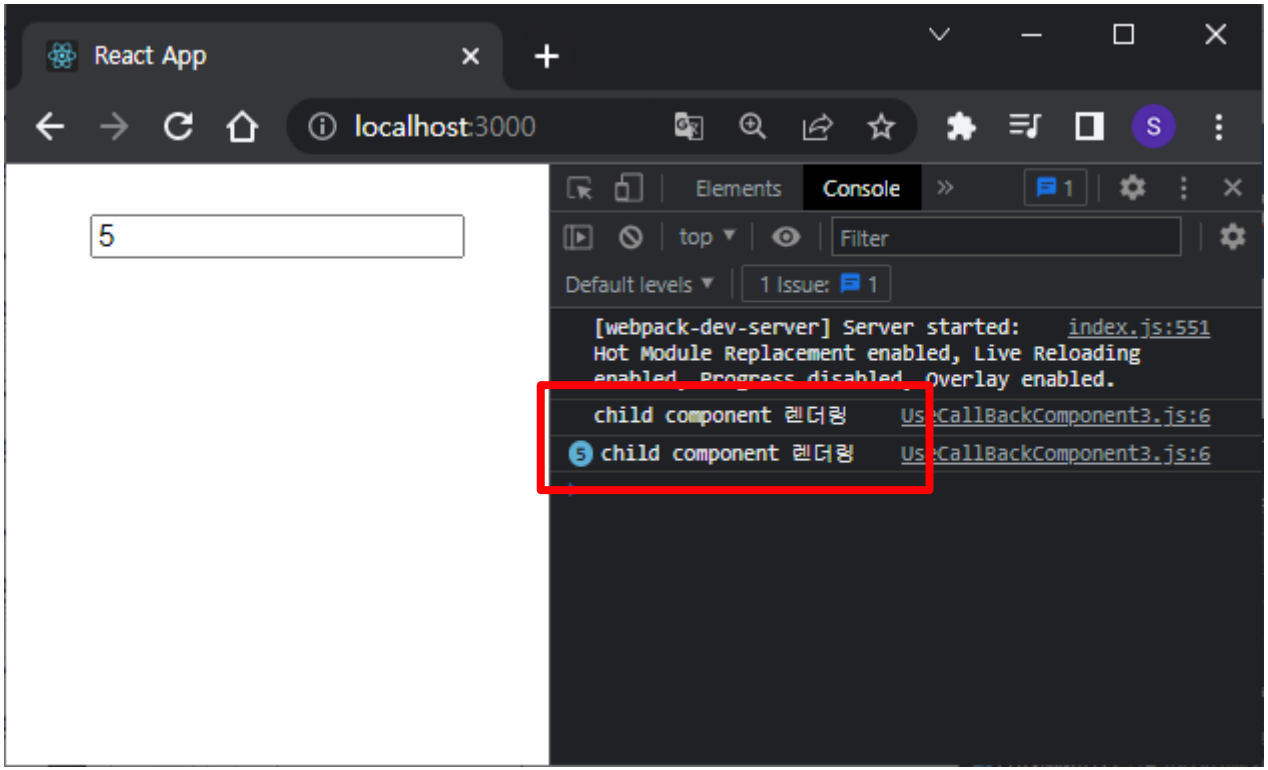
[실습] useCallback() + React.memo()

❖ **ChildComponent.js**

```
const ChildComponent = (props) => {
  const { update } = props;

  console.log("child component 렌더링");

  return <div></div>;
};

export default ChildComponent;
```

[실습] useCallback() + React.memo()

❖ 실행 결과



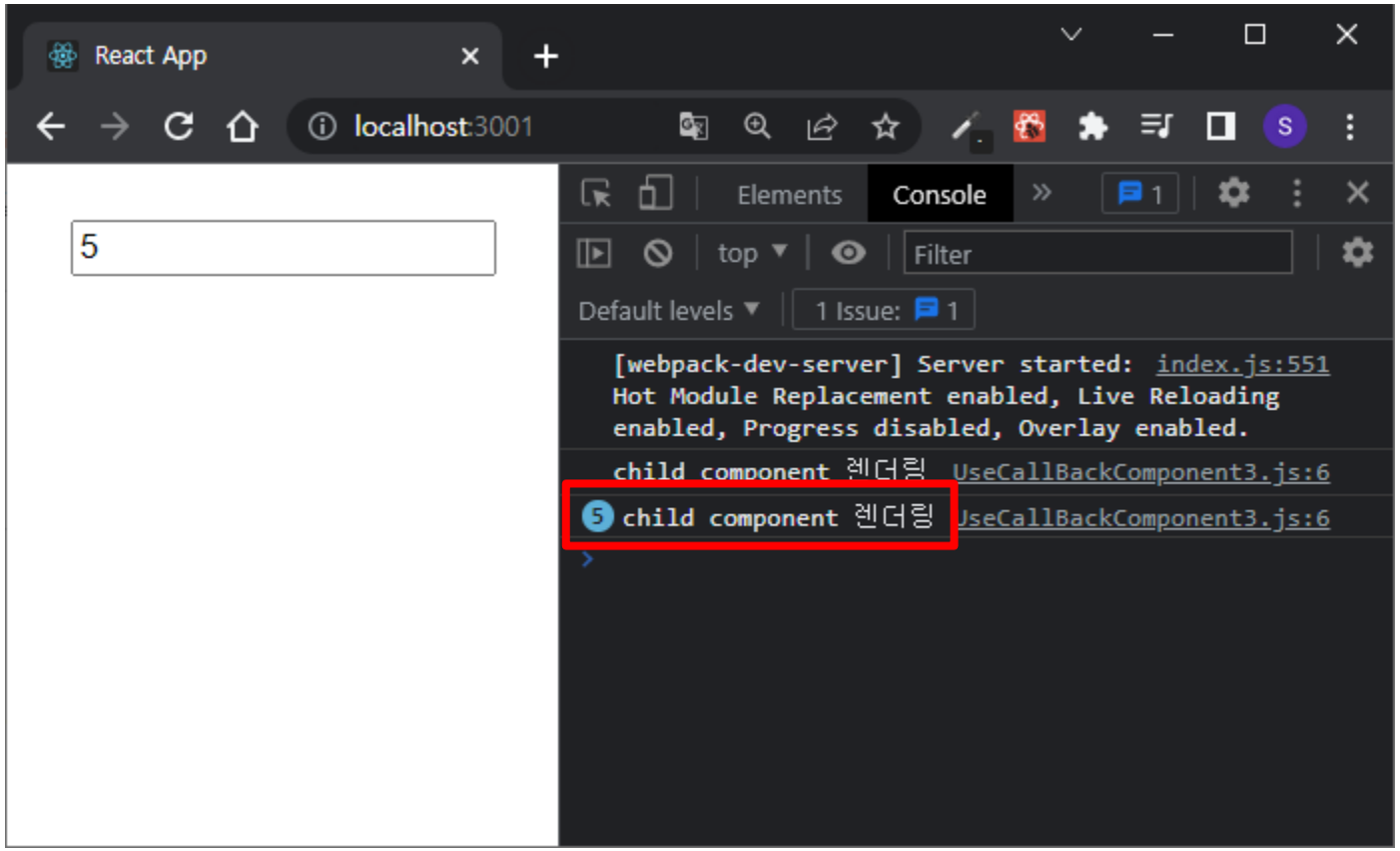
[실습] useCallback() + React.memo() - 수정

❖ App.js

```
function App() {
  const [count, setCount] = useState(0);

  const updateHandler = useCallback(() => {
     console.log("update");
  }, []);

  return (
    <div>
      <input type="number"
             onChange={(e) => setCount(e.target.value)} />

      <ChildComponent update={updateHandler} />
    </div>
  );
}

export default App;
```

[실습] useCallback() + React.memo() - 수정

❖ 실행 결과



[실습] useCallback() + React.memo() - 수정

❖ **useCallback()을 사용했지만 자식 컴포넌트가 호출 되는 이유**

- 자식 컴포넌트가 호출 될 때 매번 새로운 props 객체가 생성되기 때문

- ChildComponent.js

실행 될 때마다 새로운 props 객체가 생성

```
const ChildComponent = (props) => {
  const { update } = props;

  console.log("child component 렌더링");

  return <div></div>;
};

export default ChildComponent;
```

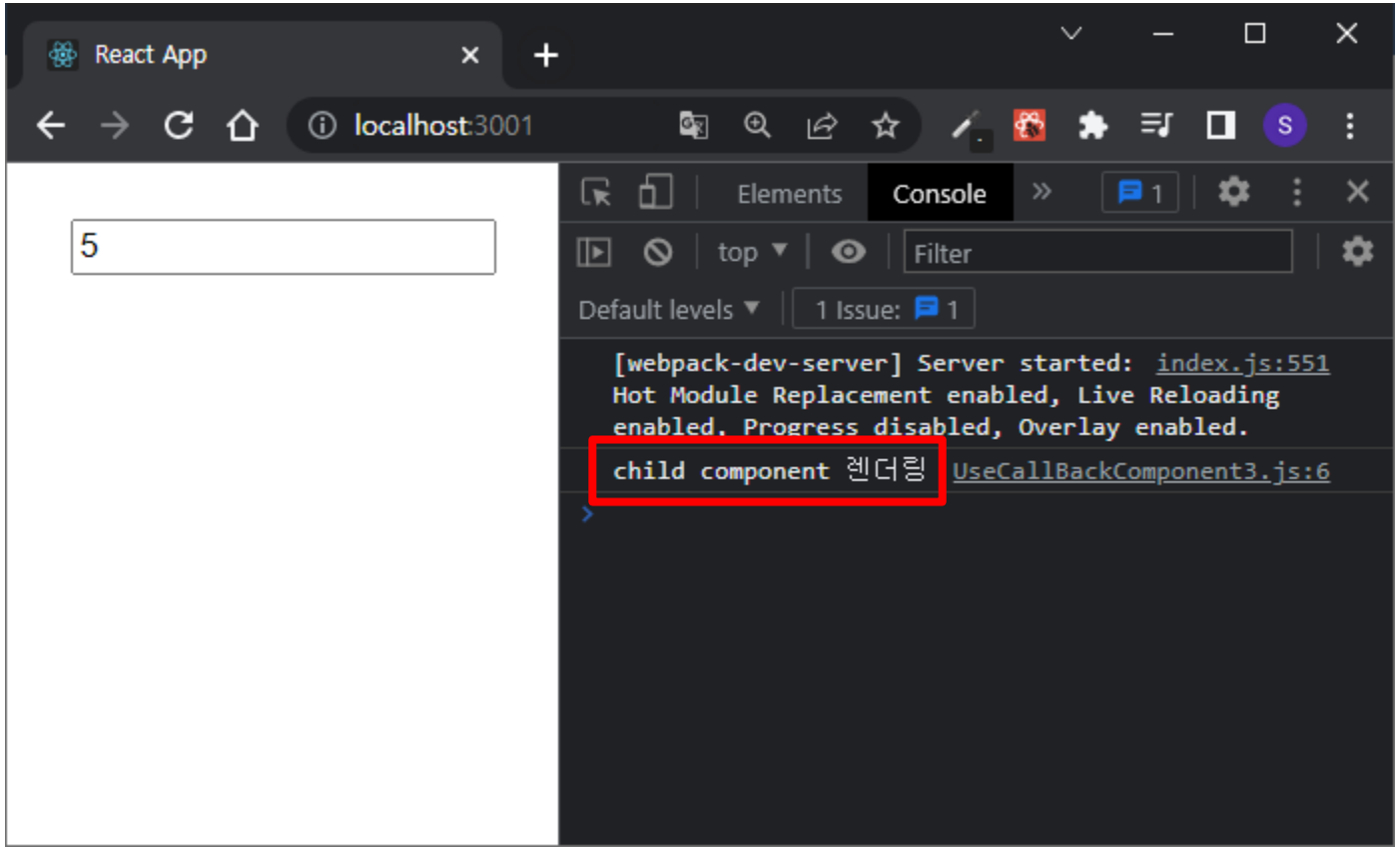
[실습] useCallback() + React.memo() - 최종

❖ ChildComponent.js

```
import React, { memo } from "react";

const ChildComponent = (props) => {
  const { update } = props;

  console.log("child component 렌더링");

  return <div></div>;
};

export default memo(ChildComponent);
```

[실습] useCallback() + React.memo() - 최종

❖ 실행 결과



React.memo()

❖ **React.memo()**
- 리액트에서 제공하는 고차 컴포넌트
  - 컴포넌트를 인자로 받아서 새로운 컴포넌트로 반환해줌
- props의 변화가 있는지를 체크
  - 변화가 있다면 렌더링 수행
  - 변화가 없다면 기존에 렌더링 된 내용을 재사용


THANK YOU