# React Hook 심화

# 1. React Hook 개요

## ❖ React Hook이란?

- 함수 컴포넌트에서 컴포넌트의 상태 관리, 생명 주기에 실행할 기능을 제공하는 함수 형태의 라이브러리
- 16.8~ 에서 지원

## ❖클래스 컴포넌트의 단점

- 다양한 생명주기 메서드를 사용할 수 있지만 작성된 코드가 간결하지 못함
- 특히 하나의 생명주기 메서드 내부에 서로 관련성 없는 코드가 작성될 우려가 있음
- 반대로 서로 관련있는 코드가 여러 생명주기 메서드에 분산되어 작성될 수 있음
- 예) componentDidMount + componentWillUnmount 를 이용한 리소스 연결-해제

## ❖함수 컴포넌트의 장점

- 코드가 간결하고 서로 관련된 코드를 묶어서 배치할 수 있음
- 하지만 클래스 컴포넌트를 완벽히 대체하지는 못함

# 1. React Hook 개요

❖react에서 제공되는 대표적인 Hook
- useState()
- useEffect(), useLayoutEffect()
- useMemo(), useCallback()
- useRef(), useContext()
- useReducer()

❖react-router에서 다뤘던 대표적인 Hook
- useParams(), useLocation(), useNavigate()
- useMatch(), useOutletContext(), useSearchParams()
- useLoaderData(), useAsyncValue(), useActionData()

❖이 과정에서는 Hook의 기본 사용법이 아닌 심화된 내용을 다룸
- useState 설명 생략

# 2. useEffect

## ❖용어 정의

- mount : 컴포넌트 트리에 컴포넌트가 추가되는 상황
- unmount : 기존 컴포넌트 트리에서 컴포넌트가 제거되는 상황
- update : mount된 컴포넌트의 State, Props가 변경되어 컴포넌트 UI가 변경되는 상황

## ❖useEffect Hook 이란?

- 컴포넌트가  mount, update, unmount 될 때의 생명주기와 관련된 부작용(Side Effect)를 수행할 수 있도록 하는 React Hook

## ❖useEffect 함수 형식

```
//첫번째 인자 : effectCallback - 부작용으로 실행할 함수
//두번째 인자 : depsList - 의존값들의 리스트
//cleanup 함수 : effectCallback 함수가 리턴하는 함수
useEffect(()=> {
    ......

  return () => {   }
}, [dep1, dep2]);
```

# 2. useEffect

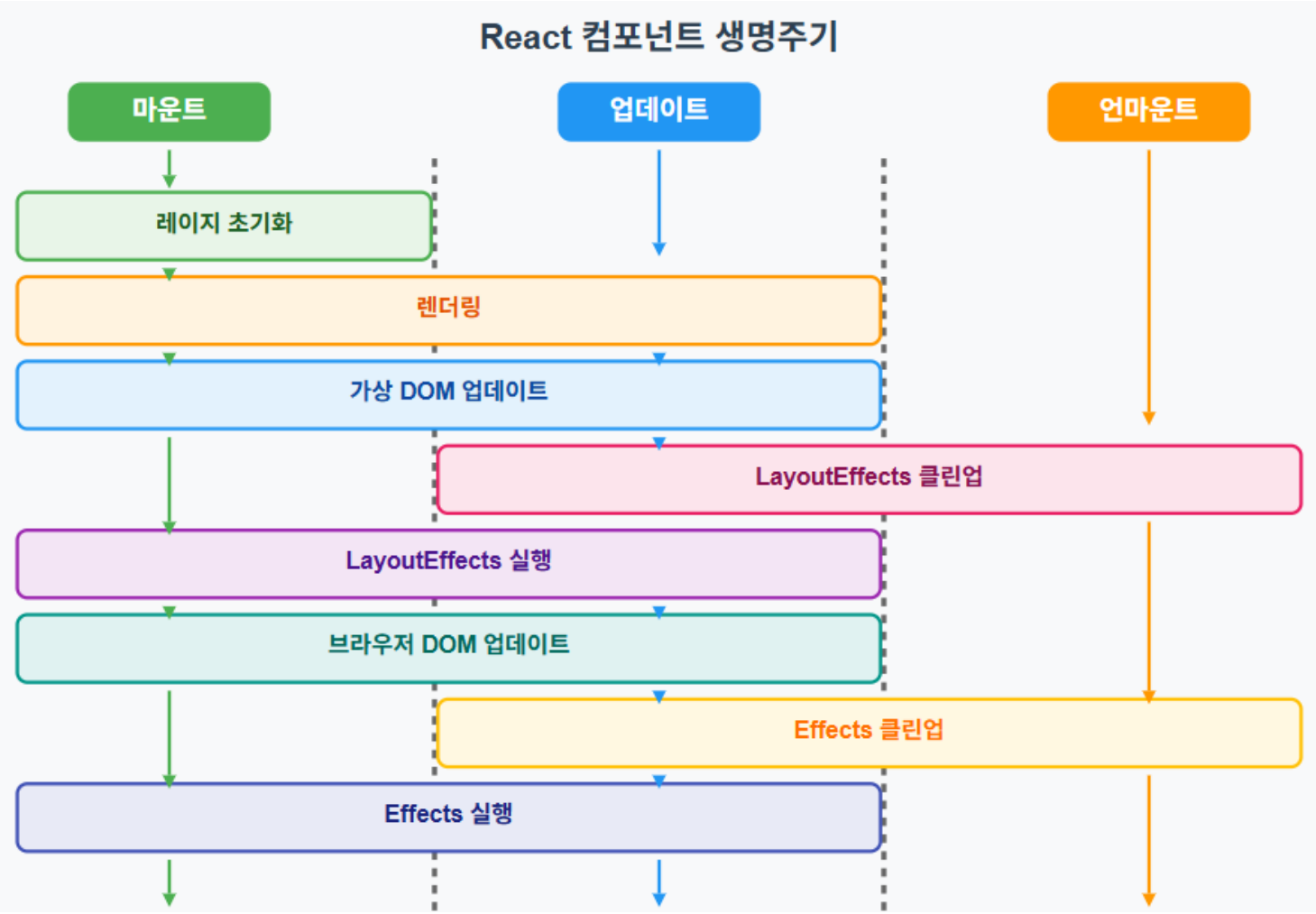
## ❖useEffect 수명주기

- 컴포넌트가 마운트될 때
  - 영역 1이 항상 실행됨
- 컴포넌트가 언마운트될 때
  - 영역 2(cleanup) 함수 내부가 항상 실행됨
  - cleanup 함수가 없다면 아무것도 실행되지 않음
- 컴포넌트가 업데이트될 때
  - 두번째 인자인 depsList를 전달하지 않았을 때
    - 컴포넌트가 업데이트될 때마다 영역2 -〉 영역1 순으로 실행
    - cleanup 함수가 먼저 실행됨
  - 두번째 인자가 빈배열( [ ] )일 때
    - 영역1, 영역2 모두 실행되지 않음
  - 두번째 인자로 [dep1, dep2]와 같이 의존 값(상태,속성)을 전달할 때
    - 지정한 의존 값이 바뀔때만 영역2에서 영역1 순으로 실행됨

```
useEffect(() => {
  //-----------
  //영역1
  //-----------

  // 클린업 함수
  return () => {
     //-------------
     //영역2
     //-------------
  };
}, depsList);
```

# 2. useEffect 생명주기

❖ useEffect 생명주기 흐름



# 3. useEffect 개념 이해 예제

## ❖use-effect-test-1 예제로 시작

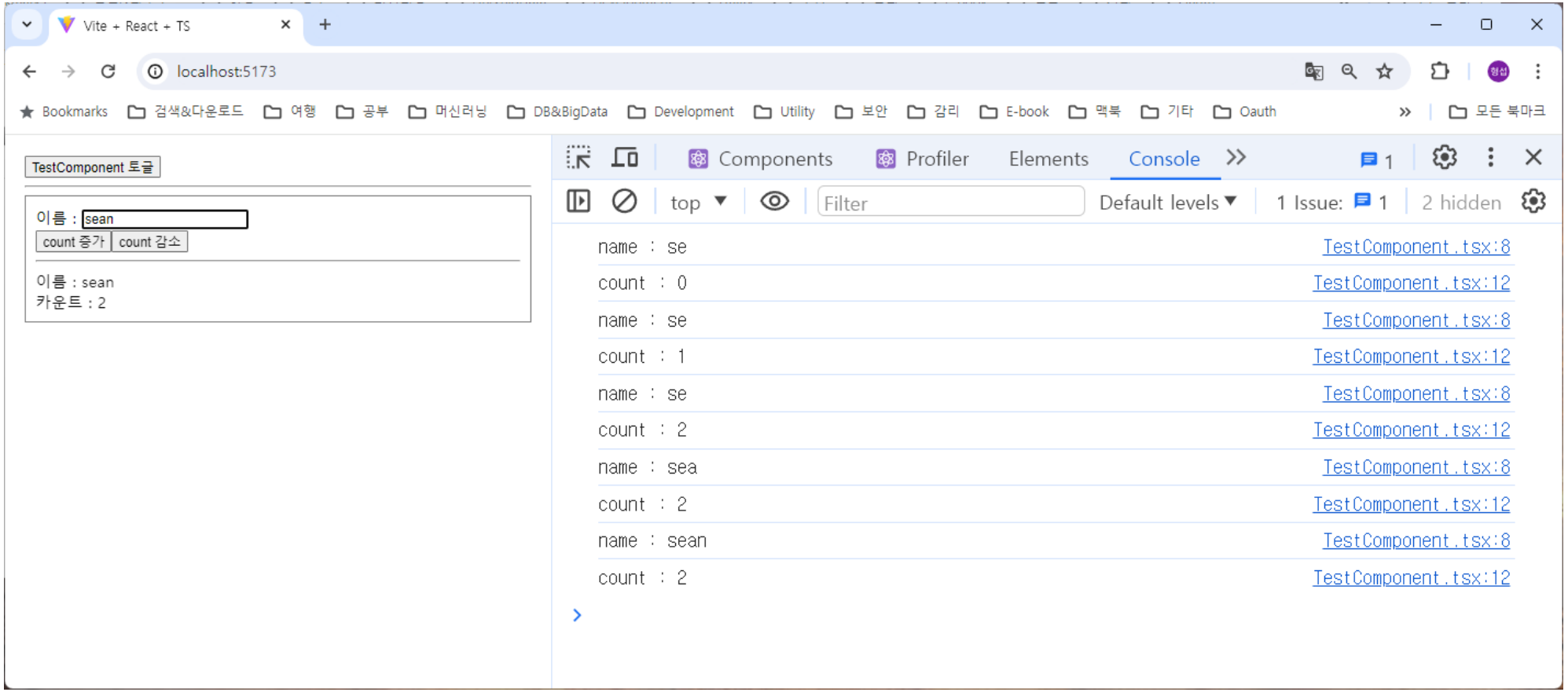
- TestComponent : useEffect를 사용하는 컴포넌트
- App 컴포넌트 : isVisible 상태를 이용해 TestComponent 를 토글하는 부모 컴포넌트
  - isVisible 속성이 false --〉 true 로 바뀔 때 : TestComponent 마운트
  - isVisible 속성이 true --〉 false 로 바뀔 때 : TestComponent 언마운트
- 초기 TestComponent의 useEffect 코드

```
useEffect(() => {
  console.log("name : " + name);
});

useEffect(() => {
  console.log("count : " + count);
});
```

- 실행 결과
  - 마운트, 업데이트될 때마다 두개의 effectCallback이 모두 실행됨

# 3. useEffect 개념 이해 예제

- 실행 결과
  - 마운트, 업데이트될 때마다 두개의 effectCallback이 모두 실행됨

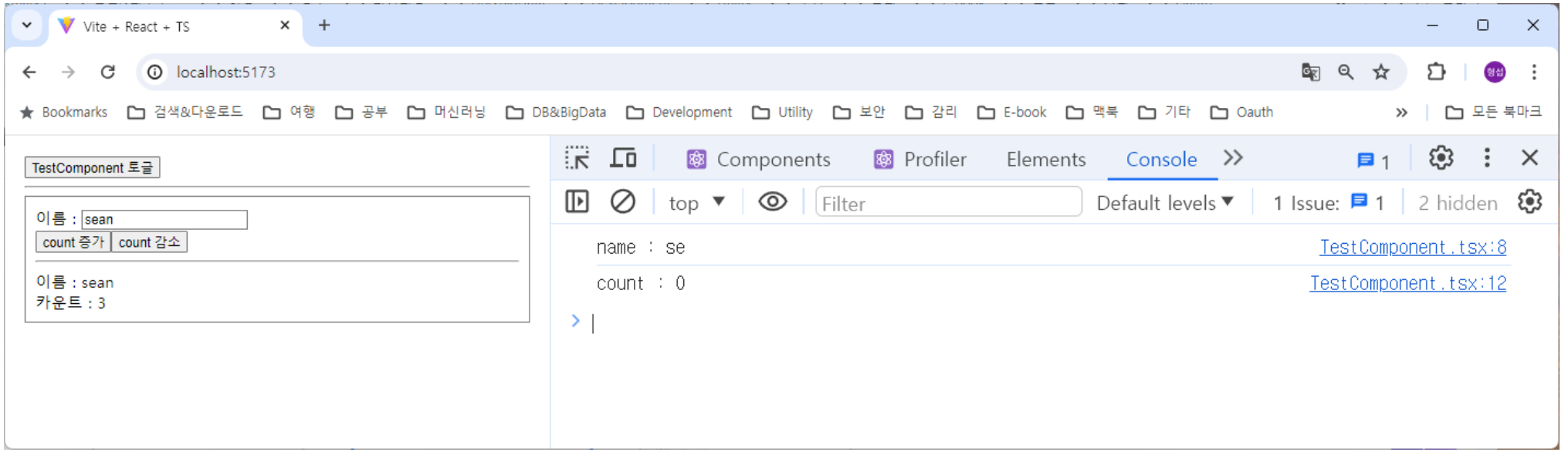


# 3. useEffect 개념 이해 예제

❖의존값 배열에 빈 배열을 지정하면?

- 컴포넌트가 마운트될 때만 effectCallback이 실행됨

```
useEffect(() => {
  console.log("name : " + name);
}, []);

useEffect(() => {
  console.log("count : " + count);
}, []);
```

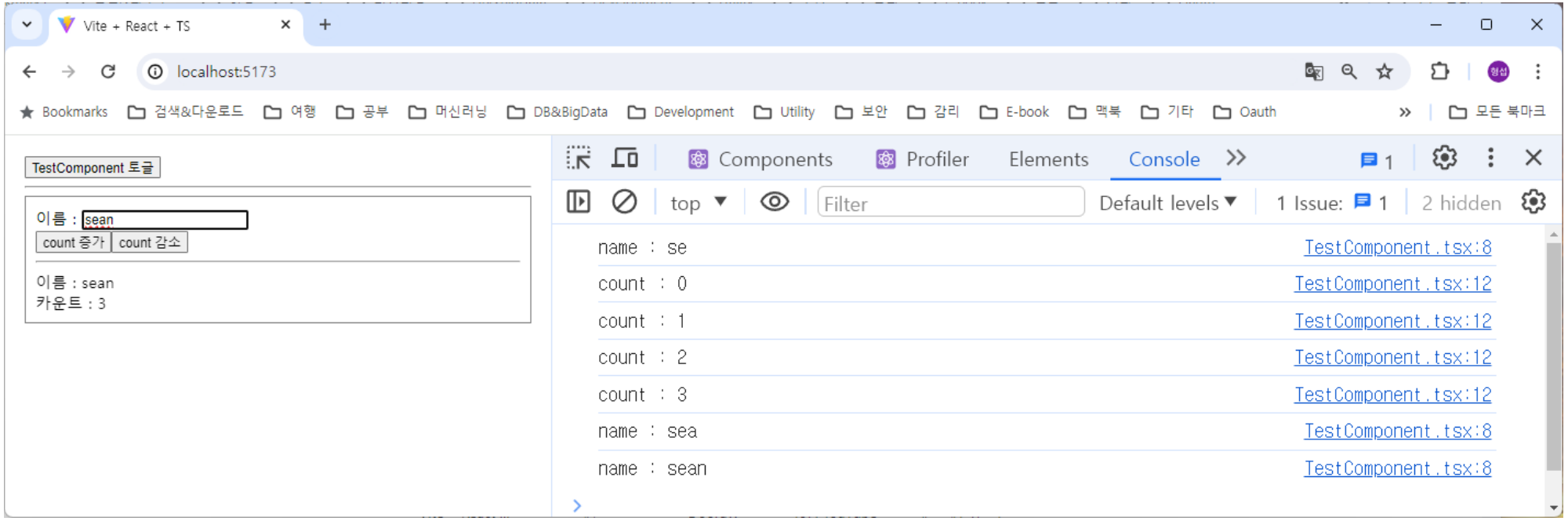


# 3. useEffect 개념 이해 예제

❖useEffect 훅의 두번째 인자에 의존값 배열(상태, 속성)를 지정하면?

- 컴포넌트가 마운트될 때와 의존 값이 바뀌어 업데이트될 때 effectCallback이 실행됨

```
useEffect(() => {
  console.log("name : " + name);
}, [name]);

useEffect(() => {
  console.log("count : " + count);
}, [count]);
```

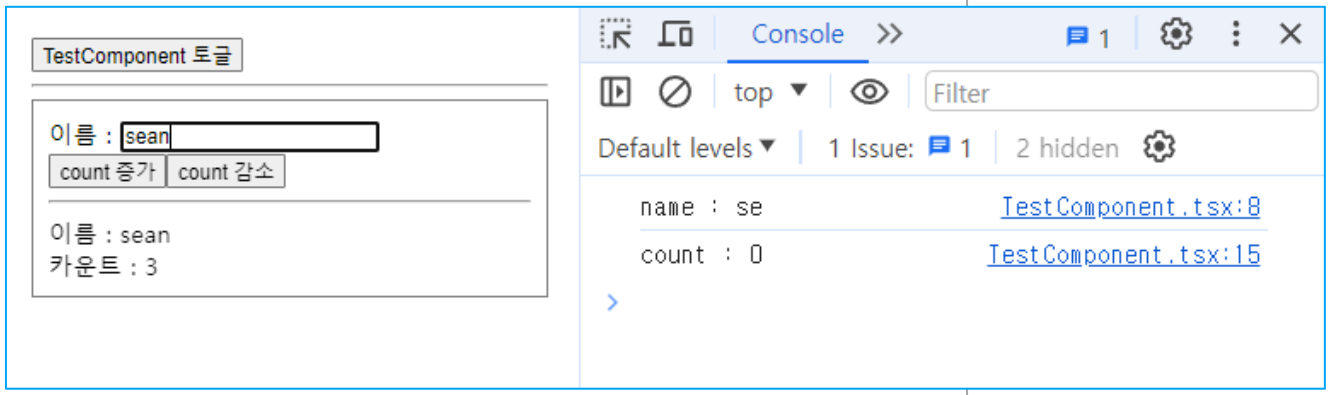


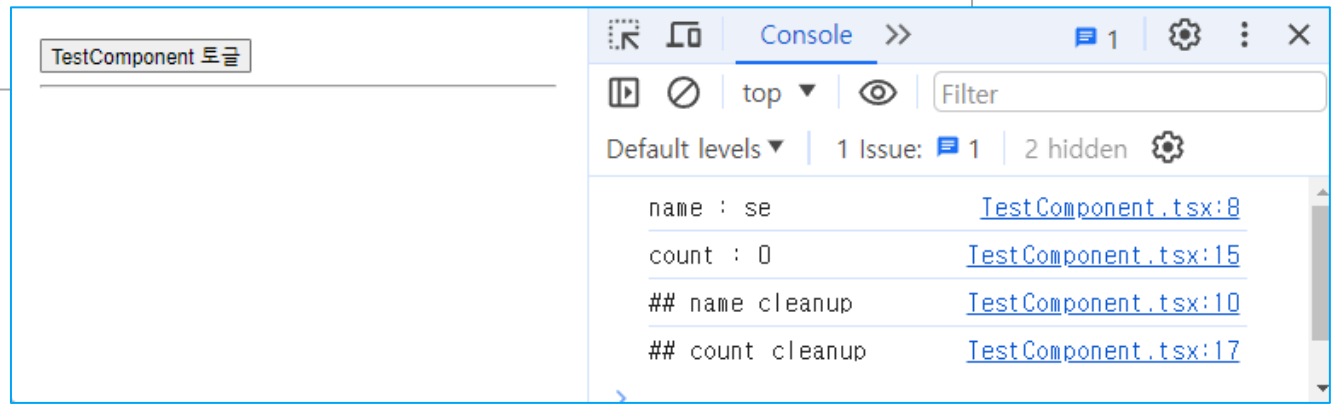
# 3. useEffect 개념 이해 예제

❖의존값 배열에 빈 배열을 지정하고 클린업 함수를 리턴하면?

- 컴포넌트가 마운트, 언마운트될 때 effectCallback이 실행됨

```
useEffect(() => {
   console.log("name : " + name);
   return () => {
     console.log("## name cleanup");
   };
 }, []);

 useEffect(() => {
   console.log("count : " + count);
   return () => {
     console.log("## count cleanup");
   };
 }, []);
```



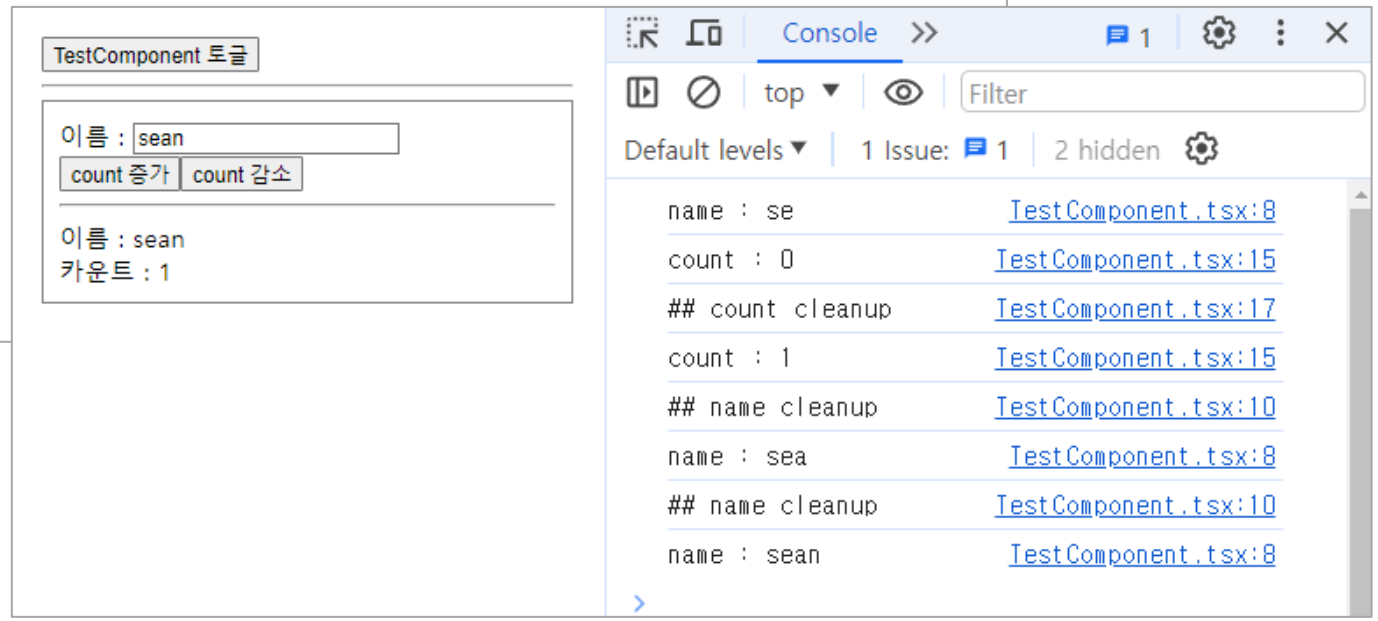



# 3. useEffect 개념 이해 예제

❖의존값을 지정하고 클린업 함수를 리턴하면?

- 컴포넌트가 언마운트될 때 클린업 함수 실행
- 컴포넌트가 업데이트될 때마다 클린업 함수 실행 후 effectCallback 내부 실행

```
useEffect(() => {
   console.log("name : " + name);
   return () => {
     console.log("## name cleanup");
   };
 }, [name]);

 useEffect(() => {
   console.log("count : " + count);
   return () => {
     console.log("## count cleanup");
   };
 }, [count]);
```



# 4. useReducer

❖지금까지의 상태는...

- useState()만을 사용
- 컴포넌트 내부에서 상태, 상태 변경 로직을 정의하였음
- 만일 복잡한 상태라면? --〉 상태 변경 로직도 포함하여 컴포넌트 내부가 복잡해짐.

❖useReducer란?

- 상태를 관리하는 기능을 컴포넌트로부터 분리시킬 수 있는 기능을 제공하는 리액트 훅
  - 여러 컴포넌트에서 상태 관리 기능을 공유할 수 있음
- useState() 와 상태 관리 기능을 대체할 수 있음

❖useReducer 훅을 제대로 사용하려면?

- reducer가 무엇인지 알아야 함

# 4. useReducer

## ❖reducer와 순수함수

- reducer란?
  - 배열(Array)의 메서드 중 reduce()에 인자로 전달하는 함수가 reducer임.
    - reduce() 메서드는 합계 값을 구할 때 사용할 수 있음
  - 두개의 인자를 이용해 연산한 값을 리턴하면 리턴값이 새로운 상태값이 됨.

```
const familyMembers = [
  { name:"홍길동", point: 10000, rel:"본인" },
  { name:"성춘향", point: 20000, rel:"처"  },
  { name:"홍예지", point: 15000, rel:"딸" },
  { name:"홍철수", point: 5000, rel:"아들" },
  { name:"홍희수", point: 10000, rel:"아들" },
];
```

```
const initialPoint = 10000;
const reducer = (totalPoint, member) => {
   totalPoint += member.point;
   return totalPoint;
}

const totalPoint = familyMembers.reduce(reducer, initialPoint);
console.log(`가족 합계 포인트 : ${totalPoint}`)
```

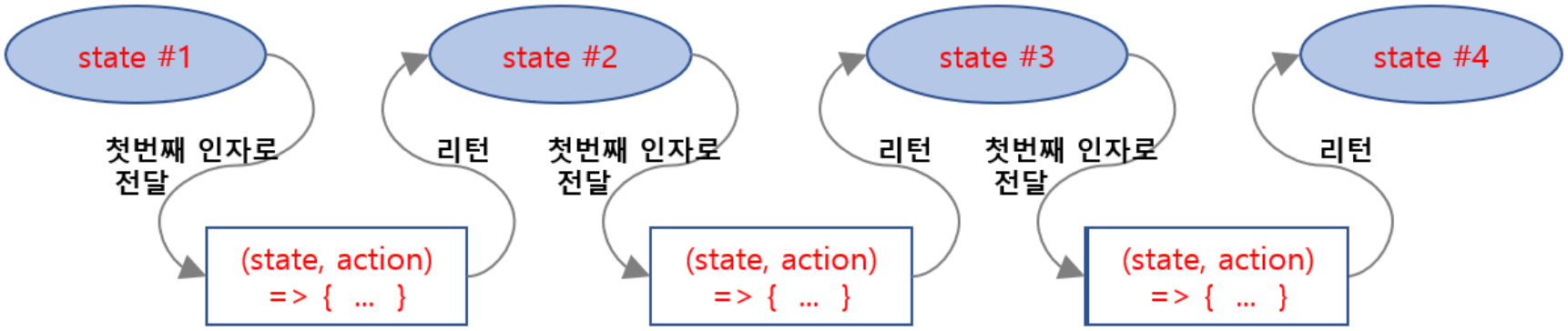
- 순수함수(pure function) : 다음의 조건을 만족해야 함.
  - 입력인자가 동일하면 리턴값도 동일해야 함
  - 부수효과(side effect)가 없어야 함
  - 함수에 전달된 인자는 불변성으로 여겨짐. 인자는 변경할 수 없음

# 4. useReducer

## ❖useReducer가 사용하는 reducer 함수 구조

```
(state, action) => {
   //state와 action을 이용해 연산을 수행한 새로운 상태(newState)를 생성하여 리턴합니다.
   return newState;
}
```

- 리턴값 : action(type, payload)와 기존 상태(state)를 이용해 **새로운 상태**를 만들어서 리턴함.
  --〉 불변성(immutability)
- 불변성을 사용하는 이유?
  - 렌더링 최적화
  - 상태 데이터에 대한 변경 추적과 효과적인 디버깅



# 4. useReducer

## ❖useReducer() Hook

- 함수 컴포넌트에서 reducer를 이용해 flux 패턴으로 상태를 변경할 수 있도록 함
  - 상태를 관리하는 기능을 컴포넌트로부터 분리시킬 수 있음
  - 단방향 흐름으로 상태 변경 추적으로 용이하게 함.
- 사용 방법
  - const [state, dispatch] = useReducer(reducer, initialState);
  - 입력값
    - initialState : 초기 상태 값
    - reducer : reducer 함수
  - 리턴값 : 배열
    - 첫번째 배열값 : 읽기 전용의 상태 값(state)
    - 두번째 배열값 : 상태 변경을 위해 액션(action: type+payload)을 인자로 전달하여 호출할 함수(dispatch)
  - 상태 변경은 반드시 dispatch 함수를 이용해야만 함
  - action의 형식 : type(액션 유형) + payload(액션을 수행할 때 필요한 데이터)
    - 예) { type:"addTodo", payload : { id: 1001, todo:"강아지 산책" } }
  - dispatch() 호출 형식
    - 예) dispatch( { type:"addTodo", payload : { id: 1001, todo:"강아지 산책" } } )

# 4. useReducer

❖예제 실습

- 제공되는 기존 예제
  - use-reducer-test-1
  - 두개의 컴포넌트가 동일한 상태 관리 기능을 이용하지만 코드가 중복된 상태
    - TodoList1
    - TodoList2
- useReducer를 이용하여 상태 변경, 관리 기능을 별도의 모듈로 분리하는 실습

# 4. useReducer

❖기존 TodoList1, TodoList2 컴포넌트 : Bold체 부분이 중복된 기능 코드

```
import { useState } from "react";

type TodoItemType = {
  id: number;
  todo: string;
};

const TodoList1 = () => {
  const [todo, setTodo] = useState<string>("");
  const [todoList, setTodoList] = useState<TodoItemType[]>([
    { id: 1, todo: "운동1" },
    { id: 2, todo: "독서1" },
    { id: 3, todo: "음악감상1" },
  ]);

  const addTodo = () => {
    const newTodoList = [...todoList, { id: new Date().getTime(), todo: todo }];
    setTodoList(newTodoList);
    setTodo("");
  };
  const deleteTodo = (id: number) => {
    const newTodoList = todoList.filter((todoItem) => todoItem.id !== id);
    setTodoList(newTodoList);
  };
```

# 4. useReducer

## ❖TodoList1, TodoList2 컴포넌트 (이어서)

```
  return (
    <div style={{ margin: "10px", padding: "10px", border: "solid 1px gray" }}>
      <input type="text" onChange={(e) => setTodo(e.target.value)} value={todo} />
      <button onClick={addTodo}>할일 추가</button>
      <ul>
        {todoList.map((item) => (
          <li key={item.id}>
            {item.todo}   
            <button onClick={() => deleteTodo(item.id)}>삭제</button>
          </li>
        ))}
      </ul>
    </div>
  );
};

export default TodoList1;
```

# 4. useReducer

## ❖useReducer 적용

- src/TodoReducer.ts

```
export type TodoItemType = { id: number; todo: string; };

export const TodoActionCreator = {
  addTodo: (todo: string) => ({ type: "TODO_ADD" as const, payload: { todo: todo } }),
  deleteTodo: (id: number) => ({ type: "TODO_DELETE" as const, payload: { id: id } }),
};
export type TodoActionType =
   ReturnType<typeof TodoActionCreator.addTodo> | ReturnType<typeof TodoActionCreator.deleteTodo>;

let newTodoList;
export const TodoReducer = (todoList: TodoItemType[], action: TodoActionType) => {
  switch (action.type) {
    case "TODO_ADD":
      newTodoList = [...todoList, { id: new Date().getTime(), todo: action.payload.todo }];
      return newTodoList;
    case "TODO_DELETE":
      newTodoList = todoList.filter((todoItem) => todoItem.id !== action.payload.id);
      return newTodoList;
    default:
      return todoList;
  }
};
```

# 4. useReducer

- src/components/TodoList1.tsx : TodoList2도 동일한 코드로 변경
  - 초기 상태(initialTodoList)만 서로 다르게 지정

```
import { useReducer, useState } from "react";
import { TodoActionCreator, TodoItemType, TodoReducer } from "../TodoReducer";

const initialTodoList1: TodoItemType[] = [
  { id: 1, todo: "운동1" },
  { id: 2, todo: "독서1" },
  { id: 3, todo: "음악감상1" },
];

const TodoList1 = () => {
  const [todoList, dispatchTodoList] = useReducer(TodoReducer, initialTodoList1);
  const [todo, setTodo] = useState("");
  const addTodo = () => {
    dispatchTodoList(TodoActionCreator.addTodo(todo));
    setTodo("");
  };
  const deleteTodo = (id: number) => {
    dispatchTodoList(TodoActionCreator.deleteTodo(id));
  };
```

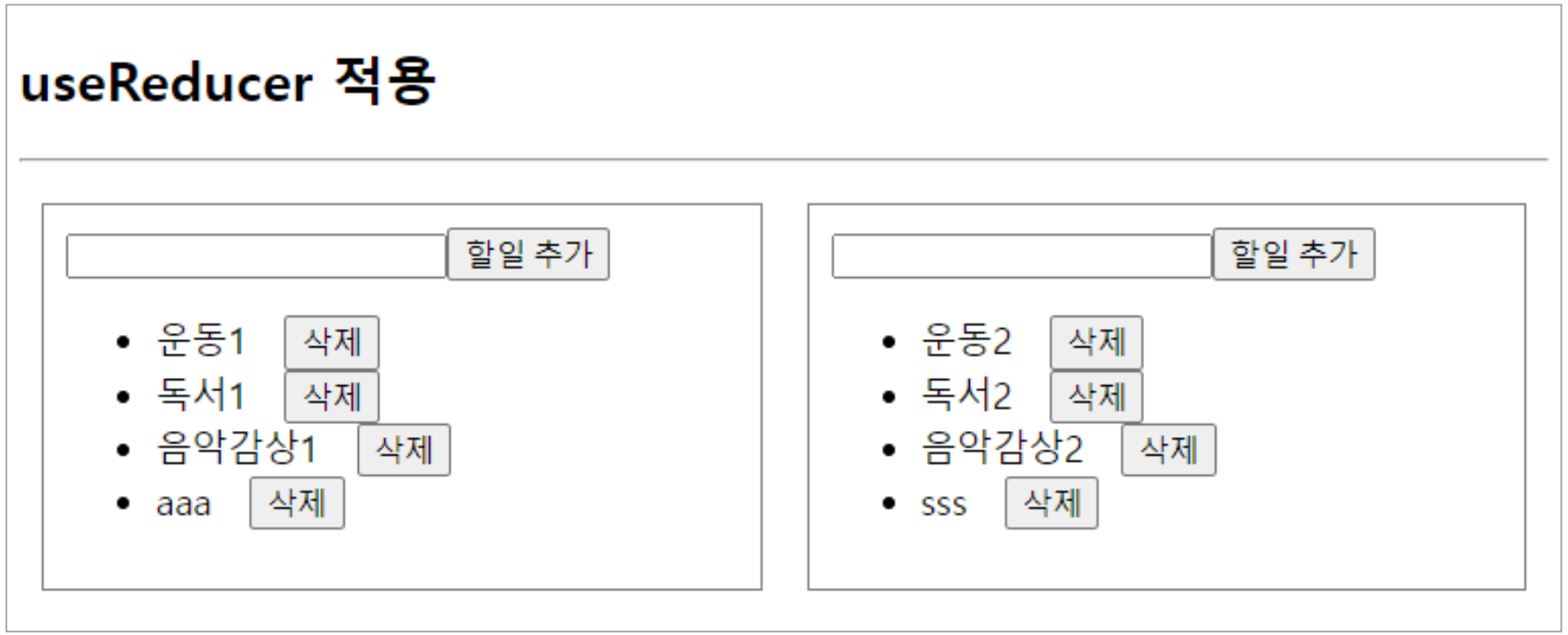
# 4. useReducer

- src/components/TodoList1.tsx (이어서)

```
  return (
    <div style={{ margin: "10px", padding: "10px", border: "solid 1px gray" }}>
      <input type="text" onChange={(e) => setTodo(e.target.value)} value={todo} />
      <button onClick={addTodo}>할일 추가</button>
      <ul>
        {todoList.map((item) => (
          <li key={item.id}>
            {item.todo}   
            <button onClick={() => deleteTodo(item.id)}>삭제</button>
          </li>
        ))}
      </ul>
    </div>
  );
};

export default TodoList1;
```

# 4. useReducer

- 실행 결과



# 5. Memoization Hook

## ❖메모이제이션이란?

- 기존에 연산된 결과값을 메모리에 캐싱하고 동일한 입력, 환경에서 재사용하는 기법
  - 적절한 사용 --〉 중복 처리를 피할 수 있음 --〉 애플리케이션 성능을 최적화할 때 사용
- 메모이제이션 기능을 제공하는 훅
  - useMemo
    - 함수가 호출된 후 리턴값을 캐싱하여 재사용.
    - 캐싱되는 것은 함수 호출 후의 리턴값
  - useCallback
    - 컴포넌트 내부의 함수를 캐싱하여 렌더링할 때마다 함수가 다시 생성되지 않게 해줌
    - 캐싱되는 것은 컴포넌트 내부의 함수!!

# 5. Memoization Hook

❖ 미리 제공되는 예제

- memorization-hook-test -1
- 간단한 TodoList 예제(App.tsx)
  - 볼드체로 보이는 부분 검토 : getTodoListCount() 함수와 이를 호출하는 부분

```
import { useState } from "react";

type TodoListItemType = { id: number; todo: string; };

const getTodoListCount = (todoList: Array<TodoListItemType>) => {
  console.log("## TodoList 카운트 : ", todoList.length);
  return todoList.length;
};

const App = () => {
  const [todoList, setTodoList] = useState<Array<TodoListItemType>>([]);
  const [todo, setTodo] = useState<string>("");

  const addTodo = (todo: string) => {
    const newTodoList = [...todoList, { id: new Date().getTime(), todo: todo }];
    setTodoList(newTodoList);
    setTodo("");
  };
```

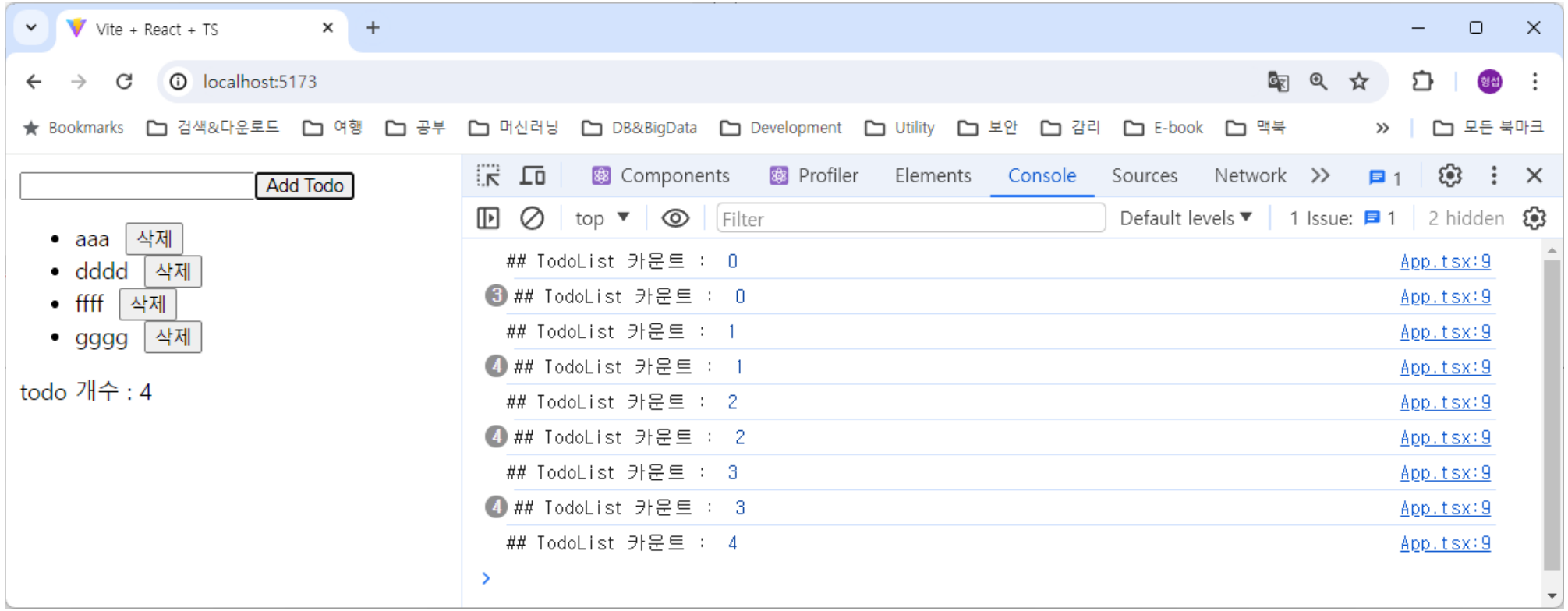
# 5. Memoization Hook

- 간단한 TodoList 예제(App.tsx) (이어서)

```
  const deleteTodo = (id: number) => {
    const index = todoList.findIndex((item) => item.id === id);
    const newTodoList = [...todoList];
    newTodoList.splice(index, 1);
    setTodoList(newTodoList);
  };

  return (
    <div className="boxStyle">
      <input type="text" value={todo} onChange={(e) => setTodo(e.target.value)} />
      <button onClick={() => addTodo(todo)}>Add Todo</button><br />
      <ul>
        {todoList.map((item) => (
          <li key={item.id}>
            {item.todo}  
            <button onClick={() => deleteTodo(item.id)}>삭제</button>
          </li>
        ))}
      </ul>
      <div>todo 개수 : {getTodoListCount(todoList)}</div>
    </div>
  );
};
export default App;
```

# 5. Memoization Hook

- 실행 결과 : 브라우저에서 실행 후 개발자도구 – 콘솔 창을 확인
  - 새로운 항목 추가를 위해 입력필드에 타이핑하는 동안에 카운트가 변경된 것이 없음에도 불구하고 getTodoListCount 함수가 매번 호출됨
  - 최적화 대상



# 5. Memoization Hook

## ❖useMemo() 훅 사용 방법

```
//factory : 캐싱할 값을 만들어내는 함수
//depsList : 의존값 리스트. 이 리스트의 값이 바뀌면 함수를 호출하여 캐시를 다시 생성함
//캐싱할 값은 제네릭으로 T에 타입을 지정함

const memoizedValue = useMemo<T>(factory: ( )=>T, depsList)
```

- depsList 의존 배열 객체가 중요
  - 이 배열에 지정된 값이 바뀌기 전까지는 캐시를 유지함
  - 이 값이 바뀌면 함수를 다시 실행하여 리턴된 값으로 캐시를 갱신

# 5. Memoization Hook

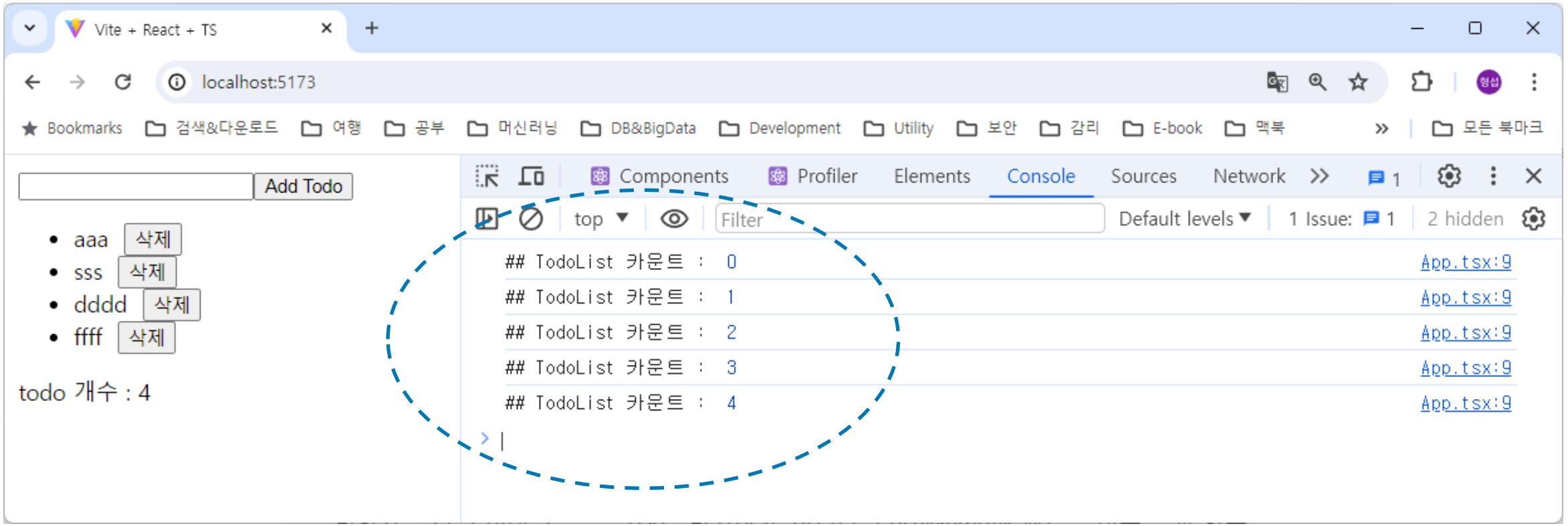
## ❖기존 예제에 useMemo() 훅 적용

- Bold체로 보이는 부분을 검토하고 수정할 것

```
import { useMemo, useState } from "react";

.....(생략)
const getTodoListCount = (todoList: Array<TodoListItemType>) => {
  console.log("## TodoList 카운트 : ", todoList.length);
  return todoList.length;
};

const App = () => {
  const [todoList, setTodoList] = useState<Array<TodoListItemType>>([]);
  const [todo, setTodo] = useState<string>("");
  //의존값으로 지정된 todoList가 바뀌면 함수를 다시 호출하여 값을 캐싱함
  const memoizedCount = useMemo<number>(() => getTodoListCount(todoList), [todoList]);
  .....(생략)
  return (
    <div className="boxStyle">
      .....(생략)
      <div>todo 개수 : {memoizedCount}</div>
    </div>
  );
};
export default App;
```

# 5. Memoization Hook

- 실행 결과



# 5. Memoization Hook

## ❖useCallback() 훅을 사용하기 전 컴포넌트 내부의 함수

- 컴포넌트가 렌더링될 때마다 매번 새롭게 만들어짐
- 함수를 속성을 이용해 자식 컴포넌트로 전달하는 경우에는 매번 새로운 함수가 전달되므로 렌더링 성능 최적화가 어려워 짐
  - 성능에 민감한 화면이라면 나쁜 영향을 끼침

## ❖useCallback() 훅 사용법

```
//callback : 캐싱하려는 대상 함수
//depsList : 의존값 리스트. 이 리스트의 값이 바뀌지 않으면 기존 함수를 이용함

const memoizedCallback = useCallback(callback, depsList)
```

- useCallback 을 이용하면 다시 렌더링되더라도 함수를 새롭게 생성하지 않고 캐싱된 함수를 이용함

# 5. Memoization Hook

## ❖useCallback 훅 적용 : 실행 결과 화면은 차이가 없음

```
import { useCallback, useMemo, useState } from "react";
.....(생략)
const App = () => {
  .....(생략)
  const addTodo = useCallback(
    (todo: string) => {
      const newTodoList = [...todoList, { id: new Date().getTime(), todo: todo }];
      setTodoList(newTodoList);
      setTodo("");
    },
    [todoList]
  );
  const deleteTodo = useCallback(
    (id: number) => {
      const index = todoList.findIndex((item) => item.id === id);
      const newTodoList = [...todoList];
      newTodoList.splice(index, 1);
      setTodoList(newTodoList);
    },
    [todoList]
  );
  .....(생략)
};
export default App;
```

# 5. Memoization Hook

## ❖useCallback() 훅 주의 사항

- useCallback의 두번째 인자인 depsList를 적절하게 지정해야 함
  - useCallback 훅에 의해 캐싱된 함수는 함수가 생성될 때의 상태나 속성을 참조하기 때문
  - 상태나 속성이 변경되면 캐싱된 함수도 갱신해줘야 함
- 만일 addTodo의 의존값 todoList를 제거하면?
  - 여러개의 할일을 추가하더라도 마지막에 추가한 하나의 항목만 남음
  - 이유는 무엇인가?
    - addTodo memoizedCallback 이 만들어진 시점에는 todoList 상태가 [ ] 이었기 때문
    - 항상 빈 리스트에 추가하는 것
- 결론
  - 의존값 리스트는 아주 중요함
  - useEffect(), useCallback(), useMemo()

# 5. Memoization Hook

❖메모이제이션 훅을 반드시 사용해야 하는가?

- 성능 이슈가 있는 곳에서만 사용함. 반드시 사용할 필요는 없음
- 캐싱으로 인해 추가적인 리소스(메모리)를 사용하므로 남용하면 오히려 성능에 나쁜 영향을 끼칠 수 있음

❖메모이제이션 훅 + memo() 고차함수

- 렌더링 성능 최적화를 위한 일반적인 조합
- 다음장에서 살펴볼 내용

# 6. Custom Hook

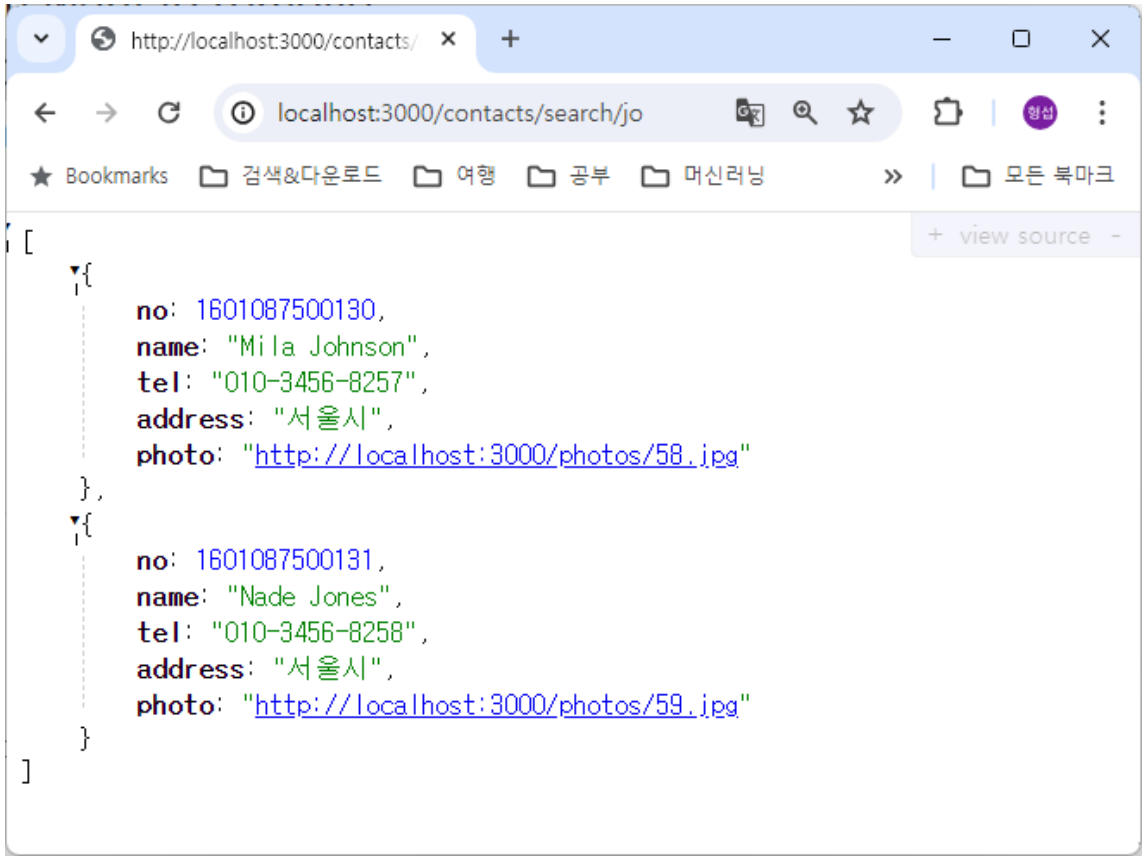
## ❖사용자 정의 훅(Custom Hook)이란?

- 개발자가 직접 작성하는 리액트 훅
- 기본으로 제공되는 훅(useState, useEffect 등)을 이용해 재사용하고자 하는 기능을 사용자 정의 훅으로 작성하고 여러 컴포넌트에서 재사용할 수 있음
- 상태
- 목적 : 상태, 기능의 재사용
- 관례적인 명명 규칙 : use~ 로 시작하는 이름을 부여함

# 6. Custom Hook

## ❖미리 제공되는 예제

- custom-hook-test-1
  - 사용자로부터 이름을 입력받아서 검색하는 예제
- backend API 사전 준비
  - 강사로부터 contactsvc 예제를 받아서 실행한다.
    - npm install
    - npm run dev
  - http://localhost:3000 으로 접속하여 백엔드 API를 테스트해봄
  - 사용할 엔드포인트 :
    http://localhost:3000/contacts/search/[name]
    - 이름에 [name]이 포함된 사람들을 조회함
    - 예) http://localhost:3000/contacts/search/se



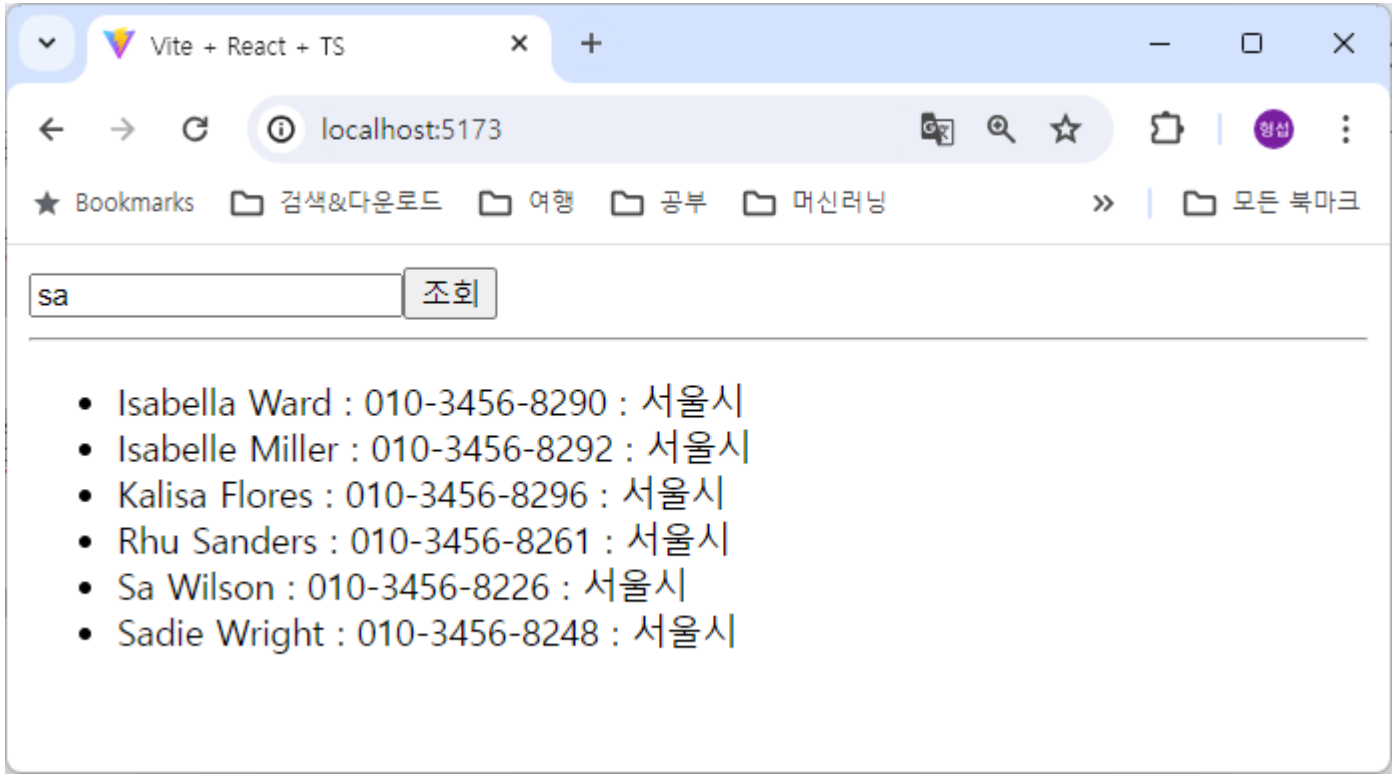
# 6. Custom Hook

- 기존 코드 검토(src/components/SearchContact.tsx)
  - 사용자로부터의 이름 입력을 위해 useState() 훅과 onChangeHandler를 직접 작성하였음
  - 하지만 입력 로직이 조금 복잡해지면?
    - 특수문자를 입력하지 못하게 하거나
    - 숫자만 입력할 수 있도록 한다든지?
  - 매번 onChange 속성에 등록할 이벤트 핸들러 함수를 컴포넌트 내부에 직접 작성해야 함

```
import { useState } from "react";
.....
const SearchContact = ({ searchByName }: PropsType) => {
  const [name, setName] = useState<string>("");
  ......(생략)
  return (
    <div>
      <input type="text" value={name} onChange={(e) => setName(e.target.value)} />
      <button onClick={searchClickHandler}>조회</button>
    </div>
  );
};

export default SearchContact;
```

# 6. Custom Hook

- 기존 코드 실행 결과



# 6. Custom Hook

## ❖첫번째 Custom Hook 적용

- Custom Hook 작성 : src/hooks/useInput.tsx

```
import { ChangeEvent, useState } from "react";

//가장 단순한 입력처리 훅
const useInput = (initialValue: string) => {
  const [value, setValue] = useState<string>(initialValue);
  const changeHandler = (e: ChangeEvent<HTMLInputElement>) => {
    setValue(e.target.value);
  };
  return [value, changeHandler] as const;
};
//특수문자 입력을 허용하지 않는 훅
const useInputNoSpecialChar = (initialValue: string) => {
  const [value, setValue] = useState<string>(initialValue);
  const changeHandler = (e: ChangeEvent<HTMLInputElement>) => {
    // 특수문자, 괄호, 점 제거를 위한 정규식, 그리고 eslint 비활성화
    // eslint-disable-next-line
    const reg = /[₩{₩}₩[₩]₩/?.,;:|₩)*~`!^₩-+<>@₩#$%&₩₩₩=₩(₩'₩"]/gi;
    setValue(e.target.value.replace(reg, ""));
  };
  return [value, changeHandler] as const;
};
```

# 6. Custom Hook

- Custom Hook 작성 : src/hooks/useInput.tsx (이어짐)

```
//숫자이외의 문자는 입력을 허용하지 않는 훅
const useInputNumber = (initialValue: number) => {
  const [value, setValue] = useState<number>(initialValue);
  const changeHandler = (e: ChangeEvent<HTMLInputElement>) => {
    const number = parseInt(e.target.value, 10);
    setValue(isNaN(number) ? 0 : number);
  };

  return [value, changeHandler] as const;
};

export { useInput, useInputNumber, useInputNoSpecialChar };
```

# 6. Custom Hook

- Custom Hook 적용 : src/components/SearchContact.tsx 변경

```
import { useInputNoSpecialChar } from "../hooks/useInput";
.....(생략)

const SearchContact = ({ searchByName }: PropsType) => {
  const [name, nameChangeHandler] = useInputNoSpecialChar("");
  .....(생략)

  return (
    <div>
      <input type="text" value={name} onChange={nameChangeHandler} />
      <button onClick={searchClickHandler}>조회</button>
    </div>
  );
};

export default SearchContact;
```

- 실행 후 이름 검색 필드에 특수 문자를 입력할 수 없음을 확인함

# 6. Custom Hook

## ❖두번째 Custom Hook 적용

- 기존 App 컴포넌트 코드 검토 : src/App.tsx
  - searchByName 함수 : axios를 이용해 백엔드 API로부터 조회하는 코드와 비동기 처리 지연동안 스피너를 보여주기 위해 3가지 상태(contacts, isLoading, error)를 변경하는 기능

```
.....(생략)
const App = () => {
  const [contacts, setContacts] = useState<ContactType[]>([]);
  const [isLoading, setIsLoading] = useState<boolean>(false);
  const [error, setError] = useState<Error>();

  const searchByName = async (name: string) => {
    try {
      setIsLoading(true);
      const response = await axios.get<ContactType[]>(`/contacts_long/search/${name}`);
      setContacts(response.data);
      setIsLoading(false);
      setError(undefined);
    } catch (e) {
      setIsLoading(false);
      setError(e as unknown as Error);
    }
  };
```

# 6. Custom Hook

- 기존 App 컴포넌트 코드 검토 : src/App.tsx(이어서)

```
 return (
    <div>
      <SearchContact searchByName={searchByName} />
      {error ? <h4>에러 메시지 : {error.message}</h4> : ""}
      <hr />
      <ContactList contacts={contacts} />
      {isLoading ? <ReactCsspin opacity={0.8} /> : ""}
    </div>
  );
};

export default App;
```

- 이와 같은 비동기 처리 기능이 여러 컴포넌트에서 사용된다면?
  - Custom Hook 으로 분리함

# 6. Custom Hook

- 두번째 Custom Hook 작성 : src/hooks/useFetch.ts
  - 응답 데이터, 에러, 로딩중을 다루는 상태를 Custom Hook 내부에 배치
  - 비동기 처리 진행 과정에서 이 상태들을 적절히 변경
  - 상태와 요청 함수를 리턴하도록 하여 컴포넌트에서 사용

```
import { useState } from "react";
import axios, { AxiosError, AxiosRequestConfig, AxiosResponse } from "axios";

const useFetch = <T>(params?: AxiosRequestConfig) => {
  const [responseData, setResponseData] = useState<T>();
  const [error, setError] = useState<AxiosError>();
  const [isLoading, setIsLoading] = useState<boolean>(false);
```

# 6. Custom Hook

- 두번째 Custom Hook 작성 : src/hooks/useFetch.ts (이어서)

```
  const fetchData = async (url: string) => {
    setResponseData(undefined);
    setError(undefined);
    setIsLoading(true);
    try {
      const response: AxiosResponse<T> = await axios.get<T, AxiosResponse<T>>(url, params);
      setResponseData(response.data);
    } catch (e) {
      setError(e as unknown as AxiosError);
    } finally {
      setIsLoading(false);
    }
  };

  const requestFetchData = (url: string) => {
    fetchData(url);
  };

  return { responseData, error, isLoading, requestFetchData };
};

export { useFetch };
```

# 6. Custom Hook

- App 컴포넌트 변경 : src/App.tsx
  - useFetch 훅을 호출하여 리턴받은 상태, 함수를 이용해 UI 구성

```
.....(생략)
import { useFetch } from "./hooks/useFetch";
.....(생략)

const App = () => {
  const { responseData, isLoading, error, requestFetchData } = useFetch<ContactType[]>({ timeout: 10000 });
  const searchByName = (name: string) => {
    requestFetchData(`/contacts_long/search/${name}`);
  };

  return (
    <div>
      <SearchContact searchByName={searchByName} />
      {error ? <h4>에러 메시지 : {error.message}</h4> : ""}
      <hr />
      <ContactList contacts={responseData ? responseData : []} />
      {isLoading ? <ReactCsspin opacity={0.8} /> : ""}
    </div>
  );
};

export default App;
```

# 7. 클로저 트랩과 exhaustive-deps

❖클로저 트랩이란?

- useEffect(), useCallback(), useMemo() 과 같이 depsList를 사용하는 훅에서 의존값을 depsList에 지정하지 않아서 상태가 변경되어도 callback이 참조하고 있는 이전의 상태를 사용하게 되는 현상.
  - callback이 호출되었을 때 참조했던 값(상태)을 클로저가 참조하고 있기 때문에 발생함. 2장 클로저 참조
  - callback 함수 : 클로저 함수
  - 상태 : 자유 변수
- 이 문제를 이해하고 해결하려면 클로저의 개념과 depsList 개념을 정확하게 이해해야 함

- 참조
  - https://betterprogramming.pub/understanding-the-closure-trap-of-react-hooks-6c560c408cde

# 7. 클로저 트랩과 exhaustive-deps

## ❖exhaustive-deps

- exhaustive-deps란?
  - 철저한 의존값
  - useEffect(), useMemo(), useCallback() 등에서 의존값을 철저하게 지정하라는 것
- 자주 나타나는 경고 메시지

> React Hook useEffect has a missing dependency: 'xxx'. Either include it or remove the dependency array

  - 컴포넌트가 마운트될 때만 실행하도록 하기위해 useEffect() 훅에서 depsList를 빈배열로 지정하면 항상 보게 되는 경고메시지
  - 의도치 않은 버그를 만날 가능성이 높아짐
    - 대표적인 것이 클로저 트랩
  - 다음 주석을 지정해서 ESLint 기능을 비활성화 할 수 있지만 권장되지 않음
    - // eslint-disable-next-line react-hooks/exhaustive-deps

# 7. 클로저 트랩과 exhaustive-deps

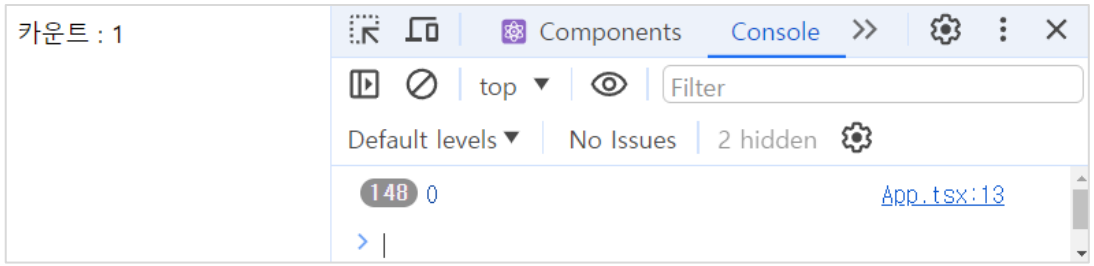
❖간단한 예제 확인 : closure-trap-test

- src/App.tsx 검토 : exhaustive-deps-lint에 의해 경고메시지 나타남

```
import { useEffect, useState } from "react";

const App = () => {
  const [count, setCount] = useState<number>(0);

  useEffect(() => {
    setInterval(() => {
      setCount(count + 1);
    }, 2000);
  }, []);

  useEffect(() => {
    setInterval(() => {
      console.log(count);
    }, 2000);
  }, []);

  return <div>카운트 : {count}</div>;
};

export default App;
```

# 7. 클로저 트랩과 exhaustive-deps

- 실행 결과
  - 2초마다 count를 1씩 증가시키는 작업을 마운트할 때 추가하고
  - 2초간격으로 count값을 확인한다.
  - 하지만 예상 결과는 오작동 : 화면 출력은 only 1, 콘솔 로그는 only 0



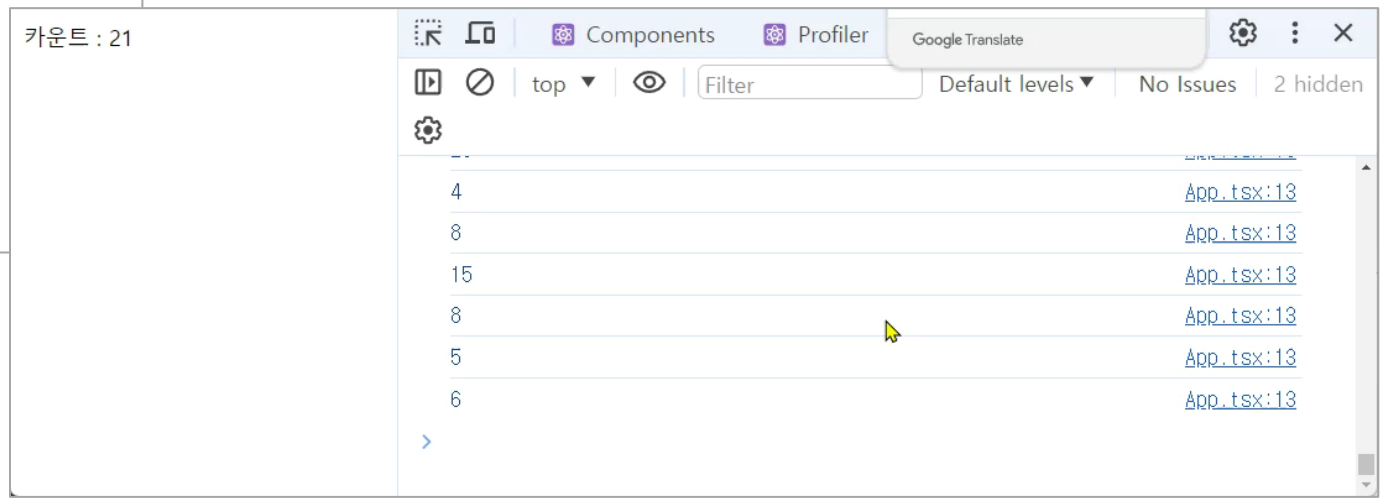

- 원인은 무엇인가?
  - 첫번째 useEffect 훅 내부의 callback이 마운트될 때 호출되면서 클로저 생성
  - callback 호출 시점의 count가 0
  - setCount(count+1) 코드로 count를 1증가 시키더라도 이 클로저는 이 시점의 count(0)을 참조함
- 해결책
  - 의존값을 지정하여 상태가 바뀔 때 callback 함수가 다시 호출되어 바뀐 count를 참조하도록 해야 함

# 7. 클로저 트랩과 exhaustive-deps

## ❖의존값을 depsList에 추가해보자

- src/App.tsx 변경

```
.....
//두개의 useEffect에 모두 의존값을 지정함
useEffect(() => {
 setInterval(() => {
   setCount(count + 1);
 }, 2000);
}, [count]);
.....
```




- 여전한 문제점 : count 가 증감을 빠르게 반복함
  - useEffect의 callback이 호출될 때마다 setInterval()을 등록하기 때문
  - 해결 방법 : cleanup을 이용해서 이전 setInterval 등록한 것을 clear 해주어야 함

# 7. 클로저 트랩과 exhaustive-deps

## ❖cleanup 추가 하여 해결

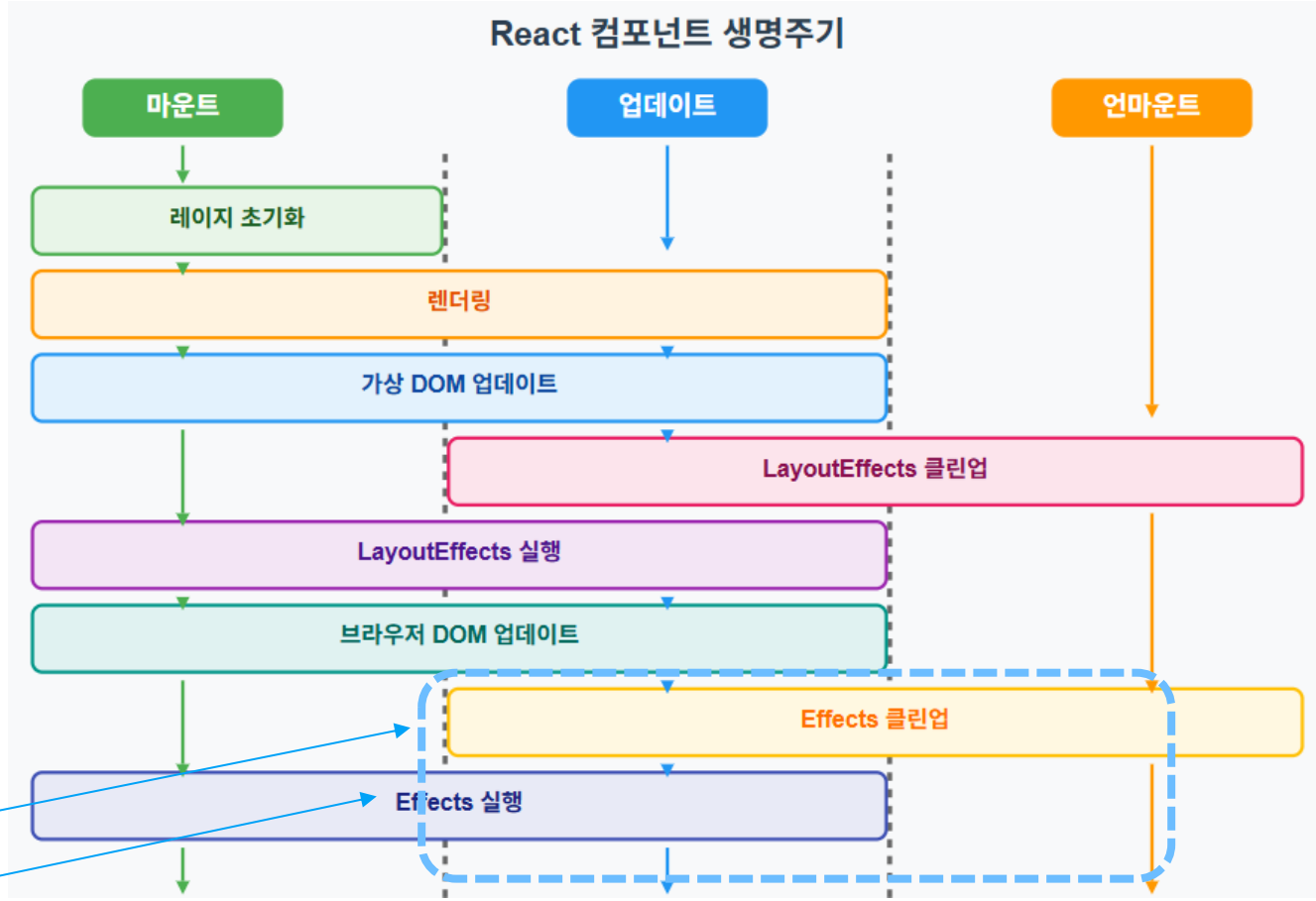
- src/App.tsx 변경

```
.....
//두개의 useEffect에 모두 의존값을 지정함
useEffect(() => {
 const handle = setInterval(() => {
   setCount(count + 1);
 }, 2000);
 return ()=>clearInterval(handle);
}, [count]);
.....
```

- 문제점
  - 정상실행이 되기는 하지만...
  - 매번 clearInterval -〉 setInterval 을 수행함

**이 시점에서 이전의 setInterval이 clear 됨**

**이 시점에서 새로운 setInterval이 설정됨**



# 7. 클로저 트랩과 exhaustive-deps

## ❖exhaustive-deps

- exhaustive-deps란?
  - 철저한 의존값
  - useEffect(), useMemo(), useCallback() 등에서  의존값을 철저하게 지정하라는 것
- 자주 나타나는 경고 메시지

> React Hook useEffect has a missing dependency: 'xxx'. Either include it or remove the dependency array

  - ESLint 규칙에 의해서 보여지는 경고
  - 컴포넌트가 마운트될 때만 실행하도록 하기위해 useEffect() 훅에서 depsList를 빈배열로 지정하면 자주 보게 되는 경고 메시지
  - 의도치 않은 버그를 만날 가능성이 높아짐
    - 대표적인 것이 클로저 트랩
  - 다음 주석을 지정해서 ESLint 기능을 비활성화 할 수 있지만 권장되지 않음
    - // eslint-disable-next-line react-hooks/exhaustive-deps

# 7. 클로저 트랩과 exhaustive-deps

❖ closure-trap-test 예제에 이어서

- 이전 예제에서 count가 바뀔 때마다 매번 clearInterval --〉 setInterval
  - 잠재적인 논리 오류 발생 가능성

- 해결 방법1
  - count 상태를 변경하는 작업을 함수를 이용한 업데이트로 변경함
    - 이전) setCount(count+1);
    - 변경) setCount((prevCount) =〉 prevCount+1 );
  - 이렇게 하면 count 상태를 의존값으로 depsList에 추가할 필요 없음
  - 이 방법은 useEffect 훅 내부에서 단순 상태를 변경할 때만 사용할 수 있음
  - src/App.tsx 변경

```
useEffect(() => {
  const handle = setInterval(() => {
    setCount((prevCount) => prevCount + 1);
  }, 2000);
  return () => clearInterval(handle);
}, []);
```

# 7. 클로저 트랩과 exhaustive-deps

- *해결 방법 2*
  - useReducer 훅을 이용해 상태 변경 기능을 컴포넌트 밖으로 분리함
  - src/CountReducer.ts 작성

```
export const CountAction = {
  increment: (value: number) => ({ type: "INCREMENT", payload: { value } }),
};

export type CountStateType = { count: number };
export type CountActionType = ReturnType<typeof CountAction.increment>;

export const CountReducer = (state: CountStateType, action: CountActionType) => {
  switch (action.type) {
    case "INCREMENT":
      return { ...state, count: state.count + action.payload.value };
    default:
      return state;
  }
};
```

# 7. 클로저 트랩과 exhaustive-deps

- 해결 방법2 (이어서)
  - src/App.tsx 변경 : useReducer를 사용하도록 변경

```
import { useEffect, useReducer } from "react";
import { CountAction, CountReducer, CountStateType } from "./CountReducer";

const initialState: CountStateType = { count: 0 };
const App = () => {
  const [state, dispatch] = useReducer(CountReducer, initialState);

  useEffect(() => {
    const handle = setInterval(() => {
      dispatch(CountAction.increment(1));
    }, 2000);
    return () => clearInterval(handle);
  }, [dispatch]);
  useEffect(() => {
    const handle = setInterval(() => {
      console.log(state.count);
    }, 2000);
    return () => clearInterval(handle);
  }, [state]);
  return <div>카운트 : {state.count}</div>;
};
export default App;
```

# 7. 클로저 트랩과 exhaustive-deps

❖ 실행 결과