**4. Create: 할 일 추가하기**

📄

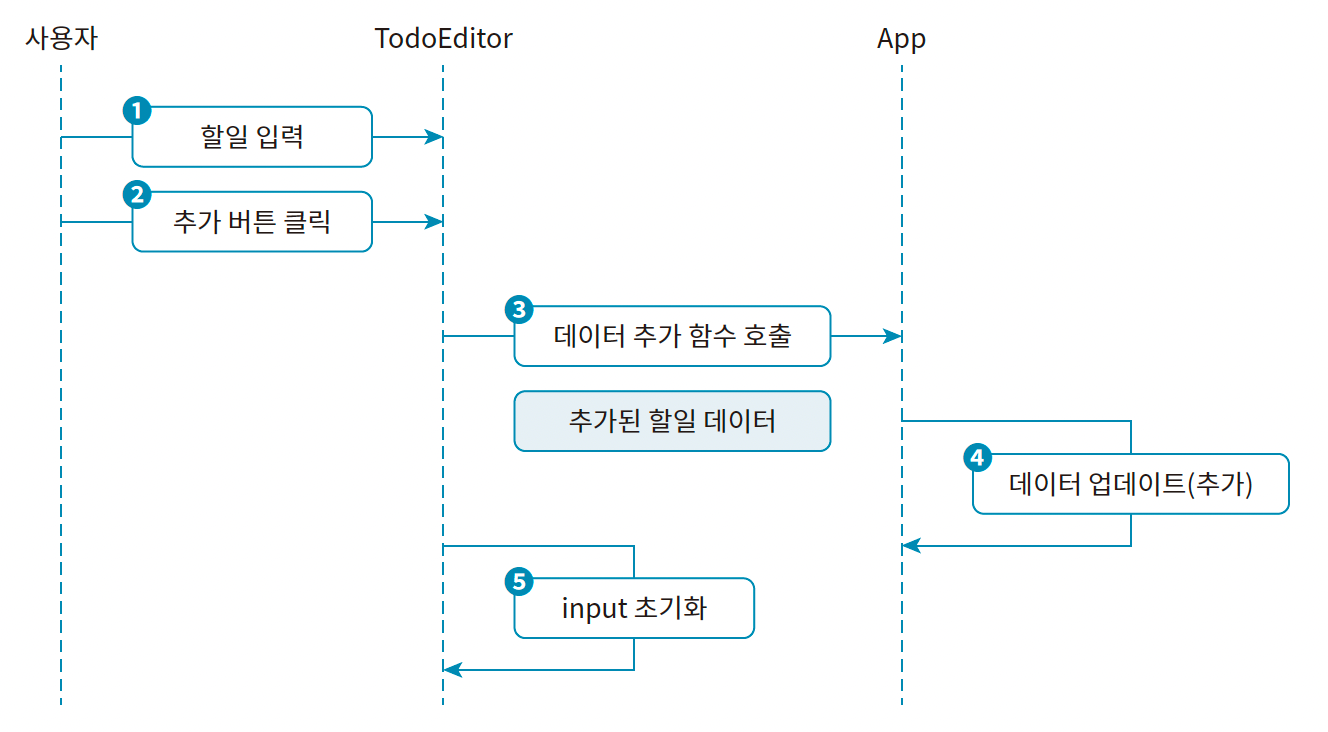
**Table Of Contents**

목 데이터 설정까지 완료했다면 이제 기능 구현을 위한 준비 과정은 모두 마쳤습니다.

이번에는 CRUD의 첫 번째 기능인 Create를 구현하겠습니다.

**기능 흐름 살펴보기**

다음 그림은 [할 일 관리] 앱에서 할 일이 추가되는 과정을 도식화한 것입니다.

[할 일 관리] 앱의 Create 기능 흐름

1. 사용자가 새로운 할 일을 입력합니다.
2. TodoEditor 컴포넌트에 있는 <추가> 버튼을 클릭합니다.
3. TodoEditor 컴포넌트는 부모인 App에게 아이템 추가 이벤트가 발생했음을 알리고 사용자가 추가한 할 일 데이터를 전달합니다.
4. App 컴포넌트는 TodoEditor 컴포넌트에서 받은 데이터를 이용해 새 아이템을 추가한 배열을 만들고 State 변수 todo 값을 업데이트합니다.
5. TodoEditor 컴포넌트는 자연스러운 사용자 경험을 위해 할 일 입력 폼을 초기화합니다.

**아이템 추가 함수 만들기**

TodoEditor 컴포넌트에서 <추가> 버튼을 클릭하면 App에 사용자가 입력한 할 일 데이터를 전달하고 추가 이벤트가 발생했음을 알려야 합니다. 자식 컴포넌트에서 발생한 이벤트를 부모가 처리하는 것에 대해서는 프로젝트 1에서 잠깐 다뤄본 적 이 있습니다.

먼저 App 컴포넌트에서 새 할 일 아이템을 추가하는 함수 onCreate를 만듭니다.

**src/App.js**

**Copy**

(...)

function App() {

const [todo, setTodo] = useState(mockTodo);

const onCreate = (content) => { ①

const newItem = {

id: 0,

content,

isDone: false,

createdDate: new Date().getTime(),

};

setTodo([newItem, ...todo]); ②

};

return (

(...)

);

}

export default App;

**①** TodoEditor 컴포넌트에서 <추가> 버튼을 클릭하면 호출할 함수 onCreate를 만듭니다. 이 함수는 TodoEdior 컴포넌트에서 사용자가 작성한 할 일 데이터를 받아 매개변수 content에 저장합니다. 이 데이터를 토대로 새 할 일 아이템 객체를 만들어 newItem에 저장합니다.

**②** 배열의 스프레드 연산자를 활용해 newItem을 포함한 새 배열을 만들어 State 변수 todo를 업데이트합니다. 이렇게 작성하면 새롭게 추가된 아이템은 항상 배열의 0번 요소가 됩니다.

그런데 지금의 함수 onCreate에는 한 가지 문제점이 있습니다. 모든 아이템은 고유한 id를 가져야 하는데, 새롭게 추가할 아이템의 id가 모두 0으로 고정되기 때문입니다. 그럼 아이템을 추가할 때마다 중복 id가 만들어져 문제가 발생합니다.

Ref 객체를 사용하면 이 문제를 간단히 해결할 수 있습니다. Ref 객체는 앞에서 살펴본 적이 있는데, 리액트 훅인 함수 useRef로 생성합니다. Ref 객체는 리액트에서 주로 돔을 조작할 때 사용하지만, 컴포넌트의 변수로도 자주 활용합니다.

다음과 같이 App.js에서 새로운 Ref 객체를 생성합니다.

**src/App.js**

**Copy**

import { useState, useRef } from "react";

(...)

function App() {

const idRef = useRef(3);

(...)

}

export default App;

초기 값이 3인 Ref 객체를 생성해 idRef에 저장합니다.

이제 다음과 같이 idRef를 이용해 아이템을 생성할 때마다 id가 1씩 늘어나도록 수정합니다.

**src/App.js**

**Copy**

(...)

function App() {

(...)

const idRef = useRef(3);

const onCreate = (content) => {

const newItem = {

id: idRef.current, ①

content,

isDone: false,

createdDate: new Date().getTime(),

};

setTodo([newItem, ...todo]);

idRef.current += 1; ②

};

return (

(...)

);

}

export default App;

**①** idRef의 현잿값을 새롭게 추가할 할 일 아이템의 id로 지정합니다. 만약 아이템이 처음으로 추가되는 경우라면 해당 아이템의 id는 3이 됩니다.

**②** idRef의 현잿값을 1 늘립니다. 아이템을 추가할 때마다 idRef의 현잿값은 1씩 늘어납니다. 따라서 모든 아이템은 고유한 id를 갖게 됩니다. 참고로 idRef의 초깃값을 3으로 설정한 이유는 앞서 작성한 목 데이터의 id가 0, 1, 2이기 때문입니다.

App 컴포넌트에서 할 일 아이템을 생성하는 함수 onCreate를 모두 작성했습니다.

함수 onCreate는 사용자가 TodoEditor 컴포넌트에서 <추가> 버튼을 클릭해야 호출 되기 때문에 이 컴포넌트에 Props로 전달해야 합니다.

**src/App.js**

**Copy**

(...)

function App() {

(...)

return (

<div className="App">

<Header />

<TodoEditor onCreate={onCreate} />

<TodoList />

</div>

);

}

export default App;

**아이템 추가 함수 호출하기**

사용자가 할 일 입력 폼에서 아이템을 입력하고 <추가> 버튼을 클릭합니다.

그러면 TodoEditor 컴포넌트는 새 할 일을 생성하기 위해 App에서 Props로 받은 함수 onCreate를 호출하고 현재 사용자가 작성한 할 일을 인수로 전달합니다.

함수 onCreate를 사용하기 위해 다음과 같이 TodoEditor 컴포넌트를 수정합니다

**src/component/TodoEditor.js**

**Copy**

import "./TodoEditor.css";

const TodoEditor = ({ onCreate }) => { ①

return (

<div className="TodoEditor">

<h4>새로운 Todo 작성하기 🔏 </h4>

<div className="editor\_wrapper">

<input placeholder="새로운 Todo..." />

<button>추가</button>

</div>

</div>

);

};

export default TodoEditor;

**①** Props 객체를 구조 분해 할당합니다.

다음으로 TodoEditor 컴포넌트의 할 일 입력 폼에서 사용자가 입력하는 새 할 일 데이터를 저장할 State를 만듭니다.

✋🏼

TodoEditor에서 사용자가 할 일을 입력하고 <추가> 버튼을 클릭했을 때 전달되는 ‘할 일 데이터’는 리액트 컴포넌트 트리 구조에서 표현되는 데이터는 아닙니다. 컴포넌트 트리 구조에서 데이터를 전달한다는 의미는 여러 컴포넌트가 동시에 동일한 데이터를 이용한다는 뜻입니다. 따라서 변하는 값인 State는 부모에서 자식으로 Props를 이용해서만 전달할 수 있습니다. 버튼을 클릭하는 이벤트가 발생했을 때 인수로 전달하는 데이터는 컴포넌트 트리 구조상의 데이터 전달이 아닙니다. 일종의 ‘이벤트’가 전달된다고 생각하면 됩니다.

**src/component/TodoEditor.js**

**Copy**

import { useState } from "react";

import "./TodoEditor.css";

const TodoEditor = ({ onCreate }) => {

const [content, setContent] = useState(""); ①

const onChangeContent = (e) => { ②

setContent(e.target.value);

};

return (

<div className="TodoEditor">

<h4>새로운 Todo 작성하기 ✏ </h4>

<div className="editor\_wrapper">

<input ③

value={content}

onChange={onChangeContent}

placeholder="새로운 Todo..."

/>

<button>추가</button>

</div>

</div>

);

};

export default TodoEditor;

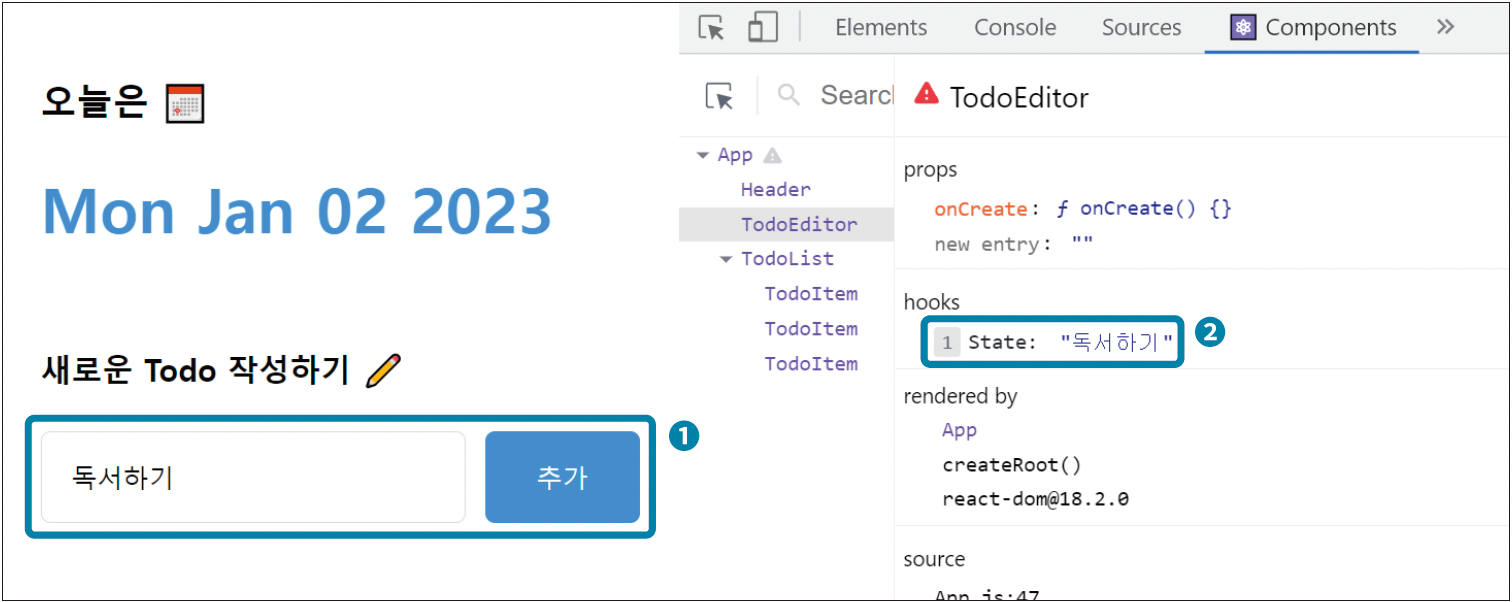
**①** 사용자가 입력 폼에 입력한 데이터를 저장할 State 변수 content를 만듭니다.

**②** 입력 폼의 onChange 이벤트 핸들러 onChangeContent를 만듭니다.

**③** 입력 폼의 value 속성으로 content 값을 설정하고, 이벤트 핸들러로 onChangeContent를 설정합니다.

개발자 도구의 [Components] 탭에서 실시간으로 State 값을 잘 반영하는지 확인합니다. 할 일 입력 폼에서 ‘독서하기’를 입력하고 개발자 도구의 [Components] 탭을 엽니다.

[Components] 탭에서 App의 TodoEditor를 클릭해 State(content)에 사용자가 지금 입력한 내용이 제대로 반영되는지 확인합니다.

[Components] 탭에서 할 일 아이템 생성 확인하기

사용자가 입력한 값이 실시간으로 content에 반영되고 있음을 알 수 있습니다.

다음으로 <추가> 버튼을 클릭하면, 함수 onCreate를 호출하는 버튼 클릭 이벤트 핸들러를 만듭니다.

**src/component/TodoEditor.js**

**Copy**

(...)

const TodoEditor = ({ onCreate }) => {

(...)

const onSubmit = () => { ①

onCreate(content);

};

return (

<div className="TodoEditor">

<h4>새로운 Todo 작성하기 ✏ </h4>

<div className="editor\_wrapper">

<input

value={content}

onChange={onChangeContent}

placeholder="새로운 Todo..."

/>

<button onClick={onSubmit}>추가</button> ②

</div>

</div>

);

};

export default TodoEditor;

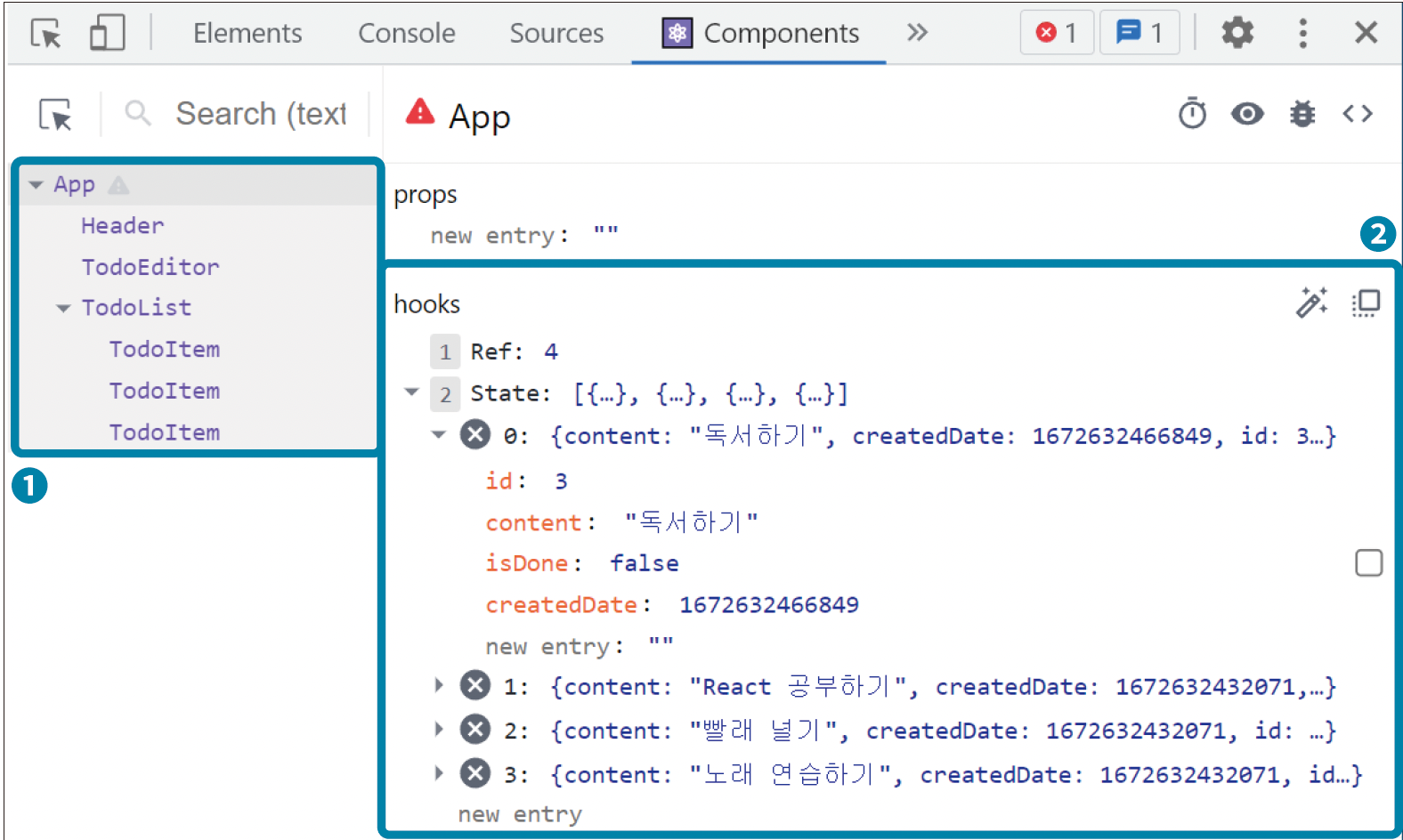
**①** <추가> 버튼에 대한 이벤트 핸들러 onSubmit을 생성합니다. onSubmit은 함수 onCreate를 호출하고 인수로 content의 값을 전달합니다.

**②** 버튼 클릭 이벤트 핸들러로 함수 onSubmit을 설정합니다.

이제 새 할 일을 작성하고 <추가> 버튼을 클릭해, App 컴포넌트의 todo에 새 아이템이 잘 추가되는지 확인합니다. 아직 App 컴포넌트의 todo 값을 페이지에 렌더링하는 Read 기능은 개발하지 않았기 때문에 개발자 도구의 [Components] 탭에서 직접 확인해야 합니다.

새 할 일 아이템으로 ‘독서하기’를 입력하고 <추가> 버튼을 클릭합니다.

[Components] 탭에서 App를 클릭해 State를 확인하면 ‘독서하기’ 아이템이 추가되어 있습니다. 다음 그림 처럼 나온다면 정상적으로 동작하는 겁니다.

[Components] 탭에서 할 일 아이템을 제대로 추가하는지 확인

이렇게 [할 일 관리] 앱에서 새 아이템을 추가하는 Create 기능을 만들었습니다. 새롭게 추가한 아이템은 App 컴포넌트의 todo 배열 맨 앞에 추가됩니다. 그 이유는 앞서 함수 onCreate를 만들 때 새로 추가할 아이템은 배열의 0번 인덱스에 위치하도록 만들었기 때문입니다.

**Create 완성도 높이기**

지금까지 만든 Create 기능은 아이템을 잘 추가하고, id도 잘 배정하지만 완성도를 높이기 위해서는 몇 가지 해야 할 작업이 있습니다.

**빈 입력 방지하기**

프로그램의 완성도를 높이기 위해 Create 기능을 좀 더 개선하겠습니다.

처음 할 일은 빈 입력을 방지하는 일입니다. 빈 입력은 말뜻 그대로 아무것도 입력하지 않은 상태에서 <추가> 버튼을 누르는 행위입니다.

아무것도 입력하지 않은 상태로 아이템을 추가하는 것을 방지하기 위한 방법은 여럿 있습니다. 이 책에서는 웹 서비스들이 일반적으로 채택하고 있는 빈 입력란에 포커스를 주는 기능을 구현합니다.

특정 페이지 요소에 포커스를 주는 방법은 Ref 객체를 소개할 때 잠시 살펴본 적이 있습니다. 할 일 입력 폼을 관리할 Ref 객체를 하나 만들고, 함수 onSubmit에서 content 값이 비어 있으면 입력 폼에 포커스를 구현하는 방식입니다.

다음과 같이 TodoEditor 컴포넌트를 수정합니다.

**src/component/TodoEditor.js**

**Copy**

import { useState, useRef } from "react";

import "./TodoEditor.css";

const TodoEditor = ({ onCreate }) => {

const [content, setContent] = useState("");

const inputRef = useRef(); ①

(...)

const onSubmit = () => {

if (!content) { ②

inputRef.current.focus();

return;

}

onCreate(content);

};

return (

<div className="TodoEditor">

<h4>새로운 Todo 작성하기 ✏ </h4>

<div className="editor\_wrapper">

<input

ref={inputRef} ③

value={content}

onChange={onChangeContent}

placeholder="새로운 Todo..."

/>

<button onClick={onSubmit}>추가</button>

</div>

</div>

);

};

export default TodoEditor;

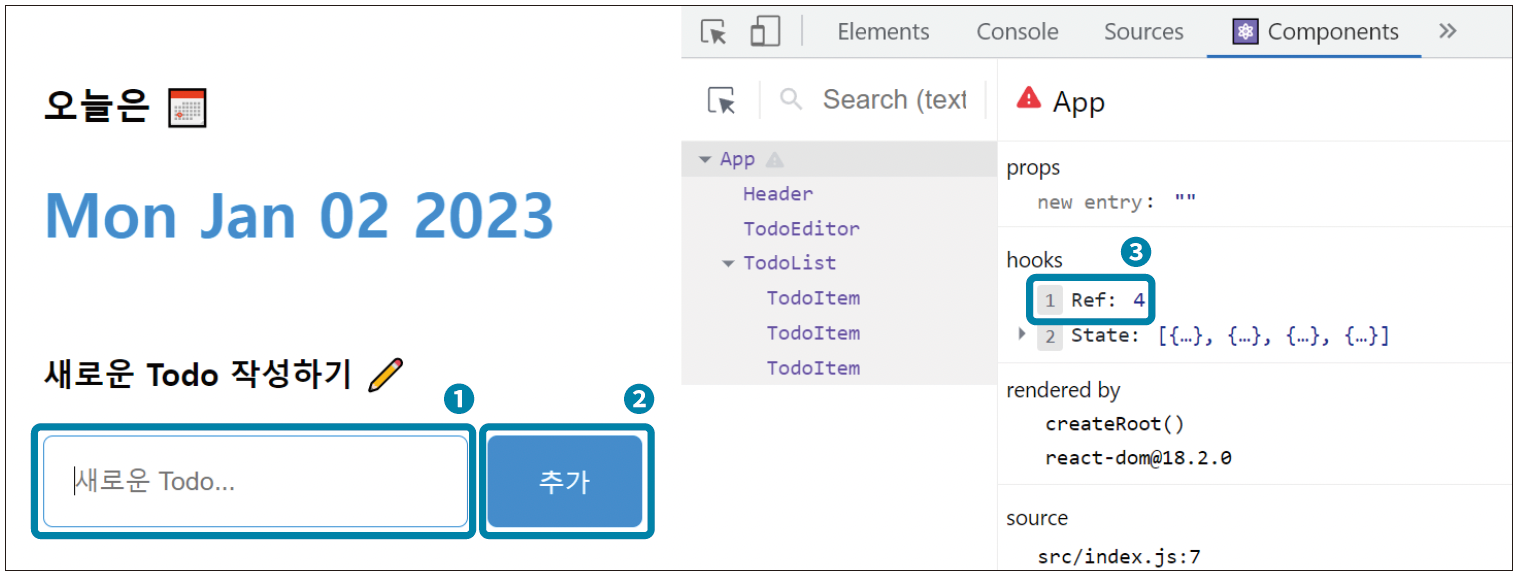
**①** 할 일 입력 폼을 제어할 객체 inputRef를 생성합니다.

**②** 함수 onSubmit은 현재 content 값이 빈 문자열이면, inputRef가 현잿값(current)으로 저장한 요소에 포커스하고 종료합니다.

**③** 할 일 입력 폼의 ref에 inputRef를 설정합니다. 이제 inputRef는 현잿값으로 이 요소를 저장합니다.

이제 입력 폼에서 아무것도 입력하지 않고 <추가> 버튼을 클릭하면, 입력 폼은 아이템을 추가하지 않고 포커스 상태로 멈춰 있게 됩니다. 아이템의 추가 여부를 확인하려면 개발자 도구 [Components] 탭에서 App를 선택한 다음, <추가> 버튼을 클릭해도 Ref 객체에 저장된 id 값이 증가하지 않는지 확인하면 됩니다.

입력 폼에서 아무것도 입력하지 않은 상태에서 <추가> 버튼을 클릭해 제대로 동작 하는지 확인합니다.

입력 폼에서 빈 입력 방지하기

<추가> 버튼을 클릭해도 빈 입력 상태에서는 입력 폼에 아무런 변화도 일어나지 않고 커서만 깜빡입니다. Ref의 값도 4로 변화가 없습니다.

**아이템 추가 후 입력 폼 초기화하기**

프로그램의 완성도를 높이기 위한 두 번째 조치로 아이템을 추가하면 자동으로 할 일 입력 폼을 초기화하는 기능을 구현합니다.

입력 폼을 초기화하는 기능이 없다면 의도치 않게 <추가> 버튼을 클릭해 중복 아이템을 생성할 수 있습니다. 또한 다른 할 일을 추가하려는 사용자에게는 기존에 작성했던 항목이 지워지지 않고 남아 있어 이를 지워야 하는 불편함이 생깁니다.

TodoEditor 컴포넌트의 함수 onSubmit을 다음과 같이 수정합니다.

**src/component/TodoEditor.js**

**Copy**

(...)

const TodoEditor = ({ onCreate }) => {

(...)

const onSubmit = () => {

if (!content) {

inputRef.current.focus();

return;

}

onCreate(content);

setContent(""); ①

};

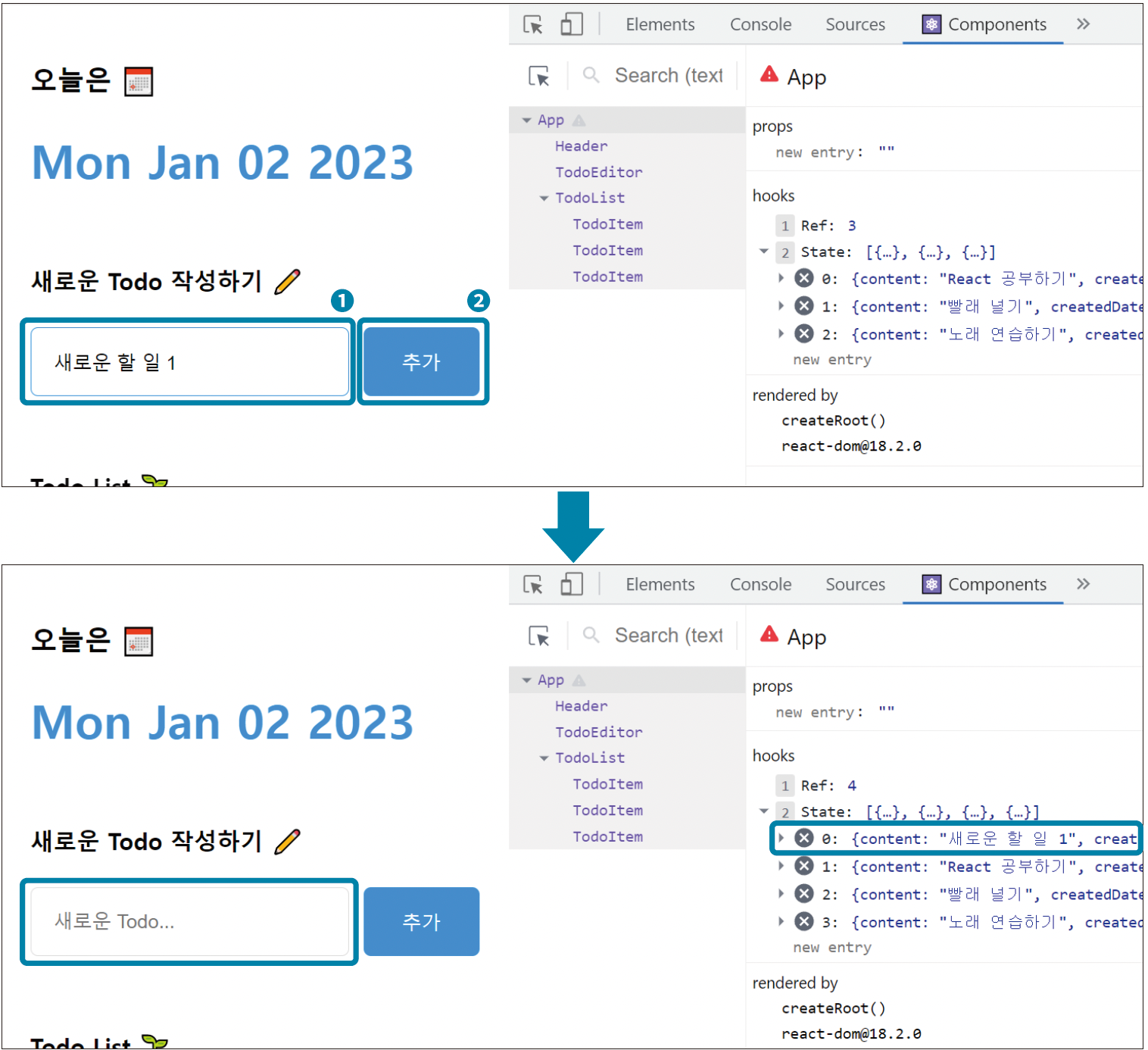
(...)

};

export default TodoEditor;

**①** 함수 setContent를 호출해 인수로 빈 문자열을 전달합니다. 그러면 새 아이템을 추가하고 난 후, content 값은 빈 문자열이 되고 입력 폼 역시 초기화됩니다.

[할 일 관리] 앱에서 새로고침(<F5>) 버튼을 클릭해 페이지를 초기화합니다. 임의로 새 할 일을 하나 입력한 다음, <추가> 버튼을 클릭합니다. [Components] 탭에서 데이터가 생성되는지 그리고 페이지의 입력 폼은 잘 초기화되는지 확인합니다.

입력 폼 초기화하기

새로운 할 일 아이템이 추가되는 것과 동시에 입력 폼 역시 초기화되는 것을 알 수 있습니다.

✋🏼

[할 일 관리] 앱 페이지에서 새로고침(<F5>)하면 기존에 입력했던 데이터(예를 들어 독서하기)들은 모두 사라지고 처음 입력한 목 데이터만 다시 렌더링합니다. 여러분의 결과가 이 책의 결과와 다르다면 새로고침하여 추가로 입력한 값들은 모두 삭제하고 새 할 일을 입력해 추가하면 됩니다.

**<Enter> 키를 눌러 아이템 추가하기**

프로그램의 완성도를 높이기 위해 마지막으로 키보드의 <Enter> 키를 누르면, <추가> 버튼을 클릭한 것과 동일한 동작을 수행하는 키 입력 이벤트를 만들겠습니다.

TodoEditor 컴포넌트를 다음과 같이 수정합니다.

**src/component/TodoEditor.js**

**Copy**

(...)

const TodoEditor = ({ onCreate }) => {

(...)

const onKeyDown = (e) => { ①

if (e.keyCode === 13) {

onSubmit();

}

};

return(

<div className="TodoEditor">

<h4>새로운 Todo 작성하기 ✏ </h4>

<div className="editor\_wrapper">

<input

ref={inputRef}

value={content}

onChange={onChangeContent}

onKeyDown={onKeyDown} ②

placeholder="새로운 Todo..."

/>

<button onClick={onSubmit}>추가</button>

</div>

</div>

)

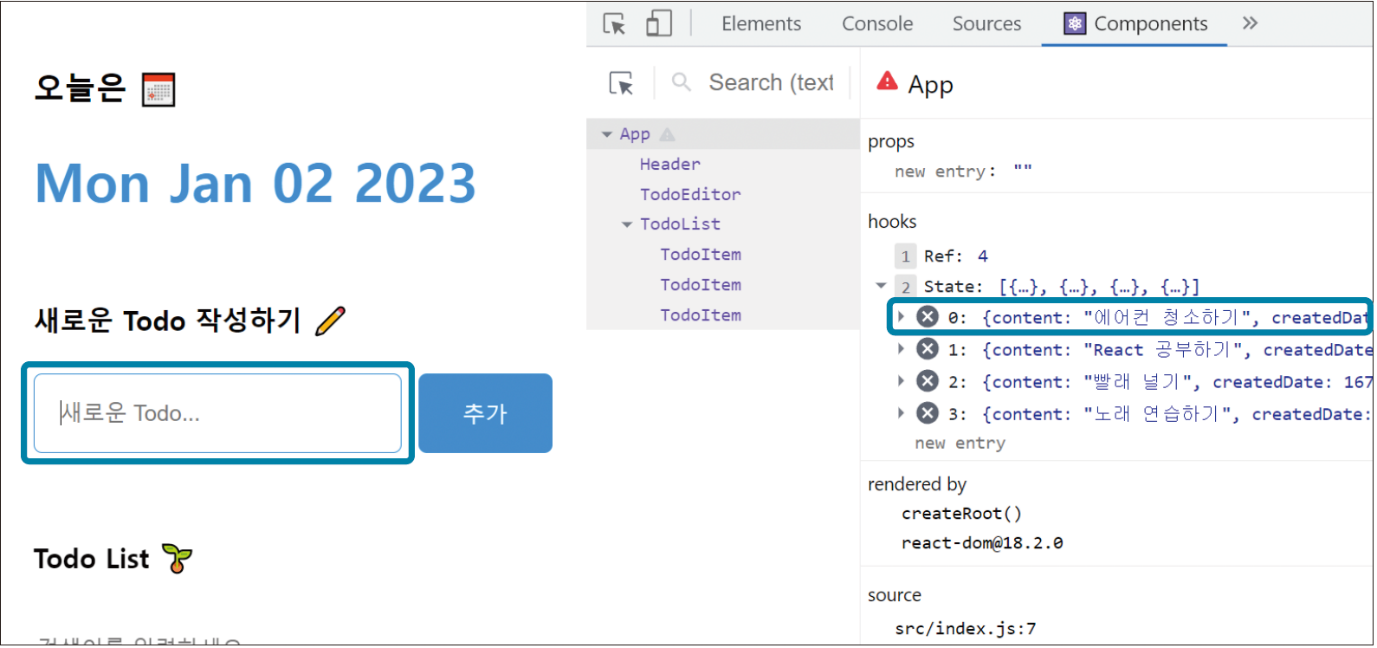
};

export default TodoEditor;

**①** 함수 onKeyDown은 사용자가 <Enter> 키를 눌렀을 때 호출할 이벤트 핸들러입니다. e.keyCode 에는 현재 사용자가 누른 키보드의 키가 숫자로 변환되어 저장되어 있는데, 13은 <Enter> 키를 의미합니다. 따라서 e.keyCode가 13이면 함수 onSubmit을 호출합니다.

**②** 입력 폼의 키 입력 이벤트 핸들러를 함수 onKeyDown으로 설정합니다.

새로고침한 다음, 새로운 할 일 ‘에어컨 청소하기’를 입력하고 <Enter> 키를 눌러 아이템이 잘 추가되는지 확인합니다. 개발자 도구의 [Components] 탭에서 App 컴포넌트의 State 값을 확인하면 됩니다.

<Enter> 키로 아이템 생성하기

이것으로 [할 일 관리] 앱의 Create 기능을 모두 완성하였습니다.