**5. Read: 할 일 리스트 렌더링하기**

📄

**Table Of Contents**

이번에는 TodoList 컴포넌트의 기능이면서 CRUD의 두 번째 요소인 Read 기능을 만들겠습니다. Read 기능을 이용하면 배열에 저장한 여러 할 일 아이템을 반복해서 페이지에 렌더링할 수 있습니다.

**배열을 리스트로 렌더링하기**

App 컴포넌트의 State 변수 todo에는 배열 형태로 여러 개의 할 일 아이템이 저장되어 있습니다. 배열 todo를 TodoList 컴포넌트에 Props로 전달합니다.

**src/App.js**

**Copy**

(...)

function App() {

(...)

return (

<div className="App">

<Header />

<TodoEditor onCreate={onCreate} />

<TodoList todo={todo} />

</div>

);

}

export default App;

TodoList 컴포넌트에서는 App에서 Props로 전달된 todo를 리스트로 렌더링해야 합니다. 리액트에서 배열 데이터를 렌더링할 때는 배열 메서드 map을 주로 이용합니다. map을 이용하면 HTML 또는 컴포넌트를 순회하면서 매 요소를 반복하여 렌더링합니다.

**map을 이용한 HTML 반복하기**

TodoList 컴포넌트에서 배열 메서드 map을 이용해 HTML 요소를 반복해 렌더링합니다. TodoList 컴포넌트를 다음과 같이 수정합니다.

**src/component/TodoList.js**

**Copy**

import TodoItem from "./TodoItem";

import "./TodoList.css";

const TodoList = ({ todo }) => { ①

return (

<div className="TodoList">

<h4>Todo List 🌱</h4>

<input className="searchbar" placeholder="검색어를 입력하세요" />

<div className="list\_wrapper">

{todo.map((it) => ( ②

<div>{it.content}</div>

))}

</div>

</div>

);

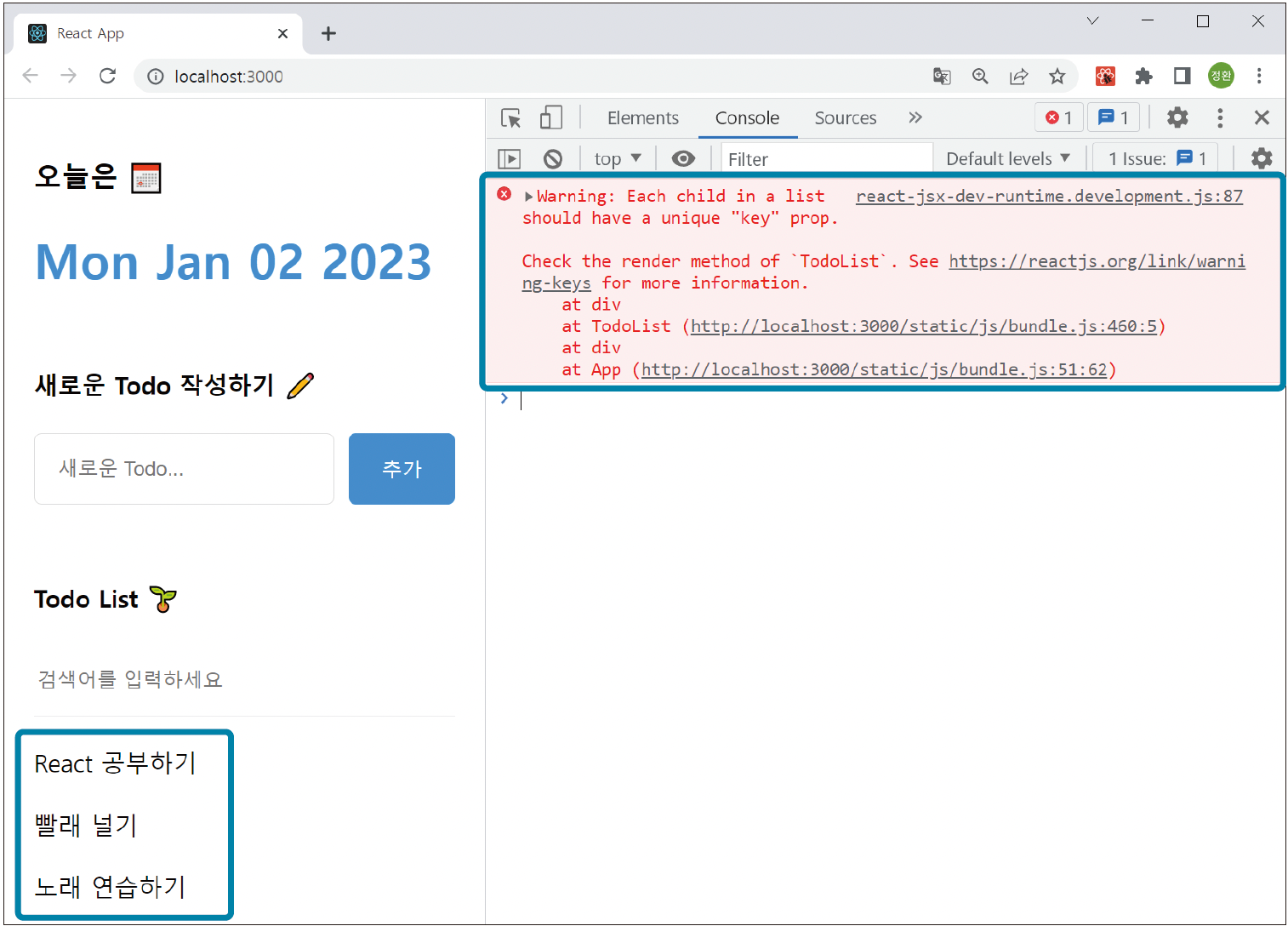
};

export default TodoList;

**①** Props를 구조 분해 할당합니다.

**②** map 메서드를 이용해 배열 todo의 모든 요소를 순차적으로 순회하며 HTML로 변환합니다. 이 식의 결괏값은 배열 todo에 저장된 모든 할 일을 <div> 태그로 감싼 것과 동일합니다.

State 변수 todo를 초기화하기 위해 페이지를 새로고침(<F5>)하고 렌더링 결과를 확인합니다.

map 메서드로 HTML 렌더링하기

todo에 저장된 3개의 할 일을 HTML로 반복해 페이지에 렌더링합니다. 이때 개발자 도구의 콘솔에 **“Each child in a list should have a unique key prop”**이라는 경고 메시지가 출력되는데, 이 메시지에 대해서는 뒤에서 자세히 다루겠습니다.

**map을 이용해 컴포넌트 반복하기**

이번에는 map 메서드의 콜백 함수가 HTML이 아닌 컴포넌트를 반환하도록 수정하겠습니다. 배열을 이용해 컴포넌트를 반복해 렌더링합니다.

**src/component/TodoList.js**

**Copy**

import TodoItem from "./TodoItem";

import "./TodoList.css";

const TodoList = ({ todo }) => {

return (

<div className="TodoList">

<h4>Todo List 🌱</h4>

<input className="searchbar" placeholder="검색어를 입력하세요" />

<div className="list\_wrapper">

{todo.map((it) => (

<TodoItem {...it} /> ①

))}

</div>

</div>

);

};

export default TodoList;

**①** map 메서드의 콜백 함수가 TodoItem 컴포넌트를 반환합니다. 이때 TodoItem 컴포넌트에 현재 순회 중인 배열 요소 it의 모든 프로퍼티를 스프레드 연산자를 이용해 Props로 전달합니다. 배열 todo에는 할 일 아이템 객체가 저장되어 있기 때문에 결과적으로 TodoItem 컴포넌트에는 이 객체 각각의 프로퍼티가 Props로 전달됩니다.

TodoItem 컴포넌트에 전달된 Props를 이 컴포넌트에서 사용할 수 있도록 다음과 같이 수정합니다.

✋🏼

map 메서드의 매개변수 it는 item을 줄여 쓴 겁니다.

**src/component/TodoItem.js**

**Copy**

import "./TodoItem.css";

const TodoItem = ({ id, content, isDone, createdDate }) => { ①

return (

<div className="TodoItem">

<div className="checkbox\_col">

<input checked={isDone} type="checkbox" /> ②

</div>

<div className="title\_col">{content}</div> ③

<div className="date\_col">

{new Date(createdDate).toLocaleDateString()} ④

</div>

<div className="btn\_col">

<button>삭제</button>

</div>

</div>

);

};

export default TodoItem;

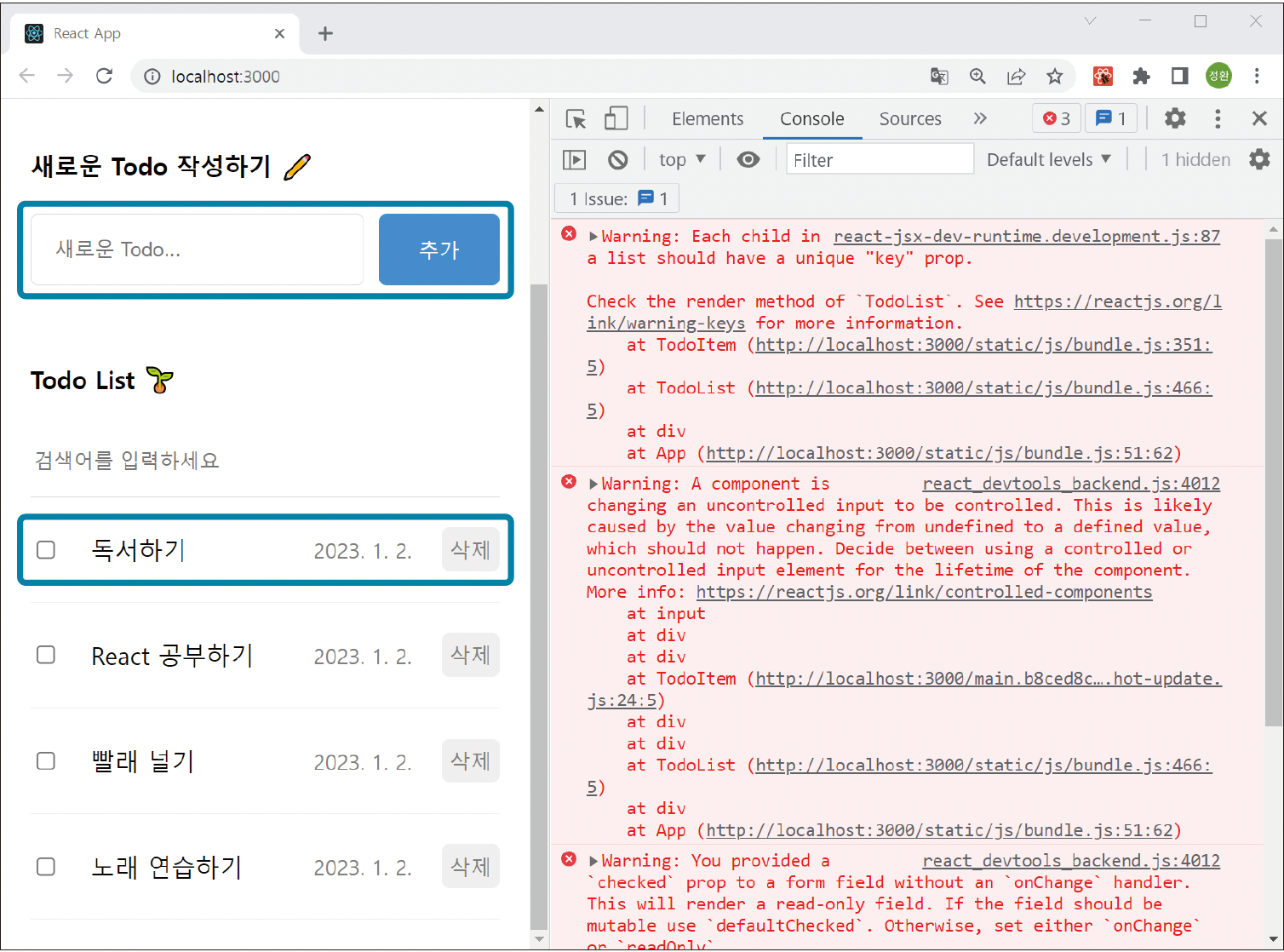
**①** Props를 구조 분해 할당합니다.

**②** 체크박스 입력 폼의 체크 여부를 isDone으로 설정합니다.

**③** 할 일을 페이지에 표시하기 위해 content를 렌더링합니다.

**④** 앞서 목 데이터를 설정할 때 createdDate를 타임 스탬프값으로 저장했습니다. new Date로 새로운 객체를 만들고, 생성자의 인수로 createDate를 전달해 타임 스탬프값을 Date 형식으로 변환합니다. 그다음 toLocaleDateString 메서드를 사용해 문자열로 변환해 렌더링합니다.

저장한 다음, 결과를 확인할 수 있도록 할 일 입력 폼에 ‘독서하기’라는 새 아이템을 추가합니다.

새 아이템 렌더링하기

목 데이터의 할 일 아이템들과 새로 입력한 아이템을 잘 렌더링합니다. 그러나 개발자 도구의 콘솔을 열면 여러 가지 경고 메시지가 출력되는 걸 볼 수 있습니다. 앞서 확인했던 것과 같은 경고 메시지입니다.

**Each child in a list should have a unique "key" prop.**

경고 메시지를 직역하면 “리스트의 모든 자식 요소는 key라는 고유한 prop을 반드시 가져야 한다”라고 해석할 수 있습니다. 그리고 다음과 같은 두 번째 경고 메시지도 발견할 수 있습니다.

**You provided a 'checked' prop to a form without an 'onChange' handler …**

이 메시지는 TodoItem 컴포넌트가 체크박스 입력 폼에 onChange 이벤트 핸들러를 설정하지 않아서 발생한 경고입니다. 나중에 이 체크박스에 onChange 이벤트 핸들러를 설정할 예정이므로 지금은 무시해도 됩니다.

**key 설정하기**

key는 리스트에서 각각의 컴포넌트를 구분하기 위해 사용하는 값입니다. 리액트는 리스트에서 특정 컴포넌트를 수정, 추가, 삭제하는 경우, 이 key로 어떤 컴포넌트를 업데이트할지 결정합니다. 따라서 리스트의 각 컴포넌트를 key로 구분하지 않으면 생성, 수정, 삭제와 같은 연산을 수행할 수 없거나 비효율적으로 탐색하게 됩니다. 심지어 성능이 나빠지거나 의도치 않은 동작을 수행할 수 있습니다.

그렇다면 무엇을 key로 사용하는 게 좋을까요? 우리는 이미 아이템마다 고유한 id를 갖도록 데이터를 모델링했습니다. 그리고 App 컴포넌트의 할 일 아이템 생성 과정에서 Ref 객체를 이용해 아이템마다 고유 id를 갖도록 만들었습니다. 따라서 id를 key로 전달하면 문제를 간단히 해결할 수 있습니다.

TodoList 컴포넌트를 다음과 같이 수정합니다.

**src/component/TodoItem.js**

**Copy**

import TodoItem from "./TodoItem";

import "./TodoList.css";

const TodoList = ({ todo }) => {

return (

<div className="TodoList">

(...)

<div className="list\_wrapper">

{todo.map((it) => (

<TodoItem key={it.id} {...it} /> ①

b ))}

</div>

</div>

);

};

export default TodoList;

**①** 리스트의 각 컴포넌트에 key로 할 일 아이템의 id를 전달합니다.

이제 개발자 도구의 [콘솔] 탭을 다시 확인해 보면 key와 관련해서는 더 이상 오류가 발생하지 않습니다.

정리하면 map을 이용해 컴포넌트를 리스트 형태로 반복적으로 렌더링하려면 반드시 리스트 내의 고유한 key를 Props로 전달해야 합니다.

**검색어에 따라 필터링하기**

TodoList 컴포넌트에서 특정 할 일을 검색하는 기능을 만들겠습니다.

**검색 기능 만들기**

이번에는 TodoList의 검색 폼에서 검색어를 입력하면, 해당 문자열을 포함하는 할 일 아이템만 필터링해 보여주는 기능을 구현합니다.

먼저 사용자가 입력하는 검색어를 처리할 State 변수를 만든 다음, 검색 폼에서 사용자가 입력한 내용을 처리하는 기능을 만듭니다. TodoList.js를 다음과 같이 수 정합니다.

**src/component/TodoList.js**

**Copy**

import { useState } from "react"; *// ①*

(...)

const TodoList = ({ todo }) => {

const [search, setSearch] = useState("");

const onChangeSearch = (e) => { *// ②*

setSearch(e.target.value);

};

return (

<div className="TodoList">

<h4>Todo List 🌱</h4>

<input

value={search} *// ③*

onChange={onChangeSearch} *// ④*

className="searchbar"

placeholder="검색어를 입력하세요"

/>

<div className="list\_wrapper">

{todo.map((it) => (

<TodoItem key={it.id} {...it} />

))}

</div>

</div>

);

};

export default TodoList;

**①** react 라이브러리에서 useState 리액트 훅을 불러옵니다.

**②** 검색 폼의 onChange 이벤트 핸들러 onChangeSearch를 만듭니다.

**③** 검색 폼의 value로 State 변수 search를 설정합니다.

**④** 검색 폼의 onChange 이벤트 핸들러를 onChangeSearch로 설정합니다.

계속해서 사용자가 입력한 검색어에 따라 할 일 아이템을 필터링하는 기능을 만듭니다.

**src/component/TodoList.js**

**Copy**

(...)

const TodoList = ({ todo }) => {

(...)

const getSearchResult = () => {

return search === ""

? todo

: todo.filter((it) => it.content.includes(search));

};

return (

<div className="TodoList">

(...)

<div className="list\_wrapper">

{getSearchResult().map((it) => ( ②

<TodoItem key={it.id} {...it} />

))}

</div>

</div>

);

};

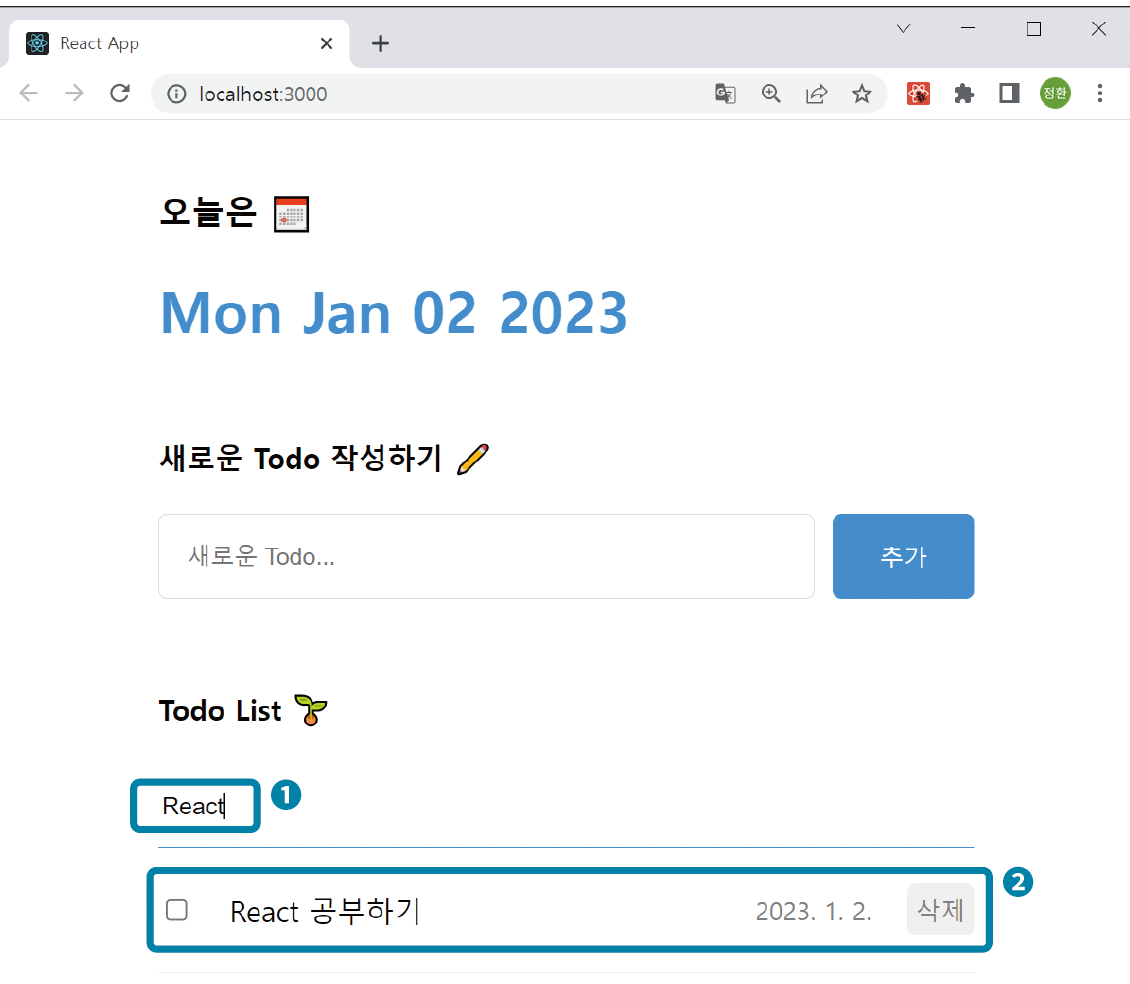
export default TodoList;

**①** 함수 getSearchResult는 현재 입력한 검색어인 search가 빈 문자열("")이면 todo를 그대로 반환하고, 그렇지 않으면 todo 배열에서 search의 내용과 일치하는 아이템만 필터링해 반환합니다.

**②** 함수 getSearchResult의 결괏값을 map 메서드를 이용해 리스트로 렌더링합니다.

이제 다 되었습니다. 검색어로 ‘React’를 입력하여 검색 결과가 잘 나타나는지 확인합니다.

검색어 ‘React’를 입력하니 ‘React 공부하기’ 아이템만 페이지에 렌더링하는 것을 볼 수 있습니다. 검색 기능이 잘 구현되었습니다. 한 가지 아쉬운 점은 검색어를 React가 아니라 ‘react’로 입력하면 대소 문자를 구별하지 못해 검색 결과가 나타나지 않습니다.

TodoList 컴포넌트의 검색 기능 만들기

**대소 문자를 구별하지 않게 하기**

이번에는 검색에서 대소 문자를 구별하지 않도록 기능을 업그레이드하겠습니다. 그럼 사용자가 더 쉽게 검색할 수 있습니다

**src/component/TodoList.js**

**Copy**

(...)

const getSearchResult = () => {

return search === ""

? todo

: todo.filter((it) =>

it.content.toLowerCase().includes(search.toLowerCase()) ①

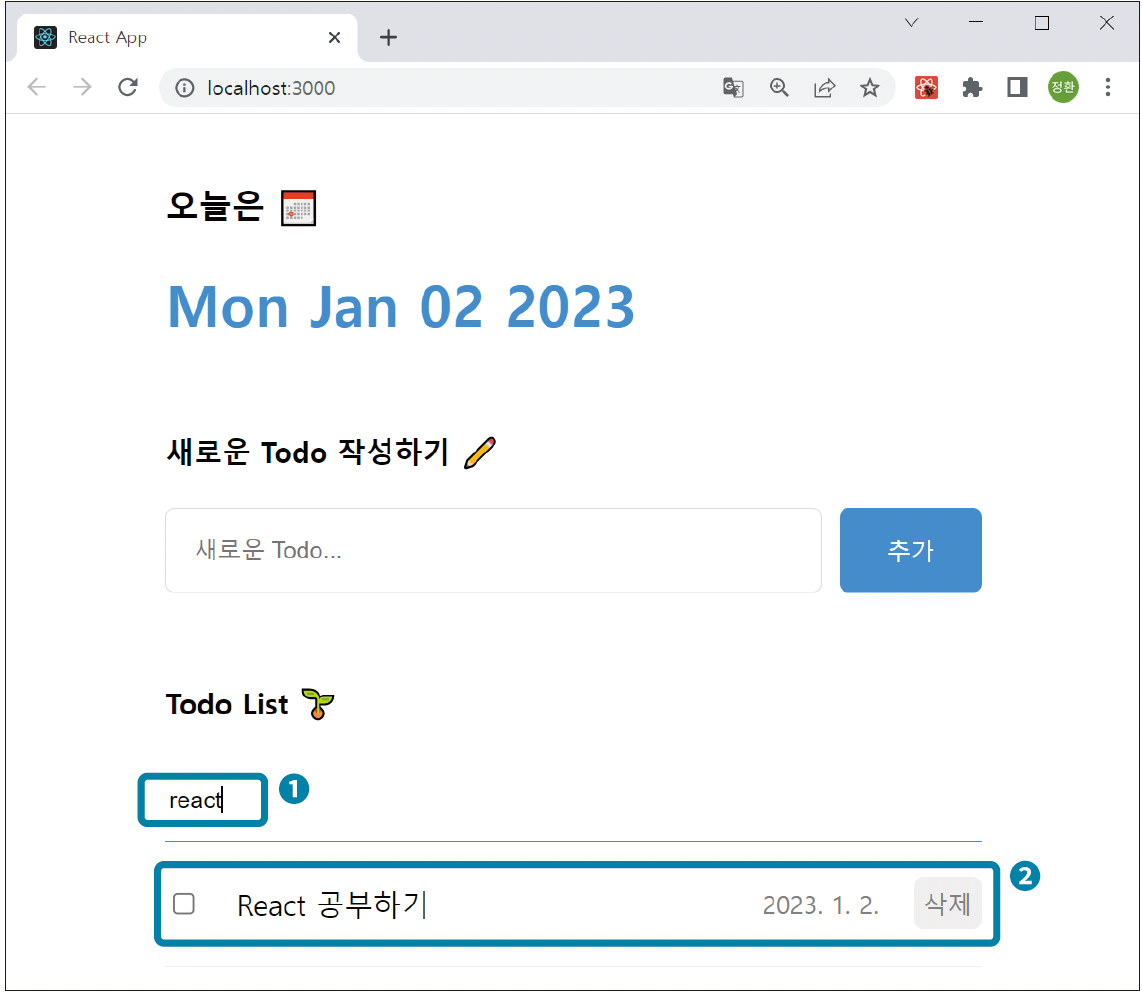
);

};

(...)

**①** toLowerCase() 메서드는 문자열에 있는 대문자를 모두 소문자로 바꿔 줍니다. toLowerCase 메서드를 이용해 검색어(search)와 todo 아이템의 content 를 모두 소문자로 바꾸면 대소 문자를 구별하지 않고 검색합니다.

이제 대소 문자를 구별하지 않기 때문에 react로 검색해도 React 공부하기 아이템이 검색 결과에 잘 나타납니다.

검색 기능에서 대소 문자 구별하지 않기