2: [할 일 관리] 앱 만들기

이번 프로젝트에서는 간단한 [할 일 관리] 앱을 만들어 보겠습니다.

이 앱은 오늘의 날짜를 표시하고, 할 일을 작성하고, 할 일 리스트를 관리할 수 있는 기능을 포함하고 있습니다.

프로젝트는 크게 준비, UI 구현, 기능 구현의 단계로 나누어 진행합니다.

목차

---

1. 프로젝트 준비하기

요구사항 분석하기

먼저 앱의 요구사항을 분석합니다.

다음은 [할 일 관리] 앱의 최종 구현 모습입니다.

기능 목록:

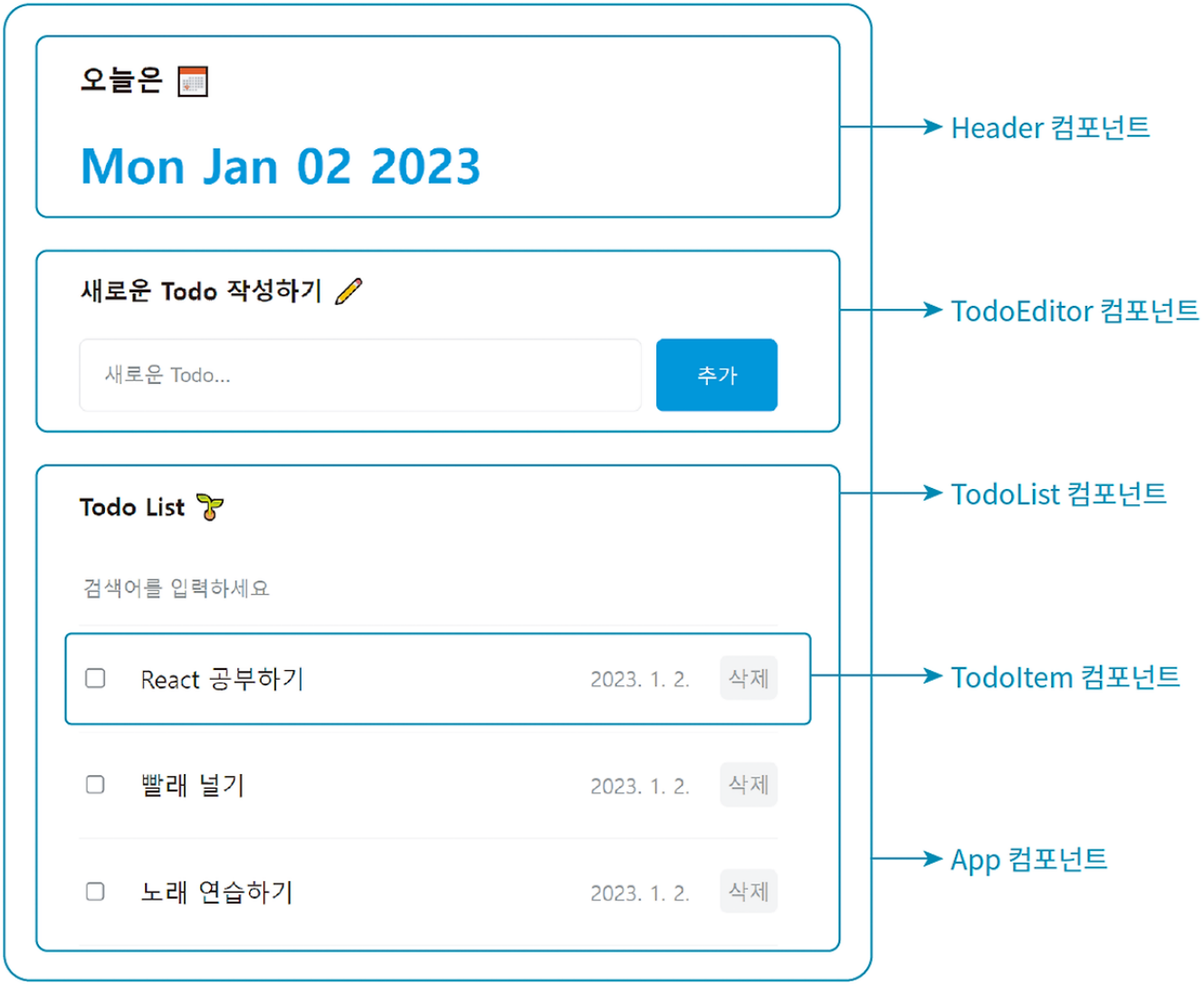
- 오늘의 날짜를 요일, 월, 일, 연도순으로 표시
- 할 일(Todo) 작성 입력 폼 및 추가 버튼
- 할 일 리스트 및 검색 기능
- 할 일 아이템 체크박스, 등록 날짜, 삭제 버튼

컴포넌트 구조

앱의 UI 요소를 컴포넌트 단위로 나눕니다.

컴포넌트를 나누는 일은 UI 요소를 역할별로 구분하는 데 중요합니다.

다음은 [할 일 관리] 앱의 컴포넌트 구조입니다.



컴포넌트 목록:

- **Header:** 오늘의 날짜를 표시합니다.
- **TodoEditor:** 새로운 할 일 아이템을 등록합니다.
- **TodoList:** 검색어에 맞게 필터링된 할 일 리스트를 렌더링합니다.
- **TodoItem:** 할 일 아이템의 정보를 표시하고, 체크박스 및 삭제 버튼을 포함합니다.

리액트 앱 만들기

새로운 리액트 앱을 생성하고 불필요한 파일을 삭제합니다.

npx create-react-app project2

cd project2

다음 파일들을 삭제합니다:

- src/App.test.js
- src/logo.svg
- src/reportWebVitals.js
- src/setupTests.js

App.js와 index.js 파일을 아래와 같이 수정합니다.

src/App.js

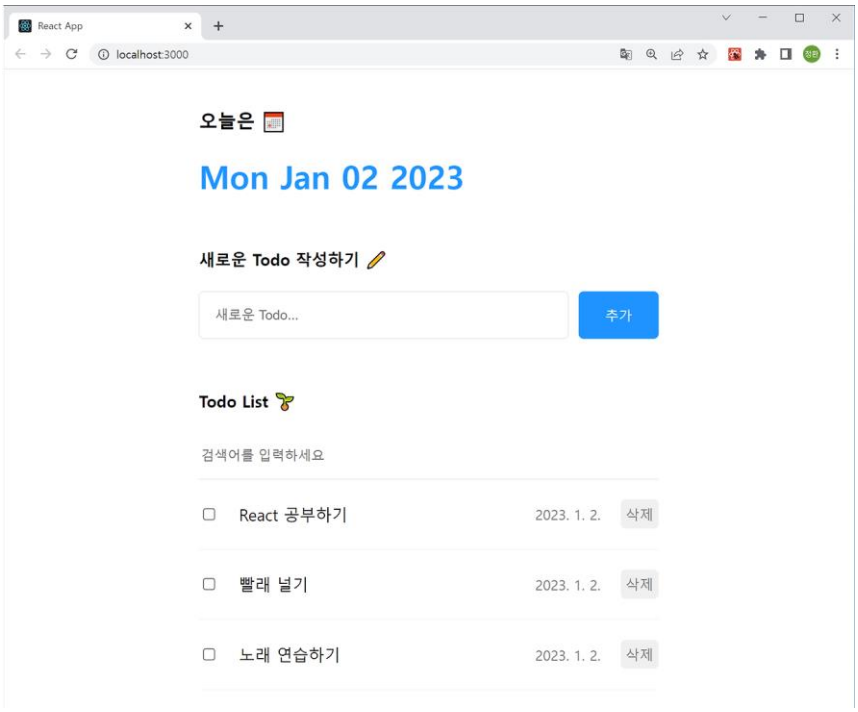
```
import "./App.css";


function App() {
  return <div className="App"></div>;
}


export default App;
```

src/index.js

```
import React from "react";
import ReactDOM from "react-dom/client";
import "./index.css";
```

```
import App from "./App";


const root = ReactDOM.createRoot(document.getElementById("root"));

root.render(<App />);
```

2. UI 구현하기



페이지 레이아웃 만들기

먼저 페이지 레이아웃을 구성합니다. App.js에 임시 내용을 추가합니다.

src/App.js

```
import "./App.css";


function App() {
  return (
    <div className="App">
```

```
        <h2>헬로 리액트</h2>
      </div>
    );
}

export default App;
```

index.css와 App.css에 기본 스타일을 추가합니다.

src/index.css

```
body {
  margin: 0px;
}
```

src/App.css

```
.App {
  max-width: 500px;
  width: 100%;
  margin: 0 auto;
  box-sizing: border-box;
  padding: 20px;
  border: 1px solid gray;
  display: flex;
  flex-direction: column;
```

```
  gap: 30px;
}
```

개발자 도구에서 요소의 flex 속성을 확인하여 UI 요소가 잘 배치되었는지 확인합니다.

Header 컴포넌트 만들기

Header 컴포넌트를 생성합니다.

src/component/Header.js

```
import "./Header.css";

const Header = () => {
  return (
    <div className="Header">
      <h3>오늘은 🗓</h3>
      <h1>{new Date().toDateString()}</h1>
    </div>
  );
};

export default Header;
```

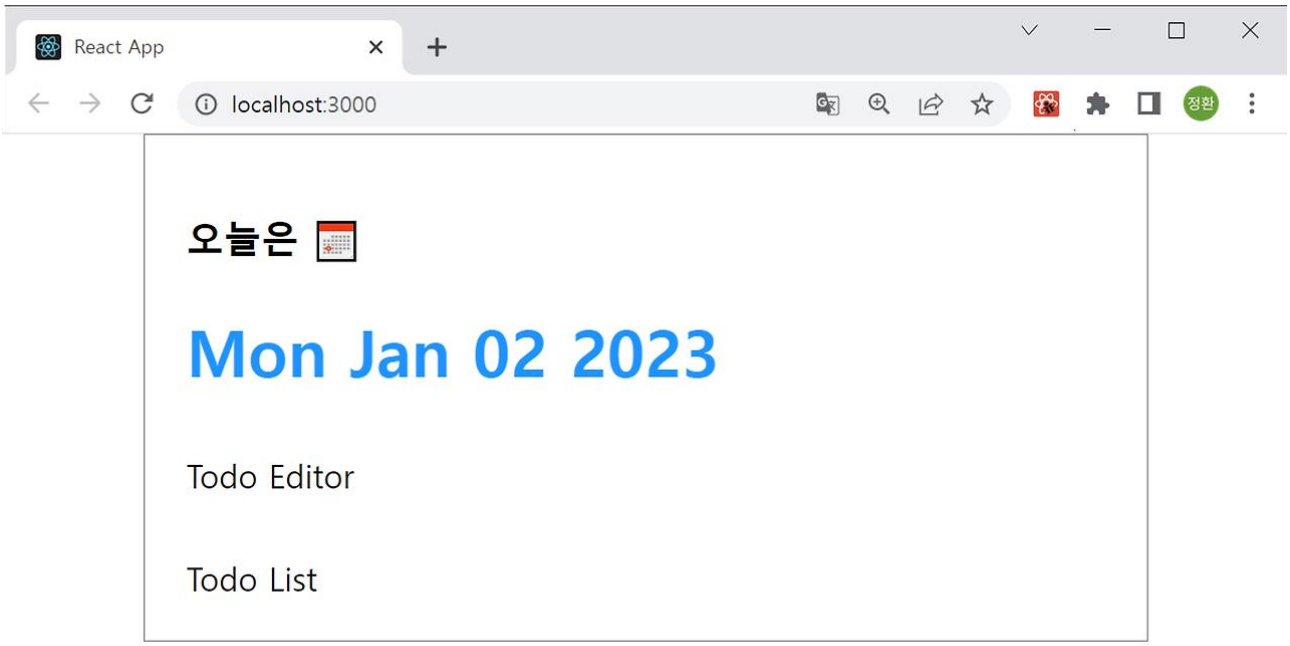
src/component/Header.css

```
.Header h1 {
```

```
    margin-bottom: 0px;
    color: #1f93ff;
}
```

App.js에 Header 컴포넌트를 추가합니다.

src/App.js



TodoEditor 컴포넌트 만들기

TodoEditor 컴포넌트를 생성합니다.

src/component/TodoEditor.js

```
import "./TodoEditor.css";

const TodoEditor = () => {
    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;
```

src/component/TodoEditor.css

```
.TodoEditor .editor_wrapper {
  width: 100%;
  display: flex;
  gap: 10px;
}

.TodoEditor input {
  flex: 1;
  box-sizing: border-box;
  border: 1px solid rgb(220, 220, 220);
```

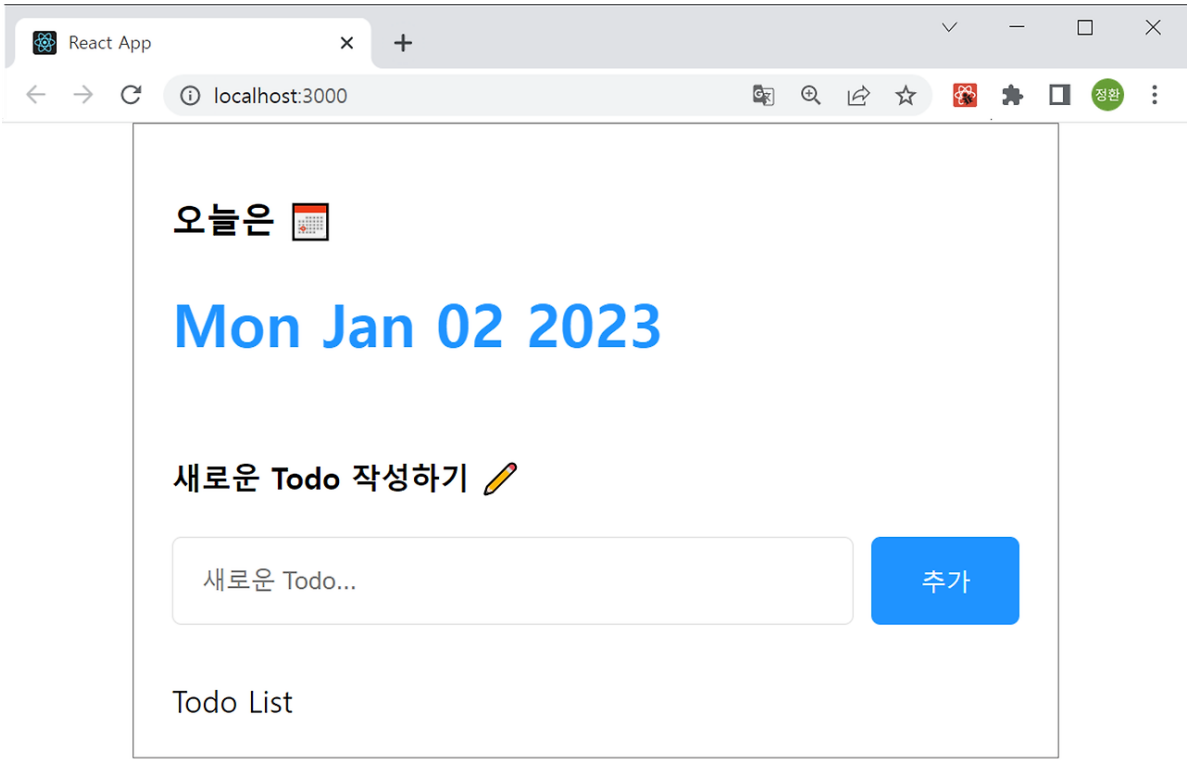
```
  border-radius: 5px;
  padding: 15px;
}

.TodoEditor input:focus {
  outline: none;
  border: 1px solid #1f93ff;
}

.TodoEditor button {
  cursor: pointer;
  width: 80px;
  border: none;
  background-color: #1f93ff;
  color: white;
  border-radius: 5px;
}
```

App.js에 TodoEditor 컴포넌트를 추가합니다.

src/App.js

```
import "./App.css";
import Header from "./component/Header";
import TodoEditor from "./component/TodoEditor";
```

```
function App() {
  return (
    <div className="App">
      <Header />
      <TodoEditor />
      <div>Todo List</div>
    </div>
  );
}

export default App;
```



TodoList, TodoItem 컴포넌트 만들기

TodoList와 TodoItem 컴포넌트를 생성합니다.

src/component/TodoList.js

```
import TodoItem from "./TodoItem";
import "./TodoList.css";

const TodoList = () => {
  return (
    <div className="TodoList">
      <h4>Todo List 🌱</h4>
      <input className="searchbar" placeholder="검색어를 입력하세요" />
      <div className="list_wrapper">
        <TodoItem />
        <TodoItem />
        <TodoItem />
      </div>
    </div>
  );
};

export default TodoList;
```

src/component/TodoList.css

```
.TodoList .searchbar {
    margin-bottom: 20px;
    width: 100%;
    border: none;
    border-bottom: 1px solid rgb(220, 220, 220);
    box-sizing: border-box;
    padding-top: 15px;
    padding-bottom: 15px;
}

.TodoList .searchbar:focus {
    outline: none;
    border-bottom: 1px solid #1f93ff;
}

.TodoList .list_wrapper {
    display: flex;
    flex-direction: column;
    gap: 20px;
}
```

src/component/TodoItem.js

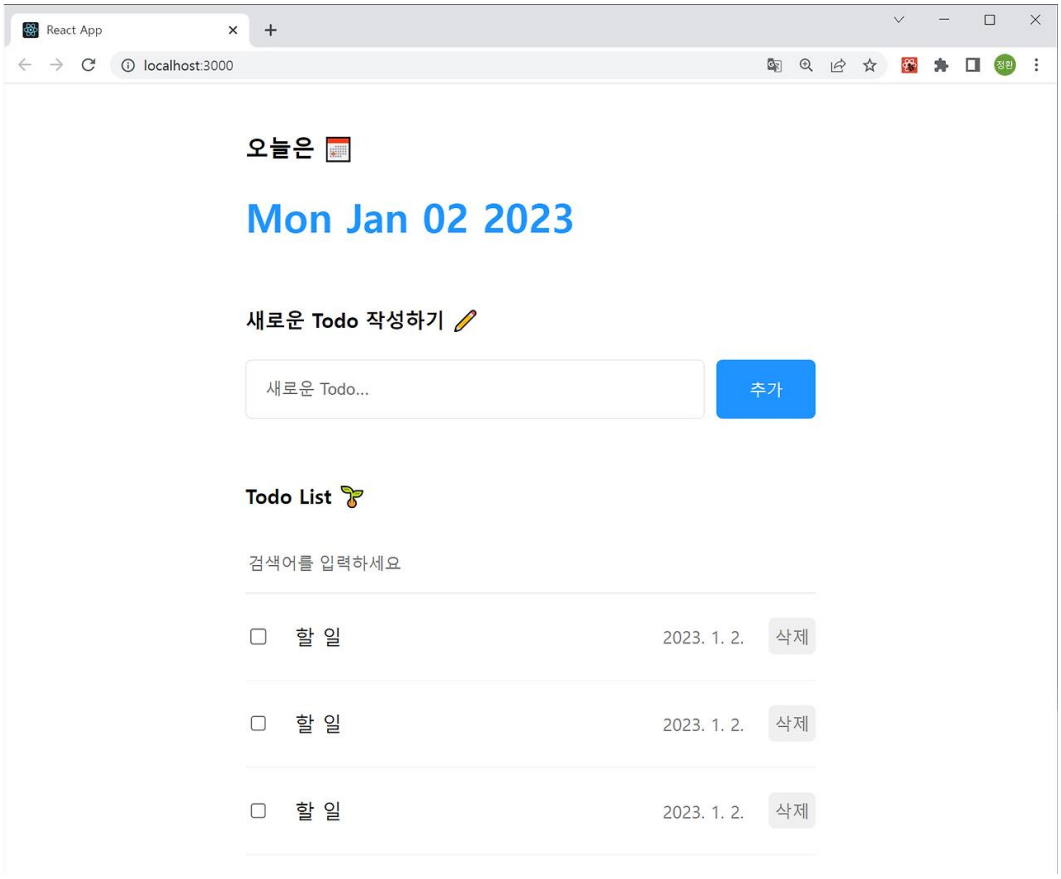
```
import "./TodoItem.css";


const TodoItem = () => {
  return (
    <div className="TodoItem">
      <div className="checkbox_col">
        <input type="checkbox" />
      </div>
      <div className="title_col">할 일</div>
      <div className="date_col">{new Date().toLocaleDateString()}</div>
      <div className="btn_col">
        <button>삭제</button>
      </div>
    </div>
  );
};


export default TodoItem;
```

App.js에 TodoList 컴포넌트를 추가합니다.

src/App.js

```
import "./App.css";
import Header from "./component/Header";
import TodoEditor from "./component/TodoEditor";
import TodoList from "./component/TodoList";

function App() {
  return (
    <div className="App">
      <Header />
      <TodoEditor />
      <TodoList />
    </div>
  );
}

export default App;
```

이제 모든 컴포넌트가 잘 렌더링되었는지 확인합니다. 모든 스타일링과 레이아웃이 제대로 적용되었는지 확인한 후 다음 단계로 넘어갑니다. UI 구현을 모두 완료했습니다.



3. 기능 구현 준비하기

UI를 완료했으니 이제 컴포넌트의 기능을 구현합니다. 먼저 컴포넌트별로 어떤 기능을 구현해야 하는지 살펴보겠습니다.

- **App 컴포넌트:** 할 일 데이터 관리
- **Header 컴포넌트:** 오늘의 날짜 표시
- **TodoEditor 컴포넌트:** 새로운 할 일 아이템 생성
- **TodoList 컴포넌트:** 검색에 따라 필터링된 할 일 아이템 렌더링
- **TodoItem 컴포넌트:** 할 일 아이템의 수정 및 삭제

데이터를 다루는 4개의 기본 기능은 추가(Create), 조회(Read), 수정(Update), 삭제(Delete) 기능입니다. 이를 CRUD라고 합니다. 이번 프로젝트의 기능 구현은 CRUD 순서에 따라 진행하겠습니다.

기초 데이터 설정하기

먼저 할 일 아이템을 생성할 상태를 설정합니다.

src/App.js

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

function App() {
  const [todo, setTodo] = useState([]);

  return (
    // ...
  );
}

export default App;
```

useState를 이용해 할 일 아이템의 상태를 관리할 State를 만들었습니다.

함수 useState에서 인수로 빈 배열을 전달해 State 변수 todo의 기본값을 빈 배열로 초기화했습니다.

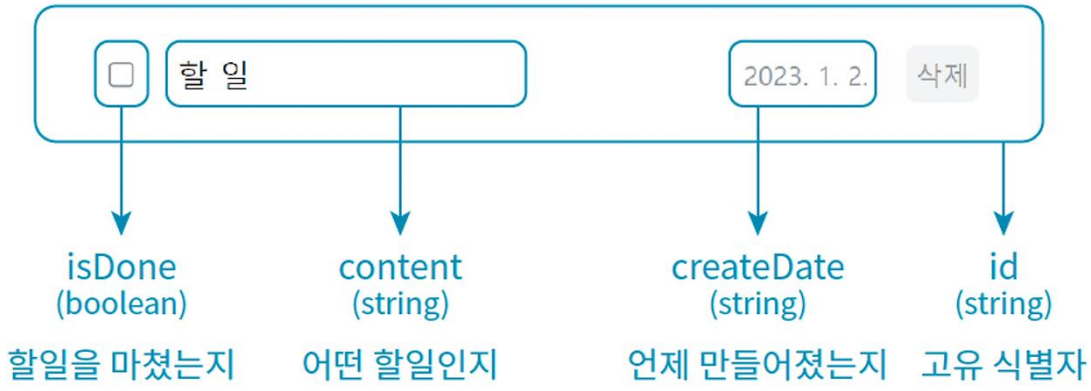
데이터 모델링하기

자바스크립트에서는 현실의 사물이나 개념을 객체로 표현합니다.

예를 들어 할 일 아이템을 다음과 같이 모델링합니다.

```
{
  id: 0,
  isDone: false,
  content: "React 공부하기",
  createdDate: new Date().getTime(),
}
```

각 할 일 아이템은 고유한 식별자 id, 완료 여부를 나타내는 isDone, 할 일 내용 content, 생성 날짜 createdDate를 포함합니다.



목 데이터 설정하기

기능을 구현하기 전에 테스트를 위한 목 데이터를 설정합니다.

src/App.js

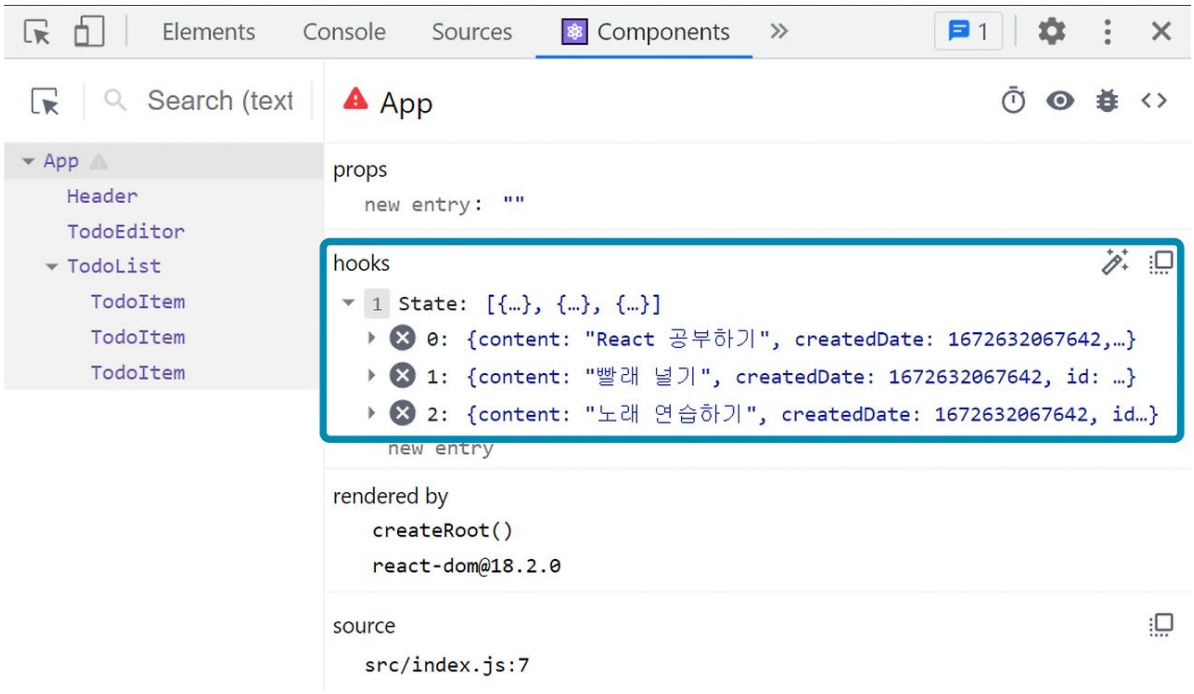
```
const mockTodo = [
  {
    id: 0,
    isDone: false,
```

```
    content: "React 공부하기",
    createdDate: new Date().getTime(),
  },
  {
    id: 1,
    isDone: false,
    content: "빨래 널기",
    createdDate: new Date().getTime(),
  },
  {
    id: 2,
    isDone: false,
    content: "노래 연습하기",
    createdDate: new Date().getTime(),
  },
];

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

  return (
    // ...
  );
```

}

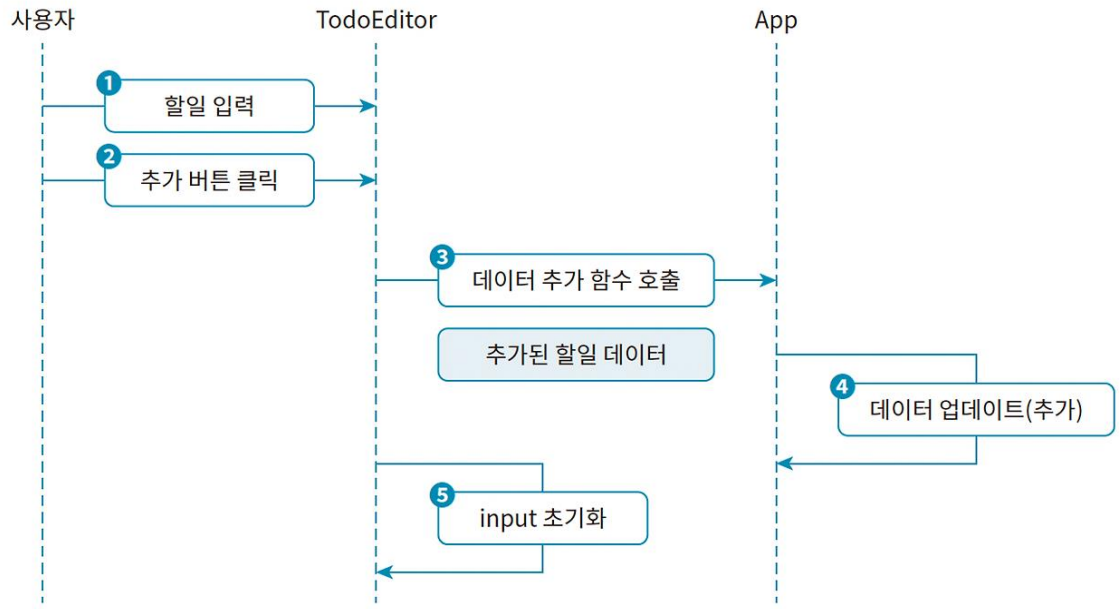
export default App;

목 데이터를 생성하고 useState의 기본값으로 설정했습니다.



4. Create: 할 일 추가하기

기능 흐름 살펴보기



할 일이 추가되는 과정은 다음과 같습니다.

1. 사용자가 새로운 할 일을 입력합니다.
2. TodoEditor 컴포넌트의 <추가> 버튼을 클릭합니다.
3. TodoEditor 컴포넌트는 App 컴포넌트에 아이템 추가 이벤트와 데이터를 전달합니다.
4. App 컴포넌트는 데이터를 이용해 새 아이템을 추가하고 State 변수 todo를 업데이트합니다.
5. TodoEditor 컴포넌트는 입력 폼을 초기화합니다.

아이템 추가 함수 만들기

새 할 일 아이템을 추가하는 함수를 만듭니다.

src/App.js

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


function App() {
```

```
  const [todo, setTodo] = useState(mockTodo);
  const idRef = useRef(3);

  const onCreate = (content) => {
    const newItem = {
      id: idRef.current,
      content,
      isDone: false,
      createdDate: new Date().getTime(),
    };
    setTodo([newItem, ...todo]);
    idRef.current += 1;
  };

  return (
    <div className="App">
      <Header />
      <TodoEditor onCreate={onCreate} />
      <TodoList />
    </div>
  );
}
```

```
export default App;
```

onCreate 함수는 새 아이템을 생성하고 todo 배열을 업데이트합니다. 고유한 id를 부여하기 위해 useRef를 사용합니다.

아이템 추가 함수 호출하기

사용자가 <추가> 버튼을 클릭하면 onCreate 함수가 호출됩니다.

src/component/TodoEditor.js

```
import { useState } from "react";
import "./TodoEditor.css";

const TodoEditor = ({ onCreate }) => {
  const [content, setContent] = useState("");

  const onChangeContent = (e) => {
    setContent(e.target.value);
  };

  const onSubmit = () => {
    if (!content) {
      inputRef.current.focus();
      return;
    }
```

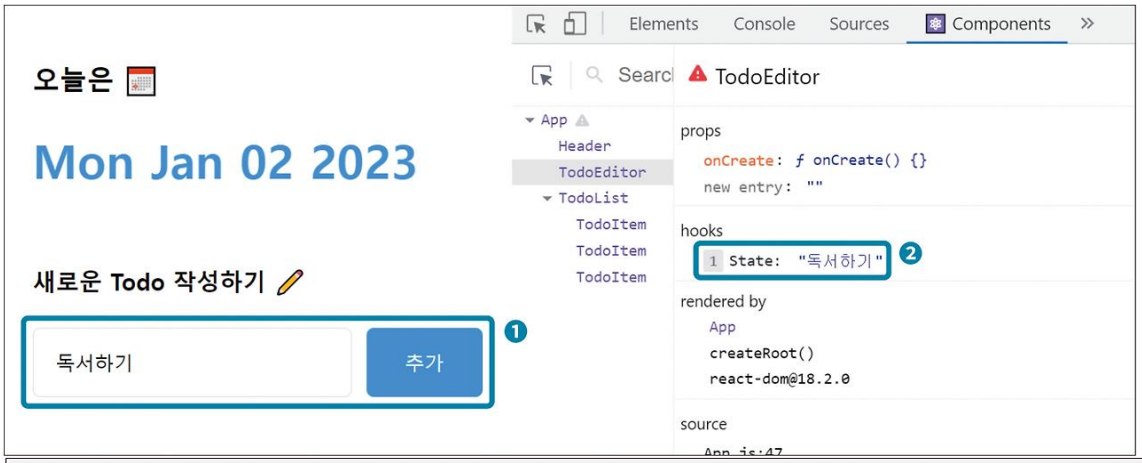
```
    onCreate(content);
    setContent("");
  };

  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>
      </div>
    </div>
  );
};

export default TodoEditor;
```

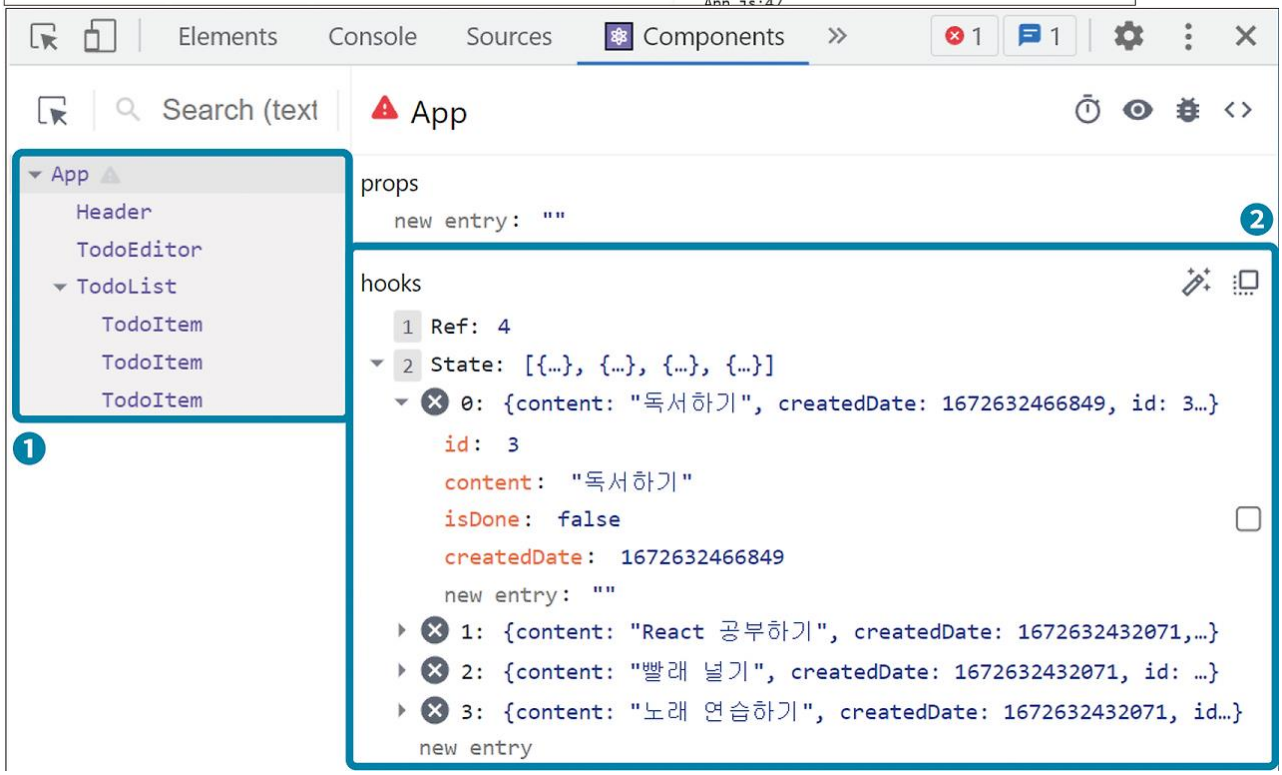
TodoEditor 컴포넌트에서 사용자가 입력한 내용을 content에 저장하고, <추가> 버튼을 클릭하면 onCreate 함수를 호출합니다. 입력 폼은 초기화됩니다.

e.target.value

e.target.value는 이벤트 객체 e의 타겟 요소에 대한 값을 가져오는 속성입니다. 주로 입력 폼에서 사용자가 입력한 값을 얻기 위해 사용됩니다.

이 코드에서 onChangeContent 함수는 입력 필드의 값이 변경될 때마다 호출됩니다. e.target.value는 입력 필드에 입력된 값을 나타내며, 이 값을 content 상태 변수에 저장합니다.





빈 입력 방지하기

할 일을 입력하지 않고 <추가> 버튼을 클릭할 경우 입력 폼에 포커스를 줍니다.

src/component/TodoEditor.js

```
import { useState, useRef } from "react";
import "./TodoEditor.css";


const TodoEditor = ({ onCreate }) => {
  const [content, setContent] = useState("");
  const inputRef = useRef(); // useRef 사용해서 DOM 요소 참조


  const onChangeContent = (e) => {
    setContent(e.target.value);
  };


  const onSubmit = () => {
        // 빈 입력 방지
    if (!content) {
      inputRef.current.focus(); // 포커스 설정
      return;
    }
    // 초기화
    onCreate(content);
    setContent("");
```

```
  };

  return (
    <div className="TodoEditor">
      <h4>새로운 Todo 작성하기 ✏ </h4>
      <div className="editor_wrapper">
        <input
          ref={inputRef}
          value={content}
          onChange={onChangeContent}
          placeholder="새로운 Todo..."
        />
        <button onClick={onSubmit}>추가</button>
      </div>
    </div>
  );
};

export default TodoEditor;
```

inputRef를 사용하여 입력 폼에 포커스를 줍니다.

```
inputRef.current.focus()
```

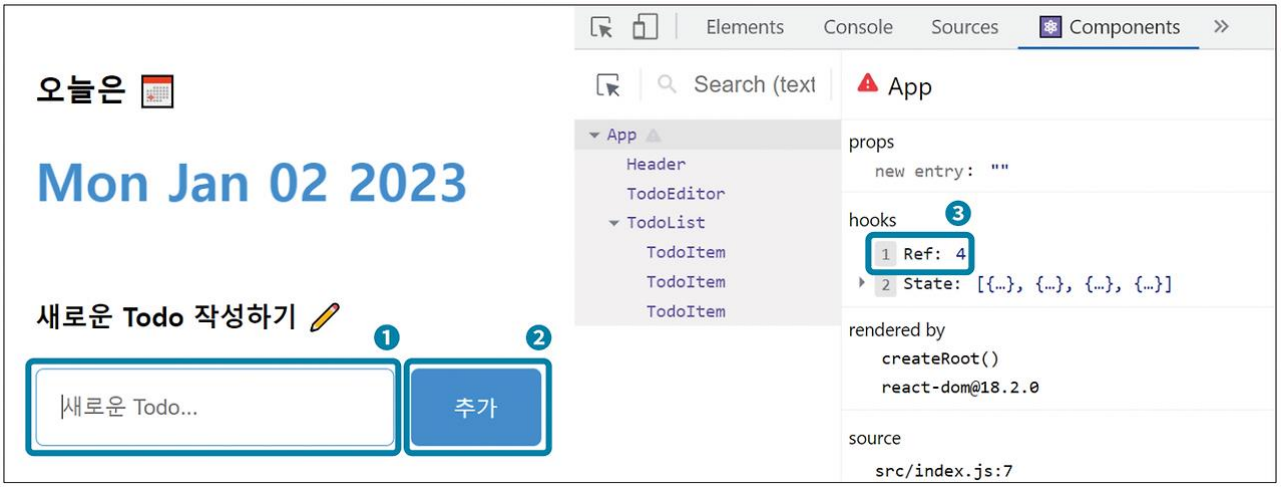
inputRef.current.focus()는 useRef 훅을 사용하여 특정 DOM 요소에 포커스를 설정하는 메서드입

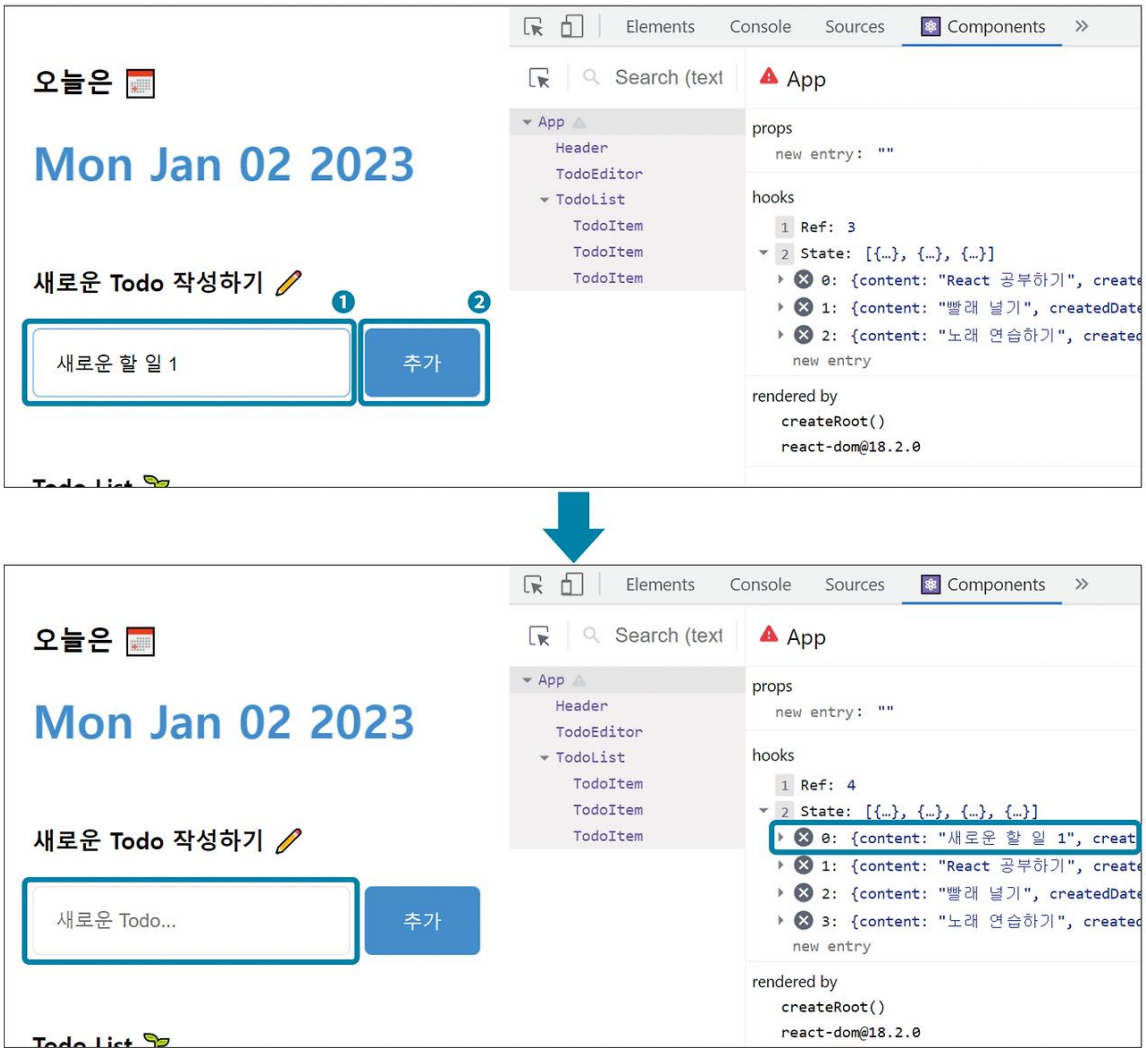
니다.

이 메서드는 주로 사용자 경험을 개선하기 위해 특정 입력 필드에 자동으로 포커스를 줄 때 사용됩니다.

코드에서 inputRef는 useRef 훅을 사용하여 입력 필드를 참조합니다.

onSubmit 함수가 호출되면 inputRef.current.focus()가 실행되어 입력 필드에 포커스가 설정됩니다.





<Enter> 키로 아이템 추가하기

키보드의 <Enter> 키를 눌렀을 때도 아이템이 추가되도록 합니다.

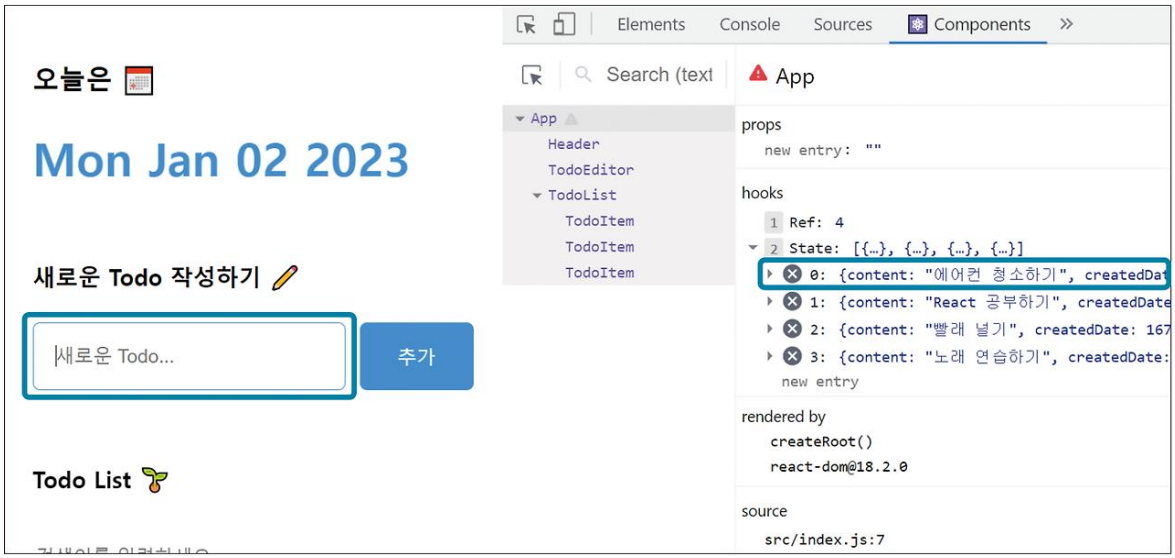
src/component/TodoEditor.js

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

import "./TodoEditor.css";


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

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

const onChangeContent = (e) => {
  setContent(e.target.value);
};

const onSubmit = () => {
  if (!content) {
    inputRef.current.focus();
    return;
  }
  onCreate(content);
  setContent("");
};

const onKeyDown = (e) => {
  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}
            placeholder="새로운 Todo..."
          />
          <button onClick={onSubmit}>추가</button>
        </div>
      </div>
    );
};

export default TodoEditor;
```

onKeyDown 이벤트 핸들러를 사용하여 <Enter> 키를 눌렀을 때도 아이템이 추가되도록 했습니다.



이로써 [할 일 관리] 앱의 Create 기능을 모두 구현했습니다.

---

5. Read: 할 일 리스트 렌더링하기

이번에는 TodoList 컴포넌트의 기능이자 CRUD의 두 번째 요소인 Read 기능을 구현하겠다.

Read 기능을 이용하면 배열에 저장한 여러 할 일 아이템을 반복해서 페이지에 렌더링할 수 있다.

배열을 리스트로 렌더링하기

App 컴포넌트의 State 변수 todo에는 배열 형태로 여러 개의 할 일 아이템이 저장되어 있다.

배열 todo를 TodoList 컴포넌트에 Props로 전달하겠다.

```
// src/App.js
function App() {
  const [todo, setTodo] = useState(mockTodo);

  return (
```

```
      <div className="App">
        <Header />
        <TodoEditor onCreate={onCreate} />
        <TodoList todo={todo} />
      </div>
    );
}
export default App;
```

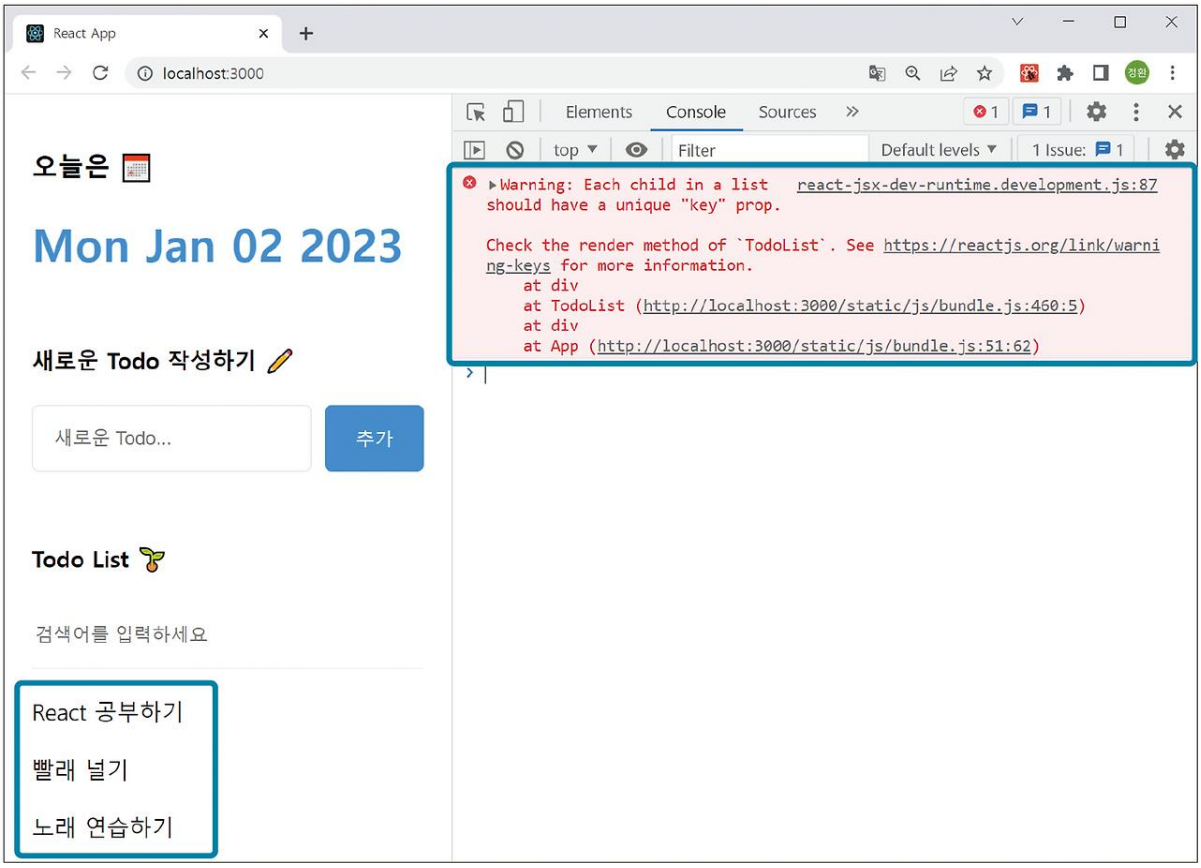
TodoList 컴포넌트에서는 App에서 Props로 전달된 todo를 리스트로 렌더링해야 한다. 리액트에서 배열 데이터를 렌더링할 때는 배열 메서드 map을 주로 이용한다.

map을 이용해 HTML 반복하기

TodoList 컴포넌트에서 배열 메서드 map을 이용해 HTML 요소를 반복해 렌더링하겠다.

```
// src/component/TodoList.js
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) => (
```

```
            <div>{it.content}</div>
          ))}
        </div>
      </div>
    );
};

export default TodoList;
```



map을 이용해 컴포넌트 반복하기

이번에는 map 메서드의 콜백 함수가 HTML이 아닌 컴포넌트를 반환하도록 수정하겠다.

// src/component/TodoList.js

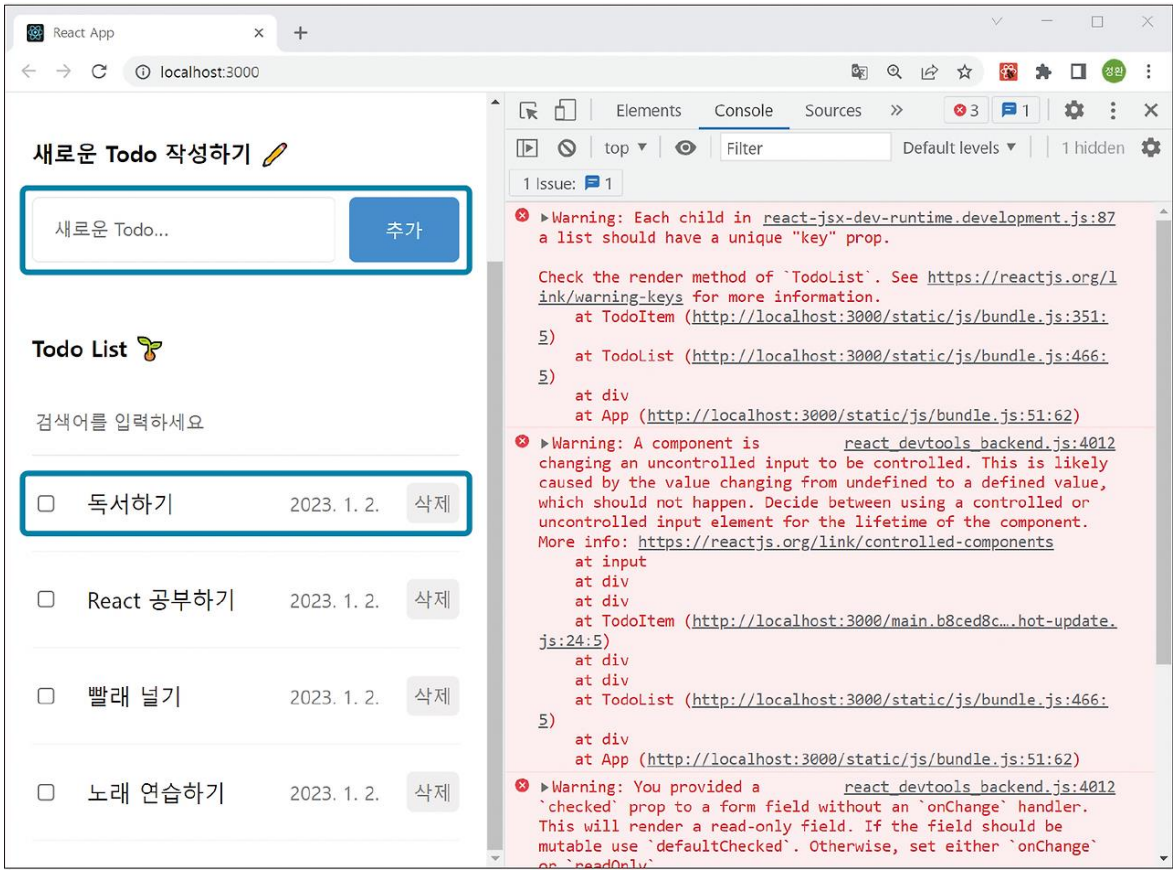
```
import TodoItem from "./TodoItem";
import "./TodoList.css";

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} />
        ))}
      </div>
    </div>
  );
};
export default TodoList;
```

TodoItem 컴포넌트에 전달된 Props를 사용할 수 있도록 다음과 같이 수정한다.

```
// src/component/TodoItem.js
import "./TodoItem.css";

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

```
  return (
    <div className="TodoItem">
      <div className="checkbox_col">
        <input checked={isDone} type="checkbox" />
      </div>
      <div className="title_col">{content}</div>
      <div className="date_col">
        {new Date(createdDate).toLocaleDateString()}
      </div>
      <div className="btn_col">
        <button>삭제</button>
      </div>
    </div>
  );
};

export default TodoItem;
```

위 코드를 작성하고 저장한 다음, 할 일 입력 폼에 '독서하기'라는 새 아이템을 추가하여 결과를 확인한다.



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

리스트의 각 컴포넌트를 고유하게 구분하기 위해 key를 설정해야 한다.

key는 리스트에서 각각의 컴포넌트를 구분하기 위해 사용하는 값이다.

고유한 id를 key로 전달하여 문제를 해결할 수 있다.

```
// src/component/TodoList.js

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

        ))}

      </div>

    </div>

  );

};

export default TodoList;
```

이제 개발자 도구의 콘솔을 확인하여 key와 관련된 경고 메시지가 더 이상 발생하지 않는지 확인한다.

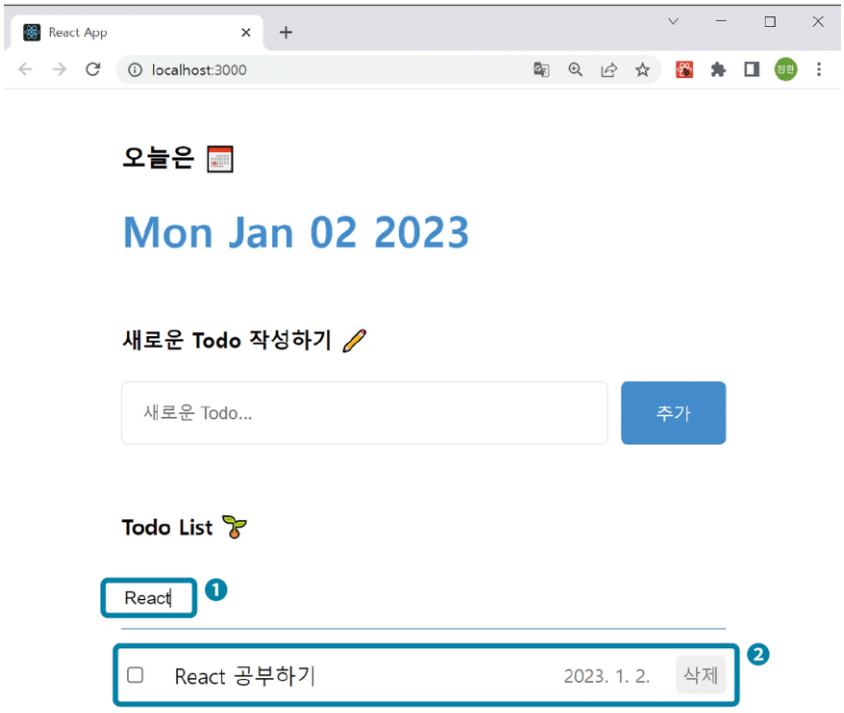
검색어에 따라 필터링하기

TodoList 컴포넌트에서 특정 할 일을 검색하는 기능을 구현하겠다.

사용자가 입력하는 검색어를 처리할 State 변수를 만든 다음, 검색 폼에서 사용자가 입력한 내용을 처리하는 기능을 만듭니다.

```
// src/component/TodoList.js

import { useState } from "react";

import TodoItem from "./TodoItem";

import "./TodoList.css";


const TodoList = ({ todo }) => {

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


  const onChangeSearch = (e) => {

    setSearch(e.target.value);

  };


  const getSearchResult = () => {

        // 빈 문자열이면 그대로 todo, 아니면 일치하는 아이템만 필터링해 반환

    return search === ""

      ? todo

      : todo.filter((it) =>

          it.content.toLowerCase().includes(search.toLowerCase()) // 대소문자 구별하지 않게

        );

  };
```

```
  return (
    <div className="TodoList">
      <h4>Todo List 🌱</h4>
      <input
        value={search}
        onChange={onChangeSearch}
        className="searchbar"
        placeholder="검색어를 입력하세요"
      />
      <div className="list_wrapper">
        {getSearchResult().map((it) => (
          <TodoItem key={it.id} {...it} />
        ))}
      </div>
    </div>
  );
};

export default TodoList;
```

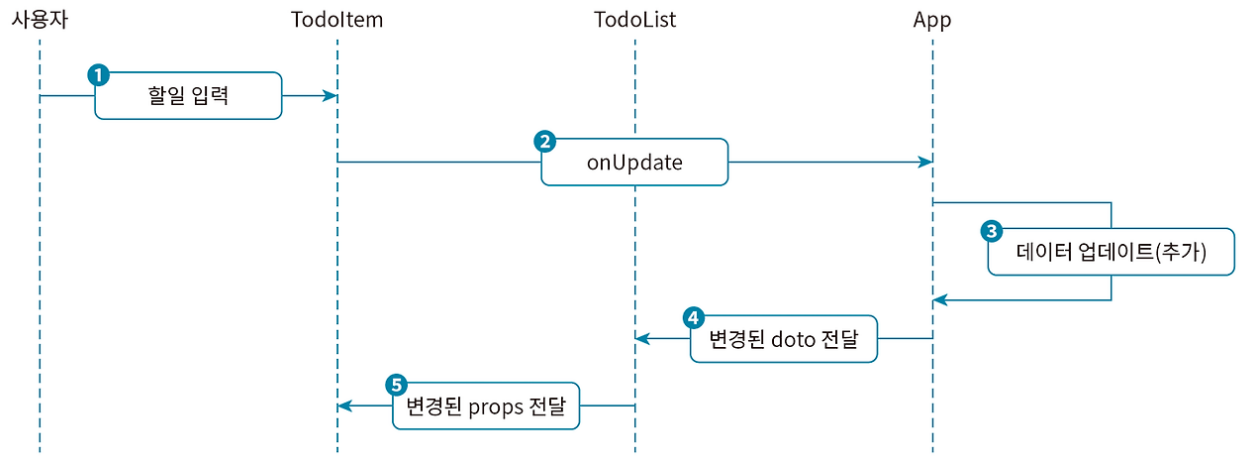


---

6. Update: 할 일 수정하기

CRUD의 세 번째 기능은 Update이다. 할 일 아이템의 수정 기능을 구현하겠다.

기능 흐름 살펴보기



할 일 아이템의 수정은 다음과 같은 일련의 과정이 필요하다.

1. 사용자가 TodoItem의 체크박스를 클릭한다.

2. TodoItem 컴포넌트는 함수 onUpdate를 호출하고, 해당 아이템의 id를 인수로 전달한다.
3. App 컴포넌트의 함수 onUpdate는 해당 아이템의 상태를 토글하기 위해 State 값을 업데이트한다.
4. State 값이 변경되면 TodoList에 전달하는 Props의 값도 변경된다.
5. TodoList는 변경된 State 값을 다시 리스트로 렌더링한다.

아이템 수정 함수 만들기

할 일 생성을 위해 함수 onCreate를 만들었듯이, 수정을 위해 함수 onUpdate를 만든다.

```
// src/App.js
function App() {
  const [todo, setTodo] = useState(mockTodo);

  const onUpdate = (targetId) => {
    setTodo(
      todo.map((it) =>
        it.id === targetId ? { ...it, isDone: !it.isDone } : it
      )
    );
  };

  return (
    <div className="App">
      <Header />
      <TodoEditor onCreate={onCreate} />
```

```
        <TodoList todo={todo} onUpdate={onUpdate} />
      </div>
    );
}

export default App;
```

TodoList 컴포넌트에서 TodoItem 컴포넌트에 함수 onUpdate를 전달한다.

getSearchResult().map((it) => (...))

- getSearchResult() 함수는 검색어에 따라 필터링된 할 일 리스트를 반환한다.
- map 메서드는 이 반환된 배열을 순회하면서 각 요소를 렌더링할 JSX를 반환한다.
- it는 배열의 각 요소를 의미한다. 즉, 하나의 할 일 아이템 객체이다.

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

이 부분은 TodoItem 컴포넌트를 렌더링하는 JSX 표현식이다. 각 TodoItem 컴포넌트에 여러 Props를 전달한다.

- key={it.id}: key는 리액트에서 리스트를 렌더링할 때 각 항목을 고유하게 식별하기 위해 사용된다. it.id는 할 일 아이템의 고유 식별자이다. 이를 통해 리액트는 리스트에서 각 아이템을 구분할 수 있다.
- **{...it}: 스프레드 연산자(...)**를 사용하여 it 객체의 모든 속성을 TodoItem 컴포넌트에 Props로 전달한다. 예를 들어, it 객체가 { id: 1, content: "할 일", isDone: false, createdDate: "2023-01-01" }이라면, TodoItem 컴포넌트에 id, content, isDone, createdDate라는 Props가 전달된다.
- onUpdate={onUpdate}: onUpdate 함수는 Props로 전달되어 TodoItem 컴포넌트에서 호출할 수 있다. 이 함수는 할 일의 상태를 업데이트하는 역할을 한다.

요약

- map 메서드는 배열을 순회하면서 각 요소를 JSX로 변환한다.
- key는 각 요소를 고유하게 식별하기 위해 사용된다.

- 스프레드 연산자(...)를 사용하여 객체의 모든 속성을 Props로 전달한다.
- onUpdate는 할 일의 상태를 업데이트하는 함수로, TodoItem 컴포넌트에서 호출할 수 있도록 전달된다.

이 코드는 할 일 리스트를 렌더링하고, 각 할 일 아이템에 필요한 데이터를 Props로 전달하여 개별 아이템이 올바르게 표시되도록 한다.

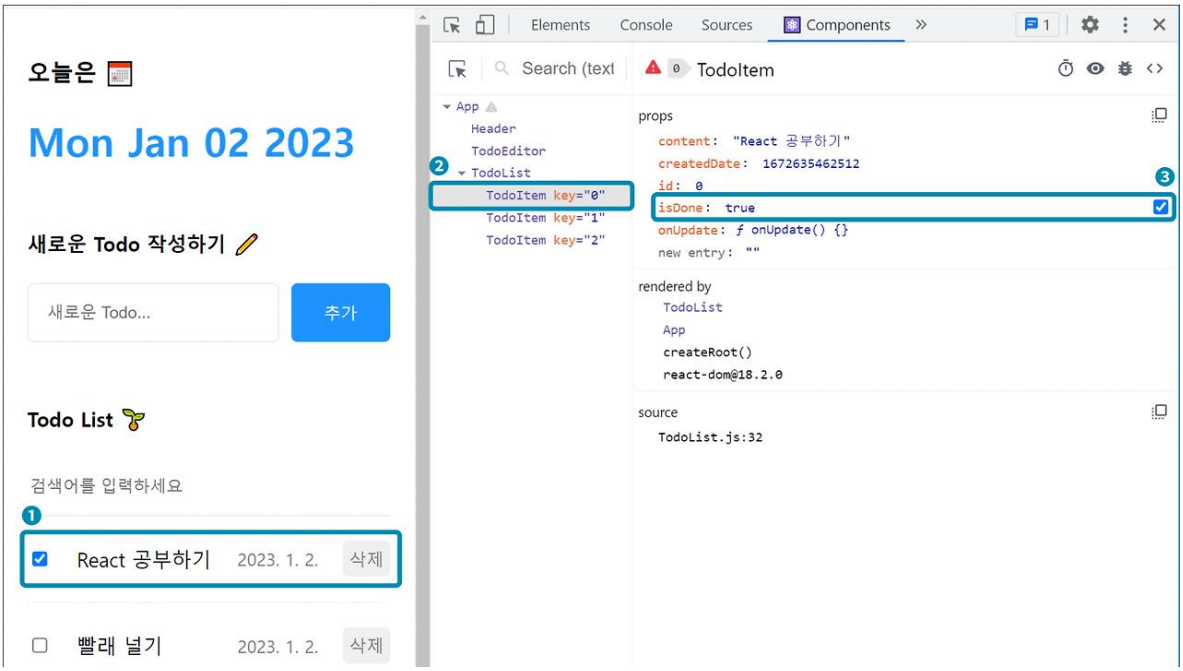
TodoItem 컴포넌트에서 아이템 수정 함수 호출하기

TodoItem 컴포넌트에서 체크박스를 클릭하면 함수 onUpdate를 호출하도록 구현한다.

```
// src/component/TodoItem.js

import "./TodoItem.css";


const TodoItem = ({ id, content, isDone, createdDate, onUpdate }) => {
  const onChangeCheckbox = () => {
    onUpdate(id);
  };


  return (
    <div className="TodoItem">
      <div className="checkbox_col">
        <input onChange={onChangeCheckbox} checked={isDone} type="checkbox" />
      </div>
      <div className="title_col">{content}</div>
      <div className="date_col">
        {new Date(createdDate).toLocaleDateString()}
```

```
        </div>
        <div className="btn_col">
          <button>삭제</button>
        </div>
      </div>
    );
};
export default TodoItem;
```

할 일 완료 여부를 확인하는 체크박스를 클릭하여 정상적으로 업데이트되는지 확인한다.

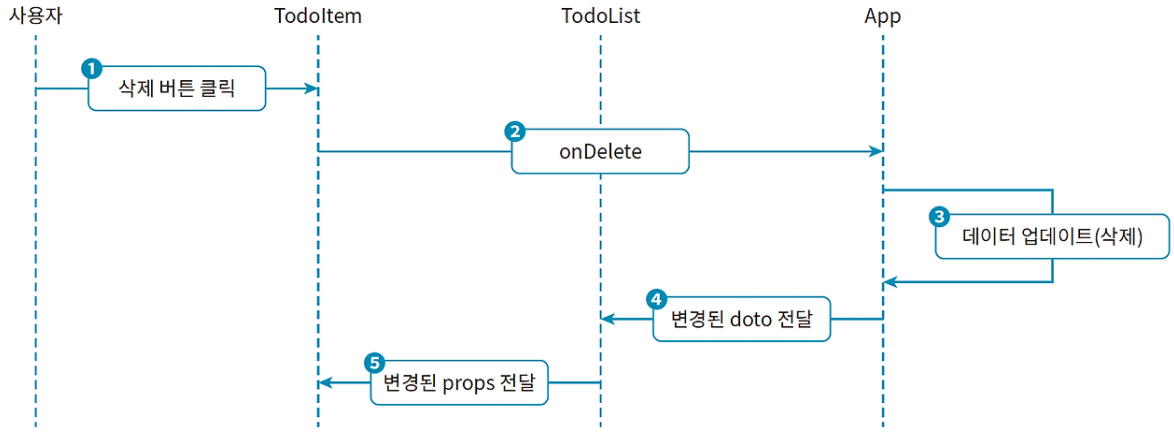


---

7. Delete: 할 일 삭제하기

마지막으로 CRUD의 Delete 기능을 구현하여 할 일 아이템을 삭제하겠다.

기능 흐름 살펴보기



할 일 아이템의 삭제는 다음과 같은 일련의 과정이 필요하다.

1. 사용자가 TodoItem의 <삭제> 버튼을 클릭한다.
2. 함수 onDelete를 호출한다.
3. <삭제> 버튼을 클릭하면 삭제할 할 일 아이템만 빼고, 새 배열을 만들어 State 값을 업데이트한다.
4. State 변수 todo가 업데이트되면 TodoList 컴포넌트에 전달한 Props의 값도 변경된다.
5. TodoList 컴포넌트는 Props의 값이 변경되면 리렌더링한다.

아이템 삭제 함수 만들기

App 컴포넌트에서 할 일을 삭제하는 함수 onDelete를 만든다.

```
// src/App.js
function App() {
  const [todo, setTodo] = useState(mockTodo);

  const onDelete = (targetId) => {
    setTodo(todo.filter((it) => it.id !== targetId));
  };
```

```
  return (
    <div className="App">
      <Header />
      <TodoEditor onCreate={onCreate} />
      <TodoList todo={todo} onUpdate={onUpdate} onDelete={onDelete} />
    </div>
  );
}

export default App;
```

TodoList 컴포넌트에서 TodoItem 컴포넌트에 함수 onDelete를 전달한다.

TodoItem의 <삭제> 버튼을 클릭 했을때 호출하는 함수 onDelete는 매개변수 targetId에 삭제 할 일기 아이템의 id를 저장합니다.

그리고 해당 id 요소를 뺀 새 배열로 todo를 업데이트 함으로써 대상 아이템을 삭제합니다.

```
// src/component/TodoList.js

const TodoList = ({ todo, onUpdate, onDelete }) => {
  return (
    <div className="TodoList">
      <h4>Todo List 🌱</h4>
      <input className="searchbar" placeholder="검색어를 입력하세요" />
      <div className="list_wrapper">
```

```
            {getSearchResult().map((it) => (
                <TodoItem key={it.id} {...it} onUpdate={onUpdate} onDelete={onDelete} />
            ))}
          </div>
        </div>
    );
};
export default TodoList;
```

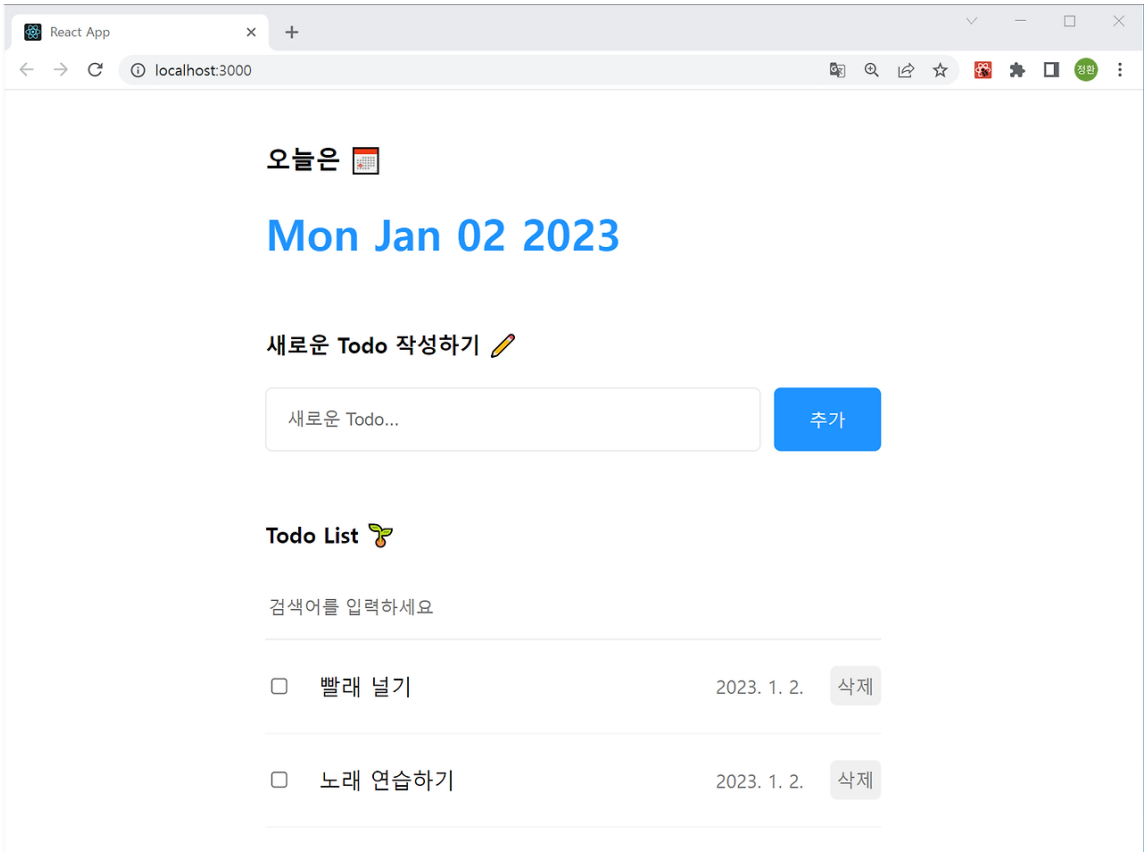
TodoItem 컴포넌트에서 삭제 함수 호출하기

TodoItem에서 <삭제> 버튼을 클릭하면 함수 onDelete를 호출하도록 구현한다.

```
// src/component/TodoItem.js
import "./TodoItem.css";

const TodoItem = ({ id, content, isDone, createdDate, onUpdate, onDelete }) => {
    const onChangeCheckbox = () => {
        onUpdate(id);
    };

    const onClickDelete = () => {
        onDelete(id);
    };
```

```
  return (
    <div className="TodoItem">
      <div className="checkbox_col">
        <input onChange={onChangeCheckbox} checked={isDone} type="checkbox" />
      </div>
      <div className="title_col">{content}</div>
      <div className="date_col">
        {new Date(createdDate).toLocaleDateString()}
      </div>
      <div className="btn_col">
        <button onClick={onClickDelete}>삭제</button>
      </div>
    </div>
  );
};

export default TodoItem;
```

할 일 아이템을 선택해 <삭제> 버튼을 클릭하여 아이템이 잘 삭제되는지 페이지에서 확인한다.



이렇게 두 번째 리액트 앱 프로젝트인 [할 일 관리] 앱을 모두 완성하였다.

Props Drilling이나 최적화 문제, 상태 관리 등 리액트 서비스와 관련해 알아야 할 내용들을 공부하면서 [할 일 관리] 앱을 한 단계 업그레이드하겠다.

[할 일 관리] 앱 업그레이드

이번 포스트에서는 useReducer 훅을 사용하여 [할 일 관리] 앱을 업그레이드합니다.

useReducer는 상태 관리 로직을 컴포넌트 외부로 분리할 수 있어 상태가 복잡해질 때 유용합니다.

기존의 useState를 useReducer로 대체해보겠습니다.

1. useState를 useReducer로 바꾸기

먼저, App.js 파일에서 useState를 useReducer로 대체합니다.

코드 변경

```
import { useReducer, useRef } from "react";


// 상태 변화 로직
function reducer(state, action) {
  switch (action.type) {
    case "CREATE":
      return [action.newItem, ...state];
    case "UPDATE":
      return state.map((it) =>
        it.id === action.targetId ? { ...it, isDone: !it.isDone } : it
      );
    case "DELETE":
      return state.filter((it) => it.id !== action.targetId);
    default:
      return state;
  }
}


function App() {
  const [todo, dispatch] = useReducer(reducer, mockTodo);
```

```
const idRef = useRef(3);


const onCreate = (content) => {
  dispatch({
    type: "CREATE",
    newItem: {
      id: idRef.current,
      content,
      isDone: false,
      createdDate: new Date().getTime(),
    },
  });
  idRef.current += 1;
};


const onUpdate = (targetId) => {
  dispatch({ type: "UPDATE", targetId });
};


const onDelete = (targetId) => {
  dispatch({ type: "DELETE", targetId });
};
```

```
  return (
    <div className="App">
      <Header />
      <TodoEditor onCreate={onCreate} />
      <TodoList todo={todo} onUpdate={onUpdate} onDelete={onDelete} />
    </div>
  );
}

export default App;
```

설명

1. useReducer를 react 라이브러리에서 불러옵니다. useState 대신 useReducer를 사용할 예정이므로 기존 useState 코드는 삭제합니다.
2. reducer 함수는 상태 변화 로직을 처리합니다. 여기서는 CREATE, UPDATE, DELETE 타입에 따라 상태를 변화시킵니다.
3. useReducer를 사용하여 todo 상태와 dispatch 함수를 초기화합니다. dispatch 함수는 상태 변화를 촉발합니다.
4. onCreate, onUpdate, onDelete 함수에서 **dispatch**를 호출하여 **상태 변화를 처리**합니다.

2. Create: 할 일 아이템 추가하기

코드 변경

```
const onCreate = (content) => {
  dispatch({
    type: "CREATE",
```

```
    newItem: {
      id: idRef.current,
      content,
      isDone: false,
      createdDate: new Date().getTime(),
    },
  });
  idRef.current += 1;
};
```

설명

- dispatch 함수는 type을 CREATE로 설정하고, 새 할 일 아이템을 newItem 속성에 저장하여 호출합니다.
- reducer 함수에서 CREATE 타입을 처리하여 새 아이템을 기존 상태에 추가합니다.

3. Update: 할 일 아이템 수정하기

코드 변경

```
const onUpdate = (targetId) => {
  dispatch({ type: "UPDATE", targetId });
};
```

설명

- dispatch 함수는 type을 UPDATE로 설정하고, targetId 속성에 수정할 아이템의 id를 저장하여 호출합니다.
- reducer 함수에서 UPDATE 타입을 처리하여 해당 아이템의 isDone 상태를 토글합니다.

4. Delete: 할 일 삭제하기

코드 변경

```
const onDelete = (targetId) => {
  dispatch({ type: "DELETE", targetId });
};
```

설명

- dispatch 함수는 type을 DELETE로 설정하고, targetId 속성에 삭제할 아이템의 id를 저장하여 호출합니다.
- reducer 함수에서 DELETE 타입을 처리하여 해당 아이템을 상태에서 제거합니다.

전체 코드

```
import { useReducer, useRef } from "react";
import Header from "./component/Header";
import TodoEditor from "./component/TodoEditor";
import TodoList from "./component/TodoList";
import mockTodo from "./mockTodo";

function reducer(state, action) {
  switch (action.type) {
    case "CREATE":
      return [action.newItem, ...state];
    case "UPDATE":
      return state.map((it) =>
```

```
            it.id === action.targetId ? { ...it, isDone: !it.isDone } : it
        );
    case "DELETE":
        return state.filter((it) => it.id !== action.targetId);
    default:
        return state;
  }
}

function App() {
  const [todo, dispatch] = useReducer(reducer, mockTodo);
  const idRef = useRef(3);

  const onCreate = (content) => {
    dispatch({
      type: "CREATE",
      newItem: {
        id: idRef.current,
        content,
        isDone: false,
        createdDate: new Date().getTime(),
      },
    });
```

```
    idRef.current += 1;
  };

  const onUpdate = (targetId) => {
    dispatch({ type: "UPDATE", targetId });
  };

  const onDelete = (targetId) => {
    dispatch({ type: "DELETE", targetId });
  };

  return (
    <div className="App">
      <Header />
      <TodoEditor onCreate={onCreate} />
      <TodoList todo={todo} onUpdate={onUpdate} onDelete={onDelete} />
    </div>
  );
}

export default App;
```

이번 포스트에서는 useReducer를 사용하여 [할 일 관리] 앱을 업그레이드했습니다.

useReducer를 이용하면 상태 변화 로직을 컴포넌트 외부로 분리할 수 있어 컴포넌트 코드가 더 간결해집니다.

앞으로 더 복잡한 상태 관리를 할 때 유용하게 사용할 수 있습니다.