렌더링과 todo앱

**학습목표**

컴포넌트가 언제 렌더링 되는지 알아보자

todo앱 구현

**목차**

[1. 렌더링 2](#_Toc131584314)

[1) 언제 렌더링 되는가? 2](#_Toc131584315)

[2) 언제 렌더링 안되는가? #1 7](#_Toc131584316)

[3) 언제 렌더링 안되는가? #2 9](#_Toc131584317)

[4) 렌더링 안되는 이유 11](#_Toc131584318)

[2. todo 앱 12](#_Toc131584319)

[1) 화면 분할 12](#_Toc131584320)

[2) 컴포넌트 분할 12](#_Toc131584321)

[3) 구현 13](#_Toc131584322)

[4) 연습 문제 19](#_Toc131584323)

# 렌더링

## 언제 렌더링 되는가?

컴포넌트가 언제 렌더링 되는지 알아보기 위한 예제를 구현하자.

### src/CounterA.tsx 생성

|  |
| --- |
| import React, { useState } from 'react'  function CounterA() {    console.log("CounterA 호출됨");    const [value, setValue] = useState<number>(0);    return (      <div className="box">        <h1>CounterA</h1>        <button onClick={() => setValue(value + 1)}>          { value }        </button>      </div>    )  }  export default CounterA; |

console.log("CounterA 호출됨");

CountA 함수형 컴포넌트가 렌더링 될 때 마다 위 코드가 실행될 것이다.

const [value, setValue] = useState<number>(0);

useState 함수로 상태를 생성할 때, 위와 같이 값의 타입을 적어주는 것이 좋다.

### src/CounterB.tsx 생성

|  |
| --- |
| import React, { useState } from 'react'  function CounterB() {    console.log("CounterB 호출됨");    const [value, setValue] = useState<number>(0);    return (      <div className="box">        <h1>CounterB</h1>        <button onClick={() => setValue(value + 1)}>          { value }        </button>      </div>    )  }  export default CounterB; |

### src/GoupComponent.tsx 생성

|  |
| --- |
| import React, { useState } from 'react'  import CounterA from './CounterA'  import CounterB from './CounterB'  function GroupComponent() {    console.log('GroupComponent 호출됨');    const [value, setValue] = useState<number>(0);    return (      <div className='box'>        <h1>GroupComponent</h1>        <button onClick={() => setValue(value + 1)}>          { value }        </button>        <CounterA />        <CounterB />      </div>    )  }  export default GroupComponent; |

### src/App.tsx 수정

|  |
| --- |
| import React, { useState } from 'react'  import './App.css'  import GroupComponent from './GroupComponent';  function App() {    console.log('App 호출됨');    const [value, setValue] = useState<number>(0);    return (      <div className="box">        <h1>App</h1>        <button onClick={() => setValue(value + 1)}>          { value }        </button>        <GroupComponent />      </div>    );  }  export default App; |

### src/App.css 수정

|  |
| --- |
| h1 { font-size: 11pt; margin: 0; }  div.box { display: inline-block; margin: 10px; padding: 10px;    border: 1px solid gray; }  button { display: block; padding: 0.2em 2em; margin: 0.5em; } |

### 실행

웹브라우저 개발자창의 콘솔 탭을 열고, 새로고침 하자.

새로고침 하면 리액트앱이 재시작하게 된다.

Graphical user interface, text, application, email

Description automatically generated

위 창에서 회색으로 출력된 메시지 부분은 무시하고, 검은색 메시지 출력만 보자.

참고: index.tsx 파일의 <React.StrictMode> 컴포넌트는 오류를 찾고 경고를 해주는 도구이다.

이 컴포넌트에 의해 회색 부분이 출력됨.

프로젝트를 빌드할 때는 이 컴포넌트가 무시되어 배포 파일에 포함되지 않는다.

처음 렌더링 될 때, 모든 컴포넌트들이 호출되었다. (모든 컴포넌트들이 렌더링 되었다)

Graphical user interface

Description automatically generated

CounterA 컴포넌트의 버튼을 클릭하자.

Graphical user interface, text, application, email

Description automatically generated

CounterA 컴포넌트 함수만 호출되었다. 즉 CounterA 컴포넌트만 렌더링 되었다.

리액트 상태가 변경되어 렌더링 될 때,

상태가 변경된 컴포넌트만 렌더링 된다는 것을 알 수 있다.

### 실습

ComponentB 컴포넌트의 버튼을 클릭하면 ComponentB 컴포넌트만 렌더링 될 것이다.

확인해 보자.

### 실습

GroupComponent 컴포넌트 버튼을 클릭하자

Graphical user interface, text, application, email

Description automatically generated

GroupComponent 컴포넌트와 그 자식 컴포넌트들이 렌더링 되었다.

### 실습

App 컴포넌트 버튼을 클릭하자

Graphical user interface, text, application, email

Description automatically generated

### 요약

부모 컴포넌트가 렌더링될 때, 자식 컴포넌트들도 전부 렌더링된다.

처음 렌더링 할 때는 당연히 모든 컴포넌트가 렌더링 되지만,

그 다음 렌더링부터는 상태가 변경된 컴포넌트와 그 자식 컴포넌트들만 렌더링 된다.

## 소스코드 파일 하나에 여러 컴포넌트 구현 가능

소스코드 파일 하나에 여러 함수형 컴포넌트를 구현해도 된다.

### src/GoupComponent.tsx 수정

|  |
| --- |
| import React, { useState } from 'react'  function CounterA() {    console.log("CounterA 호출됨");    const [value, setValue] = useState<number>(0);    return (      <div className="box">        <h1>CounterA</h1>        <button onClick={() => setValue(value + 1)}>          { value }        </button>      </div>    )  }  function CounterB() {    console.log("CounterB 호출됨");    const [value, setValue] = useState<number>(0);    return (      <div className="box">        <h1>CounterB</h1>        <button onClick={() => setValue(value + 1)}>          { value }        </button>      </div>    )  }  function GroupComponent() {    console.log('GroupComponent 호출됨');    const [value, setValue] = useState<number>(0);    return (      <div className='box'>        <h1>GroupComponent</h1>        <button onClick={() => setValue(value + 1)}>          { value }        </button>        <CounterA />        <CounterB />      </div>    )  }  export default GroupComponent; |

위와 같이 한 파일에 구현해도, 상태가 변경된 컴포넌트만 렌더링 되는 것은 전과 같다.

## 언제 렌더링 안되는가? #1

### App.tsx 수정

|  |
| --- |
| import React, { useState } from 'react'  import './App.css'  function App() {    const [arr, setArr] = useState<number[]>([0, 0]);    const increaseAt = (index: number) => {      arr[index] = arr[index] + 1;      setArr(arr);    };    return (      <div className="box">        <h1>App</h1>        <p>{ arr.join(", ") }</p>        <button onClick={() => increaseAt(0)} >a[0]++</button>        <button onClick={() => increaseAt(1)} >a[1]++</button>      </div>    );  }  export default App; |

### App.css 수정

|  |
| --- |
| h1 { font-size: 11pt; margin: 0; }  div.box { display: inline-block; margin: 10px; padding: 10px;    border: 1px solid gray; }  button { padding: 0.3em 1.3em; margin-right: 1em; } |

버튼을 클릭하면 increaseAt 함수가 호출되고

arr[index] = arr[index] + 1;

배열의 원소의 값이 변경되고

setArr(arr);

상태 변수 setter가 호출되지만

렌더링 되지 않는다.

arr 배열의 내부 값만 변경되었을 뿐, arr 배열은 이전과 같은 배열이다.

이 경우에는 렌더링 되지 않는다.

새 배열이 setArr 되어야 다시 렌더링 된다.

### App.tsx 수정

|  |
| --- |
| import React, { useState } from 'react'  import './App.css'  function App() {    const [arr, setArr] = useState<number[]>([0, 0]);    const increaseAt = (index: number) => {      arr[index] = arr[index] + 1;      setArr([...arr]);    };    return (      <div className="box">        <h1>App</h1>        <p>{ arr.join(", ") }</p>        <button onClick={() => increaseAt(0)} >a[0]++</button>        <button onClick={() => increaseAt(1)} >a[1]++</button>      </div>    );  }  export default App; |

[...arr]

새 배열이 생성되고, 그 새 배열에 arr 배열의 원소들이 채워진다.

...는 spread 연산자이다.

Graphical user interface, text, application

Description automatically generated

버튼을 클릭할 때 마다 렌더링 된다.

### 요약

컴포넌트 상태 값이 배열일 때, 배열의 원소 값만 바뀌어서는 렌더링되지 않는다.

상태 값이 새 배열로 바뀌어야, 즉 setter에 새 배열을 전달해야 렌더링 된다.

## 언제 렌더링 안되는가? #2

### App.tsx 수정

|  |
| --- |
| import React, { useState } from 'react'  import './App.css'  type Person = {    name: string,    age: number,    height: number  }  function App() {    const [person, setPerson] = useState<Person>({ name: '홍길동', age: 16, height: 180});    const increase = (key: string) => {      if (key==="height") person.height++;      else if (key==="age") person.age++;      setPerson(person);    };    return (      <div className="box">        <h1>App</h1>        <p>{ person.age }, { person.height }</p>        <button onClick={() => increase("age")} >age++</button>        <button onClick={() => increase("height")} >height++</button>      </div>    );  }  export default App; |

버튼을 클릭하면 increase 함수가 호출되고

if (key==="height") person.height++;

else if (key==="age") person.age++;

person 객체의 속성이 변경되고

setPerson(person);

상태 변수 setter가 호출되지만

렌더링 되지 않는다.

person 객체의 내부 값만 변경되었을 뿐, person 객체는 이전과 같은 객체이다.

이 경우에는 렌더링 되지 않는다.

새 객체가 setPerson 되어야 다시 렌더링 된다.

### App.tsx 수정

|  |
| --- |
| import React, { useState } from 'react'  import './App.css'  type Person = {    name: string,    age: number,    height: number  }  function App() {    const [person, setPerson] = useState<Person>({ name: '홍길동', age: 16, height: 180});    const increase = (key: string) => {      if (key==="height") person.height++;      else if (key==="age") person.age++;      setPerson({...person});    };    return (      <div className="box">        <h1>App</h1>        <p>{ person.age }, { person.height }</p>        <button onClick={() => increase("age")} >age++</button>        <button onClick={() => increase("height")} >height++</button>      </div>    );  }  export default App; |

{...person}

person 객체가 복제된어 새 객체가 만들어진다.

...는 spread 연산자이다.

Graphical user interface, text, application

Description automatically generated

버튼을 클릭할 때 마다 렌더링 된다.

### 요약

컴포넌트 상태 값이 객체일 때, 객체의 속성 값만 바뀌어서는 렌더링되지 않는다.

상태 값이 새 객체로 바뀌어야 렌더링 된다.

## 렌더링 안되는 이유

상태 값이 객체나 배열일 때,

렌더링 여부를 판단하기 위해 리액트는 객체나 배열의 참조만 비교하고,

그 내부 내용은 비교하지 않는다.

배열이나 객체의 내부 값들까지 전부 비교하는 것은 너무 느려서 비효율적이기 때문이다.

차라리 내부 값이 바뀔 때 마다, 배열이나 객체를 복제해서 set 하는 것이 더 빠르다.

### 요약

리액트 상태 값이 객체나 배열일 때,

그 내부 값이 바뀔 때 마다, 복제해서 set 해야 한다.

# todo 앱

## 화면 분할

컴포넌트 단위로 렌더링 되고, 상태 변경이 있는 컴포넌트만 렌더링 된다.

따라서 효율적인 렌더링을 위해서, 화면의 각 부분을 컴포넌트로 분할하는 것이 좋다.

input 태그가 포함된 컴포넌트의 경우,

키가 하나 눌러질 때마다 렌더링 되어야 한다.

따라서 입력 폼 부분을 컴포넌트로 분할해야 한다.

## 컴포넌트 분할

오로지 효율만 중시하는 군대는, 전황 상태를 감시하고 그 변화에 따라 작전을 변경하고 지시하는 장교와,

오로지 주어진 임무만 기계적으로 실행하는 사병의 역할을 분리한다.

### presentation component

화면을 담당한다.

화면 제어를 위한 상태만 일부 가지고 있고, 중요한 데이터나 상태를 가지고 있지 않다. (지휘자가 아님)

props로 전달된 데이터를 화면에 출력하는 부분만 담당하는, 가볍고 단순한 컴포넌트.

이벤트가 발생하면 지휘자에게 보고해야 한다.

보고하기 위한 function이 props로 주어져야 함.

### container component

중요한 데이타와 상태를 관리한다. (지휘자)

화면 출력은 하지 않는다. (화면 출력은 부하들의 역할)

이벤트 발생을 보고 받기 위한 function을 부하들에게 주어야 함. (props 전달)

화면에 출력할 데이터를 부하들에게 주어야 함. (props 전달)

부하들의 이벤트 보고를 받아서, 데이터를 수정하고 상태를 변경해야 함.

### 구현할 컴포넌트 목록

TodoContainer 지휘자 container component

TodoRoot 부하 presentation componente들 중에서 루트 컴포넌트

TodoInput 할일 입력 화면 컴포넌트

TodoList 할일 목록 화면 컴포넌트

TodoItem 할일 한 개 화면 컴포넌트

## 구현

### 프로젝트 생성

|  |
| --- |
| npx create-react-app todo --template typescript |

### types.ts

src/types.ts

|  |
| --- |
| export type Todo = {    id: number,    title: string,    done: boolean  };  export type AddTodoFunc = (text: string) => void; |

타입스크립트로 구현하려면 자료형(type)을 선언해야 한다.

데이터의 자료형

함수의 자료형

데이터나 함수를 주고 받는 컴포넌트들이 그 데이터나 함수의 자료형을 공유해야 한다.

따라서 자료형을 types.ts 파일에 구현하여 공유하자.

타입스크립트 파일 확장자

순수 타입스크립트 문법 파일: \*.ts

JSX 확장 문법 파일 즉 컴포넌트 파일: \*.tsx

export type Todo = {

  id: number,

  title: string,

  done: boolean

};

todo 항목 타입 선언

export type AddTodoFunc = (text: string) => void;

todo 입력 화면에서 저장 버튼이 클릭되면,

todo 입력 화면 컴포넌트는 컨테이너 컴포넌트에게 보고해야 한다.

보고를 받은 컨테이너 컴포넌트가, 할일 목록에 새 할일을 추가한다.

이렇게 보고하기 위한 함수의 타입이 AddTodoFunc 이다.

(text: string) 부분은 함수의 파라미터 변수의 타입이고

=> void 부분은 리턴값의 타입이다.

### TodoItem.tsx

src/TodoItem.tsx 할일 한개 화면 컴포넌트

|  |
| --- |
| import React from 'react'  import type \* as types from './types'  type Props = {    todo: types.Todo  }  function TodoItem({ todo }: Props) {    return (      <tr>        <td>{ todo.id }</td>        <td>{ todo.title }</td>      </tr>    )  }  export default TodoItem; |

import type \* as types from './types'

타입을 import 할 때는 import type으로 구현해야 한다.

type Props = {

  todo: types.Todo

}

TodoItem 컴포넌트의 props 타입

컴포넌트 함수의 파라미터는 props 이다. 다음과 같이 두 가지 방법으로 선언할 수 있다.

function TodoItem(props : Props)

이렇게 선언했다면, <td>{ props.todo.id }</td> 이렇게 사용해야 한다.

파라미터로 전달된 props 객체가 props 파라미터 변수에 대입된다.

function TodoItem({ todo }: Props)

이렇게 선언했다면, <td>{ todo.id }</td> 이렇게 사용해야 한다.

객체 구조 분해 할당 문법의 활용

파라미터로 전달된 props 객체가 쪼개져서, 그 객체의 todo 속성이 todo 파라미터 변수에 대입된다.

### TodoList.tsx

src/TodoList.tsx 할일 목록 화면 컴포넌트

|  |
| --- |
| import React from 'react'  import type \* as types from './types'  import TodoItem from './TodoItem'  type Props = {    todoList: types.Todo[]  }  function TodoList({ todoList }: Props) {    const trlist = todoList.map(todo => <TodoItem todo={todo} key={todo.id} />);    return (      <table>        <thead>          <tr><td>id</td><td>할일</td></tr>        </thead>        <tbody>          {trlist}        </tbody>      </table>    )  }  export default TodoList |

### TodoInput.tsx

src/TodoInput.tsx 할일 입력 화면 컴포넌트

|  |
| --- |
| import React, { ChangeEvent, useState } from 'react'  import type \* as types from './types'  type Props = {    addTodo : types.AddTodoFunc  }  function TodoInput({ addTodo }: Props) {    const [title, setTitle] = useState<string>('');    const onChange = (e: ChangeEvent<HTMLInputElement>) => setTitle(e.target.value);    const onClick = () => { addTodo(title); setTitle(''); }    return (      <div>        <input type='text' value={title} onChange={onChange} />        <button onClick={onClick}>추가</button>      </div>    )  }  export default TodoInput |

type Props = {

  addTodo : types.AddTodoFunc

}

새 할일이 입력되고 저장 버튼이 클릭되면, 이 이벤트가 컨테이너 컴포넌트에게 보고되어야 한다.

그 보고를 하기 위한 함수가 TodoInput 컴포넌트에 전달되어야 한다.

addTodo props 값이 바로 그 함수이다.

  const [title, setTitle] = useState<string>('');

  const onChange = (e: ChangeEvent<HTMLInputElement>) => setTitle(e.target.value);

<input type='text' value={title} onChange={onChange} />

새 할일 제목을 입력하기 위한 구현

const onClick = () => { addTodo(title); setTitle(''); }

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

추가 버튼이 클릭되면, addTodo 함수가 호출되어 컨테이너 컴포넌트에게 보고되고,

input 태그에 '' 값이 채워진다.

### TodoRoot.tsx

src/TodoRoot.tsx todo앱 화면을 담당하는 컴포넌트

|  |
| --- |
| import React from 'react'  import './TodoRoot.css'  import type \* as types from './types'  import TodoInput from './TodoInput';  import TodoList from './TodoList';  type Props = {    todoList: types.Todo[],    addTodo: types.AddTodoFunc  }  function TodoRoot({ todoList, addTodo }: Props) {    return (      <div className="todoRoot">        <h1>할일</h1>        <TodoInput addTodo={addTodo} />        <TodoList todoList={todoList} />      </div>    )  }  export default TodoRoot; |

### TodoRoot.css

|  |
| --- |
| .todoRoot { border: 1px solid gray; width: 300px; margin: 10px auto;    box-shadow: 5px 5px 5px #DDD; padding: 30px;  }  .todoRoot h1 { margin: 0  0 20px 0; }  .todoRoot input { padding: 6px; width: 220px; }  .todoRoot button { margin-left: 5px; padding: 3px 10px; }  .todoRoot table { border-collapse: collapse; margin-top: 20px; width: 100%; }  .todoRoot td { border: 1px solid gray; padding: 6px; }  .todoRoot thead { background-color: #eee; } |

### TodoContainer.tsx

src/TodoContainer.tsx 지휘자 컨테이너 컴포넌트

|  |
| --- |
| import React, { useState } from 'react'  import type \* as types from './types'  import TodoRoot from './TodoRoot';  const data : types.Todo[] = [    {id: 1, title: '프론트엔드 과제', done: false},    {id: 2, title: '백엔드 과제', done: true},    {id: 3, title: '웹프 시험공부', done: false}  ];  function TodoContainer() {    const [todoList, setTodoList] = useState<types.Todo[]>(data);    const addTodo : types.AddTodoFunc = (title: string) => {      const lastIndex = todoList.length - 1;      const lastId = todoList[lastIndex].id;      const todo = {id: lastId + 1, title: title, done: false};      setTodoList([...todoList, todo]);    }      return <TodoRoot todoList={todoList} addTodo={addTodo} />;  }  export default TodoContainer |

const [todoList, setTodoList] = useState<types.Todo[]>(data);

할일 목록 상태

할일 목록 전달 과정 TodoContainer -> TodoRoot -> TodoList

  const addTodo : types.AddTodoFunc = (title: string) => {

    const lastIndex = todoList.length - 1; // todoList 배열에서 마지막 항목의 인덱스

    const lastId = todoList[lastIndex].id; // 마지막 항목의 id

    const todo = {id: lastId + 1, title: title, done: false}; // 새 할일 객체 생성

    setTodoList([...todoList, todo]); // todoList 배열의 모든 원소와,

// 새 todo 객체가 들어있는 새 배열을 만들어서 상태 변경

  }

새 할일이 입력되고 저장 버튼이 클릭되었을 때 실행할 함수 구현

이 addTodo 함수가 TodoInput 컴포넌트에 전달되어야 함. (props로 전달)

전달과정 TodoContainer -> TodoRoot -> TodoInput

### App.tsx 수정

|  |
| --- |
| import React from 'react';  import TodoContainer from './TodoContainer';  function App() {    return (      <TodoContainer />    );  }  export default App; |

## 연습 문제

(1) 할일 항목에 체크 박스를 만들고, 해당 할일 항목의 done 값이 체크 박스에 채워지도록 구현하자.

(2) 체크가된 할일 항목은 회색 배경에 회색 글씨로 보이도록 서식을 구현하자.

(3) 체크 박스를 클릭하면, 해당 항목의 done 값이 변경되도록 구현하자.

이 변경 작업은 TodoContainer에 toggleTodo(id: number) 함수로 구현되어야 하고,

그 함수가 TodoItem 컴포넌트까지 전달되어야 한다.