React

**나의 첫번째 리액트 컴포넌트**

첫번째 리액트 컴포넌트를 만들어봅시다.

src 디렉터리에 Hello.js 라는 파일을 다음과 같이 작성해보세요.

**Hello.js**

import React from 'react';

function Hello() {

return <div>안녕하세요</div>

}

export default Hello;

리액트 컴포넌트를 만들 땐

import React from 'react';

를 통하여 리액트를 불러와주어야 합니다.

리액트 컴포넌트는 함수형태로 작성 할 수도 있고 클래스형태로도 작성 할 수 있습니다. 지금 단계에서는 함수로 작성하는 방법에 대해서만 알아보겠습니다.

리액트 컴포넌트에서는 XML 형식의 값을 반환해줄 수 있는데 이를 JSX 라고 부릅니다. 이에 대해선 다음번에 알아보도록 하겠습니다.

코드의 최하단

export default Hello;

이 코드는 Hello 라는 컴포넌트를 내보내겠다는 의미입니다. 이렇게 해주면 다른 컴포넌트에서 불러와서 사용 할 수 있습니다.

이 컴포넌트를 한번 App.js 에서 불러와서 사용해보겠습니다. (CodeSandbox 의 경우 index.js 에서 불러오세요. 그리고 App 컴포넌트 코드 아래부분은 건드리지마세요.)

**App.js**

import React from 'react';

import Hello from './Hello';

function App() {

return (

<div>

<Hello />

</div>

);

}

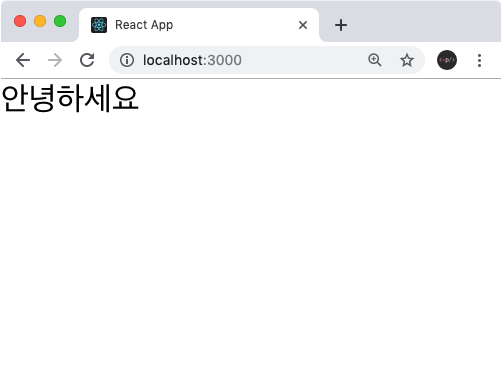
export default App;

상단에 있던

import logo from './logo.svg';

import './App.css';

는 SVG 파일을 불러오고, CSS 를 적용하는 코드인데 이는 현재 불필요하므로 제거해주었습니다.



컴포넌트는 일종의 UI 조각입니다. 그리고, 쉽게 재사용 할 수도 있습니다.

import React from 'react';

import Hello from './Hello';

function App() {

return (

<div>

<Hello />

<Hello />

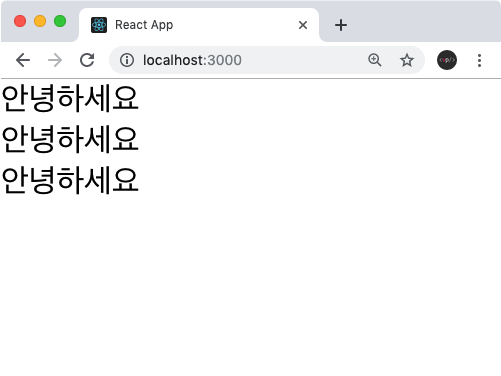
<Hello />

</div>

);

}

export default App;



이제, index.js 를 열어보세요.

이런 코드가 보일 것입니다.

import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import App from './App';

import \* as serviceWorker from './serviceWorker';

ReactDOM.render(<App />, document.getElementById('root'));

// If you want your app to work offline and load faster, you can change

// unregister() to register() below. Note this comes with some pitfalls.

// Learn more about service workers: https://bit.ly/CRA-PWA

serviceWorker.unregister();

여기서 ReactDOM.render 의 역할은 브라우저에 있는 실제 DOM 내부에 리액트 컴포넌트를 렌더링하겠다는 것을 의미합니다. id 가 root 인 DOM 을 선택하고 있는데, 이 DOM 이 어디있는지 볼까요?

public/index.html 을 열어보시면 내부에

<div id="root"></div>

을 찾아보실 수 있습니다.

결국, 리액트 컴포넌트가 렌더링 될 때에는, 렌더링된 결과물이 위 div 내부에 렌더링되는 것 입니다.

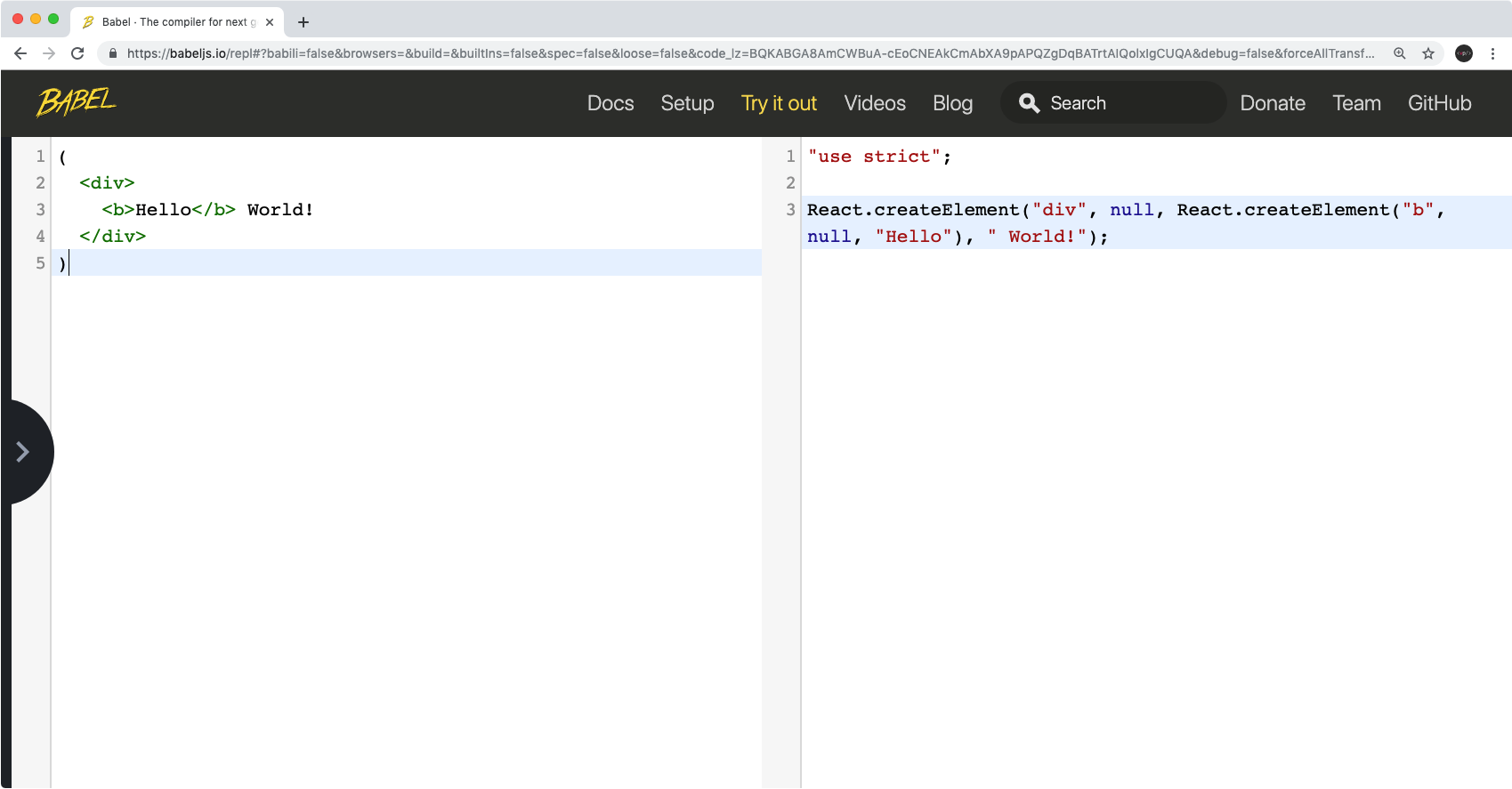
## JSX의 기본 규칙 알아보기

JSX 는 리액트에서 생김새를 정의할 때, 사용하는 문법입니다. 얼핏보면 HTML 같이 생겼지만 실제로는 JavaScript 입니다.

return <div>안녕하세요</div>;

리액트 컴포넌트 파일에서 XML 형태로 코드를 작성하면 babel 이 JSX 를 JavaScript 로 변환을 해줍니다.

어떻게 변환되는지 한번 예시를 볼까요?



<https://bit.ly/2wMpkk2>

Babel 은 자바스크립트의 문법을 확장해주는 도구입니다. 아직 지원되지 않는 최신 문법이나, 편의상 사용하거나 실험적인 자바스크립트 문법들을 정식 자바스크립트 형태로 변환해줌으로서 구형 브라우저같은 환경에서도 제대로 실행 할 수 있게 해주는 역할을 합니다.

JSX 가 JavaScript 로 제대로 변환이 되려면 지켜주어야 하는 몇가지 규칙이 있습니다. 다음 문법들을 준수해주시면 앞으로 리액트 컴포넌트를 개발함에 있어서 큰 어려움이 없을 것입니다!

### 꼭 닫혀야 하는 태그

태그는 꼭 닫혀있어야 합니다.

다음과 같은 코드는 오류가 발생하게 됩니다.

#### App.js

import React from 'react';

import Hello from './Hello';

function App() {

return (

<div>

<Hello />

<Hello />

<Hello />

<div>

</div>

);

}

export default App;

태그를 열었으면 꼭, <div></div> 이렇게 닫아주어야 합니다.

HTML 에서는 input 또는 br 태그를 사용 할 때 닫지 않고 사용하기도 합니다. 하지만 리액트에서는 그렇게 하면 안됩니다.

#### App.js

import React from 'react';

import Hello from './Hello';

function App() {

return (

<div>

<Hello />

<Hello />

<Hello />

<input>

<br>

</div>

);

}

export default App;

태그와 태그 사이에 내용이 들어가지 않을 때에는, Self Closing 태그 라는 것을 사용해야 합니다. 현재 Hello 컴포넌트를 사용 할 때에도 Self Closing 태그를 사용해주었는데요, 열리고, 바로 닫히는 태그를 의미합니다.

#### App.js

import React from 'react';

import Hello from './Hello';

function App() {

return (

<div>

<Hello />

<Hello />

<Hello />

<input />

<br />

</div>

);

}

export default App;

### 꼭 감싸져야하는 태그

두가 이상의 태그는 무조건 하나의 태그로 감싸져있어야 합니다. 한번 다음 코드를 작성해보세요.

#### App.js

import React from 'react';

import Hello from './Hello';

function App() {

return (

<Hello />

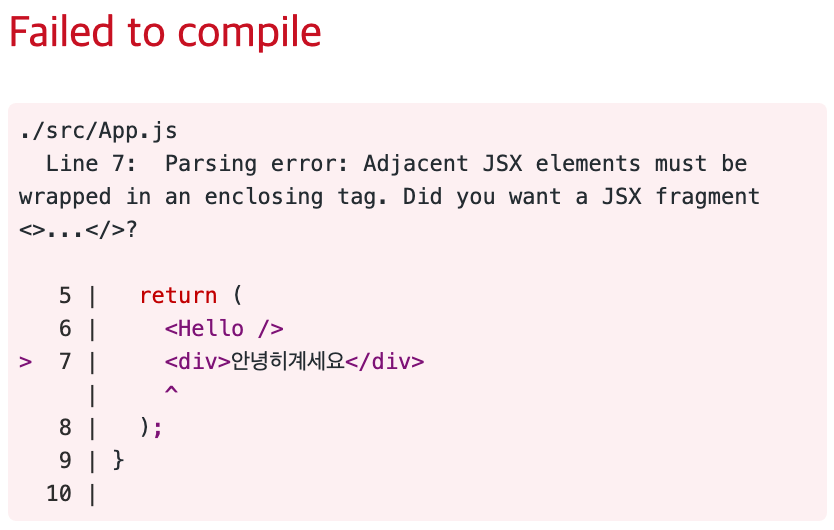
<div>안녕히계세요.</div>

);

}

export default App;

이런 코드는 오류가 발생하게 됩니다.



그 대신에 하나의 태그로 감싸주셔야 합니다.

import React from 'react';

import Hello from './Hello';

function App() {

return (

<div>

<Hello />

<div>안녕히계세요</div>

</div>

);

}

export default App;

하지만, 이렇게 단순히 감싸기 위하여 불필요한 div 로 감싸는게 별로 좋지 않은 상황도 있습니다. 예를 들어서 스타일 관련 설정을 하다가 복잡해지게 되는 상황도 올 수 있고, table 관련 태그를 작성 할 때에도 내용을 div 같은걸로 감싸기엔 애매하죠. 그럴 땐, 리액트의 Fragment 라는 것을 사용하면 됩니다.

#### App.js

import React from 'react';

import Hello from './Hello';

function App() {

return (

<>

<Hello />

<div>안녕히계세요</div>

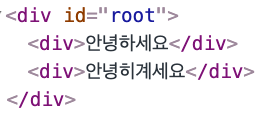
</>

);

}

export default App;

태그를 작성 할 때 이름 없이 작성을 하게 되면 Fragment 가 만들어지는데, Fragment 는 브라우저 상에서 따로 별도의 엘리먼트로 나타나지 않습니다.



### JSX 안에 자바스크립트 값 사용하기

JSX 내부에 자바스크립트 변수를 보여줘야 할 때에는 {} 으로 감싸서 보여줍니다.

import React from 'react';

import Hello from './Hello';

function App() {

const name = 'react';

return (

<>

<Hello />

<div>{name}</div>

</>

);

}

export default App;

### style 과 className

JSX 에서 태그에 style 과 CSS class 를 설정하는 방법은 HTML 에서 설정하는 방법과 다릅니다.

우선, 인라인 스타일은 객체 형태로 작성을 해야 하며, background-color 처럼 - 로 구분되어 있는 이름들은 backgroundColor 처럼 camelCase 형태로 네이밍 해주어야 합니다.

#### App.js

import React from 'react';

import Hello from './Hello';

function App() {

const name = 'react';

const style = {

backgroundColor: 'black',

color: 'aqua',

fontSize: 24, // 기본 단위 px

padding: '1rem' // 다른 단위 사용 시 문자열로 설정

}

return (

<>

<Hello />

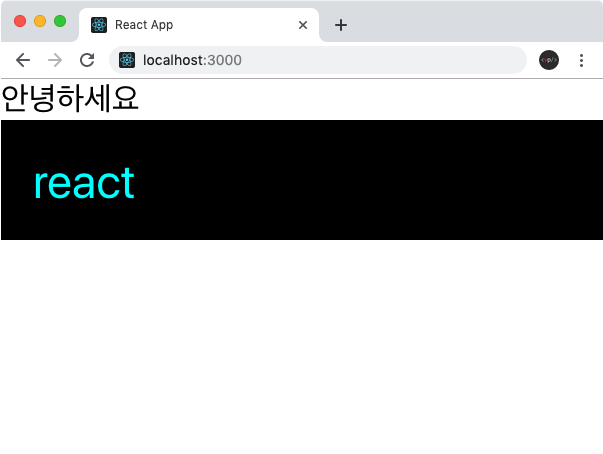
<div style={style}>{name}</div>

</>

);

}

export default App;



그리고, CSS class 를 설정 할 때에는 class= 가 아닌 className= 으로 설정을 해주어야 합니다. 한번, App.css 파일을 열어서 전체 내용을 지운 뒤 다음과 같이 수정해주세요.

#### App.css

.gray-box {

background: gray;

width: 64px;

height: 64px;

}

그 다음, App.js 를 다음과 같이 수정해보세요.

#### App.js

import React from 'react';

import Hello from './Hello';

import './App.css';

function App() {

const name = 'react';

const style = {

backgroundColor: 'black',

color: 'aqua',

fontSize: 24, // 기본 단위 px

padding: '1rem' // 다른 단위 사용 시 문자열로 설정

}

return (

<>

<Hello />

<div style={style}>{name}</div>

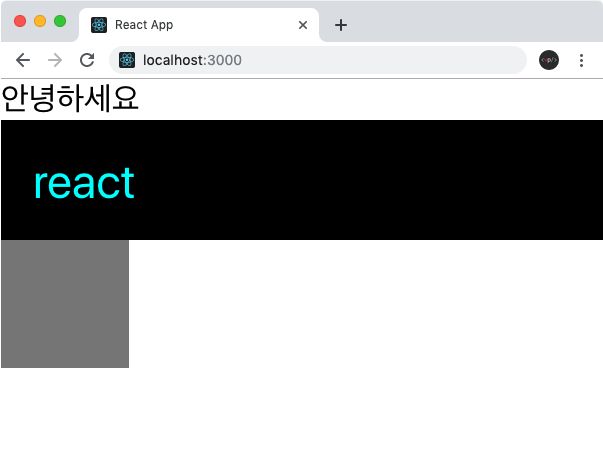
<div className="gray-box"></div>

</>

);

}

export default App;



### 주석

이제, JSX 에서 주석은 어떻게 작성하는지 알아봅시다.

JSX 내부의 주석은 {/\* 이런 형태로 \*/} 작성합니다.

한번 사용해볼까요?

#### App.js

import React from 'react';

import Hello from './Hello';

import './App.css';

function App() {

const name = 'react';

const style = {

backgroundColor: 'black',

color: 'aqua',

fontSize: 24, // 기본 단위 px

padding: '1rem' // 다른 단위 사용 시 문자열로 설정

}

return (

<>

{/\* 주석은 화면에 보이지 않습니다 \*/}

/\* 중괄호로 감싸지 않으면 화면에 보입니다 \*/

<Hello

/>

<div style={style}>{name}</div>

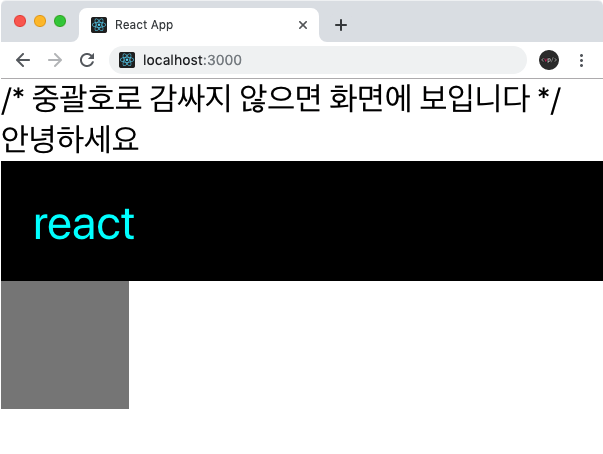
<div className="gray-box"></div>

</>

);

}

export default App;



추가적으로, 열리는 태그 내부에서는 // 이런 형태로도 주석 작성이 가능합니다.

import React from 'react';

import Hello from './Hello';

import './App.css';

function App() {

const name = 'react';

const style = {

backgroundColor: 'black',

color: 'aqua',

fontSize: 24, // 기본 단위 px

padding: '1rem' // 다른 단위 사용 시 문자열로 설정

}

return (

<>

{/\* 주석은 화면에 보이지 않습니다 \*/}

/\* 중괄호로 감싸지 않으면 화면에 보입니다 \*/

<Hello

// 열리는 태그 내부에서는 이렇게 주석을 작성 할 수 있습니다.

/>

<div style={style}>{name}</div>

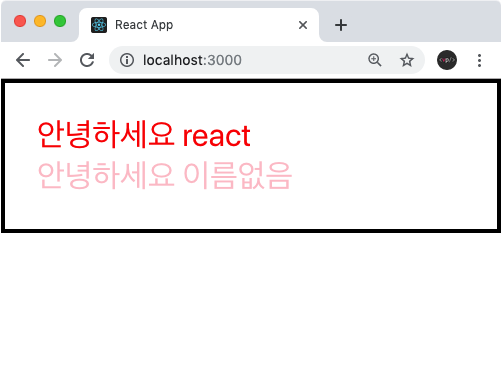
<div className="gray-box"></div>

</>

);

}

export default App;



## 조건부 렌더링

조건부 렌더링이란, 특정 조건에 따라 다른 결과물을 렌더링 하는 것을 의미합니다.

예를 들어서, App 컴포넌트에서 Hello 컴포넌트를 사용 할 때, isSpecial 이라는 props 를 설정해볼게요.

#### App.js

import React from 'react';

import Hello from './Hello';

import Wrapper from './Wrapper';

function App() {

return (

<Wrapper>

<Hello name="react" color="red" isSpecial={true}/>

<Hello color="pink" />

</Wrapper>

)

}

export default App;

여기서 true 는 자바스크립트 값이기 때문에 중괄호로 감싸주었습니다.

그리고, Hello 컴포넌트에서는 isSpecial 이 true 이냐 false 이냐에 따라서 컴포넌트의 좌측에 \* 표시를 보여줘보겠습니다.

이를 처리하는 가장 기본적인 방법은, 삼항연산자를 사용하는 것 입니다.

#### Hello.js

import React from 'react';

function Hello({ color, name, isSpecial }) {

return (

<div style={{ color }}>

{ isSpecial ? <b>\*</b> : null }

안녕하세요 {name}

</div>

);

}

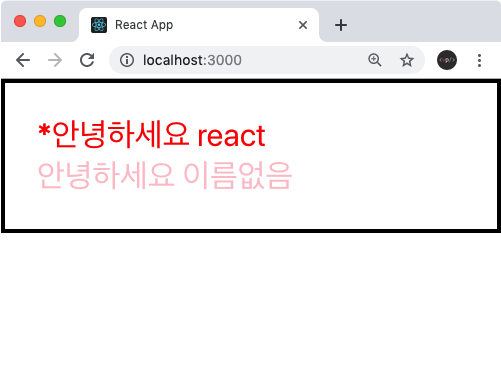
Hello.defaultProps = {

name: '이름없음'

}

export default Hello;

isSpecial 값이 true 라면 <b>\*</b> 를, 그렇지 않다면 null 을 보여주도록 했습니다. 참고로 JSX 에서 null, false, undefined 를 렌더링하게 된다면 아무것도 나타나지 않게 됩니다.



보통 삼항연산자를 사용한 조건부 렌더링을 주로 특정 조건에 따라 보여줘야 하는 내용이 다를 때 사용합니다.

지금은 내용이 달라지는게 아니라, 단순히 특정 조건이 true 이면 보여주고, 그렇지 않다면 숨겨주고 있는데요, 이러한 상황에서는 && 연산자를 사용해서 처리하는 것이 더 간편합니다.

코드를 다음과 같이 수정해보세요.

#### Hello.js

import React from 'react';

function Hello({ color, name, isSpecial }) {

return (

<div style={{ color }}>

{isSpecial && <b>\*</b>}

안녕하세요 {name}

</div>

);

}

Hello.defaultProps = {

name: '이름없음'

}

export default Hello;

isSpecial && <b>\*</b> 의 결과는 isSpecial 이 false 일땐 false 이고, isSpecial이 true 일 땐 <b>\*</b> 가 됩니다. 이게 왜 그렇게 되는지 모르겠다면 [단축 평가 논리 계산법](https://learnjs.vlpt.us/useful/03-short-circuiting.html) 을 배워보세요.

### props 값 설정을 생략하면 ={true}

컴포넌트의 props 값을 설정하게 될 때 만약 props 이름만 작성하고 값 설정을 생략한다면, 이를 true 로 설정한 것으로 간주합니다.

예를 들자면,

#### App.js

import React from 'react';

import Hello from './Hello';

import Wrapper from './Wrapper';

function App() {

return (

<Wrapper>

<Hello name="react" color="red" isSpecial />

<Hello color="pink"/>

</Wrapper>

);

}

export default App;

이렇게 isSpecial 이름만 넣어주면 isSpecial={true} 와 동일한 의미입니다.

## useState 를 통해 컴포넌트에서 바뀌는 값 관리하기

지금까지 우리가 리액트 컴포넌트를 만들 때는, 동적인 부분이 하나도 없었습니다. 값이 바뀌는 일이 없었죠. 이번에는 컴포넌트에서 보여줘야 하는 내용이 사용자 인터랙션에 따라 바뀌어야 할 때 어떻게 구현할 수 있는지에 대하여 알아보겠습니다.

리액트 16.8 이전 버전에서는 함수형 컴포넌트에서는 상태를 관리할 수 없었는데요, 리액트 16.8 에서 Hooks 라는 기능이 도입되면서 함수형 컴포넌트에서도 상태를 관리할 수 있게 되었습니다. 이번에는 useState 라는 함수를 사용해보게 되는데, 이게 바로 리액트의 Hooks 중 하나입니다.

정말 진부한 예제인, 버튼을 누르면 숫자가 바뀌는 Counter 컴포넌트를 만들어볼게요.

src 디렉터리에 Counter.js 를 다음과 같이 작성해보세요.

#### Counter.js

import React from 'react';

function Counter() {

return (

<div>

<h1>0</h1>

<button>+1</button>

<button>-1</button>

</div>

);

}

export default Counter;

그 다음엔 App 에서 Counter 를 렌더링 해보세요.

#### App.js

import React from 'react';

import Counter from './Counter';

function App() {

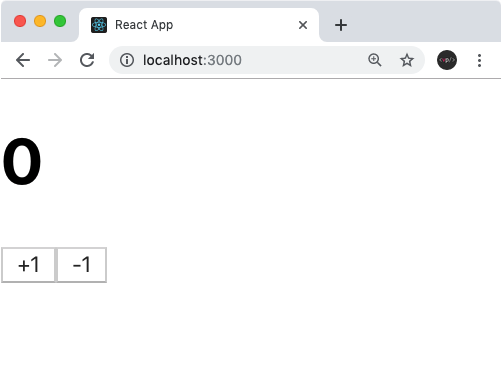
return (

<Counter />

);

}

export default App;



이런 UI 가 보여졌나요?

### 이벤트 설정

이제, Counter 에서 버튼이 클릭되는 이벤트가 발생 했을 때, 특정 함수가 호출되도록 설정을 해보겠습니다.

Counter 컴포넌트를 다음과 같이 수정해보세요.

#### Counter.js

import React from 'react';

function Counter() {

const onIncrease = () => {

console.log('+1')

}

const onDecrease = () => {

console.log('-1');

}

return (

<div>

<h1>0</h1>

<button onClick={onIncrease}>+1</button>

<button onClick={onDecrease}>-1</button>

</div>

);

}

export default Counter;

여기서 onIncrease 와 onDecrease 는 화살표 함수를 사용하여 구현을 해주었는데요, 화살표 함수에 대해서 잘 모르신다면 이 [링크](https://learnjs.vlpt.us/basics/05-function.html#%ED%99%94%EC%82%B4%ED%91%9C-%ED%95%A8%EC%88%98) 를 참고하세요.

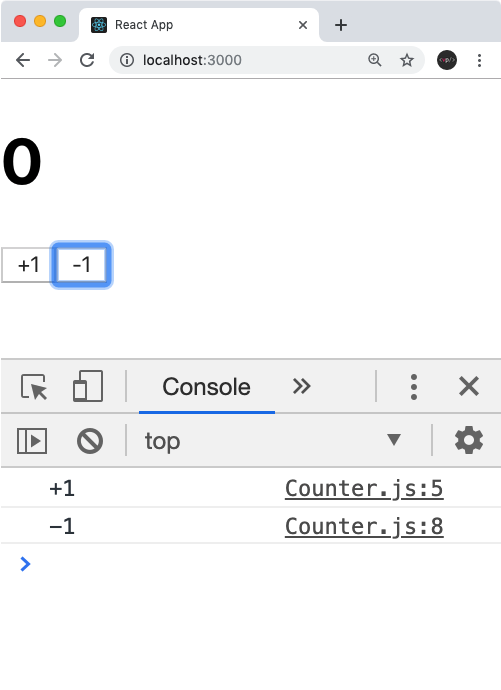
함수를 만들고, button 의 onClick 으로 각 함수를 연결해주었습니다. 리액트에서 엘리먼트에 이벤트를 설정해줄때에는 on이벤트이름={실행하고싶은함수} 형태로 설정해주어야 합니다.

여기서 주의하셔야 하는 점은, 함수형태를 넣어주어야 하지, 함수를 다음과 같이 실행하시면 안됩니다.

onClick={onIncrease()}

이렇게 하면 렌더링되는 시점에서 함수가 호출되버리기 때문입니다. 이벤트를 설정할때에는 함수타입의 값을 넣어주어야 한다는 것, 주의해주세요.

코드를 저장 후 버튼들을 눌러보세요. 콘솔에 메시지들이 잘 출력되나요?



### 동적인 값 끼얹기, useState

컴포넌트에서 동적인 값을 상태(state)라고 부릅니다. 리액트에 useState 라는 함수가 있는데요, 이것을 사용하면 컴포넌트에서 상태를 관리 할 수 있습니다.

Counter 컴포넌트를 다음과 같이 수정해보세요.

#### Counter.js

import React, { useState } from 'react';

function Counter() {

const [number, setNumber] = useState(0);

const onIncrease = () => {

setNumber(number + 1);

}

const onDecrease = () => {

setNumber(number - 1);

}

return (

<div>

<h1>{number}</h1>

<button onClick={onIncrease}>+1</button>

<button onClick={onDecrease}>-1</button>

</div>

);

}

export default Counter;

import React, { useState } from 'react';

이 코드는 리액트 패키지에서 useState 라는 함수를 불러와줍니다.

const [number, setNumber] = useState(0);

useState 를 사용 할 때에는 상태의 기본값을 파라미터로 넣어서 호출해줍니다. 이 함수를 호출해주면 배열이 반환되는데요, 여기서 첫번째 원소는 현재 상태, 두번째 원소는 Setter 함수입니다.

원래는 다음과 같이 해야하지만,

const numberState = useState(0);

const number = numberState[0];

const setNumber = numberState[1];

[배열 비구조화 할당](https://learnjs.vlpt.us/useful/06-destructuring.html#%EB%B0%B0%EC%97%B4-%EB%B9%84%EA%B5%AC%EC%A1%B0%ED%99%94-%ED%95%A0%EB%8B%B9)을 통하여 각 원소를 추출해준것입니다.

const onIncrease = () => {

setNumber(number + 1);

}

const onDecrease = () => {

setNumber(number - 1);

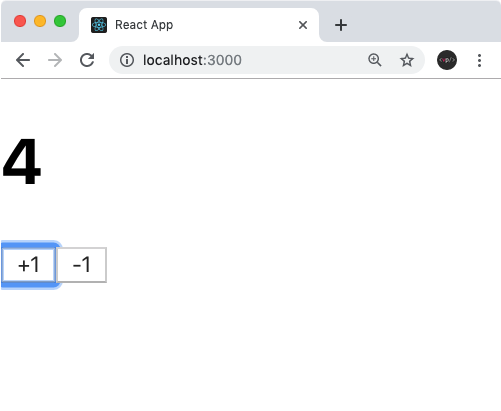
}

Setter 함수는 파라미터로 전달 받은 값을 최신 상태로 설정해줍니다.

<h1>{number}</h1>

h1 태그에서는 이제 0 이 아닌 {number} 값을 보여주어야 합니다.

코드를 다 수정 후, 버튼들을 눌러보세요. 숫자가 잘 바뀌고 있나요?



### 함수형 업데이트

지금은 Setter 함수를 사용 할 때, 업데이트 하고 싶은 새로운 값을 파라미터로 넣어주고 있는데요, 그 대신에 기존 값을 어떻게 업데이트 할 지에 대한 함수를 등록하는 방식으로도 값을 업데이트 할 수 있습니다.

Counter 컴포넌트를 다음과 같이 수정해보세요.

#### Counter.js

import React, { useState } from 'react';

function Counter() {

const [number, setNumber] = useState(0);

const onIncrease = () => {

setNumber(prevNumber => prevNumber + 1);

}

const onDecrease = () => {

setNumber(prevNumber => prevNumber - 1);

}

return (

<div>

<h1>{number}</h1>

<button onClick={onIncrease}>+1</button>

<button onClick={onDecrease}>-1</button>

</div>

);

}

export default Counter;

onIncrease 와 onDecrease 에서 setNumber 를 사용 할 때 그 다음 상태를 파라미터로 넣어준것이 아니라, 값을 업데이트 하는 함수를 파라미터로 넣어주었습니다.

함수형 업데이트는 주로 나중에 컴포넌트를 최적화를 하게 될 때 사용하게 됩니다. 지금 당장은 함수형 업데이트란게 있는 것 정도만 이해해두시면 충분합니다. 이게 왜 최적화랑 관련이 되어있는지는 나중에 알아보도록 할게요.

## input 상태 관리하기

이번에는 리액트에서 사용자가 입력 할 수 있는 input 태그의 상태를 관리하는 방법을 알아보겠습니다.

우선, src 디렉터리에 InputSample.js 라는 파일을 생성하세요.

#### InputSample.js

import React from 'react';

function InputSample() {

return (

<div>

<input />

<button>초기화</button>

<div>

<b>값: </b>

</div>

</div>

);

}

export default InputSample;

그 다음에는, 이 컴포넌트를 App 에서 렌더링하세요.

#### App.js

import React from 'react';

import InputSample from './InputSample';

function App() {

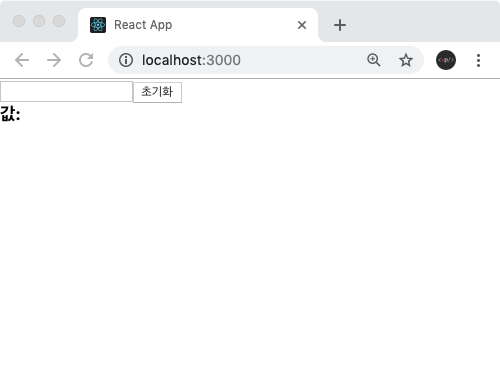
return (

<InputSample />

);

}

export default App;



input 에 입력하는 값이 하단에 나타나게 하고, 초기화 버튼을 누르면 input 이 값이 비워지도록 구현을 해보겠습니다.

이번에도, useState 를 사용합니다. 이번에는 input 의 onChange 라는 이벤트를 사용하는데요, 이벤트에 등록하는 함수에서는 이벤트 객체 e 를 파라미터로 받아와서 사용 할 수 있는데 이 객체의 e.target 은 이벤트가 발생한 DOM 인 input DOM 을 가르키게됩니다. 이 DOM 의 value 값, 즉 e.target.value 를 조회하면 현재 input 에 입력한 값이 무엇인지 알 수 있습니다.

이 값을 useState 를 통해서 관리를 해주면 됩니다.

한번 코드를 다음과 같이 수정해보세요.

#### InputSample.js

import React, { useState } from 'react';

function InputSample() {

const [text, setText] = useState('');

const onChange = (e) => {

setText(e.target.value);

};

const onReset = () => {

setText('');

};

return (

<div>

<input onChange={onChange} value={text} />

<button onClick={onReset}>초기화</button>

<div>

<b>값: {text}</b>

</div>

</div>

);

}

export default InputSample;

input 의 상태를 관리할 때에는 input 태그의 value 값도 설정해주는 것이 중요합니다. 그렇게 해야, 상태가 바뀌었을때 input 의 내용도 업데이트 됩니다.

## 여러개의 input 상태 관리하기

지난 튜토리얼에서 우리는 input 상태를 관리하는 방법에 대하여 알아보았는데요, 이번에는 input 이 여러개일때는 어떻게 관리해야 하는지 알아보겠습니다.

우선 지난번에 만든 InputSample 에서 새로운 input 을 보여주세요.

이번에는 input 이 비어져있을 때 인풋에 대한 설명을 보여주는 placeholder 값도 설정해보겠습니다.

기존에 만들었던 상태는 지워주시고, onChange 와 onReset 함수는 비워주세요.

#### InputSample.js

import React, { useState } from 'react';

function InputSample() {

const onChange = (e) => {

};

const onReset = () => {

};

return (

<div>

<input placeholder="이름" />

<input placeholder="닉네임" />

<button onClick={onReset}>초기화</button>

<div>

<b>값: </b>

이름 (닉네임)

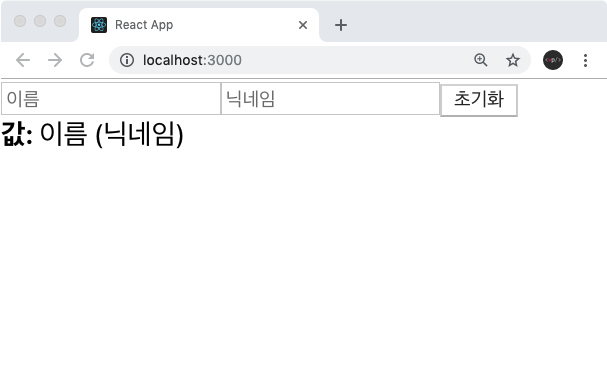
</div>

</div>

);

}

export default InputSample;



input 의 개수가 여러개가 됐을때는, 단순히 useState 를 여러번 사용하고 onChange 도 여러개 만들어서 구현 할 수 있습니다. 하지만 그 방법은 가장 좋은 방법은 아닙니다. 더 좋은 방법은, input 에 name 을 설정하고 이벤트가 발생했을 때 이 값을 참조하는 것입니다. 그리고, useState 에서는 문자열이 아니라 객체 형태의 상태를 관리해주어야 합니다.

한번 InputSample 컴포넌트를 다음과 같이 수정해보세요.

#### InputSample.js

import React, { useState } from 'react';

function InputSample() {

const [inputs, setInputs] = useState({

name: '',

nickname: ''

});

const { name, nickname } = inputs; // 비구조화 할당을 통해 값 추출

const onChange = (e) => {

const { value, name } = e.target; // 우선 e.target 에서 name 과 value 를 추출

setInputs({

...inputs, // 기존의 input 객체를 복사한 뒤

[name]: value // name 키를 가진 값을 value 로 설정

});

};

const onReset = () => {

setInputs({

name: '',

nickname: '',

})

};

return (

<div>

<input name="name" placeholder="이름" onChange={onChange} value={name} />

<input name="nickname" placeholder="닉네임" onChange={onChange} value={nickname}/>

<button onClick={onReset}>초기화</button>

<div>

<b>값: </b>

{name} ({nickname})

</div>

</div>

);

}

export default InputSample;

리액트 상태에서 객체를 수정해야 할 때에는,

inputs[name] = value;

이런식으로 직접 수정하면 안됩니다.

그 대신에, 새로운 객체를 만들어서 새로운 객체에 변화를 주고, 이를 상태로 사용해주어야 합니다.

setInputs({

...inputs,

[name]: value

});

여기서 사용한 ... 문법은 spread 문법입니다. 객체의 내용을 모두 "펼쳐서" 기존 객체를 복사해주는데요, 이 문법을 잘 모르신다면 이 [링크](https://learnjs.vlpt.us/useful/07-spread-and-rest.html) 를 참고하세요.

이러한 작업을, "불변성을 지킨다" 라고 부릅니다. 불변성을 지켜주어야만 리액트 컴포넌트에서 상태가 업데이트가 됐음을 감지 할 수 있고 이에 따라 필요한 리렌더링이 진행됩니다. 만약에 inputs[name] = value 이런식으로 기존 상태를 직접 수정하게 되면, 값을 바꿔도 리렌더링이 되지 않습니다.

추가적으로, 리액트에서는 불변성을 지켜주어야만 컴포넌트 업데이트 성능 최적화를 제대로 할 수 있습니다. 컴포넌트 최적화에 대해서는 나중에 더 자세히 알아보도록 하겠습니다.

지금은 이것만 기억하세요. 리액트에서 객체를 업데이트하게 될 때에는 기존 객체를 직접 수정하면 안되고, 새로운 객체를 만들어서, 새 객체에 변화를 주어야 됩니다.

## useRef 로 특정 DOM 선택하기

JavaScript 를 사용 할 때에는, 우리가 특정 DOM 을 선택해야 하는 상황에 getElementById, querySelector 같은 DOM Selector 함수를 사용해서 DOM 을 선택합니다.

리액트를 사용하는 프로젝트에서도 가끔씩 DOM 을 직접 선택해야 하는 상황이 발생 할 때도 있습니다. 예를 들어서 특정 엘리먼트의 크기를 가져와야 한다던지, 스크롤바 위치를 가져오거나 설정해야된다던지, 또는 포커스를 설정해줘야된다던지 등 정말 다양한 상황이 있겠죠. 추가적으로 Video.js, JWPlayer 같은 HTML5 Video 관련 라이브러리, 또는 D3, chart.js 같은 그래프 관련 라이브러리 등의 외부 라이브러리를 사용해야 할 때에도 특정 DOM 에다 적용하기 때문에 DOM 을 선택해야 하는 상황이 발생 할 수 있습니다.

그럴 땐, 리액트에서 ref 라는 것을 사용합니다.

함수형 컴포넌트에서 ref 를 사용 할 때에는 useRef 라는 Hook 함수를 사용합니다. 클래스형 컴포넌트에서는 콜백 함수를 사용하거나 React.createRef 라는 함수를 사용하는데, 이에 대해서는 나중에 클래스 컴포넌트를 배울 때 다뤄보도록 하겠습니다. (참고로, 클래스 컴포넌트를 나중에 다루는 이유는, 이제 별로 중요하지 않기 때문입니다.)

우리가 만든 InputSample 에서는 초기화 버튼을 누르면 포커스가 초기화 버튼에 그대로 남아있게 됩니다.

한번, 초기화 버튼을 클릭했을 때 이름 input 에 포커스가 잡히도록 useRef 를 사용하여 기능을 구현해보겠습니다.

#### InputSample.js

import React, { useState, useRef } from 'react';

function InputSample() {

const [inputs, setInputs] = useState({

name: '',

nickname: ''

});

const nameInput = useRef();

const { name, nickname } = inputs; // 비구조화 할당을 통해 값 추출

const onChange = e => {

const { value, name } = e.target; // 우선 e.target 에서 name 과 value 를 추출

setInputs({

...inputs, // 기존의 input 객체를 복사한 뒤

[name]: value // name 키를 가진 값을 value 로 설정

});

};

const onReset = () => {

setInputs({

name: '',

nickname: ''

});

nameInput.current.focus();

};

return (

<div>

<input

name="name"

placeholder="이름"

onChange={onChange}

value={name}

ref={nameInput}

/>

<input

name="nickname"

placeholder="닉네임"

onChange={onChange}

value={nickname}

/>

<button onClick={onReset}>초기화</button>

<div>

<b>값: </b>

{name} ({nickname})

</div>

</div>

);

}

export default InputSample;

useRef() 를 사용하여 Ref 객체를 만들고, 이 객체를 우리가 선택하고 싶은 DOM 에 ref 값으로 설정해주어야 합니다. 그러면, Ref 객체의 .current 값은 우리가 원하는 DOM 을 가르키게 됩니다.

위 예제에서는 onReset 함수에서 input 에 포커스를 하는 focus() DOM API 를 호출해주었습니다.

이제 브라우저에서 인풋에 값을 입력한 다음에 초기화를 눌러보세요. 이름 input 에 포커스가 잘 잡히나요?

## 배열 렌더링하기

이번에는 리액트에서 배열을 렌더링하는 방법을 알아보겠습니다.

이러 배열이 있다고 가정해봅시다.

const users = [

{

id: 1,

username: 'velopert',

email: 'public.velopert@gmail.com'

},

{

id: 2,

username: 'tester',

email: 'tester@example.com'

},

{

id: 3,

username: 'liz',

email: 'liz@example.com'

}

];

만약에 이 내용을 컴포넌트로 렌더링한다면 어떻게 해야 할까요?

일단, 가장 기본적인 방법으론 비효율적이지만, 그냥 그대로 코드를 작성하는 것 입니다.

src 디렉터리에 UserList.js 컴포넌트를 다음과 같이 만들어보세요.

#### UserList.js

import React from 'react';

function UserList() {

const users = [

{

id: 1,

username: 'velopert',

email: 'public.velopert@gmail.com'

},

{

id: 2,

username: 'tester',

email: 'tester@example.com'

},

{

id: 3,

username: 'liz',

email: 'liz@example.com'

}

];

return (

<div>

<div>

<b>{users[0].username}</b> <span>({users[0].email})</span>

</div>

<div>

<b>{users[1].username}</b> <span>({users[1].email})</span>

</div>

<div>

<b>{users[2].username}</b> <span>({users[1].email})</span>

</div>

</div>

);

}

export default UserList;

그런데, 재사용되는 코드를 일일히 넣는게 별로 좋지 않으니, 컴포넌트를 재사용 할 수 있도록 새로 만들어주겠습니다.

참고로, 한 파일에 여러개의 컴포넌트를 선언해도 괜찮습니다.

#### UserList.js

import React from 'react';

function User({ user }) {

return (

<div>

<b>{user.username}</b> <span>({user.email})</span>

</div>

);

}

function UserList() {

const users = [

{

id: 1,

username: 'velopert',

email: 'public.velopert@gmail.com'

},

{

id: 2,

username: 'tester',

email: 'tester@example.com'

},

{

id: 3,

username: 'liz',

email: 'liz@example.com'

}

];

return (

<div>

<User user={users[0]} />

<User user={users[1]} />

<User user={users[2]} />

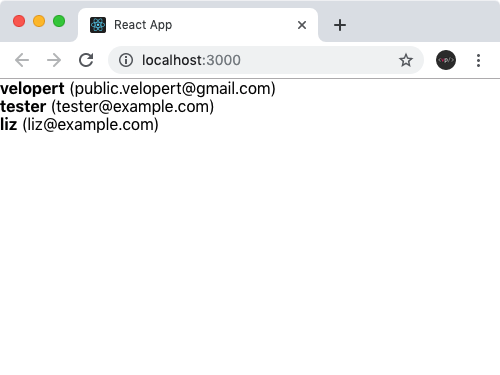
</div>

);

}

export default UserList;

한번 컴포넌트를 App 에서 렌더링 해봅시다.



배열이 고정적이라면 상관없겟지만, 배열의 인덱스를 하나하나 조회해가면서 렌더링하는 방법은 동적인 배열을 렌더링하지 못합니다.

동적인 배열을 렌더링해야 할 때에는 자바스크립트 배열의 내장함수 map() 을 사용합니다. 이 함수를 잘 모르신다면 이 [링크](https://learnjs.vlpt.us/basics/09-array-functions.html#map) 를 참고하세요.

map() 함수는 배열안에 있는 각 원소를 변환하여 새로운 배열을 만들어줍니다. 리액트에서 동적인 배열을 렌더링해야 할 때는 이 함수를 사용하여 일반 데이터 배열을 리액트 엘리먼트로 이루어진 배열로 변환해주면 됩니다.

UserList 컴포넌트를 다음과 같이 수정해보세요.

#### UserList.js

import React from 'react';

function User({ user }) {

return (

<div>

<b>{user.username}</b> <span>({user.email})</span>

</div>

);

}

function UserList() {

const users = [

{

id: 1,

username: 'velopert',

email: 'public.velopert@gmail.com'

},

{

id: 2,

username: 'tester',

email: 'tester@example.com'

},

{

id: 3,

username: 'liz',

email: 'liz@example.com'

}

];

return (

<div>

{users.map(user => (

<User user={user} />

))}

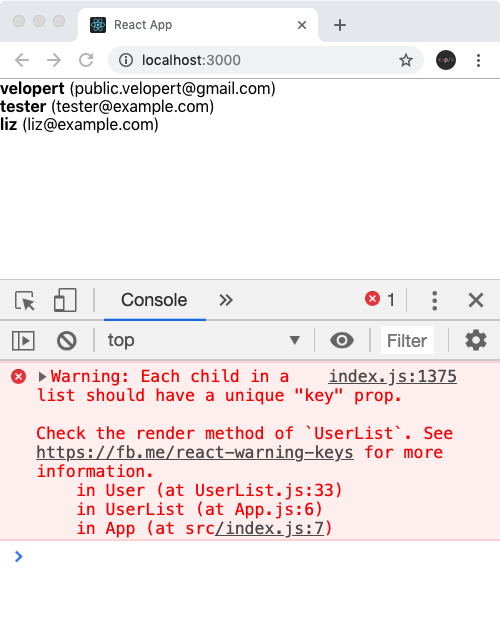
</div>

);

}

export default UserList;

간단하지요? 이렇게 하면 배열의 모든 원소가 렌더링됩니다. 하지만, 여기서 끝이 아닙니다. 브라우저에서 콘솔을 열어보면 다음과 같은 에러가 보여질것입니다.



리액트에서 배열을 렌더링 할 때에는 key 라는 props 를 설정해야합니다. key 값은 각 원소들마다 가지고 있는 고유값으로 설정을 해야합니다. 지금의 경우엔 id 가 고유 값이지요.

#### UserList.js

import React from 'react';

function User({ user }) {

return (

<div>

<b>{user.username}</b> <span>({user.email})</span>

</div>

);

}

function UserList() {

const users = [

{

id: 1,

username: 'velopert',

email: 'public.velopert@gmail.com'

},

{

id: 2,

username: 'tester',

email: 'tester@example.com'

},

{

id: 3,

username: 'liz',

email: 'liz@example.com'

}

];

return (

<div>

{users.map(user => (

<User user={user} key={user.id} />

))}

</div>

);

}

export default UserList;

만약 배열 안의 원소가 가지고 있는 고유한 값이 없다면 map() 함수를 사용 할 때 설정하는 콜백함수의 두번째 파라미터 index 를 key 로 사용하시면 됩니다.

<div>

{users.map((user, index) => (

<User user={user} key={index} />

))}

</div>

만약에 배열을 렌더링 할 때 key 설정을 하지 않게된다면 기본적으로 배열의 index 값을 key 로 사용하게 되고, 아까 봤었던 경고메시지가 뜨게 됩니다. 이렇게 경고 메시지가 뜨는 이유는, 각 고유 원소에 key 가 있어야만 배열이 업데이트 될 때 효율적으로 렌더링 될 수 있기 때문입니다.

### key 의 존재유무에 따른 업데이트 방식

예를 들어서 다음과 같은 배열이 있다고 가정해봅시다.

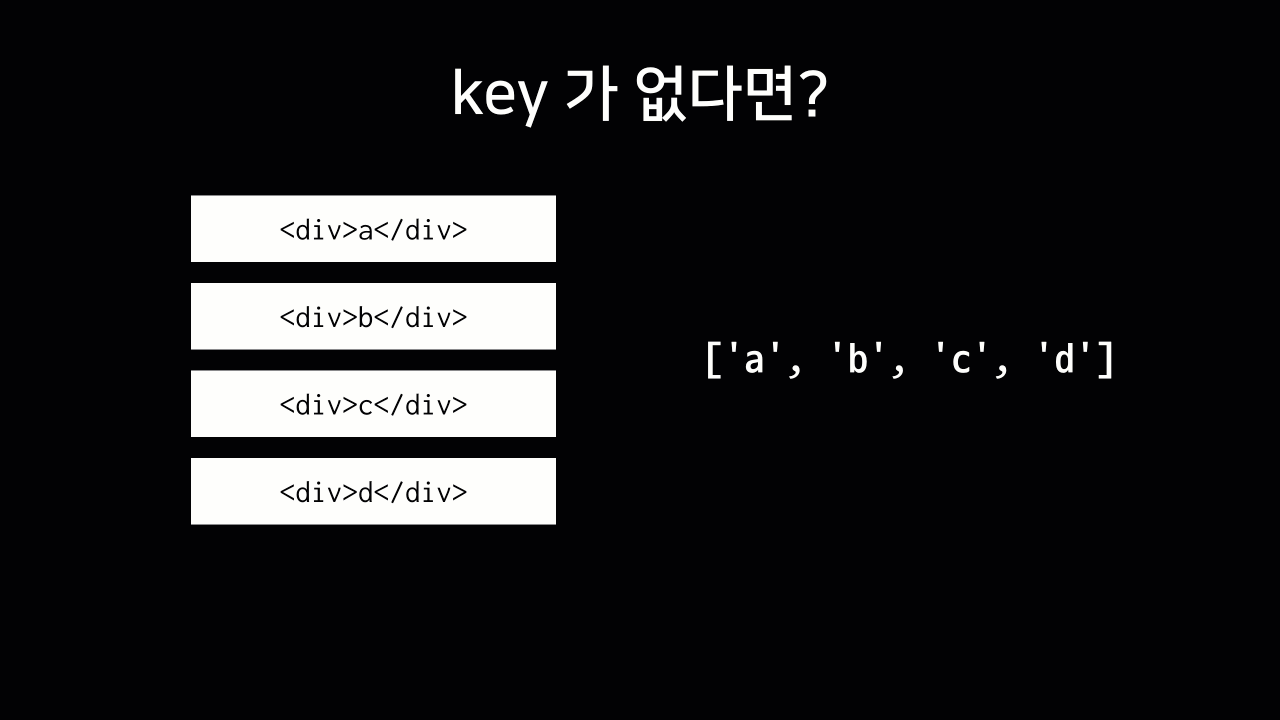
const array = ['a', 'b', 'c', 'd'];

그리고 위 배열을 다음과 같이 렌더링한다고 가정해보겠습니다.

array.map(item => <div>{item}</div>);

위 배열의 b 와 c 사이에 z 를 삽입하게 된다면, 리렌더링을 하게 될 때 <div>b</div> 와 <div>c</div> 사이에 새 div 태그를 삽입을 하게 되는 것이 아니라, 기존의 c 가 z 로바뀌고, d 는 c 로 바뀌고, 맨 마지막에 d 가 새로 삽입됩니다.

그 다음에 a 를 제거하게 된다면, 기존의 a 가 b 로 바뀌고, b 는 z 로 바뀌고, z는 c로 바뀌고, c는 d 로바뀌고, 맨 마지막에 있는 d 가 제거됩니다.



비효율적이지요? 하지만, key 가 있다면 이 작업은 개선됩니다.

객체에 다음과 같이 key 로 사용 할 수 있는 고유 값이 있고

[

{

id: 0,

text: 'a'

},

{

id: 1,

text: 'b'

},

{

id: 2,

text: 'c'

},

{

id: 3,

text: 'd'

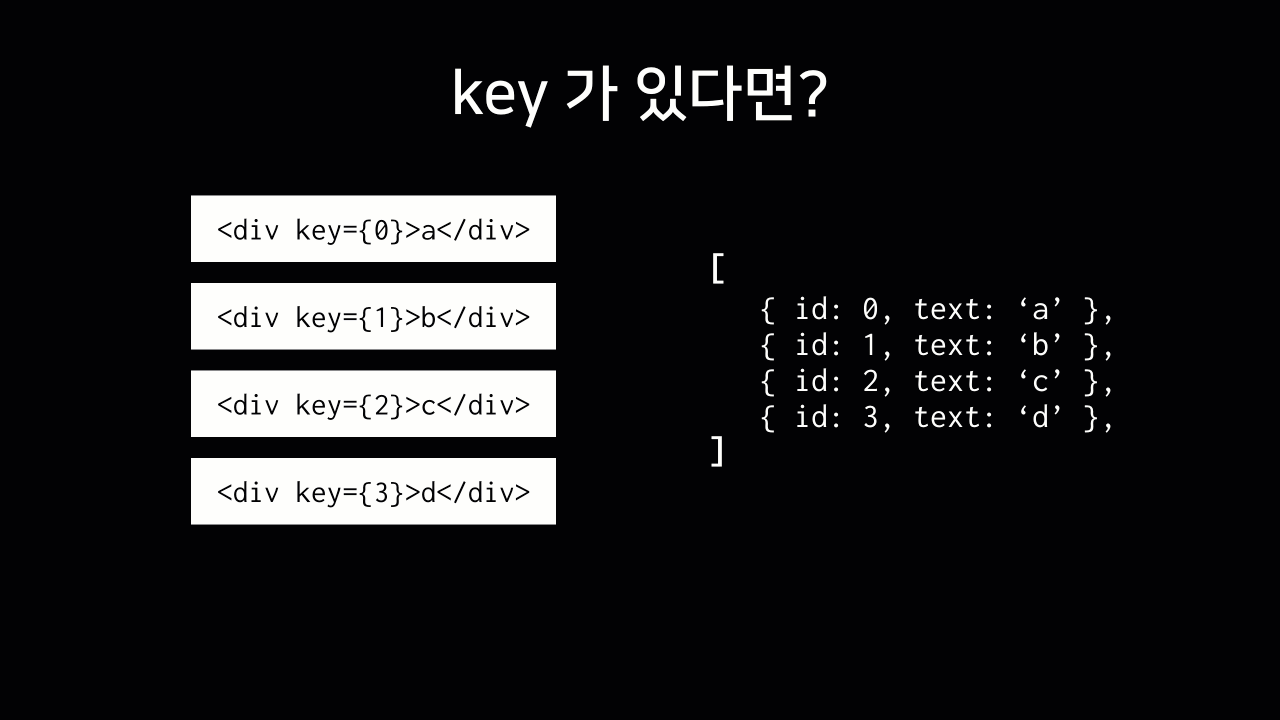
}

];

다음과 같이 렌더링이 된다면

array.map(item => <div key={item.id}>{item.text}</div>);

배열이 업데이트 될 떄 key 가 없을 때 처럼 비효율적으로 업데이트 하는 것이 아니라, 수정되지 않는 기존의 값은 그대로 두고 원하는 곳에 내용을 삽입하거나 삭제합니다.



때문에, 배열을 렌더링 할 때에는 고유한 key 값이 있는것이 중요하며, 만약에 배열안에 중복되는 key 가 있을 때에는 렌더링시에 오류메시지가 콘솔에 나타나게 되며, 업데이트가 제대로 이루어지지 않게 됩니다.