공통 컴포넌트 (예:)

모든 내장 브라우저 컴포넌트 (예: `<div>`)는 공통의 Props와 이벤트를 지원합니다.

- 포커스 이벤트 처리
- 키보드 이벤트 처리

레퍼런스

공통 컴포넌트 (예: `<div>`)

```
<div className="wrapper">Some content</div>
```

아래 예시를 참고하세요.

Props

아래의 특별한 React Props는 내장된 모든 컴포넌트에서 지원합니다.

- `children`: React 노드(요소, 문자열, 숫자, Portal, `null`, `undefined`, 불리언 타입과 같은 빈 노드, 또는 다른 React 노드의 배열) 입니다. 컴포넌트 내부의 콘텐츠를 지정합니다. JSX를 사용하면 일반적으로 `<div><span /></div>` 처럼 태그를 중첩하여 `children` Prop을 암묵적으로 지정합니다.

- `dangerouslySetInnerHTML`: 원시 HTML 문자열이 포함된 `{ __html: '<p>some html</p>' }` 형식의 객체입니다. DOM 노드의 `innerHTML` 프로퍼티를 덮어쓰고 전달된 HTML을 내부에 표시합니다. 이것은 매우 주의해서 사용해야 합니다. 내부 HTML을 신뢰할 수 없는 경우 (예: 사용자 데이터를 기반으로 하는 경우) XSS 취약점이 발생할 수 있습니다. `dangerouslySetInnerHTML` 에 대해 더 알아보려면 읽어보세요.

- `ref`: `useRef` 나 `createRef` 의 `ref` 객체, 또는 `ref` 콜백 함수거나 legacy refs의 문자열입니다. 해당 `ref` 는 해당 노드의 DOM 요소로 채워집니다. `ref` 를 사용하여 DOM을 조작하는 방법에 대해 더 자세히 알아보세요.

- `suppressContentEditableWarning`: 불리언 타입입니다. `true` 일 때, 일반적으로 같이 사용하지 않는 `children` 과 `contentEditable={true}` 가 모두 존재하는 요소에 대해 React에서 발생하는 경고를 나타내지 않습니다. 이는 `contentEditable` 콘텐츠를 수동으로 관리하는 텍스트 입력 라이브러리를 빌드할 때 사용됩니다.

- `suppressHydrationWarning`: 불리언 타입입니다. 서버 렌더링을 사용할 때, 일반적으로 서버와 클라이언트가 서로 다른 콘텐츠를 렌더링하면 경고가 표시됩니다. 일부 드문 사례(예: 타

임스탬프)에서는 정확한 일치를 보장하기가 매우 어렵거나 불가능합니다. `suppressHydrationWarning`를 `true`로 설정하면, React는 해당 요소의 어트리뷰트와 콘텐츠가 일치하지 않아도 경고를 표시하지 않습니다. 이는 한 단계의 깊이에서만 작동하며, 탈출구로 사용하기 위한 것입니다. 과도하게 사용하지 마세요. Suppressing Hydration 오류에 대해서 읽어보세요.

- `style`: `{ fontWeight: 'bold', margin: 20 }`와 같이 CSS 스타일이 있는 객체입니다. DOM의 `style` 프로퍼티에서 `fontWeight` 대신 `font-weight`로 작성하는 것과 마찬가지로 CSS 프로퍼티의 이름도 `camelCase`로 작성해야 합니다. 또한 문자열이나 숫자를 값으로 전달할 수 있습니다. `width: 100`와 같은 숫자를 전달한다면 React는 단위가 없는 프로퍼티가 아니라면 자동으로 `px` ("픽셀")로 값을 추가합니다. `style`은 스타일 값을 미리 알 수 없는 동적 스타일에만 사용하는 것을 권장합니다. 그 외의 경우에는 `className`을 사용하여 일반 CSS 클래스를 사용하는 것이 더 효율적입니다. `className`과 `style`에 대해서 더 자세히 알아보세요.

아래의 표준 DOM Props는 내장된 모든 컴포넌트에서 지원합니다.

- `accessKey`: 문자열 타입입니다. 요소의 바로 가기 키를 지정합니다. 일반적으로 권장하지 않습니다.
- `aria-*`: ARIA 속성을 사용하면 이 요소에 대한 접근성 트리 정보를 지정할 수 있습니다. 전체적인 레퍼런스는 ARIA 어트리뷰트를 참조하세요. React에서 모든 ARIA 어트리뷰트의 이름은 HTML에서의 이름과 완전히 동일합니다.
- `autoCapitalize`: 문자열 타입입니다. 사용자의 입력을 대문자로 표시할지 여부와 방법을 지정합니다.
- `className`: 문자열 타입입니다. 요소의 CSS 클래스 이름을 지정합니다. CSS 스타일 적용에 대해 자세히 알아보세요.
- `contentEditable`: 불리언 타입입니다. `true`일 때 브라우저는 사용자가 렌더링 된 요소를 직접 편집할 수 있도록 합니다. 이는 Lexical과 같은 서식이 있는 텍스트 입력 라이브러리를 구현하는 데 사용됩니다. React는 사용자가 편집한 후에 React가 그 내용을 업데이트할 수 없기 때문에 `contentEditable={true}`가 있는 요소에 React의 자식을 전달하려고 하면 경고를 표시합니다.
- `data-*`: 데이터 속성을 사용하면 요소에 일부 문자열 데이터를 첨부할 수 있습니다. (예: `data-fruit="banana"`) React에서는 일반적으로 Props나 State에서 데이터를 읽어오기 때문에 일반적으로 사용되지는 않습니다.
- `dir`: `'ltr'` 또는 `'rtl'` 입니다. 요소의 텍스트 방향을 지정합니다.
- `draggable`: 불리언 타입입니다. 요소의 드래그 가능 여부를 지정합니다. HTML 드래그 앤 드롭 API의 일부입니다.

- `enterKeyHint` : 문자열 타입입니다. 가상 키보드의 입력 키에 어떤 동작을 표시할지 지정합니다.
- `htmlFor` : 문자열 타입입니다. `<label>` 이나 `<output>` 의 경우 label을 일부 동작에 연결할 수 있습니다. 이는 HTML 어트리뷰트의 `for` 과 동일합니다. React는 HTML 어트리뷰트의 이름 대신 `htmlFor` 와 같은 표준 DOM 프로퍼티의 이름을 사용합니다.
- `hidden` : 불리언 혹은 문자열 타입입니다. 요소를 숨길지에 대한 여부를 지정합니다.
- `id` : 문자열 타입입니다. 요소의 고유 식별자를 지정하여 나중에 찾거나 다른 요소와 연결하는 데 사용할 수 있습니다. 동일한 컴포넌트의 여러 인스턴스 간의 충돌을 피하고자 `useId` 로 생성합니다.
- `is` : 문자열 타입입니다. 지정하게 되면 컴포넌트가 사용자 정의 요소처럼 작동합니다.
- `inputMode` : 문자열 타입입니다. 표시할 키보드의 종류(예시: 텍스트, 숫자 또는 전화번호)를 지정합니다.
- `itemProp` : 문자열 타입입니다. 구조화된 데이터 크롤러에 대해 요소가 나타내는 속성을 지정합니다.
- `lang` : 문자열 타입입니다. 요소의 언어를 지정합니다.
- `onAnimationEnd` : `AnimationEvent` 핸들러 함수입니다. CSS 애니메이션이 완료될 때 발생합니다.
- `onAnimationEndCapture` : 캡처 단계에서 실행되는 `onAnimationEnd` 의 버전입니다.
- `onAnimationIteration` : `AnimationEvent` 핸들러 함수입니다. CSS 애니메이션의 반복이 끝나고 다른 애니메이션이 시작될 때 발생합니다.
- `onAnimationIterationCapture` : 캡처 단계에서 실행되는 `onAnimationIteration` 의 버전입니다.
- `onAnimationStart` : `AnimationEvent` 핸들러 함수입니다. CSS 애니메이션이 시작될 때 발생합니다.
- `onAnimationStartCapture` : `onAnimationStart` 입니다. 그러나 캡처 단계에서 실행됩니다.
- `onAuxClick` : `MouseEvent` 핸들러 함수입니다. 기본 포인터가 아닌 버튼을 클릭했을 때 발생합니다.
- `onAuxClickCapture` : `onAuxClick` 의 캡처 단계에서 실행되는 버전입니다.
- `onBeforeInput` : `InputEvent` 핸들러 함수입니다. 편집할 수 있는 요소의 값이 수정되기 전에 발생합니다. React는 아직 네이티브 `beforeinput` 이벤트를 *사용하지 않습니다.* 대신 다른 이벤트를 사용하여 폴리필을 시도합니다.
- `onBeforeInputCapture` : `onBeforeInput` 의 캡처 단계에서 실행되는 버전입니다.
- `onBlur` : `FocusEvent` 핸들러 함수입니다. 요소가 포커싱을 잃었을 때 발생합니다. 브라우저에 내장된 `blur` 이벤트와 달리 React에서는 `onBlur` 이벤트가 버블링을 발생시킵니다.
- `onBlurCapture` : `onBlur` 의 캡처 단계에서 실행되는 버전입니다.

- `onClick`: `MouseEvent` 핸들러 함수입니다. 포인팅 디바이스에서 기본 버튼이 클릭 되었을 때 발생합니다.
- `onClickCapture`: `onClick` 의 캡처 단계에서 실행되는 버전입니다.
- `onCompositionStart`: `CompositionEvent` 핸들러 함수입니다. 입력 메서드 편집기가 새로운 구성 세션을 시작할 때 발생합니다.
- `onCompositionStartCapture`: `onCompositionStart` 의 캡처 단계에서 실행되는 버전입니다.
- `onCompositionEnd`: `CompositionEvent` 핸들러 함수입니다. 입력 메서드 편집기가 구성 세션을 완료하거나 취소할 때 발생합니다.
- `onCompositionEndCapture`: `onCompositionEnd` 의 캡처 단계에서 실행되는 버전입니다.
- `onCompositionUpdate`: `CompositionEvent` 핸들러 함수입니다. 입력 메서드 편집기에 새로운 문자가 입력되면 발생합니다.
- `onCompositionUpdateCapture`: `onCompositionUpdate` 의 캡처 단계에서 실행되는 버전입니다.
- `onContextMenu`: `MouseEvent` 핸들러 함수입니다. 컨텍스트 메뉴를 열려고 할 때 발생합니다.
- `onContextMenuCapture`: `onContextMenu` 의 캡처 단계에서 실행되는 버전입니다.
- `onCopy`: `ClipboardEvent` 핸들러 함수입니다. 클립보드에 무언가를 복사하려고 할 때 발생합니다.
- `onCopyCapture`: `onCopy` 의 캡처 단계에서 실행되는 버전입니다.
- `onCut`: `ClipboardEvent` 핸들러 함수입니다. 클립보드에서 무언가를 잘라내려고 할 때 발생합니다.
- `onCutCapture`: `onCut` 의 캡처 단계에서 실행되는 버전입니다.
- `onDoubleClick`: `MouseEvent` 핸들러 함수입니다. 두 번 클릭하면 발생합니다. 브라우저의 `dblclick` 이벤트에 해당합니다.
- `onDoubleClickCapture`: `onDoubleClick` 의 캡처 단계에서 실행되는 버전입니다.
- `onDrag`: `DragEvent` 핸들러 함수입니다. 무언가를 드래그하는 동안 실행됩니다.
- `onDragCapture`: `onDrag` 의 캡처 단계에서 실행되는 버전입니다.
- `onDragEnd`: `DragEvent` 핸들러 함수입니다. 드래그를 멈추면 발생합니다.
- `onDragEndCapture`: `onDragEnd` 의 캡처 단계에서 실행되는 버전입니다.
- `onDragEnter`: `DragEvent` 핸들러 함수입니다. 드래그한 콘텐츠가 유효한 드롭 대상에 들어가면 발생합니다.
- `onDragEnterCapture`: `onDragEnter` 의 캡처 단계에서 실행되는 버전입니다.
- `onDragOver`: `DragEvent` 핸들러 함수입니다. 드래그된 콘텐츠를 드래그하는 동안 유효한 드롭 대상에서 발생합니다. 드롭을 허용하려면 여기서 `e.preventDefault()` 를 호출해야 합니

다.

- `onDragOverCapture`: `onDragOver` 의 캡처 단계에서 실행되는 버전입니다.
- `onDragStart`: `DragEvent` 핸들러 함수입니다. 요소를 드래그하기 시작할 때 발생합니다.
- `onDragStartCapture`: `onDragStart` 의 캡처 단계에서 실행되는 버전입니다.
- `onDrop`: `DragEvent` 핸들러 함수입니다. 유효한 드롭 대상에 무언가를 떨어뜨리면 발동합니다.
- `onDropCapture`: `onDrop` 의 캡처 단계에서 실행되는 버전입니다.
- `onFocus`: `FocusEvent` 핸들러 함수입니다. 요소가 포커싱을 얻었을 때 발생합니다. 브라우저에 내장된 `focus` 이벤트와 달리 React에서는 `onFocus` 이벤트가 버블링을 발생시킵니다.
- `onFocusCapture`: `onFocus` 의 캡처 단계에서 실행되는 버전입니다.
- `onGotPointerCapture`: `PointerEvent` 핸들러 함수입니다. 요소가 프로그래밍 방식으로 포인터를 캡처할 때 발생합니다.
- `onGotPointerCaptureCapture`: `onGotPointerCapture` 의 캡처 단계에서 실행되는 버전입니다.
- `onKeyDown`: `KeyboardEvent` 핸들러 함수입니다. 키를 누르면 실행됩니다.
- `onKeyDownCapture`: `onKeyDown` 의 캡처 단계에서 실행되는 버전입니다.
- `onKeyPress`: `KeyboardEvent` 핸들러 함수입니다. 사용되지 않습니다. 대신 `onKeyDown` 또는 `onBeforeInput` 을 사용하세요.
- `onKeyPressCapture`: `onKeyPress` 의 캡처 단계에서 실행되는 버전입니다.
- `onKeyUp`: `KeyboardEvent` 핸들러 함수입니다. 키를 놓으면 실행됩니다.
- `onKeyUpCapture`: `onKeyUp` 의 캡처 단계에서 실행되는 버전입니다.
- `onLostPointerCapture`: `PointerEvent` 핸들러 함수입니다. 요소가 포인터 캡처를 중지하면 발생합니다.
- `onLostPointerCaptureCapture`: `onLostPointerCapture` 의 캡처 단계에서 실행되는 버전입니다.
- `onMouseDown`: `MouseEvent` 핸들러 함수입니다. 마우스 포인터를 눌렀을 때 실행됩니다.
- `onMouseDownCapture`: `onMouseDown` 의 캡처 단계에서 실행되는 버전입니다.
- `onMouseEnter`: `MouseEvent` 핸들러 함수입니다. 마우스 포인터가 요소 내부로 이동할 때 발생합니다. 캡처 단계가 없습니다. 대신 `onMouseLeave` 와 `onMouseEnter` 는 떠나는 요소에서 입력되는 요소로 전파됩니다.
- `onMouseLeave`: `MouseEvent` 핸들러 함수입니다. 마우스 포인터가 요소 외부로 이동하면 발생합니다. 캡처 단계가 없습니다. 대신 `onMouseLeave` 와 `onMouseEnter` 는 떠나는 요소에서 입력되는 요소로 전파됩니다.
- `onMouseMove`: `MouseEvent` 핸들러 함수입니다. 마우스 포인터의 좌표를 변경할 때 발생합니다.

- `onMouseMoveCapture`: `onMouseMove` 의 캡처 단계에서 실행되는 버전입니다.
- `onMouseOut`: `MouseEvent` 핸들러 함수입니다. 마우스 포인터가 요소 외부로 이동하거나 하위 요소로 이동하면 발생합니다.
- `onMouseOutCapture`: `onMouseOut` 의 캡처 단계에서 실행되는 버전입니다.
- `onMouseUp`: `MouseEvent` 핸들러 함수입니다. 마우스 포인터에서 손을 떼면 발생합니다.
- `onMouseUpCapture`: `onMouseUp` 의 캡처 단계에서 실행되는 버전입니다.
- `onPointerCancel`: `PointerEvent` 핸들러 함수입니다. 브라우저가 포인터와 상호작용을 취소할 때 발생합니다.
- `onPointerCancelCapture`: `onPointerCancel` 의 캡처 단계에서 실행되는 버전입니다.
- `onPointerDown`: `PointerEvent` 핸들러 함수입니다. 포인터가 활성화되면 발생합니다.
- `onPointerDownCapture`: `onPointerDown` 의 캡처 단계에서 실행되는 버전입니다.
- `onPointerEnter`: `PointerEvent` 핸들러 함수입니다. 포인터가 요소 내부로 이동할 때 발생합니다. 캡처 단계가 없습니다. 대신 `onPointerLeave` 와 `onPointerEnter` 는 떠나는 요소에서 입력되는 요소로 전파됩니다.
- `onPointerLeave`: `PointerEvent` 핸들러 함수입니다. 포인터가 요소 내부로 이동할 때 발생합니다. 캡처 단계가 없습니다. 대신 `onPointerLeave` 와 `onPointerEnter` 는 떠나는 요소에서 입력되는 요소로 전파됩니다.
- `onPointerMove`: `PointerEvent` 핸들러 함수입니다. 포인터의 좌표를 변경할 때 발생합니다.
- `onPointerMoveCapture`: `onPointerMove` 의 캡처 단계에서 실행되는 버전입니다.
- `onPointerOut`: `PointerEvent` 핸들러 함수입니다. 포인터가 요소 외부로 이동하거나 포인터 상호 작용이 취소되는 경우, 그리고 그 외 몇 가지 이유로 인해 발생합니다.
- `onPointerOutCapture`: `onPointerOut` 의 캡처 단계에서 실행되는 버전입니다.
- `onPointerUp`: `PointerEvent` 핸들러 함수입니다. 포인터가 더 이상 활성화되지 않을 때 발생합니다.
- `onPointerUpCapture`: `onPointerUp` 의 캡처 단계에서 실행되는 버전입니다.
- `onPaste`: `ClipboardEvent` 핸들러 함수입니다. 사용자가 클립보드에서 붙여 넣으려고 할 때 발생합니다.
- `onPasteCapture`: `onPaste` 의 캡처 단계에서 실행되는 버전입니다.
- `onScroll`: `Event` 핸들러 함수입니다. 요소를 스크롤 할 때 발생합니다. 이 이벤트는 버블링이 발생하지 않습니다.
- `onScrollCapture`: `onScroll` 의 캡처 단계에서 실행되는 버전입니다.
- `onSelect`: `Event` 핸들러 함수입니다. 입력 변경과 같이 편집할 수 있는 요소 내부에서 선택되면 실행됩니다. React는 `onSelect` 이벤트를 `contentEditable={true}` 요소에도 작동하

도록 확장합니다. 또한 React는 빈 선택과 (선택에 영향을 줄 수 있는) 편집 시에도 발동되도록 확장합니다.

- `onSelectCapture` : `onSelect` 의 캡처 단계에서 실행되는 버전입니다.
- `onTouchCancel` : `TouchEvent` 핸들러 함수입니다. 브라우저가 터치 상호작용을 취소할 때 발생합니다.
- `onTouchCancelCapture` : `onTouchCancel` 의 캡처 단계에서 실행되는 버전입니다.
- `onTouchEnd` : `TouchEvent` 핸들러 함수입니다. 하나 이상의 터치 포인트가 사라지면 발생합니다.
- `onTouchEndCapture` : `onTouchEnd` 의 캡처 단계에서 실행되는 버전입니다.
- `onTouchMove` : `TouchEvent` 핸들러 함수입니다. 하나 이상의 터치 포인트가 이동하면 발생합니다.
- `onTouchMoveCapture` : `onTouchMove` 의 캡처 단계에서 실행되는 버전입니다.
- `onTouchStart` : `TouchEvent` 핸들러 함수입니다. 하나 이상의 터치 포인트가 위치하면 발생합니다.
- `onTouchStartCapture` : `onTouchStart` 의 캡처 단계에서 실행되는 버전입니다.
- `onTransitionEnd` : `TransitionEvent` 헨들러 함수입니다. CSS 전환을 완료하면 발생합니다.
- `onTransitionEndCapture` : `onTransitionEnd` 의 캡처 단계에서 실행되는 버전입니다.
- `onWheel` : `WheelEvent` 핸들러 함수입니다. 휠 버튼을 돌리면 발생합니다.
- `onWheelCapture` : `onWheel` 의 캡처 단계에서 실행되는 버전입니다.
- `role` : 문자열 타입입니다. 보조 기술에 대한 요소의 역할을 명시적으로 지정합니다.
- `slot` : 문자열 타입입니다. 그림자 DOM을 사용할 때 슬롯의 이름을 지정합니다. React에서는 일반적으로 JSX를 프로퍼티로 전달하여 동일한 패턴을 얻을 수 있습니다. (예시: `<Layout left={<Sidebar />} right={<Content />} />` .
- `spellCheck` : 불리언 또는 null 타입입니다. `true` 또는 `false` 로 설정하여 맞춤법 검사를 활성화 또는 비활성화합니다.
- `tabIndex` : 숫자 타입입니다. 기본 탭 버튼 동작을 재정의합니다. `-1` 과 `0` 이외의 값은 사용하지 마십시오.
- `title` : 문자열 타입입니다. 요소의 툴팁 텍스트를 지정합니다.
- `translate` : `'yes'` 나 `'no'` 중 하나입니다. `'no'` 를 전달하면 요소의 콘텐츠가 번역에서 제외됩니다.

사용자 정의 어트리뷰트를 Props로 전달할 수도 있습니다. (예: `mycustomprop="someValue"`) 이는 서드파티 라이브러리와 통합할 때 유용할 수 있습니다. 사용자 정의 어트리뷰트의 이름은 소

문자여야 하며 `on` 으로 시작하지 않아야 합니다. 값은 문자열로 변환됩니다. `null` 또는 `undefined` 를 전달하면 사용자 정의 어트리뷰트가 제거됩니다.

다음의 이벤트는 `<form>` 요소에 대해서만 발생합니다.

- `onReset`: `Event` 핸들러 함수입니다. 폼을 재설정할 때 발생합니다.
- `onResetCapture`: `onReset` 의 캡처 단계에서 실행되는 버전입니다.
- `onSubmit`: `Event` 핸들러 함수입니다. 폼을 제출할 때 발생합니다.
- `onSubmitCapture`: `onSubmit` 의 캡처 단계에서 실행되는 버전입니다.

다음의 이벤트는 `<dialog>` 요소에 대해서만 발생합니다. 그리고 브라우저 이벤트와 달리 React에서는 버블링이 발생합니다.

- `onCancel`: `Event` 핸들러 함수입니다. 사용자가 대화상자를 닫으려고 할 때 발생합니다.
- `onCancelCapture`: `onCancel` 의 캡처 단계에서 실행되는 버전입니다.
- `onClose`: `Event` 핸들러 함수입니다. 대화 상자가 닫혔을 때 발생합니다.
- `onCloseCapture`: `onClose` 의 캡처 단계에서 실행되는 버전입니다.

다음의 이벤트는 `<details>` 요소에 대해서만 발생합니다. 그리고 브라우저 이벤트와 달리 React에서는 버블링이 발생합니다.

- `onToggle`: `Event` 핸들러 함수입니다. 세부사항을 토글할 때 발생합니다.
- `onToggleCapture`: `onToggle` 의 캡처 단계에서 실행되는 버전입니다.

다음의 이벤트는 `<img>`, `<iframe>`, `<object>`, `<embed>`, `<link>` 그리고 SVG `<image>` 요소들에 대해서 발생합니다. 그리고 브라우저 이벤트와 달리 React에서는 버블링이 발생합니다.

- `onLoad`: `Event` 핸들러 함수입니다. 자원이 로드되면 발생합니다.
- `onLoadCapture`: `onLoad` 의 캡처 단계에서 실행되는 버전입니다.
- `onError`: `Event` 핸들러 함수입니다. 자원을 로드할 수 없을 때 발생합니다.
- `onErrorCapture`: `onError` 의 캡처 단계에서 실행되는 버전입니다.

다음의 이벤트는 `<audio>` 및 `<video>` 와 같은 자원에 대해 발생합니다. 그리고 브라우저 이벤트와 달리 React에서는 버블링이 발생합니다.

- `onAbort`: `Event` 핸들러 함수입니다. 자원이 완전히 로드되지 않았지만 오류로 인한 것이 아닌 경우 발생합니다.
- `onAbortCapture`: `onAbort` 의 캡처 단계에서 실행되는 버전입니다.

- `onCanPlay`: `Event` 핸들러 함수입니다. 재생을 시작하기에 충분한 데이터가 있지만 버퍼링 없이 끝까지 재생할 수 없을 때 발생합니다.
- `onCanPlayCapture`: `onCanPlay` 의 캡처 단계에서 실행되는 버전입니다.
- `onCanPlayThrough`: `Event` 핸들러 함수입니다. 데이터가 충분하여 끝까지 버퍼링 없이 재생을 시작할 수 있을 때 발생합니다.
- `onCanPlayThroughCapture`: `onCanPlayThrough` 의 캡처 단계에서 실행되는 버전입니다.
- `onDurationChange`: `Event` 핸들러 함수입니다. 미디어 지속 시간이 업데이트되면 발생합니다.
- `onDurationChangeCapture`: `onDurationChange` 의 캡처 단계에서 실행되는 버전입니다.
- `onEmptied`: `Event` 핸들러 함수입니다. 미디어가 비어있을 때 발생합니다.
- `onEmptiedCapture`: `onEmptied` 의 캡처 단계에서 실행되는 버전입니다.
- `onEncrypted`: `Event` 핸들러 함수입니다. 브라우저에서 암호화된 미디어를 발견하면 발생합니다.
- `onEncryptedCapture`: `onEncrypted` 의 캡처 단계에서 실행되는 버전입니다.
- `onEnded`: `Event` 핸들러 함수입니다. 재생할 내용이 남아 있지 않아 재생이 중지되면 발생합니다.
- `onEndedCapture`: `onEnded` 의 캡처 단계에서 실행되는 버전입니다.
- `onError`: `Event` 핸들러 함수입니다. 리소스를 로딩할 수 없을 때 발생합니다.
- `onErrorCapture`: `onError` 의 캡처 단계에서 실행되는 버전입니다.
- `onLoadedData`: `Event` 핸들러 함수입니다. 현재 재생 프레임이 로딩되면 발생합니다.
- `onLoadedDataCapture`: `onLoadedData` 의 캡처 단계에서 실행되는 버전입니다.
- `onLoadedMetadata`: `Event` 핸들러 함수입니다. 메타데이터가 로딩될 때 발생합니다.
- `onLoadedMetadataCapture`: `onLoadedMetadata` 의 캡처 단계에서 실행되는 버전입니다.
- `onLoadStart`: `Event` 핸들러 함수입니다. 브라우저가 자원 로딩을 시작하면 발생합니다.
- `onLoadStartCapture`: `onLoadStart` 의 캡처 단계에서 실행되는 버전입니다.
- `onPause`: `Event` 핸들러 함수입니다. 미디어가 일시 중지되었을 때 발생합니다.
- `onPauseCapture`: `onPause` 의 캡처 단계에서 실행되는 버전입니다.
- `onPlay`: `Event` 핸들러 함수입니다. 미디어가 더 이상 일시 정지되지 않을 때 발생합니다.
- `onPlayCapture`: `onPlay` 의 캡처 단계에서 실행되는 버전입니다.
- `onPlaying`: `Event` 핸들러 함수입니다. 미디어 재생이 시작되거나 재시작될 때 발생합니다.
- `onPlayingCapture`: `onPlaying` 의 캡처 단계에서 실행되는 버전입니다.
- `onProgress`: `Event` 핸들러 함수입니다. 자원이 로드되는 동안 주기적으로 실행됩니다.
- `onProgressCapture`: `onProgress` 의 캡처 단계에서 실행되는 버전입니다.
- `onRateChange`: `Event` 핸들러 함수입니다. 재생 속도가 변경되면 발생합니다.

- `onRateChangeCapture` : `onRateChange` 의 캡처 단계에서 실행되는 버전입니다.
- `onResize` : `Event` 핸들러 함수입니다. 동영상 크기가 변경될 때 발생합니다.
- `onResizeCapture` : `onResize` 의 캡처 단계에서 실행되는 버전입니다.
- `onSeeked` : `Event` 핸들러 함수입니다. 탐색 작업이 완료되면 발생합니다.
- `onSeekedCapture` : `onSeeked` 의 캡처 단계에서 실행되는 버전입니다.
- `onSeeking` : `Event` 핸들러 함수입니다. 탐색 작업이 시작될 때 발생합니다.
- `onSeekingCapture` : `onSeeking` 의 캡처 단계에서 실행되는 버전입니다.
- `onStalled` : `Event` 핸들러 함수입니다. 브라우저가 데이터를 기다리지만 계속 로드되지 않을 때 발생합니다.
- `onStalledCapture` : `onStalled` 의 캡처 단계에서 실행되는 버전입니다.
- `onSuspend` : `Event` 핸들러 함수입니다. 자원 로딩이 일시 중단되었을 때 발생합니다.
- `onSuspendCapture` : `onSuspend` 의 캡처 단계에서 실행되는 버전입니다.
- `onTimeUpdate` : `Event` 핸들러 함수입니다. 현재 재생 시간이 업데이트될 때 발생합니다.
- `onTimeUpdateCapture` : `onTimeUpdate` 의 캡처 단계에서 실행되는 버전입니다.
- `onVolumeChange` : `Event` 핸들러 함수입니다. 볼륨이 변경되었을 때 발생합니다.
- `onVolumeChangeCapture` : `onVolumeChange` 의 캡처 단계에서 실행되는 버전입니다.
- `onWaiting` : `Event` 핸들러 함수입니다. 일시적인 데이터 부족으로 인해 재생이 중지된 경우 발생합니다.
- `onWaitingCapture` : `onWaiting` 의 캡처 단계에서 발생하는 버전입니다.

주의 사항

- `children` 과 `dangerouslySetInnerHTML` 을 동시에 전달할 수 없습니다.
- 일부 이벤트(예: `onAbort` , `onLoad`)는 브라우저에서 버블링이 발생하지 않지만, React에서는 버블링이 발생합니다.

`ref` 콜백 함수

`useRef` 에서 반환되는 `ref` 객체 대신 `ref` 속성에 함수를 전달할 수 있습니다.

```
<div ref={(node) => {
  console.log('Attached', node);

  return () => {
    console.log('Clean up', node)
```

```
    }
  }}>
```

`ref` 콜백을 사용하는 예시를 확인해 보세요.

화면에 `<div>` DOM 노드가 추가되면, React는 `ref` 콜백을 호출하고 그 인자로 DOM `node` 를 전달합니다. 해당 `<div>` DOM 노드가 제거되면, React는 콜백에서 반환한 Cleanup 함수를 호출합니다.

React는 *다른* `ref` 콜백을 전달할 때마다 `ref` 콜백도 호출합니다. 위 예시에서 `(node) => { ... }` 는 렌더링마다 서로 다른 함수입니다. 컴포넌트가 다시 렌더링 될 때, *이전* 함수는 인자로 `null` 을 받아 호출되고, *다음* 함수는 DOM 노드를 인자로 받아 호출됩니다.

매개변수

- `node` : DOM 노드. Ref가 DOM 노드에 연결될 때 React가 해당 DOM 노드를 전달합니다. 매 렌더링에서 `ref` 콜백에 동일한 함수 참조를 넘기지 않으면, 컴포넌트가 리렌더링될 때마다 콜백이 일시적으로 Cleanup 됐다가 다시 생성됩니다.

중요합니다!

React 19는 `ref` 콜백을 위한 Cleanup 함수를 추가했습니다.

하위 호환성을 위해, `ref` 콜백이 Cleanup 함수를 반환하지 않으면, `ref` 가 분리될 때 `node` 가 `null` 로 호출됩니다. 이 동작은 향후 버전에서 제거될 예정입니다.

반환값

- **optional** `Cleanup 함수` : `ref` 가 분리되면, React는 cleanup 함수를 호출합니다. `ref` 콜백에 의해 함수가 반환되지 않으면 React는 `ref` 가 분리되면 인수로 `null` 을 사용하여 다시 콜백을 호출합니다. 이 동작은 향후 버전에서 제거될 예정입니다.

주의 사항

- Strict Mode가 켜져있으면, React는 첫 번째 실제 설정 전에 **개발 전용 Setup + cleanup 주기**를 하나 더 실행할 것입니다. 이는 스트레스 테스트로, Cleanup 로직이 Setup 로직을 "거울처럼" 따라가며 Setup이 하는 일을 중지하거나 되돌리도록 보장하기 위한 것입니다. 이 때문에 문제가 발생한다면 Cleanup 함수를 구현하세요.
- *다른* `ref` 콜백을 전달하면, React는 먼저 *이전* 콜백의 Cleanup 함수가 있다면 그것을 호출합니다. Cleanup 함수가 없으면, 이전 `ref` 콜백을 `null` 을 인수로 하여 한 번 호출합니다. *다음* 함수는 DOM 노드와 함께 호출됩니다.

React 이벤트 객체

이벤트 핸들러는 *React 이벤트 객체*를 받게 되며, "합성 이벤트"라고도 합니다.

```
<button onClick={e => {
  console.log(e); // React 이벤트 객체
}} />
```

이것은 기본 DOM 이벤트와 같은 표준을 준수하지만 일부 브라우저의 불일치를 수정합니다.

일부 React의 이벤트는 브라우저의 네이티브 이벤트에 직접 매핑되지 않습니다. 예를 들어 `onMouseLeave` 에서 `e.nativeEvent` 는 `mouseout` 이벤트를 가리킵니다. 특정 매핑은 퍼블릭 API의 일부가 아니며 추후 변경될 수 있습니다. 어떠한 이유로 기본 브라우저 이벤트가 필요한 경우 `e.nativeEvent` 에서 읽어와야 합니다.

프로퍼티

React 이벤트 객체는 표준 `Event` 프로퍼티의 일부를 구현했습니다.

- `bubbles` : 불리언 타입입니다. 이벤트가 DOM을 통해 버블링되는지 여부를 반환합니다.
- `cancelable` : 불리언 타입입니다. 이벤트를 취소할 수 있는지를 반환합니다.
- `currentTarget` : DOM 노드입니다. React 트리에서 현재 핸들러가 연결된 노드를 반환합니다.
- `defaultPrevented` : 불리언 타입입니다. `preventDefault` 가 호출되었는지 여부를 반환합니다.
- `eventPhase` : 숫자 타입입니다. 이벤트가 현재 어느 단계에 있는지 반환합니다.

- `isTrusted` : 불리언 타입입니다. 사용자에 의해 이벤트가 시작되었는지에 대한 여부를 반환합니다.
- `target` : DOM 노드입니다. (멀리 있는 자식일 수도 있는) 이벤트가 발생한 노드를 반환합니다.
- `timeStamp` : 숫자 타입입니다. 이벤트가 발생한 시간을 반환합니다.

추가로 React 이벤트 객체는 다음과 같은 프로퍼티를 제공합니다.

- `nativeEvent` : DOM `Event` 이벤트입니다. 원래의 브라우저 이벤트 객체입니다.

메서드

React 이벤트 객체는 표준 `Event` 메서드의 일부를 구현했습니다.

- `preventDefault()` : 이벤트에 대한 기본 브라우저 동작을 방지합니다.
- `stopPropagation()` : React 트리를 통한 이벤트 전파를 중지합니다.

추가로 React 이벤트 객체는 다음과 같은 프로퍼티를 제공합니다.

- `isDefaultPrevented()` : `preventDefault` 가 호출되었는지에 대한 여부를 나타내는 불리언 값을 반환합니다.
- `isPropagationStopped()` : `stopPropagation` 이 호출되었는지에 대한 여부를 나타내는 불리언 값을 반환합니다.
- `persist()` : React DOM에서는 사용되지 않습니다. React Native에서는 이벤트가 발생한 후 이벤트의 프로퍼티를 읽으려면 해당 함수를 호출해야 합니다.
- `isPersistent()` : React DOM에서는 사용되지 않습니다. React Native에서는 `persist` 가 호출되었는지 여부를 반환합니다.

주의 사항

- `currentTarget` , `eventPhase` , `target` , `type` 의 값은 React 코드가 예상하는 값을 반영합니다. 내부적으로는 React는 이벤트 핸들러를 루트에 첨부하지만, React 이벤트 객체에는 반영되지 않습니다. 예를 들어 `e.currentTarget` 은 기본 `e.nativeEvent.currentTarget` 과 동일하지 않을 수 있습니다. 폴리필 된 이벤트의 경우 `e.type` (React 이벤트 타입)이 `e.nativeEvent.type` (기본 타입)과 다를 수 있습니다.

`AnimationEvent` 핸들러 함수

CSS 애니메이션 이벤트에 대한 이벤트 핸들러 유형입니다.

```
<div
  onAnimationStart={e => console.log('onAnimationStart')}
  onAnimationIteration={e => console.log('onAnimationIteration')}
  onAnimationEnd={e => console.log('onAnimationEnd')}
/>
```

매개변수

- `e`: 다음과 같은 추가 `AnimationEvent` 프로퍼티가 있는 React 이벤트 객체입니다.
  - `animationName`
  - `elapsedTime`
  - `pseudoElement`

`ClipboardEvent` 핸들러 함수

클립보드 API 이벤트에 대한 이벤트 핸들러 유형입니다.

```
<input
  onCopy={e => console.log('onCopy')}
  onCut={e => console.log('onCut')}
  onPaste={e => console.log('onPaste')}
/>
```

매개변수

- `e`: 다음과 같은 추가 `ClipboardEvent` 프로퍼티가 있는 React 이벤트 객체입니다.
  - `clipboardData`

`CompositionEvent` 핸들러 함수

입력 메서드 편집기 (IME) 이벤트에 대한 이벤트 핸들러 유형입니다.

```
<input
  onCompositionStart={e => console.log('onCompositionStart')}
  onCompositionUpdate={e => console.log('onCompositionUpdate')}
  onCompositionEnd={e => console.log('onCompositionEnd')}
/>
```

매개변수

- `e`: 다음과 같은 추가 `CompositionEvent` 프로퍼티가 있는 React 이벤트 객체입니다.
  - `data`

`DragEvent` 핸들러 함수

HTML 드래그 앤 드롭 API 이벤트의 이벤트 핸들러 유형입니다.

```
<>
  <div
    draggable={true}
    onDragStart={e => console.log('onDragStart')}
    onDragEnd={e => console.log('onDragEnd')}
  >
    Drag source
  </div>

  <div
    onDragEnter={e => console.log('onDragEnter')}
    onDragLeave={e => console.log('onDragLeave')}
    onDragOver={e => { e.preventDefault(); console.log('onDragOver'); }}
    onDrop={e => console.log('onDrop')}
  >
    Drop target
  </div>
</>
```

매개변수

- `e` : 다음과 같은 추가 `DragEvent` 프로퍼티가 있는 React 이벤트 객체입니다.
  - `dataTransfer`

  이는 상속된 `MouseEvent` 의 프로퍼티도 포함합니다.

  - `altKey`
  - `button`
  - `buttons`
  - `ctrlKey`
  - `clientX`
  - `clientY`
  - `getModifierState(key)`
  - `metaKey`
  - `movementX`
  - `movementY`
  - `pageX`
  - `pageY`
  - `relatedTarget`
  - `screenX`
  - `screenY`
  - `shiftKey`

  또한 상속된 `UIEvent` 의 프로퍼티도 포함합니다.

  - `detail`
  - `view`

`FocusEvent` 핸들러 함수

포커싱 이벤트에 대한 이벤트 핸들러 유형입니다.

```
<input
  onFocus={e => console.log('onFocus')}
  onBlur={e => console.log('onBlur')}
```

아래 예시를 참고하세요.

매개변수

- `e`: 다음과 같은 추가 `FocusEvent` 프로퍼티가 있는 React 이벤트 객체입니다.
  - `relatedTarget`

  또한 상속된 `UIEvent` 의 프로퍼티도 포함합니다.

  - `detail`
  - `view`

`Event` 핸들러 함수

일반 이벤트를 위한 이벤트 핸들러 유형입니다.

매개변수

- `e`: 추가 프로퍼티가 없는 React 이벤트 객체입니다.

`InputEvent` 핸들러 함수

`onBeforeInput` 이벤트에 대한 이벤트 핸들러 유형입니다.

```
<input onBeforeInput={e => console.log('onBeforeInput')} />
```

매개변수

- `e`: 다음과 같은 추가 `InputEvent` 프로퍼티가 있는 React 이벤트 객체입니다.
  - `data`

`KeyboardEvent` 핸들러 함수

키보드 이벤트에 대한 이벤트 핸들러 유형입니다.

```
<input
  onKeyDown={e => console.log('onKeyDown')}
  onKeyUp={e => console.log('onKeyUp')}
/>
```

아래 예시를 참고하세요.

매개변수

- `e` : 다음과 같은 추가 `KeyboardEvent` 프로퍼티가 있는 React 이벤트 객체입니다.
  - `altKey`
  - `charCode`
  - `code`
  - `ctrlKey`
  - `getModifierState(key)`
  - `key`
  - `keyCode`
  - `locale`
  - `metaKey`
  - `location`
  - `repeat`
  - `shiftKey`
  - `which`

  또한 상속된 `UIEvent` 의 프로퍼티도 포함합니다.

  - `detail`
  - `view`

`MouseEvent` 핸들러 함수

마우스 이벤트에 대한 이벤트 핸들러 유형입니다.

```
<div
  onClick={e => console.log('onClick')}
  onMouseEnter={e => console.log('onMouseEnter')}
  onMouseOver={e => console.log('onMouseOver')}
  onMouseDown={e => console.log('onMouseDown')}
  onMouseUp={e => console.log('onMouseUp')}
  onMouseLeave={e => console.log('onMouseLeave')}
/>
```

아래 예시를 참고하세요.

매개변수

- `e` : 다음과 같은 추가 `MouseEvent` 프로퍼티가 있는 React 이벤트 객체입니다.
  - `altKey`
  - `button`
  - `buttons`
  - `ctrlKey`
  - `clientX`
  - `clientY`
  - `getModifierState(key)`
  - `metaKey`
  - `movementX`
  - `movementY`
  - `pageX`
  - `pageY`
  - `relatedTarget`
  - `screenX`
  - `screenY`
  - `shiftKey`

  또한 상속된 `UIEvent` 의 프로퍼티도 포함합니다.

  - `detail`
  - `view`

`PointerEvent` 핸들러 함수

포인터 이벤트에 대한 이벤트 핸들러 유형입니다.

```
<div
  onPointerEnter={e => console.log('onPointerEnter')}
  onPointerMove={e => console.log('onPointerMove')}
  onPointerDown={e => console.log('onPointerDown')}
  onPointerUp={e => console.log('onPointerUp')}
  onPointerLeave={e => console.log('onPointerLeave')}
/>
```

아래 예시를 참고하세요.

매개변수

- `e` : 다음과 같은 추가 `PointerEvent` 프로퍼티가 있는 React 이벤트 객체입니다.
  - `height`
  - `isPrimary`
  - `pointerId`
  - `pointerType`
  - `pressure`
  - `tangentialPressure`
  - `tiltX`
  - `tiltY`
  - `twist`
  - `width`

  이는 상속된 `MouseEvent` 의 프로퍼티도 포함합니다.

  - `altKey`
  - `button`
  - `buttons`
  - `ctrlKey`
  - `clientX`

- clientY
- getModifierState(key)
- metaKey
- movementX
- movementY
- pageX
- pageY
- relatedTarget
- screenX
- screenY
- shiftKey

또한 상속된 UIEvent 의 프로퍼티도 포함합니다.

- detail
- view

TouchEvent 핸들러 함수

터치 이벤트에 대한 이벤트 핸들러 유형입니다.

```
<div
  onTouchStart={e => console.log('onTouchStart')}
  onTouchMove={e => console.log('onTouchMove')}
  onTouchEnd={e => console.log('onTouchEnd')}
  onTouchCancel={e => console.log('onTouchCancel')}
/>
```

매개변수

- e : 다음과 같은 추가 TouchEvent 프로퍼티가 있는 React 이벤트 객체입니다.
  - altKey
  - ctrlKey
  - changedTouches

- `getModifierState(key)`
- `metaKey`
- `shiftKey`
- `touches`
- `targetTouches`

또한 상속된 `UIEvent` 의 프로퍼티도 포함합니다.

- `detail`
- `view`

`TransitionEvent` 핸들러 함수

CSS 전환 이벤트에 대한 이벤트 핸들러 유형입니다.

```
<div
  onTransitionEnd={e => console.log('onTransitionEnd')}
/>
```

매개변수

- `e` : 다음과 같은 추가 `TransitionEvent` 의 프로퍼티가 있는 React 이벤트 객체입니다.
  - `elapsedTime`
  - `propertyName`
  - `pseudoElement`

`UIEvent` 핸들러 함수

일반적인 UI 이벤트를 위한 이벤트 핸들러 유형입니다.

```
<div
  onScroll={e => console.log('onScroll')}
/>
```

매개변수

- `e`: 다음과 같은 추가 `UIEvent` 프로퍼티를 가진 React 이벤트 객체입니다.
  - `detail`
  - `view`

`WheelEvent` 핸들러 함수

`onWheel` 이벤트에 대한 이벤트 핸들러 유형입니다.

```
<div
  onWheel={e => console.log('onWheel')}
/>
```

매개변수

- `e`: 다음과 같은 추가 `WheelEvent` 의 프로퍼티가 있는 React 이벤트 객체입니다.
  - `deltaMode`
  - `deltaX`
  - `deltaY`
  - `deltaZ`

  또한 다음과 같이 상속된 `MouseEvent` 의 프로퍼티도 포함합니다.

  - `altKey`
  - `button`
  - `buttons`
  - `ctrlKey`
  - `clientX`
  - `clientY`
  - `getModifierState(key)`
  - `metaKey`
  - `movementX`
  - `movementY`

- `pageX`
- `pageY`
- `relatedTarget`
- `screenX`
- `screenY`
- `shiftKey`

더불어 아래의 상속된 `UIEvent` 의 프로퍼티도 포함합니다.

- `detail`
- `view`

사용법

CSS 스타일 적용하기

React는 `className` 을 사용하여 CSS 클래스를 지정합니다. 이것은 HTML의 클래스 속성처럼 작동합니다.

```
<img className="avatar" />
```

그런 다음 별도의 CSS 파일에 CSS 규칙을 지정합니다.

```
/* In your CSS */
.avatar {
  border-radius: 50%;
}
```

React는 CSS 파일을 추가하는 방법을 규정하지 않습니다. 가장 간단한 방법은 HTML에 `<link>` 태그를 추가하는 것입니다. 빌드 도구나 프레임워크를 사용하고 있다면, 해당 기술의 문서를 참조하여 프로젝트에 CSS 파일을 추가하는 방법을 알아보세요.

때때로 스타일 값은 데이터에 따라 달라집니다. `style` 어트리뷰트를 사용하여 일부 스타일을 동적으로 전달할 수 있습니다.

```
<img
  className="avatar"
  style={{
    width: user.imageSize,
    height: user.imageSize
  }}
/>
```

위의 예시에서 `style={{}}` 은 특별한 구문이 아니라 `style={ }` 와 같이 중괄호가 있는 JSX 내에 있는 일반 `{}` 객체입니다. 스타일이 자바스크립트 변수에 의존하는 경우에만 `style` 어트리뷰트를 사용하는 것이 좋습니다.

App.js Avatar.js 새로고침 Clear 포크

```
export default function Avatar({ user }) {
  return (
    <img
      src={user.imageUrl}
      alt={'Photo of ' + user.name}
      className="avatar"
      style={{
        width: user.imageSize,
        height: user.imageSize
      }}
    />
  );
```

자세히 살펴보기

여러 CSS 클래스를 조건부로 적용하기 위해 어떻게 해야하나요?

자세히 보기

`ref`를 사용하여 DOM 노드 조작하기

때로는 JSX에서 태그와 연결된 브라우저 DOM 노드를 가져와야 하는 경우가 있습니다. 예를 들어 버튼이 클릭 될 때 `<input>`에 포커싱을 맞추려면 브라우저의 `<input>` DOM 노드에서 `focus()`를 호출하면 됩니다.

태그에 대한 브라우저의 DOM 노드를 가져오려면 `ref`를 선언하고 해당 태그에 `ref` 어트리뷰트로 전달합니다.

```
import { useRef } from 'react';

export default function Form() {
  const inputRef = useRef(null);
  // ...
  return (
    <input ref={inputRef} />
    // ...
```

React는 DOM 노드를 화면에 렌더링 한 후 `inputRef.current`에 넣습니다.

App.js 다운로드 새로고침 Clear 포크

```
import { useRef } from 'react';

export default function Form() {
  const inputRef = useRef(null);

  function handleClick() {
    inputRef.current.focus();
  }

  return (
    <>
      <input ref={inputRef} />
```

자세히 보기

Ref로 DOM 조작하기 및 더 많은 예시에 대해 더 자세히 읽어보세요.

고급 사용 사례의 경우 `ref` 어트리뷰트는 콜백 함수도 허용합니다.

내부 HTML을 위험하게 설정하는 경우

다음과 같이 원시 HTML 문자열을 요소에 전달할 수 있습니다.

```
const markup = { __html: '<p>some raw html</p>' };
return <div dangerouslySetInnerHTML={markup} />;
```

이것은 위험합니다. 기본 DOM의 `innerHTML` 프로퍼티와 마찬가지로 각별히 주의해야 합니다. 마크업이 완전히 신뢰할 수 있는 출처에서 제공되는 것이 아니라면, XSS 취약점이 쉽게 나타날 수 있습니다.

예를 들어, 마크다운을 HTML로 변환하는 라이브러리를 사용할 때, 해당 파서에 버그가 없고 사용자가 자신의 입력만 볼 수 있다고 믿는다면 다음과 같이 결과 HTML을 표시할 수 있습니다.

package.json App.js MarkdownPreview.js 새로고침 Clear 포크

```
import { Remarkable } from 'remarkable';

const md = new Remarkable();

function renderMarkdownToHTML(markdown) {
  // 출력되는 HTML이 동일한 사용자에게 표시되고,
  // 이 마크다운 파서에 버그가 없다고
  // 신뢰하기 때문에 안전합니다.
  const renderedHTML = md.render(markdown);
  return {__html: renderedHTML};
}
```

자세히 보기

위 예시의 `renderMarkdownToHTML` 함수처럼, `{__html}` 객체는 가능한 한 HTML이 만들어지는 곳 가까이에서 생성되어야 합니다. 이렇게 하면 코드에서 사용되는 모든 원시 HTML이 명시적으로 표시되고 HTML을 포함할 것으로 예상되는 변수만 `dangerouslySetInnerHTML` 로 전달됩니다. `<div dangerouslySetInnerHTML={{__html: markup}} />` 처럼 인라인으로 객체를 생성하는 것은 권장하지 않습니다.

임의의 HTML을 렌더링하는 것이 왜 위험한지를 알아보려면 위의 코드를 다음과 같이 바꿔보세요.

```
const post = {
  // 이 콘텐츠가 데이터베이스에 저장되어 있다고 가정해보겠습니다.
  content: `<img src="" onerror='alert("you were hacked")'>`
};

export default function MarkdownPreview() {
  // 🔴 보안 취약점: 신뢰할 수 없는 입력을 dangerouslySetInnerHTML로 전달했습니다.
  const markup = { __html: post.content };
  return <div dangerouslySetInnerHTML={markup} />;
}
```

HTML에 포함된 코드가 실행됩니다. 해커는 이 보안 허점을 이용하여 사용자의 정보를 훔치거나 사용자 대신 작업을 수행할 수 있습니다. **신뢰할 수 있고 유해한 정보가 포함되어 있지 않은 데이터를 사용할 때만 `dangerouslySetInnerHTML` 을 사용하세요.**

마우스 이벤트 처리

이 예시는 일반적인 마우스 이벤트와 해당 이벤트가 언제 발생하는지 보여줍니다.

App.js 다운로드 새로고침 Clear 포크

```
export default function MouseExample() {
  return (
```

```
      <div
        onMouseEnter={e => console.log('onMouseEnter (parent)')}
        onMouseLeave={e => console.log('onMouseLeave (parent)')}
      >
        <button
          onClick={e => console.log('onClick (first button)')}
          onMouseDown={e => console.log('onMouseDown (first button)')}
          onMouseEnter={e => console.log('onMouseEnter (first button)')}
          onMouseLeave={e => console.log('onMouseLeave (first button)')}
```

자세히 보기

포인터 이벤트 처리

이 예시는 일반적인 포인터 이벤트와 해당 이벤트가 언제 발생하는지 보여줍니다.

App.js 다운로드 새로고침 Clear 포크

```
export default function PointerExample() {
  return (
    <div
      onPointerEnter={e => console.log('onPointerEnter (parent)')}
      onPointerLeave={e => console.log('onPointerLeave (parent)')}
```

```
        style={{ padding: 20, backgroundColor: '#ddd' }}
      >
        <div
          onPointerDown={e => console.log('onPointerDown (first child)')}
          onPointerEnter={e => console.log('onPointerEnter (first child)')}
          onPointerLeave={e => console.log('onPointerLeave (first child)')}
```

자세히 보기

포커스 이벤트 처리

React에서는 포커스 이벤트가 버블링됩니다. 부모 요소의 바깥 부분에서 발생한 이벤트가 focus 혹은 blur인지 구분하기 위해 `currentTarget` 과 `relatedTarget` 를 사용할 수 있습니다.

App.js 다운로드 새로고침 Clear 포크

```
export default function FocusExample() {
  return (
    <div
      tabIndex={1}
      onFocus={(e) => {
        if (e.currentTarget === e.target) {
          console.log('focused parent');
```

```
      } else {
        console.log('focused child', e.target.name);
      }
      if (!e.currentTarget.contains(e.relatedTarget)) {
        // children가의 focus를 이동할 때는 발생되지 않음
```

자세히 보기

키보드 이벤트 처리

이 예시는 일반적인 키보드 이벤트와 해당 이벤트가 언제 발생하는지 보여줍니다.

App.js

다운로드 새로고침 Clear 포크

```
export default function KeyboardExample() {
  return (
    <label>
      First name:
      <input
        name="firstName"
        onKeyDown={e => console.log('onKeyDown:', e.key, e.code)}
        onKeyUp={e => console.log('onKeyUp:', e.key, e.code)}
      />
    </label>
```

```
    );
  }
```


이전
컴포넌트

다음
<form>

커뮤니티

행동 강령

팀 소개

문서 기여자

감사의 말

더 보기

블로그

React Native

개인 정보 보호

약관