

06

React.js

PNU Mini Bootcamp – Front End

리액트 소개

- React란?
 - Facebook에서 만든 UI를 작성하기 위한 자바스크립트 라이브러리
 - A JavaScript library for building user interfaces
 - Javascript 코드만을 이용해 조합형 UI를 작성함.
 - JSX와 같은 형식을 사용하지만 결국 트랜스파일되어 자바스크립트 코드로 실행함.
 - 데이터가 지속적으로 변하는 대규모 애플리케이션의 구축을 위해서 만들어졌음

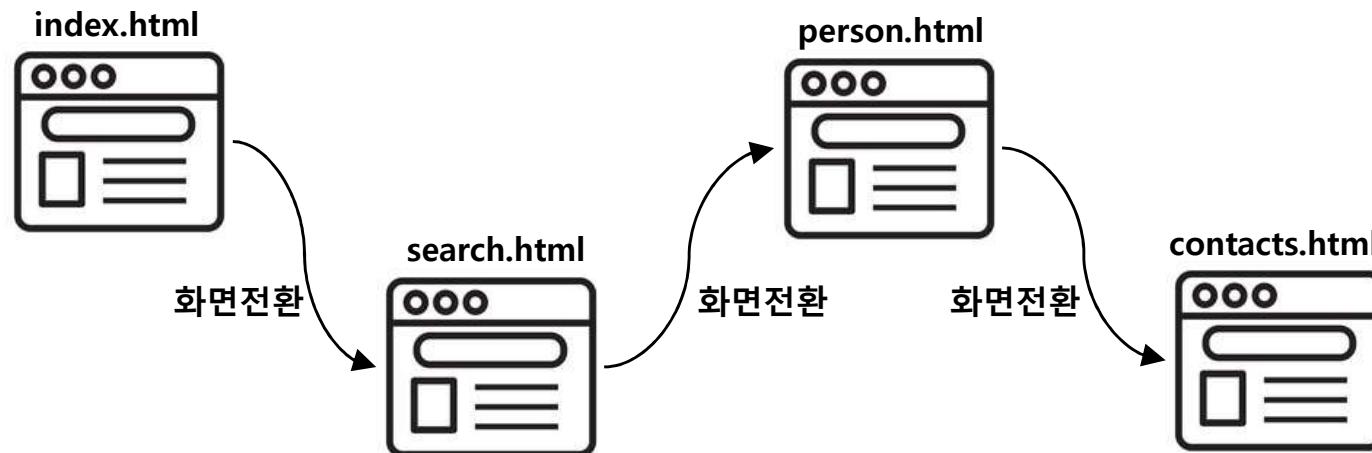
리액트 소개

- 전통적인 웹 애플리케이션
 - 요청의 단위가 페이지임
 - 화면의 페이지 단위로 구성됨. 따라서 화면의 변경을 위해 서버로부터 새로운 페이지를 내려받음
 - 페이지 단위의 요청과 응답의 문제점은 화면 일부분만 갱신하고 싶어도 페이지 전체를 HTTP 서버로부터 다시 받아와야 한다는 것을 의미
 - HTTP 서버는 브라우저의 요청이 있을 때마다 페이지 전체를 재생성하여 전송



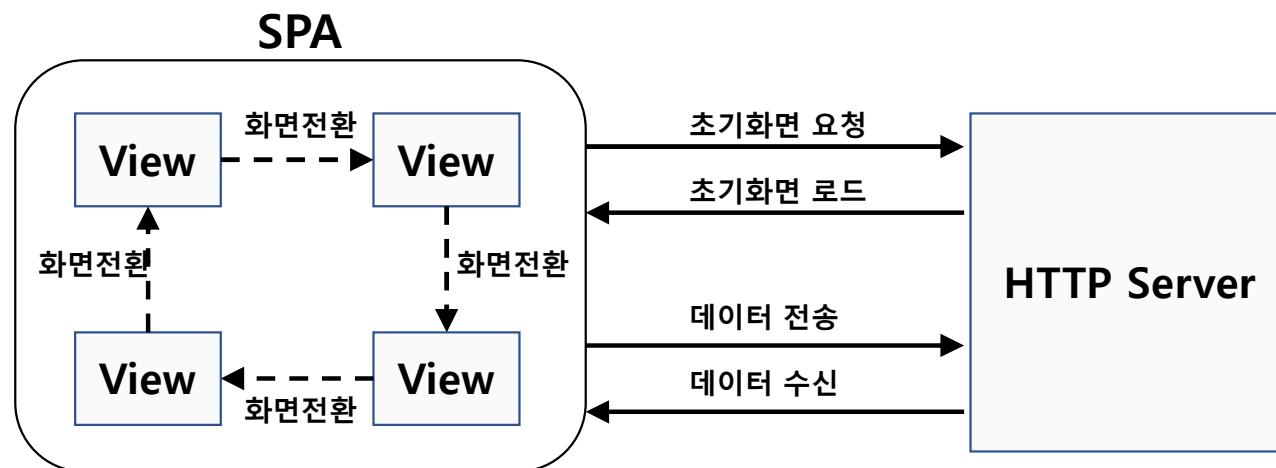
리액트 소개

- AJAX를 이용한 멀티 페이지 애플리케이션
 - 화면이 전환될 때는 새로운 HTML 페이지로 이동함
 - 전화된 HTML 페이지가 초기 화면을 구성함
 - 데이터를 이용한 UI 생성과 서버로 데이터를 전송하는 작업은 AJAX를 이용함
 - jQuery 를 주로 사용하던 시절의 개발



리액트 소개

- 단일 페이지 애플리케이션
 - SPA(Single Page Application)
 - 하나의 페이지로 애플리케이션의 모든 화면을 제공함.
 - AJAX, Socket 등의 방법을 이용해 서버와 데이터 교환
 - 하나의 HTML 내에서 화면전환이 되어야 함으로 라우팅 기능의 라이브러리 필요

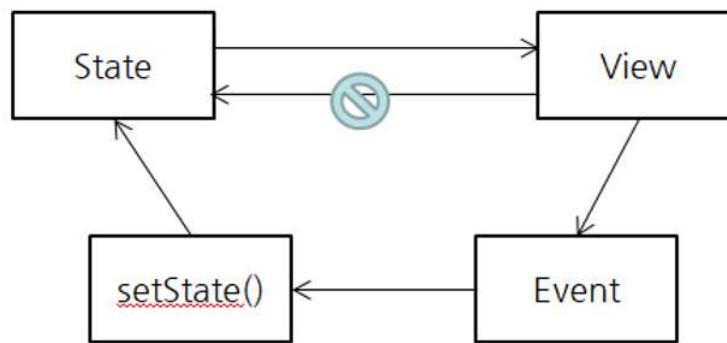


리액트 소개

- 단일 페이지 애플리케이션의 단점
 - 효과적인 상태관리가 요구됨
 - 애플리케이션에서 보여줘야 하는 데이터. 즉 상태(State)를 브라우저에서 관리해야 함.
 - 각각의 화면마다 상태를 관리하기 쉽지 않음. 한 상태 데이터를 여러 화면이 이용하는 경우가 있기 때문에...
 - 느린 DOM
 - SPA는 DOM을 아주 빈번히 갱신하는데, 브라우저의 DOM을 직접 다루는 것은 대단히 느리다.
 - 이것으로 인해 화면 전환, 데이터 변경이 일어날 때 사용자 경험이 훼손될 수 있다.
 - HTML 마크업을 생성하도록 자바스크립트 코드로 제어해야 함.
 - HTML 마크업을 문자열로 이어붙이는 작업은 개발 생산성을 떨어뜨림.

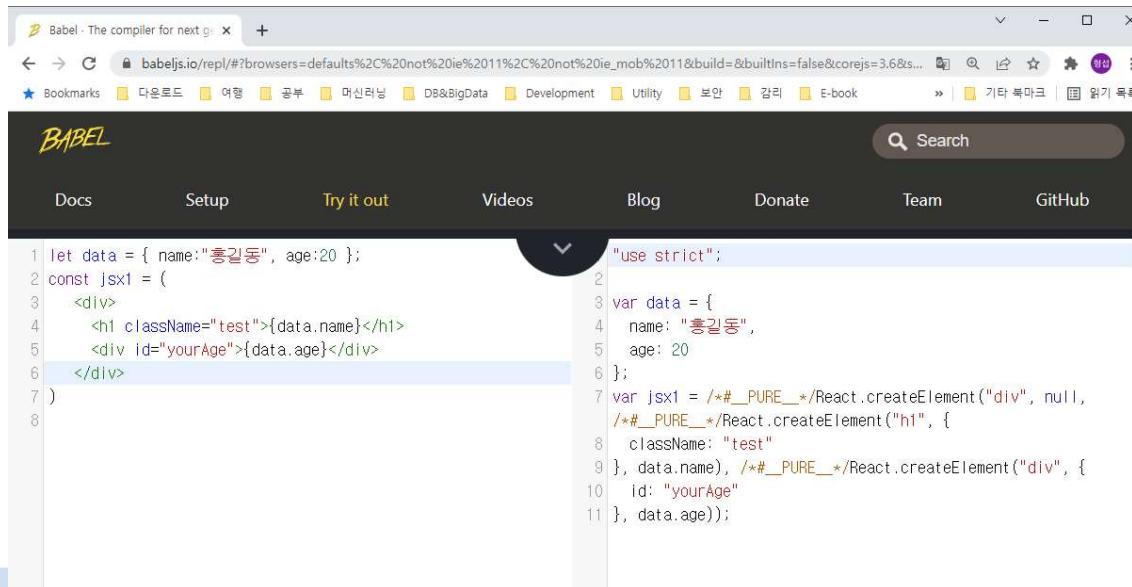
리액트 소개

- React의 특징
 - 상태 관리
 - 컴포넌트 단위로 상태를 관리할 수 있음
 - Flux, Redux와 같은 라이브러리를 사용하면 상태관리를 앱단위로 중앙집중화할 수 있음
 - 단방향 데이터 바인딩
 - 상태 데이터가 변경되면 즉시 UI가 반영됨
 - 번거로워 보일 수 있지만 상태 데이터가 변경되는 과정이 명시적으로 예측 가능하고, 추적 가능하도록 함.



리액트 소개

- React의 특징
 - JSX(Javascript XML)
 - 템플릿이 아닌 XML과 유사한 자바스크립트 확장 문법임.
 - 필수는 아니지만 자바스크립트 코드 내부에서 HTML 마크업을 사용할 수 있도록 지원함.



The screenshot shows a browser window with the Babel REPL interface. The URL is `babeljs.io/repl/#?browsers=defaults%2C%20not%20ie%2011%2C%20not%20ie_mob%2011&build=&builtIns=false&corejs=3.6&s...`. The page title is "BABEL". The main content area contains two panes of code. The left pane shows JSX code:

```
1 let data = { name:"홍길동", age:20 };
2 const jsx1 =
3   <div>
4     <h1 className="test">{data.name}</h1>
5     <div id="yourAge">{data.age}</div>
6   </div>
7 )
```

The right pane shows the generated JavaScript code:

```
"use strict";
var data = {
  name: "홍길동",
  age: 20
};
var jsx1 = /*#___PURE__*/React.createElement("div", null,
/*_PURE__*/React.createElement("h1", {
  className: "test"
}, data.name), /*#___PURE__*/React.createElement("div", {
  id: "yourAge"
}, data.age));
```

리액트 소개

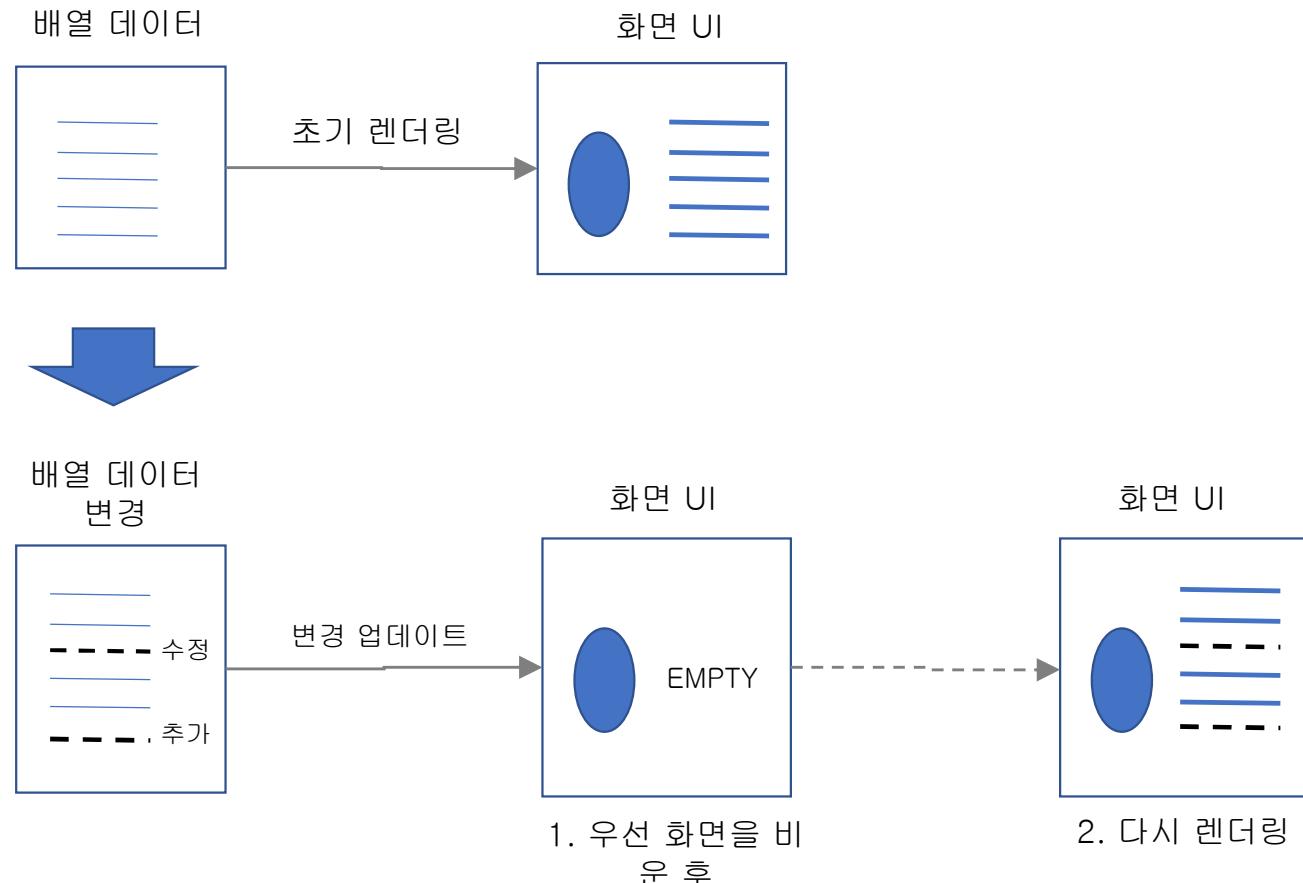
- React의 특징
 - 컴포넌트 기반의 개발
 - React 컴포넌트는 독립적인 컴포넌트 --> 뛰어난 재사용성
 - 마크업, 스타일, 자바스크립트 코드의 분리가 아닌 결합
 - 밀접한 연관성이 있으므로...
 - 관심사의 분리를 컴포넌트 단위를 만드는 방식으로 분리시킴.

리액트 소개

- React의 특징
 - 가상 DOM : Virtual DOM
 - DOM 조작은 느리므로 가상 DOM을 사용함
 - 앱의 상태가 바뀔때 전체 DOM을 업데이트하지 않고 원하는 DOM 상태와 유사한 가상 트리를 형성함.
 - React는?
 - Always re-render on update!!
 - 개발자가 원하는 출력물만을 선언적으로 작성하기 때문에 컴포넌트 전체를 re-render 하듯이 개발할 수 밖에 없음
 - 따라서 UI 성능을 위해서 가상 DOM이 반드시 필요함
 - 가상 DOM 트리를 비교하면서 필요한 부분만 업데이트함. --> 화면 업데이트 로직은 신경쓰지 않아도 됨.
 - 가상 DOM의 개념을 익힐 수 있는 동영상
 - <https://www.youtube.com/watch?v=BYbgopx44vo>

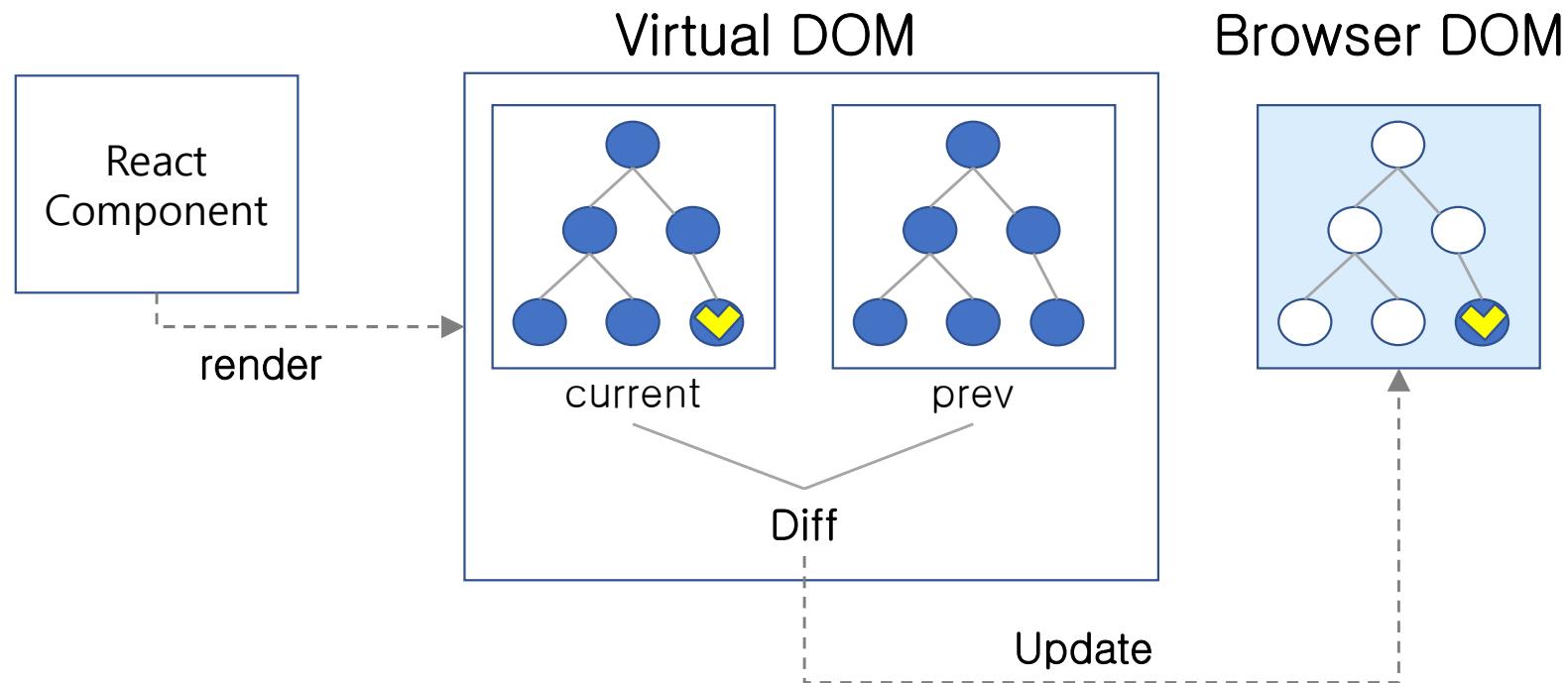
리액트 소개

- 가상 DOM 적용 전 상황



리액트 소개

- Virtual DOM의 작동 방식



Virtual DOM의 이전 상태와 현재 상태를 비교해 차이가 나 는 부분만을 Browser DOM으로 업데이트함.

리액트 프로젝트 생성

CRA

- create-react-app
- 리액트 프로젝트를 생성할 때 필요한 webpack 등의 설정을 자동화하여 개발 환경을 구축해주는 도구
- 리액트 개발을 위해서는 webpack, rollup 같은 module bundler 도구가 있어야 하며 babel, tsc 같은 transpiler 가 있어야 한다.
- Component 수 백개를 하나의 파일내에 작성할 수 없다. 그럼으로 모듈단위로 분리해서 개발해야 한다.
- 그런데 개발은 모듈단위로 분리시키지만 배포 직전에는 하나 혹은 몇 개의 js 파일로 묶어주어야 한다. 이 기능을 하는 것이 모듈 번들러 도구이며 대표적인 것이 webpack, rollup 이 있다.
- CRA 는 모듈 번들러, 프랜스파일등의 기능을 조합해서 리액트 프로젝트 템플릿을 만들어주는 도구이다.
- CRA 를 이용하면 프로젝트 생성, 개발도구, 빌드, 실행을 위한 다양한 설정들을 자동으로 해준다.

리액트 프로젝트 생성

- 생성 방법
 - ES6 기반 코드 : npx create-react-app [프로젝트명]
 - Typescript 기반 코드 : npx create-react-app [프로젝트명] --template typescript
 - 프로젝트 명으로 디렉토리가 만들어지고 기본적인 boilerplate 코드가 제공됨
- 시작 명령어
 - 빌드 명령어 : npm run build
 - 개발서버 시작 명령어 : npm run start
- npx create-react-app 실행 시 오류 대처
 - npx create-react-app hello 명령을 실행했을 때 다음 오류가 발생하는 경우가 있다.
 - "You are running 'create-react-app' x.x.x, which is behind the latest release (x.x.x). We no longer support global installation of Create React App."
 - 이 경우에는 다음 명령어를 실행한 후 프로젝트 생성을 재시도한다.

```
npm uninstall -g create-react-app  
npx clear-npx-cache
```

리액트 프로젝트 생성

- 폴더 구조
 - src : JS, TS 코드를 이곳에 작성함
 - 시작 진입점 : src/index.js
 - public : 정적 파일, 자원을 이곳에 배치함.
 - build : 빌드 후 생성된 아티팩트가 이곳에 저장됨.

리액트 프로젝트 생성

vite 도구

- vue 전용 빌드 도구로 탄생 --> react 지원
- vite(비트) : 프랑스어로 빠르다는 뜻
 - 개발과 빌드의 속도에 초점을 맞추었음
 - Webpack이 아닌 ESBUILD, Rollup 기반
 - 생성된 프로젝트의 사이즈가 작고 빠름 : 280MB vs 100MB
- Webpack 은 자바스크립트로 만들어진 번들링, 빌드 도구인데
- ESBUILD 는 Go 언어로 만들어진 빌드 도구이다. 네이티브 성격이 강한 도구이다. 사이즈도 작고 빠르다.

리액트 프로젝트 생성

- vite 도구
 - 생성 방법
 - npm init vite [프로젝트명]
 - 프롬프트 기반으로 하나씩 선택해가며 생성할 수 있음.
 - ES6 기반 코드 직접 생성
 - npm init vite [프로젝트명] -- --template react
 - yarn create vite [프로젝트명] -- --template react
 - Typescript 기반 코드 직접 생성
 - npm init vite [프로젝트명] -- --template react-ts
 - yarn create vite [프로젝트명] -- --template react-ts

리액트 프로젝트 생성

- vite 도구
 - 시작 명령어
 - 빌드 명령어 : npm run build
 - 개발서버 시작 명령어 : npm run dev

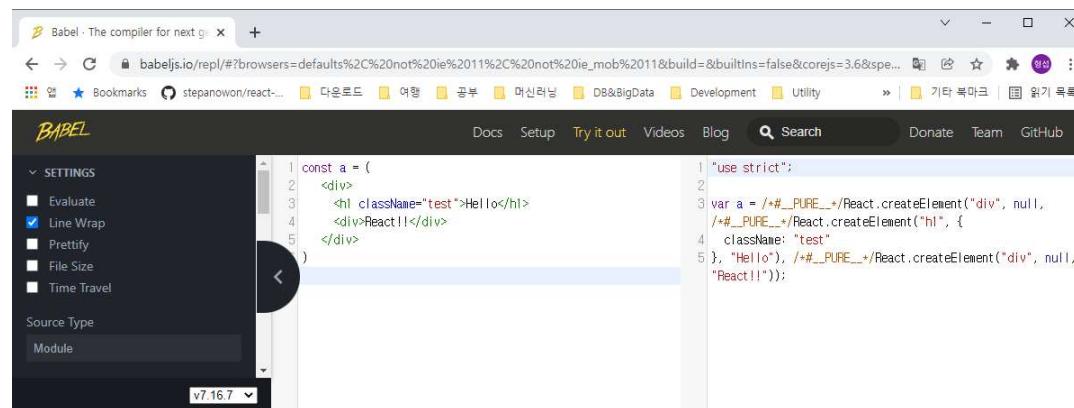
리액트 프로젝트 생성

- vite 도구
 - 폴더 구조
 - src : JS, TS 코드를 이곳에 작성함
 - 시작 진입점 : src/main.jsx
 - public : 정적 파일, 자원이 필요하다면 이곳에 배치함.
 - 이 폴더는 직접 생성해야 함.
 - dist : 빌드 후 생성된 아티팩트가 이곳에 저장됨.

리액트 기초 개념

JSX란?

- NOT HTML!!
- 선언적 XML 스타일의 자바스크립트 확장 문법
 - Javascript 코드로 변환되어 실행됨.
- babel repl 도구를 이용한 변환 (<https://babeljs.io/repl/>)



The screenshot shows the Babel REPL interface. On the left, there's a sidebar with 'SETTINGS' expanded, showing options like 'Evaluate', 'Line Wrap' (which is checked), 'Prettyfy', 'File Size', and 'Time Travel'. Below that is a 'Source Type' dropdown set to 'Module'. The main area has two panes: a code editor on the left containing JSX code, and a results pane on the right containing the generated JavaScript code. The JSX code is:

```
1 const a = (
2   <div>
3     <h1 className="test">Hello</h1>
4     <div>React!!</div>
5   </div>
)

```

The generated JavaScript code is:

```
1 "use strict";
2
3 var a = /*#__PURE__*/React.createElement("div", null,
4   /*#__PURE__*/React.createElement("h1", {
5     className: "test"
5   }, "Hello"), /*#__PURE__*/React.createElement("div", null,
6   "React!!"));

```

리액트 기초 개념

- JSX는 선택적 요소임
 - 반드시 사용해야 하는 것은 아님. 하지만 장점이 많음
 - 사실상의 필수
 - UI를 표현하기에 더 적합함. 특히 HTML, XML 의 트리 구조
 - 애플리케이션의 구조를 시각화하기에 더 좋음
 - 자바스크립트 코드이므로 언어의 의미가 변형되지 않음

리액트 기초 개념

- 주의사항
 - 태그의 Attribute는 카멜 표기법(camel casing)을 준수함
 - 예) onclick --> onClick
 - HTML : <button onclick="start()" />
 - JSX : <button onClick={start} />

리액트 기초 개념

- 주의사항
 - Attribute 이름이 DOM API 스펙에 기반을 두고 있음.
 - CSS클래스를 적용할 때 class attribute를 사용하지만 JSX에서는 className 속성을 사용해야 함. --> JS 코드이기 때문에...
 - 비교 대상
 - HTML : <div id="a" class="test"></div>
 - JS : document.getElementById("a").className="test";
 - HTML : <div class="test">Hello</div>
 - JSX : <div className="test">Hello</div>

리액트 기초 개념

- JSX 표현식에 동적으로 값을 넣고자 한다면?
 - { } 보간법(interpolation)을 사용함.
 - { } 내부에 값, 메서드의 리턴값, 속성 등이 위치할 수 있음.
 - { } 에 복잡한 자바스크립트 구문을 배치할 수 없음
 - 예1) if문을 { } 내부에 작성할 수 없음.
 - 3항 연산식은 허용함({ a? b:c })
 - 예2) for 문 반복문을 { } 내부에 작성할 수 없음
 - 외부에서 연산처리하여 변수에 값을 저장한 후 보간해야 함.
 - {} 에는 값이 있는 구문을 작성해야 함

리액트 기초 개념

- JSX 표현식에 동적으로 값을 넣고자 한다면?
 - 값은 모두 HTML Encoding 되어 출력됨
 - XSS 공격 때문에 자동으로 HTML Encoding을 수행함.
 - 그럼에도 불구하고 HTML 그대로 출력하려면 dangerouslySetInnerHTML 특성을 사용함

```
<span dangerouslySetInnerHTML={{ __html:msg }} />
```

- XSS : Cross Site Scripting
- 공격할 때 script 태그 형태를 많이 이용한다.
- <script> 태그가 브라우저 화면안에 DOM 으로 추가되지 못하게 하기 위함이다.

리액트 기초 개념

- 단일 루트 노드
 - 단일 루트 요소만 렌더링할 수 있음.
 - 여러개의 요소를 렌더링하려면 <div></div>와 같은 요소로 감싸주어야 함.
- XML 문법임으로 당연하다.
- 여러 노드를 출력하기 위해서 과거에는 <div></div> 를 사용했었는데
- 최근에는 오른쪽처럼 비어있는 노드를 사용한다.
- 비어있는 노드가 root element 역할을 한다.
- 실제 렌더링 될 때 <></>은 랜더링 되지 않는다.

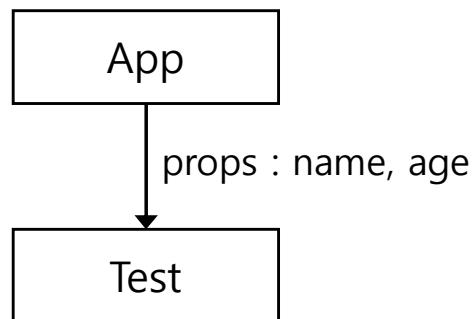
```
return (
  <div>Hello</div>
  <div>World</div>
);
```

```
return (
  <>
  <div>Hello</div>
  <div>World</div>
  </>
);
```

리액트 기초 개념

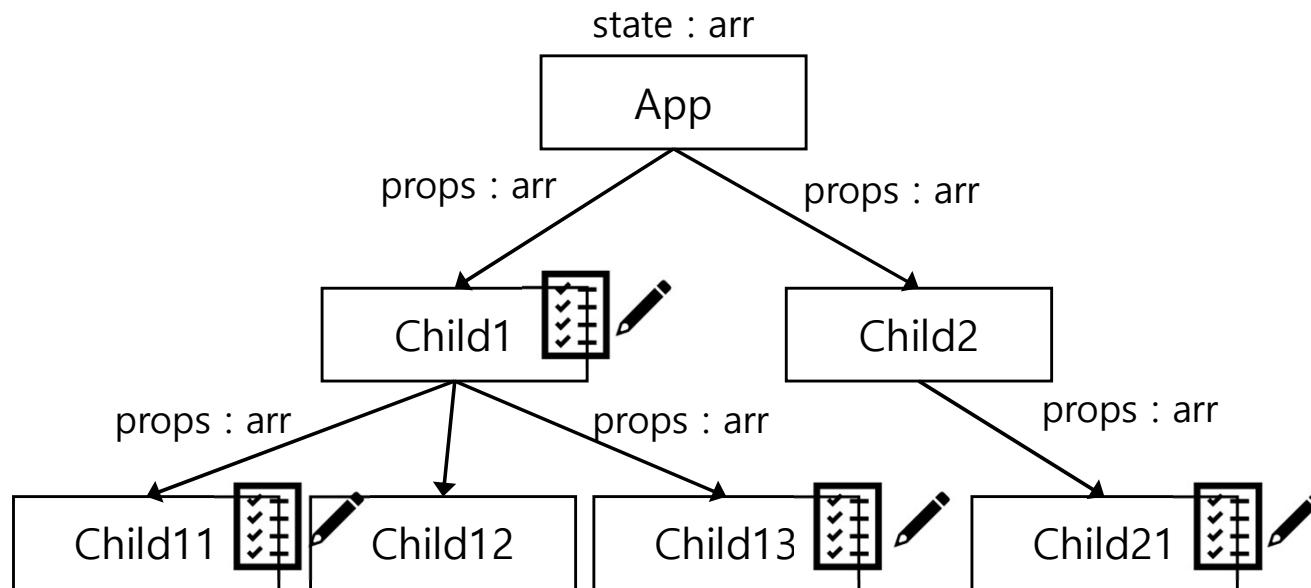
속성 : props

- 컴포넌트가 외부로부터 데이터를 전달받기 위해 사용
 - 부모 컴포넌트 --> 자식컴포넌트로 정보 전달
 - 전달받은 속성의 값은 그 컴포넌트에서 변경하지 않음



리액트 기초 개념

- Props 는 변경할 수 없다.
- 기본값(number, string, Boolean) 은 변경 안된다.
- 객체를 전달 받는 경우에는 메모리 참조임으로 객체자체를 변경할 수 없지만 객체의 내부 값은 변경이 가능하다.
- 하지만 객체의 경우에도 객체의 값을 변경하지 않는 것이 권장한다.



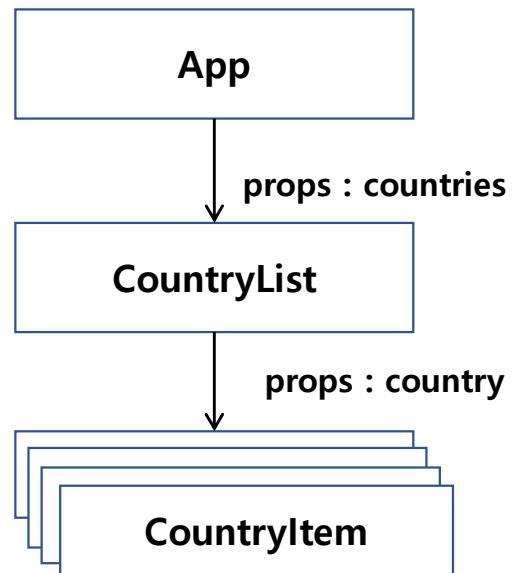
리액트 기초 개념

```
<Sub1 name={name} age="10"/>
```

```
const Sub1 = (props) => {
  return <p>I am Sub1, name : {props.name}, age : {props.age}</p>
}
```

리액트 기초 개념

- 컴포넌트 세분화!
 - 컴포넌트를 어떻게 설계할 것인지는 개발자 마음
 - 배열등의 데이터로 반복적인 UI 가 출력이 된다면 반복되는 부분을 책임지는 컴포넌트를 설계할 것이 권장



리액트 기초 개념

- 컴포넌트 세분화!
 - 컴포넌트 세분화 장점
 - 컴포넌트의 기능이 단순해짐으로 에러가 발생할 가능성이 줄어들고 디버깅이 쉬워짐
 - 컴포넌트의 재사용이 높아짐
 - 하나의 컴포넌트에 많은 내용이 포함되면 UI 와 기능이 모두 일치한 경우에만 재사용 가능하다. 하지만 세분화 시키면 그만큼 재사용성이 증가된다.
 - 렌더링 성능을 최적화 하기 용이하다.
 - 리액트는 컴포넌트 단위로 렌더링할지 여부를 결정한다.
 - List 전체의 컴포넌트로 작성되었다면 item 하나만 변경이 되어도 List 컴포넌트가 리렌더링 된다.
 - 하지만 List – Item 으로 세분화 하게 되면 전체 항목중 변경된 Item 의 컴포넌트만 리랜더링 되게 된다.

리액트 기초 개념

상태 : state

- 컴포넌트가 보유하는 데이터
- 상태의 변경은 보유하고 있는 컴포넌트에서만 수행함.
 - 속성을 통해 자식컴포넌트로 전달된 경우 자식컴포넌트에서 상태 변경 불가.
 - 속성을 통해서 자식 컴포넌트로 전달할 수 있음.
 - 상태가 변경되면 컴포넌트는 다시 랜더링 된다.

리액트 기초 개념

- 상태 컴포넌트와 무상태 컴포넌트
 - 상태 컴포넌트 : stateful component
 - 자신의 상태를 가지는 컴포넌트
 - 무상태 컴포넌트 : stateless component
 - 자신의 상태가 없으며 속성을 통해서 부모 컴포넌트로부터 데이터를 전달받아야만 하는 컴포넌트
 - 컴포넌트의 재사용성은 무상태 컴포넌트가 좋음

리액트 기초 개념

- 상태의 초기화와 변경
 - 상태의 초기화를 위해 useState()라는 React Hook을 사용함
 - const [getter, setter] = useState(defaultValue);
 - 리턴값(배열)의 첫번째 아이템은 읽기전용의 속성, 두번째 아이템은 setter 기능의 메서드
 - 상태의 변경은 setter 메서드를 이용해야 함.

스타일링

HTML에서의 스타일 지정 방법

- 인라인 스타일
 - <div style="width:200px; height:60px; color:yellow; border:solid 1px gray; background-color:purple;">Hello</div>
 - HTML에서는 동일한 스타일의 요소가 여러개 만들어질 경우 중복, 유지관리가 힘듬.
- style 태그
 - 페이지 단위 <style> 태그를 작성하여 참조하는 방법
- 외부 CSS 파일 참조
 - .css 파일을 작성하고 여러 페이지에서 <link> 태그로 참조하는 방법

스타일링

리액트에서 스타일 지정 방법

- 기존의 css 구문을 적용하는 방법
 - global style 로 적용할 수도 있고
 - css 모듈로 적용할 수도 있다.
- Js 객체로 스타일을 적용하는 방법

스타일링

전역 css 참조

- css 파일을 js 모듈처럼 import 한후 모든 컴포넌트에서 사용하는 방법
- import 'bootstrap/dist/css/bootstrap.css'
- 어디에선가 import한 css 파일의 클래스는 모든 컴포넌트에서 사용할 수 있음.

스타일링

인라인 스타일 지정

- JS 객체를 style Attribute에 지정하여 CSS 생성
- HTML에서는 권장하지 않지만 React에서는 컴포넌트 단위로 마크업+로직+스타일 을 묶어서 하나의 캡슐로 만들고자 할 때 사용

```
const styles = {  
    color:"yellow", backgroundColor:"purple"  
}
```

- css 파일 정의와 비슷해 보이지만 JS 객체이다.
- 속성과 속성 사이를 , 으로 구분
- 문자열을 “” 로 묶는다.
- Camel case 를 이용한다. css 에서는 background-color 로 사용하지만 JS 객체에서는 backgroundColor

스타일링

인라인 스타일 지정

- 선언된 JS CSS 는 컴포넌트에 style 속성으로 이용.

```
<div style={styles}>Hello</div>
```

- inline style 이라는 것은 HTML 과 다르게 JS 객체를 적용하는 것임으로 여러 컴포넌트에서 재사용이 가능
- 웹 디자이너와 협업할 때 웹디가 작성한 css 를 JS CSS 객체로 변환해주어야 한다.
- 이를 지원하는 도구가 있다.
- <https://transform.tools/css-to-js>

스타일링

CSS 모듈

- Inline style 은 js 객체로 css 를 표현함으로 충돌이 발생하지는 않지만
- CSS 를 js 문법에 맞게 작성해야 한다는 불편함이 있다.
- 전역 css 는 css 문법으로 작성해서 사용이 가능하기는 하지만 중복에 의한 문제가 있다.
- 이 문제를 해결하기 위한 것이 css 모듈이다.
- 즉 css 모듈은 css 를 그대로 이용하면서 특정 component 에 종속적으로 적용하기 위함이다.
- 알아서 hash 값을 추가해서 충돌을 피해준다.

스타일링

CSS 모듈

- 클래스 명을 해시를 이용해 충돌나지 않는 이름으로 변경
- .module.css로 끝나는 이름을 사용해야 함.
- CSS 모듈은 js object 기반이 아니라 css class 기반으로 작성
- css 파일을 그대로 이용하는데 import 할때 js object 처럼 획득해서 object 명으로 css class 를 이용한다는 개념이다.
- 결국 css 를 그대로 이용하면서 js object 로 구분해서 사용이 가능하다는 개념이다.

```
.test { color:blue; background-color: bisque; }
```

```
.....  
import AppCssModule from "./App.module.css";  
.....  
  
<h2 className={AppCssModule.test}>Hello {msg}!</h2>
```

Props Type

속성의 유효성 검증

- 컴포넌트 기반으로 개발할 때 컴포넌트의 속성은 다음을 쉽게 식별할 수 있어야 함.
 - 컴포넌트에서 사용가능한 속성이 무엇인지...
 - 필수 속성은 무엇인지...
 - 속성에 전달할 수 있는 값의 타입은 무엇인지...
- 이를 위해 속성의 유효성 검사 기능이 필요함.

Props Type

유효성 검증 방법

- PropTypes
 - 런타임시에 타입을 검사함
 - 실제로 전달되는 값을 이용해 타입을 확인하고 경고를 일으킴
 - 컴포넌트의 static 멤버로 검증 정보를 부여

Props Type

PropTypes 적용

- npm install prop-types

```
import PropTypes from "prop-types";

Calc.propTypes = {
  x: PropTypes.number.isRequired,
  y: PropTypes.number.isRequired,
};
```

- 컴포넌트 이름 + propTypes = { }
- 컴포넌트 외부에 선언되어야 한다.

Props Type

지정 가능한 유효성 검증 타입

- 단순 타입
 - PropTypes.array
 - PropTypes.bool
 - PropTypes.func
 - PropTypes.number
 - PropTypes.object
 - PropTypes.string

Props Type

지정 가능한 유효성 검증 타입

- 복잡한 객체, 배열 속성
 - PropTypes.instanceOf(Customer)
 - PropTypes.oneOf(['+', '*', '/'])
 - PropTypes.oneOfType([PropTypes.number, PropTypes.string])
 - PropTypes.arrayOf(PropTypes.object)

Props Type

- 지정 가능한 유효성 검증 타입

- 복잡한 객체 속성

```
PropTypes.shape({
  name: PropTypes.string.isRequired,
  age: PropTypes.number
})
```

- 함수를 이용한 커스텀 유효성 검증

```
const calcChecker = (props, propName, componentName) => {
  if (propName === "oper") {
    if (props[propName] !== "+" && props[propName] !== "*") {
      return new Error(`.${propName}속성의 값은
        반드시 '+' 혹은 '*'만 허용합니다(at ${componentName}).`);
    }
  }
};

Calc.propTypes = {
  x: PropTypes.number.isRequired,
  y: PropTypes.number.isRequired,
  oper: calcChecker,
};
```

Props Type

- 속성의 기본값 지정
 - 컴포넌트에 defaultProps static 속성을 추가함.

```
.....  
Calc.defaultProps = {  
  x: 100,  
  y: 20,  
  oper: "+",  
};
```

리액트 이벤트

- React 이벤트
 - 성능을 위해 이벤트를 리액트 트리가 렌더되는 root DOM container에 연결하고 이벤트를 위임(delegation) 처리함.
 - 이벤트가 발생하면 React가 DOM Root에서 적절한 컴포넌트 요소로 바인딩함.
 - camel Casing 규칙을 준수해야 함.
 - <button onClick={()=> alert('hello')}> OK </button>
- React DOM 은 가상 DOM 이다.
- 실제 유저가 이벤트를 가하는 부분은 실 DOM 이다.
- 이 부분의 차이를 해결하기 위해서 코드에서 가상 Dom 에 이벤트를 등록하면 모두 root DOM 에서 처리하게 해 준다는 개념이다.
- Root DOM 에서 실 DOM 에 적절하게 이벤트를 바인딩하게 된다.

리액트 이벤트

- 이벤트 적용 방법

- 외부 함수 바인딩

```
const eventHandler = () => { ..... }
```

```
//JSX 내부에서  
<input type="text" ... onChange={eventHandler} />
```

- 익명 함수 바인딩

```
//JSX 내부에서  
<button onClick={() => { ..... }}>버튼</button>
```

- 이벤트 핸들러 함수의 첫번째 인자를 이용해 이벤트 아규먼트 값을 받을 수 있음

```
const eventHandler = (e) => {  
  setValue(e.target.value);  
}
```

리액트 이벤트

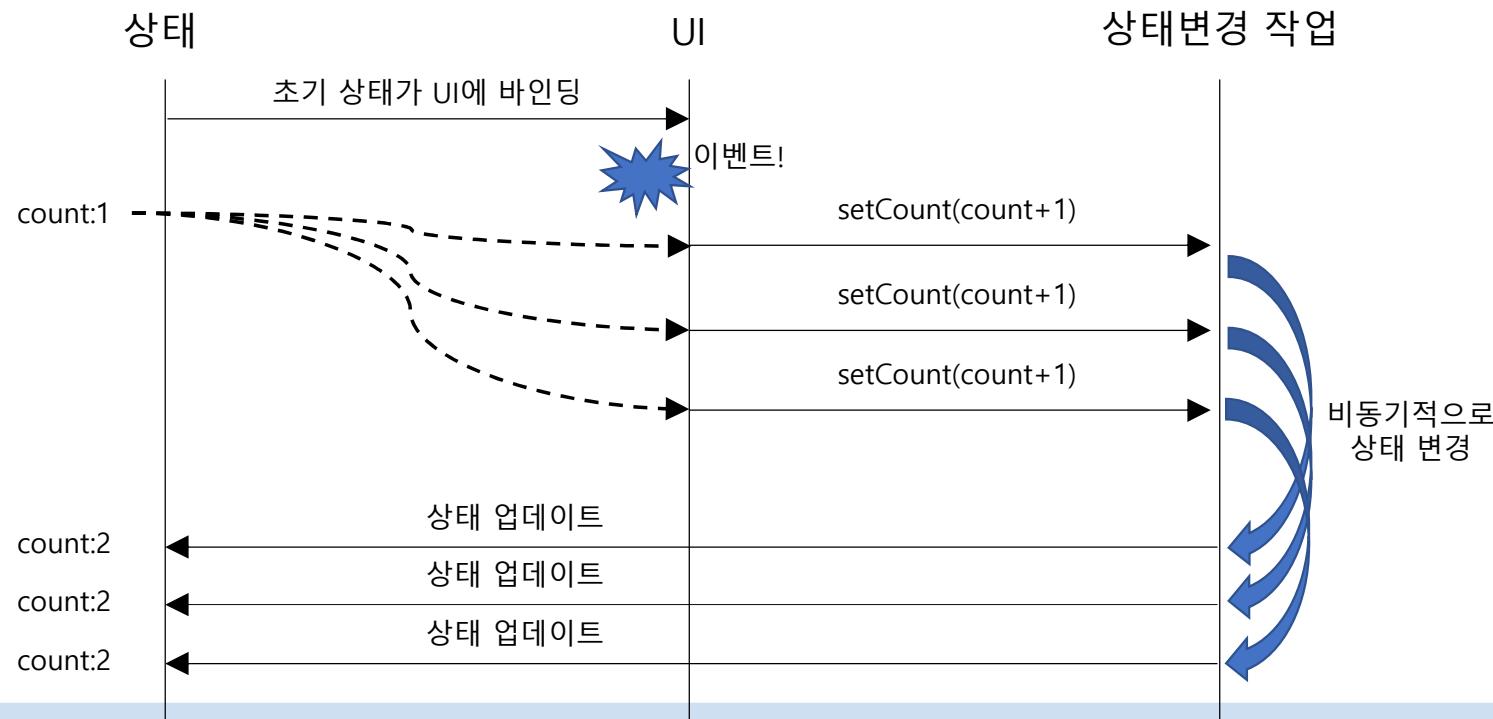
- 상태 변경을 위해서...
 - useState()로 만든 setter 함수를 이용함.
- 이벤트 핸들러 함수 내부에서 같은 상태를 여러번 변경하면?
 - increment 함수 내부에서 count 증가를 3번 해보자.

```
const increment = () => {
  setCount(count + 1);
  setCount(count + 1);
  setCount(count + 1);
};
```

- 증가 버튼을 클릭하면 3씩 증가하는가? No
- 상태 변경을 위한 setter 함수는 비동기로 실행된다.
- 그럼으로 이벤트 핸들러에서 여러 번 setter 함수를 호출하게 되면 문제 발생할 수 있다.

리액트 이벤트

- 상태변경은 기본적으로 비동기 작업
 - 왜 이렇게 설계했을까?
 - 렌더링 성능 때문에...



리액트 이벤트

- 권장사항
 - 상태 변경이 비동기적으로 진행되므로 하나의 이벤트 핸들러 함수 내부에서 동일한 상태를 여러번 변경하지 않는 것이 바람직함.
- 불가피한 상황이라면 어떤 방법을 적용해야 하나?
 - 함수를 이용해 리턴값으로 상태를 변경하도록 작성함.

```
const increment = () => {
  setCount((count) => count + 1);
  setCount((count) => count + 1);
  setCount((count) => count + 1);
};
```

제어 컴포넌트와 비제어 컴포넌트

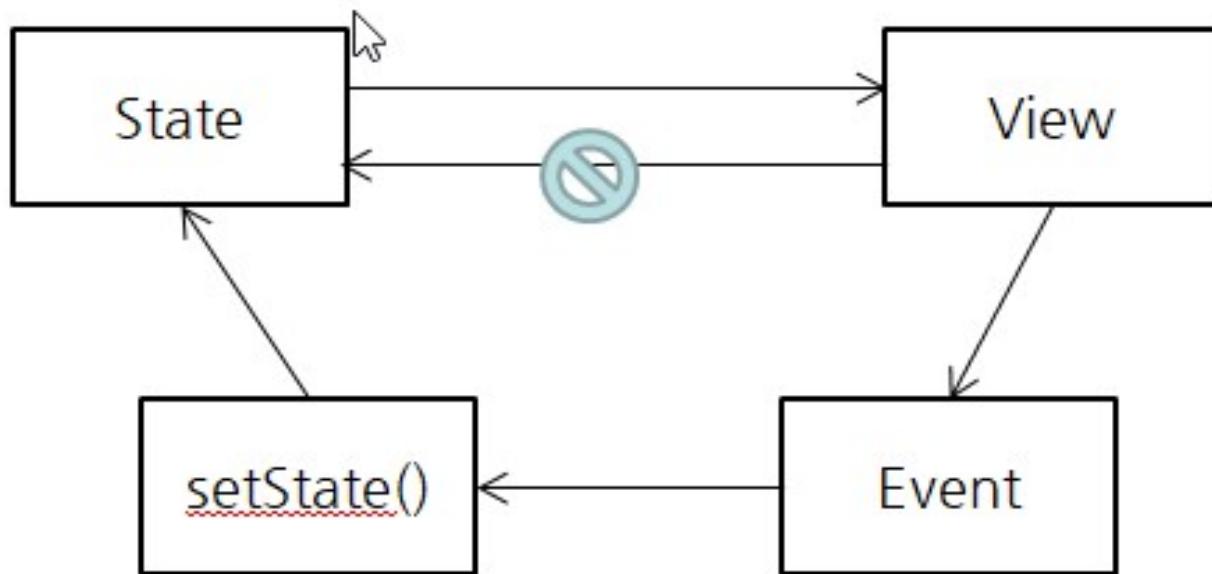
- 제어 컴포넌트와 비제어 컴포넌트
 - 제어 컴포넌트(Controlled Component)
 - 입력필드의 값이 state나 props에 의해 제어되는 컴포넌트
 - state가 바뀌지 않는 한 입력값을 변경할 수 없다. 변경하려면 상태를 바꾸도록 이벤트 처리가 요구됨.
 - 비제어 컴포넌트(Uncontrolled Component)
 - 입력필드의 값이 state나 props에 의해 제어되지 않는 컴포넌트
 - 사용자가 쉽게 변경할 수 있지만 입력값을 알아내려면 실제 HTML DOM에 접근해야 하는 단점이 있다.
- 둘 중 제어 컴포넌트를 더 권장함.
 - 비제어 컴포넌트는 실제 DOM을 접근해야 하므로 고비용임.

제어 컴포넌트와 비제어 컴포넌트

- 제어 컴포넌트
 - 제어 컴포넌트는 UI 요소의 값이 상태나 속성에 강하게 연결되어 있으므로 변경이 불가능함.
 - 상태를 변경하면 UI 요소의 값이 바뀜. 따라서 이벤트를 이용해 상태를 변경해야 함.
- <input type="text" value={x} />
- 이렇게만 하면 유저 입력이 화면에 적용되지 않는다.

제어 컴포넌트와 비제어 컴포넌트

- 제어 컴포넌트에 사용자 입력값이 반영되게 처리
 - 양방향 바인딩을 제공하지 않음. 이벤트 핸들러를 통해 상태를 변경하도록 해야 함.



제어 컴포넌트와 비제어 컴포넌트

- 제어가 가능하도록 이벤트 처리 적용

```
const changeValue = (e) => {
  let newValue: number = parseInt(e.target.value);
  if (isNaN(newValue)) newValue = 0;
  if (e.target.id === "x") setX(newValue);
  else setY(newValue);
};

.....
<input id="x" type="text" value={x} onChange={changeValue} />
```

제어 컴포넌트와 비제어 컴포넌트

비제어 컴포넌트

- state나 props에 종속되지 않기 때문에 이벤트를 이용해 처리하지 않아도 사용자가 값을 수정할 수 있음.
- 비제어 컴포넌트지만 초기값을 부여하고자 할 때
 - default~로 시작하는 Attribute를 이용함.
 - ex) value --> defaultValue
 - ex) checked --> defaultChecked
- 사용자가 HTML DOM 요소에서 입력한 값을 획득하기 위해 ref 특성을 사용할 수 있음.
 - 실제 HTML DOM에 접근하는 것은 고비용의 작업임. 꼭 필요한 경우가 아니라면 권장하지 않음
 - state나 props로 해결하기 어려운 경우에는 한가지 대안이 될 수 있음.

제어 컴포넌트와 비제어 컴포넌트

- 비제어 컴포넌트

```
const elemX = useRef(null);
const elemY = useRef(null);

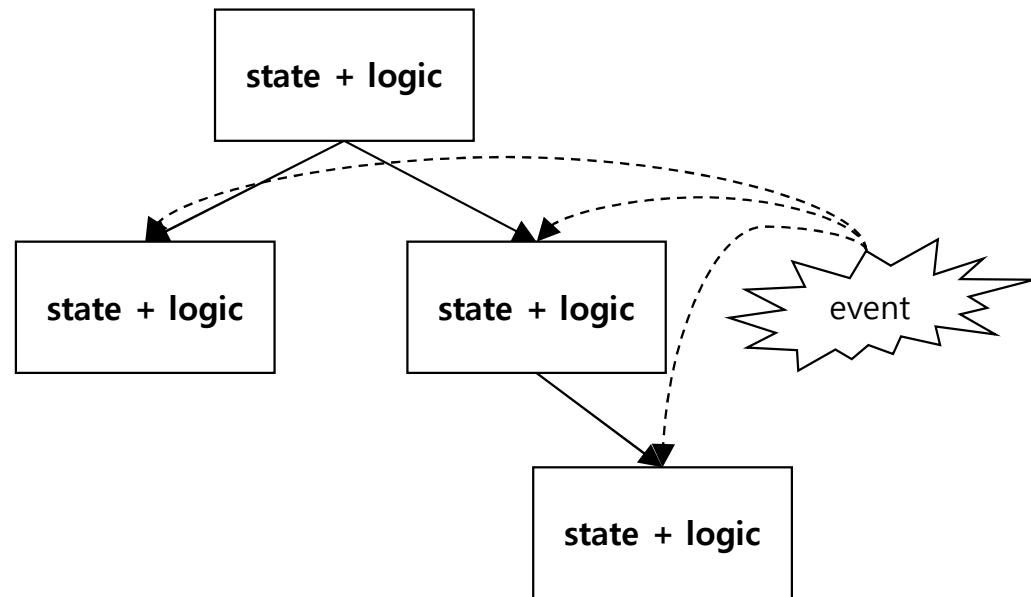
.....
return (
  <div className="container">
    X : <input id="x" type="text" defaultValue={x} ref={elemX} />
    <br />
    Y : <input id="y" type="text" defaultValue={y} ref={elemY} />
  </div>
);
```

컨테이너 컴포넌트와 표현 컴포넌트

- 상태의 보유 여부에 따른 컴포넌트의 유형
 - 상태 컴포넌트 : stateful component
 - 상태(state)와 상태를 변경하는 기능을 보유하는 컴포넌트
 - 비상태 컴포넌트 : stateless component
 - 상태가 없으며 부모 컴포넌트로부터 props를 전달받아 UI를 렌더링할 목적의 컴포넌트
- 비즈니스 로직 기능과 표현 기능으로 구분하는 방법
 - 표현 컴포넌트 : presentational component
 - 컨테이너로부터 props를 전달받아 UI를 렌더링하는 기능을 수행함
 - 높은 재사용성, 행동 로직과의 분리
 - 컨테이너 컴포넌트 : container component
 - UI와 스타일 정보를 포함하지 않음
 - 상태와 상태 변경 로직만을 가짐.

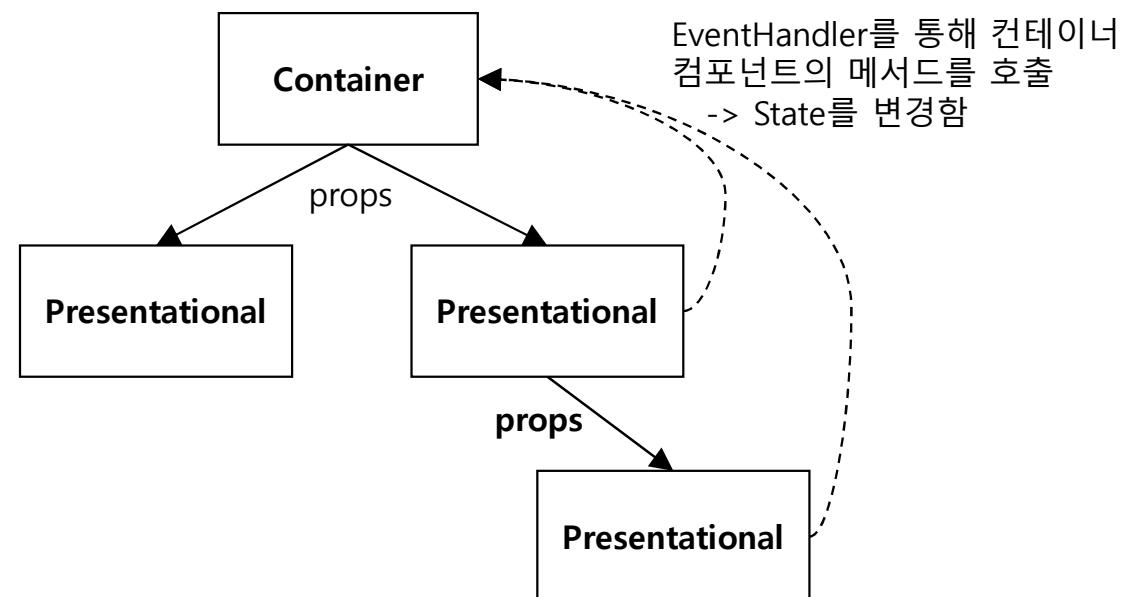
컨테이너 컴포넌트와 표현 컴포넌트

- 컴포넌트 각각 상태를 보유한다면?
 - 애플리케이션의 상태 관리가 복잡해짐.
 - 디버깅이 어려워짐



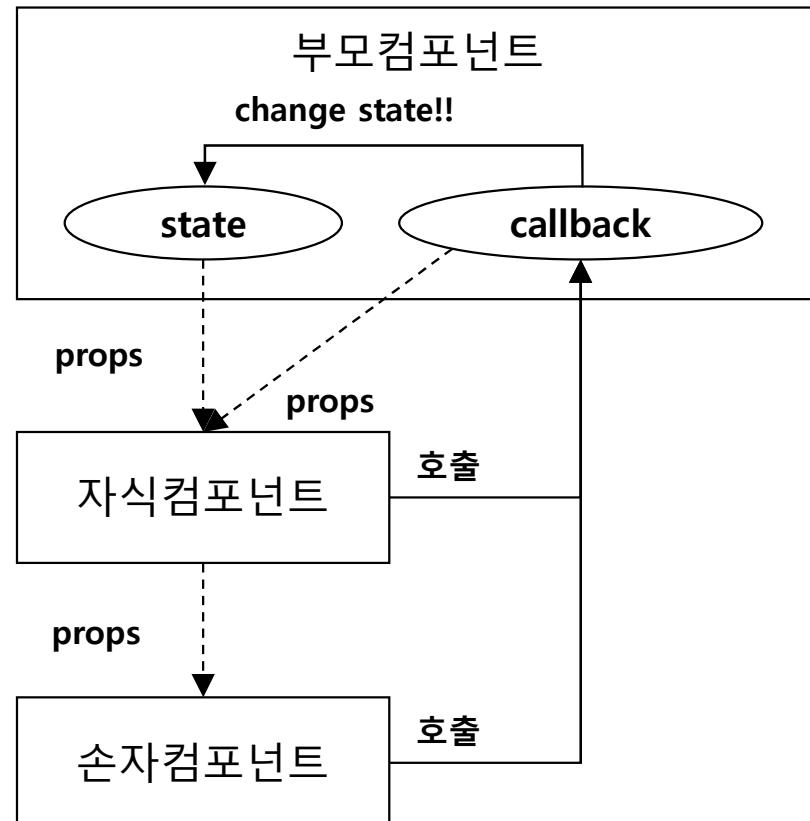
컨테이너 컴포넌트와 표현 컴포넌트

- 그렇다면 권장 사항은?
 - 하나의 컨테이너 컴포넌트
 - 여러개의 자식 표현 컴포넌트
 - 그리고 속성으로 전달



컨테이너 컴포넌트와 표현 컴포넌트

- 부모, 자식 컴포넌트 간의 정보 전달 방법
 - 부모 → 자식 으로의 정보 전달 방법
 - props를 이용해 전달함.
 - 자식 → 부모로의 정보 전달 방법
 - 부모 컴포넌트의 콜백 함수(메서드)를 props를 이용해 자식 컴포넌트로 전달함.
 - 속성을 이용해 함수(메서드)를 전달할 수 있음.



클래스 컴포넌트

- 함수 컴포넌트
 - 최근 더 많이 사용됨 → 더 간결하기 때문에...
 - 렌더링 성능이 클래스 컴포넌트에 비해 약간 좋음(큰 차이 없음)
 - 컴포넌트의 생명주기 이벤트 처리를 React Hook을 이용함
- 클래스 컴포넌트
 - 다양한 생명주기 이벤트 메서드를 사용할 수 있음
 - 작성은 더 복잡하지만 자바, c# 등의 개발자라면 오히려 익숙할 수 있음

클래스 컴포넌트

- 무엇을 사용할까?
 - 정답은 없음
 - 다양한 생명주기 이벤트가 필요하다면? 클래스 컴포넌트
 - 간결한 코드가 필요하다면? 함수 컴포넌트
 - 한 프로젝트 내에서 혼용해서 사용할 수 있음

클래스 컴포넌트

- 기본 형태
 - React.Component 를 상속받도록 작성

```
import React, { Component } from 'react'

export default class Test extends Component {
  render() {
    return (
      <div>Test</div>
    )
  }
}
```

클래스 컴포넌트

- 속성 사용

```
import React, { Component } from "react";

export default class Test extends Component {
  render() {
    return <div>{this.props.name}님의 나이는
      {this.props.age}입니다.</div>;
  }
}
```

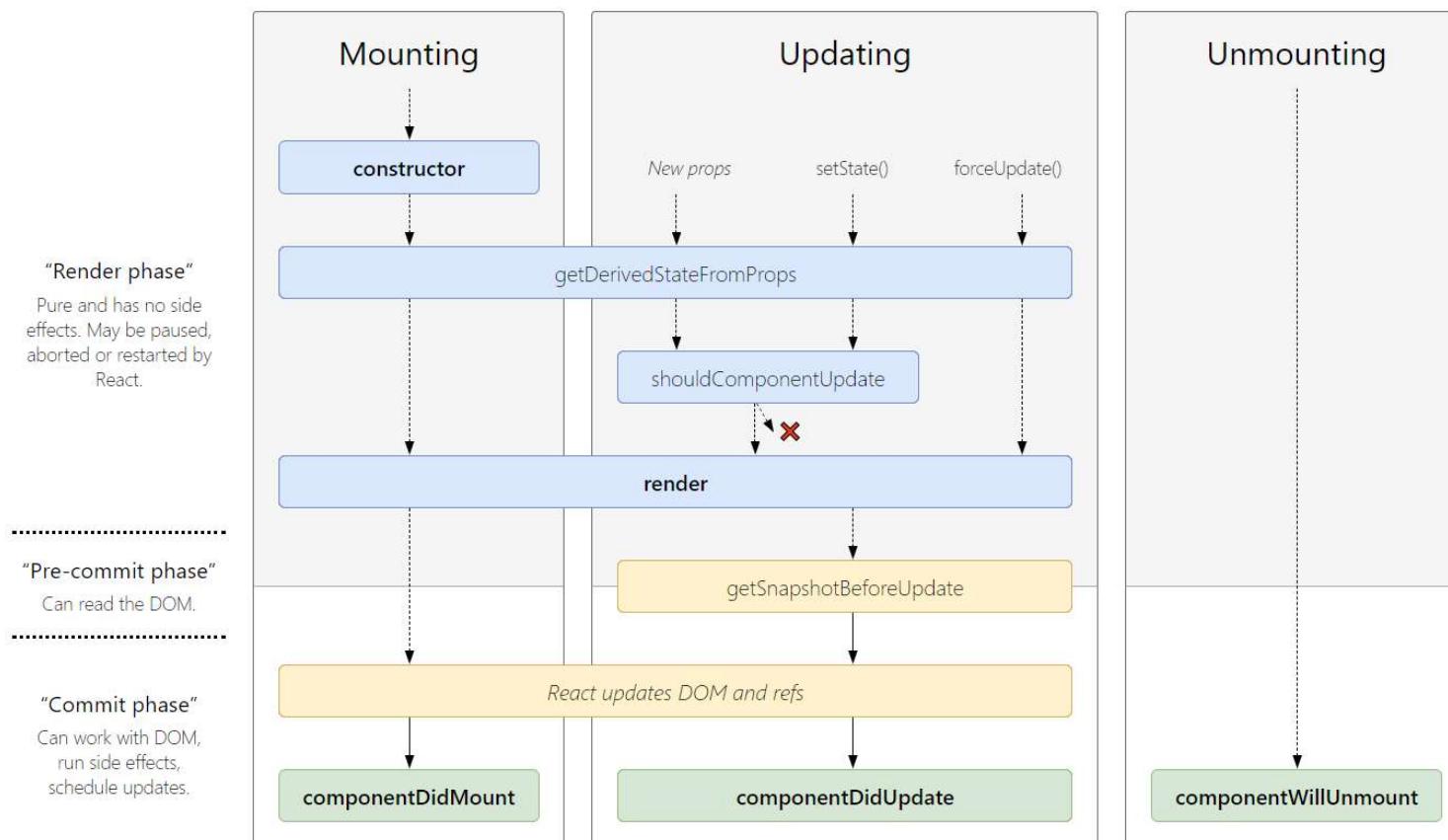
클래스 컴포넌트

- 상태 사용

```
import React, { Component } from "react";

export default class Test extends Component {
  state = {
    name: "홍길동",
  };
  render() {
    return <div>Test</div>;
  }
}
```

컴포넌트 생명주기



컴포넌트 생명주기

- 클래스 컴포넌트에서만 사용 가능
- 컴포넌트 마운트, 언마운트, 업데이터될때 자동 실행되는 메서드
- 마운트/언마운트는 화면에 보이는 것과는 상관 없이 컴포넌트 트리에 새로운 컴포넌트가 추가되거나 제거되는 순간을 의미한다.
- 마운트
 - Constructor -> getDerivedStateFromProps -> render -> componentDidMount
- Update
 - getDerivedStateFromProps -> shouldComponentUpdate -> render -> getSnapshotBeforeUpdate -> componentDidUpdate
- Unmounting
 - componentWillUnmount

컴포넌트 생명주기

constructor(props) : 생성자

- 마운트 되기 전에 호출되며 상태를 초기화하기 위한 최적의 시점
- 인자는 속성이며 constructor 내부의 첫 줄에 반드시 super(props)가 포함되어야 함.
- 상태가 없다면 생성자를 구현할 필요가 없음
- 부모 컴포넌트로부터 속성을 전달받아 상태를 초기화

```
constructor(props) {  
    super(props)  
    this.state = { name: "홍길동" }  
}
```

컴포넌트 생명주기

render()

- Component의 render() 메서드
- 일반적으로 JSX를 사용하여 작성함.
- return 된 객체들이 Virtual DOM에 렌더링됨.
- 이 메서드 내부에서 setState() 해서는 안됨 --> 무한 루프

컴포넌트 생명주기

- componentDidMount()
 - 컴포넌트의 마운트가 완료된 후에 호출.
 - DOM의 구조를 확인하고 난 뒤 DOM에 대한 초기화를 수행할 때
 - 이 단계에서 setState()가 호출되면 re-render가 일어나지만 실제 브라우저의 HTML DOM에 반영되기 전에 호출
 - 따라서 render() 가 두번 호출되지만 사용자는 최종 상태만 조회하게 됨.
 - 가상 Drender 는 가상 DOM 에 쓰기를 하는 것이다.
 - 그리고 현재의 OM 과 이전 DOM 을 비교해서 브라우저 DOM 을 업데이트 된다.
 - 주로 이 함수에서는 브라우저에 출력된 후에 외부 리소스 연결(websocket 등)하는 작업을 한다.
 - 먼저 화면을 출력한후 시간이 오래 걸리는 작업을 비동기로 진행해서 화면업데이트 하는데 이런 작업을 주로 이 함수에서 한다.

컴포넌트 생명주기

- 속성, 상태 변경 시의 흐름
 - static getDerivedStateFromProps()
 - 부모의 상태값이 변경될 때마다 호출.
 - 최초에도 호출.
 - 부모가 지정한 속성 값을 받아서 상태를 변경할 때 주로 이용
 - 인자는 props – 부모가 지정한 속성, state – 이 컴포넌트의 상태
 - 이 함수에서는 객체로 변경된 상태를 리턴해야 한다.
 - 이 메서드에서 null을 리턴하면 상태를 업데이트하지 않겠다는 의미.

```
static getDerivedStateFromProps(nextProps, prevState){  
    if(nextProps.someValue!==prevState.someValue){  
        return { someState: nextProps.someValue};  
    }  
    else return null;  
}
```

컴포넌트 생명주기

shouldComponentUpdate(nextProps, nextState)

- 전달 인자 : 새로운 속성, 새로운 상태
- 리턴값이 중요함
 - 리턴값이 true : 이후 단계의 메서드를 실행함(render)
 - 리턴값이 false : 이후 단계의 메서드를 실행하지 않음.
- 이 메서드를 작성하지 않으면 기본적으로 true를 리턴함

컴포넌트 생명주기

- `getSnapshotBeforeUpdate()`
 - `render()`가 호출된 직후 Virtual DOM에는 업데이트가 되고 나면 호출됨
 - 아직은 Browser DOM에 업데이트 되기 전임
 - 브라우저 업데이트 이전에 DOM 상태를 확인해서 설정작업등에 사용
 - 리턴 값을 스냅샷이라고 하는데 이곳에서 리턴한 스냅샷이 `componentDidUpdate` 세번째 인자로 전달된다.
 - 자주 사용되는 메서드는 아님.

컴포넌트 생명주기

- componentDidUpdate(prevProps, prevState)
 - DOM 업데이트가 일어난 직후에 실행됨.
 - 마운트 시에는 실행되지 않음.
 - 실제 HTML DOM이 업데이트된 후이므로 실제 DOM을 이용한 작업을 하기에 적절함
 - 현재 속성과 이전 속성을 비교하여 다른 경우에만 지정된 작업을 실행하도록 할 수 있음
 - 예) 현재의 속성과 이전 속성이 다르면 서버측으로 데이터를 요청하고자 할 때

컴포넌트 생명주기

- 컴포넌트 언마운트 시의 흐름
 - componentWillUnmount
 - 컴포넌트가 언마운트될 때 실행됨(예: 화면이 완전히 전환될 때)
 - 소켓 서버로의 네트워크 연결 해제, 이벤트 구독 해제 등의 작업을 수행할 수 있음

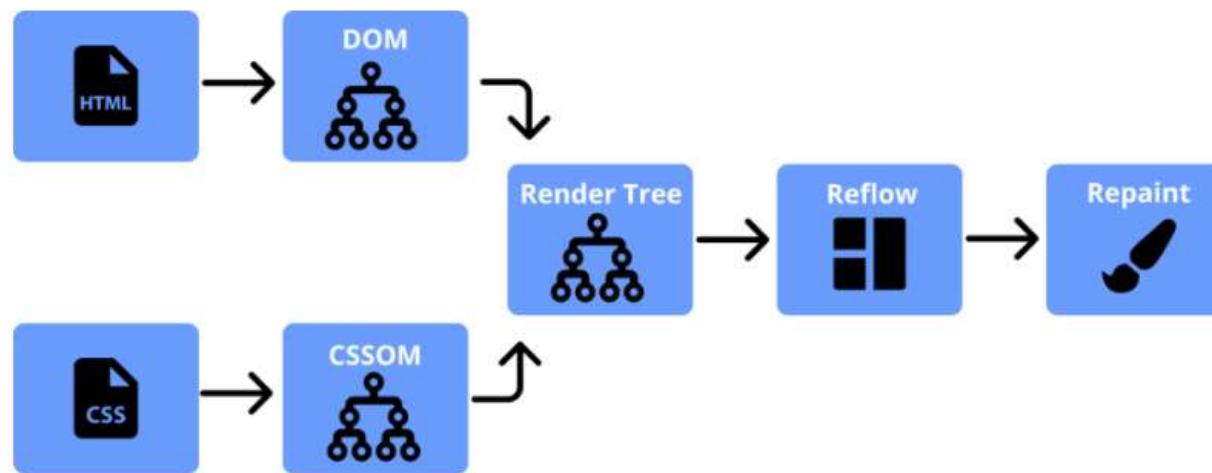
컴포넌트 생명주기

- 에러 경계 컴포넌트
 - componentDidCatch, getDerivedStateFromError
 - 위의 생명주기 메서드는 평소에는 실행안된다.
 - 앱에 에러가 발생한 경우에만 실행된다.
 - 이 함수들을 이용해서 에러 발생시에 유저에게 콜백 화면을 제공할 수 있다.
 - 혹은 에러 운영 로그를 출력할 때 사용될 수 있다.
 -
 - 위의 함수로 에러를 처리하는 컴포넌트를 에러 경계 컴포넌트라고 부른다.
 -
 - 이런 컴포넌트는 기존의 Container Component 혹은 Presentational Component에서 하면 안되고 별도의 에러 전담 컴포넌트를 만들어 주어야 한다.
 -
 - 이 라이프사이클은 함수 컴포넌트는 지원하지 않는다.

```
<ErrorBoundary>
  <Component1 />
  <Component2 />
  ...
</ErrorBoundary>
```

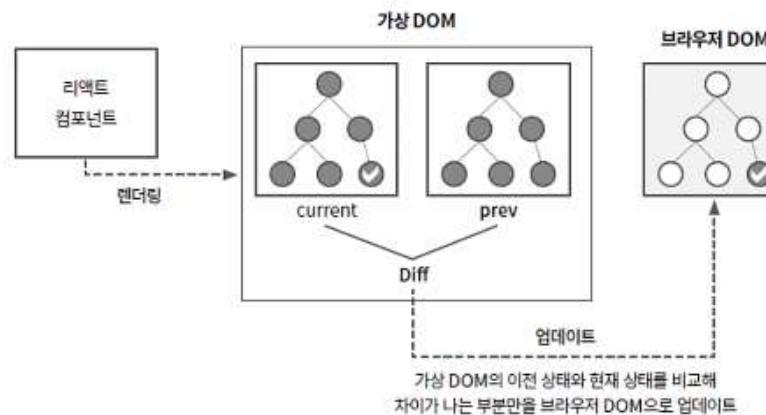
가상 DOM과 Key

- HTML DOM이 느리다?
 - DOM 조작은 빠르지만 브라우저에서 reflow, repaint 과정이 느림
 - Reflow : layout이라고도 부름. 렌더링할 DOM Tree를 새로이 만들고 HTML Element 각각의 위치를 계산하고 배치함.
 - Repaint : HTML Element에 스타일을 요소에 입히고 그려냄



가상 DOM과 Key

- React는?
 - Always re-render on update!!
 - 개발자가 원하는 출력물만을 선언적으로 작성하기 때문에 컴포넌트 전체를 re-render 하듯이 개발할 수 밖에 없음
 - 따라서 UI 성능을 위해서 Virtual DOM이 반드시 필요함
 - Virtual DOM 트리를 비교하면서 차이가 나는 부분만을 업데이트함.
 - 이것을 조정(Reconciliation) 작업이라 부름
 - 브라우저 DOM의 업데이트 로직은 개발자가 신경쓰지 않아도 됨.



가상 DOM과 Key

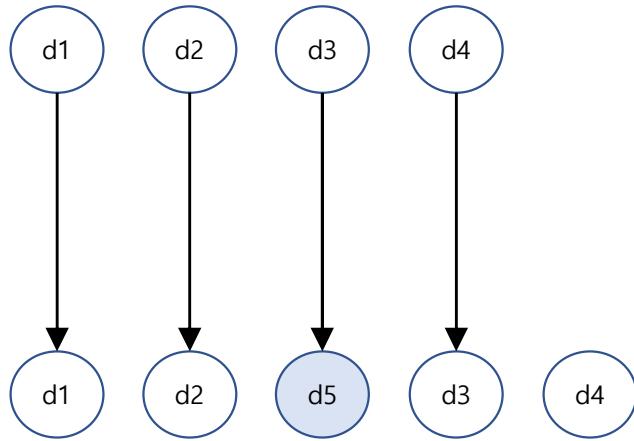
- Virtual DOM
 - DOM에 대한 추상화된 객체
 - 업데이트해야 할 정보를 메모리에 저장함.
 - Virtual DOM의 업데이트 비용은 작음
 - 실제 HTML DOM을 업데이트하는 것이 아니므로 Reflow, Repaint가 일어나지 않음
 - 이전 스냅샷과 Virtual DOM 변경 후의 스냅샷을 비교해 차이가 발생한 부분에 대해서만 실제 HTML DOM을 업데이트함.

가상 DOM과 Key

- key 특성
 - 컴포넌트 내부에서 반복적으로 자식 컴포넌트, 요소를 렌더링할 때 사용
 - 반복적인 리스트의 변경 사항을 추적하기 힘듬
 - 새로운 요소가 추가, 삽입되는 경우
 - 요소들의 순서가 변경되는 경우
 - 특정 요소가 삭제되는 경우
 - 반복적으로 렌더링할 때 key를 부여하지 않으면 React는 경고를 일으킴.
 - key 특성의 값으로는 고유한 값을 부여해야 함.(index번호(X))

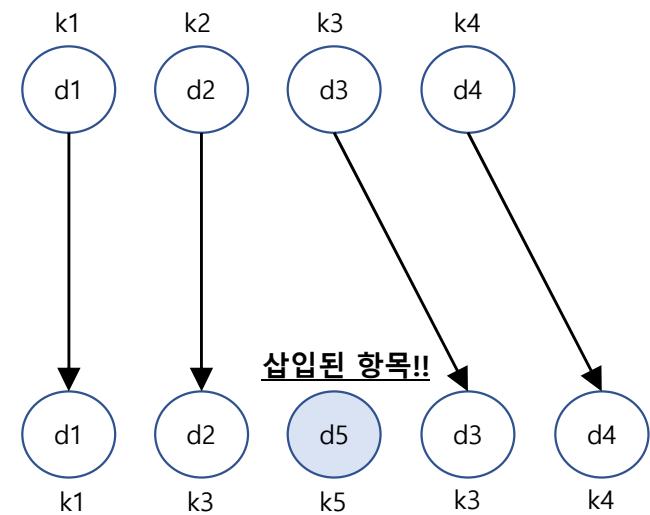
가상 DOM과 Key

Key가 사용되지 않는 경우



배열 데이터의 어떤 값이 어느 요소에
렌더링되었는지를 추적할 방법이 없으므로
배열 데이터에 대한 전체 리스트를 갱신할 수
밖에 없음

Key가 사용되는 경우

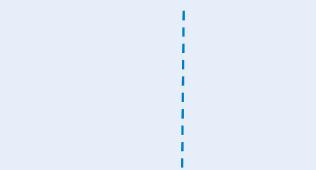


배열 데이터의 어떤 값이 어느 요소에
렌더링되었는지를 key를 통해 추적할
수 있으므로 배열 데이터의 추가, 삽입,
삭제된 것을 쉽게 추적해 비교할 수 있음



감사합니다

단단히 마음먹고 떠난 사람은
산꼭대기에 도착할 수 있다.
산은 올라가는 사람에게만 정복된다.



윌리엄 셰익스피어
William Shakespeare