

# React 19

---

# Next 14

@시코 - 시니어코딩

# What's React

## History

- 2011년 **Jordan Walke** in Facebook
- 2011년 - Facebook NewSpeed에 적용
- 2012년 - Instagram
- 2013년 - JSConf - OpenSource로 발표 (V of MVC)
- 2015년 - Netflix (Ember + **React**)
- 2016년 - react-router, redux, mobX
- 2017년 - React **Fiber** (virtual-dom)
- 2019년 - React Hooks & Fiber Reconciler (**v16.8**)
- 2021/2022년 - **React v18**
- 2024/2025년 - **React v19** 발표 예정



## 특장점 (Only **View**)

- 순수 함수로 UI 표현
  - **Virtual-DOM** vs Real-DOM
  - **JSX**를 사용하고 다른 Framework(Astro, Next, Remix, etc)과 호환
- 코 - 시니어코딩

# React Concept 리액트의 개념

## virtual-dom

- from **Fibers**: Tree & Scheduler Algorithm
- **renderer**: 렌더링 담당
- **reconciler**: DOM의 변경 관리 (비동기)
- React(Virtual)DOM vs Real(Shadow)DOM

## ReactDOM

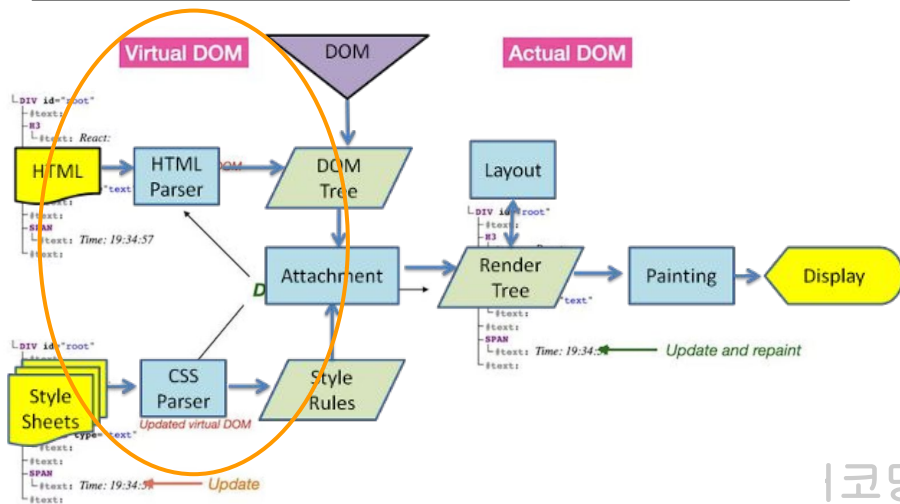
→ → →

- render the **ReactElements**.
- **createElement**(tag, attr, ...children);
- **render**(reactElement, reactRoot);

## React.Component extends ReactDOM

- render / ReactDOM / life-cycle
  - 컴포넌트 생성
- ```
const Sample = React.createClass({...});  
class Sample extends React.Component {...}  
const Sample = (props) => (<>...</>);
```

```
ReactDOM.render(  
  React.createElement('h1', null, 'A'),  
  document.getElementById('root')  
);  
  
const root = ReactDOM.createRoot(  
  document.getElementById('root')  
);  
const element = <h1>Hello, world</h1>;  
root.render(element);
```



# 설치 및 환경 설정

## 필수 설치

- node/npm: <https://nodejs.org> (by **NVM** / **Volta**)

```
$> nvm install --lts
```

```
$> nvm use v20.10.0
```

```
$> nvm alias default v20.10.0
```

```
[12:44:45:jade ~]$ nvm install --lts
Installing latest LTS version.
Downloading and installing node v16.18.0...
Downloading https://nodejs.org/dist/v16.18.0...
#####
Computing checksum with shasum -a 256
Checksums matched!
Now using node v16.18.0 (npm v8.19.2)
```

- npx: \$> npx -v # 없다면..

- \$> npm i npx -g

- vscode: <https://code.visualstudio.com/download>

## 기타

- **React DevTools** # 리액트 개발자 도구
- **yarn**: **npm i yarn -g** # brew install yarn
- vite: <https://vitejs-kor.github.io/>
- git: <https://git-scm.com/download>

Node.js v22 is now available! ↗

nodejs

Survey Help us shape the next 10 years of Node.js →

## Run JavaScript Everywhere

Node.js® is a free, open-source, cross-platform JavaScript runtime environment that lets developers create servers, web apps, command line tools and scripts.


**Download Node.js (LTS)** 📄

Downloads Node.js v20.13.1<sup>1</sup> with long-term support. Node.js can also be installed via **package managers**.

Want new features sooner? Get **Node.js v22.2.0<sup>1</sup>** instead.


## Download Visual Studio Code

Free and built on open source. Integrated Git, debugging and extensions.



↓ Windows  
Windows 8, 10, 11

User Installer 64 bit 32 bit ARM  
System Installer 64 bit 32 bit ARM  
zip 64 bit 32 bit ARM




↓ .deb  
Debian, Ubuntu

↓ .rpm  
Red Hat, Fedora, SUSE

.deb 64 bit ARM 64 bit ARM 64 bit ARM  
.rpm 64 bit ARM 64 bit ARM 64 bit ARM  
tar.gz 64 bit ARM 64 bit ARM 64 bit ARM

Snap Store

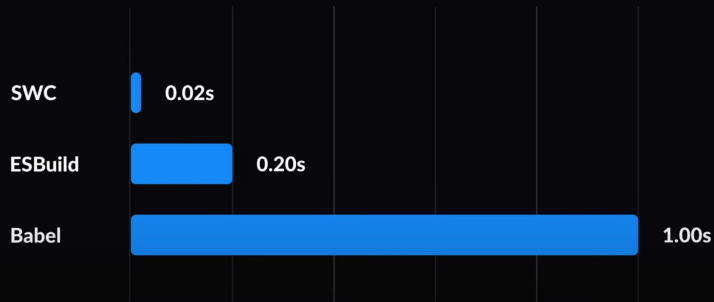


↓ Mac  
macOS 10.11+

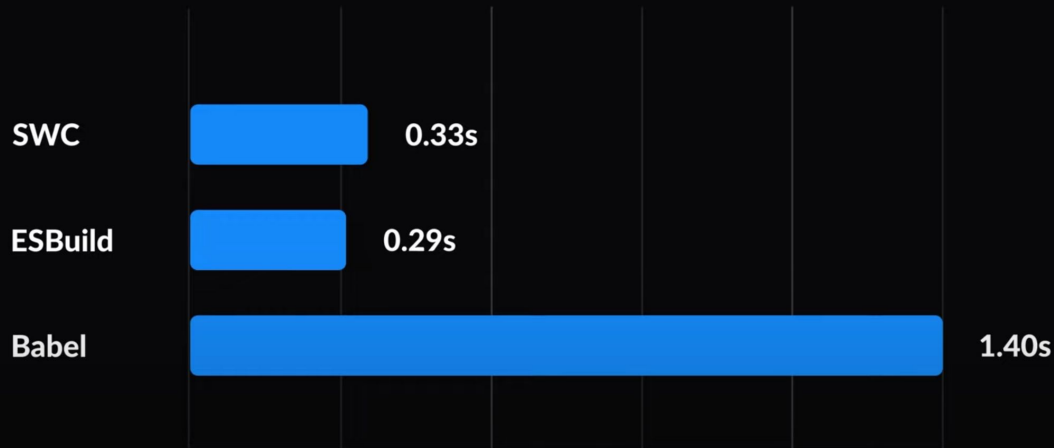
zip Universal Intel Chip Apple Silicon

cf. SWC(Rust) vs ESBuild(Go) vs Babel

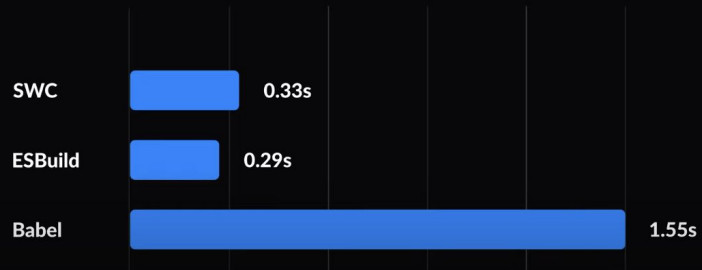
Bootstrap JavaScript



Material Design Components



Three.js



cf. Rollup

- rollup4: SWC parser
- rolldown

# Vite vs Traditional Module Bundlers



Bundles all your JS modules, CSS and other assets

Does this every time you make a change, which can get really slow as your app grows



Vite takes advantage of native ES Modules. Serves directly to the browser

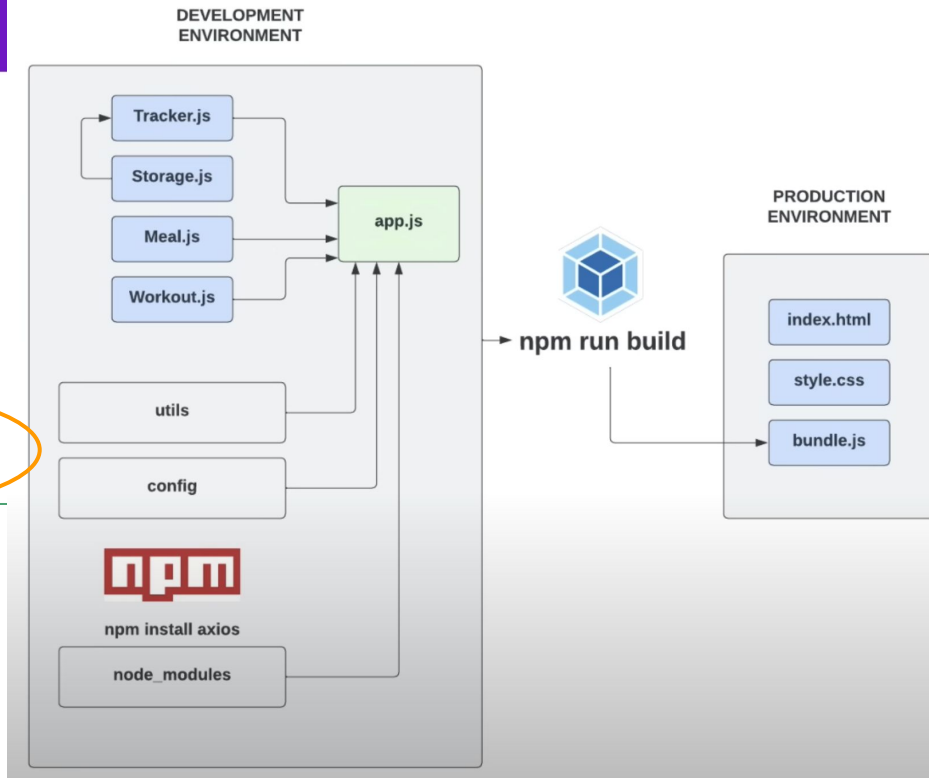
Uses Rollup to bundle files for production (npm run build)

## Replacement For Create React App?

CRA is great for beginners. It's easy to use and has some great features

It can get quite laggy as your application grows as it uses Webpack under the hood

Vite has a React plugin, which makes it a great replacement for a faster experience



## vite로 시작 (esbuild + rollup)

**vite**      **>= node v18+ or 20+**

\$> **yarn create vite** <project-name>

# yarn create vite rbvite

# npm init vite@latest rbvite # in rbvite

\$> cd rbvite      # react-basic-vite

\$> yarn      # 또는 **npm i**

\$> yarn build      # npm run build

\$> yarn dev

# npm

npm create vite@latest sample

# yarn

**yarn create vite sample**

# pnpm

pnpm create vite sample

# bun

bunx **create-vite** sample

```
[15:41:17:jade ~/workspace/fullstack4/react]$ yarn create vite rbvite
yarn create v1.22.19
[1/4] 🔍 Resolving packages...
[2/4] 📦 Fetching packages...
[3/4] 🔗 Linking dependencies...
[4/4] 🔨 Building fresh packages...

success Installed "create-vite@4.4.1" with binaries:
  - create-vite
  - cva

[#####] 46/46? Select a framework:
> Vanilla
  Vue
  React
  Preact
  Lit
  Svelte
  Solid
  Qwik
  Others
```

```
? Select a variant: > - Use arrow-keys. Return to submit.
> TypeScript
  TypeScript + SWC
  JavaScript
  JavaScript + SWC
```

**yarn install v1.22.19**

info No lockfile found.

[1/4] 🔍 Resolving packages...

[2/4] 📦 Fetching packages...

[3/4] 🔗 Linking dependencies...

[4/4] 🔨 Building fresh packages...

success Saved lockfile.

🌟 Done in 10.97s.

VITE v5.3.1 ready in 100 ms

→ Local: http://localhost:5173/  
→ Network: use --host to expose  
→ press h + enter to show help

h

### Shortcuts

press r + enter to restart the server  
press u + enter to show server url  
press o + enter to open in browser  
press c + enter to clear console  
press q + enter to quit

q

✨ Done in 29.32s.

```
$ tree --gitignore
```

```
├── README.md
├── index.html
├── package.json
├── public
│   └── vite.svg
├── src
│   ├── App.css
│   ├── App.jsx
│   ├── assets
│   │   └── react.svg
│   ├── index.css
│   └── main.jsx
├── vite.config.js
└── yarn.lock
```



# React Folder 구조 예시 비교

## react org (webpack, CRA)

- build
- public
  - index.html
- src
  - App.js → App.jsx
  - index.js → index.jsx
- dist
- vite.config.js
- index.html

## vite react (JS)

- dist
- public
  - vite.svg
- src
  - assets
  - App.jsx
  - index.css
  - main.jsx
- index.html
- vite.config.js

## vite react (TS)

- dist
- public
  - vite.svg
- src
  - assets
  - App.tsx
  - index.css
  - main.tsx
- vite-env.d.ts
- index.html
- vite.config.ts
- tsconfig.json
- tsconfig.node.json
- .eslintrc.cjs
- .gitignore
- package.json

index.html → main.jsx → App.jsx

# VSCode Extensions code formatter



## Prettier - Code formatter

🕒 945ms

Code formatter using prettier

Prettier



## Prettier ESLint

A Visual Studio Extensio

Rebecca Vest



## ESLint v2.2.6

🔗 Microsoft | 📦 23,052,518 | ★★★★★

Integrates ESLint JavaScript into VS Code.

Disable ▾

Uninstall ▾



This extension is enabled globally.

### Editor: Format On Save

- ☒ Format a file on save. A formatter must be available, the file must not be saved after delay, and the editor must not be shutting down.

single quote

default formatter

User Workspace

Text Edit... (1) ⚙️

✖ Extensions... (11)

✖ CSS La... (3)

CSS (1)

LESS (1)

SCSS ... (1)

Editor: **Default Formatter** (Also mo...  
Defines a default formatter which...  
the identifier of an extension cor...

**Prettier ESLint**

User Workspace

Text Editor (1)

Workbench (2)

✖ Extensions (6)

HTML (1)

**Prettier (2)**

Turbo Console Lo... (1)

TypeScript (2)

**Prettier: Jsx Single Quote**

☒ Use single quotes instead of double quotes in JSX

**Prettier: Single Quote**

☒ If true, will use single instead of doub

# VSCode 갱신: Ctrl + Shift + P

>dev reloa

Developer: **Reload Window**

# create-react-app 프로젝트

## ~~npx~~

```
$> npx create-react-app reactbasic
# npx create-react-app . # in directory
# npx create-react-app <app> --template
typescript
🌟 Done in 8.90s.
⇒ @testing-library 함께 설치됨!
```

```
$> yarn build # npm run build
$> yarn start # npm run start
```

Compiled successfully!

You can now view **reactbasic** in the browser.

Local: http://localhost:3000  
On Your Network: http://192.168.35.200:3000

Note that the development build is not optimized.  
To create a production build, use `npm run build`.

webpack compiled **successfully**

## vite

```
$> cd rbvite
$> yarn prettier
$> yarn build # npm run build
$> yarn dev
```

🌟 Done in 2.52s

VITE v3.1.8 ready in 498 ms  
→ Local: http://127.0.0.1:5173/  
→ Network: use --host to expose

VITE v5.3.1 ready in 98 ms

→ Local: http://localhost:5173/  
→ Network: use --host to expose  
→ press h + enter to show help

# JSX JavaScript as XML

<https://babeljs.io/>

```
<div className="shopping-list">
  <h1>Shopping List for {props.name}</h1>
  <ul>
    <li>Instagram</li>
    <li>WhatsApp</li>
    <li>Oculus</li>
  </ul>
</div>;
```

babel  
or  
rollup

```
/*#__PURE__*/React.createElement("div", {
  className: "shopping-list"
}), /*#__PURE__*/React.createElement("h1", null, "Shopping List for ",
props.name), /*#__PURE__*/React.createElement("ul", null,
/*#__PURE__*/React.createElement("li", null, "Instagram"),
/*#__PURE__*/React.createElement("li", null, "WhatsApp"),
/*#__PURE__*/React.createElement("li", null, "Oculus"))));
```

## JSX Rules

<https://reactjs.org/docs/introducing-jsx.html>

- Tree 자료구조 ⇒ Root Element ⇒ <React.Fragment /> (<>...</>)
- {} 사용, JS 구문 표현 (Type 평가 방식, 표현식 가능!)  
`const element = <h1>Hello, {name}</h1>;`
- if 대신 3항 / 논리(&&, ||) 연산자 사용 (단, 논리에 NaN, 0은 표시됨!) `{id && name}`  
`<h1>Hello, {name} {age && (isMale ? '군' : '양')}</h1>;`
- CSS class는 className으로, 그 외 style 키값은 CamelCase로! (warning)
- 태그를 열었으면, 꼭 닫자! `<br />`
- 주석은 `{/* ~ */}`
- HTML tag는 소문자, 그 외 대문자로 시작 또는 CamelCase
- Injection Attacks에 안전함!

# React Component 리액트의

## 개념 Component

- 속성 (Props)을 받고, 상태 (state)/Method를 가진다.
- 반복되는 UI 단위 (JS Code 또는 HTML-JSX)
- 즉, 재사용성과 가독성을 위한 도구
- 가능한 독립적 (순수함수)으로 실행되도록 작성하고, 데이터 영역과 UI를 분리
- `MyComponent.defaultProps = {...}`

## Class Component

- extends `React.Component` 또는 `createClass`
- constructor, `this.state`
- LifeCycle 관리가 쉬움
- soft-deprecated



A LONG TIME AGO...

react < 16.8

## Functional Component

- from React 16.8 ~
- 직관적이고 가벼우며, 번들 크기 더 작음  
(`:` 기본 변수-상태-와 생명주기가 없다)
- render 필요없고 `JSX return;`

```
import React from 'react';
import './App.css';

class App extends React.Component {
  render() {
    return <div className='App'>Class
Component</div>;
  }
}

export default App;
```

```
// src/App.js - default
import './App.css';

function App() {
  return <div className='App'>Function
Component</div>;
}

export default App;
```

# React Component Props

## Props

- props, children (React.Node)
- 속성(변수, 함수등) 전달
- `MyComponent.defaultProps = {...}`

// cf. props 정의 (**without TS**)

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

```
Hello.propTypes = {  
  name: PropTypes.string.isRequired,  
  age: PropTypes.number,  
  isMale: PropTypes.bool,  
  plusCount: PropTypes.func,  
  // children: PropTypes.children,  
  children: PropTypes.func,  
}
```

```
type propTypes = {  
  name: string,  
}
```

Cf. <https://flow.org/>

// App.js (클래스형 컴포넌트)

```
const Hello = ({ name }) => {  
  return <h1>Hello, {name}</h1>;  
};  
  
class App extends React.Component {  
  render() {  
    return <Hello name='Kim'>World</Hello>;  
  }  
}
```

// 별도 파일로 작성한다면... src/components/Hello.tsx (함수형 컴포넌트)  
import { PropsWithChildren } from 'react';

```
type Props = {  
  name: string;  
  age: number;  
};
```

```
const Hello = ({ name, age, children }: PropsWithChildren<Props>) => {  
  return (  
    <>  
      <h1>Hello, {name} {age}</h1>  
      {children}  
    </>  
  );  
};  
  
export default Hello; // => <Hello name='Hong' age={32} />
```

# Component Props / States / Variables

## Variables / Functions

- 변경된다고 해서 **re-rendering** 영향 없음!
- 값은 바뀌지만 화면(View)에는 아무런 영향이 없음!

## Props

- **read-only in receiver**
- 렌더링될 때 한번만 설정된다.  
즉, 렌더링 후에 변경되지 않는다!

## States

- 상태변경은 무조건 **setState**를 사용하자!
- 부모의 상태가 변경되면,  
그 상태를 참조하는 모든 자식 컴포넌트 **re-render**!  
→ 상태를 통합하면 **re-rendering**은 더 많이 일어난다!  
(상태를 성격-연관성-에 맞게 분리하자): 좁게 사용!  
단, **Actual(Real Active) DOM**은 해당 상태를 참조하는 부분만 다시 그린다(**paint & display**)

```
const plusAge = () => {  
  props.age += 1;  
  console.log('age=', props.age);  
};
```

✖ ▶ Uncaught TypeError: Cannot assign to read only property 'age' of object '#<Object>' [Hello](#)

({age}) => age++; // 가능 (다른 메모리)

## Component 정리

- 함수형 컴포넌트 == 순수 함수형 컴포넌트
- **Container Component**를 적절히 사용하자
  - 데이터를 가져오는 것과 그리는 것은 분리하자!
  - 상태 공유 단위로 분리하자 (cf. **useContext**)
  - 단순 담는 용도의 아주 깊은 **container** 금지  
즉, **container(context)**를 남발하지 말자!
- 최대한 독립적으로 작성한다!
  - **coupling**을 최소화하여 단독으로 사용 가능하도록!

# Component LifeCycle

## class component & Props

1. constructor (this.state, this.props)

2. componentWillMount()

p. componentWillMountReceiveProps  
(getDerivedStateFromProps)

p. shouldComponentUpdate(){...}

p. ComponentWillUpdate()

3. render (VirtualDOM.render): afterRenderTree

4. componentDidMount  
(getSnapshotBeforeUpdate)

5. componetDidUpdate

6. componentWillUnmount: clean-up code

\*. componentDidCatch

## functional component (hooks)

1/2. declarative area (variables / functions), useState 초기화

3. return JSX ⇒ render()

⇒ **useLayoutEffect** : 설계도 완성단계  
paint 호출 전에 실행(화면 깜빡임 없음!)

4. paint와 동시 **useEffect** : 집 지은 단계  
그리면서 re-render므로 상태변경 시 화면 깜빡임

5. return of useEffect, useLayoutEffect

Cf. index.jsx에서 **React.StrictMode**로  
App이 감싸져 있으면 **LifeCycle**이 두 번씩  
호출됨!  
(개발 환경에서만!) cf. useCallback(..., [])

```
<React.StrictMode>
```

```
<App />
```

```
</React.StrictMode>
```



# Sample Code 소스 구조(최초)

```
// src/main.jsx
```

```
// src/App.jsx
```

```
const SampleSession = {
  loginUser: { id: 1, name: 'Hong' },
  cart: [
    { id: 100, name: '라면', price: 3000 },
    { id: 101, name: '컵라면', price: 2000 },
    { id: 200, name: '과', price: 5000 },
  ],
};
```

```
function App() {
  console.info('@@@App');
```

```
  const [session, setSession] = useState(SampleSession);
```

```
  return (
```

```
    <My session={session}
      login={login}
      logout={logout} />
```

```
    <Hello name='홍길동' age={30}
      plusCount={plusCount}>
```

```
      <h3>반갑습니다~</h3>
```

```
    </Hello>
```

```
  );
```

```
}
```

```
export default App;
```

```
// src/components/Hello.jsx
```

```
export const Hello = (props) => { // props: {name, age, plusCount}
  console.log('@@@ Hello');
  const [isActive, setActive] = useState(false);
  ...
}
```

```
// src/components/My.jsx
```

```
// props ⇔ {session, login, logout}
export const My = ({session, logout}) => {
  console.log('@@@My>>>', session);
  return (
    <>
      {session.loginUser ? (
        <Profile session={session}
          logout={logout} />
      ) : (
        <Login login={login} />
      )}
    <ul>
      {session?.cart.map((item, i) => (
        ...
      )}
    </ul>
  )}
  </>
}
```

```
// src/components/Profile.jsx
```

```
// props: {session, logout}
const Profile = ({ session, logout }) => {
  console.log('@@@Profile>>', session);
  return (
    <>
      <div>User ID: {session.loginUser?.name}</div>
      <button onClick={logout}>Logout</button>
    </>
  );
};
```

```
// src/components/Login.jsx
```

```
const Login = ({login}) => (
  <><div>User ID(숫자): <input type='number' /></div>
  <div>User Name: <input type='text' /></div>
  <button onClick={login}>Login</button></>
);
```

# React Hooks

---

@시코 - 시니어코딩

# React Hooks - react의 상태 및 life-cycle 관리

## 상태관리 Hooks

- useState
- useContext (createContext)
- useReducer

## 시점 Hooks

- useLayoutEffect (paint 전)
- useEffect (paint 후)

## 메모화 Hooks

- useMemo
- useCallback
- memo

## 그 외 Hooks

- useRef, useImperativeHandle
- useDebugValue, useId, useTra...

## Hooks의 3 원칙

### 1. 컴포넌트의 영역 안에서만 작동한다!

- 컴포넌트 / 커스텀훅 내부에서만 호출해야 한다!

### 2. 기능을 여러 훅으로 나누면 좋다!

- 나누어 있어도 컴포넌트에서 한번에 **순차** 호출
  - 함수형 컴포넌트 ⇒ 함수 ⇒ 기능 단위 분리 ⇒ 가독성 ⇒ 테스트 및 유지보수에 유리

### 3. 컴포넌트의 최상위 레이어(스코프)에서만 호출해야 한다!

- <https://reactjs.org/docs/hooks-rules.html>  
- 블록 내부에서는 호출할 수 없다!

# useState 상태 Hook

## state

- 순수 함수  $\Rightarrow$  컴포넌트 내에서 변경될 수 있는 값
- 해당 상태를 소유하지 않은 컴포넌트에서는 **props**
- **const** [변수, 세터] = **useState**(초깃값);
- 초깃값은 동기(**sync**) 방식이면 오래걸려도 가능.

**Promise**의 경우 **async/await**로 처리해도 **Promise**가 세팅

- 변수는 **read-only**므로, 변경은 반드시 세터를 이용한다!
- 변수가 객체(참조)형이면 **spread** 연산자로 사본을 만들어
- **Setter**에서는 값 수신
  - $\rightarrow$  **Dispatcher** 값 세팅  $\rightarrow$  **re-rendering**을 **trigger** (**DOM** 갱신)
  - $\rightarrow$  **painting + display** (**re-render**는 **16ms throttle async for batch**)
  - $\rightarrow$  컴포넌트 **state** 값 변경 (주의 - **DOM** 갱신 보다 늦음!)

- **react-dom's flushSync**는 **동기**로 **DOM** 갱신 (**화면 그려지고 난 후에 처리하고 싶을 때**)

**flushSync(() => setCount(c => c + 1));** // **DOM** 즉시 갱신 (**화면 후처리 가능**), **No-batch!!**

- **React 18**부터는 **Promise/Timeout/EventHandler** 내에서

여러 상태를 변경해도 **batch-render** - 한번만 **re-rendering** - 됨! **react-doc** 참고!

- 부모로 상태를 전달하기 위해서는 부모에서 자식으로 접근자 함수를 전달 (**ex. plusCount, logout**)

```
// src/App.jsx
const SampleSession = {
  loginUser: { id: 1, name: 'Hong' },
  cart: [
    { id: 100, name: '라면', price: 3000 },
    { id: 101, name: '컵라면', price: 2000 },
    { id: 200, name: '파', price: 5000 },
  ],
};
...
const [session, setSession] = useState(SampleSession);

const logout = () => {
  // session.loginUser = null; // Bad!!
  setSession({ ...session, loginUser: null });
};
```

```
const plusCount = () => {
  console.log('111>>', count); // 0
  // setCount((count) => {
  //   count += 1;
  //   console.log('222>>', count); // 1
  // });
  // return count;
  // });
  flushSync(() => setCount((c) => c + 1));
  console.log('333>>', count); // 0
};
```

# Try This - useState

앞 장 session의 cart에 담긴 물건을 삭제하는 함수를 작성하시오.

```
// src/App.jsx
const SampleSession = {
  loginUser: { id: 1, name: 'Hong' },
  cart: [
    { id: 100, name: '라면', price: 3000 },
    { id: 101, name: '컵라면', price: 2000 },
    { id: 200, name: '파', price: 5000 },
  ],
};
...
const [session, setSession] = useState(SampleSession);

const logout = () => {
  setSession({ ...session, loginUser: null });
};

const removeCartItem = (itemId) => {
  // 이곳에 작성하세요~
}
...
<My session={session} logout={logout}
  removeCartItem={removeCartItem} />
```

```
// src/components/My.jsx
import Profile from './Profile';
import Login from './Login';

export const My = ({ session = {}, logout, removeCartItem }) => {
  console.log('@@My>>>', session);
  return (
    <>
      {session.loginUser ? (
        <Profile session={session} logout={logout} />
      ) : (
        <Login />
      )}
      <ul>
        {session?.cart.map((item) => (
          <li key={item.id}>
            {item.name} ({item.price})
            <button onClick={() => removeCartItem(item.id)}>DEL</button>
          </li>
        ))}
      </ul>
    </>
  );
};
```

이 코드는 React 18.0.0 버전에서 실행됩니다. React 18.0.0 버전에서는 useState의 초기값을 null로 설정할 수 있습니다. React 18.0.0 버전에서는 useState의 초기값을 null로 설정할 수 있습니다.

# useContext 데이터(상태) 직항!

## React ContextAPI (a.k.a React context)

- Provider로 부터 Consumer까지 **direct**로 value(상태) 공유! (ex. App → Profile)
- Provider의 Children들은 해당 context의 상태를 바로 접근 가능!
- Context.Provider ~ Context.Consumer

```
// context pattern 1 (render-prop)
import { createContext } from 'react';
const MyContext = createContext({x: 1});
export MyContext;
```

```
// context pattern 2 (provider-useContext)
export const MyContext = createContext();
render(
  <MyContext.Provider value={{xObj}}>
    <App />
  </MyContext.Provider>
);
```

```
// context consumer
import { MyContext } from '../myContext';
...
<MyContext.Consumer>
  { value => <h1>x: {value.x}</h1> }
</MyContext.Consumer>
```

```
// context consumer
import { useContext } from 'react';
import { MyContext } from '../MyContext';
const {xObj} = useContext(MyContext);
```

# useContext (Cont'd)

## 상태를 공유하는 context (useState 보유)

- 상태가 변경되면, **Provider**는 물론, 모든 자식 **Component(Consumer)**는 모두 다시 렌더링된다!
- ⇒ setState를 바로 공유하지 말고, 상태를 변경하는 함수를 전달하자! (ex. **setCount**가 아닌, **plusCount** 노출)

```
// context pattern 3 (state-provider) BAD!
const CountContext = createContext();
export function CountProvider({children}) {
  const [count, setCount] = useState(0);
  return (
    <CountContext.Provider
      value={{count, setCount}}>
      {children}
    </CountContext.Provider>
  );
};
```



```
// src/hooks/counter-context.jsx GOOD!
export const CountProvider = ({children}) => {
  const [count, setCount] = useState(0);
  const plusCount = () => setCount(count + 1);
  return (
    <CountContext.Provider
      value={{count, plusCount}}>
    ...
  );
};
export const useCount = () => useContext(CountContext);
```

```
// src/index.jsx - Providing
import { CountProvider } from '../hooks/counter-context';
root.render(
  <React.StrictMode>
    <CountProvider> <App /> </CountProvider>
  </React.StrictMode>
);
```

```
// Consumer
import { useCount } from '../hooks/counter-co...';
const { plusCount } = useCount();
```

# Try This - useContext

App → My → Profile 순으로 전달했던 session Props 객체를 contextAPI를 이용하여 바로 사용할 수 있도록 session-context를 하시오.

(value: {session, logout, removeCart})

```
// src/App.jsx
<My session={session}
  logout={logout}
  removeCartItem={removeCartItem} />
```

```
// src/components/My.jsx
export const My = ({session, logout}) =>
...
<Profile session={session}
  logout={logout} />
```

```
// src/components/Profile.jsx
export const Profile = ({session, logout}) =>
...
const {session} = useContext(SessionContext);
```

```
// src/App.jsx
<SessionProvider> <My /> </SessionProvider>
```

```
// src/hooks/session-context.jsx
export const SessionProvider =
({children}) =>
...
export const useSession = ...
...
```

```
// src/components/My.jsx
export const My = () =>
...
```

```
// src/components/Profile.jsx
export const Profile = () =>
...
const {session} = useContext(SessionContext);
```





# useRef 무엇이든 참조해 주겠소!

**useRef** 직접 DOM에 접근할 때는 날 불러주시오!

```
const xRef = useRef();
```

→ `<span ref={xRef}>...`, `<input type='text' ref={xRef} />`, `<MyComponent ref={xRef} />`

→ `xRef.current.[value, focus, ...]` ← 해당 DOM의 속성

- `onChange`와 같은 무거운 **Event Handler**를 사용하지 않아도 된다! (DOM을 외부에 공유하는 것도 가능)

```
// onChange와 useState 사용
const Login = () => {
  const {login} = useSession();
  const [userId, setUserId] = useState(0);
  const [userName, setUserName] = useState('시코');

  const submit = (evt) => {
    evt.preventDefault();
    login(userId, userName);
  };

  return ( ...
    <form onSubmit={submit}>
      User ID: <input type='number' value={userId}
        onChange={(evt) => setUserId(evt.target.value)} />
```

// useRef 사용

```
const Login = () => {
  const {login} = useSession();
  const userIdRef = useRef();
  const userNameRef = useRef();

  const submit = (evt) => {
    evt.preventDefault();
    login(userIdRef.current.value,
      userNameRef.current.value);
  };

  return ( ...
    <form onSubmit={submit}>
      User ID: <input type='number' ref={userIdRef} />
```

# useEffect rendering 후에 뭘 할까

## useEffect rendering 후 mount 시점에 실행

- componentDidMount 시점에 비동기로 실행 (ex. alert, focus, sound 등 After DOM 처리)
- 즉, 함수형 컴포넌트에서 return JSX(render) 다음에 실행된다. (ex. focus주기, 미디어 재생, 지도에 추가 등)
- 정리(cleanup)코드는 componentWillUnmount 시점에 실행된다. (useEffect's return function)
- 의존 관계 배열 (Dependency Array)가 비워져 있다면 1회만 실행된다(re-render시 skip except StrictMode)

```
// Login component에 로그인하라는 alert 띄우기
const Login = () => {
  ...
  useEffect(() => {
    alert('로그인해주세요!');

    return () => { // 정리(cleanup) 함수
      alert('로그인 되셨어요!');
    };
  }, []); // 단 1번 마운트, 단 1번의 효과!
} // → 의존관계배열 생략시 re-render마다 효과!
```

```
// (주의) 의존 관계 지정 시 고려 사항
const [, rerender] = useState();

let primitive = 123;
useEffect(() => {
  console.log('effect primitive!!!');
}, [primitive]);

const array = [1, 2, 3];
useEffect(() => {
  console.log('effect Array!!!');
}, [array]);

<input type='text' onChange={rerender} />
```

# useEffect (Cont'd) call twice

## useEffect call twice

- 개발환경 + **StrictMode**에서 의존 관계 배열이 비워([])져있을 때 2회 호출 (Mount - UnMount - Mount)
  - 2회 마운트/호출 가능한 부분의 안전한 코딩을 위해! (**setState/useEffect/useReduce의 callback을 2회 실행!**)
  - **StrictMode**를 꺼두면 잘못 된 코드를 작성할 가능성 높음! (**cleanup** 유실, 중복 데이터 처리 등)
  - 즉, 제대로 **cleanup** 해주고 **2회 호출되어도 안전한 코딩**을 하자! (ex. timer, eventListener, data add/remove)
  - **cleanup** 해줄게 있는지를 항상 염두하자! (socket.on('evt') ⇒⇒ return () => socket.off('evt'))

```
// Bad!
const [badSec, setBadSec] = useState(0);

useEffect(() => {
  setInterval(() => {
    setBadSec((pre) => pre + 1);
  }, 1000);
}, []);
...
<span style={{ float: 'left', color: 'red' }}>
  {badSec} sec
</span>
```



```
// Good!
const [goodSec, setGoodSec] = useState(0);

useEffect(() => {
  // ?
}, []);
...

const intl = setInterval(() => {
  setGoodSec((pre) => pre + 1);
}, 1000);

return () => clearInterval(intl);
```

# useEffect (Cont'd) with Promise

## useEffect with Promise

- Promise는 실행될 때 callback 함수를 background로 제어를 넘긴다.
- re-render 등으로 UnMount시에 Promise는 background에 존재하므로 cleanup되지 않는다.
- 정리(cleanup)코드에서 Promise를 중지(abort)해줘야 2번 render되지 않는다. (단, request server는 2회)

```
controller = new AbortController();  
const { signal } = controller;  
fetch(url, { signal });  
// non-signal: 3회 render (TQ에 2번 들어감)  
// with-signal: 2회 render (처음 cb가 abort → TQ 1번)  
controller.abort() at cleanup function!!
```

```
// src/hooks/session-context.jsx Bad!  
useEffect(() => {  
  
  const url = '/data/sample.json';  
  fetch(url)  
    .then((res) => res.json())  
    .then((data) => {  
      setSession(data);  
    })  
    .catch(console.error);  
  
}, []);
```



```
// src/hooks/session-context.jsx Good!  
useEffect(() => {  
  const controller = new AbortController();  
  const { signal } = controller;  
  fetch(url, { signal })  
    .then((res) => res.json())  
    .then((data) => {  
      setSession(data);  
    })  
    .catch(console.error);  
  
  return () => controller.abort();  
}, []);
```

⇒ SWR, React-Query 고려

# useCallback `함수`를 기억해 둘게!

**useCallback**    함수 cache    (**function memoization**)

```
const 메모화된_함수 = useCallback(() => {  
  함수_코드;  
}, [의존관계배열]);
```

- 메모화된 함수는 자식 컴포넌트에서 **re-render** 시 **재 호출** 방지.

즉, 한번만 실행되어야 하는 함수를 전달할 때 사용!

- 단, 함수를 그냥 전달하는 비용보다 비싸기 때문에 비용이 많이드는 또는 자주 불리면 안되는 함수에만 적용하자!

// 부모 컴포넌트

```
const [rrr, render] = useState();
```

```
const fn = useCallback(() => console.log('useCallbacked.fn'), []);  
useEffect(() => console.log('*****'), [fn]); // 여기서 한번!
```

```
<Child fn={fn} />  
<hr />  
<div>  
  rrr: {rrr}  
  <input type='text'  
    onChange={e => render(e.target.value)} />  
</div>
```

// 자식 컴포넌트

```
const Child = ({ fn }) => {  
  console.log('Child rendering!');  
  
  useEffect(() => {  
    console.log('call fn()!!!!!!');  
    fn(); // fn 변경 없으면 1회만 실행!  
  }, [fn]);  
  
  return <button>Compo</button>;  
};
```

# useLayoutEffect rendering 후(그리기 전)에 뭘

**useLayoutEffect** rendering 직후 mount(painting) 전 실행!

- 순서: rendering ⇒ **useLayoutEffect** ⇒ [painting & display 동시(비동기) **useEffect(didMount)**]  
즉, DOM을 **화면에 그리기 직전**에 호출. (**Reconciler**가 **Virtual-DOM**을 **Real-Native-DOM**에 적용 후 호출)
- DOM의 좌표와 크기 등은 모두 결정된 후 화면을 그리기 전에 완료해야 하므로 동기(**sync, block**)로 실행!  
"useLayoutEffect가 끝나기 전에 **painting** 하지마!" 즉, 너무 연산이 많은 부담스러운 작업은 금물!
- **useEffect**와 같이 **cleanup** 및 의존 관계 배열 사용은 동일함.

↓↓↓ App.jsx에서 현재 마우스 포지션(x,y)을 표시

```
const [position, setPosition] = useState({ x: 0, y: 0 });
const catchMousePosition = ({ x, y }) => {
  setPosition({ x, y });
};
```

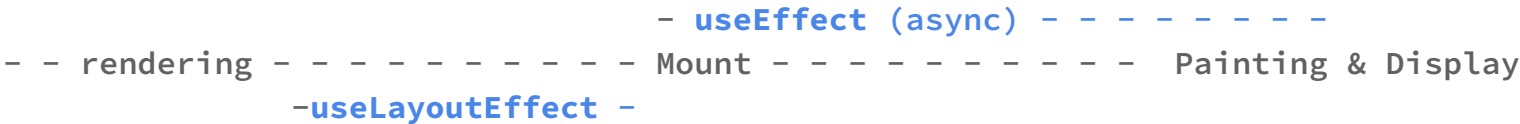
```
useLayoutEffect(() => { // ← 만약 useEffect로 하면?? 거의(컴이 빠르면 찰나의 차이) 동일
  window.addEventListener('mousemove', catchMousePosition);
  return () => window.removeEventListener('mousemove', catchMousePosition);
}, []);
```

...

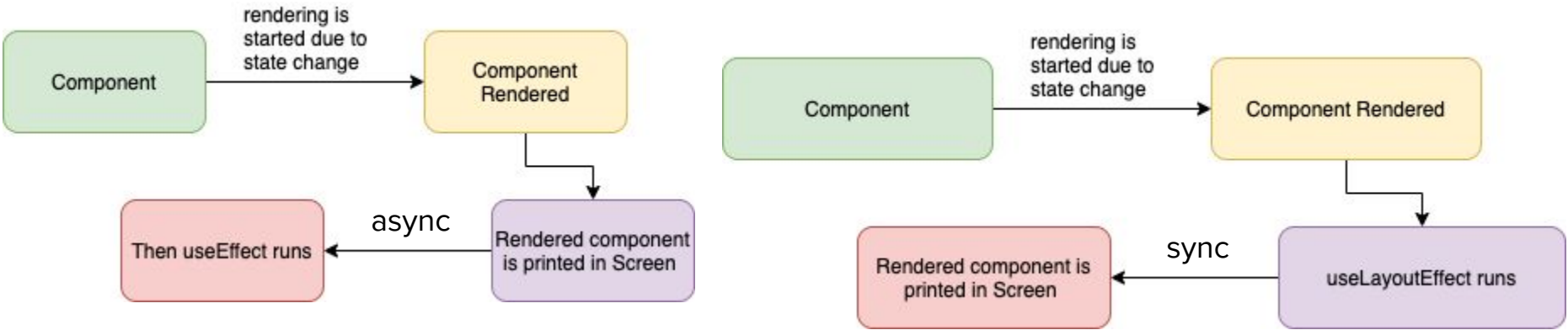
```
<small>{JSON.stringify(position)}</small>
```

# useEffect vs useEffect

**useEffect** mount 직후 (painting과 비동기로 실행!)



**useLayoutEffect** rendering 직후 mount(painting) 전 실행!



# useReducer 1. 위험한 상태 변경(setter)은 이제

## useReducer from reduce function

- **reduce**: 기존(현재) 값을 바탕으로 추가 액션 ("현재 값은 모르겠고, 암튼 이 처리를 해줘!") ← 예측 가능한 처리!

ex) `[1,2,3].reduce((currSum, a) => currSum + a, 0);`

- **reducer**: 기존 값에 무엇인 액션을 해주는 함수

ex) `setIsActive(!isActive); setCount(count => count + x);` // bad  
`toggle();` // good  
`plusCount(x);`

- 가장 큰 장점은, **setter**를 직접적으로 노출하지 않으면서 함수를 컴포넌트 외부에 둘 수 있다.

↓↓↓ `src/components/Hello.jsx`에 `toggle` 만들기

```
// src/components/Hello.jsx
import { useReducer } from 'react';
// const [isActive, setActive] = useState(false);
const [isActive, toggleActive]
= useReducer(isActive => !isActive, false);

<button onClick={toggleActive}>Toggle</button>
```

`reduce` 함수 (`session-context.jsx`) ↓↓↓

```
const totalPrice =
  session.cart.map(item => item.price)
    .reduce((tot, price) => tot + price, 0);
```

⇒ ⇒ 이것을 활용해서 **reducer**를 만들면...  
기존 **totalPrice**값에 신규 **item.price**를 더하는 것!!

```
const [totalPrice, addPrice] // 사용: addPrice(100)
= useReducer( (prePrice, newPrice) =>
  prePrice + newPrice, 0);
```



# useReducer 2. 상태 관련 함수를 한곳에..

Don't use Redux!

Mathias Oehler Dec 24, 2017 · 5min read

🏠 📄 🔍



**useReducer** 상태 변경하는 여러 함수를 하나로. **reduce & dispatcher**

- **reducer**: 같은 상태를 변경하는 함수들을 한곳에 모아 놓은 것.
- **dispatcher(action)**: **action**을 전달하는 함수(**StrictMode**에서는 2회 Call)
- 가장 큰 장점은, 컴포넌트 외부에 상태 변경하는 함수(**reducer**)를 둘 수 있다!

`const reducer = (state, action) => {...} // action: {type, payload} ← payload는 신규 데이터`  
→ `const [state, dispatcher] = useReducer(reducer, 초깃값);`

↓↓↓ **plusCount, minusCount** 합치기

```
// src/App.jsx : reducer
import { useReducer } from 'react';
const reducer = (count, action) => {
  switch (action.type) {
    case 'plus':
      return count + (action.payload ?? 1);
    case 'minus':
      return count - 1;
    default:
      return count;
  }
};
```

```
// src/hooks/count-context.jsx : useReducer
function App() {
  // const [count, setCount] = useState(0);
  const [count, dispatch] = useReducer(reducer, 0);
  const plusCount = (x = 1) => {
    // setCount((c) => c + 1);
    dispatch({ type: 'plus', payload: x });
  };
  ...
}
```

# memo `컴포넌트`를 기억해 둘게!

**memo** Component Memoization 성능 최적화의 끝판왕!

- memo(Component): 컴포넌트를 cache해서, 해당 컴포넌트의 불필요한 re-render를 막는다!  
ex) `const MemoedComponent = memo(Component);` // 이 컴포넌트를 사용하는 컴포넌트 외부에 선언!
- re-render timing: props가 변경되었을 때! (즉, props가 dependency array)  
ex) `<MemoedComponent name={name} />` // Good Cf. useMemo  
`<MemoedComponent param={{name}} />` // Bad (Reference Type is always new memory address!)  
⇒ `const MemoedComponent = memo(Component, (pre, next) => pre.param.name === next.param.name);`  
⇒ 또 다른 방법은? `const param = useMemo(xparam);` `<MemoedComponent param={param} />`
- setState(useState), dispatch(useReducer)등의 함수는 자체 memo! (React Static 영역)
- useContext를 사용하는 컴포넌트는 memo되지 않는다! → 다음 장 Try This!
- React는 rendering(reconciler)까지 무지 빠르다! 즉, memo를 굳이 사용해야 할까?!



```
// src/App.jsx
import { memo } from 'react';
const ColorTitle = ({ color }) => {
  console.log('@@@ ColorTitle!!', color);
  return <h2 style={{ color }}>MEMO</h2>;
};
const MemoedColorTitle = memo(ColorTitle);
```

```
function Color() {
  const [state, setState] = useState(0);
  ...
  <MemoedColorTitle color={state % 3 === 0 ? 'gray' : 'yellow'} />
  {state}
  <button onClick={() => setState((state) => state + 1)}>MM</button>
```

# Thank you

수고 많으셨습니다. 2탄을 기대해 주세요~