



**Seoul  
Software  
ACademy**

**with**







Formatting,  
Linting,  
Typescript 세팅!



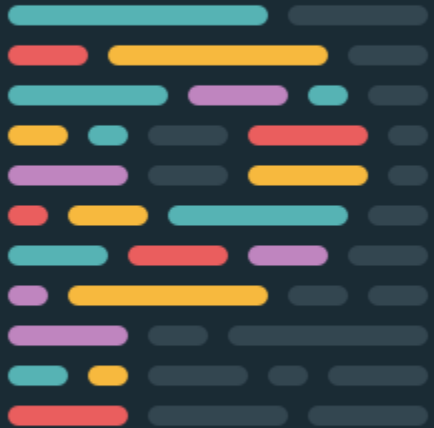
# Formatting

# Formatting?



```
README.md — prettier-config-example

1 # Prettier support for other languages
2
3 This folder has JavaScript, CSS, HTML, JSON and even this Markdown file all formatted using
  Prettier
4
5 Note that while formatting [index.js](index.js) Prettier uses 2 spaces per tab, but in the
  Markdown code blocks it uses 4 spaces.
6
7 ```js
8 const code = true; if (code) { console.log('code is on')}
9 ```
10
11 This is because we override options inside the [.prettierrc.json](.prettierrc.json) file.
12
```



# Prettier



# Linting

# Linting?









# Typescript

Type 관련 문제는  
TS가 처리했으니 안심하너굴!





자이언트콩  
명수, 위술고 70기 (10)

EBS 11

잠

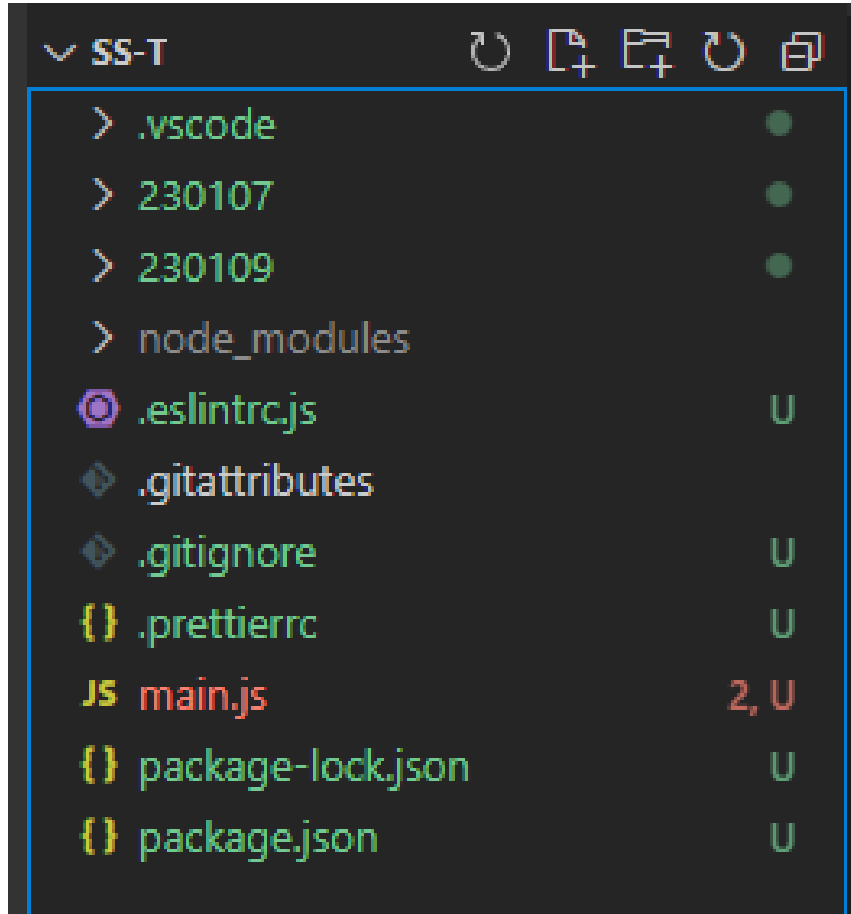
관



**확실히 세팅하고  
넘어갑시다!**



# 리액트 폴더 최상위에 세팅하기!



- .vscode 는 최상위에 두면 하위 폴더의 모든 코드를 관리 합니다!
- 기존 .vscode 코드와 .prettierrc / .eslintrc.js 파일을 리액트 폴더 최상위로 옮깁시다!



# 리액트 폴더 최상위에 세팅하기!

- 물론 기존 수업 코드에서 `.vscode` / `.prettierrc` / `.eslintrc.js` 파일이 있다면 삭제해 주세요!



# 필요 모듈 설치하기!

```
{  
  "devDependencies": {  
    "eslint-config-airbnb-base": "^15.0.0",  
    "eslint-plugin-prettier": "^4.2.1",  
    "typescript": "^4.9.3"  
  }  
}
```

- 명령어 치기 귀찮으니, package.json 파일을 만들고 위의 코드를 넣어 주세요!
- npm i 를 쳐서 모듈 인스톨!





# Es-lint 설정 변경

```
module.exports = {  
  extends: ['airbnb-base'],  
  rules: {  
    'linebreak-style': 0,  
    'no-console': 'off',  
  },  
  parserOptions: {  
    ecmaFeatures: {  
      jsx: true,  
    },  
    ecmaVersion: 'latest',  
    sourceType: 'module',  
  },  
};
```

- 기본 eslint 모듈로도 jsx 문법을 문제 없이 처리할 수 있도록 .eslintrc.js 코드를 이렇게 변경 합시다!



# .git 폴더가 안생기도록 리액트 앱 생성!

- `npx create-react-app --skip-git`

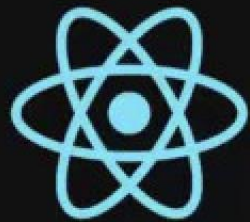
```
npx create-react-app projectname --skip-git
```

- 혹, 까먹고 그냥 생성하셨을 때 프로젝트 폴더에 .git 폴더가 생성 된다면 그냥 지워 주시면 됩니다!
- 이제, 매번 리액트 프로젝트 폴더에 세팅을 하실 필요가 없어졌습니다!



# State

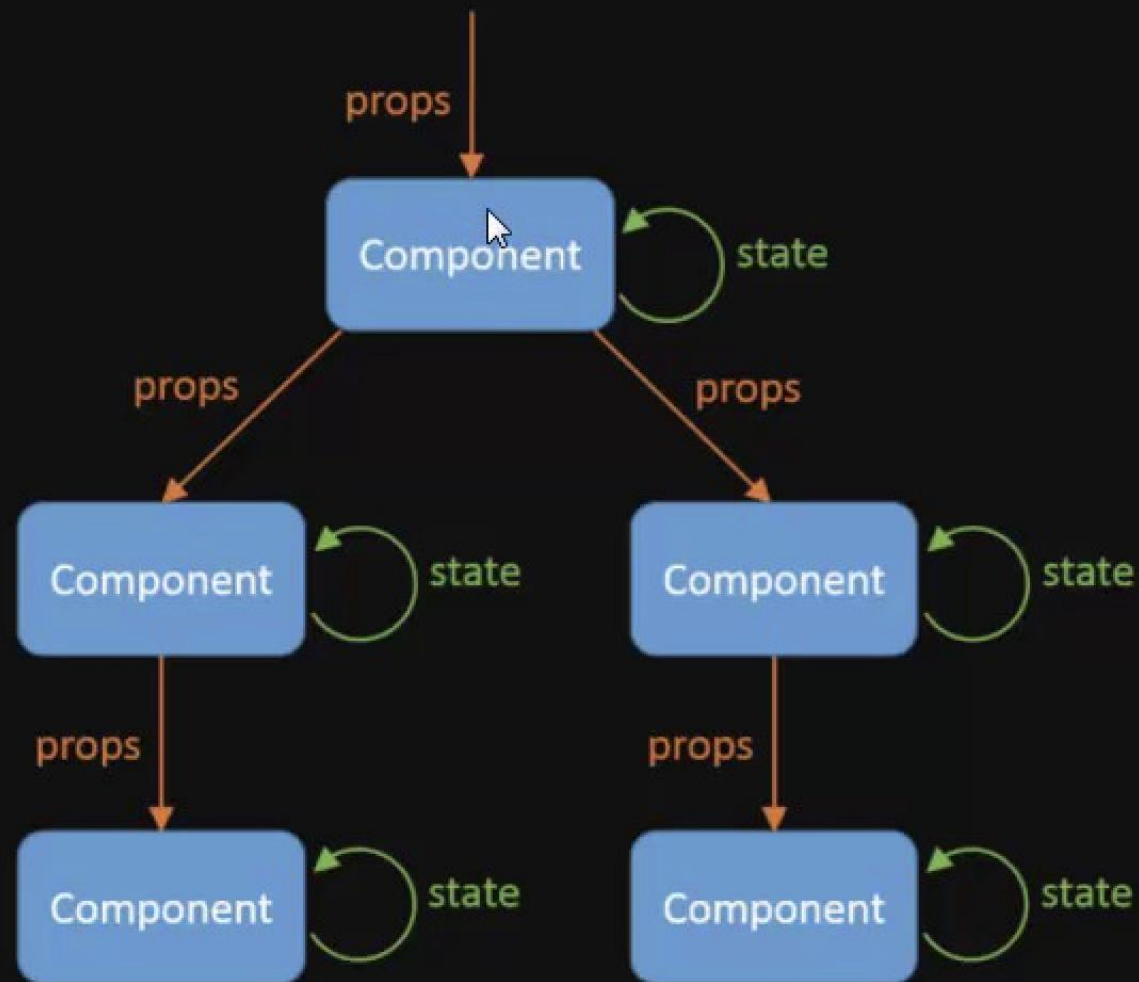




# Data Flow - React



Components can either be passed data (PROPS), or  
materialize their own state and manage it over time (STATES)





# State

# 체험하기



```
import { useState } from "react";

function App() {
  const [teacher, setTeacher] = useState("이효석");

  return (
    <div className="App">
      <button onClick={() => setTeacher("tetz")}>영어로!</button>
      <br />
      <span>{teacher}</span>
    </div>
  );
}

export default App;
```



# useState 문법!

```
import { useState } from "react";  
  
const [스테이트이름, 스테이트변경함수] = useState(초기값);
```

- 먼저 상태를 관리하는 state 를 배열의 첫번째로 정해주고, 해당 state 를 변경 할 수 있는 함수를 두번째로 지정해 주면 됩니다!
- 그리고 useState 의 () 안에는 state 의 초기 값을 넣어 줍니다!





```
const [state, setState] = useState(initialState)
```



The name of  
your state



The function you'll  
eventually use to  
change the value of this  
state



The initial value  
of your state

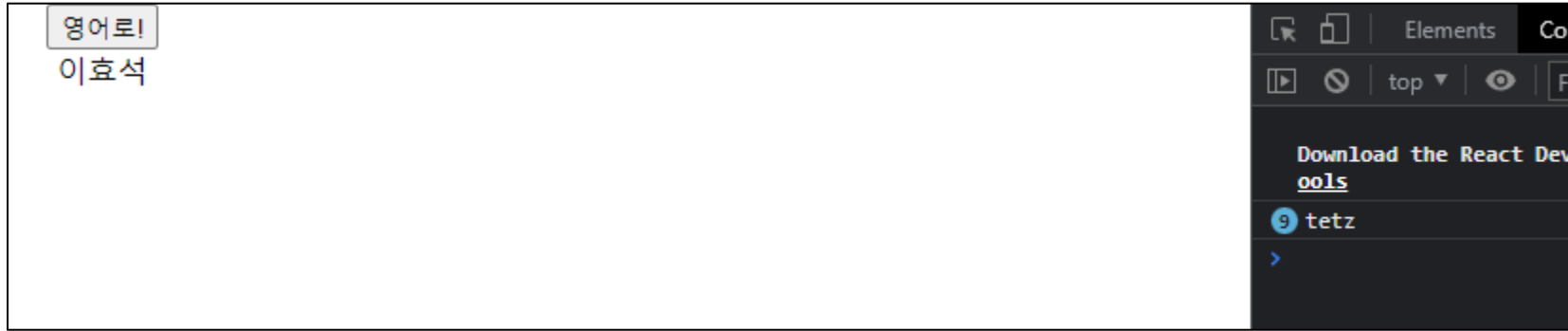


# 그럼 이런 코드도 한번 봅시다~!

```
function App() {  
  let [teacher, setTeacher] = useState("이효석");  
  
  function customSetTeacher(name) {  
    teacher = name;  
    console.log(teacher);  
  }  
  
  return (  
    <div className="App">  
      <button onClick={() => customSetTeacher("tetz")}>영어로!</button>  
      <br />  
      <span>{teacher}</span>  
    </div>  
  );  
}  
export default App;
```



# 그럼 이런 코드도 한번 봅시다~!



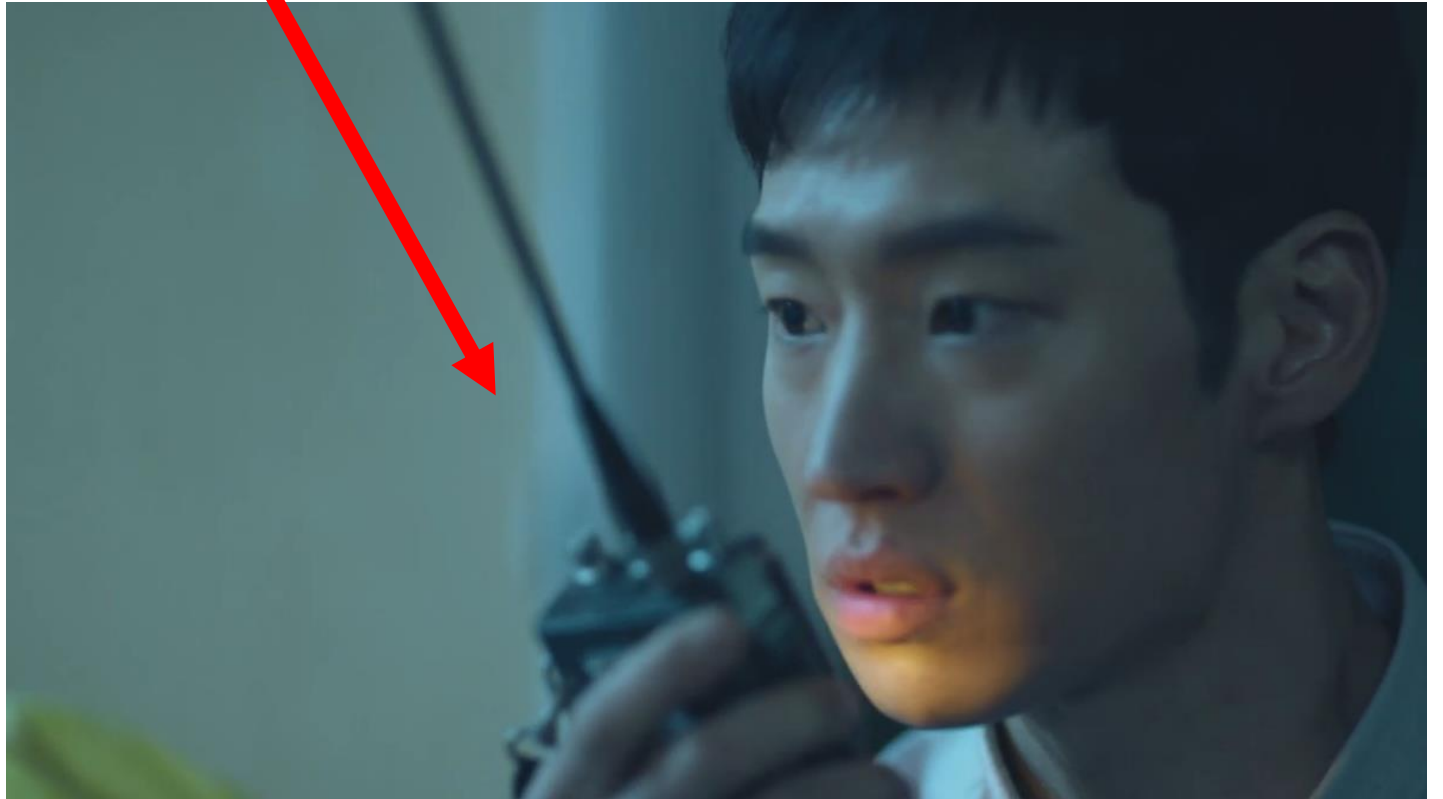
- 보시는 것처럼! state 가 변경이 되네요?
- 그런데 리렌더링은 발생하지 않습니다!



```
import { useState } from "react";
```

```
const [스태이트이름, 스테이트변경함수] = useState(초기값);
```

- 상태 변경 함수로 변경을 해야만 React 가 state 변경을 알아차리고 html 을 변경해 줍니다!





# 주의 사항!



```
function App() {  
  let [teacher, setTeacher] = useState("이효석");  
  
  return (  
    <div className="App">  
      <button onClick={setTeacher("tetz")}>영어로!</button>  
      <br />  
      <span>{teacher}</span>  
    </div>  
  );  
}  
export default App;
```

```
✖ ▶ Uncaught Error: Too many re-renders. React limits the number of renders to prevent an infinite loop.  
    at renderWithHooks (react-dom.development.js:16317:1)  
    at mountIndeterminateComponent (react-dom.development.js:20074:1)  
    at beginWork (react-dom.development.js:21587:1)  
    at HTMLUnknownElement.callCallback (react-dom.development.js:4164:1)  
    at Object.invokeGuardedCallbackDev (react-dom.development.js:4213:1)  
    at invokeGuardedCallback (react-dom.development.js:4277:1)  
    at beginWork$1 (react-dom.development.js:27451:1)  
    at performUnitOfWork (react-dom.development.js:26557:1)  
    at workLoopSync (react-dom.development.js:26466:1)  
    at renderRootSync (react-dom.development.js:26434:1)
```

```
✖ ▶ Uncaught Error: Too many re-renders. React limits the number of renders to prevent an infinite loop.
```



그럼 약간 더  
'REACT'스럽게  
만들어 볼까요?



```
function App() {
  let [condition, setCondition] = useState(true);

  return (
    <div className="App">
      <button onClick={() => setCondition(!condition)}>컨디션 변경!!</button>
      <br />
      <span style={{ fontSize: "100px" }}>{condition ? "👍" : "😞"}</span>
    </div>
  );
}
export default App;
```





# 클래스형 컴포넌트의 State



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

class ClassState extends Component {
  constructor(props) {
    super(props);
    this.state = {
      teacher: "이효석",
    };
  }
  render() {
    const { teacher } = this.state;

    return (
      <div>
        <button onClick={() => this.setState({ teacher: "tetz" })}>
          영어로!
        </button>
        <br />
        <span style={{ fontSize: "100px" }}>{teacher}</span>
      </div>
    );
  }
}

export default ClassState;
```

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

class ClassState extends Component {
  // 현재 버전
  state = {
    teacher: "이효석",
  };
  render() {
    const { teacher } = this.state;

    return (
      <div>
        <button onClick={() => this.setState({ teacher: "tetz" })}>
          영어로!
        </button>
        <br />
        <span style={{ fontSize: "100px" }}>{teacher}</span>
      </div>
    );
  }
}

export default ClassState;
```





# State 와 변수!

```
function App() {
  let [state, setState] = useState(0);
  let variable = 0;
  const setVariable = () => {
    variable += 1;
    console.log(`state: ${state} / variable: ${variable}`);
  };

  return (
    <div className="App">
      {state} / {variable}
      <br />
      <button
        onClick={() => {
          setState((state) => state + 1);
          setVariable();
        }}
      >
        +1
      </button>
    </div>
  );
}
```

```
state: 1 / variable: 1
state: 2 / variable: 1
state: 3 / variable: 1
state: 4 / variable: 1
state: 5 / variable: 1
state: 6 / variable: 1
state: 7 / variable: 1
state: 8 / variable: 1
state: 9 / variable: 1
state: 10 / variable: 1
state: 11 / variable: 1
state: 12 / variable: 1
state: 13 / variable: 1
```





# State

## 사용시 주의 사항



# 객체 또는 배열을 State 로 사용!

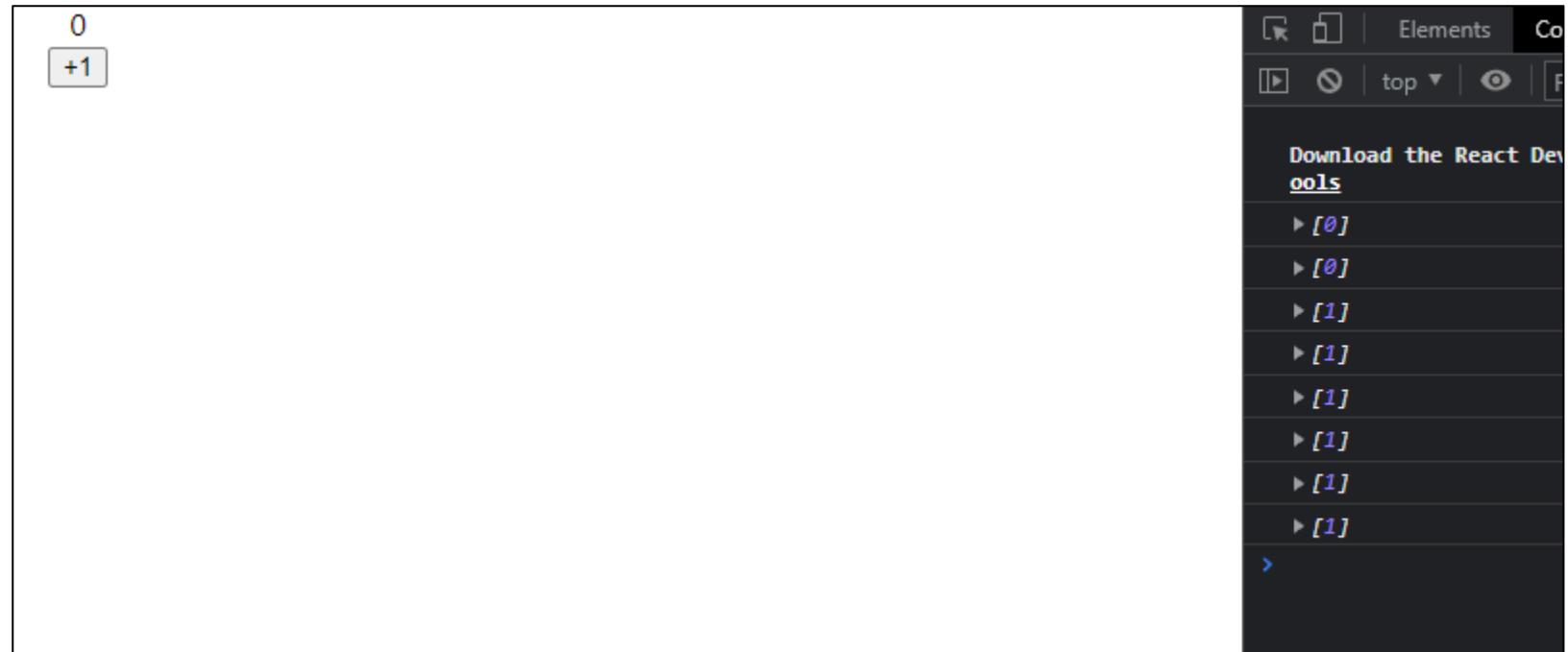
```
function App() {  
  let [state, setState] = useState([0]);  
  console.log(state);  
  return (  
    <div className="App">  
      {state}  
      <br />  
      <button  
        onClick={() => {  
          state[0] = 1;  
          setState(state);  
          console.log(state);  
        }}  
      >  
        +1  
      </button>  
    </div>  
  );  
}
```

- State 의 초기 값을 배열로 세팅해 보겠습니다!



# 객체 또는 배열을 State 로 사용!

- 아무리 눌러도 변화가 없네요!?
- 분명 값이 0 에서 1 로 변화 했는데 왜? 변경이 안되죠?
- 아시는 분???





# 원시 (Primitive) 타입

String  
Number  
Boolean  
Null  
Undefined  
BigInt  
Symbol

# 객체 (Object) 타입

원시 타입을 제외한 모든 것

Object  
Array  
...

# 원시 (Primitive) 타입

```
const location = "korea"
```

location

"korea"

# 객체 (Object) 타입

```
const location = {  
  country: "korea"  
}
```

location

#12345



#12345

{ country: "korea" }

## 원시 (Primitive) 타입

```
const locationOne = "korea"
```

```
const locationTwo = "korea"
```

```
locationOne === locationTwo
```

```
> true
```

## 객체 (Object) 타입

```
const locationOne = {  
  country: "korea"  
}
```

```
const locationTwo = {  
  country: "korea"  
}
```

```
locationOne === locationTwo
```

```
> false
```



# 객체 또는 배열을 State 로 사용!

- 그렇다면!? 새롭게 배열을 만들어서 메모리 주소를 바꿉시다!

```
function App() {  
  let [state, setState] = useState([0]);  
  
  return (  
    <div className="App">  
      {state}  
      <br />  
      <button  
        onClick={() => { setState([1]); }}  
      >  
        +1  
      </button>  
    </div>  
  );  
}
```



# 객체 또는 배열을 State 로 사용!

- 또는?

```
function App() {  
  let [state, setState] = useState([0]);  
  
  return (  
    <div className="App">  
      {state}  
      <br />  
      <button  
        onClick={() => {  
          state[0] = 1;  
          const copyArr = [...state];  
          setState(copyArr);  
        }}  
      >  
        +1  
      </button>  
    </div>  
  );  
}
```



**지금 시작합니다**



컴포넌트로

코드를 줄이자~!



# 컴포넌트 활용하기 - 코드 줄이기!

- 기존 HTML 에서는 기존의 것과 동일한 것을 만들어도 코드를 동일하게 써야만 했습니다~!
- 즉, 템플릿 기능이 없었죠!
- 리액트에서는 이러한 HTML 의 문제를 컴포넌트를 활용하여 손쉽게 해결이 가능합니다~! 느낌이 오시죠!?



```
function List() {  
  return (  
    <div>  
      <h1>오늘 해야할일</h1>  
      <hr />  
      <h2>리액트 공부하기</h2>  
      <p>state 사용법 익히기</p>  
      <hr />  
      <h2>코테 문제 풀기</h2>  
      <p>Lv 0 정복 가즈아</p>  
      <hr />  
      <div className="modal">  
        <h2>오늘 해야할 일 2개</h2>  
        <h2>오늘 완료한 일 0개</h2>  
      </div>  
    </div>  
  );  
}  
  
export default List();
```

갑자기 요 modal DIV를  
최상단에도 넣고 싶다면!?



```
export default function List() {  
  return (  
    <div>  
      <h1>오늘 해야할 일</h1>  
      <div className="modal">  
        <h2>오늘 해야할 일 2개</h2>  
        <h2>오늘 완료한 일 0개</h2>  
      </div>  
      <hr />  
      <h2>리액트 공부하기</h2>  
      <p>state 사용법 익히기</p>  
      <hr />  
      <h2>코테 문제 풀기</h2>  
      <p>Lv 0 정복 가즈아</p>  
      <hr />  
      <div className="modal">  
        <h2>오늘 해야할 일 2개</h2>  
        <h2>오늘 완료한 일 0개</h2>  
      </div>  
    </div>  
  );  
}  
export default List();
```

HTML에서는 어쩔 수 없이  
HTML 코드를 그대로 복사해서  
써야만 했었죠!





# 컴포넌트 활용하기 - 코드 줄이기!

- 그런데 저런 Modal 을 약 1000 개의 페이지 마다 계속 써줘야 한다면!?
- 그리고 Modal 에 갑자기 내용이 추가 되어야 한다면?
- 매우 골치가 아플 것입니다~!





# 컴포넌트 활용하기 - 코드 줄이기!

- 이럴 땐 해당 Modal 을 컴포넌트로 만들고 불러서 쓰면 쉽겠죠?
- 그럼 한번 해보시죠!



```
function Modal() {  
  return (  
    <div className="modal">  
      <h2>오늘 해야할 일 2개</h2>  
      <h2>오늘 완료한 일 0개</h2>  
    </div>  
  );  
}  
export default Modal;
```

```
import Modal from "./Modal";  
  
function List() {  
  return (  
    <div>  
      <h1>오늘 해야할일</h1>  
      <Modal />  
      <hr />  
      <h2>리액트 공부하기</h2>  
      <p>state 사용법 익히기</p>  
      <hr />  
      <h2>코테 문제 풀기</h2>  
      <p>Lv 0 정복 가즈아</p>  
      <hr />  
      <Modal />  
    </div>  
  );  
}  
export default List;
```



## 오늘 해야할일

오늘 해야할 일 2개

오늘 완료한 일 0개

### 리액트 공부하기

state 사용법 익히기

### 코테 문제 풀기

Lv 0 정복 가즈아

오늘 해야할 일 2개

오늘 완료한 일 0개





# 컴포넌트 활용하기 - 코드 줄이기!

- 이렇게 컴포넌트를 쓰면 코드를 크게 줄일 수 있으며, 재사용도 편리 합니다!
- 그리고, 모달에 내용이 추가 되어도 원본 파일의 코드만 바꾸면 모든 모달이 변하므로 이전 처럼 1000 번이나 각각 코드에 가서 수정을 할 필요도 없죠~!



엇, 그렇다면!?





```
import Modal from "./Modal";

function List() {
  return (
    <div>
      <h1>오늘 해야할일</h1>
      <Modal />
      <hr />
      <h2>리액트 공부하기</h2>
      <p>state 사용법 익히기</p>
      <hr />
      <h2>코테 문제 풀기</h2>
      <p>Lv 0 정복 가즈아</p>
      <hr />
      <Modal />
    </div>
  );
}

export default List;
```

이 친구들도 뭔가 반복적으로  
사용되는데.....

컴포넌트로 처리할 수 있지 않을까요?



# 그런데...



```
import Modal from "./Modal";

function List() {
  return (
    <div>
      <h1>오늘 해야할일</h1>
      <Modal />
      <hr />
      <h2>리액트 공부하기</h2>
      <p>state 사용법 익히기</p>
      <hr />
      <h2>코테 문제 풀기</h2>
      <p>Lv 0 정복 가즈아</p>
      <hr />
      <Modal />
    </div>
  );
}

export default List;
```

- 애들은 내용이 서로 좀 다르네요?
- 그럼 컴포넌트로 못하는 거 아니가요?





# Props



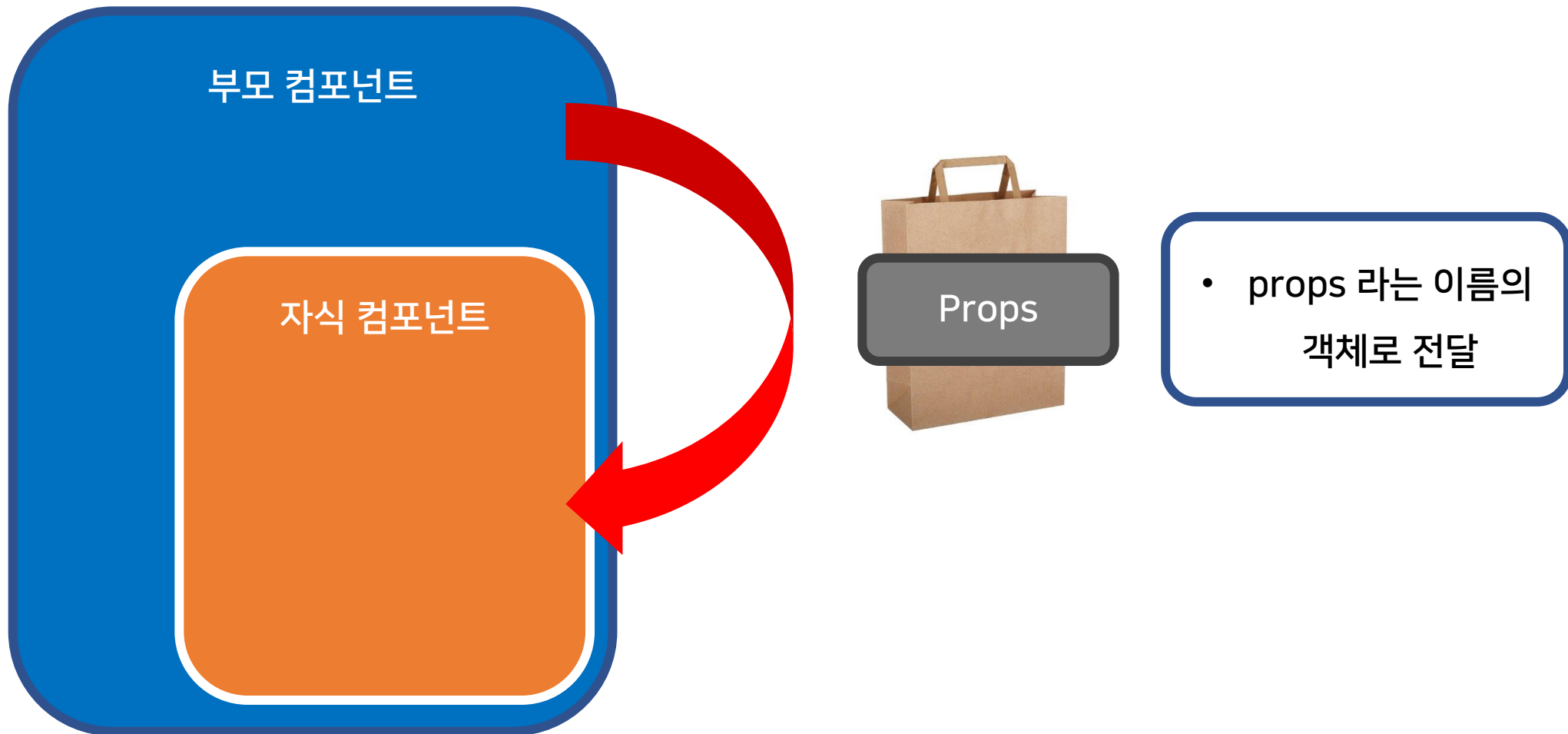
# 리액트는 이런 걸 다 생각했죠!

- 그래서 Props 라는 것을 준비 했습니다~!
- EJS 파일에서 백엔드 데이터를 받아서, 그것을 통해 HTML 을 그려주던 방법이 생각나시나요!?
- 리액트에서는 props(properties) 라는 것으로 손쉽게 전달이 가능합니다!



# Props?

- 부모 컴포넌트에서 자식 컴포넌트에 원하는 데이터를 보내는 방법입니다!



# 간단한 MainHeader 라는 컴포넌트 만들기



- MainHeader 의 매개변수 전달 부분에 {} 를 추가하고 부모로 부터 받아올 props 명을 적어 주시면 됩니다!

```
import React from "react";

function MainHeader({ text }) {
  return (
    <h1>{text}</h1>
  )
}

export default MainHeader;
```





# App.js 에서 props 사용하기

- App.js 의 컴포넌트 옆에 html 태그에 속성을 부여하는 것 처럼 props 명과 전달 하고자하는 데이터를 적어서 전달 하면 됩니다!

```
function App() {  
  return (  
    <div className="App">  
      <MainHeader text="Hello, props world!" />  
    </div>  
  );  
}
```

```
export default App;
```

src/App.js



```
function App() {  
  return (  
    <div className="App">  
      <MainHeader text="Hello, props world!" />  
      <MainHeader text="Bye Bye, props world!" />  
      <MainHeader text="Well come back, props world!" />  
    </div>  
  );  
}  
  
export default App;
```

src/App.js

**Hello, props world!**

**Bye Bye, props world!**

**Well come back, props world!**



# 다양한 데이터 받아오기!

- Props 는 다양한 데이터를 한꺼번에 받아 올 수 있습니다!
- 그리고 다양한 데이터는 props 라는 객체 하나로 받아서 사용이 가능합니다!

```
function App() {  
  return (  
    <div className="App">  
      <MainHeader text="Go naver" href="https://naver.com" userID="tetz!" />  
    </div>  
  );  
}
```

```
export default App;
```

src/App.js



```
function MainHeader(props) {  
  return (  
    <div>  
      <h1>{props.userID} 님 반갑습니다.</h1>  
      <a href={props.href}>{props.text}</a>  
    </div>  
  )  
}
```

src/components/MainHeader.js

- 물론 구조 분해 할당 문법도 사용이 가능합니다!



```
function MainHeader({ userID, href, text }) {  
  return (  
    <div>  
      <h1>{userID} 님 반갑습니다.</h1>  
      <a href={href}>{text}</a>  
    </div>  
  );  
}
```

src/components/MainHeader.js

```
function MainHeader(props) {  
  const { userID, text, href } = props;  
  return (  
    <div>  
      <h1>{userID} 님 반갑습니다.</h1>  
      <a href={href}>{text}</a>  
    </div>  
  )  
}
```

export default MainHeader;

src/components/MainHeader.js



# 클래스형 컴포넌트

## Props



# 클래스형 컴포넌트에서의 props

- 클래스형 컴포넌트인 만큼 기존의 props 로 접근하던 것을 this.props 로 접근 하시면 됩니다!
- 클래스이기 때문에 매개변수를 따로 받지 않습니다!
- This.props 를 쓰면 코드가 길어져서 보통 구조 분해 할당으로 변수로 만들어 사용합니다



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

class ClassProps extends Component {
  render() {
    const { name, age, nickName } = this.props;
    return (
      <div>
        <h1>이름: {name}</h1>
        <h2>나이: {age}</h2>
        <h2>별명: {nickName}</h2>
      </div>
    )
  }
}
```

```
export default ClassProps;
```

src/components/ClassProps.js





```
function App() {  
  return (  
    <div className="App">  
      <ClassProps name="뽀로로" age="5" nickName="사고뭉치" />  
    </div>  
  );  
}
```

src/App.js

**이름: 뽀로로**

**나이: 5**

**별명: 사고뭉치**



# 실습, 컴포넌트와 props 활용하기!

```
import Modal from "./Modal";

function List() {
  return (
    <div>
      <h1>오늘 해야할일</h1>
      <hr />
      <h2>리액트 공부하기</h2>
      <p>state 사용법 익히기</p>
      <hr />
      <h2>코테 문제 풀기</h2>
      <p>Lv 0 정복 가즈아</p>
      <hr />
    </div>
  );
}
export default List;
```

- 해당 코드를 컴포넌트와 Props 를 사용해서 수정하기!
- <h2>, <p>, <hr> 태그로 구성 된 오늘 할 일 목록을 ListChild 라는 컴포넌트로 만들어 주세요~!
- 내용은 Props 로 전달하여 오른쪽 코드의 결과물과 같은 결과물이 나올 수 있도록 만들어 주세요~!



리액트에서  
배열 활용하기!



# 리액트에서 배열 활용하기!

- 이제 컴포넌트와 Props 를 사용하는 방법을 배웠습니다!
- 그런데 props 로 전달하는 부분을 매번 저희가 작성을 해줘야 할까요?
- 아닙니다. 보통은 백엔드에서 데이터를 받아오게 되고 받은 데이터를 props 에 전달하는 형태를 많이 사용합니다~!

# 이런 데이터를 받았다고 가정을 해봅시다~!



```
const dataArr = [
  {
    title: "리액트 공부하기",
    detail: "state 사용법 익히기",
  },
  {
    title: "코테 문제 풀기",
    detail: "Lv0 정복 가즈아",
  },
];
```

- 데이터를 보니 뭔가 딱, 아까 ListChild  
에 props로 전달하면 딱이다 싶으시죠?



```
import ListChild from "./ListChild";
```

```
function List() {  
  const dataArr = [  
    {  
      title: "리액트 공부하기",  
      detail: "state 사용법 익히기",  
    },  
    {  
      title: "코테 문제 풀기",  
      detail: "Lv 0 정복 가즈아",  
    },  
  ],  
  
  return (  
    <div>  
      <h1>오늘 해야할일</h1>  
      <hr />  
      <ListChild title={dataArr[0].title} detail={dataArr[0].detail} />  
      <ListChild title={dataArr[1].title} detail={dataArr[1].detail} />  
    </div>  
  );  
}  
export default List;
```



# 크흠...

- 저렇게 코드를 쓰면 편할까요?? 당연히 불편하겠죠!?
- 그리고 데이터 배열의 길이가 변화한다면?
- 이래저래 안 좋은 코드가 될 겁니다!
- 그런데 지금 우리가 쓰고 있는건!? R.E.A.C.T 입니다!
- 즉, JSX 를 통해 HTML 과 JS 를 혼용해서 사용이 가능하죠!



# 배열 데이터를 리액트로 그리는 방법!

- React 에서는 배열 데이터를 그려 줄 때 배열 함수인 map 을 사용합니다!

```
return (  
  <div>  
    <h1>오늘 해야할일</h1>  
    <hr />  
    {dataArr.map((el) => {  
      return <ListChild title={el.title} detail={el.detail} />;  
    })}  
  </div>  
);
```





# 오늘 해야할일

## 리액트 공부하기

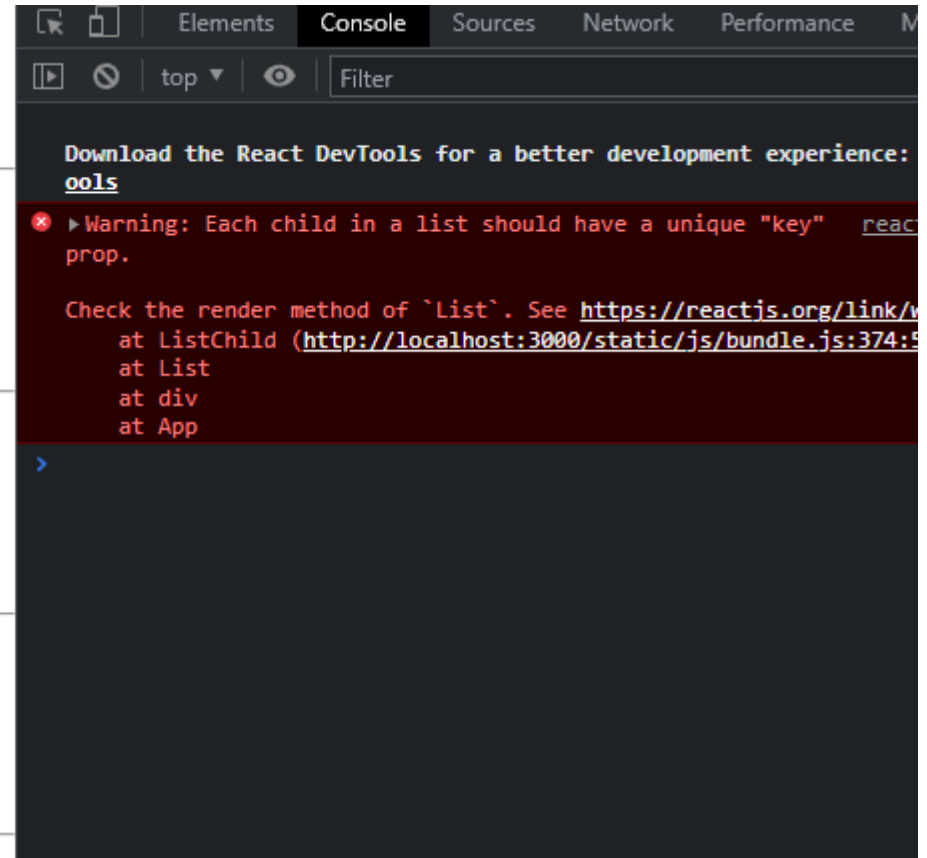
state 사용법 익히기

## 코테 문제 풀기

Lv 0 정복 가즈아

## 리액트 정복하기

리액트로 취뽀 가즈아





# 그런데 Warning 이!?

- React 에서 map 으로 배열 데이터를 그려줄 때에는 해당 요소에 Unique 한 Key 값 props 로 부여해 줘야 합니다~!
- 그래야만 나중에 리액트에서 map 으로 그려진 엘리먼트 중에서 어떤 것을 컨트롤(변경, 추가, 삭제 등)을 할 지 구분 할 수 있기 때문입니다!
- 구분을 해야하기 때문에 반드시 unique 한 값으로 넣어줘야 합니다!
- Key 의 값은 전역에서 unique 할 필요는 없지만, 반드시 형제 사이에서는 unique 하여야 합니다~!



```
import ListChild from "./ListChild";
function List() {
  const dataArr = [
    {
      title: "리액트 공부하기",
      detail: "state 사용법 익히기",
    },
    {
      title: "코테 문제 풀기",
      detail: "Lv 0 정복 가즈아",
    },
  ];
  return (
    <div>
      <h1>오늘 해야할일</h1>
      <hr />
      {dataArr.map((el, index) => {
        return <ListChild title={el.title} detail={el.detail} key={index} />;
      })}
    </div>
  );
}
export default List;
```

구별이 가능한 unique 한 key 값을  
props 로 부여하기!

단, index 는 최후의 수단으로 사용!





# Map 을 쓰면~!

- 귀찮게 하나하나 배열의 데이터를 넣어줄 필요도 없습니다!
- 그리고 배열 데이터가 변경이 되어도 자연스럽게 적용이 됩니다!
- 아 그리고, 당연히 컴포넌트에만 적용할 필요도 없습니다!
- Map 의 리턴 값으로 원하는 것을 넣어줘도 됩니다!
  - 단, 리턴되는 요소는 반드시 하나의 부모 요소를 가져야만 합니다!

```
return (  
  <div>  
    <h1>오늘 해야할일</h1>  
    <hr />  
    {dataArr.map((el, index) => {  
      return (  
        <div key={index}>  
          <h2>{el.title}</h2>  
          <p>{el.detail}</p>  
          <hr />  
        </div>  
      );  
    })}  
  </div>  
);
```





# 실습, 배열을 map 으로 그리기!

- 왼쪽과 같은 데이터를 map 을 이용해서 다음과 같은 화면으로 출력하기!
- 컴포넌트와 Props 사용 / HTML 코드를 리턴 둘 다 만들어 주세요~!

```
const items = [  
  {  
    item: "PS5",  
    price: "685,000원",  
  },  
  {  
    item: "에어 프라이어",  
    price: "50,000원",  
  },  
  {  
    item: "사세 치킨왕",  
    price: "11,500원",  
  },  
];
```

**품목명 : PS5**

가격 : 685,000원

**품목명 : 에어 프라이어**

가격 : 50,000원

**품목명 : 사세 치킨왕**

가격 : 11,500원



# Props

# 활용하기



# 배열을 전달하고 props 로 받아서 처리!

- Props 로는 배열 같은 다양한 자료형도 전달이 가능합니다!
- 배열을 받아서 처리하는 CustomList.js 컴포넌트를 만들어 봅시다!

```
function CustomList(props) {  
  return (  
    <ul>  
      {props.arr.map((el) => {  
        return <li>{el}</li>  
      })}  
    </ul>  
  )  
}
```

```
export default CustomList;
```

src/components/CustomList.js





# App.js 에서 배열을 전달

- App 에서 임의의 배열을 만들어서 전달하기!

```
function App() {  
  const nameArr = ['뽀로로', '루피', '크롱이'];  
  return (  
    <div className="App">  
      <CustomList arr={nameArr} />  
    </div>  
  );  
}
```

```
export default App;
```

src/App.js

뽀로로  
루피  
크롱이



# 객체를 전달하고 props 로 받아서 처리!

- 객체를 받아서 처리하는 CustomObj.js 컴포넌트를 만들어 봅시다!

```
function CustomObj(props) {  
  const { name, age, nickName } = props.obj;  
  return (  
    <div>  
      <h1>이름 : {name}</h1>  
      <h2>나이 : {age}</h2>  
      <h3>별명 : {nickName}</h3>  
    </div>  
  )  
}
```

```
export default CustomObj;
```

src/components/CustomObj.js



# App.js 에서 객체를 전달

- App 에서 임의의 객체를 만들어서 전달하기!

```
function App() {  
  const pororoObj = {  
    name: "뽀로로",  
    age: "5",  
    nickName: "사고뭉치"  
  }  
  return (  
    <div className="App">  
      <CustomObj obj={pororoObj} />  
    </div>  
  );  
}
```

src/App.js

이름 : 뽀로로

나이 : 5

별명 : 사고뭉치



**안전하게  
사용하기!**



# 좀 더 안전하게 사용하기!

- Props 로는 배열 같은 다양한 자료형도 전달이 가능합니다!
- 배열을 받아서 처리하는 CustomList.js 컴포넌트를 만들어 봅시다!

```
function CustomList(props) {  
  return (  
    <ul>  
      {props.arr.map((el) => {  
        return <li>{el}</li>  
      })}  
    </ul>  
  )  
}
```

```
export default CustomList;
```

src/components/CustomList.js



# 좀 더 안전하게 사용하기!

- 만약 props 로 배열 또는 객체 데이터가 안들어 온다면 어떻게 될까요?
- 이럴 때를 대비는 해주는 게 좋습니다!

```
function App() {  
  return (  
    <div className="App">  
      <CustomObj />  
    </div>  
  );  
}  
export default App;
```



```
✖ ▶ Uncaught TypeError: Cannot destructure property 'name' of 'props.obj' as it is undefined.  Cu
    at CustomObj (CustomObj.js:3:1)
    at renderWithHooks (react-dom.development.js:16305:1)
    at updateFunctionComponent (react-dom.development.js:19588:1)
    at beginWork (react-dom.development.js:21601:1)
    at HTMLUnknownElement.callCallback (react-dom.development.js:4164:1)
    at Object.invokeGuardedCallbackDev (react-dom.development.js:4213:1)
    at invokeGuardedCallback (react-dom.development.js:4277:1)
    at beginWork$1 (react-dom.development.js:27451:1)
    at performUnitOfWork (react-dom.development.js:26557:1)
    at workLoopSync (react-dom.development.js:26466:1)

✖ ▶ Uncaught TypeError: Cannot destructure property 'name' of 'props.obj' as it is undefined.  Cu
    at CustomObj (CustomObj.js:3:1)
    at renderWithHooks (react-dom.development.js:16305:1)
    at updateFunctionComponent (react-dom.development.js:19588:1)
    at beginWork (react-dom.development.js:21601:1)
    at HTMLUnknownElement.callCallback (react-dom.development.js:4164:1)
    at Object.invokeGuardedCallbackDev (react-dom.development.js:4213:1)
    at invokeGuardedCallback (react-dom.development.js:4277:1)
    at beginWork$1 (react-dom.development.js:27451:1)
    at performUnitOfWork (react-dom.development.js:26557:1)
    at workLoopSync (react-dom.development.js:26466:1)

✖ ▶ The above error occurred in the <CustomObj> component:  react devtools back
    at CustomObj (http://localhost:3000/main.41f7405...hot-update.js:26:13)
    at div
    at App

Consider adding an error boundary to your tree to customize error handling behavior.
Visit https://reactjs.org/link/error-boundaries to learn more about error boundaries.

✖ ▶ Uncaught TypeError: Cannot destructure property 'name' of 'props.obj' as it is undefined.  react-refresh-runtime.develop
    at CustomObj (CustomObj.js:3:1)
    at renderWithHooks (react-dom.development.js:16305:1)
    at updateFunctionComponent (react-dom.development.js:19588:1)
    at beginWork (react-dom.development.js:21601:1)
    at beginWork$1 (react-dom.development.js:27426:1)
    at performUnitOfWork (react-dom.development.js:26557:1)
    at workLoopSync (react-dom.development.js:26466:1)
    at renderRootSync (react-dom.development.js:26434:1)
    at recoverFromConcurrentError (react-dom.development.js:25850:1)
```



# IF문 사용하기!

- If 문을 사용해서 props 가 들어 왔는지 확인 후, 처리를 해줍니다
- 단, if 문은 JSX 문법 내부(return 내부)에서는 사용할 수 없습니다! → 그래서 3항 연산자를 사용했죠!
- 대신 return 밖에서 사용을 하고 상황에 따라 다른 return 을 주는 방식으로 가능합니다!





데이터가 없습니다.

```
function CustomObj(props) {  
  if (props.obj) {  
    const { name, age, nickName } = props.obj;  
    return (  
      <div>  
        <h1>이름 : {name}</h1>  
        <h2>나이 : {age}</h2>  
        <h3>별명 : {nickName}</h3>  
      </div>  
    )  
  } else {  
    return (  
      <div>  
        데이터가 없습니다.  
      </div>  
    )  
  }  
}  
export default CustomObj;
```



# Prototype Chaining 에 ? 사용하기

```
props.arr.map
```

- 해당 객체 또는 Key 값이 들어올지에 대한 확신이 없을 때에는 ? 를 사용해서 에러를 막을 수 있습니다~!
- JS 가 에러를 일으키는 상황
  - key 에 값이 없어서 undefined 값이 리턴 된 상황에서, 다시 key 값을 참조하려고 할 때
  - key 에 값이 없어서 undefined 값이 리턴 된 상황에서, 메소드를 실행하려고 할 때



# Prototype Chaining 에 ? 사용하기

- 참조한 key 에 값이 없어서 undefined 를 띄우는 것 까지는 괜찮습니다
- 하지만 undefined 값에서 뭔가를 하려고 하면~!?





# Prototype Chaining 에 ? 사용하기

```
function CustomList(props) {  
  return (  
    <ul>  
      {props.arr.map((el) => {  
        return <li>{el}</li>  
      })}  
    </ul>  
  )  
}  
export default CustomList;
```

```
✖ ▶ Uncaught TypeError: Cannot read properties of undefined (reading 'map')  
    at CustomList (CustomList.js:3:1)  
    at renderWithHooks (react-dom.development.js:16305:1)  
    at mountIndeterminateComponent (react-dom.development.js:20074:1)  
    at beginWork (react-dom.development.js:21587:1)  
    at HTMLUnknownElement.callCallback (react-dom.development.js:4164:1)  
    at Object.invokeGuardedCallbackDev (react-dom.development.js:4213:1)  
    at invokeGuardedCallback (react-dom.development.js:4277:1)  
    at beginWork$1 (react-dom.development.js:27451:1)  
    at performUnitOfWork (react-dom.development.js:26557:1)  
    at workLoopSync (react-dom.development.js:26466:1)
```



# Prototype Chaining 에 ? 사용하기

```
{props.arr?.map((el) => {
```

- 하지만 값이 들어올지 확실하지 않은 key 에 ? 를 붙여 주면, 해당 key 의 값이 들어 왔을 때에만 뒤의 chain 을 참조합니다!
- 즉, ? 를 붙인 key 가 undefined 면 뒤의 부분은 실행 되지 않습니다!
- If 문으로 처리 한 것 대비 깔끔하지는 않지만 Err 는 막을 수 있죠!
- 그런데 오히려 Err 를 발생시키고 깔끔하게 수정하는 편이 나을 수도 있습니다.
- 데이터를 받아오는 곳에 대한 확신이 없을 때 사용하면 좋습니다!

# Prototype Chaining 에 ? 사용하기



```
function CustomList(props) {  
  return (  
    <ul>  
      {props.arr?.map((el) => {  
        return <li>{el}</li>  
      })}  
    </ul>  
  )  
}  
export default CustomList;
```





# 실습, props 와 state 활용하기!

- 프로필 변경하기 버튼을 클릭하면 props 로 전달 된, 오브젝트 배열의 값이 순서대로 변하는 ChangeObj.js 컴포넌트를 만들어 주세요!
- 전달되는 오브젝트의 배열은 다음 장의 코드를 참고해 주세요!
- 함수형으로 구현해 주세요!

```
import ChangeObj from "./ChangeObj";
```

```
function Ex5() {  
  const pororoObjArr = [  
    {  
      name: "뽀로로",  
      age: "5",  
      nickName: "사고뭉치",  
    },  
    {  
      name: "루피",  
      age: "4",  
      nickName: "공주님",  
    },  
    {  
      name: "크롱이",  
      age: "5",  
      nickName: "장난꾸러기",  
    },  
  ];  
  return (  
    <div className="App">  
      <ChangeObj objArr={pororoObjArr} />  
    </div>  
  );  
}  
export default Ex5;
```







# useRef



# useRef? 이 친구는 또 뭐죠?

- 무언가 컴포넌트가 리렌더링 되어도 값을 유지하고 싶을 때가 있습니다!
- 어? 그럼 state 쓰면 되는거 아니가요?
- 그런데 값이 변경 되어도 컴포넌트 리렌더링을 하고 싶지는 않아요!
- 변수를 쓰면? → 리렌더링 시 변수 값은 초기화가 됩니다!
- 이럴 때 쓰는 것이 useRef 입니다!



# useRef? 이 친구는 또 뭐죠?

- 자 Input 태그를 사용해서 무언가를 입력 받는 상황을 그려 봅시다!
- 만약 이럴 때, useState 를 사용한다면?
- 인풋 태그 자체가 계속 리렌더링이 되어서 문제 + 낭비가 생길 겁니다!
- 이럴 때 쓰라고 있는 것이 useRef 되겠습니다!



# useRef

- useRef 를 사용하면 참조하고자 하는 DOM 요소에 ref 속성을 주고 해당 태그의 변화를 감지하거나 DOM 요소를 컨트롤 할 수 있습니다
- 보통은 컴포넌트에 존재하는 인풋 태그의 값을 받거나, JS에서 DOM 요소를 관리하던 역할을 해줍니다!



# JS 방식으로 구현하기

- Components 폴더에 TestRef.js 컴포넌트를 작성해 주세요!
- 인풋에 입력 된 값이, 상단에 있는 h1 태그의 컨텐츠가 되도록 구성하여 봅시다!
- 일단은, 기존의 JS 처럼 onChange 속성을 사용해서 이벤트 객체를 이용하여 구현하여 봅시다!



```
import TestRef from './components/TestRef';
```

```
function App() {  
  return (  
    <div className="App">  
      <TestRef />  
    </div>  
  );  
}
```

```
export default App;
```

src/App.js



```
import { useState, useRef } from "react";

export default function TestRef() {
  const [text, setText] = useState("안녕하세요!");

  const onChangeText = (e) => {
    const inputText = e.target.value;
    setText(inputText);
  }

  return (
    <div>
      <h1>{text}</h1>
      <input onChange={e => { onChangeText(e) }}></input>
    </div>
  )
}
```

dasdadada

dasdadada

src/components/TestRef.js



# useRef 로 구현하기

- useRef 를 사용하여 변화를 감지하거나 DOM 요소를 컨트롤 하고 싶은 태  
그에 ref 속성을 부여하기!



```
const ref = useRef(value)
```



```
<input ref={ ref } />
```



```
import { useState, useRef } from "react";

export default function TestRef() {
  const [text, setText] = useState("안녕하세요!");

  const inputValue = useRef();

  const onChangeText = () => {
    setText(inputValue.current.value);
  }

  return (
    <div>
      <h1>{text}</h1>
      <input ref={inputValue} onChange={onChangeText}></input>
    </div>
  )
}
```

src/components/TestRef.js



# useRef 값 확인하기

- useRef 로 설정한 값을 console.log 에 찍어 봅시다!

```
const onChangeText = () => {  
  console.log(inputValue);  
  setText(inputValue.current.value);  
}
```

```
▼ {current: input} ⓘ  
  ▼ current: input  
    value: "dsadad"  
    ▶ __reactEvents$3ew9d0gbjvx: Set(1) {'invalid__bubble'}  
    ▶ __reactFiber$3ew9d0gbjvx: FiberNode {tag: 5, key: null, el  
    ▶ __reactProps$3ew9d0gbjvx: {onChange: f}  
    ▶ _valueTracker: {getValue: f, setValue: f, stopTracking: f}  
    ▶ _wrapperState: {initialChecked: undefined, initialValue: '...
```



# useRef 로

# 포커스 이동 시키기!



# useRef 로 포커스 이동!

- useRef 가 자주 사용되는 포커스 사용 방법을 알아 봅시다!
- 컴포넌트 폴더에 ChangeFocus.js 컴포넌트를 만들어 봅시다!
- 2개의 인풋 태그를 만들고 해당 인풋에 ref 속성을 부여!
- useRef 로 각각 인풋의 속성 값을 변수에 담고 해당 변수를 통해 input 태그에 포커스를 부여해 봅시다!
- 해당 값에 대한 접근은 current 객체를 통해 해야합니다!



`focus()`



hello123

로그인



```
import { useState, useRef } from "react";
```

```
export default function TestRef() {
```

```
  const input1 = useRef();
```

```
  const input2 = useRef();
```

```
  const changeFocusOne = () => {  
    input1.current.focus();  
  }
```

```
  const changeFocusTwo = () => {  
    input2.current.focus();  
  }
```

```
  return (  
    <div>  
      <input ref={input1}></input>  
      <input ref={input2}></input>  
      <br></br>  
      <button onClick={changeFocusOne}>1</button>  
      <button onClick={changeFocusTwo}>2</button>  
    </div>  
  )  
}
```

src/components/ChangeFocus.js



# useRef 로 DOM 컨트롤



```
const ref = useRef(value)
```



```
<input ref={ ref } />
```



# useRef 를 사용하여 DOM 컨트롤!

- useRef 로 선언한 변수를 DOM 요소에 ref 속성으로 부여하면 해당 요소에 접근할 수 있습니다
- 마치 이전 VanilaJS 의 querySelector 또는 getElementById 같은 역할을 손쉽게 구현할 수 있죠! → 2개를 대체해서 사용이 가능합니다!
- 그럼 ref 로 접근한 DOM 요소를 컨트롤 해봅시다!



# useRef 를 사용하여 DOM 컨트롤!

- H1 태그 2개를 사용하고 useRef 로 DOM 요소에 접근하여 해당 DOM 요소의 스타일을 변경해 보겠습니다!
- 인풋 태그를 초기화도 해보죠!

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

export default function RefDOM() {
  const orangeEl = useRef();
  const skyblueEl = useRef();
  const inputEl = useRef();

  const adjustCSS = () => {
    orangeEl.current.style.backgroundColor = "orange";
    skyblueEl.current.style.backgroundColor = "skyblue";
  };

  const clearInput = () => {
    inputEl.current.value = "";
  };

  return (
    <div>
      <h1 ref={orangeEl}>Orange</h1>
      <h1 ref={skyblueEl}>Skyblue</h1>
      <input type="text" ref={inputEl} />
      <br />
      <button onClick={adjustCSS}>CSS 적용</button>
      <button onClick={clearInput}>인풋 초기화</button>
    </div>
  );
}
```





# 실습, useRef 활용하기!

- DIV 요소의 배경색을 직접 입력 받아서 변경하는 ColorInput.js 컴포넌트를 만들어 봅시다~!

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

export default function ColorInput() {
  return (
    <div>
      <input />
      <br />
      <button>색 적용</button>
    </div>
  )
}
```



# 실습, useRef 활용하기!

- 위의 코드를 변경해서 Input 창에 색을 입력하고 색 변경 버튼을 누르면 컴포넌트의 배경색이 변경되는 컴포넌트를 완성해 주세요~!
- Input 값을 받는 것과 DOM 컨트롤은 useRef 를 활용해서 구현해야 합니다!

색 적용

orange

색 적용

#ccc

색 적용

skyblue

색 적용





# 실습, useRef 활용하기!

- 숫자 퀴즈 프로그램을 만들어 봅시다!
- 컴포넌트가 랜더링이 되면 0 ~ 9 사이의 수를 랜덤으로 2개 생성합니다.
- 그리고 랜덤으로 더하기, 빼기, 곱하기 중에서 어떤 연산을 할지 정하여 퀴즈를 출제합니다. (나눗셈은 소수가 나오므로 X)

**2 x 8**

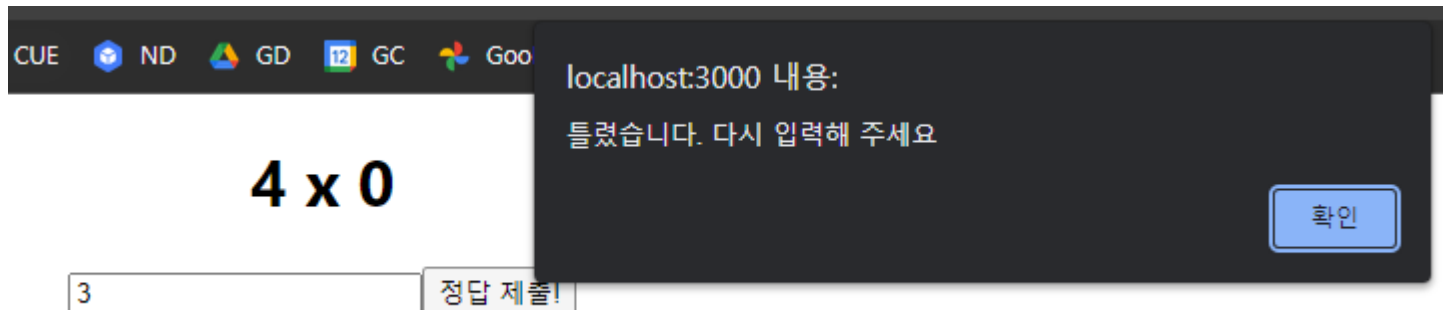
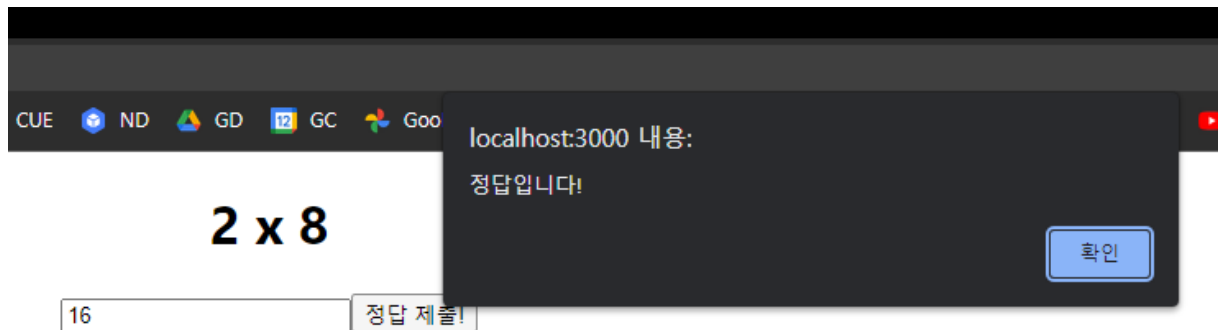
정답 제출!





# 실습, useRef 활용하기!

- useRef 를 활용하여 input 태그의 값을 입력 받아서 퀴즈의 정답이 맞을 경우, 틀릴 경우 각각 alert 을 띄워 줍니다





# 실습, useRef 활용하기!

- 정답을 입력한 경우 input 태그의 내용을 비워주고, input 태그로 포커스를 이동 시킨 뒤 컴포넌트를 리랜더링 하여 새로운 퀴즈를 출제합니다
- 오답을 입력한 경우 input 태그의 내용을 비워주고, input 태그로 포커스를 이동 시켜 줍니다
- 나누기는 소수점 계산이 필요하여 제외 합니다
- 답은 음수가 될 수 있습니다!



useState

useRef

Variable



# 3가지 타입에 대해서 정리해 봅시다!

- 지금까지 배운 useState 의 state 그리고 useRef 의 ref 와 리액트 내부의 변수가 렌더링에 따라 어떤 식으로 변화하는지 알아 봅시다!
- 그리고 state 에 변화에 따른 컴포넌트 리렌더링의 개념도 다시 한번 잡아 봅시다!



State

변경



Ref

변경



Variable

변경





# 코드로 봅시다!

- 세 가지 버튼으로 각각 state / ref / variable 값을 올리는 컴포넌트를 구성해 봅시다!
- 그리고 값의 변화 없이 컴포넌트 리렌더링을 위해서 하나의 버튼을 더 만들어 봅시다
- 그리고 각각의 버튼을 클릭하면서 해당 값의 변화에 대해 관찰해 봅시다!

```
import { useRef, useState } from "react";

const Comparing = () => {
  const [countState, setCountState] = useState(0);
  const [render, setRender] = useState(0);
  const countRef = useRef(0);
  let countVar = 0;

  const countUpState = () => {
    setCountState(countState + 1);
    console.log('State: ', countState);
  }

  const countUpRef = () => {
    countRef.current = countRef.current + 1;
    console.log('Ref: ', countRef.current);
  }

  const countUpVar = () => {
    countVar = countVar + 1;
    console.log('Variable: ', countVar);
  }

  const reRender = () => {
    setRender(render + 1);
  }
}
```

```
return (
  <>
    <h1>State: {countState}</h1>
    <h1>Ref: {countRef.current}</h1>
    <h1>Variable: {countVar}</h1>
    <button onClick={countUpState}>State UP!</button>
    <button onClick={countUpRef}>Ref UP!</button>
    <button onClick={countUpVar}>Variable UP!</button>
    <button onClick={reRender}>렌더링!</button>
  </>
)

src/components/Comparing.js

export default Comparing;
```



```
import { useRef, useState } from "react";
```

```
const Comparing = () => {  
  const [countState, setState] = useState(0);  
  const [render, setRender] = useState(0);  
  const countRef = useRef(0);  
  let countVar = 0;  
  
  const countUpState = () => {  
    setState(countState + 1);  
    console.log('State: ', countState);  
  }  
  
  const countUpRef = () => {  
    countRef.current = countRef.current + 1;  
    console.log('Ref: ', countRef.current);  
  }  
  
  const countUpVar = () => {  
    countVar = countVar + 1;  
    console.log('Variable: ', countVar);  
  }  
  
  const reRender = () => {  
    setRender(render + 1);  
  }  
  
  return (  
    <>  
      <h1>State: {countState}</h1>  
      <h1>Ref: {countRef.current}</h1>  
      <h1>Variable: {countVar}</h1>  
      <button onClick={countUpState}>State UP!</button>  
      <button onClick={countUpRef}>Ref UP!</button>  
      <button onClick={countUpVar}>Variable UP!</button>  
      <button onClick={reRender}>렌더링!</button>  
    </>  
  )  
}
```

```
export default Comparing;
```

**State: 4**

**Ref: 14**

**Variable: 0**

State UP! Ref UP! Variable UP! 렌더링!

src/components/Comparing.js





# React.Fragment



# React.Fragment?

- 자, 정말 간단한 컴포넌트를 만들어 볼게요!

```
export default function ReactFragment() {  
  return (  
    <div>  
      <h1>안녕하세요!</h1>  
      <span>반갑습니다!</span>  
    </div>  
  );  
}
```

src/components/ReactFragment.js



# React.Fragment?

- 그리고 페이지에서 개발자 도구를 열어 보면!

A screenshot of the React DevTools component inspector. It shows a tree view of the component structure. The root component is a <div id="root">. Inside it is a <div class="App">. Inside the App component is another <div> component. This inner div contains two children: an <h1>안녕하세요!</h1> and a <span>반갑습니다!</span>. A red arrow points to the inner <div> component in the tree view.

```
<div id="root">
  <div class="App">
    <div>
      <h1>안녕하세요!</h1>
      <span>반갑습니다!</span>
    </div>
  </div>
</div>
```

- 간단한 컴포넌트임에도 div 요소가 하나 추가 되었네요!
- 그럼, 저 div 를 없앨수 있을까요?



# React.Fragment?

- 리턴 값에서 최상위 태그 역할을 하는 DIV를 빼보시죠!

```
export default function ReactFragment() {  
  return [  
    <h1>안녕하세요!</h1>  
    <span>반갑습니다!</span>  
  ];  
}
```

```
ERROR in [eslint]  
src\components\ReactFragment.js  
  Line 4:6:  Parsing error: Adjacent JSX elements must be wrapped in an enclosing tag. Did you want a JSX fragment <>...</>? (4:6)
```

- 바로 에러가 뜹니다! 그런데 React 가 JSX fragment 를 추천하네요!?



# React.Fragment!

- 실제 리액트에서 컴포넌트를 조합할 때, 최상위에 div 를 사용하지 않고 반환해야만 하는 경우가 생기게 됩니다!
- CSS 가 깨진다거나, 테이블 요소 사이에 div 요소가 들어가면 예러가 뜨기 때문이죠!
- 그럴 때 쓰는 것이 바로 React.Fragment 입니다!



# React.Fragment!

- 이제 컴포넌트를 React.Fragment 로 감싸 봅시다!
- 이건 React 라이브러리의 기능이므로 React 라이브러리 추가 필요

```
import React from "react";

export default function ReactFragment() {
  return (
    <React.Fragment>
      <h1>안녕하세요!</h1>
      <span>반갑습니다!</span>
    </React.Fragment>
  );
}
```

src/components/ReactFragment.js



# React.Fragment!

- 이제는 div 가 사라졌네요!?

```
▼ <div id="root">
  ▼ <div class="App">
    <h1>안녕하세요!</h1>
    <span>반갑습니다!</span>
  </div>
</div>
```

- 바로 이러한 역할을 해주는 것이 React.Fragment 입니다!



<> </>

- 개발자들은 축약의 만족이기 때문에 이렇게 긴 코드를 용납 못합니다!
- <React.Fragment> 는 <> 로 대체가 가능합니다! :)

```
import React from "react";

export default function ReactFragment() {
  return (
    <>
      <h1>안녕하세요!</h1>
      <span>반갑습니다!</span>
    </>
  );
}
```

src/components/ReactFragment.js





# React.Fragment

## 가 필요할 때!



# CSS 님이 화가 나셨어!

- 먼저 App.js 에 가서 ReactFragment 컴포넌트와 동일한 코드를 작성해 봅시다

```
function App() {  
  return (  
    <div className="App">  
      <h1>안녕하세요</h1>  
      <span>반갑습니다</span>  
    </div>  
  );  
}
```

```
export default App;
```

src/App.js



# CSS 님이 화가 나셨어!

- 그리고 App.css 에 아래의 코드도 추가해 봅시다!

```
.App {  
  display: flex;  
  justify-content: space-between;  
}  
  
.App h1 {  
  background-color: skyblue;  
}  
  
.App span {  
  background-color: orange;  
}
```

src/App.css

# 결과물 화면~!



안녕하세요

반갑습니다

- 이런 결과물 화면이 나와야 정상입니다!



# ReactFragment 컴포넌트를 적용!

- ReactFragment 컴포넌트의 최상위 요소를 div 로 변경해서 적용시켜 봅시다!

```
import React from "react";

export default function ReactFragment() {
  return (
    <div>
      <h1>안녕하세요!</h1>
      <span>반갑습니다!</span>
    </div>
  );
}
```

src/components/ReactFragment.js



# ReactFragment 컴포넌트를 적용!

안녕하세요!

반갑습니다!

- Div 가 생기게 되어서 이전 결과물과는 완전히 다른 결과물이 나오게 됩니다!
- 따라서, 이런 일을 피하려면 React.Fragment 를 쓰셔야 합니다!



# 나는 너에게 속할 수 없어!

- 테이블 요소에 테이블 내용을 컴포넌트로 삽입하는 경우를 생각해 봅시다  
(사실 이게 쓸 일이 거의 없긴 합니다 ㅎㅎㅎ;;)
- 그런데 테이블 요소 안에는 div 태그가 들어가지 못합니다!
- 이럴때 문제가 생기는데 이것을 React.Fragment 로 해결이 가능합니다!

```
import TableColumn from
"./components/TableColumn";
```

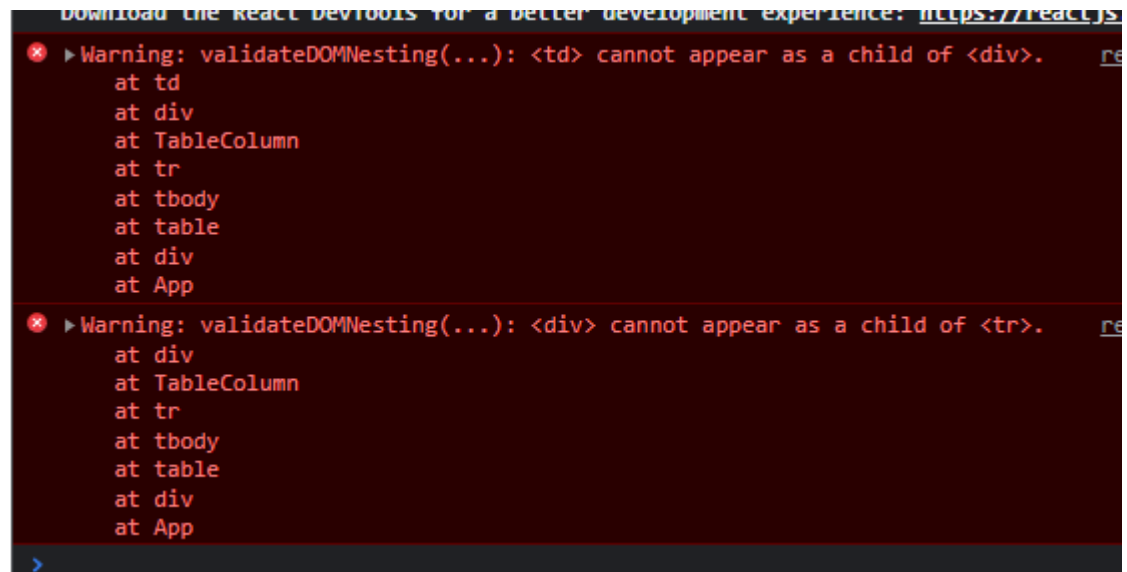
```
function App() {
  return (
    <div className="App">
      <table border="1">
        <tbody>
          <tr border="1">
            <td>1</td>
            <td>2</td>
            <td>3</td>
          </tr>
          <tr>
            <TableColumn />
          </tr>
        </tbody>
      </table>
    </div>
  );
}
```

```
export default App; src/App.css
```

```
import React from "react";
```

```
export default function ReactFragment() {
  return (
    <div>
      <td>a</td>
      <td>b</td>
      <td>c</td>
    </div>
  );
}
```

src/components/TableColumn.js







# 조건부 렌더링!



# 컴포넌트를 상황에 따라 켜고 끄기!

- 상황에 따라서 컴포넌트를 보여줄지 여부를 정해야 할 때가 있습니다!
- 그럴 때 사용하는 것이 조건부 렌더링 입니다!
- 이전 JS, HTML 에서는 Display 속성을 none 으로 해서 처리하곤 했습니다
- 리엑트는 JSX 문법을 사용 하므로, if 문 또는 3항 연산자, 논리 연산자와 HTML 태그를 같이 쓰면 되기 때문에 상당히 쉽습니다!
- 그럼 일단 한번 보시죠!



# 조건부 렌더링을 위한 컴포넌트 만들기

- Components 폴더에 ConditionalRender.js 파일 만들기

```
function ConditionalRender() {  
  return (  
    <h1>보이나요?</h1>  
  )  
}  
export default ConditionalRender;
```

src/components/ConditionalRender.js



# App.js 에서 조건부 렌더링 처리

- App.js 에서 useState 를 활용해서 condition 에 따른 조건부 렌더링을 처리해 봅시다!
- Condition 이 감추기이면 ConditionalRender 를 보여주고, 보이기이면 감춰 봅시다!
- 이럴 땐, 보통 && 연산자를 사용합니다!



```
import ConditionalRender from './components/ConditionalRender';
import { useState } from 'react';

function App() {

  const [condition, setCondition] = useState("보이기");

  const onChange = () => {
    condition === "보이기" ? setCondition("감추기") : setCondition("보이기");
  }

  return (
    <div className="App">
      {condition === "감추기" && <ConditionalRender />}
      <button onClick={onChange}>{condition}</button>
    </div>
  );
}

export default App;
```

src/App.js



# 변수로 처리해서 코드 정리!

- 조건부 렌더링 자체를 변수에 넣어서 처리해 봅시다!
- 이렇게 하면 코드가 깔끔해 지는 효과가 있습니다!
- 이런 것이 가능한 이유는 바로! → JSX 문법 덕분이죠!



```
import ConditionalRender from './components/ConditionalRender';
import { useState } from 'react';

function App() {

  const [condition, setCondition] = useState("보이기");

  const onChange = () => {
    condition === "보이기" ? setCondition("감추기") : setCondition("보이기");
  }

  const conditionRender = condition === "감추기" && <ConditionalRender />;

  return (
    <div className="App">
      {conditionRender}
      <button onClick={onChange}>{condition}</button>
    </div>
  );
}

export default App;
```

src/App.js



# 실습, 조건부 렌더링 처리!

- PracticeOne, PracticeTwo 컴포넌트를 만들어 주세요!
- props 로 데이터를 받아서 h1 태그로 출력하는 간단한 구조를 가집시다!
- App.js 에서 버튼을 클릭하면 PracticeOne 과 PracticeTwo 가 번갈아서 렌더링 되는 조건부 렌더링 처리를 해주세요!
- 버튼의 컨텐츠도 렌더링 되는 컨포넌트가 나오게 해주시면 됩니다!





**1번 컴포넌트**

1번



**2번 컴포넌트**

2번



# Life Cycle



# Mount

화면에 첫 렌더링



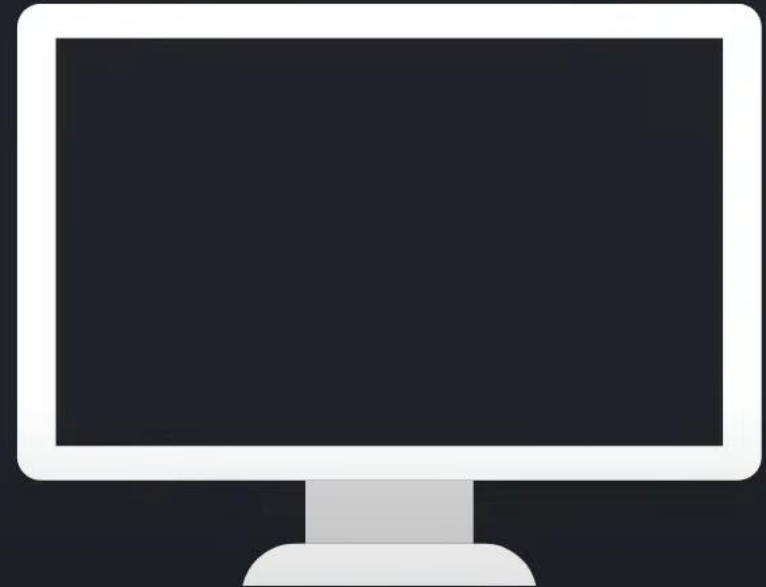
# Update

다시 렌더링



# Unmount

화면에서 사라질때





# 컴포넌트의 Life Cycle

- 컴포넌트는 최초에 화면에 등장 할 때 → Mount
- 컴포넌트의 state 변화로 리렌더링 될 때 → Update
- 화면에서 사라질 때 → Unmount
- 생명 주기를 가집니다!



# 클래스형 컴포넌트의 Life Cycle

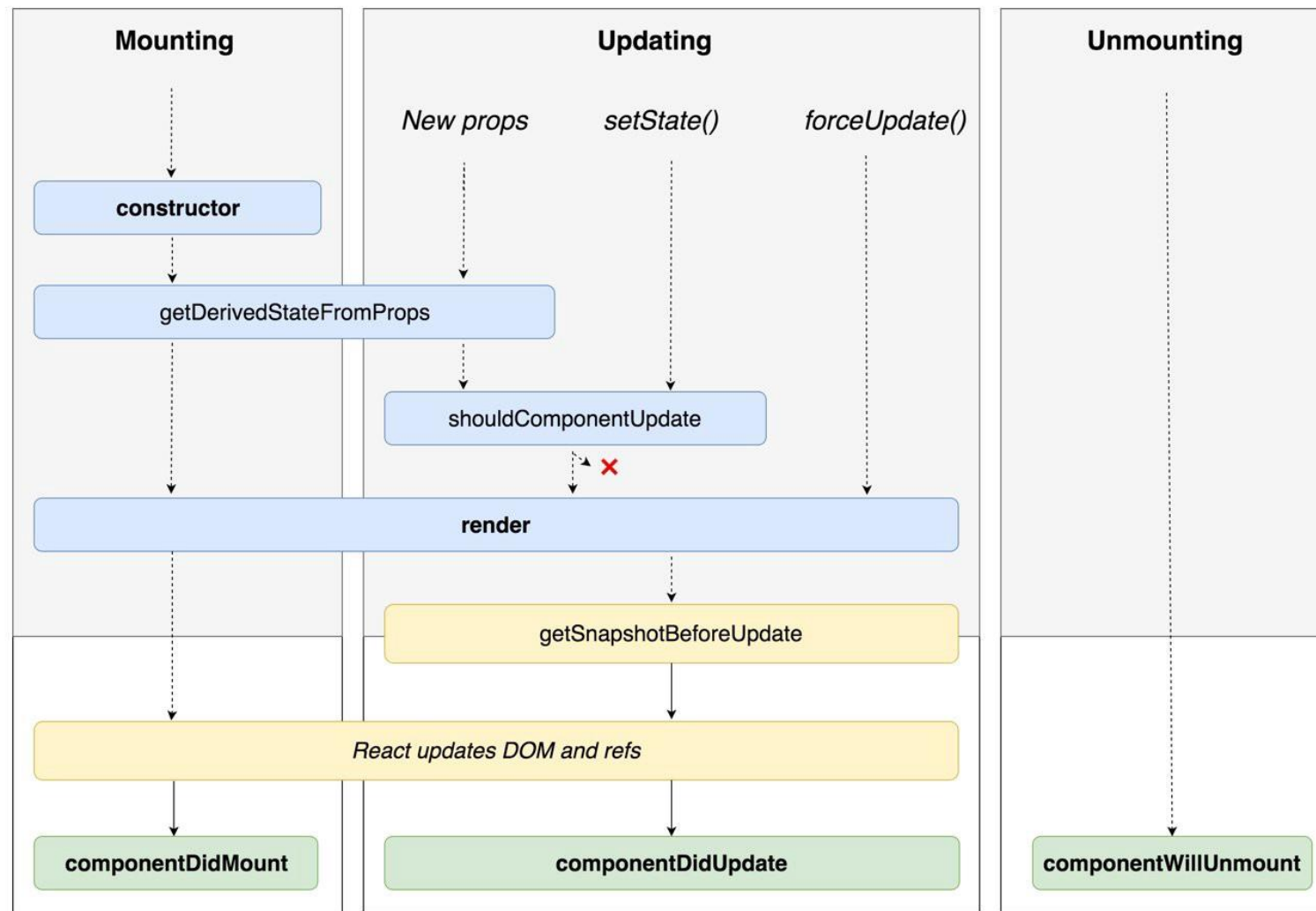
- 리액트의 장점은 이러한 상황에서 컴포넌트별 상태 관리 및 리렌더링에 있기 때문에 리액트는 Life Cycle 에 대한 기능이 많습니다!
- 따라서 각각의 Life Cycle 상황에 맞게 특정 기능을 수행할 수 있도록 다양한 기능을 제공 했습니다



**“Render Phase”**  
Pure and has no side effects.  
May be paused, aborted or  
restarted by React.

**“Pre-Commit Phase”**  
Can read the DOM.

**“Commit Phase”**  
Can work with DOM,  
run side effects,  
schedule updates.



componentDidMount

componentDidUpdate

componentWillUnmount



# Mount

화면에 첫 렌더링



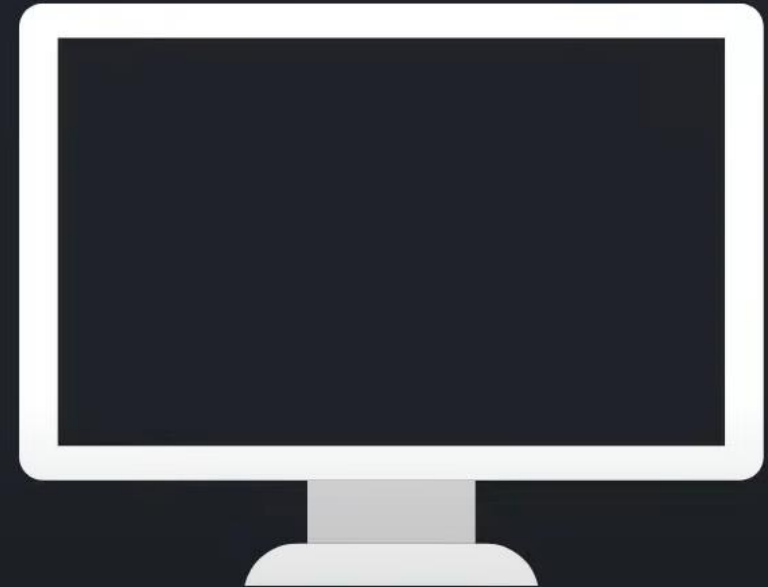
# Update

다시 렌더링



# Unmount

화면에서 사라질때





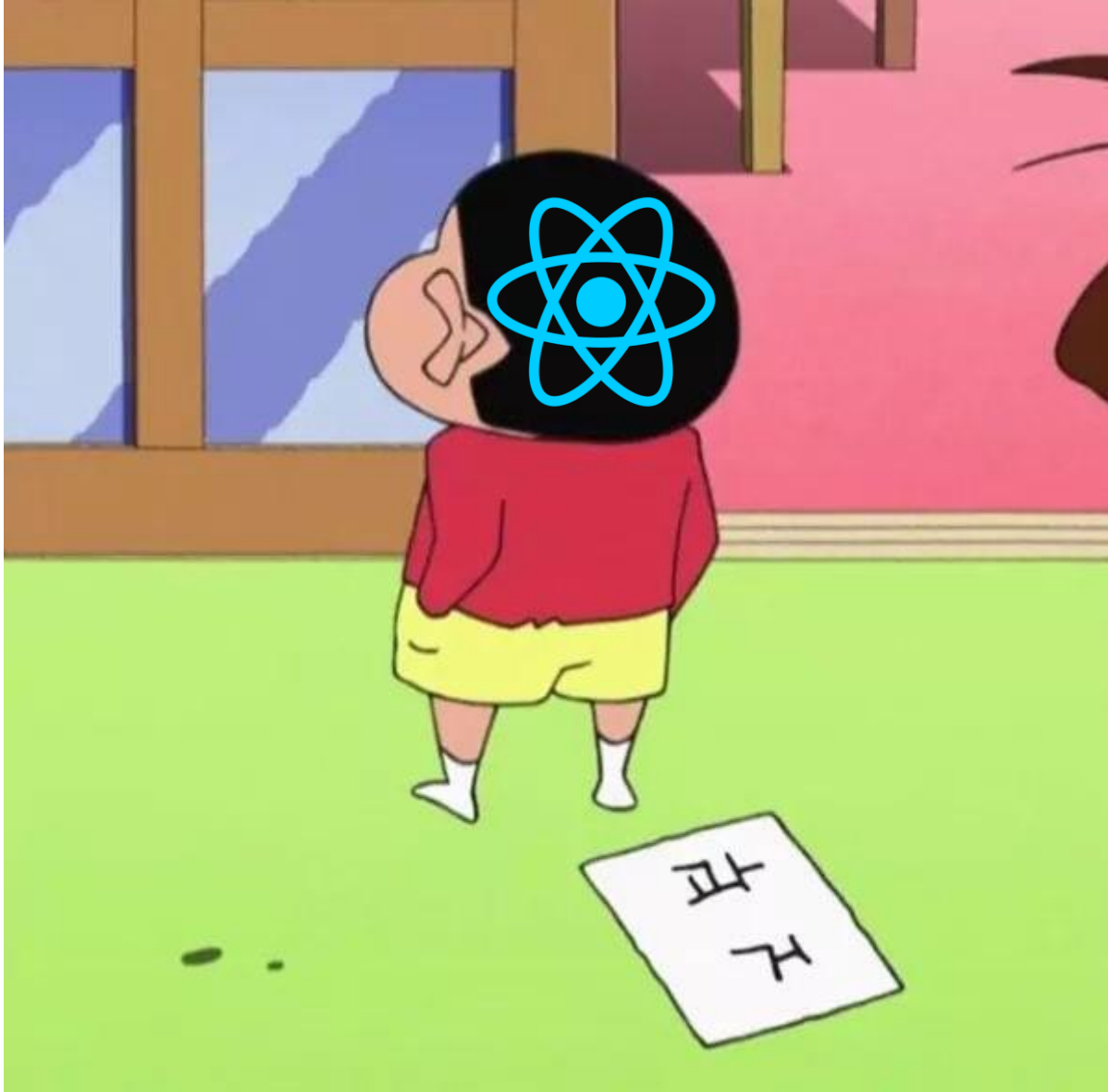
```
class Cycle extends React.Component {  
  componentWillMount(){  
    alert("render전 가장 먼저 호출")  
  }  
  componentDidMount(){  
    alert("ComponentDidMount호출 됨")  
  }  
  
  shouldComponentUpdate(nextProps, nextState){  
    return this.props.name !== nextProps.name;  
  }  
  
  render(){  
    alert("render 호출");  
    return <h2>hello, {this.props.name}</h2>  
  }  
}
```





# 클래스형 컴포넌트의 Life Cycle

- 리액트는 위와 같은 기능을 통해서 컴포넌트가 처음 불러 왔을 때, 리렌더링 되었을 때, 컴포넌트가 사라질 때 마다 특정한 기능을 수행 할 수 있도록 처리를 했습니다!



# Mount

화면에 첫 렌더링



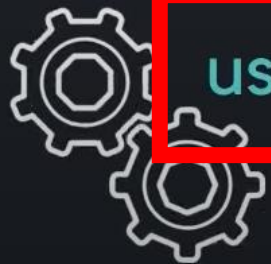
# Update

다시 렌더링



# Unmount

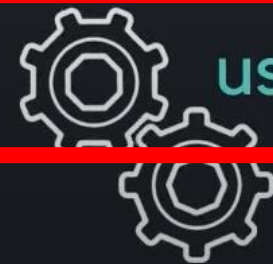
화면에서 사라질때



useEffect



useEffect



useEffect



# 함수형 컴포넌트의 Life Cycle

- 하지만 우리 리액트에게 과거는 과거일 뿐 더 빠르고 편리한 미래만을 그립니다!
- 따라서, 클래스형 컴포넌트에서 함수형 컴포넌트로 넘어 가면서 기존의 Life Cycle 을 담당하던 기능을 하나의 HOOK에 추가 시켰습니다!
- 바로 UseEffect 입니다!



# useEffect

별코딩★

<https://www.youtube.com/watch?v=kyodvzc5GHU>

1

```
useEffect( ( ) => {
```

```
  // 작업...
```

```
});
```

렌더링 될때 마다 실행

2

```
useEffect( ( ) => {  
    // 작업...  
}, [ value ] );
```

화면에 첫 렌더링 될때 실행

value 값이 바뀔때 실행

# Clean Up - 정리



```
useEffect( ( ) => {  
    // 구독 ...  
  
    return ( ) => {  
        // 구독 해지 ...  
    }  
}, [ ] );
```





# 그럼 코드로 한번 확인해 봅시다!

- 버튼을 클릭하면 count 가 +1 이 되어 숫자가 증가되는 컴포넌트를 만들어 봅시다!
- 그리고 useEffect 를 사용해서 state 값 변경에 따라 컴포넌트라 렌더링 될 때 마다 useEffect 가 작동하는지 알아 봅시다!



```
import { useEffect, useState } from "react";

export default function TestUseEffect() {
  const [count, setCount] = useState(0);

  const onClick = () => {
    console.log("👉 버튼 클릭");
    setCount(count + 1);
  }

  useEffect(() => {
    console.log("🌀 렌더링 할 때마다 실행되는 useEffect");
  })

  return (
    <>
      <h1>{count}</h1>
      <button onClick={onClick}>+1 버튼</button>
    </>
  )
}
```

src/components/TestUseEffect.js



🖱️ 버튼 클릭

🔄 렌더링 할 때마다 실행되는 `useEffect`

🖱️ 버튼 클릭

🔄 렌더링 할 때마다 실행되는 `useEffect`

🖱️ 버튼 클릭

🔄 렌더링 할 때마다 실행되는 `useEffect`

🖱️ 버튼 클릭

🔄 렌더링 할 때마다 실행되는 `useEffect`

🖱️ 버튼 클릭

🔄 렌더링 할 때마다 실행되는 `useEffect`



# 그럼 코드로 한번 확인해 봅시다!

- 지금은 버튼이 클릭 되면 state 값의 변경이 일어나기 때문에 컴포넌트가 다시 렌더링이 되고, 그로 인해서 useEffect 내부의 함수가 실행 되고 있습니다!
- 그럼 여기에다가 input 태그를 하나 추가하고, input 태그의 입력 내용이 h1 태그에 출력 되도록 코드를 변경해 봅시다!

```
import { useEffect, useRef, useState } from "react";
```

```
export default function TestUseEffect() {  
  const [count, setCount] = useState(0);  
  const [text, setText] = useState("입력 하세요!");  
  const inputValue = useRef();  
  
  const onClick = () => {  
    console.log("🖱️ 버튼 클릭");  
    setCount(count + 1);  
  }  
  
  const onChange = () => {  
    console.log("💻 키 입력");  
    setText(inputValue.current.value);  
  }  
  
  useEffect(() => {  
    console.log("🔄 렌더링 할 때마다 실행되는 useEffect");  
  })  
  
  return (  
    <>  
      <h1>{count}</h1>  
      <button onClick={onClick}>+1 버튼</button>  
      <br /><br /><br /><br />  
      <input ref={inputValue} onChange={onChange}></input>  
      <h1>{text}</h1>  
    </>  
  )  
}
```

src/components/TestUseEffect.js





2

+1 버튼

qwe

qwe

👤 렌더링 할 때마다 실행되는 <code>useEffect</code>
🖱️ 버튼 클릭
👤 렌더링 할 때마다 실행되는 <code>useEffect</code>
💻 키 입력
👤 렌더링 할 때마다 실행되는 <code>useEffect</code>
💻 키 입력
👤 렌더링 할 때마다 실행되는 <code>useEffect</code>
💻 키 입력
👤 렌더링 할 때마다 실행되는 <code>useEffect</code>
🖱️ 버튼 클릭
👤 렌더링 할 때마다 실행되는 <code>useEffect</code>
>



# useEffect Dependency

2

```
useEffect( ( ) => {  
    // 작업...  
}, [ value ] );
```

화면에 첫 렌더링 될때 실행

value 값이 바뀔때 실행





# useEffect 의 Dependency Array

- useEffect 는 두번째 인자로 Dependency Array 를 받습니다
- 해당 Array 에는 변수를 넣을 수가 있으며, 해당 변수가 변경 될 때에만 useEffect 내부의 함수가 실행 됩니다!
- + 빈 배열 [] 을 넣으면 최초 마운트 시에만 실행이 됩니다!
- 그럼 아래의 3가지 useEffect 코드를 추가한 뒤 테스트를 해봅시다!



```
useEffect(() => {  
  console.log("👁 렌더링 할 때마다 실행되는 useEffect");  
})
```

```
useEffect(() => {  
  console.log("👉 버튼 클릭 시에만 실행되는 useEffect");  
}, [count])
```

```
useEffect(() => {  
  console.log("🖱 인풋 입력 시에만 실행되는 useEffect");  
}, [text])
```

src/components/TestUseEffect.js



src/components/TestUseEffect.js

전체 코드

```
import { useEffect, useRef, useState } from "react";

export default function TestUseEffect() {
  const [count, setCount] = useState(0);
  const [text, setText] = useState("입력 하세요!");
  const inputValue = useRef();

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

  const onChange = () => {
    setText(inputValue.current.value);
  }

  useEffect(() => {
    console.log("👁 렌더링 할 때마다 실행되는 useEffect");
  })

  useEffect(() => {
    console.log("👉 버튼 클릭 시에만 실행되는 useEffect");
  }, [count])

  useEffect(() => {
    console.log("🖱 인풋 입력 시에만 실행되는 useEffect");
  }, [text])

  return (
    <>
      <h1>{count}</h1>
      <button onClick={onClick}>+1 버튼</button>
      <br /><br /><br /><br />
      <input ref={inputValue} onChange={onChange}></input>
      <h1>{text}</h1>
    </>
  )
}
```



# useEffect 의 Dependency Array

- 이제 버튼을 클릭하면 Dependency 로 count 를 전달한 useEffect 만 작동하고, 인풋에 값을 입력하면 Dependency 로 value 를 전달한 useEffect 만 작동하게 됩니다!
- 물론 Dependency Arr 를 전달하지 않은 경우는 렌더링 때마다 실행

🌀 렌더링 할 때마다 실행되는 `useEffect`

👉 버튼 클릭 시에만 실행되는 `useEffect`

🌀 렌더링 할 때마다 실행되는 `useEffect`

👉 인풋 입력 시에만 실행되는 `useEffect`



# Dependency Array 를 전달하면?

- 두번째 인자 자체를 전달 하지 않으면, 매번 랜더링 마다 실행이 되지만 빈 배열을 두번째 인자로 전달 한다면?

```
useEffect(() => {  
    console.log("초기 마운트 시에만 실행");  
}, []);
```

- 요 친구는 변화를 감지할 값이 없으므로 최초 마운트 시에만 실행이 됩니다!



# useEffect

# Clean-Up



# useEffect 정리하기

- 지금까지 useEffect 를 컴포넌트가 마운트 되는 순간과 리렌더링 되는 순간에 적용하여 사용하는 방법을 배웠습니다
- 그럼, 컴포넌트가 Unmount 되는 순간에는 어찌 처리하면 될까요?
- 클래스형에서는 componentWillUnmount 라는 메소드를 사용했지만 useEffect HOOK 에서는 useEffect 의 리턴에 함수를 부여하면 됩니다!

# Clean Up - 정리



```
useEffect( ( ) => {  
    // 구독 ...  
  
    return ( ) => {  
        // 구독 해지 ...  
    }  
}, [ ] );
```





# 코드로 보기!

- 간단한 Timer 컴포넌트를 만들어 봅시다!
- 버튼을 클릭하면 setInterval 함수를 통해 1초에 한번씩 console.log 를 찍는 컴포넌트 입니다!
- 그리고 해당 컴포넌트를 조건부 렌더링 처리하여 Mount 와 Unmount 를 시킬 수 있도록 만들어 봅시다!



```
import { useEffect } from "react";

export default function Timer() {
  useEffect(() => {
    setInterval(() => {
      console.log(`타이머 실행중`)
    }, 1000);
  }, []);

  return (
    <>
    <h1>타이머가 실행 중입니다!</h1 >
    </>
  )
}
```

src/components/Timer.js



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

function App() {
  const [show, setShow] = useState(false);

  return (
    <div className="App">
      {show && <Timer />}
      <button onClick={() => setShow(!show)}>버튼</button>
    </div>
  );
}

export default App;
```

src/App.js

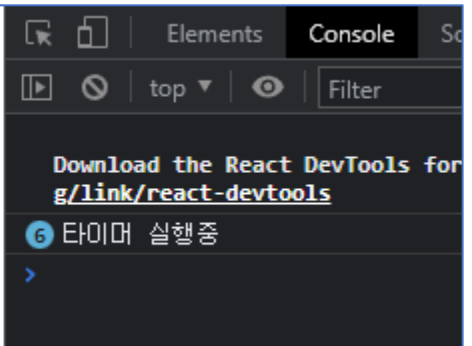


# Timer 컴포넌트

- 버튼을 클릭하면 Timer 가 돌기 시작합니다!

타이머가 실행 중입니다!

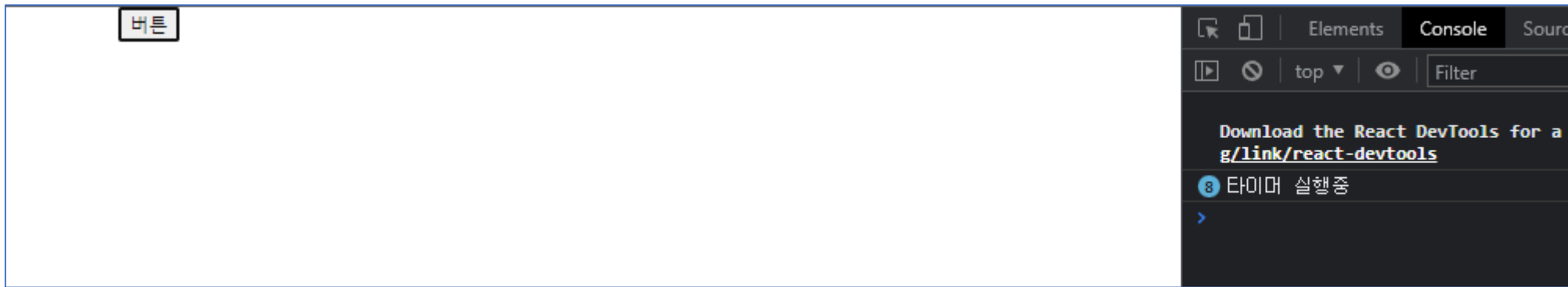
버튼





# Timer 컴포넌트

- 하지만 버튼을 다시 눌러서 Unmount 를 시켜 봅시다!



- 컴포넌트가 unmount 가 되어도 타이머는 계속 돌아가게 됩니다!

# Clean-up



- 이런 상황을 방지하기 위해서 Unmount 가 되면 실행되는 Clean-up 을 이용, 타이머를 제거해 줍시다!
- 기존의 useEffect 코드에 return 으로 Clean-up 함수를 지정하여 줍시다



```
import { useEffect } from "react";

export default function Timer() {
  useEffect(() => {
    const timer = setInterval(() => {
      console.log(`타이머 실행중`)
    }, 1000);

    return (() => {
      clearInterval(timer);
    })
  }, []),

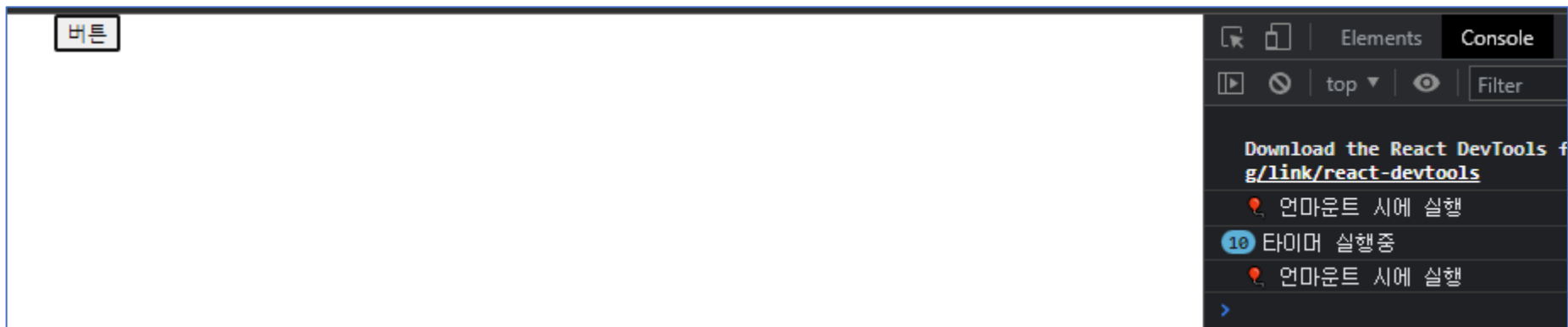
  return (
    <>
    <h1>타이머가 실행 중입니다!</h1 >
    </>
  )
}
```

src/components/Timer.js

# Clean-up



- 이제 Unmount 가 되는 상황에서는 return 에 인자로 전달한 함수가 실행이 되고 타이머가 정상 종료 됩니다!







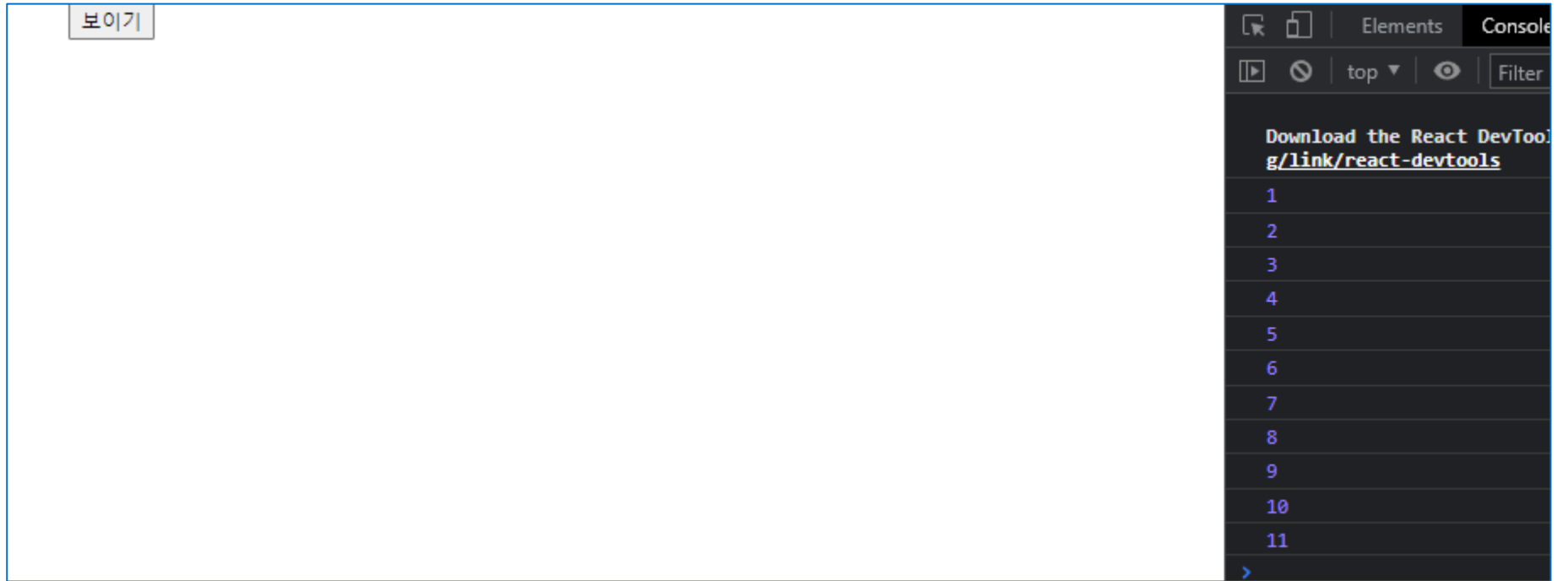
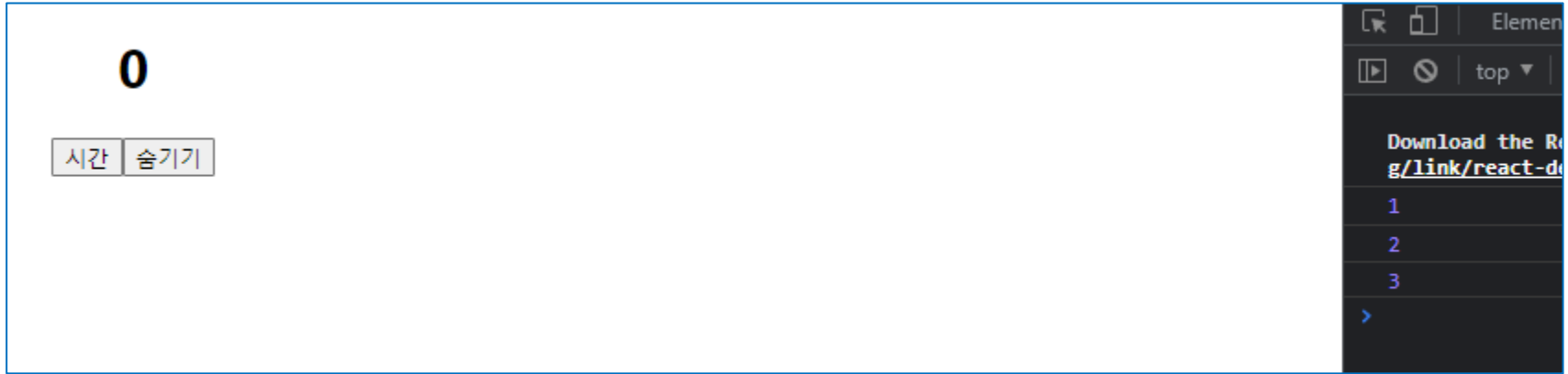
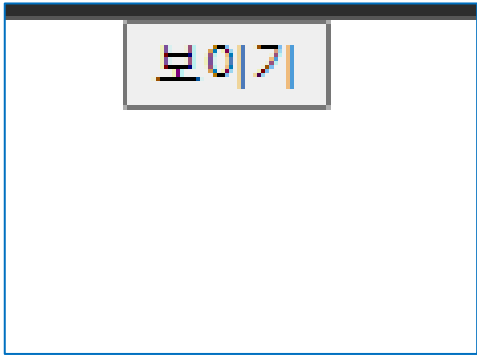
# 실습, 컴포넌트 타이머!

- PracticeTimer 라는 컴포넌트를 만들어 줍시다!
- App.js 에는 '보이기' 라는 버튼이 하나 있습니다. 해당 버튼을 클릭하면 PracticeTimer 가 마운트 됩니다.
- 버튼을 클릭하면 PracticeTimer 컴포넌트가 마운트 되고 마운트가 된 시간을 초단위로 기록하는 타이머가 실행 됩니다.
- PracticeTimer 에도 '시간' 버튼이 존재하며, 해당 버튼을 누르면 지금까지 마운트가 된 시간을 출력해 줍니다!



# 실습, 컴포넌트 타이머!

- '보이기' 버튼을 한번 더 클릭하면, PracticeTimer 가 Unmount 되고 타이머도 종료 되어야 합니다!
- 추가, '보이기' 버튼이 클릭 되면 버튼 이름을 '숨기기'로 변경 하기!
- 추가, 페이지가 처음 시작 되면 '보이기' 버튼에 포커스가 이동하도록 처리





# useEffect

## 실전 활용!



# 실전 활용?

- useEffect 는 다양한 곳에서 활용이 가능합니다!
- 보통 컴포넌트가 서버로 부터 데이터를 받아와야 하는 상황에서 많이 사용
- 컴포넌트가 최초 마운트 → 서버로 부터 데이터를 요청 → 데이터를 State  
에 등록 → 해당 내용을 렌더링
- 위와 같은 흐름을 많이 사용합니다!



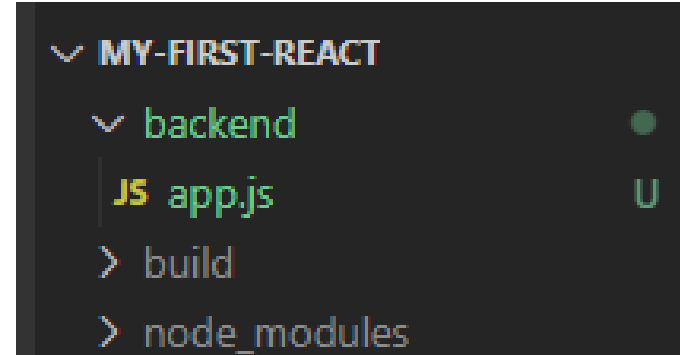
# 간단한 백엔드 서버 구성하기!

- 지난번 실습으로 했었던 pororoObjArr 라는 배열을 전달해 주는 백엔드 서버를 간단하게 구축해 봅시다!
- 따로 만들기는 귀찮으니 리액트 폴더에 backend 라는 폴더를 만들어서 간단하게 구현해 봅시다!
- 왜냐면 리액트는 이미 npm 이 관리하는 폴더이기 때문에 간단하게 필요 패키지 모듈만 설치하여 사용하면 되기 때문이죠!

# 간단한 백엔드 서버 구성하기!



- 일단 리액트 폴더에 backend 폴더 생성
- Express 와 cors 모듈 설치
- Npm i express cors



# server.js 코드



```
const express = require("express");
const cors = require("cors");

const PORT = 4000;
const app = express();

app.use(cors());
app.get("/", (req, res) => {
  const pororoObjArr = [];
  res.send(JSON.stringify(pororoObjArr));
});

app.listen(PORT, () => {
  console.log(`데이터 통신 서버가 ${PORT}에서 작동 중입니다!`);
});
```

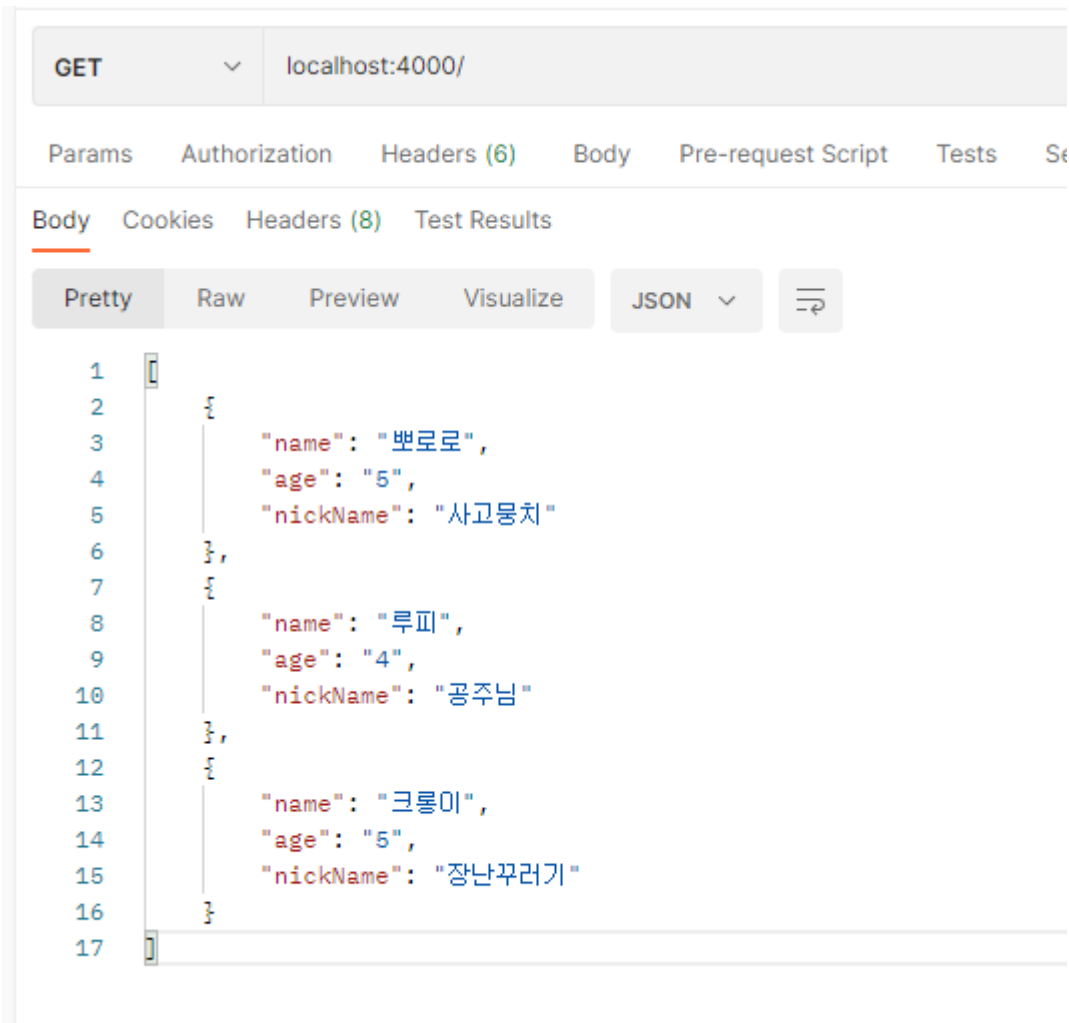
```
const pororoObjArr = [
  {
    name: "뽀로로",
    age: "5",
    nickName: "사고뭉치",
  },
  {
    name: "루피",
    age: "4",
    nickName: "공주님",
  },
  {
    name: "크롱이",
    age: "5",
    nickName: "장난꾸러기",
  },
];
```



# 간단한 백엔드 서버 구성하기!



- 서버 실행 후 Postman 으로 테스트



# 리액트 컴포넌트 만들기!



- 백엔드 데이터를 받아서 그려줄 UseEffectFetch.jsx 컴포넌트 생성
- 함수가 마운트 되면 Fetch 함수를 이용해서 만들어 둔, API로 데이터를 요청하고 해당 데이터를 받아서 state 에 부여해 봅시다!



```
import React, { useEffect, useState } from "react";

export default function UseEffectFetch() {
  const [dataArr, setDataArr] = useState([]);

  async function fetchData() {
    const resFetch = await fetch("http://localhost:4000/", {
      method: "GET",
      headers: {
        "Content-type": "application/json",
      },
    });
    const data = await resFetch.json();
    setDataArr(data);
    console.log(dataArr);
  }

  useEffect(() => {
    fetchData();
  }, []);
}
```

```
▼ (3) [{...}, {...}, {...}] ⓘ
  ▶ 0: {name: '뽀로로', age: '5', nickName: '사고뭉치'}
  ▶ 1: {name: '루피', age: '4', nickName: '공주님'}
  ▶ 2: {name: '크롱이', age: '5', nickName: '장난꾸러기'}
    length: 3
  ▶ [[Prototype]]: Array(0)
```

# 받은 데이터를 그려 봅시다!



- 그런데 데이터가 일정하죠? 그럼 하나하나 그릴 필요가 없겠죠?
- 바로 컴포넌트와 props 를 활용해 봅시다!
- 데이터를 그려주는 ProfileComponent.jsx 를 생성해 봅시다!

```
export default function ProfileComponent({ name, age, nickName }) {  
  return (  
    <div>  
      <h1>이름 : {name}</h1>  
      <h1>나이 : {age}</h1>  
      <h1>별명 : {nickName}</h1>  
      <hr />  
    </div>  
  );  
}
```



# 컴포넌트를 импорт 하고 map 으로 그려주기!



- 생성한 ProfileComponent 컴포넌트를 импорт 하고, map 을 이용하여 그려 줍시다!



```
import ProfileComponent from "../ProfileComponent";  
// 기존 코드
```

```
return (  
  <>  
    {dataArr.map((el) => {  
      return (  
        <ProfileComponent  
          key={el.name}  
          name={el.name}  
          age={el.age}  
          nickName={el.nickName}  
        />  
      );  
    })}  
  </>  
);
```



이름 : 뽀로로

나이 : 5

별명 : 사고뭉치

이름 : 루피

나이 : 4

별명 : 공주님

이름 : 크롱이

나이 : 5

별명 : 장난꾸러기

```
Elements Console Sources Network Performance
top Filter
Download the React DevTools for a better development experience
act-devtools
(3) [{...}, {...}, {...}]
  0: {name: '뽀로로', age: '5', nickName: '사고뭉치'}
  1: {name: '루피', age: '4', nickName: '공주님'}
  2: {name: '크롱이', age: '5', nickName: '장난꾸러기'}
  length: 3
  [[Prototype]]: Array(0)
(3) [{...}, {...}, {...}]
>
```







수고하셨습니다!