



**Seoul
Software
ACademy**

with





useRef

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



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



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

export default function TestUseRef() {
  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/TestUseRef.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!



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

export default function ChangeFocus() {
  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



```
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>
  );
}
```



useState

useRef

Variable



State

변경



Ref

변경



Variable

변경





```
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



<> </>

- 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



조건부 렌더링!



```
import { useState } from 'react';
import Item from './Item';

function ConditionalRender() {
  const [condition, setCondition] = useState('보이기');

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

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

export default ConditionalRender;
```




Life Cycle



Mount

화면에 첫 렌더링



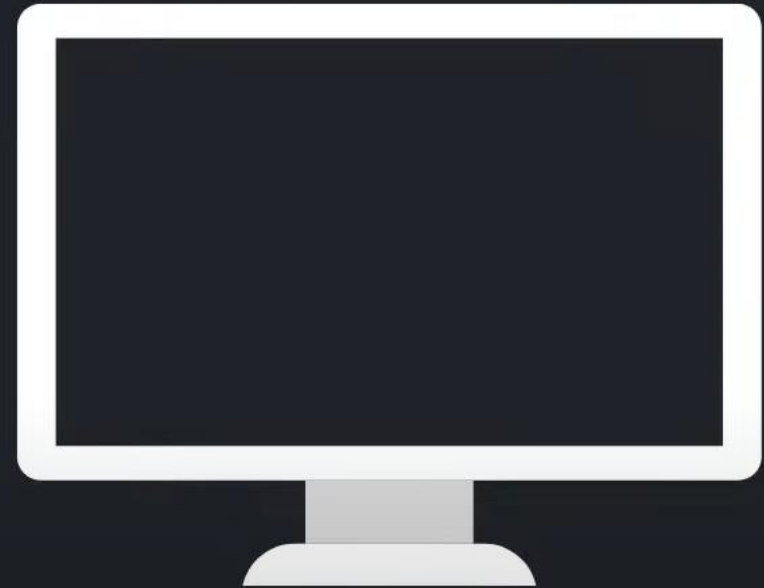
Update

다시 렌더링



Unmount

화면에서 사라질때



componentDidMount

componentDidUpdate

componentWillUnmount



Mount

화면에 첫 렌더링



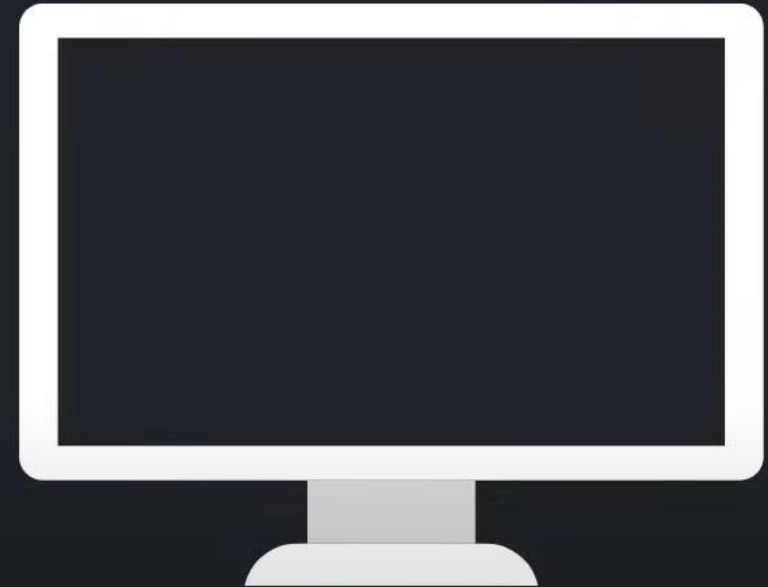
Update

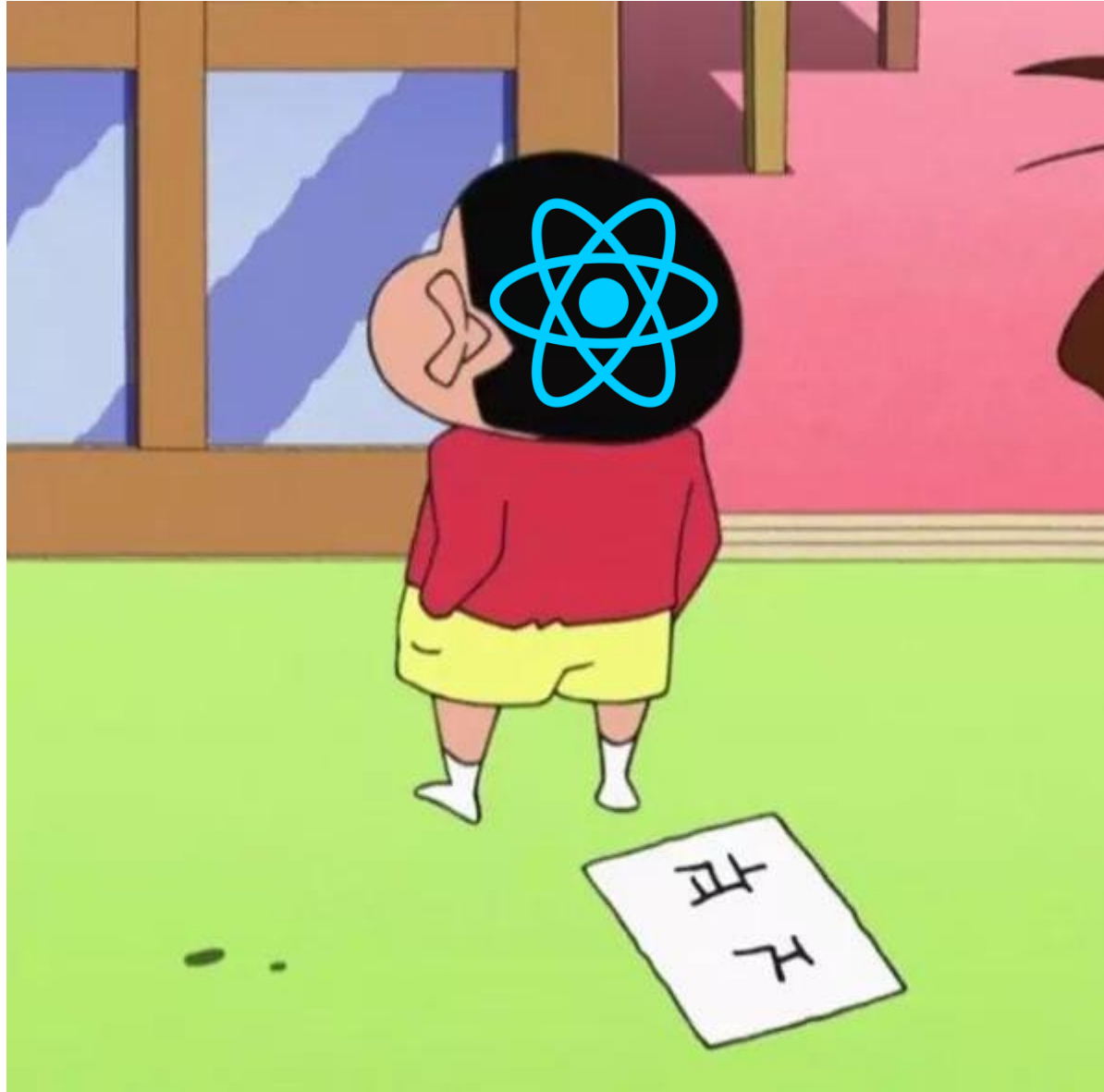
다시 렌더링



Unmount

화면에서 사라질때





Mount

화면에 첫 렌더링



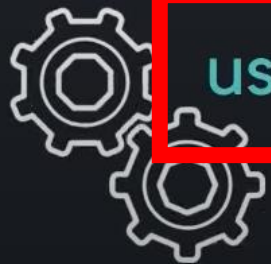
Update

다시 렌더링



Unmount

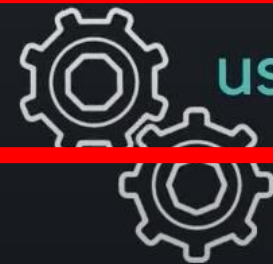
화면에서 사라질때



useEffect



useEffect



useEffect



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 ...  
    }  
}, [ ] );
```

```
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(() => {  
    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

Clean-Up

Clean Up - 정리



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



```
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



지금 시작합니다



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!



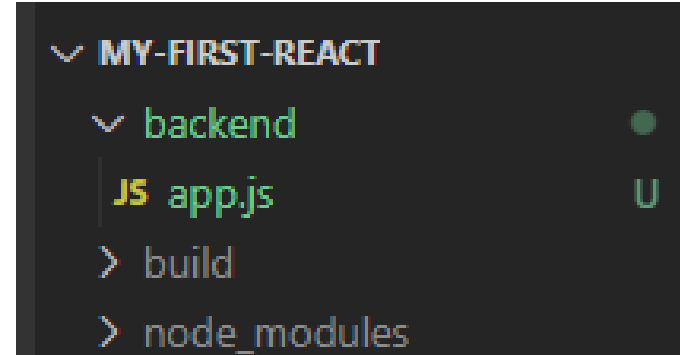
간단한 백엔드 서버 구성하기!

- 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.status(200).send(JSON.stringify(pororoObjArr));
});

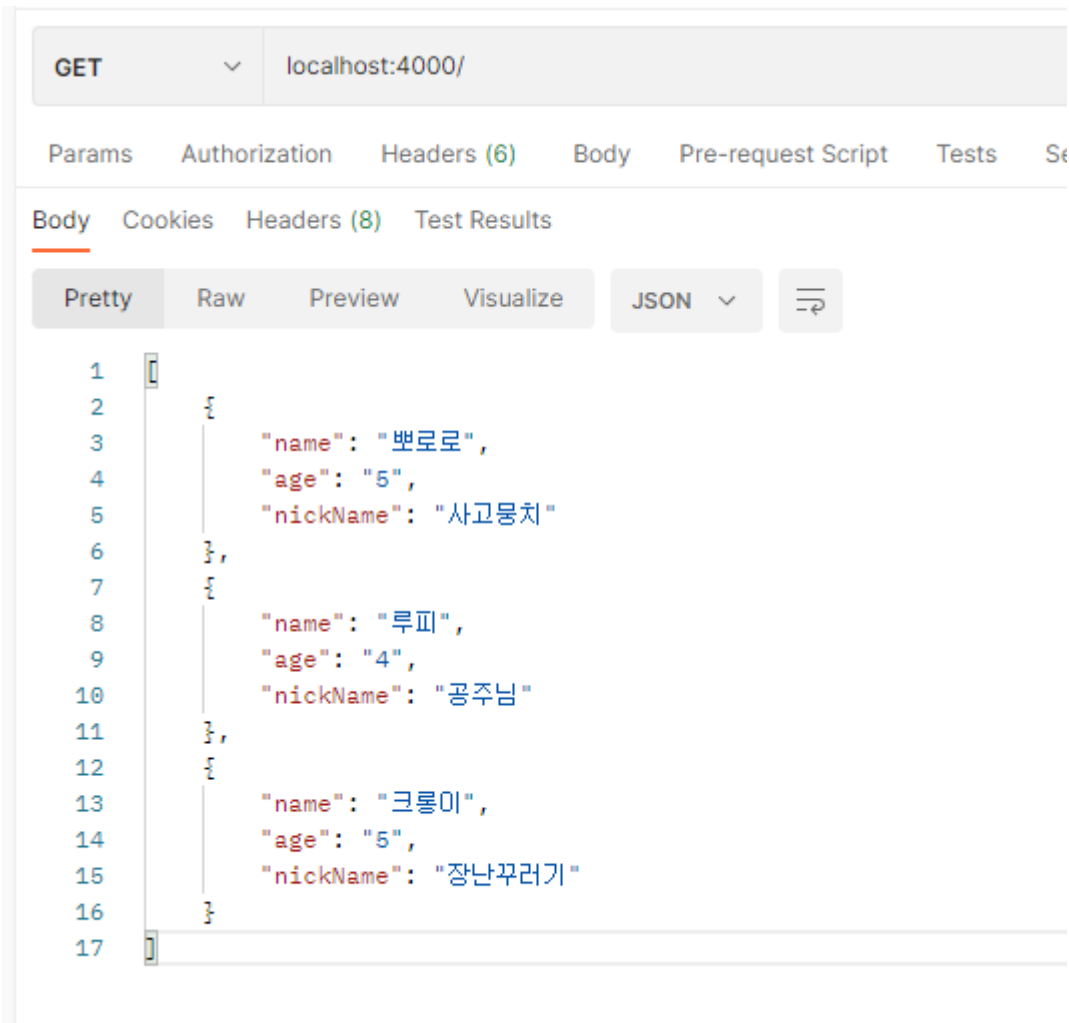
app.listen(PORT, () => {
  console.log(`데이터 통신 서버가 ${PORT}에서 작동 중입니다!`);
});
```

```
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",
      },
    });
    if (resFetch.status !== 200) return alert('통신 에러');

    const data = await resFetch.json();
    setDataArr(data);
    console.log(data);
  }

  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) [{...}, {...}, {...}]
>
```




useState
useRef
useEffect



컴포넌트 꾸미기



인라인으로 꾸미기!



인라인 스타일을 바로 적용

- JSX 문법을 통해 style 속성을 지정 가능하므로 스타일 속성 값을 객체로 선언한 다음 직접 삽입하는 방법도 가능합니다!
- 다만 아무래도 불편함이 많아서 실제로는 급한 상황이 아니면 사용하지 않습니다!



```
const divStyle = {  
  backgroundColor: "orange"  
}
```

src/style/InlineCss.jsx

```
const headingStyle = {  
  color: "blue"  
}
```

```
const spanStyle = {  
  backgroundColor: "pink",  
  fontWeight: "700"  
}
```

```
export default function InlineCss() {  
  return (  
    <div className="component-root" style={divStyle}>  
      <h1 style={headingStyle}>CSS 테스트 컴포넌트 입니다</h1>  
      <span style={spanStyle}>해당 컴포넌트를 기본 CSS로 꾸며 보아요!</span>  
    </div>  
  )  
}
```

CSS 테스트 컴포넌트 입니다

해당 컴포넌트를 기본 CSS로 꾸며 보아요!



기본 CSS로 꾸미기



기본 CSS 로 꾸미기!

- 컴포넌트의 결과물은 결국 html 코드이기 때문에 CSS 로 꾸미기가 가능합니다!
- 보통, src 폴더에 컴포넌트와 같은 이름의 css 파일을 만들어서 사용합니다!
- Src 폴더에 style 폴더를 만들고 거기에 컴포넌트와 같은 이름의 css 파일을 만들어 줍니다!
- 그리고 컴포넌트에 css 파일을 импорт 시켜주시면 됩니다!



```
div.component-root {  
  background-color: orange;  
}
```

```
div.component-root h1 {  
  color: red;  
}
```

```
div.component-root span {  
  background-color: white;  
  font-weight: 700;  
}
```

src/style/TestCss.css



```
import '../style/TestCss.css';

export default function TestCss() {
  return (
    <div className="component-root">
      <h1>CSS 테스트 컴포넌트 입니다</h1>
      <span>해당 컴포넌트를 기본 CSS로 꾸며 보아요!</span>
    </div>
  )
}
```

src/components/TestCss.jsx

CSS 테스트 컴포넌트 입니다

해당 컴포넌트를 기본 CSS로 꾸며 보아요!



Styled Components



styled
components



Styled Components

- 리엑트는 기본적으로 컴포넌트 단위가 중심이 되는 구조입니다
- 컴포넌트는 상대적으로 작은 규모로 운영이 되기 때문에 기존의 방식처럼 css 파일을 분리해서 운영할 필요가 크지 않습니다!
- 따라서, 이전에 `<style>` 태그를 사용해서 css 를 썼던 것 처럼 사용하고 싶은 요구가 많았습니다!
- 그래서 탄생한 것이 Styled Components 입니다!



Styled Components 설치 및 불러오기

- 먼저 설치 부터 하시죠!
- Npm install styled-components
- 그리고 TestStyled.jsx 파일 만들기
- Styled 컴포넌트 모듈 불러오기

```
import styled from "styled-components";
```



Styled Components 사용하기

- Styled Components 는 독특하게 사용이 됩니다!

```
export default function TestStyled() {  
  return (  
    <MyDiv>  
      <MyHeading>h1 태그 입니다!</MyHeading>  
      <MySpan>span 태그 입니다!</MySpan>  
    </MyDiv>  
  )  
}
```

src/components/TestStyled.jsx

- 먼저 자기만의 이름으로 태그를 구성 합니다!



Styled Components 사용하기

- 각각의 태그를 변수에 할당하고 해당 태그의 실제적인 태그명을 styled 를 이용하여 지정합니다!

```
const MyDiv = styled.div`  
  background-color: orange;  
`;  
;
```

```
const MyHeading = styled.h1`  
  color: blue;  
`;
```

```
const MySpan = styled.span`  
  background-color: pink;  
  font-weight: 700;  
`;
```

src/components/TestStyled.jsx



Styled Components 사용하기

- 자신이 지정한 태그의 이름은 일종의 빈 태그로 만들어 지지만 styled 모듈을 사용하여 해당 태그의 실질적 태그를 지정할 수 있습니다
- 그리고 CSS 요소는 뒤에 따라오는 `` 백틱 문자의 내부에 써주시면 됩니다!



요소 보기



Styled Components 작동

- Styled Components 는 html 태그의 class 명을 기반으로 작동하며, 해당 class 명은 임의로 생성 됩니다

```
▼<div class="App">
  ▼<div class="sc-bczRLJ jkucQt">
    <h1 class="sc-gsnTZi b0qcB0">h1 태그 입니다!</h1>
    <span class="sc-dkzDqf jmLKCS">span 태그 입니다!</span>
  </div>
```



불편함 해결



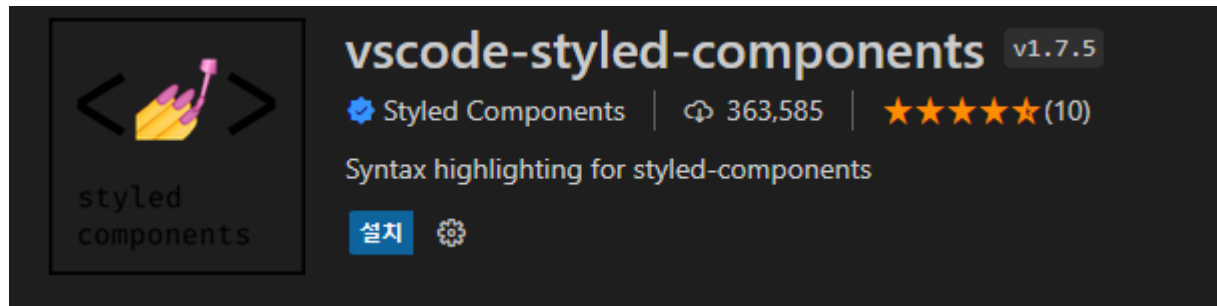
Styled Components 사용의 불편함

- Styled Components 사용시 CSS 를 입력하는 부분을 백틱 문자(` `) 내부에 작성을 해야합니다.
- 하지만 해당 부분은 JS에서 문자열로 취급 받기 때문에 CSS 의 스니펫을 활용할 수 없습니다!
- VS-Code 의 확장 프로그램을 설치하여 불편함을 해결 합니다!



Styled Components 사용의 불편함

- 확장 프로그램의 검색 창에 vscode-styled-components 입력

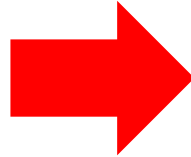


- 위의 확장 설치!

Styled Components 문법 표기



```
const MyDiv = styled.div`  
  background-color: orange;  
`;  
  
const MyHeading = styled.h1`  
  color: blue;  
`;  
  
const MySpan = styled.span`  
  background-color: pink;  
  font-weight: 700;  
`;
```



```
const MyDiv = styled.div`  
  background-color: orange;  
`;  
  
const MyHeading = styled.h1`  
  color: blue;  
`;  
  
const MySpan = styled.span`  
  background-color: pink;  
  font-weight: 700;  
`;
```

Styled Components snippet 기능!



```
const MyDiv = styled.div`  
  background-color: orange;  
  color: do;  
  x  
const MyHeadi  
  color: bl  
const MySpan  
  backgroun  
  font-weight: 700;  
  dodgerblue  
  darkolivegreen  
  darkorange  
  darkorchid  
  darkgoldenrod  
  darkviolet  
  darksalmon  
  darkturquoise
```





실습, React 초기 페이지를 Styled 로!

- 리액트 앱을 최초 생성 시, 만들어지는 app.js 컴포넌트를 App.css 가 아닌 Styled Components 를 이용해서 구현하기!
- 아래에 주어진 App.js 기본 코드와 Styled Components 를 위해 변경 된 코드를 비교하여 Styled Components 의 스타일을 적용시켜 주세요!
- 추가, 그림이 회전하는 keyframes 를 Styled Components 에서 적용하는 방법은 검색을 통해 구현해 주세요!



기본 App.js

```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

```
import logo from './logo.svg';

function App() {
  return (
    <RootDiv>
      <AppHeader>
        <AppLogo src={logo} alt="app" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <MyA
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </MyA>
      </AppHeader>
    </RootDiv>
  );
}

export default App;
```

Styled 적용을 위해 변경 된 App.js



```
.App {
  text-align: center;
}

.App-logo {
  height: 40vmin;
  pointer-events: none;
}

@media (prefers-reduced-motion: no-preference) {
  .App-logo {
    animation: App-logo-spin infinite 20s linear;
  }
}

.App-header {
  background-color: #282c34;
  min-height: 100vh;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  font-size: calc(10px + 2vmin);
  color: white;
}

.App-link {
  color: #61dafb;
}
```

```
@keyframes App-logo-spin {
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(360deg);
  }
}
```

App.css 코드



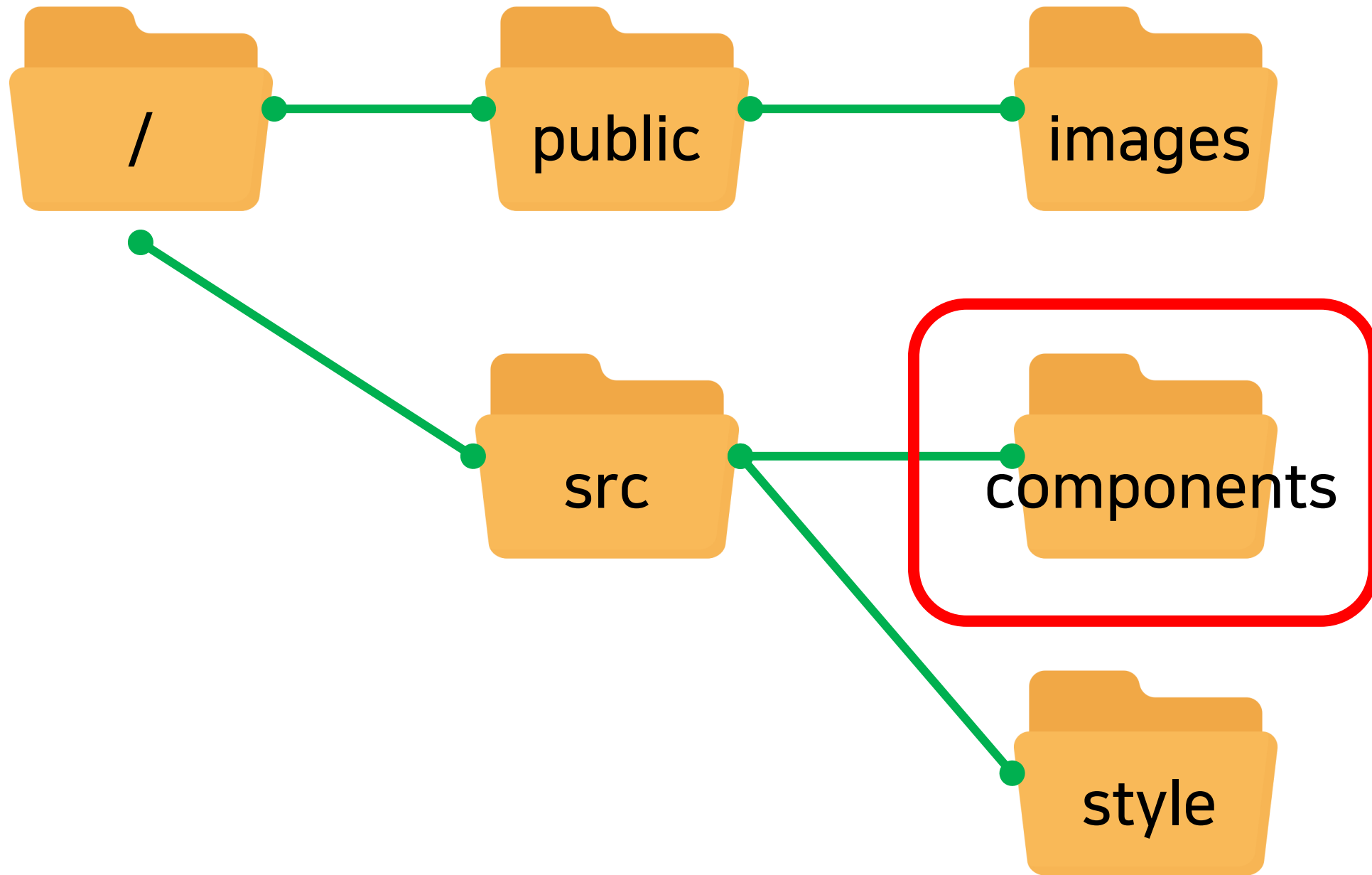
Public

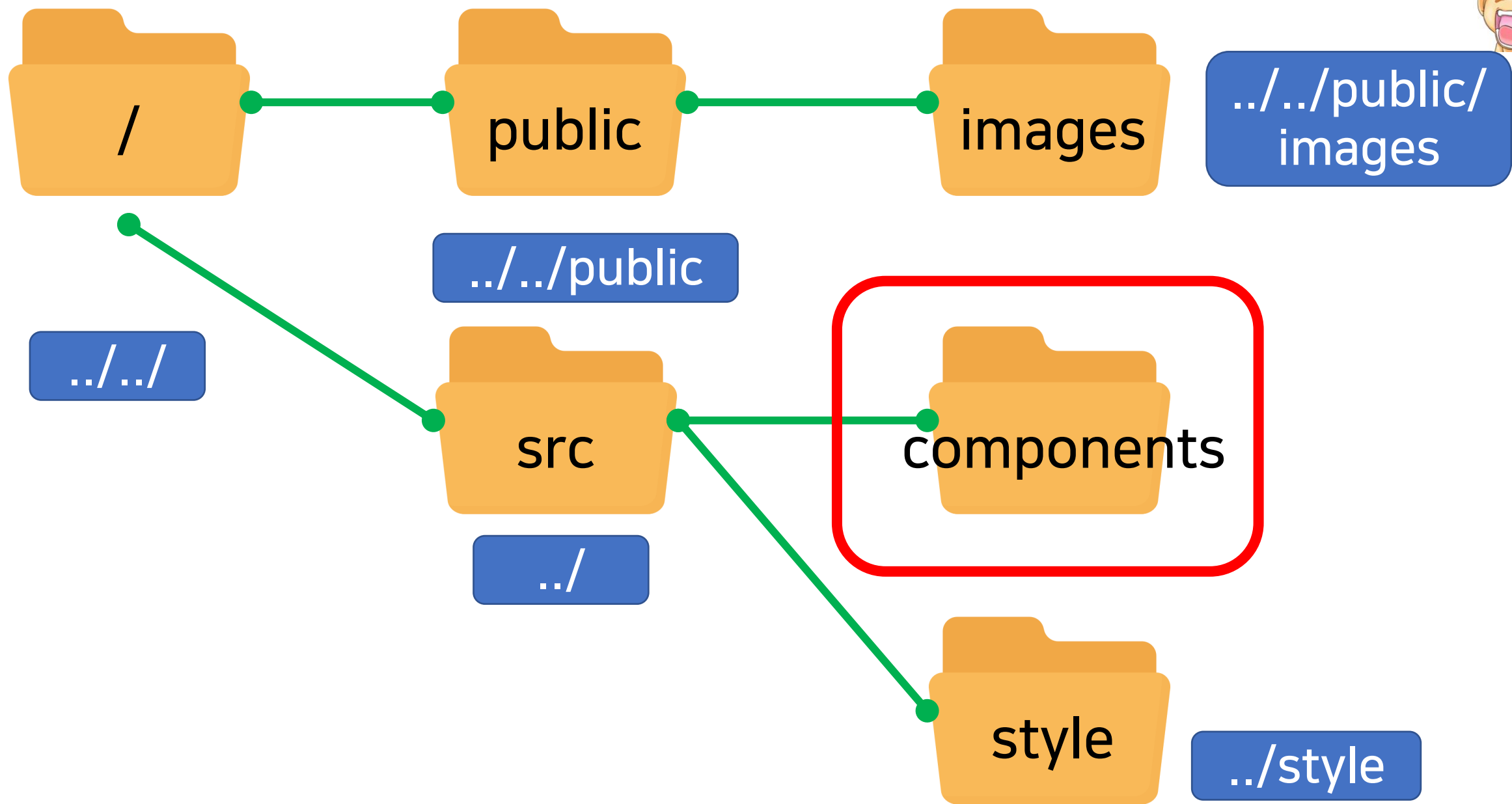
폴더 사용



퍼블릭 폴더로 접근하기!

- 기존 Backend 에서는 public 폴더를 static 이라는 Express 메소드를 사용해서 특정 주소 값을 요청하면 바로 public 폴더로 연결 해줬습니다!
- Public 폴더에 **/images** 라는 폴더를 만들고 원하는 사진 한 장을 넣기!
- 간단하게 **<Image>** 라는 컴포넌트를 만들고 public 폴더에 있는 이미지 파일에 접근해 봅시다!







상대 경로로 과연 가질까요!?

- 위에서 표현한 접근 방법으로 표시해서 접근해 봅시다!

```
import dogImg from "../../public/images/dog.jpg"

export default function Image() {
  return (
    <>
      <img src={dogImg} alt="강아지" />
    </>
  )
}
```

src/components/Image.js



Compiled with problems:

ERROR in ./src/components/test/Image.js 4:0-52

Module not found: Error: You attempted to import ../../../../public/images/dog.jpg which falls outside of the project src/ directory. Relative imports outside of src/ are not supported.
You can either move it inside src/, or add a symlink to it from project's node_modules/.

- Error 를 읽어보니 컴포넌트에서 다른 폴더에 접근을 하려고 해도 src 폴더 밖으로 나가는 것은 지원하지 않는다고 하네요
- 그럼, 백엔드 처럼 static 설정하면 되겠죠?



잠깐! 그럼 절대 경로는!?

```
import dogImg from "/"

export default function() {
  return (
    <img src=
  )
}
```

- \$Recycle.Bin
- \$WINDOWS.~BT
- \$WinREAgent
- \$Windows.~WS
- Documents and Settings
- ESD
- Microsoft VS Code
- PerfLogs
- Program Files
- Program Files (x86)
- ProgramData
- Recovery

- 진짜 컴퓨터의 Root 경로가 뜨네요
- 이건 아무리 봐도 답이 없겠죠?
- 그럼 Static 설정을 해봅시다!





폐북의 가호

- `Npx create-react-app` 을 통해 만들어진 리액트 앱의 경우는 public 폴더가 자동으로 static 처리가 됩니다!
- 따라서, 어느 위치에서건 `/` 를 써서 접근하면 public 폴더가 호출 됩니다!
- 그럼 강아지 사진의 주소 값을 `/images/dog.jpg` 로 변경해 봅시다!



```
export default function Image() {  
  return (  
    <>  
      
    </>  
  )  
}
```

src/components/Image.js





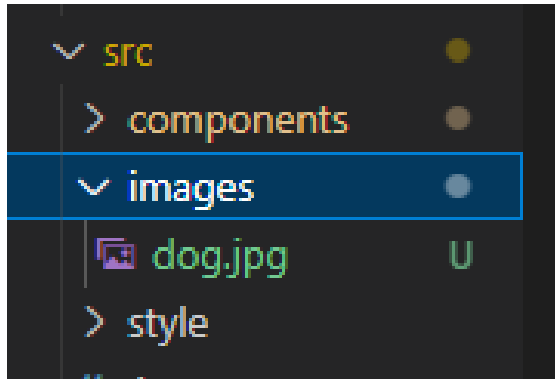
예전에 말씀드린 보안적 측면!

- 주소를 통해 폴더 구조가 드러나는 경우를 막기 위해 static 처리를 하고 있습니다!
- 상대 경로를 이용해서 src 폴더 이상으로 가서 다른 폴더에 접근해서 다시 내려가는 방식은 막힙니다!
- 또는, 절대 경로를 이용하는 방법 역시도 막힙니다!



하지만 src 폴더 이내라면!?

- 아무래도 src 폴더 내부에서는 서로가 서로를 참조하는 일이 비일비재 할 것 이다보니 아까 경고문에서도 src 외부로 나가는 것을 지원하지 않는다고 한 것 같네요!
- 그럼, 강아지 사진을 `src/images` 폴더로 옮기고 테스트 해봅시다!



```
import dogImg from "../images/dog.jpg"

export default function Image() {
  return (
    <>
      <img src={dogImg} alt="강아지" />
    </>
  )
}
```

src/components/Image.js

src 폴더 내부 참조는 문제 없습니다!





Props.children



Props.children

- Props 로 값을 보내는 것은 이미 배우셨습니다!
- 다만, 이미 정해진 props에 전달하는 값이 아닌 상황에 따라 html 요소 또는 컴포넌트 자체를 보내고 싶을 땐 어떻게 하면 될까요?
- 이럴 때에는 컴포넌트의 자식 요소를 한꺼번에 전달해 주는 `props.children` 을 사용할 수 있습니다.



```
React.createElement(  
  type,  
  [props],  
  [...children]  
);
```



```
1 function WelcomeDialog(props) {  
2   return (  
3     <FancyBorder color="blue">  
4       <h1 className="Dialog-title">  
5         어서오세요  
6       </h1>  
7       <p className="Dialog-message">  
8         우리 사이트에 방문하신 것을 환영합니다!  
9       </p>  
10    </FancyBorder>  
11  );  
12 }
```

Props

Props.children



코드로 확인해 봅시다!

- 전달 받은 요소를 전달 받은 color 의 border 로 감싸는 `<FancyBorder>` 라는 컴포넌트를 작성하여 봅시다!
- 그리고 App.js 에서 해당 컴포넌트가 감쌀 요소를 `props.children` 으로 한 꺼번에 전달해 봅시다!



```
export default function FancyBorder(props) {  
  return (  
    <div style={{ border: `3px solid ${props.color}` }}>  
      {props.children}  
    </div>  
  );  
}
```

src/components/FancyBorder.js

```
import FancyBorder from "../components/FancyBorder";  
  
function App() {  
  return (  
    <div className="App">  
      <FancyBorder color="blue">  
        <h1>Hello, props.children</h1>  
        <p>이건 매우 유용한 기술입니다요!</p>  
      </FancyBorder>  
    </div>  
  );  
}
```

src/App.js



Hello, props.children

이건 매우 유용한 기술입니다요!



```
export default function FancyBorder(props) {  
  return (  
    <div style={{ border: `3px solid ${props.color}` }}>  
      {props.children}  
    </div>  
  );  
}
```

src/components/FancyBorder.js

```
import FancyBorder from "../components/FancyBorder";  
  
function App() {  
  return (  
    <div className="App">  
      <FancyBorder color="blue">  
        <h1>Helle, props.children</h1>  
        <p>It's so convinient tools for us</p>  
      </FancyBorder>  
    </div>  
  );  
}
```

src/App.js

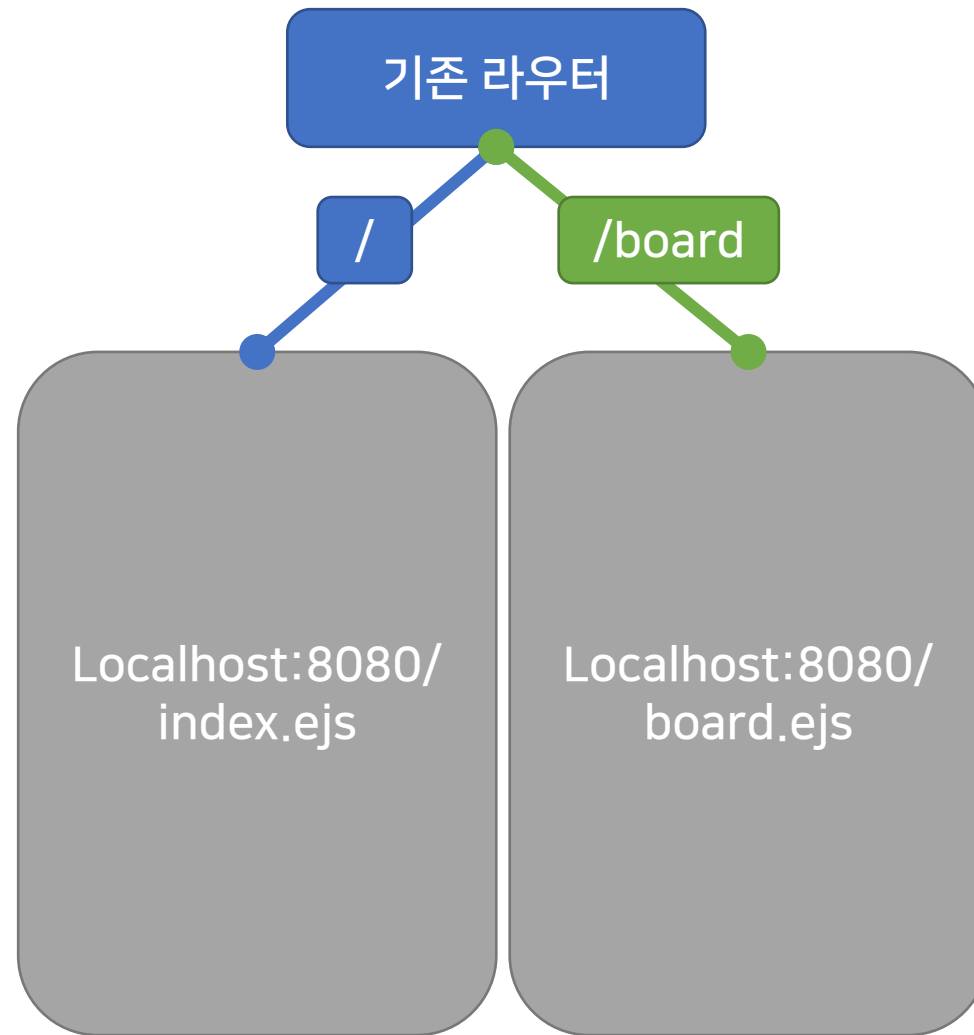


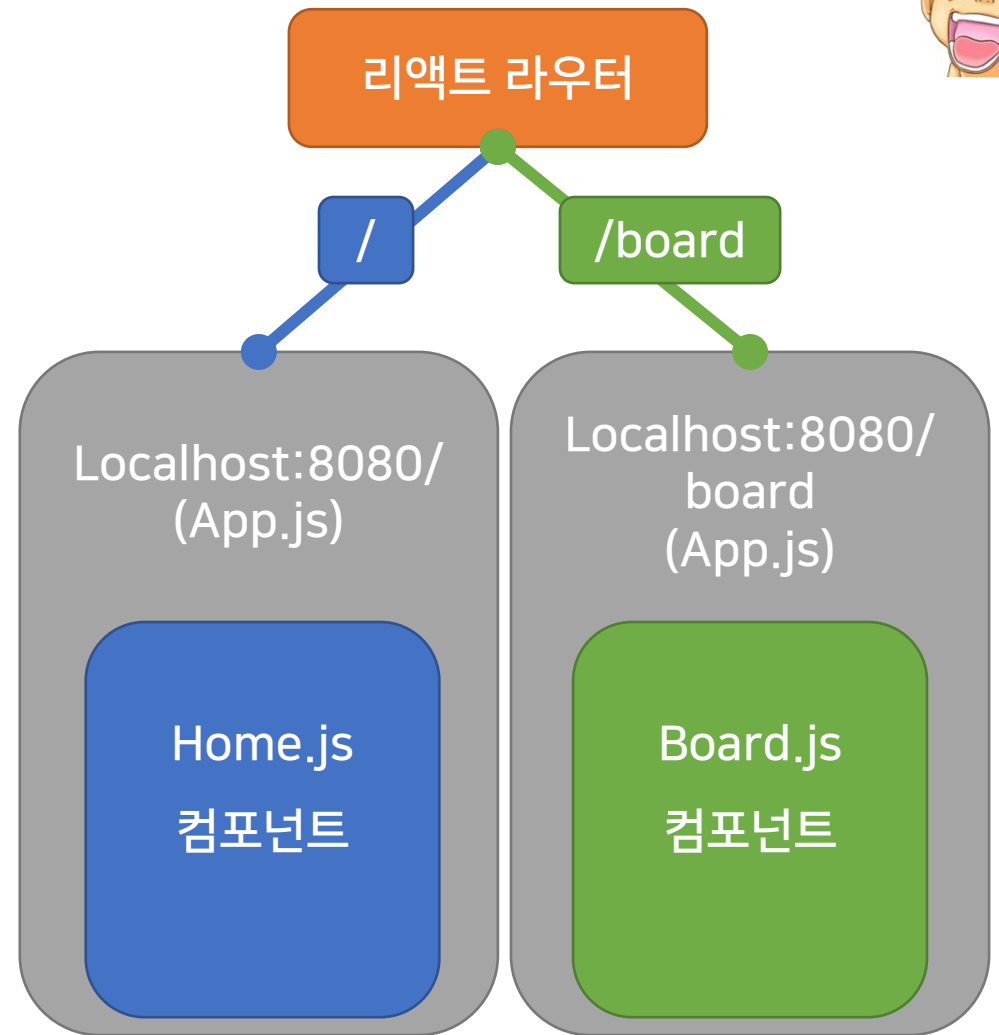
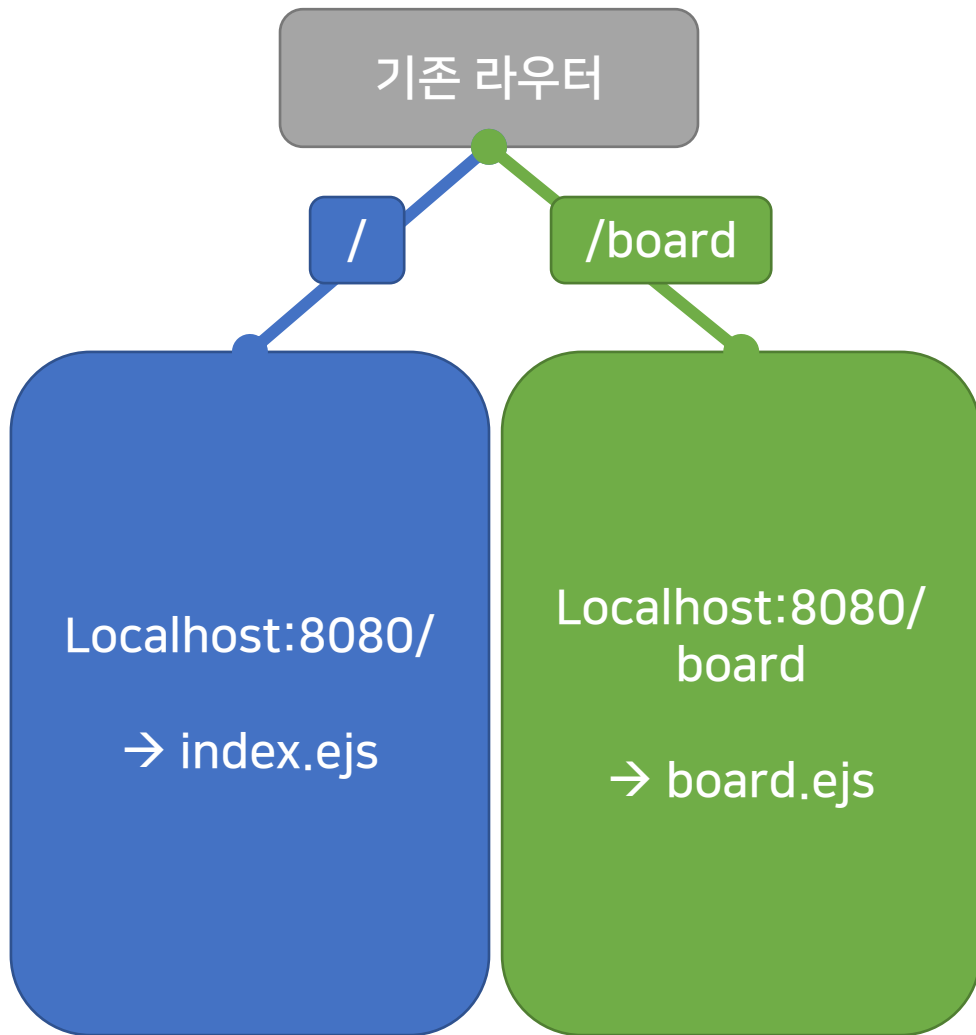
React Router



React 에서 Router 활용하기

- 지금까지 Router 는 입력받은 주소에 따라 페이지를 변경해주는 역할로 많이 사용이 되었습니다.
- 하지만, React 에서는 컴포넌트별(DOM) 라우팅이 가능합니다!
- 조건부 렌더링으로 처리가 가능하지만 서비스의 경우 주소에 따른 구분을 해주어야만 서비스별 구분이 가능하므로 라우팅 기능을 활용이 필요 합니다!
- 그리고 해당 모듈을 쓰면 페이지 깜박임 없이 부드러운 브라우징 가능







React Router 모듈 설치 및 적용

- Npm install react-router-dom
- 그리고 index.js 컴포넌트로 가서 App 컴포넌트를 `<BrowserRouter>` 라는 react-router-dom 이라는 컴포넌트로 감싸 주세요!
- `<BrowserRouter>` 로 감싸 주어야만 `<App>` 컴포넌트에서 발생하는 주소 값의 변경을 감지 할 수 있습니다
- 그외의 라우터로는 `<HashRouter>` 가 유명하며 주소의 해시 주소 `localhost:3000/#hash` 를 감지할 수 있는 라우터 입니다!



```
import { BrowserRouter } from 'react-router-dom';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </>
);

reportWebVitals();
```

Src/index.js



React Router 용 컴포넌트 만들기

- 라우팅 구현을 위해 h1 태그로만 구성 된 간단한 컴포넌트 2개(Board, Profile)를 만들어 봅시다!



```
export default function Profile() {  
  return (  
    <>  
      <h1>프로필 페이지 입니다</h1>  
    </>  
  )  
}
```

Src/component/Profile.jsx

```
export default function Board() {  
  return (  
    <>  
      <h1>게시판 페이지 입니다</h1>  
    </>  
  )  
}
```

Src/component/Board.jsx

App.js 에 Nav 메뉴 구성



- 간단하게 페이지 이동이 가능한 Nav 메뉴도 구현해 봅시다!



```
function App() {  
  return (  
    <div className="App">  
      <nav>  
        <ul>  
          <li>  
            <a href="/profile">프로필 페이지 이동</a>  
          </li>  
          <li>  
            <a href="/board">게시판 페이지 이동</a>  
          </li>  
        </ul>  
      </nav>  
    </div>  
    <Profile />  
    <Board />  
  );  
}
```

```
export default App;
```

Src/App.js



홈 페이지 이동
프로필 페이지 이동
게시판 페이지 이동

프로필 페이지 입니다

게시판 페이지 입니다

- 지금은 라우팅 구현이 안되어 모든 컴포넌트가 하나의 페이지에서 보이고 있습니다! → 이를 해결하기 위해 라우팅을 구현!



React Router 구현

- React-router-dom 의 Route 모듈을 App.js 에 추가하여 컴포넌트 라우팅을 구현해 봅시다!

```
import { Route } from "react-router-dom";
```



React Router 구현

- React Router 는 Routes 컴포넌트 내부에 Route 컴포넌트를 넣어주고 각각의 주소 값은 path 속성에, 호출할 컴포넌트는 element 속성으로 불러 주면 됩니다!

```
<Routes>  
  <Route path="/profile" element={<Profile />} />  
  <Route path="/board" element={<Board />} />  
</Routes>
```

Src/App.js

- 여기서 Route 는 반드시 Routes 내부에 있어야 합니다!



React Router 구현

- `<a>` 태그는 브라우저 레벨에서 페이지를 자동으로 새로고침 하기 때문에 React 에서는 이를 막고자 `<Link to="">` 라는 컴포넌트를 사용합니다!
- Link 컴포넌트는 html 상에서는 `<a>` 태그로 변경이 되지만 브라우저 새로고침 없이 주소만 변경해 주는 역할을 합니다!
- `<a>` 태그를 `<Link>` 컴포넌트로 변경!



```
<nav>
  <ul>
    <li>
      <Link to="/">홈 페이지 이동</Link>
    </li>
    <li>
      <Link to="/profile">프로필 페이지 이동</Link>
    </li>
    <li>
      <Link to="/board">게시판 페이지 이동</Link>
    </li>
  </ul>
</nav>
```

Src/App.js



```
import { Link, Outlet, Route, Routes } from "react-router-dom";
import Profile from "../components/Profile";
import Board from "../components/Board";
```

```
function App() {
  return (
    <div className="App">
      <nav>
        <ul>
          <li>
            <Link to="/">홈 페이지 이동</Link>
          </li>
          <li>
            <Link to="/profile">프로필 페이지 이동</Link>
          </li>
          <li>
            <Link to="/board">게시판 페이지 이동</Link>
          </li>
        </ul>
      </nav>
      <Routes>
        <Route path="/profile" element={<Profile />} />
        <Route path="/board" element={<Board />} />
      </Routes>
    </div>
  );
}
```

```
export default App;
```

전체 코드

Src/App.js



Localhost:3000/



[홈 페이지 이동](#)
[프로필 페이지 이동](#)
[게시판 페이지 이동](#)

Localhost:3000/profile



[홈 페이지 이동](#)
[프로필 페이지 이동](#)
[게시판 페이지 이동](#)

프로필 페이지 입니다

Localhost:3000/board



[홈 페이지 이동](#)
[프로필 페이지 이동](#)
[게시판 페이지 이동](#)

게시판 페이지 입니다



React Router

심화 활용



React Router 심화 활용

- 이번에는 `<Header>` 컴포넌트를 만들고 해당 `<Header>` 를 통한 라우팅 처리를 구현해 봅시다!
- 그 외에도 주소 예외 처리 및 주소의 parameter 사용에 대해서도 배워 봅시다!



Header 컴포넌트 작성

- 부드러운 브라우징을 위해 `<Link>` 컴포넌트 사용
- Header 에 맞게 `display: "flex"` 처리



```
import { Link } from "react-router-dom";
export default function Header() {
  return (
    <>
      <nav>
        <ul style={{ display: "flex", justifyContent: "space-around" }}>
          <li>
            <Link to="/">홈 페이지 이동</Link>
          </li>
          <li>
            <Link to="/profile">프로필 페이지 이동</Link>
          </li>
          <li>
            <Link to="/board">게시판 페이지 이동</Link>
          </li>
        </ul>
      </nav>
    </>
  );
}
```

Src/component/Header.js



App.js 에서의 라우팅 처리

- 그럼 **<Header>** 컴포넌트에서 작성한 주소에 맞게 라우팅 처리를 하고 각각의 컴포넌트를 제작해 봅시다!
- 메인 페이지는 **<Header>** 컴포넌트만 불러오면 되므로 **"/"** 주소에는 **<Header>** 컴포넌트만 부여
- 각각의 주소는 각각 주소에 맞는 컴포넌트 부여



```
import { Route, Routes } from "react-router-dom";
import Profile from "../components/Profile";
import Board from "../components/Board";
import Header from "../components/Header";
import NotFound from "../components/NotFound";

function App() {
  return (
    <div className="App">
      <Routes>
        <Route path="/" element={<Header />} />
        <Route path="profile" element={<Profile />} />
        <Route path="board" element={<Board />} />
      </Routes>
    </div>
  );
}

export default App;
```

Src/App.js



Profile, Board 컴포넌트 업데이트!

- 이제는 App.js 에서 컴포넌트를 부르는 방식이 아니라 해당 컴포넌트 자체가 그려지는 구조를 가지므로 각각 컴포넌트에도 `<Header>` 컴포넌트 추가 필요!



```
import Header from './Header'

export default function Profile() {
  return (
    <>
      <Header />
      <h1>프로필 페이지 입니다</h1>
    </>
  )
}
```

Src/component/Profile.js

```
import Header from './Header'

export default function Board() {
  return (
    <>
      <Header />
      <h1>게시판 페이지 입니다</h1>
    </>
  )
}
```

Src/component/Board.js

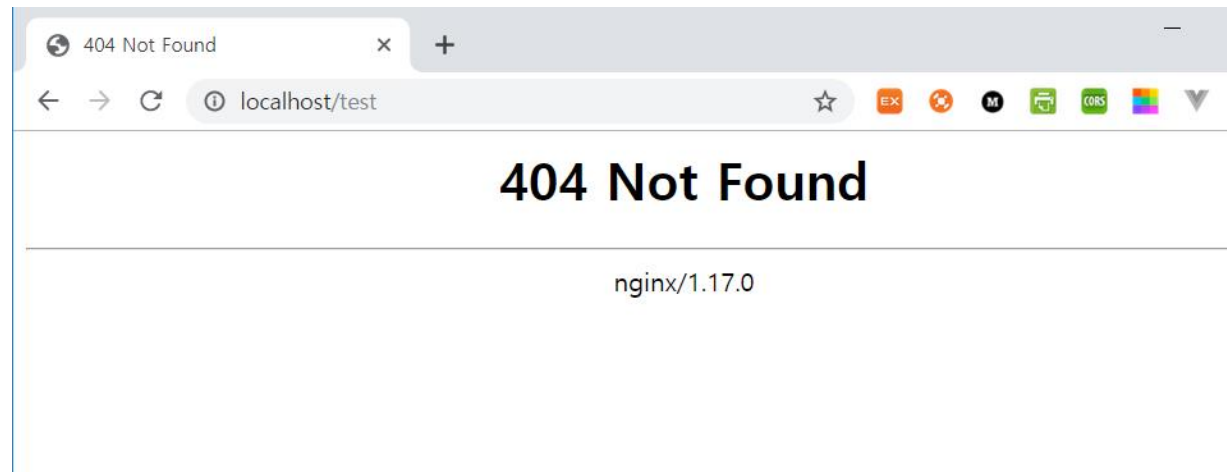


주소 예외 처리



Page Not Found

- 실제 서비스의 경우 사용자가 예상하지 못한 주소 값을 입력하는 경우가 발생 합니다.
- 이럴 때, 브라우저에서 제공하는 **404 Not Found** 페이지를 띄우면 일단 서버 응답을 기다리는 시간도 오래 걸릴 뿐더러, 결과 페이지가 서비스의 신뢰를 깨는 역할을 하게 됩니다!





Page Not Found

- 따라서 잘못 입력 된 주소에 대한 예외 처리가 필요합니다!
- React-router-dom 모듈은 해당 부분에 있어서 * 라는 편리한 방법을 제공합니다!
- * 는 모든 주소 입력을 의미하며 아래와 같이 사용합니다!

```
<Route path="*" element={<NotFound />} />
```



```
function App() {  
  return (  
    <div className="App">  
      <Routes>  
        <Route path="/" element={<Header />} />  
        <Route path="profile" element={<Profile />} />  
        <Route path="board" element={<Board />} />  
        <Route path="*" element={<NotFound />} />  
      </Routes>  
    </div>  
  );  
}  
  
export default App;
```

코드 처리 방향

Src/App.js



Page Not Found

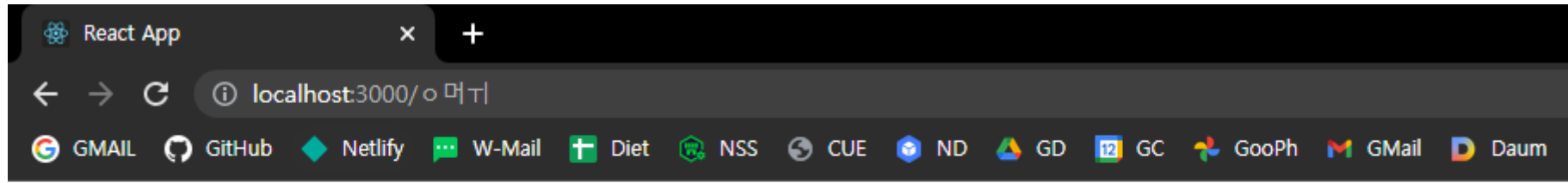
- React-router-dom 라우터도 백엔드 라우터와 마찬가지로 코드 선언 순서에 따라 처리가 됩니다!
- 위에서 주소 처리를 해도 일치가 되는 부분이 없으면 아래의 라우터로 내려오게 되는데 마지막 라우터에서 주소를 * 를 사용하여 처리 하면 일치가 안된 주소는 한꺼번에 처리가 가능합니다!
- 해당 라우터에서 <NotFound> 컴포넌트로 연결하여 모든 예외를 처리!



```
import Header from './Header'

export default function NotFound() {
  return (
    <>
      <Header />
      <h1>Page Not Found</h1>
    </>
  )
}
```

Src/component/NotFound.js



- [홈 페이지 이동](#)

- [프로필 페이지 이동](#)

- [게시판 페이지 이동](#)

Page NotFound



주소

Parameter 활용



주소의 Parameter 값 활용

- 백엔드에서 주소로 전달되는 Parameter 는 중요하게 사용이 됩니다!
- 물론 리액트 라우터에서도 두 가지 모두를 사용할 수 있습니다!
- <Board> 컴포넌트에 2개의 게시글이 있다고 가정하여 parameter 를 활용하여 봅시다!



Board 컴포넌트 변경

- <Board> 컴포넌트는 이제 2개의 게시글의 목록을 보여주고 해당 게시글로 이동하는 역할을 합니다!

```
import { Link, Route, Routes } from 'react-router-dom';
import Header from './Header'

export default function Board() {
  return (
    <>
      <Header />
      <h1>게시판 페이지 입니다</h1>
      <Link to="1"><h2>게시글 1번 보여주기</h2></Link>
      <Link to="2"><h2>게시글 2번 보여주기</h2></Link>
    </>
  )
}
```

Src/component/Board.js



주소 처리 방법, to??

- 어떤 주소는 / 로 시작하고 어떤 주소는 / 가 없이 시작하죠?
- / 로 시작
 - 앱의 기본 주소인 `Localhost:3000` 뒤에 / 뒤의 주소가 이어짐
 - `/profile` → `Localhost:3000/profile`



주소 처리 방법, to??

- / 없이 시작
 - 현재 라우팅 된 주소의 뒤에 해당 주소의 문자열이 추가 됨
 - 현재 라우팅 주소가 localhost:3000/board 일 때
 - 1 → localhost:3000/board/1



Route 에 Parameter 선언

- App.js 에 선언 된 라우터 선언부에 Parameter 를 선언해 봅시다!
- 기존과 같은 방법으로 주소/:parameter 로 선언하면 됩니다!

```
<Routes>
  <Route path="/" element={<Header />} />
  <Route path="profile" element={<Profile />} />
  <Route path="board" element={<Board />} />
  <Route path="board/:boardID" element={<BoardDetail />} />
  <Route path="*" element={<NotFound />} />
</Routes>
```

Src/App.js

BoardDetail 컴포넌트에서 parameter 받기



- 게시글 내용은 <BoardDetail> 컴포넌트에서 받으므로 해당 컴포넌트를 만들어 줍시다!
- Parameter 로 전달 받은 값도 받아서 활용해 줍시다!
- Parameter 는 useParams 로 받을 수 있으며, useParams 로 선언한 객체 변수에 담기게 됩니다
- parameter 로 선언한 이름이 Key 로 설정 됩니다!



```
import { useParams } from "react-router-dom"
import Header from "../Header";
```

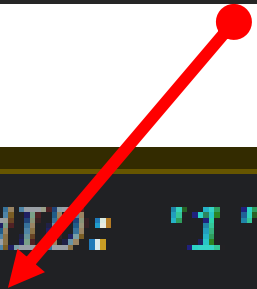
```
export default function BoardDetail() {
  const params = useParams();
  console.log(params);
  return (
    <>
      <Header />
      <h2>{params.boardID} 번 게시글 내용입니다!</h2>
    </>
  )
}
```

Src/component/BoardDetail.js



전달한 이름 그대로 객체의 Key 값이 배정

```
<Route path="board/:boardID" element={<BoardDetail />} />
```

A red arrow originates from the `boardID` prop in the code above and points to the `boardID` key in the object below.

```
▼ {boardID: '1'} ⓘ  
  boardID: "1"  
  ► [[Prototype]]: Object
```



구조 분해 할당 문법!? 가능!!

```
import { useParams } from "react-router-dom"
import Header from "../Header";

export default function BoardDetail() {
  const { boardID } = useParams();
  return (
    <>
      <Header />
      <h2>{boardID} 번 게시물 내용입니다!</h2>
    </>
  )
}
```

Src/component/BoardDetail.js



Redux



React 입문자들이 알아야할 Redux 쉽게설명 (8분컷)

조회수 5.7만회 · 1년 전



코딩애플

React 하다보면 Redux를 필히 만나게 되는데 **한 해 리덕스 포기자가 10만명이나 되기 때문에 준비했습니다** 리액트 강의 ...



React 입문자들이 알아야할 Redux 쉽게설명 (8분컷)

조회수 5.7만회 · 1년 전



코딩애플

React 하다보면 Redux를 필히 만나게 되는데 한 해 리덕스 포기자가 10만명이나 되기 때문에 준비했습니다 리액트 강의 ...







© Dave Stamboulis



Redux



<https://coinpan.com> 코인판 on 2018-01-09



코딩애플

구독자 10.2만명

<https://www.youtube.com/watch?v=QZcYz2NrDIs&t=212s>



그래서 Redux 가 뭐죠?

- Redux 는 상태 관리 라이브러리 입니다!
- R로 시작해서 React 랑만 쓰는 것 같지만, 상태 관리가 필요한 다른 프레임 워크(Angular.js / Vue.js / 심지어 JQuery) 에서도 사용이 가능합니다!
- 물론 Redux 하나를 전체가 공유하는 것은 아니고 Redux 의 개념을 각각의 프레임 워크에 맞춘 라이브러리를 사용합니다
- 따라서, 우리가 쓰는 건 React-Redux 입니다!

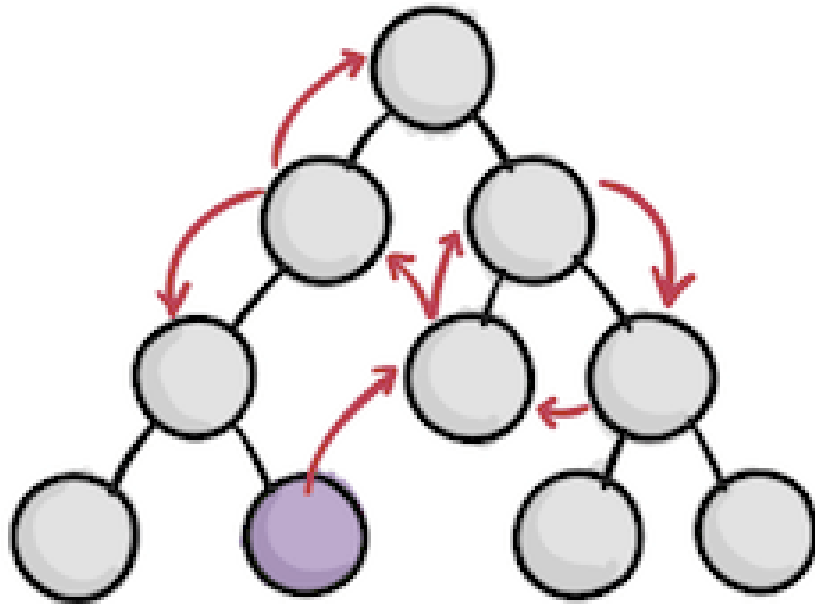


그래서 Redux 가 뭐죠?

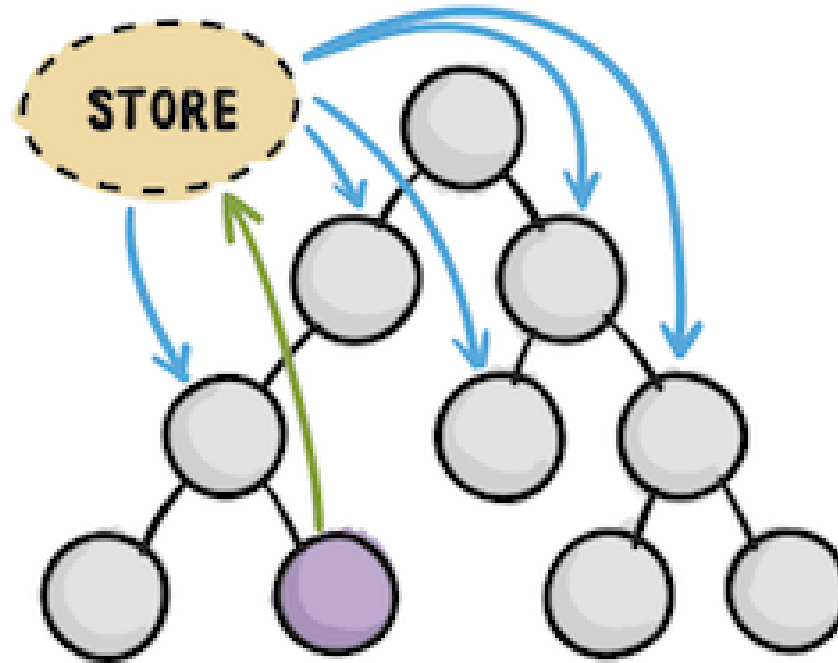
- 컴포넌트의 상태를 하나하나 Props 로 전달하면 너무 힘들기 때문에 이를 해결하고자 나온 라이브러리 입니다
- 컴포넌트의 상태를 각각 컴포넌트 별 State 에 따라 관리하는 것이 아닌 하나의 Store 라는 곳에서 관리 합니다!
- 따라서, 상태 변화 값을 중첩 된 컴포넌트 수 만큼 Props 로 전달하는 방식 이 아니라 Store 에서 한번에 꺼내서 사용하는 편리함을 제공합니다!
- 물론 그 편리함을 쓰려면 어려움이 발생하죠 ㅎㅎㅎ 😊



WITHOUT REDUX

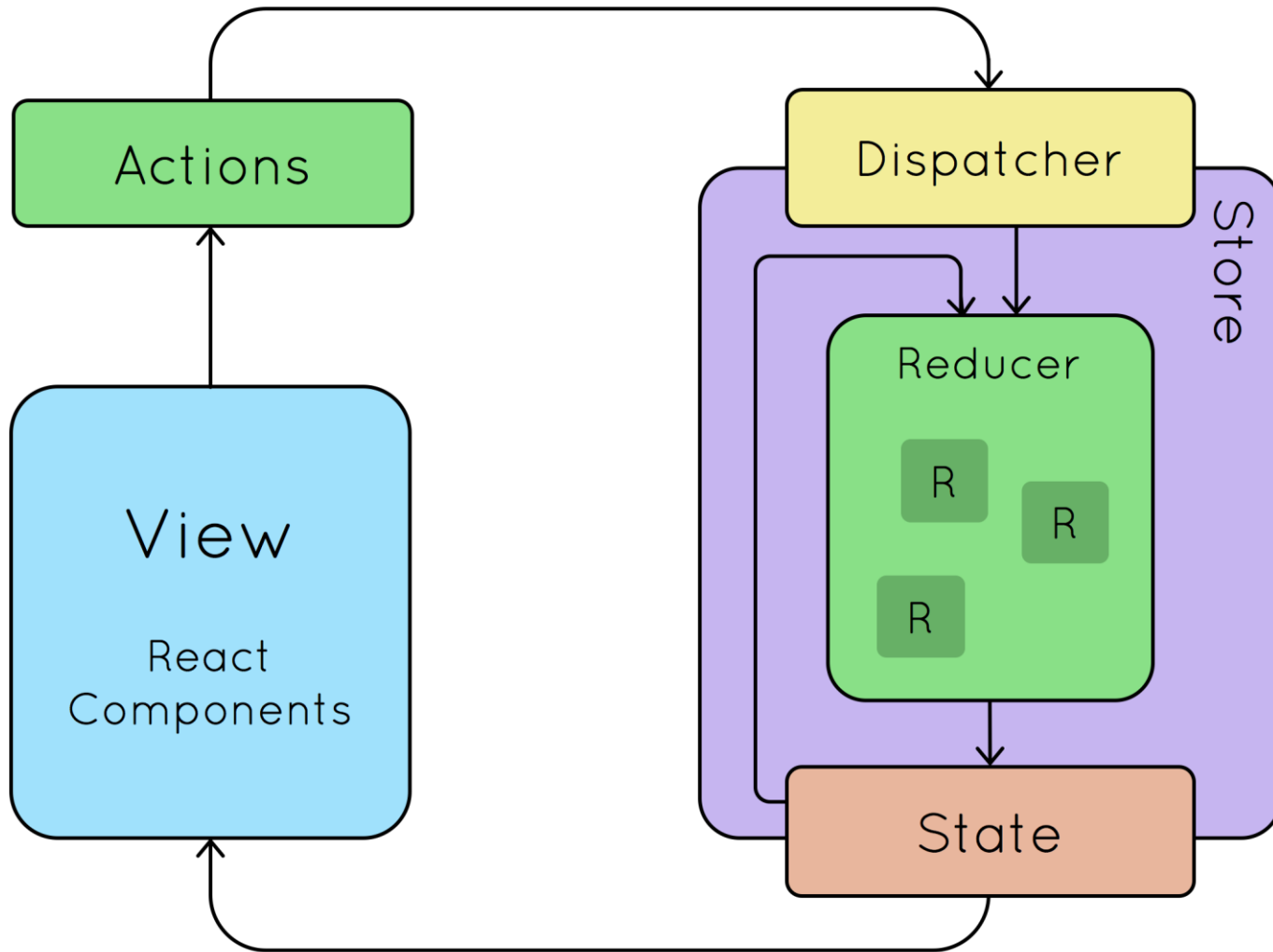


WITH REDUX



 COMPONENT INITIATING CHANGE

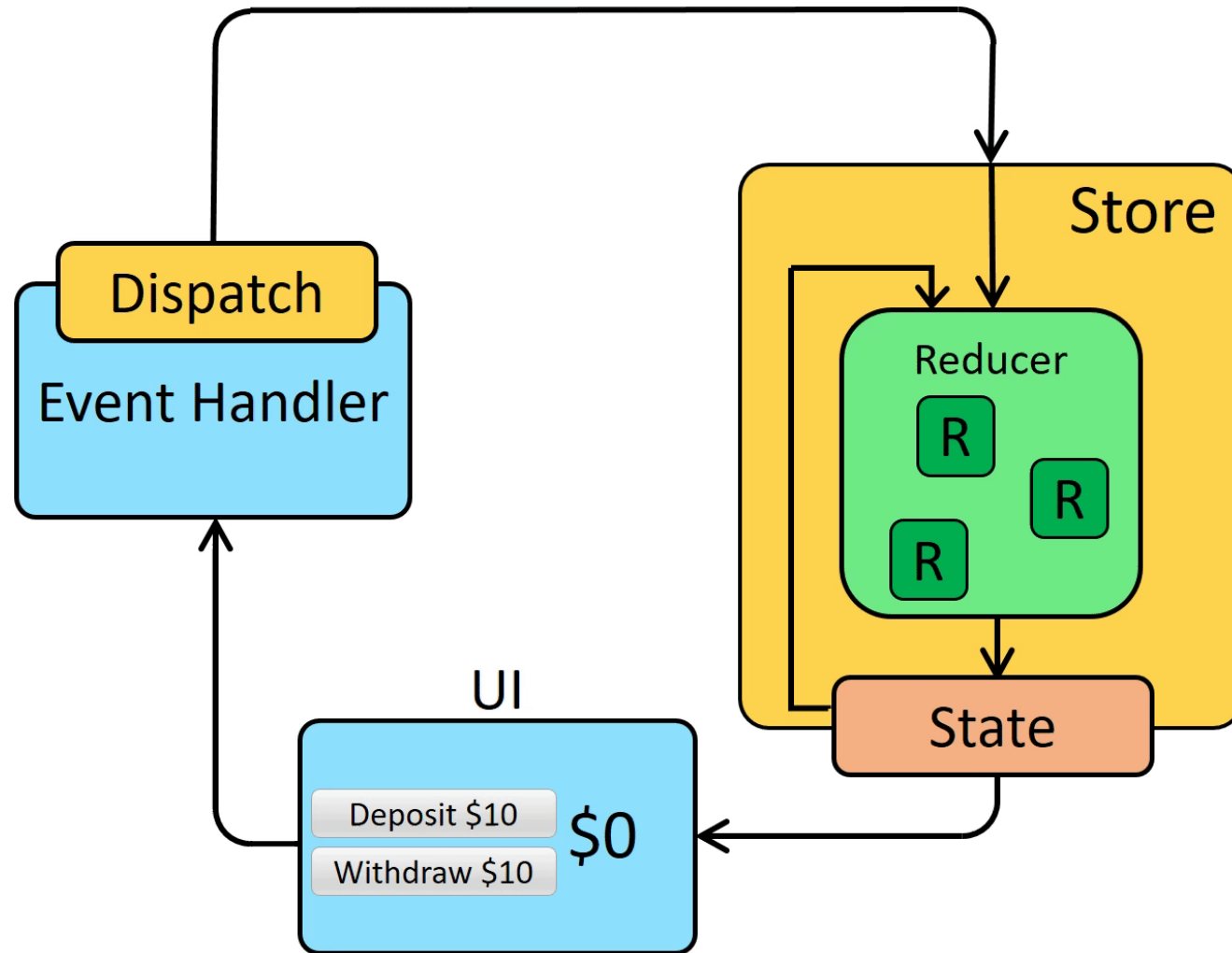
Redux 동작 순서





Redux 동작 순서

1. 디스패치(Dispatch) 함수를 실행하면
2. 액션(Action)이 발생합니다
3. 이 액션을 리듀서(Reducer)가 받아서
4. 상태(State)를 변경합니다
5. 상태가 변경 되면 컴포넌트를 리렌더링 됩니다!





코딩 애플 코드로 하나하나 확인하기



Redux 기초 세팅!

- Redux 를 위한 새로운 App 을 만듭시다!
- `Npx create-react-app redux-app`
- 앱 생성 후, redux 관련 모듈을 설치 합시다!
- `Npm install redux`
- `Npm install react-redux`



Redux 기초 세팅!

- 라우팅 처리 하던 것처럼! Redux 적용을 위해서는 `<Provider>` 컴포넌트를 임포트하고 해당 컴포넌트로 `<App>` 컴포넌트를 감싸줘야만 합니다!
- `Index.js` 에 가서 코드를 처리!

```
import { Provider } from 'react-redux';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <>
    <Provider>
      <App />
    </Provider>
  </>
);
```

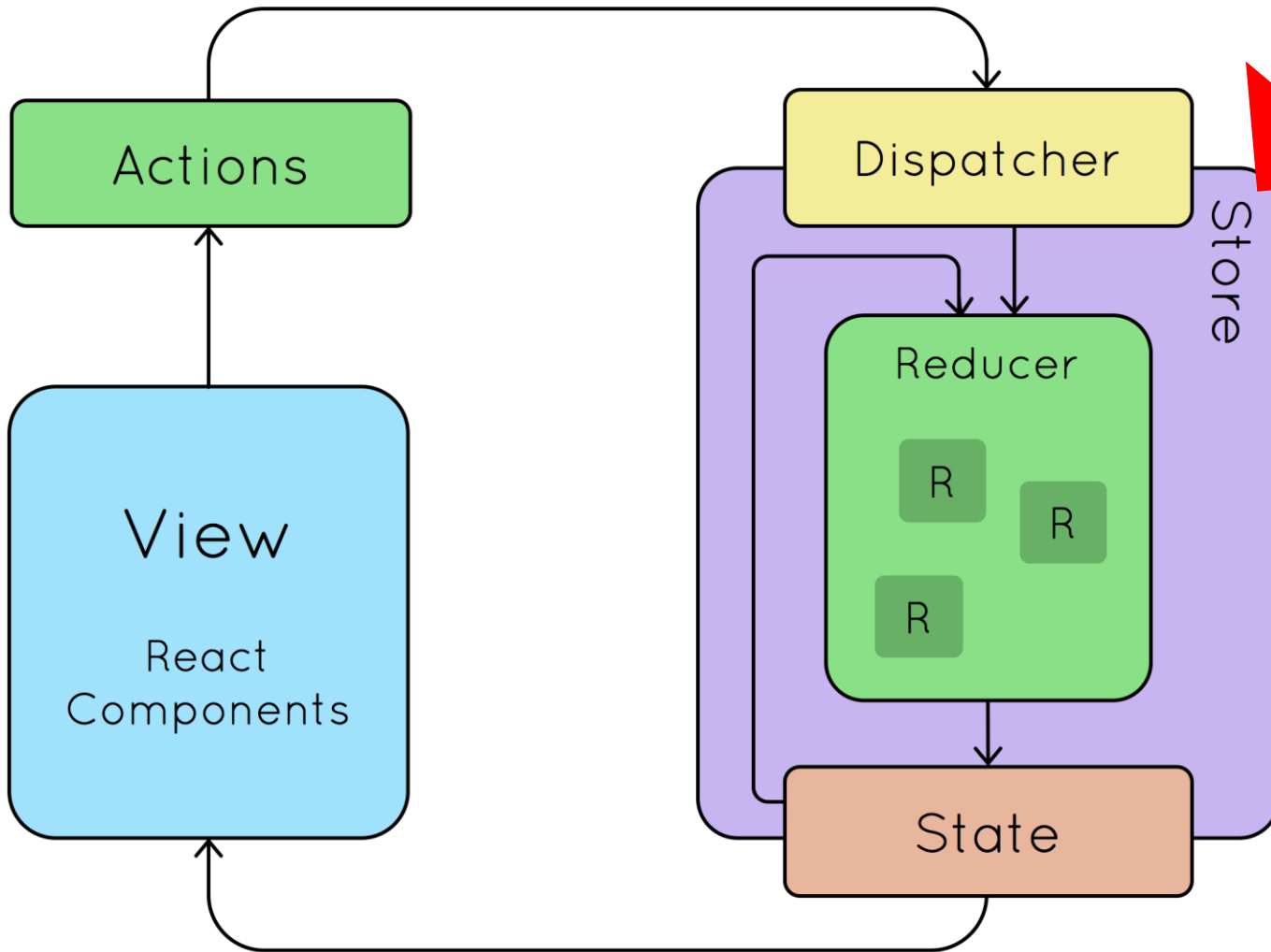
Src/index.js





Store

Store 만들기!



- 상태를 저장할 Store 부터 만들어 봅시다!



Store 만들기!

- Redux 에서 `createStore` 를 импорт 한 뒤, store 를 만들어 줍니다
- 그리고 `<Provider>` 컴포넌트에게 상태 관리를 할 store 속성에 만들어진 store 를 부여해 줍니다!



Store

앱에는 단 하나의 스토어가 존재
현재 상태, 리듀서가 포함



```
import { createStore } from 'redux';
```

```
let store = createStore(reducer);
```

```
const root = ReactDOM.createRoot(document.getElementById('root'));
```

```
root.render(  
  <>  
    <Provider store={store}>  
      <App />  
    </Provider>  
  </>  
);
```

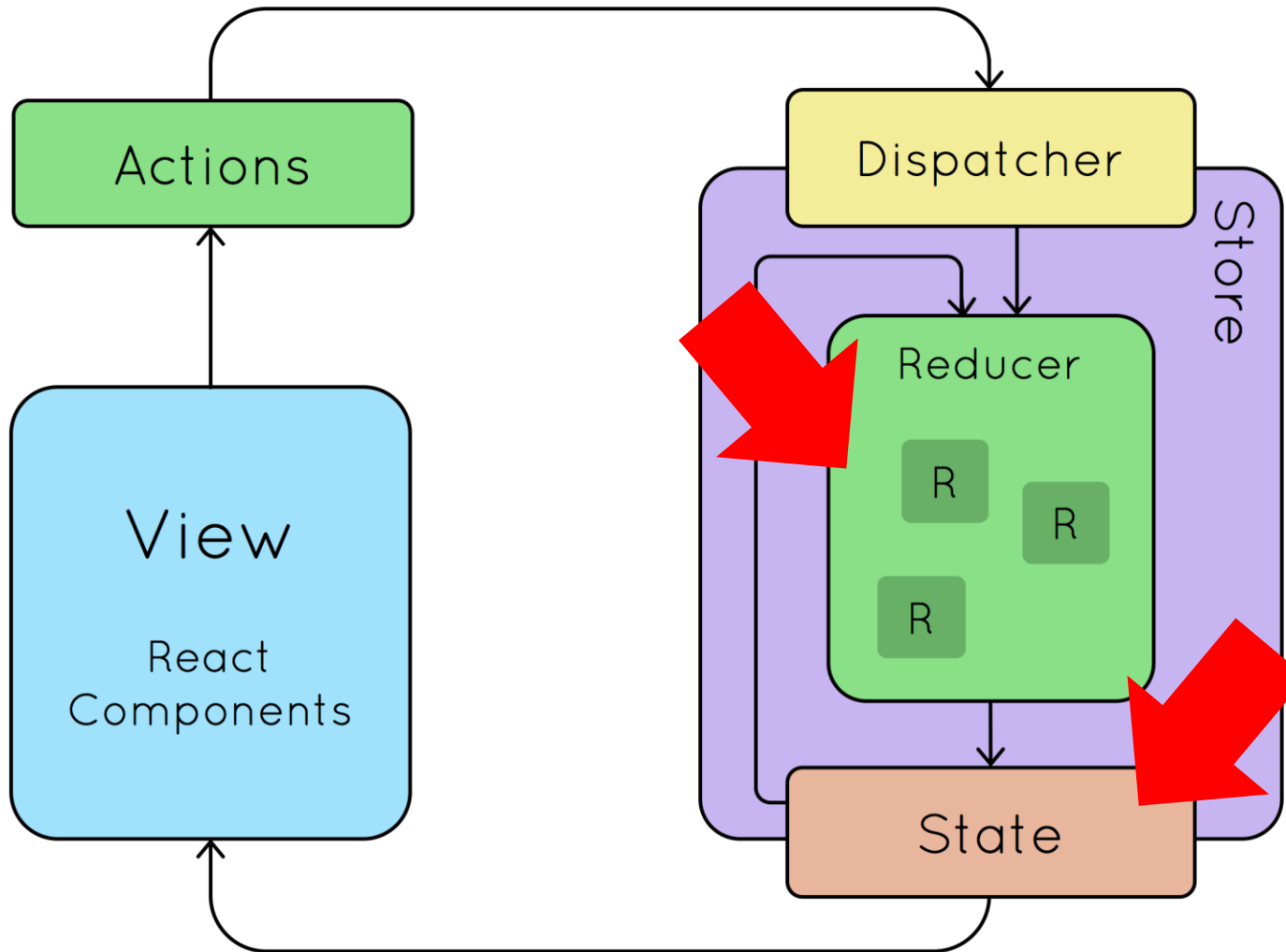
Src/index.js



State & Reducer



State 값 설정 및 Reducer 만들기!



- 상태 역할을 해줄 State 값 설정을 하고
- State 값을 변경해 줄 Reducer 를 만들어 봅시다!



State 설정

- State 는 하나의 변수 또는 객체를 사용하면 됩니다!
- 저희는 간단한 형태로 알아만 볼 것이기 때문에 변수 하나를 선언해서 사용해 봅시다!

```
const weight = 100;
```



Reducer 만들기!

- 실제로 State 값을 관리하는 Reducer 를 만들어 봅시다!
- 먼저 State 로 사용할 변수를 매개 변수로 전달해 주면 됩니다!
- 지금은 Reducer 에 상태 관리 기능(Action 에 따른 동작)을 제외하고 간단하게 State 를 전달하는 기능만 만들어 봅시다!



```
let weight = 100;

function reducer(state = weight) {
  return state;
}

let store = createStore(reducer);

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <Provider store={store}>
    <App />
  </Provider>
);

reportWebVitals();
```

Src/index.js

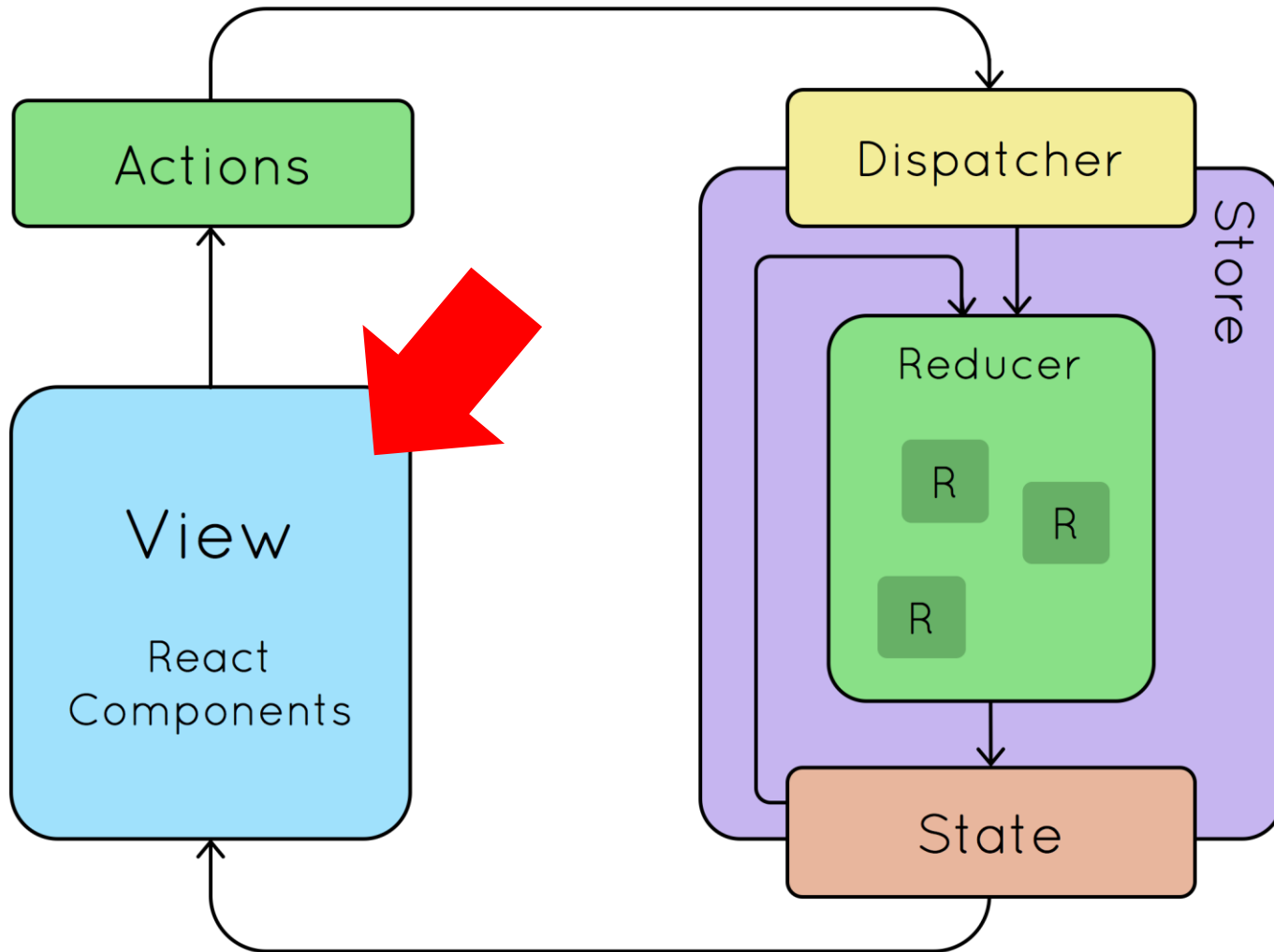
- 아직은 Action 에 따라 상태 값을 변경하는 기능은 없고, index.js 에 저장된 상태 값을 전역에 있는 컴포넌트에 전달하는 기능만을 수행 합니다!



Store 에 저장 된
값 받아오기!



Store 에 저장 된 값 받아오기!



- App.js 내부의 컴포넌트에서 Store 에 저장 되어있던 State 값을 받아 옵시다!



Store 의 상태 값을 받아올 컴포넌트 작성!

- Store 의 상태 값을 받아올 **<Test>** 컴포넌트를 작성해 봅시다!
- Store 의 상태 값을 받아올 때에는 React-redux 모듈의 **useSelector** 를 사용하면 됩니다!



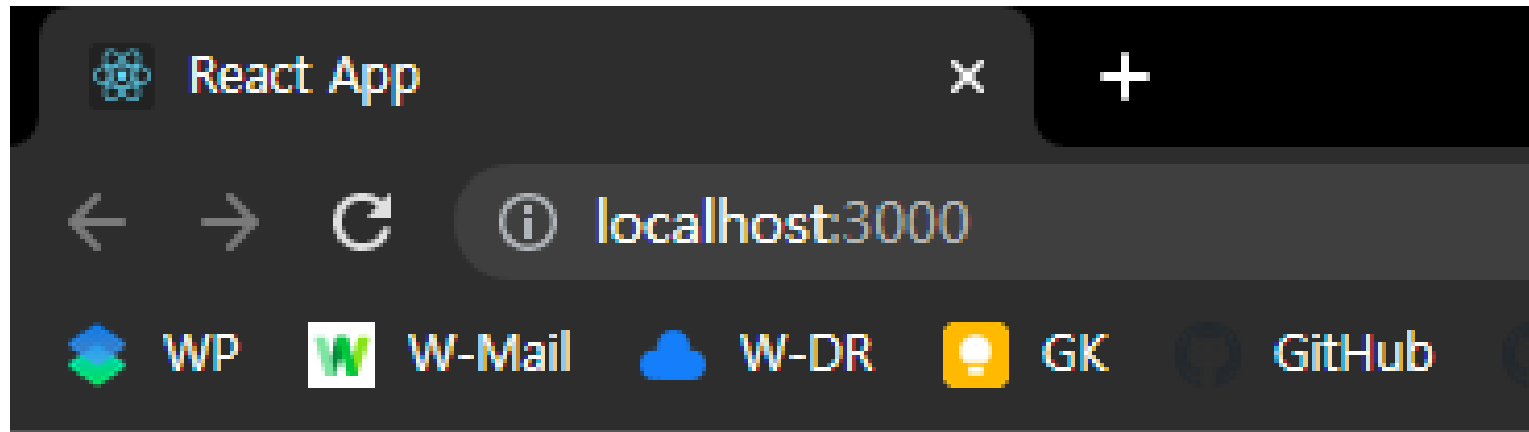
```
import React from 'react'
import { useSelector } from 'react-redux'

export default function Test() {
  const weight = useSelector((state) => state);

  return (
    <>
      <h1>당신의 몸무게는 {weight}</h1>
    </>
  )
}
```

Src/component/Test.js

- weight 라는 변수에 Store 에 저장 되어있던 상태 값을 받고, 활용하면 됩니다!

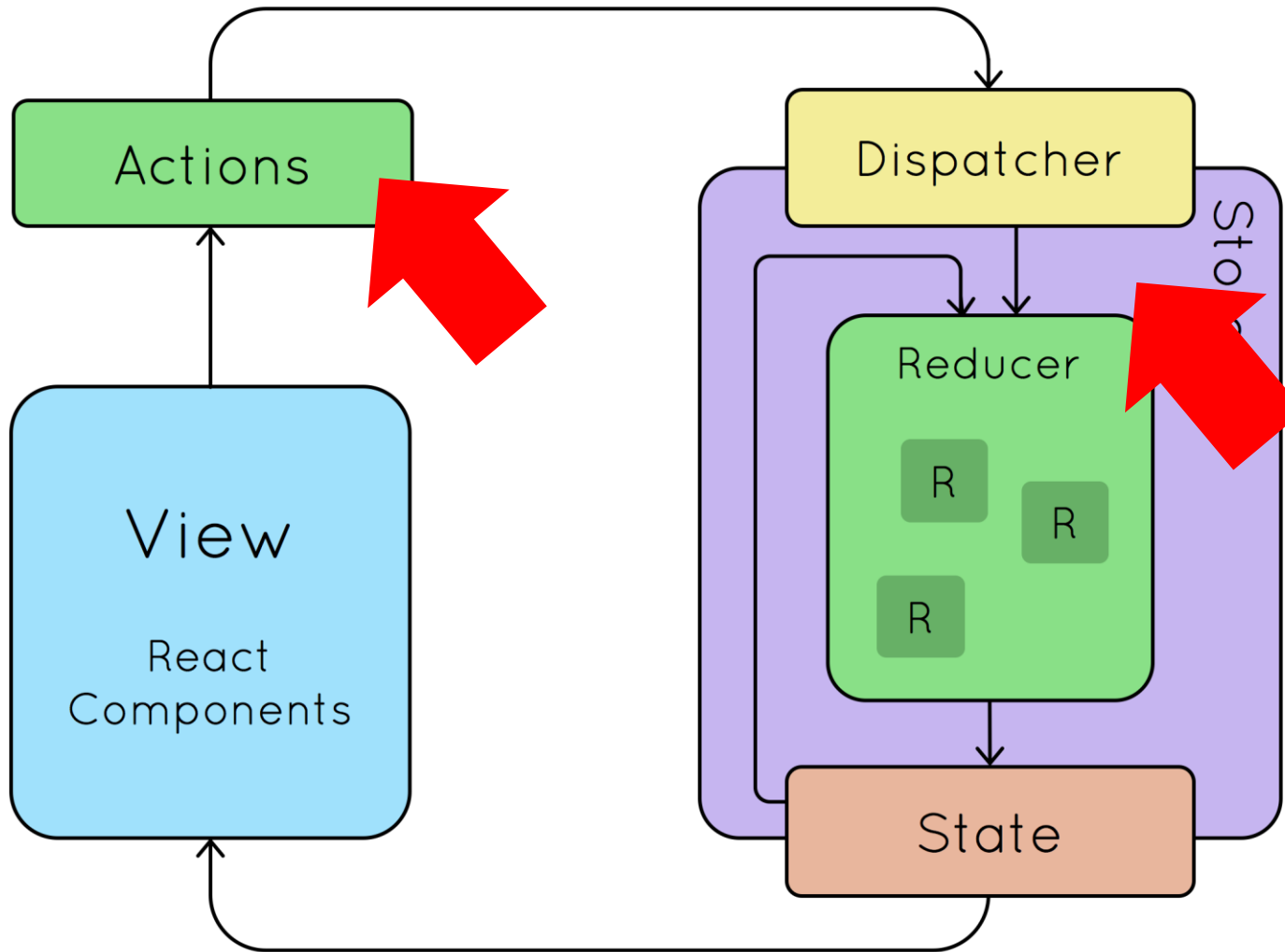


당신의 몸무게는 100



Action & Reducer

Action 설정과 Dispatch 로 Action 보내기



- Store 의 State 값 변경을 위해서는 Action 을 설정
- Action 을 Dispatch 를 사용해서 Reducer 에 전달
- Reducer 가 State 를 변경

Action



pixtastock.com - 77572318



Action & Reducer

- Action 은 Reducer 에게 어떤 처리를 해야하는지 알려주는 역할을 합니다!
- Action 은 객체 내부에 type 라는 키를 가지고 있으며, 해당 type 에는 리듀서에 전달할 액션을 "문자열" 형태로 가지고 있습니다!
- Action 은 매개변수로 Reducer 에게 전달이 되며, Reducer 는 Action 객체 내부의 type 키 값의 문자열을 읽어서 State 를 어떤 방식으로 처리할 지 결정 합니다!



Action

{

type: "some text"

}



Reducer

```
function reducer (state, action) {  
  return changeState;  
}
```



```
function reducer(state = weight, action) {  
  if (action.type === "증가") {  
    state++;  
    return state;  
  } else if (action.type === "감소") {  
    state--;  
    return state;  
  } else {  
    return state;  
  }  
}
```

Src/index.js



Action & Dispatch

- 컴포넌트에 있는 Action 을 index.js 에 있는 reducer 에 까지 보내려면 React-redux 에 있는 `useDispatch()` 를 사용해야 합니다!
- `useDispatch()` 를 하나의 변수에 담고 해당 변수를 통해 Action 의 값을 Reducer 로 전달해 주면 됩니다!
- Dispatch 로 전달 된 Action 의 type 의 값에 따라 Reducer 는 State 값을 변경하고, 해당 State 값은 컴포넌트에 반영 됩니다!



Dispatch

스토어 내장 함수.
액션을 발생시키는 함수.
Ex) dispatch(action)



```
import React from 'react'
import { useSelector, useDispatch } from 'react-redux'

export default function Test() {
  const weight = useSelector((state) => state);
  const dispatch = useDispatch();

  return (
    <>
      <h1>당신의 몸무게는 {weight}</h1>
      <button onClick={() => { dispatch({ type: "증가" }) }}>살 찌기</button>
      <button onClick={() => { dispatch({ type: "감소" }) }}>살 빼기</button>
    </>
  )
}
```

Src/component/Test.js



당신의 몸무게는 100

살 찌기

살 빼기

당신의 몸무게는 105

살 찌기

살 빼기

당신의 몸무게는 92

살 찌기

살 빼기



전체 코드



```
import { Provider } from 'react-redux';
import { createStore } from 'redux';

const weight = 100;

function reducer(state = weight, action) {
  if (action.type === "증가") {
    state++;
    return state;
  } else if (action.type === "감소") {
    state--;
    return state;
  } else {
    return state;
  }
}

let store = createStore(reducer);

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <>
    <Provider store={store}>
      <App />
    </Provider>
  </>
);
```

Src/index.js



```
import React from 'react'
import { useSelector, useDispatch } from 'react-redux'

export default function Test() {
  const weight = useSelector((state) => state);
  const dispatch = useDispatch();

  return (
    <>
      <h1>당신의 몸무게는 {weight}</h1>
      <button onClick={() => { dispatch({ type: "증가" }) }}>살 찌기</button>
      <button onClick={() => { dispatch({ type: "감소" }) }}>살 빼기</button>
    </>
  )
}
```

Src/component/Test.js



실전이야!



React 입문자들이 알아야할 Redux 쉽게설명 (8분컷)

조회수 5.7만회 · 1년 전



코딩애플

React 하다보면 Redux를 필히 만나게 되는데 **한 해 리덕스 포기자가 10만명이나 되기 때문에 준비했습니다** 리액트 강의 ...



그래서 우리가
만들 것은!?



What's the Plan for Today?

Add Todo

Like and Subscribe!



Wash the Dishes



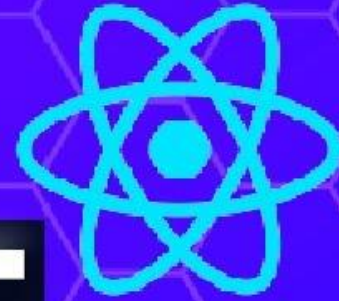
Walk the Gorilla



Skydive over Everest



REACT TODO LIST



Todo 리스트 만들기!



- 정말 간단하게 만들 겁니다!
- 할 일 추가, 할 일 완료를 누르면 완료 목록으로 옮기는 기능 정도만 추가해 볼게요!



Store

폴더 생성



Store 폴더 생성

- 기능 이전에 저장할 근간부터 만들어야 겠죠?
- Src 폴더 내부에 store 폴더를 만들어 주세요!
- Store 전체를 총괄하는 모듈은 `index.js` 가 담당할 예정입니다!



>	node_modules	
>	public	
▼	src	●
>	components	●
▼	store	●
	JS index.js	U
#	App.css	
JS	App.js	M
JS	App.test.js	
#	index.css	
JS	index.js	M
🖼️	logo.svg	
JS	reportWebVitals.js	
JS	setupTests.js	
📄	.gitignore	
{ }	package-lock.json	M
{ }	package.json	M
📖	README.md	

Store 모듈 분할

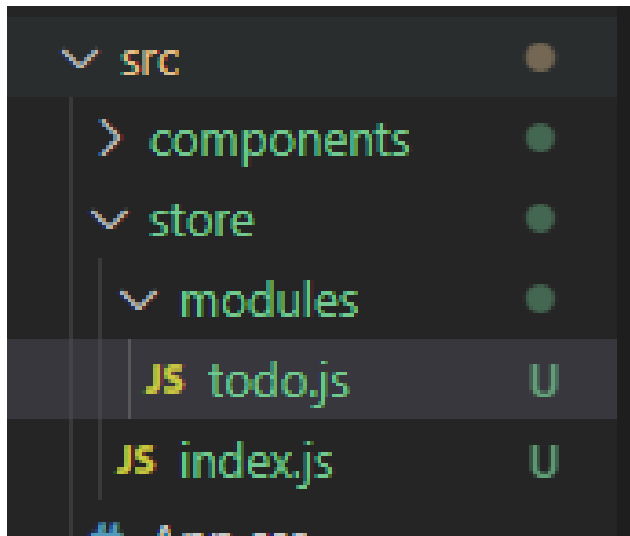


- 모든 컴포넌트에 대한 글로벌 상태 값을 하나의 파일에서 관리한다면?
- 해당 파일이 하는 일이 너무 많겠죠?
- 당연히 코드 확장성 및 관리에도 어려움이 생깁니다! → 각 기능별 Store 모듈을 분할 합니다!



Store 모듈 분할

- 먼저 store 내부에 modules 폴더를 만들어 봅시다!
- 우리는 ToDo List 를 만들 것이므로 해당 리스트를 관리하는 모듈인 **todo.js** 모듈을 modules 폴더 내부에 만들어 줍시다!





초기 State 값 선언하기



최초의 State 값 설정

- 컴포넌트가 최초 렌더링 될 때 보여줘야할 최초의 State 값을 설정해 봅시다!
- 물론 DB 에서 데이터를 받아서 설정해 주는 방법이 맞지만, 이전 백엔드와 마찬가지로 편의를 위해서 변수로 설정해 봅시다!



최초의 State 값 설정

- Todo List 이므로 객체가 담긴 배열 형태로 선언할 예정입니다!
- List 객체에는 아래의 값이 구성 될 예정입니다!
 - **id**: 고유 id 값
 - **text**: 할 일 내용
 - **done**: 완료 여부



```
// 초기 상태 설정
const initState = {
  list: [
    {
      id: 0,
      text: '리액트 공부하기',
      done: false,
    },
    {
      id: 1,
      text: '척추의 요정이 말합니다! 척추 펴기!',
      done: false,
    },
    {
      id: 2,
      text: '취업 하기',
      done: false,
    },
  ],
};
```

Src/store/modules/todo.js



Reducer 로
값 리턴 시키기!



Reducer 를 통해 State 전달!

- 설정한 State 값을 외부에서 접근 하기 위해서는 Reducer 를 통해 값을 return 시켜줘야 합니다!
- 설정한 State 값을 바로 return 시켜주는 간단한 Reducer 를 작성해 봅시다!

```
export default function todo(state = initState, action) {  
  return state;  
}
```

Src/store/modules/todo.js



- 원래는 전달 된 2번째 매개 변수인 action 의 type 에 따라 다른 동작을 수행하는 것이 진짜 Reducer 입니다.
- 지금은 초기 State 값을 외부로 전달하는 목적만 달성하면 되므로 state 매개 변수에 initState 값을 넣어서 바로 return 시켜 주면 됩니다!



Store

통합 관리



Store 통합 관리!

- Store 는 모듈 별로 관리하고, 모듈 들은 Store 폴더의 **index.js** 에 의해서 통합 관리 됩니다!
- Store 폴더의 **Index.js** 파일에 가서 모듈 들을 통합 관리 해봅시다!
- 먼저 초기 값을 선언한 todo.js 를 import 해서 todo.js 의 reducer 를 불러오고(export default 로 설정 하였음) redux 의 **combineReducer** 를 이용하여 todo.js 의 reducer 를 하나로 합쳐서 다시 내보내 줍시다!



```
// 통합 관리 파일
import { combineReducers } from 'redux';
import todo from './modules/todo';

export default combineReducers({
  todo,
});
```

Src/store/index.js

- 각각의 Reducer 들을 합쳐주는 **combineReducer** 를 이용해서 각각의 store 모듈에서 export 된 reducer 를 합쳐 줍시다!
- 그리고 다시 합쳐진 reducer 를 export default 로 보내내 줍시다!



Redux 기초 세팅 및 Store 연결



Redux 기초 세팅!

- 라우팅 처리 하던 것처럼! Redux 적용을 위해서는 `<Provider>` 컴포넌트를 임포트하고 해당 컴포넌트로 `<App>` 컴포넌트를 감싸줘야만 합니다!
- Src 폴더의 최상위 `Index.js` 에 가서 코드를 처리!(Store 의 index.js X)
- `combineReducer` 를 통해 하나로 합쳐서 내보낸 Reducer 는 `rootReducer` 라는 값으로 받아 줍시다!



```
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import rootReducer from './store';

const store = createStore(rootReducer);

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <>
    <Provider store={store}>
      <App />
    </Provider>
  </>
);
```

Src/index.js



TodoList

컴포넌트 작성



Todo List 의 기본이 될 컴포넌트 만들기!

- 할 일 목록을 보여주고, 추가하는 기능을 가지는 → `<TodoList>` 컴포넌트
- 완료된 목록을 보여주는 → `<DoneList>` 컴포넌트
- 위의 두 컴포넌트를 “포함” 하여 전체 앱을 그려주는 `<ListContainer>` 컴포넌트를 제작해 봅시다!



TodoList

컴포넌트



<TodoList> 컴포넌트

- 할 일을 추가하는 Input 요소와, 추가 버튼 요소를 만들어 줍시다!
- 할 일 목록을 redux 를 통해 Store 에서 받아온 다음, 해당 목록을 `` 태그의 `` 요소로 그려 줍시다!



```
import { useRef } from 'react';
import { useSelector } from 'react-redux';

export default function TodoList() {
  const list = useSelector((state) => state.todo.list);
  const inputRef = useRef();

  return (
    <section>
      <h1>할일 목록</h1>
      <div>
        <input type="text" ref={inputRef} />
        <button>추가</button>
      </div>
      <ul>
        {list.map((el) => {
          return <li key={el.id}>{el.text}</li>;
        })}
      </ul>
    </section>
  );
}
```

Src/component/TodoList.jsx



DoneList

컴포넌트



<DoneList> 컴포넌트

- List 는 일단 useSelector 를 이용해서 state 값을 받아 옵시다!
- 완료된 List 를 받으면, 해당 List 를 ui 요소로 그려주면 도비니다!
- <TodoList> 에서 인풋 입력을 뺀 상태로 비슷하게 구현하면 됩니다!



```
import { useSelector } from "react-redux";

export default function DoneList() {
  const list = useSelector((state) => state.todo.list);
  return (
    <section>
      <h1>완료된 목록</h1>
      <ul>
        {list.map((el) => {
          return (
            <li key={el.id}>
              {el.text}
              <button>완료</button>
            </li>
          );
        })}
      </ul>
    </section>
  );
}
```

Src/component/DoneList.jsx



ListContainer

컴포넌트



<ListContainer> 컴포넌트

- <ListContainer> 컴포넌트는 <TodoList> 와 <DoneList> 를 순서대로 포함만 하면 되므로, 각각 컴포넌트를 Import 한 다음 자식 요소로 만들어 줍니다!



```
import TodoList from './TodoList';
import DoneList from './DoneList';

export default function ListContainer() {
  return (
    <>
      <TodoList />
      <DoneList />
    </>
  );
}
```

Src/component/ListContainer.jsx



할일 목록

- 리액트 공부하기
- 척추의 요정이 말합니다 : 척추 펴기!
- 취업 하기



Redux 를 위한 편의 도구



편의 도구가 왜 필요하죠?

- Redux 에 저장 된 Store 의 값은 src 폴더의 `index.js` 파일에서 `getState()` 메소드를 이용하여 확인을 합니다!
- 지금은 간단한 Todo List 이기 때문에 하나의 값 만을 처리하고 있지만 프로젝트가 커지게 되면 다양한 전역 상태 값을 redux 로 관리 해야 합니다.
- 그럴때 마다 redux 의 모든 값을 하나하나 `console.log` 로 찍어가면서 확인하기는 매우 귀찮기 때문에 사용합니다!



```
import { createStore } from 'redux';
import rootReducer from './store';

import { Provider } from 'react-redux';

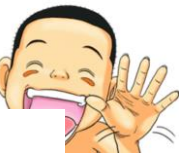
const store = createStore(rootReducer);
console.log(store.getState());

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <>
    <Provider store={store}>
      <App />
    </Provider>
  </>
);
```

Src/index.js



```
▼ Object i  
  ▼ todo:  
    ▼ list: Array(3)  
      ▶ 0: {id: 0, text: '리액트 공부하기', done: false}  
      ▶ 1: {id: 1, text: '척추의 요정이 말합니다 : 척추 펴기!', done: false}  
      ▶ 2: {id: 2, text: '취업 하기', done: false}  
      length: 3  
      ▶ [[Prototype]]: Array(0)  
      ▶ [[Prototype]]: Object  
      ▶ [[Prototype]]: Object
```



홈 > 확장 프로그램 > Redux DevTools



Redux DevTools

추천

★★★★★ 571 ⓘ | 개발자 도구 | 사용자 1,000,000+명

Chrome에 추가

<https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklieibfkpmmfibljd?hl=ko>

사용법!



- <https://github.com/reduxjs/redux-devtools/tree/main/extension#installation>

1.1 Basic store

For a basic Redux store simply add:

```
const store = createStore(  
  reducer, /* preloadedState, */  
+ window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__()  
);
```

사용법!



- redux store 를 만들 때, 약속 된 코드를 삽입해 주면 됩니다!

```
const reduxDevTool =  
  window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__();  
  
const store = createStore(rootReducer, reduxDevTool);
```

Src/index.js

Actions Settings

filter...

@@INIT 8:14:01.26

State Action State Diff Trace Test

Tree Chart Raw

- ▼ todo (pin)
 - ▼ list (pin)
 - ▶ 0 (pin): { id: 0, text: "리액트 공부하기", done: false }
 - ▶ 1 (pin): { id: 1, text: "책추의 요정이 말함...", done: false }
 - ▶ 2 (pin): { id: 2, text: "취업 하기", done: false }

@@INIT (0)

Inspector Log monitor Chart RTK Query

React App

Live 1x 2x





Action 타임

설정



Action 타입 정의하기

- Action 타입은 "문자열"로 보통 정의합니다!
- Todo 리스트에 필요한 생성, 완료 액션을 정의합니다!

```
// 액션 타입 정의하기  
const CREATE = "todo/CREATE";  
const DONE = "todo/DONE";
```

Src/store/modules/todo.js



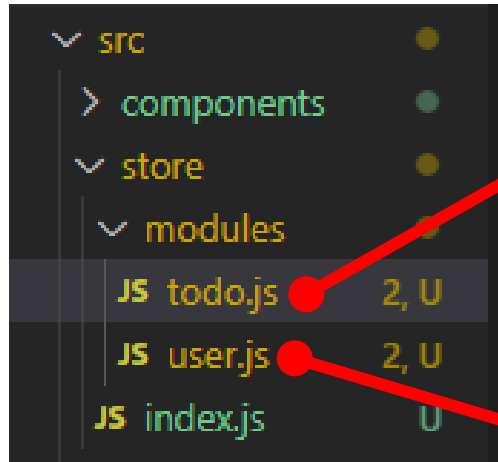
Action 타입 정의하기

- 앞에 **todo/** 는 왜 붙이는 건가요?
- 이 것은 잘못 된 사용을 막기 위한 하나의 방법입니다!
- 모듈이 달라도 Action 타입으로 CREATE, DELETE, DONE 같은 변수명은 상당히 많이 사용이 됩니다.
- 그럴 때 잘못 된 모듈 Import 로 인해, 다른 Reducer 의 기능이 호출되면 문제가 발생합니다



Action 타입 정의하기

- 이럴 때 Action 타입 앞에 지금 이 액션의 타입이 어떤 모듈의 타입인지를 알려주는 문자열을 추가하여 위와 같은 문제가 발생하는 것을 막아 줍니다!



```
// 액션 타입 정의하기  
const CREATE = "todo/CREATE";  
const DONE = "todo/DONE";
```

```
// 액션 타입 정의하기  
const CREATE = "user/CREATE";  
const DONE = "user/DONE";
```



Action 생성 함수

작성



Action 생성 함수 작성

- 외부 컴포넌트에서 Action 을 만들어주는 함수부터 작성을 해봅시다!
- Action 생성 함수는 type 정보와 전달해야 할 정보를 payload 객체에 담아서 **Dispatch** 를 통해 전달 합니다!
- 결과적으로 Reducer 가 Action 함수에 들어있는 type 을 확인해서 어떤 행동을 할지 정하고, payload 에 있는 데이터를 받아서 처리 합니다!



Action 생성 함수 작성 - Create

- 새로운 할 일 목록을 만드는 create 함수부터 작성해 봅시다!
- 먼저 Action type 설정 부터 CREATE 로 해줍니다!
- 그리고 전달 해야할 정보는 payload 라는 매개 변수에 담아서 전달 합니다!



```
// 액션 생성 함수 작성
export function create(payload) {
  return {
    type: CREATE,
    payload,
  };
}
```

Src/store/modules/todo.js



Action 생성 함수 작성 - Done

- 할 일을 완료하는 역할을 하는 Done 함수도 작성해 봅시다!
- 먼저 Action type 설정 부터 DONE 으로 해줍니다!
- 이번에는 새로운 정보를 전달 할 필요가 없이 어떤 목록이 완료 되었는지만 알면 되기 때문에 id 값만 전달 하면 됩니다!

```
export function done(id) {  
  return {  
    type: DONE,  
    id,  
  };  
}
```

Src/store/modules/todo.js





```
// 액션 타입 정의하기
const CREATE = 'todo/CREATE';
const DONE = 'todo/DONE';

// 액션 생성 함수 작성
export function create(payload) {
  return {
    type: CREATE,
    payload,
  };
}

export function done(id) {
  return {
    type: DONE,
    id,
  };
}
```

외부에서 직접 요청하지 않고
Todo.js 의 함수를 import 해서
사용하는 이유는

이렇게 type 값 등을 외부에서는
알 수가 없기 때문에
약속 된 함수만 사용하여
접근하는 것이 편하기 때문입니다!

Src/store/modules/todo.js

전체 코드



Reducer

구조 구현



Action Type 에 따라 작동하는 Reducer

- 이제는 Action Type 에 따라 작동하는 Reducer 를 구현해 봅시다!
- 먼저, switch 문을 이용해서 action type 에 따라서 각각의 역할을 한 뒤 값을 return 하는 구조로 만들어 주시면 됩니다!



// 리듀서 설정(실제 작업은 이친구가 합니다!)

```
export default function todo(state = initState, action) {  
  switch (action.type) {  
    case CREATE:  
      return console.log('CREATE 호출');  
    case DONE:  
      return console.log('DONE 호출');  
    default:  
      return state;  
  }  
}
```

Src/store/modules/todo.js



Dispatch 로 Action 함수 전달



Dispatch 로 Action 함수 전달

- 그럼 이번에는 컴포넌트에서 **Dispatch** 로 정의한 Action 함수를 Reducer 에 전달하여 정상적으로 호출이 되는지 확인해 봅시다!
- Dispatch 활용을 위해 **useDispatch** 를 dispatch 변수에 넣어주기!
- **Src/store/modules/todo.js** 에서 create, done 함수 불러오기!
- **Dispatch** 의 인자로 create, done 함수를 전달하여 호출 상태 확인!



```
import { useRef } from 'react';  
import { useDispatch, useSelector } from 'react-redux';  
import { create, done } from '../store/modules/todo';
```

```
export default function TodoList() {  
  const list = useSelector((state) => state.todo.list);
```

```
  const inputRef = useRef();  
  const dispatch = useDispatch();
```

```
  return (  
    <section>  
      <h1>할일 목록</h1>  
      <div>  
        <input type="text" ref={inputRef} />  
        <button  
          onClick={() => {  
            dispatch(create(''));  
          }}  
        >  
          추가  
        </button>  
      </div>  
      <ul>  
        {list.map((el) => {  
          return <li key={el.id}>{el.text}</li>;  
        })}  
      </ul>  
    </section>  
  );  
}
```

Src/component/TodoList.jsx



Download the React DevTools for a better development experience

```
▶ {todo: {...}}
```

CREATE 호출

```
2 ▶ Uncaught Error: When called with an action of type
  an action, you must explicitly return the previous
    at combination (redux.js:564:1)
    at k (<anonymous>:2235:16)
    at D (<anonymous>:2251:13)
    at <anonymous>:2464:20
    at Object.dispatch (redux.js:288:1)
    at e (<anonymous>:2494:20)
    at onClick (TodoList.js:18:1)
    at HTMLUnknownElement.callCallback (react-dom.development
    at Object.invokeGuardedCallbackDev (react-dom.development
    at invokeGuardedCallback (react-dom.development
```

Download the React DevTools for a better development experience

```
▶ {todo: {...}}
```

DONE 호출

```
2 ▶ Uncaught Error: When called with an action of type
  an action, you must explicitly return the previous
    at combination (redux.js:564:1)
    at k (<anonymous>:2235:16)
    at D (<anonymous>:2251:13)
    at <anonymous>:2464:20
    at Object.dispatch (redux.js:288:1)
    at e (<anonymous>:2494:20)
    at onClick (TodoList.js:18:1)
    at HTMLUnknownElement.callCallback (react-dom.development
    at Object.invokeGuardedCallbackDev (react-dom.development
    at invokeGuardedCallback (react-dom.development
```



Reducer

CREATE 구현



Reducer 의 CREATE 동작 구현

- 이제 들어온 Action Type 에 따른 reducer 의 실제 동작을 구현해 봅시다!
- 먼저 CREATE 부터 구현을 해봅시다!
- 혹시 모를 다른 초기 값이 있을지 모르므로 state 를 전개 연산자로 먼저 리턴해 줍니다.
- List 의 경우는 새롭게 입력 받은 값을 list 의 배열에 넣어 주면 됩니다!



```
export default function todo(state = initState, action) {  
  switch (action.type) {  
    case CREATE:  
      return {  
        ...state,  
        list: state.list.concat({  
          id: action.payload.id,  
          text: action.payload.text,  
          done: false,  
        }),  
        nextID: action.payload.id + 1,  
      };  
    case DONE:  
      return {  
        ...state,  
      };  
    default:  
      return state;  
  }  
}
```

Push 말고 Concat 을 사용하는 이유는?

지금은 list 라는 배열에 변경 된 값을
리턴해 줘야 하는 상황입니다!

Push 는 배열에 값을 추가하고 배열의
길이를 리턴해 주고, concat 은 값이 추
가된 배열을 리턴해 줍니다!

따라서 push 를 쓰면 list 에는 숫자 값
만 들어가므로 문제가 생깁니다!

Src/store/modules/todo.js



Dispatch 로 CREATE 호출



CREATE 호출

- 이번에는 CREATE 의 함수에 제대로 된 인자를 전달하여 정상적으로 기능이 작동하도록 해봅시다!
- 리듀서에서 할 일 목록 추가로 필요한 정보는 id 값과 새롭게 추가될 할 일의 text 값이 필요합니다 → 두 데이터를 객체에 담아서 인자로 전달해 봅시다!



```
import { useRef } from "react";
import { useDispatch, useSelector } from "react-redux";
import { create, done } from '../store/modules/todo';

export default function TodoList() {
  const list = useSelector((state) => state.todo.list);
  const inputRef = useRef();
  const dispatch = useDispatch();

  return (
    <section>
      <h1>할일 목록</h1>
      <div>
        <input type="text" ref={inputRef} />
        <button
          onClick={() => {
            dispatch(create({ id: list.length, text: inputRef.current.value }));
            inputRef.current.value = "";
          }}
        >
          추가
        </button>
      </div>
    </section>
  );
}
```

Src/component/TodoList.jsx

할일 목록



추가

- 리액트 공부하기
- 척추의 요정이 말합니다 : 척추 펴기!
- 취업 하기
- 추가가 되나요!?



Reducer

DONE 구현



Reducer 의 DONE 동작 구현

- 동일하게 list 이외의 초기 state 값은 그대로 전달이 되어야 하므로 전개 연산자를 사용!
- List 의 경우는 컴포넌트에서 전달 받은 id 값과 동일한 객체를 찾은 다음 해당 객체의 done 항목을 true 로 변경하면 됩니다!
- 이럴 때는 `map()` 을 쓰면 편합니다! `map()` 은 배열의 모든 값을 순회 하면서 배열의 값을 return 된 값으로 변경해 줍니다!



```
case DONE:
  return {
    ...state,
    list: state.list.map((el) => {
      if (el.id === action.id) {
        return {
          ...el,
          done: true,
        };
      } else {
        return el;
      }
    }),
  };
default:
  return state;
```

Src/store/modules/todo.js



```
// 액션 타입 정의하기
const CREATE = 'todo/CREATE';
const DONE = 'todo/DONE';

// 액션 생성 함수 작성
export function create(payload) {
  return {
    type: CREATE,
    payload,
  };
}

export function done(id) {
  return {
    type: DONE,
    id,
  };
}

// 초기 상태 설정
const initState = {
  list: [
    {
      id: 0,
      text: '리액트 공부하기',
      done: false,
    },
    {
      id: 1,
      text: '책추의 요정이 말합니다 : 책추 펴
기!',
      done: false,
    },
    {
      id: 2,
      text: '취업 하기',
      done: false,
    },
  ],
};
```

```
// 리듀서 설정(실제 작업은 이친구가 합니다!)
export default function todo(state = initState, action) {
  switch (action.type) {
    case CREATE:
      return {
        ...state,
        list: state.list.concat({
          id: action.payload.id,
          text: action.payload.text,
          done: false,
        }),
        nextID: action.payload.id + 1,
      };
    case DONE:
      return {
        ...state,
        list: state.list.map((el) => {
          if (el.id === action.id) {
            return {
              ...el,
              done: true,
            };
          } else {
            return el;
          }
        }),
      };
    default:
      return state;
  }
}
```

Src/store/modules/todo.js

전체 코드



Dispatch 로

DONE 호출



DONE 호출

- 이번에는 DONE 의 함수에 제대로 된 인자를 전달하여 정상적으로 기능이 작동하도록 해봅시다!
- 리듀서에서 완료 된 목록의 id 값만 받아서 해당 목록의 done 항목을 true 로 변경만 하면 됩니다!
- Done 함수에 인자로 id 값을 전달해 봅시다!



```
<ul>
  {list.map((el) => {
    return (
      <li key={el.id}>
        {el.text}
        <button
          onClick={() => {
            dispatch(done(el.id));
          }}
        >
          완료
        </button>
      </li>
    );
  })}
</ul>
```

Src/component/ToDoList.js



각각 컴포넌트에 Filter 걸기!



Filter 처리!

- 지금 `<TodoList>` 컴포넌트와 `<DoneList>` 컴포넌트는 동일한 List 를 출력하고 있습니다!
- 이제 done 의 값을 통해 필터링 하여 `<TodoList>` 에는 할 일 목록만, `<DoneList>` 에는 완료 된 목록만 남겨 봅시다!



TodoList 컴포넌트

- `<TodoList>` 컴포넌트 List 의 항목 중에서 done 의 값이 `false` 인 친구들만 가져오면 됩니다!
- 배열의 filter 메소드는 조건식을 만족하는 배열만 남겨서 리턴해 주므로 해당 메소드를 사용 하면 됩니다!


```
export default function TodoList() {  
  const list = useSelector((state) => state.todo.list).filter(  
    (el) => el.done === false  
  );  
};
```

Src/component/TodoList.jsx





DoneList 컴포넌트

- **<DoneList>** 컴포넌트 List 의 항목 중에서 done 의 값이 true 인 친구들만 가져오면 됩니다!

```
export default function DoneList() {  
  const list = useSelector((state) => state.todo.list).filter(  
    (el) => el.done === true  
  );  
}
```

Src/component/DoneList.js





하지만
언제나 조져지는 건
나였다



킹치만...

- 역시 테스트를 해보니 에러가 뜨네요!

```
✖ Warning: Encountered two children with the same key, react-dom.development.js:86  
`0`. Keys should be unique so that components maintain their identity across  
updates. Non-unique keys may cause children to be duplicated and/or omitted – the  
behavior is unsupported and could change in a future version.  
    at ul  
    at section  
    at DoneList (http://localhost:3000/static/js/bundle.js:103:72)  
    at ListContainer  
    at div  
    at App  
    at Provider (http://localhost:3000/static/js/bundle.js:37892:5)
```



원인은...

- List 요소의 key 값은 고유해야 하지만 고유하지 않아서 생기는 문제 입니다!
- TodoList 에서 할 일을 추가 할 때, Store 에서 받아온 list 의 length 값을 넘기고 있습니다 → 이미 완료를 몇 개 하면 list 의 길이가 짧아짐 → 새로 생성 되는 요소는 이전의 key 와 동일한 값을 가지게 됨 → 에러 발생!



해결책은...

- 할 일 목록의 id를 목록의 순번으로 부여를 하고 있으므로, 해당 순번도 store 에서 전역으로 관리하여 문제를 막아 봅시다!
- 이런 부분이 리얼 redux 실전 입니다! 😊



문제 해결하기!



ID 를 관리하기 위한 State 생성!

- Store 의 todo 모듈에 ID 관리를 위한 값을 설정해 봅시다!
- 일단 초기 List 의 길이 값을 구하고, 해당 값을 다음에 생성 될 할 일 목록의 ID 값으로 넘겨주는 구조를 그려 봅시다!



```
const initState = {  
  // 초기 상태 설정  
};  
  
let counts = initState.list.length;  
initState['nextID'] = counts;
```

Src/store/modules/todo.js



CREATE 리듀서에 해당 내용 추가!

- CREATE 액션이 호출 되면 nextID 의 값을 새로운 할 일 목록의 id 로 전달 되기 때문에, 그 다음에 CREATE 가 호출 되기 전에 nextID 값은 +1 상태가 되어야 합니다!
- 따라서, action 에서 받아온 id 값(이전 상태의 nextID 값)에 +1 을 해주면 됩니다!

```
export default function todo(state = initState, action) {  
  switch (action.type) {  
    case CREATE:  
      return {  
        ...state,  
        list: state.list.concat({  
          id: action.payload.id,  
          text: action.payload.text,  
          done: false,  
        }),  
        nextID: action.payload.id + 1,  
      };  
  }  
}
```





할 일 목록 추가 시 기능 수정



<TodoList> 컴포넌트 기능 수정

- 이제는 새롭게 만들어질 할 일 목록의 id 는 list.length 로 보내는 것이 아니라 Store 의 **todo.js** 모듈에서 받아오면 됩니다!
- 해당 값을 CREATE 액션 호출 시 전달해 주면, 리듀서에서 그 값을 받아 다음 할 일 목록의 id 값을 +1 시켜 주므로 논리적으로 문제 없이 구성이 가능합니다!

```
export default function TodoList() {
  const list = useSelector((state) => state.todo.list).filter(
    (el) => el.done === false
  );

  const nextID = useSelector((state) => state.todo.nextID);

  const inputRef = useRef();
  const dispatch = useDispatch();

  return (
    <section>
      <h1>할일 목록</h1>
      <div>
        <input type="text" ref={inputRef} />
        <button
          onClick={() => {
            dispatch(create({ id: nextID, text: inputRef.current.value }));
            inputRef.current.value = '';
          }}
        >
          추가
        </button>
      </div>
    </section>
  );
}
```





수고하셨습니다!