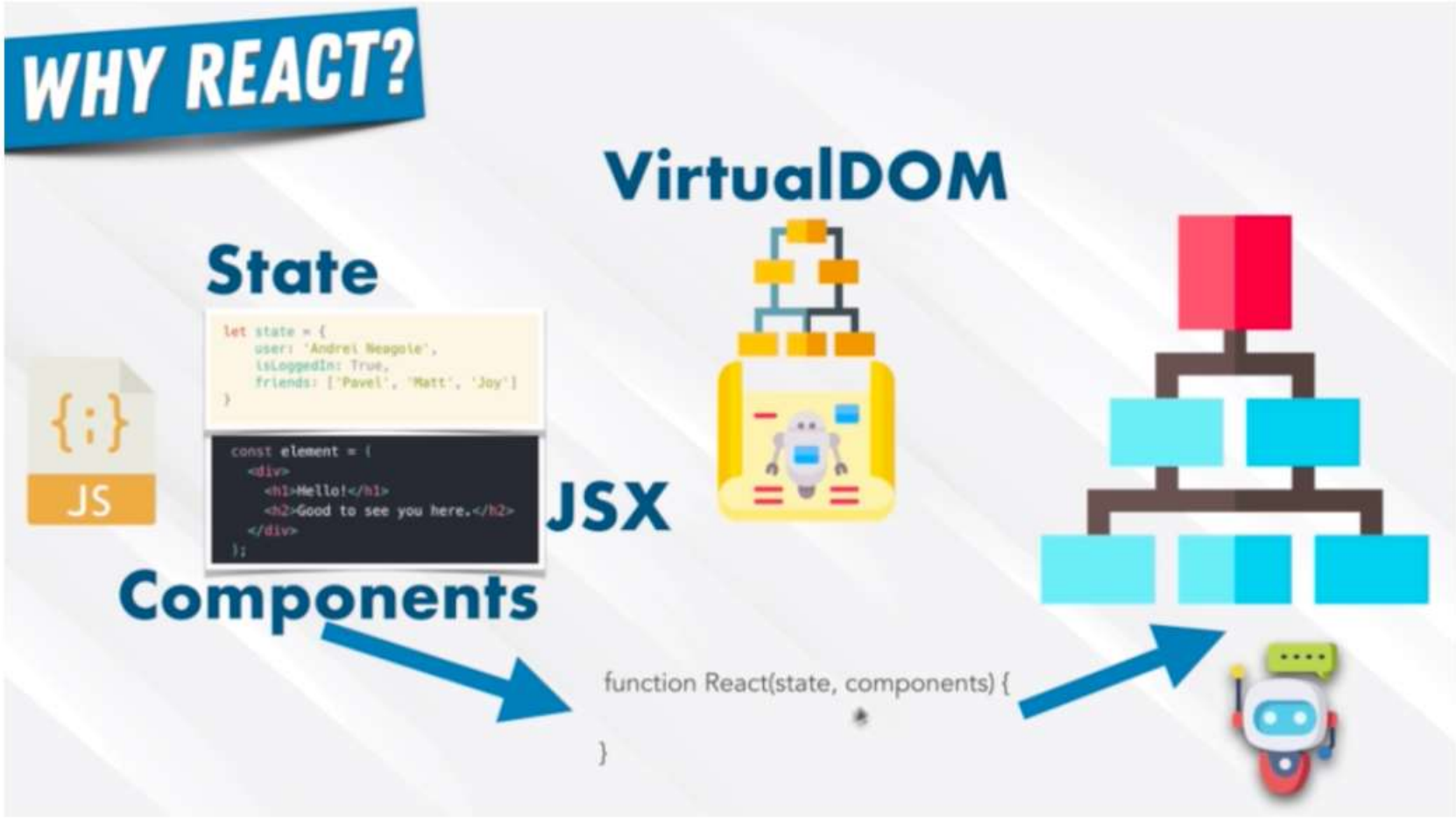
React 기초


WHY REACT?
VirtualDOM
State
let state = {
user: 'Andrei Neagoie',
isLoggedIn: True,
friends: ['Pavel', 'Matt', 'Joy']
}
JS
const element = (

<h1>Hello!</h1>
<h2>Good to see you here.</h2>

);
JSX
Components
function React(state, components) {
}

React의 역사화 역할, 기본 개념 소개
JSX 문법 소개
함수형 컴포넌트와 클래스형 컴포넌트
Props와 State 개념
useState 훅 활용
이벤트 처리 방법/조건부 렌더링
간단한 Todo List 예제 프로젝트 만들기
컴포넌트 간 데이터 흐름과 상태 관리 심화
React Hooks
React Router를 이용한 페이지 네비게이션
스타일링 컴포넌트
과정 정리 및 평가

ReactJS

- 2013년 Facebook에서 개발한 JavaScript 라이브러리
- UI의 "View"만을 담당하는 FrontEnd Framework
- 컴포넌트 기반 개발과 선언형 프로그래밍을 지원
- React는 페이스북(메타), 넷플릭스, 인스타그램, 에어비앤비 등 다양한 기업에서 사용

2011년 페이스북의 엔지니어 Jordan Walke가 React의 초기 버전을 개발
2012년 React가 페이스북의 뉴스피드(News Feed)에 처음 사용됨
2013년 React가 오픈 소스로 공개됨
2015년 React 0.14에서 함수형 컴포넌트(Function Component) 도입
2016년 React Fiber(새로운 렌더링 엔진) 개발 시작
2017년 React 16 출시 (React Fiber 적용)
2020년 React 17 출시 (대규모 개선보다는 점진적 업그레이드 초점)
2022년 React 18 출시 (Concurrent Rendering, Automatic Batching 도입)

ReactJS

- 상태 기반(State-based)으로 컴포넌트를 관리함
  - 부모-자식 관계를 통해 단방향으로 데이터를 전달 (Props)
- JSX(JavaScript XML) 문법 사용
  - JavaScript 코드 안에서 HTML을 작성, UI 구조를 유연하게 구성
- 가상 DOM(Virtual DOM) 사용
  - 변경 사항을 비교하여 필요한 부분만 업데이트하는 Reconciliation(조정) 과정 수행
  - React는 UI의 상태(State)를 정의하면 자동으로 렌더링되는 선언형 프로그래밍 방식
- React 16.8(2019년)부터 클래스형 컴포넌트 없이도 상태 관리 가능하도록 useState, useEffect 등의 Hook 기능 도입
  - 함수형 컴포넌트 중심 개발이 가능해짐
- React는 상태 관리나 라우팅 같은 기능이 기본적으로 포함되어 있지 않으며, Redux, React Router 외부 라이브러리를 사용
- Create React App (CRA)를 이용하여 프로젝트를 생성하고, Webpack 기반으로 빌드 및 배포합니다.
- https://react.dev
- https://react.dev/reference/react

ReactJS 활용 분야

- React Router를 활용하여 페이지 전환이 없는 싱글 페이지 애플리케이션(SPA)의 동적 UI 구현에 사용됨
- 상태 관리 라이브러리(Redux, Recoil)를 사용하여 대규모 웹 애플리케이션 구현에 활용될 수 있음
- React Native를 이용하여 iOS/Android 모바일 앱 개발에 활용됨
- 대규모 애플리케이션 및 기업용 애플리케이션 개발에 적합하며, 유연한 설계로 다양한 환경에서 활용할 수 있습니다.
- 정적 사이트 생성을 위해 Gatsby, 서버 사이드 렌더링을 위해 Next.js와 같은 다양한 생태계가 존재합니다

전통적으로 웹 페이지는 모든 페이지마다 HTML , CSS , JavaScript 파일을 각각 가지고 있다. SPA 는 웹 또는 앱 실행 시 HTML , CSS , JavaScript 를 '최초 1번만 load' 하고, 이 후에는 JavaScript 파일을 통해 DOM 또는 필요한 HTML 파일을 조작하는 방식으로 동작한다.

React 개발 환경 구성

- Node.js : 서버 측에서 JavaScript를 실행할 수 있게 해주는 런타임 환경
- npm(Node Package Manager) : Node.js의 패키지 관리 도구(package.json 파일을 통해 의존성 패키지들을 관리)
- npx (Node Package eXecute) : 설치된 패키지를 로컬 또는 글로벌로 실행시키는 npm에 포함된 명령어
- **React** : UI를 구성하는 주요 라이브러리
- **ReactDOM** : 실제 DOM과 React의 가상 DOM 간의 연동을 담당하는 라이브러리
- Babel : ECMAScript6를 지원하지 않는 환경에서 ECMAScript6 Syntax를 사용할 수 있게 해줍니다.
- Webpack : 애플리케이션의 코드, 스타일, 이미지 등을 번들링하여 하나의 파일로 묶어주는 빌드 도구입니다. React 애플리케이션의 모든 파일들을 처리하고 최적화하여 bundle.js로 출력합니다
  웹 브라우저가 해석할 수 없는 .hbs, .cjs, .sass 등의 파일을 웹팩이 분석하여 .js, .css등의 파일로 변환
  js, png, jpg 등과 같은 파일을 적절한 크기로 자르거나 묶어주는 역할도 합니다.
  불필요한 파일을 제외하거나 압축하여 프로젝트의 용량을 줄여줍니다.
- webpack-dev-server : webpack에서 지원하는 간단한 개발 서버
  hot-loader를 통하여 코드가 수정 될 때마다 자동으로 리로드 되게 할 수 있습니다.

React 개발 환경 구성

1. Node.js와 npm(Node Package Manager)을 기반으로 동작하므로 Node.js와 npm을 먼저 구성
   https://nodejs.org
2. Create React App 라이브러리 설치

```
npm install -g create-react-app
```

3. VS Code 설치
   https://code.visualstudio.com

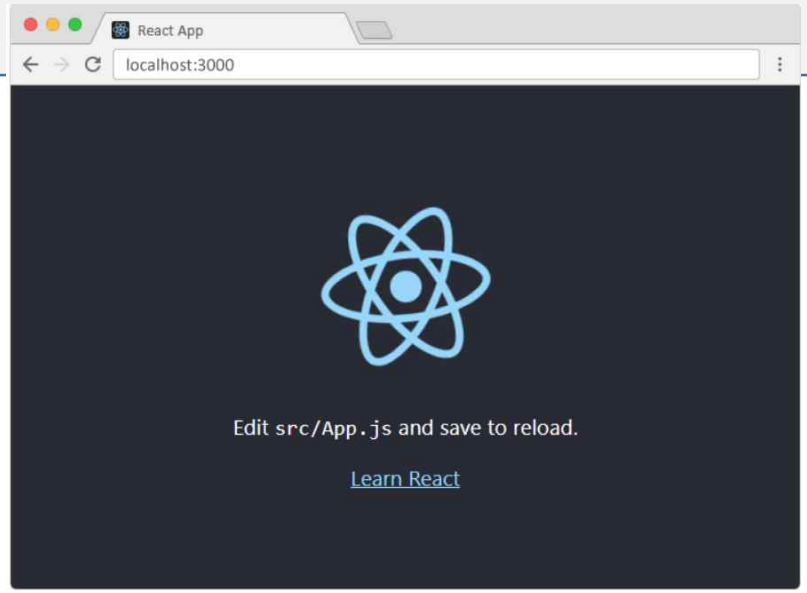
React 개발 환경 구성

1. React 프로젝트 생성

```
npx create-react-app my-app
```

2. 프로젝트 디렉토리로 이동, React 개발 서버 실행

```
cd my-app
npm start
```



npx는 npm 패키지 실행기로, create-react-app 패키지를 로컬에 설치하지 않고도 바로 실행할 수 있게 해줍니다.

React 개발 환경 구성

- React 프로젝트 구조

```
my-app/
├── node_modules/          # 설치된 npm 패키지들
├── public/
│   ├── index.html         # 애플리케이션의 HTML 템플릿
├── src/                   # 애플리케이션 소스 코드
│   ├── App.js             # 기본 앱 컴포넌트
│   ├── index.js           # React 앱의 진입점
├── package.json           # 프로젝트의 의존성 및 설정
└── README.md              # 프로젝트 설명
```

```
npm i react@latest react-dom@latest
```

React 개발 환경 구성

- package.json

key	설명
name	프로젝트의 이름을 정의
version	프로젝트의 현재 버전
private : true	프로젝트를 npm에 게시하지 못하도록 합니다
dependencies	프로덕션 및 개발에 필요한 라이브러리 @testing-library/* React 컴포넌트 테스트를 위한 라이브러리 "react": "^19.1.0" React 라이브러리의 주요 패키지 "react-dom": "^19.1.0" 는 React를 브라우저에서 렌더링할 때 필요한 패키지 "react-scripts": "5.0.1"는 create-react-app을 기반으로 프로젝트를 실행하는 스크립트 및 설정을 포함하는 패키지 Webpack, Babel 등의 설정이 포함되어 있습니다. "web-vitals": "^2.1.4" 는 웹 성능을 측정하기 위한 라이브러리
scripts	npm 명령어 설정 "start": "react-scripts start" 개발 서버를 실행하여 React 애플리케이션을 실행 "build": "react-scripts build" 프로덕션용으로 React 애플리케이션을 빌드 "test": "react-scripts test" Jest 기반으로 테스트를 실행 "eject": "react-scripts eject" create-react-app의 기본 설정(Webpack, Babel 등)을 프로젝트 내에서 직접 수정할 수 있도록 설정을 추출

React 개발 환경 구성

- package.json

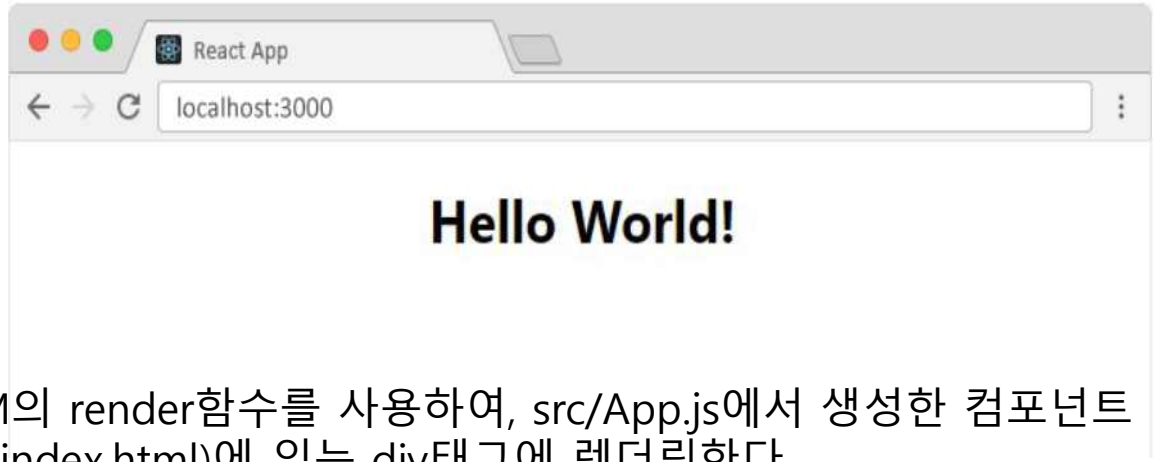
key	설명
eslintConfig	ESLint(정적 코드 분석 도구) 설정을 확장 react-app: create-react-app의 기본 ESLint 규칙을 따릅니다. react-app/jest: Jest(테스트 프레임워크) 관련 ESLint 규칙을 적용합니다.
browserslist	브라우저 호환성 설정 (production모드, development 모드) ">0.2%": 전 세계 사용률이 0.2% 이상인 브라우저를 지원합니다. "not dead": 공식적으로 지원이 중단되지 않은 브라우저만 지원합니다. "not op_mini all": Opera Mini 브라우저는 지원하지 않습니다. "last 1 chrome version": 최신 버전의 Chrome을 사용합니다. "last 1 firefox version": 최신 버전의 Firefox를 사용합니다. "last 1 safari version": 최신 버전의 Safari를 사용합니다.

React 개발 환경 구성

- React Application으로 실행 (/react프로젝트폴더/src/App.js)

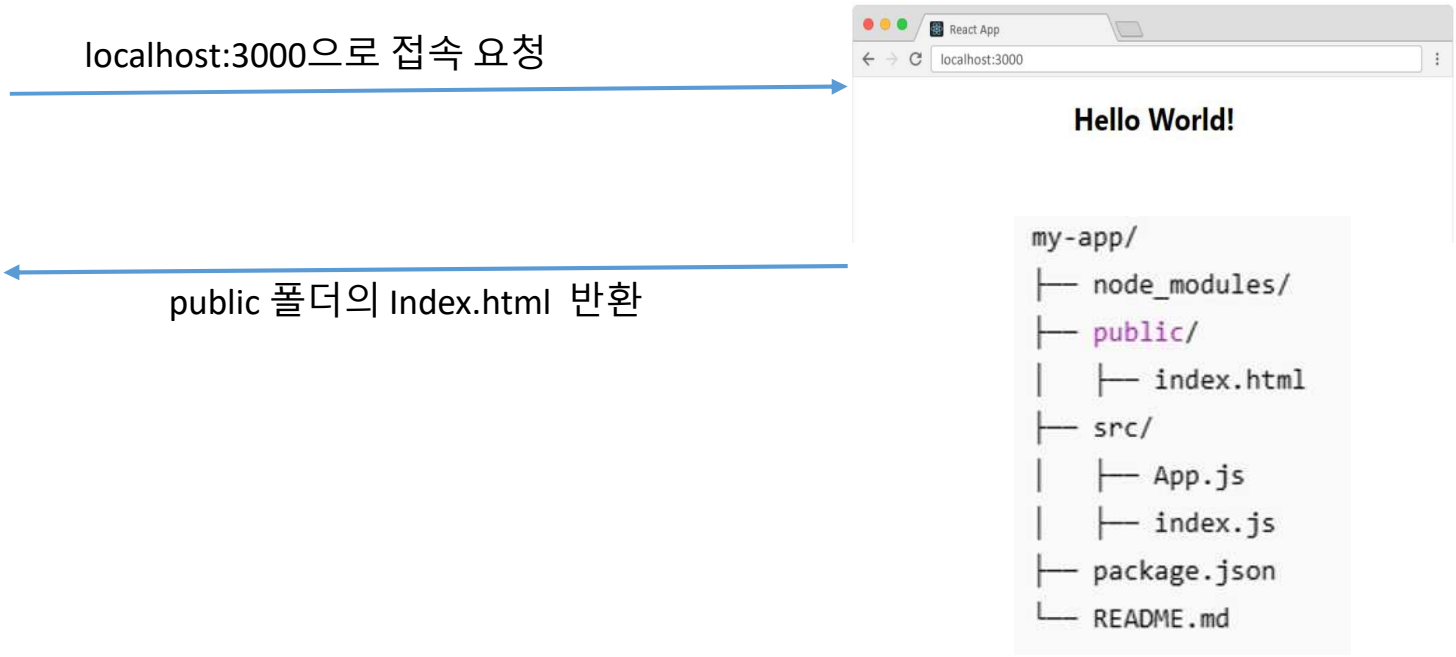
```
function App() {
  return (
    <div className="App">
      <h1>Hello World!</h1>
    </div>
  );
}

export default App;
```

- /react프로젝트폴더/npm start
- http://localhost:3000



src/index.js에서 ReactDOM의 render함수를 사용하여, src/App.js에서 생성한 컴포넌트를 DOM 컨테이너(public/index.html)에 있는 div태그에 렌더링한다.

React 개발 환경 구성



- index.html에는 src 폴더의 index.js와 해당 파일이 가져오는 자바스크립트 파일을 한데 묶어 놓은 bundle.js를 불러온다. <script> 태그에서 자동으로 추가한다.
- bundle.js는 Webpack과 같은 번들링 도구에 의해 생성, 애플리케이션의 모든 코드와 의존성을 하나의 파일로 묶은 결과물 (애플리케이션의 모든 React 코드와 다른 JavaScript 라이브러리, CSS, 이미지 등 모든 필요한 파일을 포함)
- index.js는 React 애플리케이션의 진입점으로, ReactDOM.render() 또는 ReactDOM.createRoot().render() [React 18에서 도입]를 사용하여 React 앱을 DOM에 연결합니다.
- render()를 사용해 DOM의 루트 아래에 자식 컴포넌트를 추가한다. (App 컴포넌트가 렌더링된다.)

React 개발 환경 구성

- HTML 파일에서 React실행

```
<!DOCTYPE html>
<html>
  <head>
    <script src=https://unpkg.com/react@18/umd/react.development.js crossorigin></script>
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
crossorigin></script>
    <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>
  <body>

    <div id="mydiv"></div>  <!– 루트 요소 : React 애플리케이션의 진입점 역할 -->

    <script type="text/babel">
      function Hello() {
        return <h1>Hello World!</h1>;
      }

      const container = document.getElementById('mydiv');
      const root = ReactDOM.createRoot(container);
      root.render(<Hello />)
    </script>

  </body>
</html>
```

React 개발 환경 구성

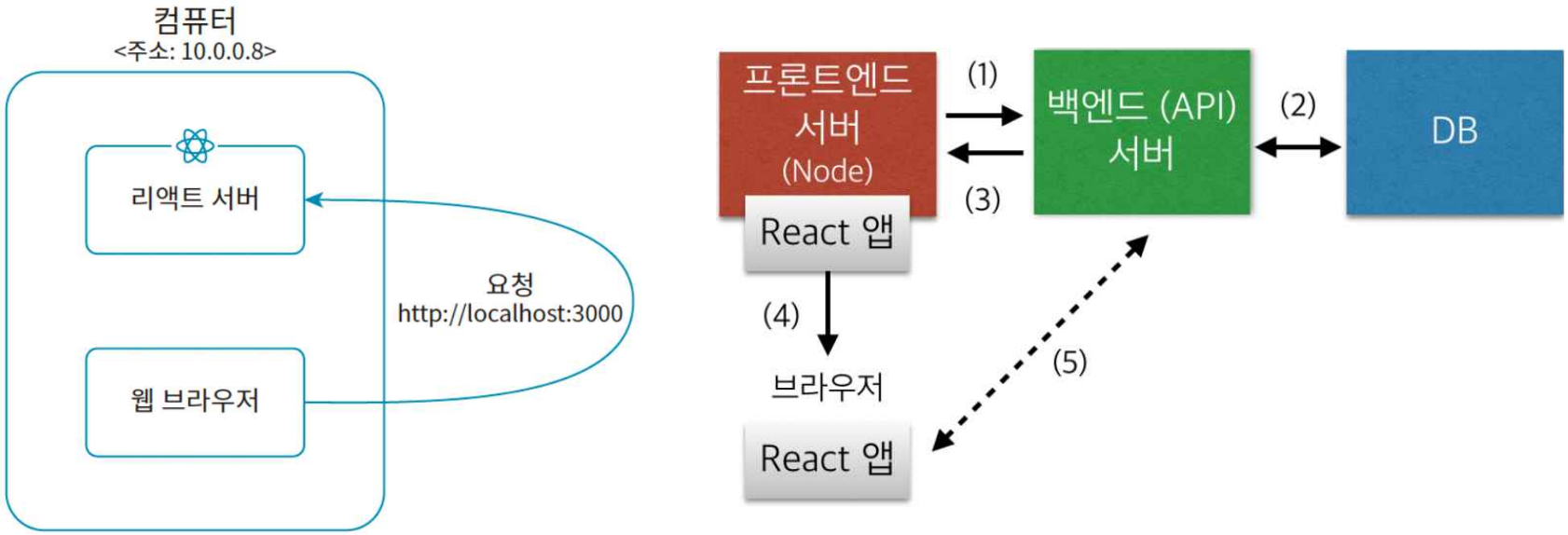
- HTML 파일에서 React실행

React 스크립트	역할
React 라이브러리 (react.development.js)	React의 핵심 기능을 제공 컴포넌트 생성, 상태 관리, 가상 DOM 처리 등의 기능 포함 개발용(Development) 버전으로, 경고 메시지와 디버깅 기능 포함
ReactDOM 라이브러리 (react-dom.development.js)	React 컴포넌트를 실제 DOM에 렌더링하는 기능을 제공 브라우저 환경에서 ReactDOM.createRoot() 등을 사용하여 React 요소를 마운트 개발용(Development) 버전으로 디버깅 기능이 활성화
Babel (babel.min.js)	JSX 문법을 일반적인 JavaScript 코드로 변환하는 역할을 합니다. JSX를 사용하여 <script type="text/babel"> 태그 안에 작성된 코드를 변환하여 실행할 수 있도록 합니다. React의 최신 문법(예: 화살표 함수, ES6+ 문법)도 변환

- createRoot()는 React의 새로운 동시성 렌더링(Concurrent Mode) 기능을 활성화 합니다.
- render()는 React.createElement() 또는 JSX로 만든 컴포넌트를 container에 기존 DOM을 지우고 새로운 React 요소를 마운트하고 렌더링 합니다.

React의 핵심

- React Application 프로젝트의 진입점 src/index.js
- import React from 'react'; 는 JSX라는 문법을 사용하기 위한 필수 선언
- 'react-dom'은 컴포넌트를 화면에 그리기 위한 라이브러리
- 열어놓은 태그는 꼭 닫아야 합니다
- 최상위 태그는 꼭 1개여야 합니다.
- 컴포넌트 이름의 첫 번째 글자는 반드시 대문자여야 합니다.
- import 문에서 파일 이름의 확장자를 생략해도 해당 파일을 자동으로 찾을 수 있게 설정되어 있습니다. (webpack modeul resolution)



React의 핵심

- 컴포넌트(Component) : 함수형 컴포넌트 vs 클래스형 컴포넌트
- JSX(JavaScript XML) : HTML과 유사한 문법을 JavaScript에서 사용 가능
- Props (Properties) : 부모 → 자식으로 전달되는 데이터 (읽기 전용)
- State : 컴포넌트 내부에서 관리하는 동적인 데이터, useState Hook 사용
- 이벤트 핸들링 & 조건부 렌더링

Angular.js 프레임워크 :
구글에 의해 개발된 웹 또는 앱을 만드는 데 필요한 모든 것이 갖춰진 프레임워크
작은 컨테이너들이 모여 거대한 앱을 구성하도록 설계됨
양방향 데이터 바인딩을 사용하여 모델과 뷰가 자동으로 동기화됨
데이터 변경이 발생할 때마다 "소화 과정(Digest Cycle)" 이 실행되면서 전체 스코프 트리가 검사됨.
모든 바인딩을 감시($watch)하며 변경을 추적.
대형 애플리케이션에서는 많은 Watcher가 생성되어 렌더링 속도가 느려짐.
DI를 활용하여 모듈 간 의존성을 관리하지만, 설정 및 사용 방식이 복잡하여 디버깅이 어려움.
$scope 객체를 통해 데이터 바인딩을 관리하지만, 부모-자식 컴포넌트 간 데이터 흐름이 직관적이지 않음.

React의 핵심


```
import React from "react";
// console.log(React);
```

- React는 react 모듈에서 기본(default)으로 export된 객체

```
import { useState, useEffect } from "react";
console.log(typeof useState); // "function"
console.log(useState); // function useState() { ... }
```

- 함수형 컴포넌트에서 상태(state)와 라이프사이클(lifecycle) 기능을 사용할 수 있도록 useState와 useEffect 등의 Hooks를 제공
- {}를 사용하면 React.useState()가 아닌 useState()만으로 사용할 수 있음
- {} (중괄호)는 "구조 분해 할당(Destructuring)"을 위한 문법

```
import React from "react";

React.useState(); // React 객체에서 직접 useState 사용
React.useEffect();
```

React의 핵심

- React 컴포넌트
  - UI 요소를 표현하는 최소한의 단위, 화면의 특정 부분이 어떻게 생길지 정하는 선언체

```
//DOM을 직접 제어 (명령형 프로그래밍)
const root = document.getElementById('root');
const header = document.createElement('h1');
const headerContent = document.createTextNode(
            'Hello, World!'
);

header.appendChild(headerContent);
root.appendChild(header);
```

```
//UI을 선언하고 render 함수를 호출하면 React가 화면에 출력
const header = <h1>Hello World</h1>; // jsx
ReactDOM.render(header, document.getElementById('root'));
```

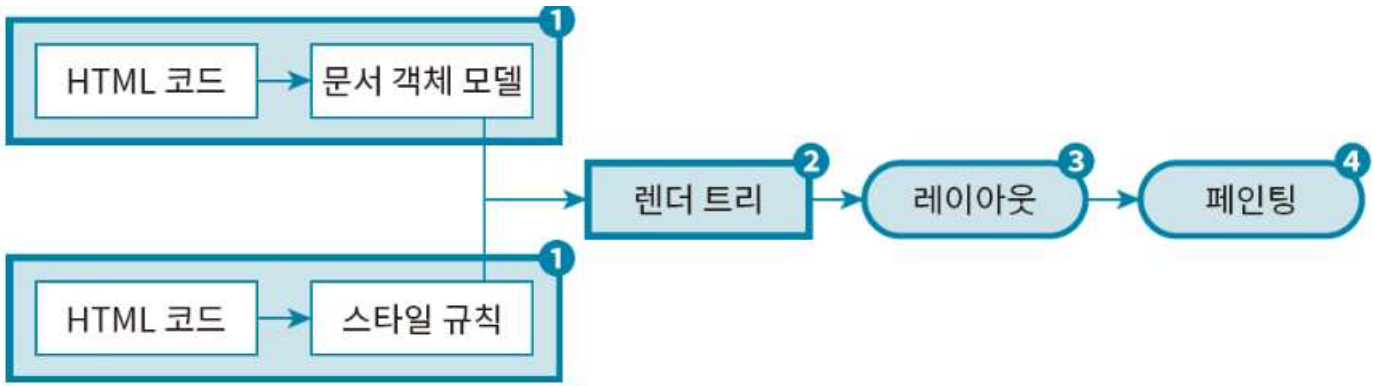
React의 핵심

- React 컴포넌트의 render()
  - JSX를 사용하여 반환할 UI를 선언
  - 컴포넌트로 전달된 데이터는 render() 안에서 this.state와 this.props 를 통해 접근할 수 있습니다.
  - 데이터를 입력 받아 화면에 표시할 내용을 반환
  - 컴포넌트의 상태나 props를 변경하지 않으며, 항상 동일한 입력에 대해 동일한 출력을 반환하는 순수 함수로 정의
  - 상태나 props를 변경하지 않음
  - 반드시 하나의 루트 요소(root element)를 반환해야 합니다.
  - 여러 개의 엘리먼트를 반환하려면 div, React.Fragment, 또는 배열을 사용하여 감싸야 합니다.

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

root.render( <HelloMessage name="Taylor" />);
```

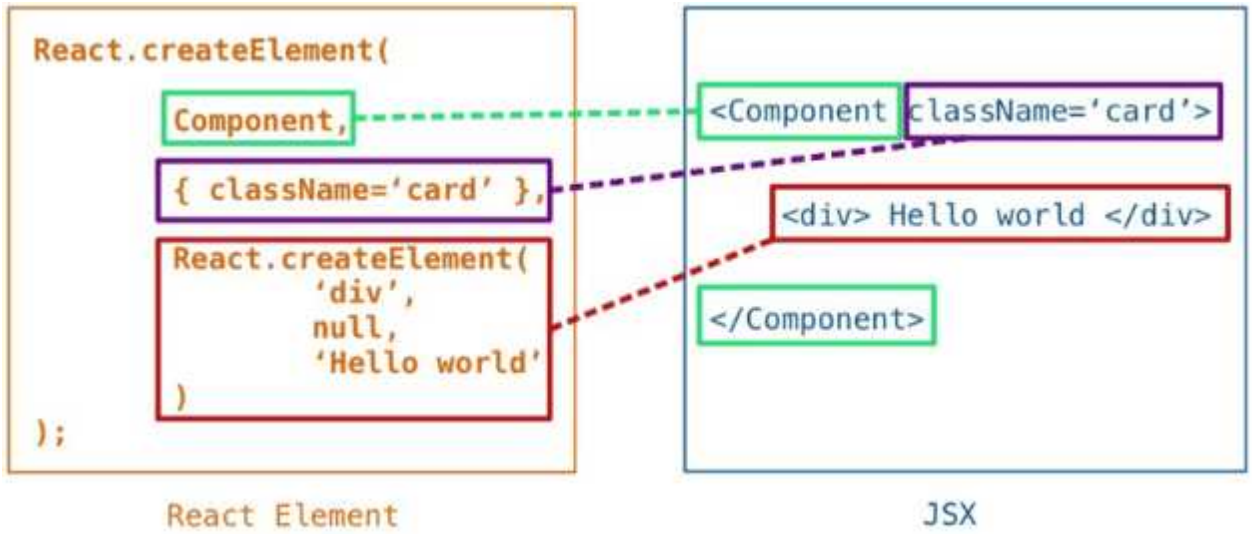
React의 핵심

- React 컴포넌트의 render()
  - React의 상태(state)가 변경될 때마다 호출됩니다. (shouldComponentUpdate 메서드를 사용하거나 React.memo를 활용해 불필요한 리렌더링을 방지할 수 있습니다.)
  - 여러 엘리먼트를 배열 형태로 반환할 수 있으며, 각 엘리먼트는 고유한 key prop을 가져야 합니다.
  - 컴포넌트를 렌더링하지 않도록 할 때 null이나 false를 반환할 수 있습니다.
  - 함수 내에서 다른 컴포넌트를 호출하여 UI를 구성하는 방식으로, 컴포넌트를 계층적으로 관리할 수 있습니다.
  - 예외 처리나 디버깅을 위한 로직을 최소화하여 작성하는 것이 좋습니다.
  - 현재 리액트 내부에 어떤 상태(state)에 변경이 발생했을 때, 컴포넌트에 새로운 props가 들어올 때, 리렌더링이 발생
  - 여러 상태가 변경됐다면 리액트는 이를 큐 자료구조에 넣어 순서를 관리합니다.



Javascript

```
<Component className="card">
    <div>Hello world</div>
</Component>
```



JSX

- JSX(JavaScript XML) : React에서 UI를 선언적으로 표현하는 방식
- HTML 요소를 JavaScript로 작성하여 createElement() 또는 appendChild() 메서드 없이 DOM에 배치할 수 있습니다
- JSX는 브라우저에서 직접 실행되지 않고, Babel을 통해 JavaScript로 변환됨
- JSX는 React.createElement() 호출로 변환

```
//JSX코드
const myElement = <h1>I Love JSX!</h1>;

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

```
//JavaScript 코드
const myElement = React.createElement('h1', {}, 'I do not use JSX!');

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

Babel은 JavaScript(ES6 이상) 코드나 JSX를 이전 버전(ES5)으로 변환하여 모든 환경에서 실행할 수 있도록 변환하는 JavaScript 컴파일러입니다.

JSX

- JSX 내부에서 중괄호 {}를 사용하여 변수, 표현식을 활용할 수 있음.

```
const name = "React";
const element = <h1>Hello, {name}!</h1>;
```

- HTML 속성을 사용할 때 CamelCase 방식으로 작성해야 함

```
const element = <h1 className="title">JSX 속성 사용</h1>;
const imageUrl = "https://via.placeholder.com/150";
const element = <img src={imageUrl} alt="Sample Image" />;
```

- XML 규칙을 따르므로 HTML 요소를 닫아야 합니다
- HTML 태그 속성 class는 키워드이므로 className 속성을 사용합니다

JSX

- JSX 내부에서는 if 문을 지원되지 않습니다.
- JSX에서 조건문을 사용하려면 if 문을 JSX 외부에 배치하거나 내부에서 삼항식을 사용할 수 있습니다
- JSX 내부에서 조건문을 사용할 때는 {} 내부에 표현식으로 작성

```
const x = 5;

const element = <p> { (x) < 10  && "새로운 메시지가 있습니다!"} </p>;
```

```
const x = 5;
let text = "Goodbye";
if (x < 10) {
  text = "Hello";
}

const myElement = <h1>{text}</h1>;
```

```
const x = 5;

const myElement = <h1>{(x) < 10 ? "Hello" : "Goodbye"}</h1>;
```

JSX

- if(조건) return (실행문)을 사용하여 복수 조건을 처리합니다.

```
import React, { Component, Fragment } from 'react';
class App extends Component {
  render() {
     const value = 1;
     return (
        <div>
              {
                (function(){
                        if(value===1) return ( <div>하나</div> );
                        if(value===2) return ( <div>둘</div> );
                        if(value===3) return ( <div>셋</div> );
                  })()
              }
        </div>
        );
  }
}
export default App;
```

JSX

- switch 구문은 화살표함수를 사용하여 복수 조건을 처리합니다.

```
import React, { Component, Fragment } from 'react';
class App extends Component {
  render() {
    const value = 1;
    return (
      <div>
        {    /* 주석내용 */
          (() => {
                  if(value===1) return ( <div>하나</div> );
                  if(value===2) return ( <div>둘</div> );
                  if(value===3) return ( <div>셋</div> );
                })()
            }
      </div>
      );
  }
}
export default App;
```

JSX

- 배열 데이터를 JSX에서 리스트 렌더링할 때 map을 사용

```
const items = ["React", "JSX", "Props", "State"];
const list = items.map( (item) => <li key={item}> {item} </li> );
const element = <ul>{list}</ul>;
```

JSX

- key는 React에서 리스트의 각 항목을 식별하는 고유한 값으로, key를 올바르게 설정하면 리스트의 전체 항목을 다시 렌더링하지 않고 변경된 부분만 업데이트할 수 있습니다.

```
function ItemList() {
  const [items, setItems] = useState([
    { id: 1, name: "Apple" },
    { id: 2, name: "Banana" },
    { id: 3, name: "Cherry" },
  ]);
const removeItem = (id) => {          // 리스트에서 항목 제거
    setItems(items.filter((item) => item.id !== id));
  };
  return (
    <div>    <h3>Fruit List</h3>
      <ul>
        {items.map((item) => (
          <li  key={item.id}>
            {item.name} <button onClick={ () => removeItem(item.id) }>삭제</button>
          </li>
        ))}
      </ul>    </div>
  );
}
```

JSX

- JSX 내부의 복수의 div를 감싸는 tag는 Fragment를 사용
- HTML 코드는 하나의 최상위 요소로 감싸야 합니다.
- HTML을 여러 줄로 작성하려면 HTML을 괄호 안에 넣습니다

```
import React, { Component, Fragment } from 'react';
class App extends Component {
  render() {
    return  (
              <Fragment>
                      <div>     Hello      </div>
                      <div >    Hello      </div>
               </Fragment>
    );
  }
}
export default App;
```

JSX

- Reactjs JSX에서 CSS 사용
  - const css설정이름={}를 사용하여 css객체를 정의하고 return문의 tag안에 style={css설정이름}으로 CSS 적용합니다.

```
import React, { Component, Fragment } from 'react';
class App extends Component {
 render() {
     const style = {
         backgroundColor:'black',
         padding:'16px',
         color:'white',
         fontSize:'36px'
         };
   return <div style={style}>안녕하세요!</div>;
   }
}
export default App;
```

React CSS 스타일 적용

- inline styling : 몇 가지 스타일 속성 만 추가하려는 경우
- JavaScript Object로 정의하고 element의 style 속성에 설정합니다
- CSS속성 키의 – 을 camel case 문법을 사용합니다 (background-color => backgroundColor)

```
const Header = () => {
  return (
    <>
      <h1 style={{color: "red"}}>Hello Style!</h1>
      <p>Add a little style!</p>
    </>
  );
}
```

React CSS 스타일 적용

- regular CSS stylesheets : 응용 프로그램이 복잡한 경우
- .css 파일을 생성하고 스타일을 작성하고 import 키워드를 사용하여 해당 파일을 컴포넌트에서 불러옵니다
- 모든 컴포넌트에서 해당 스타일이 글로벌하게 적용됩니다.

```
/* App.css */
.container {
  background-color: lightblue;
  padding: 20px;
  text-align: center;
}
```

```
/* App.js */
import React from "react";
import "./App.css";  // CSS 파일을 import

function App() {
  return <div className="container">Hello, CSS Stylesheets!</div>;
}

export default App;
```

React CSS 스타일 적용

- CSS Modules : CSS를 컴포넌트 단위로 캡슐화하여 전역 오염을 방지하는 방식
- .module.css 파일을 생성
- import styles from "파일명.module.css" 형식으로 불러오기
- className={styles.클래스명} 형태로 적용
- CSS 클래스가 컴포넌트별로 격리되어 있어 스타일 충돌이 없음

```
/* Button.module.css*/
.button {
  background-color: blue;
  color: white;
  padding: 10px 20px;
  border: none;
  cursor: pointer;
}
```

```
/* Button.js*/
import React from "react";
import styles from "./Button.module.css";            // CSS 모듈 import

function Button() {
  return <button className={styles.button}>Click Me</button>;
}

export default Button;
```

React CSS 스타일 적용

- Styled Components (CSS-in-JS) : CSS를 JavaScript 안에서 작성하는 방법으로, 컴포넌트 기반 스타일링을 지원
- JavaScript 코드 안에서 조건부 스타일링이 가능
- props를 활용한 동적 스타일 적용 가능
- styled-components 패키지 설치

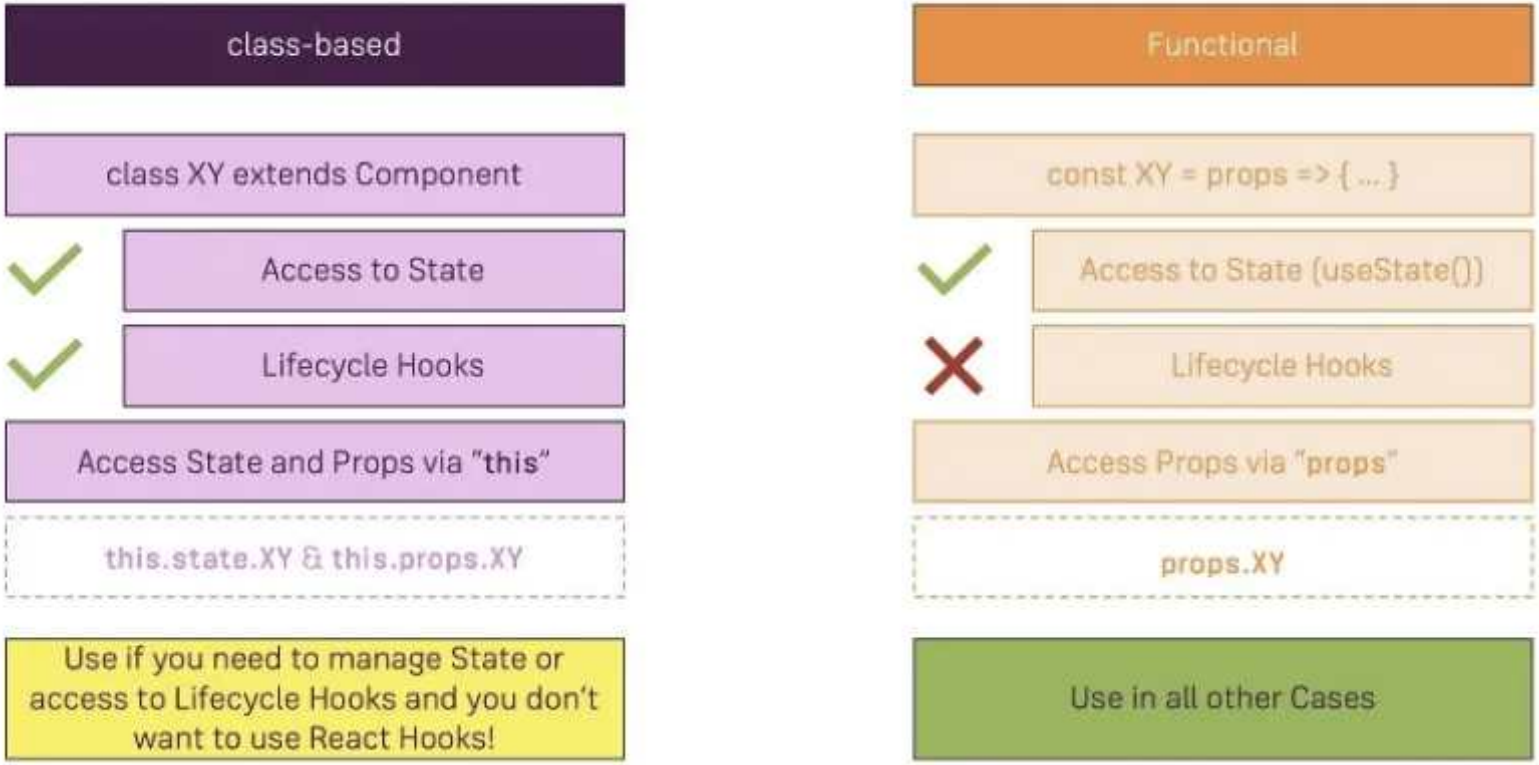
```
npm install styled-components
```

```
/* Button.js*/
import React from "react";
import styled from "styled-components"; // styled-components import

const StyledButton = styled.button`
  background-color: green;
  color: white;
  padding: 10px 20px;
  border: none;
  cursor: pointer;
  border-radius: 5px;
`;
function Button() {
  return <StyledButton>Click Me</StyledButton>;
}
export default Button;
```

JSX

JSX 문법을 활용하여 작은 React 컴포넌트 구현 실습
- 사용자 이름 입력받기 (prompt)
- 좋아하는 색상 입력받기 (prompt)
- 입력된 정보를 화면에 출력 (JSX 활용)

Class-based vs Functional Components



React Component

- React Component는 HTML 요소를 반환하는 함수와 같습니다.
- Component의 재사용성을 높이기 위해 .js 파일로 저장할때 Component단위로 저장하고 파일이름은 대문자로 시작해야 합니다.
- React 16.8에서 추가된 Hooks와 함께 Function Component를 사용하는 것이 권장됩니다
- React Component의 이름은 대문자로 시작해야 합니다

```
/*  "Car.js" */
function Car() {
  return <h2>Hi, I am a Car!</h2>;
}

export default Car;
```

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import Car from './Car.js';

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

React Component

- 함수형 컴포넌트 : 상태 관리와 Side Effect(렌더링 과정과 직접 관련되지 않은 동작)를 위해 React Hooks를 사용 (React 16.8 이후 Hooks가 도입되면서 함수형 컴포넌트가 더 많이 사용됨)
- 클래스형 컴포넌트 : 상태 관리와 Side Effect 를 위해 state와 lifecycle methods를 사용

비교 항목	함수형 컴포넌트	클래스형 컴포넌트
선언 방식	function 키워드 사용 JavaScript 순수 함수 React 요소(JSX)를 반환하는 함수	class 키워드 사용 React.Component의 확장 React 요소를 반환하는 렌더링 함수 정의
state 사용	useState() 훅 사용	this.state 사용
lifecycle 메서드	useEffect()로 대체	componentDidMount(), componentDidUpdate(), componentWillUnmount() 등 사용
props 접근 방식	props 매개변수 사용	this.props 사용
this 사용 여부	this 없음	this 사용

React Component

- Class Component는 extends React.Component 로 상속을 반드시 선언하며 React.Component의 모든 기능을 액세스할 수 있습니다.
- HTML을 반환하는 render() 메서드를 정의해야 합니다

```
class Car extends React.Component {
  render() {
    return <h2>Hi, I am a Car!</h2>;
  }
}
```

React Component

- props는 속성(property)으로 함수의 파라미터처럼 Component에 전달됩니다.
- props는 HTML 속성을 통해 모 컴포넌트에서 자식 컴포넌트로 전달하는 읽기 전용 데이터 입니다.

```
function Car(props) {
  return <h2>I am a {props.color} Car!</h2>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car color="red"/>);
```

React Component

- 다른 component 내부에서 component를 참조할 수 있습니다.

```
function Car() {
  return <h2>I am a Car!</h2>;
}

function Garage() {
  return (
    <>
      <h1>Who lives in my Garage?</h1>
      <Car />
    </>
  );
}

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

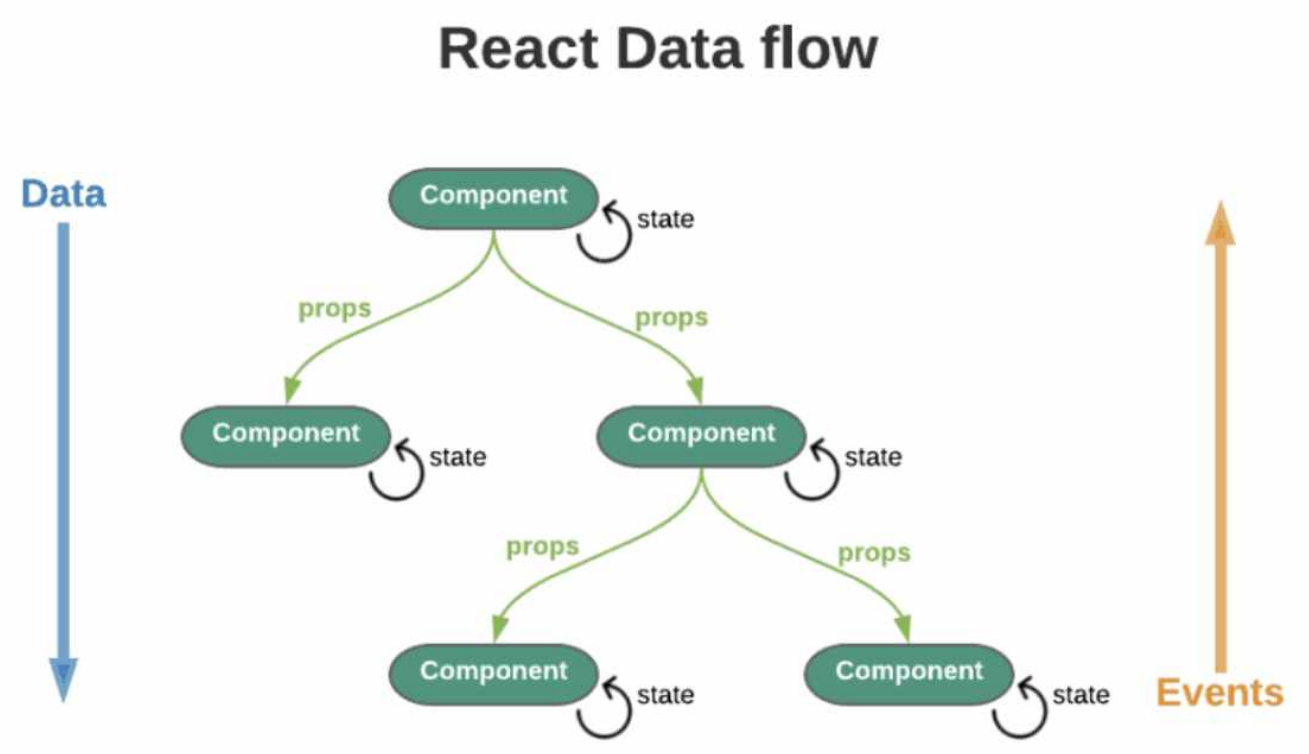
React Component

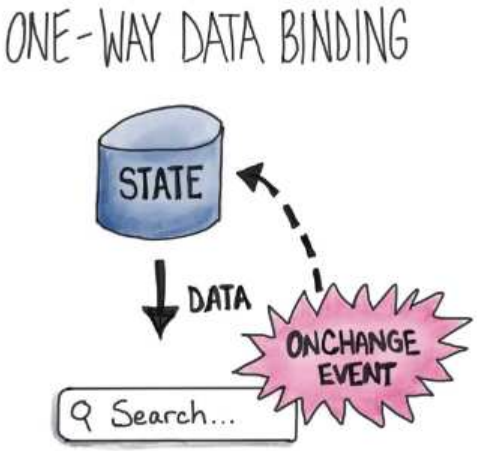
- state는 컴포넌트 내부에서 관리하는 변경 가능한 데이터로 해당 컴포넌트 내부에서 관리하며 state가 변경되면 컴포넌트가 다시 렌더링됩니다
- Constructor() 또는 useState() 를 사용한 state정의하고 setState(), usetState Hook을 사용하여 변경 할 수 있습니다.

```
import React, { Component } from 'react';
class Counter extends Component {
    constructor(props) {
        super(props);
        this.state = {
             number: 0
            }
      }
     handleIncrease = () => {
            this.setState({
                  number: this.state.number + 1
                       });
             }
     handleDecrease = () => {
           this.setState({
                  number: this.state.number - 1
                       });
            }
```

React Component


```
 render() {
     return (
             <div>
             <h1> 카운터 </h1>
             <div> 값 : {this.state.number} </div>
             <button onClick={this.handleIncrease}>+</button>
             <button onClick={this.handleDecrease}>-</button>
             </div>
       );
     }
}

export default Counter;
```


React Data flow
Data
Component
state
props
props
Component
state
Component
state
props
props
Component
state
Component
state
Events


ONE-WAY DATA BINDING
STATE
DATA
ONCHANGE EVENT
Search...

Component 상태 관리

- 함수형 컴포넌트에서 useState는 상태(state)를 관리하는 Hook
- 클래스형 컴포넌트의 this.state와 setState()를 대체
- 배열 구조 분해(destructuring)를 사용하여 [값, setter 함수]를 반환

```
const [state, setState] = useState(초기값);
```

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

function Counter() {
  const [count, setCount] = useState(0); // 초기값 0

  return (
    <div>
      <h1>카운트: {count}</h1>
      <button onClick={() => setCount(count + 1)}>증가</button>
    </div>
  );
}

export default Counter;
```

Component 상태 관리

- React에서 상태(state)를 직접 수정하면 리렌더링이 발생하지 않습니다

```
state.count = 10;     // 직접 수정
setCount(10);         // React가 감지하고 자동으로 리렌더링
```

- state로 저장된 배열의 요소를 변경하고 리렌더링이 되게 하려면, 기존 배열을 직접 수정하지 않고 새로운 배열을 만들어 setState 해야 합니다.
- push, splice, unshift, pop 같은 내장함수는 배열 자체를 직접 수정하게 되므로 적합하지 않습니다.
- 기존의 배열에 기반하여 새 배열을 만들어내는 함수인 concat, slice, map, filter 같은 함수, …spread 연산자를 사용해야 합니다.

```
const a = [1, 2, 3, 4, 5];
const b = [];
b.forEach(number => b.push(number * 2));
```

```
const a = [1, 2, 3, 4, 5];
const b = a.mpa(number =>  number * 2);
```

Component 상태 관리

- state에 저장된 배열의 요소 제거 :
  1. 첫번째 방법은 slice 와 concat 을 이용
  2. 배열 전개 연산자(...)를 사용

```
array.slice(0, 2).concat(array.slice(3, 5))
```

```
[ ...array.slice(0,2), ...array.slice(3, 5 )];
```

Component 상태 관리

- useEffect는 함수형 컴포넌트의 라이프사이클을 관리하는 Hook
- 클래스형 컴포넌트의 componentDidMount, componentDidUpdate, componentWillUnmount를 대체
- API 호출, 이벤트 리스너 등록, 타이머 설정 등의 부작용(side effects) 처리

```
useEffect(() => {
  // 실행할 코드
}, [의존성]);
```

✔ 의존성 배열([])에 값이 변경될 때마다 effect 실행
✔ 빈 배열([]) → 처음 마운트될 때만 실행
✔ 의존성 없이 사용하면 → 매번 렌더링될 때 실행됨 (비효율적 주의!)

Component 상태 관리

- 함수형 Component의 상태 관리

```
//컴포넌트가 마운트될 때 한번 실행
import React, { useState, useEffect } from "react";

function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log("컴포넌트가 처음 마운트됨!");
  }, []); // [] 빈 배열 → 최초 1회 실행

  return (
    <div>
      <h1>카운트: {count}</h1>
      <button onClick={() => setCount(count + 1)}>증가</button>
    </div>
  );
}

export default Timer;
```

Component 상태 관리

- 함수형 Component의 상태 관리

```
//특정 값이 변경될 때 실행
import React, { useState, useEffect } from "react";

function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
  console.log(`카운트 변경됨: ${count}`);
}, [count]); // count가 변경될 때만 실행

  return (
    <div>
      <h1>카운트: {count}</h1>
      <button onClick={() => setCount(count + 1)}>증가</button>
    </div>
  );
}

export default Timer;
```

Component 상태 관리

- 함수형 Component의 상태 관리

```
//컴포넌트가 언마운트될 때 정리 (clean-up)
import React, { useState, useEffect } from "react";

function TimerComponent() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setCount((prevCount) => prevCount + 1);
    }, 1000);

    return () => {
      console.log("타이머 정리됨!");
      clearInterval(interval);
    };
  }, []);     // 최초 실행 후 언마운트 시 정리

  return <h1>타이머: {count}초</h1>;
}

export default TimerComponent;
```

React Component 상태관리

- 클래스형 Component의 상태 관리
  - 클래스 Component는 내부적인 상태 데이터를 this.state로 접근할 수 있고 상태 데이터가 바뀌면 render()가 다시 호출되어 리렌더링됩니다.

```
class Timer extends React.Component {
   constructor(props) {
      super(props);
      this.state = { seconds: 0 };
  }

  tick() {
     this.setState(state => ({
        seconds: state.seconds + 1
    }));
  }

}
```

React Component 상태 관리

- 클래스형 Component의 상태

```
  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>
        Seconds: {this.state.seconds}
      </div>
    );
  }
}

root.render(<Timer />);
```

Component 상태 관리

- **로컬 상태 관리** : useState와 useReducer 같은 내장된 Hooks를 사용
- **전역 상태 관리** : Context API를 사용하거나, Redux, MobX, Recoil 같은 외부 라이브러리를 사용

```
//useState는 컴포넌트 내부에서 상태를 관리할 때 사용
import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>현재 카운트: {count}</p>
      <button onClick={() => setCount(count + 1)}>증가</button>
      <button onClick={() => setCount(count - 1)}>감소</button>
    </div>
  );
}

export default Counter;
```

Component 상태 관리


```
//useReducer는 복잡한 상태 관리(여러 개의 상태, 상태 변경 로직이 복잡한 경우)에 사용
import { useReducer } from "react";
const initialState = { count: 0 };
function reducer(state, action) {
  switch (action.type) {
    case "increment":
      return { count: state.count + 1 };
    case "decrement":
      return { count: state.count - 1 };
    default:
      return state;
  }
}
function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <div>
      <p>현재 카운트: {state.count}</p>
      <button onClick={() => dispatch({ type: "increment" })}>증가</button>
      <button onClick={() => dispatch({ type: "decrement" })}>감소</button>
    </div>
  );
}
export default Counter;
```

Component 상태 관리

- props(Properties) : 부모 컴포넌트에서 자식 컴포넌트로 전달하는 읽기 전용 (Read-Only, 변경 불가) 데이터
- props는 부모 컴포넌트가 관리하며, 변경되면 컴포넌트가 다시 렌더링됨
- 객체 형태로 전달됨
- 부모 컴포넌트에서 Child 컴포넌트로 초기 데이터, 설정 값을 전달에 사용

```
<자식컴포넌트 속성이름="값" />
```

```
//부모 컴포넌트에서 Props 전달
import React from "react";
import Greeting from "./Greeting";

function App() {
  return <Greeting name="홍길동" />;
}

export default App;
```

```
//자식 컴포넌트에서 props를 받아 출력
import React from "react";

function Greeting(props) {
  return <h1>안녕하세요, { props.name }
님!</h1>;
}

export default Greeting;
```

Component 상태 관리

- props 여러 개 전달
- {}를 사용하여 숫자와 같은 JavaScript 표현식도 전달 가능

```
//Props는 변경할 수 없다 (Read-Only)
import React from "react";

 function Greeting(props) {
  props.name = "변경된 값"; //  오류 발생
  return <h1>안녕하세요, {props.name}님!</h1>;
}
export default Greeting;
```

```
function UserInfo(props) {
  return (
    <div>
      <h1>이름: {props.name}</h1>
      <p>나이: {props.age}</p>
    </div>
  );
}
function App() {
  return <UserInfo name="홍길동" age={25} />;
}
```

Component 상태 관리

- 변수를 props로 전달

```
function Car(props) {
  return <h2>I am a { props.brand }!</h2>;
}

function Garage() {
  const carName = "Ford";
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <Car brand={ carName } />
    </>
  );
```

Component 상태 관리

- 객체를 props로 전달

```
//객체를 Props로 전달
function Car(props) {
  return <h2>I am a { props.brand.model }!</h2>;
}

function Garage() {
  const carInfo = { name: "Ford", model: "Mustang" };
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <Car brand={ carInfo } />
    </>
  );
}
```

Component 상태 관리

- Props의 기본값 설정 (defaultProps)

```
function Greeting(props) {
  return <h1>안녕하세요, {props.name}님!</h1>;
}

Greeting.defaultProps = {
  name: "손님",
};

function App() {
  return <Greeting />;      // name을 전달하지 않아도 기본값 '손님' 사용
}
```

- Props를 구조 분해 할당 (Destructuring)으로 받기

```
function Greeting({ name }) {
  return <h1>안녕하세요, {name}님!</h1>;
}
```

Component 상태 관리

- React는 prop-types 라이브러리(v15.5부터 별도의 라이브러리로 분리)의 propTypes 라는 기능을 통해 타입체크를 지원합니다.
- PropTypes는 전달받은 데이터의 유효성을 검증하기 위해서 다양한 유효성 검사기 (Validator)를 내보냅니다.
- propTypes는 성능상의 이유로 개발 모드(Development mode) 에서만 확인됩니다

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

class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}
Greeting.propTypes = {
  name: PropTypes.string
};
```

Component 상태 관리

- prop-types 패키지를 사용하면 props의 자료형을 검사할 수 있습니다.
- propTypes를 활용하면 props의 자료형을 검증할 수 있으며, 잘못된 타입을 전달하면 콘솔에서 경고가 표시됩니다.

```
import React from "react";
import PropTypes from "prop-types";
const Greeting = ({ name, age, isMember }) => {
  return (
    <div>
      <h1>Hello, {name}!</h1>
      <p>Age: {age}</p>
      <p>Membership: {isMember ? "Active" : "Inactive"}</p>
    </div>
  );
};
// props의 자료형 선언
Greeting.propTypes = {
  name: PropTypes.string.isRequired,  // 문자열 (필수)
  age: PropTypes.number,              // 숫자
  isMember: PropTypes.bool            // 불리언
};
// 기본 props 값 설정
Greeting.defaultProps = {
  age: 20,
  isMember: false
};
export default Greeting;
```

Component 상태 관리

- PropTypes.element를 이용하여 컴포넌트의 자식들(Children)에 단 하나의 자식(Child)만이 전달될 수 있도록 제한 할 수 있습니다.
- 여러 개의 요소가 전달되면 오류가 발생합니다.
- string, number 같은 기본 자료형은 element 타입이 아니므로 허용되지 않습니다.
- 여러 개의 요소가 필요하면 React.Fragment 또는 div로 감싸야 합니다.

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

const SingleChildComponent = ({ children }) => {
  return <div className="container">{children}</div>;
};

// children이 단 하나의 React 요소만 가능하도록 설정
SingleChildComponent.propTypes = {
  children: PropTypes.element.isRequired,
};
```

Component 상태 관리

- PropTypes.element

```
export default function App() {
  return (
    <div>
      { /* 올바른 사용 (하나의 요소만 전달) */ }
      <SingleChildComponent>
        <p>This is a single child element.</p>
      </SingleChildComponent>

      { /* 잘못된 사용 (여러 개의 요소 전달) - 콘솔 경고 발생 */ }
      <SingleChildComponent>
        <p>Child 1</p>
        <p>Child 2</p>
      </SingleChildComponent>
    </div>
  );
}
```

Component 상태 관리

- defaultProps : props의 일부 속성에 기본값을 설정
- 함수형 component에서는 class 밖에 class와 export문 사이에 독립적으로 defaultProps를 설정합니다

```
import React, { Component } from 'react';
class MyWork extends Component {
  static defaultProps = {
      name: '작품명이 없습니다.'
  }
 render() {
   return (
       <div> 작품명은 <b> {this.props.name} </b> 입니다. </div>
     );
   }
}
export default MyWork;
```

```
import React, { Component } from 'react';
const MyWork = ({name}) => {
    return (
      <div> 작품명은 <b> {name} </b> 입니다. </div>
       );
    }
MyWork defaultProps = {
       name: '작품명이 없습니다.'
  }
export default MyWork;
```

Component 상태 관리

- forceUpdate() : 클래스형 컴포넌트에서 강제로 리렌더링을 수행하는 메서드
- forceUpdate는 상태를 변경하지 않고 강제로 리렌더링만 수행합니다.
- React는 상태(State)나 Props가 변경될 때 자동으로 컴포넌트를 리렌더링하기 때문에, forceUpdate를 사용할 필요가 거의 없습니다.

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

class MyComponent extends Component {
  forceRerender = () => {
    this.forceUpdate(); // 강제 리렌더링
  };

  render() {
    console.log("Component is rendering...");
    return (
      <div>
        <h1>Hello, React!</h1>
        <button onClick={this.forceRerender}>Force Update</button>
      </div>
    );
  }
}

export default MyComponent;
```

Component 상태 관리

- 함수형 컴포넌트에서는 useState나 useReducer를 활용하여 리렌더링 유도할 수 있습니다

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

const MyComponent = () => {
  const [, setRerender] = useState(0);

  const forceRerender = () => {
    setRerender((prev) => prev + 1); // 상태 변경 -> 리렌더링 발생
  };

  console.log("Component is rendering...");

  return (
    <div>
      <h1>Hello, React!</h1>
      <button onClick={forceRerender}>Force Update</button>
    </div>
  );
};

export default MyComponent;
```

Component 상태 관리

- useReducer를 활용하면 불필요한 상태값 없이도 리렌더링을 수행할 수 있습니다.

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

const MyComponent = () => {
   const [, forceUpdate] = useReducer((x) => x + 1, 0);

  console.log("Component is rendering...");

  return (
    <div>
      <h1>Hello, React!</h1>
      <button onClick={forceUpdate}>Force Update</button>
    </div>
  );
};

export default MyComponent;
```

Component 상태 관리

- 함수형 컴포넌트에서는 useState나 useReducer를 활용하여 리렌더링 유도할 수 있습니다
- React에서 관리하지 않는 외부 데이터가 변경될 때 [예) 전역 객체(window, localStorage, etc.)가 변경] 강제 리렌더링이 필요할 수 있습니다.

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

const MyComponent = () => {
  const [time, setTime] = useState(new Date().toLocaleTimeString());

  useEffect(() => {
    const interval = setInterval(() => {
      setTime(new Date().toLocaleTimeString()); // 상태 변경으로 리렌더링 유도
    }, 1000);
    return () => clearInterval(interval);
  }, []);

  return <h1>Current Time: {time}</h1>;
};

export default MyComponent;
```

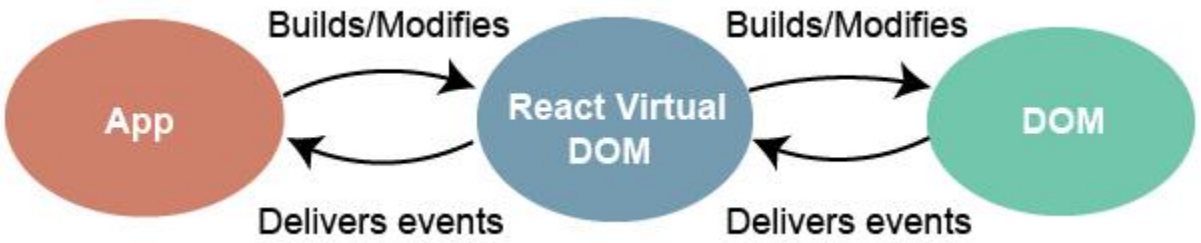
React Hook

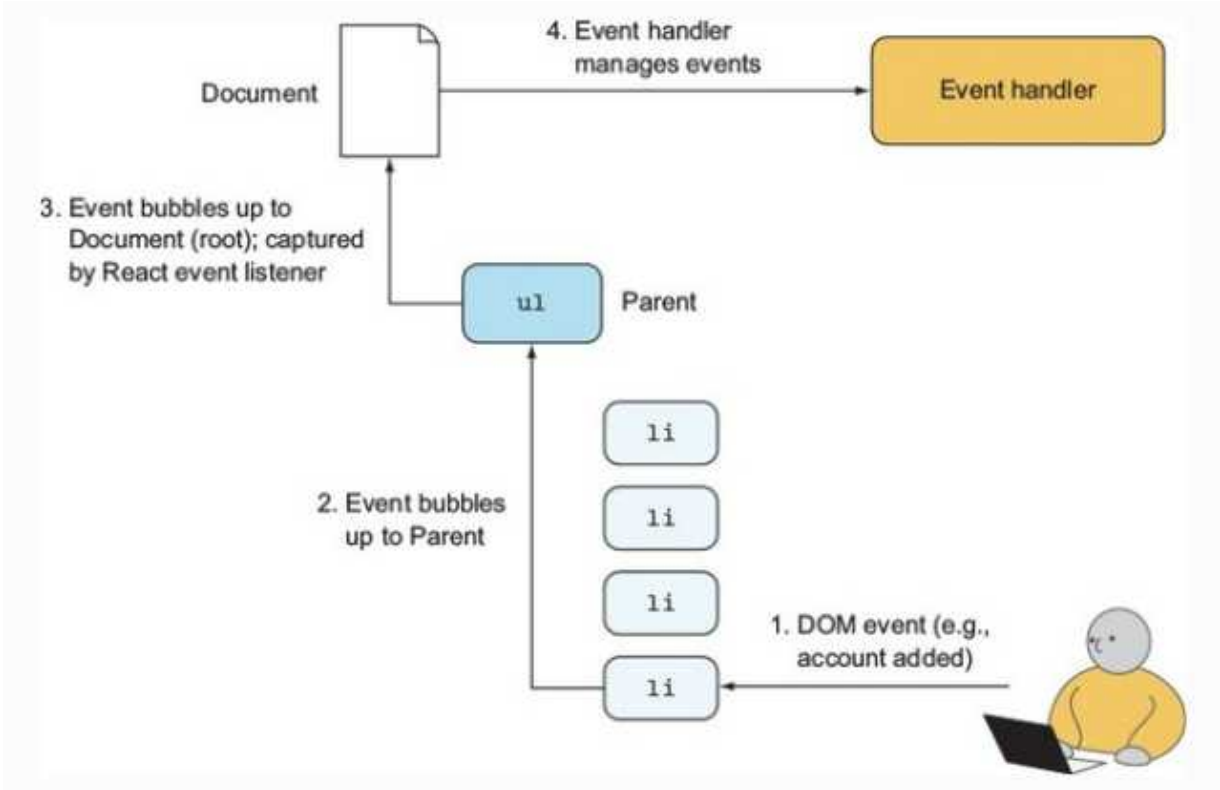
- Hook : React의 함수형 컴포넌트에서 상태(state)와 라이프사이클(lifecycle) 기능을 사용할 수 있도록 해주는 기능
- 클래스형 컴포넌트에서는 state와 라이프사이클 메서드를 사용해야만 상태 관리를 할 수 있었음
- 클래스형 컴포넌트는 코드가 길어지고, 복잡한 로직을 여러 곳에서 중복해서 사용해야 하는 경우 유지보수가 어려워져서 함수형 컴포넌트에서도 상태 관리가 가능하도록 Hook이 도입됨 (React 16.8부터 사용 가능)
- Hook은 함수 Component 내부에서만 호출할 수 있습니다.
- Hook은 Component의 최상위 레벨에서만 호출할 수 있습니다.
- Hook은 조건부일 수 없습니다

- useState (상태 관리): 컴포넌트 내부에서 상태 값을 관리할 때 사용합니다.

```
const [state, setState] = useState(초기값);
```

- useState Hook은 문자열, 숫자, 부울, 배열, 객체 및 이들의 조합을 추적하는 데 사용할 수 있습니다
- 개별 값을 추적하기 위해 여러 개의 상태 Hook을 만들 수 있습니다.


Builds/Modifies
Builds/Modifies
App
React Virtual DOM
DOM
Delivers events
Delivers events


4. Event handler manages events
Document
Event handler
3. Event bubbles up to Document (root); captured by React event listener
ul
Parent
li
li
li
li
2. Event bubbles up to Parent
1. DOM event (e.g., account added)

React 이벤트

- React는 HTML과 동일한 이벤트를 가집니다
- React의 이벤트는 camelCase 를 사용합니다.
- JSX를 사용하여 문자열이 아닌 함수로 이벤트 핸들러를 {} 사용하여 전달합니다.

```
//DOM 엘리먼트에서 이벤트를 처리하는 방식
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

```
//react이벤트를 처리하는 방식
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

React 이벤트

- 이벤트 핸들러에 인수를 전달하려면 화살표 함수 또는 **bind**를 사용 사용합니다

```
function Football() {
  const shoot = (a) => {
    alert(a);
  }

  return (
    <button onClick={() => shoot("Goal!")}>Take the shot!</button>
  );
}
```

```
function Football() {
  const shoot = (a) => {
    alert(a);
  }

  return (
    <button onClick={shoot.bind("Goal!")}>Take the shot!</button>
  );
}
```

React 이벤트

- React의 이벤트 핸들러는 React 이벤트 객체에 접근할 수 있습니다.
- React에서 제공하는 이벤트 객체는 SyntheticEvent(합성 이벤트)입니다.

```
function Football() {
  const shoot = (a, b) => {
    alert(b.type);
  }

  return (
    <button onClick={(event) => shoot("Goal!", event)}>Take the shot!</button>
  );
}
```

SyntheticEvent :
React는 브라우저의 네이티브 이벤트 대신 SyntheticEvent를 사용합니다.
SyntheticEvent는 네이티브 이벤트를 감싼 객체로, 브라우저 간의 차이를 해결하기 위해 사용됩니다.
네이티브 이벤트와 비슷하지만, React 내부에서 최적화되어 동작합니다.
event.nativeEvent를 사용하면 원래의 네이티브 이벤트 객체에 접근할 수도 있습니다.

React 이벤트

- React에서는 false를 반환해도 기본 동작을 방지할 수 없으며, preventDefault를 명시적으로 호출해야 합니다.

```
//DOM 엘리먼트에서  기본 이벤트 취소
<form onsubmit="console.log('You clicked submit.'); return false">
  <button type="submit">Submit</button>
</form>
```

```
//react이벤트를 기본 이벤트 취소
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('You clicked submit.');
  }

  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">Submit</button>
    </form>
  );
```

React 이벤트

- JavaScript에서 클래스 메서드는 기본적으로 바인딩되어 있지 않습니다.
- this.이벤트핸들러를 바인딩하지 않고 on이벤트속성에 전달하면, 함수가 실제 호출될 때 this는 undefined가 됩니다.

```
class LoggingButton extends React.Component {
  handleClick = () => {
    console.log('this is:', this);
  };
  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }

  render() {
    return (
      <button onClick={() => this.handleClick()}>
        Click me
      </button>
    );
  }
```

React 조건부 렌더링

- React에서 if를 사용하여 Component를 조건부로 렌더링할 수 있습니다.

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}
```

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Greeting isLoggedIn={false} />);
```

React Form

- React에서 form 데이터는 component 의 state로 저장됩니다

```
import { useState } from 'react';
import ReactDOM from 'react-dom/client';

function MyForm() {
  const [name, setName] = useState("");

  return (
    <form>
      <label>Enter your name:
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </label>
    </form>
  )
}
```

React Form

- onSubmit 속성에 이벤트 핸들러를 추가하면 submit 작업을 제어할 수 있습니다.

```
import { useState } from 'react';
import ReactDOM from 'react-dom/client';
function MyForm() {
 const [name, setName] = useState("");
 const handleSubmit = (event) => {
  event.preventDefault();
  alert(`The name you entered was: ${name}`)
 }
 return (
  <form onSubmit={handleSubmit}>
   <label>Enter your name:
    <input
     type="text"
     value={name}
     onChange={(e) => setName(e.target.value)}
    />
   </label>
   <input type="submit" />
  </form>
 )
}
```

React Form

- 이벤트 핸들러의 필드에 액세스하려면 event.target.name 과 event.target.value 구문을 사용합니다.
- 상태를 업데이트하려면 속성 이름 주위에 대괄호 []을 사용합니다.

```
import { useState } from 'react';
import ReactDOM from 'react-dom/client';

function MyForm() {
 const [inputs, setInputs] = useState({});

 const handleChange = (event) => {
  const name = event.target.name;
  const value = event.target.value;
  setInputs(values => ({...values, [name]: value}))
 }

 const handleSubmit = (event) => {
  event.preventDefault();
  alert(inputs);
 }
```

React Form


```
return (
  <form onSubmit={handleSubmit}>
   <label>Enter your name:
   <input
    type="text"
    name="username"
    value={inputs.username || ""}
    onChange={handleChange}
   />
   </label>
   <label>Enter your age:
    <input
     type="number"
     name="age"
     value={inputs.age || ""}
     onChange={handleChange}
    />
    </label>
    <input type="submit" />
  </form>
 )
}
```

React textarea

- React에서 textarea의 값은 value 속성에 배치됩니다.
- textarea의 값을 관리하기 위해 usageState Hook을 사용합니다

```
import { useState } from 'react';
import ReactDOM from 'react-dom/client';

function MyForm() {
  const [textarea, setTextarea] = useState(
    " textarea의 값은 value 속성에 포함됩니다 "
  );

  const handleChange = (event) => {
    setTextarea(event.target.value)
  }

  return (
    <form>
      <textarea value={textarea} onChange={handleChange} />
    </form>
  )
}
```

React textarea

- React에서 drop down list 의 선택된 값은 selected 속성에 배치됩니다.
- React에서 select box의 선택된 값은 value 속성에 배치됩니다.

```
function MyForm() {
  const [myCar, setMyCar] = useState("Volvo");

  const handleChange = (event) => {
    setMyCar(event.target.value)
  }

  return (
    <form>
      <select value={myCar} onChange={handleChange}>
        <option value="Ford">Ford</option>
        <option value="Volvo">Volvo</option>
        <option value="Fiat">Fiat</option>
      </select>
    </form>
  )
}
```

props와 state를 사용한 Todo 애플리케이션

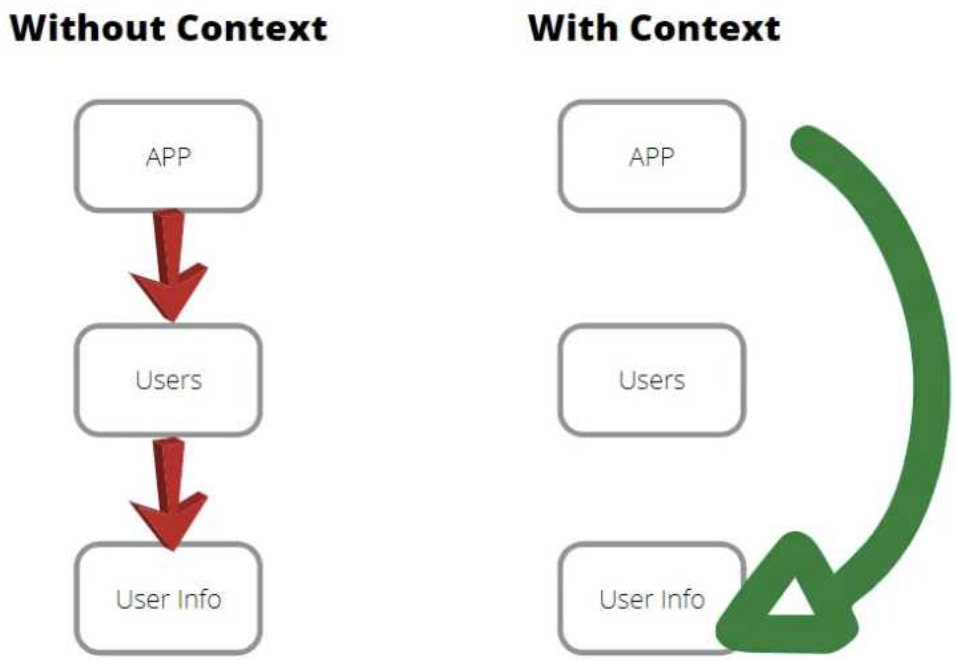

```
class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = { items: [], text: '' };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
 render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={this.handleSubmit}>
          <label htmlFor="new-todo">
            What needs to be done?
          </label>
          <input
            id="new-todo"
            onChange={this.handleChange}
            value={this.state.text}
          />
```

props와 state를 사용한 Todo 애플리케이션


```
<button>
            Add #{this.state.items.length + 1}
          </button>
        </form>
      </div>
    );
  }

handleChange(e) {
    this.setState({ text: e.target.value });
  }
handleSubmit(e) {
    e.preventDefault();
    if (this.state.text.length === 0) {
      return;
    }
    const newItem = {
      text: this.state.text,
      id: Date.now()
    };
```

props와 state를 사용한 Todo 애플리케이션


```
 this.setState(state => ({
       items: state.items.concat(newItem),
       text: ''
     }));
  }
}

class TodoList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.items.map(item => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    );
  }
}

root.render(<TodoApp />);
```


Without Context
APP
Users
User Info
With Context
APP
Users
User Info

React Context API

- React version 16부터 사용 가능한 React의 내장 API
- props를 사용하지 않고 필요한 데이터(state)를 쉽게 공유
- props drilling을 해결하기 위해 사용
- 깊이 여부와 무관하게 데이터가 필요한 컴포넌트에서만 불러다가 사용할 수 있다
- 앱의 모든 컴포넌트에서 사용할 수 있는 데이터(state)를 전달할 때 유용
- Context API는 자주 업데이트할 필요가 없는 데이터에 사용한다.
- Context API에서 State값을 변경하면, Provider로 감싼 모든 자식 컴포넌트들이 리렌더링되므로 전역 상태 관리를 위한 도구가 아닌, 데이터를 쉽게 전달하고 공유하기 위한 목적으로 사용하는 것이 적합하다.

중간에 여러 컴포넌트를 거쳐야 하거나 앱의 여러 컴포넌트에서 동일한 데이터를 필요로 하는 경우에는 계속 props로 넘겨줘야 하는 prop drilling을 해야만 한다
Props drilling: 중첩된 여러 계층의 컴포넌트들에게 props를 전달해 주는 것

React Context API

- Context API를 통해 전달하는 데이터의 종류
  - 테마 데이터 (다크 모드, 라이트 모드)
  - 사용자 데이터 (현재 인증된 사용자)
  - 언어 혹은 지역 데이터

- Context API 사용 방법
  - createContext 메서드를 사용하여 context 생성한다.
  - 생성한 context를 대상 컴포넌트에 값을 내려주기 위해서 Provider로 대상 컴포넌트를 감싼다.
  - Provider의 프로퍼티인 value에 전달할 데이터를 넣는다.
  - Provider의 value에 담은 데이터를 전달 할 때는Consumer 컴포넌트 또는 useContext라는 훅을 이용하여 전달한다.

React Context API

```
//Context 생성
import { createContext } from "react";

const MyContext = createContext();
```

```
// Provider로 데이터 제공
import { useState } from "react";

export const MyProvider = ({ children }) => {
  const [user, setUser] = useState("John Doe");

  return (
    <MyContext.Provider value={{ user, setUser }}>
      {children}
    </MyContext.Provider>
  );
};
```

React Context API

```
// Consumer 또는 useContext로 데이터 사용
import { useContext } from "react";

const MyComponent = () => {
  const { user, setUser } = useContext(MyContext);

  return (
    <div>
      <p>User: {user}</p>
      <button onClick={() => setUser("Jane Doe")}>Change User</button>
    </div>
  );
};
```

```
import { MyProvider } from "./MyProvider";
import MyComponent from "./MyComponent";

const App = () => (
  <MyProvider>
    <MyComponent />
  </MyProvider>
);

export default App;
```

하이오더(Higher-Order Component, HOC)

- 다른 컴포넌트를 인자로 받아 새로운 컴포넌트를 반환하는 함수
- 여러 컴포넌트에서 공통된 기능을 분리하여 컴포넌트 로직을 재사용하는 패턴을 제공
- 기존의 컴포넌트를 수정하거나 확장하는 역할을 함
- 직접적으로 렌더링되는 것이 아니라, 기존 컴포넌트를 인자로 받아 새로운 컴포넌트를 반환합니다.
- 원래의 컴포넌트를 수정하지 않고, 래핑(wrap)하여 새로운 기능을 추가합니다.
- 원본 컴포넌트에 필요한 props를 그대로 전달합니다

```
const MyHOC = (WrappedComponent) => {
  return class extends React.Component {
    render() {
      // HOC가 제공하는 기능을 추가
      return <WrappedComponent {...this.props} />;
    }
  };
};
```

하이오더(Higher-Order Component, HOC)

- HOC는 로딩 상태를 관리하고, 로딩 중일 때 로딩 메시지를 보여주는 기능을 추가합니다.

```
import React from 'react';
// HOC 정의
const withLoading = (WrappedComponent) => {
  return class extends React.Component {
    state = {
      loading: true,
    };
    componentDidMount() {
      setTimeout(() => {
        this.setState({ loading: false });
      }, 2000); // 2초 후에 로딩 상태 종료
    }
    render() {
      const { loading } = this.state;
      if (loading) {
        return <div>Loading...</div>;
      }
      return <WrappedComponent {...this.props} />;
    }
  };
};
```

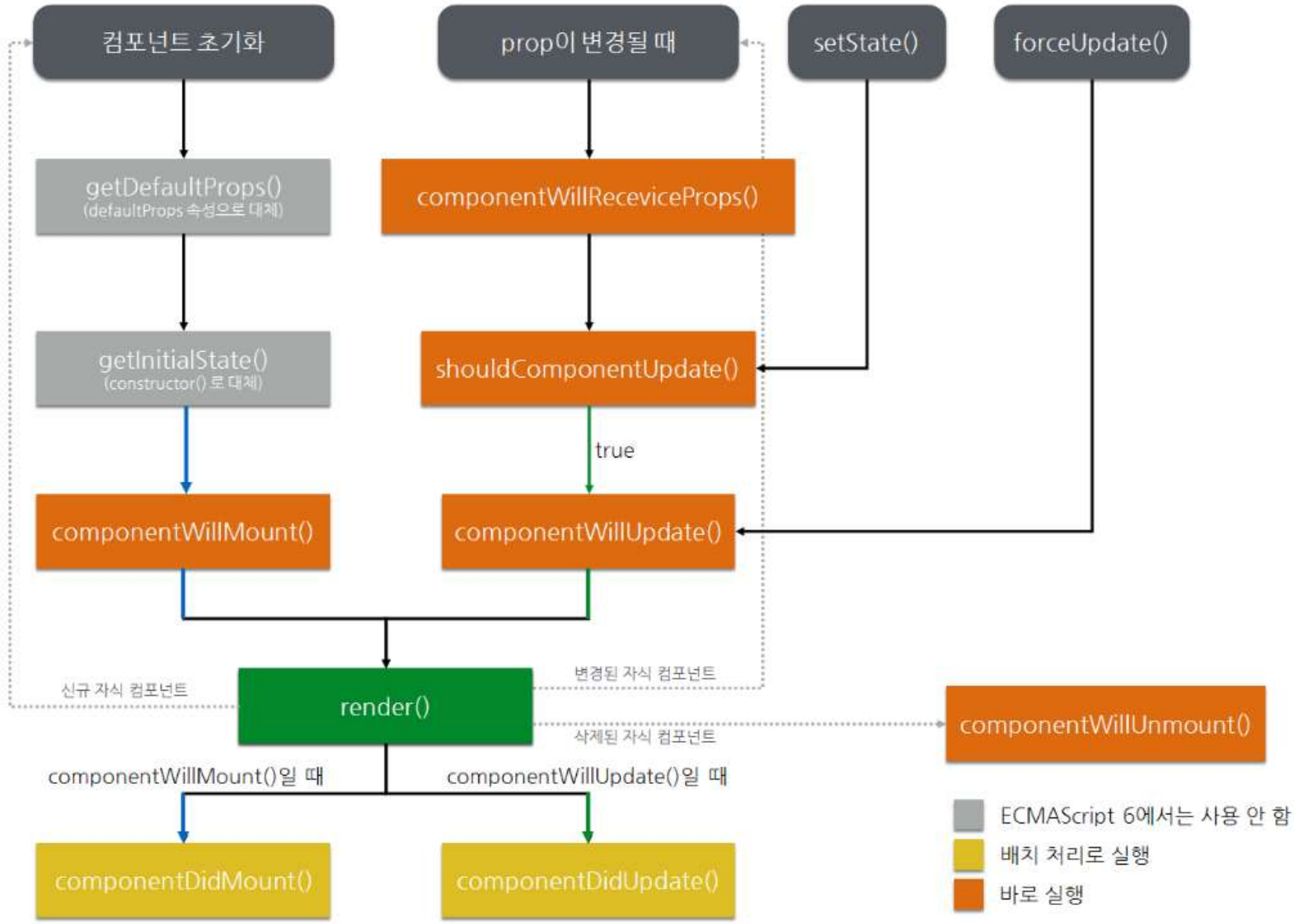
하이오더(Higher-Order Component, HOC)

- HOC는 로딩 상태를 관리하고, 로딩 중일 때 로딩 메시지를 보여주는 기능을 추가합니다.

```
// HOC를 사용하여 컴포넌트 확장
const MyComponent = () => {
  return <div>My component content</div>;
};

const MyComponentWithLoading = withLoading(MyComponent);

export default MyComponentWithLoading;
```

- HOC에서 전달하는 props와 원본 컴포넌트에서 사용하는 props 이름이 겹칠 수 있기 때문에, props 이름을 관리하는 것이 중요합니다.
- HOC는 클래스 컴포넌트에서 사용될 때, 래핑된 컴포넌트의 정적 메서드가 사라질 수 있습니다. (hoist-non-react-statics와 같은 라이브러리를 사용하여 해결할 수 있습니다)

React Component Lifecycle



React Component Lifecycle

Lifecycle 메서드	설명
constructor()	컴포넌트가 처음 생성될 때 한번만 호출됨 생성자 내에서 항상 super()함수를 가장 먼저에 호출해야 합니다. 상태를 초기화하거나 인스턴스를 설정하는 데 사용됩니다.
getDerivedStateFromProps()	props가 변경될 때마다 호출되며, 새 props에 대해 상태를 업데이트할 수 있습니다. static 메서드로, 클래스 메서드 안에서 this를 사용할 수 없습니다.
render()	UI를 반환하는 메서드로 JSX를 반환합니다. 데이터가 변경되어 새 화면을 그려야 할 때 자동으로 호출됨
componentDidMount()	컴포넌트가 DOM에 렌더링된 후에 DOM에 삽입된 직후 호출됨 API 호출, 타이머 설정, 외부 라이브러리와의 통합 등 DOM과 관련된 작업 및 비동기 작업을 수행 할 수 있습니다.
getSnapshotBeforeUpdate()	컴포넌트의 DOM이 업데이트되기 직전에 호출됩니다. 렌더링 결과를 DOM에 반영하기 전에 필요한 값을 캡처할 때 유용합니다
componentDidUpdate(prevProps, prevState, snapshot)	컴포넌트가 업데이트된 후에 호출 이전 props와 state를 비교하여 추가 작업을 할 수 있습니다.
componentWillUnmount	컴포넌트가 제거되기 직전에 호출 주로 타이머를 정리하거나 이벤트 리스너를 제거하는 데 사용됩니다.

React Component Lifecycle

1. constructor() : 컴포넌트가 처음 생성될 때 한번만 호출됨
   - 생성자 내에서 항상 super()함수를 가장 먼저에 호출해야 합니다.
   - 상태를 초기화하거나 인스턴스를 설정하는 데 사용됩니다.
2. componentWillMount() : 컴포넌트가 렌더링되기 직전에 호출
   - 상태를 설정하거나 초기화 작업을 할 때 유용
   - render() 메서드가 호출되기 직전에 호출
   - 컴포넌트가 렌더링되기 전에 변경을 일으킬 수 있기 때문에, React 16.3부터 deprecated(사용되지 않음) 되어 UNSAFE_componentWillMount()로 대체
3. componentDidMount() : 컴포넌트가 DOM에 렌더링된 후에 DOM에 삽입된 직후 호출됨
   - API 호출, 타이머 설정, 외부 라이브러리와의 통합 등 DOM과 관련된 작업을 수행

-

React Component Lifecycle


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

class LifecycleExample extends Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
    console.log('Constructor: 초기화');
  }

  static getDerivedStateFromProps(nextProps, nextState) {
    console.log('getDerivedStateFromProps: props로부터 상태 업데이트');
    return null;  // 상태를 수정하지 않으면 null을 반환
  }

  shouldComponentUpdate(nextProps, nextState) {
    console.log('shouldComponentUpdate: 렌더링 여부를 결정');
    return true;  // 항상 렌더링하도록 true를 반환
  }
```

React Component Lifecycle

```
render() {
  console.log('Render: UI 반환');
  return (
    <div>
      <h1>카운트: {this.state.count}</h1>
      <button onClick={() => this.setState({ count: this.state.count + 1 })}>증가</button>
    </div>
  );
}

getSnapshotBeforeUpdate(prevProps, prevState) {
  console.log('getSnapshotBeforeUpdate: 업데이트 직전');
  return null;  // 반환값은 componentDidUpdate에서 사용됨
}

componentDidMount() {
  console.log('componentDidMount: 마운트 후');
}

componentDidUpdate(prevProps, prevState, snapshot) {
  console.log('componentDidUpdate: 업데이트 후');
}
```

React Component Lifecycle


```
  componentWillUnmount() {
    console.log('componentWillUnmount: 제거 직전');
  }
}

export default LifecycleExample;
```

컴포넌트가 처음 마운트

- constructor
- getDerivedStateFromProps
- render
- componentDidMount

버튼 클릭으로 상태가 변경되면

- shouldComponentUpdate
- render
- getSnapshotBeforeUpdate
- componentWillUnmount

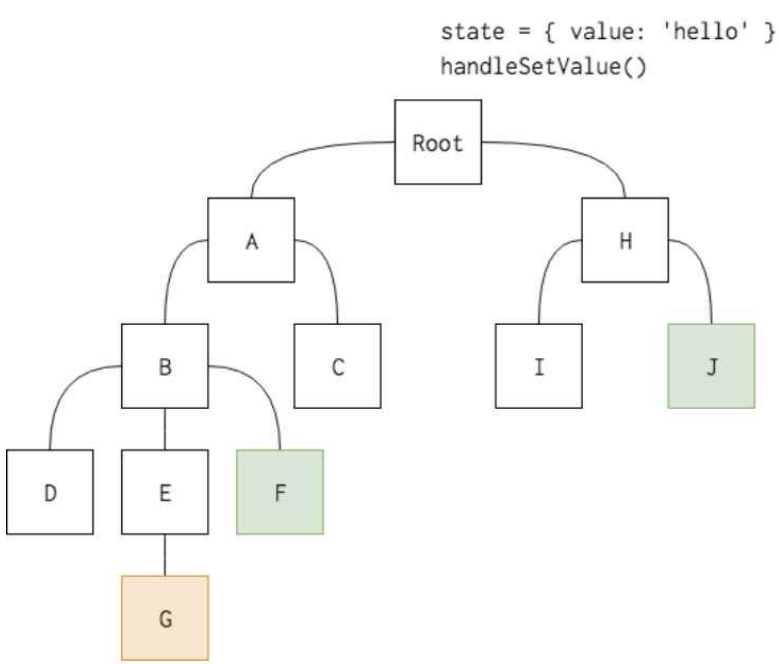
React Component

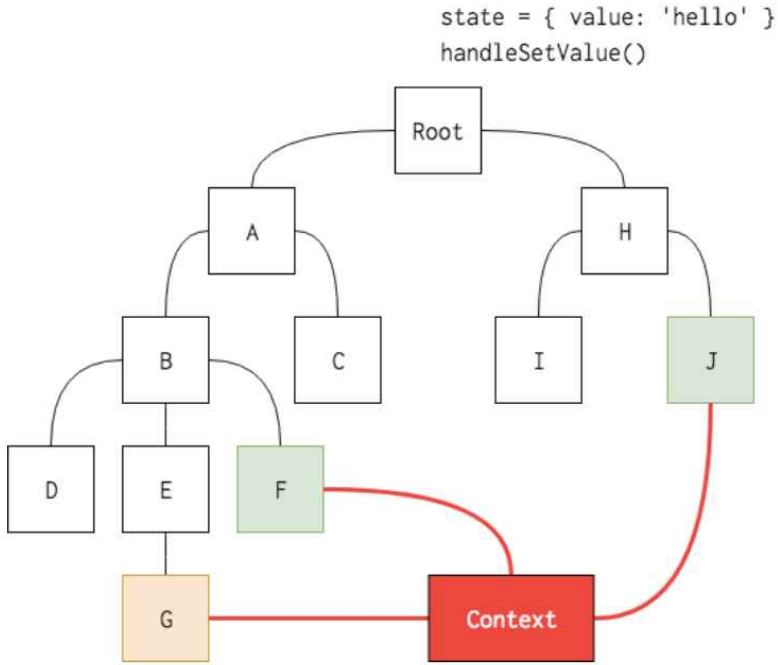
- 클래스형 컴포넌트

```
import React, { Component } from "react";
class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }
  componentDidMount() {
    console.log("컴포넌트가 마운트됨");
  }
  componentDidUpdate() {
    console.log("컴포넌트가 업데이트됨");
  }
  componentWillUnmount() {
    console.log("컴포넌트가 언마운트됨");
  }
  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>        +        </button>
      </div>
    );
  }
}
export default Counter;
```

React Context API

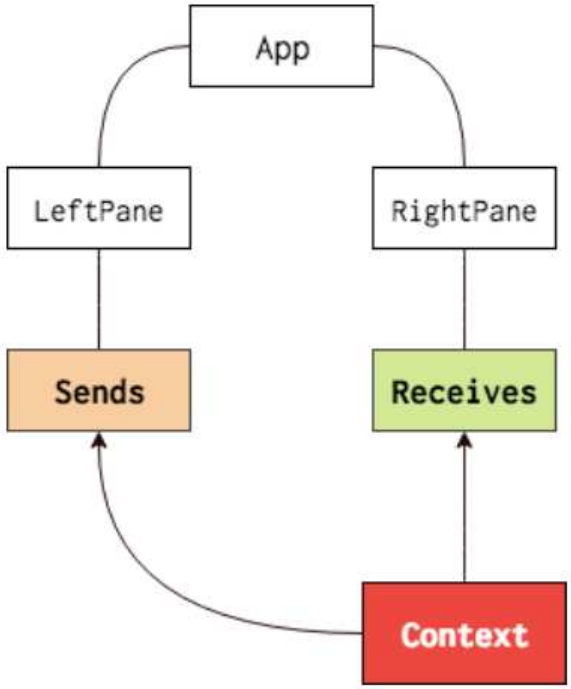
- context는 React 컴포넌트 트리 안에서 전역적(global)이라고 볼 수 있는 데이터를 공유할 수 있도록 고안된 방법
- context를 이용하면 단계마다 일일이 props를 넘겨주지 않고도 컴포넌트 트리 전체에 데이터를 제공할 수 있습니다.
- 사용자 로그인 정보, 애플리케이션 설정, 테마, 등 전역적으로 데이터를 관리하는 용도로 사용
- 여러 컴포넌트를 거치지 않고 Context 를 통해서 원하는 값이나 함수를 바로 전달할 수 있습니다





React Context API

- App 컴포넌트 내부에 LeftPane 과 RightPane 라는 컴포넌트를 만들고, 한쪽에는 값을 설정시킬 Sends 컴포넌트, 반대쪽에는 Receives 컴포넌트를 만들어서 Context 를 사용하여 값 전달



React Context API

- Context 객체 생성

```
const MyContext = React.createContext(defaultValue);
```

- Context.Provider 를 이용하여 Context 변경 사항을 자손들에게 제공
- Provider 의 value는 하위의 모든 Consumer 에서 사용할 수 있으며, Provider 하위의 모든 Consumer 는 Provider 의 value가 변경 될 때마다 다시 렌더링 됩니다.

```
<MyContext.Provider value={/* some value */}></MyContext.Provider>
```

- Context.Consumer는 Provider의 Value의 변경 사항을 구독하며, Context 에서 가장 가까운 Provider 의 Value 를 참조한다.

```
<MyContext.Consumer>
{(value) => (/* render something based on the context value */)}
</MyContext.Consumer>
```

- Class.contextType - Class 의 contextType 에 Context 객체를 할당 할 수 있다.

React Context API

- Context 는 createContext 라는 함수를 사용해서 만들며, 이 함수를 호출하면 Provider 와 Consumer 라는 컴포넌트들이 반환됩니다.
- Provider 는 Context 에서 사용 할 값을 설정할 때 사용되고, Consumer 는 Context에 설정한 값을 불러와야 할 때 사용됩니다.

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

const Context = createContext(); // Context 를 만듭니다.

const { Provider, Consumer: SampleConsumer } = Context;

class SampleProvider extends Component {
 state = {
  value: '기본값입니다'
 }
 actions = {
  setValue: (value) => {
   this.setState({value});
  }
 }
```

React Context API


```
render() {
    const { state, actions } = this;
    const value = { state, actions };
    return (
      <Provider value={value}>
        {this.props.children}
      </Provider>
    )
  }
}

export {
  SampleProvider,
  SampleConsumer,
};
```

React Context API

- Provider 사용하기

```
# src/App.js
import React from 'react';
import LeftPane from './components/LeftPane';
import RightPane from './components/RightPane';
import { SampleProvider } from './contexts/sample';

const App = () => {
  return (
    <SampleProvider>
      <div className="panes">
        <LeftPane />
        <RightPane />
      </div>
    </SampleProvider>
  );
};

export default App;
```

React Context API

- Consumer 사용하기

```
# src/components/Receives.js
import React from 'react';
import { SampleConsumer } from '../contexts/sample';

const Receives = () => {
  return (
    <SampleConsumer>
      {
        (sample) => (
          <div>
            현재 설정된 값: { sample.state.value }
          </div>
        )
      }
    </SampleConsumer>
  );
};

export default Receives;
```

React Context API

```
# src/contexts/sample.js
import React, { Component } from 'react';
import { SampleConsumer } from '../contexts/sample';

class Sends extends Component {

 state = {
   input: ''
 }

 componentDidMount() {
  this.setState({                                  // 초기 값 설정
     input: this.props.value,
   })
 }

 handleChange = (e) => {
   this.setState({ input: e.target.value });
 }
 handleSubmit = (e) => {
     e.preventDefault();
    this.props.setValue(this.state.input);                    // props로 받은 setValue 호출
 }
```

- Container 를 만들 때 사용했던 로직을, 쉽게 재사용 할 수 있도록 HoC 를 사용

```
# src/contexts/sample.js
import React, { Component, createContext } from 'react';
……
// :: HoC 를 사용
function useSample(WrappedComponent) {
  return function UseSample(props) {
    return (
      <SampleConsumer>
        {
          ({ state, actions }) => (
            <WrappedComponent
              value={state.value}
              setValue={actions.setValue}
            />
          )
        }
      </SampleConsumer>
    )
  }
}
export {
   ……
  useSample
```

- Context 를 여러개 만들 수도 있습니다.

```
import React, { Component, createContext } from 'react';
const Context = createContext();
const {   Provider,  Consumer: AnotherConsumer} = Context;

class AnotherProvider extends Component {
  state = {
    number: 1,
  }
  actions = {
    increment: () => {
      this.setState(
        ({ number }) => ({ number: number + 1 })
      );
    }
  }
  render() {
    const { state, actions } = this;
    const value = { state, actions };
    return (
      <Provider value={value}>
        {this.props.children}
      </Provider>
    );
  }
}
```

- Context 를 여러 개 만들 수도 있습니다.

```
const AppProvider = ({ contexts, children }) => contexts.reduce(
  (prev, context) => React.createElement(context, {
    children: prev
  }),
  children
);

const App = () => {
  return (
    <AppProvider
      contexts={[SampleProvider, AnotherProvider]}
    >
      <div className="panes">
        <LeftPane />
        <RightPane />
      </div>
    </AppProvider>
  );
};

export default App;
```

- Context 를 여러 개 만들 수도 있습니다.

```
import React from 'react';
import { useAnother } from '../contexts/another';

const Counter = ({ number, increment}) => {
  return (
    <div>
      <h1>{number}</h1>
      <button onClick={increment}>더하기</button>
    </div>
  );
};

export default useAnother(Counter);
```

- props로 전달하는 테마를 Context로 전달

```
class App extends React.Component {
  render() {
    return <Toolbar theme="dark" />;
  }
}
// Toolbar 컴포넌트는 불필요한 테마 prop를 받아서  ThemeButton에 전달해야 합니다.
function Toolbar(props) {
return (
    <div>
      <ThemedButton theme={props.theme} />
    </div>
  );
}

class ThemedButton extends React.Component {
  render() {
    return <Button  theme={this.props.theme } />;
  }
}
```

- props로 전달하는 테마를 Context로 전달

```
const ThemeContext = React.createContext('light');
class App extends React.Component {
  render() {                    // Provider를 이용해 하위 트리에 테마 값을 보내줍니다.
    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}
function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}
class ThemedButton extends React.Component {          // 현재 선택된 테마 값을 읽기 위해
contextType을 지정
  // React는 가장 가까이 있는 테마 Provider를 찾아 그 값을 사용
  static contextType = ThemeContext;
  render() {
    return <Button theme={this.context} />;
  }
}
```

- context의 주된 용도는 다양한 레벨에 네스팅된 많은 컴포넌트에게 데이터를 전달하는 것입니다.
- context를 사용하면 컴포넌트를 재사용하기가 어려워지므로 꼭 필요할 때만 사용해야 합니다.
- 여러 레벨에 걸쳐 props 넘기는 걸 대체하는 데에 context보다 컴포넌트 합성이 간단한 해결책일 수도 있습니다.

#Link 와 Avatar 컴포넌트에게 user 와 avatarSize 라는 props를 전달해야 하는 Page 컴포넌트

```
<Page  user={user} avatarSize={avatarSize} />
// ... 그 아래에 ...
<PageLayout user={user}  avatarSize={avatarSize} />
// ... 그 아래에 ...
<NavigationBar user={user} avatarSize={avatarSize} />
// ... 그 아래에 ...
<Link href={user.permalink}>
  <Avatar user={user} size={avatarSize} />
</Link>
```

```
function Page(props) {
  const user = props.user;
  const userLink = (
    <Link  href = {user.permalink} >
      <Avatar  user={user}  size={props.avatarSize} />
    </Link>
  );
  return <PageLayout  userLink={userLink} />;
}
// 이제 이렇게 쓸 수 있습니다.
<Page  user={user}  avatarSize={avatarSize} />
// ... 그 아래에 ...
<PageLayout  userLink={...}  />
// ... 그 아래에 ...
<NavigationBar  userLink={...} />
// ... 그 아래에 ...
{props.userLink}
```

Appendix

Sass(Syntactically Awesome Stylesheets)

- CSS를 보다 효율적으로 작성할 수 있도록 도와주는 CSS 전처리기(preprocessor)
- 반복되는 스타일을 줄이고 유지보수가 쉬움
- 규모가 큰 프로젝트에서 CSS 코드를 모듈화하고 체계적으로 관리할 수 있음
- 변수, 중첩, 믹스인 등의 기능을 활용하여 생산성을 높일 수 있음

비교항목	CSS	Sass
변수사용	불가능	$primary-color:blue;
중첩(Nesting)	불가능	.parent {.child {…}}
Mixin(재사용 함수)	불가능	@mixin button { … }
상속(inheritance)	불가능	@extend .base-class
연산 기능	불가능	Width: (100% / 3);
코드 모듈화	불가능	@import, @use
파일 분리	불가능	@import, @use 활용 가능

Sass(Syntactically Awesome Stylesheets)

- CSS에서는 색상이나 폰트 크기를 일일이 반복해서 입력해야 하지만, Sass에서는 변수를 사용해 코드의 재사용성을 높일 수 있습니다.

```
$primary-color: #3498db;
body {
  background-color: $primary-color;
}
```

- Sass에서는 요소의 계층 구조를 직관적으로 표현할 수 있습니다.

```
.nav {
  ul {
    list-style: none;
    li {
      display: inline-block;
    }
  }
}
```

Sass(Syntactically Awesome Stylesheets)

- 자주 사용하는 스타일을 함수처럼 정의하고 여러 곳에서 재사용할 수 있습니다.

```
@mixin flex-center {
  display: flex;
  justify-content: center;
  align-items: center;
}

.container {
  @include flex-center;
}
```

- CSS에서는 불가능한 수학 연산이 가능하여 가변적인 스타일 적용이 가능합니다.

```
@import "header";  // header.scss 파일을 가져옴
@import "footer";
```

Sass(Syntactically Awesome Stylesheets)

- 비슷한 스타일을 공유하는 클래스 간에 @extend를 사용하여 코드 중복을 줄일 수 있습니다.

```
.btn {
  padding: 10px;
  border-radius: 5px;
}

.btn-primary {
  @extend .btn;
  background-color: blue;
}
```

- @import 또는 @use를 활용하여 여러 개의 Sass 파일을 하나의 파일처럼 사용할 수 있습니다

React 개발 VS Code 유용한 extension

- JavaScript (ES6) Code Snippets : 자바스크립트 코드 스니펫을 제공하는 확장 프로그램. 반복되는 코드를 간편하게 작성할 수 있으며, 사용자 정의 스니펫을 지원합니다.
- ES7 React / Redux / GraphQL / React-Native snippets : 키워드를 사용하여 빠르게 코드를 작성할 수 있는 확장
- ESLint : 코드를 자동으로 형식화하고 오류를 경고메세지로 출력합니다.
- Bracket Pair Colorzier : 코드에서 대괄호를 색상으로 구분하여 쉽게 찾을 수 있게 도와주는 확장
- Live Server : 로컬 호스트 서버를 시작하고 코드 변경 시 자동으로 새로고침 되는 기능을 제공
- Browser Preview : VS Code내에 브라우저 미리보기를 할 수 있는 확장
- Path Intellisense : 파일 경로를 자동 완성해주는 확장
- Reactjs code snippets : React 개발을 위한 코드 스니펫을 제공하는 확장
- Prettier - Code formatter : 코드 스타일을 자동으로 관리해주는 확장
- vscode-styled-components : styled-components를 사용할 때 자동완성을 도와주는 확장

reactstrap

- React를 위한 UI 컴포넌트 라이브러리
- Bootstrap을 기반으로 하여, Bootstrap의 스타일과 컴포넌트를 React 애플리케이션에서 쉽게 사용할 수 있도록 만들어진 라이브러리
- React 컴포넌트로 Bootstrap의 구성 요소들을 제공합니다.
- Bootstrap의 전통적인 CSS를 그대로 사용할 수 있기 때문에, Bootstrap을 알고 있거나 사용하는 개발자에게 친숙합니다.
- Bootstrap을 이미 사용 중인 프로젝트에서 쉽게 통합할 수 있습니다.
- 버튼, 카드, 알림, 모달, 내비게이션, 폼 등 다양한 UI 컴포넌트를 제공합니다.
- React의 컴포넌트 기반 설계 방식을 따르기 때문에, React에서 쉽게 사용하고 관리할 수 있습니다.

```
npm install reactstrap bootstrap
```

Reactstrap


```
import React from 'react';
import { Button } from 'reactstrap';

const Example = () => {
  return (
    <Button color="primary">Primary Button</Button>
  );
};

export default Example;
```