React

2019.4

박재성

이 문서는 나눔글꼴로 작성되었습니다. 설치하기

목차

1. React Basic

React?

React

개요

- Facebook이 2013년 5월 발표
- 컴포넌트 단위의 UI를 개발할 수 있으며, MVC 구조의 V(Views) 영역을 구현
- React는 선언적, 효율적 그리고 유연한 UI 개발을 위한 JavaScript 라이브러리/프레임워크
- 작고 isolated된 '컴포넌트'라고 불리는 작은 조각의 코드들을 통해 복잡한 UI를 개발할 수 있도록 한다.
- View만을 담당하기 때문에, 라우팅과 데이터 흐름 등을 담당하는 추가적인 도구를 같이 사용해야 한다.
- React 앱은 React Native 도구를 통해 네이티브 모바일 앱으로 개발될 수 있어, 단일 코드베이스를 통해 웹, 모바일 앱 등의 개발이 가능하다.

환경설정 #1

프로젝트 생성

package.json 파일을 생성하기 위해 다음의 명령을 실행한다.

```
$ npm init
```

React 패키지들을 설치한다.

```
$ npm install --save react react-dom
```

환경설정 #2

Babel과 Webpack 패키지 설치

Babel과 Webpack 패키지들을 각각 설치한다.

→ Babel에서 React를 사용하기 위해선 '**@babel/preset-react**' 패키지가 필요하다.

→ Webpack의 **html-webpack-plugin**은 번들링된 파일을 로딩하는 코드가 추가된 html 파일을 생성하는 플러그인이다.

```
# Babel 패키지
npm install --save-dev @babel/cli @babel/core @babel/preset-env @babel/preset-react
babel-loader

# Webpack 패키지
npm install --save-dev webpack webpack-cli webpack-dev-server html-webpack-plugin
```

환경설정: Babel

babel.config.js

```
module.exports = {
    presets: [
        [
            "@babel/preset-env", {
                targets: {
                    browsers: [ "last 2 versions",  "ie >= 9",  "iOS >= 8" ]
                },
                    useBuiltIns: "usage",
            }
        ],
        [
            // https://babeljs.io/docs/en/babel-preset-react#options
            "@babel/preset-react", {
                "throwIfNamespace": false // defaults to true
            }
        ]
    ]
};
```

환경설정: Webpack

webpack.config.js

```
module: {
  rules: [{
      test: /₩.m?js$/,
      exclude: /(node_modules)/,
      use: {
        loader: "babel-loader",
        options: {
          presets: [
            "@babel/preset-env", "@babel/preset-react"
          ]
        }
      }
    }]
},
plugins: [
    new HtmlWebPackPlugin({
        title: "My ReactApp",
        template: "./src/index.html",
        filename: "../index.html"
    })
]
```

```
const path = require("path");
const HtmlWebPackPlugin = require("html-webpack-plugin");

module.exports = function(env, arg) {
    const distPath = path.join(__dirname, "dist");

    return {
        entry: "./src/index.js",
        output: {
            path: distPath,
            filename: "webapp.bundle.js",
            publicPath: "dist"
        },
        module: {
          rules: [
            {
              test: /₩.m?js$/,
              exclude: /(node_modules)/,
              use: {
                loader: "babel-loader",
                options: {
                  presets: ["@babel/preset-env", "@babel/preset-react"]
                }
              }
            }
          ]
        },
        devtool: "cheap-module-source-map",
        mode: arg.mode || "none",
        plugins: [
            new HtmlWebPackPlugin({
                title: "My ReactApp",
                template: "./src/index.html",
                filename: "../index.html"
            })
        ]
    };
};
```

환경설정: npm script

package.json

기본 빌드 명령어와 개발서버 실행을 위한 script command를 설정한다.

```
{
   …
       "scripts": {
         "build": "webpack",
         "build:prod": "webpack --mode production",
         "start": "webpack-dev-server --open"
       },
    …
}
```

첫 번째 React App #1

- /src 폴더 내에 아래 코드를 "index.html" 이름으로 저장한다.
- "npm run build"를 실행해, index.html 파일을 생성한다.

→ 아래 템플릿은 "html-webpack-plugin"의 템플릿으로 사용된다.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <!----- App holder ----->
    <div id="app"></div>
  </body>
</html>
```

첫 번째 React App #2

- /src 폴더 내에 아래 코드를 "index.js" 이름으로 저장한다.
- CLI에서 "npm start"를 실행해 확인한다.

```
import React, { Component } from "react";
import ReactDOM from "react-dom";

class SayHello extends React.Component {
    render() {
        return (
            <div>Hello World!</div>
        );
    }
}

ReactDOM.render(<SayHello />, document.getElementById("app"));
```

브라우저에서 빠르게 실행하는 방법

React, ReactDOM과 Babel standalone 버전을 CDN에서 로딩

```
<head>
  <script crossorigin src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
  <script src="https://unpkg.com/babel-standalone/babel.min.js"></script>
</head>
```

→ https://reactjs.org/docs/cdn-links.html

<script type="text/babel"> 블록에서 코드 작성

```
<!-- container -->
<div id="root"></div>

<script type="text/babel">
    ReactDOM.render(<h1>Hello World!</h1>,
        document.getElementById("root")
    );
</script>
```

2. JSX (JavaScript XML)

JSX?

특징

- JSX는 태그와 같은 형태로 작성되는 JavaScript 확장 문법
- React 앱 내에서 UI를 표현할 때 사용되며, JSX는 React "element"를 생성한다.
- React는 마크업과 로직이 분리된 개별적 파일들을 통한 접근보다, loose 하게 커플링되는 "컴포넌트"를 통해 마크업과 로직이 한곳에 포함된다.

```
const element = <h1>Hello, world!</h1>;
```

React element 렌더링

React element가 문서에 실제로 노출되기 위해선 렌더링 과정이 필요하며,
HTML 문서에는 React app이 렌더링 될 container element가 필요하다.

```
ReactDOM.render(element, container[,callback]);
```

```
<!-- app container -->
<div id="app"></div>

// React element
const element = <h1>Hello, world!</h1>;

// Rendering
ReactDOM.render(element, document.getElementById("app"));
```

JSX 표현식

Embedding Expressions

JSX는 실제로 JavaScript 코드이므로, JSX내에서 JavaScript의 모든 표현식의 사용이 가능하다.

JSX내에서 expression은 중괄호(curly braces) "**{ ··· }**" 문자로 감싸 표현한다.

```
const name = "John Doe";
const element = <h1>Hello, {name}</h1>;  // Hello John Doe

<div>{ 2 + 2 }</div>  // 4
<p>{getData(123)}</p>  // getData 함수를 호출
```

JSX 속성

속성의 사용

JSX는 HTML 보다는 JavaScript에 보다 가깝기 때문에, 속성을 지정할 때는 camelCase 네이밍 컨벤션을 따른다.
따라서 기존의 TAG 속성들은 모두 JSX내에서는 camelCase로 표현해야 한다.

ex)
class → className
tab-index → tabIndex

```
// h1.title { … } css rule에 매칭
const element = <h1 className="title">Hello, {name}</h1>;
```

JSX children

자식 노드

Tag가 자식 노드를 포함하지 않는 경우에는 XML과 같이 "/>"를 통해 닫을 수 있다.

```
const element = <img src="…" />;
```

자식 노드를 포함하는 경우에는 아래와 같이 표현할 수 있다.

```
const element = (
    <div>
        <h1>Hello!</h1>
        <h2>Good to see you here.</h2>
    </div>
);
```

JSX 객체 #1

JSX Object

JSX는 React.createElement()를 통해 생성되는 객체이다.
JSX 자체는 유효한 문법이 아니기 때문에 최종적으로는 transpiling 되어야 한다.
→ 아래의 2개의 코드는 동일한 코드이다.

```
const element = (
  <h1 className="greeting">Hello, world!</h1>
);

// 위의 코드가 Babel을 통해 transpile 된 형태
// 또는 transpiler를 사용하지 않는 경우, 아래와 같이 React element를 생성한다.
const element = React.createElement("h1",
      {className: "greeting"},
      "Hello, world!"
);
```

JSX 객체 #2

Object Structure

생성된 JSX 객체는 다음과 같은 구조를 갖는다.

```
const element = (
    <h1 className="greeting">Hello, world!</h1>
);

// 위의 React element는 다음과 같은 구조를 갖는 객체가 생성된다.
{
  ...
    type: "h1",
    props: {
        className: "greeting",
        children: "Hello, world!"
    }
};
```

3. VDOM / SSR

VDOM (Virtual DOM)

JavaScript로 표현된 가상 DOM 트리

React는 내부적으로 VDOM을 통해 실제 브라우저가 렌더링 하는 DOM 영역과 React 내에서 사용되는 React element 간의 비교를 통해 실제로 변경이 발생이 필요한 요소만 업데이트 하도록 구성한다.

VDOM을 사용하는 이유는?

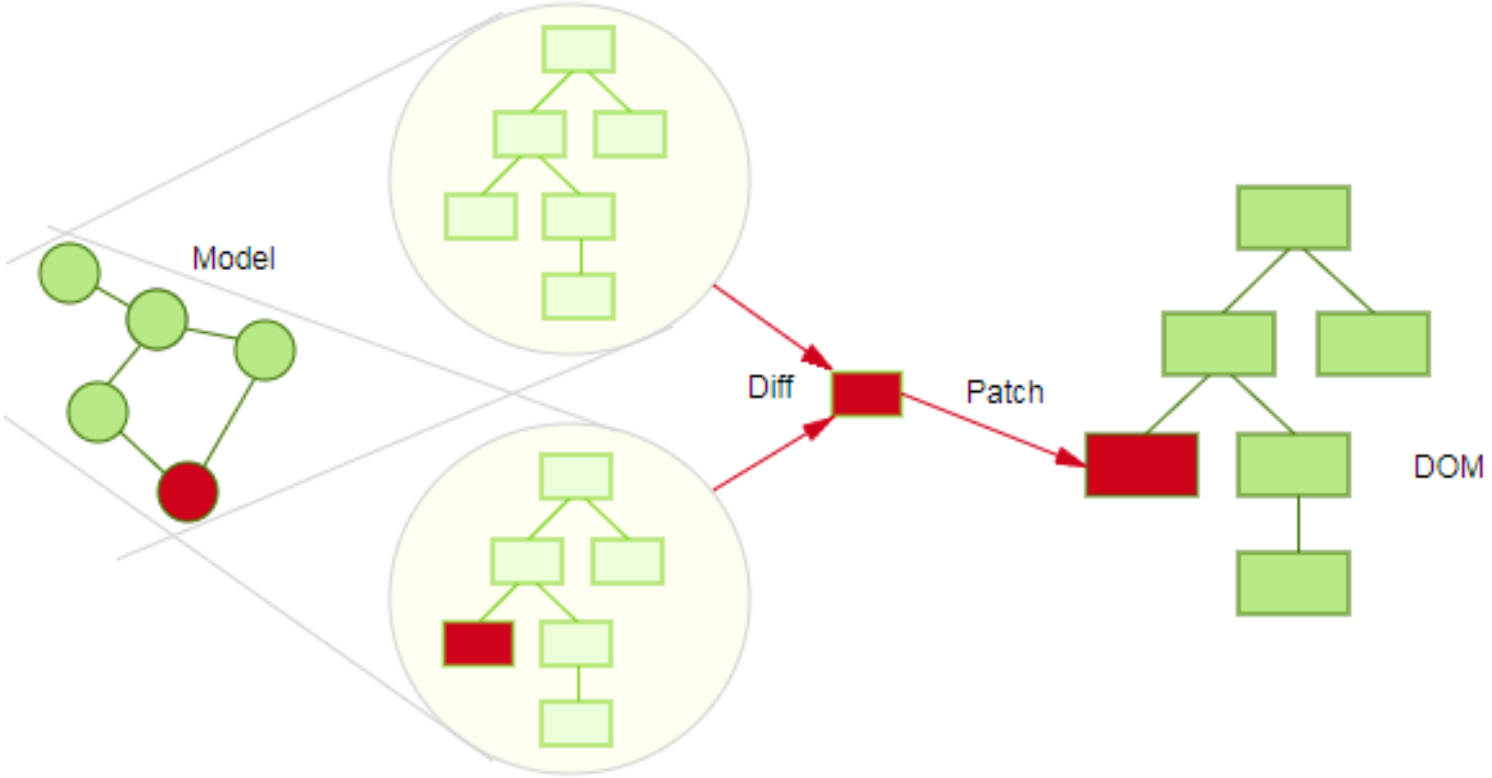
- DOM 노드 핸들링은 비용이 많이 들기 때문
- Diff 알고리즘을 통해 실제 변경이 필요한 부분만 효율적 렌더링

→ React의 diff 알고리즘: http://calendar.perfplanet.com/2013/diff/

→ Virtual DOM and diffing algorithm
https://gist.github.com/Raynos/8414846

VDOM (Virtual DOM)

가상의 DOM을 내부적으로 유지하면서, 모델이 변경되면 실제 DOM 트리와 비교 과정을 통해, 업데이트가 필요한 부분만 렌더링 한다.



SSR (Server-side Rendering)

SSR은 SPA 형태로 실행되는 앱들이 갖는 문제점(초기 정적 DOM은 app holder만을 갖는)을 해결하기 위해, 초기 렌더링 되는 정적 코드를 서버에서 반환하는 기술이다.

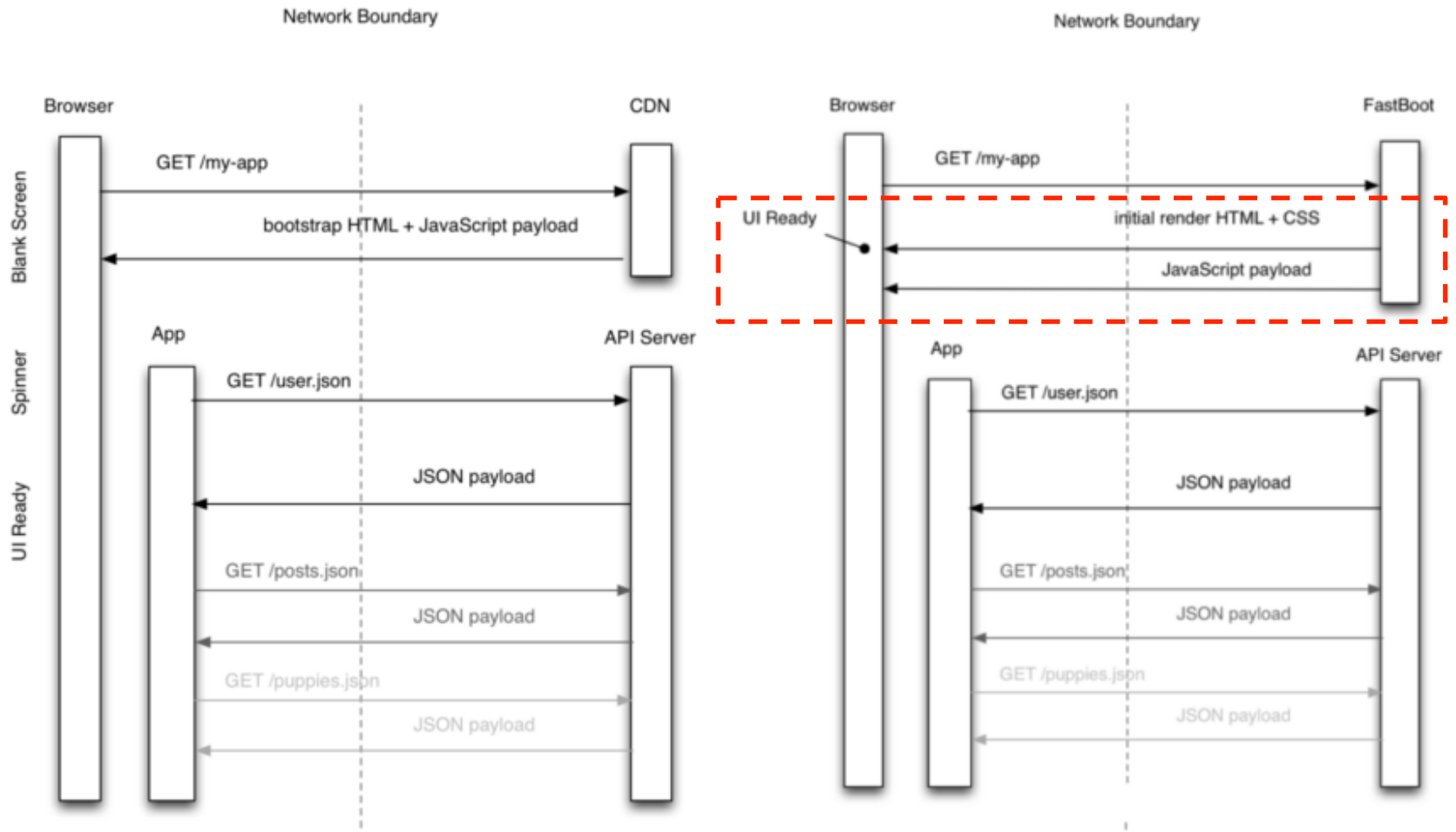
- **SEO 문제**
  → Ajax API로 구성 페이지 초기 화면이 blank로 구성되면 SEO 문제

- **초기 로딩속도**
  → 정적 자원 로딩 후, 컨텐츠 구성을 위한 추가 Ajax API 호출 시간소요

- **JavaScript 사용이 제한된 환경 대비**
  → 클라이언트 렌더링은 무용지물

실제 구현은 복잡하며, 비용이 많이 든다. React에서 SSR의 구현을 보다 쉽게 구현해 주는 대표적인 프레임워크로는 "next.js"가 있다. → https://nextjs.org/

SSR (Server-side Rendering)



→ http://tomdale.net/2015/02/youre-missing-the-point-of-server-side-rendered-javascript-apps/

4. Component & props

Component #1

컴포넌트는 UI를 독립된 형태로 나누어 재사용이 가능하게 만든다.
컴포넌트는 2가지 형태로 작성될 수 있으며, 컨벤션에 따라 첫 글자를 대문자로 작성한다.

- **Function**: Function component
- **Class**: React.Component를 상속받고, render() 메서드를 구현해야 한다.

```
function SayHello(props) {
    return <h1>Hello World!</h1>
}

class SayHello extends React.Component {
    // render 메서드가 반드시 포함돼야 한다.
    render() {
        return <h1>Hello World!</h1>
    }
}
```

Component #2

작성된 컴포넌트는 HTML tag의 형태와 같이 사용될 수 있다.
컴포넌트는 한 개의 상위 노드로 감싸지며, 여러 개의 상위 노드는 허용되지 않는다.

```
function SayHello(props) {
    return <h1>Hello World!</h1>
}

// 아래와 같은 유형은 허용되지 않는다.
function SayHello(props) {
    return <h1>Hello</h1><h1>World!</h1>;
}

const element = <SayHello />;
```

Component #3

만약 컴포넌트가 렌더링 되지 않게 하려면, null 을 반환한다.

```
function SayHello(props) {
    // 조건에 따라 null을 반환하면, <h1> 노드가 DOM에 추가되지 않는다.
    if (!props.rendering) {
        return null;
    }

    return <h1>Hello World!</h1>
}
```

props

"props"는 properties를 의미하며, 컴포넌트에 전달되는 속성 값을 의미한다.

```
const element = <SayHello name="John Doe" />;


function SayHello(props) {
    return <h1>Hello {props.name}!</h1>
}

class SayHello extends React.Component {
    render() {
        return <h1>Hello {this.props.name}!</h1>
    }
}
```

props: argument

constructor를 작성하지 않더라도, instance 멤버로 접근할 수 있다.
필요에 따라 constructor 작성이 필요한 경우라면, 반드시 생성자를 호출해야 한다.

```
class SayHello extends React.Component {
    constructor(props) {
        // constructor 작성시에는 부모 생성자를 반드시 호출해야 한다.
        super(props);

        props;  // ← 이곳에서는 argument로 전달
    }

    render() {
        // 이곳에서는 instance로 접근
        return <h1>Hello {this.props.name}!</h1>
    }
}
```

props: Read-only

React 컴포넌트는 유연하지만, 하나의 엄격한 룰을 갖는다.
모든 컴포넌트는 props에 대해 "**Pure function**"처럼 동작해야 한다.

Pure function은 전달되는 arguments(input)의 값을 변경하지 않는 함수를 의미한다.

```
// pure function
function sum(a, b) {
    return a + b;
}

// impure function
function total(obj, val) {
    return obj.total += val;
}
```

→ https://en.wikipedia.org/wiki/Pure_function

props: overriding

- props를 override 하는 경우에는 props 뒤에 명시한다.
- 앞에 명시하는 경우에는, 전달된 props로 override 된다.

```
cost props = {
    className: "myClass"
}

const element = <div {...props} />  // className="myClass"

// props의 className으로 override: className="myClass"
const element = <div className="myAClass" {...props} />

// myAClass로 지정: className="myAClass"
const element = <div {...props} className="myAClass" />
```

props: style

style 속성은 객체로 작성한다.

```
const element = (
    <div style={
            {backgroundColor: "red", color: "#fff"}
        } className="box">
        I'm Text
    </div>
);
```

노드 레퍼런스: Refs

상황에 따라 element에 대한 레퍼런스를 직접 얻어와 핸들링하는 필요성이 생긴다.
이 경우, Refs를 통해 element에 대한 레퍼런스를 얻어올 수 있다.

→ **참고:** Function component는 인스턴스가 생성되지 않으므로, ref attribute를 사용하면 안된다.

```
class MyComponent extends React.Component {
   constructor(props) {
      super(props);
      this.myRef = React.createRef();  // (1) Ref를 생성
   }

   clickHandler(event) {
      this.myRef.current;  // (3) <div>에 대한 레퍼런스에 접근시 'current' 속성으로 접근
   }

   render() {
      // (2) element에 ref 속성을 통해 레퍼런스를 얻을 수 있도록 설정
      return <div ref={this.myRef} onClick={this.clickHandler} />;
   }
}
```

노드 레퍼런스: Callback Refs

Refs를 생성(React.createRef())하지 않고, ref 속성에 callback 함수를 설정해 element의 레퍼런스를 저장하고 사용할 수도 있다.

```
class MyComponent extends React.Component {
    this.inputNode = null;

    clickHandler(event) {
        this.inputNode.value;  // 전달된 element 레퍼런스를 통해 값을 얻을 수 있다.
    }

    render() {
        // callback 함수에 React component 인스턴스 또는 HTML DOM element가 전달
        return  <input type="text" name="username" ref={
            node => {this.inputNode = node}
        } onClick={this.clickHandler} />
    }
}
```

Component의 재사용

작성된 컴포넌트는 자유롭게 재사용이 가능하다.

```
function Welcome(props) {
    return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Jessica" />
      <Welcome name="Maria" />
    </div>
  );
}

ReactDOM.render(<App />, document.getElementById("app"));
```

Component 분리 #1

컴포넌트의 특징은 유연함과 재사용에 있다.

한 개의 컴포넌트를 너무 큰 단위로 구성하면 복잡도의 증가와 재사용에 대한 유연성을 떨어뜨리게 된다.

```
function UserInfo(props) {
  return (
      <div className="UserInfo">
          <img className="Avatar"
              src={props.author.avatarUrl}
              alt={props.author.name} />
          <div className="UserInfo-name">{props.author.name}</div>
      </div>
  );
}
```

Component 분리 #2

보다 작은 단위의 컴포넌트들로 구분해 재사용성을 높이는 것이 좋다.

```
function Avatar(props) {
    return (
      <img className="Avatar"
        src={props.author.avatarUrl}  alt={props.author.name} />
    );
}

function UserInfo(props) {
    return (
        <div className="UserInfo">
            <Avatar user={props.user} />
            <div className="UserInfo-name">{props.author.name}</div>
        </div>
    );
}
```

5. PropTypes: 타입 체크

PropTypes?

- PropTypes는 built-in 타입 검사 라이브러리이다.
- React component로 전달되는 props의 값에 대한 타입을 정의 또는 값의 필수 유무를 지정할 수 있다.

```
# prop-types 패키지 설치
$ npm install --save prop-types

<!-- 브라우저에서 로딩하는 경우 -->
<script src="https://unpkg.com/prop-types@15.6.1/prop-types.js"></script>
```

- 전달된 props 값의 데이터 타입이 일치 하지 않거나 또는 값이 필수인데 지정되지 않는 경우 오류가 발생된다.

```
▶Warning: Failed prop type: Invalid prop `firstName` of type `number`
supplied to `SayHello`, expected `string`.
    in SayHello
```

→ https://reactjs.org/docs/typechecking-with-proptypes.html

PropTypes: 타입 정의

PropTypes는 built-in 타입 검사 라이브러리로, props의 값에 대한 타입을 정의하고 검사한다.

```
// props 타입에 대한 정의
MyComponent.propTypes = {
  optionalArray: PropTypes.array,
  optionalBool: PropTypes.bool,
  optionalFunc: PropTypes.func,
  optionalNumber: PropTypes.number,
  optionalObject: PropTypes.object,
  optionalString: PropTypes.string,
  optionalSymbol: PropTypes.symbol,
  ...
};
```

→ 데이터 타입: https://reactjs.org/docs/typechecking-with-proptypes.html#proptypes

PropTypes: 필수 지정 및 기본 값

props의 값의 필수 조건 또는 기본 값을 정의할 수 있다.

- 필수 조건: 타입 뒤에 ".isRequired" 지정
- 기본 값: defaultProps 객체 값을 설정

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

class SayHello extends React.Component {
    // props 타입에 대한 정의
    static get propTypes() {
        return {
            name: PropTypes.string,
            age: PropTypes.number.isRequired  // age는 필수 값으로 설정
        };
    }

    // props 타입 기본 값 정의
    static get defaultProps {
        return {
            name: "John Doe"  // name의 기본 값은 "John Doe"로 설정
        }
    };

    render() { … }
}
```

PropTypes: 타입 정의 제거

- dev 모드에선 타입 체크가 필요하지만, production에서는 불필요한 오버헤드와 타입 정의는 불필요한 용량을 차지하기 때문에 reatc-dom.production에서는 제거되어 있다.
- babel-plugin-transform-react-remove-prop-types Babel plugin을 사용해 transpile시 제거하도록 구성할 수도 있다.

```
# 설치
$ npm install --save-dev babel-plugin-transform-react-remove-prop-types

// babel.config.js 추가
plugins: [ "transform-react-remove-prop-types" ],
```

```
// input
const Baz = (props) => (
    <div {...props} />
);

Baz.propTypes = {
    className: PropTypes.string
};
```

```
// output
const Baz = (props) => (
    <div {...props} />
);
```

6. State

State

- State는 props와 유사하나, private 이면서 컴포넌트를 통해 관리된다는 점에서 다르다.
- State는 컴포넌트 내에서 갖는 '상태'에 대한 값을 의미한다.
- **State 값이 변경되면 .render() 메서드가 재호출된다.**

```
// props를 사용하는 경우
function Clock(props) {
    return (<div>현재시간: {props.date.toTimeString()}</div>);
}

ReactDOM.render(<Clock date={new Date()} />, root);
```

```
// state를 사용하는 경우
class Clock extends React.Component {
    constructor(props) {
        super(props);  // ← 전달되는 props 값이 있는 경우, 상위 컴포넌트에 값을 전달해야 한다.
        this.state = {date: new Date()};
    }

    render() {
        return (<div>현재시간: {this.state.date.toTimeString()}</div>);
    }
}

ReactDOM.render(<Clock />, root);
```

State 값의 설정

State의 값을 설정하는 경우에는, 항상 this.setState() 메서드를 사용해야 한다.

```
// state를 사용하는 경우
class Clock extends React.Component {
    constructor(props) {
        super(props);  // ← 전달되는 props 값이 있는 경우, 상위 컴포넌트에 값을 전달해야 한다.
        this.state = {date: new Date()};
    }

    tick() {
        this.setState({
            date: new Date()
        });
    }

    render() {
        (<div>현재시간: {this.state.date.toTimeString()}</div>);
    }
}

ReactDOM.render(<Clock />, root);
```

State 주의사항 #1

- 메서드를 사용하지 않는 직접 값 변경은 피해야 한다.
- 직접적인 접근을 통해 값을 변경하게 되면 컴포넌트가 재 렌더링 되지 않는다.

```
// 잘못된 사용
this.state.name = "John Doe";

// 올바른 사용
this.setState({ name: "John Doe" });
```

State 주의사항 #2

- React는 내부적으로 여러 개의 setState() 작업을 단일 업데이트로 최적화해 수행한다.
- this.props와 this.state는 비동기로 값이 업데이트 될수도 있기 때문에, 직접적인 값의 접근은 최적화 또는 업데이트된 올바른 값을 얻지 못하게 만들 수 있다.

```
// 잘못된 사용
this.setState({
    counter: this.state.counter + this.props.increment,
});

// 올바른 사용
this.setState((state, props) => ({
    counter: state.counter + props.increment
}));
```

State 주의사항 #3

setState()가 호출되면, React는 기존의 state 객체와 병합(merge)한다.

```
constructor(props) {
    super(props);

    this.state = {
       posts: [],
       comments: "Hello World"
    };
}

 method() {
      // this.state = { posts: 12345, comments: "Hello World" }
      this.setState({ posts: 12345 });
}
```

하위 컴포넌트로 state 전달

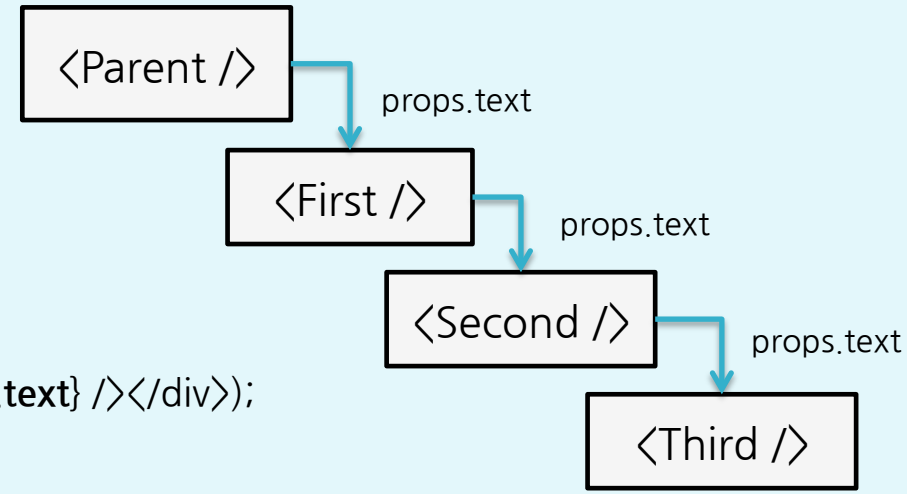
- 각 컴포넌트들은 상태 값을 갖거나(stateful) 갖지 않을 수(stateless) 있다.
- 상태 값은 주로 해당 컴포넌트에 국한되어 있기 때문에 "local" 또는 "encapsulated"라 할수 있다.
- 하지만, 상황에 따라 컴포넌트의 '상태'를 하위 컴포넌트로 전달해야 하는 경우가 필요할 수도 있으며, 이 경우 props를 통해 값을 전달할 수 있다.

```
// Parent 컴포넌트
<MyDate date={this.state.date} />

function MyDate(props) {
    return <h2>현재시간: {props.date.toTimeString()}.</h2>;
}
```

prop drilling (or threading)

- "prop drilling"은 상위 컴포넌트의 '상태' 값을 하위 컴포넌트로 전달하기 위해 "drilling" 하는 것과 같다는 데서 유래 되었다.
- 값이 전달되어야 할 하위 컴포넌트가 많고, 중첩된 형태로 사용되는 경우, 관리가 어렵고 복잡해 진다.

```
class Parent extends React.Component {
    constructor() {
        super();
        this.state = {text: "Hello World!"};
    }

    render() {
        return <First text={this.state.text} />;
    }
}

const First = props => (
    <div><h1>First</h1> <Second text={props.text} /></div>);

const Second = props => (
    <div><h2>Second</h2> <Third text={props.text} /></div>);

const Third = props => (
    <div><h3>Third</h3> Text: {props.text}</div>);
```



7. Context API

Context API? (v16.3+)

Context는 React component 트리에서 데이터를 '전역'(global) 변수와 같이 공유하고 싶을때 사용될 수 있다. (ex. 사용자 정보는 모든 영역에서 공유필요)

- **Provider**: 해당 context로 값을 설정/전달 하는 경우에 사용
- **Consumer**: Context의 값을 가져오는 경우에 사용

```
// Context 생성. 초기 값을 지정할 수도 있다.
const MyContext = React.createContext("light");

const MyProvider = MyContext.Provider;
const MyConsumer = MyContext.Consumer;

// Destructuring으로 별도 변수에 할당해 사용하는 경우
const { Provider: MyProvider, Consumer: MyConsumer } = React.createContext();
```

Context API #1

생성된 context의 Provider로 하위 element들을 래핑해 context를 전달한다.

```
class App extends React.Component {
  render() {
    // Context Provider를 통해 하위 노드들로 전달한다.
    // value 속성을 통해 사용될 값을 설정한다.
    return (
      <MyContext.Provider value="dark">
        <Toolbar />
      </MyContext.Provider>
    );
  }
}
```

Context API #2

- Provider로 래핑된 하위 element에서 Consumer를 통해 context의 값에 접근할 수 있다.
- Context consumer는 단일 child 형태인 function이여야 한다.

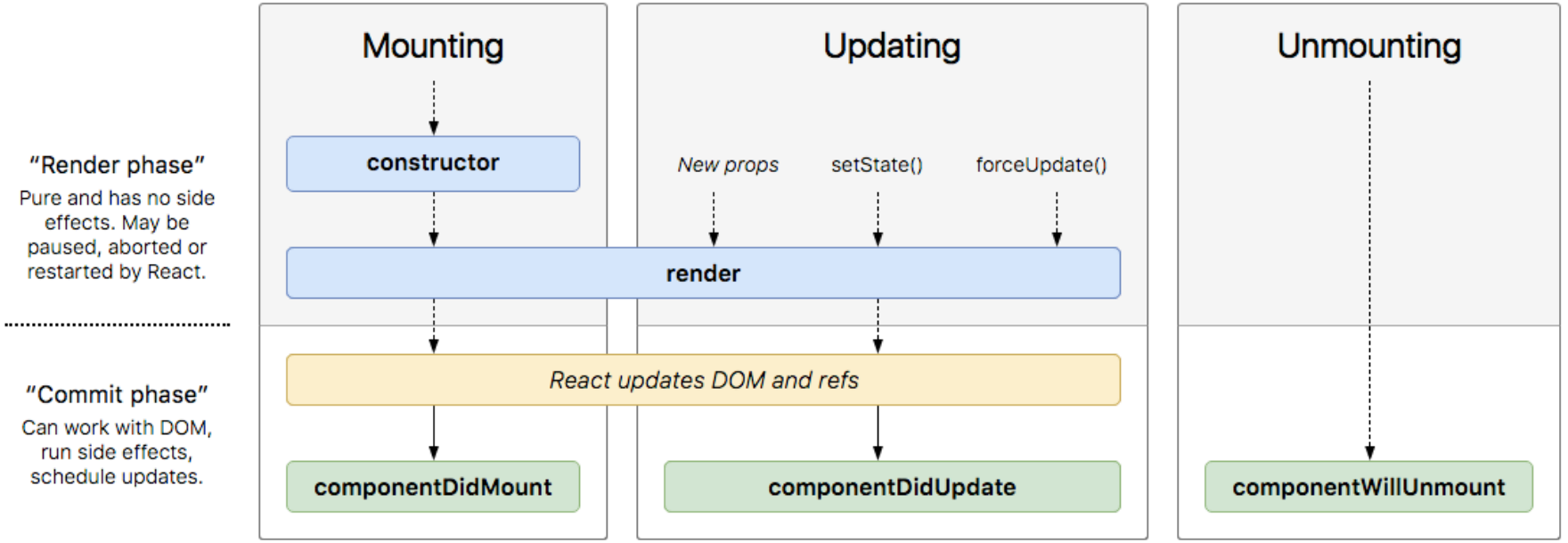
```
// 중간에 위치한 컴포넌트는 하위로 별도의 값을 전달하지 않아도 된다.
function Toolbar(props) {
    return (
        <div><MyButton /></div>
    );
}

class MyButton extends React.Component {
    render() {
        return
            <MyContext.Consumer>
                {context => <Button theme={context} />}
            </MyContext.Consumer>;
    }
}
```

8. Component Lifecycle

Lifecycle (v16.3 기준)

각각의 컴포넌트들은 생명주기에 따른 "Lifecycle Methods"들을 갖는다.

컴포넌트의 인스턴스가 생성되고 DOM에 추가되는 프로세스 과정에서 메서드를 override 하면, 특정 시점에 필요한 코드를 실행시킬 수 있다.



→ http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

Lifecycle Methods

컴포넌트의 Lifecycle methods 들은 각 단계에 따라 아래의 나열된 순서대로 호출된다.
→ 굵게 표현된 메서드들이 일반적으로 사용되며, 나머지들은 사용은 일부의 경우에 한해 사용된다.

1) Mounting
- **constructor()**
- static getDerivedStateFromProps()
- **render()**
- **componentDidMount()**

2) Updating
- static getDerivedStateFromProps()
- shouldComponentUpdate()
- **render()**
- getSnapshotBeforeUpdate()
- **componentDidUpdate()**

3) Unmounting
- **componentWillUnmount()**: 컴포넌트가 DOM에서 제거될 때 실행

```
// unmount는 아래 메서드를 통해 할수 있다.
ReactDOM.unmountComponentAtNode(
    containerElement
);
```

Lifecycle Methods: Deprecation

- React v16.3 릴리스를 통해 기존의 lifecycle 들중 아래의 lifecycle들이 deprecate 되었다.
- v17 릴리스까지는 지원될 예정이라고 밝히고 있지만, 제거될 항목이기 때문에 사용을 하지 않는 것이 좋다.
  → v17에서는 "UNSAFE_" prefix로 alias로 처리해 deprecation을 경고

- componentWillMount
- componentWillReceiveProps
- componentWillUpdate

→ https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html#gradual-migration-path

9. Event Handling

Event binding

- React에서 이벤트 처리는 HTML에서의 인라인 이벤트 바인딩과 유사하다.
- 이벤트명은 React element 네이밍 컨벤션에 따라 camelCase 형태로 사용된다.

```
<!-- HTML의 인라인 이벤트 바인딩 -->
<button onclick="showLayer()">Show Layer</button>

// React element의 이벤트 바인딩
<button onClick={showLayer()}>Show Layer</button>
```

대다수의 기존 네이티브 이벤트들의 사용이 가능하다.
지원되는 이벤트 목록들은 아래 링크를 통해 확인할 수 있다.
→ https://reactjs.org/docs/events.html#supported-events

Synthetic(합성) Event

- 이벤트 핸들러에는 네이티브 이벤트가 아닌, SyntheticEvent로 래핑된 이벤트 인스턴스가 전달된다.
- SyntheticEvent는 네이티브 이벤트 객체와 유사하며, 다음과 같은 구조를 갖는다.
- 래핑되지 않은 네이티브 이벤트로의 접근은 "*event*.nativeEvent" 속성을 통해 접근할 수 있다.

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent  # 네이티브 이벤트에 접근할 수 있다.
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
DOMEventTarget target
number timeStamp
string type
```

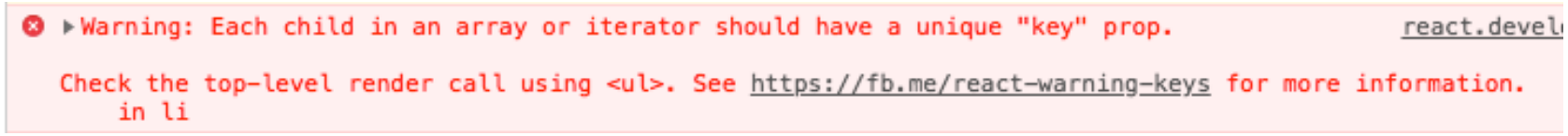
10. List & key props

List 처리

반복문을 통해 목록을 갖는 React component의 생성이 가능하다.

```
const listItems = [1, 2, 3, 4, 5].map(number => <li>{number}</li>);

ReactDOM.render(<ul>{listItems}</ul>, document.getElementById("app"));
```

그러나 위의 코드를 실행해 보면, 아래와 같이 unique한 "key" prop가 지정되지 않았다는 경고 메세지가 출력된다.




```
const listItems = [1, 2, 3, 4, 5].map(number => (
    <li key={number}>{number}</li>
));
```

key props?

오류를 해결하기 위해선, 'key' props를 아래와 같이 지정하면 해결할 수 있다.

```
const listItems = [1, 2, 3, 4, 5].map(number => (
    <li key={number}>{number}</li>
));
```

그렇다면 "key" props는 왜 필요할까?

- key는 React로 하여금, 해당 요소가 변경, 추가, 삭제 등의 작업 수행시 해당 element를 구분할 수 있도록 해준다.
- key는 unique 해야 하지만, Sibling 레벨에서만 그렇다. 전역영역에서 unique할 필요는 없다.

key props - 유의사항

"key"는 배열을 통해 생성되는 element 처리에서만 의미를 갖는다.

```
function ListItem(props) {
  const value = props.value;
  return (
      // 이곳에 key props를 지정하는 것은 의미없다.
      <li key={value.toString()}>{value}</li>
  );
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
      // key props는 여기에 지정되어야 한다.
      <ListItem key={number} value={number} />
  );

  return (<ul>{listItems}</ul>);
}
```

11. Form handling

Form

- 일반적인 <form> 요소는 입력/선택된 값을 기준으로 상태가 유지된다.
- React는 변경(mutable)이 발생하는 '상태'의 관리는 setState()를 통해서 이뤄지게 된다.
- 이처럼 React의 '상태' 관리와 <form> 요소의 '상태' 관리의 차이가 존재하기 때문에, <form>에 대한 '상태'를 "**Single source of truth**"로 만드는 것이 필요하다.

Form

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ""};
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
      this.setState({value: event.target.value});
  }

  handleSubmit(event) {
      alert("Submit된 값 " + this.state.value);
      event.preventDefault();
  }

  render() {
    return (<form onSubmit={this.handleSubmit}>
          Name: <input type="text" value={this.state.value} onChange={this.handleChange} />
          <input type="submit" value="Submit" />
      </form>);
   }
}
```

Form: HTML 요소와의 차이점

<textarea>: 값에 대한 접근

- HTML: text 값은 childNode로 접근
- React: <textarea> element의 *textElement*.**value** 속성으로 접근

<select>: 기본 option 선택

- HTML: <select><option **selected** value="…"></select> 에서 selected를 통해 기본값 선택
- React: 상위 <select **value**={선택자-값}> … </select> 요소에서 'value'를 통해 기본 값 선택

입력 제어

- input 요소에 value 속성의 값이 지정된 경우에는 값의 입력이 허용되지 않는다.
- value가 undefined 또는 null인 경우에는 값이 입력된다.

```
// input 요소의 값이 수정되지 않는다.
ReactDOM.render(<input value="hi" />, document.getElementById("root"));

// input 요소에 값을 수정할 수 있다.
setTimeout(function() {
    ReactDOM.render(<input value={null} />, document.getElementById("root"));
}, 1000);
```

Form: element value

input element의 값을 얻어오는 방법:
- 이벤트 객체를 통한 직접 접근
- Element의 ref 속성

```
class NameForm extends React.Component {
    handleSubmit = event => {
        // 네이티브 이벤트 객체를 통해 접근하는 경우
        console.log(event.target[0].value);

        // name 속성으로 접근하는 경우
        console.log(event.target.elements.username.value);

        // ref 속성을 통해 전달한 경우
        console.log(this.inputNode.value);
    }

    render() {
        return (<form onSubmit={this.handleSubmit}>
                Name: <input type="text" name="username" ref={node => {this.inputNode = node}} />
                <button type="submit">Submit</button>
            </form>);
    }
}
```

12. React Hooks

React Hooks

React v16.8에 추가된 기능으로 함수형(not class) 컴포넌트에서 사용되는 함수(빌트인, 또는 커스텀)를 말하며, 함수형 컴포넌트내에서 React의 기능을 **“Hook”** 할 수 있게 만들어 준다.

Hook은 기존 컴포넌트 모델이 갖는 다음의 문제들의 해결을 위해 만들어 졌다.

- 컴포넌트들 간에 ‘상태’를 갖는 로직이 재사용되기 어려운 문제
- 복잡한 컴포넌트는 이해되기 어렵다.

→ Hook은 컴포넌트의 기존 구조를 변경하지 않고 상태 값을 갖는 로직을 재사용할 수 있게 만들어 준다.

→ Hook은 컴포넌트를 작은 함수들로 분리해 보다 단순화 시켜준다.

React Hooks (v16.8+)

Hook을 사용하는 경우, 2가지 룰을 따라야 한다.

- 상위 레벨(top level)에서 호출되어야 한다.
  → 이는 반복문, 조건식 또는 중첩된 함수 내에서 호출되지 않아야 하는 것을 의미

- React 함수형 컴포넌트(또는 custom hook)에서 사용되어야 한다.
  → 일반 JavaScript 함수에서 호출되어서는 안된다는 것을 의미

또한, Hook은 접두사로 소문자 '**use**'를 갖는 네이밍 컨벤션을 갖는다.
→ ex. useState, useEffect, etc.

→ https://reactjs.org/docs/hooks-rules.html

(Built-in) State Hook: useState #1

상태를 갖는(stateful) 값과 그 상태를 업데이트 할수 있는 함수를 반환하는 빌트인 hook 이다.

```
const [state, setState] = useState("초기-상태-값");
```

```
// 다음은 버튼 클릭 시, 값이 증가하는 클래스 컴포넌트의 예이다.
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 0};
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

useState #2

기존의 클래스 컴포넌트는 hook을 사용해 함수형 컴포넌트로 단순화 시킬 수 있다.

```
Import React, {useState} from "React";

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

  return (
      <div>
          <p>You clicked {count} times</p>
          <button onClick={() => setCount(count + 1)}>
            Click me
          </button>
      </div>
  );
}
```

useState #3: Functional update

"setState" 함수에 함수를 전달하는 경우, 이전 상태에 대한 값이 파라미터로 전달된다.

```
Import React, {useState} from "React";

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

  return (
      <div>
          <p>You clicked {count} times</p>
          <button onClick={() => setCount(prev => prev + 1)}>+</button>
          <button onClick={() => setCount(prev => prev - 1)}>-</button>
      </div>
  );
}
```

useState #4: Lazy initial state

useState의 초기값이 비용이 많이 발생하는 작업으로 부터 얻어온다면, 매 렌더링 시마다 수행하는 것은 성능 저하의 요인이 될수 있다.

이 경우, useState의 **초기값으로 함수**를 설정하면 초기 렌더링 시에만 수행된다.

```
Import React, {useState} from "React";

function Counter() {
  // useState()에 함수를 설정하면, 초기 렌더링시에만 수행된다.
  const initCount = () => Number(window.localStorage.getItem("count") || 0);
  const [count, setCount] = useState(initCount);

  return (
      <div>
          <p>You clicked {count} times</p>
          <button onClick={() => setCount(count + 1)}>+</button>
      </div>
  );
}
```

(Built-in) Effect Hook: useEffect #1

"effect" 명령을 수행하는 함수를 설정하며, lifecycle method인 componentDidMount와 componentDidUpdate의 실행 시점과 유사하게 동작한다.

단, effect에 설정된 함수는 lifecycle method와는 다르게, 모든 렌더링 시점(layout과 paint)마다 수행된다.

"Side Effect"(또는 줄여서 effect)란 함수의 개념적(conceptual) 행위에 대한 것을 말한다.

함수가 "side effect"를 갖고 있다라고 한다면, 이는 함수 밖의 영역에 대한 변경을 포함한다는 것을 의미한다.
→ 예) 전역 변수를 변경하는 코드가 포함된 함수는 "side effect"를 갖는다.

```
useEffect(() => {
    // componentDidMount 시점에 수행될 코드
    // componentDidUpdate 시점에 수행될 코드
    ...
});
```

useEffect #2: Conditionally firing

조건에 따른 수행은 2번째 파라미터를 통해 가능하다.

비어있는 배열을 지정하는 경우, 컴포넌트가 마운트 되는 시점에만 수행되게 한다.

```
useEffect(() => {
    ...
}, []);  // ← 컴포넌트가 마운트 되는 시점에만 실행
```

값이 변경되는 경우에만 수행되게 하려면, 해당 값을 배열로 전달하면 된다.

```
useEffect(() => {
    ...
}, [value]);  // ← value가 업데이트 되는 시점에만 실행
```

useEffect #3: Clean-up

컴포넌트가 unmount 될 때, clean-up 작업의 수행이 필요할 때가 있다.
→ ex) 컴포넌트에서 바인딩 된 이벤트가 있다면, umount시 이벤트 제거가 필요하다.

이 경우, useEffect에 전달하는 함수에서 다시 clean-up 수행 함수를 반환하면 된다.

```
const [width, setWidth] = useState(window.innerWidth);

useEffect(() => {
    const handler = e => setWidth(window.innerWidth);

    window.addEventListener("resize", handler);

    // clean-up 코드를 포함하는 함수를 반환
    return () => {
        window.removeEventListener("resize", handler);
    };
});
```

(Built-in) Context Hook: useContext

React.createContext(Context API)를 통해 반환된 context 객체를 설정하며, 현재 context 값을 반환한다.

```
const TextContext = React.createContext();

// 잘못된 사용: useContext(TextContext.Consumer)
// 잘못된 사용: useContext(TextContext.Provider)
const value = useContext(MyContext);
```

```
// using Context API
<MyContext.Consumer>
   {context => <Button theme={context} />}
</MyContext.Consumer>;


// using with context hook
const context = useContext(MyContext);

<Button theme={context} />}
```

Custom Hook #1

재사용이 가능한 로직을 분리해 별도의 커스텀 hook(함수)을 만들 수 있다.
함수명은 반드시 "use" 접두사를 사용해 네이밍 되어야 한다.

1) 다음 코드는 온라인 상태를 출력하는 함수 컴포넌트의 예이다.

```
function FriendListItem(props) {
    const [isOnline, setIsOnline] = useState(null);

    useEffect(() => {
        function handleStatusChange(status) {
            setIsOnline(status.isOnline);
        }

        ChatAPI.subscribe(props.id, handleStatusChange);

        return () => { ChatAPI.unsubscribe(props.id, handleStatusChange); };
    });

    return <li style={{ color: isOnline ? 'green' : 'black' }}>{props.name}</li>;
}
```

→ https://reactjs.org/docs/hooks-custom.html

Custom Hook #2

2) 온라인 상태를 확인하는 로직을 재사용이 가능한 custom hook으로 분리하고 사용될 수 있다.

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribe(props.id, handleStatusChange);

    return () => { ChatAPI.unsubscribe(props.id, handleStatusChange); };
  });

  return isOnline;
}

function FriendListItem(props) {
  const isOnline = useFriendStatus(props.id);

  return <li style={{ color: isOnline ? 'green' : 'black' }}>{props. name}</li>;
}
```

Additional Hooks

Basic Hooksd 외에도 추가적으로 다음의 hook들이 제공되며, 모든 경우에 hook을 사용해야 하는 것은 아니기 때문에 필요한 경우에 한해 사용을 고려하면 된다.

- **useReducer:**
  useState의 또다른 대체제 형태로, reducer와 상태값을 지정하면 현재 상태 값과 dispatch 메서드를 반환한다.
- **useCallback:** memoized 콜백을 반환한다.
- **useMemo:** memoized 값을 반환한다.
- **useRef:** 노드의 refs에 대한 hook
- **useImperativeHandle:** refs가 사용되는 컴포넌트의 인스턴스 커스터마이징
- **useLayoutEffect:**
  useEffect와 동일하나, DOM의 모든 변경된 후 동기적으로 처리되는 점이 다르다. (가능한 모든 경우에 useEffect의 사용이 권장)
- **useDebugValue:** 커스텀 hook 디버그시 커스텀 hook 이름 텍스트를 출력해 준다.

13. React Router

React Router

React Router

- React는 View에 집중된 프레임워크이기 때문에, 라우팅 기능은 built-in으로 지원하지 않는다.
- 라우팅을 사용하기 위해선 third-party 패키지를 사용해야 하며, React Router는 그 중 하나이다.
  → https://reacttraining.com/react-router/

```
# react-router 패키지 설치
$ npm install --save react-router-dom
```

브라우저에서 로딩해서 사용하는 경우

```
<script src="https://unpkg.com/react-router/umd/react-router.min.js"></script>
<script src="https://unpkg.com/react-router-dom/umd/react-router-dom.min.js"></script>

<script type="text/babel">
    const {BrowserRouter, Route, Link} = ReactRouterDOM;
</script>
```

React Router: <BrowserRouter>

- HTML5 history API를 사용한 라우팅 기능을 제공
- 라우팅 대상 element들을 래핑하는 최상위 요소

```
<BrowserRouter>
    <div>
      <nav>
        <Link to="/">Home</Link> /
      </nav>
    </div>
  </BrowserRouter>
```

→ API: https://reacttraining.com/react-router/web/api/BrowserRouter

React Router: <Link>

Anchor 요소로, 다른 곳으로의 이동을 위한 link를 정의

```
<BrowserRouter>
    …
    <Link to="/">Home</Link> /
    <Link to="/about">About</Link>
</BrowserRouter>
```

→ API: https://reacttraining.com/react-router/web/api/Link

React Router: <Route>

매칭되는 location(path props)에 대한 UI를 렌더링 한다.
렌더링 UI는 다음 중 한가지의 형태를 지정할 수 있다.

- **component** (React component): 렌더링할 컴포넌트
- **rendering** (Function): 렌더링 요소를 반환하는 함수
- **children** (Function): 매칭 여부와는 상관없이 항상 렌더링

```
const Index = () => <h2>Home</h2>;

  <BrowserRouter>
      …
        <Route path="/" exact component={Index} />
        <Route path="/about/" render={() => <h2>About</h2>} />
        <Route children={({ match, …rest }) => (<span>항상노출</span>)} />
  </BrowserRouter>
```

→ API: https://reacttraining.com/react-router/web/api/Route

React Router: 예제

3개의 link를 갖는 라우팅 예제 앱

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

const Index = () => <h2>Home</h2>;
const About = () => <h2>About</h2>;
const Users = () => <h2>Users</h2>;

const AppRouter = () => (
  <BrowserRouter>
    <div>
      <nav>
        <Link to="/">Home</Link> /
        <Link to="/about/">About</Link> /
        <Link to="/users/">Users</Link>
      </nav>

      <Route path="/" exact component={Index} />
      <Route path="/about/" component={About} />
      <Route path="/users/" component={Users} />
    </div>
  </BrowserRouter>
);

ReactDOM.render(<AppRouter />, document.getElementById("root"));
```

14. 도구

Create React App

React 앱을 개발 환경을 구성해 주는 스캐폴딩(scaffolding) 도구
→ https://facebook.github.io/create-react-app/

- 의존성 모듈들에 대한 설치와 구성
- 개발서버 환경 구성
- 기본 작업 파일 구조 생성
- npm script 명령어 구성
  → npm start : 개발서버 실행
  → npm run build : 빌드 실행

```
my-app
├── README.md
├── node_modules
├── package.json
├── .gitignore
├── public
│   ├── favicon.ico
│   ├── index.html
│   └── manifest.json
└── src
    ├── App.css
    ├── App.js
    ├── App.test.js
    ├── index.css
    ├── index.js
    ├── logo.svg
    └── serviceWorker.js
```

```
# 'my-app'이라는 React 앱을 만드는 경우
# 패키지 설치없이 한번만 패키지를 사용하는 경우에는 npm의 'npx' 명령어를 사용한다.
$ npx create-react-app my-app

# 자주 사용하는 경우에는 global로 설치
$ npm install -g create-react-app
```

React Developer Tools

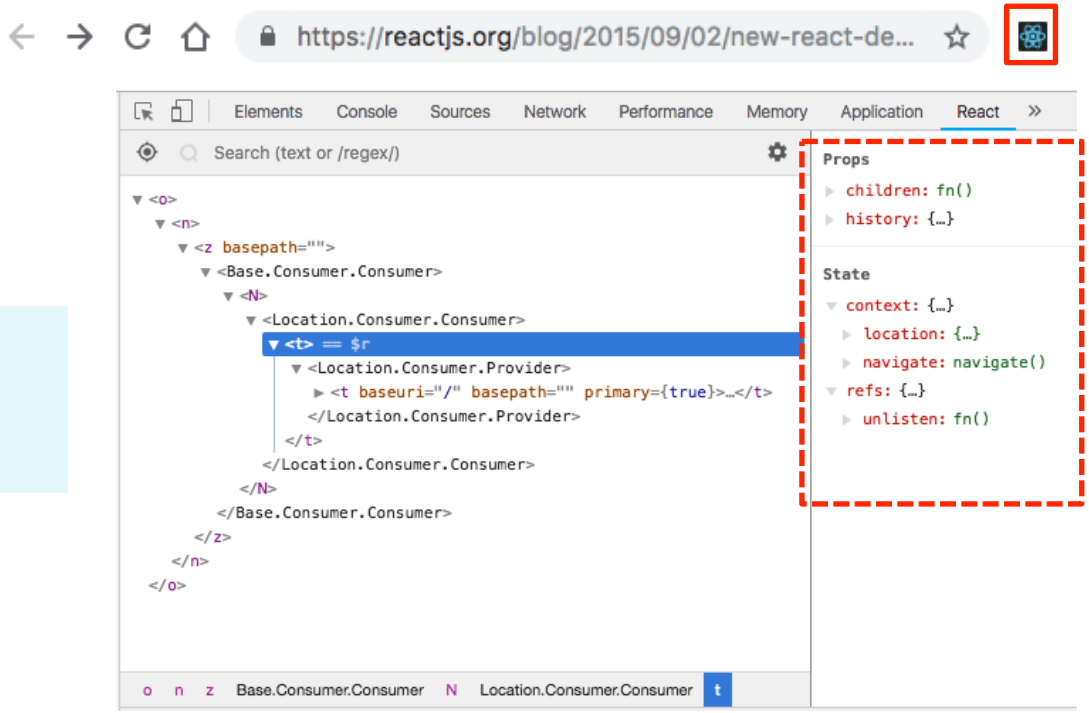
React Developer Tools
제공자: Facebook

React Developer Tools는 크롬 익스텐션 또는 Mozilla add-ons로 설치되는 브라우저 확장 프로그램이다.

기본 기능:

- React element tree 구조를 확인
- Props와 state 값의 확인 및 수정

크롬 웹 스토어에서 설치
https://goo.gl/kNFjs9



→ https://reactjs.org/blog/2015/09/02/new-react-developer-tools.html

15. CSS-in-JS

styled components

- CSS-in-JS는 JavaScript내에 CSS 코드가 포함되는 것을 말한다.
- 컴포넌트 단위의 UI에서 관리되기 어려운 CSS를 컴포넌트 자체에 포함시켜 관리할 수 있게 한다.

- 다양한 CSS-in-JS 도구/라이브러리들이 있으며, styled component는 React 생태계에서 가장 인기있는 라이브러리다.
  → https://www.styled-components.com/

```
# styled components 패키지 설치
$ npm install --save styled-components
```

브라우저에서 로딩해서 사용하는 경우, **window.styled** 객체를 통해 접근한다.

```
<script src="https://unpkg.com/styled-components/dist/styled-components.min.js"></script>

<script>
    const Component = window.styled.div`color: red;`
</script>
```

기본 사용 방법

- Tagged Template Literal을 사용해 스타일을 갖는 element 생성한다.
- "styled.태그" 형식의 Tagged Template Literal을 통해 스타일을 정의한다.

```
import styled from "styled-component";

// 지정한 스타일을 갖는 <Element>를 정의한다.
const Title = styled.h1`
    font-size: 1.5em;
    text-align: center;
    color: red;
`
// 생성된 노드는 임의의 클래스가 부여되며, 정의한 스타일을 클래스에 포함된다.
// <h1 class='fRAMMT'>Hello</h1>
render() { return <Title>Hello</Title>; }
```

props 값을 통한 조건식

생성된 '스타일 컴포넌트'의 props 값을 통해 조건식을 사용할 수도 있다.

```
// 아래의 스타일을 갖는 <button> element를 생성
const Button = styled.button`
  font-size: 1.5em;
  text-align: center;
  color: ${props => props.primary ? "green" : "yellow" };
`;

render() {
    return <>
            <Title primary>Hello A</Title>  // green
            <Title>Hello B</Title>  // yellow
     </>;
}
```

스타일 확장

기존에 작성된 '스타일 컴포넌트'를 확장해 새로운 컴포넌트를 만들 수 있다.
기존 값은 override 되고, 새로운 값은 추가된다.

```
const Button = styled.button`
  color: red;
  font-size: 1em;
  margin: 1em;
  padding: 0.25em 1em;
  border: 2px solid red;
  border-radius: 3px;
`;

// 기존 <Button>을 확장한 새로운 컴포넌트
const TomatoButton = styled(Button)`
  color: tomato;
  border-color: tomato;
  opacity: 0.5;
`;
```

기존 컴포넌트 스타일링

기존의 일반 컴포넌트들도 '스타일 컴포넌트'로 쉽게 만들 수 있다.

```
// 일반 컴포넌트
const Link = ({ className, children }) => (
  <a className={className}>
    {children}
  </a>
);

// 스타일 컴포넌트를 생성
const StyledLink = styled(Link)`
  color: palevioletred;
  font-weight: bold;
`;
```

고맙습니다.

이 문서는 나눔글꼴로 작성되었습니다. 설치하기