

REACT Docs - 한국어

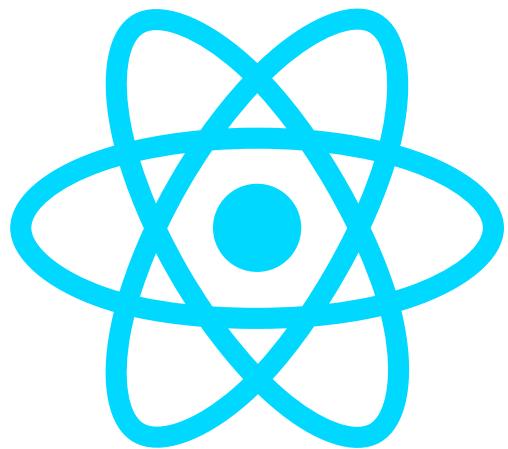


Table of contents

• Accessibility	5
• Add react /to /a /website	16
• Addons animation	21
• Addons create /fragment	28
• Addons perf	30
• Addons pure /render /mixin	34
• Addons shallow /compare	35
• Addons shallow /renderer	36
• Addons test /utils	38
• Addons two /way /binding /helpers	44
• Addons update	47
• Addons	50
• Cdn links	52
• Code splitting	53
• Codebase overview	58
• Components and /props	62
• Composition vs /inheritance	67
• Conditional rendering	70
• Context	75
• Create a /new /react /app	81
• Cross origin /errors	84
• Design principles	86
• Error boundaries	92
• Error decoder	96
• Faq ajax	97
• Faq build	100
• Faq functions	101
• Faq internals	108
• Faq state	109
• Faq structure	112
• Faq styling	114
• Faq versioning	115
• Forms	117
• Forwarding refs	123

• Fragments	125
• Getting started	128
• Handling events	132
• Hello world	135
• Higher order /components	137
• Hooks custom	145
• Hooks effect	150
• Hooks faq	161
• Hooks intro	181
• Hooks overview	185
• Hooks reference	191
• Hooks rules	207
• Hooks state	210
• How to /contribute	216
• Implementation notes	221
• Integrating with /other /libraries	238
• Introducing jsx	247
• Jsx in /depth	251
• Legacy context	259
• Legacy event /pooling	264
• Lifting state /up	265
• Lists and /keys	272
• Optimizing performance	277
• Portals	285
• React without /es6	288
• React without /jsx	292
• Reconciliation	293
• Reference dom /elements	297
• Reference events	301
• Reference glossary	309
• Reference javascript /environment /requirements	313
• Reference profiler	314
• Reference pure /render /mixin	317
• Reference react /component	319
• Reference react /dom /client	333
• Reference react /dom /server	335

• Reference react /dom	340
• Reference react	345
• Reference test /renderer	354
• Refs and /the /dom	360
• Release channels	366
• Render props	369
• Rendering elements	375
• State and /lifecycle	377
• Static type /checking	385
• Strict mode	392
• Testing environments	397
• Testing recipes	399
• Testing	410
• Thinking in /react	411
• Typechecking with /proptypes	416
• Uncontrolled components	421
• Web components	423

접근성

접근성이 필요한 이유

웹 접근성(별칭: **a11y**)은 모두가 사용할 수 있도록 웹사이트를 디자인, 개발하는 것을 의미합니다. 보조과학기술(assistive technology)들이 웹페이지들을 해석할 수 있도록 접근성을 갖추는 것이 필요합니다.

React는 접근성을 갖춘 웹사이트를 만들 수 있도록 모든 지원을 하고 있으며, 대부분은 표준 HTML 기술이 사용됩니다.

표준 및 지침

WCAG

[Web Content Accessibility Guidelines](#)는 접근성을 갖춘 웹사이트를 만드는 데 필요한 지침을 제공합니다.

아래 WCAG 체크리스트를 통해 간략하게 살펴볼 수 있습니다.

- [Wuhcag의 WCAG 체크리스트](#)
- [WebAIM의 WCAG 체크리스트](#)
- [The A11Y Project의 체크리스트](#)

WAI-ARIA

[Web Accessibility Initiative - Accessible Rich Internet Applications](#) 문서에는 접근성을 갖춘 JavaScript 위젯을 만드는데 필요한 기술들이 담겨있습니다.

참고로, JSX에서는 모든 `aria-*` HTML 어트리뷰트를 지원하고 있습니다. React에서 대부분의 DOM 프로퍼티와 어트리뷰트에 대한 값이 캐멀 케이스로 지원되는 반면, `aria-*` 와 같은 어트리뷰트는 일반적인 HTML과 마찬가지로 hyphen-case(혹은 kebab-case, lisp-case 등)로 작성해야 합니다.

```
<input
  type="text"
  aria-label={labelText}
  aria-required="true"
  onChange={onchangeHandler}
  value={inputValue}
  name="name"
/>
```

시맨틱 HTML

시맨틱 HTML은 웹 애플리케이션에 있어 접근성의 기초입니다. 정보의 의미가 강조되는 HTML 엘리먼트를 웹 사이트에서 사용하면 자연스럽게 접근성이 갖추어지곤 합니다.

- [MDN HTML 엘리먼트 참고](#)

가끔 React로 구성한 코드가 돌아가게 만들기 위해 `<div>` 와 같은 엘리먼트를 사용해 HTML의 의미를 깨트리곤 합니다. 특히, 목록 (``, ``, `<dl>`)과 HTML `<table>`을 사용할 때 문제가 두드러집니다. 이 경우에는, [React Fragment](#)를 사용하여 여러 엘리먼트를 하나로 묶어주는 것을 권장합니다.

예시는 아래와 같습니다.

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

function ListItem({ item }) {
  return (
    <Fragment>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </Fragment>
  );
}

function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        <ListItem item={item} key={item.id} />
      ))}
    </dl>
  );
}
```

다른 엘리먼트와 마찬가지로, Fragment는 배열의 각 항목을 매핑할 때에도 사용할 수 있습니다.

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // 항목을 매핑할 때 Fragment는 반드시 `key` 프로퍼티가 있어야 합니다.
        <Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </Fragment>
      ))}
    </dl>
  );
}
```

Fragment 태그에 어떤 props도 필요하지 않고, 사용하고 있는 도구에서 지원한다면, 아래와 같이 [짧게 줄여 쓸 수](#) 있습니다.

```
function ListItem({ item }) {
  return (
    <>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </>
  );
}
```

더 자세한 내용은 [Fragment](#) 문서를 참고해주시기 바랍니다.

접근성 있는 폼

라벨링

`<input>` 과 `<textarea>` 같은 모든 HTML 폼 컨트롤은 구분할 수 있는 라벨이 필요합니다. 스크린 리더를 사용하는 사용자를 위해 자세한 설명이 담긴 라벨을 제공해야 합니다.

다음은 라벨을 제공하는 방법에 관한 자료입니다.

- [W3C에서 제공하는 엘리먼트 라벨링 방법](#)
- [WebAIM에서 제공하는 엘리먼트 라벨링 방법](#)
- [The Paciello Group이 설명한 접근 가능한 이름들](#)

이와 같은 표준 HTML에 대한 예시들이 React에 바로 사용될 수 있으나, `for` 어트리뷰트 만은 JSX에서 `htmlFor`로 사용하는 것에 주의하시기 바랍니다.

```
<label htmlFor="namedInput">Name:</label>
<input id="namedInput" type="text" name="name"/>
```

사용자들에게 오류 안내하기

오류 상황은 모든 사용자가 알 수 있어야 합니다. 아래 링크는 스크린 리더에 오류 문구를 노출하는 방법을 설명합니다.

- [The W3C demonstrates user notifications](#)
- [WebAIM looks at form validation](#)

포커스 컨트롤

모든 웹 애플리케이션은 키보드만 사용하여 모든 동작을 할 수 있어야 합니다.

- [WebAIM이 말하는 키보드 접근성](#)

키보드 포커스와 포커스 윤곽선

키보드 포커스는 키보드 입력을 받아들일 수 있는 DOM 내의 현재 엘리먼트를 나타냅니다. 아래 이미지와 비슷하게 포커스 윤곽선이 표시됩니다.



위와 같은 윤곽선을 다른 포커스 윤곽선으로 교체할 때만 `outline: 0`과 같은 윤곽선을 제거하는 CSS를 사용합니다.

원하는 콘텐츠로 건너뛸 수 있는 방법

애플리케이션은 사용자들의 키보드 탐색을 돋고 탐색 속도를 높일 수 있도록, 이전에 탐색한 영역을 건너뛸 방법을 제공해야 합니다.

Skiplinks 또는 Skip Navigation Link들은 키보드 사용자가 페이지와 상호작용할 때만 표시되는 숨겨진 탐색 링크입니다. 내부의 페이지 앵커와 약간의 스타일링으로 매우 쉽게 구현할 수 있습니다.

- [WebAIM - Skip Navigation Links](#)

또한, 보조과학기술에 의해 사용자들이 이러한 섹션으로 빠르게 이동할 수 있도록, `<main>` 과 `<aside>` 같이 대표성을 띠는 랜드마크 엘리먼트와 역할들을 사용해 페이지 영역을 나누어야 합니다.

이러한 엘리먼트들을 사용해 접근성을 높이는 방법은 아래의 글을 참고해주시기 바랍니다.

- [접근 가능한 랜드마크\(Accessible Landmarks\)](#)

프로그래밍적으로 포커스 관리하기

React 애플리케이션들은 런타임 동안 지속해서 HTML DOM을 변경하기 때문에, 가끔 키보드 포커스를 잃거나 예상치 못한 엘리먼트에 포커스를 맞추곤 합니다. 이를 수정하기 위해, 프로그래밍적으로 키보드 포커스를 옮바른 방향으로 변경해주어야 합니다. 예를 들어, 모달이 닫힌 후에는 모달을 열었던 버튼으로 키보드 포커스를 다시 맞춰주어야 합니다.

MDN Web Docs에서 [키보드로 탐색이 가능한 JavaScript 위젯](#)을 만드는 방법에 관해 설명한 글이 있습니다.

React에서 포커스를 지정하려면, [DOM 엘리먼트에 ref를 사용](#)할 수 있습니다.

이를 사용해 JSX 컴포넌트 클래스 안에서 엘리먼트에 대한 ref를 먼저 생성합니다.

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // DOM 엘리먼트를 저장할 TextInput이라는 ref를 생성합니다.
    this.TextInput = React.createRef();
  }
  render() {
    // `ref` 콜백으로 텍스트 input DOM을 저장합니다.
    // 인스턴스 필드의 엘리먼트 (예를 들어, this.TextInput)
    return (
      <input
        type="text"
        ref={this.TextInput}
      />
    );
  }
}
```

그 후에, 컴포넌트 내에서 필요할 때마다 포커스를 지정할 수 있습니다.

```
focus() {
  // DOM API를 사용해 텍스트 input에 정확히 포커스를 맞춥니다.
  // 주의: '현재'의 DOM 노드에 접근하고 있습니다.
  this.TextInput.current.focus();
}
```

가끔씩 부모 컴포넌트가 자식 컴포넌트 내의 엘리먼트에 포커스를 잡아야 할 때가 있습니다. 이때는 자식 컴포넌트에 특별한 프로퍼티를 주어 [DOM ref를 부모 컴포넌트로 노출](#)하는 방식으로 부모의 ref를 자식의 DOM 노드에 넘겨줄 수 있습니다.

```
function CustomTextInput(props) {
  return (
    <div>
      <input ref={props.inputRef} />
    </div>
  );
}
```

```

}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.inputElement = React.createRef();
  }
  render() {
    return (
      <CustomTextInput inputRef={this.inputElement} />
    );
  }
}

// 이제 필요할 때마다 포커스를 잡을 수 있습니다.
this.inputElement.current.focus();

```

고차 컴포넌트(Higher Order Component)를 사용하여 컴포넌트를 확장할 때는 감싸진 컴포넌트에 React에서 제공하는 `forwardRef` 함수를 사용하여 `ref`를 넘겨줄 수 있습니다. 서드 파티 고차 컴포넌트에서 `ref`를 넘겨줄 수 없다면, 위와 같은 패턴을 여전히 차선책으로 사용할 수 있습니다.

매우 좋은 포커스 관리 예시로 [react-aria-modal](#)을 들 수 있습니다. 완전히 접근 가능한 모달 창에 대한 드문 예시입니다. 첫 포커스를 취소 버튼에 맞출 뿐 만 아니라(키보드 사용자가 실수로 확인 동작을 일으키지 않도록 막아줌), 키보드 포커스를 모달 안으로 한정해주며, 모달이 닫힐 때 모달을 열게 했던 엘리먼트에 포커스를 잡아줍니다.

주의

키보드 포커스는 매우 중요한 접근성 기능이지만, 동시에 매우 조심해서 사용해야 하는 기능이기도 합니다. 사용자가 애플리케이션을 어떻게 사용하길 원하는지 예측하지 말고 키보드 포커스 흐름이 흐트러졌을 때 이를 고치려는 방법으로 사용하기 바랍니다.

마우스와 포인터 이벤트

마우스 혹은 포인터 이벤트로 노출된 모든 기능을 키보드만으로 사용할 수 있도록 보장해야 합니다. 포인터 장치만 고려할 경우, 키보드 사용자들이 애플리케이션을 사용하지 못하는 경우가 많습니다.

아래는 클릭 이벤트로 인해 접근성이 떨어지게 되는 예시입니다. 열린 팝오버의 바깥을 클릭해 팝오버를 닫을 수 있는 '외부 클릭 패턴(outside click pattern)'입니다.

Select an option

Load the option

Remove the option

일반적으로 팝오버를 닫는 `click` 이벤트를 `window` 객체에 붙여 구현합니다.

```
class OuterClickExample extends React.Component {
  constructor(props) {
    super(props);

    this.state = { isOpen: false };
    this.toggleContainer = React.createRef();

    this.onClickHandler = this.onClickHandler.bind(this);
    this.onClickOutsideHandler = this.onClickOutsideHandler.bind(this);
  }

  componentDidMount() {
    window.addEventListener('click', this.onClickOutsideHandler);
  }

  componentWillUnmount() {
    window.removeEventListener('click', this.onClickOutsideHandler);
  }

  onClickHandler() {
    this.setState(currentState => ({
      isOpen: !currentState.isOpen
    }));
  }

  onClickOutsideHandler(event) {
    if (this.state.isOpen && !this.toggleContainer.current.contains(event.target))
    {
      this.setState({ isOpen: false });
    }
  }

  render() {
    return (
      <div ref={this.toggleContainer}>
        <button onClick={this.onClickHandler}>Select an option</button>
        {this.state.isOpen && (
          <ul>
            <li>Option 1</li>
            <li>Option 2</li>
            <li>Option 3</li>
          </ul>
        )}
      </div>
    );
  }
}
```

```

    );
}
}

```

이는 포인터 장치 사용자들에게는 괜찮으나, 키보드 사용자들에게는 기능적으로 문제가 생깁니다. 다음 엘리먼트로 탭을 이동할 때 `window` 객체가 `click` 이벤트를 받을 수 없기 때문입니다. 이로 인해, 기능이 가려져 사용자들이 애플리케이션을 제대로 사용할 수 없게 됩니다.

Select an option

Load the option

Remove the option

`onBlur` 와 `onFocus` 같은 적절한 이벤트 핸들러를 사용하여 같은 기능을 제공할 수 있습니다.

```

class BlurExample extends React.Component {
  constructor(props) {
    super(props);

    this.state = { isOpen: false };
    this.timeOutId = null;

    this.onClickHandler = this.onClickHandler.bind(this);
    this.onBlurHandler = this.onBlurHandler.bind(this);
    this.onFocusHandler = this.onFocusHandler.bind(this);
  }

  onClickHandler() {
    this.setState(currentState => ({
      isOpen: !currentState.isOpen
    }));
  }

  // setTimeout을 사용해 다음 순간에 팝오버를 닫습니다.
  // 엘리먼트의 다른 자식에 포커스가 맞춰져있는지 확인하기 위해 필요합니다.
  // 새로운 포커스 이벤트가 발생하기 전에
  // 블러(blur) 이벤트가 발생해야 하기 때문입니다.
  onBlurHandler() {
    this.timeOutId = setTimeout(() => {
      this.setState({
        isOpen: false
      });
    });
  }

  // 자식이 포커스를 받으면, 팝오버를 닫지 않습니다.
  onFocusHandler() {
    clearTimeout(this.timeOutId);
  }
}

```

```

render() {
  // React는 블러와 포커스 이벤트를 부모에 버블링해줍니다.
  return (
    <div onBlur={this.onBlurHandler}
      onFocus={this.onFocusHandler}>
      <button onClick={this.onClickHandler}
        aria-haspopup="true"
        aria-expanded={this.state.isOpen}>
        Select an option
      </button>
      {this.state.isOpen && (
        <ul>
          <li>Option 1</li>
          <li>Option 2</li>
          <li>Option 3</li>
        </ul>
      )}
    </div>
  );
}

```

이 코드는 포인터 장치 사용자와 키보드 사용자 모두에게 기능을 제공합니다. 동시에 스크린 리더 사용자들을 지원하기 위해 `aria-* props`를 추가했습니다. 단순함을 위해 `방향키`로 조작하는 기능은 구현하지 않았습니다.

Select an option

Load the option

Remove the option

이것은 포인터와 마우스 이벤트에만 의존해 키보드 사용자의 사용성을 해치는 많은 예시 중 하나입니다. 항상 키보드로 테스트하면 바로 문제가 되는 영역을 확인하고, 키보드 핸들러를 추가하여 수정할 수 있습니다.

더욱 복잡한 위젯

복잡한 사용자 경험으로 접근성이 떨어져서는 안 됩니다. 접근성을 쉽게 지원하는 방법은 가능한 한 HTML에 맞게 코딩하는 것이며, 복잡한 위젯 역시 접근성있게 코딩할 수 있습니다.

여기서는 [ARIA 역할과 ARIA 상태 및 프로퍼티](#)에 대한 지식이 필요합니다. 이들은 JSX에서 모두 지원되는 HTML 어트리뷰트로 채워진 도구 상자로, 이를 통해 완전히 접근성 있고 기능이 우수한 React 컴포넌트를 구성할 수 있습니다.

각각의 위젯 타입은 명확한 디자인 패턴이 있으며, 사용자와 사용자 애이전트 모두 특정 방향으로 기능하는 것이 요구됩니다.

- [WAI-ARIA Authoring Practices - 디자인 패턴과 위젯](#)
- [Heydon Pickering - ARIA 예시](#)
- [포괄적 컴포넌트](#)

기타 고려사항

언어 설정

스크린 리더 소프트웨어들이 올바른 음성을 선택할 수 있도록, 페이지 텍스트에 인간 언어(human language)를 나타내야 합니다.

- [WebAIM - 문서 언어](#)

문서 제목 설정

문서의 `<title>` 이 현재 페이지에 대한 올바른 설명을 담아야 합니다. 이를 통해 사용자들이 현재 페이지의 맥락을 놓치지 않도록 할 수 있습니다.

- [WCAG - 문서 제목 요건 이해하기](#)

React에서는 [React Document Title 컴포넌트](#)를 사용해 설정할 수 있습니다.

색 대비

읽을 수 있는 모든 글에 충분한 색 대비를 주어, 저시력 사용자들이 최대한 읽을 수 있도록 해야 합니다.

- [WCAG - 색 대비 요건 이해하기](#)
- [색 대비에 대한 모든 것과 이를 다시 생각해야 하는 이유](#)
- [The A11Y Project - 색 채도란](#)

웹사이트의 모든 항목에 대해 적절한 색 조합을 일일이 계산하는 것은 지루할 수 있습니다. [Colorable](#)을 사용해 접근 가능한 모든 색 조합 표를 계산할 수 있습니다.

아래에 언급된 aXe와 WAVE 도구 모두 색 대비에 대한 테스트가 포함되어 있어, 색 대비에 대한 오류를 알려줍니다.

색 대비에 대한 테스트 기능을 확장할 경우, 아래 도구를 사용할 수 있습니다.

- [WebAIM - 색 채도 검사기](#)
- [The Paciello Group - 색 채도 분석기](#)

개발 및 테스트 도구

접근 가능한 웹 애플리케이션을 만들 수 있도록 도와주는 여러 도구가 있습니다.

키보드

가장 쉬우면서도 가장 중요한 검사 중 하나는 웹사이트 전체가 키보드만으로도 사용될 수 있는지 테스트하는 것입니다. 방법은 아래와 같습니다.

1. 마우스의 연결을 해제하세요.
2. `Tab` 과 `Shift+Tab` 을 사용해 이동하세요.
3. `Enter` 를 사용해 엘리먼트를 활성화하세요.
4. 메뉴와 드롭다운과 같은 일부 엘리먼트는 필요하다면 키보드 방향키를 사용해 조작합니다.

개발 보조 도구

일부 접근성 기능들은 JSX 코드에서 바로 확인할 수 있습니다. 종종 ARIA 역할, 상태 및 프로퍼티에 대한 인텔리센스(intellisense) 검사 기능이 JSX를 인식하는 IDE에 미리 제공되는 경우가 있습니다. 아래와 같은 도구 역시 사용할 수 있습니다.

eslint-plugin-jsx-a11y

ESLint 플러그인인 [eslint-plugin-jsx-a11y](#)는 JSX 내의 접근성 문제에 대해 즉각적인 AST 릴팅 피드백을 제공합니다. 많은 IDE가 코드 분석과 소스 코드 창에 이런 결과를 통합할 수 있도록 해줍니다.

[Create React App](#)에서는 해당 플러그인의 일부 규칙들이 활성화되어 있습니다. 더 많은 접근성 기능을 활성화하려면, 프로젝트 최상위에 아래와 같이 `.eslintrc` 파일을 생성합니다.

```
{
  "extends": ["react-app", "plugin:jsx-a11y/recommended"],
  "plugins": ["jsx-a11y"]
}
```

브라우저에서 접근성 테스트하기

브라우저에서 접근성 검수를 받을 수 있는 여러 도구가 있습니다. 여기서 소개하는 것들은 HTML의 기술적인 접근성만을 테스트하기 때문에, 다른 도구들과 함께 사용하는 것을 권장합니다.

aXe와 aXe-core, react-axe

Deque Systems에서는 자동으로 애플리케이션의 종단 간(end-to-end) 접근성을 테스트하는 [aXe-core](#)를 제공합니다. 이 모듈은 Selenium과의 연동이 포함되어있습니다.

[The Accessibility Engine](#) 또는 aXe는 aXe-core 기반의, 접근성 검사를 위한 브라우저 확장기능입니다.

또는, [@axe-core/react](#) 모듈을 사용해 개발 혹은 디버깅 중에 이러한 접근성 문제를 콘솔에 바로 띄울 수 있습니다.

WebAIM WAVE

[Web Accessibility Evaluation Tool](#)은 또 다른 브라우저 확장 기능입니다.

접근성 검사기와 접근성 트리

접근성 트리는 스크린 리더와 같은 보조과학기술에 노출되어야 하는 DOM 엘리먼트에 접근 가능한 객체가 담긴 DOM 트리의 하위 집합입니다.

일부 브라우저에서는 접근성 트리 안의 각 엘리먼트의 접근성 정보를 손쉽게 확인할 수 있습니다.

- [Firefox에서 접근성 검사기를 사용하는 방법](#)
- [Chrome에서 접근성 검사기를 사용하는 방법](#)
- [OS X Safari에서 접근성 검사기를 사용하는 방법](#)

스크린 리더

접근성 테스트의 일환으로 스크린 리더를 사용한 테스트 역시 진행되어야 합니다.

브라우저와 스크린 리더 조합에 주의해주시기 바랍니다. 선택한 스크린 리더에 가장 적합한 브라우저에서 애플리케이션을 테스트하기 바랍니다.

일반적으로 사용되는 스크린 리더

Firefox의 NVDA

[NonVisual Desktop Access](#)(별칭: NVDA)는 널리 사용되는 오픈소스 윈도우 스크린 리더입니다.

NVDA를 효과적으로 사용하는 방법은 아래를 참조해주시기 바랍니다.

- [WebAIM - NVDA를 사용한 웹 접근성 측정](#)
- [Deque - NVDA 키보드 단축키](#)

Safari의 VoiceOver

VoiceOver는 애플 기기에 통합된 스크린 리더입니다.

VoiceOver를 활성화 및 사용하는 방법은 아래를 참조해주시기 바랍니다.

- [WebAIM - VoiceOver를 사용한 웹 접근성 측정](#)
- [Deque - OS X용 VoiceOver 키보드 단축키](#)
- [Deque - iOS용 VoiceOver 단축키](#)

Internet Explorer의 JAWS

[Job Access With Speech](#) 또는 JAWS는 윈도우에서 주로 쓰이는 스크린 리더입니다.

JAWS를 효과적으로 사용하는 방법은 아래를 참조해주시기 바랍니다.

- [WebAIM - JAWS를 사용한 웹 접근성 측정](#)
- [Deque - JAWS 키보드 단축키](#)

기타 스크린 리더

Google Chrome의 ChromeVox

ChromeVox는 Chromebook에 통합된 스크린 리더이며 Google Chrome의 [확장기능](#)으로 사용할 수 있습니다.

ChromeVox를 효과적으로 사용하는 방법은 아래를 참조해주시기 바랍니다.

- [Google Chromebook 도움말 - 내장 스크린 리더 사용법](#)
- [ChromeVox Classic 키보드 단축키](#)

웹사이트에 React 추가

React는 필요한 만큼만 사용하면 됩니다. 적게든 많게든 말이죠.

React는 처음부터 점진적으로 도입할 수 있게 설계되었습니다. 그래서 **React는 필요한 만큼만 사용하면 됩니다.** 아마 기존 페이지에 "상호작용"이라는 조미료를 첨가하기만 하고 싶을 수도 있습니다. 이럴 때 React 컴포넌트를 쓰면 좋습니다.

오늘날 웹사이트의 대부분은 싱글 페이지가 아니고 그래야 할 필요도 없습니다. **빌드 도구 없이 몇 줄의 코드만으로** 웹사이트의 작은 부분에 먼저 React를 도입해보세요. React의 비중을 천천히 늘릴 수도 있고 아니면 동적인 위젯을 몇 개 추가하는 것으로 끌낼 수도 있습니다.

- [React 1분 내로 추가하기](#)
- [선택사항: JSX로 React 해보기](#) (번들러가 필요없습니다!)

1분 내로 React 추가하기

이 글에서는 기존 페이지에 React를 추가하는 법에 대해서 다룹니다. 기존의 웹사이트에서 시도 해보셔도 좋고, 연습삼아 새로운 HTML 파일에서 시도해보셔도 좋습니다.

복잡한 도구를 쓰거나 뭔가를 따로 설치해야 할 필요도 없습니다. **약간의 시간과 함께 인터넷만 연결 되어 있다면** 이 문서에서 다루는 것을 모두 해볼 수 있습니다.

선택사항: [예시의 전체 소스코드 다운로드 \(2KB로 압축됨\)](#)

1단계: HTML 파일에 DOM 컨테이너 설치

먼저, 편집할 HTML 파일을 엽니다. 그 다음 비어있는 `<div>` 태그를 추가해줍니다. 이 태그가 바로 React를 통해 원하는 내용을 표시할 수 있는 위치가 됩니다. 다음과 같이 말이죠.

```
<!-- ... existing HTML ... -->
<div id="like_button_container"></div>
<!-- ... existing HTML ... -->
```

이 `<div>` 태그에 유일한 HTML 속성인 `id` 를 부여해줍니다. 이를 통해 JavaScript가 태그를 찾고 찾은 태그 안에 React 컴포넌트를 표시할 수 있게됩니다.

팁

컨테이너 `<div>` 태그는 이처럼 `<body>` 태그 안 쪽 어디서든 추가할 수 있습니다. 한 페이지에서도 독립적인 DOM 컨테이너를 원하는 만큼 추가할 수 있습니다. 보통 이 태그는 비어있습니다. React가 DOM 컨테이너 안에 내용을 추가해줍니다.

2단계: 스크립트 태그 추가하기

그 다음, `<script>` 태그 3개를 닫는 태그인 `</body>` 앞에 추가해줍니다.

```
<!-- ... 다른 HTML ... -->
<!-- React를 실행. -->
<!-- 주의: 사이트를 배포할 때는 "development.js"를 "production.min.js"로 대체하세요. -->
<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>

<!-- 만든 React 컴포넌트를 실행. -->
<script src="like_button.js"></script>

</body>
```

처음 두 태그는 React를 실행시키고 3번 째 코드는 만든 컴포넌트를 실행시킵니다.

3단계: React 컴포넌트 만들기

`like_button.js`라는 이름으로 HTML 페이지 옆에 새 파일을 만듭니다.

이 [스타터 코드](#)를 열고 코드를 방금 만든 파일에 복사해줍니다.

팁

이 코드는 `LikeButton`이라는 React 컴포넌트를 정의해줍니다. 지금 당장 이해가 안 되어도 걱정 마세요. React에 대한 개념을 쌓아 나가는 것은 나중에 [자습서](#)와 [주요 개념 가이드](#)에서 다룰 겁니다. 그러니 지금 당장은, 컴포넌트를 화면에 띄우는 데 집중해봅시다!

`like_button.js`의 맨 뒷 줄, 그러니까 아까 붙여넣은 [스타터 코드](#) 뒤에 다음 코드 세 줄을 추가해줍니다.

```
// ... 복사했던 스타터 코드 ...

const domContainer = document.querySelector('#like_button_container');
const root = ReactDOM.createRoot(domContainer);
root.render();
```

이 세 줄의 코드는 첫 단계에서 HTML 페이지에 추가했던 `<div>` 태그를 찾아주고, 해당 태그에 React 앱을 만들어주고, 그 안에 "좋아요" 버튼 React 컴포넌트를 추가해줍니다.

다 끝났습니다!

다음 단계는 없습니다. 방금 웹사이트에 처음으로 React 컴포넌트를 추가했습니다.

다음 차례들에 React를 기존 프로젝트에 결합하는데 도움이 될만한 정보들이 더 있습니다.

[예시 전체 소스 코드 보기](#)

[예시 전체 다운로드 \(2KB로 압축됨\)](#)

팁: 컴포넌트 재사용

보통은 하나의 웹페이지에 여러 React 컴포넌트를 설치하게 됩니다. 다음 예시는 "좋아요" 버튼 3개를 만들고 그 컴포넌트들에 데이터를 넘겨주는 코드입니다.

[예시 전체 소스 코드 보기](#)

[예시 전체 다운로드 \(2KB로 압축됨\)](#)

주의

이 방법은 대개 페이지에서 React로 만들어진 부분들이 서로 격리 되어있을 때 유용합니다. React 코드 내에서는 [컴포넌트 합성](#)을 사용하는 편이 더 쉽습니다.

팁: 프로덕션을 위한 JavaScript의 압축

프로덕션을 위해 웹사이트를 배포하기 전에 JavaScript 파일을 압축하지 않는다면 웹사이트를 사용할 때 눈에 띄는 성능 저하가 일어날 겁니다.

애플리케이션 스크립트를 이미 압축했을 경우 배포된 HTML 파일이 `production.min.js`로 끝나는 React 파일을 확실히 실행하기만 하면 사이트는 프로덕션 준비가 완료된 겁니다.

```
<script src="https://unpkg.com/react@18/umd/react.production.min.js" crossorigin>
</script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.production.min.js"
crossorigin></script>
```

스크립트를 압축하는 절차가 따로 없다면 [이 사이트를 참고해서 설정해보세요](#).

선택사항: JSX로 React 해보기

지금까지 다뤘던 예시들은 브라우저가 기본적으로 지원하는 요소들만을 사용했습니다. 때문에 React가 어떤 것을 표시할지 결정 해주는 JavaScript의 함수들을 사용했습니다.

```
const e = React.createElement;

// "좋아요" <button>을 표시
return e(
  'button',
  { onClick: () => this.setState({ liked: true }) },
  'Like'
);
```

하지만, React에서는 [JSX](#)라는 또 다른 선택지가 있습니다.

```
// "좋아요" <button>을 표시
return (
  <button onClick={() => this.setState({ liked: true })}>
    Like
  </button>
);
```

위 두 코드들은 같은 기능을 수행합니다. **JSX는 필수가 아닌 선택사항**이지만 사람들은 UI 코드를 짤 때 JSX를 쓰는 것이 더 편리하다고 생각합니다. React와 다른 라이브러리들에서도 말이죠.

이 [온라인 변화기](#)를 통해서 JSX로 여러 가지 작업을 해볼 수 있습니다.

JSX 빠르게 시도해보기

기존 프로젝트에서 JSX 태그를 써보는 제일 빠른 방법은 이 `<script>` 태그를 집어넣는 겁니다.

```
<script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
```

이제 어떤 `<script>` 태그에서든 `type="text/babel"` 성질을 추가하면 JSX를 사용할 수 있습니다. 이 [JSX를 사용한 예시 HTML 파일](#)로 여러 가지 작업들을 해보세요.

이런 식의 접근 방법은 공부 목적이나 간단한 데모 사이트를 만들기 위함이라면 괜찮습니다. 그러나, 이 방법은 사이트를 느리게 만들고 프로덕션에서는 맞지 않습니다. 한 단계 앞으로 나아갈 준비가 되었다면 새로 추가한 `<script>` 태그와 `type="text/babel"` 어트리뷰트를 제거해보세요. 다음 차례에서는 `<script>` 태그를 자동으로 변환시켜줄 JSX 전처리기를 만들 겁니다.

프로젝트에 JSX 추가하기

JSX를 프로젝트에 추가하는 데에는 복잡한 번들러나 개발 서버가 필요하지 않습니다. 핵심은, JSX를 추가하는 건 **마치 CSS 전처리기를 추가하는 것과 같다**는 겁니다. 오직 필요한 건 컴퓨터에 [Node.js](#)를 설치하는 겁니다.

터미널에서 프로젝트 파일에 접근하고 다음 두 명령어를 붙여넣기 하세요.

1. **1단계:** `npm init -y` 를 실행하세요. (실패한다면 [여기서 해결할 수 있습니다.](#))
2. **2단계:** `npm install babel-cli@6 babel-preset-react-app@3` 를 실행하세요.

팁

JSX 전처리기를 설치할 때 npm은 여기서만 쓰면 됩니다. 다른 단계에서는 npm이 쓰일 일이 없습니다. React와 애플리케이션 코드는 둘다 똑같이 `<script>` 태그로 유지할 수 있습니다.

축하합니다! 프로젝트에 프로덕션 준비가 된 **JSX 설정**을 끝마쳤습니다.

JSX 전처리기 실행하기

`src` 폴더를 만들고 다음 터미널 명령어를 실행하세요.

```
npx babel --watch src --out-dir . --presets react-app/prod
```

주의

`npx` 는 오타가 아닙니다. -- [npm 5.2버전 이상에 내장된 패키지 실행 도구](#)입니다.

"babel 패키지 설치가 잘못 수행되었습니다."라는 메시지를 보았다면, [그 전 단계들](#)에서 몇 가지 놓쳤을 수도 있습니다.
동일한 폴더에서 이를 실행하고 다시 시도해보세요.

끝날 때 까지 기다릴 필요가 없습니다. 이 명령어는 자동화 된 JSX 감시기를 실행합니다.

JSX 스타터 코드를 통해 `src/like_button.js`라는 파일을 만들어주면, 감시기가 전처리 되어 브라우저와 호환되는 순수 JavaScript로 구성된 `like_button.js`를 생성합니다. JSX를 포함한 소스 파일을 편집하면 이 과정이 자동으로 다시 실행됩니다.

덤으로 이 감시기는 구형 브라우저와의 호환성 문제를 걱정할 필요 없이 클래스와 같은 모던 JavaScript 문법을 쓸 수 있게 해줍니다. 아까 사용했던 도구는 Babel이라고 부릅니다. Babel에 대한 자세한 정보는 [공식 문서](#)에서 볼 수 있습니다!

이런 빌드 도구들에 익숙해지고 더 많은 것을 해보고 싶어진다고 느끼기 시작한다면, [다음 문서](#)에서 가장 널리 쓰이고 접근성이 좋은 툴체인들을 소개하고 있으니 확인해보세요. 그렇지 않아도 괜찮다면, 이 스크립트 태그만으로도 충분합니다!

[Go to TOC](#)

Animation Add-Ons

Note:

`ReactTransitionGroup` and `ReactCSSTransitionGroup` have been moved to the `react-transition-group` package that is maintained by the community. Its 1.x branch is completely API-compatible with the existing addons. Please file bugs and feature requests in the [new repository](#).

The `ReactTransitionGroup` add-on component is a low-level API for animation, and `ReactCSSTransitionGroup` is an add-on component for easily implementing basic CSS animations and transitions.

High-level API: ReactCSSTransitionGroup

`ReactCSSTransitionGroup` is a high-level API based on `ReactTransitionGroup` and is an easy way to perform CSS transitions and animations when a React component enters or leaves the DOM. It's inspired by the excellent [ng-animate](#) library.

Importing

```
import ReactCSSTransitionGroup from 'react-transition-group'; // ES6
var ReactCSSTransitionGroup = require('react-transition-group'); // ES5 with npm
```

```
class TodoList extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: ['hello', 'world', 'click', 'me']};
    this.handleAdd = this.handleAdd.bind(this);
  }

  handleAdd() {
    const newItems = this.state.items.concat([
      prompt('Enter some text')
    ]);
    this.setState({items: newItems});
  }

  handleRemove(i) {
    let newItems = this.state.items.slice();
    newItems.splice(i, 1);
    this.setState({items: newItems});
  }

  render() {
    const items = this.state.items.map((item, i) => (
      <div key={i} onClick={() => this.handleRemove(i)}>
        {item}
      </div>
    ));
    return (

```

```

        <div>
          <button onClick={this.handleAdd}>Add Item</button>
          <ReactCSSTransitionGroup
            transitionName="example"
            transitionEnterTimeout={500}
            transitionLeaveTimeout={300}>
            {items}
          </ReactCSSTransitionGroup>
        </div>
      );
    }
  }
}

```

Note:

You must provide the `key` attribute for all children of `ReactCSSTransitionGroup`, even when only rendering a single item. This is how React will determine which children have entered, left, or stayed.

In this component, when a new item is added to `ReactCSSTransitionGroup` it will get the `example-enter` CSS class and the `example-enter-active` CSS class added in the next tick. This is a convention based on the `transitionName` prop.

You can use these classes to trigger a CSS animation or transition. For example, try adding this CSS and adding a new list item:

```

.example-enter {
  opacity: 0.01;
}

.example-enter.example-enter-active {
  opacity: 1;
  transition: opacity 500ms ease-in;
}

.example-leave {
  opacity: 1;
}

.example-leave.example-leave-active {
  opacity: 0.01;
  transition: opacity 300ms ease-in;
}

```

You'll notice that animation durations need to be specified in both the CSS and the render method; this tells React when to remove the animation classes from the element and -- if it's leaving -- when to remove the element from the DOM.

Animate Initial Mounting

`ReactCSSTransitionGroup` provides the optional prop `transitionAppear`, to add an extra transition phase at the initial mount of the component. There is generally no transition phase at the initial mount as the default value of `transitionAppear` is `false`. The following is an example which passes the prop `transitionAppear` with the value `true`.

```

render() {
  return (
    <ReactCSSTransitionGroup
      transitionName="example"
      transitionAppear={true}
      transitionAppearTimeout={500}
      transitionEnter={false}
      transitionLeave={false}>
      <h1>Fading at Initial Mount</h1>
    </ReactCSSTransitionGroup>
  );
}

```

During the initial mount `ReactCSSTransitionGroup` will get the `example-appear` CSS class and the `example-appear-active` CSS class added in the next tick.

```

.example-appear {
  opacity: 0.01;
}

.example-appear.example-appear-active {
  opacity: 1;
  transition: opacity .5s ease-in;
}

```

At the initial mount, all children of the `ReactCSSTransitionGroup` will `appear` but not `enter`. However, all children later added to an existing `ReactCSSTransitionGroup` will `enter` but not `appear`.

Note:

The prop `transitionAppear` was added to `ReactCSSTransitionGroup` in version 0.13. To maintain backwards compatibility, the default value is set to `false`.

However, the default values of `transitionEnter` and `transitionLeave` are `true` so you must specify `transitionEnterTimeout` and `transitionLeaveTimeout` by default. If you don't need either enter or leave animations, pass `transitionEnter={false}` or `transitionLeave={false}`.

Custom Classes

It is also possible to use custom class names for each of the steps in your transitions. Instead of passing a string into `transitionName` you can pass an object containing either the `enter` and `leave` class names, or an object containing the `enter`, `enter-active`, `leave-active`, and `leave` class names. If only the `enter` and `leave` classes are provided, the `enter-active` and `leave-active` classes will be determined by appending '`-active`' to the end of the class name. Here are two examples using custom classes:

```

// ...
<ReactCSSTransitionGroup
  transitionName={{ 
    enter: 'enter',
    enterActive: 'enterActive',
    leave: 'leave',
    leaveActive: 'leaveActive',
  }}

```

```

        appear: 'appear',
        appearActive: 'appearActive'
    } }>
{item}
</ReactCSSTransitionGroup>

<ReactCSSTransitionGroup
    transitionName={ {
        enter: 'enter',
        leave: 'leave',
        appear: 'appear'
    } }>
{item2}
</ReactCSSTransitionGroup>
// ...

```

Animation Group Must Be Mounted To Work

In order for it to apply transitions to its children, the `ReactCSSTransitionGroup` must already be mounted in the DOM or the prop `transitionAppear` must be set to `true`.

The example below would **not** work, because the `ReactCSSTransitionGroup` is being mounted along with the new item, instead of the new item being mounted within it. Compare this to the [Getting Started](#) section above to see the difference.

```

render() {
  const items = this.state.items.map((item, i) =>
    <div key={item} onClick={() => this.handleRemove(i)}>
      <ReactCSSTransitionGroup transitionName="example">
        {item}
      </ReactCSSTransitionGroup>
    </div>
  );
}

return (
  <div>
    <button onClick={this.handleAdd}>Add Item</button>
    {items}
  </div>
);
}

```

Animating One or Zero Items

In the example above, we rendered a list of items into `ReactCSSTransitionGroup`. However, the children of `ReactCSSTransitionGroup` can also be one or zero items. This makes it possible to animate a single element entering or leaving. Similarly, you can animate a new element replacing the current element. For example, we can implement a simple image carousel like this:

```

import ReactCSSTransitionGroup from 'react-transition-group';

function ImageCarousel(props) {
  return (
    <div>
      <ReactCSSTransitionGroup
        transitionName="carousel"
        transitionEnterTimeout={300}

```

```

        transitionLeaveTimeout={300}>
          <img src={props.imageSrc} key={props.imageSrc} />
        </ReactCSSTransitionGroup>
      </div>
    );
}

```

Disabling Animations

You can disable animating `enter` or `leave` animations if you want. For example, sometimes you may want an `enter` animation and no `leave` animation, but `ReactCSSTransitionGroup` waits for an animation to complete before removing your DOM node. You can add `transitionEnter={false}` or `transitionLeave={false}` props to `ReactCSSTransitionGroup` to disable these animations.

Note:

When using `ReactCSSTransitionGroup`, there's no way for your components to be notified when a transition has ended or to perform any more complex logic around animation. If you want more fine-grained control, you can use the lower-level `ReactTransitionGroup` API which provides the hooks you need to do custom transitions.

Low-level API: `ReactTransitionGroup`

Importing

```

import ReactTransitionGroup from 'react-addons-transition-group' // ES6
var ReactTransitionGroup = require('react-addons-transition-group') // ES5 with
npm

```

`ReactTransitionGroup` is the basis for animations. When children are declaratively added or removed from it (as in the [example above](#)), special lifecycle methods are called on them.

- `componentWillAppear()`
- `componentDidAppear()`
- `componentWillEnter()`
- `componentDidEnter()`
- `componentWillLeave()`
- `componentDidLeave()`

Rendering a Different Component

`ReactTransitionGroup` renders as a `span` by default. You can change this behavior by providing a `component` prop. For example, here's how you would render a ``:

```

<ReactTransitionGroup component="ul">
  {/* ... */}
</ReactTransitionGroup>

```

Any additional, user-defined, properties will become properties of the rendered component. For example, here's how you would render a `` with CSS class:

```
<ReactTransitionGroup component="ul" className="animated-list">
  {/* ... */}
</ReactTransitionGroup>
```

Every DOM component that React can render is available for use. However, `component` does not need to be a DOM component. It can be any React component you want; even ones you've written yourself! Just write `component={List}` and your component will receive `this.props.children`.

Rendering a Single Child

People often use `ReactTransitionGroup` to animate mounting and unmounting of a single child such as a collapsible panel. Normally `ReactTransitionGroup` wraps all its children in a `span` (or a custom `component` as described above). This is because any React component has to return a single root element, and `ReactTransitionGroup` is no exception to this rule.

However if you only need to render a single child inside `ReactTransitionGroup`, you can completely avoid wrapping it in a `` or any other DOM component. To do this, create a custom component that renders the first child passed to it directly:

```
function FirstChild(props) {
  const childrenArray = React.Children.toArray(props.children);
  return childrenArray[0] || null;
}
```

Now you can specify `FirstChild` as the `component` prop in `<ReactTransitionGroup>` props and avoid any wrappers in the result DOM:

```
<ReactTransitionGroup component={FirstChild}>
  {someCondition ? <MyComponent /> : null}
</ReactTransitionGroup>
```

This only works when you are animating a single child in and out, such as a collapsible panel. This approach wouldn't work when animating multiple children or replacing the single child with another child, such as an image carousel. For an image carousel, while the current image is animating out, another image will animate in, so `<ReactTransitionGroup>` needs to give them a common DOM parent. You can't avoid the wrapper for multiple children, but you can customize the wrapper with the `component` prop as described above.

Reference

`componentWillAppear()` **#componentwillappear**

```
componentWillAppear(callback)
```

This is called at the same time as `componentDidMount()` for components that are initially mounted in a `TransitionGroup`. It will block other animations from occurring until `callback` is called. It is only called on the initial render of a `TransitionGroup`.

`componentDidAppear()` **{#componentdidappear}**

`componentDidAppear()`

This is called after the `callback` function that was passed to `componentWillAppear()` is called.

`componentWillEnter()` **{#componentwillenter}**

`componentWillEnter(callback)`

This is called at the same time as `componentDidMount()` for components added to an existing `TransitionGroup`. It will block other animations from occurring until `callback` is called. It will not be called on the initial render of a `TransitionGroup`.

`componentDidEnter()` **{#componentdidenter}**

`componentDidEnter()`

This is called after the `callback` function that was passed to `componentWillEnter()` is called.

`componentWillLeave()` **{#componentwillleave}**

`componentWillLeave(callback)`

This is called when the child has been removed from the `ReactTransitionGroup`. Though the child has been removed, `ReactTransitionGroup` will keep it in the DOM until `callback` is called.

`componentDidLeave()` **{#componentdidleave}**

`componentDidLeave()`

This is called when the `willLeave` `callback` is called (at the same time as `componentWillUnmount()`).

[Go to TOC](#)

Keyed Fragments

Note:

`React.addons` entry point is deprecated as of React v15.5. We now have first class support for fragments which you can read about [here](#).

Importing

```
import createFragment from 'react-addons-create-fragment'; // ES6
var createFragment = require('react-addons-create-fragment'); // ES5 with npm
```

Overview

In most cases, you can use the `key` prop to specify keys on the elements you're returning from `render`. However, this breaks down in one situation: if you have two sets of children that you need to reorder, there's no way to put a key on each set without adding a wrapper element.

That is, if you have a component such as:

```
function Swapper(props) {
  let children;
  if (props.swapped) {
    children = [props.rightChildren, props.leftChildren];
  } else {
    children = [props.leftChildren, props.rightChildren];
  }
  return <div>{children}</div>;
}
```

The children will unmount and remount as you change the `swapped` prop because there aren't any keys marked on the two sets of children.

To solve this problem, you can use the `createFragment` add-on to give keys to the sets of children.

```
Array<ReactNode> createFragment(object children) {#arrayreactnode-createfragmentobject-children}
```

Instead of creating arrays, we write:

```
import createFragment from 'react-addons-create-fragment';

function Swapper(props) {
  let children;
  if (props.swapped) {
    children = createFragment({
      right: props.rightChildren,
      left: props.leftChildren
    });
  }
  return <div>{children}</div>;
}
```

```
    } else {
      children = createFragment({
        left: props.leftChildren,
        right: props.rightChildren
      });
    }
    return <div>{children}</div>;
}
```

The keys of the passed object (that is, `left` and `right`) are used as keys for the entire set of children, and the order of the object's keys is used to determine the order of the rendered children. With this change, the two sets of children will be properly reordered in the DOM without unmounting.

The return value of `createFragment` should be treated as an opaque object; you can use the `React.Children` helpers to loop through a fragment but should not access it directly. Note also that we're relying on the JavaScript engine preserving object enumeration order here, which is not guaranteed by the spec but is implemented by all major browsers and VMs for objects with non-numeric keys.

Performance Tools

Note:

As of React 16, `react-addons-perf` is not supported. Please use [your browser's profiling tools](#) to get insight into which components re-render.

Importing

```
import Perf from 'react-addons-perf'; // ES6
var Perf = require('react-addons-perf'); // ES5 with npm
```

Overview

React is usually quite fast out of the box. However, in situations where you need to squeeze every ounce of performance out of your app, it provides a `shouldComponentUpdate()` method where you can add optimization hints to React's diff algorithm.

In addition to giving you an overview of your app's overall performance, `Perf` is a profiling tool that tells you exactly where you need to put these methods.

See these articles for an introduction to React performance tooling:

- ["How to Benchmark React Components"](#)
- ["Performance Engineering with React"](#)
- ["A Deep Dive into React Perf Debugging"](#)

Development vs. Production Builds

If you're benchmarking or seeing performance problems in your React apps, make sure you're testing with the [minified production build](#). The development build includes extra warnings that are helpful when building your apps, but it is slower due to the extra bookkeeping it does.

However, the perf tools described on this page only work when using the development build of React. Therefore, the profiler only serves to indicate the *relatively* expensive parts of your app.

Using Perf

The `Perf` object can be used with React in development mode only. You should not include this bundle when building your app for production.

Getting Measurements

- `start()`
- `stop()`

- `getLastMeasurements()`

Printing Results

The following methods use the measurements returned by `Perf.getLastMeasurements()` to pretty-print the result.

- `printInclusive()`
 - `printExclusive()`
 - `printWasted()`
 - `printOperations()`
 - `printDOM()`
-

Reference

`start() {#start}`

`stop() {#stop}`

```
Perf.start()  
// ...  
Perf.stop()
```

Start/stop the measurement. The React operations in-between are recorded for analyses below. Operations that took an insignificant amount of time are ignored.

After stopping, you will need `Perf.getLastMeasurements()` to get the measurements.

`getLastMeasurements() {#getlastmeasurements}`

```
Perf.getLastMeasurements()
```

Get the opaque data structure describing measurements from the last start-stop session. You can save it and pass it to the other print methods in `Perf` to analyze past measurements.

Note

Don't rely on the exact format of the return value because it may change in minor releases. We will update the documentation if the return value format becomes a supported part of the public API.

`printInclusive() {#printinclusive}`

```
Perf.printInclusive(measurements)
```

Prints the overall time taken. When no arguments are passed, `printInclusive` defaults to all the measurements from the last recording. This prints a nicely formatted table in the console, like so:

(index)	Owner > component	Inclusive time (ms)	Instances
0	"<root> > App"	15.31	37
1	"App > Box"	6.47	37
2	"Box > Box2"	1.96	37

printExclusive() `{#printexclusive}`

`Perf.printExclusive(measurements)`

"Exclusive" times don't include the times taken to mount the components: processing props, calling `componentWillMount` and `componentDidMount`, etc.

(index)	Component c...	Total inclus...	Exclusive m...	Exclusive r...	Mount time ...	Render time...	Instances
0	"App"	15.31	1.76	1.23	0.04	0.03	37

printWasted() `{#printwasted}`

`Perf.printWasted(measurements)`

The most useful part of the profiler.

"Wasted" time is spent on components that didn't actually render anything, e.g. the render stayed the same, so the DOM wasn't touched.

(index)	Owner > component	Wasted time (ms)	Instances
0	"<root> > App"	15.317999947001226	37
1	"App > Box"	6.479999952716753	37
2	"Box > Box2"	1.9630000460892916	37

printOperations() `{#printoperations}`

`Perf.printOperations(measurements)`

Prints the underlying DOM manipulations, e.g. "set innerHTML" and "remove".

(index)	data-reactid	type	args
0	"0"	"set innerHTML"	"<div data-reactid=".0\..."

printDOM() `{#printdom}`

`Perf.printDOM(measurements)`

This method has been renamed to `printOperations()`. Currently `printDOM()` still exists as an alias but it prints a deprecation warning and will eventually be removed.

PureRenderMixin

Note:

`PureRenderMixin` is a legacy add-on. Use `React.PureComponent` instead.

Importing

```
import PureRenderMixin from 'react-addons-pure-render-mixin'; // ES6
var PureRenderMixin = require('react-addons-pure-render-mixin'); // ES5 with npm
```

Overview

If your React component's render function renders the same result given the same props and state, you can use this mixin for a performance boost in some cases.

Example:

```
const createReactClass = require('create-react-class');

createReactClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

Under the hood, the mixin implements `shouldComponentUpdate`, in which it compares the current props and state with the next ones and returns `false` if the equalities pass.

Note:

This only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only mix into components which have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using [immutable objects](#) to facilitate fast comparisons of nested data.

Furthermore, `shouldComponentUpdate` skips updates for the whole component subtree. Make sure all the children components are also "pure".

[Go to TOC](#)

Shallow Compare

Note:

`shallowCompare` is a legacy add-on. Use `React.memo` or `React.PureComponent` instead.

Importing

```
import shallowCompare from 'react-addons-shallow-compare'; // ES6
var shallowCompare = require('react-addons-shallow-compare'); // ES5 with npm
```

Overview

Before `React.PureComponent` was introduced, `shallowCompare` was commonly used to achieve the same functionality as `PureRenderMixin` while using ES6 classes with React.

If your React component's render function is "pure" (in other words, it renders the same result given the same props and state), you can use this helper function for a performance boost in some cases.

Example:

```
export class SampleComponent extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    return shallowCompare(this, nextProps, nextState);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

`shallowCompare` performs a shallow equality check on the current `props` and `nextProps` objects as well as the current `state` and `nextState` objects.

It does this by iterating on the keys of the objects being compared and returning true when the values of a key in each object are not strictly equal.

`shallowCompare` returns `true` if the shallow comparison for props or state fails and therefore the component should update.

`shallowCompare` returns `false` if the shallow comparison for props and state both pass and therefore the component does not need to update.

[Go to TOC](#)

얕은 렌더러

불러오기

```
import ShallowRenderer from 'react-test-renderer/shallow'; // ES6
var ShallowRenderer = require('react-test-renderer/shallow'); // npm에서 ES5를 사용하는 경우
```

개요

React를 위한 유닛 테스트를 작성할 때 얕은 렌더링이 유용할 수 있습니다. 얕은 렌더링은 컴포넌트를 "한 단계 깊이"로 렌더링할 수 있으며 인스턴스화 또는 렌더링 되지 않는 자식 컴포넌트의 동작에 대해 걱정 없이 렌더링 메서드가 무엇을 반환하는지에 대해 검증할 수 있습니다. 이 작업은 DOM이 필요하지 않습니다.

예를 들어 다음 컴포넌트가 있는 경우

```
function MyComponent() {
  return (
    <div>
      <span className="heading">Title</span>
      <Subcomponent foo="bar" />
    </div>
  );
}
```

다음과 같이 검증할 수 있습니다.

```
import ShallowRenderer from 'react-test-renderer/shallow';

// in your test:
const renderer = new ShallowRenderer();
renderer.render(<MyComponent />);
const result = renderer.getRenderOutput();

expect(result.type).toBe('div');
expect(result.props.children).toEqual([
  <span className="heading">Title</span>,
  <Subcomponent foo="bar" />
]);
```

얕은 테스팅은 현재 몇 가지 제한 사항이 있습니다. 다시 말해 refs를 지원하지 않습니다.

주의

우리는 또한 Enzyme의 [Shallow Rendering API](#)를 확인해 볼 것을 권장합니다. 같은 기능에 대해 더 높은 수준의 API를 제공합니다.

참조

`shallowRenderer.render() {#shallowrendererrender}`

`shallowRenderer.render()`는 테스트 중인 컴포넌트를 렌더링하는 "장소(place)"로 생각할 수 있으며 이것으로부터 컴포넌트의 출력을 추출할 수 있습니다.

`shallowRenderer.render()`는 `root.render()`와 비슷하지만 DOM을 요구하지 않으며 오직 한 단계 깊이만을 렌더링합니다. 이것은 컴포넌트의 자식들이 어떻게 구현되었는지 신경 쓰지 않고 독립적으로 테스트할 수 있음을 의미합니다.

`shallowRenderer.getRenderOutput() {#shallowrenderergetrenderoutput}`

`shallowRenderer.render()`가 호출된 후 `shallowRenderer.getRenderOutput()`을 사용하여 얕게 렌더링 된 출력을 얻을 수 있습니다.

그러면 출력에 대해 검증을 시작할 수 있습니다.

테스팅 도구

Importing

```
import ReactTestUtils from 'react-dom/test-utils'; // ES6
var ReactTestUtils = require('react-dom/test-utils'); // npm과 ES5
```

개요

`ReactTestUtils` 는 여러분이 선택한 테스팅 프레임워크에서 테스트를 쉽게 진행할 수 있도록 해 줍니다. Facebook에서는 [Jest](#)를 이용해 더욱 쉽게 JavaScript 테스트를 하고 있습니다. Jest 웹사이트의 [React 자습서](#) 문서를 통해 Jest를 시작하는 방법에 대해서 알아보세요.

주의

Facebook에서는 [React Testing Library](#) 사용을 권장합니다. 이 라이브러리는 사용자가 컴포넌트를 사용하는 것처럼 테스트를 작성할 수 있도록 설계되었습니다.

React v16 이하에서는 [Enzyme](#)을 통해 React 컴포넌트의 출력을 쉽게 검증하고 조작하고 탐색할 수 있습니다.

- `act()`
- `mockComponent()`
- `isElement()`
- `isElementOfType()`
- `isDOMComponent()`
- `isCompositeComponent()`
- `isCompositeComponentWithType()`
- `findAllInRenderedTree()`
- `scryRenderedDOMComponentsWithClass()`
- `findRenderedDOMComponentWithClass()`
- `scryRenderedDOMComponentsWithTag()`
- `findRenderedDOMComponentWithTag()`
- `scryRenderedComponentsWithType()`
- `findRenderedComponentWithType()`
- `renderIntoDocument()`
- `Simulate`

참조사항

`act() {#act}`

컴포넌트의 진단을 준비하기 위해서는 컴포넌트를 렌더링하고 간신히주는 코드를 `act()` 를 호출한 것의 안에 넣어줘야 합니다. 이를 통해 React를 브라우저 내에서 동작하는 것과 비슷한 환경에서 테스트할 수 있습니다.

주의

`react-test-renderer` 를 사용한다면, 똑같이 작동하는 `act` `export`가 제공됩니다.

예를 들어, 다음과 같은 `Counter` 컴포넌트가 있다고 해봅시다.

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 0};
    this.handleClick = this.handleClick.bind(this);
  }
  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }
  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }
  handleClick() {
    this.setState(state => ({
      count: state.count + 1,
    }));
  }
  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={this.handleClick}>
          Click me
        </button>
      </div>
    );
  }
}
```

이런 방식으로 테스트 할 수 있습니다.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { act } from 'react-dom/test-utils';
import Counter from './Counter';

let container;

beforeEach(() => {
  container = document.createElement('div');
  document.body.appendChild(container);
});

afterEach(() => {
  document.body.removeChild(container);
  container = null;
});

it('can render and update a counter', () => {
  // 첫 render와 componentDidMount를 테스트
  act(() => {
    ReactDOM.createRoot(container).render(<Counter />);
  });
});
```

```
});  
const button = container.querySelector('button');  
const label = container.querySelector('p');  
expect(label.textContent).toBe('You clicked 0 times');  
expect(document.title).toBe('You clicked 0 times');  
  
// 두 번째 render와 componentDidUpdate를 테스트  
act(() => {  
  button.dispatchEvent(new MouseEvent('click', {bubbles: true}));  
});  
expect(label.textContent).toBe('You clicked 1 times');  
expect(document.title).toBe('You clicked 1 times');  
});
```

- DOM 이벤트 발행은 DOM 컨테이너가 `document` 객체에 추가되었을 때만 작동한다는 점을 잊지마세요. 불필요하게 반복 되는 코드를 줄이기 위해서 `react-testing-library` 와 같은 라이브러리를 사용할 수 있습니다.
- [테스트 방법](#) 문서에 `act()` 의 동작 방식에 대한 자세한 내용이 예시와 사용법과 함께 포함되어 있습니다.

mockComponent() [\[#mockcomponent\]](#)

```
mockComponent(  
  componentClass,  
  [mockTagName]  
)
```

모의 컴포넌트 모듈을 이 메서드에 넘겨 유용한 메서드들을 붙여 증강해 더미 React 컴포넌트로 사용할 수 있습니다. 보통의 경우처럼 렌더링 하지 않고 그 대신 컴포넌트는 간단하게 `<div>` 태그가 됩니다. `mockTagName` 값을 넘겨준다면 `<div>` 대신 다른 태그로 만들어 줄 수 있습니다.

주의

`mockComponent()` 는 더 이상 쓰이지 않는 API입니다. `jest.mock()` 사용을 추천합니다.

isElement() [\[#iselement\]](#)

```
isElement(element)
```

`element` 가 React의 element라면 `true` 를 반환합니다.

isElementOfType() [\[#iselementoftype\]](#)

```
isElementOfType(  
  element,  
  componentClass  
)
```

`element` 가 `componentClass` 타입의 React element라면 `true` 를 반환합니다.

isDOMComponent() {#isdomcomponent}

```
isDOMComponent(instance)
```

`instance` 가 `<div>` 나 `` 같은 DOM 컴포넌트라면 `true` 를 반환합니다.

isCompositeComponent() {#iscompositecomponent}

```
isCompositeComponent(instance)
```

`instance` 가 클래스나 함수 같이 사용자가 정의한 컴포넌트라면 `true` 를 반환합니다.

isCompositeComponentWithType() {#iscompositecomponentwithtype}

```
isCompositeComponentWithType(
  instance,
  componentClass
)
```

`instance` 가 `componentClass` 타입을 가진 컴포넌트라면 `true` 를 반환합니다.

findAllInRenderedTree() {#findallinrenderedtree}

```
findAllInRenderedTree(
  tree,
  test
)
```

`tree` 의 모든 컴포넌트를 탐색하여 `test(component)` 가 `true` 일 때 모든 컴포넌트를 축적합니다. 이 함수는 그 자체만으로는 유용하지 않지만, 다른 테스트 도구의 기반이 됩니다.

scryRenderedDOMComponentsWithClass() {#scryrendereddomcomponentswithclass}

```
scryRenderedDOMComponentsWithClass(
  tree,
  className
)
```

렌더링 된 트리에서 조건 `className` 에 만족하는 `className` 명을 가지고 있는 DOM 컴포넌트의 DOM 엘리먼트를 모두 검색합니다.

findRenderedDOMComponentWithClass() {#findrendereddomcomponentwithclass}

```
findRenderedDOMComponentWithClass(
  tree,
  className
)
```

`scryRenderedDOMComponentsWithClass()` 와 기능이 유사하나 결과값이 하나라고 가정하고 그 결과값만을 반환합니다. 두 개 이상의 결과값이 있다면 예외를 반환합니다.

`scryRenderedDOMComponentsWithTag()` {#scryrendereddomcomponentswithtag}

```
scryRenderedDOMComponentsWithTag(
  tree,
  tagName
)
```

렌더링 된 트리 내에서 조건 `tagName` 에 만족하는 tagName을 가진 DOM 컴포넌트의 DOM 엘리먼트를 모두 검색합니다.

`findRenderedDOMComponentWithTag()` {#findrendereddomcomponentwithtag}

```
findRenderedDOMComponentWithTag(
  tree,
  tagName
)
```

`scryRenderedDOMComponentsWithTag()` 와 기능이 유사하나 결과값이 하나라고 가정하고 그 결과값만을 반환합니다. 두 개 이상의 결과값이 있다면 예외를 뱉습니다.

`scryRenderedComponentsWithType()` {#scryrenderedcomponentswithtype}

```
scryRenderedComponentsWithType(
  tree,
  componentClass
)
```

`componentClass` 타입을 가진 모든 인스턴스를 검색합니다.

`findRenderedComponentWithType()` {#findrenderedcomponentwithtype}

```
findRenderedComponentWithType(
  tree,
  componentClass
)
```

`scryRenderedComponentsWithType()` 와 기능이 유사하나 결과값이 하나라고 가정하고 그 결과값만을 반환합니다. 두 개 이상의 결과값이 있다면 예외를 뱉습니다.

`renderIntoDocument()` {#renderintodocument}

```
renderIntoDocument(element)
```

React 엘리먼트를 document내의 떨어져 있는 DOM 노드에 렌더링합니다. 이 함수를 쓰려면 **DOM**이 필요합니다. 이 함수는 다음 코드와 같은 기능을 합니다.

```
const domContainer = document.createElement('div');
ReactDOM.createRoot(domContainer).render(element);
```

주의

`window`, `window.document` 와 `window.document.createElement` 는 React 를 가져와서 사용하기 전에도 전역적으로 사용할 수 있습니다. 그렇지 않다면 React는 DOM에 접근할 수 없다고 간주할 것이며 `setState` 와 같은 메서드들이 작동하지 않을 것 입니다.

다른 테스팅 도구들

Simulate {#simulate}

```
Simulate.{eventName}(
  element,
  [eventData]
)
```

이벤트 데이터인 `eventData` 를 옵션으로 준 DOM 노드에 붙이는 이벤트를 시뮬레이팅합니다.

`Simulate` 는 React가 이해하는 모든 이벤트를 위한 메서드를 가집니다.

엘리먼트 클릭

```
// <button ref={(node) => this.button = node}>...</button>
const node = this.button;
ReactTestUtils.Simulate.click(node);
```

입력 필드의 값을 바꾼 뒤 ENTER키 누르기

```
// <input ref={(node) => this.textInput = node} />
const node = this.textInput;
node.value = 'giraffe';
ReactTestUtils.Simulate.change(node);
ReactTestUtils.Simulate.keyDown(node, {key: "Enter", keyCode: 13, which: 13});
```

주의

컴포넌트 내에서 사용하고 있는 `keyCode`, `which`과 같은 이벤트 프로퍼티는 별도로 제공해주어야 합니다. React에서는 이러한 이벤트 프로퍼티를 자동으로 만들어 주지 않습니다.

[Go to TOC](#)

Two-way Binding Helpers

Note:

`LinkedStateMixin` is deprecated as of React v15. The recommendation is to explicitly set the value and change handler, instead of using `LinkedStateMixin`.

Importing

```
import LinkedStateMixin from 'react-addons-linked-state-mixin'; // ES6
var LinkedStateMixin = require('react-addons-linked-state-mixin'); // ES5 with npm
```

Overview

`LinkedStateMixin` is an easy way to express two-way binding with React.

In React, data flows one way: from owner to child. We think that this makes your app's code easier to understand. You can think of it as "one-way data binding."

However, there are lots of applications that require you to read some data and flow it back into your program. For example, when developing forms, you'll often want to update some React `state` when you receive user input. Or perhaps you want to perform layout in JavaScript and react to changes in some DOM element size.

In React, you would implement this by listening to a "change" event, read from your data source (usually the DOM) and call `setState()` on one of your components. "Closing the data flow loop" explicitly leads to more understandable and easier-to-maintain programs. See [our forms documentation](#) for more information.

Two-way binding -- implicitly enforcing that some value in the DOM is always consistent with some React `state` -- is concise and supports a wide variety of applications. We've provided `LinkedStateMixin`: syntactic sugar for setting up the common data flow loop pattern described above, or "linking" some data source to React `state`.

Note:

`LinkedStateMixin` is just a thin wrapper and convention around the `onChange / setState()` pattern. It doesn't fundamentally change how data flows in your React application.

LinkedStateMixin: Before and After

Here's a simple form example without using `LinkedStateMixin`:

```
var createReactClass = require('create-react-class');

var NoLink = createReactClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  handleChange: function(event) {
    this.setState({message: event.target.value});
  },
  render: function() {
    var message = this.state.message;
    return <input type="text" value={message} onChange={this.handleChange} />;
  }
});
```

This works really well and it's very clear how data is flowing, however, with a lot of form fields it could get a bit verbose. Let's use `LinkedStateMixin` to save us some typing:

```
var createReactClass = require('create-react-class');

var WithLink = createReactClass({
  mixins: [LinkedStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    return <input type="text" valueLink={this.linkState('message')} />;
  }
});
```

`LinkedStateMixin` adds a method to your React component called `linkState()`. `linkState()` returns a `valueLink` object which contains the current value of the React state and a callback to change it.

`valueLink` objects can be passed up and down the tree as props, so it's easy (and explicit) to set up two-way binding between a component deep in the hierarchy and state that lives higher in the hierarchy.

Note that checkboxes have a special behavior regarding their `value` attribute, which is the value that will be sent on form submit if the checkbox is checked (defaults to `on`). The `value` attribute is not updated when the checkbox is checked or unchecked. For checkboxes, you should use `checkedLink` instead of `valueLink`:

```
<input type="checkbox" checkedLink={this.linkState('booleanValue')} />
```

Under the Hood

There are two sides to `LinkedStateMixin`: the place where you create the `valueLink` instance and the place where you use it. To prove how simple `LinkedStateMixin` is, let's rewrite each side separately to be more explicit.

valueLink Without LinkedStateMixin

```
var createReactClass = require('create-react-class');

var WithoutMixin = createReactClass({
```

```

getInitialState: function() {
  return {message: 'Hello!'};
},
handleChange: function(newValue) {
  this.setState({message: newValue});
},
render: function() {
  var valueLink = {
    value: this.state.message,
    requestChange: this.handleChange
  };
  return <input type="text" valueLink={valueLink} />;
}
);

```

As you can see, `valueLink` objects are very simple objects that just have a `value` and `requestChange` prop. And `LinkedStateMixin` is similarly simple: it just populates those fields with a `value` from `this.state` and a callback that calls `this.setState()`.

LinkedStateMixin Without valueLink

```

var LinkedStateMixin = require('react-addons-linked-state-mixin');
var createReactClass = require('create-react-class');

var WithoutLink = createReactClass({
  mixins: [LinkedStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    var valueLink = this.linkState('message');
    var handleChange = function(e) {
      valueLink.requestChange(e.target.value);
    };
    return <input type="text" value={valueLink.value} onChange={handleChange} />;
  }
});

```

The `valueLink` prop is also quite simple. It simply handles the `onChange` event and calls `this.props.valueLink.requestChange()` and also uses `this.props.valueLink.value` instead of `this.props.value`. That's it!

Immutability Helpers

Note:

`update` is a legacy add-on. Use `immutability-helper` instead.

Importing

```
import update from 'react-addons-update'; // ES6
var update = require('react-addons-update'); // ES5 with npm
```

Overview

React lets you use whatever style of data management you want, including mutation. However, if you can use immutable data in performance-critical parts of your application it's easy to implement a fast `shouldComponentUpdate()` method to significantly speed up your app.

Dealing with immutable data in JavaScript is more difficult than in languages designed for it, like [Clojure](#). However, we've provided a simple immutability helper, `update()`, that makes dealing with this type of data much easier, *without* fundamentally changing how your data is represented. You can also take a look at Facebook's [Immutable-js](#) and the [Advanced Performance](#) section for more detail on Immutable-js.

The Main Idea

If you mutate data like this:

```
myData.x.y.z = 7;
// or...
myData.a.b.push(9);
```

You have no way of determining which data has changed since the previous copy has been overwritten. Instead, you need to create a new copy of `myData` and change only the parts of it that need to be changed. Then you can compare the old copy of `myData` with the new one in `shouldComponentUpdate()` using triple-equals:

```
const newData = deepCopy(myData);
newData.x.y.z = 7;
newData.a.b.push(9);
```

Unfortunately, deep copies are expensive, and sometimes impossible. You can alleviate this by only copying objects that need to be changed and by reusing the objects that haven't changed. Unfortunately, in today's JavaScript this can be cumbersome:

```
const newData = extend(myData, {
  x: extend(myData.x, {
    y: extend(myData.x.y, {z: 7}),
```

```
  },
  a: extend(myData.a, {b: myData.a.b.concat(9)})
});
```

While this is fairly performant (since it only makes a shallow copy of `log n` objects and reuses the rest), it's a big pain to write. Look at all the repetition! This is not only annoying, but also provides a large surface area for bugs.

update() {#update}

`update()` provides simple syntactic sugar around this pattern to make writing this code easier. This code becomes:

```
import update from 'react-addons-update';

const newData = update(myData, {
  x: {y: {z: {$set: 7}}},
  a: {b: {$push: [9]}}
});
```

While the syntax takes a little getting used to (though it's inspired by MongoDB's query language) there's no redundancy, it's statically analyzable and it's not much more typing than the mutative version.

The `$`-prefixed keys are called *commands*. The data structure they are "mutating" is called the *target*.

Available Commands

- `={$push: array} push()` all the items in `array` on the target.
- `={$unshift: array} unshift()` all the items in `array` on the target.
- `={$splice: array of arrays} for each item in arrays call splice()` on the target with the parameters provided by the item.
- `={$set: any} replace the target entirely.`
- `={$merge: object} merge the keys of object with the target.`
- `={$apply: function} passes in the current value to the function and updates it with the new returned value.`

Examples

Simple push

```
const initialArray = [1, 2, 3];
const newArray = update(initialArray, {$push: [4]}); // => [1, 2, 3, 4]
```

`initialArray` is still `[1, 2, 3]`.

Nested collections

```
const collection = [1, 2, {a: [12, 17, 15]}];
const newCollection = update(collection, {2: {a: {$splice: [[1, 1, 13, 14]]}}});
// => [1, 2, {a: [12, 13, 14, 15]}]
```

This accesses `collection`'s index `2`, key `a`, and does a splice of one item starting from index `1` (to remove `17`) while inserting `13` and `14`.

Updating a value based on its current one

```
const obj = {a: 5, b: 3};
const newObj = update(obj, {b: {$apply: function(x) {return x * 2;}}});
// => {a: 5, b: 6}
// This is equivalent, but gets verbose for deeply nested collections:
const newObj2 = update(obj, {b: {$set: obj.b * 2}});
```

(Shallow) Merge

```
const obj = {a: 5, b: 3};
const newObj = update(obj, {$merge: {b: 6, c: 7}}); // => {a: 5, b: 6, c: 7}
```

Add-Ons

Note:

`React.addons` entry point is deprecated as of React v15.5. The add-ons have moved to separate modules, and some of them have been deprecated.

The React add-ons are a collection of useful utility modules for building React apps. **These should be considered experimental** and tend to change more often than the core.

- `createFragment`, to create a set of externally-keyed children.

The add-ons below are in the development (unminified) version of React only:

- `Perf`, a performance profiling tool for finding optimization opportunities.
- `ReactTestUtils`, simple helpers for writing test cases.

Legacy Add-ons

The add-ons below are considered legacy and their use is discouraged. They will keep working in observable future, but there is no further development.

- `PureRenderMixin`. Use `React.PureComponent` instead.
- `shallowCompare`, a helper function that performs a shallow comparison for props and state in a component to decide if a component should update. We recommend using `React.PureComponent` instead.
- `update`. Use `kolodny/immutability-helper` instead.
- `ReactDOMFactories`, pre-configured DOM factories to make React easier to use without JSX.

Deprecated Add-ons

- `LinkedStateMixin` has been deprecated.
- `TransitionGroup` and `CSSTransitionGroup` have been deprecated in favor of [their drop-in replacements](#).

Using React with Add-ons

You can install the add-ons individually from npm (e.g. `npm install react-addons-create-fragment`) and import them:

```
import createFragment from 'react-addons-create-fragment'; // ES6
var createFragment = require('react-addons-create-fragment'); // ES5 with npm
```

When using React 15 or earlier from a CDN, you can use `react-with-addons.js` instead of `react.js`:

```
<script src="https://unpkg.com/react@15/dist/react-with-addons.js"></script>
```

The add-ons will be available via the `React.addons` global (e.g. `React.addons.TestUtils`).

[Go to TOC](#)

CDN 링크

React와 ReactDOM 모두 CDN을 통해 사용할 수 있습니다.

```
<script crossorigin src="https://unpkg.com/react@18/umd/react.development.js">
</script>
<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"></script>
```

위의 코드는 개발용으로 적합하며 배포용 버전에는 적합하지 않습니다. React의 용량 및 성능 최적화된 배포용 버전은 아래와 같이 제공되고 있습니다.

```
<script crossorigin src="https://unpkg.com/react@18/umd/react.production.min.js">
</script>
<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.production.min.js"></script>
```

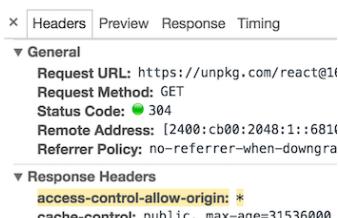
`react` 와 `react-dom`의 특정 버전을 로딩하려면 `18` 을 사용하고자 하는 버전 넘버로 대체하면 됩니다.

crossorigin 속성이 필요한 이유

CDN을 통해 React를 사용한다면, `crossorigin` 어트리뷰트(attribute)와 함께 사용하는 것을 권장합니다.

```
<script crossorigin src="..."></script>
```

또한 사용 중인 CDN이 `Access-Control-Allow-Origin: *` HTTP 헤더 설정을 사용하는지 확인하는 것이 좋습니다.



`Access-Control-Allow-Origin: *`

이를 통해 React 16 버전과 다음 버전에서 더 쉽게 [에러 처리](#)를 할 수 있습니다.

[Go to TOC](#)

코드 분할

번들링

대부분 React 앱들은 [Webpack](#), [Rollup](#) 또는 [Browserify](#) 같은 툴을 사용하여 여러 파일을 하나로 병합한 "번들 된" 파일을 웹 페이지에 포함하여 한 번에 전체 앱을 로드 할 수 있습니다.

예시

App

```
// app.js
import { add } from './math.js';

console.log(add(16, 26)); // 42
```

```
// math.js
export function add(a, b) {
  return a + b;
}
```

Bundle

```
function add(a, b) {
  return a + b;
}

console.log(add(16, 26)); // 42
```

주의

실제 번들은 위 예시와는 많이 다르게 보일 겁니다.

[Create React App](#)이나 [Next.js](#), [Gatsby](#) 혹은 비슷한 툴을 사용한다면 여러분이 설치한 앱에서 Webpack을 같이 설치했을 겁니다.

이런 툴을 사용하지 않는다면 여러분이 스스로 번들링을 설정해야 합니다. 이 경우 Webpack의 [설치하기](#) 문서와 [시작하기](#) 문서를 참조해주세요.

코드 분할

번들링은 훌륭하지만 여러분의 앱이 커지면 번들도 커집니다. 특히 큰 규모의 서드 파티 라이브러리를 추가할 때 실수로 앱이 커져서 로드 시간이 길어지는 것을 방지하기 위해 코드를 주의 깊게 살펴야 합니다.

번들이 거대해지는 것을 방지하기 위한 좋은 해결방법은 번들을 "나누는" 것입니다. 코드 분할은 런타임에 여러 번들을 동적으로 만들고 불러오는 것으로 [Webpack](#), [Rollup](#)과 [Browserify](#) ([factor-bundle](#)) 같은 번들러가 지원하는 기능입니다.

코드 분할은 여러분의 앱을 "지연 로딩" 하게 도와주고 앱 사용자에게 획기적인 성능 향상을 하게 합니다. 앱의 코드 양을 줄이지 않고도 사용자가 필요하지 않은 코드를 불러오지 않게 하며 앱의 초기화 로딩에 필요한 비용을 줄여줍니다.

import() {#import}

앱에 코드 분할을 도입하는 가장 좋은 방법은 동적 `import()` 문법을 사용하는 방법입니다.

Before

```
import { add } from './math';
console.log(add(16, 26));
```

After

```
import("./math").then(math => {
  console.log(math.add(16, 26));
});
```

Webpack이 이 구문을 만나게 되면 앱의 코드를 분할합니다. Create React App을 사용하고 있다면 이미 Webpack이 구성이 되어 있기 때문에 즉시 사용할 수 있습니다. [Next.js](#) 역시 지원합니다.

[코드 분할 가이드](#)를 참조하세요. Webpack 설정은 [가이드](#)에 있습니다.

Babel을 사용할 때는 Babel이 동적 `import`를 인식할 수 있지만 변환하지는 않도록 합니다. 이를 위해 [@babel/plugin-syntax-dynamic-import](#)를 사용하세요.

React.lazy {#reactlazy}

`React.lazy` 함수를 사용하면 동적 `import`를 사용해서 컴포넌트를 렌더링 할 수 있습니다.

Before

```
import OtherComponent from './OtherComponent';
```

After

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

`MyComponent` 가 처음 렌더링 될 때 `OtherComponent` 를 포함한 번들을 자동으로 불러옵니다.

`React.lazy` 는 동적 `import()` 를 호출하는 함수를 인자로 가집니다. 이 함수는 React 컴포넌트를 `default export`로 가진 모듈 객체가 이행되는 `Promise` 를 반환해야 합니다.

`lazy` 컴포넌트는 `Suspense` 컴포넌트 하위에서 렌더링되어야 하며, `Suspense` 는 `lazy` 컴포넌트가 로드되길 기다리는 동안 로딩 화면과 같은 예비 컨텐츠를 보여줄 수 있게 해줍니다.

```
import React, { Suspense } from 'react';
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

```
function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
      </Suspense>
    </div>
  );
}
```

`fallback` prop은 컴포넌트가 로드될 때까지 기다리는 동안 렌더링하려는 React 엘리먼트를 받아들입니다. `Suspense` 컴포넌트는 `lazy` 컴포넌트를 감쌉니다. 하나의 `Suspense` 컴포넌트로 여러 `lazy` 컴포넌트를 감쌀 수도 있습니다.

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

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </div>
  );
}
```

Avoiding fallbacks

Any component may suspend as a result of rendering, even components that were already shown to the user. In order for screen content to always be consistent, if an already shown component suspends, React has to hide its tree up to the closest `<Suspense>` boundary. However, from the user's perspective, this can be disorienting.

Consider this tab switcher:

```
import React, { Suspense } from 'react';
import Tabs from './Tabs';
import Glimmer from './Glimmer';

const Comments = React.lazy(() => import('./Comments'));
const Photos = React.lazy(() => import('./Photos'));

function MyComponent() {
  const [tab, setTab] = React.useState('photos');

  function handleTabSelect(tab) {
    setTab(tab);
  };

  return (
    <div>
      <Tabs onSelect={handleTabSelect} />
      <Suspense fallback={<Glimmer />}>
```

```

        {tab === 'photos' ? <Photos /> : <Comments />}
      </Suspense>
    </div>
  );
}

```

In this example, if tab gets changed from `'photos'` to `'comments'`, but `Comments` suspends, the user will see a glimmer. This makes sense because the user no longer wants to see `Photos`, the `Comments` component is not ready to render anything, and React needs to keep the user experience consistent, so it has no choice but to show the `Glimmer` above.

However, sometimes this user experience is not desirable. In particular, it is sometimes better to show the "old" UI while the new UI is being prepared. You can use the new `startTransition` API to make React do this:

```

function handleTabSelect(tab) {
  startTransition(() => {
    setTab(tab);
  });
}

```

Here, you tell React that setting tab to `'comments'` is not an urgent update, but is a `transition` that may take some time. React will then keep the old UI in place and interactive, and will switch to showing `<Comments />` when it is ready. See [Transitions](#) for more info.

Error boundaries

네트워크 장애 같은 이유로 다른 모듈을 로드에 실패할 경우 에러를 발생시킬 수 있습니다. 이때 [Error Boundaries](#)를 이용하여 사용자의 경험과 복구 관리를 처리할 수 있습니다. Error Boundary를 만들고 lazy 컴포넌트를 감싸면 네트워크 장애가 발생했을 때 에러를 표시할 수 있습니다.

```

import React, { Suspense } from 'react';
import MyErrorBoundary from './MyErrorBoundary';

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

const MyComponent = () => (
  <div>
    <MyErrorBoundary>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </MyErrorBoundary>
  </div>
);

```

Route-based code splitting

앱에 코드 분할을 어느 곳에 도입할지 결정하는 것은 조금 까다롭습니다. 여러분은 사용자의 경험을 해치지 않으면서 번들을 균등하게 분배 할 곳을 찾고자 합니다.

이를 시작하기 좋은 장소는 라우트입니다. 웹 페이지를 불러오는 시간은 페이지 전환에 어느 정도 발생하며 대부분 페이지를 한번에 렌더링하기 때문에 사용자가 페이지를 렌더링하는 동안 다른 요소와 상호작용하지 않습니다.

`React.lazy` 를 `React Router` 라이브러리를 사용해서 애플리케이션에 라우트 기반 코드 분할을 설정하는 예시입니다.

```
import React, { Suspense, lazy } from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </Suspense>
  </Router>
);
```

Named Exports

`React.lazy` 는 현재 default exports만 지원합니다. named exports를 사용하고자 한다면 default로 이름을 재정의한 중간 모듈을 생성할 수 있습니다. 이렇게 하면 tree shaking이 계속 동작하고 사용하지 않는 컴포넌트는 가져오지 않습니다.

```
// ManyComponents.js
export const MyComponent = /* ... */;
export const MyUnusedComponent = /* ... */;
```

```
// MyComponent.js
export { MyComponent as default } from "./ManyComponents.js";
```

```
// MyApp.js
import React, { lazy } from 'react';
const MyComponent = lazy(() => import("./MyComponent.js"));
```

Codebase Overview

이 섹션은 React 코드베이스 구조와 관례, 그리고 구현에 대한 개요를 설명하고 있습니다.

[React에 기여](#)하고자 한다면, 이 문서를 참고하여 프로젝트를 수정하길 바랍니다.

React 앱을 개발하는데 있어서 아래 관례들의 사용을 반드시 추천하지는 않습니다. 대부분의 관례들은 이전에 많이 사용된 것들이며, 차차 수정될 것입니다.

최상위 폴더

[React 저장소](#)를 클론하게 되면, 다음과 같은 몇 개의 최상위 폴더를 볼 수 있습니다.

- `packages` 폴더는 React 저장소에 있는 모든 패키지들에 대해 `package.json`과 같은 메타데이터와 `src` 폴더를 포함하고 있습니다. 변경하고자 하는 부분이 코드와 관련되어 있다면, `src` 폴더에 대부분을 할애하게 될 것입니다.
- `fixtures`에는 기여자들을 위한 몇 가지 React 테스트 애플리케이션이 존재합니다.
- `build` 폴더에는 React의 빌드 결과가 생성됩니다. 저장소에는 존재하지 않지만, React를 클론하고 처음 [빌드](#)할 시 생성됩니다.

설명 문서는 React로부터 [분리된 저장소](#)에 존재합니다.

위에서 언급한 것 외에 별도의 최상위 폴더가 존재하지만, 대부분 기여하는데 있어서 사용되지 않을 것입니다.

Colocated 테스트

해당 프로젝트에는 유닛 테스트를 위한 상위 디렉토리가 존재하지 않습니다. 대신 소스 코드와 함께 `__tests__` 디렉토리 내부에 테스트 코드를 함께 넣어뒀습니다.

예를 들어, `setInnerHTML.js`에 대한 테스트 코드는 `__tests__/setInnerHTML-test.js`에 존재합니다.

경고와 불변식

React 코드베이스에서는 `console.error`를 사용해 경고를 표시합니다.

```
if (__DEV__) {
  console.error('Something is wrong.');
}
```

경고는 개발 시에만 표시되며 배포 시에는 완전히 무시됩니다. 실행 시에 몇 가지 코드를 무시하기 위해서, 다음과 같이 `invariant` 모듈을 활용할 수 있습니다.

```
var invariant = require('invariant');

invariant(
  2 + 2 === 4,
  'You shall not pass!'
);
```

`invariant`의 대상 조건식이 `false` 일 때 예외가 발생합니다.

'불변식'은 '해당 조건식이 항상 참일 경우'를 의미합니다. 코드의 동작을 확실하게 하기 위한 요소라고 생각할 수 있습니다.

개발용과 배포용 프로그램이 서로 유사하게 동작하는 것은 중요합니다. 따라서 `invariant` 예외는 개발용과 배포용 프로그램에서 동시에 발생합니다. 오류 메시지는 배포용 프로그램에서 오류 코드로 자동 변환됩니다.

개발과 배포

의사 전역 변수 `__DEV__`를 사용하여 개발 시에만 작동하는 코드를 작성할 수 있습니다.

해당 변수는 컴파일 단계에서 한번에 처리되며, CommonJS 빌드 시에 `process.env.NODE_ENV !== 'production'`로 변환됩니다.

스탠드얼론 빌드의 경우, 해당 변수는 압축하지 않을 시 `true`로 취급되며 압축 시 `if` 블록을 포함하여 전부 제거됩니다.

```
if (__DEV__) {
  // This code will only run in development.
}
```

Flow

React는 최근 [Flow](#) 검사를 도입하기 시작했습니다. 라이센스 주석에 `@flow` 표시가 포함된 파일은 자료형 검사를 받게 됩니다.

해당 프로젝트는 [이미 존재하는 코드에 대한 Flow 형식 추가](#)에 대한 풀 리퀘스트를 허용하고 있습니다. Flow 검사 형식은 다음과 같이 구성되어 있습니다.

```
ReactRef.detachRefs = function(
  instance: ReactInstance,
  element: ReactElement | string | number | null | false,
): void {
  // ...
}
```

가능하면 새로 작성하는 코드에 Flow 검사 형식을 포함해 주길 바랍니다. `yarn flow` 명령어를 통해 Flow에 대한 형식 검사를 직접 수행해 볼 수 있습니다.

다양한 패키지

React는 [monorepo](#)입니다. 해당 저장소는 여러 분리된 패키지를 포함하고 있으며, 각 변경점들은 함께 반영되고 모든 이슈는 한 곳에서 관리됩니다.

React 코어

React의 '코어'는 모든 [최상위 React API](#)를 포함합니다. 예를 들면 다음과 같습니다.

- `React.createElement()`
- `React.Component`
- `React.Children`

React 코어는 컴포넌트를 정의하는 데에 필요한 API만 포함하고 있습니다. React 코어는 [재조정](#) 알고리즘과 플랫폼 전용 코드를 포함하고 있지 않습니다. 해당 코어는 React DOM과 React 네이티브의 컴포넌트에서 동시에 사용됩니다.

React 코어의 소스 코드는 `packages/react`에 위치하고 있습니다. 해당 코드는 npm에서 `react` 패키지로 받을 수 있습니다. 관련된 스탠드얼론 브라우저 빌드는 `react.js`이며, 전역에서 `React` 키워드를 통해 접근할 수 있습니다.

렌더러

React는 원래 DOM을 대상으로 하여 개발됐지만, 이후 [React 네이티브](#)를 통해 네이티브 플랫폼 또한 지원하게 되었습니다. 본 문단은 React 내부의 '렌더러'에 대해 간략히 소개합니다.

렌더러는 **React** 트리의 플랫폼 종속적인 변환 방법을 관리합니다.

렌더러는 `packages/` 폴더 안에 다음과 같이 존재합니다.

- [React DOM Renderer](#)는 React 컴포넌트를 DOM으로 렌더링합니다. 해당 렌더러는 [최상위 레벨 ReactDOM API](#)에 구현되어 있으며 npm에서 `react-dom` 패키지로 받을 수 있습니다. 또한 스탠드얼론 브라우저에서 `react-dom.js`를 통해 전역 `ReactDOM` 키워드로 접근할 수 있습니다.
- [React Native Renderer](#)는 React 컴포넌트를 네이티브 뷰로 렌더링합니다. 해당 렌더러는 React 네이티브 내부에서 사용됩니다.
- [React Test Renderer](#)는 React 컴포넌트를 JSON 형식으로 렌더링합니다. 해당 렌더러는 [Jest](#)의 [Snapshot 테스트](#)에 사용되며, npm에서 `react-test-renderer` 패키지로 받을 수 있습니다.

언급한 것 외에 공식적으로 지원하는 렌더러는 `react-art`로 유일합니다. [GitHub 저장소](#)에서 분리되어 있었지만, 현재는 메인 소스 트리로 옮겨졌습니다.

주의

`react-native-renderer`는 React 네이티브 구현체와 상호작용하는 매우 얇은 레이어입니다. 네이티브 뷰를 관리하는 실제 플랫폼 관련 코드는 [React 네이티브 저장소](#)에 존재합니다.

재조정자

React DOM과 React 네이티브 같은 매우 다른 경우를 포함하여, 렌더러들은 상당 부분 동작 방식을 서로 공유해야 합니다. 특히 [재조정](#) 알고리즘의 경우는 더욱 그렇습니다. 이를 통해 렌더링, 사용자 정의 컴포넌트, 상태, 생명주기 메서드, 레퍼런스가 플랫폼에 상관없이 일관적으로 작동해야 합니다.

이를 해결하기 위해 서로 다른 렌더러들은 몇 가지의 코드를 공유하며, 해당 부분을 '재조정자'라고 부릅니다. `setState()`와 같은 함수가 수정되어야 할 때, 재조정자는 트리에 있는 컴포넌트의 `render()` 함수를 호출한 후 마운트나 업데이트, 혹은 마운트해제를 실시합니다.

재조정자는 현재로서는 어떠한 public API도 존재하지 않기 때문에 개별적인 패키지로서 존재하지 않습니다. 대신 React DOM과 React 네이티브 같은 렌더러에 개별적으로 사용됩니다.

스택 재조정자

'스택' 재조정자는 React 15 이하에서 구현되었습니다. 현재는 더 이상 지원하지 않지만, [다음 섹션](#)에서 자세한 사항을 살펴볼 수 있습니다.

파이버 재조정자

'파이버' 재조정자는 스택 재조정자에서 발생한 문제를 해결하고 몇 가지 오래된 오류를 고치는 데에 중점을 두기 위해 고안되었습니다. 해당 재조정자는 React 16부터 기본적으로 사용됩니다.

해당 재조정자의 목표는 다음과 같습니다.

- 중단 불가능한 작업을 청크로 분할하는 기능
- 진행 중인 작업의 우선 순위 지정, 재배치 및 재사용에 관한 기능
- React의 레이아웃을 지원하기 위한 계층별 배치 순서 처리 기능
- `render()` 함수로부터의 다중 요소 반환 기능
- 여러 경계에 대한 향상된 지원

React 파이버 구조에 대해 [여기](#)와 [여기](#)를 통해 더 자세히 확인할 수 있습니다. 파이버 재조정자는 React 16과 함께 제공되지만, 비동기 기능은 아직 기본적으로 활성화되지 않습니다.

해당 소스 코드는 `packages/react-reconciler`에서 확인할 수 있습니다.

이벤트 시스템

React는 네이티브 이벤트 위에 레이어를 구현하여 크로스 브라우저 차이를 제거했습니다. 해당 코드는 `packages/react-dom/src/events`에서 확인할 수 있습니다.

해당 코드에 대한 상세한 설명은 다음의 [영상](#) (66분)을 참고하세요.

다음 내용은?

[다음 섹션](#)을 읽고 재조정자에 대한 pre-React 16 구현에 대해 더 자세하게 배워보세요. 새로운 재조정자에 대한 설명은 아직 작성되지 않았습니다.

[Go to TOC](#)

Components와 Props

컴포넌트를 통해 UI를 재사용 가능한 개별적인 여러 조각으로 나누고, 각 조각을 개별적으로 살펴볼 수 있습니다. 이 페이지에서는 컴포넌트의 개념을 소개합니다. [자세한 컴포넌트 API 레퍼런스는 여기](#)에서 확인할 수 있습니다.

개념적으로 컴포넌트는 JavaScript 함수와 유사합니다. "props"라고 하는 임의의 입력을 받은 후, 화면에 어떻게 표시되는지를 기술하는 React 엘리먼트를 반환합니다.

함수 컴포넌트와 클래스 컴포넌트

컴포넌트를 정의하는 가장 간단한 방법은 JavaScript 함수를 작성하는 것입니다.

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

이 함수는 데이터를 가진 하나의 "props" (props는 속성을 나타내는 데이터입니다) 객체 인자를 받은 후 React 엘리먼트를 반환하므로 유효한 React 컴포넌트입니다. 이러한 컴포넌트는 JavaScript 함수이기 때문에 말 그대로 "함수 컴포넌트"라고 호칭합니다.

또한 [ES6 class](#)를 사용하여 컴포넌트를 정의할 수 있습니다.

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

React의 관점에서 볼 때 위 두 가지 유형의 컴포넌트는 동일합니다.

`class`는 몇 가지 추가 기능이 있으며 이에 대해서는 [다음 장](#)에서 설명합니다. 그때까지는 간결성을 위해 함수 컴포넌트를 사용하겠습니다. 함수 컴포넌트와 클래스 컴포넌트 둘 다 몇 가지 추가 기능이 있으며 이에 대해서는 [다음 장](#)에서 설명합니다.

컴포넌트 렌더링

이전까지는 DOM 태그만을 사용해 React 엘리먼트를 나타냈습니다.

```
const element = <div />;
```

React 엘리먼트는 사용자 정의 컴포넌트로도 나타낼 수 있습니다.

```
const element = <Welcome name="Sara" />;
```

React가 사용자 정의 컴포넌트로 작성한 엘리먼트를 발견하면 JSX 어트리뷰트와 자식을 해당 컴포넌트에 단일 객체로 전달합니다. 이 객체를 "props"라고 합니다.

다음은 페이지에 "Hello, Sara"를 렌더링하는 예시입니다.

```

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

const root = ReactDOM.createRoot(document.getElementById('root'));
const element = <Welcome name="Sara" />;
root.render(element);

```

[CodePen에서 시험해보기](#)

이 예시에서는 다음과 같은 일들이 일어납니다.

1. `<Welcome name="Sara" />` 엘리먼트로 `root.render()` 를 호출합니다.
2. React는 `{name: 'Sara'}` 를 props로 하여 `Welcome` 컴포넌트를 호출합니다.
3. `Welcome` 컴포넌트는 결과적으로 `<h1>Hello, Sara</h1>` 엘리먼트를 반환합니다.
4. React DOM은 `<h1>Hello, Sara</h1>` 엘리먼트와 일치하도록 DOM을 효율적으로 업데이트합니다.

주의: 컴포넌트의 이름은 항상 대문자로 시작합니다.

React는 소문자로 시작하는 컴포넌트를 DOM 태그로 처리합니다. 예를 들어 `<div />` 는 HTML div 태그를 나타내지만, `<welcome />` 은 컴포넌트를 나타내며 범위 안에 `welcome` 이 있어야 합니다.

이 규칙에 대한 자세한 내용은 [여기](#)에서 확인할 수 있습니다.

컴포넌트 합성

컴포넌트는 자신의 출력에 다른 컴포넌트를 참조할 수 있습니다. 이는 모든 세부 단계에서 동일한 추상 컴포넌트를 사용할 수 있음을 의미합니다. React 앱에서는 버튼, 폼, 다이얼로그, 화면 등의 모든 것들이 흔히 컴포넌트로 표현됩니다.

예를 들어 `Welcome` 을 여러 번 렌더링하는 `App` 컴포넌트를 만들 수 있습니다.

```

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

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

```

[CodePen에서 시험해보기](#)

일반적으로 새 React 앱은 최상위에 단일 `App` 컴포넌트를 가지고 있습니다. 하지만 기존 앱에 React를 통합하는 경우에는 `Button` 과 같은 작은 컴포넌트부터 시작해서 뷰 계층의 상단으로 올라가면서 점진적으로 작업해야 할 수 있습니다.

컴포넌트 추출

컴포넌트를 여러 개의 작은 컴포넌트로 나누는 것을 두려워하지 마세요.

다음 `Comment` 컴포넌트를 살펴봅시다.

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

[CodePen에서 시험해보기](#)

이 컴포넌트는 `author` (객체), `text` (문자열) 및 `date` (날짜)를 `props`로 받은 후 소셜 미디어 웹 사이트의 코멘트를 나타냅니다.

이 컴포넌트는 구성요소들이 모두 중첩 구조로 이루어져 있어서 변경하기 어려울 수 있으며, 각 구성요소를 개별적으로 재사용하기도 힘듭니다. 이 컴포넌트에서 몇 가지 컴포넌트를 추출하겠습니다.

먼저 `Avatar` 를 추출하겠습니다.

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

`Avatar` 는 자신이 `Comment` 내에서 렌더링 된다는 것을 알 필요가 없습니다. 따라서 `props`의 이름을 `author`에서 더욱 일반화된 `user`로 변경하였습니다.

`props`의 이름은 사용될 `context`가 아닌 컴포넌트 자체의 관점에서 짓는 것을 권장합니다.

아래 `Comment` 가 살짝 단순해졌습니다.

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
```

```

        <Avatar user={props.author} />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}

```

다음으로 `Avatar` 옆에 사용자의 이름을 렌더링하는 `UserInfo` 컴포넌트를 추출하겠습니다.

```

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

```

`Comment` 가 더욱 단순해졌습니다.

```

function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}

```

[CodePen에서 시험해보기](#)

처음에는 컴포넌트를 추출하는 작업이 지루해 보일 수 있습니다. 하지만 재사용 가능한 컴포넌트를 만들어 놓는 것은 더 큰 앱에서 작업할 때 두각을 나타냅니다. UI 일부가 여러 번 사용되거나 (`Button`, `Panel`, `Avatar`), UI 일부가 자체적으로 복잡한 (`App`, `FeedStory`, `Comment`) 경우에는 별도의 컴포넌트로 만드는 게 좋습니다.

props는 읽기 전용입니다.

함수 컴포넌트나 클래스 컴포넌트 모두 컴포넌트의 자체 `props`를 수정해서는 안 됩니다. 다음 `sum` 함수를 살펴봅시다.

```

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

```

이런 함수들은 [순수 함수](#)라고 호칭합니다. 입력값을 바꾸려 하지 않고 항상 동일한 입력값에 대해 동일한 결과를 반환하기 때문입니다.

반면에 다음 함수는 자신의 입력값을 변경하기 때문에 순수 함수가 아닙니다.

```
function withdraw(account, amount) {  
  account.total -= amount;  
}
```

React는 매우 유연하지만 한 가지 엄격한 규칙이 있습니다.

모든 React 컴포넌트는 자신의 props를 다룰 때 반드시 순수 함수처럼 동작해야 합니다.

물론 애플리케이션 UI는 동적이며 시간에 따라 변합니다. [다음 장](#)에서는 "state"라는 새로운 개념을 소개합니다. React 컴포넌트는 state를 통해 위 규칙을 위반하지 않고 사용자 액션, 네트워크 응답 및 다른 요소에 대한 응답으로 시간에 따라 자신의 출력값을 변경할 수 있습니다.

[Go to TOC](#)

합성 (Composition) vs 상속 (Inheritance)

React는 강력한 합성 모델을 가지고 있으며, 상속 대신 합성을 사용하여 컴포넌트 간에 코드를 재사용하는 것이 좋습니다.

이번 문서에서는 React를 처음 접한 개발자들이 종종 상속으로 인해 부딪히는 몇 가지 문제들과 합성을 통해 이러한 문제를 해결하는 방법을 살펴볼 것입니다.

컴포넌트에서 다른 컴포넌트를 담기

어떤 컴포넌트들은 어떤 자식 엘리먼트가 들어올 지 미리 예상할 수 없는 경우가 있습니다. 범용적인 '박스' 역할을 하는 `Sidebar` 혹은 `Dialog`와 같은 컴포넌트에서 특히 자주 볼 수 있습니다.

이러한 컴포넌트에서는 특수한 `children` prop을 사용하여 자식 엘리먼트를 출력에 그대로 전달하는 것이 좋습니다.

```
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}
```

이러한 방식으로 다른 컴포넌트에서 JSX를 중첩하여 임의의 자식을 전달할 수 있습니다.

```
function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        Welcome
      </h1>
      <p className="Dialog-message">
        Thank you for visiting our spacecraft!
      </p>
    </FancyBorder>
  );
}
```

[CodePen에서 실행하기](#)

`<FancyBorder>` JSX 태그 안에 있는 것들이 `FancyBorder` 컴포넌트의 `children` prop으로 전달됩니다. `FancyBorder`는 `{props.children}` 을 `<div>` 안에 렌더링하므로 전달된 엘리먼트들이 최종 출력됩니다.

흔하지 않지만 종종 컴포넌트에 여러 개의 "구멍"이 필요할 수도 있습니다. 이런 경우에는 `children` 대신 자신만의 고유한 방식을 적용할 수도 있습니다.

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
      </div>
      <div className="SplitPane-right">
        {props.right}
      </div>
    </div>
  );
}
```

```

        </div>
      </div>
    );
}

function App() {
  return (
    <SplitPane
      left={
        <Contacts />
      }
      right={
        <Chat />
      } />
    );
}

```

[CodePen에서 실행하기](#)

<Contacts /> 와 <Chat /> 같은 React 엘리먼트는 단지 객체이기 때문에 다른 데이터처럼 prop으로 전달할 수 있습니다. 이러한 접근은 다른 라이브러리의 "슬롯 (slots)"과 비슷해보이지만 React에서 prop으로 전달할 수 있는 것에는 제한이 없습니다.

특수화

때로는 어떤 컴포넌트의 "특수한 경우"인 컴포넌트를 고려해야 하는 경우가 있습니다. 예를 들어, `WelcomeDialog` 는 `Dialog` 의 특수한 경우라고 할 수 있습니다.

React에서는 이 역시 합성을 통해 해결할 수 있습니다. 더 "구체적인" 컴포넌트가 "일반적인" 컴포넌트를 렌더링하고 props를 통해 내용을 구성합니다.

```

function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
    </FancyBorder>
  );
}

function WelcomeDialog() {
  return (
    <Dialog
      title="Welcome"
      message="Thank you for visiting our spacecraft!" />
  );
}

```

[CodePen에서 실행하기](#)

합성은 클래스로 정의된 컴포넌트에서도 동일하게 적용됩니다.

```

function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
      {props.children}
    </FancyBorder>
  );
}

class SignUpDialog extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.handleSignUp = this.handleSignUp.bind(this);
    this.state = {login: ''};
  }

  render() {
    return (
      <Dialog title="Mars Exploration Program"
        message="How should we refer to you?">
        <input value={this.state.login}
          onChange={this.handleChange} />
        <button onClick={this.handleSignUp}>
          Sign Me Up!
        </button>
      </Dialog>
    );
  }

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

  handleSignUp() {
    alert(`Welcome aboard, ${this.state.login}!`);
  }
}

```

[CodePen에서 실행하기](#)

그렇다면 상속은?

Facebook에서는 수천 개의 React 컴포넌트를 사용하지만, 컴포넌트를 상속 계층 구조로 작성은 권장할 만한 사례를 아직 찾지 못했습니다.

props와 합성은 명시적이고 안전한 방법으로 컴포넌트의 모양과 동작을 커스터마이징하는데 필요한 모든 유연성을 제공합니다. 컴포넌트가 원시 타입의 값, React 엘리먼트 혹은 함수 등 어떠한 props도 받을 수 있다는 것을 기억하세요.

UI가 아닌 기능을 여러 컴포넌트에서 재사용하기를 원한다면, 별도의 JavaScript 모듈로 분리하는 것이 좋습니다. 컴포넌트에서 해당 함수, 객체, 클래스 등을 import 하여 사용할 수 있습니다. 상속받을 필요 없이 말이죠.

[Go to TOC](#)

조건부 렌더링

React에서는 원하는 동작을 캡슐화하는 컴포넌트를 만들 수 있습니다. 이렇게 하면 애플리케이션의 상태에 따라서 컴포넌트 중 몇 개만을 렌더링할 수 있습니다.

React에서 조건부 렌더링은 JavaScript에서의 조건 처리와 같이 동작합니다. `if` 나 `조건부 연산자` 와 같은 JavaScript 연산자를 현재 상태를 나타내는 엘리먼트를 만드는 데에 사용하세요. 그러면 React는 현재 상태에 맞게 UI를 업데이트할 것입니다.

아래 두 컴포넌트가 있다고 가정해 봅시다.

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

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

이제 사용자의 로그인 상태에 맞게 위 컴포넌트 중 하나를 보여주는 `Greeting` 컴포넌트를 만듭니다.

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

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

[CodePen에서 실행하기](#)

이 예시는 `isLoggedIn` prop에 따라서 다른 인사말을 렌더링 합니다.

엘리먼트 변수

엘리먼트를 저장하기 위해 변수를 사용할 수 있습니다. 출력의 다른 부분은 변하지 않은 채로 컴포넌트의 일부를 조건부로 렌더링 할 수 있습니다.

로그아웃과 로그인 버튼을 나타내는 두 컴포넌트가 있다고 가정해 보세요.

```
function LoginButton(props) {
  return (
    <button onClick={props.onClick}>
      Login
    </button>
  );
}

function LogoutButton(props) {
  return (
    <button onClick={props.onClick}>
```

```

        Logout
      </button>
    );
}

```

아래의 예시에서는 `LoginControl`이라는 유상태 컴포넌트를 만들 것입니다.

이 컴포넌트는 현재 상태에 맞게 `<LoginButton />`이나 `<LogoutButton />`을 렌더링합니다. 또한 이전 예시에서의 `<Greeting />`도 함께 렌더링합니다.

```

class LoginControl extends React.Component {
  constructor(props) {
    super(props);
    this.handleLoginClick = this.handleLoginClick.bind(this);
    this.handleLogoutClick = this.handleLogoutClick.bind(this);
    this.state = {isLoggedIn: false};
  }

  handleLoginClick() {
    this.setState({isLoggedIn: true});
  }

  handleLogoutClick() {
    this.setState({isLoggedIn: false});
  }

  render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;

    if (isLoggedIn) {
      button = <LogoutButton onClick={this.handleLogoutClick} />;
    } else {
      button = <LoginButton onClick={this.handleLoginClick} />;
    }

    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
  }
}

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

```

[CodePen에서 실행하기](#)

변수를 선언하고 `if` 를 사용해서 조건부로 렌더링 하는 것은 좋은 방법이지만 더 짧은 구문을 사용하고 싶을 때가 있을 수 있습니다. 여러 조건을 JSX 안에서 인라인(inline)으로 처리할 방법 몇 가지를 아래에서 소개하겠습니다.

논리 `&&` 연산자로 If를 인라인으로 표현하기

JSX 안에는 중괄호를 이용해서 표현식을 포함 할 수 있습니다. 그 안에 JavaScript의 논리 연산자 `&&` 를 사용하면 쉽게 엘리먼트를 조건부로 넣을 수 있습니다.

```

function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
        </h2>
      }
    </div>
  );
}

const messages = ['React', 'Re: React', 'Re:Re: React'];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Mailbox unreadMessages={messages} />);

```

CodePen에서 실행하기

JavaScript에서 `true && expression` 은 항상 `expression` 으로 평가되고 `false && expression` 은 항상 `false` 로 평가됩니다.

따라서 `&&` 뒤의 엘리먼트는 조건이 `true` 일때 출력이 됩니다. 조건이 `false` 라면 React는 무시하고 건너뜁니다.

`falsy` 표현식을 반환하면 여전히 `&&` 뒤에 있는 표현식은 건너뛰지만 `falsy` 표현식이 반환된다는 것에 주의해주세요. 아래 예시에서, `<div>0</div>` 이 render 메서드에서 반환됩니다.

```

render() {
  const count = 0;
  return (
    <div>
      {count && <h1>Messages: {count}</h1>}
    </div>
  );
}

```

조건부 연산자로 If-Else 구문 인라인으로 표현하기

엘리먼트를 조건부로 렌더링하는 다른 방법은 조건부 연산자인 `condition ? true: false` 를 사용하는 것입니다.

아래의 예시에서는 짧은 구문을 조건부로 렌더링합니다.

```

render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.
    </div>
  );
}

```

가독성은 좀 떨어지지만, 더 큰 표현식에도 이 구문을 사용할 수 있습니다.

```

render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn
        ? <LogoutButton onClick={this.handleLogoutClick} />
        : <LoginButton onClick={this.handleLoginClick} />
      }
    </div>
  );
}

```

JavaScript와 마찬가지로, 가독성이 좋다고 생각하는 방식을 선택하면 됩니다. 또한 조건이 너무 복잡하다면 [컴포넌트를 분리하기](#) 좋을 때 일 수도 있다는 것을 기억하세요.

컴포넌트가 렌더링하는 것을 막기

가끔 다른 컴포넌트에 의해 렌더링될 때 컴포넌트 자체를 숨기고 싶을 때가 있을 수 있습니다. 이때는 렌더링 결과를 출력하는 대신 `null`을 반환하면 해결할 수 있습니다.

아래의 예시에서는 `<WarningBanner />` 가 `warn` prop의 값에 의해서 렌더링됩니다. prop이 `false`라면 컴포넌트는 렌더링하지 않게 됩니다.

```

function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }

  return (
    <div className="warning">
      Warning!
    </div>
  );
}

class Page extends React.Component {
  constructor(props) {
    super(props);
    this.state = {showWarning: true};
    this.handleToggleClick = this.handleToggleClick.bind(this);
  }

  handleToggleClick() {
    this.setState(state => ({
      showWarning: !state.showWarning
    }));
  }

  render() {
    return (
      <div>
        <WarningBanner warn={this.state.showWarning} />
        <button onClick={this.handleToggleClick}>
          {this.state.showWarning ? 'Hide' : 'Show'}
        </button>
      </div>
    );
  }
}

```

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

[CodePen에서 실행하기](#)

컴포넌트의 `render` 메서드로부터 `null` 을 반환하는 것은 생명주기 메서드 호출에 영향을 주지 않습니다. 그 예로 `componentDidUpdate` 는 계속해서 호출되게 됩니다.

[Go to TOC](#)

Context

context를 이용하면 단계마다 일일이 props를 넘겨주지 않고도 컴포넌트 트리 전체에 데이터를 제공할 수 있습니다.

일반적인 React 애플리케이션에서 데이터는 위에서 아래로 (즉, 부모로부터 자식에게) props를 통해 전달되지만, 애플리케이션 안의 여러 컴포넌트들에 전해줘야 하는 props의 경우 (예를 들면 선호 로케일, UI 테마) 이 과정이 번거로울 수 있습니다. context를 이용하면, 트리 단계마다 명시적으로 props를 넘겨주지 않아도 많은 컴포넌트가 이러한 값을 공유하도록 할 수 있습니다.

- 언제 context를 써야 할까
- context를 사용하기 전에 고려할 것
- API
 - `React.createContext`
 - `Context.Provider`
 - `Class.contextType`
 - `Context.Consumer`
 - `Context.displayName`
- 예시
 - 값이 변하는 context
 - 하위 컴포넌트에서 context 업데이트하기
 - 여러 context 구독하기
- 주의사항
- 예전 API

언제 context를 써야 할까

context는 React 컴포넌트 트리 안에서 전역적(global)이라고 볼 수 있는 데이터를 공유할 수 있도록 고안된 방법입니다. 그러한 데이터로는 현재 로그인한 유저, 테마, 선호하는 언어 등이 있습니다. 예를 들어, 아래의 코드는 버튼 컴포넌트를 꾸미기 위해 테마(theme) props를 명시적으로 넘겨주고 있습니다.

```
embed:context/motivation-problem.js
```

context를 사용하면 중간에 있는 엘리먼트들에게 props를 넘겨주지 않아도 됩니다.

```
embed:context/motivation-solution.js
```

context를 사용하기 전에 고려할 것

context의 주된 용도는 다양한 레벨에 네스팅된 많은 컴포넌트에게 데이터를 전달하는 것입니다. context를 사용하면 컴포넌트를 재사용하기가 어려워지므로 꼭 필요할 때만 쓰세요.

여러 레벨에 걸쳐 props 넘기는 걸 대체하는 데에 context보다 [컴포넌트 합성](#)이 더 간단한 해결책일 수도 있습니다.

예를 들어 여러 단계 아래에 있는 `Link` 와 `Avatar` 컴포넌트에게 `user` 와 `avatarSize`라는 props를 전달해야 하는 `Page` 컴포넌트를 생각해봅시다.

```
<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>
```

실제로 사용되는 곳은 `Avatar` 컴포넌트 뿐인데 `user` 와 `avatarSize` props를 여러 단계에 걸쳐 보내줘야 한다는 게 번거로워 보일 수 있습니다. 게다가 위에서 `Avatar` 컴포넌트로 보내줘야하는 props가 추가된다면 그 또한 중간 레벨에 모두 추가해줘야 합니다.

`Avatar` 컴포넌트 자체를 넘겨주면 **context**를 사용하지 않고 이를 해결할 수 있습니다. 그러면 중간에 있는 컴포넌트들이 `user` 나 `avatarSize`에 대해 전혀 알 필요가 없습니다.

```
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}
```

이렇게 바꾸면 `Link` 와 `Avatar` 컴포넌트가 `user` 와 `avatarSize` props를 쓴다는 걸 알아야 하는 건 가장 위에 있는 `Page` 뿐입니다.

이러한 **제어의 역전(inversion of control)**을 이용하면 넘겨줘야 하는 props의 수는 줄고 최상위 컴포넌트의 제어력은 더 커지기 때문에 더 깔끔한 코드를 쓸 수 있는 경우가 많습니다. 하지만 이러한 역전이 항상 옳은 것은 아닙니다. 복잡한 로직을 상위로 옮기면 이 상위 컴포넌트들은 더 난해해지기 마련이고 하위 컴포넌트들은 필요 이상으로 유연해져야 합니다.

자식으로 둘 수 있는 컴포넌트의 수에 제한은 없습니다. 여러 컴포넌트, 혹은 여러 개로 구분된 "슬롯"을 넘기는 방법에 대해서는 [여기](#)를 참조하세요.

```
function Page(props) {
  const user = props.user;
  const content = <Feed user={user} />;
  const topBar = (
    <NavigationBar>
      <Link href={user.permalink}>
        <Avatar user={user} size={props.avatarSize} />
      </Link>
    </NavigationBar>
  );
  return (
```

```
<PageLayout
  topBar={topBar}
  content={content}
/>
);
}
```

이 패턴을 사용하면 자식 컴포넌트와 직속 부모를 분리(decouple)하는 문제는 대개 해결할 수 있습니다. 더 나아가 `render props`를 이용하면 렌더링 되기 전부터 자식 컴포넌트가 부모 컴포넌트와 소통하게 할 수 있습니다.

하지만 같은 데이터를 트리 안 여러 레벨이 있는 많은 컴포넌트에 주어야 할 때도 있습니다. 이런 데이터 값이 변할 때마다 모든 하위 컴포넌트에게 널리 "방송"하는 것이 context입니다. 흔히 예시로 드는 선호 로케일, 테마, 데이터 캐시 등을 관리하는 데 있어서는 일반적으로 context를 사용하는 게 가장 편리합니다.

API

`React.createContext` [\[#reactcreatecontext\]](#)

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

Context 객체를 만듭니다. Context 객체를 구독하고 있는 컴포넌트를 렌더링할 때 React는 트리 상위에서 가장 가까이 있는 짹이 맞는 `Provider`로부터 현재값을 읽습니다.

`defaultValue` 매개변수는 트리 안에서 적절한 `Provider`를 찾지 못했을 때만 쓰이는 값입니다. 이 기본값은 컴포넌트를 독립적으로 테스트할 때 유용한 값입니다. `Provider`를 통해 `undefined`을 값으로 보낸다고 해도 구독 컴포넌트들이 `defaultValue`를 읽지 않는다는 점에 유의하세요.

`Context.Provider` [\[#contextprovider\]](#)

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

Context 오브젝트에 포함된 React 컴포넌트인 `Provider`는 `context`를 구독하는 컴포넌트들에게 `context`의 변화를 알리는 역할을 합니다.

`Provider` 컴포넌트는 `value` prop을 받아서 이 값을 하위에 있는 컴포넌트에게 전달합니다. 값을 전달받을 수 있는 컴포넌트의 수에 제한은 없습니다. `Provider` 하위에 또 다른 `Provider`를 배치하는 것도 가능하며, 이 경우 하위 `Provider`의 값이 우선시됩니다.

`Provider` 하위에서 `context`를 구독하는 모든 컴포넌트는 `Provider`의 `value` prop가 바뀔 때마다 다시 렌더링 됩니다. `Provider`로부터 하위 `consumer`(`.contextType` 와 `useContext`을 포함한)로의 전파는 `shouldComponentUpdate` 메서드가 적용되지 않으므로, 상위 컴포넌트가 업데이트를 건너 뛰더라도 `consumer`가 업데이트됩니다.

`context` 값의 바뀌었는지 여부는 `Object.is` 와 동일한 알고리즘을 사용해 이전 값과 새로운 값을 비교해 측정됩니다.

주의

위와 같은 방식으로 변화를 측정하기 때문에 객체를 `value`로 보내는 경우 다소 문제가 생길 수 있습니다. [주의사항](#)을 참조하세요.

Class.contextType {#classcontexttype}

```
class MyClass extends React.Component {
  componentDidMount() {
    let value = this.context;
    /* MyContext의 값을 이용한 코드 */
  }
  componentDidUpdate() {
    let value = this.context;
    /* ... */
  }
  componentWillUnmount() {
    let value = this.context;
    /* ... */
  }
  render() {
    let value = this.context;
    /* ... */
  }
}
MyClass.contextType = MyContext;
```

`React.createContext()`로 생성한 Context 객체를 원하는 클래스의 `contextType` 프로퍼티로 지정할 수 있습니다. 이 프로퍼티를 활용해 클래스 안에서 `this.context`를 이용해 해당 Context의 가장 가까운 Provider를 찾아 그 값을 읽을 수 있게됩니다. 이 값은 `render`를 포함한 모든 컴포넌트 생명주기 매서드에서 사용할 수 있습니다.

주의

이 API를 사용하면 하나의 `context`만 구독할 수 있습니다. 여러 `context`를 구독하기 위해서는 [여러 context 구독하기](#)를 참조하세요.

실험적 기능인 [public class fields syntax](#)를 사용하고 있다면 정적 클래스 프로퍼티로 `contextType`을 지정할 수 있습니다.

```
class MyClass extends React.Component {
  static contextType = MyContext;
  render() {
    let value = this.context;
    /* context 값을 이용한 렌더링 */
  }
}
```

Context.Consumer {#contextconsumer}

```
<MyContext.Consumer>
  {value => /* context 값을 이용한 렌더링 */}
</MyContext.Consumer>
```

`context` 변화를 구독하는 React 컴포넌트입니다. 이 컴포넌트를 사용하면 [함수 컴포넌트](#)안에서 `context`를 구독할 수 있습니다.

`Context.Consumer`의 자식은 [함수](#)여야합니다. 이 함수는 `context`의 현재값을 받고 React 노드를 반환합니다. 이 함수가 받는 `value` 매개변수 값은 해당 `context`의 Provider 중 상위 트리에서 가장 가까운 Provider의 `value` prop과 동일합니다. 상위에 Provider가 없다면 `value` 매개변수 값은 `createContext()`에 보냈던 `defaultValue`와 동일할 것입니다.

주의

함수를 자식으로 받는 패턴에 대해서는 [render props](#)을 참조하세요.

Context.displayName {#contextdisplayname}

Context 객체는 `displayName` 문자열 속성을 설정할 수 있습니다. React 개발자 도구는 이 문자열을 사용해서 context를 어떻게 보여줄지 결정합니다.

예를 들어, 아래 컴포넌트는 개발자 도구에 `MyDisplayName`로 표시됩니다.

```
const MyContext = React.createContext(/* some value */);
MyContext.displayName = 'MyDisplayName';

<MyContext.Provider> // "MyDisplayName.Provider" in DevTools
<MyContext.Consumer> // "MyDisplayName.Consumer" in DevTools
```

예시**값이 변하는 context**

`theme` 값이 변하는 좀 더 복잡한 예시입니다.

theme-context.js [embed:context/theme-detailed-theme-context.js](#)

themed-button.js [embed:context/theme-detailed-themed-button.js](#)

app.js [embed:context/theme-detailed-app.js](#)

하위 컴포넌트에서 context 업데이트하기

컴포넌트 트리 하위 깊숙이 있는 컴포넌트에서 context를 업데이트 해야 할 때가 종종 있습니다. 그럴 때는 context를 통해 매서드를 보내면 됩니다.

theme-context.js [embed:context/updating-nested-context-context.js](#)

theme-toggler-button.js [embed:context/updating-nested-context-theme-toggler-button.js](#)

app.js [embed:context/updating-nested-context-app.js](#)

여러 context 구독하기

각 context마다 Consumer를 개별 노드로 만들게 설계되어 있는데, 이것은 context 변화로 인해 다시 렌더링하는 과정을 빠르게 유지하기 위함입니다.

[embed:context/multiple-contexts.js](#)

둘 이상의 context 값이 함께 쓰이는 경우가 많다면 그 값을 한 번에 받는 render prop 컴포넌트를 만드는 것을 고려해보세요.

주의사항

다시 렌더링할지 여부를 정할 때 참조(reference)를 확인하기 때문에, Provider의 부모가 렌더링 될 때마다 불필요하게 하위 컴포넌트가 다시 렌더링 되는 문제가 생길 수도 있습니다. 예를 들어 아래 코드는 `value` 가 바뀔 때마다 매번 새로운 객체가 생성되므로 Provider가 렌더링 될 때마다 그 하위에서 구독하고 있는 컴포넌트 모두가 다시 렌더링 될 것입니다.

```
embed:context/reference-caveats-problem.js
```

이를 피하기 위해서는 값을 부모의 `state`로 끌어올리세요.

```
embed:context/reference-caveats-solution.js
```

예전 API

주의

이전 버전의 React에 실험적인 단계의 context API가 존재한 적이 있습니다. 예전 API는 모든 16.x 버전에서 지원될 예정이지만 새로운 API로 옮길 것을 권장합니다. 다음 메이저 배포에서 예전 API는 삭제될 것입니다. 예전 API 문서는 [여기](#)에 있습니다.

[Go to TOC](#)

새로운 React 앱 만들기

최고의 사용자와 개발자 경험을 위해 통합된 툴체인을 사용하세요.

이 페이지에서는 도움이 되는 몇 가지 인기 있는 React 툴체인을 소개합니다.

- 많은 파일과 컴포넌트 스케일링
- 서드 파티 npm 라이브러리 사용
- 일반적인 실수를 조기에 발견
- CSS와 JS를 실시간으로 편집
- 프로덕션 코드 최적화

이 페이지에서 추천하는 툴체인은 시작하는데, 별도의 환경설정이 필요 없습니다.

툴체인이 필요하지 않을 수 있습니다.

위에서 설명한 문제를 경험하지 못했거나 아직 JavaScript 도구를 사용하는 것이 편하지 않다면, [React를 HTML 페이지에서 일반적인 <script> 태그에 추가하거나 JSX를 고려하세요.](#)

이 방법이 제일 쉽게 **React**를 이미 만들어진 웹사이트에 추가하는 방법입니다. 그리고 언제나 도움이 될 것 같으면 더 많은 툴체인을 추가할 수가 있습니다.

추천 툴체인

React 팀의 추천 방법은 아래와 같습니다

- **React**를 배우고 있거나 아니면 새로운 [싱글 페이지 앱](#)을 만들고 싶다면 [Create React App](#).
- 서버 렌더링 **Node.js** 웹사이트를 만들고 있다면 [Next.js](#)를 시도해보세요..
- 고정적인 콘텐츠 지향적 웹사이트를 만들고 있다면 [Gatsby](#)를 시도해보세요..
- 컴포넌트 라이브러리 혹은 이미 있는 코드 베이스에 통합을 한다면 [더 유연한 툴체인](#).

Create React App

[Create React App](#)은 **React** 배우기에 간편한 환경입니다. 그리고 시작하기에 최고의 방법은 [새로운 싱글 페이지 애플리케이션](#)입니다.

이것은 개발 환경을 설정하고, 최신 JavaScript를 사용하게 해주며, 좋은 개발 경험과 프로덕션 앱 최적화를 해줍니다. [Node 14.0.0 혹은 상위 버전 및 npm 5.6 혹은 상위 버전](#)이 필요합니다. 새로운 프로젝트를 만들기 위해 아래의 명령어를 실행합니다.

```
npx create-react-app my-app
cd my-app
npm start
```

주의

첫 번째 줄의 'npx'는 실수가 아니며 [npm 5.2+ 버전의 패키지 실행 도구입니다.](#)

Create React App 은 백 앤드 로직이나 데이터베이스를 제어할 수 없습니다. Create React App 은 프런트 앤드 빌드 파이프라인만 생성하기 때문에 원하는 어떤 백엔드와도 함께 사용할 수 있습니다. Create React App은 Babel이나 webpack같은 build 도구를 사용하나, 설정 없이도 동작합니다.

프로덕션을 배포할 준비가 되었을 때, `npm run build` 를 실행하면 build 폴더 안에 제작한 앱의 최적화된 Build를 생성합니다. [README](#) 나 [사용자 가이드](#)에서 더 자세한 사항을 볼 수 있습니다.

Next.js

[Next.js](#)는 인기 있는 경량의 프레임워크로 React로 만들어진 스태틱 서버 렌더링 애플리케이션입니다. 기본적으로 [스타일링과 라우팅 해결책](#)을 가지고 있으며, 사용자가 Node.js를 서버 환경으로 사용하고 있다고 생각합니다.

Next.js를 [정식 가이드](#)를 보면서 배워보세요.

Gatsby

[Gatsby](#)는 정적 웹사이트를 React로 만들기에는 최고의 방법입니다. React 컴포넌트를 사용하게 해주지만 미리 렌더링 된 HTML과 CSS를 사용하여 가장 빠르게 로드됩니다.

Gatsby를 [정식 가이드](#)와 [스타터 키트](#)를 보면서 배워보세요

더 유연한 툴체인

밑에 있는 툴체인은 조금 더 많은 선택과 다르기 쉬운 옵션입니다. 숙련된 사용자들에게 추천합니다.

- [Neutrino](#)는 [webpack](#)의 장점과 React의 단순함과 미리 설정된 앱과 [컴포넌트](#)를 합친 것입니다.
- [Nx](#)는 폴스택 모노레포 개발을 위한 도구이며, React, Next.js, Express 등을 기본적으로 지원합니다.
- [Parcel](#)은 React와 함께 사용할 수 있고 빠르고 설정이 필요 없는 웹 애플리케이션 bundler입니다.
- [Razzle](#)은 서버 렌더링 프레임워크며 설정이 필요 없지만, Next.js보다 다루기 쉽습니다.

툴체인을 직접 만들기

JavaScript build 툴체인은 주로 아래와 같이 구성되어있습니다

- [Yarn](#) 혹은 [npm](#)같은 package 매니저는 서드 파티 패키지의 방대한 생태계를 활용할 수 있게 하며, 쉽게 설치하고 업데이트할 수 있게 합니다.
- [webpack](#) 아니면 [Parcel](#) 같은 **bundler**는 코드를 모듈방식으로 작성할 수 있게 하고 이를 작은 package로 묶어서 로딩 시간을 최적화할 수 있습니다.
- [Babel](#) 같은 **컴파일러**는 최신 JavaScript 코드를 구형 브라우저에도 실행되게 도와줍니다.

만든 JavaScript 툴체인을 원하신다면, [이 가이드를 보세요](#).

커스텀 툴체인이 제대로 설정되어 있는지 [잊지 마세요](#).

[Go to TOC](#)

Cross-origin Errors

주의:

해당 부분은 오직 리액트 개발 모드에서만 적용됩니다. 배포 모드에서 오류 처리는 일반 try/catch 구문으로 수행됩니다.

개발 모드에서 리액트는 전역 `오류` 이벤트 핸들러를 사용하여 브라우저 DevTools의 "예외 시 일시 중지" 동작을 유지해줍니다. 또한 개발자 콘솔에 오류를 기록합니다.

다른 출처로 인한 오류가 발생한다면 브라우저는 해당 세부 정보를 숨기고 리액트는 오리지널 오류 메시지를 기록할 수가 없습니다. 왜냐하면 민감한 정보가 노출되는 것을 방지하기 위해서 브라우저에서 보안 예방 조치 취하기 때문입니다.

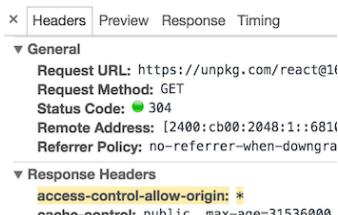
동일 출처 정책으로 오류가 발생하도록 개발/디버깅 프로세스를 단순화할 수 있습니다. 다음은 교차 출처 오류가 발생하는 일반적인 원인과 해결 방법입니다.

CDN

CDN에서 리액트(혹은 오류를 발생시킬 수 있는 다른 라이브러리)를 로딩할 때, `<script>` 태그에 `crossorigin` 어트리뷰트를 추가하세요:

```
<script crossorigin src="..."></script>
```

또한 CDN이 `Access-Control-Allow-Origin: *` HTTP 헤더로 응답하는지 확인합니다.



Access-Control-Allow-Origin: *

Webpack

소스 맵

일부 자바스크립트 번들러는 개발 중에 `eval` 문으로 코드로 감싸져 있을 수 있습니다. (예를 들어 웹팩은 `devtool`이 "eval"이라는 단어가 포함된 값으로 설정된 경우 이 작업을 수행합니다) 이로 인해 크로스-오리진 오류로 처리될 수 있습니다.

웹팩을 사용하는 경우, 이러한 문제를 피하고자 개발 시에 `cheap-module-source-map` 설정하는 것을 추천합니다.

코드 분할

애플리케이션이 여러 번들로 분할된 경우 JSONP를 사용하여 로드될 수 있습니다. 이로 인해 번들링 된 코드가 크로스-오리진 오류로 처리될 수도 있습니다.

이러한 문제를 해결하려면, 개발 시에 `crossOriginLoading` 설정을 사용하여 JSON 요청에 대해 생성된 `<script>` 태그에 `crossorigin` 속성을 추가하세요.

Design Principles

우리는 이 문서를 작성함으로써 React는 무엇을 하고 무엇을 하지 않는지, 우리의 개발 철학은 무엇인지에 대해 우리가 어떻게 결정하는지 여러분이 더 잘 알 수 있도록 했습니다. 우리는 커뮤니티에 기여하는 것을 즐거워하기도 하지만 이 원칙들 중 하나 이상을 위반하는 길은 선택하지 않을 것입니다.

주의

이 문서는 React에 대한 깊은 이해가 있음을 전제로 합니다. React 컴포넌트나 애플리케이션이 아니라 *React 자체*에 대한 설계 원칙(design principles)을 설명합니다.

React 소개는 [React로 사고하기](#)를 살펴보세요.

합성

React의 핵심 기능은 컴포넌트의 합성입니다. 컴포넌트는 서로 다른 사람들에 의해 작성되지만 잘 동작해야 합니다. 코드베이스에 변화의 파장을 일으키지 않고 컴포넌트에 기능을 추가할 수 있는 것이 중요합니다.

예를 들어, 컴포넌트를 사용하는 쪽을 변경하지 않고 컴포넌트에 어떤 로컬 state를 도입할 수 있어야 합니다. 마찬가지로, 필요한 경우 어떤 컴포넌트에 초기화 및 해체 코드를 추가할 수 있어야 합니다.

컴포넌트에서 state나 생명주기 메서드를 사용하는 것에 대해 "나쁜 것"은 없습니다. 다른 강력한 기능과 마찬가지로 적당히 사용해야 할 필요가 있지만, 우리는 그것들을 제거할 생각은 없습니다. 오히려 우리는 그것들이 React를 유용하게 만드는 데에 매우 중요한 부분이라고 생각합니다. 장래에 [보다 함수적인 패턴](#)을 사용 가능하게 할지도 모르겠습니다만, 로컬 state와 생명주기 메서드는 모두 그 모델의 일부가 될 것입니다.

컴포넌트는 종종 "단순한 함수"로 묘사되지만 우리의 관점에서는 유용한 것 이상의 것이 필요합니다. React에서 컴포넌트는 구성 가능한 모든 동작을 기술합니다. 그리고 여기에는 렌더링, 생명주기와 state가 포함됩니다. [Relay](#)와 같은 어떤 외부 라이브러리는 데이터 의존성을 기술하는 것과 같은 다른 책임을 컴포넌트에 덧붙입니다. 이런 아이디어들은 어떤 형태로 React로 다시 받아들여질 수도 있습니다.

공통의 추상화

일반적으로 우리는 사용자 영역에서 구현할 수 있는 [기능을 추가하지 않습니다](#). 불필요한 라이브러리 코드로 여러분의 앱을 비대화화하고 싶지는 않습니다. 그렇지만, 여기에는 예외가 있습니다.

예를 들어, React가 로컬 state나 생명주기 메서드를 지원하지 않았다면 사람들은 사용자 정의 추상화를 만들게 될 것입니다. 여러 개의 추상화가 충돌하는 경우 React는 어느 한 쪽의 특성을 강요하거나 이용할 수는 없습니다. 그것은 최소 공통 분모로 작용해야 합니다.

이것이 우리가 React 그 자체에 때때로 기능을 추가하는 이유입니다. 많은 컴포넌트가 호환성이 없거나 비효율적인 방법으로 어떤 기능을 구현한다는 것을 알게 된다면 차라리 그것을 React에 녹여낼지도 모르겠습니다. 하지만 우리는 선뜻 그렇게 하지는 않습니다. 우리가 그렇게 한다면 그것은 추상화 레벨을 올리는 것이 전체 생태계에 이익이 된다는 확신이 있기 때문입니다. State, 생명주기 메서드, 크로스 브라우저의 이벤트 정규화가 좋은 예입니다.

우리는 언제나 커뮤니티와 이러한 개선안을 논의합니다. React 이후 트래커에서 ["big picture"](#) 라벨로 이러한 논의를 찾을 수 있습니다.

해결책

React는 실용적입니다. Facebook에서 작성된 제품의 필요에 의해 발전되었습니다. 함수형 프로그래밍과 같은, 아직은 완전한 주류는 아닌 몇 가지 패러다임에 의해 영향을 받지만, 다양한 기술과 경험을 가진 광범위한 개발자에게 접근 가능하도록 유지하는 것은 프로젝트의 명백한 목표입니다.

우리가 원하지 않는 패턴을 비권장하는 경우는 그것을 비권장하기 전에 존재하는 모든 유스케이스를 고려하고 [대체방법에 대해 커뮤니티에 교육하는 것은](#) 우리의 책임입니다. 앱을 구축하는 데 유용한 어떤 패턴을 선언적 방식으로 표현하기가 아주 어려운 경우는 우리는 [명령적 API를 제공할 것입니다](#). 많은 앱에서 필요한 어떤 것에 대한 최적의 API를 찾아내지 못 한 경우, 가능한 추후에 제거할 수 있고 장래의 개선의 여지가 있는 경우에 한해서 [보통 수준 이하의 임시적인 작업용 API를 제공할 것입니다](#).

안정성

우리는 API 안정성에 가치를 둡니다. Facebook에는 React를 사용한 5만 개 이상의 컴포넌트가 사용됩니다. Twitter와 Airbnb를 포함한 다른 많은 기업들 또한 React의 헤비 유저(heavy user)입니다. 이것이 바로 우리가 공용 API나 behavior의 변경을 꺼리는 이유입니다.

그러나 "아무 것도 변하지 않는다"는 의미로 안정성을 과대평가하고 있다고 생각합니다. 그것은 곧 정체로 바뀝니다. 대신, 우리는 "프로덕션(production) 환경에서 자주 사용되는 무엇인가가 변경되었을 때는 명확한 (그리고 가능하면 자동화 된) 마이그레이션 방법이 있다"는 의미에서의 안정성을 선호합니다.

어떤 패턴을 비권장하는 경우 Facebook 내부에서 그 사용법을 조사하고 비권장 경고를 추가합니다. 그럼으로써 변화의 영향도를 평가할 수 있습니다. 변경이 너무 시기상조이거나 변경에 대한 준비가 될 때까지 코드베이스를 가져가는 데에 있어 전략적으로 조금 더 생각할 필요가 있는 경우, 우리는 그 변경을 되돌릴 수 있습니다.

변경이 그닥 파괴적이지 않고 모든 유스케이스에서 마이그레이션 전략이 실행 가능하다고 확신하는 경우 오픈소스 커뮤니티에 비추천 경고를 공개합니다. 우리는 Facebook 이외의 많은 React 사용자와 긴밀하게 연락하고 있으며 인기있는 오픈소스 프로젝트를 모니터링하고 비추천 경고 수정을 안내하고 있습니다.

Facebook의 React 코드베이스의 순수 사이즈를 생각해 볼 때 사내 마이그레이션이 성공하는 것은 다른 기업들에서도 문제 없을 것이라는 좋은 지표입니다. 그럼에도 불구하고 가끔 사람들은 우리가 생각하지 못 한 새로운 유스케이스를 지적하고, 우리는 해결책을 추가하거나 접근법을 다시 생각합니다.

정당한 이유 없이는 우리는 어떤 것도 비권장하지 않습니다. 때때로 비권장 경고가 좌절을 야기함을 인정하기는 하지만 비권장은 우리와 커뮤니티의 많은 사람들이 가치있다고 생각하는 개선과 새로운 기능에 대해 깨끗하게 길을 닦기 때문에 우리는 그것을 추가합니다.

예를 들어, 우리는 React 15.2.0에서는 [알려지지 않은 DOM props에 대한 경고](#)를 추가합니다. 이것은 많은 프로젝트에 영향이 있습니다. 그렇지만, React에 [사용자 attributes](#)의 지원을 React에 도입할 수 있도록 이 경고를 수정하는 것은 중요합니다. 우리가 추가하는 모든 비권장에는 이와 같은 이유가 있습니다.

우리가 비권장 경고를 추가하면 현재의 메이저 버전의 나머지 부분에서는 경고를 남긴 채 유지하고 [다음 메이저 버전에서는 동작을 변경합니다](#). 반복적인 수작업이 많은 경우에는 변경의 상당 부분을 자동화하는 [codemod](#) 스크립트를 공개합니다. Codemod를 사용하면 대규모 코드베이스도 지체없이 이행을 진행할 수 있고, 우리는 이것을 사용하기를 독려합니다.

여러분은 우리가 [react-codemod](#) 저장소에 공개한 codemod를 찾을 수 있습니다.

상호운용성

우리는 기존 시스템과의 상호운용성과 점진적인 도입에 높은 가치를 두고 있습니다. Facebook은 거대한 non-React 코드베이스를 가지고 있습니다. Facebook 웹사이트는 XHP라고 부르는 서버사이드 컴포넌트 시스템과 React 이전부터 개발된 내부 UI 라이브러리, 그리고 React 그 자체를 조합해서 사용하고 있습니다. 우리에게는 어떤 프로덕트 팀도 내기하듯이 코드를 다시 작성하기보다는 [작은 기능에 React를 사용하여 시작할 수 있다](#)는 것이 중요합니다.

이것이 React가 변경 가능한 모델을 다루기 위한 해결책(escape hatches)을 제시하고, 다른 UI 라이브러리들과 함께 잘 동작하도록 노력하는 이유입니다. 기존의 명령형 UI를 선언형 컴포넌트로 감싸는 것도, 또 그 반대도 가능합니다. 이것은 점진적인 도입을 위해서 반드시 필요한 것입니다.

스케줄링

컴포넌트가 함수로 기술되어 있어도 React를 사용할 때 그것을 직접 호출하지는 마세요. 모든 컴포넌트는 [무엇을 렌더링할 필요가 있는 것인지에 대한 설명](#)을 반환하고 `<LikeButton>` 과 같은 사용자가 정의한 컴포넌트와 `<div>` 와 같은 플랫폼 고유의 컴포넌트를 모두 포함할 수 있습니다. 미래의 언젠가 `<LikeButton>` 을 풀고 컴포넌트의 렌더링 결과에 따라 재귀적으로 UI 트리를 변경하는 것은 실제로 React의 책임입니다.

이것은 매우 미묘한 차이지만 매우 강력합니다. 여러분이 그 컴포넌트 함수를 호출하지 않고 React가 호출하는데, 이것은 필요한 경우 자연할 수 있는 권한이 React에 있다는 것을 의미합니다. 현재의 구현에서는 React는 트리를 재귀적으로 조사하고 단일 Tick 동안에 전체 갱신된 트리의 렌더링 함수를 호출합니다.

이것은 React 설계의 공통된 테마입니다. 어떤 인기있는 라이브러리는 새 데이터가 사용 가능해졌을 때 계산이 수행되는 "push" 접근 방식을 구현합니다. 그러나 React는 필요할 때까지 계산을 지연할 수 있는 "pull" 접근 방식을 택합니다.

React는 범용적인 데이터 처리 라이브러리가 아닙니다. 그것은 사용자 인터페이스를 만들기 위한 라이브러리입니다. 우리는 React가 앱에서 어떤 계산이 지금 필요하고 어떤 계산이 지금 필요하지 않은지 알 수 있는 특별한 위치를 점유했다고 생각합니다.

무엇인가가 화면 밖에 있다면 우리는 그것과 관련된 어떤 로직을 지연시킬 수 있습니다. 데이터가 프레임 속도보다 좀 더 빠르게 도착하는 경우 통합 및 일괄 업데이트를 할 수 있습니다. (네트워크에서 갓 로드된 새로운 컨텐츠의 렌더링과 같은) 중요도가 낮은 백그라운드 작업 보다 (버튼 클릭에 의한 애니메이션과 같은) 사용자 인터렉션을 우선할 수 있습니다.

명확하게 하자면, 우리는 지금 이것을 이용하고 있지 않습니다. 그러나 이런 일을 할 수 있는 자유는 왜 우리가 스케줄링에 대한 통제권을 선호하는지, 왜 `setState()` 는 비동기적인지에 대한 이유입니다. 개념적으로 우리는 이것을 "스케줄링 업데이트"라고 생각합니다.

[Functional Reactive Programming](#)의 몇 가지 변형에서 흔하게 볼 수 있는 "push" 기반 패러다임으로 사용자가 직접 뷰를 구성한다면 스케줄링에 대한 제어를 얻는 것이 어려워질 것입니다. 우리는 "글루(glue)" 코드를 관리하고 싶습니다.

React의 핵심 목표는 React로 되돌아 가기 전에 수행하는 사용자 코드의 양을 최소화하는 것입니다. 이것은 React가 UI에 대해서 알고 있는 지에 따라 작업을 스케줄하고 청크로 분할하는 능력을 유지한다는 것을 보증합니다.

React는 완전히 "반응적(reactive)"이고 싶지 않았기 때문에 React는 "Schedule"로 불리웠어야 한다는 팀내 농담이 있습니다.

개발자 경험

좋은 개발자 경험을 제공하는 것은 우리에게 중요합니다.

예를 들어, 우리는 Chrome과 Firefox에서 React 컴포넌트 트리를 살펴볼 수 있는 [React DevTools](#)를 유지합니다. 이것은 Facebook 엔지니어와 커뮤니티 모두에게 커다란 생산성 향상을 가져왔다고 들었습니다.

또한, 도움이 될 만한 개발자 경고를 제공하기 위해 우리는 더 노력하고 있습니다. 예를 들어, React는 개발 중에 브라우저가 이해할 수 없는 방법으로 태그를 중첩하거나 API에서 일반적인 오타를 작성한 경우 이에 대해 경고합니다. 개발자 경고와 이와 관련된 검사는 React 개발자 버전이 프로덕션 버전보다 조금 더 느린 주요한 이유입니다.

Facebook 내부에서 볼 수 있는 사용성 패턴은 보편적인 실수가 무엇이고 어떻게 방지할 수 있는지 이해하는 데 도움이 됩니다. 새 기능을 추가할 때 우리는 일반적인 실수를 예상하고 그것을 경고하려고 노력합니다.

우리는 항상 개발자 경험을 향상할 수 있는 방법을 찾고 있습니다. 우리는 여러분의 제안을 듣고 여러분의 기여를 수용하여 더 나은 개발자 경험을 만들 수 있기를 바랍니다.

디버깅

문제가 발생했을 때 코드베이스로 실수의 원인을 추적할 수 있는 표식을 만드는 것은 중요합니다. React에서의 props와 state가 이러한 표식입니다.

화면에서 무엇인가 문제가 발생한 경우 React DevTools를 열고 렌더링을 담당한 컴포넌트를 찾은 후 props와 state가 올바른지 확인할 수 있습니다. 그렇다면 컴포넌트의 `render()` 함수 안에 문제가 있거나 `render()`를 호출하는 어떤 함수에 문제가 있다는 것을 알 수 있습니다. 이제 문제가 분리되었습니다.

`state`가 바르지 않다면 이 파일 내의 `setState()` 호출 중 하나에서 발생한 문제입니다. 또한, 보통 하나의 파일 내에서 `setState()` 호출은 많지 않기 때문에 검색이나 수정이 비교적 간단합니다.

`props`가 바르지 않다면 인스펙터로 거슬러 올라가며 트리를 이리저리 탐색하여 나쁜 `props`를 전달함으로써 최초로 "우물에 둑을 탄" 범인 컴포넌트를 찾을 수 있습니다.

현재의 `props`와 `state`의 형태로 어떤 UI를 생성한 데이터까지 추적할 수 있는 이 능력은 React에는 매우 중요합니다. `state`가 Closure와 연결자(combinator)에 갇혀 있지 않고 React에 직접적으로 이용할 수 있는 것은 명확한 설계 목표입니다.

UI는 동적이지만, `props`와 `state`의 동기적인 `render()` 함수는 디버깅 작업을 단순한 추적에서 지루하지만 유한한 과정으로 바꿀 수 있다고 믿습니다. 복잡한 애니메이션과 같은 몇 가지의 유스케이스가 보다 어렵게 되겠지만, 우리는 React에서 이 제한점을 고수했으면 합니다.

설정

우리는 글로벌 런타임 설정 옵션에 문제가 있음을 발견했습니다.

가령, 때때로 `React.configure(options)` 또는 `React.register(component)`과 같은 기능을 구현해 달라는 요청을 받습니다. 그렇지만 이것은 여러 가지 문제를 일으킬 수 있는데 우리는 그것에 대한 좋은 해결책을 알지 못합니다.

서드 파티 컴포넌트 라이브러리에서 이런 함수를 호출한다면? 하나의 React 앱이 다른 React 앱을 포함했는데 설정이 불완전하다면? 서드 파티 컴포넌트는 특정한 설정이 필요한지 아닌지 어떻게 구체화할 것인지? 우리는 글로벌 설정이 합성에서 제대로 동작하지 않을 것이라고 생각합니다. 합성이란 React의 중심이기 때문에 우리는 코드에서 글로벌 설정을 제공하지 않습니다.

그러나 우리는 빌드 레벨에서의 몇 가지 글로벌 설정을 제공합니다. 예를 들어, 분리된 개발 빌드와 프로덕션 빌드를 제공하고 있습니다. 추후에는 [프로파일링 빌드를 추가할](#)지도 모르고, 또 다른 빌드 플래그의 검토에 열려 있습니다.

DOM을 넘어서

우리는 더 적은 버그의 컴포넌트를 작성하여 구성할 수 있다는 면에서 React의 가치를 높니다. DOM은 React에서 기본적 렌더링 대상이지만, [React Native](#)는 Facebook과 커뮤니티에서 모두 중요합니다.

렌더러에 구속받지 않은은 React의 중요한 설계상의 제한점입니다. 그것은 내부 표현에 약간의 오버헤드를 더하게 됩니다. 반면, 코어의 개선은 모든 플랫폼에 통용됩니다.

단일 프로그래밍 모델을 가짐으로써 우리는 플랫폼 대신 프로덕트 중심의 엔지니어링 팀을 구성할 수 있습니다. 지금까지 우리에게 트레이드 오프는 그만한 가치가 있습니다.

구현

우리는 가능한 한 세련된 API를 제공하려고 노력합니다. 그러나 구현이 화려한 것에는 그다지 관심이 없습니다. 실세계는 완벽한 것에서 거리가 있습니다. 사용자가 못생긴 코드를 작성하지 않아도 된다면, 합리적인 범위에서 우리는 라이브러리에 그 못생긴 코드를 삽입하는 것을 선호합니다. 새로운 코드를 평가할 때 올바르고 성능이 좋고 뛰어난 개발자 경험을 제공하는 구현을 기대합니다. 우아함은 그 다음 문제입니다.

우리는 현명한 코드보다 지루한 코드를 선호합니다. 코드는 일회성이며 자주 변경됩니다. 그래서 [절대적으로 필요한 것이 아니라면 새로운 내부 추상화를 도입하지 않는 것이 중요합니다](#). 이동, 변경 또는 제거가 쉬운 장황한 코드는 시기상조로 추상화하여 변경하기 어려운 우아한 코드보다 우선합니다.

도구의 최적화

몇 가지 일반적으로 사용되고 있는 API들은 장황한 이름을 가지고 있습니다. 예를 들어, 우리는 `didMount()` 나 `onMount()` 대신 `componentDidMount()`를 사용합니다. 이것은 [의도적입니다](#). 목적은 라이브러리와의 인터렉션 포인트를 눈에 띄게 하는 것입니다.

Facebook과 같이 거대한 코드베이스에서 특정한 API의 사용을 검색할 수 있다는 것은 매우 중요합니다. 구별이 쉬운 장황한 이름, 특히 조심스럽게 사용해야만 하는 기능을 소중히 하고 있습니다. 예를 들어, 코드 리뷰에서 `dangerouslySetInnerHTML` 을 간과한다는 것은 아주 어려울 것입니다.

파괴적인 변경을 가할 때 [codemods](#)에 의존하고 있기 때문에 검색 최적화 또한 중요합니다. 우리는 거대한 자동화된 변경을 코드베이스 전체에 적용하는 것이 쉽고 안전했으면 하는 바람인데, 독특한 웅장한 이름을 사용하여 이것을 실현할 수 있습니다. 마찬가지로, 다른 것과 구별되는 이름을 사용하여, 거짓양성(false positives)을 걱정하는 일 없이, React 사용에 대한 사용자 [lint 규칙](#)을 쉽게 만들 수 있습니다.

[JSX](#)도 유사한 역할을 합니다. React에서 꼭 필요한 것은 아니지만 우리는 미적, 실용적인 이유로 Facebook에서 광범위하게 사용합니다.

우리의 코드베이스에서는, JSX는 React 엘리먼트 트리를 다루는 둘에 대한 명확한 힌트를 제공합니다. 이로 인해 [hoisting constant elements](#)와 안전한 [lint](#) 및 [codemod](#) 내부 컴포넌트 사용과 같은 빌드 시의 최적화를 추가하거나 JSX 소스 위치를 경고에 포함시킬 수 있습니다.

독푸딩

우리는 커뮤니티에서 제기한 문제를 해결하려고 최선을 다하고 있습니다. 그러나 우리는 사람들이 "또한" Facebook 내부적으로 겪고 있는 이슈를 우선시할 수 도 있습니다. 반 직관적으로 우리는 이것이 커뮤니티가 React에 내기할 수 있는 주요한 이유라고 생각합니다.

내부적으로 많이 사용하기 때문에 React가 내일 사라지지는 않을 거라는 확신을 가질 수 있습니다. React는 Facebook에서의 문제를 해결하기 위해 만들어졌습니다. React는 기업에 확실한 비지니스 가치를 가져다 주며 많은 프로덕트에 사용됩니다. [독푸딩](#) 그것은 우리의 전망을 날카롭게 유지하며 앞으로 나아가는 방향으로 초점을 맞추고 있다는 것을 의미합니다.

이것은 우리가 커뮤니티에서 제기하는 이슈를 무시한다는 것을 의미하지 않습니다. 가령, 우리가 Facebook 내부에서는 어느 하나에도 의존하지 않음에도 불구하고 웹 컴포넌트와 SVG에 대한 지원을 React에 추가했습니다. 우리는 적극적으로 여러분의 고통 포인트에 대해 듣고 있고 우리의 최선을 다해 그것들을 처리하고 있습니다. 커뮤니티는 React를 우리에게 특별하게 만들고, 우리는 다시 기여할 수 있음에 감사합니다.

Facebook에서 많은 오픈 소스 프로젝트를 출시한 이후에, 우리는 동시에 모두가 행복하도록 노력하는 것이 잘 성장하지 못 하는 약한 집중력의 프로젝트를 생성해냈다는 것을 배웠습니다. 대신, 우리는 작은 청중을 골라내고 그들을 행복하게 만드는 것에 집중하는 것이 긍정적인 순수 효과를 가져온다는 것을 발견했습니다. 이것이 우리가 React에서 한 바로 그 일이고, 지금까지 Facebook 프로덕트 팀이 직면한 문제를 해결한 것은 오픈 소스 커뮤니티로 잘 전달되었습니다.

이 접근법의 단점은 "시작하기"와 같은 경험처럼 때때로 Facebook 팀이 다를 필요가 없는 것들에 충분한 집중을 하지 못하는 경우가 있다는 것입니다. 우리는 이것을 정확하게 알고 있고 커뮤니티에서 우리가 이전에 오픈소스 프로젝트에서 했던 것과 같은 동일한 실수 없이 모든 사람에게 이익이 되는 방식으로 어떻게 개선할 것인가 생각하고 있습니다.

에러 경계(Error Boundaries)

과거에는 컴포넌트 내부의 자바스크립트 에러가 React의 내부 상태를 훼손하고 다음 렌더링에서 암호화 에러 방출을 유발하곤 했습니다. 이러한 에러는 항상 애플리케이션 코드의 이전 단계의 에러로 인해 발생했지만, React는 컴포넌트 내에서 에러를 정상적으로 처리할 수 있는 방법을 제공하지 않아 이를 복구할 수가 없었습니다.

에러 경계의 소개

UI의 일부분에 존재하는 자바스크립트 에러가 전체 애플리케이션을 중단시켜서는 안 됩니다. React 사용자들이 겪는 이 문제를 해결하기 위해 React 16에서는 에러 경계("error boundary")라는 새로운 개념이 도입되었습니다.

에러 경계는 **하위 컴포넌트 트리의 어디에서든 자바스크립트 에러를 기록하며 깨진 컴포넌트 트리 대신 폴백 UI를 보여주는 React 컴포넌트입니다.** 에러 경계는 렌더링 도중 생명주기 메서드 및 그 아래에 있는 전체 트리에서 에러를 잡아냅니다.

Note

에러 경계는 다음과 같은 에러는 포착하지 않습니다.

- 이벤트 핸들러 ([더 알아보기](#))
- 비동기적 코드 (예: `setTimeout` 혹은 `requestAnimationFrame` 콜백)
- 서버 사이드 렌더링
- 자식에서가 아닌 에러 경계 자체에서 발생하는 에러

생명주기 메서드인 `static getDerivedStateFromError()` 와 `componentDidCatch()` 중 하나 (혹은 둘 다)를 정의하면 클래스 컴포넌트 자체가 에러 경계가 됩니다. 에러가 발생한 뒤에 폴백 UI를 렌더링하려면 `static getDerivedStateFromError()` 를 사용하세요. 에러 정보를 기록하려면 `componentDidCatch()` 를 사용하세요.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // 다음 렌더링에서 폴백 UI가 보이도록 상태를 업데이트 합니다.
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // 에러 리포팅 서비스에 에러를 기록할 수도 있습니다.
    logErrorToMyService(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      // 폴백 UI를 커스텀하여 렌더링할 수 있습니다.
      return <h1>Something went wrong.</h1>;
    }
  }
}
```

```

    return this.props.children;
}

```

그런 다음엔 아래와 같이 일반 컴포넌트로 사용할 수 있습니다.

```

<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>

```

에러 경계는 자바스크립트의 `catch {}` 구문과 유사하게 동작하지만 컴포넌트에 적용됩니다. 오직 클래스 컴포넌트만이 에러 경계가 될 수 있습니다. 실제로도 대부분의 경우 에러 경계 컴포넌트를 한 번만 선언하여 애플리케이션 전체에서 활용하려고 할 것입니다.

에러 경계는 트리 내에서 하위에 존재하는 컴포넌트의 에러만을 포착합니다. 에러 경계 자체적으로는 에러를 포착할 수 없습니다. 에러 경계가 에러 메시지를 렌더링하는 데에 실패한다면 에러는 그 위의 가장 가까운 에러 경계로 전파될 것입니다. 이 또한 자바스크립트의 `catch {}` 구문이 동작하는 방식과 유사합니다.

라이브 데모

[에러 경계 선언과 사용 예시](#)를 참고하세요.

에러 경계를 배치할 위치

에러 경계의 좀 더 세분화된 부분은 개발자에게 달려있습니다. 서버 사이드 프레임워크가 충돌을 해결하는 것처럼 최상위 경로의 컴포넌트를 감싸서 유저에게 “문제가 발생했습니다”라는 메시지를 보여줄 수 있습니다. 또한 에러 경계의 각 위젯을 에러 경계로 감싸서 애플리케이션의 나머지 부분이 충돌하지 않도록 보호할 수도 있습니다.

포착되지 않는 에러에 대한 새로운 동작

이 변경사항은 중요한 의미를 갖습니다. **React 16**부터는 에러 경계에서 포착되지 않은 에러로 인해 전체 React 컴포넌트 트리의 마운트가 해제됩니다.

우리는 이 결정에 대해서 논의했지만 우리의 경험에 따르면 손상된 UI를 완전히 제거하는 것보다 그대로 남겨두는 것이 더 좋지 않습니다. 예를 들어 메신저와 같은 제품에서 손상된 UI를 그대로 남겨두면 누군가가 잘못된 사람에게 메시지를 보내게 될 가능성이 있습니다. 마찬가지로 결제 앱의 예를 들면 잘못된 금액을 보여주는 것이 아무 것도 렌더링하지 않는 것보다 더 나쁩니다.

이 변경사항은 React 16으로 마이그레이션 할 때 애플리케이션에서 이전에 알려지지 않았던 기존에 존재하던 충돌을 발견할 수 있음을 의미합니다. 에러 경계를 추가함으로써 문제가 발생했을 때 더 나은 사용자 경험을 제공할 수 있습니다.

예를 들어 페이스북 메신저는 사이드 바, 정보 패널, 대화 기록과 메시지 입력을 각각 별도의 에러 경계로 감싸두었습니다. 이 UI 영역 중 하나의 컴포넌트에서 충돌이 발생하면 나머지 컴포넌트는 대화형으로 유지됩니다.

또한 프로덕션 환경에서 발생한 처리되지 않은 예외 상황에 대하여 학습하고 수정할 수 있도록 자바스크립트 에러 리포팅 서비스를 활용하거나 직접 작성하는 것을 권장합니다.

컴포넌트 스택 추적

React 16은 애플리케이션이 실수로 에러를 집어삼킨 경우에도 개발 과정에서 렌더링하는 동안 발생한 모든 에러를 콘솔에 출력합니다. 에러 메시지 및 자바스크립트 스택과 더불어 React 16은 컴포넌트 스택 추적 또한 제공합니다. 이제 정확히 컴포넌트 트리의 어느 부분에서 에러가 발생했는지 확인할 수 있게 되었습니다.

```
▶ React caught an error thrown by BuggyCounter. You should fix this error in your code. react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.

Error: I crashed!

The error is located at:
in BuggyCounter (created by App)
in ErrorBoundary (created by App)
in div (created by App)
in App
```

또한 컴포넌트 스택 추적 내에서 파일 이름과 줄 번호도 확인할 수 있습니다. 이는 [Create React App](#) 프로젝트 내에서 기본적으로 동작합니다.

```
▶ React caught an error thrown by BuggyCounter. You should fix this error in your code. react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.

Error: I crashed!

The error is located at:
in BuggyCounter (at App.js:26)
in ErrorBoundary (at App.js:21)
in div (at App.js:8)
in App (at index.js:5)
```

Create React App을 사용하지 않는 경우에는 수동으로 [이 플러그인](#)을 Babel 설정을 추가할 수 있습니다. 이 기능은 개발 단계를 위해서만 제작되었으며 프로덕션 환경에서는 비활성화 해야합니다.

주의

스택 추적에 표시되는 컴포넌트 이름은 `Function.name` 프로퍼티에 따라 다릅니다. 아직 기본적으로 제공하지 않는 구형 브라우저 혹은 디바이스(예: IE11)를 지원하는 경우 번들 애플리케이션에 `function.name-polyfill`과 같은 `Function.name` 폴리필(Polyfill)을 포함시키는 것을 고려해볼 수 있습니다. 또는 모든 컴포넌트에서 `displayName` 프로퍼티를 명시적으로 설정할 수도 있습니다.

try/catch는 어떤가요?

`try` / `catch`는 헐륭하지만 명령형 코드에서만 동작합니다.

```
try {
  showButton();
} catch (error) {
  // ...
}
```

그러나 React 컴포넌트는 선언적이며 무엇을 렌더링할지 구체화합니다.

```
<Button />
```

에러 경계는 React의 선언적인 특성을 보존하고 예상한 대로 동작합니다. 예를 들어 트리의 깊숙한 어딘가에 있는 `setState`에 의해 유발된 `componentDidUpdate` 메서드에서 에러가 발생하더라도 가장 가까운 에러 경계에 올바르게 전달됩니다.

이벤트 핸들러는 어떤가요?

에러 경계는 이벤트 핸들러 내부에서는 에러를 포착하지 않습니다.

React는 이벤트 핸들러의 에러를 해결하기 위해서 에러 경계를 필요로 하지 않습니다. `render` 메서드 및 생명주기 메서드와 달리 이벤트 핸들러는 렌더링 중에 발생하지 않습니다. 따라서 이벤트 핸들러가 에러를 던져도 React는 여전히 화면에 무엇을 표시해야 할지 알고 있습니다.

이벤트 핸들러 내에서 에러를 잡아야 하는 경우에 일반 자바스크립트의 `try` / `catch` 구문을 사용하세요.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { error: null };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    try {
      // 에러를 던질 수 있는 무언가를 해야합니다.
    } catch (error) {
      this.setState({ error });
    }
  }

  render() {
    if (this.state.error) {
      return <h1>Caught an error.</h1>
    }
    return <button onClick={this.handleClick}>Click Me</button>
  }
}
```

위의 예시는 일반적인 자바스크립트의 동작을 보여주며 에러 경계를 사용하지 않습니다.

React 15에서 변경된 명명법

React 15는 `unstable_handleError`라는 다른 메서드 이름으로 에러 경계에 대해 매우 제한적인 지원을 포함하고 있었습니다. 이 메서드는 더 이상 동작하지 않으며 첫 16 베타 릴리즈부터 코드에서 `componentDidCatch`로 변경해야 합니다.

이 변경사항을 위해 코드를 자동으로 마이그레이션하기 위한 [codemod](#)를 제공했습니다.

[Go to TOC](#)

에러 해독기(Error Decoder)

React의 축소된 배포용 빌드에서는 전송되는 바이트 수를 줄이기 위해 전체 오류 메시지를 전송하지 않습니다.

저희는 추가 디버그 정보를 추적하고 앱의 잠재적인 문제에 대한 유용한 경고를 제공하므로 로컬에서 앱을 디버깅할 때는 개발용 빌드를 사용하는 것이 좋습니다. 그러나 배포용 빌드를 사용하는 동안 예외가 발생하면 이 페이지가 오류의 원본 텍스트를 다시 조합합니다.

[Go to TOC](#)

AJAX 와 APIs

어떻게 AJAX 호출을 할 수 있을까요?

선호하는 AJAX 라이브러리를 React와 함께 사용할 수 있습니다. 유명한 라이브러리로는 [Axios](#), [jQuery AJAX](#), 그리고 브라우저에 내장된 `window.fetch` 등이 있습니다.

컴포넌트의 생명주기 중 어디에서 AJAX 호출을 할 수 있나요?

AJAX 호출을 통한 데이터는 생명주기 메서드 중 `componentDidMount` 안에 추가되어야 합니다. 이는 데이터를 받아 올 때 `setState`를 통하여 컴포넌트를 업데이트하기 위함입니다.

예시: 로컬 state를 설정하기 위해 AJAX 결과 사용하기

아래 컴포넌트는 로컬 컴포넌트의 state를 채우기 위하여 `componentDidMount` 안에서 어떻게 AJAX 호출을 만드는지 보여 줍니다.

API 예시는 다음과 같은 JSON 객체를 반환합니다.

```
{
  "items": [
    { "id": 1, "name": "Apples", "price": "$2" },
    { "id": 2, "name": "Peaches", "price": "$5" }
  ]
}
```

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      error: null,
      isLoading: false,
      items: []
    };
  }

  componentDidMount() {
    fetch("https://api.example.com/items")
      .then(res => res.json())
      .then(
        result => {
          this.setState({
            isLoading: true,
            items: result.items
          });
        },
        // 주의: 컴포넌트에 있는 실제 버그로 인해 발생한 예외를
        // 놓치지 않고 처리하기 위해서는
        // catch() 블록보다는 여기서 에러를 다뤄주는 게 중요합니다.
        error => {
          this.setState({
            isLoading: true,
            error
          });
        }
      );
  }
}
```

```

        )
    }

    render() {
        const { error, isLoading, items } = this.state;
        if (error) {
            return <div>Error: {error.message}</div>;
        } else if (!isLoading) {
            return <div>Loading...</div>;
        } else {
            return (
                <ul>
                    {items.map(item => (
                        <li key={item.id}>
                            {item.name} {item.price}
                        </li>
                    ))}
                </ul>
            );
        }
    }
}

```

이는 아래처럼 Hook으로 작성한 코드와 동일합니다.

```

function MyComponent() {
    const [error, setError] = useState(null);
    const [isLoading, setIsLoaded] = useState(false);
    const [items, setItems] = useState([]);

    // 주의: 빈 deps 배열 []은
    // useEffect가 componentDidMount()처럼
    // 한 번만 실행되는 걸 의미합니다.
    useEffect(() => {
        fetch("https://api.example.com/items")
            .then(res => res.json())
            .then(
                (result) => {
                    setIsLoaded(true);
                    setItems(result);
                },
                // 주의: 컴포넌트에 있는 실제 버그로 인해 발생한 예외를
                // 놓치지 않고 처리하기 위해서는
                // catch() 블록보다는 여기서 에러를 다뤄주는 게 중요합니다.
                (error) => {
                    setIsLoaded(true);
                    setError(error);
                }
            )
        }, [])
    }

    if (error) {
        return <div>Error: {error.message}</div>;
    } else if (!isLoading) {
        return <div>Loading...</div>;
    } else {
        return (
            <ul>
                {items.map(item => (
                    <li key={item.id}>
                        {item.name} {item.price}
                    </li>
                ))}
            </ul>
        );
    }
}

```

```
</li>
});}
</ul>
);
}
}
```

Babel, JSX, 그리고 빌드 과정들

React에 JSX를 꼭 사용해야 하나요?

아니요! 더 자세한 내용을 위해 "[JSX 없이 React 사용하기](#)" 를 확인해 주세요.

React에 ES6 (+)를 꼭 사용해야 하나요?

아니요! 더 자세한 내용을 위해 "[ES6 없이 React 사용하기](#)" 를 확인해 주세요.

JSX에서 어떻게 주석을 달 수 있나요?

```
<div>
  /* 주석은 여기에 */
  Hello, {name}!
</div>
```

```
<div>
  /* 여러 줄의
  주석도 가능합니다. */
  Hello, {name}!
</div>
```

[Go to TOC](#)

컴포넌트에 함수 전달하기

컴포넌트로 onClick과 같은 이벤트 핸들러를 어떻게 전달 할까요?

자식 컴포넌트에 프로퍼티로 이벤트 핸들러와 다른 함수들을 전달합니다.

```
<button onClick={this.handleClick}>
```

핸들러 안에서 부모 컴포넌트에 접근할 필요가 있으면 컴포넌트 인스턴스에 함수를 바인딩해 주어야 합니다.

컴포넌트 인스턴스로 함수를 어떻게 바인딩할까요?

사용하고 있는 문법과 빌드 단계에 따라 `this.props`, `this.state` 와 같은 컴포넌트의 어트리뷰트에 함수들이 확실히 접근할 수 있도록 만드는 방법은 여러 가지가 있습니다.

생성자에서 바인딩하기 (ES2015)

```
class Foo extends Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

클래스 프로퍼티 (ES2022)

```
class Foo extends Component {
  handleClick = () => {
    console.log('Click happened');
  };
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

render 메서드 안에서 바인딩하기

```
class Foo extends Component {
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={this.handleClick.bind(this)}>Click Me</button>;
  }
}
```

주의

`Function.prototype.bind` 를 `render` 메서드에서 사용하면 컴포넌트가 렌더링할 때마다 새로운 함수를 생성하기 때문에 성능에 영향을 줄 수 있습니다.

render 메서드 안에서 화살표 함수 사용

```
class Foo extends Component {
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={() => this.handleClick()}>Click Me</button>;
  }
}
```

주의

`render` 메서드 안에서 화살표 함수를 사용하면 컴포넌트가 렌더링할 때마다 새로운 함수를 만들기 때문에 엄격한 비교에 의해 최적화가 깨질 수 있습니다.

render 메서드 안에서 화살표 함수를 사용해도 괜찮을까요?

이 방법은 대체로 사용해도 괜찮고, 콜백 함수로 매개변수를 전달해 주는 가장 쉬운 방법입니다.

성능 문제가 있다면 반드시 최적화를 해야 합니다.

바인딩이 필요한 이유는 무엇일까요?

자바스크립트에서 아래 두 개의 코드 조각은 동일하지 않습니다.

```
obj.method();
```

```
var method = obj.method;
method();
```

바인딩 메서드는 두 번째 코드 조각이 첫 번째 코드조각과 같은 방식으로 작동하도록 만들어 줍니다.

일반적으로 React에서 다른 컴포넌트에 메서드를 전달해 줄 때만 바인딩해 주면 됩니다. 예를 들어 `<button onClick={this.handleClick}>` 는 `this.handleClick` 을 전달하여 바인딩합니다. 그렇지만 `render` 메서드나 생명주기 메서드는 다른 컴포넌트로 전달하지 않기 때문에 바인딩할 필요가 없습니다.

[Yehuda Katz의 글](#)에서 바인딩이 무엇인지, JavaScript에서 어떻게 함수가 작동하는지에 대해 상세히 알 수 있습니다.

왜 컴포넌트가 렌더링할 때마다 함수가 호출될까요?

컴포넌트로 함수를 전달할 때 호출하지 않는지 확인합니다.

```
render() {
  // 잘못된 방법: handleClick은 레퍼런스로 전달되지 않고 호출되었습니다!
  return <button onClick={this.handleClick}>Click Me</button>
}
```

위와 같은 방식이 아니라 괄호 없이 함수 그 자체를 전달해야 합니다.

```
render() {
  // 올바른 방법 : handleClick이 레퍼런스로 전달되었습니다.
  return <button onClick={this.handleClick}>Click Me</button>
}
```

이벤트 핸들러나 콜백에 어떻게 매개변수를 전달할까요?

이벤트 핸들러에 화살표 함수를 사용하여 감싼 다음에 매개변수를 넘겨줄 수 있습니다.

```
<button onClick={() => this.handleClick(id)} />
```

.bind 를 호출한 것과 같습니다.

```
<button onClick={this.handleClick.bind(this, id)} />
```

예시: 화살표 함수를 이용하여 매개변수 전달하기

```
const A = 65 // ASCII character code

class Alphabet extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      justClicked: null,
      letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
    };
  }
  handleClick(letter) {
    this.setState({ justClicked: letter });
  }
  render() {
    return (
      <div>
        Just clicked: {this.state.justClicked}
        <ul>
          {this.state.letters.map(letter =>
            <li key={letter} onClick={() => this.handleClick(letter)}>
              {letter}
            </li>
          )}
        </ul>
      </div>
    );
  }
}
```

예시: data-attributes를 사용해서 매개변수 전달하기

다른 방법으로 이벤트 핸들러에 필요한 데이터를 저장하기 위해 DOM API를 사용할 수 있습니다. 이 방법은 아주 많은 요소를 최적화하거나 React.PureComponent 동일성 검사에 의존하는 렌더링 트리를 사용할 때 고려해 볼 만합니다.

```

const A = 65 // ASCII character code

class Alphabet extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.state = {
      justClicked: null,
      letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
    };
  }

  handleClick(e) {
    this.setState({
      justClicked: e.target.dataset.letter
    });
  }

  render() {
    return (
      <div>
        Just clicked: {this.state.justClicked}
        <ul>
          {this.state.letters.map(letter =>
            <li key={letter} data-letter={letter} onClick={this.handleClick}>
              {letter}
            </li>
          )}
        </ul>
      </div>
    );
  }
}

```

어떻게 함수가 너무 빨리, 너무 많이 호출되는 것을 막을 수 있나요?

`onClick` 또는 `onScroll`과 같은 이벤트 핸들러를 사용하고 있을 때 콜백이 너무 빠르게 호출되지 않도록 콜백이 실행되는 속도를 제어할 수 있습니다. 다음의 함수들을 사용하면 됩니다.

- **throttling**: 시간 기반 빈도에 따른 변경 샘플링 (예시 `_.throttle`)
- **debouncing**: 비활성 주기 이후에 변경 적용 (예시 `_.debounce`)
- **requestAnimationFrame throttling**: `requestAnimationFrame` (예시 `raf-schd`)을 기반으로 한 변경 샘플링

`throttle`과 `debounce` 함수를 비교하고 싶으면 [시각화](#)를 확인하면 됩니다.

주의

`_.debounce`, `_.throttle`, `raf-schd`는 지연되는 콜백을 취소하는 메서드 `cancel`을 제공합니다. `componentWillUnmount`에서 이 함수를 사용하거나 또는 지연된 함수 내에서 컴포넌트가 마운트가 되어있음을 확인해야 합니다.

Throttle

Throttling은 함수가 주어진 시간 동안에 한 번 이상 호출되는 것을 막습니다. 아래는 "click" 핸들러에 throttling을 사용하여 초당 한 번만 호출되도록 한 예시입니다.

```

import throttle from 'lodash.throttle';

class LoadMoreButton extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.handleClickThrottled = throttle(this.handleClick, 1000);
  }

  componentWillUnmount() {
    this.handleClickThrottled.cancel();
  }

  render() {
    return <button onClick={this.handleClickThrottled}>Load More</button>;
  }

  handleClick() {
    this.props.loadMore();
  }
}

```

Debounce

Debouncing은 함수가 마지막으로 호출된 후 특정 시간까지 실행되지 않도록 해줍니다. 빠르게 발생하는 이벤트(예시 스크롤, 키보드 이벤트)의 응답으로 어떤 비싼 계산을 수행해야 할 때 사용하면 좋습니다. 아래의 예시는 250 밀리초 이내의 텍스트 입력을 Debouncing했습니다.

```

import debounce from 'lodash.debounce';

class Searchbox extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.emitChangeDebounced = debounce(this.emitChange, 250);
  }

  componentWillUnmount() {
    this.emitChangeDebounced.cancel();
  }

  render() {
    return (
      <input
        type="text"
        onChange={this.handleChange}
        placeholder="Search..."
        defaultValue={this.props.value}
      />
    );
  }

  handleChange(e) {
    this.emitChangeDebounced(e.target.value);
  }

  emitChange(value) {
    this.props.onChange(value);
  }
}

```

requestAnimationFrame throttling

`requestAnimationFrame`은 렌더링 성능을 위해 브라우저에서 최적화된 시간에 함수가 실행되도록 함수를 큐잉하는 방법입니다. `requestAnimationFrame`의 큐로 들어간 함수는 다음 프레임에서 실행됩니다. 브라우저는 1초당 60 프레임(60 fps)을 보장하기 위해 열심히 일합니다. 하지만 브라우저가 이를 하지 못할 때 저절로 프레임을 제한합니다. 예를 들면 한 기기가 30 fps만 처리할 수 있다면 1초 동안 30 프레임만 얻을 수 있습니다. throttling을 위해 `requestAnimationFrame`을 사용하면 1초에 60번 이상 업데이트하는 것을 막을 수 있습니다. 1초당 100번 업데이트하도록 브라우저에 일을 만들어 주어도, 유저는 이를 확인할 수 없습니다.

주의

이 기법을 사용하면, 프레임에 가장 마지막으로 계재된 값만 사용하게 됩니다. 최적화가 어떻게 작동하는지에 대한 예시는 MDN에서 확인할 수 있습니다.

```
import rafSchedule from 'raf-sched';

class ScrollListener extends React.Component {
  constructor(props) {
    super(props);

    this.handleScroll = this.handleScroll.bind(this);

    // 업데이트 일정을 정하는 함수를 만듭니다.
    this.scheduleUpdate = rafSchedule(
      point => this.props.onScroll(point)
    );
  }

  handleScroll(e) {
    // 스크롤 이벤트를 받게 되면 업데이트를 일정에 추가합니다.
    // 한 프레임 안에 많은 업데이트를 받으면 오직 마지막 값만 계재합니다.
    this.scheduleUpdate({ x: e.clientX, y: e.clientY });
  }

  componentWillUnmount() {
    // 마운트 해제 중에 임시상태의 업데이트들을 모두 취소합니다.
    this.scheduleUpdate.cancel();
  }

  render() {
    return (
      <div
        style={{ overflow: 'scroll' }}
        onScroll={this.handleScroll}
      >
        
      </div>
    );
  }
}
```

속도 제한 테스트 방법

속도 제한 코드가 잘 작동하는지 테스트할 때, 빨리 감기 기능을 사용하는 것이 좋습니다. `jest` 를 사용한다면 `mock timers` 를 빨리 감기 도구로 사용할 수 있습니다. `requestAnimationFrame throttling`을 사용한다면 애니메이션 프레임의 틱을 제어하기 위한 툴로 `raf-stub` 를 보면 좋습니다.

Virtual DOM과 Internals

Virtual DOM은 무엇인가요?

Virtual DOM (VDOM)은 UI의 이상적인 또는 "가상"적인 표현을 메모리에 저장하고 ReactDOM과 같은 라이브러리에 의해 "실제" DOM과 동기화하는 프로그래밍 개념입니다. 이 과정을 [재조정](#)이라고 합니다.

이 접근방식이 React의 선언적 API를 가능하게 합니다. React에게 원하는 UI의 상태를 알려주면 DOM이 그 상태와 일치하도록 합니다. 이러한 방식은 앱 구축에 사용해야 하는 어트리뷰트 조작, 이벤트 처리, 수동 DOM 업데이트를 추상화합니다.

"virtual DOM"은 특정 기술이라기보다는 패턴에 가깝기 때문에 사람들마다 의미하는 바가 다릅니다. React의 세계에서 "virtual DOM"이라는 용어는 보통 사용자 인터페이스를 나타내는 객체이기 때문에 [React elements](#)와 연관됩니다. 그러나 React는 컴포넌트 트리에 대한 추가 정보를 포함하기 위해 "fibers"라는 내부 객체를 사용합니다. 또한 React에서 "virtual DOM" 구현의 일부로 간주할 수 있습니다.

Shadow DOM은 Virtual DOM과 같은가요?

아니요, 둘은 다릅니다. Shadow DOM은 주로 웹 컴포넌트의 범위 지정 변수 및 CSS용으로 설계된 브라우저 기술입니다. virtual DOM은 브라우저 API 위에 있는 JavaScript 라이브러리에서 구현되는 개념입니다.

"React Fiber"는 무엇인가요?

Fiber는 React 16의 새로운 재조정 엔진입니다. 이 프로그램의 주요 목표는 virtual DOM의 증분 렌더링을 활성화하는 것입니다. [더 읽기](#).

[Go to TOC](#)

컴포넌트 State

setState 는 어떤 일을 하나요?

`setState()` 는 컴포넌트의 `state` 객체에 대한 업데이트를 실행합니다. `state`가 변경되면, 컴포넌트는 리렌더링됩니다.

state 와 props 의 차이점은 무엇인가요?

`props` ("properties"의 줄임말) 와 `state` 는 일반 JavaScript 객체입니다. 두 객체 모두 렌더링 결과물에 영향을 주는 정보를 갖고 있는데, 한 가지 중요한 방식에서 차이가 있습니다. `props` 는 (함수 매개변수처럼) 컴포넌트에 전달되는 반면 `state` 는 (함수 내에 선언된 변수처럼) 컴포넌트 안에서 관리됩니다.

언제 `props` 와 `state` 를 사용하는지 더 알고 싶다면 아래의 자료를 확인해보세요.

- [Props vs State](#)
- [ReactJS: Props vs. State](#)

왜 setState 가 잘못된 값을 주는 걸까요?

React에서 `this.props` 와 `this.state` 는 모두 렌더링된 값을 나타냅니다. 다시 말해 현재 화면에 보이는 것을 말합니다.

`setState` 호출은 비동기적으로 이루어집니다. 따라서 `setState` 호출 직후 새로운 값이 `this.state` 에 반영될 거라고 믿어서는 안 됩니다. 이전 `state` 값을 기준으로 값을 계산해야 한다면 객체 대신 `update` 함수를 전달하세요. (자세한 내용은 아래를 확인하세요.)

예시 코드는 예상대로 동작하지 않을 것입니다.

```
incrementCount() {
  // 주의: 이 코드는 예상대로 동작하지 *않을 것*입니다.
  this.setState({count: this.state.count + 1});
}

handleSomething() {
  // `this.state.count`가 0에서 시작한다고 해봅시다.
  this.incrementCount();
  this.incrementCount();
  this.incrementCount();
  // React가 컴포넌트를 리렌더링할 때 `this.state.count`는 3이 될 것 같은 예상과 달리 1이 됩니다.

  // 이것은 `incrementCount()` 함수가 `this.state.count`에서 값을 읽어 오는데
  // React는 컴포넌트가 리렌더링될 때까지 `this.state.count`를 갱신하지 않기 때문입니다.
  // 그러므로 `incrementCount()`는 매번 `this.state.count`의 값을 0으로 읽은 뒤에 이 값을 1로
  // 설정합니다.

  // 이 문제의 해결 방법은 아래에 설명되어 있습니다.
}
```

이 문제를 어떻게 해결하는지 알아봅시다.

어떻게 하면 이전 state 값을 기준으로 state 값을 업데이트할 수 있나요?

항상 `setState` 가 가장 최신의 state 값을 사용하도록 보장하기 위해서는 `setState` 에 객체 대신 함수를 전달하세요. (아래를 참조하세요.)

`setState`에 객체를 전달하는 것과 함수를 전달하는 것은 어떤 차이가 있나요?

updater 함수를 전달하면 updater 함수 안에서 이전 state 값에 접근할 수 있습니다. `setState` 호출은 일괄적으로 처리되기 때문에 여러 업데이트 사항이 충돌 없이 차례대로 반영되도록 합니다.

```
incrementCount() {
  this.setState((state) => {
    // 중요: 값을 업데이트할 때 `this.state` 대신 `state` 값을 읽어옵니다.
    return {count: state.count + 1}
  });
}

handleSomething() {
  // `this.state.count` 가 0에서 시작한다고 해봅시다.
  this.incrementCount();
  this.incrementCount();
  this.incrementCount();

  // 지금 `this.state.count` 값을 읽어 보면 이 값은 여전히 0일 것입니다.
  // 하지만 React가 컴포넌트를 리렌더링하게 되면 이 값은 3이 됩니다.
}
```

[setState에 대해 더 알아보기](#)

언제 `setState` 가 비동기적인가요?

현재 `setState` 는 이벤트 핸들러 내에서 비동기적입니다.

이로 인해 부모 와 자식 이 모두 click 이벤트에서 `setState` 를 호출한다면 자식 은 두 번 렌더링되지 않습니다. 대신 React는 브라우저 이벤트가 끝날 시점에 state를 일괄적으로 업데이트합니다. 이는 더 큰 규모의 앱에서 뚜렷한 성능 향상을 만들어냅니다.

이것은 구현 세부사항이므로 전적으로 의존해선 안됩니다. 추후 React는 기본적으로 더 많은 경우에서 일괄적으로 업데이트를 처리할 예정입니다.

왜 React는 `this.state` 를 동기적으로 업데이트하지 않나요?

이전 절에서 설명했듯이 모든 컴포넌트가 자신의 이벤트 핸들러에서 `setState()` 를 호출할 때까지 React는 리렌더링을 하지 않고 내부적으로 "기다리고 있습니다". 이를 통해 불필요한 렌더링을 방지하면서 성능을 향상시킵니다.

그러나 왜 React는 리렌더링 대신 즉시 `this.state` 를 업데이트하지 않는지 여전히 궁금해 하실 수도 있습니다.

여기에는 두 가지 중요한 이유가 존재합니다.

- `props` 와 `state` 사이의 일관성을 해칠 수 있으며 이것은 디버깅하기 매우 힘든 이슈를 일으킬 수 있기 때문입니다.
- 현재 작업 중인 새로운 기능들을 구현하기 힘들게 만들 수 있기 때문입니다.

이 [GitHub 코멘트](#)에서 더욱 자세한 예시를 확인할 수 있습니다.

Redux 나 MobX 같은 상태('state') 관리 라이브러리를 사용해야 하나요?

아마 필요할 수도 있습니다.

추가적인 라이브러리를 사용하기 전에 먼저 React에 익숙해지는 게 좋습니다. React만으로도 꽤 복잡한 애플리케이션을 만들 수 있습니다.

[Go to TOC](#)

파일 구조

React 프로젝트 구조를 설계하기 위해 추천할 만한 방법이 있을까요?

React는 파일을 어떤 식으로 폴더에 분류할 것인지에 대해서 제시하고 있지는 않습니다. 그러나 React 생태계 내에서 고려할만한 몇 가지 인기 있는 일반적인 접근법들이 있습니다.

파일의 기능이나 라우트에 의한 분류

프로젝트 구조에 대한 일반적인 방법 중 하나는 CSS, JS 그리고 테스트 파일을 기능이나 라우트로 분류된 폴더에 같이 두는 방법입니다.

```
common/
  Avatar.js
  Avatar.css
  APIUtils.js
  APIUtils.test.js
feed/
  index.js
  Feed.js
  Feed.css
  FeedStory.js
  FeedStory.test.js
  FeedAPI.js
profile/
  index.js
  Profile.js
  ProfileHeader.js
  ProfileHeader.css
  ProfileAPI.js
```

"기능"의 정의는 보편적인 것이 아닙니다. 그것은 얼마나 세분화할 것인지에 달려 있습니다. 최상위 폴더 리스트를 생각해낼 수 없다면, 서비스 사용자들에게 해당 서비스를 구성하는 중요한 부분이 무엇인지 물어볼 수 있습니다. 그리고 사용자들의 의견을 토대로 만들어진 구조를 청사진으로 사용할 수 있습니다.

파일 유형에 의한 분류

프로젝트 구조를 정하는 또 다른 인기 있는 방법은 비슷한 파일끼리 묶어주는 것입니다. 아래 예시를 참고하세요.

```
api/
  APIUtils.js
  APIUtils.test.js
  ProfileAPI.js
  UserAPI.js
components/
  Avatar.js
  Avatar.css
  Feed.js
  Feed.css
  FeedStory.js
  FeedStory.test.js
  Profile.js
  ProfileHeader.js
  ProfileHeader.css
```

어떤 사람들은 더 나아가서 해당 애플리케이션 내에서의 역할에 따라 컴포넌트를 다른 폴더로 분리하는 것을 선호합니다. 예를 들어 [아토믹 디자인](#)은 이와 같은 원칙에 기반한 디자인 방법론입니다. 지켜야 할 엄격한 규칙이라기보다 도움이 되는 예시로 이러한 방법론을 활용하는 것은 종종 더욱 생산적이라는 사실을 잊지 마세요.

너무 많은 중첩을 피하세요

JavaScript 프로젝트에서 깊은 디렉토리 중첩과 관련된 문제점들이 많이 있습니다. 프로젝트 간에 상대 경로를 통한 임포트를 작성하거나 또는 파일이 옮겨졌을 때 그러한 임포트들을 업데이트하는 것이 더 어려워집니다. 깊은 폴더 구조를 사용해야 하는 매우 설득력 있는 이유가 없다면, 단일 프로젝트 내에서는 3번 혹은 4번을 최대한으로 폴더를 중첩하도록 제한하는 것을 고려해보세요. 물론 이것은 단지 권장 사항일 뿐이며, 여러분의 프로젝트와 관련 없을 수도 있습니다.

너무 깊게 생각하지 마세요

이제 막 프로젝트를 시작하는 단계라면 파일 구조를 선택하는 것에 있어서 [5분 이상 시간을 투자하지 마세요](#). 앞서 살펴본 접근법 중에 아무거나 선택하고(혹은 자신만의 방법을 찾아내세요) 코드를 우선 작성해보세요.

완전히 난관에 놓착해있다면, 모든 파일을 하나의 폴더에 보관하는 방법으로 우선 시작해보세요. 결국에는 프로젝트가 충분히 커져서 일부 파일을 나머지로부터 분리해 보관하기 원하게 될 것입니다. 그 시점까지 어떤 파일들을 가장 자주 묶어서 수정하는지 충분히 알 수 있게 될 것입니다. 일반적으로, 자주 함께 변경되는 파일들을 같이 보관하는 것이 좋은 방법입니다. 이러한 원칙을 "코로케이션(colocation)"이라고 부릅니다.

프로젝트가 커지게 되면서, 실제로는 앞서 언급한 방법들을 섞어서 사용하게 됩니다. 그래서 처음부터 "옳은" 방법 하나를 선택하는 것이 대단히 중요하지는 않습니다.

스타일링과 CSS

CSS 클래스를 컴포넌트에 어떻게 추가하나요?

`className` prop에 문자열을 넘깁니다.

```
render() {
  return <span className="menu navigation-menu">Menu</span>
}
```

컴포넌트의 props나 state에 CSS 클래스가 의존하는 것은 자주 사용되는 방식입니다.

```
render() {
  let className = 'menu';
  if (this.props.isActive) {
    className += ' menu-active';
  }
  return <span className={className}>Menu</span>
}
```

팁

이러한 방식으로 코드를 자주 작성한다면, `classnames` 패키지를 통해 이 작업을 간소화할 수 있습니다.

인라인 스타일을 사용할 수 있나요?

네, [여기](#)에서 스타일링에 관한 문서를 참고해 주세요.

인라인 스타일은 좋지 않은 방식인가요?

보통의 경우 CSS 클래스가 인라인 스타일보다 더 나은 성능을 보입니다.

CSS-in-JS가 무엇인가요?

"CSS-in-JS"는 외부의 파일에 CSS를 정의하는 대신에 JavaScript와 결합하는 패턴을 의미합니다.

이러한 기능은 React에 포함된 기능이 아닌, 별도의 라이브러리로 제공되고 있는 것에 주의해주세요. React는 어떻게 스타일이 정의되는지에 대한 의견이 없습니다. 의구심이 생긴다면, 평소처럼 별도의 `*.css` 파일에 정의한 뒤 `className`을 통해 참조하는 것으로 시작해볼 수 있습니다.

React에서 애니메이션을 할 수 있나요?

React는 애니메이션을 표현할 수 있습니다. 이에 대한 예시로 [React Transition Group](#), [React Motion](#), [React Spring](#) 또는 [Framer Motion](#)을 확인해 보세요.

[Go to TOC](#)

버전 정책

React는 유의적 버전 ([semver](#)) 원칙을 따릅니다.

즉, 버전 번호는 **x.y.z** 형태가 됩니다.

- 치명적인 버그를 수정할 때는 **z** 번호를 바꾸고 수 버전을 올려 배포합니다. (예를 들어, 15.6.2에서 15.6.3)
- 새로운 기능을 출시하거나 치명적이지 않은 버그를 수정할 때는 **y** 번호를 바꾸고 부 버전을 올려 배포를 합니다. (예를 들어, 15.6.2에서 15.7.0)
- 호환성이 유지되지 않는 변경이 있을 때는 **x** 번호를 바꾸고 주 버전을 올려 배포를 합니다. (예를 들어, 15.6.2에서 16.0.0)

주 버전 배포는 새 기능을 포함할 수 있으며 모든 배포에는 버그 수정을 포함할 수도 있습니다.

부 버전을 변경시킨 배포가 가장 보편적인 배포 형태입니다.

해당 버전 정책은 Next 또는 Experimental 채널의 [prerelease](#) 빌드에는 적용되지 않습니다. [prerelease](#)에 대해 여기서 알아보세요.

호환성이 유지되지 않는 변경

호환성이 유지되지 않는 변경은 모든 사람에게 불편할 수 있습니다. 그래서 주 버전 배포는 최소화하도록 노력합니다. 예를 들어, React 15는 2016년 4월에 배포되었고 React 16은 2017년 9월에 배포되었습니다. 그리고 React 17은 2020년 10월에 배포되었습니다.

대신 부 버전을 변경시킨 배포로 새 기능을 배포합니다. 부 버전 배포는 겸손한 이름에도 불구하고, 주 버전 배포보다 가끔 더 흥미롭고 주목할 만합니다.

안정성을 위한 노력

React가 서서히 변화하는 가운데 우리는 새 기능을 도입하기 위해 필요한 노력을 최소화하고 있습니다. 오래된 API를 다른 패키지로 분리하더라도, 오래된 API가 계속 동작하도록 가능한 유지할 예정입니다. 예를 들어, [믹스인은 수년 동안 그 기세가 꺾여오고 있지만](#), [create-react-class](#)를 통해서 오늘날까지 지원되고 있고, 많은 코드 베이스가 안정된 레거시 코드에서 오래된 API를 계속 사용되고 있습니다.

백만 명 이상의 개발자가 React를 사용하고 있고, React에는 통틀어 수백만 개의 컴포넌트가 유지되고 있습니다. 페이스북 코드 베이스만 고려해 봐도 50,000개 이상의 React 컴포넌트가 있습니다. 이것은 React의 새 버전으로의 업그레이드를 가능한 한 쉽게 만들 필요가 있음을 의미합니다. 마이그레이션 수단도 없이 대단위의 큰 변화가 발생한다면 개발자들은 구버전에 갇히게 될 것입니다. 우리는 이 업그레이드 방법을 페이스북 자체에 테스트하고 있습니다. 열 명도 되지 않는 우리 팀이 단독적으로 5만 개 이상의 컴포넌트를 업데이트할 수 있다면 React를 사용하는 모든 개발자가 업그레이드를 관리할 수 있기를 기대합니다. 대부분의 경우, 컴포넌트 구문을 업그레이드하기 위한 [자동화 스크립트](#)를 오픈소스 배포에 포함해서 누구나 사용할 수 있도록 하고 있습니다.

경고를 통한 점진적인 업그레이드

React 개발 빌드는 상당수의 유용한 경고를 포함합니다. 가능한 한 미래의 큰 변화에 대비하여 경고를 추가합니다. 최신 배포에서 여러분의 앱이 경고를 표시하지 않았다면 다음에 배포될 주 버전과 호환될 겁니다. 경고 덕분에 여러분은 컴포넌트 하나씩 앱을 업그레이드할 수 있게 해줍니다.

개발 경고는 앱의 런타임 수행에 영향을 주지는 않습니다. 다만, 그로 인해 앱의 개발 빌드와 프로덕션 빌드가 동일한 방식으로 동작할 거라는 확신을 가질 수 있습니다. 개발 빌드와 유일한 차이점이라면 프로덕션 빌드는 경고를 출력하지 않으며 조금 더 효율적입니다. (또 다른 점이 발견된다면 이슈를 제출해주세요)

무엇이 호환되지 않는 변화일까요?

보통 우리는 다음의 변경에는 주 버전 번호를 *변화시키지 않습니다.*

- **개발 환경에서의 경고.** 프로덕션 빌드의 동작에는 영향을 미치지 않기 때문에 주요 버전 간에 새로운 경고를 추가하거나 기존 경고를 수정할 수 있습니다. 실제로, 이를 통해 다가올 호환되지 않는 변화에 대해 확실하게 경고할 수 있게 해줍니다.
- **`unstable_`로 시작하는 API.** 아직 신뢰할 수 있지 않은 API를 실험적인 기능으로 제공합니다. `unstable_` 접두어를 사용한 버전을 배포해서 개발 주기를 조금 더 빠르게 하고 안정적인 API를 조금 더 빠르게 얻을 수 있습니다.
- **React의 알파 버전과 카나리아(canary) 버전.** 새 기능의 이론 테스트를 위한 방법으로 React의 알파 버전을 제공합니다. 그러나 동시에, 알파 기간에 습득한 것을 바탕으로 변경을 자유롭게 할 수 있는 유연함도 필요합니다. 알파 버전을 사용하고 있다면 안정 버전이 출시되기 전에는 API가 얼마든지 변경될 수 있으니 주의해주세요.
- **문서화되지 않은 API와 내부 데이터 구조.** `__SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED` 라든지 `__reactInternalInstance$uk43rzhitjg` 와 같은 내부 프로퍼티에 접근한다면 책임은 스스로 지셔야 합니다.

이 정책은 실용적으로 설계되었습니다. 특히, 여러분의 머리가 아프지 않았으면 합니다. 이 모든 변화에 대해 주 버전 번호를 변화시킨다면, 더 많은 주 버전을 배포하게 되고 결국 커뮤니티에 버전 관리에 대한 고통을 더 가져다줄 겁니다. 이것은 우리가 바라는 만큼 빠르게 React를 발전시킬 수 없다는 것을 의미하기도 합니다.

그런데도 위의 목록과 같은 변경이 커뮤니티에서 광범위한 문제를 일으키게 된다면 점진적으로 마이그레이션 할 수 있는 수단을 제공할 수 있도록 최선을 다할 것입니다.

부 버전 배포에 새로운 기능이 포함되어 있지 않다면, 왜 수 버전을 변경하지 않나요?

부 버전 배포에 새 기능이 포함되지 않을 수도 있습니다. [유의적 버전 관리에서 허용되며](#), "[부 버전]은 개인 코드에 상당한 새 기능이나 개선이 도입되는 경우 증가할 수 있다. 수 버전 수준의 변경이 포함될 수 있다."라고 기재되어 있습니다.

그렇지만, 이런 배포는 왜 수 버전으로 배포하지 않는지 의문을 제기합니다.

정답은 React (또는 다른 소프트웨어)의 변경은 예상하지 못한 방식으로 고장 낼 위험이 있습니다. 하나의 버그를 수정하는 수 버전 배포가 실수로 다른 버그를 만들어내는 시나리오를 상상해 보세요. 이것은 개발자에게 혼란을 줄 뿐만 아니라 수 버전 배포에 대한 개발자의 신뢰를 떨어뜨립니다. 특히 원래의 해결책이 거의 마주치지 않는 버그에 대한 수정이라면 더욱 유감스러운 일입니다.

우리는 React 배포를 버그 없이 지켜온 꽤 좋은 기록을 가지고 있지만, 대부분의 개발자는 수 버전 배포가 부정적인 결과 없이 적용할 수 있다고 가정하기 때문에 수 버전 배포는 안정성에 대한 기준이 훨씬 높습니다.

이런 이유로 가장 중요한 버그와 보안 취약점에 대해서만 수 버전 배포를 사용합니다.

내부 코드 리팩토링이나 구현 세부사항에 대한 변화, 성능 개선, 사소한 버그 수정처럼 핵심적이지 않은 변경이 포함된 배포라면, 새로운 기능이 없을 때라도 부 버전을 증가시킬 겁니다.

[Go to TOC](#)

폼

HTML 폼 엘리먼트는 폼 엘리먼트 자체가 내부 상태를 가지기 때문에, React의 다른 DOM 엘리먼트와 다르게 동작합니다. 예를 들어, 순수한 HTML에서 이 폼은 name을 입력받습니다.

```
<form>
  <label>
    Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

이 폼은 사용자가 폼을 제출하면 새로운 페이지로 이동하는 기본 HTML 폼 동작을 수행합니다. React에서 동일한 동작을 원한다면 그대로 사용하면 됩니다. 그러나 대부분의 경우, JavaScript 함수로 폼의 제출을 처리하고 사용자가 폼에 입력한 데이터에 접근하도록 하는 것이 편리합니다. 이를 위한 표준 방식은 "제어 컴포넌트 (controlled components)"라고 불리는 기술을 이용하는 것입니다.

제어 컴포넌트 (Controlled Component)

HTML에서 `<input>`, `<textarea>`, `<select>` 와 같은 폼 엘리먼트는 일반적으로 사용자의 입력을 기반으로 자신의 state를 관리하고 업데이트합니다. React에서는 변경할 수 있는 state가 일반적으로 컴포넌트의 state 속성에 유지되며 `setState()`에 의해 업데이트됩니다.

우리는 React state를 "신뢰 가능한 단일 출처 (single source of truth)"로 만들어 두 요소를 결합할 수 있습니다. 그러면 폼을 렌더링하는 React 컴포넌트는 폼에 발생하는 사용자 입력값을 제어합니다. 이러한 방식으로 React에 의해 값이 제어되는 입력 폼 엘리먼트를 "제어 컴포넌트 (controlled component)"라고 합니다.

예를 들어, 이전 예시가 전송될 때 이름을 기록하길 원한다면 폼을 제어 컴포넌트 (controlled component)로 작성할 수 있습니다.

```
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('A name was submitted: ' + this.state.value);
    event.preventDefault();
  }

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

```

    />
    </label>
    <input type="submit" value="Submit" />
  </form>
);
}
}
}

```

[CodePen에서 실행하기](#)

`value` 어트리뷰트는 폼 엘리먼트에 설정되므로 표시되는 값은 항상 `this.state.value`가 되고 React state는 신뢰 가능한 단일 출처 (single source of truth)가 됩니다. React state를 업데이트하기 위해 모든 키 입력에서 `handleChange`가 동작하기 때문에 사용자가 입력할 때 보여지는 값이 업데이트됩니다.

제어 컴포넌트로 사용하면, `input`의 값은 항상 React state에 의해 결정됩니다. 코드를 조금 더 작성해야 한다는 의미이지만, 다른 UI 엘리먼트에 `input`의 값을 전달하거나 다른 이벤트 핸들러에서 값을 재설정할 수 있습니다.

textarea 태그

HTML에서 `<textarea>` 엘리먼트는 텍스트를 자식으로 정의합니다.

```

<textarea>
  Hello there, this is some text in a text area
</textarea>

```

React에서 `<textarea>`는 `value` 어트리뷰트를 대신 사용합니다. 이렇게하면 `<textarea>`를 사용하는 폼은 한 줄 입력을 사용하는 폼과 비슷하게 작성할 수 있습니다.

```

class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'Please write an essay about your favorite DOM element.'
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

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

  handleSubmit(event) {
    alert('An essay was submitted: ' + this.state.value);
    event.preventDefault();
  }

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

```

```

    );
}
}
```

`this.state.value` 를 생성자에서 초기화하므로 `textarea`는 일부 텍스트를 가진채 시작되는 점을 주의해주세요.

select 태그

HTML에서 `<select>` 는 드롭 다운 목록을 만듭니다. 예를 들어, 이 HTML은 파일 드롭 다운 목록을 만듭니다.

```

<select>
  <option value="grapefruit">Grapefruit</option>
  <option value="lime">Lime</option>
  <option selected value="coconut">Coconut</option>
  <option value="mango">Mango</option>
</select>
```

`selected` 옵션이 있으므로 `Coconut` 옵션이 초기값이 되는 점을 주의해주세요. React에서는 `selected` 어트리뷰트를 사용하는 대신 최상단 `select` 태그에 `value` 어트리뷰트를 사용합니다. 한 곳에서 업데이트만 하면되기 때문에 제어 컴포넌트에서 사용하기 더 편합니다. 아래는 예시입니다.

```

class FlavorForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 'coconut'};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

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

  handleSubmit(event) {
    alert('Your favorite flavor is: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Pick your favorite flavor:
          <select value={this.state.value} onChange={this.handleChange}>
            <option value="grapefruit">Grapefruit</option>
            <option value="lime">Lime</option>
            <option value="coconut">Coconut</option>
            <option value="mango">Mango</option>
          </select>
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

CodePen에서 실행하기

전반적으로 `<input type="text">`, `<textarea>` 및 `<select>` 모두 매우 비슷하게 동작합니다. 모두 제어 컴포넌트를 구현하는데 `value` 어트리뷰트를 허용합니다.

주의

`select` 태그에 `multiple` 옵션을 허용한다면 `value` 어트리뷰트에 배열을 전달할 수 있습니다.

```
<select multiple={true} value={['B', 'C']}>
```

file input 태그

HTML에서 `<input type="file">`는 사용자가 하나 이상의 파일을 자신의 장치 저장소에서 서버로 업로드하거나 [File API](#)를 통해 JavaScript로 조작할 수 있습니다.

```
<input type="file" />
```

값이 읽기 전용이기 때문에 React에서는 [비제어 컴포넌트](#)입니다. [문서 뒷부분](#)에서 다른 비제어 컴포넌트와 함께 설명하고 있습니다.

다중 입력 제어하기

여러 `input` 엘리먼트를 제어해야 할 때, 각 엘리먼트에 `name` 어트리뷰트를 추가하고 `event.target.name` 값을 통해 핸들러가 어떤 작업을 할 지 선택할 수 있게 해줍니다.

아래는 예시입니다.

```
class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      number0fGuests: 2
    };
    this.handleInputChange = this.handleInputChange.bind(this);
  }

  handleInputChange(event) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
    const name = target.name;

    this.setState({
      [name]: value
    });
  }

  render() {
    return (
      <form>
        <label>
```

```

Is going:
<input
  name="isGoing"
  type="checkbox"
  checked={this.state.isGoing}
  onChange={this.handleInputChange} />
</label>
<br />
<label>
  Number of guests:
  <input
    name="numberOfGuests"
    type="number"
    value={this.state.numberOfGuests}
    onChange={this.handleInputChange} />
</label>
</form>
);
}
}

```

[CodePen에서 실행하기](#)

주어진 input 태그의 name에 일치하는 state를 업데이트하기 위해 ES6의 `computed property name` 구문을 사용하고 있습니다.

```

this.setState({
  [name]: value
});

```

ES5 코드는 아래와 같습니다.

```

var partialState = {};
partialState[name] = value;
this.setState(partialState);

```

또한, `setState()` 는 자동적으로 `현재 state`에 일부 `state`를 병합하기 때문에 바뀐 부분에 대해서만 호출하면 됩니다.

제어되는 Input Null 값

제어 컴포넌트에 `value` prop을 지정하면 의도하지 않는 한 사용자가 변경할 수 없습니다. `value` 를 설정했는데 여전히 수정할 수 있다면 실수로 `value` 를 `undefined` 나 `null` 로 설정했을 수 있습니다.

아래 코드가 이것을 보여줍니다. (첫 번째 입력은 잠겨있지만 잠시 후 입력이 가능해집니다.)

```

ReactDOM.createRoot(mountNode).render(<input value="hi" />);

setTimeout(function() {
  ReactDOM.createRoot(mountNode).render(<input value={null} />);
}, 1000);

```

제어 컴포넌트의 대안

데이터를 변경할 수 있는 모든 방법에 대해 이벤트 핸들러를 작성하고 React 컴포넌트를 통해 모든 입력 상태를 연결해야 하기 때문에 때로는 제어 컴포넌트를 사용하는 게 지루할 수 있습니다. 특히 기존의 코드베이스를 React로 변경하고자 할 때나 React가 아닌 라이브러리와 React 애플리케이션을 통합하고자 할 때 짜증날 수 있습니다. 이러한 경우에 입력 폼을 구현하기 위한 대체 기술인 [비제어 컴포넌트](#)를 확인할 수 있습니다.

완전한 해결책

유효성 검사, 방문한 필드 추적 및 폼 제출 처리와 같은 완벽한 해결을 원한다면 [Formik](#)이 대중적인 선택 중 하나입니다. 그러나 Formik은 제어 컴포넌트 및 state 관리에 기초하기 때문에 배우는 걸 쉽게 생각하면 안 됩니다.

[Go to TOC](#)

Ref 전달하기

`ref` 전달은 컴포넌트를 통해 자식 중 하나에 `ref`를 자동으로 전달하는 기법입니다. 일반적으로 애플리케이션 대부분의 컴포넌트에 필요하지는 않습니다. 그렇지만, 특히 재사용 가능한 컴포넌트 라이브러리와 같은 어떤 컴포넌트에서는 유용할 수 있습니다. 가장 보편적인 시나리오를 아래에 설명하겠습니다.

DOM 에 refs 전달하기

기본 `button` DOM 요소를 렌더링하는 `FancyButton` 컴포넌트를 가정해 봅시다. `embed:forwarding-refs/fancy-button-simple.js`

`FancyButton`를 사용하는 다른 컴포넌트들은 일반적으로 내부 `button` DOM 요소에 대한 `ref`를 얻을 필요가 없습니다. 이는 컴포넌트들이 서로의 DOM 구조에 지나치게 의존하지 않기 때문에 괜찮습니다.

이런 캡슐화는 `FeedStory`나 `Comment` 같은 애플리케이션 레벨의 컴포넌트에서는 바람직하지만, `FancyButton`이나 `MyTextInput`과 같은 재사용성이 높은 "말단" 요소에서는 불편할 수도 있습니다. 이런 컴포넌트들은 일반적인 DOM `button`, `input`과 유사한 방법으로 애플리케이션 전체에 걸쳐 사용되는 경향이 있습니다. 그리고 포커스, 선택, 애니메이션을 관리하기 위해서는 이런 DOM 노드에 접근하는 것이 불가피할 수 있습니다.

Ref 전달하기는 일부 컴포넌트가 수신한 `ref`를 받아 조금 더 아래로 전달(즉, "전송")할 수 있는 옵트인 기능입니다.

아래의 예에서 `FancyButton`은 `React.forwardRef`를 사용하여 전달된 `ref`를 얻고, 그것을 렌더링 되는 DOM `button`으로 전달합니다.

`embed:forwarding-refs/fancy-button-simple-ref.js`

이런 방법으로 `FancyButton`을 사용하는 컴포넌트들은 `button` DOM 노드에 대한 참조를 가져올 수 있고, 필요한 경우 DOM `button`을 직접 사용하는 것처럼 접근할 수 있습니다.

위의 예시에서 어떤 일이 일어나는지 단계별로 설명하겠습니다.

1. `React.createRef`를 호출해서 `React ref`를 생성하고 `ref` 변수에 할당합니다.
2. `ref`를 JSX 속성으로 지정해서 `<FancyButton ref={ref}>`로 전달합니다.
3. React는 이 `ref`를 `forwardRef` 내부의 `(props, ref) => ...` 함수의 두 번째 인자로 전달합니다.
4. 이 `ref`를 JSX 속성으로 지정해서 `<button ref={ref}>`으로 전달합니다.
5. `ref`가 첨부되면 `ref.current`는 `<button>` DOM 노드를 가리키게 됩니다.

알아두기

두 번째 `ref` 인자는 `React.forwardRef` 와 같이 호출된 컴포넌트를 정의했을 때에만 생성됩니다. 일반 함수나 클래스 컴포넌트는 `ref` 인자를 받지도 않고 `props`에서 사용할 수도 없습니다.

Ref 전달은 DOM 컴포넌트에만 한정적이지 않습니다. 클래스 컴포넌트 인스턴스에도 전달할 수 있습니다.

컴포넌트 라이브러리 유지관리자를 위한 주의사항

컴포넌트 라이브러리에서 `forwardRef`를 사용하기 시작할 때 이것을 변경사항으로 간주하고 라이브러리의 새로운 중요 버전을 릴리즈 해야 합니다. 이는 라이브러리에 주목할 만하게 (`ref`가 할당되는 것이 무엇이며 내보내는 유형은 무엇인가와 같은) 다른 동작을 할 가능성이 높고 이전 동작에 의존하는 앱이나 다른 라이브러리들이 손상될 가능성이 크기 때문입니다.

조건적으로 `React.forwardRef` 가 존재할 때 조건부로 적용하는 것도 같은 이유로 권장하지 않습니다: 라이브러리가 동작하는 방식을 변경하고 React 그 자체를 업데이트할 때 사용자 앱을 손상시킬 수도 있습니다.

고차원 컴포넌트에서의 `ref` 전달하기

이 기술은 (HOC로 알려진) [고차원 컴포넌트](#)에서 부분적으로 유용할 수 있습니다. 콘솔에 컴포넌트 `props`를 로깅 하는 HOC 예시로 설명을 시작해 보겠습니다. `embed:forwarding-refs/log-props-before.js`

"`logProps`" HOC는 모든 `props`를 래핑하는 컴포넌트로 전달하므로 렌더링 된 결과가 동일하게 됩니다. 예를 들어, 이 HOC를 사용해서 "`fancy button`" 컴포넌트로 전달하는 모든 `props`를 기록 할 수 있습니다. `embed:forwarding-refs/fancy-button.js`

위 예시에서 한 가지 주의사항이 있습니다: `refs`는 전달되지 않는다는 것입니다. 그것은 `ref` 는 `prop`이 아니기 때문입니다. `key` 와 마찬가지로 `ref` 는 React에서 다르게 처리합니다. HOC에 `ref`를 추가하면 `ref`는 래핑 된 컴포넌트가 아니라 가장 바깥쪽 컨테이너 컴포넌트를 참조합니다.

`FancyButton` 컴포넌트를 위한 `refs`가 실제로는 `LogProps` 컴포넌트에 첨부된다는 것을 의미합니다. `embed:forwarding-refs/fancy-button-ref.js`

다행히도 `React.forwardRef` API를 사용하여 내부 `FancyButton` 컴포넌트에 대한 `refs`를 명시적으로 전달할 수 있습니다. `React.forwardRef` 는 `props` 와 `ref` 파라미터를 받아 React 노드를 반환하는 렌더링 함수를 받습니다. 예를 들어: `embed:forwarding-refs/log-props-after.js`

DevTools에 사용자 정의 이름 표시하기

`React.forwardRef` 는 렌더링 함수를 받습니다. React DevTools는 이 함수를 사용하여 `ref` 전달 컴포넌트에 대해서 무엇을 표시 할 것인지 정의합니다.

예로, 다음의 컴포넌트는 DevTools에 "`ForwardRef`"로 나타날 것입니다.

`embed:forwarding-refs/wrapped-component.js`

렌더링 함수를 지정하면 DevTools에 해당 이름도 포함됩니다. (예, "`ForwardRef(myFunction)`")

`embed:forwarding-refs/wrapped-component-with-function-name.js`

감싸고 있는 컴포넌트를 포함하도록 함수의 `displayName` 속성을 설정할 수도 있습니다.

`embed:forwarding-refs/customized-display-name.js`

[Go to TOC](#)

Fragments

React에서 컴포넌트가 여러 엘리먼트를 반환하는 것은 흔한 패턴입니다. Fragments는 DOM에 별도의 노드를 추가하지 않고 여러 자식을 그룹화할 수 있습니다.

```
render() {
  return (
    <React.Fragment>
      <ChildA />
      <ChildB />
      <ChildC />
    </React.Fragment>
  );
}
```

이를 선언하는 새로운 단축 문법이 있습니다.

동기

컴포넌트가 여러 자식을 반환하는 것은 흔한 패턴입니다. 다음 React 예시를 보세요.

```
class Table extends React.Component {
  render() {
    return (
      <table>
        <tr>
          <Columns />
        </tr>
      </table>
    );
  }
}
```

렌더링 된 HTML이 유효하려면 `<Columns />` 가 여러 `<td>` 엘리먼트만 반환해야 합니다. `<Columns />` 의 `render()` 안에 부모 `div`로 자식들을 감싸다면 렌더링 된 HTML은 유효하지 않습니다.

```
class Columns extends React.Component {
  render() {
    return (
      <div>
        <td>Hello</td>
        <td>World</td>
      </div>
    );
  }
}
```

`<Table />` 의 출력 결과는 다음과 같습니다.

```
<table>
  <tr>
    <div>
      <td>Hello</td>
```

```

<td>World</td>
</div>
</tr>
</table>

```

Fragments는 이 문제를 해결해줍니다.

사용법

```

class Columns extends React.Component {
  render() {
    return (
      <React.Fragment>
        <td>Hello</td>
        <td>World</td>
      </React.Fragment>
    );
  }
}

```

올바른 `<Table />`의 출력 결과는 아래와 같습니다.

```

<table>
  <tr>
    <td>Hello</td>
    <td>World</td>
  </tr>
</table>

```

단축 문법

Fragments를 선언하는 더 짧고 새로운 문법이 있습니다. 마치 빈 태그와 같습니다.

```

class Columns extends React.Component {
  render() {
    return (
      <>
        <td>Hello</td>
        <td>World</td>
      </>
    );
  }
}

```

`key` 또는 어트리뷰트를 지원하지 않는다는 것을 빼고 다른 엘리먼트처럼 `<></>`을 사용할 수 있습니다.

key가 있는 Fragments

Fragments에 `key` 가 있다면 `<React.Fragment>` 문법으로 명시적으로 선언해야 합니다. 예를 들어 정의 목록을 만들기 위해 컬렉션을 fragments 배열로 맵핑하는 사용 사례입니다.

```

function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // React는 `key`가 없으면 key warning을 발생합니다.
      ))}
    </dl>
  );
}

```

```
<React.Fragment key={item.id}>
  <dt>{item.term}</dt>
  <dd>{item.description}</dd>
</React.Fragment>
);
});
```

`key` 는 `Fragment` 에 전달할 수 있는 유일한 어트리뷰트입니다. 추후 이벤트 핸들러와 같은 추가적인 어트리뷰트를 지원할 수도 있습니다.

라이브 데모

[CodePen](#)에서 새로운 JSX fragment 문법을 사용해 볼 수 있습니다.

시작하기

- "download.html"
- "downloads.html"

이 페이지는 React 문서 및 관련된 리소스의 개요입니다.

React는 사용자 인터페이스를 만들기 위한 JavaScript 라이브러리입니다. [홈페이지](#)나 [자습서](#)에서 React에 대한 모든 것을 알아보세요.

- [React 시도해보기](#)
- [React 배우기](#)
- [최신 React 소식](#)
- [버전 관리된 문서](#)
- [빠진 항목이 있다면?](#)

React 시도해보기

React는 처음부터 점진적으로 적용할 수 있도록 설계되었으며 필요한 만큼 **React**를 사용할 수 있습니다. React를 맛보기로 접해보거나 간단한 HTML 페이지에 약간의 대화형 기능(interactivity)을 추가하거나 복잡한 React 기반의 앱을 시작하고자 하는 경우, 이 섹션의 링크를 통해 시작할 수 있습니다.

온라인 플레이그라운드

React를 사용하는데 관심이 있다면, 온라인 코드 편집기를 사용할 수 있습니다. [CodePen](#), [CodeSandbox](#) 또는 [Stackblitz](#)에서 Hello World 템플릿을 사용해 보세요.

여러분이 사용하고 있는 코드 편집기를 사용하길 원한다면, [이 HTML 파일을 다운로드](#)하고 편집한 다음 브라우저의 로컬 파일 시스템에서 열 수도 있습니다. 런타임 코드 변환이 느리므로 간단한 데모에만 이 코드를 사용하는 것이 좋습니다.

웹사이트에 React를 추가하기

[1분 안에 HTML 페이지에 React를 추가할 수 있습니다.](#) 그리고 조금씩 React의 비중을 늘리거나 몇 개의 동적 위젯에 포함할 수 있습니다.

새 React 앱 만들기

React 프로젝트를 시작한다면 [스크립트 태그를 사용한 간단한 HTML 페이지](#)를 만드는 것이 최고의 방법일 것입니다. 설치하는 데 1분밖에 걸리지 않습니다!

그러나 애플리케이션이 커진다면 보다 통합된 설정을 고려하는 것이 좋습니다. 대규모 애플리케이션에 권장되는 여러 개의 JavaScript 툴체인들이 있습니다. 각 툴체인은 많은 설정 없이 작동할 수 있고 풍부한 React 에코시스템을 최대한 활용할 수 있습니다. [어떻게 하는지 배워보세요.](#)

React 배우기

사람들은 다양한 배경과 다른 학습 스타일을 가지고 React에 입문합니다. 이론적인 접근 방식을 선호하는 사람들과 실용적인 접근 방식을 선호하는 사람들 모두에게 이 섹션이 도움되길 바랍니다.

- 직접 구현해보면서 학습하는 것을 원하시는 경우, [실용적인 자습서](#)부터 시작하세요.
- 개념을 차근차근 익히며 학습하는 것을 원하시는 경우, [주요 개념 가이드](#)부터 시작하세요.

다른 낯선 기술과 마찬가지로 React는 러닝 커브(learning curve)가 있습니다. 약간의 인내심을 가지고 연습하면, React에 대한 요령을 익힐 수 있습니다.

첫 번째 예시

[React 홈페이지](#)에는 라이브 편집기와 함께 제공되는 몇 가지 React 예시가 있습니다. React에 대해 전혀 모르더라도 코드를 변경하면서 그것이 결과에 어떤 영향을 미치는지 보실 수 있습니다.

입문자를 위한 React

React 문서가 어렵게 느껴진다면, [Tania Rascia가 쓴 React 개요](#)를 먼저 보고 오시는 것이 좋습니다. 위 글은 가장 중요한 React 개념들을 자세하고 입문자도 쉽게 이해할 수 있도록 소개합니다. 그다음에 이 문서를 다시 봄주세요!

디자이너를 위한 React

디자인 분야에서 일하고 있다면 [이 리소스](#)가 시작하기 좋습니다.

JavaScript 리소스

React 문서는 JavaScript 프로그래밍에 익숙한 사람을 대상으로 쓰여져 있습니다. 전문가일 필요는 없지만, React와 JavaScript를 동시에 배우기는 어렵습니다.

[이 JavaScript 개요](#)를 읽고 여러분의 JavaScript 수준을 확인하는 것을 권장합니다. 30분에서 1시간 정도 걸리겠지만 React를 익히는 데에 더 자신감이 생길 것입니다.

팁

JavaScript 내용이 혼동될 때마다, [MDN](#)과 [javascript.info](#)는 참고하기 좋은 웹사이트입니다. 또한, 문의를 할 수 있는 [커뮤니티 지원 포럼](#)도 있습니다.

실용적인 자습서

직접 구현해보면서 학습하는 것을 원하시면, [실용적인 자습서](#)를 살펴보세요. 본 자습서에서는 React로 tic-tac-toe 게임을 만들어 봅니다. 게임 제작에는 관심이 없어서 이 자습서를 건너뛰고 싶은 마음이 들지도 모릅니다. 하지만 조금만 기다려 보세요. 자습서에서 학습하는 기술은 [대부분의 React](#) 앱을 구축하는 데 있어 기본이며, 이를 완전히 익히면 훨씬 더 깊은 이해할 수 있습니다.

단계별 가이드

개념을 단계별로 배우려면 [주요 개념 가이드](#)를 시작하는 것이 가장 좋습니다. 모든 다음 장은 앞 장에 소개된 지식을 바탕으로 하기 때문에, 천천히 따라가기만 한다면 개념을 놓치지 않고 습득할 수 있습니다.

React에 대해 생각해 보기

많은 React 사용자는 [React에 대해 생각해보기](#)를 읽는 것을 React를 마지막으로 이해하는 것으로 생각합니다. 이것은 아마 가장 오래 된 React 문서이지만 여전히 유용합니다.

권장 과정

사람들은 때때로 공식 문서보다 타사 책과 비디오 강좌가 더 도움이 된다고 생각합니다. 우리는 [일반적으로 권장되는 리소스 목록](#)을 유지하며, 이 중 일부는 무료입니다.

고급 개념

[주요 개념](#)에 익숙해지고 React를 더 많이 다루게 되면 보다 고급 주제에 관심을 가지게 될 것입니다. 이 섹션에서는 [context](#) 및 [refs](#)와 같이 강력하지만 일반적으로 사용되지는 않는 React 기능을 소개합니다.

API 참조

이 문서 섹션은 특정 React API를 자세히 알아보고 싶을 때 유용합니다. 예를 들어, [React.Component API 참조](#)에서는 `setState()`의 작동 방식과 각 상황에 알맞는 생명주기 메서드를 알 수 있습니다.

용어 사전과 FAQ

[용어 사전](#)에는 React 문서에 나타나는 가장 일반적인 용어의 개요가 포함되어 있습니다. 또한 [AJAX 요청 만들기](#), [컴포넌트 상태](#), [파일 구조](#) 등 일반적인 주제에 대한 짧은 질문과 답변을 전문으로 하는 FAQ 섹션도 있습니다.

최신 React 소식

[React 블로그](#)는 React 팀의 업데이트에 대한 공식 소스입니다. 릴리즈 노트 또는 사용자제 공지를 포함한 중요한 내용이 먼저 게시됩니다.

트위터의 [@reactjs 계정](#)을 팔로우할 수도 있지만, 블로그만 읽어도 꼭 알아야 하는 정보는 놓치지 않을 수 있습니다.

모든 React 릴리즈가 자체 블로그 게시물을 사용할 만한 것은 아니지만 [릴리즈](#) 페이지와 React의 [CHANGELOG.md 파일](#)에서 모든 릴리즈에 대한 자세한 변경 기록을 확인할 수 있습니다.

버전 관리된 문서

이 문서는 항상 안정된 최신 버전의 React를 반영합니다. React 16부터는 이전 버전의 문서를 [별도의 페이지](#)에서 찾을 수 있습니다. 이전 버전의 문서는 릴리즈 시점에 스냅샷으로 작성되며 지속적으로 업데이트되지 않는다는 점에 유의하세요.

빠진 항목이 있다면?

이 문서에서 빠진 부분이나 혼동되는 부분을 발견하시면, 개선을 위한 제안과 함께 [문서의 레퍼지토리](#)에 이슈를 올려주시거나, [@reactjs 계정](#)에 tweet 해주세요. 여러분의 참여를 기다립니다!

이벤트 처리하기

React 엘리먼트에서 이벤트를 처리하는 방식은 DOM 엘리먼트에서 이벤트를 처리하는 방식과 매우 유사합니다. 몇 가지 문법 차이는 다음과 같습니다.

- React의 이벤트는 소문자 대신 캐멀 케이스(camelCase)를 사용합니다.
- JSX를 사용하여 문자열이 아닌 함수로 이벤트 핸들러를 전달합니다.

예를 들어, HTML은 다음과 같습니다.

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

React에서는 약간 다릅니다.

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

또 다른 차이점으로, React에서는 `false`를 반환해도 기본 동작을 방지할 수 없습니다. 반드시 `preventDefault`를 명시적으로 호출해야 합니다. 예를 들어, 일반 HTML에서 폼을 제출할 때 가지고 있는 기본 동작을 방지하기 위해 다음과 같은 코드를 작성할 수 있습니다.

```
<form onSubmit="console.log('You clicked submit.'); return false">
  <button type="submit">Submit</button>
</form>
```

React에서는 다음과 같이 작성할 수 있습니다.

```
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('You clicked submit.');
  }

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

여기서 `e`는 합성 이벤트입니다. React는 [W3C 명세](#)에 따라 합성 이벤트를 정의하기 때문에 브라우저 호환성에 대해 걱정할 필요가 없습니다. React 이벤트는 브라우저 고유 이벤트와 정확히 동일하게 동작하지는 않습니다. 더 자세한 사항은 [합성 이벤트](#)를 참고하시기 바랍니다.

React를 사용할 때 DOM 엘리먼트가 생성된 후 리스너를 추가하기 위해 `addEventListener`를 호출할 필요가 없습니다. 대신, 엘리먼트가 처음 렌더링될 때 리스너를 제공하면 됩니다.

ES6 클래스를 사용하여 컴포넌트를 정의할 때, 일반적인 패턴은 이벤트 핸들러를 클래스의 메서드로 만드는 것입니다. 예를 들어, 다음 Toggle 컴포넌트는 사용자가 "ON"과 "OFF" 상태를 토글 할 수 있는 버튼을 렌더링합니다.

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // 콜백에서 `this`가 작동하려면 아래와 같이 바인딩 해주어야 합니다.
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(prevState => ({
      isToggleOn: !prevState.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}
```

CodePen에서 실행하기

JSX 콜백 안에서 `this`의 의미에 대해 주의해야 합니다. JavaScript에서 클래스 메서드는 기본적으로 **바인딩**되어 있지 않습니다. `this.handleClick`을 바인딩하지 않고 `onClick`에 전달하였다면, 함수가 실제 호출될 때 `this`는 `undefined`가 됩니다.

이는 React만의 특수한 동작이 아니며, [JavaScript에서 함수가 작동하는 방식](#)의 일부입니다. 일반적으로 `onClick={this.handleClick}`과 같이 뒤에 `()`를 사용하지 않고 메서드를 참조할 경우, 해당 메서드를 바인딩 해야 합니다.

`bind`를 호출하는 것이 불편하다면, 이를 해결할 수 있는 두 가지 방법이 있습니다. 콜백을 올바르게 바인딩하기 위해 [퍼블릭 클래스 필드 문법](#)을 활용할 수 있다.

```
class LoggingButton extends React.Component {
  // 이 문법은 `this`가 handleClick 내에서 바인딩되도록 합니다.
  // 주의: 이 문법은 *실험적인* 문법입니다.
  handleClick = () => {
    console.log('this is:', this);
  }

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

Create React App에서는 이 문법이 기본적으로 설정되어 있습니다.

클래스 필드 문법을 사용하고 있지 않다면, 콜백에 [화살표 함수](#)를 사용하는 방법도 있습니다.

```

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

  render() {
    // 이 문법은 `this`가 handleClick 내에서 바인딩되도록 합니다.
    return (
      <button onClick={() => this.handleClick()}>
        Click me
      </button>
    );
  }
}

```

이 문법의 문제점은 `LoggingButton`이 렌더링될 때마다 다른 콜백이 생성된다는 것입니다. 대부분의 경우 문제가 되지 않으나, 콜백이 하위 컴포넌트에 `props`로서 전달된다면 그 컴포넌트들은 추가로 다시 렌더링을 수행할 수도 있습니다. 이러한 종류의 성능 문제를 피하고자, 생성자 안에서 바인딩하거나 클래스 필드 문법을 사용하는 것을 권장합니다.

이벤트 핸들러에 인자 전달하기

루프 내부에서는 이벤트 핸들러에 추가적인 매개변수를 전달하는 것이 일반적입니다. 예를 들어, `id` 가 행의 ID일 경우 다음 코드가 모두 작동합니다.

```

<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>

```

위 두 줄은 동등하며 각각 [화살표 함수](#)와 `Function.prototype.bind`를 사용합니다.

두 경우 모두 React 이벤트를 나타내는 `e` 인자가 ID 뒤에 두 번째 인자로 전달됩니다. 화살표 함수를 사용하면 명시적으로 인자를 전달해야 하지만 `bind` 를 사용할 경우 추가 인자가 자동으로 전달됩니다.

[Go to TOC](#)

Hello World

가장 단순한 React 예시는 다음과 같이 생겼습니다.

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<h1>Hello, world!</h1>);
```

위 코드는 페이지에 "Hello, world!"라는 제목을 보여줍니다.

[CodePen에서 실행하기](#)

온라인 에디터로 열어보려면 상단의 링크를 클릭하세요. 코드를 자유롭게 수정하고, 결과가 어떻게 변하는지 관찰해보세요. 이 안내서의 거의 모든 페이지에서는 이런 형태로 수정 가능한 예시를 함께 제공합니다.

이 안내서를 읽는 방법

이 안내서에서 우리는 엘리먼트와 컴포넌트라고 불리는 React 앱의 구성 블록들에 대해 살펴볼 것입니다. 이들을 숙달하고 나면 재사용 가능한 조각들로부터 복잡한 앱을 만들 수도 있습니다.

팁

이 안내서는 **개념들을 단계적으로 배우길 원하는 사람들을 대상으로 합니다.** 실습을 통해 배우길 원한다면, [실전 튜토리얼](#)을 확인하세요. 이 안내서와 실전 튜토리얼은 상호보완적인 내용을 담고 있습니다.

이 페이지는 React 주요 개념에 대한 단계별 안내서의 첫 번째 장입니다. 내비게이션 사이드바에서 모든 장의 목록을 확인할 수 있습니다. 모바일 기기에서 읽고 있다면, 화면 오른쪽 하단 끝에 있는 버튼을 클릭하여 내비게이션에 접근할 수 있습니다.

이 안내서의 모든 장들은 이전 장의 내용을 토대로 작성되었습니다. 사이드바에 보이는 순서대로 "**주요 개념**" 안내서의 장들을 읽으면 **React의 거의 모든 내용을 익힐 수 있습니다.** "[JSX 소개](#)"가 바로 이 다음 장입니다.

지식수준 가정

React는 JavaScript 라이브러리이며, 따라서 JavaScript 언어에 대한 기본적인 이해가 필요합니다. **아직 자신이 없다면, [JavaScript 튜토리얼 살펴보기](#)를 통해 자신의 지식수준을 확인해보길 권장드리며** 이를 통해 길을 잊지 않고 이 안내서를 잘 따라올 수 있게 될 것입니다. 30분에서 1시간 가량 소요되지만, 결과적으로 더 이상 React와 JavaScript를 동시에 배운다는 느낌을 받지 않을 수 있습니다.

주의

이 가이드에서는 가끔 최신 JavaScript 문법을 예시에 사용합니다. 지난 몇 년간 JavaScript로 작업하지 않았다면, [0| 3가지 사항](#)을 통해 대부분의 것을 얻을 수 있습니다.

시작해봅시다!

아래로 계속 스크롤하면, 푸터 바로 앞에서 [다음 장](#)으로 연결된 링크를 찾을 수 있습니다.

[Go to TOC](#)

고차 컴포넌트

고차 컴포넌트(HOC, Higher Order Component)는 컴포넌트 로직을 재사용하기 위한 React의 고급 기술입니다. 고차 컴포넌트(HOC)는 React API의 일부가 아니며, React의 구성적 특성에서 나오는 패턴입니다.

구체적으로, 고차 컴포넌트는 컴포넌트를 가져와 새 컴포넌트를 반환하는 함수입니다.

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

컴포넌트는 props를 UI로 변환하는 반면에, 고차 컴포넌트는 컴포넌트를 새로운 컴포넌트로 변환합니다.

고차 컴포넌트(HOC)는 Redux의 `connect` 와 Relay의 `createFragmentContainer` 와 같은 서드 파티 React 라이브러리에서 흔하게 볼 수 있습니다.

이 문서에서는 고차 컴포넌트가 유용한 이유를 보여주고, 직접 작성하는 방법에 대해 알아보겠습니다.

횡단 관심사(Cross-Cutting Concerns)에 고차 컴포넌트 사용하기

주의

이전에는 횡단 관심사 문제를 제어하기 위해 mixin 사용을 권장했습니다. 하지만 mixin을 사용하는 것은 더 많은 문제를 일으킨다는 것을 알게 되었습니다. 우리가 mixin을 더 이상 권장하지 않는 이유와 기존 컴포넌트를 어떻게 변환하는지에 대해서 [이 글](#)을 읽어보세요.

컴포넌트는 React에서 코드 재사용의 기본 단위입니다. 그러나 어떤 패턴은 기존 컴포넌트에 잘 적용되지 않을 수 있습니다.

외부로부터 데이터를 구독하여 댓글 목록을 렌더링하는 `CommentList` 컴포넌트를 예로 들겠습니다.

```
class CommentList extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      // "DataSource" 는 글로벌 데이터 소스입니다.
      comments: DataSource.getComments()
    };
  }

  componentDidMount() {
    // 변화감지를 위해 리스너를 추가합니다.
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    // 리스너를 제거합니다.
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    // 데이터 소스가 변경될때마다 comments를 업데이트합니다.
  }
}
```

```

    this.setState({
      comments: DataSource.getComments()
    });

    render() {
      return (
        <div>
          {this.state.comments.map((comment) => (
            <Comment comment={comment} key={comment.id} />
          ))}
        </div>
      );
    }
}

```

그리고 블로그 포스트를 구독하기 위해 위와 비슷한 패턴으로 컴포넌트를 작성합니다.

```

class BlogPost extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      blogPost: DataSource.getBlogPost(props.id)
    };
  }

  componentDidMount() {
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    this.setState({
      blogPost: DataSource.getBlogPost(this.props.id)
    });
  }

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

```

`CommentList` 와 `BlogPost` 컴포넌트는 동일하지 않습니다. 두 컴포넌트는 `DataSource`에서 서로 다른 메서드를 호출하며 다른 렌더링 결과를 보여줍니다. 하지만 대부분의 구현체는 동일합니다.

- 컴포넌트가 마운트되면, `change` 리스너를 `DataSource`에 추가합니다.
- 리스너 안에서, 데이터 소스가 변경되면 `setState`를 호출합니다.
- 컴포넌트가 마운트 해제되면 `change` 리스너를 제거합니다.

규모가 큰 애플리케이션에서 `DataSource`를 구독하고 `setState`를 호출하는 동일한 패턴이 반복적으로 발생한다고 가정해봅시다. 그렇게 된다면 이 로직을 한 곳에서 정의하고 많은 컴포넌트에서 로직을 공유할 수 있게 하는 추상화가 필요하게 됩니다. 이러한 경우에 고차 컴포넌트를 사용하면 좋습니다.

`DataSource` 를 구독하는 `CommentList` 나 `BlogPost` 같은 컴포넌트를 생성하는 함수를 작성할 수 있습니다. 구독한 데이터를 `prop`으로 전달받는 자식 컴포넌트를 파라미터 중 하나로 받는 함수를 만듭니다. 이 함수를 `withSubscription` 라고 합니다.

```
const CommentListWithSubscription = withSubscription(
  CommentList,
  (DataSource) => DataSource.getComments()
);

const BlogPostWithSubscription = withSubscription(
  BlogPost,
  (DataSource, props) => DataSource.getBlogPost(props.id)
);
```

첫 번째 파라미터는 래핑된 컴포넌트입니다. 두 번째 파라미터에는 `DataSource` 와 현재 `props`를 가지고 컴포넌트에서 관심 있는 데이터를 검색합니다.

`CommentListWithSubscription` 과 `BlogPostWithSubscription` 가 렌더링될 때 `CommentList` 와 `BlogPost` 는 `DataSource` 에서 가장 최근에 검색된 데이터를 `data` `prop`으로 전달합니다.

```
// 이 함수는 컴포넌트를 매개변수로 받고..
function withSubscription(WrappedComponent, selectData) {
  // ... 다른 컴포넌트를 반환하는데...
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.state = {
        data: selectData(DataSource, props)
      };
    }

    componentDidMount() {
      // ... 구독을 담당하고...
      DataSource.addChangeListener(this.handleChange);
    }

    componentWillUnmount() {
      DataSource.removeChangeListener(this.handleChange);
    }

    handleChange() {
      this.setState({
        data: selectData(DataSource, this.props)
      });
    }

    render() {
      // ... 래핑된 컴포넌트를 새로운 데이터로 렌더링 합니다!
      // 컴포넌트에 추가로 props를 내려주는 것에 주목하세요.
      return <WrappedComponent data={this.state.data} {...this.props} />;
    }
  };
}
```

고차 컴포넌트는 입력된 컴포넌트를 수정하지 않으며 상속을 사용하여 동작을 복사하지도 않습니다. 오히려 고차 컴포넌트는 원본 컴포넌트를 컨테이너 컴포넌트로 포장(*Wrapping*)하여 조합(*compose*)합니다. 고차 컴포넌트는 사이드 이펙트가 전혀 없는 순수 함수입니다.

이게 전부입니다! 래핑된 컴포넌트는 새로운 props, `data` 와 함께 컨테이너의 모든 props를 전달받으며, 이 데이터들은 출력을 렌더링하는 데 사용됩니다. 고차 컴포넌트는 데이터가 사용되는 이유 및 방법과 연관이 없으며 래핑된 컴포넌트는 데이터가 어디서부터 왔는지와 관련이 없습니다.

`withSubscription` 는 일반 함수이기 때문에 원하는 개수의 인수를 추가할 수 있습니다. 예를 들어 래핑된 컴포넌트로부터 고차 컴포넌트를 더 격리시키기 위해 `data` prop 이름을 설정할 수 있게 만들 수 있습니다. 혹은 `shouldComponentUpdate` 설정을 위한 인수를 받게 하거나 데이터 소스를 설정하는 인수를 받게할 수도 있습니다. 고차 컴포넌트가 컴포넌트 정의 방법을 완전히 제어할 수 있기 때문에 이런 작업이 모두 가능합니다.

컴포넌트와 마찬가지로 `withSubscription` 과 래핑된 컴포넌트 간 계약(contract)은 완전히 props 기반입니다. 이렇게하면 래핑된 컴포넌트에 동일한 props를 제공한다면 다른 고차 컴포넌트를 쉽게 변경할 수 있습니다. 예를 들어 데이터를 가져오는 라이브러리를 변경하는 경우 유용하게 사용할 수 있습니다.

원본 컴포넌트를 변경하지 마세요. 조합(Composition)하세요.

고차 컴포넌트 내부에서 컴포넌트의 프로토타입을 수정(또는 변경)하지 않도록 합니다.

```
function logProps(Component) {
  Component.prototype.componentDidUpdate = function(prevProps) {
    console.log('Current props: ', this.props);
    console.log('Previous props: ', prevProps);
  };
  // 원본의 입력을 반환한다는 것은 이미 변형되었다는 점을 시사합니다.
  return Component;
}

// EnhancedComponent 는 props를 받을 때마다 log를 남깁니다.
const EnhancedComponent = logProps(Component);
```

여기엔 몇 가지의 문제가 있습니다. 그 중 하나는 입력된 컴포넌트를 확장된(enhanced) 컴포넌트와 별도로 재사용 할 수 없다는 것입니다. 더 중요한 것은, `componentDidUpdate` 를 변형하는 `EnhancedComponent` 에 또 다른 HOC를 적용하면 첫 번째 HOC의 기능은 무시됩니다! 이 HOC는 생명주기 메서드가 없는 함수 컴포넌트에서도 작동하지 않습니다.

변경(mutation)된 HOC는 누출된 추상화(leaky abstraction)입니다. Consumer는 다른 HOC와의 충돌을 피하기 위하여 어떻게 구현되어있는지 반드시 알아야 합니다.

HOC는 변경(mutation)대신에 입력 컴포넌트를 컨테이너 구성요소로 감싸서 조합(composition)을 사용해야 합니다.

```
function logProps(WrappedComponent) {
  return class extends React.Component {
    componentDidUpdate(prevProps) {
      console.log('Current props: ', this.props);
      console.log('Previous props: ', prevProps);
    }
    render() {
      // 들어온 component를 변경하지 않는 container입니다. 좋아요!
      return <WrappedComponent {...this.props} />;
    }
  }
}
```

위 고차 컴포넌트는 총돌 가능성을 피하면서 프로토타입을 직접 변경하는 버전과 동일하게 작동합니다. 이 방법은 클래스 컴포넌트와 함수 컴포넌트에서도 동일하게 작동합니다. 그리고 순수한 함수이기 때문에 다른 고차 컴포넌트와 같이 조합하거나 심지어 자체적으로 조합할 수 있습니다.

고차 컴포넌트와 컨테이너 컴포넌트라 불리는 패턴이 유사하다고 느낄 수 있습니다. 컨테이너 컴포넌트는 **high-level**과 **low-level** 관심사를 분리하는 전략 중 하나입니다. 컨테이너는 구독 및 **state** 같은 것을 관리하고 UI 렌더링 같은 것을 처리하는 컴포넌트에 **props**를 전달합니다. 고차 컴포넌트는 컨테이너를 그 구현체 중 일부에 사용하고 있습니다. 고차 컴포넌트는 매개변수화된 컨테이너 컴포넌트 정의로 생각할 수 있습니다.

컨벤션: 래핑된 컴포넌트를 통해 관련없는 **Props** 전달하기

고차 컴포넌트는 컴포넌트에 기능을 추가합니다. 고차 컴포넌트는 정의(**contract**)를 과감하게 변경해서는 안됩니다. 고차 컴포넌트에서 반환된 컴포넌트는 래핑된 컴포넌트와 비슷한 인터페이스가 있어야합니다.

고차 컴포넌트는 특정 관심사와 관련이 없는 **props**를 활용해야 합니다. 대부분의 고차 컴포넌트에는 다음과 같은 렌더링 메서드가 포함되어 있습니다.

```
render() {
  // 이 HOC에만 해당되므로 추가된 props는 걸러내어 이 HOC에 전달되지 않도록 합니다.
  const { extraProp, ...passThroughProps } = this.props;

  // 이 Props는 일반적으로 Status값 또는 Instance method 입니다.
  const injectedProp = someStateOrInstanceMethod;

  // wrapped component에 props를 전달합니다.
  return (
    <WrappedComponent
      injectedProp={injectedProp}
      {...passThroughProps}
    />
  );
}
```

이 컨벤션은 고차 컴포넌트의 유연성과 재사용성을 보장하는데 도움이 됩니다.

컨벤션: 조합 가능성(**Composability**) 끌어올리기

고차 컴포넌트는 여러 가지 방법으로 작성할 수 있습니다. 때때로 단일 인수로 래핑된 컴포넌트만 받을 때도 있습니다.

```
const NavbarWithRouter = withRouter(Navbar);
```

일반적으로 고차 컴포넌트는 추가 인수를 허용합니다. Relay 예시에서 config 객체는 컴포넌트의 데이터 의존성을 지정하기 위해 사용합니다.

```
const CommentWithRelay = Relay.createContainer(Comment, config);
```

고차 컴포넌트에 대한 가장 일반적인 사용은 다음과 같습니다.

```
// React Redux의 `connect`
const ConnectedComment = connect(commentSelector, commentActions)(CommentList);
```

주목! 위 코드를 분해해보면 어떤 동작을 하는지 쉽게 알 수 있습니다.

```
// connect는 다른 함수를 반환하는 함수입니다.
const enhance = connect(commentListSelector, commentListActions);
// 반환된 함수는 Redux store에 연결된 컴포넌트를 반환하는
// 고차 함수 컴포넌트입니다.
const ConnectedComment = enhance(CommentList);
```

다르게 말하면 `connect` 는 고차 컴포넌트를 반환하는 고차 함수입니다.

이 형태는 혼란스럽거나 불필요하게 보일 수 있지만 매우 유용한 속성입니다. `connect` 함수에 의해 반환된 것과 같은 단일 인수 고차 컴포넌트는 `Component => Component` 특징을 가지고 있습니다. 출력 타입이 입력 타입과 동일한 함수는 정말 쉽게 조합할 수 있습니다.

```
// 이렇게 하는 대신에...
const EnhancedComponent = withRouter(connect(commentSelector)(WrappedComponent))

// ... 함수 구성 유ти리티를 사용할 수 있습니다.
// compose(f, g, h)는 (...args) => f(g(h(...args)))와 같습니다.
const enhance = compose(
  // 둘 다 단일 매개변수의 HOC입니다.
  withRouter,
  connect(commentSelector)
)
const EnhancedComponent = enhance(WrappedComponent)
```

(이와 동일한 속성을 통해서 실험단계인 `connect` 와 기타 인해서 스타일의 HOC를 데코레이터로 사용할 수 있습니다.)

`compose` 유ти리티 기능(효용 함수)은 `lodash` (as `lodash.flowRight`), `Redux`, and `Ramda`를 포함한 많은 서드 파티 라이브러리에서 제공하고 있습니다.

컨벤션: 간단한 디버깅을 위한 디스플레이 네임 작성 방법

다른 구성 요소와 마찬가지로 HOC로 만든 컨테이너 구성 요소도 [React Developer Tools](#)에 표시됩니다. 디버깅을 쉽게 하려면 HOC의 결과임을 알리는 디스플레이 네임을 작성합니다.

가장 일반적인 방법은 HOC의 이름으로 내부 컴포넌트명을 감싸는 것입니다. 따라서 HOC의 이름이 `withSubscription`이고, HOC 내부의 컴포넌트의 이름이 `CommentList` 인 경우, 디스플레이 네임은 `WithSubscription(CommentList)` 을 사용합니다.

```
function withSubscription(WrappedComponent) {
  class WithSubscription extends React.Component {/* ... */}
  WithSubscription.displayName =
`WithSubscription(${getDisplayName(WrappedComponent)})`;
  return WithSubscription;
}

function getDisplayName(WrappedComponent) {
  return WrappedComponent.displayName || WrappedComponent.name || 'Component';
}
```

주의사항

고차 컴포넌트에는 React가 처음이라면 알아차리기 어려운 몇 가지 주의사항이 있습니다.

render 메서드 안에서 고차 컴포넌트를 사용하지 마세요

재조정(reconciliation)으로 알려진 React의 비교 알고리즘은 컴포넌트의 개별성(identity)을 가지고 기존 서브트리를 업데이트 해야 하는지 아니면 버리고 새로운 노드를 마운트 해야 할지 결정합니다. `render`에서 반환된 컴포넌트가 이전에 렌더링 된 컴포넌트와 동일하다면(`==`) React가 새로운 서브트리와 비교하여 재귀적으로 서브트리를 업데이트합니다. 동일하지 않다면 이전 서브트리는 완전히 마운트 해제됩니다.

일반적으로 위 내용에 대해 생각할 필요는 없습니다. 하지만 컴포넌트의 `render` 메서드 안에서 고차 컴포넌트를 사용할 수 없기 때문에 고차 컴포넌트를 사용할 때는 위 내용을 짚고 넘어가야 합니다.

```
render() {
  // render가 호출될 때마다 새로운 버전의 EnhancedComponent가 생성됩니다.
  // EnhancedComponent1 !== EnhancedComponent2
  const EnhancedComponent = enhance(MyComponent);
  // 때문에 매번 전체 서브트리가 마운트 해제 후 다시 마운트 됩니다!
  return <EnhancedComponent />;
}
```

여기서 성능상의 문제뿐만 아니라 컴포넌트가 다시 마운트 되면서 컴포넌트의 `state`와 컴포넌트의 하위 항목들이 순실됩니다.

대신에 컴포넌트의 정의 바깥에 HOC를 적용하여 컴포넌트가 한 번만 생성되도록 합니다. 그러면 해당 component는 여러번 렌더링이 되더라도 일관성을 유지합니다. 일반적으로 렌더링이 여러번 되어도 바뀌길 원하는 사람은 없을 것이라고 생각합니다.

드문 경우로 HOC를 동적으로 적용해야 할 경우에는 컴포넌트의 생명주기 메서드 또는 생성자 내에 작성 할 수 있습니다.

정적 메서드는 반드시 따로 복사하세요

React 컴포넌트에 정적 메서드를 정의하는 것이 유용할 때도 있습니다. 예를 들어 Relay 컨테이너는 GraphQL 구성을 용이하게 하기 위해 정적 메서드 `getFragment`를 노출합니다.

컴포넌트에 HOC를 적용하면, 기존 컴포넌트는 컨테이너의 컴포넌트로 감싸집니다. 즉, 새 컴포넌트는 기존 컴포넌트의 정적 메서드를 가지고 있지 않습니다.

```
// 정적 함수를 정의합니다
WrappedComponent.staticMethod = function() {/*...*/}
// HOC를 적용합니다
const EnhancedComponent = enhance(WrappedComponent);

// 향상된 컴포넌트에는 정적 메서드가 없습니다.
typeof EnhancedComponent.staticMethod === 'undefined' // true
```

이 문제를 해결하려면 메서드를 반환하기 전에 컨테이너에 복사합니다.

```
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  // 복사 할 메서드를 정확히 알아야 합니다.
  Enhance.staticMethod = WrappedComponent.staticMethod;
  return Enhance;
}
```

그러나 복사해야 할 메서드를 정확히 알아야 할 필요가 있습니다. [hoist-non-react-statics](#)를 사용하여 모든 non-React 정적 메서드를 자동으로 복사할 수 있습니다.

```
import hoistNonReactStatic from 'hoist-non-react-statics';
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  hoistNonReactStatic(Enhance, WrappedComponent);
  return Enhance;
}
```

또 다른 해결 방법은 정적 메서드를 컴포넌트와 별도로 내보내는 것입니다.

```
// 대신에...
MyComponent.someFunction = someFunction;
export default MyComponent;

// ...메서드를 각각 내보내고...
export { someFunction };

// ...불러오는 모듈에서 두개를 다 임포트합니다.
import MyComponent, { someFunction } from './MyComponent.js';
```

ref는 전달되지 않는다

고차 컴포넌트는 모든 props를 래핑된 컴포넌트에 전달하는 것이 원칙이지만, refs에서는 작동하지 않습니다. 이는 React에서 `ref`가 실제 prop이 아닌 `key`처럼 특별하게 취급되기 때문입니다. 컴포넌트가 HOC의 결과인 엘리먼트에 ref를 추가하는 경우, ref는 래핑된 컴포넌트가 아닌 가장 바깥쪽 컨테이너 컴포넌트의 인스턴스를 나타냅니다.

이 문제의 해결 방법은 `React.forwardRef` API를 사용하는 것입니다. (React 16.3에 도입됨) [자세한 내용은 Forwarding Refs 섹션을 참조](#).

[Go to TOC](#)

자신만의 Hook 만들기

Hook은 React 16.8 버전에 새로 추가되었습니다. Hook은 클래스 컴포넌트를 작성하지 않아도 state와 같은 특징들을 사용할 수 있습니다.

자신만의 Hook을 만들면 컴포넌트 로직을 함수로 뽑아내어 재사용할 수 있습니다.

[Effect Hook 사용하기](#)를 배울 때, 채팅 애플리케이션에서 친구가 온라인 상태인지 아닌지에 대한 메시지를 표시하는 컴포넌트를 보았을 것입니다.

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

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

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

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

이제 채팅 애플리케이션에 연락처 목록이 있으며 그중에서 온라인 상태인 사용자들의 이름을 초록색으로 표시하는 상황을 가정해 보겠습니다. 위의 코드와 비슷한 로직을 복사하여 `FriendListItem` 컴포넌트 안에 붙여넣을 수도 있지만, 가장 좋은 방법이라고 할 수는 없습니다.

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

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

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

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

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

```

        </li>
    );
}

```

그 대신 이 로직을 `FriendStatus` 와 `FriendListItem` 에서 공유하도록 하고 싶습니다.

React에는 상태 관련 로직을 컴포넌트에서 공유하는 두 가지 전통적인 방법이 있는데 `render props`와 `고차 컴포넌트`입니다. Hook을 사용하여 트리에 컴포넌트를 더하지 않고 위의 문제를 해결하는 방법을 보도록 하겠습니다.

사용자 정의 Hook 추출하기

두 개의 자바스크립트 함수에서 같은 로직을 공유하고자 할 때는 또 다른 함수로 분리합니다. 컴포넌트와 Hook 또한 함수이기 때문에 같은 방법을 사용할 수 있습니다!

사용자 정의 **Hook**은 이름이 `use`로 시작하는 자바스크립트 함수입니다. 사용자 **Hook**은 다른 **Hook**을 호출할 수 있습니다. 예를 들자면, 아래의 `useFriendStatus` 가 우리의 첫 번째 사용자 정의 Hook입니다.

```

import { useState, useEffect } from 'react';

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

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

    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}

```

딱히 새로운 것은 없습니다. 로직은 위의 컴포넌트로부터 복사해왔습니다. 다만 컴포넌트에서처럼 다른 Hook들은 사용자 Hook의 위로 놓여야 하며 사용자 정의 Hook은 조건부 함수가 아니어야 합니다.

React 컴포넌트와는 다르게 사용자 정의 Hook은 특정한 시그니처가 필요 없습니다. 무엇을 인수로 받아야 하며 필요하다면 무엇을 반환해야 하는지를 사용자가 결정할 수 있습니다. 다시 말하지만, 보통의 함수와 마찬가지입니다. 이름은 반드시 `use`로 시작해야 하는데 그 래야만 한눈에 보아도 **Hook 규칙**이 적용되는지를 파악할 수 있기 때문입니다.

`useFriendStatus` Hook의 목표는 친구의 상태를 구독하기 위함입니다. 이를 위하여 `friendID` 를 인수로 받고 온라인 상태의 여부를 반환합니다.

```

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

  // ...

  return isOnline;
}

```

이제 사용자 정의 Hook을 어떻게 이용하는지 보겠습니다.

사용자 정의 Hook 이용하기

처음 우리의 목표는 `FriendStatus` 와 `FriendListItem` 컴포넌트에 종복되어있는 로직을 제거하는 것이었습니다. 두 컴포넌트 모두 친구의 온라인 상태 여부를 알아야 하죠.

이제 이 로직을 `useFriendStatus` hook으로 뽑아내었으니, 바로 사용할 수 있습니다.

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}

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

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

본래의 예시와 동일한 코드인가요? 네 정확히 같은 방식으로 작동합니다. 자세히 보면 작동방식에 어떤 변화도 없다는 것을 알 수 있습니다. 바뀐 것은 오로지 공통의 코드를 뽑아내 새로운 함수로 만든 것뿐입니다. **사용자 정의 Hook은 React의 특별한 기능이라기보다 기본적으로 Hook의 디자인을 따르는 관습입니다.**

사용자 정의 Hook의 이름은 “`use`”로 시작되어야 하나요? 네 그렇습니다. 이 관습은 아주 중요합니다. 이를 따르지 않으면 특정한 함수가 그 안에서 Hook을 호출하는지를 알 수 없기 때문에 **Hook 규칙**의 위반 여부를 자동으로 체크할 수 없습니다.

같은 Hook을 사용하는 두 개의 컴포넌트는 state를 공유하나요? 아니요. 사용자 정의 Hook은 상태 관련 로직(구독을 설정하고 현재 변수값을 기억하는 것)을 재사용하는 메커니즘이지만 사용자 Hook을 사용할 때마다 그 안의 state와 effect는 완전히 독립적입니다.

사용자 정의 Hook은 어떻게 독립된 state를 얻는 건가요? 각각의 Hook에 대한 호출은 서로 독립된 state를 받습니다. `useFriendStatus` 를 직접적으로 호출하기 때문에 React의 관점에서 이 컴포넌트는 `useState` 와 `useEffect` 를 호출한 것과 다름없습니다. 또한 우리가 **이전에 배웠듯이**, 하나의 컴포넌트 안에서 `useState` 와 `useEffect` 를 여러 번 부를 수 있고 이들은 모두 완전히 독립적입니다.

팁: Hook에서 Hook으로 정보 전달하기

Hook은 함수이기 때문에 Hook 사이에서도 정보를 전달할 수 있습니다.

상황설명을 위해 채팅 예시에 있는 다른 컴포넌트를 사용하겠습니다. 현재 선택된 친구가 온라인 상태인지를 표시하는 채팅 수신자 선택기입니다.

```
const friendList = [
];
```

```

function ChatRecipientPicker() {
  const [recipientID, setRecipientID] = useState(1);
  const isRecipientOnline = useFriendStatus(recipientID);

  return (
    <>
      <Circle color={isRecipientOnline ? 'green' : 'red'} />
      <select
        value={recipientID}
        onChange={e => setRecipientID(Number(e.target.value))}>
        >
        {friendList.map(friend => (
          <option key={friend.id} value={friend.id}>
            {friend.name}
          </option>
        ))}
      </select>
    </>
  );
}

```

현재 선택된 친구의 ID를 `recipientID` state 변수에 저장하고 사용자가 `<select>` 선택기에 있는 다른 친구를 선택하면 이를 업데이트합니다.

`useState` Hook 호출은 `recipientID` state 변수의 최신값을 돌려주기 때문에 이를 `useFriendStatus` Hook에 인수로 보낼 수 있습니다.

```

const [recipientID, setRecipientID] = useState(1);
const isRecipientOnline = useFriendStatus(recipientID);

```

이를 통해 지금 선택되어있는 친구의 온라인 상태 여부를 알 수 있습니다. 다른 친구를 선택하고 `recipientID` state 변수를 업데이트하면 `useFriendStatus` Hook은 이미 선택되어있는 친구의 구독을 해지하고 새로이 선택된 친구의 상태를 구독할 것입니다.

useYourImagination() {#useyourimagination}

사용자 정의 Hook은 이전 React 컴포넌트에서는 불가능했던 로직공유의 유연성을 제공합니다. 사용자 정의 Hook을 만들어 품 다루기, 애니메이션, 선언형 구독, 타이머, 그 외에 생각하지 않은 부분까지 훨씬 다양한 쓰임새에 적용할 수 있습니다. 또한 React의 내장된 기능 만큼이나 사용하기 쉬운 Hook을 만들 수도 있습니다.

너무 이론 단계에서 로직을 뽑아내려고 하지는 않는 게 좋습니다. 함수 컴포넌트가 할 수 있는 일이 더 다양해졌기 때문에 여러분의 코드에 있는 함수 컴포넌트의 길이도 길어졌을 것입니다. 이는 지극히 평범한 일이며 지금 바로 Hook으로 분리해야만 한다고 느낄 필요는 없습니다. 하지만 동시에 사용자 정의 Hook이 복잡한 로직을 단순한 인터페이스 속에 숨길 수 있도록 하거나 복잡하게 뒤엉킨 컴포넌트를 풀어내도록 돋는 경우들을 찾아내는 것을 권장합니다.

예를 들자면, 내부에 많은 state를 지니고 있지만, 이것들이 적절하게 관리되지 않는 컴포넌트가 있다고 가정하겠습니다. `useState`는 업데이트 로직을 모아주는 데에는 도움이 되지 않기 때문에 대신 [Redux](#) reducer의 이용을 선호할 수도 있겠지요.

```

function todosReducer(state, action) {
  switch (action.type) {
    case 'add':
      return [...state, {
        text: action.text,
        completed: false
      }];
  }
}

```

```
// ... other actions ...
default:
  return state;
}
```

reducer는 독립적으로 테스트하기에 편리하며 복잡한 업데이트 로직의 표현이 늘어나는 경우에도 잘 맞습니다. 필요하다면 더 작은 reducer로 나누는 것도 가능합니다. 하지만 React state의 장점을 누리는 것은 선택일 뿐, 이 때문에 또 다른 라이브러리를 설치하고 싶지 않을 수도 있습니다.

컴포넌트의 `안의 state`를 reducer로 관리하는 `useReducer` Hook을 작성한다면 어떨까요? 이 Hook을 간단히 표현하면 다음과 같습니다.

```
function useReducer(reducer, initialState) {
  const [state, setState] = useState(initialState);

  function dispatch(action) {
    const nextState = reducer(state, action);
    setState(nextState);
  }

  return [state, dispatch];
}
```

컴포넌트 안에서 이 Hook을 사용하여 reducer가 state 관리를 합니다.

```
function Todos() {
  const [todos, dispatch] = useReducer(todosReducer, []);

  function handleAddClick(text) {
    dispatch({ type: 'add', text });
  }

  // ...
}
```

복잡한 컴포넌트에서 내에서 state를 reducer로 관리해야 하는 보편적 필요성을 고려하여 React에는 `useReducer` 가 내장되어 있습니다. 이는 [Hook API 참고서](#)에서 다른 내장 Hook과 함께 찾아볼 수 있습니다.

Using the Effect Hook

*Hooks*는 React 16.8버전에 새로 추가되었습니다. Hook은 클래스 컴포넌트를 작성하지 않아도 state와 같은 특징들을 사용할 수 있습니다.

*Effect Hook*을 사용하면 함수 컴포넌트에서 side effect를 수행할 수 있습니다.

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

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

  // componentDidMount, componentDidUpdate와 같은 방식으로
  useEffect(() => {
    // 브라우저 API를 이용하여 문서 타이틀을 업데이트합니다.
    document.title = `You clicked ${count} times`;
  });

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

위의 코드는 [이전 페이지의 카운터 예시](#)를 바탕으로 하지만, 문서의 타이틀을 클릭 횟수가 포함된 문장으로 표현할 수 있도록 새로운 기능을 더했습니다.

데이터 가져오기, 구독(subscription) 설정하기, 수동으로 React 컴포넌트의 DOM을 수정하는 것까지 이 모든 것이 *side effects*입니다. 이런 기능들(operations)을 *side effect*(혹은 *effect*)라 부르는 것이 익숙하지 않을 수도 있지만, 아마도 이전에 만들었던 컴포넌트에서 위의 기능들을 구현해보았을 것입니다.

팁

React의 class 생명주기 메서드에 친숙하다면, `useEffect` Hook을 `componentDidMount` 와 `componentDidUpdate`, `componentWillUnmount` 가 합쳐진 것으로 생각해도 좋습니다.

React 컴포넌트에는 일반적으로 두 종류의 *side effects*가 있습니다. 정리(clean-up)가 필요한 것과 그렇지 않은 것. 이 둘을 어떻게 구분해야 할지 자세하게 알아봅시다.

정리(Clean-up)를 이용하지 않는 Effects

React가 DOM을 업데이트한 뒤 추가로 코드를 실행해야 하는 경우가 있습니다. 네트워크 리퀘스트, DOM 수동 조작, 로깅 등은 정리(clean-up)가 필요 없는 경우들입니다. 이러한 예들은 실행 이후 신경 쓸 것이 없기 때문입니다. class와 hook이 이러한 *side effects*를 어떻게 다르게 구현하는지 비교해봅시다.

Class를 사용하는 예시

React의 class 컴포넌트에서 `render` 메서드 그 자체는 side effect를 발생시키지 않습니다. 이때는 아직 이를 시기로서 이러한 effect를 수행하는 것은 React가 DOM을 업데이트하고 난 이후입니다.

React class에서 side effect를 `componentDidMount` 와 `componentDidUpdate`에 두는 것이 바로 이 때문입니다. 예시로 돌아와서 React가 DOM을 바꾸고 난 뒤 문서 타이틀을 업데이트하는 React counter 클래스 컴포넌트를 봅시다.

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

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }

  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

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

위 코드에서 `class` 안의 두 개의 생명주기 메서드에 같은 코드가 중복되는 것에 주의합시다

이는 컴포넌트가 이제 막 마운트된 단계인지 아니면 업데이트되는 것인지에 상관없이 같은 side effect를 수행해야 하기 때문입니다. 개념적으로 렌더링 이후에는 항상 같은 코드가 수행되기를 바라는 것이죠. 하지만 React 클래스 컴포넌트는 그러한 메서드를 가지고 있지 않습니다. 함수를 별개의 메서드로 뽑아낸다고 해도 여전히 두 장소에서 함수를 불러내야 합니다.

이제 `useEffect` Hook에서 같은 기능을 어떻게 구현하는지 보겠습니다.

Hook을 이용하는 예시

아래의 코드는 위에서 이미 보았던 것이지만 이번에는 좀 더 자세히 살펴보겠습니다.

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

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
}
```

```

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

```

useEffect 가 하는 일은 무엇일까요? **useEffect** Hook을 이용하여 우리는 React에게 컴포넌트가 렌더링 이후에 어떤 일을 수행해야하는지를 말합니다. React는 우리가 넘긴 함수를 기억했다가(이 함수를 'effect'라고 부릅니다) DOM 업데이트를 수행한 이후에 불러낼 것입니다. 위의 경우에는 effect를 통해 문서 타이틀을 지정하지만, 이 외에도 데이터를 가져오거나 다른 명령형(imperative) API를 불러내는 일도 할 수 있습니다.

useEffect 를 컴포넌트 안에서 불러내는 이유는 무엇일까요? **useEffect** 를 컴포넌트 내부에 둘으로써 effect를 통해 `count` state 변수(또는 그 어떤 prop에도)에 접근할 수 있게 됩니다. 함수 범위 안에 존재하기 때문에 특별한 API 없이도 값을 얻을 수 있는 것입니다. Hook은 자바스크립트의 클로저를 이용하여 React에 한정된 API를 고안하는 것보다 자바스크립트가 이미 가지고 있는 방법을 이용하여 문제를 해결합니다.

useEffect 는 렌더링 이후에 매번 수행되는 걸까요? 네, 기본적으로 첫번째 렌더링과 이후의 모든 업데이트에서 수행됩니다.(나중에 **effect**를 필요에 맞게 수정하는 방법에 대해 다를 것입니다.) 마운팅과 업데이트라는 방식으로 생각하는 대신 effect를 렌더링 이후에 발생하는 것으로 생각하는 것이 더 쉬울 것입니다. React는 effect가 수행되는 시점에 이미 DOM이 업데이트되었음을 보장합니다.

상세한 설명

effect에 대해 좀 더 알아보았으니 아래의 코드들을 더 쉽게 이해할 수 있을 것입니다.

```

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
}

```

`count` state 변수를 선언한 뒤 React에게 effect를 사용함을 말하고 있습니다. **useEffect** Hook에 함수를 전달하고 있는데 이 함수가 바로 effect입니다. 이 effect 내부에서 `document.title`이라는 브라우저 API를 이용하여 문서 타이틀을 지정합니다. 같은 함수 내부에 있기 때문에 최신의 `count` 를 바로 얻을 수 있습니다. 컴포넌트를 렌더링할 때 React는 우리가 이용한 effect를 기억하였다가 DOM을 업데이트한 이후에 실행합니다. 이는 맨 첫 번째 렌더링은 물론 그 이후의 모든 렌더링에 똑같이 적용됩니다.

숙련된 자바스크립트 개발자라면 **useEffect** 에 전달된 함수가 모든 렌더링에서 다르다는 것을 알아챘을지도 모릅니다. 이는 의도된 것으로서, `count` 값이 제대로 업데이트 되는지에 대한 걱정 없이 effect 내부에서 그 값을 읽을 수 있게 하는 부분이기도 합니다. 리렌더링하는 때마다 모두 이전과 다른 effect로 교체하여 전달합니다. 이 점이 렌더링의 결과의 한 부분이 되게 만드는 점인데, 각각의 effect는 특정한 렌더링에 속합니다. [이 페이지의 뒷부분](#)에서 이것이 왜 유용한지에 대해서 더 자세히 다를 것입니다.

팁

`componentDidMount` 혹은 `componentDidUpdate` 와는 달리 `useEffect`에서 사용되는 effect는 브라우저가 화면을 업데이트하는 것을 차단하지 않습니다. 이를 통해 애플리케이션의 반응성을 향상해줍니다. 대부분의 effect는 동기적으로 실행될 필요가 없습니다. 흔하지는 않지만 (레이아웃의 측정과 같은) 동기적 실행이 필요한 경우에는 `useEffect` 와 동일한 API를 사용하는 `useLayoutEffect` 라는 별도의 Hook이 존재합니다.

정리(clean-up)를 이용하는 Effects

위에서 정리(clean-up)가 필요하지 않은 side effect를 보았지만, 정리(clean-up)가 필요한 effect도 있습니다. 외부 데이터에 구독(subscription)을 설정해야 하는 경우를 생각해보겠습니다. 이런 경우에 메모리 누수가 발생하지 않도록 정리(clean-up)하는 것은 매우 중요합니다. class와 Hook을 사용하는 두 경우를 비교해보겠습니다.

Class를 사용하는 예시

React class에서는 흔히 `componentDidMount`에 구독(subscription)을 설정한 뒤 `componentWillUnmount`에서 이를 정리(clean-up)합니다. 친구의 온라인 상태를 구독할 수 있는 ChatAPI 모듈의 예를 들어보겠습니다. 다음은 class를 이용하여 상태를 구독(subscribe)하고 보여주는 코드입니다.

```
class FriendStatus extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }

  render() {
    if (this.state.isOnline === null) {
      return 'Loading...';
    }
    return this.state.isOnline ? 'Online' : 'Offline';
  }
}
```

`componentDidMount` 와 `componentWillUnmount` 가 어떻게 대칭을 이루고 있는지를 봅시다. 두 개의 메서드 내에 개념상 똑같은 effect에 대한 코드가 있음에도 불구하고 생명주기 메서드는 이를 분리하게 만듭니다.

주의

눈썰미가 좋은 독자들은 이 예시가 완전하기 위해서는 `componentDidUpdate` 가 필요하다는 것을 눈치챘을 것입니다. 이에 대해서는 [다음 섹션](#)에서 다를 것입니다.

Hook을 이용하는 예시

이제 이 컴포넌트를 Hook을 이용하여 구현해봅시다.

정리(clean-up)의 실행을 위해 별개의 effect가 필요하다고 생각할 수도 있습니다. 하지만 구독(subscription)의 추가와 제거를 위한 코드는 결합도가 높기 때문에 `useEffect`는 이를 함께 다루도록 고안되었습니다. effect가 함수를 반환하면 React는 그 함수를 정리가 필요한 때에 실행시킬 것입니다.

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

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

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

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // effect 이후에 어떻게 정리(clean-up)할 것인지 표시합니다.
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

effect에서 함수를 반환하는 이유는 무엇일까요? 이는 effect를 위한 추가적인 정리(clean-up) 메커니즘입니다. 모든 effect는 정리를 위한 함수를 반환할 수 있습니다. 이 점이 구독(subscription)의 추가와 제거를 위한 로직을 가까이 묶어둘 수 있게 합니다. 구독(subscription)의 추가와 제거가 모두 하나의 effect를 구성하는 것입니다.

React가 effect를 정리(clean-up)하는 시점은 정확히 언제일까요? React는 컴포넌트가 마운트 해제되는 때에 정리(clean-up)를 실행합니다. 하지만 위의 예시에서 보았듯이 effect는 한번이 아니라 렌더링이 실행되는 때마다 실행됩니다. React가 다음 차례의 effect를 실행하기 전에 이전의 렌더링에서 파생된 effect 또한 정리하는 이유가 바로 이 때문입니다. 이것이 [버그를 방지하는 데에 어떻게 도움이 되는지](#) 그리고 [성능 저하 문제가 발생할 경우 effect를 건너뛰는 방법](#)에 대해서 이다음으로 논의해봅시다.

주의

effect에서 반드시 유명함수(named function)를 반환해야 하는 것은 아닙니다. 목적을 분명히 하기 위해 정리(clean-up)라고 부르고 있지만 화살표 함수를 반환하거나 다른 이름으로 불러도 무방합니다.

요약

`useEffect` 가 컴포넌트의 렌더링 이후에 다양한 side effects를 표현할 수 있음을 위에서 배웠습니다. effect에 정리(clean-up)가 필요한 경우에는 함수를 반환합니다.

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

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
});
```

정리(clean-up)가 필요없는 경우에는 어떤 것도 반환하지 않습니다.

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
});
```

이처럼 effect Hook은 두 가지 경우를 한 개의 API로 통합합니다.

effect hook이 어떻게 작동하는지에 대해 충분히 이해했거나, 내용이 이해하기 어렵다고 생각된다면 [다음 페이지의 Hook의 규칙](#)로 넘어가도 좋습니다.

effect를 이용하는 팁

이제 숙련된 React 사용자들이라면 보다 궁금해할 `useEffect`에 대해 좀 더 깊이 알아보겠습니다. 이 부분은 지금 바로 읽어야 하는 것은 아니며, 언제라도 effect hook의 자세한 이해가 필요할 때 돌아와서 읽어도 좋습니다.

팁: 관심사를 구분하려고 한다면 **Multiple Effect**를 사용합니다

Hook이 탄생한 동기가 된 문제 중의 하나가 생명주기 class 메서드가 관련이 없는 로직들은 모아놓고, 관련이 있는 로직들은 여러 개의 메서드에 나누어 놓는 경우가 자주 있다는 것입니다. 이전의 예시에서도 보았던 카운터와 친구의 상태 지표 로직을 결합한 컴포넌트를 보겠습니다.

```
class FriendStatusWithCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0, isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
}
```

```

componentDidUpdate() {
  document.title = `You clicked ${this.state.count} times`;
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

handleStatusChange(status) {
  this.setState({
    isOnline: status.isOnline
  });
}
// ...

```

`document.title` 을 설정하는 로직이 `componentDidMount` 와 `componentDidUpdate`에 나누어져 있습니다. 구독(subscription)로직 또한 `componentDidMount` 와 `componentWillUnmount`에 나누어져 있네요. `componentDidMount` 가 두 가지의 작업을 위한 코드를 모두 가지고 있습니다.

Hook을 이용하여 이 문제를 어떻게 해결할 수 있을까요? [State Hook을 여러 번 사용할 수 있는 것처럼 effect](#) 또한 여러 번 사용할 수 있습니다. Effect를 이용하여 서로 관련이 없는 로직들을 갈라놓을 수 있습니다.

```

function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
}
// ...

```

Hook을 이용하면 생명주기 메서드에 따라서가 아니라 코드가 무엇을 하는지에 따라 나눌 수가 있습니다. React는 컴포넌트에 사용된 모든 effect를 지정된 순서에 맞춰 적용합니다.

설명: effect가 업데이트 시마다 실행되는 이유

class에 익숙하다면 왜 effect 정리(clean-up)가 마운트 해제되는 때에 한번만이 아니라 모든 리렌더링 시에 실행되는지가 궁금할 것입니다. 이러한 디자인이 버그가 적은 컴포넌트를 만드는 데에 어떻게 도움이 되는지 다음의 예시를 통해 알아봅시다.

이 페이지의 위에서 봤던 친구가 온라인인지 아닌지 표시하는 `FriendStatus` 컴포넌트 예시를 생각해봅시다. class는 `this.props`로부터 `friend.id`를 읽어내고 컴포넌트가 마운트된 이후에 친구의 상태를 구독하며 컴포넌트가 마운트를 해제할 때에 구독을 해지합니다.

```

componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

```

그런데 컴포넌트가 화면에 표시되어있는 동안 `friend prop`이 변한다면 무슨 일이 일어날까요? 컴포넌트는 다른 친구의 온라인 상태를 계속 표시할 것입니다. 버그인 거죠. 또한 마운트 해제가 일어날 동안에는 구독 해지 호출이 다른 친구 ID를 사용하여 메모리 누수나 충돌이 발생할 수도 있습니다.

클래스 컴포넌트에서는 이런 경우들을 다루기 위해 `componentDidUpdate`를 사용합니다.

```

componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentDidUpdate(prevProps) {
  // 이전 friend.id에서 구독을 해지합니다.
  ChatAPI.unsubscribeFromFriendStatus(
    prevProps.friend.id,
    this.handleStatusChange
  );
  // 다음 friend.id를 구독합니다.
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

```

React 애플리케이션의 흔한 버그 중의 하나가 `componentDidUpdate`를 제대로 다루지 않는 것입니다.

이번에는 Hook을 사용하는 컴포넌트를 생각해봅시다.

```

function FriendStatus(props) {
  // ...
  useEffect(() => {
    // ...
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {

```

```
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
}
```

이 경우에는 버그에 시달리지 않습니다.(달리 바꾼 것도 없는데 말이죠.)

`useEffect` 가 기본적으로 업데이트를 다루기 때문에 더는 업데이트를 위한 특별한 코드가 필요 없습니다. 다음의 `effect`를 적용하기 전에 이전의 `effect`는 정리(clean-up)합니다. 구독과 구독 해지 호출을 반복해서 만들어내는 컴포넌트를 통해 이를 가시화해봅시다.

```
ChatAPI.subscribeToFriendStatus(100, handleStatusChange); // 첫번째 effect가 작동합니다.

ChatAPI.unsubscribeFromFriendStatus(100, handleStatusChange); // 이전의 effect를 정리(clean-up)합니다.
ChatAPI.subscribeToFriendStatus(200, handleStatusChange); // 다음 effect가 작동합니다.

ChatAPI.unsubscribeFromFriendStatus(200, handleStatusChange); // 이전의 effect를 정리(clean-up)합니다.
ChatAPI.subscribeToFriendStatus(300, handleStatusChange); // 다음 effect가 작동합니다.

// 마운트를 해제합니다.
ChatAPI.unsubscribeFromFriendStatus(300, handleStatusChange); // 마지막 effect를 정리(clean-up)합니다.
```

이러한 방식으로 동작하는 것이 일관성을 유지해주며 클래스 컴포넌트에서는 흔히 업데이트 로직을 빼먹으면서 발생할 수 있는 버그를 예방합니다.

팁: Effect를 건너뛰어 성능 최적화하기

모든 렌더링 이후에 `effect`를 정리(clean-up)하거나 적용하는 것이 때때로 성능 저하를 발생시키는 경우도 있습니다. 클래스 컴포넌트의 경우에는 `componentDidUpdate`에서 `prevProps`나 `prevState`와의 비교를 통해 이러한 문제를 해결할 수 있습니다.

```
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    document.title = `You clicked ${this.state.count} times`;
  }
}
```

이러한 요구 조건은 흔하기 때문에 `useEffect` Hook API에 이미 내재하여 있습니다. 특정 값들이 리렌더링 시에 변경되지 않는다면 React로 하여금 `effect`를 건너뛰도록 할 수 있습니다. `useEffect`의 선택적 인수인 두 번째 인수로 배열을 넘기면 됩니다.

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // count가 바뀔 때만 effect를 재실행합니다.
```

위의 예시에서 우리는 `[count]`를 두 번째 인수로 넘깁니다. 이것이 의미하는 바는 다음과 같습니다. `count` 가 `5`이고 컴포넌트가 리렌더링된 이후에도 여전히 `count` 는 변함없이 `5`라면 React는 이전 렌더링 시의 값 `[5]`를 그다음 렌더링 때의 `[5]`와 비교합니다. 배열 내의 모든 값이 같기 때문에(`5 === 5`) React는 `effect`를 건너뛰게 됩니다. 이런 식으로 최적화가 가능합니다.

`count` 가 `6` 으로 업데이트된 뒤에 렌더링하면 React는 이전에 렌더링된 값 `[5]` 를 그다음 렌더링 시의 `[6]` 와 비교합니다. 이때 `5 !== 6` 이기 때문에 React는 `effect`를 재실행합니다. 배열 내에 여러 개의 값이 있다면 그중의 단 하나만 다를지라도 React는 `effect`를 재실행합니다.

이것은 정리(clean-up)를 사용하는 effect의 경우에도 동일하게 작용합니다.

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

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
}, [props.friend.id]); // props.friend.id가 바뀔 때만 재구독합니다.
```

두 번째 인자는 빌드 시 변환에 의해 자동으로 추가될 수도 있습니다.

주의

이 최적화 방법을 사용한다면 배열이 컴포넌트 범위 내에서 바뀌는 값들과 **effect**에 의해 사용되는 값을 모두 포함하는 것을 기억해주세요. 그렇지 않으면 현재 값이 아닌 이전의 렌더링 때의 값을 참고하게 됩니다. 이에 대해서는 [함수를 다루는 방법](#)과 [의존성 배열이 자주 바뀔 때는 어떻게 해야 하는가](#)에서 더 자세히 알아볼 수 있습니다.

effect를 실행하고 이를 정리(clean-up)하는 과정을 (마운트와 마운트 해제 시에) 딱 한 번씩만 실행하고 싶다면, 빈 배열([])을 두 번째 인수로 넘기면 됩니다. 이렇게 함으로써 React로 하여금 여러분의 effect가 prop이나 state의 그 어떤 값에도 의존하지 않으며 따라서 재실행되어야 할 필요가 없음을 알게 하는 것입니다. 이는 의존성 배열의 작동 방법을 그대로 따라서 사용하는 것일 뿐이며 특별한 방법인 것은 아닙니다.

빈 배열([])을 넘기게 되면, **effect** 안의 prop과 state는 초기값을 유지하게 됩니다. 빈 배열([])을 두 번째 인수로 넘기는 것이 기존에 사용하던 `componentDidMount` 와 `componentWillUnmount` 모델에 더 가깝지만, **effect**의 잦은 재실행을 피할 수 있는 [더 나은 해결방법](#)이 있습니다. 또한 React는 브라우저가 다 그려질 때까지 `useEffect`의 실행을 지연하기 때문에 추가적인 작업을 더하는 것이 큰 문제가 되지는 않습니다.

`exhaustive-deps` 규칙을 `eslint-plugin-react-hooks` 패키지에 포함하는 것을 추천합니다. 이 패키지는 의존성이 바르지 않게 지정되었을 때 경고하고 수정하도록 알려줍니다.

다음 단계

여기까지 오느라 수고가 많으셨습니다. 긴 글이지만 끝까지 읽으면서 많은 궁금증이 해결되었기를 바랍니다. 지금까지 배운 State Hook과 Effect Hook을 결합하면 많은 것을 할 수 있습니다. `class`를 이용하는 대부분의 경우를 구현할 수 있습니다. 구현되지 않는 경우에는 [additional Hooks](#)이 도움이 될 것입니다.

우리는 [동기](#)에 서술되어있는 문제들을 Hook이 어떻게 해결할 수 있는지 알아가고 있습니다. 어떻게 effect의 정리(clean-up)가 `componentDidUpdate` 와 `componentWillUnmount`에서의 중복을 피하고, 관련 있는 코드들을 한곳에 모아게 하며 버그를 줄일 수 있게 도와주는지 알아보았습니다. 또한 effect를 그 목적에 따라 어떻게 구분하는지도 보았습니다. 이는 클래스 컴포넌트에서는 할 수 없는 일이었죠.

지금 이 시점에서 여러분은 Hook이 어떻게 작동하는 것인지 궁금해하고 있을 것입니다. 어떻게 React는 리렌더링 시에 특정한 `useState`의 실행이 어떤 state 변수에 상응하는지 아는 것일까요? 어떻게 React는 이전과 그다음 effect를 업데이트 때마다 "매치업하는(일치하는)" 것일까요? 이에 대해서는 다음 장에서 **Hook**의 작동에 핵심적인 [Hook의 규칙](#)을 통해서 배워볼 것입니다.

react

[Go to TOC](#)

Hook 자주 묻는 질문

Hook은 React 16.8에 새로 추가되었습니다. Class를 작성하지 않고 state 및 기타 React 기능을 사용할 수 있습니다.

이 페이지는 [Hook](#) 자주 묻는 질문에 대한 답변입니다.

- [적용 전략](#)

- 어떤 버전의 React가 Hook을 포함합니까?
- 모든 class 컴포넌트를 다시 작성해야 합니까?
- Class로 하지 못하는 것 중에 Hook으로 가능한 것이 무엇인가요?
- React 지식은 얼마나 관련이 있습니까?
- Hook이나 class 또는 두 가지를 모두 사용해야 합니까?
- Hook이 class의 모든 사용 사례를 커버합니까?
- Hook이 render props 및 고차 컴포넌트를 대체합니까?
- Redux connect()와 React Router와 같은 인기 있는 API에 대해 Hook은 무엇을 의미합니까?
- Hook은 정적 타이핑과 함께 작동합니까?
- Hook을 사용하는 컴포넌트 테스트하는 방법?
- Lint 규칙은 정확히 무엇을 시행합니까?

- [Class에서 Hook으로](#)

- 생명주기 메서드가 Hook에 어떻게 대응합니까?
- Hook을 사용하여 데이터 가져오기를 수행하려면 어떻게 해야 합니까?
- 인스턴스 변수와 같은 것이 있습니까?
- 하나 또는 여러 state 변수를 사용해야 합니까?
- 업데이트에만 effect를 실행할 수 있습니까?
- 이전 props 또는 state를 얻는 방법?
- 함수 컴포넌트 안에 오래된 props나 state가 보이는 이유는 무엇입니까?
- getDerivedStateFromProps를 어떻게 구현합니까?
- forceUpdate와 같은 것이 있습니까?
- 함수 컴포넌트에 ref를 만들 수 있습니까?
- DOM 노드를 측정하려면 어떻게 해야 합니까?
- const [thing, setThing] = useState()는 무엇을 의미합니까?

- [성능 최적화](#)

- 업데이트 시 effect를 건너뛸 수 있습니까?
- 종속성 목록에서 함수 컴포넌트를 생략하는 것이 안전합니까?
- effect 종속성이 너무 자주 변경되면 어떻게 해야 합니까?
- shouldComponentUpdate는 어떻게 구현합니까?
- 계산을 메모이제이션 하는 법?
- 고비용의 객체를 지연해서 생성하는 법?
- 렌더링에서 함수 컴포넌트를 만들기 때문에 Hook이 느려집니까?
- 콜백 전달을 피하는 법?
- useCallback에서 자주 변경되는 값을 읽는 방법?

- [Hook의 이면](#)

- React는 Hook 호출을 컴포넌트와 어떻게 연관시키는가?

- Hook에 대한 선행 기술은 무엇입니까?

적용 전략

어떤 버전의 React가 Hook을 포함합니까?

16.8.0부터 React에는 React Hook의 안정적인 구현이 포함됩니다.

- React DOM
- React Native
- React DOM Server
- React 테스트 렌더러
- React 얇은 렌더러

Hook을 사용하려면 모든 React 패키지가 16.8.0 이상이어야합니다. 업데이트하는 것을 (예: React DOM) 잊어버리면 Hook이 작동하지 않습니다.

React Native 0.59 이상은 Hook을 지원합니다.

모든 class 컴포넌트를 다시 작성해야 합니까?

아닙니다. React에서 class를 삭제할 계획은 없습니다. 우리는 제품을 출시할 때마다 재작성을 할 여유가 없습니다. 새 코드에서 Hook을 사용하는 것이 좋습니다.

Class로 하지 못하는 것 중에 Hook으로 가능한 것이 무엇인가요?

Hook은 컴포넌트 간에 기능을 재사용할 수 있는 강력하고 표현적인 새로운 방법을 제공합니다. "자신만의 Hook 만들기"는 가능한 것을 엿볼 수 있게 해줍니다. React 핵심 팀 구성원이 작성한 [이 기사](#)에서는 Hook이 제공할 새로운 기능에 대해 자세히 설명합니다.

React 지식은 얼마나 관련이 있습니까?

Hook은 state, 생명주기, context 및 ref와 같은 이미 알고 있는 React 기능을 사용하는 보다 직접적인 방법입니다. React가 어떻게 작동하는지 근본적으로 바꿀 수 없으며 컴포넌트, props 및 하향식 데이터 흐름에 대한 지식도 마찬가지로 중요합니다.

Hook에는 독자적인 학습 곡선이 있습니다. 이 문서에 누락된 것이 있으면 [문제를 제기](#)하면 도움을 제공해 드리겠습니다.

Hook이나 class 또는 두 가지를 모두 사용해야 합니까?

준비가 되면 작성하는 새 컴포넌트에서 Hook을 시도해 보는 것이 좋습니다. 팀의 모든 구성원이 사용하고 이 문서에 익숙한지 확인해주세요. 일부러 다시 작성하지 않는 이상 (예: 버그 수정) 기존 class를 Hook으로 고쳐 쓰는 것은 추천하지 않습니다.

Class 컴포넌트 내부에서 Hook을 사용할 수는 없지만, class와 함수 컴포넌트를 단일 트리에서 Hook과 섞어서 사용할 수 있습니다. 컴포넌트가 class인지 Hook을 사용하는 함수 컴포넌트인지 여부는 해당 컴포넌트의 구현 세부 사항입니다. 장기적으로 우리는 Hook이 사람들이 React 컴포넌트를 작성하는 주요 방법이 될 것으로 기대합니다.

Hook이 class의 모든 사용 사례를 커버합니까?

우리의 목표는 Hook이 class의 모든 사용 사례를 가능한 한 빨리 커버하게 하는 것입니다. 드문 `getSnapshotBeforeUpdate`, `getDerivedStateFromError` 및 `componentDidCatch` 생명주기에 해당하는 Hook은 아직 없지만, 곧 추가할 계획입니다.

Hook의 초기 단계이며 일부 타사 라이브러리는 현재 Hook과 호환되지 않을 수 있습니다.

Hook이 render props 및 고차 컴포넌트를 대체할까요?

종종 render props와 고차 컴포넌트는 하나의 자식만 렌더링합니다. 우리는 Hook이 이 사용 사례를 처리하는 더 간단한 방법이라고 생각합니다. 여전히 두 패턴 모두를 쓸 수 있습니다. (예를 들어, 가상 스크롤러 컴포넌트에는 renderItem props가 있거나 시각적 컨테이너 컴포넌트에는 자체 DOM 구조가 있을 수 있습니다) 그러나 대부분의 경우 Hook은 충분하며 코드 트리의 중첩을 줄이는 데 도움이 될 수 있습니다.

Redux connect()와 React Router와 같은 인기 있는 API에 대해 Hook은 무엇을 의미합니까?

여태껏 쓰던 API를 계속 사용할 수 있습니다; 앞으로도 계속 작동할 것 입니다.

v7.1.0부터 React Redux는 [Hook API를 지원하고](#) `useDispatch` 또는 `useSelector` 와 같은 Hook을 노출합니다.

v5.1 이후 React Router는 [Hook을 지원합니다](#).

다른 라이브러리도 나중에 Hook을 지원할 수 있습니다.

Hook은 정적 타이핑과 함께 작동합니까?

Hook은 정적 타이핑을 염두에 두고 설계되었습니다. 함수 컴포넌트이기 때문에 고차 컴포넌트와 같은 패턴보다 타입을 명시하기가 더 쉽습니다. 최신 Flow 및 TypeScript React 정의에는 React Hook 지원이 포함됩니다.

중요한 점은, 커스텀 Hook은 더 엄격하게 타이핑하려는 경우 React API를 제한할 수 있는 기능을 제공합니다. React는 기초 요소를 제공하지만, 기본 제공 방식과 다른 방식으로 조합 할 수 있습니다.

Hook을 사용하는 컴포넌트 테스트하는 방법?

React의 관점에서 Hook을 사용하는 컴포넌트는 일반적인 컴포넌트입니다. 테스트 솔루션이 React internals에 종속하지 않는 경우 Hook이 있는 컴포넌트 테스트는 일반적으로 컴포넌트를 테스트하는 방법과 다르지 않아야 합니다.

주의

[테스팅 방안](#)에는 복사하여 붙여넣을 수 있는 많은 예시가 포함되어 있습니다.

예를 들어 여기 이 계수기 컴포넌트가 있다고 가정해 보겠습니다.

```
function Example() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

```

    </div>
  );
}

```

React DOM을 사용하여 테스트하겠습니다. 브라우저에서 발생하는 상황과 동작이 일치하도록 코드 렌더링을 래핑하고 이를 `ReactTestUtils.act()` 호출로 업데이트합니다.

```

import React from 'react';
import ReactDOM from 'react-dom/client';
import { act } from 'react-dom/test-utils';
import Counter from './Counter';

let container;

beforeEach(() => {
  container = document.createElement('div');
  document.body.appendChild(container);
});

afterEach(() => {
  document.body.removeChild(container);
  container = null;
});

it('can render and update a counter', () => {
  // 첫 번째 렌더링 및 effect 테스트
  act(() => {
    ReactDOM.createRoot(container).render(<Counter />);
  });
  const button = container.querySelector('button');
  const label = container.querySelector('p');
  expect(label.textContent).toBe('You clicked 0 times');
  expect(document.title).toBe('You clicked 0 times');

  // 두 번째 렌더링 및 effect 테스트
  act(() => {
    button.dispatchEvent(new MouseEvent('click', {bubbles: true}));
  });
  expect(label.textContent).toBe('You clicked 1 times');
  expect(document.title).toBe('You clicked 1 times');
});

```

`act()` 호출은 그 안의 effect를 플러시합니다.

커스텀 Hook을 테스트해야 하는 경우 테스트에서 컴포넌트를 작성하고 Hook을 사용하여 이를 수행 할 수 있습니다. 그런 다음 작성한 컴포넌트를 테스트 할 수 있습니다.

상용구를 줄이려면 [React Testing Library](#)를 사용하는 것이 좋습니다. 이 라이브러리는 최종 사용자와 마찬가지로 컴포넌트를 사용하는 테스트 작성을 장려하도록 설계되었습니다.

자세한 내용은 [테스팅 방안](#)을 확인해주세요.

Lint 규칙은 정확히 무엇을 시행합니까?

버그를 피하고자 Hook 규칙을 시행하는 [ESLint 플러그인](#)을 제공합니다. "use"로 시작하는 모든 함수 컴포넌트와 Hook 바로 뒤에 대 문자가 있다고 가정합니다. 우리는 이 휴리스틱이 완벽하지 않고 오 텀지가 있을 수 있다는 점을 인식하지만, 생태계 전반의 협약이 없으면 흐를 제대로 작동시킬 수 있는 방법이 없습니다 -- 더 긴 이름은 사람들이 Hook을 채택하거나 협약을 따르지 못하게 합니다.

특히, 규칙은 이것들을 시행합니다.

- Hook에 대한 호출은 `PascalCase` 함수 컴포넌트 (컴포넌트로 가정) 또는 다른 `useSomething` 함수 컴포넌트 (커스텀 Hook으로 가정) 내에 있습니다.
- 모든 렌더링에서 Hook은 동일한 순서로 호출됩니다.

휴리스틱이 몇 가지 더 있으며, 추후 오 템지를 피해 버그를 찾기 위해 규칙을 미세 조정함에 따라 변경될 수 있습니다.

Class에서 Hook으로

생명주기 메서드가 Hook에 어떻게 대응합니까?

- `constructor` : 함수 컴포넌트는 `constructor`가 필요하지 않습니다. `useState` 호출에서 `state`를 초기화 할 수 있습니다. 초기 `state`를 계산하는 것이 비싸면 `useState`에 함수 컴포넌트를 전달할 수 있습니다.
- `getDerivedStateFromProps` : 대신 렌더링하는 동안 업데이트 예약.
- `shouldComponentUpdate` : 아래의 `React.memo`를 참조해주세요.
- `render` : 이것은 함수 컴포넌트 본체 자체입니다.
- `componentDidMount`, `componentDidUpdate`, `componentWillUnmount` : `useEffect` Hook은 이들의 모든 조합을 표현할 수 있습니다. (흔하거나 그렇지 않은 경우 포함).
- `getSnapshotBeforeUpdate`, `componentDidCatch` 그리고 `getDerivedStateFromError` : 이러한 메서드에 대한 Hook은 없지만, 곧 추가될 예정입니다.

Hook을 사용하여 데이터 가져오기를 수행하려면 어떻게 해야 합니까?

다음은 시작하기 위한 짧은 [데모](#)입니다. 자세한 내용은 Hook을 사용한 데이터 가져오기를 다룬 [이 기사](#)를 확인해주세요.

인스턴스 변수와 같은 것이 있습니까?

네! `useRef()` Hook은 DOM ref만을 위한 것이 아닙니다. "ref" 객체는 현재 프로퍼티가 변경할 수 있고 어떤 값이든 보유할 수 있는 일반 컨테이너입니다. 이는 class의 인스턴스 프로퍼티와 유사합니다.

`useEffect` 내부에서 쓸 수 있습니다.

```
function Timer() {
  const intervalRef = useRef();

  useEffect(() => {
    const id = setInterval(() => {
      // ...
    });
    intervalRef.current = id;
    return () => {
      clearInterval(intervalRef.current);
    };
  });
  // ...
}
```

인터벌을 설정하고 싶다면 ref가 필요하지 않지만 (`id`는 로컬 effect일 수 있습니다), 이벤트 처리에서 인터벌을 지우고 싶을 때 유용합니다.

```
// ...
function handleCancelClick() {
  clearInterval(intervalRef.current);
}
// ...
```

개념적으로, class의 인스턴스 변수와 ref를 비슷하게 생각할 수 있습니다. [지연 초기화](#)를 수행하지 않는 한, 렌더링 중에 ref 설정을 피해주세요. -- 이것은 놀라운 상황을 초래할 수 있습니다. 대신, 일반적으로 이벤트 처리와 effect에서 ref를 수정하는 것이 좋습니다.

하나 또는 여러 state 변수를 사용해야 합니까?

Class를 배운 후라면, `useState()` 를 한 번만 호출하고 모든 state를 단일 객체에 넣고 싶을 수 있습니다. 원하시면 그렇게 할 수 있습니다. 다음은 마우스 움직임을 따르는 컴포넌트의 예입니다. 포인터의 위치와 크기를 로컬 state에 유지합니다.

```
function Box() {
  const [state, setState] = useState({ left: 0, top: 0, width: 100, height: 100 });
}
// ...
```

이제 사용자가 마우스를 움직일 때 `left` 과 `top` 의 포지션을 변경하는 로직을 작성하고 싶다고 가정해 보겠습니다. 이러한 필드를 이전 state 개체에 수동으로 병합하는 방법에 유의해주세요.

```
// ...
useEffect(() => {
  function handleWindowMouseMove(e) {
    // "... state"를 spread 하여 너비와 높이가 "손실"되지 않습니다
    setState(state => ({ ...state, left: e.pageX, top: e.pageY }));
  }
  // 주의: 이 구현은 약간 단순화되었습니다
  window.addEventListener('mousemove', handleWindowMouseMove);
  return () => window.removeEventListener('mousemove', handleWindowMouseMove);
}, []);
// ...
```

이는 state 변수를 업데이트할 때 그 값을 대체하기 때문입니다. 이것은 업데이트된 필드를 객체에 병합하는 class의 `this.setState` 와 다릅니다.

자동 병합이 그리운 경우 개체 state 업데이트를 병합하는 커스텀 `useLegacyState` Hook을 작성할 수 있습니다. 그러나, 함께 변경되는 값에 따라 state를 여러 state 변수로 분할하는 것을 추천합니다.

예를 들어 컴포넌트 state를 `position` 및 `size` 객체로 분할하고 병합할 필요 없이 항상 `position` 을 대체 할 수 있습니다.

```
function Box() {
  const [position, setPosition] = useState({ left: 0, top: 0 });
  const [size, setSize] = useState({ width: 100, height: 100 });

  useEffect(() => {
    function handleWindowMouseMove(e) {
      setPosition({ left: e.pageX, top: e.pageY });
    }
  }, []);
}
```

독립된 state 변수를 분리하면 또 다른 이점이 있습니다. 예를 들어 나중에 관련 로직을 커스텀 Hook으로 쉽게 추출 할 수 있습니다.

```
function Box() {
  const position = useWindowPosition();
  const [size, setSize] = useState({ width: 100, height: 100 });
  // ...
}

function useWindowPosition() {
  const [position, setPosition] = useState({ left: 0, top: 0 });
  useEffect(() => {
    // ...
  }, []);
  return position;
}
```

코드를 변경하지 않고 `position` state 변수에 대한 `useState` 호출과 관련 effect를 커스텀 Hook으로 옮길 수 있었던 방법에 유의해주세요. 모든 state가 단일 객체에 있으면 추출하기가 더 어려울 것입니다.

모든 state를 단일 `useState` 호출에 넣고 필드마다 `useState` 호출을 두는 방법도 쓸 수 있습니다. 컴포넌트는 이러한 두 극단 사이의 균형을 찾고 관련 state를 몇 개의 독립 state 변수로 그룹화할 때 가장 읽기 쉬운 경향이 있습니다. State 로직이 복잡해지면 [리듀서로 관리](#), 또는 커스텀 Hook을 사용하는 것이 좋습니다.

업데이트에만 effect를 실행할 수 있습니까?

이것은 드문 사용 사례입니다. 필요한 경우 [변경 가능한 ref를 사용하여](#) 첫 번째 또는 후속 렌더링에 있는지에 해당하는 부울 값을 수동으로 저장한 다음, 해당 플래그를 확인할 수 있습니다. (이 작업을 자주 수행하는 경우 커스텀 Hook을 만들 수 있습니다.)

이전 props 또는 state를 얻는 방법?

현재는 수동으로 [ref랑 같이](#) 사용할 수 있습니다.

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

  const prevCountRef = useRef();
  useEffect(() => {
    prevCountRef.current = count;
  });
  const prevCount = prevCountRef.current;

  return <h1>Now: {count}, before: {prevCount}</h1>;
}
```

약간 복잡 할 수 있지만, 커스텀 툥으로 추출 할 수 있습니다.

```
function Counter() {
  const [count, setCount] = useState(0);
  const prevCount = usePrevious(count);
  return <h1>Now: {count}, before: {prevCount}</h1>;
}

function usePrevious(value) {
  const ref = useRef();
  useEffect(() => {
    ref.current = value;
  });
  return ref.current;
}
```

```

    });
    return ref.current;
}

```

이것이 props, state 또는 기타 계산된 값에 대해 어떻게 작동하는지 확인해주세요.

```

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

  const calculation = count + 100;
  const prevCalculation = usePrevious(calculation);
  // ...
}

```

상대적으로 일반적인 사용 사례이기 때문에 향후 React에서 'usePrevious' Hook을 제공 할 수 있습니다.

[파생 state에 권장되는 패턴](#)도 참조하세요.

함수 컴포넌트 안에 오래된 props나 state가 보이는 이유는 무엇입니까?

이벤트 처리 및 effect를 포함한 컴포넌트 내부의 모든 함수 컴포넌트는 생성된 렌더링에서 props와 state를 "확인"합니다. 예를 들어 이와 같은 코드를 고려해주세요.

```

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

  function handleAlertClick() {
    setTimeout(() => {
      alert('You clicked on: ' + count);
    }, 3000);
  }

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

```

먼저 "알림 표시"를 클릭한 다음 카운터를 늘리면 "알림 표시" 버튼을 클릭할 때 경고 문구로 `count` 변수가 표시됩니다. 이것은 props와 state가 변경되지 않는다고 가정하는 상황에서 코드로 인한 버그를 방지합니다.

일부 비동기 콜백에서 최근 state를 의도적으로 읽으려면 `ref`에 보관해서 변경하고 읽으면 됩니다.

마지막으로, 부실한 props 또는 state를 볼 수 있는 또 다른 이유는 "종속성 배열" 최적화를 사용하지만 모든 종속성을 올바르게 지정하지 않은 경우입니다. 예를 들어, effect가 `[]`를 두 번째 인수로 지정하지만, 내부에서 `someProp`을 읽는 경우 `someProp`의 초기 값을 계속 "보고"합니다. 해결책은 종속성 배열을 제거하거나 수정하는 것입니다. 다음은 [함수 컴포넌트를 처리하는 방법](#)과 종속성을 잘못 건너뛰지 않고 effect를 덜 실행하는 [일반적인 전략](#)입니다.

주의

`eslint-plugin-react-hooks` 패키지의 일부로 `exhaustive-deps` ESLint 규칙을 제공합니다. 종속성이 잘못 지정되면 경고하고 수정을 제안합니다.

getDerivedStateFromProps를 어떻게 구현합니까?

필요하지 않을 수 있지만 드물게 수행하는 경우 (예: `<Transition>` 컴포넌트 구현) 렌더링 중에 `state`를 바로 업데이트 할 수 있습니다. React는 첫 번째 렌더링을 종료한 후 즉시 업데이트된 `state`로 컴포넌트를 다시 실행하므로 비용이 많이 들지 않습니다.

여기서는 비교할 수 있도록 `row` props의 이전 값을 `state` 변수에 저장합니다.

```
function ScrollView({row}) {
  const [isScrollingDown, setIsScrollingDown] = useState(false);
  const [prevRow, setPrevRow] = useState(null);

  if (row !== prevRow) {
    // 마지막 렌더링 이후 행이 변경되었습니다. isScrollingDown을 업데이트합니다.
    setIsScrollingDown(prevRow !== null && row > prevRow);
    setPrevRow(row);
  }

  return `Scrolling down: ${isScrollingDown}`;
}
```

처음에는 이상하게 보일 수 있지만, 렌더링 중 업데이트는 정확히 'getDerivedStateFromProps'가 개념적으로 항상 그랬던 것과 같습니다.

forceUpdate와 같은 것이 있습니까?

`useState` 와 `useReducer` Hook은 다음 값이 이전 값과 같으면 업데이트에서 제외됩니다. State를 변경하고 `setState`를 호출해도 다시 렌더링 되지 않습니다.

일반적으로 React에서 로컬 `state`를 변경해서는 안 됩니다. 그러나 도피 수단으로 증가하는 카운터를 사용하여 `state`가 변경되지 않은 경우에도 강제로 다시 렌더링 할 수 있습니다.

```
const [ignored, forceUpdate] = useReducer(x => x + 1, 0);

function handleClick() {
  forceUpdate();
}
```

가능하면 이 패턴을 피해주세요.

함수 컴포넌트에 ref를 만들 수 있습니까?

자주 필요하지는 않지만 `useImperativeHandle` Hook을 사용하여 일부 명령형 메서드를 부모 컴포넌트에 노출 할 수 있습니다.

DOM 노드를 측정하려면 어떻게 해야 합니까?

DOM 노드의 위치나 크기를 측정하는 기본적인 방법의 하나는 `콜백 ref`를 사용하는 것입니다. React는 `ref`가 다른 노드에 연결될 때마다 해당 콜백을 호출합니다. 다음은 짧은 데모입니다.

```

function MeasureExample() {
  const [height, setHeight] = useState(0);

  const measuredRef = useCallback(node => {
    if (node !== null) {
      setHeight(node.getBoundingClientRect().height);
    }
  }, []);

  return (
    <>
      <h1 ref={measuredRef}>Hello, world</h1>
      <h2>The above header is {Math.round(height)}px tall</h2>
    </>
  );
}

```

이 예시에서는 객체 ref가 현재 ref 값의 변경 사항에 대해 알려주지 않기 때문에 `useRef`를 선택하지 않았습니다. 콜백 ref를 사용하면 [자식 컴포넌트가 나중에 측정된 노드를 표시하더라도](#) (예: 클릭에 대한 응답으로) 여전히 부모 컴포넌트에서 이에 대한 알림을 받고 측정을 업데이트 할 수 있습니다.

`[]` 를 'useCallback'에 종속성 배열로 전달합니다. 이렇게 하면 ref 콜백이 다시 렌더링 간에 변경되지 않음으로 React가 불필요하게 호출하지 않습니다.

이 예시에서 콜백 ref는 렌더링 된 `<h1>` 컴포넌트가 모든 리렌더 동안 존재하기 때문에 컴포넌트가 마운트 및 마운트 해제될 때만 호출됩니다. 컴포넌트의 크기가 조정될 때마다 알림을 받으려면 `ResizeObserver` 또는 여기에 빌드된 제삼자 Hook을 사용할 수 있습니다.

원한다면 재사용 가능한 Hook으로 [이 로직을 추출](#) 할 수 있습니다.

```

function MeasureExample() {
  const [rect, ref] = useClientRect();
  return (
    <>
      <h1 ref={ref}>Hello, world</h1>
      {rect !== null &&
        <h2>The above header is {Math.round(rect.height)}px tall</h2>
      }
    </>
  );
}

function useClientRect() {
  const [rect, setRect] = useState(null);
  const ref = useCallback(node => {
    if (node !== null) {
      setRect(node.getBoundingClientRect());
    }
  }, []);
  return [rect, ref];
}

```

const [thing, setThing] = useState()는 무엇을 의미합니까?

이 구문에 익숙하지 않은 경우 State Hook 문서의 [explanation](#)을 확인하세요.

성능 최적화

업데이트 시 effect를 건너뛸 수 있습니까?

예. [조건부 effect 실행](#)을 참조해주세요. 업데이트 처리를 잊어 버리면 종종 [버그가 발생](#)하므로 이것은 기본 세팅이 아닙니다.

종속성 목록에서 함수 컴포넌트를 생략하는 것이 안전합니까?

일반적으로는 아닙니다.

```
function Example({ someProp }) {
  function doSomething() {
    console.log(someProp);
  }

  useEffect(() => {
    doSomething();
  }, []); // 🔴 이것은 안전하지 않습니다 (`someProp`을 사용하는 `doSomething`을 호출합니다)
}
```

Effect 외부의 함수 컴포넌트에서 어떤 props 또는 state를 사용하는지 기억하기 어렵습니다. 이것이 [일반적으로 그 내부의 effect에 필요한 함수 컴포넌트를 선언하려는 이유입니다](#). 그러면 effect가 미치는 컴포넌트 범위의 값을 쉽게 확인할 수 있습니다.

```
function Example({ someProp }) {
  useEffect(() => {
    function doSomething() {
      console.log(someProp);
    }

    doSomething();
  }, [someProp]); // ✅ OK (우리 effect는 `someProp` 만 사용합니다)
}
```

그 후에도 컴포넌트 범위의 값을 사용하지 않으면 `[]`를 지정하는 것이 안전합니다.

```
useEffect(() => {
  function doSomething() {
    console.log('hello');
  }

  doSomething();
}, []); // ✅ 이 예에서는 컴포넌트 범위의 *어떤* 값도 사용하지 않기 때문에 좋습니다
```

사용 사례에 따라 아래에 설명된 몇 가지 옵션이 더 있습니다.

주의

`eslint-plugin-react-hooks` 패키지의 일부로 `exhaustive-deps` ESLint 규칙을 제공합니다. 업데이트를 일관되게 처리하지 않는 컴포넌트를 찾는 데 도움이 됩니다.

이것이 왜 중요한지 봅시다.

`useEffect`, `useLayoutEffect`, `useMemo`, `useCallback` 또는 `useImperativeHandle`의 마지막 인수로 종속성 목록을 지정하는 경우 콜백 내에서 사용되는 모든 값을 포함하고 React 데이터 흐름에 참여해야 합니다. 여기에는 props, state 및 그로부터 파생된 모든 것이 포함됩니다.

함수 컴포넌트 (또는 함수 컴포넌트가 호출하는 함수 컴포넌트)가 props, state 또는 파생된 값을 참조하지 않는 경우에만 종속성 목록에서 함수 컴포넌트를 생략하는 것이 안전합니다. 이 예에는 버그가 있습니다.

```
function ProductPage({ productId }) {
  const [product, setProduct] = useState(null);

  async function fetchProduct() {
    const response = await fetch('http://myapi/product/' + productId); // productId props 사용
    const json = await response.json();
    setProduct(json);
  }

  useEffect(() => {
    fetchProduct();
  }, []); // ❌ `fetchProduct` 가 `productId` 를 사용하므로 잘못되었습니다
  // ...
}
```

권장되는 해결 방법은 해당 기능을 **effect 내부로 이동하는 것입니다**. 이를 통해 effect가 사용하는 props 또는 state를 쉽게 확인하고 모두 선언되었는지 확인할 수 있습니다.

```
function ProductPage({ productId }) {
  const [product, setProduct] = useState(null);

  useEffect(() => {
    // 이 함수 컴포넌트를 effect 내부로 이동하면 사용하는 값을 명확하게 볼 수 있습니다.
    async function fetchProduct() {
      const response = await fetch('http://myapi/product/' + productId);
      const json = await response.json();
      setProduct(json);
    }

    fetchProduct();
  }, [productId]); // ✅ 효과는 productId 만 사용하므로 유효합니다
  // ...
}
```

이를 통해 effect 내부의 로컬 변수를 사용하여 비순차적 인 응답을 처리 할 수도 있습니다.

```
useEffect(() => {
  let ignore = false;
  async function fetchProduct() {
    const response = await fetch('http://myapi/product/' + productId);
    const json = await response.json();
    if (!ignore) setProduct(json);
  }

  fetchProduct();
  return () => { ignore = true };
}, [productId]);
```

Effect 내부로 함수를 옮겼으므로 의존성 배열에 있을 필요가 없습니다.

팁

이 [짧은 데모](#)와 [이 기사](#)를 확인해 Hook을 사용한 데이터 가져오기에 대해 자세히 알아보세요.

만일 어떤 이유로 함수를 **effect** 내부로 이동할 수 없는 경우 몇 가지 옵션이 더 있습니다.

- 해당 함수를 컴포넌트 외부로 이동해 볼 수 있습니다. 이 경우 함수는 props나 state를 참조하지 않도록 보장되며 종속성 목록에 있을 필요가 없습니다.
- 호출하는 함수가 순수한 계산이고 렌더링하는 동안 호출해도 안전하다면, 대신에 **effect** 외부에서 호출하고 반환된 값에 따라 effect 가 달라지도록 할 수 있습니다.
- 마지막 수단으로 **Effect** 의존성 배열에 함수를 추가하되, 정의를 `useCallback` Hook에 감싸주세요. 이렇게 하면 자체 종속성도 변경되지 않는 한 모든 렌더링에서 변경되지 않습니다.

```
function ProductPage({ productId }) {
  // ✅ 모든 렌더링에서 변경되지 않도록 useCallback으로 래핑
  const fetchProduct = useCallback(() => {
    // ... productId로 무언가를 합니다 ...
  }, [productId]); // ✅ 모든 useCallback 종속성이 지정됩니다

  return <ProductDetails fetchProduct={fetchProduct} />;
}

function ProductDetails({ fetchProduct }) {
  useEffect(() => {
    fetchProduct();
  }, [fetchProduct]); // ✅ 모든 useEffect 종속성이 지정됩니다
  // ...
}
```

위의 예에서 종속성 목록에 함수 컴포넌트를 유지할 필요가 있습니다. 이렇게 하면 `ProductPage`의 `productId` props가 변경되면 `ProductDetails` 컴포넌트에서 자동으로 다시 가져오기가 트리거 됩니다.

effect 종속성이 너무 자주 변경되면 어떻게 해야 합니까?

때로는 effect가 너무 자주 변경되는 state를 사용할 수도 있습니다. 종속성 목록에서 해당 state를 생략하고 싶을 수 있지만, 이 경우 일반적으로 버그가 발생합니다.

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

  useEffect(() => {
    const id = setInterval(() => {
      setCount(count + 1); // 이 effect는 'count' state에 따라 달립니다
    }, 1000);
    return () => clearInterval(id);
  }, []); // ❗️ 버그: `count`가 종속성으로 지정되지 않았습니다

  return <h1>{count}</h1>;
}
```

빈 종속성 세트, `[]` 는 컴포넌트가 마운트 될 때마다 effect가 한 번만 실행되고 매번 렌더링 시에는 실행되지 않음을 의미합니다. 문제는 `setInterval` 콜백 내에서 `count` 값이 변경되지 않는다는 것입니다. Effect 콜백이 실행되었을 때와 마찬가지로 `count` 값이 0으로 설정된 클로저를 생성했기 때문입니다. 이 콜백은 매초 `setCount(0 + 1)` 를 호출하므로 카운트가 1을 초과하지 않습니다.

종속성 목록으로 `[count]` 를 지정하면 버그가 수정되지만, 변경될 때마다 간격이 재설정됩니다. 효과적으로, 각 `setInterval` 은 지워지기 전에 한 번의 실행 기회를 갖게 됩니다 (`setTimeout` 과 유사). 이는 바람직하지 않을 수 있습니다. 이를 해결하기 위해 `useState` 의 [함수 컴포넌트 업데이트 풀](#)을 사용할 수 있습니다. 현재 `state`를 참조하지 않고 `state`를 변경해야 하는 방법을 지정할 수 있습니다.

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

  useEffect(() => {
    const id = setInterval(() => {
      setCount(c => c + 1); // ✓ 이것은 외부의 'count' 변수에 의존하지 않습니다
    }, 1000);
    return () => clearInterval(id);
  }, []); // ✓ 우리의 effect는 컴포넌트 범위의 변수를 사용하지 않습니다

  return <h1>{count}</h1>;
}
```

(`setCount` 함수 컴포넌트의 정체성은 안정적이므로 생략해도 안전합니다.)

이제 `setInterval` 콜백이 1초에 한 번 실행되지만 `setCount` 에 대한 내부 호출이 `count` 에 최신 값을 사용할 수 있습니다. (여기서는 콜백에서 `c` 라고 함).

더 복잡한 경우 (예: 한 `state`가 다른 `state`에 의존하는 경우) `useReducer` Hook을 사용하여 `state` 업데이트 로직을 effect 외부로 이동해보세요. [이 문서](#)에서는 이를 수행하는 방법에 대한 예를 제공합니다. `useReducer` 의 `dispatch` 함수 컴포넌트의 정체성은 항상 안정적입니다 — 리듀서 함수 컴포넌트가 컴포넌트 내부에서 선언되고 해당 `props`를 읽는 경우에도 마찬가지입니다.

마지막 수단으로, class에서 `this` 와 같은 것을 원한다면 [ref를 사용](#)하여 가변 변수를 보유 할 수 있습니다. 그런 다음 그것을 쓰고 읽을 수 있습니다. 예를 들면.

```
function Example(props) {
  // 최신 props를 ref에 보관해주세요.
  const latestProps = useRef(props);
  useEffect(() => {
    latestProps.current = props;
  });

  useEffect(() => {
    function tick() {
      // 언제든지 최신 props 읽기
      console.log(latestProps.current);
    }
  });

  const id = setInterval(tick, 1000);
  return () => clearInterval(id);
}, []); // 이 effect는 다시 실행되지 않습니다
}
```

변형에 의존하면 컴포넌트를 예측하기 어렵기 때문에 더 나은 대안을 찾을 수 없는 경우에만 이 작업을 수행해주세요. 제대로 옮겨지지 않는 특정 패턴이 있는 경우 실행 가능한 예시 코드로 [문제를 제출하면](#) 도움을 드릴 수 있습니다.

shouldComponentUpdate는 어떻게 구현합니까?

함수 컴포넌트를 `React.memo`로 래핑하여 props를 얇게 비교할 수 있습니다.

```
const Button = React.memo((props) => {
  // 여러분의 컴포넌트
});
```

Hook처럼 구성하지 않기 때문에 Hook이 아닙니다. `React.memo`는 `PureComponent`와 동일하지만, props만 비교합니다. (두 번째 인수를 추가하여 이전 및 새 props를 받는 커스텀 비교 함수 컴포넌트를 지정할 수도 있습니다. `true`를 반환하면 업데이트를 건너뜁니다.)

`React.memo`는 비교할 단일 state 객체가 없기 때문에 state를 비교하지 않습니다. 하지만 자식들을 순수하게 만들 수도 있고 `useMemo`를 사용하여 개별 자식들을 최적화 할 수도 있습니다.

계산을 메모이제이션 하는 법?

`useMemo` Hook을 사용하면 이전 계산을 "기억"하여 여러 렌더링 간에 계산을 캐시 할 수 있습니다.

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

이 코드는 `computeExpensiveValue(a, b)`를 호출합니다. 그러나 종속성 `[a, b]`가 마지막 값 이후로 변경되지 않은 경우 `useMemo`는 두 번째 호출을 건너뛰고 반환된 마지막 값을 재사용합니다.

`useMemo`에 전달된 함수는 렌더링 중에 실행됩니다. 렌더링하는 동안 일반적으로 하지 않는 작업은 하지마세요. 예를 들어 부작용은 `useMemo`가 아니라 `useEffect`에 속합니다.

의미론적 보장이 아닌 성능 최적화로 `useMemo`를 사용할 수 있습니다. 미래에 React는 이전에 메모한 일부 값을 "잊고" 다음 렌더링에서 다시 계산하도록 선택할 수 있습니다. 오프 스크린 컴포넌트를 위한 메모리를 확보합니다. `useMemo` 없이도 계속 작동하도록 코드를 작성한 다음 추가하여 성능을 최적화해주세요. (값을 다시 계산해서는 안 되는 드문 경우의 경우, `ref`를 [느리게 초기화](#) 할 수 있습니다.)

편리하게, `useMemo`를 사용하면 자식의 값비싼 리렌더링을 건너뛸 수 있습니다.

```
function Parent({ a, b }) {
  // 'a'가 변경된 경우에만 다시 렌더링 됩니다:
  const child1 = useMemo(() => <Child1 a={a} />, [a]);
  // 'b'가 변경된 경우에만 다시 렌더링 됩니다:
  const child2 = useMemo(() => <Child2 b={b} />, [b]);
  return (
    <>
      {child1}
      {child2}
    </>
  )
}
```

Hook 호출은 루프 내부에 배치될 수 [없기](#) 때문에 이 방법은 루프에서 작동하지 않습니다. 그러나 목록 항목에 대해 별도의 컴포넌트를 추출하고 거기에서 `useMemo`를 호출 할 수 있습니다.

고비용의 객체를 지연해서 생성하는 법?

`useMemo` 를 사용하면 종속성이 동일한 경우 값비싼 계산을 메모 할 수 있습니다. 그러나 힌트 역할을 할 뿐이며 계산이 다시 실행되지 않는다는 보장은 없습니다. 때로는 객체가 한 번만 생성되었는지 확인해야 합니다.

첫 번째 일반적인 사용 사례는 초기 **state**를 만드는 데 비용이 많이 드는 경우입니다.

```
function Table(props) {
  //⚠️ createRows()는 모든 렌더링에서 호출됩니다
  const [rows, setRows] = useState(createRows(props.count));
  // ...
}
```

무시된 초기 **state**를 다시 생성하지 않으려면 `useState`에 함수 컴포넌트를 전달할 수 있습니다.

```
function Table(props) {
  //✅ createRows()는 한 번만 호출됩니다
  const [rows, setRows] = useState(() => createRows(props.count));
  // ...
}
```

React는 첫 번째 렌더링 중에만 함수 컴포넌트를 호출합니다. [useState API 참조](#)를 확인하세요.

때때로 `useRef()` 초기값을 다시 작성하지 않으려고 할 수도 있습니다. 예를 들어 명령형 class 인스턴스가 한 번만 생성되도록 하고 싶을 수 있습니다.

```
function Image(props) {
  //⚠️ IntersectionObserver는 모든 렌더링에서 생성됩니다
  const ref = useRef(new IntersectionObserver(onIntersect));
  // ...
}
```

`useRef`는 `useState`와 같은 특수 함수 컴포넌트 오버로드를 허용하지 않습니다. 대신 느리게 생성하고 설정하는 자체 함수 컴포넌트를 작성할 수 있습니다.

```
function Image(props) {
  const ref = useRef(null);

  //✅ IntersectionObserver는 한 번 느리게 생성됩니다
  function getObserver() {
    if (ref.current === null) {
      ref.current = new IntersectionObserver(onIntersect);
    }
    return ref.current;
  }

  // 필요할 때 getObserver()를 호출해주세요
  // ...
}
```

이렇게 하면 처음에 진정으로 필요할 때까지는 값비싼 객체를 만들지 않아도 됩니다. Flow 또는 TypeScript를 사용하는 경우 편의를 위해 `getObserver()`에 nullable이 아닌 유형을 제공 할 수도 있습니다.

렌더링에서 함수 컴포넌트를 만들기 때문에 Hook이 느려집니까?

아니요. 최신 브라우저에서 class와 비교해 클로저의 원시적 성능은 극단적인 시나리오를 제외하고는 크게 다르지 않습니다.

또한 Hook 디자인이 몇 가지 면에서 더 효율적이라는 것을 고려해주세요.

- Hook은 class 인스턴스를 만들고 생성자에서 이벤트 핸들러를 바인딩하는 비용과 같이 class에 필요한 많은 오버헤드를 방지합니다.
- **Hook을 사용하는 관용적 코드에는 고차 컴포넌트, 렌더링 props 및 context를 사용하는 코드 베이스에서 널리 알려진 깊은 컴포넌트 트리 중첩이 필요하지 않습니다.** 컴포넌트 트리가 작을수록 React는 할 일이 적습니다.

전통적으로 React의 인라인 함수 컴포넌트와 관련된 성능 문제는 각 렌더에서 새 콜백을 전달하면 자식 컴포넌트에서 `shouldComponentUpdate` 최적화가 중단되는 방식과 관련이 있습니다. Hook은 세 가지 측면에서 이 문제에 접근합니다.

- `useCallback` Hook을 사용하면 `shouldComponentUpdate` 가 계속 작동하도록 다시 렌더링간에 동일한 콜백 참조를 유지 할 수 있습니다.

```
// `a` 또는 `b`가 변경되지 않으면 변경되지 않습니다
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);
```

- `useMemo` Hook을 사용하면 개별 자식들이 업데이트되는 시기를 보다 쉽게 제어할 수 있음으로 순수한 컴포넌트의 필요성이 줄어듭니다.
- 마지막으로, `useReducer` Hook은 아래에 설명된 것처럼 콜백을 깊이 전달할 필요성을 줄여줍니다.

콜백 전달을 피하는 법?

우리는 대부분의 사람이 모든 레벨의 컴포넌트 트리를 통해 콜백을 수동으로 전달하는 것을 좋아하지 않는다는 것을 발견했습니다. 더 명백 하지만 마치 "배관"이 많은 것처럼 느껴질 수 있습니다.

큰 컴포넌트 트리에서 권장되는 대안은 `context`를 통해 `useReducer`에서 `dispatch` 함수 컴포넌트를 전달하는 것입니다.

```
const TodosDispatch = React.createContext(null);

function TodosApp() {
  // 주의: `dispatch`는 다시 렌더링 간에 변경되지 않습니다
  const [todos, dispatch] = useReducer(todosReducer);

  return (
    <TodosDispatch.Provider value={dispatch}>
      <DeepTree todos={todos} />
    </TodosDispatch.Provider>
  );
}
```

`TodosApp` 내의 트리에 있는 모든 자식은 `dispatch` 기능을 사용하여 `TodosApp`에 작업을 전달할 수 있습니다.

```
function DeepChild(props) {
  // 작업을 수행하려면 context에서 dispatch를 얻을 수 있습니다.
  const dispatch = useContext(TodosDispatch);
```

```

function handleClick() {
  dispatch({ type: 'add', text: 'hello' });
}

return (
  <button onClick={handleClick}>Add todo</button>
);
}

```

이것은 유지 보수 관점에서 더 편리하고 (콜백을 계속 전달할 필요가 없음) 콜백 문제를 모두 방지합니다. 이처럼 `dispatch`를 전달하는 것이 심층 업데이트에 권장되는 패턴입니다.

애플리케이션 `state`를 `props` (더욱 명백한) 또는 `context`로 (매우 심층적인 업데이트에 더 편리함) 전달할지 여부를 선택할 수 있습니다. `context`를 사용하여 `state`를 전달하는 경우에도 두 가지 `context` 유형을 사용해주세요. `dispatch` `context`는 변경되지 않음으로 이를 읽는 컴포넌트는 애플리케이션 `state`가 필요하지 않은 한 다시 렌더링할 필요가 없습니다.

useCallback에서 자주 변경되는 값을 읽는 방법?

주의

`props`의 개별 콜백보다는 `context`에서 `dispatch`를 전달하는 것이 좋습니다. 아래 접근 방식은 완전성과 탈출구로만 여기에서 언급됩니다.

드물게 `useCallback`을 사용하여 콜백을 메모해야 할 수도 있지만, 내부 함수 컴포넌트를 너무 자주 다시 만들어야 하므로 메모가 제대로 작동하지 않습니다. 메모하는 함수 컴포넌트가 이벤트 핸들러이고 렌더링 중에 사용되지 않는 경우 `ref`를 인스턴스 변수로 사용하고 마지막으로 커밋 된 값을 수동으로 저장할 수 있습니다.

```

function Form() {
  const [text, updateText] = useState('');
  const textRef = useRef();

  useEffect(() => {
    textRef.current = text; // ref에 쓰기
  });

  const handleSubmit = useCallback(() => {
    const currentText = textRef.current; // ref에서 읽기
    alert(currentText);
  }, [textRef]); // [text]처럼 handleSubmit를 다시 만들지 마세요

  return (
    <>
      <input value={text} onChange={e => updateText(e.target.value)} />
      <ExpensiveTree onSubmit={handleSubmit} />
    </>
  );
}

```

이것은 다소 복잡한 패턴이지만 필요한 경우에 탈출구 최적화를 수행 할 수 있음을 보여줍니다. 커스텀 Hook으로 추출하면 견딜 수 있습니다.

```

function Form() {
  const [text, updateText] = useState('');
  // 'text'가 변경되어도 메모합니다:
  const handleSubmit = useEventCallback(() => {
    alert(text);
  }, [text]);

  return (
    <>
      <input value={text} onChange={e => updateText(e.target.value)} />
      <ExpensiveTree onSubmit={handleSubmit} />
    </>
  );
}

function useEventCallback(fn, dependencies) {
  const ref = useRef() => {
    throw new Error('Cannot call an event handler while rendering.');
  };

  useEffect(() => {
    ref.current = fn;
  }, [fn, ...dependencies]);

  return useCallback(() => {
    const fn = ref.current;
    return fn();
  }, [ref]);
}

```

두 경우 모두 이 패턴을 권장하지 않으며 완전성을 위해 여기에 표시합니다. 대신 콜백을 깊게 전달하지 않는 것이 좋습니다.

Hook의 이면

React는 Hook 호출을 컴포넌트와 어떻게 연관시키는가?

React는 현재 렌더링 컴포넌트를 추적합니다. [Rules of Hook](#) 덕분에 Hook은 React 컴포넌트 (또는 React 컴포넌트에서만 호출되는 커스텀 Hook)에서만 호출된다는 것을 알고 있습니다.

각 컴포넌트와 관련된 "메모리 셀"의 내부 목록이 있습니다. 이것은 단지 데이터를 넣을 수 있는 JavaScript 객체입니다. `useState()` 와 같은 Hook을 호출하면 현재 셀을 읽거나 첫 번째 렌더링 중에 초기화한 다음 포인터를 다음 셀로 이동합니다. 이것이여 `useState()` 호출이 각각 독립적인 로컬 state를 얻는 방법입니다.

Hook에 대한 선행 기술은 무엇입니까?

Hook은 여러 소스에서 아이디어를 합성합니다.

- [react-future](#) 저장소에서 기능적 API를 사용한 이전 실험.
- [Ryan Florence](#)의 [Reactions](#) 컴포넌트를 포함하여 렌더링 props API를 사용한 React 커뮤니티의 실험.
- 렌더링 props를 위한 편의 문법으로 제안된 [Dominic Gannaway](#)의 `adopt` 키워드.
- [DisplayScript](#)의 state 변수 및 state 셀.
- [ReasonReact](#)의 리듀서 컴포넌트.
- Rx의 구독.
- Multicore OCaml의 대수 효과.

Sebastian Markbåge는 Hook의 원래 디자인을 생각해냈고, 나중에 Andrew Clark, Sophie Alpert, Dominic Gannaway 및 React 팀의 다른 구성원에 의해 개선되었습니다.

[Go to TOC](#)

Hook의 개요

Hook은 React 버전 16.8부터 React 요소로 새로 추가되었습니다. Hook을 이용하여 기존 Class 바탕의 코드를 작성할 필요 없이 상태 값과 여러 React의 기능을 사용할 수 있습니다.

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

function Example() {
  // "count"라는 새로운 상태 값을 정의합니다.
  const [count, setCount] = useState(0);

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

`useState`는 우리가 "Hook"에서 처음 배우게 될 함수입니다. 이 예시는 단지 맛보기에 불과합니다. 아직 이해되지 않아도 걱정하지 마세요!

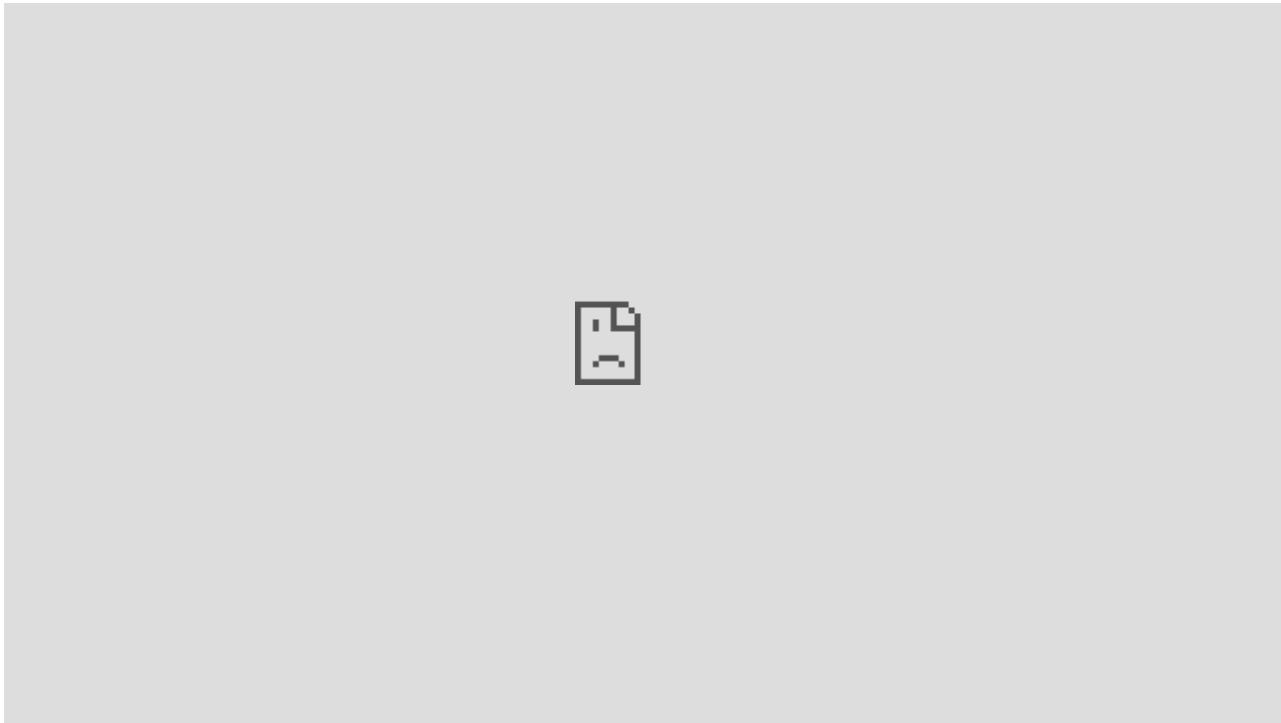
[다음 페이지](#)에서 **Hook**에 대해 배울 수 있습니다. 이 페이지에서는 우리가 왜 Hook을 React에 추가했는지, 그리고 Hook이 애플리케이션을 작성하는 데 어떠한 도움을 주는지 설명할 것입니다.

주의

React 16.8.0은 Hook을 지원하는 첫 번째 배포입니다. 업그레이드 시 React DOM을 포함한 모든 패키지의 업데이트를 진행해주세요. React Native는 [v0.59](#)부터 Hook을 지원합니다.

소개 영상

React Conf 2018에서 Sophie Alpert와 Dan Abramov가 Hook을 소개했습니다. 이어서 Ryan Florence가 Hook을 사용한 애플리케이션 리팩토링의 과정을 보여주었습니다. 아래 영상에서 확인해보세요.



이미 작성된 코드에 필요한 변화사항은 없습니다.

얘기하기에 앞서, Hook의 특징은 다음과 같습니다.

- 선택적 사용 기존의 코드를 다시 작성할 필요 없이 일부의 컴포넌트들 안에서 Hook을 사용할 수 있습니다. 그러나 당장 Hook이 필요 없다면, Hook을 사용할 필요는 없습니다.
- 100% 이전 버전과의 호환성** Hook은 호환성을 깨뜨리는 변화가 없습니다.
- 현재 사용 가능 Hook은 배포 v16.8.0에서 사용할 수 있습니다.

React에서 **Class**를 제거할 계획은 없습니다. Hook의 점진적 적용 전략에 대해 [아래 영역](#) 페이지에서 읽을 수 있습니다.

Hook은 알고 있는 React 컨셉을 대체하지 않습니다. 대신에, Hook은 props, state, context, refs, 그리고 lifecycle와 같은 React 개념에 좀 더 직관적인 API를 제공합니다. 또한 Hook은 이 개념들을 여기 위해 새로운 강력한 방법을 제공합니다.

단지 **Hook에 대해 배우길 원한다면, 다음 페이지**로 바로 이동해도 됩니다. 또한 왜 우리가 Hook을 추가했는지, 그리고 어떻게 애플리케이션을 다시 작성할 필요 없이 어떻게 Hook을 이용하는지 알고 싶다면 이 페이지를 계속 읽을 필요가 있습니다.

동기

Hook은 우리가 5년간 React로 컴포넌트를 작성하고 유지하는 동안 부딪혔던 수 많은 문제들을 해결했습니다. React를 배우는 중이든, 매일 사용하든, 심지어 비슷한 컴포넌트 모델과 함께 다른 라이브러리를 선호하든지 간에, 사용자는 이러한 문제를 인식해 왔을 것입니다.

컴포넌트 사이에서 상태 로직을 재사용하기 어렵습니다.

React는 컴포넌트간에 재사용 가능한 로직을 붙이는 방법을 제공하지 않습니다. (예를 들어, 스토어에 연결하는 것) 이전부터 React를 사용해왔다면, [render props](#)이나 [고차 컴포넌트](#)와 같은 패턴을 통해 이러한 문제를 해결하는 방법에 익숙할 것입니다. 그러나 이런 패턴의 사용은 컴포넌트의 재구성을 강요하며, 코드의 추적을 어렵게 만듭니다. React 개발자 도구에서 React 애플리케이션을 본다면, providers, consumers, 고차 컴포넌트, render props 그리고 다른 추상화에 대한 레이어로 둘러싸인 "래퍼 지옥(wrapper hell)"을 볼 가능성이 높습니다. [개발자 도구에서 걸려낼 수 있지만](#), 이 문제의 요점은 심층적이었습니다. React는 상태 관련 로직을 공유하기 위해 좀 더 좋은 기초 요소가 필요했습니다.

Hook을 사용하면 컴포넌트로부터 상태 관련 로직을 추상화할 수 있습니다. 이를 이용해 독립적인 테스트와 재사용이 가능합니다. **Hook**은 계층의 변화 없이 상태 관련 로직을 재사용할 수 있도록 도와줍니다. 이것은 많은 컴포넌트 혹은 커뮤니티 사이에서 Hook을 공유하기 쉽게 만들어줍니다.

우리는 [자신만의 Hook을 구축하는 법](#)에 대해 좀 더 알아볼 것입니다.

복잡한 컴포넌트들은 이해하기 어렵습니다.

우리는 때론 간단하게 시작했지만 관리하기가 힘들어지는 상태 관련 로직들과 사이드 이펙트가 있는 컴포넌트들을 유지보수해야 합니다. 각 생명주기 메서드에는 자주 관련 없는 로직이 섞여들어가고는 합니다. 예시로 `componentDidMount` 와 `componentDidUpdate` 는 컴포넌트안에서 데이터를 가져오는 작업을 수행할 때 사용 되어야 하지만, 같은 `componentDidMount`에서 이벤트 리스너를 설정하는 것과 같은 관계없는 로직이 포함되기도 하며, `componentWillUnmount`에서 `cleanup` 로직을 수행하기도 합니다. 함께 변경되는 상호 관련 코드는 분리되지만 이와 연관 없는 코드들은 단일 메서드로 결합합니다. 이로 인해 버그가 쉽게 발생하고 무결성을 너무나 쉽게 해칩니다.

위와 같은 예시에서, 상태 관련 로직은 한 공간안에 둑여 있기 때문에 이런 컴포넌트들을 작게 분리하는 것은 불가능하며 테스트하기도 어렵습니다. 이 때문에 많은 사용자들은 React를 별도의 상태 관리 라이브러리와 함께 결합해서 사용해왔습니다. 그러나, 이런 상태 관리 라이브러리는 종종 너무 많은 추상화를 하고, 서로 다른 파일들 사이에서 건너뛰기를 요구하며 컴포넌트 재사용을 더욱더 어렵게 만들었습니다.

이같은 문제를 해결하기위해, 생명주기 메서드를 기반으로 쪼개는 것 보다는, **Hook을 통해 서로 비슷한 것을 하는 작은 함수의 묶음으로 컴포넌트를 나누는 방법**을 사용할 수 있습니다. ([구독 설정 및 데이터를 불러오는 것과 같은 로직](#)) 또한 이러한 로직의 추적을 쉽게 할 수 있도록 리듀서를 활용해 컴포넌트의 지역 상태 값을 관리하도록 할 수 있습니다.

이에 대해 [Effect Hook](#)에서 더 알아볼 것입니다.

Class은 사람과 기계를 혼동시킵니다.

React에서의 Class 사용을 위해서는 JavaScript의 `this` 키워드가 어떻게 작동하는지 알아야만 합니다. JavaScript의 `this` 키워드는 대부분의 다른 언어에서와는 다르게 작동함으로 사용자에게 큰 혼란을 주었으며, 코드의 재사용성과 구성을 매우 어렵게 만들고는 했습니다. 또한 class의 사용을 위해 이벤트 핸들러가 등록되는 방법을 정확히 파악해야 했으며, 이는 [ES2022 public class fields](#)가 없을 때 코드를 매우 장황하게 만들었습니다. 사용자들은 `props`, `state`, 그리고 `top-down` 데이터 흐름을 완벽하게 하고도, Class의 이해에는 어려움을 겪고는 했습니다. React 내의 함수와 Class 컴포넌트의 구별, 각 요소의 사용 타이밍 등은 숙련된 React 개발자 사이에서도 의견이 일치하지 않습니다.

React는 지난 5년 동안 널리 사용되어 왔으며, React의 개발진은 5년 뒤에도 React가 지금과 같이 널리 이용되길 원합니다. [Svelte](#), [Angular](#), [Glimmer](#) 등에서 보여주듯이, 컴포넌트를 [미리 컴파일해놓는 방식](#)에는 높은 잠재력이 있습니다. 템플릿에 한정하지 않는다면 더 그렇고요. 개발진은 최근 [Prepack](#)을 사용한 [컴포넌트 folding](#)에 대해서 실험해왔고 긍정적인 결과를 보았습니다만, Class 컴포넌트가

이러한 최적화를 더 느린 경로로 되돌리는 의도하지 않은 패턴을 장려할 수 있다는 것을 발견했습니다. Class는 최근 사용되는 도구에서도 많은 문제를 일으킵니다. 예를 들어 Class는 코드의 최소화를 힘들게 만들고, 핫 리로딩을 깨지기 쉽고 신뢰할 수 없게 만듭니다. 우리는 코드가 최적화 가능한 경로에서 유지될 가능성이 더 높은 API를 제공하길 원하였습니다.

이러한 문제를 해결하기 위해, **Hook은 Class없이 React 기능들을 사용하는 방법을 제시합니다.** 개념적으로 React 컴포넌트는 항상 함수에 더 가깝습니다. Hook은 React의 정신을 희생하지 않고 함수의 사용을 권장합니다. Hook은 명령형 코드로 해결책을 찾을 수 있게 해주며 복잡한 함수형 또는 반응형 프로그래밍 기술을 배우도록 요구하지 않습니다.

예시

[Hook 살펴보기](#)는 Hook을 배우는데 좋은 공간입니다.

점진적 적용 전략

요약: React로부터 Class를 제거할 계획은 없습니다.

우리는 React를 사용하는 개발자들이 프로덕트의 개발에 더 초점을 맞추고 있으며, 변경되는 API의 요소들을 새롭게 공부할 시간이 없다는 걸 알고 있습니다. Hook은 매우 새로운 기술이며, Hook을 배우거나 적용하기 전에 더 많은 예시와 자습서를 기다리는 게 나을 수도 있습니다.

또한 React에 추가되는 새로운 기능에 대한 기준점이 매우 높다는 것을 알고 있습니다. 궁금해하는 독자들을 위해 동기부여에 대한 구체적인 내용을 담고 있는 [자세한 RFC](#)를 준비했고 특정 설계 결정 및 관련 선행 기술에 대한 추가적인 관점을 제공합니다.

결정적으로, Hook은 존재하는 코드와 함께 나란히 작동함으로써 점진적으로 적용할 수 있습니다. Hook의 적용을 서두를 필요는 없습니다. 이미 사용중인 복잡한 Class 컴포넌트들에 대한 "큰 리팩토링"을 피하기를 권장합니다. "Hook을 적용하기"에는 고민의 시간이 필요합니다. 경험을 토대로 하여, 상대적으로 중요성이 덜한 새 컴포넌트에서 Hook을 사용하는 것이 최고의 연습입니다. 그리고 여러분의 모든 팀원이 Hook에 대해 안정감을 느끼는지 확인하는 게 좋습니다. Hook을 이용 후, 긍정적, 부정적 의견 모두 [의견 보내기](#)를 이용해 보내주시면 감사하겠습니다.

React의 개발자들은 현재 사용중인 Class 사례를 Hook으로 교체하는 것을 염두에 두고는 있지만, **미래에도 계속 Class 컴포넌트들을 지원할 예정입니다.** 페이스북은 수만 개의 Class 컴포넌트들을 작성했지만 이를 재작성할 계획은 없습니다. 대신에, 새로운 코드에서 기존 코드와 나란히 Hook을 사용할 계획입니다.

자주 묻는 질문

[Hook FAQ 페이지](#)에서 Hook에 대한 자주 묻는 질문과 답변을 확인할 수 있습니다.

다음 단계

이 페이지를 다 읽었을 때, Hook이 해결하려는 문제들에 대한 대략적인 개념을 이해하고 있어야 합니다. 그러나 아직 구체적으로 명확화된 않을 것입니다. 걱정하지 마세요! [다음 페이지](#)로 가서 예시를 통해서 Hook에 대해 배워 봅시다!

[Go to TOC](#)

Hook 개요

*Hook*은 React 16.8에 새로 추가된 기능입니다. *Hook*은 class를 작성하지 않고도 state와 다른 React의 기능들을 사용할 수 있게 해줍니다.

*Hook*은 하위 호환성을 가지고 있습니다. 이 문서는 React에 경험이 있는 사용자를 대상으로 *Hook*에 대해 간략히 소개합니다. 이 문서는 빠르게 진행됩니다. 혼란스러운 경우에는 다음과 같은 노란색 박스를 참고하세요.

자세한 설명

React에 *Hook*을 도입하는 이유를 알고 싶다면 [Motivation](#) 파트를 읽어보세요.

↑↑↑ 각 섹션 마지막에는 이런 박스가 있습니다. 자세한 설명을 보시려면 링크를 따라가시면 됩니다.

📌 State Hook

버튼을 클릭하면 값이 증가하는 간단한 카운터 예시가 여기 있습니다.

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

function Example() {
  // "count"라는 새 상태 변수를 선언합니다
  const [count, setCount] = useState(0);

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

여기서 `useState` 가 바로 *Hook*입니다(이게 무슨 의미인지는 앞으로 알아보겠습니다). *Hook*을 호출해 함수 컴포넌트(function component) 안에 `state`를 추가했습니다. 이 `state`는 컴포넌트가 다시 렌더링 되어도 그대로 유지될 것입니다. `useState`는 현재의 `state` 값과 이 값을 업데이트하는 함수를 쌍으로 제공합니다. 우리는 이 함수를 이벤트 핸들러나 다른 곳에서 호출할 수 있습니다. 이것은 class의 `this.setState` 와 거의 유사하지만, 이전 `state`와 새로운 `state`를 합치지 않는다는 차이점이 있습니다. (`useState` 와 `this.state` 를 비교하는 예시가 [Using the State Hook](#) 문서에 있으니 한번 보세요.)

`useState` 는 인자로 초기 `state` 값을 하나 받습니다. 카운터는 0부터 시작하기 때문에 위 예시에서는 초기값으로 `0` 을 넣어준 것입니다. `this.state` 와는 달리 `useState` Hook의 `state`는 객체일 필요가 없습니다. 물론 원한다면 그렇게도 가능하지만요. 이 초기값은 첫 번째 렌더링에만 딱 한번 사용됩니다.

여러 `state` 변수 선언하기

하나의 컴포넌트 내에서 *State Hook*을 여러 개 사용할 수도 있습니다.

```
function ExampleWithManyStates() {
  // 상태 변수를 여러 개 선언했습니다!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
  // ...
}
```

배열 구조 분해(destructuring) 문법은 `useState`로 호출된 state 변수들을 다른 변수명으로 할당할 수 있게 해줍니다. 이 변수명은 `useState` API와 관련이 없습니다. 대신에 React는 매번 렌더링할 때 `useState`가 사용된 순서대로 실행할 것입니다. 왜 이렇게 동작하는지는 나중에 살펴보겠습니다.

근데 Hook이 뭔가요?

Hook은 함수 컴포넌트에서 React state와 생명주기 기능(lifecycle features)을 "연동(hook into)"할 수 있게 해주는 함수입니다. Hook은 class 안에서는 동작하지 않습니다. 대신 class 없이 React를 사용할 수 있게 해주는 것입니다. (하지만 이미 짜놓은 컴포넌트를 모조리 재작성하는 것은 권장하지 않습니다. 대신 새로 작성하는 컴포넌트부터는 Hook을 이용하시면 됩니다.)

React는 `useState` 같은 내장 Hook을 몇 가지 제공합니다. 컴포넌트 간에 상태 관련 로직을 재사용하기 위해 Hook을 직접 만드는 것도 가능합니다. 일단 내장 Hook을 먼저 보겠습니다.

자세한 설명

State Hook에 대해서는 독립된 문서 [Using the State Hook](#)에서 더 알아보세요.

⚡ Effect Hook

React 컴포넌트 안에서 데이터를 가져오거나 구독하고, DOM을 직접 조작하는 작업을 이전에도 종종 해보셨을 것입니다. 우리는 이런 모든 동작을 "side effects"(또는 짧게 "effects")라고 합니다. 왜냐하면 이것은 다른 컴포넌트에 영향을 줄 수도 있고, 렌더링 과정에서는 구현할 수 없는 작업이기 때문입니다.

Effect Hook, 즉 `useEffect`는 함수 컴포넌트 내에서 이런 side effects를 수행할 수 있게 해줍니다. React class의 `componentDidMount`나 `componentDidUpdate`, `componentWillUnmount`와 같은 목적으로 제공되지만, 하나의 API로 통합된 것입니다. (`useEffect`와 이 세 가지 메서드를 비교하는 예시가 [Using the Effect Hook](#) 문서에 있습니다.)

예를 들어, 이 예시는 React가 DOM을 업데이트한 뒤에 문서의 타이틀을 바꾸는 컴포넌트입니다.

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

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

  // componentDidMount, componentDidUpdate와 비슷합니다
  useEffect(() => {
    // 브라우저 API를 이용해 문서의 타이틀을 업데이트합니다
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
    </div>
  );
}
```

```

        <button onClick={() => setCount(count + 1)}>
          Click me
        </button>
      </div>
    );
}

```

`useEffect` 를 사용하면, React는 DOM을 바꾼 뒤에 "effect" 함수를 실행할 것입니다. Effects는 컴포넌트 안에 선언되어있기 때문에 `props`와 `state`에 접근할 수 있습니다. 기본적으로 React는 매 렌더링 이후에 effects를 실행합니다. 첫 번째 렌더링도 포함해서요. (Class 생명주기(lifecycle)와 다른 점은 [Using the Effect Hook](#) 문서에서 더 자세히 다루고 있습니다.)

Effect를 "해제"할 필요가 있다면, 해제하는 함수를 반환해주면 됩니다. 이는 선택적입니다(optional). 예를 들어, 이 컴포넌트는 친구의 접속 상태를 구독하는 effect를 사용했고, 구독을 해지함으로써 해제해줍니다.

```

import React, { useState, useEffect } from 'react';

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

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);

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

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}

```

이 예시에서 컴포넌트가 unmount될 때 React는 `ChatAPI`에서 구독을 해지할 것입니다. 또한 재 렌더링이 일어나 effect를 재실행하기 전에도 마찬가지로 구독을 해지합니다. (원한다면 `props.friend.id` 가 바뀌지 않았을 때 [재구독을 건너뛰도록 설정](#)할 수 있습니다.)

`useState` 와 마찬가지로 컴포넌트 내에서 여러 개의 effect를 사용할 수 있습니다.

```

function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  function handleStatusChange(status) {

```

```

    setIsOnline(status.isOnline);
}
// ...

```

Hook을 사용하면 구독을 추가하고 제거하는 로직과 같이 서로 관련 있는 코드들을 한군데에 모아서 작성할 수 있습니다. 반면 class 컴포넌트에서는 생명주기 메서드(lifecycle methods) 각각에 쪼개서 넣어야만 했습니다.

자세한 설명

`useEffect`에 대해서는 독립된 문서 [Using the Effect Hook](#)에서 더 알아보세요.

👉 Hook 사용 규칙

Hook은 그냥 JavaScript 함수이지만, 두 가지 규칙을 준수해야 합니다.

- **최상위(at the top level)**에서만 Hook을 호출해야 합니다. 반복문, 조건문, 중첩된 함수 내에서 Hook을 실행하지 마세요.
- **React 함수 컴포넌트** 내에서만 Hook을 호출해야 합니다. 일반 JavaScript 함수에서는 Hook을 호출해서는 안 됩니다. (Hook을 호출할 수 있는 곳이 딱 한 군데 더 있습니다. 바로 직접 작성한 custom Hook 내입니다. 이것에 대해서는 나중에 알아보겠습니다.)

이 규칙들을 강제하기 위해서 [linter plugin](#)을 제공하고 있습니다. 이 규칙들이 제약이 심하고 혼란스럽다고 처음에는 느낄 수 있습니다. 하지만 이것은 Hook이 제대로 동작하기 위해서는 필수적인 조건입니다.

자세한 설명

이 규칙들에 대해서는 독립된 문서 [Rules of Hooks](#)에서 더 알아보세요.

💡 나만의 Hook 만들기

개발을 하다 보면 가끔 상태 관련 로직을 컴포넌트 간에 재사용하고 싶은 경우가 생깁니다. 이 문제를 해결하기 위한 전통적인 방법이 두 가지 있었는데, [higher-order components](#)와 [render props](#)가 바로 그것입니다. Custom Hook은 이들 둘과는 달리 컴포넌트 트리에 새 컴포넌트를 추가하지 않고도 이것을 가능하게 해줍니다.

친구의 접속 상태를 구독하기 위해서 `useState`와 `useEffect` Hook을 사용한 `FriendStatus` 컴포넌트 예시를 다시 한번 보겠습니다. 이 로직을 다른 컴포넌트에서도 재사용하고 싶다고 가정을 해봅시다.

먼저, 이 로직을 `useFriendStatus`라는 custom Hook으로 뽑아냅니다.

```

import React, { useState, useEffect } from 'react';

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

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {

```

```

    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  );
  return isOnline;
}

```

이 Hook은 `friendID`를 인자로 받아서 친구의 접속 상태를 반환해줍니다.

이제 우리는 이것을 여러 컴포넌트에서 사용할 수 있습니다.

```

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

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}

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

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

```

각 컴포넌트의 `state`는 완전히 독립적입니다. Hook은 `state` 그 자체가 아니라, 상태 관련 로직을 재사용하는 방법입니다. 실제로 각각의 Hook 호출은 완전히 독립된 `state`를 가집니다. 그래서 심지어는 한 컴포넌트 안에서 같은 custom Hook을 두 번 쓸 수도 있습니다.

Custom Hook은 기능이라기보다는 컨벤션(convention)에 가깝습니다. 이름이 "`use`"로 시작하고, 안에서 다른 Hook을 호출한다면 그 함수를 custom Hook이라고 부를 수 있습니다. `useSomething`이라는 네이밍 컨벤션은 linter 플러그인이 Hook을 인식하고 버그를 찾을 수 있게 해줍니다.

폼 핸들링, 애니메이션, 선언적 구독(declarative subscriptions), 타이머 등 많은 경우에 custom Hook을 사용할 수 있습니다. 우리는 React 커뮤니티에서 어떤 custom Hook이 만들어질지 정말 기대됩니다.

자세한 설명

Custom Hook에 대해서는 독립된 문서 [Building Your Own Hooks](#)에서 더 알아보세요.

다른 내장 Hook

보편적이지는 않지만 유용하다고 느낄만한 내장 Hook이 몇 가지 더 있습니다. 예를 들어, `useContext` 는 컴포넌트를 종첩하지 않고도 React context를 구독할 수 있게 해줍니다.

```
function Example() {  
  const locale = useContext(LocaleContext);  
  const theme = useContext(ThemeContext);  
  // ...  
}
```

그리고 `useReducer` 는 복잡한 컴포넌트들의 state를 reducer로 관리할 수 있게 해줍니다.

```
function Todos() {  
  const [todos, dispatch] = useReducer(todosReducer);  
  // ...  
}
```

자세한 설명

모든 내장 Hook에 대해서는 독립된 문서 [Hooks API Reference](#)에서 더 알아보세요.

다음 단계

휴, 정말 순식간이었죠! 잘 이해가 안 되는 부분이 있거나 좀 더 깊이 공부하고 싶으면 [State Hook](#)부터 시작해서 다른 문서들을 읽어보시기 바랍니다.

또한 [Hooks API reference](#)와 [Hooks FAQ](#)도 참고하시기 바랍니다.

마지막으로, 우리가 왜 Hook을 추가하는지 그 이유와 앱을 재작성하지 않고도 class와 함께 Hook을 사용하는 방법을 설명한 [소개 페이지](#)도 놓치지 마세요.

[Go to TOC](#)

Hooks API Reference

*Hook*는 React 16.8에서 새로 추가된 개념입니다. Hook을 통해 class를 작성하지 않고도 state와 같은 React 기능들을 사용할 수 있습니다.

이 페이지는 React에 내장된 Hook API를 설명합니다.

Hook이 생소하다면 [Hook 개요](#)를 먼저 읽어 보기 바랍니다. 혹은 [frequently asked questions](#)에서 유용한 정보를 찾을 수도 있습니다.

- [기본 Hook](#)
 - `useState`
 - `useEffect`
 - `useContext`
- [추가 Hooks](#)
 - `useReducer`
 - `useCallback`
 - `useMemo`
 - `useRef`
 - `useImperativeHandle`
 - `useLayoutEffect`
 - `useDebugValue`
 - `useDeferredValue`
 - `useTransition`
 - `useId`
- [Library Hooks](#)
 - `useSyncExternalStore`
 - `useInsertionEffect`

기본 Hook

`useState` [\[#usestate\]](#)

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

상태 유지 값과 그 값을 갱신하는 함수를 반환합니다.

최초로 렌더링을 하는 동안, 반환된 state(`state`)는 첫 번째 전달된 인자(`initialState`)의 값과 같습니다.

`setState` 함수는 state를 갱신할 때 사용합니다. 새 state 값을 받아 컴포넌트 리렌더링을 큐에 등록합니다.

```
setState(newState);
```

다음 리렌더링 시에 `useState` 를 통해 반환받은 첫 번째 값은 항상 갱신된 최신 state가 됩니다.

주의

React는 `setState` 함수 동일성이 안정적이고 리렌더링 시에도 변경되지 않을 것이라는 것을 보장합니다. 이것이 `useEffect`나 `useCallback` 의존성 목록에 이 함수를 포함하지 않아도 무방한 이유입니다.

함수적 갱신

이전 `state`를 사용해서 새로운 `state`를 계산하는 경우 함수를 `useState`로 전달할 수 있습니다. 그 함수는 이전 값을 받아 갱신된 값을 반환할 것입니다. 여기에 `useState`의 양쪽 형태를 사용한 카운터 컴포넌트의 예가 있습니다.

```
function Counter({initialCount}) {
  const [count, setCount] = useState(initialCount);
  return (
    <>
      Count: {count}
      <button onClick={() => setCount(initialCount)}>Reset</button>
      <button onClick={() => setCount(prevCount => prevCount - 1)}>-</button>
      <button onClick={() => setCount(prevCount => prevCount + 1)}>+</button>
    </>
  );
}
```

"+"와 "-" 버튼은 함수 형식을 사용하고 있습니다. 이것은 갱신된 값이 갱신되기 이전의 값을 바탕으로 계산되기 때문입니다. 반면, "Reset" 버튼은 카운트를 항상 0으로 설정하기 때문에 일반적인 형식을 사용합니다.

업데이트 함수가 현재 상태와 정확히 동일한 값을 반환한다면 바로 뒤에 일어날 리렌더링은 완전히 건너뛰게 됩니다.

주의

클래스 컴포넌트의 `setState` 메서드와는 다르게, `useState`는 갱신 객체(update objects)를 자동으로 합치지는 않습니다. 함수 업데이터 품을 객체 전개 연산자와 결합함으로써 이 동작을 복제할 수 있습니다.

```
const [state, setState] = useState({});
setState(prevState => {
  // Object.assign would also work
  return {...prevState, ...updatedValues};
});
```

다른 방법으로는 `useReducer` 가 있는데 이는 여럿값들을 포함한 `state` 객체를 관리하는 데에 더 적합합니다.

지연 초기 state

`initialState` 인자는 초기 렌더링 시에 사용하는 `state`입니다. 그 이후의 렌더링 시에는 이 값은 무시됩니다. 초기 `state`가 고비용 계산의 결과라면, 초기 렌더링 시에만 실행될 함수를 대신 제공할 수 있습니다.

```
const [state, setState] = useState(() => {
  const initialState = someExpensiveComputation(props);
  return initialState;
});
```

state 갱신의 취소

State Hook을 현재의 state와 동일한 값으로 갱신하는 경우 React는 자식을 렌더링 한다거나 무엇을 실행하는 것을 회피하고 그 처리를 종료합니다. (React는 `Object.is` 비교 알고리즘을 사용합니다.)

실행을 회피하기 전에 React에서 특정 컴포넌트를 다시 렌더링하는 것이 여전히 필요할 수도 있다는 것에 주의하세요. React가 불필요하게 트리에 그 이상으로 「더 깊은」는 관여하지 않을 것이므로 크게 신경 쓰지 않으셔도 됩니다만, 렌더링 시에 고비용의 계산을 하고 있으면 `useMemo`를 사용하여 그것들을 최적화할 수 있습니다.

Batching of state updates

React may group several state updates into a single re-render to improve performance. Normally, this improves performance and shouldn't affect your application's behavior.

Before React 18, only updates inside React event handlers were batched. Starting with React 18, [batching is enabled for all updates by default](#). Note that React makes sure that updates from several *different* user-initiated events -- for example, clicking a button twice -- are always processed separately and do not get batched. This prevents logical mistakes.

In the rare case that you need to force the DOM update to be applied synchronously, you may wrap it in `flushSync`. However, this can hurt performance so do this only where needed.

useEffect {#useeffect}

```
useEffect(didUpdate);
```

명령형 또는 어떤 effect를 발생하는 함수를 인자로 받습니다.

변형, 구독, 타이머, 로깅 또는 다른 부작용(side effects)은 (React의 렌더링 단계에 따르면) 함수 컴포넌트의 본문 안에서는 허용되지 않습니다. 이를 수행한다면 그것은 매우 혼란스러운 버그 및 UI의 불일치를 야기하게 될 것입니다.

대신에 `useEffect`를 사용하세요. `useEffect`에 전달된 함수는 화면에 렌더링이 완료된 후에 수행되게 될 것입니다. React의 순수한 함수적인 세계에서 명령적인 세계로의 탈출구로 생각하세요.

기본적으로 동작은 모든 렌더링이 완료된 후에 수행됩니다만, 어떤 값이 변경되었을 때만 실행되게 할 수도 있습니다.

effect 정리

effect는 종종 컴포넌트가 화면에서 제거될 때 정리해야 하는 리소스를 만듭니다. 가령 구독이나 타이머 ID와 같은 것입니다. 이것을 수행하기 위해서 `useEffect`로 전달된 함수는 정리(clean-up) 함수를 반환할 수 있습니다. 예를 들어 구독을 생성하는 경우는 아래와 같습니다.

```
useEffect(() => {
  const subscription = props.source.subscribe();
  return () => {
    // Clean up the subscription
    subscription.unsubscribe();
  };
});
```

정리 함수는 메모리 누수 방지를 위해 UI에서 컴포넌트를 제거하기 전에 수행됩니다. 더불어, 컴포넌트가 (그냥 일반적으로 수행하는 것처럼) 여러 번 렌더링 된다면 다음 **effect**가 수행되기 전에 이전 **effect**는 정리됩니다. 위의 예에서, 매 갱신마다 새로운 구독이 생성되고 볼 수 있습니다. 갱신마다 불필요한 수행이 발생하는 것을 회피하기 위해서는 다음 절을 참고하세요.

effect 타이밍

`componentDidMount` 와 `componentDidUpdate` 와는 다르게, `useEffect`로 전달된 함수는 지역 이벤트 동안에 레이아웃 배치와 그리기를 완료한 후 발생합니다. 이것은 구독이나 이벤트 핸들러를 설정하는 것과 같은 다수의 공통적인 부작용에 적합합니다. 왜냐면 대부분의 작업이 브라우저에서 화면을 업데이트하는 것을 차단해서는 안 되기 때문입니다.

그렇지만, 모든 **effect**가 지역될 수는 없습니다. 예를 들어 사용자에게 노출되는 DOM 변경은 사용자가 노출된 내용의 불일치를 경험하지 않도록 다음 화면을 다 그리기 이전에 동기화 되어야 합니다. (그 구분이란 개념적으로는 수동적 이벤트 리스너와 능동적 이벤트 리스너의 차이와 유사합니다) 이런 종류의 **effect**를 위해 React는 `useLayoutEffect`라는 추가적인 Hook을 제공합니다. 그것은 `useEffect`와 동일한 시그니처를 가지고 있고 그것이 수행될 때에만 차이가 납니다.

Additionally, starting in React 18, the function passed to `useEffect` will fire synchronously **before** layout and paint when it's the result of a discrete user input such as a click, or when it's the result of an update wrapped in `flushSync`. This behavior allows the result of the effect to be observed by the event system, or by the caller of `flushSync`.

Note

This only affects the timing of when the function passed to `useEffect` is called - updates scheduled inside these effects are still deferred. This is different than `useLayoutEffect`, which fires the function and processes the updates inside of it immediately.

Even in cases where `useEffect` is deferred until after the browser has painted, it's guaranteed to fire before any new renders. React will always flush a previous render's effects before starting a new update.

조건부 effect 발생

effect의 기본 동작은 모든 렌더링을 완료한 후 **effect**를 발생하는 것입니다. 이와 같은 방법으로 의존성 중 하나가 변경된다면 **effect**는 항상 재생성됩니다.

그러나 이것은 이전 섹션의 구독 예시와 같이 일부 경우에는 과도한 작업일 수 있습니다. `source` props가 변경될 때에만 필요한 것이라면 매번 갱신할 때마다 새로운 구독을 생성할 필요는 없습니다.

이것을 수행하기 위해서는 `useEffect`에 두 번째 인자를 전달하세요. 이 인자는 **effect**가 종속되어 있는 값의 배열입니다. 이를 적용한 예는 아래와 같습니다.

```
useEffect(
  () => {
    const subscription = props.source.subscribe();
    return () => {
      subscription.unsubscribe();
    };
  },
  [props.source],
);
```

자 이제, `props.source` 가 변경될 때에만 구독이 재생성될 것입니다.

주의

이 최적화를 사용하는 경우, 값의 배열이 시간이 지남에 따라 변경되고 **effect**에 사용되는 컴포넌트 범위의 모든 값들(예를 들어, **props와 state와 같은 값들**)을 포함하고 있는지 확인하세요. 그렇지 않다면 여러분의 코드는 이전 렌더링에서 설정된 오래된 값을 참조하게 될 것입니다. [how to deal with functions](#)와 [array values change too often](#) 할 때 무엇을 할 것인지에 대해서 조금 더 알아보세요.

effect를 수행하고 (`mount`를 하거나 `unmount` 할 때) 그것을 한 번만 실행하고 싶다면 두 번째 인자로 빈 배열(`[]`)을 전달할 수 있습니다. 이를 통해 **effect**는 React에게 `props`나 `state`에서 가져온 어떤 값에도 의존하지 않으므로, 다시 실행할 필요가 전혀 없다는 것을 알려주게 됩니다. 이것을 특별한 경우로 간주하지는 않고, 의존성 값의 배열이 항상 어떻게 동작하는지 직접적으로 보여주는 것뿐입니다.

빈 배열(`[]`)을 전달한다면 **effect** 안에 있는 `props`와 `state`는 항상 초기값을 가지게 될 것입니다. 두 번째 인자로써 `[]`을 전달하는 것이 친숙한 `componentDidMount` 와 `componentWillUnmount`에 의한 개념과 비슷하게 느껴지겠지만, **effect**가 너무 자주 리렌더링 되는 것을 피하기 위한 보통 [더 나은 해결책](#)이 있습니다. 또한 브라우저가 모두 그려질 때까지 React는 `useEffect`의 수행을 지연하기 때문에 다른 작업의 수행이 문제가 되지는 않는다는 것을 잊지 마세요.

`eslint-plugin-react-hooks` 패키지의 `exhaustive-deps` 규칙을 사용하기를 권장합니다. 그것은 의존성이 바르지 않게 정의되었다면 그에 대해 경고하고 수정하도록 알려줍니다.

의존성 값의 배열은 **effect** 함수의 인자로 전달되지는 않습니다. 그렇지만 개념적으로는, 이 기법은 **effect** 함수가 무엇일지를 표현하는 방법입니다. **effect** 함수 안에서 참조되는 모든 값은 의존성 값의 배열에 드러나야 합니다. 나중에는 충분히 발전된 컴파일러가 이 배열을 자동적으로 생성할 수 있을 것입니다.

useContext {#usecontext}

```
const value = useContext(MyContext);
```

`context` 객체(`React.createContext`에서 반환된 값)을 받아 그 `context`의 현재 값을 반환합니다. `context`의 현재 값은 트리 안에서 이 Hook을 호출하는 컴포넌트에 가장 가까이에 있는 `<MyContext.Provider>`의 `value` prop에 의해 결정됩니다.

컴포넌트에서 가장 가까운 `<MyContext.Provider>`가 갱신되면 이 Hook은 그 `MyContext provider`에게 전달된 가장 최신의 `context value`를 사용하여 렌더러를 트리거 합니다. 상위 컴포넌트에서 `React.memo` 또는 `shouldComponentUpdate`를 사용하더라도 `useContext`를 사용하고 있는 컴포넌트 자체에서부터 다시 렌더링됩니다.

`useContext`로 전달한 인자는 `context` 객체 그 자체이어야 함을 잊지 마세요.

- 맞는 사용: `useContext(MyContext)`
- 틀린 사용: `useContext(MyContext.Consumer)`
- 틀린 사용: `useContext(MyContext.Provider)`

`useContext`를 호출한 컴포넌트는 `context` 값이 변경되면 항상 리렌더링 될 것입니다. 컴포넌트를 리렌더링 하는 것에 비용이 많이 든다면, [메모이제이션을 사용하여 최적화할 수 있습니다](#).

팁

여러분이 Hook 보다 context API에 친숙하다면 `useContext(MyContext)` 는 클래스에서의 `static contextType = MyContext` 또는 `<MyContext.Consumer>` 와 같다고 보면 됩니다.

`useContext(MyContext)` 는 context를 읽고 context의 변경을 구독하는 것만 가능합니다. context의 값을 설정하기 위해서는 여전히 트리의 윗 계층에서의 `<MyContext.Provider>` 가 필요합니다.

useContext를 Context.Provider와 같이 사용해주세요

```
const themes = {
  light: {
    foreground: "#000000",
    background: "#eeeeee"
  },
  dark: {
    foreground: "#ffffff",
    background: "#222222"
  }
};

const ThemeContext = React.createContext(themes.light);

function App() {
  return (
    <ThemeContext.Provider value={themes.dark}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

function ThemedButton() {
  const theme = useContext(ThemeContext);

  return (
    <button style={{ background: theme.background, color: theme.foreground }}>
      I am styled by theme context!
    </button>
  );
}
```

해당 예시는 [Context 고급 안내서](#)에서 사용했던 예시가 hook으로 수정되었으며 안내서에서 Context를 언제, 어떻게 사용하는지 자세히 알 수 있습니다.

추가 Hook

다음의 Hook은 이전 섹션에서의 기본 Hook의 변경이거나 특정한 경우에만 필요한 것입니다. 익히는 것에 너무 압박받지는 마세요.

useReducer {#usereducer}

```
const [state, dispatch] = useReducer(reducer, initialArg, init);
```

`useState`의 대체 함수입니다. `(state, action) => newState`의 형태로 reducer를 받고 `dispatch` 메서드와 짹의 형태로 현재 state를 반환합니다. (Redux에 익숙하다면 이것이 어떻게 동작하는지 여러분은 이미 알고 있을 것입니다.)

다수의 하위값을 포함하는 복잡한 정적 로직을 만드는 경우나 다음 state가 이전 state에 의존적인 경우에 보통 `useState` 보다 `useReducer`를 선호합니다. 또한 `useReducer`는 자세한 업데이트를 트리거 하는 컴포넌트의 성능을 최적화할 수 있게 하는데, 이것은 콜백 대신 `dispatch`를 전달 할 수 있기 때문입니다.

아래는 `useState` 내용에 있던 카운터 예시인데 reducer를 사용해서 다시 작성한 것입니다.

```
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:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

주의

React는 `dispatch` 함수의 동일성이 안정적이고 리렌더링 시에도 변경되지 않으리라는 것을 보장합니다. 이것이 `useEffect`나 `useCallback` 의존성 목록에 이 함수를 포함하지 않아도 괜찮은 이유입니다.

초기 state의 구체화

`useReducer` state의 초기화에는 두 가지 방법이 있습니다. 유스케이스에 따라서 한 가지를 선택하세요. 가장 간단한 방법은 초기 state를 두 번째 인자로 전달하는 것입니다.

```
const [state, dispatch] = useReducer(
  reducer,
  {count: initialCount}
);
```

주의

React에서는 Reducer의 인자로써 `state = initialState` 와 같은 초기값을 나타내는, Redux에서는 보편화된 관습을 사용하지 않습니다. 때때로 초기값은 `props`에 의존할 필요가 있어 Hook 호출에서 지정되기도 합니다. 초기값을 나타내는 것이 정말 필요하다면 `useReducer(reducer, undefined, reducer)` 를 호출하는 방법으로 Redux를 모방할 수는 있겠지만, 이 방법을 권장하지는 않습니다.

초기화 지연

초기 `state`를 조금 지연해서 생성할 수도 있습니다. 이를 위해서는 `init` 함수를 세 번째 인자로 전달합니다. 초기 `state`는 `init(initialArg)` 에 설정될 것입니다.

이것은 `reducer` 외부에서 초기 `state`를 계산하는 로직을 추출할 수 있도록 합니다. 또한, 어떤 행동에 대한 대응으로 나중에 `state`를 재설정하는 데에도 유용합니다.

```
function init(initialCount) {
  return {count: initialCount};
}

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    case 'reset':
      return init(action.payload);
    default:
      throw new Error();
  }
}

function Counter({initialCount}) {
  const [state, dispatch] = useReducer(reducer, initialCount, init);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'reset', payload: initialCount})}>
        Reset
      </button>
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

dispatch의 회피

Reducer Hook에서 현재 `state`와 같은 값을 반환하는 경우 React는 자식을 리렌더링하거나 effect를 발생하지 않고 이것들을 회피할 것입니다. (React는 `Object.is` 비교 알고리즘을 사용합니다.)

실행을 회피하기 전에 React에서 특정 컴포넌트를 다시 렌더링하는 것이 여전히 필요할 수도 있다는 것에 주의하세요. React가 불필요하게 트리에 그 이상으로 「더 깊게」 까지는 가지 않을 것이므로 크게 신경 쓰지 않으셔도 됩니다. 렌더링 시에 고비용의 계산을 하고 있다면 `useMemo`를 사용하여 그것들을 최적화할 수 있습니다.

useCallback {#usecallback}

```
const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b],
);
```

[메모이제이션된](#) 콜백을 반환합니다.

인라인 콜백과 그것의 의존성 값의 배열을 전달하세요. `useCallback`은 콜백의 메모이제이션된 버전을 반환할 것입니다. 그 메모이제이션된 버전은 콜백의 의존성이 변경되었을 때에만 변경됩니다. 이것은, 불필요한 렌더링을 방지하기 위해 (예로 `shouldComponentUpdate`를 사용하여) 참조의 동일성에 의존적인 최적화된 자식 컴포넌트에 콜백을 전달할 때 유용합니다.

`useCallback(fn, deps)`은 `useMemo(() => fn, deps)`와 같습니다.

주의

의존성 값의 배열이 콜백에 인자로 전달되지는 않습니다. 그렇지만 개념적으로는, 이 기법은 콜백 함수가 무엇일지를 표현하는 방법입니다. 콜백 안에서 참조되는 모든 값은 의존성 값의 배열에 나타나야 합니다. 나중에는 충분히 발전된 컴파일러가 이 배열을 자동적으로 생성할 수 있을 것입니다.

`eslint-plugin-react-hooks` 패키지의 일부로써 `exhaustive-deps` 규칙을 사용하기를 권장합니다. 그것은 의존성이 바르지 않게 정의되었다면 그에 대해 경고하고 수정하도록 알려줍니다.

useMemo {#usememo}

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

[메모이제이션된](#) 값을 반환합니다.

"생성(create)" 함수와 그것의 의존성 값의 배열을 전달하세요. `useMemo`는 의존성이 변경되었을 때에만 메모이제이션된 값만 다시 계산 할 것입니다. 이 최적화는 모든 렌더링 시의 고비용 계산을 방지하게 해 줍니다.

`useMemo`로 전달된 함수는 렌더링 중에 실행된다는 것을 기억하세요. 통상적으로 렌더링 중에는 하지 않는 것을 이 함수 내에서 하지 마세요. 예를 들어, 사이드 이펙트(side effects)는 `useEffect`에서 하는 일이지 `useMemo`에서 하는 일이 아닙니다.

배열이 없는 경우 매 렌더링 때마다 새 값을 계산하게 될 것입니다.

`useMemo`는 성능 최적화를 위해 사용할 수는 있지만 의미상으로 보장이 있다고 생각하지는 마세요. 가까운 미래에 React에서는, 이전 메모이제이션된 값들의 일부를 "잊어버리고" 다음 렌더링 시에 그것들을 재계산하는 방향을 택할지도 모르겠습니다. 예를 들면, 오프스크린 컴포넌트의 메모리를 해제하는 등이 있을 수 있습니다. `useMemo`를 사용하지 않고도 동작할 수 있도록 코드를 작성하고 그것을 추가하여 성능을 최적화하세요.

주의

의존성 값의 배열은 함수에 인자로 전달되지는 않습니다. 그렇지만 개념적으로는, 이 기법은 함수가 무엇일지를 표현하는 방법입니다. 함수 안에서 참조되는 모든 값은 의존성 값의 배열에 나타나야 합니다. 나중에는 충분히 발전된 컴파일러가 이 배열을 자동으로 생성할 수 있을 것입니다.

`eslint-plugin-react-hooks` 패키지의 일부로써 `exhaustive-deps` 규칙을 사용하기를 권장합니다. 그것은 의존성이 바르지 않게 정의되었다면 그에 대해 경고하고 수정하도록 알려줍니다.

useRef {#useref}

```
const refContainer = useRef(initialValue);
```

`useRef` 는 `.current` 프로퍼티로 전달된 인자(`initialValue`)로 초기화된 변경 가능한 `ref` 객체를 반환합니다. 반환된 객체는 컴포넌트의 전 생애주기를 통해 유지될 것입니다.

일반적인 유스케이스는 자식에게 명령적으로 접근하는 경우입니다.

```
function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  const onButtonClick = () => {
    // `current` points to the mounted text input element
    inputEl.current.focus();
  };
  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the input</button>
    </>
  );
}
```

본질적으로 `useRef` 는 `.current` 프로퍼티에 변경 가능한 값을 담고 있는 "상자"와 같습니다.

아마도 여러분은 [DOM에 접근](#)하는 방법으로 `refs`에 친숙할지도 모르겠습니다. `<div ref={myRef} />` 를 사용하여 React로 `ref` 객체를 전달한다면, React는 노드가 변경될 때마다 변경된 DOM 노드에 그것의 `.current` 프로퍼티를 설정할 것입니다.

그렇지만, `ref` 속성보다 `useRef()` 가 더 유용합니다. 이 기능은 클래스에서 인스턴스 필드를 사용하는 방법과 유사한 [어떤 가변값을 유지하는 데에 편리합니다](#).

이것은 `useRef()` 가 순수 자바스크립트 객체를 생성하기 때문입니다. `useRef()` 와 `{current: ...}` 객체 자체를 생성하는 것의 유일한 차이점이라면 `useRef` 는 매번 렌더링을 할 때 동일한 `ref` 객체를 제공한다는 것입니다.

`useRef` 는 내용이 변경될 때 그것을 알려주지는 않는다는 것을 유념하세요. `.current` 프로퍼티를 변형하는 것이 리렌더링을 발생시키지는 않습니다. React가 DOM 노드에 `ref`를 attach하거나 detach할 때 어떤 코드를 실행하고 싶다면 대신 [콜백 ref](#)를 사용하세요.

useImperativeHandle {#useimperativehandle}

```
useImperativeHandle(ref, createHandle, [deps])
```

`useImperativeHandle` 은 `ref` 를 사용할 때 부모 컴포넌트에 노출되는 인스턴스 값을 사용자화(customizes)합니다. 항상 그렇듯이, 대부분의 경우 `ref`를 사용한 명령형 코드는 피해야 합니다. `useImperativeHandle` 는 `forwardRef` 와 더불어 사용하세요.

```
function FancyInput(props, ref) {
  const inputRef = useRef();
  useImperativeHandle(ref, () => ({
    focus: () => {
      inputRef.current.focus();
    }
  }));
  return <input ref={inputRef} ... />;
}
FancyInput = forwardRef(FancyInput);
```

위의 예시에서 `<FancyInput ref={inputRef} />` 를 렌더링한 부모 컴포넌트는 `inputRef.current.focus()` 를 호출할 수 있습니다.

useLayoutEffect {#uselayouteffect}

이 함수의 시그니처는 `useEffect` 와 동일하긴 한데, 모든 DOM 변경 후에 동기적으로 발생합니다. 이것은 DOM에서 레이아웃을 읽고 동기적으로 리렌더링하는 경우에 사용하세요. `useLayoutEffect` 의 내부에 예정된 갱신은 브라우저가 화면을 그리기 이전 시점에 동기적으로 수행될 것입니다.

화면 갱신 차단의 방지가 가능할 때 표준 `useEffect` 를 먼저 사용하세요.

팁

클래스 컴포넌트에서 코드를 변환하는 경우에 `useLayoutEffect` 는 `componentDidMount` 나 `componentDidUpdate` 와 동일한 단계를 실행하게 된다는 것에 주의하기 바랍니다. 그렇기는 하지만, 먼저 `useEffect` 를 사용해 보고 문제가 있다면 그다음으로 `useLayoutEffect` 를 사용해 보기를 권합니다.

서버 렌더링을 사용하는 경우라면 자바스크립트가 모두 다운로드될 때까지는 `useLayoutEffect` 와 `useEffect` 어느 것도 실행되지 않는다는 것을 명심해야 합니다. 이것이 서버에서 렌더링 되는 컴포넌트에서 `useLayoutEffect` 가 사용되는 경우에 대해 React가 경고하는 이유입니다. 이를 수정하기 위해서는 (최초 렌더링 시에 필요하지 않다면) 로직을 `useEffect` 로 이동하거나 (`useLayoutEffect` 가 수행될 때까지 HTML이 깨져 보이는 경우는) 클라이언트 렌더링이 완료될 때까지 컴포넌트 노출을 지연하도록 하세요.

서버에서 렌더링된 HTML에서 레이아웃 effect가 필요한 컴포넌트를 배제하고 싶다면, `showChild && <Child />` 를 사용하여 조건적으로 렌더링 하고 `useEffect(() => { setShowChild(true); }, [])` 를 사용하여 노출을 지연시키세요. 이런 방법으로 자바스크립트 코드가 주입되기 전에 깨져 보일 수 있는 UI는 표현되지 않게 됩니다.

useDebugValue {#usedebugvalue}

```
useDebugValue(value)
```

`useDebugValue` 는 React 개발자도구에서 사용자 Hook 레이블을 표시하는 데에 사용할 수 있습니다.

예를 들어, "나만의 Hook 만들기"에 설명하고 있는 `useFriendStatus` 사용자 Hook에 대해서 생각해 봅시다.

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

  // ...

  // Show a label in DevTools next to this Hook
  // e.g. "FriendStatus: Online"
  useDebugValue(isOnline ? 'Online' : 'Offline');

  return isOnline;
}
```

팁

모든 사용자 Hook에 디버그 값을 추가하기를 권하지는 않습니다. 이것은 사용자 Hook이 공유된 라이브러리의 일부일 때 가장 유용합니다.

디버그 값 포맷팅 지연하기

경우에 따라 디스플레이 값을 포맷팅하는 것이 고비용의 연산일 수 있습니다. 또한, 사실상 Hook이 감지되지 않는다면 불필요하기도 합니다.

이런 이유로 `useDebugValue` 는 옵션 두 번째 파라미터로 포맷팅 함수를 전달할 수도 있습니다. 이 함수는 Hook가 감지되었을 때만 호출됩니다. 이것은 파라미터로써 디버그 값을 전달받아 포맷된 노출값을 반환해야 합니다.

예를 들어 사용자 Hook은 다음의 포맷 형식을 사용해서 `toDateString` 함수를 불필요하게 호출하는 것을 방지할 수 있습니다.

```
useDebugValue(date, date => date.toDateString());
```

useDeferredValue {#usedeferredvalue}

```
const deferredValue = useDeferredValue(value);
```

`useDeferredValue` accepts a value and returns a new copy of the value that will defer to more urgent updates. If the current render is the result of an urgent update, like user input, React will return the previous value and then render the new value after the urgent render has completed.

This hook is similar to user-space hooks which use debouncing or throttling to defer updates. The benefit to using `useDeferredValue` is that React will work on the update as soon as other work finishes (instead of waiting for an arbitrary amount of time), and like `startTransition`, deferred values can suspend without triggering an unexpected fallback for existing content.

Memoizing deferred children

`useDeferredValue` only defers the value that you pass to it. If you want to prevent a child component from re-rendering during an urgent update, you must also memoize that component with `React.memo` or `React.useMemo`:

```

function Typeahead() {
  const query = useSearchQuery('');
  const deferredQuery = useDeferredValue(query);

  // Memoizing tells React to only re-render when deferredQuery changes,
  // not when query changes.
  const suggestions = useMemo(() =>
    <SearchSuggestions query={deferredQuery} />,
    [deferredQuery]
  );

  return (
    <>
      <SearchInput query={query} />
      <Suspense fallback="Loading results...">
        {suggestions}
      </Suspense>
    </>
  );
}

```

Memoizing the children tells React that it only needs to re-render them when `deferredQuery` changes and not when `query` changes. This caveat is not unique to `useDeferredValue`, and it's the same pattern you would use with similar hooks that use debouncing or throttling.

useTransition {#usetransition}

```
const [isPending, startTransition] = useTransition();
```

Returns a stateful value for the pending state of the transition, and a function to start it.

`startTransition` lets you mark updates in the provided callback as transitions:

```

startTransition(() => {
  setCount(count + 1);
})

```

`isPending` indicates when a transition is active to show a pending state:

```

function App() {
  const [isPending, startTransition] = useTransition();
  const [count, setCount] = useState(0);

  function handleClick() {
    startTransition(() => {
      setCount(c => c + 1);
    })
  }

  return (
    <div>
      {isPending && <Spinner />}
      <button onClick={handleClick}>{count}</button>
    </div>
  );
}

```

Note:

Updates in a transition yield to more urgent updates such as clicks.

Updates in a transitions will not show a fallback for re-suspended content. This allows the user to continue interacting with the current content while rendering the update.

useId {#useid}

```
const id = useId();
```

`useId` is a hook for generating unique IDs that are stable across the server and client, while avoiding hydration mismatches.

Note

`useId` is **not** for generating `keys` in a list. Keys should be generated from your data.

For a basic example, pass the `id` directly to the elements that need it:

```
function Checkbox() {
  const id = useId();
  return (
    <>
      <label htmlFor={id}>Do you like React?</label>
      <input id={id} type="checkbox" name="react"/>
    </>
  );
};
```

For multiple IDs in the same component, append a suffix using the same `id`:

```
function NameFields() {
  const id = useId();
  return (
    <div>
      <label htmlFor={id + '-firstName'}>First Name</label>
      <div>
        <input id={id + '-firstName'} type="text" />
      </div>
      <label htmlFor={id + '-lastName'}>Last Name</label>
      <div>
        <input id={id + '-lastName'} type="text" />
      </div>
    </div>
  );
};
```

Note:

`useId` generates a string that includes the `:` token. This helps ensure that the token is unique, but is not supported in CSS selectors or APIs like `querySelectorAll`.

`useId` supports an `identifierPrefix` to prevent collisions in multi-root apps. To configure, see the options for `hydrateRoot` and `ReactDOMServer`.

Library Hooks

The following Hooks are provided for library authors to integrate libraries deeply into the React model, and are not typically used in application code.

`useSyncExternalStore` {#usesyncexternalstore}

```
const state = useSyncExternalStore(subscribe, getSnapshot[, getServerSnapshot]);
```

`useSyncExternalStore` is a hook recommended for reading and subscribing from external data sources in a way that's compatible with concurrent rendering features like selective hydration and time slicing.

This method returns the value of the store and accepts three arguments:

- `subscribe` : function to register a callback that is called whenever the store changes.
- `getSnapshot` : function that returns the current value of the store.
- `getServerSnapshot` : function that returns the snapshot used during server rendering.

The most basic example simply subscribes to the entire store:

```
const state = useSyncExternalStore(store.subscribe, store.getSnapshot);
```

However, you can also subscribe to a specific field:

```
const selectedField = useSyncExternalStore(
  store.subscribe,
  () => store.getSnapshot().selectedField,
);
```

When server rendering, you must serialize the store value used on the server, and provide it to `useSyncExternalStore`. React will use this snapshot during hydration to prevent server mismatches:

```
const selectedField = useSyncExternalStore(
  store.subscribe,
  () => store.getSnapshot().selectedField,
  () => INITIAL_SERVER_SNAPSHOT.selectedField,
);
```

Note:

`getSnapshot` must return a cached value. If `getSnapshot` is called multiple times in a row, it must return the same exact value unless there was a store update in between.

A shim is provided for supporting multiple React versions published as `use-sync-external-store/shim`. This shim will prefer `useSyncExternalStore` when available, and fallback to a user-space implementation when it's not.

As a convenience, we also provide a version of the API with automatic support for memoizing the result of `getSnapshot` published as `use-sync-external-store/with-selector`.

useInsertionEffect `{#useinsertioneffect}`

 `useInsertionEffect(didUpdate);`

The signature is identical to `useEffect`, but it fires synchronously *before* all DOM mutations. Use this to inject styles into the DOM before reading layout in `useLayoutEffect`. Since this hook is limited in scope, this hook does not have access to refs and cannot schedule updates.

Note:

`useInsertionEffect` should be limited to css-in-js library authors. Prefer `useEffect` or `useLayoutEffect` instead.

[Go to TOC](#)

Hook의 규칙

Hook은 React 16.8에 새로 추가된 기능입니다. Hook은 class를 작성하지 않고도 state와 다른 React의 기능들을 사용할 수 있도록 해줍니다.

Hook은 JavaScript 함수입니다. 하지만 Hook을 사용할 때는 두 가지 규칙을 준수해야 합니다. 우리는 이러한 규칙들을 자동으로 강제하기 위한 [linter 플러그인](#)을 제공하고 있습니다.

최상위(at the Top Level)에서만 Hook을 호출해야 합니다

반복문, 조건문 혹은 중첩된 함수 내에서 **Hook을 호출하지 마세요**. 대신 early return이 실행되기 전에 항상 React 함수의 최상위(at the top level)에서 Hook을 호출해야 합니다. 이 규칙을 따르면 컴포넌트가 렌더링 될 때마다 항상 동일한 순서로 Hook이 호출되는 것이 보장됩니다. 이러한 점은 React가 `useState` 와 `useEffect` 가 여러 번 호출되는 중에도 Hook의 상태를 올바르게 유지할 수 있도록 해줍니다. 이 점에 대해서 궁금하다면 [아래](#)에서 자세히 설명해 드리겠습니다.

오직 React 함수 내에서 Hook을 호출해야 합니다

Hook을 일반적인 JavaScript 함수에서 호출하지 마세요. 대신 아래와 같이 호출할 수 있습니다.

- React 함수 컴포넌트에서 Hook을 호출하세요.
- Custom Hook에서 Hook을 호출하세요. ([다음 페이지](#)에서 이 부분을 살펴볼 예정입니다)

이 규칙을 지키면 컴포넌트의 모든 상태 관련 로직을 소스코드에서 명확하게 보이도록 할 수 있습니다.

ESLint 플러그인

우리는 이 두 가지 규칙을 강제하는 `eslint-plugin-react-hooks`라는 ESLint 플러그인을 출시했습니다. 이 플러그인을 프로젝트에 추가할 수 있습니다.

이 플러그인은 [Create React App](#)에 기본적으로 포함되어 있습니다.

```
npm install eslint-plugin-react-hooks --save-dev
```

```
// ESLint 설정 파일
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
    // ...
    "react-hooks/rules-of-hooks": "error", // Checks rules of Hooks
    "react-hooks/exhaustive-deps": "warn" // Checks effect dependencies
  }
}
```

어떻게 [나만의 Hook](#)를 작성할 수 있는지 설명하는 다음 장으로 지금 넘어가도 좋습니다. 이번 장에서는 계속해서 이러한 규칙들의 논리적 근거에 대해 설명할 예정입니다.

설명

이전에 배웠듯이 한 컴포넌트에서 State나 Effect Hook을 여러 개 사용할 수도 있습니다.

```
function Form() {
  // 1. name이라는 state 변수를 사용하세요.
  const [name, setName] = useState('Mary');

  // 2. Effect를 사용해 폼 데이터를 저장하세요.
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });

  // 3. surname이라는 state 변수를 사용하세요.
  const [surname, setSurname] = useState('Poppins');

  // 4. Effect를 사용해서 제목을 업데이트합니다.
  useEffect(function updateTitle() {
    document.title = name + ' ' + surname;
  });

  // ...
}
```

그렇다면 React는 어떻게 특정 state가 어떤 useState 호출에 해당하는지 알 수 있을까요? 정답은 **React가 Hook이 호출되는 순서에 의존한다는 것입니다.** 모든 렌더링에서 Hook의 호출 순서는 같기 때문에 예시가 올바르게 동작할 수 있습니다.

```
// -----
// 첫 번째 렌더링
// -----
useState('Mary')           // 1. 'Mary'라는 name state 변수를 선언합니다.
useEffect(persistForm)     // 2. 폼 데이터를 저장하기 위한 effect를 추가합니다.
useState('Poppins')        // 3. 'Poppins'라는 surname state 변수를 선언합니다.
useEffect(updateTitle)     // 4. 제목을 업데이트하기 위한 effect를 추가합니다.

// -----
// 두 번째 렌더링
// -----
useState('Mary')           // 1. name state 변수를 읽습니다.(인자는 무시됩니다)
useEffect(persistForm)     // 2. 폼 데이터를 저장하기 위한 effect가 대체됩니다.
useState('Poppins')        // 3. surname state 변수를 읽습니다.(인자는 무시됩니다)
useEffect(updateTitle)     // 4. 제목을 업데이트하기 위한 effect가 대체됩니다.

// ...
```

Hook의 호출 순서가 렌더링 간에 동일하다면 React는 지역적인 state를 각 Hook에 연동시킬 수 있습니다. 하지만 Hook을 조건문 안에서(예를 들어 persistForm effect) 호출한다면 어떤 일이 일어날까요?

```
// ● 조건문에 Hook을 사용함으로써 첫 번째 규칙을 깰습니다
if (name !== '') {
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });
}
```

name !== '' 조건은 첫 번째 렌더링에서 true 기 때문에 Hook은 동작합니다. 하지만 사용자가 그다음 렌더링에서 폼을 초기화하면서 조건을 false로 만들 겁니다. 렌더링 간에 Hook을 건너뛰기 때문에 Hook 호출 순서는 달라지게 됩니다.

```
useState('Mary')           // 1. name state 변수를 읽습니다. (인자는 무시됩니다)
// useEffect(persistForm) // ❶ Hook을 건너뛰었습니다!
useState('Poppins')        // ❷ 2 (3이었던). surname state 변수를 읽는 데 실패했습니다.
useEffect(updateTitle)     // ❸ 3 (4였던). 제목을 업데이트하기 위한 effect가 대체되는 데 실패
했습니다.
```

React는 두 번째 `useState` Hook 호출에 대해 무엇을 반환할지 몰랐습니다. React는 이전 렌더링 때처럼 컴포넌트 내에서 두 번째 Hook 호출이 `persistForm` effect와 일치할 것이라 예상했지만 그렇지 않았습니다. 그 시점부터 건너뛴 Hook 다음에 호출되는 Hook이 순서가 하나씩 밀리면서 버그를 발생시키게 됩니다.

이것이 컴포넌트 최상위(**the top of level**)에서 Hook이 호출되어야만 하는 이유입니다. 조건부로 effect를 실행하기를 원한다면, 조건문을 Hook 내부에 넣을 수 있습니다.

```
useEffect(function persistForm() {
  // 👍 더 이상 첫 번째 규칙을 어기지 않습니다
  if (name !== '') {
    localStorage.setItem('formData', name);
  }
});
```

제공된 [lint 규칙](#)을 활용한다면 이 문제에 대해 걱정할 필요는 없습니다. 그러나 이제 왜 Hook이 이런 식으로 동작하는지 그리고 이 규칙이 어떤 문제를 방지하는지 알고 있습니다.

다음 단계

마침내 [Custom Hook](#)을 작성하는 법을 배울 준비가 되었습니다! Custom Hook은 React에서 제공하는 Hook을 추상화된 로직으로 사용할 수 있도록 결합해주고 다른 컴포넌트 사이에서 공통의 상태 관련 로직을 재사용 할 수 있도록 해줍니다.

Using the State Hook

Hook은 React 16.8버전에 새로 추가되었습니다. Hook은 클래스 컴포넌트를 작성하지 않아도 state와 같은 특징들을 사용할 수 있습니다.

[Hook 소개](#)에서 아래 예시를 통해 Hook과 친해졌습니다.

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

function Example() {
  // 새로운 state 변수를 선언하고, count라 부르겠습니다.
  const [count, setCount] = useState(0);

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

아래의 클래스 예시와 비교하며 Hook의 특징에 대해 배울 예정입니다.

Hook과 같은 기능을 하는 클래스 예시

React에서 클래스를 사용해봤다면, 아래의 코드는 익숙할 겁니다.

```
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>
    );
  }
}
```

위 코드에서 state는 `{ count: 0 }`이며 사용자가 `this.setState()`를 호출하는 버튼을 클릭했을 때 `state.count`를 증가시킵니다. 위의 클래스 예시를 해당 페이지에서 계속 사용할 예정입니다.

주의

좀 더 현실적인 예시가 아닌, counter 예시를 사용하는지 궁금할 수 있습니다. counter 예시를 사용한 이유는, Hook을 잘 이해할 수 있도록 도와주는 가장 기초적인 내용이 될 수 있기 때문입니다.

Hook과 함수 컴포넌트

React의 함수 컴포넌트는 이렇게 생겼습니다.

```
const Example = (props) => {
  // 여기서 Hook을 사용할 수 있습니다!
  return <div />;
}
```

또는 이렇게 생겼습니다.

```
function Example(props) {
  // 여기서 Hook을 사용할 수 있습니다!
  return <div />;
}
```

함수 컴포넌트를 "state가 없는 컴포넌트"로 알고 있었을 겁니다. 하지만 Hook은 React state를 함수 안에서 사용할 수 있게 해줍니다.

Hook은 클래스 안에서 **동작하지 않습니다**. 하지만 클래스를 작성하지 않고 사용할 수 있습니다.

Hook이란?

React의 `useState` Hook을 사용해봅시다!

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

function Example() {
  // ...
}
```

Hook이란? Hook은 특별한 함수입니다. 예를 들어 `useState` 는 state를 함수 컴포넌트 안에서 사용할 수 있게 해줍니다. 다른 Hook들은 나중에 살펴봅시다!

언제 Hook을 사용할까? 함수 컴포넌트를 사용하던 중 state를 추가하고 싶을 때 클래스 컴포넌트로 바꾸곤 했을 겁니다. 하지만 이제 함수 컴포넌트 안에서 Hook을 이용하여 state를 사용할 수 있습니다.

주의

컴포넌트 안에서 Hook을 사용할 때 몇 가지 특별한 규칙이 있습니다. 나중에 [Hook의 규칙](#)에서 살펴보도록 할게요!

state 변수 선언하기

클래스를 사용할 때, constructor 안에서 `this.state` 를 `{ count: 0 }` 로 설정함으로써 `count` 를 `0` 으로 초기화했습니다.

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

함수 컴포넌트는 `this` 를 가질 수 없기 때문에 `this.state` 를 할당하거나 읽을 수 없습니다. 대신, `useState` Hook을 직접 컴포넌트에 호출합니다.

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

function Example() {
  // 새로운 state 변수를 선언하고, 이것을 count라 부르겠습니다.
  const [count, setCount] = useState(0);
```

`useState` 를 호출하는 것은 무엇을 하는 걸까요? "state 변수"를 선언할 수 있습니다. 위에 선언한 변수는 `count` 라고 부르지만 `banana` 처럼 아무 이름으로 지어도 됩니다. `useState` 는 클래스 컴포넌트의 `this.state` 가 제공하는 기능과 똑같습니다. 일반적으로 일반 변수는 함수가 끝날 때 사라지지만, state 변수는 React에 의해 사라지지 않습니다.

`useState` 의 인자로 무엇을 넘겨주어야 할까요? `useState()` Hook의 인자로 넘겨주는 값은 state의 초기 값입니다. 함수 컴포넌트의 `state` 는 클래스와 달리 객체일 필요는 없고, 숫자 타입과 문자 타입을 가질 수 있습니다. 위의 예시는 사용자가 버튼을 얼마나 많이 클릭했는지 알기를 원하므로 `0` 을 해당 state의 초기 값으로 선언했습니다. (2개의 다른 변수를 저장하기를 원한다면 `useState()` 를 두 번 호출해야 합니다.)

`useState` 는 무엇을 반환할까요? state 변수, 해당 변수를 갱신할 수 있는 함수 이 두 가지 쌍을 반환합니다. 이것이 바로 `const [count, setCount] = useState()` 라고 쓰는 이유입니다. 클래스 컴포넌트의 `this.state.count` 와 `this.setState` 와 유사합니다. 이러한 문법에 익숙하지 않다면 [현재 페이지의 끝](#)에서 살펴볼게요.

이제 `useState` 를 이용하여 많은 것을 만들 수 있습니다.

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

function Example() {
  // 새로운 state 변수를 선언하고, 이것을 count라 부르겠습니다.
  const [count, setCount] = useState(0);
```

`count` 라고 부르는 state 변수를 선언하고 `0` 으로 초기화합니다. React는 해당 변수를 리렌더링할 때 기억하고, 가장 최근에 갱신된 값을 제공합니다. `count` 변수의 값을 갱신하려면 `setCount` 를 호출하면 됩니다.

주의

왜 `createState` 가 아닌, `useState` 로 이름을 지었을까요?

컴포넌트가 렌더링할 때 오직 한 번만 생성되기 때문에 "Create"라는 이름은 꽤 정확하지 않을 수 있습니다. 컴포넌트가 다음 렌더링을 하는 동안 `useState`는 현재 `state`를 줍니다. Hook 이름이 항상 `use`로 시작하는 이유도 있습니다. [Hook의 규칙](#)에서 나중에 살펴보도록 할게요.

state 가져오기

클래스 컴포넌트는 `count`를 보여주기 위해 `this.state.count`를 사용합니다.

```
<p>You clicked {this.state.count} times</p>
```

반면 함수 컴포넌트는 `count`를 직접 사용할 수 있습니다.

```
<p>You clicked {count} times</p>
```

state 갱신하기

클래스 컴포넌트는 `count`를 갱신하기 위해 `this.setState()`를 호출합니다.

```
<button onClick={() => this.setState({ count: this.state.count + 1 })}>
  Click me
</button>
```

반면 함수 컴포넌트는 `setCount`와 `count` 변수를 가지고 있으므로 `this`를 호출하지 않아도 됩니다.

```
<button onClick={() => setCount(count + 1)}>
  Click me
</button>
```

요약

아래 코드를 한 줄 한 줄 살펴보고, 얼마나 이해했는지 체크해봅시다.

```
1: import React, { useState } from 'react';
2:
3: function Example() {
4:   const [count, setCount] = useState(0);
5:
6:   return (
7:     <div>
8:       <p>You clicked {count} times</p>
9:       <button onClick={() => setCount(count + 1)}>
10:         Click me
11:       </button>
12:     </div>
13:   );
14: }
```

- 첫 번째 줄: `useState` Hook을 React에서 가져옵니다.
- 네 번째 줄: `useState` Hook을 이용하면 `state` 변수와 해당 `state`를 갱신할 수 있는 함수가 만들어집니다. 또한, `useState`의 인자의 값으로 `0`을 넘겨주면 `count` 값을 `0`으로 초기화할 수 있습니다.

- **아홉 번째 줄:** 사용자가 버튼 클릭을 하면 `setCount` 함수를 호출하여 state 변수를 갱신합니다. React는 새로운 `count` 변수를 `Example` 컴포넌트에 넘기며 해당 컴포넌트를 리렌더링합니다.

많은 것들이 있기 때문에 처음에는 다소 어려울 수 있습니다. 설명이 이해가 잘 안 된다면, 위의 코드를 천천히 다시 읽어보세요. 클래스 컴포넌트에서 사용하던 state 동작 방식을 잊고, 새로운 눈으로 위의 코드를 보면 분명히 이해가 갈 것입니다.

팁: 대괄호가 의미하는 것은 무엇일까요?

대괄호를 이용하여 state 변수를 선언하는 것을 보셨을 겁니다.

```
const [count, setCount] = useState(0);
```

대괄호 왼쪽의 state 변수는 사용하고 싶은 이름으로 선언할 수 있습니다.

```
const [fruit, setFruit] = useState('banana');
```

위 자바스크립트 문법은 "배열 구조 분해"라고 하고, `fruit` 과 `setFruit`, 총 2개의 값을 만들고 있습니다. 즉, `useState` 를 사용하면 `fruit` 이라는 첫 번째 값과 `setFruit` 라는 두 번째 값을 반환합니다. 아래의 코드와 같은 효과를 낼 수 있습니다.

```
var fruitStateVariable = useState('banana'); // 두 개의 아이템이 있는 쌍을 반환
var fruit = fruitStateVariable[0]; // 첫 번째 아이템
var setFruit = fruitStateVariable[1]; // 두 번째 아이템
```

`useState` 를 이용하여 변수를 선언하면 2개의 아이템 쌍이 들어있는 배열로 만들어집니다. 첫 번째 아이템은 현재 변수를 의미하고, 두 번째 아이템은 해당 변수를 갱신해주는 함수입니다. 배열 구조 분해라는 특별한 방법으로 변수를 선언해주었기 때문에 `[0]` 이나 `[1]` 로 배열에 접근하는 것은 좋지 않을 수 있습니다.

주의

`this` 를 React에 알리지 않았는데, 어떻게 React가 특정 컴포넌트에서 `useState` 를 사용한 것을 아는지 궁금해할 수 있습니다. [이 질문](#)과 다른 궁금 사항들은 나중에 살펴보겠습니다.

팁: 여러 개의 state 변수를 사용하기

`[something, setSomething]` 의 쌍처럼 state 변수를 선언하는 것은 유용합니다. 왜냐하면 여러 개의 변수를 선언할 때 각각 다른 이름을 줄 수 있기 때문입니다.

```
function ExampleWithManyStates() {
  // 여러 개의 state를 선언할 수 있습니다!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
}
```

위의 코드는 `age`, `fruit`, `todos` 라는 지역 변수를 가지며 개별적으로 갱신할 수 있습니다.

```
function handleOrangeClick() {
  // this.setState({ fruit: 'orange' })와 같은 효과를냅니다.
  setFruit('orange');
}
```

여러 개의 state 변수를 사용하지 않아도 됩니다. state 변수는 객체와 배열을 잘 가지고 있을 수 있으므로 서로 연관있는 데이터를 묶을 수 있습니다. 하지만 클래스 컴포넌트의 `this.setState` 와 달리 state를 갱신하는 것은 병합하는 것이 아니라 대체하는 것입니다.

독립적인 state 변수 분할에 대한 추가적인 권장 사항을 자주 묻는 질문에서 볼 수 있습니다. [자주 묻는 질문](#).

다음 단계

이번 페이지에서 React의 Hook 중 하나인 `useState` 에 대해 배웠습니다. `useState` 를 이용하면 함수 컴포넌트 안에서 state를 사용할 수 있습니다.

Hook은 함수 컴포넌트에서 React의 특징을 갖게 해주는 함수입니다. Hook은 항상 `use` 라는 키워드로 시작하며 `useState` 이외에 아직 보지 못한 많은 Hook들이 있습니다.

다음 강좌를 이어서 학습해보세요. [다음 Hook 강좌: useEffect](#). 다음에 배울 Hook은 클래스 컴포넌트의 생명주기와 비슷한 퍼포먼스를 낼 수 있습니다.

기여하는 방법

React는 페이스북 최초 오픈소스 프로젝트 중 하나로 매우 활발히 개발하고 있으며 facebook.com의 모든 사용자에게 전달되고 있습니다. 이 프로젝트에 최대한 쉽고 간편하게 기여할 수 있도록 노력하고 있지만, 아직 부족한 상태입니다. 이 문서를 통해 여러분이 프로젝트에 기여하는 절차에 대해 명확하게 인지하고 궁금한 점들을 해결할 수 있기를 바랍니다.

행동 강령 {#code-of-conduct}

Facebook은 [Contributor Covenant](#)를 행동 강령으로 채택했으며 모든 프로젝트 참여자가 준수하기를 기대합니다. [전문](#)을 읽고 어떤 행동이 허용되고 허용되지 않는지 이해할 수 있습니다.

오픈소스

React에 대한 모든 개발 작업은 [GitHub](#)에서 이루어집니다. 코어 팀 구성원과 외부 기여자가 모두 동일한 검토 과정을 통해 풀 리퀘스트를 제출합니다.

유의적 버전

React는 [유의적 버전](#)을 따릅니다. 중요한 버그 수정은 수 버전으로, 핵심적이지 않은 변화나 새로운 기능은 부 버전으로 그리고 호환성이 유지되지 않는 변경은 주 버전으로 배포합니다. 호환성이 유지되지 않는 변경을 만들 때, 부 버전에서 사용을 권장하지 않는 주의 메세지를 통해 React를 사용하는 개발자가 다가올 변화를 알아차리고 미리 코드를 변경할 수 있게 합니다. 안정성과 점진적 마이그레이션과 관련된 약속에 대해 [버전 정책](#)에서 자세히 알 수 있습니다.

모든 중요한 변화는 [changelog 파일](#)에 기록되어 있습니다.

브랜치 구성

모든 변화는 `main` 브랜치로 제출해주세요. 개발이나 다가오는 배포를 위해 따로 브랜치를 관리하지는 않습니다.

`main`에 반영된 코드는 가장 최근의 안정된 배포와 반드시 호환돼야 합니다. 추가적인 기능을 포함할 수 있지만, 호환되지 않는 변화는 포함되면 안 됩니다. 언제든 `main`의 가장 최근 커밋으로부터 새로운 부 버전을 배포할 수 있어야 합니다.

Feature Flags

`main` 브랜치를 배포 가능한 상태로 유지하기 위해, 호환되지 않는 변화나 실험적인 기능은 feature flag를 통해 활성화 유무를 관리할 수 있어야 합니다.

Feature flag는 `packages/shared/ReactFeatureFlags.js`에 정의되어 있습니다. React의 몇 가지 빌드는 서로 다른 feature flag 조합을 가질 수 있습니다. 예를 들어, React Native 빌드는 React DOM과 다르게 구성됩니다. 이러한 flag는 `packages/shared/forks`에서 찾을 수 있습니다. Feature flag는 Flow에 의해 정적으로 타입 추론되며, `yarn flow`를 통해 필요한 모든 파일을 변경했는지 확인할 수 있습니다.

React의 빌드 시스템은 배포하기 전에 비활성화된 기능을 제거합니다. CI의 job에서 모든 커밋에 대해 번들 크기 변화를 점검합니다. 크기 변화를 원하는 기능의 활성화 유무가 올바르게 되었는지 확인하는 신호로써 사용할 수 있습니다.

버그

알려진 이슈는 어디서 찾아야 할까요?

공개 버그 관리에 [GitHub Issues](#)를 사용하고 있습니다. 이슈에 주의를 기울이고 내부 수정이 진행 중이라면 이슈를 해결하려고 노력합니다. 새로운 이슈를 등록하기 전에, 이미 등록된 이슈가 아닌지 확인해주세요.

새로운 이슈 보고

버그가 발생하는 작은 테스트 케이스를 제공하는 게 버그를 수정하기 위한 가장 좋은 방법입니다. 이때 [JSFiddle 템플릿](#)이 좋은 시작점입니다.

보안 관련 버그

Facebook은 보안 버그를 안전하게 공개하기 위한 [포상금 제도](#)가 존재합니다. 이러한 점을 고려해서 보안 관련 버그는 공개 이슈에 문제를 제기하지 말고 이 페이지에 설명된 순서에 따라 진행해 주세요.

연락 방법

- IRC: [#reactjs on freenode](#)
- [Discussion forums](#)

또한, React에 대한 도움이 필요한 경우 [Discord 채팅 플랫폼](#)에서 React 사용자 커뮤니티를 이용할 수 있습니다.

변경 제안

Public API를 변경하거나 구현을 간단하게 변경하려 할 때 [이슈를 먼저 제출](#)하길 권장합니다. 수정하려고 많은 노력을 기울이기 전에 제안에 대한 합의에 도달할 수 있도록 해줍니다.

버그만 수정할 경우, 곧바로 풀 리퀘스트를 제출해도 괜찮지만, 여전히 수정하려는 사항을 자세히 설명하는 이슈를 제출하는 것이 좋습니다. 받아들여지지 않은 특정 변화가 있지만, 이슈를 추적하기 원할 때 도움이 됩니다.

첫 번째 풀 리퀘스트

풀 리퀘스트를 처음 해보시나요? 무료 영상 시리즈를 통해 다음과 같이 기여하는 방법을 배울 수도 있습니다.

[GitHub에서 오픈 소스 프로젝트에 기여하는 방법](#)

처음 발을 디딜 때 기여 과정에 익숙해질 수 있도록 비교적 영향력이 적은 버그를 포함하는 [good first issues](#) 목록은 기여 입문에 최적입니다.

이슈를 해결하려는 경우, 다른 누군가 이미 수정 작업을 진행 중일 수도 있으므로 코멘트 쓰레드를 꼭 확인해 주세요. 아무도 작업하고 있지 않은 경우 다른 사람이 실수로 중복 작업을 하지 않도록 작업할 예정이라고 코멘트를 남겨주세요.

누군가 하겠다고 선언한 이슈가 2주 이상 방치된 경우 다른 사람이 넘겨받는 것은 상관없지만 그러한 경우에도 코멘트를 남겨야 합니다.

풀 리퀘스트 보내기

코어 팀은 풀 리퀘스트를 모니터링하고 있습니다. 여러분이 요청한 풀 리퀘스트를 리뷰, 병합, 변경 요청을 하거나 설명과 함께 풀 리퀘스트를 닫을 것입니다. Facebook.com 내부에서 사용법 검토가 필요한 API 변경은 시간이 더 걸릴 수 있습니다. 코어 팀은 프로세스 전반에 걸쳐 최신 정보 업데이트와 피드백을 제공하도록 최선을 다할 것입니다.

풀 리퀘스트를 보내기 전에, 다음 사항을 확인해 주세요.

1. 저장소를 포크하고 `main`로부터 새로운 브랜치를 생성합니다.
2. 저장소 루트에서 `yarn` 명령을 실행합니다.
3. 버그를 수정했거나 테스트가 필요한 코드를 추가했다면 테스트를 추가해 주세요.
4. 테스트가 통과하는지 확인해 주세요(`yarn test`). Tip : `yarn test --watch TestName` 명령은 개발할 때 도움이 됩니다.
5. `yarn test --prod` 명령을 실제 환경에서 테스트하기 위해 실행합니다.
6. 디버거가 필요한 경우 `yarn debug-test --watch TestName` 을 실행하고 `chrome://inspect` 을 열어 "Inspect"를 누르세요.
7. `prettier`로 코드를 포맷하세요. (`yarn prettier`).
8. 코드를 린트하세요 (`yarn lint`). Tip: `yarn linc` 는 변경된 파일만 확인할 수 있습니다.
9. `Flow` 타입 검사를 실행하세요. (`yarn flow`).
10. 아직 CLA에 서명하지 않았다면, 서명을 완료해주세요.

Contributor License Agreement (CLA)

제출해주신 풀 리퀘스트를 받아들이기 위해서는 CLA를 완료해주셔야 합니다. 한 번만 하면 되기 때문에 다른 Facebook 오픈 소스 프로젝트에서 이미 완료했다면 이 과정은 필요하지 않습니다. 풀 리퀘스트를 처음 제출한다면 CLA를 완료했다고 알려주세요. 그러면 우리는 GitHub의 사용자 이름을 확인하여 이에 대한 점검을 시행합니다.

[여기서 CLA를 완료하세요.](#)

기여 수행 조건

- Node LTS와 Yarn v1.2.0+가 설치되어 있어야 합니다.
- JDK가 설치되어 있어야 합니다.
- gcc 가 설치되어 있어야 하는 데 필요하다면 컴파일러를 설치하는 게 편합니다. 일부 의존성은 컴파일 과정이 필요할 수 있습니다. OS X에서는 Xcode 커맨드 라인 도구가 도움이 됩니다. Ubuntu에서는 `apt-get install build-essential` 명령으로 필요한 패키지를 설치할 수 있습니다. 다른 Linux 배포판에서도 비슷한 명령으로 작업할 수 있습니다. Windows에서는 몇 가지 추가 단계가 필요할 수 있으므로 자세한 내용은 `node-gyp installation instructions`를 참조해 주세요.
- Git 사용에 익숙해야 합니다.

개발 워크플로우

React 저장소를 복사한 후 `yarn` 명령을 사용하면 다음과 같은 여러 명령을 실행할 수 있습니다.

- `yarn lint` 코드 스타일을 점검합니다.
- `yarn linc` 는 `yarn lint` 와 비슷하지만, 브랜치에서 변경된 파일만 검사하기 때문에 더 빠릅니다.
- `yarn test` 는 모든 테스트를 실행합니다.
- `yarn test --watch` 대화형 테스트 watcher를 실행합니다.
- `yarn test <pattern>` 은 일치하는 파일 이름으로 테스트를 실행합니다.
- `yarn test --prod` 는 실제 환경에서 테스트를 실행합니다.
- `yarn debug-test` 는 `yarn test` 와 비슷하지만, 디버거를 활용합니다. `chrome://inspect` 를 열고 "Inspect"를 누르세요.
- `yarn flow` 는 Flow 타입검사를 실행합니다.
- `yarn build` 는 모든 패키지와 함께 `build` 폴더를 만듭니다.

- `yarn build react/index,react-dom/index --type=UMD` 는 React와 ReactDOM의 UMD 빌드를 만듭니다.

변경 작업이 이상이 없는지 확인하기 위해 `yarn test` (또는 이것과 비슷한 명령)을 통해 확인해주세요. 직접 수정한 React 빌드를 사용해보는 게 도움이 될 수 있습니다.

우선, `yarn build`를 실행합니다. 이렇게 하면 `build` 폴더에 미리 빌드된 번들 파일이 만들어지고 동시에 `build/packages` 안에 npm 패키지도 준비가 됩니다.

`yarn build react/index,react-dom/index --type=UMD` 를 실행하고 `fixtures/packaging/babel-standalone/dev.html` 을 실행하는 게 변경을 시도해보는 가장 쉬운 방법입니다. 이 파일은 `build` 폴더의 `react.development.js` 를 이미 사용하고 있으므로 변경 사항을 확인할 수 있습니다.

기존 React 프로젝트에서 변경한 내용을 확인하고자 한다면 `build/node_modules/react/umd/react.development.js`, `build/node_modules/react-dom/umd/react-dom.development.js` 또는 다른 빌드 파일을 애플리케이션에 복사하여 안정된 버전 대신 사용할 수 있습니다.

npm을 통해 React를 사용하고 있다면, 의존성에서 `react` 와 `react-dom` 을 삭제하고 `yarn link` 를 사용해서 로컬 `build` 폴더를 가리키게 해주세요. **빌드할 때 --type=UMD 대신 --type=NODE 을 전달해야 한다는 점을 주의해주세요.** 또한 `scheduler` 패키지도 아래처럼 빌드해야 합니다.

```
cd ~/path_to_your_react_clone/
yarn build react/index,react/jsx,react-dom/index,scheduler --type=NODE

cd build/node_modules/react
yarn link
cd build/node_modules/react-dom
yarn link

cd ~/path/to/your/project
yarn link react react-dom
```

React 폴더에서 `yarn build` 를 실행할 때마다 업데이트된 버전이 프로젝트 `node_modules` 에 나타납니다. 그런 다음 프로젝트를 재빌드하여 변경할 수 있습니다.

몇 가지 패키지는 여전히 빠져있을 수 있는데 (예를 들어, 프로젝트에서 `react-dom/server` 를 사용하는 경우), 항상 `yarn build` 를 통해 전체를 빌드 할 수 있습니다. 옵션 없이 `yarn build` 를 실행한다면 많은 시간이 필요합니다.

다만, 풀 리퀘스트에 새로운 기능에 대한 단위 테스트가 포함되어야 합니다. 포함되어야 새로 작성해준 코드를 코어 팀이 미래에 고장 내지 않을 수 있습니다.

스타일 가이드

Prettier라고 불리는 자동 코드 포맷터를 사용합니다. 코드를 변경한 뒤 `yarn prettier` 를 실행해주세요.

그러면 linter는 코드에 존재할 수 있는 문제를 잡아냅니다. 단순히 변경한 코드의 스타일을 점검하고 싶을 땐 `yarn linc` 를 사용해주세요.

그러나, 아직 linter에서도 점검할 수 없는 스타일이 존재합니다. 모르는 것이 있다면 [Airbnb's Style Guide](#)에서 적절한 방법을 안내받을 수 있습니다.

Request for Comments (RFC)

버그 수정이나 문서 개선을 포함한 많은 변경 사항은 일반적인 GitHub 풀 리퀘스트의 워크플로우를 통해 구현될 수 있고 검토될 수 있습니다.

다만, 일부 변화가 "크고 본질적"이라면, 약간의 설계 프로세스를 거쳐서 React 코어 팀에서 합의를 하도록 요청합니다.

"RFC" (request for comments) 프로세스는 새로운 기능이 프로젝트에 들어가기까지 일관되고 정비된 경로를 제공하는 것을 목적으로 하고 있습니다. [rfcs 저장소](#)를 방문해서 기여할 수 있습니다.

라이선스

React에 기여할 때, 여러분은 그 기여가 MIT 라이선스에 따라 라이선스가 부여되는 것에 동의했다고 간주합니다.

다음은 무엇인가요?

[다음 섹션](#)을 읽고 코드 구조가 어떻게 구성되어 있는지 확인해보세요.

[Go to TOC](#)

구현 참고사항

이 부분은 [스택 재조정자\(reconciler\)](#)에 대한 구현 참고사항입니다.

이는 매우 기술적이고 공개된 React API뿐만 아니라 어떻게 코어, 렌더러, 재조정자로 나누어지는지에 대해 깊은 이해가 필요합니다. 아직 React 코드 베이스에 친숙하지 않다면, 먼저 [the codebase overview](#)를 읽기를 바랍니다.

이는 [React 컴포넌트와 인스턴스 그리고 엘리먼트 사이의 차이점](#)을 이해한다고 가정합니다.

스택 재조정자는 React 15와 그 이전 버전에 사용되었습니다. 이는 [src/renderers/shared/stack/reconciler](#)에 위치해 있습니다.

비디오: React 처음부터 만들기

[Paul O'Shannessy](#)는 이 문서에 크게 영감을 주었던 [building React from scratch](#)에 대해 이야기 하였습니다.

이 문서와 그의 말은 모두 현실 코드베이스의 단순화했기 때문에 여러분은 두 가지 모두 친숙해 짐으로써 더 깊은 이해를 가질 것입니다.

개요

재조정자는 공개된 API를 가지지 않습니다. React DOM과 React Native와 같은 [렌더러](#)는 사용자가 쓴, React 컴포넌트에 따른 사용자 인터페이스를 효율적으로 업데이트를 하기 위해서 사용합니다.

재귀적인 과정으로써의 마운트

여러분들이 컴포넌트를 처음 마운트할 때를 고려해 보겠습니다.

```
const root = ReactDOM.createRoot(rootEl);
root.render(<App />);
```

`root.render` 는 재조정자를 통해 `<App />` 를 통과하게 할 것입니다. `<App />` 은 React 엘리먼트이며, 렌더링 할 것을 설명해놓은 것임을 기억합시다. 이것을 평범한 객체로 생각해도 좋습니다.

```
console.log(<App />);
// { type: App, props: {} }
```

재조정자가 `App` 이 `class`인지 함수인지 확인합니다.

`App` 이 함수라면, 재조정자는 렌더링 엘리먼트를 가져오기 위해 `App(props)` 를 호출합니다.

`App` 이 `class`면, 재조정자는 `App` 을 `new App(props)` 로 인스턴스화 하고, `componentWillMount()` 생명주기 메서드를 호출한 후, `render()` 메서드를 호출하여 랜더링 엘리먼트를 가져오게 할 것입니다.

어느 경우든, 재조정자는 `App` 이 렌더링 되는 엘리먼트를 학습하게 됩니다.

이러한 과정은 재귀적입니다. `App` 은 `<Greeting />` 으로 렌더링 될 수도 있고, `Greeting` 은 `<Button />` 또는 다른 곳으로 렌더링 될 수 있습니다. 재조정자는 각 컴포넌트가 무엇을 렌더링하는지 학습할 때 사용자가 정의한 컴포넌트를 재귀적으로 조사합니다.

여러분들은 의사코드로 작성된 이 과정을 생각해봅시다.

```

function isClass(type) {
  // React.Component subclasses have this flag
  return (
    Boolean(type.prototype) &&
    Boolean(type.prototype.isReactComponent)
  );
}

// This function takes a React element (e.g. <App />)
// and returns a DOM or Native node representing the mounted tree.
function mount(element) {
  var type = element.type;
  var props = element.props;

  // We will determine the rendered element
  // by either running the type as function
  // or creating an instance and calling render().
  var renderedElement;
  if (isClass(type)) {
    // Component class
    var publicInstance = new type(props);
    // Set the props
    publicInstance.props = props;
    // Call the lifecycle if necessary
    if (publicInstance.componentWillMount) {
      publicInstance.componentWillMount();
    }
    // Get the rendered element by calling render()
    renderedElement = publicInstance.render();
  } else {
    // Component function
    renderedElement = type(props);
  }

  // This process is recursive because a component may
  // return an element with a type of another component.
  return mount(renderedElement);

  // Note: this implementation is incomplete and recurses infinitely!
  // It only handles elements like <App /> or <Button />.
  // It doesn't handle elements like <div /> or <p /> yet.
}

var rootEl = document.getElementById('root');
var node = mount(<App />);
rootEl.appendChild(node);

```

주의

이는 의사코드입니다. 실제 구현과 비슷하지 않습니다. 우리가 이 과정을 언제 멈출 지 결정을 한 적이 없기 때문에 스택 오버플로 우 또한 야기할 수 있습니다.

위의 예에서 몇가지 핵심 아이디어를 요약해 봅시다.

- React 엘리먼트는 컴포넌트 타입(예: `App`)과 `props`를 나타내는 일반 객체입니다.
- 사용자 정의된 컴포넌트(예: `App`)은 `class`이거나 함수일 수 있지만 모두 엘리먼트로 렌더링됩니다.
- "마운팅"은 최상위 React 엘리먼트(예: `<App />`)로부터 DOM 또는 네이티브 트리를 만드는 재귀적인 과정입니다.

호스트 엘리먼트 마운팅

이 과정은 우리가 스크린에 무언가를 렌더링하지 않는다면 무의미해집니다.

사용자 정의된("composite") 컴포넌트 외에도, React 엘리먼트는 플랫폼 특유의("host") 컴포넌트를 나타낼 수 있습니다. 예를 들어, `Button`은 렌더링된 메서드에서 `<div />`를 리턴할 수 있습니다.

엘리먼트의 `type`이 문자열인 경우, 우리는 호스트 엘리먼트로 처리합니다.

```
console.log(<div />);
// { type: 'div', props: {} }
```

이 곳에는 호스트 엘리먼트와 관련된 사용자 정의된 코드가 없습니다.

재조정자가 호스트 엘리먼트를 만나게 되면, 렌더러가 호스트 엘리먼트를 마운트할 수 있도록 관리합니다. 예를 들어, React DOM은 DOM 노드를 생성할 것입니다.

호스트 엘리먼트가 자식을 가지고 있으면, reconciler가 위와 동일한 알고리즘에 따라 재귀적으로 자식을 마운트합니다. 이는 자식이 호스트(like `<div><hr /></div>`)인지 사용자 정의(like `<div><Button /></div>`) 되었는지는 상관이 없습니다.

자식에 의해 만들어진 DOM 노드는 부모 DOM 노드로 추가되며, 재귀적으로 전체 DOM 구조가 조립됩니다.

주의

재조정자 자체는 DOM에 연결되어 있지 않습니다. 마운트의 정확한 결과(소스 코드에서 "mount image"로 불리는)는 렌더러에 의존하고, DOM 노드(React DOM), 문자열(React DOM Server) 또는 네이티브 뷰어(React Native)를 나타내는 숫자가 될 수도 있습니다.

호스트 엘리먼트를 다루기 위해 코드를 확장하는 경우, 다음과 같이 보일 수 있습니다.

```
function isClass(type) {
  // React.Component subclasses have this flag
  return (
    Boolean(type.prototype) &&
    Boolean(type.prototype.isReactComponent)
  );
}

// This function only handles elements with a composite type.
// For example, it handles <App /> and <Button />, but not a <div />.
function mountComposite(element) {
  var type = element.type;
  var props = element.props;

  var renderedElement;
  if (isClass(type)) {
    // Component class
    var publicInstance = new type(props);
    // Set the props
    publicInstance.props = props;
    // Call the lifecycle if necessary
    if (publicInstance.componentWillMount) {
      publicInstance.componentWillMount();
    }
  }
}
```

```

    }
    renderedElement = publicInstance.render();
} else if (typeof type === 'function') {
  // Component function
  renderedElement = type(props);
}

// This is recursive but we'll eventually reach the bottom of recursion when
// the element is host (e.g. <div />) rather than composite (e.g. <App />):
return mount(renderedElement);
}

// This function only handles elements with a host type.
// For example, it handles <div /> and <p /> but not an <App />.
function mountHost(element) {
  var type = element.type;
  var props = element.props;
  var children = props.children || [];
  if (!Array.isArray(children)) {
    children = [children];
  }
  children = children.filter(Boolean);

  // This block of code shouldn't be in the reconciler.
  // Different renderers might initialize nodes differently.
  // For example, React Native would create iOS or Android views.
  var node = document.createElement(type);
  Object.keys(props).forEach(propName => {
    if (propName !== 'children') {
      node.setAttribute(propName, props[propName]);
    }
  });

  // Mount the children
  children.forEach(childElement => {
    // Children may be host (e.g. <div />) or composite (e.g. <Button />).
    // We will also mount them recursively:
    var childNode = mount(childElement);

    // This line of code is also renderer-specific.
    // It would be different depending on the renderer:
    node.appendChild(childNode);
  });

  // Return the DOM node as mount result.
  // This is where the recursion ends.
  return node;
}

function mount(element) {
  var type = element.type;
  if (typeof type === 'function') {
    // User-defined components
    return mountComposite(element);
  } else if (typeof type === 'string') {
    // Platform-specific components
    return mountHost(element);
  }
}

```

```
var rootEl = document.getElementById('root');
var node = mount(<App />);
rootEl.appendChild(node);
```

이는 동작하지만 실제로 재조정자가 구현되는 방식과는 거리가 멍니다. 누락된 핵심 요소는 업데이트에 대한 지원입니다.

내부 인스턴스의 소개

React의 가장 큰 특징은 모든 것을 다시 렌더링할 수 있고, DOM을 다시 생성하거나 상태를 초기화시키지 않아도 된다는 점입니다.

```
root.render(<App />);
// Should reuse the existing DOM:
root.render(<App />);
```

그러나, 위의 구현은 초기 트리를 어떻게 마운트 하는지만 알고 있습니다. 모든 `publicInstance` 와 어떤 DOM `node` 가 각 컴포넌트에 대응되는지와 같은 필수 정보를 담고 있지 않기 때문에 업데이트를 할 수 없습니다.

스택 재조정자의 코드베이스가 `mount()` 함수를 메서드로 만들고 `class`에 배치하여 위와 같은 문제를 해결합니다. 이러한 접근에는 여러 단점이 있고, 현재 우리는 재조정자를 다시 작성하고 있으며 스택 재조정자와는 다른 반대 방향으로 나아가고 있습니다. 그렇지만, 스택 재조정자가 지금 작동하는 방식입니다.

`mountHost` 와 `mountComposite` 함수를 분리하는 것 대신에, 우리는 `DOMComponent` 와 `CompositeComponent` 의 두 가지 `class`를 생성합니다.

두 `class` 모두 `element` 를 받아들이는 생성자 뿐만 아니라 마운트된 노드를 반환해주는 `mount()` 메서드를 가지고 있습니다. 최상위 `mount()` 함수를 올바른 `class`로 인스턴스화 하는 팩토리로 대체합니다.

```
function instantiateComponent(element) {
  var type = element.type;
  if (typeof type === 'function') {
    // User-defined components
    return new CompositeComponent(element);
  } else if (typeof type === 'string') {
    // Platform-specific components
    return new DOMComponent(element);
  }
}
```

먼저, `CompositeComponent` 을 구현한 것을 봅시다.

```
class CompositeComponent {
  constructor(element) {
    this.currentElement = element;
    this.renderedComponent = null;
    this.publicInstance = null;
  }

  getPublicInstance() {
    // For composite components, expose the class instance.
    return this.publicInstance;
  }

  mount() {
    var element = this.currentElement;
    var type = element.type;
    var props = element.props;
```

```

var publicInstance;
var renderedElement;
if (isClass(type)) {
  // Component class
  publicInstance = new type(props);
  // Set the props
  publicInstance.props = props;
  // Call the lifecycle if necessary
  if (publicInstance.componentWillMount) {
    publicInstance.componentWillMount();
  }
  renderedElement = publicInstance.render();
} else if (typeof type === 'function') {
  // Component function
  publicInstance = null;
  renderedElement = type(props);
}

// Save the public instance
this.publicInstance = publicInstance;

// Instantiate the child internal instance according to the element.
// It would be a DOMComponent for <div /> or <p />,
// and a CompositeComponent for <App /> or <Button />;
var renderedComponent = instantiateComponent(renderedElement);
this.renderedComponent = renderedComponent;

// Mount the rendered output
return renderedComponent.mount();
}
}

```

이는 이전 `mountComposite()` 구현과 크게 다르지 않지만, `this.currentElement`, `this.renderedComponent`, `this.publicInstance` 와 같이 업데이트에 사용할 수 있도록 정보를 저장할 수 있습니다.

`CompositeComponent` 의 인스턴스가 사용자가 제공하는 `element.type`의 인스턴스와 다르다는 것을 주의해주세요. `CompositeComponent` 는 재조정자의 세부 구현 내용이고, 사용자에게는 노출되지 않습니다. 사용자 정의된 `class`는 `element.type`로부터 얻고, `CompositeComponent` 가 이에 대한 인스턴스를 생성합니다.

흔동을 막기 위해, `CompositeComponent` 와 `DOMComponent` 의 인스턴스를 "내부 인스턴스"라 부릅니다. 이를 통해 몇 가지 오래 지속되는 데이터를 내부 인스턴스와 연결시킬 수 있습니다. 오직 렌더러와 재조정자만 내부 인스턴스를 알 수 있습니다.

반면, 사용자 정의된 `class`의 인스턴스를 "공개된 인스턴스(public instance)"라고 부릅니다. 공개된 인스턴스는 `render()` 와 사용자가 직접 작성한 여러 메서드에서 `this` 로 표시됩니다.

`DOMComponent` `class`의 `mount()` 메서드로 리팩터링된 `mountHost()` 메서드 또한 비슷하게 보입니다.

```

class DOMComponent {
  constructor(element) {
    this.currentElement = element;
    this.renderedChildren = [];
    this.node = null;
  }

  getPublicInstance() {
    // For DOM components, only expose the DOM node.
    return this.node;
  }
}

```

```

mount() {
  var element = this.currentElement;
  var type = element.type;
  var props = element.props;
  var children = props.children || [];
  if (!Array.isArray(children)) {
    children = [children];
  }

  // Create and save the node
  var node = document.createElement(type);
  this.node = node;

  // Set the attributes
  Object.keys(props).forEach(propName => {
    if (propName !== 'children') {
      node.setAttribute(propName, props[propName]);
    }
  });

  // Create and save the contained children.
  // Each of them can be a DOMComponent or a CompositeComponent,
  // depending on whether the element type is a string or a function.
  var renderedChildren = children.map(instantiateComponent);
  this.renderedChildren = renderedChildren;

  // Collect DOM nodes they return on mount
  var childNodes = renderedChildren.map(child => child.mount());
  childNodes.forEach(childNode => node.appendChild(childNode));

  // Return the DOM node as mount result
  return node;
}

```

`mountHost()`로 리팩터링한 후의 주요 차이점은 `this.node`와 내부 DOM 컴포넌트 인스턴스와 연결된 `this.renderedChildren`을 유지한다는 것입니다. 향후 non-destructive 업데이트 적용에도 활용할 예정입니다.

결과적으로, 복합 또는 호스트인 각 내부 인스턴스는 이제 자식 내부 인스턴스를 가리킵니다. 이를 시각화하기 위해 함수 `<App>` 컴포넌트가 `<Button>` class 컴포넌트를 렌더링하고 `Button` class가 `<div>`를 렌더링하는 경우 내부 인스턴스 트리는 다음과 같이 보일 것입니다.

```

[object CompositeComponent] {
  currentElement: <App />,
  publicInstance: null,
  renderedComponent: [object CompositeComponent] {
    currentElement: <Button />,
    publicInstance: [object Button],
    renderedComponent: [object DOMComponent] {
      currentElement: <div />,
      node: [object HTMLDivElement],
      renderedChildren: []
    }
  }
}

```

DOM에서는 `<div>`만 보일 것입니다. 그러나 내부 인스턴스 트리에는 복합적인 것과 호스트 내부 인스턴스가 모두 포함되어 있습니다.

복합 내부 인스턴스는 다음을 저장해야 합니다.

- 현재 엘리먼트
- 엘리먼트 타입이 클래스라면 `public` 인스턴스
- 단일 렌더링된 내부 인스턴스. `DOMComponent` 또는 `CompositeComponent` 가 될 수 있습니다.

호스트 내부 인스턴스는 다음을 저장해야 합니다.

- 현재 엘리먼트
- DOM 노드
- 모든 자식 내부 인스턴스. 각 인스턴스는 `DOMComponent` 또는 `CompositeComponent` 일 수 있습니다.

더 복잡한 애플리케이션에서 내부 인스턴스 트리가 어떻게 구성되는지 상상하기 어려울 경우, [React DevTools](#)는 호스트 인스턴스를 회색 그리고 복합 인스턴스를 보라색으로 강조하므로 가까운 근사치를 줄 수 있습니다.

```
<TodoApp>
  ▼<div>
    <h3>TODO</h3>
    ▼<TodoList items=[...]>
      ▼<ul>
        <li key="1475260226151">Buy milk</li>
      </ul>
    </TodoList>
    ▼<form onSubmit=bound handleSubmit()>
      <input onChange=bound onChange() value="" />
      <button>Add #2</button>
    </form>
  </div>
</TodoApp>
```

리팩토링을 완료하기 위해 전체 트리를 컨테이너 노드와 공개 인스턴스에 마운트하는 기능을 소개합니다.

```
function mountTree(element, containerNode) {
  // Create the top-level internal instance
  var rootComponent = instantiateComponent(element);

  // Mount the top-level component into the container
  var node = rootComponent.mount();
  containerNode.appendChild(node);

  // Return the public instance it provides
  var publicInstance = rootComponent.getPublicInstance();
  return publicInstance;
}

var rootEl = document.getElementById('root');
mountTree(<App />, rootEl);
```

마운트 해제

이제 자식들과 DOM 노드를 유지하는 내부 인스턴스를 가지고 있으므로, 마운트 해제를 구현할 수 있습니다. 복합 컴포넌트의 경우, 마운트 해제가 생명주기 메서드를 재귀적으로 호출합니다.

```
class CompositeComponent {
  // ...

  unmount() {
    // Call the lifecycle method if necessary
    var publicInstance = this.publicInstance;
    if (publicInstance) {
      if (publicInstance.componentWillUnmount) {
        publicInstance.componentWillUnmount();
      }
    }

    // Unmount the single rendered component
    var renderedComponent = this.renderedComponent;
    renderedComponent.unmount();
  }
}
```

`DOMComponent`의 경우 마운트 해제는 각 자식에게 마운트 해제를 지시합니다.

```
class DOMComponent {
  // ...

  unmount() {
    // Unmount all the children
    var renderedChildren = this.renderedChildren;
    renderedChildren.forEach(child => child.unmount());
  }
}
```

실제로 DOM 컴포넌트를 마운트 해제하면 이벤트 리스너가 제거되고 캐시가 일부 지워지지만 이러한 자세한 내용은 넘어가겠습니다.

이제 `ReactDOM.unmountComponentAtNode()` 와 유사한 `unmountTree(containerNode)`라는 새로운 최상위 함수를 추가할 수 있습니다.

```
function unmountTree(containerNode) {
  // Read the internal instance from a DOM node:
  // (This doesn't work yet, we will need to change mountTree() to store it.)
  var node = containerNode.firstChild;
  var rootComponent = node._internalInstance;

  // Unmount the tree and clear the container
  rootComponent.unmount();
  containerNode.innerHTML = '';
}
```

이것이 작동하려면 DOM 노드에서 내부 루트 인스턴스를 읽어야 합니다. `_internalInstance` 속성을 루트 DOM 노드에 추가하도록 `mountTree()`를 수정합니다. 또한 `mountTree()`를 가르쳐 기존 트리를 여러 번 파괴할 수 있도록 할 것입니다.

```

function mountTree(element, containerNode) {
  // Destroy any existing tree
  if (containerNode.firstChild) {
    unmountTree(containerNode);
  }

  // Create the top-level internal instance
  var rootComponent = instantiateComponent(element);

  // Mount the top-level component into the container
  var node = rootComponent.mount();
  containerNode.appendChild(node);

  // Save a reference to the internal instance
  node._internalInstance = rootComponent;

  // Return the public instance it provides
  var publicInstance = rootComponent.getPublicInstance();
  return publicInstance;
}

```

이제 `mountTree()` 또는 `unmountTree()`를 반복적으로 실행하면 오래된 트리가 제거되고 컴포넌트에서 `componentWillUnmount()` 생명주기 메서드가 실행됩니다.

업데이트

이전 섹션에서, 마운트 해제를 구현했습니다. 그러나 각각의 prop 변화가 전체 트리를 마운트 해제하고 마운트한다면 React는 그다지 유용하지 않을 것입니다. 조정자의 목표는 DOM과 상태를 보전하기 위해 가능한 경우 기존 인스턴스를 재사용 하는 것입니다.

```

var rootEl = document.getElementById('root');

mountTree(<App />, rootEl);
// Should reuse the existing DOM:
mountTree(<App />, rootEl);

```

내부 인스턴스 계약을 메서드 하나를 추가해서 확장할 것입니다. `mount()` 와 `unmount()` 외에도 , `DOMComponent` `CompositeComponent` 모두 `receive(nextElement)` 라고 불리는 새로운 메서드를 구현합니다.

```

class CompositeComponent {
  // ...

  receive(nextElement) {
    // ...
  }
}

class DOMComponent {
  // ...

  receive(nextElement) {
    // ...
  }
}

```

`nextElement`에 의해 제공된 설명을 통해 컴포넌트(또한 어떠한 자식)를 최신 상태로 만들기 위해 필요한 모든 것을 하는게 이 메서드의 일입니다.

실제로 일어나는 일은 내부 트리를 반복적으로 순회하고 각 내부 인스턴스가 업데이트를 받도록 하는 것이지만, 이 부분은 종종 "가상 DOM 비교"로 설명됩니다.

복합 컴포넌트의 업데이트

복합 컴포넌트가 새로운 엘리먼트를 받으면 `componentWillUpdate()` 생명주기 메서드를 실행합니다.

그런 다음 새로운 props와 함께 컴포넌트를 다시 렌더링 하고, 다음 렌더링 된 엘리먼트를 얻습니다.

```
class CompositeComponent {
  // ...

  receive(nextElement) {
    var prevProps = this.currentElement.props;
    var publicInstance = this.publicInstance;
    var prevRenderedComponent = this.renderedComponent;
    var prevRenderedElement = prevRenderedComponent.currentElement;

    // Update *own* element
    this.currentElement = nextElement;
    var type = nextElement.type;
    var nextProps = nextElement.props;

    // Figure out what the next render() output is
    var nextRenderedElement;
    if (isClass(type)) {
      // Component class
      // Call the lifecycle if necessary
      if (publicInstance.componentWillUpdate) {
        publicInstance.componentWillUpdate(nextProps);
      }
      // Update the props
      publicInstance.props = nextProps;
      // Re-render
      nextRenderedElement = publicInstance.render();
    } else if (typeof type === 'function') {
      // Component function
      nextRenderedElement = type(nextProps);
    }
  }
}
```

그 다음, 렌더링된 엘리먼트의 `type`을 살펴 볼 수 있습니다. 마지막 렌더링 이후 `type`이 변경되지 않았다면 아래 컴포넌트도 업데이트할 수 있습니다.

예를 들어, 처음에 `<Button color="red" />`를 반환하고 두 번째로 `<Button color="blue" />`를 반환했다면, 대응되는 내부 인스턴스에게 다음 엘리먼트를 `receive()` 하라고 말해줄 수 있습니다.

```
// ...

// If the rendered element type has not changed,
// reuse the existing component instance and exit.
if (prevRenderedElement.type === nextRenderedElement.type) {
  prevRenderedComponent.receive(nextRenderedElement);
  return;
```

```
}
```

```
// ...
```

그러나, 다음 렌더링된 엘리먼트가 이전에 렌더링된 엘리먼트와 다른 `type` 을 가지고 있다면, 우리는 내부 인스턴스를 업데이트할 수 없습니다. `<button>` 은 `<input>` 이 될 수 없습니다.

대신에, 기존 내부 인스턴스를 마운트 해제 하고 렌더링된 엘리먼트 태입에 해당하는 새 인스턴스를 마운트 해야 합니다. 예를 들어 이전에 `<button />` 을 렌더링한 컴포넌트가 `<input />` 을 렌더링할 때 다음과 같이 됩니다.

```
// ...

// If we reached this point, we need to unmount the previously
// mounted component, mount the new one, and swap their nodes.

// Find the old node because it will need to be replaced
var prevNode = prevRenderedComponent.getHostNode();

// Unmount the old child and mount a new child
prevRenderedComponent.unmount();
var nextRenderedComponent = instantiateComponent(nextRenderedElement);
var nextNode = nextRenderedComponent.mount();

// Replace the reference to the child
this.renderedComponent = nextRenderedComponent;

// Replace the old node with the new one
// Note: this is renderer-specific code and
// ideally should live outside of CompositeComponent:
prevNode.parentNode.replaceChild(nextNode, prevNode);

}
```

이를 요약하면 복합 컴포넌트가 새로운 엘리먼트를 수신할 때, 해당 컴포넌트는 렌더링된 내부 인스턴스에 업데이트를 넘기거나, 마운트 해제 하여 그 위치에 컴포넌트를 마운트 할 수 있습니다.

엘리먼트를 받는 대신 컴포넌트를 다시 마운트하는 또 다른 조건이 있는데, 엘리먼트의 `key` 가 변경될 때입니다. 이미 복잡한 자습서를 더 복잡하게 만들기 때문에 이 문서에서 `key` 처리에 대해서는 논의하지 않습니다.

특정 플랫폼 관련 노드를 찾아 업데이트하는 동안 교체할 수 있도록 내부 인스턴스 계약에 `getHostNode()` 라는 메서드를 추가해야 한다는 점을 주의해주세요. 구현은 두 클래스 모두에서 간단합니다.

```
class CompositeComponent {
// ...

getHostNode() {
  // Ask the rendered component to provide it.
  // This will recursively drill down any composites.
  return this.renderedComponent.getHostNode();
}

class DOMComponent {
// ...

getHostNode() {
```

```

        return this.node;
    }
}

```

호스트 컴포넌트 업데이트

`DOMComponent` 와 같은 호스트 컴포넌트 구현은 다르게 업데이트 됩니다. 엘리먼트를 수신할 때 특정 플랫폼 관련 뷰를 업데이트해야 합니다. React DOM의 경우 DOM 특성을 업데이트 하는 것을 의미합니다.

```

class DOMComponent {
// ...

receive(nextElement) {
    var node = this.node;
    var prevElement = this.currentElement;
    var prevProps = prevElement.props;
    var nextProps = nextElement.props;
    this.currentElement = nextElement;

    // Remove old attributes.
    Object.keys(prevProps).forEach(propName => {
        if (propName !== 'children' && !nextProps.hasOwnProperty(propName)) {
            node.removeAttribute(propName);
        }
    });
    // Set next attributes.
    Object.keys(nextProps).forEach(propName => {
        if (propName !== 'children') {
            node.setAttribute(propName, nextProps[propName]);
        }
    });
}

// ...

```

그리고 나서 호스트 컴포넌트는 자식들을 업데이트 할 필요가 있습니다. 복합 컴포넌트와 다르게 둘 이상의 자식이 포함될 수 있습니다.

단순화된 예시에서 내부 인스턴스 배열을 사용하며, 수신된 `type` 과 이전 `type` 의 일치 여부에 따라 내부 인스턴스를 업데이트하거나 교체하는 방식으로 순회합니다. 실제 재조정자는 삽입과 삭제 외에 엘리먼트의 `key` 를 가진 뒤 변경 여부를 추적하지만 여기서는 관련 코드를 생략했습니다.

리스트에서 자식에 대한 DOM 연산을 수집하여 일괄적으로 실행할 수 있도록 합니다.

```

// ...

// These are arrays of React elements:
var prevChildren = nextProps.children || [];
if (!Array.isArray(prevChildren)) {
    prevChildren = [prevChildren];
}
var nextChildren = nextProps.children || [];
if (!Array.isArray(nextChildren)) {
    nextChildren = [nextChildren];
}
// These are arrays of internal instances:
var prevRenderedChildren = this.renderedChildren;
var nextRenderedChildren = [];

// As we iterate over children, we will add operations to the array.

```

```

var operationQueue = [];

// Note: the section below is extremely simplified!
// It doesn't handle reorders, children with holes, or keys.
// It only exists to illustrate the overall flow, not the specifics.

for (var i = 0; i < nextChildren.length; i++) {
  // Try to get an existing internal instance for this child
  var prevChild = prevRenderedChildren[i];

  // If there is no internal instance under this index,
  // a child has been appended to the end. Create a new
  // internal instance, mount it, and use its node.
  if (!prevChild) {
    var nextChild = instantiateComponent(nextChildren[i]);
    var node = nextChild.mount();

    // Record that we need to append a node
    operationQueue.push({type: 'ADD', node});
    nextRenderedChildren.push(nextChild);
    continue;
  }

  // We can only update the instance if its element's type matches.
  // For example, <Button size="small" /> can be updated to
  // <Button size="large" /> but not to an <App />.
  var canUpdate = prevChildren[i].type === nextChildren[i].type;

  // If we can't update an existing instance, we have to unmount it
  // and mount a new one instead of it.
  if (!canUpdate) {
    var prevNode = prevChild.getHostNode();
    prevChild.unmount();

    var nextChild = instantiateComponent(nextChildren[i]);
    var nextNode = nextChild.mount();

    // Record that we need to swap the nodes
    operationQueue.push({type: 'REPLACE', prevNode, nextNode});
    nextRenderedChildren.push(nextChild);
    continue;
  }

  // If we can update an existing internal instance,
  // just let it receive the next element and handle its own update.
  prevChild.receive(nextChildren[i]);
  nextRenderedChildren.push(prevChild);
}

// Finally, unmount any children that don't exist:
for (var j = nextChildren.length; j < prevChildren.length; j++) {
  var prevChild = prevRenderedChildren[j];
  var node = prevChild.getHostNode();
  prevChild.unmount();

  // Record that we need to remove the node
  operationQueue.push({type: 'REMOVE', node});
}

// Point the list of rendered children to the updated version.

```

```
this.renderedChildren = nextRenderedChildren;
// ...
```

마지막 단계로, DOM 연산을 실행합니다. 또한, 실제 재조정자 코드는 이동도 처리하기 때문에 매우 복잡합니다.

```
// ...

// Process the operation queue.
while (operationQueue.length > 0) {
  var operation = operationQueue.shift();
  switch (operation.type) {
    case 'ADD':
      this.node.appendChild(operation.node);
      break;
    case 'REPLACE':
      this.node.replaceChild(operation.nextNode, operation.prevNode);
      break;
    case 'REMOVE':
      this.node.removeChild(operation.node);
      break;
  }
}
```

그리고 그것은 호스트 컴포넌트를 업데이트하기 위한 것입니다.

최상위 업데이트

이제 `CompositeComponent` 와 `DOMComponent` 모두 `receive(nextElement)` 메서드를 구현하므로 엘리먼트 `type`이 이전과 같을 때 사용하도록 최상위 `mountTree()` 함수를 변경할 수 있습니다.

```
function mountTree(element, containerNode) {
  // Check for an existing tree
  if (containerNode.firstChild) {
    var prevNode = containerNode.firstChild;
    var prevRootComponent = prevNode._internalInstance;
    var prevElement = prevRootComponent.currentElement;

    // If we can, reuse the existing root component
    if (prevElement.type === element.type) {
      prevRootComponent.receive(element);
      return;
    }

    // Otherwise, unmount the existing tree
    unmountTree(containerNode);
  }

  // ...
}
```

이제 동일한 타입으로 `mountTree()` 를 두 번 호출해도 파괴적이지 않습니다.

```
var rootEl = document.getElementById('root');

mountTree(<App />, rootEl);
// Reuses the existing DOM:
mountTree(<App />, rootEl);
```

These are the basics of how React works internally.

우리가 놓치고 간 것들

이 문서는 실제 코드베이스에 비해 단순합니다. 우리가 다루지 않은 몇 가지 중요한 측면들이 있습니다.

- 컴포넌트는 `null`을 렌더링할 수 있으며, 재조정자는 배열 및 렌더링된 출력에서 "빈 슬롯"을 처리할 수 있습니다.
- 재조정자는 또한 엘리먼트에서 `key`를 읽고, 이를 사용하여 배열의 엘리먼트와 일치하는 내부 인스턴스를 설정합니다. 실제 React 구현의 많은 복잡성은 이와 관련이 있습니다.
- 복합 및 호스트 내부 인스턴스 `class` 외에도 "text" 및 "empty" 컴포넌트에 대한 `class`도 있습니다. 텍스트 노드와 `null`을 렌더링해서 얻는 "empty slots"을 나타냅니다.
- 렌더러는 `주입`을 사용하여 재조정자에게 호스트 내부 `class`를 전달합니다. 예를 들어, React DOM은 재조정자에게 호스트 내부 인스턴스 구현으로 `ReactDOMComponent`를 사용하도록 지시합니다.
- 자식 목록을 업데이트하는 논리는 React DOM과 React Native에서 호스트 내부 인스턴스 `class` 구현에 사용되는 `ReactMultiChild`라는 mixin으로 추출됩니다.
- 재조정자는 복합 컴포넌트의 `setState()`에 대한 지원도 구현합니다. 이벤트 핸들러 내부의 여러 업데이트가 단일 업데이트로 일괄 처리됩니다.
- 재조정자는 또한 복합 컴포넌트 및 호스트 노드에 `ref`를 연결 및 분리하는 작업을 수행합니다.
- `componentDidMount()` 및 `componentDidUpdate()`와 같이 DOM이 준비된 후 호출되는 생명주기 메서드는 "콜백 큐"로 수집되어 단일 배치로 실행됩니다.
- React는 현재 업데이트에 대한 정보를 "트랜잭션"이라고 하는 내부 객체에 넣습니다. 트랜잭션은 보류 중인 생명주기 메서드 대기열의 추적, 현재 DOM 중첩에 대한 경고 및 특정 업데이트에 "전역적인" 다른 모든 것을 추적하는 데 유용합니다. 또한 트랜잭션은 업데이트 후 React가 모든 것을 정리하도록 보장합니다. 예를 들어 React DOM에서 제공하는 트랜잭션 `class`는 업데이트 후 입력 선택을 복원합니다.

코드에 대해 알아보기

- `ReactMount` 는 이 자습서에서 `mountTree()` 및 `unmountTree()` 와 같은 코드가 사용되는 곳입니다. 최상위 컴포넌츠의 마운트와 마운트 해제를 관리합니다. `ReactNativeMount` 는 React Native 아날로그입니다.
- `ReactDOMComponent` 는 본 자습서의 `DOMComponent` 와 동등합니다. React DOM 렌더러에 대한 호스트 컴포넌트 `class`를 구현합니다. `ReactNativeBaseComponent` 는 React Native 아날로그입니다.
- `ReactCompositeComponent` 는 본 자습서의 `CompositeComponent` 와 동등한 것입니다. 사용자 정의 컴포넌트 호출 및 상태 유지 관리 작업을 처리합니다.
- `instantiateReactComponent` 에는 엘리먼트에 대해 구성할 올바른 내부 인스턴스 `class`를 선택하는 스위치가 포함되어 있습니다. 이 자습서에서는 `instantiateComponent()` 와 같습니다.

- `ReactReconciler` 는 `mountComponent()`, `receiveComponent()` 및 `unmountComponent()` 메서드가 있는 wrapper입니다. 내부 인스턴스에 대한 기본 구현을 호출하지만, 또한 모든 내부 인스턴스 구현에 의해 공유되는 그들 주변의 일부 코드를 포함합니다.
- `ReactChildReconciler` 는 자식의 엘리먼트 `key` 에 따라 자식을 마운트, 업데이트 및 마운트 해제하는 코드를 구현합니다.
- `ReactMultiChild` 는 자식 삽입, 삭제 및 렌더러와 독립적으로 이동하기 위한 작업 대기열 처리를 구현합니다.
- 레거시를 위해 react codebase에 `mount()`, `receive()` 및 `unmount()` 를 실제로 각각 `mountComponent()`, `receiveComponent()`, `unmountComponent()` 라고 불러지지만, 엘리먼트를 받습니다.
- 내부 인스턴스의 속성은 `_currentElement` 와 같이 밑줄로 시작합니다. 코드베이스 전체에 걸쳐 읽기 전용 퍼블릭 필드로 간주됩니다.

미래의 방향

스택 재조정자는 작업을 중단하거나 청크로 분할할 수 없는 것과 같은 태그난 한계가 있습니다. [완전히 다른 아키텍처](#)를 가진 [새로운 Fiber 재조정자](#)에 대한 작업이 진행 중입니다. 향후, 스택 재조정자를 이것으로 대체하려고 하지만, 현재는 피쳐 패리티와는 거리가 멉니다.

다음 단계

[다음 섹션](#)을 읽고 React 개발에 사용하는 설계 원칙에 대해 알아봅시다.

다른 라이브러리와 통합하기

React는 어떤 웹 애플리케이션에서든 사용할 수 있습니다. 다른 애플리케이션에 포함될 수 있으며 약간의 노력으로 React 안에 다른 애플리케이션 포함할 수 있습니다. 이 가이드는 [jQuery](#)와 [Backbone](#)의 통합에 중점을 맞추어 일반적인 몇 가지 사용 사례를 살펴봅니다. 동일한 아이디어로 기존 코드와 컴포넌트를 통합하는 데도 적용할 수 있습니다.

DOM 조작 플러그인과 통합하기

React는 React의 외부 DOM에서 일어나는 변화를 인식하지 못합니다. 자체 내부 표현에 따라서 업데이트를 할지 말지 결정합니다. 그리고 다른 라이브러리와 같은 DOM 노드를 다룬다면 React는 혼란스러울 것이며 복구할 방법이 없습니다.

React를 DOM에 영향을 미치는 다른 방법과 결합하는 것이 불가능하거나 심지어 어렵다는 것을 의미하지 않습니다. 각각의 작업을 염두에 두기만 하면 됩니다.

충돌을 피하는 가장 쉬운 방법은 React 컴포넌트가 업데이트되지 않게 막는 것입니다. React가 업데이트할 필요가 없는 빈 `<div>` 같은 요소를 렌더링하면 됩니다.

어떻게 문제에 접근하는가

이를 설명하기 위해 일반적인 jQuery 플러그인을 위한 래퍼에 대해 간략하게 알아보겠습니다.

최상위 DOM 엘리먼트에 `ref`를 붙입니다. `componentDidMount` 내부에서 jQuery 플러그인에 전달하기 위해 최상위 DOM 엘리먼트에 대한 참조를 얻습니다.

마운팅 후 React가 DOM에 건드리는 것을 방지하기 위해 `render()` 메서드에서 빈 `<div />`를 반환합니다. 해당 `<div />` 요소는 프로퍼티나 자식을 가지지 않기 때문에 React가 업데이트할 이유가 없습니다. jQuery 플러그인이 DOM의 일부를 다룰수 있게 자유롭게 관리할 수 있습니다.

```
class SomePlugin extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
    this.$el.somePlugin();
  }

  componentWillUnmount() {
    this.$el.somePlugin('destroy');
  }

  render() {
    return <div ref={el => this.el = el} />;
  }
}
```

`componentDidMount`, `componentWillUnmount` 두 가지의 생명주기 메서드를 정의했다는 것을 주의합니다. 많은 jQuery 플러그인은 DOM에 이벤트 리스너를 등록하므로 `componentWillUnmount` 안에서 해제하는 것이 중요합니다. 플러그인이 해제를 위한 메서드를 제공하지 않는다면 자체적으로 해당 메서드를 제공해야 합니다. 메모리 누수를 방지하기 위해 플러그인이 등록한 모든 이벤트 리스너를 제거해야 하는 것을 잊어서는 안 됩니다.

jQuery Chosen 플러그인과 통합하기

이러한 컨셉의 더 구체적인 예시를 위해 `<select>` 입력을 다루는 플러그인 [Chosen](#)에 대한 간단한 래퍼를 작성해 보겠습니다.

주의

이 방법이 가능하다고 해서 React 앱에 대한 최상의 접근 방법임을 의미하지는 않습니다. 가능하다면 React 컴포넌트를 사용하는 것을 권장합니다. React 컴포넌트는 React 애플리케이션에서 더 쉽게 재사용할 수 있으며, 해당 동작과 모양에 대해 더 많은 제어를 제공해 줍니다.

먼저, Chosen이 DOM에 무엇을 하는지 봅시다.

`<select>` DOM 노드에서 Chosen을 호출하면 원본 DOM 노드의 어트리뷰트를 읽고 인라인 스타일로 숨깁니다. 그리고 `<select>` 바로 뒤에 고유의 시각적 표현을 가진 별도의 DOM 노드를 추가합니다. 그런 다음 jQuery 이벤트를 발생시켜 변경 사항에 대해 알립니다.

`<Chosen>` 래퍼 React 컴포넌트로 만든 API라고 가정해 보겠습니다.

```
function Example() {
  return (
    <Chosen onChange={value => console.log(value)}>
      <option>vanilla</option>
      <option>chocolate</option>
      <option>strawberry</option>
    </Chosen>
  );
}
```

간결함을 위해 [비제어 컴포넌트](#)로 구현하겠습니다.

먼저, `<div>`로 감싸인 `<select>`를 반환하는 `render()` 메서드가 있는 빈 컴포넌트를 생성합니다.

```
class Chosen extends React.Component {
  render() {
    return (
      <div>
        <select className="Chosen-select" ref={el => this.el = el}>
          {this.props.children}
        </select>
      </div>
    );
  }
}
```

별도의 `<div>`로 `<select>`를 어떻게 감쌌는지 주의하세요. Chosen이 전달한 `<select>` 노드 바로 다음에 다른 DOM 요소를 추가하기 때문에 필요합니다. 하지만 React가 관여하는 한, `<div>`는 항상 단일 자식만 가집니다. React 업데이트가 Chosen이 추가한 DOM 노드와 충돌하지 않게 하는 방법입니다. React 흐름 외부에서 DOM을 수정하는 경우 React가 해당 DOM 노드를 건드릴 이유가 없는지 확인해야 합니다.

다음으로 생명주기 메서드를 구현해 보겠습니다. `componentDidMount`에서 `<select>` 노드의 `ref`를 사용하여 Chosen을 초기화합니다. 그리고 `componentWillUnmount`에서 이를 해제해야 합니다.

```

componentDidMount() {
  this.$el = $(this.el);
  this.$el.chosen();
}

componentWillUnmount() {
  this.$el.chosen('destroy');
}

```

[CodePen에서 사용해보세요](#)

React는 `this.el` 필드에 특별한 의미를 부여하지 않습니다. 이전에 `render()` 메서드에서 `ref`에 이 필드를 할당했기 때문에 작동합니다.

```
<select className="Chosen-select" ref={el => this.el = el}>
```

컴포넌트를 렌더링하기에 충분하지만, 값이 변경될 때마다 알림을 받기를 원합니다. 이를 위해 Chosen이 관리하는 `<select>`에서 jQuery `change` 이벤트를 구독합니다.

Chosen에 `this.props.onChange`를 바로 전달하지 않습니다. 왜냐하면 컴포넌트의 `props`가 여러 번 변경될 수 있으며 이벤트 핸들러를 포함하고 있기 때문입니다. 그 대신에 `this.props.onChange`를 호출하는 `handleChange()` 메서드를 선언하고 jQuery `change` 이벤트로 구독합니다.

```

componentDidMount() {
  this.$el = $(this.el);
  this.$el.chosen();

  this.handleChange = this.handleChange.bind(this);
  this.$el.on('change', this.handleChange);
}

componentWillUnmount() {
  this.$el.off('change', this.handleChange);
  this.$el.chosen('destroy');
}

handleChange(e) {
  this.props.onChange(e.target.value);
}

```

[CodePen에서 사용해보세요](#)

마지막으로 해야 할 남은 한 가지 있습니다. React에서 `prop`는 여러 번 바꿀 수 있습니다. 예를 들어 부모 컴포넌트의 `state`가 변경되면 컴포넌트가 다른 자식을 가질 수 있습니다. 통합을 사용하는 위치에서는 `prop`이 업데이트할 때 마다 수동으로 DOM을 업데이트해야 합니다. 더 이상 React가 DOM을 관리하지 않습니다.

Chosen 문서에서 따르면 jQuery `trigger()` API를 사용하여 원본 DOM 엘리먼트의 변경 사항에 대해 알 수 있습니다. React가 `<select>` 안에 `this.props.children`를 업데이트하지만 Chosen에게 자식 목록의 변경에 알려주는 `componentDidUpdate()` 생명주기 메서드도 추가합니다.

```

componentDidUpdate(prevProps) {
  if (prevProps.children !== this.props.children) {
    this.$el.trigger("chosen:updated");
  }
}

```

React 변화로 `<select>` 자식 관리가 될 때 Chosen이 해당 DOM 엘리먼트 업데이트를 알 수 있습니다.

Chosen 컴포넌트의 완전한 구현은 다음과 같습니다.

```
class Chosen extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
    this.$el.chosen();

    this.handleChange = this.handleChange.bind(this);
    this.$el.on('change', this.handleChange);
  }

  componentDidUpdate(prevProps) {
    if (prevProps.children !== this.props.children) {
      this.$el.trigger("chosen:updated");
    }
  }

  componentWillUnmount() {
    this.$el.off('change', this.handleChange);
    this.$el.chosen('destroy');
  }

  handleChange(e) {
    this.props.onChange(e.target.value);
  }

  render() {
    return (
      <div>
        <select className="Chosen-select" ref={el => this.el = el}>
          {this.props.children}
        </select>
      </div>
    );
  }
}
```

[CodePen에서 사용해보세요](#)

다른 뷰 라이브러리와 통합하기

`createRoot()`의 유연성 덕분에 다른 애플리케이션에 React를 포함할 수 있습니다.

React는 일반적으로 시작 시에 단일 루트 React 컴포넌트를 DOM에 로드하는 데 사용되지만 `createRoot()`는 앱처럼 크거나 버튼처럼 작은 UI의 독립적인 부분에 대해 여러 번 호출 할 수 있습니다.

실제로 Facebook에서 React를 사용되는 방식입니다. 이렇게 하면 React에서 애플리케이션을 한 부분씩 작성할 수 있으며 이를 기준의 서버에서 생성한 템플릿 및 다른 클라이언트 사이드 코드와 결합을 할 수 있습니다.

문자열 기반 렌더링을 React로 바꾸기

이전 웹 애플리케이션의 일반적인 패턴은 `$el.html(htmlString)`처럼 DOM의 청크를 문자열로 기술하고 DOM에 삽입하는 것입니다. 코드 베이스의 이러한 점들은 React를 소개하는데 완벽합니다. 문자열 기반 렌더링을 React 컴포넌트로 다시 작성하면 됩니다.

다음 jQuery 구현은...

```
$('#container').html('<button id="btn">Say Hello</button>');
$('#btn').click(function() {
  alert('Hello!');
});
```

React 컴포넌트를 사용해서 재작성할 수 있습니다.

```
function Button() {
  return <button id="btn">Say Hello</button>;
}

$('#btn').click(function() {
  alert('Hello!');
});
```

여기에서 더 많은 로직을 컴포넌트로 옮기고 일반적인 React practices를 채택할 수 있습니다. 예를 들어 컴포넌트에서 동일한 컴포넌트를 여러 번 렌더링할 수 있으므로 ID에 의존하지 않는 것이 좋습니다. 대신 [React 이벤트 시스템](#)을 사용하고 React `<button>` 요소에 클릭 핸들러를 직접 등록하면 됩니다.

```
function Button(props) {
  return <button onClick={props.onClick}>Say Hello</button>;
}

function HelloButton() {
  function handleClick() {
    alert('Hello!');
  }
  return <Button onClick={handleClick} />;
}
```

CodePen에서 사용해보세요

이런 격리된 컴포넌트를 원하는 만큼 가질 수 있으며 `ReactDOM.createRoot()`를 사용해서 다른 DOM 컨테이너로 렌더링할 수 있습니다. 점점 앱의 더 많은 부분을 React로 변환하면 더 큰 컴포넌트로 결합할 수 있고 `ReactDOM.createRoot()` 호출을 계층 구조 상위로 옮길 수 있습니다.

Backbone 뷰 안에 React 포함하기

[Backbone](#) 뷰는 일반적으로 HTML 문자열 또는 문자열로 제공되는 템플릿 함수를 사용하여 DOM 엘리먼트를 위한 콘텐츠를 생성합니다. 이 프로세스 또한 React 컴포넌트 렌더링으로 대체할 수 있습니다.

아래에서 `ParagraphView`라는 Backbone 뷰를 생성합니다. Backbone (`this.el`)이 제공하는 DOM 요소에 React `<Paragraph>` 컴포넌트를 렌더링하기 위해 Backbone의 `render()` 함수를 오버라이드합니다. 여기서도 `ReactDOM.createRoot()` 사용하고 있습니다.

```
function Paragraph(props) {
  return <p>{props.text}</p>;
}

const ParagraphView = Backbone.View.extend({
  initialize(options) {
    this.reactRoot = ReactDOM.createRoot(this.el);
  },
});
```

```

render() {
  const text = this.model.get('text');
  this.reactRoot.render(<Paragraph text={text} />);
  return this;
},
remove() {
  this.reactRoot.unmount();
  Backbone.View.prototype.remove.call(this);
}
);

```

[CodePen에서 사용해보세요](#)

`remove` 메서드 안에서 `root.unmount()` 호출하여 분리가 됐을 때 React가 컴포넌트 트리와 관련된 이벤트 핸들러와 다른 리소스를 등록 해제하는 것이 중요합니다.

React 트리안에서 컴포넌트가 사라질 때 자동으로 클린업이 실행되지만, 전체 트리를 수동으로 제거하기 때문에 이 메서드를 반드시 호출해야 합니다.

모델 레이어와 통합하기

React state, Flux, 또는 Redux 와 같이 일반적으로 단방향 데이터 흐름을 사용하는 것을 권장하지만 React 컴포넌트는 다른 프레임워크 또는 라이브러리에 있는 모델 레이어를 사용할 수 있습니다.

React 컴포넌트 안에서 Backbone 모델 사용하기

React 컴포넌트로부터 Backbone 모델, 컬렉션을 소비하는 가장 쉬운 방법은 다양한 변경 이벤트를 감지하거나 수동으로 업데이트를 하는 것입니다.

모델 렌더링을 담당하는 컴포넌트는 '`change`' 이벤트에 수신하는 반면 컬렉션 렌더링을 담당하는 컴포넌트는 '`add`', '`remove`' 이벤트를 수신합니다. 두 경우 새로운 데이터와 함께 컴포넌트를 다시 렌더링하기 위해 `this.forceUpdate()` 를 호출합니다.

아래의 예시에서, `List` 컴포넌트는 개별적으로 렌더링하기 위해 `Item` 컴포넌트를 사용해서 Backbone 컬렉션을 렌더링합니다.

```

class Item extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange() {
    this.forceUpdate();
  }

  componentDidMount() {
    this.props.model.on('change', this.handleChange);
  }

  componentWillUnmount() {
    this.props.model.off('change', this.handleChange);
  }

  render() {
    return <li>{this.props.model.get('text')}</li>;
  }
}

```

```

class List extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange() {
    this.forceUpdate();
  }

  componentDidMount() {
    this.props.collection.on('add', 'remove', this.handleChange);
  }

  componentWillUnmount() {
    this.props.collection.off('add', 'remove', this.handleChange);
  }

  render() {
    return (
      <ul>
        {this.props.collection.map(model => (
          <Item key={model.cid} model={model} />
        ))}
      </ul>
    );
  }
}

```

[CodePen에서 사용해보세요](#)

Backbone 모델에서 데이터 추출하기

위에 대한 접근 방법은 React 컴포넌트가 Backbone 모델과 컬렉션을 알고 있어야 합니다. 나중에 다른 데이터 관리 솔루션으로 이전할 계획이 있다면 Backbone에 대한 지식을 코드의 가능한 한 적은 부분에 집중하고자 할 수 있습니다.

해결책은 변경할 때마다 순수 데이터로 모델의 어트리뷰트를 추출하는 것입니다. 그리고 이 논리를 한곳에 모아둡니다. 그다음은 Backbone 모델의 모든 어트리뷰트를 state로 추출하여 깃сан 컴포넌트로 데이터를 전달하는 [고차함수 컴포넌트](#)입니다.

이 방법은, 고차함수 컴포넌트만 Backbone 모델 내부에 대해서 알고 있으면 됩니다. 그리고 앱 안에 있는 대부분 컴포넌트는 Backbone에 대해 몰라도 됩니다.

아래의 예시에서 모델의 어트리뷰트를 복사해서 초기 상태로 만듭니다. `change` 이벤트를 구독하고 언마운트시 구독 해제합니다. 이벤트가 발생했을 때 모델의 현재 어트리뷰트와 함께 state를 업데이트합니다. 마지막으로 `model` prop 자체가 변경되면 이전 모델을 구독 해제하고 새로운 모델을 구독합니다.

이 예시는 Backbone을 사용한 작업과 관련하여 모든 것을 망라하는 것이 아니라 일반적인 방법으로 어떻게 접근하는지에 아이디어를 줄 수 있습니다.

```

function connectToBackboneModel(WrappedComponent) {
  return class BackboneComponent extends React.Component {
    constructor(props) {
      super(props);
      this.state = Object.assign({}, props.model.attributes);
      this.handleChange = this.handleChange.bind(this);
    }
  }
}

```

```

componentDidMount() {
  this.props.model.on('change', this.handleChange);
}

componentWillReceiveProps(nextProps) {
  this.setState(Object.assign({}, nextProps.model.attributes));
  if (nextProps.model !== this.props.model) {
    this.props.model.off('change', this.handleChange);
    nextProps.model.on('change', this.handleChange);
  }
}

componentWillUnmount() {
  this.props.model.off('change', this.handleChange);
}

handleChange(model) {
  this.setState(model.changedAttributes());
}

render() {
  const propsExceptModel = Object.assign({}, this.props);
  delete propsExceptModel.model;
  return <WrappedComponent {...propsExceptModel} {...this.state} />;
}
}
}

```

사용하는 방법은 `NameInput` React 컴포넌트를 Backbone 모델과 연결합니다. 그리고 input의 값이 바뀔 때마다 `firstName` 어트리뷰트를 업데이트합니다.

```

function NameInput(props) {
  return (
    <p>
      <input value={props.firstName} onChange={props.handleChange} />
      <br />
      My name is {props.firstName}.
    </p>
  );
}

const BackboneNameInput = connectToBackboneModel(NameInput);

function Example(props) {
  function handleChange(e) {
    props.model.set('firstName', e.target.value);
  }

  return (
    <BackboneNameInput
      model={props.model}
      handleChange={handleChange}
    />
  );
}

const model = new Backbone.Model({ firstName: 'Frodo' });
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Example model={model} />);

```

[CodePen에서 사용해보세요](#)

이 기술은 Backbone에 제한되지 않습니다. 생명주기 메서드 안에서 변화를 구독하고 선택적으로 로컬 React state로 데이터를 복사한다면 React를 다른 모델 라이브러리와 사용할 수 있습니다.

[Go to TOC](#)

JSX 소개

아래 변수 선언을 살펴봅시다.

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

위에 희한한 태그 문법은 문자열도, HTML도 아닙니다.

JSX라 하며 JavaScript를 확장한 문법입니다. UI가 어떻게 생겨야 하는지 설명하기 위해 React와 함께 사용할 것을 권장합니다. JSX라고 하면 템플릿 언어가 떠오를 수도 있지만, JavaScript의 모든 기능이 포함되어 있습니다.

JSX는 React "엘리먼트(element)"를 생성합니다. 다음 섹션에서는 DOM에 어떻게 렌더링하는지 알아보겠습니다. 아래를 보면 JSX를 시작하기 위해 필요한 기본사항을 찾으실 수 있습니다.

JSX란?

React에서는 본질적으로 렌더링 로직이 UI 로직(이벤트가 처리되는 방식, 시간에 따라 state가 변하는 방식, 화면에 표시하기 위해 데이터가 준비되는 방식 등)과 연결된다는 사실을 받아들입니다.

React는 별도의 파일에 마크업과 로직을 넣어 기술을 인위적으로 분리하는 대신, 둘 다 포함하는 "컴포넌트"라고 부르는 느슨하게 연결된 유닛으로 관심사를 분리합니다. 이후 섹션에서 다시 컴포넌트로 돌아오겠지만, JS에 마크업을 넣는 게 익숙해지지 않는다면 이 이야기가 확신을 줄 것입니다.

React는 JSX 사용이 필수가 아니지만, 대부분의 사람은 JavaScript 코드 안에서 UI 관련 작업을 할 때 시각적으로 더 도움이 된다고 생각합니다. 또한 React가 더욱 도움이 되는 에러 및 경고 메시지를 표시할 수 있게 해줍니다.

일단 한번 시작해보겠습니다!

JSX에 표현식 포함하기

아래 예시에서는 `name`이라는 변수를 선언한 후 중괄호로 감싸 JSX 안에 사용하였습니다.

```
const name = 'Josh Perez';
const element = <h1>Hello, {name}</h1>;
```

JSX의 중괄호 안에는 유효한 모든 [JavaScript 표현식](#)을 넣을 수 있습니다. 예를 들어 `2 + 2`, `user.firstName` 또는 `formatName(user)` 등은 모두 유효한 JavaScript 표현식입니다.

아래 예시에서는 JavaScript 함수 호출의 결과인 `formatName(user)`을 `<h1>` 엘리먼트에 포함했습니다.

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};

const element = (
  <h1>
```

```
Hello, {formatName(user)}!
  </h1>
);
```

[CodePen에서 실행하기](#)

가독성을 좋게 하기 위해 JSX를 여러 줄로 나눴습니다. 필수는 아니지만, 이 작업을 수행할 때 [자동 세미콜론 삽입](#)을 피하고자 괄호로 묶는 것을 권장합니다.

JSX도 표현식입니다

컴파일이 끝나면, JSX 표현식이 정규 JavaScript 함수 호출이 되고 JavaScript 객체로 인식됩니다.

즉, JSX를 `if` 구문 및 `for` loop 안에 사용하고, 변수에 할당하고, 인자로서 받아들이고, 함수로부터 반환할 수 있습니다.

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

JSX 속성 정의

어트리뷰트에 따옴표를 이용해 문자열 리터럴을 정의할 수 있습니다.

```
const element = <a href="https://www.reactjs.org"> link </a>;
```

중괄호를 사용하여 어트리뷰트에 JavaScript 표현식을 삽입할 수도 있습니다.

```
const element = <img src={user.avatarUrl}></img>;
```

어트리뷰트에 JavaScript 표현식을 삽입할 때 중괄호 주변에 따옴표를 입력하지 마세요. 따옴표(문자열 값에 사용) 또는 중괄호(표현식에 사용) 중 하나만 사용하고, 동일한 어트리뷰트에 두 가지를 동시에 사용하면 안 됩니다.

경고

JSX는 HTML보다는 JavaScript에 가깝기 때문에, React DOM은 HTML 어트리뷰트 이름 대신 `camelCase` 프로퍼티 명명 규칙을 사용합니다.

예를 들어, JSX에서 `class`는 `className`가 되고 `tabindex`는 `tabIndex`가 됩니다.

JSX로 자식 정의

태그가 비어있다면 XML처럼 `/>` 를 이용해 바로 닫아주어야 합니다.

```
const element = <img src={user.avatarUrl} />;
```

JSX 태그는 자식을 포함할 수 있습니다.

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

JSX는 주입 공격을 방지합니다

JSX에 사용자 입력을 삽입하는 것은 안전합니다.

```
const title = response.potentiallyMaliciousInput;
// 이것은 안전합니다.
const element = <h1>{title}</h1>;
```

기본적으로 React DOM은 JSX에 삽입된 모든 값을 렌더링하기 전에 [이스케이프](#) 하므로, 애플리케이션에서 명시적으로 작성되지 않은 내용은 주입되지 않습니다. 모든 항목은 렌더링 되기 전에 문자열로 변환됩니다. 이런 특성으로 인해 [XSS \(cross-site-scripting\)](#) 공격을 방지할 수 있습니다.

JSX는 객체를 표현합니다.

Babel은 JSX를 `React.createElement()` 호출로 컴파일합니다.

다음 두 예시는 동일합니다.

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

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

`React.createElement()`는 버그가 없는 코드를 작성하는 데 도움이 되도록 몇 가지 검사를 수행하며, 기본적으로 다음과 같은 객체를 생성합니다.

```
// 주의: 다음 구조는 단순화되었습니다
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world!'
}
```

이러한 객체를 "React 엘리먼트"라고 하며, 화면에서 보고 싶은 것을 나타내는 표현이라 생각하면 됩니다. React는 이 객체를 읽어서, DOM을 구성하고 최신 상태로 유지하는 데 사용합니다.

[다음 섹션](#)에서는 DOM에 React 엘리먼트를 렌더링하는 방법에 대해 살펴보겠습니다.

팁

ES6 및 JSX 코드가 올바르게 표시되도록 편집기에 "Babel" 언어 설정을 사용하는 것을 권장합니다.

[Go to TOC](#)

JSX 이해하기

근본적으로, JSX는 `React.createElement(component, props, ...children)` 함수에 대한 문법적 설정을 제공할 뿐입니다.

```
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>
```

위의 코드는 아래와 같이 컴파일됩니다.

```
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Click Me'
)
```

자식 컴포넌트가 없다면 아래와 같이 자기 자신을 닫는 형태의 태그를 쓸 수 있습니다.

```
<div className="sidebar" />
```

위의 코드는 아래와 같이 컴파일 됩니다.

```
React.createElement(
  'div',
  {className: 'sidebar'}
)
```

특정 JSX가 어떻게 JavaScript로 변환되는지 시험해보고 싶다면 [온라인 babel 컴파일러](#)를 사용해보세요.

React Element의 타입 지정하기

JSX 태그의 첫 부분은 React element의 타입을 결정합니다.

대문자로 시작하는 JSX 태그는 React 컴포넌트를 지정합니다. 이 태그들은 같은 이름을 가진 변수들을 직접 참조합니다. `<Foo />` 와 같은 JSX 표현을 쓰려고 한다면 Foo가 반드시 스코프 내에 존재해야 합니다.

React가 스코프 내에 존재해야 합니다

JSX는 `React.createElement`를 호출하는 코드로 컴파일 되기 때문에 `React` 라이브러리 역시 JSX 코드와 같은 스코프 내에 존재해야만 합니다.

아래의 예시를 통해 보면, `React` 와 `CustomButton` 는 JavaScript 코드에선 직접적으로 사용되진 않지만 JSX 태그로 사용하기 위해 꼭 `import` 해야합니다.

```
import React from 'react';
import CustomButton from './CustomButton';

function WarningButton() {
```

```
// return React.createElement(CustomButton, {color: 'red'}, null);
return <CustomButton color="red" />;
}
```

JavaScript 번들러를 사용하지 않고 `<script>` 태그를 통해 React를 불러왔다면 `React`는 전역 변수로 존재하기 때문에 별도로 불러올 필요가 없습니다.

JSX 타입을 위한 점 표기법 사용

JSX 내에서도 점 표기법을 사용하여 React 컴포넌트를 참조할 수 있습니다. 이 방법은 하나의 모듈에서 복수의 React 컴포넌트들을 `export` 하는 경우에 편리하게 사용할 수 있습니다. 예를 들어, `MyComponents.DatePicker`가 컴포넌트라면, 아래와 같은 방법으로 직접 사용할 수 있습니다.

```
import React from 'react';

const MyComponents = {
  DatePicker: function DatePicker(props) {
    return <div>Imagine a {props.color} datepicker here.</div>;
  }
}

function BlueDatePicker() {
  return <MyComponents.DatePicker color="blue" />;
}
```

사용자 정의 컴포넌트는 반드시 대문자로 시작해야합니다

`Element`가 소문자로 시작하는 경우에는 `<div>` 나 `` 같은 내장 컴포넌트라는 것을 뜻하며 `'div'` 나 `'span'` 같은 문자열 형태로 `React.createElement`에 전달됩니다. `<foo />` 와 같이 대문자로 시작하는 타입들은 `React.createElement(Foo)`의 형태로 컴파일 되며 JavaScript 파일 내에 사용자가 정의했거나 `import` 한 컴포넌트를 가리킵니다.

컴포넌트의 이름은 대문자로 시작하는 것을 추천합니다. 소문자로 시작하는 컴포넌트를 사용해야 한다면, 대문자로 시작하는 변수에 할당한 뒤 JSX에서 이 변수를 사용하세요.

예를 들어 아래의 코드는 예상대로 실행되지 않을 것입니다.

```
import React from 'react';

// 잘못된 사용법입니다! 아래는 컴포넌트이므로 대문자화 해야 합니다.
function hello(props) {
  // 올바른 사용법입니다! 아래의 <div> 사용법은 유효한 HTML 태그이기 때문에 유효합니다.
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // 잘못된 사용법입니다! React는 <hello />가 대문자가 아니기 때문에 HTML 태그로 인식하게 됩니다.
  return <hello toWhat="World" />;
}
```

이를 고치기 위해 우리는 `hello` 를 `Hello`로 바꾸고 이를 참조할 때 `<Hello />` 를 사용할 것 입니다.

```
import React from 'react';

// 올바른 사용법입니다. 아래는 컴포넌트이므로 대문자로 시작해야 합니다.
```

```

function Hello(props) {
  // 올바른 사용법입니다! 아래의 <div> 사용법은 유효한 HTML 태그이기 때문에 유효합니다.
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // 올바른 사용법입니다! React는 <Hello />가 대문자로 시작하기 때문에 컴포넌트로 인식합니다.
  return <Hello toWhat="World" />;
}

```

실행 중에 타입 선택하기

React element 타입에 일반적인 표현식은 사용할 수 없습니다. element 타입을 지정할 때 일반적인 표현식을 사용하고자 한다면 대문자로 시작하는 변수에 배정한 후 사용할 수 있습니다. 예를 들어 아래와 같이 prop에 따라 다른 컴포넌트를 render 해야 하는 경우들이 종종 있습니다.

```

import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // 잘못된 사용법입니다! JSX 타입은 표현식으로 사용할 수 없습니다.
  return <components[props.storyType] story={props.story} />;
}

```

이를 고치기 위해 우선 타입을 대문자로 시작하는 변수에 지정합니다.

```

import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // 올바른 사용법입니다! 대문자로 시작하는 변수는 JSX 타입으로 사용할 수 있습니다.
  const SpecificStory = components[props.storyType];
  return <SpecificStory story={props.story} />;
}

```

JSX 안에서의 prop 사용

JSX 안에서 prop을 사용하는 방법은 여러 가지가 있습니다.

JavaScript Expressions as Props

아래의 예시와 같이 JavaScript 표현을 `{ }` 안에 넣어서 JSX 안에서 prop으로 사용할 수 있습니다.

```
<MyComponent foo={1 + 2 + 3 + 4} />
```

MyComponent 의 `props.foo`의 값은 `1 + 2 + 3 + 4`의 표현식이 계산되기 때문에 `10`입니다.

`if` 구문과 `for` 루프는 JavaScript 표현식이 아니기 때문에 JSX 안에서 그대로 사용할 수 없습니다. 하지만 아래의 예시와 같이 JSX 밖의 주변 코드에서 사용할 수 있습니다.

```
function NumberDescriber(props) {
  let description;
  if (props.number % 2 == 0) {
    description = <strong>even</strong>;
  } else {
    description = <i>odd</i>;
  }
  return <div>{props.number} is an {description} number</div>;
}
```

더 자세한 [조건부 렌더링](#) 과 [리스트와 Key](#) 관련 문서를 참고해주세요.

문자열 리터럴

문자열 리터럴은 prop으로 넘겨줄 수 있습니다. 아래의 두 JSX 표현은 동일한 표현입니다.

```
<MyComponent message="hello world" />
<MyComponent message={['hello world']} />
```

문자열 리터럴을 넘겨줄 때, 그 값은 HTML 이스케이프 처리가 되지 않습니다. 따라서 아래의 두 JSX 표현은 동일한 표현입니다.

```
<MyComponent message=&lt;3&gt; />
<MyComponent message={['<3'']} />
```

이 동작은 보통 신경 쓰지 않아도 되지만, 문서의 완전성을 위해서 언급해둡니다.

Props의 기본값은 "True"

Prop에 어떤 값도 넘기지 않을 경우, 기본값은 `true`입니다. 아래의 두 JSX 표현은 동일한 표현입니다.

```
<MyTextBox autocomplete />
<MyTextBox autocomplete={true} />
```

일반적으로 prop에 대한 값을 전달하지 않는 것을 권장하지 않는데 이는 [ES6 object shorthand](#) 와 헷갈릴 수 있기 때문입니다.

`{foo}` 는 `{foo: true}` 가 아닌 `{foo: foo}` 와 동일합니다. 이는 HTML 동작 방식과 일치하기 위해 남겨두었습니다.

속성 편집기

`props`에 해당하는 객체를 이미 가지고 있다면, `...` 를 "전개" 연산자로 사용해 전체 객체를 그대로 넘겨줄 수 있습니다. 아래의 두 컴포넌트는 동일합니다.

```
function App1() {
  return <Greeting firstName="Ben" lastName="Hector" />;
}

function App2() {
```

```
const props = {firstName: 'Ben', lastName: 'Hector'};
return <Greeting {...props} />;
}
```

컴포넌트가 사용하게 될 특정 prop을 선택하고 나머지 prop은 전개 연산자를 통해 넘길 수 있습니다.

```
const Button = props => {
  const { kind, ...other } = props;
  const className = kind === "primary" ? "PrimaryButton" : "SecondaryButton";
  return <button className={className} {...other} />;
};

const App = () => {
  return (
    <div>
      <Button kind="primary" onClick={() => console.log("clicked!")}>
        Hello World!
      </Button>
    </div>
  );
};
```

위의 예시의 `kind` prop은 소비되고 DOM의 `button` element에 넘겨지지 않습니다. 다른 모든 prop은 `...other` 객체를 통해서 넘겨지며 이 컴포넌트를 유연하게 만들어줍니다. `onClick`과 `children` prop으로 넘겨지는 것을 볼 수 있습니다.

전개 연산자는 유용하지만 불필요한 prop을 컴포넌트에 넘기거나 유효하지 않은 HTML 속성들을 DOM에 넘기기도 합니다. 꼭 필요할 때만 사용하는 것을 권장합니다.

JSX에서 자식 다루기

여는 태그와 닫는 태그가 있는 JSX 표현에서 두 태그 사이의 내용은 `props.children`이라는 특수한 prop으로 넘겨집니다. 자식을 넘기는 방법은 여러 가지가 있습니다.

문자열 리터럴

여는 태그와 닫는 태그 사이에 문자열 리터럴을 넣을 수 있고 이 때 `props.children` 은 그 문자열이 됩니다. 이는 아래의 예시와 같이 많은 HTML 내장 element에 유용합니다.

```
<MyComponent>Hello world!</MyComponent>
```

이는 유효한 JSX입니다. 여기서 `MyComponent` 의 `props.children` 은 "Hello world!" 입니다. HTML은 이스케이프 처리가 되지 않으며, 일반적으로 아래와 같이 HTML을 쓰는 방식으로 JSX를 쓸 수 있습니다.

```
<div>This is valid HTML &amp; JSX at the same time.</div>
```

JSX는 각 줄의 처음과 끝에 있는 공백을 제거합니다. 빈 줄 역시 제거합니다. 태그에 붙어있는 개행도 제거되며 문자열 리터럴 중간에 있는 개행은 한 개의 공백으로 대체됩니다. 따라서 아래의 예시들은 전부 똑같이 렌더링됩니다.

```
<div>Hello World</div>

<div>
  Hello World
</div>
```

```
<div>
  Hello
  World
</div>

<div>
  Hello World
</div>
```

JSX를 자식으로 사용하기

JSX element를 자식으로 넘겨 줄 수 있습니다. 이는 중첩된 컴포넌트를 보여줄 때 유용합니다.

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

다양한 타입의 자식들을 섞어서 사용할 수 있습니다. 그래서 문자열 리터럴을 JSX 자식과 함께 사용할 수 있습니다. 이는 JSX를 HTML과 같은 방식으로 구동되는 점 중 하나입니다. 아래의 예시는 JSX와 HTML 모두에서 유효합니다.

```
<div>
  Here is a list:
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

React 컴포넌트는 element로 이루어진 배열을 반환할 수 있습니다.

```
render() {
  // 리스트 아이템들을 추가적인 엘리먼트로 둘러쌀 필요 없습니다!
  return [
    // key 지정을 잊지 마세요 :)
    <li key="A">First item</li>,
    <li key="B">Second item</li>,
    <li key="C">Third item</li>,
  ];
}
```

JavaScript 표현식을 자식으로 사용하기

{ }에 감싸서 JavaScript 표현식도 자식으로 넘길 수 있습니다. 아래의 예시들은 동일한 표현입니다.

```
<MyComponent>foo</MyComponent>

<MyComponent>{'foo'}</MyComponent>
```

이는 임의의 길이를 가진 JSX 표현식의 배열을 랜더링 할 때 종종 유용하게 사용됩니다. 아래의 예시는 HTML 배열로 랜더됩니다.

```
function Item(props) {
  return <li>{props.message}</li>;
}

function TodoList() {
```

```
const todos = ['finish doc', 'submit pr', 'nag dan to review'];
return (
  <ul>
    {todos.map((message) => <Item key={message} message={message} />)}
  </ul>
);
}
```

JavaScript 표현식은 다른 타입의 자식과 같이 쓸 수 있습니다. 이는 문자열 템플릿을 대신해서 종종 유용합니다.

```
function Hello(props) {
  return <div>Hello {props.addressee}!</div>;
}
```

함수를 자식으로 사용하기

보통 JSX에 삽입된 JavaScript 표현식은 문자열, React element 혹은 이들의 배열로 환산됩니다. 하지만 `props.children` 은 다른 prop들과 마찬가지로 React가 렌더링 할 수 있는 데이터의 형태뿐만 아니라 어떤 형태의 데이터도 넘겨질 수 있습니다. 아래의 예시와 같이 직접 만든 컴포넌트가 있다면 `props.children` 을 통해서 콜백을 넘겨받을 수 있습니다.

```
// 자식 콜백인 numTimes를 호출하여 반복되는 컴포넌트를 생성합니다.
function Repeat(props) {
  let items = [];
  for (let i = 0; i < props.numTimes; i++) {
    items.push(props.children(i));
  }
  return <div>{items}</div>;
}

function ListOfTenThings() {
  return (
    <Repeat numTimes={10}>
      {(index) => <div key={index}>This is item {index} in the list</div>}
    </Repeat>
  );
}
```

직접 만든 컴포넌트에 넘겨지는 자식들은 렌더되기 전에 React가 이해할 수 있는 형태로 변환된다면 어떤 것인든 넘겨질 수 있습니다. 이런 사용법은 일반적이지 않지만, JSX의 기능의 확장성을 확인하고 싶다면 사용할 수 있습니다.

boolean, null, undefined는 무시됩니다.

`false`, `null`, `undefined` 와 `true` 는 유효한 자식입니다. 그저 렌더링 되지 않을 뿐입니다. 아래의 JSX 표현식들은 동일하게 렌더링됩니다.

```
<div />
<div></div>
<div>{false}</div>
<div>{null}</div>
<div>{undefined}</div>
<div>{true}</div>
```

이는 React element들을 조건부 렌더링할 때 유용합니다. 아래의 JSX는 `showHeader` 가 `true` 일 때 동일하게 `<Header />` 를 렌더하게 됩니다.

```
<div>
  {showHeader && <Header />}
  <Content />
</div>
```

한 가지 주의해야 할 점은 `0` 과 같은 `"falsy"` 값들은 React가 렌더링 한다는 점입니다. 예를 들어, 아래의 예시는 `props.messages` 가 빈 배열일 때 예상과는 다르게 0을 출력하게 됩니다.

```
<div>
  {props.messages.length &&
   <MessageList messages={props.messages} />
  }
</div>
```

이를 고치려면 `&&` 앞의 표현식이 언제나 진리값이 되도록 해야합니다.

```
<div>
  {props.messages.length > 0 &&
   <MessageList messages={props.messages} />
  }
</div>
```

반대로 `false`, `true`, `null` 또는 `undefined`와 같은 값을 출력하고 싶다면 먼저 [문자열로 전환](#) 해야합니다.

```
<div>
  My JavaScript variable is {String(myVariable)}.
</div>
```

Legacy Context

Note:

The legacy context API will be removed in a future major version. Use the [new context API](#) introduced with version 16.3. The legacy API will continue working for all 16.x releases.

How To Use Context

This section documents a legacy API. See the [new API](#).

Suppose you have a structure like:

```
class Button extends React.Component {
  render() {
    return (
      <button style={{background: this.props.color}}>
        {this.props.children}
      </button>
    );
  }
}

class Message extends React.Component {
  render() {
    return (
      <div>
        {this.props.text} <Button color={this.props.color}>Delete</Button>
      </div>
    );
  }
}

class MessageList extends React.Component {
  render() {
    const color = "purple";
    const children = this.props.messages.map((message) =>
      <Message text={message.text} color={color} />
    );
    return <div>{children}</div>;
  }
}
```

In this example, we manually thread through a `color` prop in order to style the `Button` and `Message` components appropriately. Using context, we can pass this through the tree automatically:

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

class Button extends React.Component {
```

```

render() {
  return (
    <button style={{background: this.context.color}}>
      {this.props.children}
    </button>
  );
}

Button.contextTypes = {
  color: PropTypes.string
};

class Message extends React.Component {
  render() {
    return (
      <div>
        {this.props.text} <Button>Delete</Button>
      </div>
    );
  }
}

class MessageList extends React.Component {
  getChildContext() {
    return {color: "purple"};
  }

  render() {
    const children = this.props.messages.map((message) =>
      <Message text={message.text} />
    );
    return <div>{children}</div>;
  }
}

MessageList.childContextTypes = {
  color: PropTypes.string
};

```

By adding `childContextTypes` and `getChildContext` to `MessageList` (the context provider), React passes the information down automatically and any component in the subtree (in this case, `Button`) can access it by defining `contextTypes`.

If `contextTypes` is not defined, then `context` will be an empty object.

Note:

`React.PropTypes` has moved into a different package since React v15.5. Please use [the prop-type library instead](#) to define `contextTypes`.

We provide a [codemod script](#) to automate the conversion.

Parent-Child Coupling

This section documents a legacy API. See the [new API](#).

Context can also let you build an API where parents and children communicate. For example, one library that works this way is [React Router V4](#):

```
import { BrowserRouter as Router, Route, Link } from 'react-router-dom';

const BasicExample = () => (
  <Router>
    <div>
      <ul>
        <li><Link to="/">Home</Link></li>
        <li><Link to="/about">About</Link></li>
        <li><Link to="/topics">Topics</Link></li>
      </ul>

      <hr />

      <Route exact path="/" component={Home} />
      <Route path="/about" component={About} />
      <Route path="/topics" component={Topics} />
    </div>
  </Router>
);
```

By passing down some information from the `Router` component, each `Link` and `Route` can communicate back to the containing `Router`.

Before you build components with an API similar to this, consider if there are cleaner alternatives. For example, you can pass entire React components as props if you'd like to.

Referencing Context in Lifecycle Methods

This section documents a legacy API. See the [new API](#).

If `contextTypes` is defined within a component, the following [lifecycle methods](#) will receive an additional parameter, the `context` object:

- `constructor(props, context)`
- `componentWillReceiveProps(nextProps, nextContext)`
- `shouldComponentUpdate(nextProps, nextState, nextContext)`
- `componentWillUpdate(nextProps, nextState, nextContext)`

Note:

As of React 16, `componentDidUpdate` no longer receives `prevContext`.

Referencing Context in Function Components

This section documents a legacy API. See the [new API](#).

Function components are also able to reference `context` if `contextTypes` is defined as a property of the function. The following code shows a `Button` component written as a function component.

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

const Button = ({children}, context) =>
  <button style={{background: context.color}}>
    {children}
  </button>

Button.contextTypes = {color: PropTypes.string};
```

Updating Context

This section documents a legacy API. See the [new API](#).

Don't do it.

React has an API to update context, but it is fundamentally broken and you should not use it.

The `getChildContext` function will be called when the state or props changes. In order to update data in the context, trigger a local state update with `this.setState`. This will trigger a new context and changes will be received by the children.

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

class MediaQuery extends React.Component {
  constructor(props) {
    super(props);
    this.state = {type: 'desktop'};
  }

  getChildContext() {
    return {type: this.state.type};
  }

  componentDidMount() {
    const checkMediaQuery = () => {
      const type = window.matchMedia("(min-width: 1025px)").matches ? 'desktop' : 'mobile';
      if (type !== this.state.type) {
        this.setState({type});
      }
    };
    checkMediaQuery();
    window.addEventListener('resize', checkMediaQuery);
  }
}
```

```
};

window.addEventListener('resize', checkMediaQuery);
checkMediaQuery();
}

render() {
  return this.props.children;
}

MediaQuery.childContextTypes = {
  type: PropTypes.string
};
```

The problem is, if a context value provided by component changes, descendants that use that value won't update if an intermediate parent returns `false` from `shouldComponentUpdate`. This is totally out of control of the components using context, so there's basically no way to reliably update the context. [This blog post](#) has a good explanation of why this is a problem and how you might get around it.

Event Pooling

Note

This page is only relevant for React 16 and earlier, and for React Native.

React 17 on the web **does not** use event pooling.

[Read more](#) about this change in React 17.

The `SyntheticEvent` objects are pooled. This means that the `SyntheticEvent` object will be reused and all properties will be nullified after the event handler has been called. For example, this won't work:

```
function handleChange(e) {
  // This won't work because the event object gets reused.
  setTimeout(() => {
    console.log(e.target.value); // Too late!
  }, 100);
}
```

If you need to access event object's properties after the event handler has run, you need to call `e.persist()`:

```
function handleChange(e) {
  // Prevents React from resetting its properties:
  e.persist();

  setTimeout(() => {
    console.log(e.target.value); // Works
  }, 100);
}
```

[Go to TOC](#)

State 끌어올리기

종종 동일한 데이터에 대한 변경사항을 여러 컴포넌트에 반영해야 할 필요가 있습니다. 이럴 때는 가장 가까운 공통 조상으로 state를 끌어올리는 것이 좋습니다. 이런 일을 어떻게 할 수 있을지 지금부터 살펴봅시다.

이번 섹션에서는 주어진 온도에서 물의 끓는 여부를 추정하는 온도 계산기를 만들어볼 것입니다.

먼저 `BoilingVerdict`라는 이름의 컴포넌트부터 만들어봅시다. 이 컴포넌트는 섭씨온도를 의미하는 `celsius` prop를 받아서 이 온도가 물이 끓기에 충분한지 여부를 출력합니다.

```
function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>The water would boil.</p>;
  }
  return <p>The water would not boil.</p>;
}
```

그 다음으로 `Calculator`라는 컴포넌트를 만들어보겠습니다. 이 컴포넌트는 온도를 입력할 수 있는 `<input>`을 렌더링하고 그 값을 `this.state.temperature`에 저장합니다.

또한 현재 입력값에 대한 `BoilingVerdict` 컴포넌트를 렌더링합니다.

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

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

  render() {
    const temperature = this.state.temperature;
    return (
      <fieldset>
        <legend>Enter temperature in Celsius:</legend>
        <input
          value={temperature}
          onChange={this.handleChange} />
        <BoilingVerdict
          celsius={parseFloat(temperature)} />
      </fieldset>
    );
  }
}
```

[CodePen에서 실행하기](#)

두 번째 Input 추가하기

새 요구사항으로써 섭씨 입력 필드뿐만 아니라 화씨 입력 필드를 추가하고 두 필드 간에 동기화 상태를 유지하도록 해보겠습니다.

`Calculator`에서 `TemperatureInput` 컴포넌트를 빼내는 작업부터 시작해봅시다. 또한 "c" 또는 "f"의 값을 가질 수 있는 `scale` prop를 추가할 것입니다.

```
const scaleNames = {
  c: 'Celsius',
  f: 'Fahrenheit'
};

class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

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

  render() {
    const temperature = this.state.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Enter temperature in {scaleNames[scale]}:</legend>
        <input value={temperature}>
          onChange={this.handleChange}
        </input>
      </fieldset>
    );
  }
}
```

이제 `Calculator`가 분리된 두 개의 온도 입력 필드를 렌더링하도록 변경할 수 있습니다.

```
class Calculator extends React.Component {
  render() {
    return (
      <div>
        <TemperatureInput scale="c" />
        <TemperatureInput scale="f" />
      </div>
    );
  }
}
```

[CodePen에서 실행하기](#)

이제 두 개의 입력 필드를 갖게 되었습니다. 그러나 둘 중 하나에 온도를 입력하더라도 다른 하나는 갱신되지 않는 문제가 있습니다. 이것은 두 입력 필드 간에 동기화 상태를 유지하고자 했던 원래 요구사항과는 맞지 않습니다.

또한 `Calculator`에서 `BoilingVerdict`도 역시 보여줄 수 없는 상황입니다. 현재 입력된 온도 정보가 `TemperatureInput` 안에 숨겨져 있으므로 `Calculator`는 그 값을 알 수 없기 때문입니다.

변환 함수 작성하기

먼저, 섭씨를 화씨로, 또는 그 반대로 변환해주는 함수를 작성해보겠습니다.

```

function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9;
}

function toFahrenheit(celsius) {
  return (celsius * 9 / 5) + 32;
}

```

이 두 함수는 숫자를 변환합니다. 이제 `temperature` 문자열과 변환 함수를 인수로 취해서 문자열을 반환하는 또 다른 함수를 작성해 보겠습니다. 그리고 그것을 한 입력값에 기반해 나머지 입력값을 계산하는 용도로 사용할 것입니다.

이 함수는 올바르지 않은 `temperature` 값에 대해서는 빈 문자열을 반환하고 값을 소수점 세 번째 자리로 반올림하여 출력합니다.

```

function tryConvert(temperature, convert) {
  const input = parseFloat(temperature);
  if (Number.isNaN(input)) {
    return '';
  }
  const output = convert(input);
  const rounded = Math.round(output * 1000) / 1000;
  return rounded.toString();
}

```

예를 들어 `tryConvert('abc', toCelsius)` 는 빈 문자열을 반환하고 `tryConvert('10.22', toFahrenheit)` 는 '`50.396`' 을 반환합니다.

State 끌어올리기

현재는 두 `TemperatureInput` 컴포넌트가 각각의 입력값을 각자의 `state`에 독립적으로 저장하고 있습니다.

```

class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

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

  render() {
    const temperature = this.state.temperature;
    // ...
  }
}

```

그러나 우리는 두 입력값이 서로의 것과 동기화된 상태로 있길 원합니다. 섭씨온도 입력값을 변경할 경우 화씨온도 입력값 역시 변환된 온도를 반영할 수 있어야 하며, 그 반대의 경우에도 마찬가지여야 합니다.

React에서 `state`를 공유하는 일은 그 값을 필요로 하는 컴포넌트 간의 가장 가까운 공통 조상으로 `state`를 끌어올림으로써 이뤄낼 수 있습니다. 이렇게 하는 방법을 "state 끌어올리기"라고 부릅니다. 이제 `TemperatureInput` 이 개별적으로 가지고 있던 지역 `state`를 지우는 대신 `Calculator` 로 그 값을 옮겨놓을 것입니다.

`Calculator` 가 공유될 `state`를 소유하고 있으면 이 컴포넌트는 두 입력 필드의 현재 온도에 대한 "진리의 원천(source of truth)"이 됩니다. 이를 통해 두 입력 필드가 서로 간에 일관된 값을 유지하도록 만들 수 있습니다. 두 `TemperatureInput` 컴포넌트의 `props`가 같은 부모인 `Calculator`로부터 전달되기 때문에, 두 입력 필드는 항상 동기화된 상태를 유지할 수 있게 됩니다.

어떻게 동작하는지 차근차근 살펴봅시다.

우선, `TemperatureInput` 컴포넌트에서 `this.state.temperature` 를 `this.props.temperature` 로 대체할 것입니다. 지금은 `this.props.temperature` 가 이미 존재한다고 가정해봅시다. 나중에는 이 값을 `Calculator`로부터 건네야 할 것입니다.

```
render() {
  // Before: const temperature = this.state.temperature;
  const temperature = this.props.temperature;
  // ...
```

`props`는 읽기 전용입니다. `temperature` 가 지역 `state`였을 때는 그 값을 변경하기 위해서 그저 `TemperatureInput` 의 `this.setState()` 를 호출하는 걸로 충분했습니다. 그러나 이제 `temperature` 가 부모로부터 `prop`로 전달되기 때문에 `TemperatureInput` 은 그 값을 제어할 능력이 없습니다.

React에서는 보통 이 문제를 컴포넌트를 "제어" 가능하게 만드는 방식으로 해결합니다. DOM `<input>` 이 `value` 와 `onChange` `prop`를 건네받는 것과 비슷한 방식으로, 사용자 정의된 `TemperatureInput` 역시 `temperature` 와 `onTemperatureChange` `props`를 자신의 부모인 `Calculator`로부터 건네받을 수 있습니다.

이제 `TemperatureInput` 에서 온도를 갱신하고 싶으면 `this.props.onTemperatureChange` 를 호출하면 됩니다.

```
handleChange(e) {
  // Before: this.setState({temperature: e.target.value});
  this.props.onTemperatureChange(e.target.value);
  // ...
```

주의

사용자 정의 컴포넌트에서 `temperature` 와 `onTemperatureChange` `prop`의 이름이 특별한 의미를 갖진 않습니다. 일관된 컨벤션으로 `value` 와 `onChange` 를 사용할 수도 있으며, 여러분이 원하는 그 어떤 이름이든지 사용할 수 있습니다.

`onTemperatureChange` `prop`는 부모 컴포넌트인 `Calculator`로부터 `temperature` `prop`와 함께 제공될 것입니다. 이를 이용해 자신의 지역 `state`를 수정해서 변경사항을 처리하므로, 변경된 새 값을 전달받은 두 입력 필드는 모두 리렌더링될 것입니다. `Calculator` 의 새로운 구현체는 조금 뒤에 살펴보겠습니다.

`Calculator` 의 변경사항을 들여다보기 전에 `TemperatureInput` 컴포넌트에 대한 변경사항부터 요약해보겠습니다. 이 컴포넌트의 지역 `state`를 제거했으며 `this.state.temperature` 대신에 `this.props.temperature` 를 읽어오도록 변경했습니다. `state`를 변경하고 싶을 경우 `this.setState()` 대신에 `Calculator`로부터 건네받은 `this.props.onTemperatureChange()` 를 호출하도록 만들었습니다.

```
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.props.onTemperatureChange(e.target.value);
  }
```

```

render() {
  const temperature = this.props.temperature;
  const scale = this.props.scale;
  return (
    <fieldset>
      <legend>Enter temperature in {scaleNames[scale]}:</legend>
      <input value={temperature}
             onChange={this.handleChange} />
    </fieldset>
  );
}

```

이제 다시 `Calculator` 컴포넌트로 와 봅시다.

`temperature` 와 `scale`의 현재 입력값을 이 컴포넌트의 지역 state에 저장합니다. 이것은 우리가 입력 필드들로부터 "끌어올린" state이며 그들에 대한 "진리의 원천(source of truth)"으로 작용할 것입니다. 또한 두 입력 필드를 렌더링하기 위해서 알아야 하는 모든 데이터를 최소한으로 표현한 것이기도 합니다.

예를 들어서 섭씨 입력 필드에 37을 입력하면 `Calculator` 컴포넌트의 state는 다음과 같을 것입니다.

```
{
  temperature: '37',
  scale: 'c'
}
```

이후에 화씨 입력 필드의 값을 212로 수정하면 `Calculator` 의 state는 다음과 같은 모습일 것입니다.

```
{
  temperature: '212',
  scale: 'f'
}
```

두 입력 필드에 모두 값을 저장하는 일도 가능했지만 결국은 불필요한 작업이었던 것입니다. 가장 최근에 변경된 입력값과 그 값이 나타내는 단위를 저장하는 것만으로도 충분합니다. 그리고 나면 현재의 `temperature` 와 `scale` 에 기반해 다른 입력 필드의 값을 추론할 수 있습니다.

두 입력 필드의 값이 동일한 state로부터 계산되기 때문에 이 둘은 항상 동기화된 상태를 유지하게 됩니다.

```

class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleChangeCelsius = this.handleChangeCelsius.bind(this);
    this.handleChangeFahrenheit = this.handleChangeFahrenheit.bind(this);
    this.state = {temperature: '', scale: 'c'};
  }

  handleChangeCelsius(temperature) {
    this.setState({scale: 'c', temperature});
  }

  handleChangeFahrenheit(temperature) {
    this.setState({scale: 'f', temperature});
  }

  render() {
    const scale = this.state.scale;
  }
}

```

```

const temperature = this.state.temperature;
const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) :
temperature;
const fahrenheit = scale === 'c' ? tryConvert(temperature, toFahrenheit) :
temperature;

return (
  <div>
    <TemperatureInput
      scale="c"
      temperature={celsius}
      onTemperatureChange={this.handleCelsiusChange} />
    <TemperatureInput
      scale="f"
      temperature={fahrenheit}
      onTemperatureChange={this.handleFahrenheitChange} />
    <BoilingVerdict
      celsius={parseFloat(celsius)} />
  </div>
);
}
}

```

[CodePen에서 실행하기](#)

이제 어떤 입력 필드를 수정하든 간에 `Calculator`의 `this.state.temperature`와 `this.state.scale`이 갱신됩니다. 입력 필드 중 하나는 있는 그대로의 값을 받으므로 사용자가 입력한 값이 보존되고, 다른 입력 필드의 값은 항상 다른 하나에 기반해 재계산됩니다.

입력값을 변경할 때 일어나는 일들을 정리해보겠습니다.

- React는 DOM `<input>`의 `onChange`에 지정된 함수를 호출합니다. 위 예시의 경우 `TemperatureInput`의 `handleChange` 메서드에 해당합니다.
- `TemperatureInput` 컴포넌트의 `handleChange` 메서드는 새로 입력된 값과 함께 `this.props.onTemperatureChange()`를 호출합니다. `onTemperatureChange`를 포함한 이 컴포넌트의 `props`는 부모 컴포넌트인 `Calculator`로부터 제공받은 것입니다.
- 이전 렌더링 단계에서, `Calculator`는 섭씨 `TemperatureInput`의 `onTemperatureChange`를 `Calculator`의 `handleCelsiusChange` 메서드로, 화씨 `TemperatureInput`의 `onTemperatureChange`를 `Calculator`의 `handleFahrenheitChange` 메서드로 지정해놓았습니다. 따라서 우리가 둘 중에 어떤 입력 필드를 수정하느냐에 따라서 `Calculator`의 두 메서드 중 하나가 호출됩니다.
- 이들 메서드는 내부적으로 `Calculator` 컴포넌트가 새 입력값, 그리고 현재 수정한 입력 필드의 입력 단위와 함께 `this.setState()`를 호출하게 함으로써 React에게 자신을 다시 렌더링하도록 요청합니다.
- React는 UI가 어떻게 보여야 하는지 알아내기 위해 `Calculator` 컴포넌트의 `render` 메서드를 호출합니다. 두 입력 필드의 값은 현재 온도와 활성화된 단위를 기반으로 재계산됩니다. 온도의 변환이 이 단계에서 수행됩니다.
- React는 `Calculator`가 전달한 새 `props`와 함께 각 `TemperatureInput` 컴포넌트의 `render` 메서드를 호출합니다. 그러면서 UI가 어떻게 보여야 할지를 파악합니다.
- React는 `BoilingVerdict` 컴포넌트에게 섭씨온도를 `props`로 건네면서 그 컴포넌트의 `render` 메서드를 호출합니다.
- React DOM은 물의 끓는 여부와 올바른 입력값을 일치시키는 작업과 함께 DOM을 갱신합니다. 값을 변경한 입력 필드는 현재 입력 값을 그대로 받고, 다른 입력 필드는 변환된 온도 값으로 갱신됩니다.

입력 필드의 값을 변경할 때마다 동일한 절차를 거치고 두 입력 필드는 동기화된 상태로 유지됩니다.

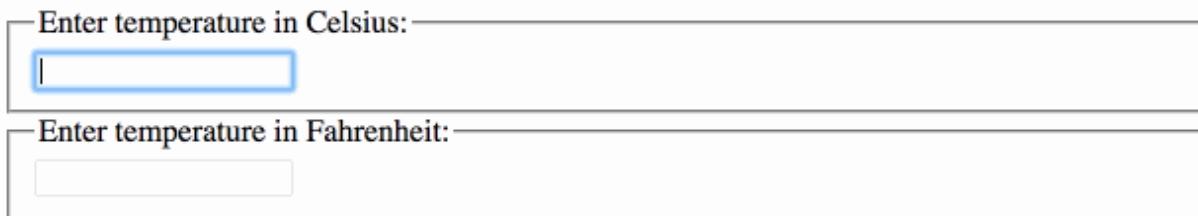
교훈

React 애플리케이션 안에서 변경이 일어나는 데이터에 대해서는 "진리의 원천(source of truth)"을 하나만 두어야 합니다. 보통의 경우, state는 렌더링에 그 값을 필요로 하는 컴포넌트에 먼저 추가됩니다. 그리고 나서 다른 컴포넌트도 역시 그 값이 필요하게 되면 그 값을 그들의 가장 가까운 공통 조상으로 끌어올리면 됩니다. 다른 컴포넌트 간에 존재하는 state를 동기화시키려고 노력하는 대신 [하향식 데이터 흐름](#)에 기대는 걸 추천합니다.

state를 끌어올리는 작업은 양방향 바인딩 접근 방식보다 더 많은 "보일러 플레이트" 코드를 유발하지만, 버그를 찾고 격리하기 더 쉽게 만든다는 장점이 있습니다. 어떤 state든 간에 특정 컴포넌트 안에서 존재하기 마련이고 그 컴포넌트가 자신의 state를 스스로 변경할 수 있으므로 버그가 존재할 수 있는 범위가 크게 줄어듭니다. 또한 사용자의 입력을 거부하거나 변형하는 자체 로직을 구현할 수도 있습니다.

어떤 값이 props 또는 state로부터 계산될 수 있다면, 아마도 그 값을 state에 두어서는 안 됩니다. 예를 들어 `celsiusValue` 와 `fahrenheitValue` 를 둘 다 저장하는 대신, 단지 최근에 변경된 `temperature` 와 `scale` 만 저장하면 됩니다. 다른 입력 필드의 값은 항상 그 값들에 기반해서 `render()` 메서드 안에서 계산될 수 있습니다. 이를 통해 사용자 입력값의 정밀도를 유지한 채 다른 필드의 입력값에 반올림을 지우거나 적용할 수 있게 됩니다.

UI에서 무언가 잘못된 부분이 있을 경우, [React Developer Tools](#)를 이용하여 props를 검사하고 state를 간신히 책임이 있는 컴포넌트를 찾을 때까지 트리를 따라 탐색해보세요. 이렇게 함으로써 소스 코드에서 버그를 추적할 수 있습니다.



The screenshot shows the React Developer Tools component tree for the `<Calculator>` component. The tree includes `<TemperatureInput scale="c" temperature="" onTemperatureChange="..."/>`, `<TemperatureInput scale="f" temperature="" onTemperatureChange="..."/>`, and `<BoilingVerdict celsius=null>...</BoilingVerdict>`. The props for the `<Calculator>` component are shown as an empty object. The state is shown with `scale: "c"` and `temperature: ""`.

[Go to TOC](#)

리스트와 Key

먼저 JavaScript에서 리스트를 어떻게 변환하는지 살펴봅시다.

아래는 `map()` 함수를 이용하여 `numbers` 배열의 값을 두배로 만든 후 `map()`에서 반환하는 새 배열을 `doubled` 변수에 할당하고 로그를 확인하는 코드입니다.

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);
console.log(doubled);
```

이 코드는 콘솔에 `[2, 4, 6, 8, 10]`을 출력합니다.

React에서 배열을 [엘리먼트](#) 리스트로 만드는 방식은 이와 거의 동일 합니다.

여러개의 컴포넌트 렌더링 하기

엘리먼트 모음을 만들고 종괄호 `{}`를 이용하여 [JSX](#)에 포함 시킬 수 있습니다.

아래의 JavaScript `map()` 함수를 사용하여 `numbers` 배열을 반복 실행합니다. 각 항목에 대해 `` 엘리먼트를 반환하고 엘리먼트 배열의 결과를 `listItems`에 저장합니다.

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

Then, we can include the entire `listItems` array inside a `` element: 그러면 `` 엘리먼트 안에 전체 `listItems` 배열을 포함할 수 있습니다.

```
<ul>{listItems}</ul>
```

[CodePen](#)에서 실행하기

이 코드는 1부터 5까지의 숫자로 이루어진 리스트를 보여줍니다.

기본 리스트 컴포넌트

일반적으로 [컴포넌트](#) 안에서 리스트를 렌더링합니다.

이전 예시를 `numbers` 배열을 받아서 순서 없는 엘리먼트 리스트를 출력하는 컴포넌트로 리팩토링할 수 있습니다.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li>{number}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}
```

```
const numbers = [1, 2, 3, 4, 5];
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<NumberList numbers={numbers} />);
```

이 코드를 실행하면 리스트의 각 항목에 key를 넣어야 한다는 경고가 표시됩니다. "key"는 엘리먼트 리스트를 만들 때 포함해야 하는 특수한 문자열 어트리뷰트입니다. 다음 섹션에서 key의 중요성에 대해서 더 설명하겠습니다. 이제 `numbers.map()` 안에서 리스트의 각 항목에 `key`를 할당하여 키 누락 문제를 해결하겠습니다.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}
```

[CodePen에서 실행하기](#)

Key

Key는 React가 어떤 항목을 변경, 추가 또는 삭제할지 식별하는 것을 돋습니다. key는 엘리먼트에 안정적인 고유성을 부여하기 위해 배열 내부의 엘리먼트에 지정해야 합니다.

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

Key를 선택하는 가장 좋은 방법은 리스트의 다른 항목들 사이에서 해당 항목을 고유하게 식별할 수 있는 문자열을 사용하는 것입니다. 대부분의 경우 데이터의 ID를 key로 사용합니다.

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}>
    {todo.text}
  </li>
);
```

렌더링 한 항목에 대한 안정적인 ID가 없다면 최후의 수단으로 항목의 인덱스를 key로 사용할 수 있습니다.

```
const todoItems = todos.map((todo, index) =>
  // Only do this if items have no stable IDs
  <li key={index}>
    {todo.text}
  </li>
);
```

항목의 순서가 바뀔 수 있는 경우 `key`에 인덱스를 사용하는 것은 권장하지 않습니다. 이로 인해 성능이 저하되거나 컴포넌트의 `state`와 관련된 문제가 발생할 수 있습니다. Robin Pokorný's가 작성한 글인 [인덱스를 key로 사용할 경우 부정적인 영향에 대한 상세 설명](#)을 참고하시길 바랍니다. 리스트 항목에 명시적으로 `key`를 지정하지 않으면 React는 기본적으로 인덱스를 `key`로 사용합니다.

더 자세히 알고 싶다면 [왜 `key`가 필요한가에 대한 더 자세한 설명](#)을 읽어보세요.

Key로 컴포넌트 추출하기

키는 주변 배열의 `context`에서만 의미가 있습니다.

예를 들어 `ListItem` 컴포넌트를 추출 한 경우 `ListItem` 안에 있는 `` 엘리먼트가 아니라 배열의 `<ListItem />` 엘리먼트가 `key`를 가져야 합니다.

예시: 잘못된 Key 사용법

```
function ListItem(props) {
  const value = props.value;
  return (
    // 틀렸습니다! 여기에는 key를 지정할 필요가 없습니다.
    <li key={value.toString()}>
      {value}
    </li>
  );
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // 틀렸습니다! 여기에 key를 지정해야 합니다.
    <ListItem value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

예시: 올바른 Key 사용법

```
function ListItem(props) {
  // 맞습니다! 여기에는 key를 지정할 필요가 없습니다.
  return <li>{props.value}</li>;
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // 맞습니다! 배열 안에 key를 지정해야 합니다.
    <ListItem key={number.toString()} value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

CodePen에서 실행하기

경험상 `map()` 함수 내부에 있는 엘리먼트에 key를 넣어 주는 게 좋습니다.

Key는 형제 사이에서만 고유한 값이어야 한다.

Key는 배열 안에서 형제 사이에서 고유해야 하고 전체 범위에서 고유할 필요는 없습니다. 두 개의 다른 배열을 만들 때 동일한 key를 사용할 수 있습니다.

```
function Blog(props) {
  const sidebar = (
    <ul>
      {props.posts.map((post) =>
        <li key={post.id}>
          {post.title}
        </li>
      )}
    </ul>
  );
  const content = props.posts.map((post) =>
    <div key={post.id}>
      <h3>{post.title}</h3>
      <p>{post.content}</p>
    </div>
  );
  return (
    <div>
      {sidebar}
      <hr />
      {content}
    </div>
  );
}

const posts = [
];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Blog posts={posts} />);
```

CodePen에서 실행하기

React에서 key는 힌트를 제공하지만 컴포넌트로 전달하지는 않습니다. 컴포넌트에서 key와 동일한 값이 필요하면 다른 이름의 prop으로 명시적으로 전달합니다.

```
const content = posts.map((post) =>
  <Post
    key={post.id}
    id={post.id}
    title={post.title} />
);
```

위 예시에서 `Post` 컴포넌트는 `props.id`를 읽을 수 있지만 `props.key`는 읽을 수 없습니다.

JSX에 map() 포함시키기

위 예시에서 별도의 `listItems` 변수를 선언하고 이를 JSX에 포함했습니다.

```

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <ListItem key={number.toString()} value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}

```

JSX를 사용하면 중괄호 안에 모든 표현식을 포함 시킬 수 있으므로 `map()` 함수의 결과를 인라인으로 처리할 수 있습니다.

```

function NumberList(props) {
  const numbers = props.numbers;
  return (
    <ul>
      {numbers.map((number) =>
        <ListItem key={number.toString()} value={number} />
      )}
    </ul>
  );
}

```

[CodePen에서 실행하기](#)

이 방식을 사용하면 코드가 더 깔끔해 지지만, 이 방식을 낭비하는 것은 좋지 않습니다. JavaScript와 마찬가지로 가독성을 위해 변수로 추출해야 할지 아니면 인라인으로 넣을지는 개발자가 직접 판단해야 합니다. `map()` 함수가 너무 중첩된다면 [컴포넌트로 추출](#) 하는 것이 좋습니다.

[Go to TOC](#)

성능 최적화

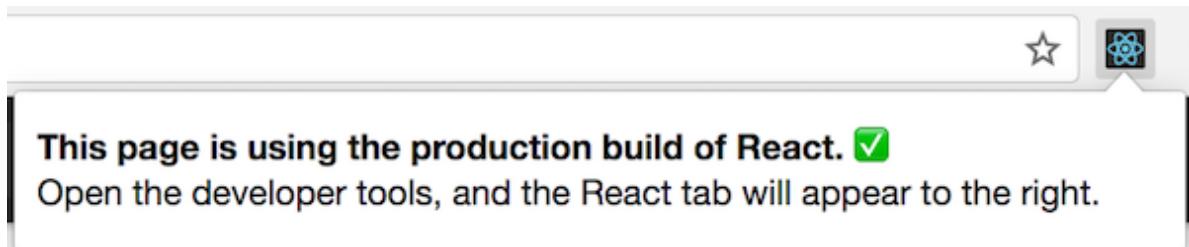
내부적으로 React는 UI를 최신화하기 위해 비용이 많이 드는 DOM 작업의 수를 최소화하기 위해 몇 가지 기발한 방법을 활용합니다. 많은 애플리케이션에서 React를 사용하면 성능을 특별히 최적화하기 위해 많은 작업을 수행하지 않고도 빠른 사용자 인터페이스로 이어질 수 있습니다. 그럼에도 불구하고 React 애플리케이션의 속도를 높일 수 있는 몇 가지 방법이 있습니다.

프로덕션 빌드를 활용하세요

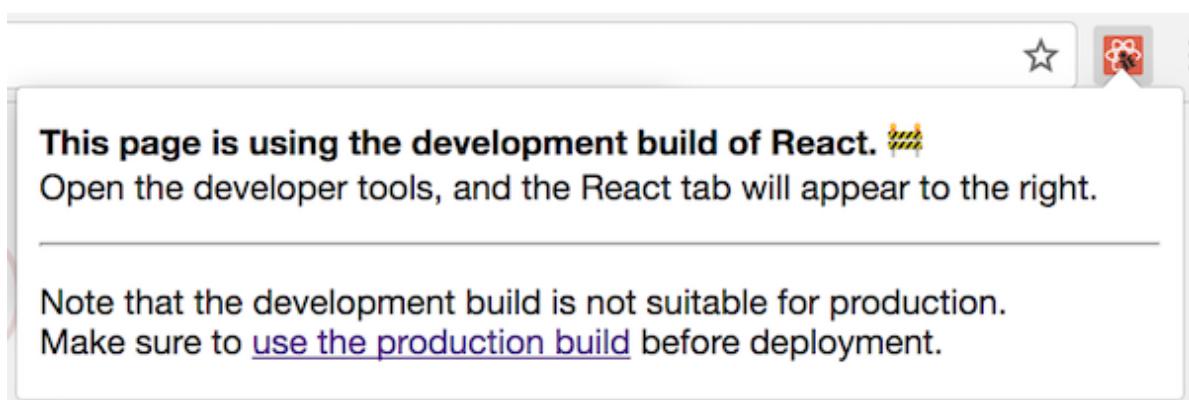
React 앱에서 성능 문제를 겪고 있거나 벤치마크하고 있는 경우, 축소된 프로덕션 빌드를 사용하여 테스트를 수행하고 있는지 확인하세요.

기본적으로 React에는 유용한 경고가 많이 포함되어 있습니다. 이 경고들은 개발하는데 있어 매우 유용합니다. 그러나 그 경고는 React를 더 크고 느리게 만들기 때문에 앱을 배포할 때 프로덕션 버전을 사용해야합니다.

빌드 프로세스가 올바르게 설정되었는지 잘 모르는 경우에는 [React Developer Tools for Chrome](#)를 설치하여 확인할 수 있습니다. 프로덕션 모드의 React 기반 사이트에 접속하면 아이콘의 배경이 어두운 색으로 표시됩니다.



개발 모드의 React 기반 사이트에 접속하면 아이콘의 배경이 빨간색으로 표시됩니다.



앱을 개발할 때는 개발 모드를, 사용자에게 앱을 배포할 때는 프로덕션 모드를 사용해야 합니다.

아래에서 프로덕션 용도의 앱을 제작할 수 있는 방법을 확인할 수 있습니다.

Create React App

프로젝트가 Create React App 기반이라면 아래 명령어를 실행하세요.

npm run build

명령어를 실행하면 프로젝트의 `build/` 폴더에 애플리케이션의 프로덕션 빌드파일이 만들어집니다.

프로덕션 환경에 배포하기 전에만 필요하다는 것을 기억하세요. 정상적인 개발 환경에선 `npm start`를 이용하세요.

단일 파일 빌드

React 및 React DOM의 프로덕션 준비 버전을 단일 파일로 제공합니다.

```
<script src="https://unpkg.com/react@18/umd/react.production.min.js"></script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.production.min.js">
</script>
```

`.production.min.js`로 끝나는 React 파일만이 프로덕션 환경에 적합합니다.

Brunch

가장 효율적인 Brunch 프로덕션 빌드를 위해 `terser-brunch`를 설치하세요.

```
# npm을 사용한다면
npm install --save-dev terser-brunch

# Yarn을 사용한다면
yarn add --dev terser-brunch
```

다음 프로덕션 빌드를 생성하기 위해 `build` 명령어에 `-p` 플래그를 추가합니다.

brunch build -p

프로덕션 빌드에 대해서만 이 작업을 수행하면 됩니다. React의 유용한 경고문구를 숨기고 빌드를 훨씬 느리게 만들기 때문에 개발 환경에서 플러그인을 적용하거나 `-p` 플래그를 추가하지 마세요.

Browserify

가장 효율적인 Browserify 프로덕션 빌드를 위해 몇 가지 플러그인을 설치하세요.

```
# npm을 사용하는 경우
npm install --save-dev envify terser uglifyify

# Yarn을 사용하는 경우
yarn add --dev envify terser uglifyify
```

프로덕션 빌드를 만들려면, 다음 변환을 추가하세요. (**순서는 중요합니다.**)

- `envify` 변환은 올바른 빌드 환경이 설정되도록 합니다. 또한 전역 (`-g`)으로 변환시킵니다.
- `uglifyify` 변환은 개발에서만 사용하는 `package`를 제거합니다. 또한 전역 (`-g`)으로 변환시킵니다.
- 마지막으로 최종 `bundle`은 `mangling`을 위해 `terser`로 연결됩니다. ([원리](#))

예시를 확인하세요.

```
browserify ./index.js \
-g [ envify --NODE_ENV production ] \
-g uglifyify \
| terser --compress --mangle > ./bundle.js
```

프로덕션 빌드에서만 필요한 작업이라는 점을 기억하세요. 이러한 플러그인은 React의 유용한 경고를 숨기고 빌드를 훨씬 느리게 만들기 때문에 개발 중에는 적용하지 마세요.

Rollup

가장 효율적인 Rollup 프로덕션 빌드를 위해 몇 가지 플러그인을 설치하세요.

```
# npm을 사용하는 경우
npm install --save-dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-terser

# Yarn을 사용하는 경우
yarn add --dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-terser
```

프로덕션 빌드를 만들려면, 다음 플러그인을 추가하세요. (순서는 중요합니다.)

- `replace` 플러그인은 올바른 빌드 환경이 설정되도록 해줍니다.
- `commonjs` 플러그인은 CommonJS를 지원하도록 해줍니다.
- `terser` 플러그인은 최종 bundle을 압축하고 mangle 해줍니다.

```
plugins: [
  // ...
  require('rollup-plugin-replace')({
    'process.env.NODE_ENV': JSON.stringify('production')
  }),
  require('rollup-plugin-commonjs')(),
  require('rollup-plugin-terser')(),
  // ...
]
```

전체적인 설정 예시는 [gist](#)를 참고하세요.

프로덕션 빌드에서만 필요한 작업이라는 점을 기억하세요. React의 유용한 경고를 숨기고 빌드를 훨씬 느리게 만들기 때문에 `terser` 플러그인이나 `replace` 플러그인을 개발 중에 '`'production'`' 값으로 적용하지 마세요.

webpack

주의

Create React App을 사용한다면 [위 설명](#)을 참고하세요.

이 부분은 webpack을 직접 구성할 경우에만 해당합니다.

Webpack v4 이상에서는 프로덕션 모드에서 기본적으로 코드를 축소합니다.

```
const TerserPlugin = require('terser-webpack-plugin');
module.exports = {
```

```

    mode: 'production',
    optimization: {
      minimizer: [new TerserPlugin({ /* additional options here */ })],
    },
  };

```

이 부분에 대해 더 알고 싶다면 [webpack 문서](#)를 참고하세요.

production 빌드에서만 필요한 작업이라는 점을 기억하세요. React의 유용한 경고를 숨기고 빌드를 훨씬 느리게 만들기 때문에 `TerserPlugin`을 개발 중에 적용하지 마세요.

DevTools Profiler로 컴포넌트 프로파일링

`react-dom` 16.5+와 `react-native` 0.57+는 React DevTools Profiler를 사용하여 개발 모드에서 향상된 프로파일링 기능을 제공합니다. Profiler에 대한 내용은 블로그 포스트 "[Introducing the React Profiler](#)"에서 확인할 수 있습니다. Profiler에 대한 영상도 [YouTube](#)에서 확인 가능합니다.

아직 React DevTools를 설치하지 않은 경우 다음 링크에서 확인하세요.

- [Chrome Browser Extension](#)
- [Firefox Browser Extension](#)
- [Standalone Node Package](#)

주의

`react-dom`의 프로덕션 프로파일링 `bundle`은 `react-dom/profiling`으로 이용할 수 있습니다. `bundle`을 사용하는 법에 대한 자세한 내용은 fb.me/react-profiling에서 확인하세요.

주의

React 17 이전에는 크롬 퍼포먼스 탭으로 컴포넌트를 프로파일링 하기 위해 [User Timing API](#) 표준을 사용했습니다. 자세한 내용은 [Ben Schwarz](#)이 작성한 글을 참조해주세요.

긴 목록 가상화하세요

애플리케이션에서 긴 목록(수백 또는 수천행)을 렌더링하는 경우 'windowing'이라는 기법을 사용하는 것을 추천합니다. 이 기법은 주어진 시간에 목록의 부분 목록만 렌더링하며 컴포넌트를 다시 렌더링하는 데 걸리는 시간과 생성된 DOM 노드의 수를 크게 줄일 수 있습니다.

`react-window`과 `react-virtualized`는 널리 알려진 windowing 라이브러리입니다. 목록, 그리드 및 표 형식 데이터를 표시하기 위한 몇 가지 재사용 가능한 컴포넌트를 제공합니다. 애플리케이션의 특정한 활용 사례에 더 적합한 것을 원한다면 Twitter처럼 자신만의 windowing 컴포넌트를 만들 수 있습니다.

재조정을 피하세요

React는 렌더링 된 UI의 internal representation을 빌드하고 유지 관리합니다. 여기에는 컴포넌트에서 반환되는 React 엘리먼트가 포함됩니다. representation은 React가 JavaScript 객체에서의 작업보다 느릴 수 있기 때문에 필요에 따라 DOM 노드를 만들고 기존 노드에 접근하지 못하도록 합니다. 때론 "virtual DOM"이라고 불리기도 하지만, React Native에서 같은 방식으로 동작합니다.

컴포넌트의 prop이나 state가 변경되면 React는 새로 반환된 엘리먼트를 이전에 렌더링된 엘리먼트와 비교해서 실제 DOM 업데이트가 필요한지 여부를 결정합니다. 같지 않을 경우 React는 DOM을 업데이트합니다.

React가 변경된 DOM 노드만 업데이트하더라도 리렌더링에는 여전히 다소 시간이 걸립니다. 대부분의 경우 문제가 되지 않지만 속도 저하가 눈에 띄는 경우 다시 렌더링이 시작되기 전에 실행되는 생명주기 함수 `shouldComponentUpdate`로 이 기능을 무시함으로써 속도를 높일 수 있습니다. 이 함수의 기본 implementation은 `true`를 반환하고 React는 업데이트를 진행합니다.

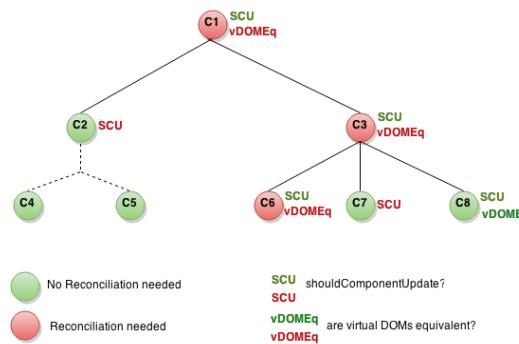
```
shouldComponentUpdate(nextProps, nextState) {
  return true;
}
```

몇몇 상황에서 컴포넌트를 업데이트할 필요가 없다는 것을 알고 있다면 `shouldComponentUpdate`에서 `false`를 반환해서 컴포넌트와 그 자식 컴포넌트에서 `render()`를 호출하는 것을 포함해서 전체 렌더링 프로세스를 건너뛰게 할 수 있습니다.

대부분의 경우 `shouldComponentUpdate()`를 직접 작성하는 대신 `React.PureComponent`에서 상속 받을 수 있습니다. 그것은 현재와 이전의 prop과 state의 양은 비교로 `shouldComponentUpdate()`를 호출하는 것과 같습니다.

shouldComponentUpdate In Action

컴포넌트의 하위트리를 살펴보세요. 각 항목에 대해 SCU 는 `shouldComponentUpdate` 가 반환한 것을 나타내며, vDOMEq 는 React 엘리먼트가 동일한지 여부를 표시합니다. 마지막으로 원의 색은 컴포넌트를 조정해야 하는지 여부를 나타냅니다.



`shouldComponentUpdate` 는 C2에 뿌리를 둔 하위트리에서 `false`를 반환했기 때문에 React는 C2를 렌더링하려고 시도하지 않았으므로 C4 및 C5에서 `shouldComponentUpdate`를 호출할 필요가 없었습니다.

C1과 C3의 경우 `shouldComponentUpdate` 가 `true` 를 반환했으므로 React가 트리의 가장 하위에 가서 확인해야 했습니다. C6의 경우 `shouldComponentUpdate` 는 `true` 를 반환했고 렌더링 된 엘리먼트는 동일하지 않기 때문에 React는 DOM을 업데이트해야 했습니다.

마지막 흥미로운 경우는 C8입니다. React는 이 컴포넌트를 렌더링 해야 했지만 이전에 렌더링 된 React 엘리먼트와 동일했기 때문에 DOM을 업데이트할 필요가 없었습니다.

React는 C6에 대해 DOM 변경(mutation)을 수행하면 되는데, 이는 필연적입니다. C8의 경우 렌더링 된 React 엘리먼트를 비교해서 빠져(bail out)나오고 C2의 하위트리와 C7의 경우 `shouldComponentUpdate`를 구제할 때 엘리먼트를 비교할 필요조차 없었고 `render`도 호출되지 않았습니다.

예시

컴포넌트가 변경되는 유일한 방법이 `props.color` 또는 `state.count` 변수가 변경되는 경우라면 `shouldComponentUpdate`에서 다음을 확인하세요.

```
class CounterButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  shouldComponentUpdate(nextProps, nextState) {
    if (this.props.color !== nextProps.color) {
      return true;
    }
    if (this.state.count !== nextState.count) {
      return true;
    }
    return false;
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
        Count: {this.state.count}
      </button>
    );
  }
}
```

이 코드에서 `shouldComponentUpdate`는 `props.color` 또는 `state.count`에 변화가 있는지 확인합니다. 해당 값들이 변경되지 않으면 컴포넌트가 업데이트되지 않습니다. 컴포넌트가 더 복잡해지면 `props` 와 `state`의 모든 필드 사이에 "얕은 비교"를 하는 것과 유사한 패턴을 사용해서 컴포넌트가 업데이트되어야만 하는지 여부를 결정할 수 있습니다. 이 패턴은 React가 `React.PureComponent`에서 단순히 상속받아 활용되는 이 로직을 사용하는 데 도움을 줄 만큼 일반적입니다. 따라서 이 코드는 동일한 효과를 얻을 수 있는 더 간단한 방법입니다.

```
class CounterButton extends React.PureComponent {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
        Count: {this.state.count}
      </button>
    );
  }
}
```

```

    );
}
}

```

대부분의 경우 `shouldComponentUpdate` 를 쓰는 대신 `React.PureComponent` 를 사용할 수 있습니다. 같은 비교만 수행하기 때문에 같은 비교로는 지나칠 수 있는 방식으로 `state`나 `props`가 변화한다면 사용할 수 없습니다.

이런 부분은 복잡한 데이터 구조인 경우 문제가 될 수 있습니다. 예를 들어 `listOfWords` 라는 컴포넌트가 쉼표로 구분된 단어 목록을 렌더링하고 단어를 목록에 추가할 수 있는 버튼을 클릭하도록 해주는 상위의 `WordAdder` 컴포넌트를 사용한다고 가정을 해보겠습니다. 이 코드는 정상적으로 작동하지 않습니다.

```

class ListOfWords extends React.PureComponent {
  render() {
    return <div>{this.props.words.join(',')}</div>;
  }
}

class WordAdder extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      words: ['marklar']
    };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // This section is bad style and causes a bug
    const words = this.state.words;
    words.push('marklar');
    this.setState({words: words});
  }

  render() {
    return (
      <div>
        <button onClick={this.handleClick} />
        <ListOfWords words={this.state.words} />
      </div>
    );
  }
}

```

문제는 `PureComponent` 가 `this.props.words` 의 이전 값과 새로운 값을 간단하게 비교한다는 점입니다. 이 코드는 `WordAdder` 의 `handleClick` 메서드에서 `words` 배열을 변경시키기 때문에 배열의 실제 단어가 변경되었다 하더라도 `this.props.words` 의 이전 값과 새로운 값은 동일하게 비교됩니다. 따라서 `ListOfWords` 는 렌더링 되어야 하는 새로운 단어가 있어도 업데이트되지 않습니다.

데이터를 변형시키지 않음으로써 얻는 효과

이 문제를 피하는 가장 간단한 방법은 `props`와 `state`로 사용중인 값의 변경을 피하는 것입니다. 예를 들어 `handleClick` 메서드는 `concat` 을 사용해서 다시 작성될 수 있습니다.

```
handleClick() {
  this.setState(state => ({
    words: state.words.concat(['marklar'])
  }));
}
```

ES6는 이런 동작을 쉽게 만들어주는 [spread syntax](#)를 배열에서 지원합니다. Create React App을 사용하고 있다면 이 문법은 기본적으로 사용할 수 있습니다.

```
handleClick() {
  this.setState(state => ({
    words: [...state.words, 'marklar'],
  }));
}
```

비슷한 방법으로 mutation을 피하기 위해 객체를 변경하는 코드를 다시 쓸 수 있습니다. 예를 들어 `colormap`이란 객체가 있고 `colormap.right`를 'blue'로 변경시키는 함수를 만들고 싶다고 가정해봅시다. 그렇다면 아래와 같이 작성할 수 있습니다.

```
function updateColorMap(colormap) {
  colormap.right = 'blue';
}
```

객체 원본을 변경시키지 않고 작성하려면 [Object.assign](#)메서드를 사용하세요.

```
function updateColorMap(colormap) {
  return Object.assign({}, colormap, {right: 'blue'});
}
```

`updateColorMap`은 기존 객체를 변경하지않고 새로운 객체를 반환합니다. `Object.assign`은 ES6의 문법이고 폴리필(polyfill)을 필요로합니다.

객체 또한 변경 없이 손쉽게 업데이트될 수 있도록 [object spread properties](#)를 추가하자는 JavaScript 제안이 있습니다.

```
function updateColorMap(colormap) {
  return {...colormap, right: 'blue'};
}
```

이 기능은 ES2018에 추가되었습니다.

Create React App을 사용하고 있다면 `Object.assign`과 `object spread` 문법은 기본적으로 활용 가능합니다.

깊게 중첩된 객체를 처리할 때 불변성을 지키는 방식으로 객체를 업데이트하면 복잡하다고 느낄 수 있습니다. 이런 문제를 마주했다면 [Immer](#) 혹은 [immutability-helper](#)를 살펴보세요. 불변성이 가져다주는 이득을 잊지 않고 조금 더 가독성 있는 코드를 작성할 수 있게 해줄겁니다.

Portals

Portal은 부모 컴포넌트의 DOM 계층 구조 바깥에 있는 DOM 노드로 자식을 렌더링하는 최고의 방법을 제공합니다.

ReactDOM.createPortal(child, container)

첫 번째 인자(`child`)는 엘리먼트, 문자열, 혹은 `fragment`와 같은 어떤 종류이든 [렌더링할 수 있는 React 자식](#)입니다. 두 번째 인자(`container`)는 DOM 엘리먼트입니다.

사용법

보통 컴포넌트 렌더링 메서드에서 엘리먼트를 반환할 때 그 엘리먼트는 부모 노드에서 가장 가까운 자식으로 DOM에 마운트됩니다.

```
render() {
  // React는 새로운 div를 마운트하고 그 안에 자식을 렌더링합니다.
  return (
    <div>
      {this.props.children}
    </div>
  );
}
```

그런데 가끔 DOM의 다른 위치에 자식을 삽입하는 것이 유용할 수 있습니다.

```
render() {
  // React는 새로운 div를 생성하지 *않고* `domNode` 안에 자식을 렌더링합니다.
  // `domNode`는 DOM 노드라면 어떠한 것이든 유효하고, 그것은 DOM 내부의 어디에 있든지 상관없습니다.
  return ReactDOM.createPortal(
    this.props.children,
    domNode
  );
}
```

portal의 전형적인 유스케이스는 부모 컴포넌트에 `overflow: hidden`이나 `z-index`가 있는 경우이지만, 시각적으로 자식을 "튀어나오도록" 보여야 하는 경우도 있습니다. 예를 들어, 다이얼로그, 호버카드나 툴팁과 같은 것입니다.

주의

portal을 이용하여 작업할 때 [키보드 포커스 관리](#)가 매우 중요하다는 것을 염두에 두세요.

모달 다이얼로그(modal dialogs)의 경우 [WAI-ARIA Modal Authoring Practices](#)에 따라 모든 모달 다이얼로그(modal dialogs)와 상호작용할 수 있는지 확인해주세요.

[CodePen에서 실행하기](#)

Portal을 통한 이벤트 버블링

portal이 DOM 트리의 어디에도 존재할 수 있다 하더라도 모든 다른 면에서 일반적인 React 자식처럼 동작합니다. context와 같은 기능은 자식이 portal이든지 아니든지 상관없이 정확하게 같게 동작합니다. 이는 *DOM 트리*에서의 위치에 상관없이 portal은 여전히 *React 트리*에 존재하기 때문입니다.

이것에는 이벤트 버블링도 포함되어 있습니다. portal 내부에서 발생한 이벤트는 *React 트리*에 포함된 상위로 전파될 것입니다. *DOM 트리*에서는 그 상위가 아니라 하더라도 말입니다. 다음의 HTML 구조를 가정해 봅시다.

```
<html>
  <body>
    <div id="app-root"></div>
    <div id="modal-root"></div>
  </body>
</html>
```

`#app-root` 안에 있는 `Parent` 컴포넌트는 형제 노드인 `#modal-root` 안의 컴포넌트에서 전파된 이벤트가 포착되지 않았을 경우 그것을 포착할 수 있습니다.

```
// 여기 이 두 컨테이너는 DOM에서 형제 관계입니다.
const appRoot = document.getElementById('app-root');
const modalRoot = document.getElementById('modal-root');

class Modal extends React.Component {
  constructor(props) {
    super(props);
    this.el = document.createElement('div');
  }

  componentDidMount() {
    // Portal 엘리먼트는 Modal의 자식이 마운트된 후 DOM 트리에 삽입됩니다.
    // 요컨대, 자식은 어디에도 연결되지 않은 DOM 노드로 마운트됩니다.
    // 자식 컴포넌트가 마운트될 때 그것을 즉시 DOM 트리에 연결해야만 한다면,
    // 예를 들어, DOM 노드를 계산한다든지 자식 노드에서 'autoFocus'를 사용한다든지 하는 경우에,
    // Modal에 state를 추가하고 Modal이 DOM 트리에 삽입되어 있을 때만 자식을 렌더링해주세요.
    modalRoot.appendChild(this.el);
  }

  componentWillUnmount() {
    modalRoot.removeChild(this.el);
  }

  render() {
    return ReactDOM.createPortal(
      this.props.children,
      this.el
    );
  }
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {clicks: 0};
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
```

```
// 이것은 Child에 있는 버튼이 클릭 되었을 때 발생하고 Parent의 state를 갱신합니다.
// 비록 버튼이 DOM 상에서 직계 자식이 아니라고 하더라도 말입니다.
this.setState(state => ({
  clicks: state.clicks + 1
}));
```

```
}
```

```
render() {
  return (
    <div onClick={this.handleClick}>
      <p>Number of clicks: {this.state.clicks}</p>
      <p>
        Open up the browser DevTools
        to observe that the button
        is not a child of the div
        with the onClick handler.
      </p>
      <Modal>
        <Child />
      </Modal>
    </div>
  );
}
```

```
function Child() {
  // 이 버튼에서의 클릭 이벤트는 부모로 버블링됩니다.
  // 왜냐하면 'onClick' 속성이 정의되지 않았기 때문입니다.
  return (
    <div className="modal">
      <button>Click</button>
    </div>
  );
}

const root = ReactDOM.createRoot(appRoot);
root.render(<Parent />);
```

[CodePen에서 실행하기](#)

portal에서 버블링된 이벤트를 부모 컴포넌트에서 포착한다는 것은 본질적으로 portal에 의존하지 않는 조금 더 유연한 추상화 개발이 가능함을 나타냅니다. 예를 들어, `<Modal />` 컴포넌트를 렌더링할 때 부모는 그것이 portal을 사용했는지와 관계없이 `<Modal />`의 이벤트를 포착할 수 있습니다.

ES6 없이 사용하는 React

보통 React 컴포넌트를 정의할 때 JavaScript의 class를 사용한다면 이와 같을 겁니다.

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

아직 ES6를 사용하지 않는다면, 그 대신 `create-react-class` 모듈을 사용할 수도 있습니다.

```
var createReactClass = require('create-react-class');
var Greeting = createReactClass({
  render: function() {
    return <h1>Hello, {this.props.name}</h1>;
  }
});
```

ES6 class의 API는 몇몇 차이점을 제외하고는 `createReactClass()` 와 비슷합니다.

Props 기본값 선언

함수와 ES6의 class를 통해 `defaultProps` 를 컴포넌트 그 자체의 속성으로서 정의할 수 있습니다.

```
class Greeting extends React.Component {
  // ...
}

Greeting.defaultProps = {
  name: 'Mary'
};
```

`createReactClass()` 를 사용한다면, 인자로 넘겨지는 객체 내에서 `getDefaults()` 를 함수로 정의해야 합니다.

```
var Greeting = createReactClass({
  getDefaults: function() {
    return {
      name: 'Mary'
    };
  },
  // ...
});
```

초기 State 정의

ES6 class의 생성자에서 `this.state` 에 값을 할당하면 state의 초기값을 정의할 수 있습니다.

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
  }
  // ...
}
```

`createReactClass()` 를 사용할 때마다 초기 state를 반환하는 `getInitialState` 메서드를 제공해야만 합니다.

```
var Counter = createReactClass({
  getInitialState: function() {
    return {count: this.props.initialCount};
  },
  // ...
});
```

자동 바인딩

ES6 class로서 선언된 React 컴포넌트에서 메서드는 일반적인 ES6 class일 때와 비슷합니다. 즉, `this` 를 인스턴스에 자동으로 바인딩하지 않습니다. 따라서 이 경우에는 생성자에서 별도로 `.bind(this)` 를 사용해 주어야 합니다.

```
class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: 'Hello!'};
    // 이 부분이 중요합니다!
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    alert(this.state.message);
  }

  render() {
    // `this.handleClick`이 바인딩 되었기 때문에, 이를 이벤트 핸들러로 사용할 수 있습니다.
    return (
      <button onClick={this.handleClick}>
        Say hello
      </button>
    );
  }
}
```

반면에 `createReactClass()` 를 사용한다면, 알아서 모든 메서드를 바인딩하기 때문에 위의 과정이 필요하지는 않습니다.

```
var SayHello = createReactClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },

  handleClick: function() {
    alert(this.state.message);
  },

  render: function() {
    return (
      <button onClick={this.handleClick}>
```

```

    Say hello
  </button>
);
}
});

```

이는 ES6의 class를 사용해 이벤트 핸들러를 만드는 경우에는 다른 방법으로 처리할 때 보다 반복되는 코드가 많아진다는 뜻입니다. 하지만 큰 규모의 애플리케이션에서는 class를 사용하는 경우에 성능이 조금 더 좋아집니다.

보일러플레이트 코드를 정 쓰기 싫다면, [ES2022 Class Properties](#) 문법을 사용할 수도 있습니다.

```

class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: 'Hello!'};

  }

  // 화살표 함수를 통해 메서드를 바인딩합니다.
  handleClick = () => {
    alert(this.state.message);
  };

  render() {
    return (
      <button onClick={this.handleClick}>
        Say hello
      </button>
    );
  }
}

```

다음과 같은 몇 가지 다른 선택지도 있습니다.

- 생성자에서 메서드들을 바인딩합니다.
- `onClick={(e) => this.handleClick(e)}` 와 같이 화살표 함수를 사용합니다.
- `createReactClass` 를 계속 사용합니다.

Mixins

주의

ES6에서는 처음부터 mixin에 대한 어떠한 지원도 없었습니다. 따라서, React에서 ES6 class를 사용하고자 하는 경우에도 mixin에 대한 별도의 지원은 없습니다.

또한 저희 팀은 **mixin**을 사용한 **Codebase**에서 수 많은 문제점들을 발견했습니다. 이에 따라 저희는 [새로 작성하는 코드에서는 mixin을 사용하지 않는 것을 추천드립니다.](#)

해당 글은 참고목적으로만 보시길 바랍니다.

가끔은 전혀 다른 컴포넌트들이 어느 정도 유사한 기능을 공유할 수도 있습니다. 간혹 발생하는 이러한 경우를 번역과 링크에 관한 [횡단 관 심사](#)라고 부릅니다. 이 문제에 대한 대처법으로서 `createClass`를 통해 더 이상 쓰이지 않는 코드인 `mixins`을 사용할 수 있습니다.

`mixin`을 사용하는 흔한 예로는 시간 간격을 두고 반복적으로 스스로 내용을 갱신하는 컴포넌트를 만들고자 할 경우가 있습니다. 이는 `setInterval()`을 사용하면 간단하게 만들 수 있지만, 메모리를 절약하기 위해서 컴포넌트를 더 이상 사용하지 않을 때 이를 취소하는 것이 중요합니다. React는 [생명주기 메서드](#)를 제공합니다. 생명주기 메서드는 컴포넌트가 생성되거나 파괴되기 직전에 이를 알려주는 역할을 합니다. 생명주기 메서드를 이용하는 간단한 `mixin`을 만들어 보겠습니다. 이 `mixin`은 컴포넌트가 파괴될 때 자동으로 정리되는 `setInterval()` 함수 기능을 제공해 줍니다.

```
var SetIntervalMixin = {
  componentWillMount: function() {
    this.intervals = [];
  },
  setInterval: function() {
    this.intervals.push(setInterval.apply(null, arguments));
  },
  componentWillUnmount: function() {
    this.intervals.forEach(clearInterval);
  }
};

var createReactClass = require('create-react-class');

var TickTock = createReactClass({
  mixins: [SetIntervalMixin], // mixin을 사용
  getInitialState: function() {
    return {seconds: 0};
  },
  componentDidMount: function() {
    this.setInterval(this.tick, 1000); // mixin에서 메서드를 호출
  },
  tick: function() {
    this.setState({seconds: this.state.seconds + 1});
  },
  render: function() {
    return (
      <p>
        React has been running for {this.state.seconds} seconds.
      </p>
    );
  }
});

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

하나의 컴포넌트가 같은 생명주기 메서드를 정의한 여러 `mixin`을 사용한다고 생각해봅시다. 예를 든다면, `mixin`들이 컴포넌트가 파괴될 때 어떠한 정리 동작을 하려고 할 수도 있습니다. 이 때는 모든 생명주기 메서드의 호출이 보장됩니다. `mixin`에서 정의된 생명주기 메서드들은 `mixin`이 나열된 순서대로 작동되며 그 뒤에 컴포넌트의 메서드가 호출됩니다.

JSX 없이 사용하는 React

React를 사용할 때 JSX는 필수가 아닙니다. 빌드 환경에서 컴파일 설정을 하고 싶지 않을 때 JSX 없이 React를 사용하는 것은 특히 편리합니다.

각 JSX 엘리먼트는 `React.createElement(component, props, ...children)` 를 호출하기 위한 문법 설탕입니다. 그래서 JSX로 할 수 있는 모든 것은 순수 JavaScript로도 할 수 있습니다.

예를 들어 다음의 JSX로 작성된 코드는

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

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Hello toWhat="World" />);
```

아래처럼 JSX를 사용하지 않은 코드로 컴파일될 수 있습니다.

```
class Hello extends React.Component {
  render() {
    return React.createElement('div', null, `Hello ${this.props.toWhat}`);
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(React.createElement(Hello, {toWhat: 'World'}, null));
```

JSX가 JavaScript로 변환되는 예시를 더 보고 싶다면 [the online Babel compiler](#)를 참고하세요.

컴포넌트는 문자열이나 `React.Component` 의 하위 클래스 또는 컴포넌트를 위한 일반 함수로 제공됩니다.

`React.createElement` 를 너무 많이 입력하는 것이 피곤하다면 짧은 변수에 할당하는 방법이 있습니다.

```
const e = React.createElement;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(e('div', null, 'Hello World'));
```

`React.createElement` 를 짧은 변수에 할당하면 편리하게 JSX 없이 React를 사용할 수 있습니다.

더 간결한 구문을 제공하는 `react-hyperscript` 나 `hyperscript-helpers` 같은 커뮤니티 프로젝트를 참고해도 좋습니다.

[Go to TOC](#)

재조정 (Reconciliation)

React는 선언적 API를 제공하기 때문에 갱신이 될 때마다 매번 무엇이 바뀌었는지를 걱정할 필요가 없습니다. 이는 애플리케이션 작성을 무척 쉽게 만들어주지만, React 내부에서 어떤 일이 일어나고 있는지는 명확히 눈에 보이지 않습니다. 이 글에서는 우리가 React의 “비교 (diffing)” 알고리즘을 만들 때 어떤 선택을 했는지를 소개합니다. 이 비교 알고리즘 덕분에 컴포넌트의 갱신이 예측 가능해지면서도 고성능 앱이라고 불러도 손색없을 만큼 충분히 빠른 앱을 만들 수 있습니다.

동기

React를 사용하다 보면, '`render()`' 함수는 React 엘리먼트 트리를 만드는 것이다.'라고 생각이 드는 순간이 있을 것입니다. `state`나 `props`가 갱신되면 `render()` 함수는 새로운 React 엘리먼트 트리를 반환할 것입니다. 이때 React는 방금 만들어진 트리에 맞게 가장 효과적으로 UI를 갱신하는 방법을 알아낼 필요가 있습니다.

하나의 트리를 가지고 다른 트리로 변환하기 위한 최소한의 연산 수를 구하는 알고리즘 문제를 풀기 위한 일반적인 해결책들이 있습니다. 하지만 이러한 최첨단의 알고리즘도 n 개의 엘리먼트가 있는 트리에 대해 $O(n^3)$ 의 복잡도를 가집니다.

React에 이 알고리즘을 적용한다면, 1000개의 엘리먼트를 그리기 위해 10억 번의 비교 연산을 수행해야 합니다. 너무나도 비싼 연산이죠. React는 대신, 두 가지 가정을 기반하여 $O(n)$ 복잡도의 휴리스틱 알고리즘을 구현했습니다.

1. 서로 다른 타입의 두 엘리먼트는 서로 다른 트리를 만들어낸다.
2. 개발자가 `key` prop을 통해, 여러 렌더링 사이에서 어떤 자식 엘리먼트가 변경되지 않아야 할지 표시해 줄 수 있다.

실제로 거의 모든 사용 사례에서 이 가정들은 들어맞습니다.

비교 알고리즘 (Diffing Algorithm)

두 개의 트리를 비교할 때, React는 두 엘리먼트의 루트(root) 엘리먼트부터 비교합니다. 이후의 동작은 루트 엘리먼트의 타입에 따라 달라집니다.

엘리먼트의 타입이 다른 경우

두 루트 엘리먼트의 타입이 다르면, React는 이전 트리를 버리고 완전히 새로운 트리를 구축합니다. `<a>`에서 ``로, `<Article>`에서 `<Comment>`로, 혹은 `<Button>`에서 `<div>`로 바뀌는 것 모두 트리 전체를 재구축하는 경우입니다.

트리를 버릴 때 이전 DOM 노드들은 모두 파괴됩니다. 컴포넌트 인스턴스는 `componentWillUnmount()`가 실행됩니다. 새로운 트리가 만들어질 때, 새로운 DOM 노드들이 DOM에 삽입됩니다. 그에 따라 컴포넌트 인스턴스는 `UNSAFE_componentWillMount()`가 실행되고 `componentDidMount()`가 이어서 실행됩니다. 이전 트리와 연관된 모든 state는 사라집니다.

루트 엘리먼트 아래의 모든 컴포넌트도 언마운트되고 그 state도 사라집니다. 예를 들어, 아래와 같은 비교가 일어나면,

```
<div>
  <Counter />
</div>

<span>
  <Counter />
</span>
```

이전 `Counter` 는 사라지고, 새로 다시 마운트가 될 것입니다.

주의

아래 메서드들은 레거시이며 새로 작성하는 코드에서는 [피해야 합니다](#).

- `UNSAFE_componentWillMount()`

DOM 엘리먼트의 타입이 같은 경우

같은 타입의 두 React DOM 엘리먼트를 비교할 때, React는 두 엘리먼트의 속성을 확인하여, 동일한 내역은 유지하고 변경된 속성들만 갱신합니다. 예를 들어,

```
<div className="before" title="stuff" />
<div className="after" title="stuff" />
```

이 두 엘리먼트를 비교하면, React는 현재 DOM 노드 상에 `className` 만 수정합니다.

`style` 이 갱신될 때, React는 또한 변경된 속성만을 갱신합니다. 예를 들어,

```
<div style={{color: 'red', fontWeight: 'bold'}} />
<div style={{color: 'green', fontWeight: 'bold'}} />
```

위 두 엘리먼트 사이에서 변경될 때, React는 `fontWeight` 는 수정하지 않고 `color` 속성 만을 수정합니다.

DOM 노드의 처리가 끝나면, React는 이어서 해당 노드의 자식들을 재귀적으로 처리합니다.

같은 타입의 컴포넌트 엘리먼트

컴포넌트가 갱신되면 인스턴스는 동일하게 유지되어 렌더링 간 `state`가 유지됩니다. React는 새로운 엘리먼트의 내용을 반영하기 위해 현재 컴포넌트 인스턴스의 `props`를 갱신합니다. 이때 해당 인스턴스의 `UNSAFE_componentWillReceiveProps()`, `UNSAFE_componentWillUpdate()`, `componentDidUpdate` 를 호출합니다.

다음으로 `render()` 메서드가 호출되고 비교 알고리즘이 이전 결과와 새로운 결과를 재귀적으로 처리합니다.

주의

아래 메서드들은 레거시이며 새로 작성하는 코드에서는 [피해야 합니다](#).

- `UNSAFE_componentWillUpdate()`
- `UNSAFE_componentWillReceiveProps()`

자식에 대한 재귀적 처리

DOM 노드의 자식들을 재귀적으로 처리할 때, React는 기본적으로 동시에 두 리스트를 순회하고 차이점이 있으면 변경을 생성합니다.

예를 들어, 자식의 끝에 엘리먼트를 추가하면, 두 트리 사이의 변경은 잘 작동할 것입니다.

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>

<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

React는 두 트리에서 `first` 가 일치하는 것을 확인하고, `second` 가 일치하는 것을 확인합니다. 그리고 마지막으로 `third` 를 트리에 추가합니다.

하지만 위와 같이 단순하게 구현하면, 리스트의 맨 앞에 엘리먼트를 추가하는 경우 성능이 좋지 않습니다. 예를 들어, 아래의 두 트리 변환은 형편없이 작동합니다.

```
<ul>
  <li>Duke</li>
  <li>Villanova</li>
</ul>

<ul>
  <li>Connecticut</li>
  <li>Duke</li>
  <li>Villanova</li>
</ul>
```

React는 `Duke` 와 `Villanova` 종속 트리를 그대로 유지하는 대신 모든 자식을 변경합니다. 이러한 비효율은 문제가 될 수 있습니다.

Keys

이러한 문제를 해결하기 위해, React는 `key` 속성을 지원합니다. 자식들이 `key`를 가지고 있다면, React는 `key`를 통해 기존 트리와 이후 트리의 자식들이 일치하는지 확인합니다. 예를 들어, 위 비효율적인 예시에 `key` 를 추가하여 트리의 변환 작업이 효율적으로 수행되도록 수정할 수 있습니다.

```
<ul>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>

<ul>
  <li key="2014">Connecticut</li>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

이제 React는 `'2014'` `key`를 가진 엘리먼트가 새로 추가되었고, `'2015'` 와 `'2016'` `key`를 가진 엘리먼트는 그저 이동만 하면 되는 것을 알 수 있습니다.

실제로, `key`로 사용할 값을 정하는 것은 어렵지 않습니다. 그리려고 하는 엘리먼트는 일반적으로 식별자를 가지고 있을 것이고, 그대로 해당 데이터를 `key`로 사용할 수 있습니다.

```
<li key={item.id}>{item.name}</li>
```

이러한 상황에 해당하지 않는다면, 여러분의 데이터 구조에 ID라는 속성을 추가해주거나 데이터 일부에 해시를 적용해서 key를 생성할 수 있습니다. 해당 key는 오로지 형제 사이에서만 유일하면 되고, 전역에서 유일할 필요는 없습니다.

최후의 수단으로 배열의 인덱스를 key로 사용할 수 있습니다. 항목들이 재배열되지 않는다면 이 방법도 잘 동작할 것이지만, 재배열되는 경우 비효율적으로 동작할 것입니다.

인덱스를 key로 사용 중 배열이 재배열되면 컴포넌트의 state와 관련된 문제가 발생할 수 있습니다. 컴포넌트 인스턴스는 key를 기반으로 갱신되고 재사용됩니다. 인덱스를 key로 사용하면, 항목의 순서가 바뀌었을 때 key 또한 바뀔 것입니다. 그 결과로, 컴포넌트의 state가 엉망이 되거나 의도하지 않은 방식으로 바뀔 수도 있습니다.

인덱스를 key로 사용하여 문제가 발생하는 [Codepen 예시](#)는 여기에 있습니다. 그리고 [개선한 예시](#)에서는 인덱스를 key로 사용하지 않으면서도 앞에서 다뤘던 재배열, 정렬 그리고 이어서 발생하는 문제들을 어떻게 해결하는지 여기에서 확인할 수 있습니다.

고려 사항

재조정 알고리즘은 구현상의 세부사항이라는 것을 명심하세요. React는 항상 전체 앱을 재렌더링할 수도 있지만, 최종적으로 출력되는 결과는 항상 같을 것입니다. 좀 더 정확히 말하자면, 여기서 말하는 재렌더링은 모든 컴포넌트의 `render`를 호출하는 것이지 React가 언마운트시키고 다시 마운트하는 것은 아닙니다. 즉, 앞서 설명했던 규칙에 따라 렌더링 전후에 변경된 부분만을 적용할 것입니다.

우리는 일반적인 사용 사례에서 더 빠르게 작동할 수 있도록 계속 휴리스틱 알고리즘을 개선하고 있습니다. 현재 구현체에서는 한 종속 트리가 그 형제 사이에서 이동했다는 사실을 표현할 수는 있지만, 아예 다른 곳으로 이동했다는 사실은 표현할 수 없습니다. 알고리즘은 전체 종속 트리를 재렌더링할 것입니다.

React는 휴리스틱에 의존하고 있기 때문에, 휴리스틱이 기반하고 있는 가정에 부합하지 않는 경우 성능이 나빠질 수 있습니다.

1. 알고리즘은 다른 컴포넌트 타입을 갖는 종속 트리들의 일치 여부를 확인하지 않습니다. 매우 비슷한 결과물을 출력하는 두 컴포넌트를 교체하고 있다면, 그 둘을 같은 타입으로 만드는 것이 더 나을 수도 있습니다. 우리는 실제 사용 사례에서 이 가정이 문제가 되는 경우를 발견하지 못했습니다.
2. key는 반드시 변하지 않고, 예상 가능하며, 유일해야 합니다. 변하는 key(`Math.random()`으로 생성된 값 등)를 사용하면 많은 컴포넌트 인스턴스와 DOM 노드를 불필요하게 재생성하여 성능이 나빠지거나 자식 컴포넌트의 state가 유실될 수 있습니다.

[Go to TOC](#)

DOM 엘리먼트

React는 성능 및 브라우저 간 호환성을 위해 브라우저의 독립적인 DOM 시스템을 구현합니다. React에서는 브라우저의 DOM 구현에서 몇 가지 어려운 부분을 정리할 수가 있었습니다.

React에서 모든 프로퍼티 및 어트리뷰트(이벤트 핸들러 포함)은 캐멀 케이스를 사용합니다. 예를 들어 HTML 어트리뷰트인 `tabindex` 는 React의 `tabIndex` 으로 표현합니다. 예외는 `aria-*` 및 `data-*` 어트리뷰트입니다. 이는 소문자로 표현합니다. 예를 들어, `aria-label` 은 `aria-label` 로 동일하게 유지됩니다.

어트리뷰트의 차이

React와 HTML 사이에는 다르게 작동하는 여러가지의 어트리뷰트들이 있습니다.

`checked`

`checked` 어트리뷰트는 `checkbox` 또는 `radio` 타입의 `<input>` 컴포넌트에 의해 지원됩니다. 이 어트리뷰트를 사용해서 컴포넌트의 선택 여부를 설정할 수 있습니다. 이는 제어 컴포넌트를 만들 때 유용합니다. `defaultChecked` 는 비제어 컴포넌트가 사용되는 동등한 의미를 가지는 어트리뷰트이며 컴포넌트가 처음 마운트될 때 선택 여부를 설정합니다.

`className`

CSS class를 사용하려면 `className` 어트리뷰트를 사용하세요. 이는 `<div>`, `<a>` 등과 같은 모든 일반적인 DOM 및 SVG 엘리먼트에 적용됩니다.

일반적이진 않지만, React를 웹 컴포넌트에 사용하는 경우 `class` 어트리뷰트를 사용하세요.

`dangerouslySetInnerHTML`

`dangerouslySetInnerHTML` 은 브라우저 DOM에서 `innerHTML` 을 사용하기 위한 React의 대체 방법입니다. 일반적으로 코드에서 HTML을 설정하는 것은 사이트 간 스크립팅 공격에 쉽게 노출될 수 있기 때문에 위험합니다. 따라서 React에서 직접 HTML을 설정할 수는 있지만, 위험하다는 것을 상기시키기 위해 `dangerouslySetInnerHTML` 을 작성하고 `__html` 키로 객체를 전달해야 합니다. 아래는 예시입니다.

```
function createMarkup() {
  return {__html: 'First &middot; Second'};
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

`htmlFor`

`for` 는 JavaScript에서 예약어이므로 React 엘리먼트는 `htmlFor` 를 대신 사용합니다.

onChange

`onChange` 이벤트는 예상한대로 폼 필드가 변경될 때 이벤트가 발생합니다. 의도적으로 기존 브라우저의 동작을 사용하지 않는데 `onChange` 는 이러한 동작에 대해 잘못된 명칭이며 React는 실시간으로 유저 입력을 처리하는 이벤트에 의존하기 때문입니다.

selected

`<option>` 을 선택됐다고 표시하고 싶다면, `<select>` 의 `value`에서 해당 option의 값을 대신 참조하세요.

자세한 설명은 "[The select Tag](#)"를 확인해주세요.

style

주의

문서의 일부 예시는 편의상 `style`을 사용하지만, **style** 어트리뷰트를 스타일링의 주요 수단으로 사용하는 것은 일반적으로 권장되지 않습니다. 대부분의 경우 `className` 외부 CSS stylesheet에 정의된 class를 참조하는데 사용해야 합니다. `style`은 보통 React 애플리케이션에서 렌더링 시점에 동적으로 계산된 스타일을 추가하기 위해 사용됩니다. [FAQ: Styling and CSS](#)를 참조해주세요.

`style` 어트리뷰트는 CSS 문자열 대신 캐멀 케이스 프로퍼티를 가진 JavaScript 객체로 받아들입니다. 이는 DOM style JavaScript 프로퍼티와 일관되며 더 효율적이며 XSS 보안 허점을 방지합니다. 아래는 예시입니다.

```
const divStyle = {
  color: 'blue',
  backgroundImage: 'url(' + imgUrl + ')',
};

function HelloWorldComponent() {
  return <div style={divStyle}>Hello World!</div>;
}
```

스타일에는 autoprefixer가 붙지 않습니다. 구형 브라우저에서 사용하려면 해당 스타일 프로퍼티를 입력해야 합니다.

```
const divStyle = {
  WebkitTransition: 'all', // note the capital 'W' here
  msTransition: 'all' // 'ms' is the only lowercase vendor prefix
};

function ComponentWithTransition() {
  return <div style={divStyle}>This should work cross-browser</div>;
}
```

스타일 키는 JS에서 DOM 노드의 프로퍼티에 접근하는 것과 일관되게 유지하기 위해 캐멜 케이스를 사용합니다. (예: `node.style.backgroundImage`). 벤더 프리픽스 `ms` 제외는 대문자로 시작해야 합니다. 이것이 바로 `WebkitTransition`이 대문자 "W"를 갖는 이유입니다.

React는 특정 숫자 인라인 스타일 프로퍼티는 "px" 접미사를 자동으로 추가합니다. "px"이 아닌 다른 단위를 사용하길 원한다면, 원하는 단위와 함께 값을 문자열로 지정해주세요. 아래는 예시입니다.

```
// Result style: '10px'
<div style={{ height: 10 }}>
  Hello World!
</div>

// Result style: '10%'
<div style={{ height: '10%' }}>
  Hello World!
</div>
```

모든 스타일 프로퍼티가 픽셀 문자열로 변환되는 것은 아닙니다. 어떤 속성들은 단위가 없습니다 (예: `zoom`, `order`, `flex`). 단위가 없는 모든 프로퍼티에 대한 목록을 [여기에서](#) 확인할 수 있습니다.

suppressContentEditableWarning

일반적으로, 자식이 있는 엘리먼트가 `contentEditable`로 표시된다면 제대로 동작하지 않으므로 경고가 표시됩니다. `suppressContentEditableWarning` 어트리뷰트는 경고를 표시하지 않도록 합니다. `contentEditable`을 수동으로 관리하는 `Draft.js`와 같은 라이브러리를 만들지 않는 한 이 옵션으로 사용하지 마세요.

suppressHydrationWarning

서버 사이드 렌더링을 사용하는 경우, 일반적으로 서버와 클라이언트가 다른 내용을 렌더링할 때 경고가 표시됩니다. 그러나 매우 드물게 정확히 일치시키는 게 힘들거나 불가능합니다. 예를 들어 타임 스탬프 같은 경우 서버와 클라이언트에서는 다를 것으로 예상됩니다.

`suppressHydrationWarning` 을 `true`로 설정하면, React는 어트리뷰트와 그 엘리먼트 내용의 불일치에 대해 경고하지 않습니다. 바로 밑 한 단계 깊이를 기준으로만 작동하며 해결책으로 사용하도록 되어 있습니다. 남용하지 마세요. 이벤트 보충에 대한 자세한 내용은 [ReactDOM.hydrateRoot\(\)](#) 문서를 참조해주세요.

value

`value` 어트리뷰트는 `<input>`, `<select>` 와 `<textarea>` 컴포넌트에 의해 지원됩니다. 이를 이용해 컴포넌트의 값을 설정할 수 있습니다. 이는 제어 컴포넌트를 만드는 데 유용합니다. `defaultValue` 비제어 컴포넌트에서 사용되는 동등한 의미를 가지는 어트리뷰트이며, 처음 마운트될 때 컴포넌트의 값을 설정합니다.

지원되는 모든 HTML 어트리뷰트

React 16부터는 모든 표준 [또는 사용자 정의](#) DOM 어트리뷰트가 완벽하게 지원됩니다.

React는 항상 JavaScript 중심 API를 DOM에 제공했습니다. React 컴포넌트는 사용자 지정 및 DOM 관련 `props` 둘 다 받아들이기 때문에, React는 DOM API처럼 camelCase를 사용합니다.

```
<div tabIndex={-1} /> // Just like node.tabIndex DOM API
<div className="Button" /> // Just like node.className DOM API
<input readOnly={true} /> // Just like node.readOnly DOM API
```

이러한 `props`는 위에 설명 된 특별한 경우를 제외하고는 해당 HTML 어트리뷰트와 유사하게 작동합니다.

React가 지원하는 DOM 어트리뷰트 중 일부는 다음과 같습니다.

```
accept acceptCharset accessKey action allowFullScreen alt autoComplete
autoFocus autoPlay capture cellPadding cellSpacing challenge charSet checked
cite classID className cols content contentEditable contextMenu controls
```

```
controlsList coords crossOrigin data dateTime default defer dir disabled
download draggable encType form formAction formEncType formMethod formNoValidate
formTarget frameBorder headers height hidden high href hrefLang htmlFor
httpEquiv icon id inputMode integrity is keyParams keyType kind label lang list
loop low manifest marginHeight marginWidth max maxLength media mediaGroup method
min minLength multiple muted name noValidate nonce open optimum pattern
placeholder poster preload profile radioGroup readOnly rel required reversed
role rowSpan rows sandbox scope scoped scrolling seamless selected shape size
sizes span spellCheck src srcDoc srcLang srcSet start step style summary
tabIndex target title type useMap value width wmode wrap
```

마찬가지로, 모든 SVG 어트리뷰트가 완벽하게 지원됩니다.

```
accentHeight accumulate additive alignmentBaseline allowReorder alphabetic
amplitude arabicForm ascent attributeName attributeType autoReverse azimuth
baseFrequency baseProfile baselineShift bbox begin bias by calcMode capHeight
clip clipPath clipPathUnits clipRule colorInterpolation
colorInterpolationFilters colorProfile colorRendering contentScriptType
contentStyleType cursor cx cy d decelerate descent diffuseConstant direction
display divisor dominantBaseline dur dx dy edgeMode elevation enableBackground
end exponent externalResourcesRequired fill fillOpacity fillRule filter
filterRes filterUnits floodColor floodOpacity focusable fontFamily fontSize
fontSizeAdjust fontStretch fontStyle fontVariant fontWeight format from fx fy
g1 g2 glyphName glyphOrientationHorizontal glyphOrientationVertical glyphRef
gradientTransform gradientUnits hanging horizAdvX horizOriginX ideographic
imageRendering in in2 intercept k k1 k2 k3 k4 kernelMatrix kernelUnitLength
kerning keyPoints keySplines keyTimes lengthAdjust letterSpacing lightingColor
limitingConeAngle local markerEnd markerHeight markerMid markerStart
markerUnits markerWidth mask maskContentUnits maskUnits mathematical mode
numOctaves offset opacity operator order orient orientation origin overflow
overlinePosition overlineThickness paintOrder panose1 pathLength
patternContentUnits patternTransform patternUnits pointerEvents points
pointsAtX pointsAtY pointsAtZ preserveAlpha preserveAspectRatio primitiveUnits
r radius refX refY renderingIntent repeatCount repeatDur requiredExtensions
requiredFeatures restart result rotate rx ry scale seed shapeRendering slope
spacing specularConstant specularExponent speed spreadMethod startOffset
stdDeviation stemh stemv stitchTiles stopColor stopOpacity
strikethroughPosition strikethroughThickness string stroke strokeDasharray
strokeDashoffset strokeLinecap strokeLinejoin strokeMiterlimit strokeOpacity
strokeWidth surfaceScale systemLanguage tableValues targetX targetY textAnchor
textDecoration textLength textRendering to transform u1 u2 underlinePosition
underlineThickness unicode unicodeBidi unicodeRange unitsPerEm vAlphabetic
vHanging vIdeographic vMathematical values vectorEffect version vertAdvY
vertOriginX vertOriginY viewBox viewTarget visibility widths wordSpacing
writingMode x x1 x2 xChannelSelector xHeight xlinkActuate xlinkArcrole
xlinkHref xlinkRole xlinkShow xlinkTitle xlinkType xmlns xmlnsXlink xmlBase
xmlLang xmlSpace y y1 y2 yChannelSelector z zoomAndPan
```

또한 소문자로만 이뤄진 경우 사용자 지정 어트리뷰트를 사용할 수도 있습니다.

[Go to TOC](#)

합성 이벤트(SyntheticEvent)

이 문서는 React의 이벤트 시스템 일부를 구성하는 `SyntheticEvent` 래퍼를 설명합니다. 더 많은 정보는 [이벤트 처리하기](#) 문서를 보세요.

개요

이벤트 핸들러는 모든 브라우저에서 이벤트를 동일하게 처리하기 위한 이벤트 래퍼 `SyntheticEvent` 객체를 전달받습니다. `stopPropagation()` 와 `preventDefault()`를 포함해서 인터페이스는 브라우저의 고유 이벤트와 같지만 모든 브라우저에서 동일하게 동작합니다.

브라우저의 고유 이벤트가 필요하다면 `nativeEvent` 어트리뷰트를 참조하세요. 합성 이벤트는 브라우저 고유 이벤트에 직접 대응되지 않으며 다릅니다. 예를 들어 `onMouseLeave`에서 `event.nativeEvent`는 `mouseout` 이벤트를 가리킵니다. 구체적인 연결은 공개된 API의 일부가 아니며 언제든지 변경될 수 있습니다. 모든 `합성 이벤트` 객체는 다음 어트리뷰트를 가집니다.

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
void persist()
DOMEventTarget target
number timeStamp
string type
```

주의

v17부터 `e.persist()` 는 `SyntheticEvent` 가 더 이상 풀링되지 않기 때문에 아무런 동작을 하지 않습니다.

주의

v0.14부터 이벤트 핸들러에서 `false` 가 반환되더라도 이벤트 전파가 더 이상 중지되지 않습니다. 대신 `e.stopPropagation()` 또는 `e.preventDefault()` 를 적절하게 수동으로 호출해야 합니다.

지원하는 이벤트

React는 이벤트들을 다른 브라우저에서도 같은 속성을 가지도록 표준화합니다.

다음 이벤트 핸들러는 이벤트 버블링 단계에서 호출됩니다. 캡처 단계에 이벤트 핸들러를 등록하기 위해서는 이벤트 이름에 `Capture` 를 덧붙이세요. 예를 들어 `onClick` 대신 `onClickCapture` 를 사용해서 캡처 단계에서 클릭 이벤트 핸들러를 사용할 수 있습니다.

- [Clipboard 이벤트](#)
 - [Composition 이벤트](#)
 - [Keyboard 이벤트](#)
 - [Focus 이벤트](#)
 - [Form 이벤트](#)
 - [Generic 이벤트](#)
 - [Mouse 이벤트](#)
 - [Pointer 이벤트](#)
 - [Selection 이벤트](#)
 - [Touch 이벤트](#)
 - [UI 이벤트](#)
 - [Wheel 이벤트](#)
 - [Media 이벤트](#)
 - [Image 이벤트](#)
 - [Animation 이벤트](#)
 - [Transition 이벤트](#)
 - [기타 이벤트](#)
-

참조

Clipboard 이벤트

이벤트 이름

`onCopy onCut onPaste`

속성

`DOMDataTransfer clipboardData`

Composition 이벤트

이벤트 이름

`onCompositionEnd onCompositionStart onCompositionUpdate`

속성

`string data`

Keyboard 이벤트

이벤트 이름

`onKeyDown onKeyPress onKeyUp`

속성

```
boolean altKey
number charCode
boolean ctrlKey
boolean getModifierState(key)
string key
number keyCode
string locale
number location
boolean metaKey
boolean repeat
boolean shiftKey
number which
```

`key` 속성은 DOM 레벨 3 이벤트 명세에 있는 어떤 값이든 가질 수 있습니다.

Focus 이벤트

이벤트 이름

`onFocus onBlur`

포커스 이벤트는 form 엘리먼트 뿐만이 아니라 모든 React DOM 엘리먼트에 작동합니다.

속성

`DOMEventTarget relatedTarget`

onFocus

`onFocus` 이벤트는 엘리먼트 (또는 자식 엘리먼트)가 포커스될 때 호출됩니다. 예를 들어, 유저가 텍스트 인풋을 클릭했을 때 호출됩니다.

```
function Example() {
  return (
    <input
      onFocus={(e) => {
        console.log('Focused on input');
      }}
      placeholder="onFocus is triggered when you click this input."
    />
  )
}
```

onBlur

`onBlur` 이벤트 핸들러는 엘리먼트 (또는 자식 엘리먼트)에서 포커스가 사라졌을 때 호출됩니다. 예를 들어, 유저가 포커스된 텍스트 인풋의 바깥 영역을 클릭했을 때 호출됩니다.

```
function Example() {
  return (
    <input
      onBlur={(e) => {
        console.log('Triggered because this input lost focus');
      }}
      placeholder="onBlur is triggered when you click this input and then you
      click outside of it."
    />
  )
}
```

Detecting Focus Entering and Leaving

부모 엘리먼트 바깥 영역으로부터 발생한 이벤트가 포커스 또는 블러인지 구분하기 위해 `currentTarget` 과 `relatedTarget` 을 사용할 수 있습니다. 다음은 복사해서 붙여넣을 수 있는 데모로 자식 엘리먼트 또는 엘리먼트 자체에 포커스 중인지, 전체 하위 트리에 포커스가 되고 있는지 사라지고 있는지 구별할 수 있는 방법을 보여줍니다.

```
function Example() {
  return (
    <div
      tabIndex={1}
      onFocus={(e) => {
        if (e.currentTarget === e.target) {
          console.log('focused self');
        } else {
          console.log('focused child', e.target);
        }
        if (!e.currentTarget.contains(e.relatedTarget)) {
          // Not triggered when swapping focus between children
          console.log('focus entered self');
        }
      }}
      onBlur={(e) => {
        if (e.currentTarget === e.target) {
          console.log('unfocused self');
        } else {
          console.log('unfocused child', e.target);
        }
        if (!e.currentTarget.contains(e.relatedTarget)) {
          // Not triggered when swapping focus between children
          console.log('focus left self');
        }
      }}
    >
      <input id="1" />
      <input id="2" />
    </div>
  );
}
```

Form 이벤트

이벤트 이름

`onChange onInput onInvalid onReset onSubmit`

`onChange` 이벤트에 대한 더 자세한 정보는 [폼 문서](#)를 참조하세요.

Generic 이벤트

이벤트 이름

`onError onLoad`

Mouse 이벤트

이벤트 이름

`onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit
onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave
onMouseMove onMouseOut onMouseOver onMouseUp`

`onMouseEnter` 및 `onMouseLeave` 이벤트는 일반적인 버블링 대신 마우스가 떠나는 엘리먼트에서 들어가는 엘리먼트로 전파되고 캡처 단계가 없습니다.

속성

```
boolean altKey
number button
number buttons
number clientX
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX
number pageY
DOMEventTarget relatedTarget
number screenX
number screenY
boolean shiftKey
```

Pointer 이벤트

이벤트 이름

`onPointerDown onPointerMove onPointerUp onPointerCancel onGotPointerCapture
onLostPointerCapture onPointerEnter onPointerLeave onPointerOver onPointerOut`

`onPointerEnter` 및 `onPointerLeave` 이벤트는 일반적인 버블링 대신 포인터가 떠나는 엘리먼트에서 들어가는 엘리먼트로 전파되고 캡처 단계가 없습니다.

속성

W3 명세에 정의된 대로 포인터 이벤트는 [마우스 이벤트](#)와 다음 속성을 포함해 확장합니다.

```
number pointerId
number width
number height
number pressure
number tangentialPressure
number tiltX
number tiltY
number twist
string pointerType
boolean isPrimary
```

크로스 브라우저 지원 주의사항

포인터 이벤트는 아직 모든 브라우저에서 지원되지 않습니다(이 문서를 작성하는 시점엔 Chrome, Firefox, Edge 및 Internet Explorer가 지원합니다). 표준 폴리필은 `react-dom` 번들을 무겁게 만들기 때문에 React가 직접 브라우저 호환성을 위해 폴리필을 제공하지 않습니다.

애플리케이션이 포인터 이벤트를 의존한다면 직접 서드 파티 포인터 폴리필을 추가하세요.

Selection 이벤트

이벤트 이름

```
onSelect
```

Touch 이벤트

이벤트 이름

```
onTouchCancel onTouchEnd onTouchMove onTouchStart
```

속성

```
boolean altKey
DOMTouchList changedTouches
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
boolean shiftKey
DOMTouchList targetTouches
DOMTouchList touches
```

UI 이벤트

이벤트 이름

```
onScroll
```

주의

React 17부터 `onScroll` 이벤트는 버블링되지 않습니다. 이는 브라우저 동작과 일치하며 스크롤 가능한 엘리먼트가 중첩된 상황에서 자식 엘리먼트가 멀리 떨어져 있는 부모 엘리먼트에 이벤트를 발생시킬 때 가질 수 있는 혼동을 막습니다.

속성

```
number detail
DOMAbstractView view
```

Wheel 이벤트**이벤트 이름**

```
onWheel
```

속성

```
number deltaMode
number deltaX
number deltaY
number deltaZ
```

Media 이벤트**이벤트 이름**

```
onAbort onCanPlay onCanPlayThrough onDurationChange onEmptied onEncrypted
onEnded onError onLoadedData onLoadedMetadata onLoadStart onPause onPlay
onPlaying onProgress onRateChange onSeeked onSeeking onStalled onSuspend
onTimeUpdate onVolumeChange onWaiting
```

Image 이벤트**이벤트 이름**

```
onLoad onError
```

Animation 이벤트**이벤트 이름**

```
onAnimationStart onAnimationEnd onAnimationIteration
```

속성

```
string animationName  
string pseudoElement  
float elapsedTime
```

Transition 이벤트

이벤트 이름

```
onTransitionEnd
```

속성

```
string propertyName  
string pseudoElement  
float elapsedTime
```

기타 이벤트

이벤트 이름

```
onToggle
```

[Go to TOC](#)

React 기술 용어 모음

싱글 페이지 애플리케이션

싱글 페이지 애플리케이션(Single-page application, SPA)은 하나의 HTML 페이지와 애플리케이션 실행에 필요한 JavaScript와 CSS 같은 모든 자산을 로드하는 애플리케이션입니다. 페이지 또는 후속 페이지의 상호작용은 서버로부터 새로운 페이지를 불러오지 않으므로 페이지가 다시 로드되지 않습니다.

React를 사용하여 싱글 페이지 애플리케이션을 만들 수 있지만, 필수 사항은 아닙니다. 기존 웹사이트 일부분의 상호작용을 개선하기 위하여 React를 사용할 수 있습니다. React로 작성된 코드는 PHP와 같은 서버에 의해 렌더된 마크업 또는 다른 클라이언트사이드 라이브러리와 함께 문제없이 공존할 수 있습니다. 사실, Facebook에서는 위와 같은 방식으로 React를 사용하고 있습니다.

ES6, ES2015, ES2016 등

이 약어들은 모두 ECMAScript 언어 명세의 최신 버전을 나타내며, JavaScript는 이를 구현한 것입니다. ES6 버전(ES2015로도 알려져 있습니다)에는 이전 버전에 없던 화살표 함수(arrow function), class, 템플릿 리터럴(template literal), `let` 과 `const` 구문과 같은 많은 추가 사항이 포함되어 있습니다. 특정 버전에 대한 자세한 내용은 [여기](#)에서 확인할 수 있습니다.

컴파일러

JavaScript 컴파일러(Compiler)는 JavaScript 코드를 변환하고 다른 형식으로 JavaScript 코드를 반환합니다. 일반적으로 ES6 문법을 구형 브라우저에서도 동작할 수 있도록 변환하는 데 많이 사용합니다. [Babel](#)은 React와 함께 가장 널리 사용되는 컴파일러입니다.

번들러

번들러(Bundler)는 분리된 JavaScript와 CSS 모듈 코드를 브라우저에 최적화된 여러 개의 파일로 결합합니다. React 애플리케이션에서 널리 사용되는 번들러에는 [Webpack](#)과 [Browserify](#)가 있습니다.

패키지 관리자

패키지 관리자는 프로젝트의 종속성을 관리할 수 있는 도구입니다. [npm](#)과 [Yarn](#)은 React 애플리케이션에서 자주 사용되는 패키지 관리자입니다. 두 패키지 관리자 모두 같은 npm 패키지 레지스트리의 클라이언트입니다.

CDN

CDN은 Content Delivery Network의 약자입니다. CDN은 전 세계의 서버 네트워크에서 캐시된 정적 콘텐츠를 제공합니다.

JSX

JSX는 JavaScript의 확장 문법입니다. JSX는 템플릿 언어와 비슷해 보이지만, JavaScript의 강력한 기능들을 모두 사용할 수 있습니다. JSX는 `React.createElement()`의 호출을 통해 일반 JavaScript 객체인 "React 엘리먼트"(React element)로 컴파일됩니다. JSX에 대한 기본 소개는 [여기](#)에서 확인할 수 있으며 JSX에 대한 자세한 튜토리얼은 [여기](#)에서 확인할 수 있습니다.

React DOM은 HTML 어트리뷰트(attribute) 이름 대신 캐멀케이스(camelCase)를 네이밍 컨벤션으로 사용합니다. 예를 들어, JSX에서 `tabindex` 는 `tabIndex`로 작성합니다. `class` 어트리뷰트는 JavaScript의 예약어이므로 `className`으로 작성합니다.

```
<h1 className="hello">My name is Clementine!</h1>
```

엘리먼트 {#elements}

React 엘리먼트(React Element)는 React 애플리케이션을 구성하는 블록입니다. 엘리먼트는 "컴포넌트(Component)"라는 널리 알려진 개념과 혼동되기 쉽습니다. 엘리먼트는 화면에 보이는 것들을 기술하며, React 엘리먼트는 변경되지 않습니다.

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

일반적으로 엘리먼트는 직접 사용되지 않고 컴포넌트로부터 반환됩니다.

컴포넌트 {#components}

React 컴포넌트는 페이지에 렌더링할 React 엘리먼트를 반환하는 작고 재사용 가능한 코드 조각입니다. 가장 간단한 React 컴포넌트는 React 엘리먼트를 반환하는 일반 JavaScript 함수입니다.

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

컴포넌트는 ES6 class로도 작성할 수 있습니다.

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

컴포넌트는 기능별로 나눌 수 있으며 다른 컴포넌트 안에서 사용할 수 있습니다. 컴포넌트는 다른 컴포넌트, 배열, 문자열 그리고 숫자를 반환할 수 있습니다. 화면을 구성하는 데 자주 사용되는 UI(Button, Panel, Avatar), 혹은 복잡한 UI(App, FeedStory, Comment) 컴포넌트는 재사용 가능한 컴포넌트가 될 수 있습니다. 컴포넌트의 이름은 항상 대문자로 시작해야 합니다 (`<Wrapper/>` (o) `<wrapper/>` (x)). 컴포넌트 렌더링에 대한 자세한 내용은 [이 문서](#)를 참고하세요.

props {#props}

`props` 는 컴포넌트의 입력값입니다. `props` 는 부모 컴포넌트로부터 자식 컴포넌트로 전달된 데이터입니다.

`props` 는 읽기 전용이라는 것에 주의하세요. `props` 는 어떤 방식으로든 수정해서는 안 됩니다.

```
// 틀린 예
props.number = 42;
```

사용자의 입력 또는 네트워크 응답에 반응하여 어떤 값을 수정해야 한다면 `state`를 사용하세요.

props.children {#propschildren}

모든 컴포넌트에서 `props.children` 를 사용할 수 있습니다. `props.children` 은 컴포넌트의 여는 태그와 닫는 태그 사이의 내용을 포함합니다. 예를 들어,

```
<Welcome>Hello world!</Welcome>
```

`Hello world!` 문자열은 `Welcome` 컴포넌트의 `props.children` 으로 사용할 수 있습니다.

```
function Welcome(props) {
  return <p>{props.children}</p>;
}
```

Class로 정의된 컴포넌트에서는 `this.props.children` 을 사용합니다.

```
class Welcome extends React.Component {
  render() {
    return <p>{this.props.children}</p>;
  }
}
```

state {#state}

컴포넌트와 관련된 일부 데이터가 시간에 따라 변경될 경우 `state` 가 필요합니다. 예를 들어, `Checkbox` 컴포넌트는 `isChecked` state가 필요할 수 있으며, `NewsFeed` 컴포넌트는 `fetchedPosts` 를 컴포넌트의 `state`를 통해 계속 주시하려고 할 수 있습니다.

`state` 와 `props` 의 가장 중요한 차이점은 `props` 는 부모 컴포넌트로부터 전달받지만, `state` 는 컴포넌트에서 관리된다는 것입니다. 컴포넌트는 `props` 를 변경할 수 없지만, `state` 는 변경할 수 있습니다.

데이터가 변경되는 각 특정한 부분에 대해, 해당 상태(`state`)를 "소유"하는 컴포넌트는 하나만 존재해야 합니다. 서로 다른 두 컴포넌트의 상태를 동기화하려고 하지마세요. 대신, 공통 상태를 두 컴포넌트의 공통 조상으로 끌어올리고 해당 데이터를 두 컴포넌트에 `props`로 전달하세요.

생명주기 메서드 {#lifecycle-methods}

생명주기 메서드(Lifecycle method)는 컴포넌트의 각각의 단계에서 실행되는 커스텀 기능입니다. 컴포넌트가 만들어지고 DOM에 삽입될 때(`mounting`), 컴포넌트가 업데이트될 때 및 컴포넌트가 DOM에서 마운트 해제될 때(`unmounted`) 혹은 제거될 때 사용할 수 있는 기능을 제공합니다.

제어 컴포넌트 vs. 비제어 컴포넌트

React는 두 가지 방식으로 form 입력을 처리합니다.

React에 의해 입력값이 제어되는 엘리먼트를 **제어 컴포넌트**(controlled component) 라고 합니다. 사용자가 제어 컴포넌트에 데이터를 입력하면 변경 이벤트 핸들러가 호출되고 코드가 (업데이트된 값으로 다시 렌더링에 의해) 입력의 유효 여부를 결정합니다. 다시 렌더링하지 않으면 form 엘리먼트는 변경되지 않은 상태로 유지됩니다.

비제어 컴포넌트(uncontrolled component)는 form 엘리먼트가 React 외부에서 작동하는 것처럼 작동합니다. 사용자가 form 필드(input box, dropdown 등)에 데이터를 입력하면 업데이트된 정보가 React에서 별도 처리할 필요 없이 엘리먼트에 반영됩니다. 그러나, 이는 특정 필드가 특정 값을 갖도록 강제할 수 없다는 의미이기도 합니다.

대부분은 controlled component를 사용해야 합니다.

Key {#keys}

"key"는 엘리먼트의 배열을 만들 때 포함해야 하는 특별한 문자열입니다. key는 React가 어떤 항목을 변경, 추가 혹은 삭제할지 식별하는 것을 돕습니다. 엘리먼트들을 안정적으로 식별할 수 있도록 배열 내의 엘리먼트에 key를 제공해야 합니다.

Key는 같은 배열에 포함된 다른 요소 사이에서만 고윳값을 가지면 됩니다. 전체 애플리케이션 또는 단일 컴포넌트 전체에서 고윳값을 가질 필요는 없습니다.

`Math.random()` 같은 값을 Key로 사용하면 안 됩니다. React가 항목 추가, 제거 또는 다시 정렬할 시기를 결정할 수 있도록 Key는 다시 렌더링하는 과정 동안 "안정적으로 식별 가능"해야 합니다. 이상적으로, Key는 `post.id`와 같이 데이터에서 사용되는 유일하고 안정적인 식별자를 사용하는 것이 좋습니다.

Ref {#refs}

React는 컴포넌트에 접근할 수 있는 특수한 어트리뷰트를 지원합니다. `ref` 어트리뷰트 `React.createRef()` 함수, 콜백 함수, 혹은 문자열(레거시 API에서)로 생성할 수 있습니다. `ref` 어트리뷰트가 콜백 함수인 경우, 함수는 DOM 엘리먼트나 class 인스턴스를 인자로 받습니다. 이를 통해 컴포넌트 인스턴스나 DOM 엘리먼트에 직접 접근할 수 있습니다.

Ref를 가능한 한 적게 사용하세요. 앱에서 Ref를 사용하여 "작동되는 부분"이 많다면 [하향식 데이터 흐름](#)을 사용하는 것이 더 좋습니다.

이벤트 {#events}

React 엘리먼트에서 이벤트를 처리할 때는 일반적인 방식과는 구문상의 차이점이 있습니다.

- React 이벤트 핸들러는 소문자가 아닌 캐멀케이스(camelCase)를 사용합니다.
- JSX를 사용하면 문자열이 아닌 함수로 이벤트 핸들러를 전달합니다.

재조정 {#reconciliation}

컴포넌트의 state나 props가 변경되면 React는 새로 반환된 컴포넌트를 이전에 렌더링된 컴포넌트와 비교하여 실제 DOM을 업데이트 해야하는지 결정합니다. 두 컴포넌트가 동일하지 않다면, React는 DOM을 업데이트 합니다. 이 과정을 재조정(Reconciliation)이라고 합니다.

JavaScript 환경 요구사항

React 18 supports all modern browsers (Edge, Firefox, Chrome, Safari, etc).

If you support older browsers and devices such as Internet Explorer which do not provide modern browser features natively or have non-compliant implementations, consider including a global polyfill in your bundled application.

Here is a list of the modern features React 18 uses:

- `Promise`
- `Symbol`
- `Object.assign`

The correct polyfill for these features depend on your environment. For many users, you can configure your [Browserlist](#) settings. For others, you may need to import polyfills like `core-js` directly.

Profiler API

`Profiler` 는 React 애플리케이션이 렌더링하는 빈도와 렌더링 "비용"을 측정합니다. Profiler의 목적은 [메모이제이션](#) 같은 성능 최적화 방법을 활용할 수 있는 애플리케이션의 느린 부분들을 식별해내는 것입니다.

주의

프로파일링은 약간의 오버헤드를 만들기 때문에 [프로덕션 빌드](#)에서는 비활성화되어 있습니다..

프로덕션에서 프로파일링을 활성화하길 원하신다면 React에서 제공하는 특별 프로덕션 빌드를 통해서 활성화하실 수 있습니다.

다음 링크에서 이 특별 빌드에 대해서 더 읽어보실 수 있습니다 fb.me/react-profiling

사용법

`Profiler` 는 React 트리 내에 어디에나 추가될 수 있으며 트리의 특정 부분의 렌더링 비용을 계산해줍니다. 이는 두 가지 `props`를 요구합니다: `id` (문자열) 와 `onRender` 콜백 (함수)이며 React 트리 내 컴포넌트에 업데이트가 "커밋"되면 호출됩니다.

예를 들어, `Navigation` 컴포넌트와 자손 컴포넌트들을 프로파일하기 위해서는

```
render(
  <App>
    <Profiler id="Navigation" onRender={callback}>
      <Navigation {...props} />
    </Profiler>
    <Main {...props} />
  </App>
);
```

복수의 `Profiler` 컴포넌트로 애플리케이션의 다른 부분들을 계산할 수 있습니다

```
render(
  <App>
    <Profiler id="Navigation" onRender={callback}>
      <Navigation {...props} />
    </Profiler>
    <Profiler id="Main" onRender={callback}>
      <Main {...props} />
    </Profiler>
  </App>
);
```

`Profiler` 컴포넌트는 하위 트리의 다른 컴포넌트들을 계산하기 위해 중첩해서 사용할 수 있습니다

```
render(
  <App>
    <Profiler id="Panel" onRender={callback}>
      <Panel {...props}>
        <Profiler id="Content" onRender={callback}>
          <Content {...props} />
        </Profiler>
      </Panel>
    </Profiler>
  </App>
);
```

```

        <Profiler id="PreviewPane" onRender={callback}>
          <PreviewPane {...props} />
        </Profiler>
      </Panel>
    </Profiler>
  </App>
);

```

주의사항

Profiler는 가벼운 컴포넌트이지만 필요할 때만 사용해야 합니다. 각 Profiler는 애플리케이션에 조금의 CPU와 메모리 비용을 추가하게 됩니다.

onRender 콜백

Profiler는 `onRender` 함수를 prop으로 요구합니다. React는 프로파일 트리 내의 컴포넌트에 업데이트가 "커밋"될 때마다 이 함수를 호출합니다. 이 함수는 무엇이 렌더링 되었는지 그리고 얼마나 걸렸는지 설명하는 입력값을 받게 됩니다.

```

function onRenderCallback(
  id, // 방금 커밋된 Profiler 트리의 "id"
  phase, // "mount" (트리가 방금 마운트가 된 경우) 혹은 "update" (트리가 리렌더링된 경우)
  actualDuration, // 커밋된 업데이트를 렌더링하는데 걸린 시간
  baseDuration, // 메모이제이션 없이 하위 트리 전체를 렌더링하는데 걸리는 예상시간
  startTime, // React가 현재 해당 업데이트를 렌더링하기 시작했는지
  commitTime, // React가 해당 업데이트를 현재 커밋했는지
  interactions // 이 업데이트에 해당하는 상호작용들의 집합
) {
  // 렌더링 타이밍을 집합하거나 로그...
}

```

각 prop에 대해 좀 더 자세히 알아보겠습니다.

방금 커밋된 Profiler 트리의 `id` prop. 복수의 프로파일러를 사용하고 있다면 트리의 어느 부분이 커밋되었는지 식별하는데 사용할 수 있습니다.

- **`phase: "mount" | "update"`** - 해당 트리가 방금 마운트된 건지 prop, state 혹은 hooks의 변화로 인하여 리렌더링 된 건지 식별합니다.
- **`actualDuration: number`** - 현재 업데이트에 해당하는 Profiler 와 자손 컴포넌트들을 렌더하는데 걸린 시간 이것은 하위 트리가 얼마나 메모이제이션을 잘 활용하고 있는지를 암시합니다 (e.g. `React.memo`, `useMemo`, `shouldComponentUpdate`). 이상적으로 대다수의 자손 컴포넌트들은 특정 prop이 변할 경우에만 리렌더링이 필요하기 때문에 이 값은 초기 렌더링 이후에 상당 부분 감소해야 합니다.
- **`baseDuration: number`** - Profiler 트리 내 개별 컴포넌트들의 가장 최근 `render` 시간의 지속기간 이 값은 렌더링 비용의 최악 케이스를 계산해줍니다(e.g. 초기 마운트 혹은 메모이제이션이 없는 트리)
- **`startTime: number`** - React가 현재 업데이트에 대해 렌더링을 시작한 시간의 타임 스탬프.
- **`commitTime: number`** - React가 현재 업데이트를 커밋한 시간의 타임 스탬프 이 값은 모든 프로파일러들이 공유하기 때문에 원한다면 그룹을 지을 수 있습니다.
- **`interactions: Set`** - 업데이트가 계획되었을 때 추적하고 있던 "상호작용"의 집합 (e.g. `render` 혹은 `setState` 가 호출되었을 때).

주의

상호작용을 추적하는 API는 아직 시험단계에 있지만, 상호작용은 업데이트의 원인을 식별하는데 사용할 수 있습니다

다음의 링크에서 더 자세히 알아볼 수 있습니다 <fb.me/react-interaction-tracing>

[Go to TOC](#)

PureRenderMixin

Note

The `PureRenderMixin` mixin predates `React.PureComponent`. This reference doc is provided for legacy purposes, and you should consider using `React.PureComponent` instead.

If your React component's render function renders the same result given the same props and state, you can use this mixin for a performance boost in some cases.

Example:

```
var PureRenderMixin = require('react-addons-pure-render-mixin');
var createReactClass = require('create-react-class');
createReactClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

Example using ES6 class syntax:

```
import PureRenderMixin from 'react-addons-pure-render-mixin';
class FooComponent extends React.Component {
  constructor(props) {
    super(props);
    this.shouldComponentUpdate = PureRenderMixin.shouldComponentUpdate.bind(this);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

Under the hood, the mixin implements `shouldComponentUpdate`, in which it compares the current props and state with the next ones and returns `false` if the equalities pass.

Note:

This only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only mix into components which have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using [immutable objects](#) to facilitate fast comparisons of nested data.

Furthermore, `shouldComponentUpdate` skips updates for the whole component subtree. Make sure all the children components are also "pure".

React.Component

이 문서에서는 React 컴포넌트 class를 다루는 API들을 자세히 소개합니다. 이 문서는 [컴포넌트와 props, state와 생명주기](#) 등과 같은 기초적인 React의 개념들에 익숙하다고 가정하고 있습니다. 그렇지 않다면, 먼저 읽으시길 바랍니다.

개요

React를 사용할 때는 컴포넌트를 class 또는 함수로 정의할 수 있습니다. class로 정의된 컴포넌트는 아래에 자세히 설명하고 있듯 보다 많은 기능을 제공합니다. React 컴포넌트 class를 정의하려면 `React.Component` 를 상속받아야 합니다.

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

`render()` 는 `React.Component` 의 하위 class에서 반드시 정의해야 하는 메서드입니다. 그 외에 이 문서에서 설명하는 메서드들은 선택 사항입니다.

컴포넌트 클래스를 직접 만들어서 사용하지 마세요. React 컴포넌트를 사용할 때에는 [상속보다 합성을 주로 사용합니다](#).

주의

React를 사용할 때 반드시 ES6 class 문법을 사용하지 않아도 됩니다. 그 대신 `create-react-class` 모듈 또는 이와 유사한 별도의 추상화를 사용해도 됩니다. 자세한 정보는 [ES6 없이 사용하는 React](#) 문서에서 확인할 수 있습니다.

컴포넌트 생명주기

모든 컴포넌트는 여러 종류의 "생명주기 메서드"를 가지며, 이 메서드를 오버라이딩하여 특정 시점에 코드가 실행되도록 설정할 수 있습니다. [이 생명주기 도표](#)를 필요할 때마다 활용하면 좋습니다. 아래 목록에서 자주 사용되는 생명주기 메서드를 **진하게** 표시했습니다. 나머지 것들은 상대적으로 자주 사용되지 않습니다.

마운트

아래 메서드들은 컴포넌트의 인스턴스가 생성되어 DOM 상에 삽입될 때에 순서대로 호출됩니다.

- `constructor()`
- `static getDerivedStateFromProps()`
- `render()`
- `componentDidMount()`

주의

아래 메서드는 기존에 사용되었지만 이제는 [사용하면 안 됩니다](#).

- `UNSAFE_componentWillMount()`

업데이트

`props` 또는 `state`가 변경되면 갱신이 발생합니다. 아래 메서드들은 컴포넌트가 다시 렌더링될 때 순서대로 호출됩니다.

- `static getDerivedStateFromProps()`
- `shouldComponentUpdate()`
- `render()`
- `getSnapshotBeforeUpdate()`
- `componentDidUpdate()`

주의

아래 메서드는 기존에 사용되었지만 이제는 [사용하면 안 됩니다](#).

- `UNSAFE_componentWillUpdate()`
- `UNSAFE_componentWillReceiveProps()`

마운트 해제

아래 메서드는 컴포넌트가 DOM 상에서 제거될 때에 호출됩니다.

- `componentWillUnmount()`

오류 처리

아래 메서드들은 자식 컴포넌트를 렌더링하거나, 자식 컴포넌트가 생명주기 메서드를 호출하거나, 또는 자식 컴포넌트가 생성자 메서드를 호출하는 과정에서 오류가 발생했을 때에 호출됩니다.

- `static getDerivedStateFromError()`
- `componentDidCatch()`

기타 API

이 외에도 컴포넌트는 몇몇 API를 제공합니다.

- `setState()`
- `forceUpdate()`

class 프로퍼티

- `defaultProps`
- `displayName`

인스턴스 프로퍼티

- `props`

- state

참고서

자주 사용되는 생명주기 메서드

이 섹션에서 다루는 메서드들을 사용하면 React 컴포넌트를 만들 때에 마주치는 대부분의 경우를 해결할 수 있습니다. [이 생명주기 도표](#)를 시각 자료로 활용하면 좋습니다.

render() {#render}

render()

`render()` 메서드는 클래스 컴포넌트에서 반드시 구현돼야하는 유일한 메서드입니다.

이 메서드가 호출되면 `this.props` 와 `this.state` 의 값을 활용하여 아래의 것 중 하나를 반환해야 합니다.

- **React 엘리먼트.** 보통 JSX를 사용하여 생성됩니다. 예를 들어, `<div />` 와 `<MyComponent />` 는 React가 DOM 노드 또는 사용자가 정의한 컴포넌트를 만들도록 지시하는 React 엘리먼트입니다.
- **배열과 Fragment.** `render()` 를 통하여 여러 개의 엘리먼트를 반환합니다. 자세한 정보는 [Fragments](#) 문서를 통하여 확인할 수 있습니다.
- **Portal.** 별도의 DOM 하위 트리에 자식 엘리먼트를 렌더링합니다. 자세한 정보는 [Portals](#)에서 확인할 수 있습니다.
- **문자열과 숫자.** 이 값들은 DOM 상에 텍스트 노드로서 렌더링됩니다.
- **Boolean 또는 null.** 아무것도 렌더링하지 않습니다. (대부분의 경우 `return test && <Child />` 패턴을 지원하는 데에 사용되며, 여기서 `test` 는 boolean 값입니다.)

`render()` 함수는 순수해야 합니다. 즉, 컴포넌트의 `state`를 변경하지 않고, 호출될 때마다 동일한 결과를 반환해야 하며, 브라우저와 직접적으로 상호작용을 하지 않습니다.

브라우저와 상호작용하는 작업이 필요하다면, 해당 작업을 `componentDidMount()` 이나 다른 생명주기 메서드 내에서 수행하세요. `render()` 를 순수하게 유지하여야 컴포넌트의 동작을 이해하기 쉽습니다.

주의

`shouldComponentUpdate()` 가 `false`를 반환하면 `render()` 는 호출되지 않습니다.

constructor() {#constructor}

constructor(props)

메서드를 바인딩하거나 `state`를 초기화하는 작업이 없다면, 해당 **React** 컴포넌트에는 생성자를 구현하지 않아도 됩니다.

React 컴포넌트의 생성자는 해당 컴포넌트가 마운트되기 전에 호출됩니다. `React.Component` 를 상속한 컴포넌트의 생성자를 구현 할 때에는 다른 구문에 앞서 `super(props)` 를 호출해야 합니다. 그렇지 않으면 `this.props` 가 생성자 내에서 정의되지 않아 버그로 이어질 수 있습니다.

React에서 생성자는 보통 아래의 두 가지 목적을 위하여 사용됩니다.

- `this.state`에 객체를 할당하여 지역 state를 초기화
- 인스턴스에 이벤트 처리 메서드를 바인딩

`constructor()` 내부에서 `setState()`를 호출하면 안 됩니다. 컴포넌트에 지역 state가 필요하다면 생성자 내에서 `this.state`에 초기 state 값을 할당하면 됩니다.

```
constructor(props) {
  super(props);
  // 여기서 this.setState()를 호출하면 안 됩니다!
  this.state = { counter: 0 };
  this.handleClick = this.handleClick.bind(this);
}
```

생성자는 `this.state`를 직접 할당할 수 있는 유일한 곳입니다. 그 외의 메서드에서는 `this.setState()`를 사용해야 합니다.

생성자 내에서는 부수 효과를 발생시키거나 구독 작업(subscription)을 수행하면 안 됩니다. 해당 경우에는 `componentDidMount()`를 대신 사용하세요.

주의

state에 props를 복사하면 안 됩니다! 가장 흔히 범하는 실수 중 하나입니다.

```
constructor(props) {
  super(props);
  // 이렇게 하지 마세요!
  this.state = { color: props.color };
}
```

이것은 불필요한 작업이며(`this.props.color`를 직접 사용하면 됩니다), 버그를 발생시킵니다(`color` props의 값이 변하더라도 state에 반영되지 않습니다).

props의 간신을 의도적으로 무시해야 할 때만 이와 같은 패턴을 사용하기 바랍니다. 이 경우, 해당 props의 이름을 `initialColor` 또는 `defaultColor` 등으로 변경하는 편이 자연스럽습니다. 그러면 이후 필요에 따라 컴포넌트가 `key`를 변경하여 초기 state를 "재설정"하도록 강제할 수 있습니다.

props의 값에 의존하는 state가 필요할 때 어떻게 해야 하는지에 대하여 알고 싶다면, 우리가 작성한 `state로부터 값을 가져오지 않는 법에 대한 블로그 글`을 읽어보세요.

componentDidMount() {#componentdidmount}

componentDidMount()

`componentDidMount()`는 컴포넌트가 마운트된 직후, 즉 트리에 삽입된 직후에 호출됩니다. DOM 노드가 있어야 하는 초기화 작업은 이 메서드에서 이루어지면 됩니다. 외부에서 데이터를 불러와야 한다면, `componentWillUnmount()`에서 구독 해제 작업을 반드시 수행하기 바랍니다.

`componentDidMount()`에서 즉시 `setState()`를 호출하는 경우도 있습니다. 이로 인하여 추가적인 렌더링이 발생하지만, 브라우저가 화면을 갱신하기 전에 이루어질 것입니다. 이 경우 `render()`가 두 번 호출되지만, 사용자는 그 중간 과정을 볼 수 없을 것입니다. 이런 사용 방식은 성능 문제로 이어지기 쉬우므로 주의가 필요합니다. 대부분의 경우, 앞의 방식을 대신하여 `constructor()` 메서드에서 초기 state를 할당할 수 있습니다. 하지만 모달(Modal) 또는 툴팁과 같이 렌더링에 앞서 DOM 노드의 크기나 위치를 먼저 측정해야 하는 경우 이러한 방식이 필요할 수 있습니다.

componentDidUpdate() {#componentdidupdate}

componentDidUpdate(prevProps, prevState, snapshot)

`componentDidUpdate()`는 갱신이 일어난 직후에 호출됩니다. 이 메서드는 최초 렌더링에서는 호출되지 않습니다.

컴포넌트가 갱신되었을 때 DOM을 조작하기 위하여 이 메서드를 활용하면 좋습니다. 또한, 이전과 현재의 `props`를 비교하여 네트워크 요청을 보내는 작업도 이 메서드에서 이루어지면 됩니다 (가령, `props`가 변하지 않았다면 네트워크 요청을 보낼 필요가 없습니다).

```
componentDidUpdate(prevProps) {
  // 전형적인 사용 사례 (props 비교를 잊지 마세요)
  if (this.props.userID !== prevProps.userID) {
    this.fetchData(this.props.userID);
  }
}
```

`componentDidUpdate()`에서 `setState()`를 즉시 호출할 수도 있지만, 위의 예시처럼 조건문으로 감싸지 않으면 무한 반복이 발생할 수 있다는 점에 주의하세요. 또한 추가적인 렌더링을 유발하여, 비록 사용자는 눈치채지 못할지라도 컴포넌트 성능에 영향을 미칠 수 있습니다. 상위에서 내려온 `prop`을 그대로 `state`에 저장하는 것은 좋지 않으며, 그 대신 `prop`을 직접 사용하는 것이 좋습니다. 이와 관련된 자세한 정보는 [props를 state에 복사하는 것이 버그를 유발하는 이유](#)에서 확인할 수 있습니다.

컴포넌트에서 `getSnapshotBeforeUpdate()`를 구현한다면, 해당 메서드가 반환하는 값은 `componentDidUpdate()`에 세 번째 "snapshot" 인자로 넘겨집니다. 반환값이 없다면 해당 인자는 `undefined`를 가집니다.

주의

`componentDidUpdate()`는 `shouldComponentUpdate()`가 `false`를 반환하면 호출되지 않습니다.

componentWillUnmount() {#componentwillunmount}

componentWillUnmount()

`componentWillUnmount()`는 컴포넌트가 마운트 해제되어 제거되기 직전에 호출됩니다. 이 메서드 내에서 타이머 제거, 네트워크 요청 취소, `componentDidMount()` 내에서 생성된 구독 해제 등 필요한 모든 정리 작업을 수행하세요.

이제 컴포넌트는 다시 렌더링되지 않으므로, `componentWillUnmount()` 내에서 `setState()`를 호출하면 안 됩니다. 컴포넌트 인스턴스가 마운트 해제되고 나면, 절대로 다시 마운트되지 않습니다.

잘 사용하지 않는 생명주기 메서드

이 섹션에서 다루는 메서드들은 잘 사용되지 않습니다. 유용하게 사용되는 경우가 아주 가끔 있지만, 대부분의 컴포넌트에서는 필요하지 않습니다. 대부분의 메서드들은 [이 생명주기 도표](#)의 최상단에 위치하는 "덜 일반적인 라이프 사이클 표시" 체크박스를 클릭하면 확인할 수 있습니다.

`shouldComponentUpdate()` [\[#shouldcomponentupdate\]](#)

`shouldComponentUpdate(nextProps, nextState)`

`shouldComponentUpdate()` 를 사용하면 현재 state 또는 props의 변화가 컴포넌트의 출력 결과에 영향을 미치는지 여부를 React가 알 수 있습니다. 기본 동작은 매 state 변화마다 다시 렌더링을 수행하는 것이며, 대부분의 경우 기본 동작에 따라야 합니다.

`shouldComponentUpdate()` 는 props 또는 state가 새로운 값으로 갱신되어서 렌더링이 발생하기 직전에 호출됩니다. 기본값은 `true` 입니다. 이 메서드는 초기 렌더링 또는 `forceUpdate()` 가 사용될 때에는 호출되지 않습니다.

이 메서드는 오직 [성능 최적화](#)만을 위한 것입니다. 렌더링을 방지하는 목적으로 사용할 경우 버그로 이어질 수 있습니다. `shouldComponentUpdate()` 의 내용을 직접 작성하는 대신에 `PureComponent` 를 사용하는 것이 좋습니다. `PureComponent` 는 props와 state에 대하여 같은 비교를 수행하고, 해야 할 갱신 작업을 건너뛸 확률을 낮춥니다.

이 메서드를 직접 작성할 자신이 있다면, `this.props` 와 `nextProps`, 그리고 `this.state` 와 `nextState` 를 비교한 뒤 `false` 를 반환하는 것으로 React가 갱신 작업을 건너뛰게 만들 수 있습니다. 여기서 `false` 를 반환하는 것이 자식 컴포넌트들이 각자가 가진 state의 변화에 따라 다시 렌더링을 수행하는 것을 막는 것은 아니라는 점에 주의하시길 바랍니다.

`shouldComponentUpdate()` 내에서 깊은 동일성 검사를 수행하거나 `JSON.stringify()` 를 사용하는 것을 권하지 않습니다. 아주 비효율적이며 성능을 떨어트릴 수 있습니다.

현재, `shouldComponentUpdate()` 가 `false` 를 반환할 경우 `UNSAFE_componentWillUpdate()`, `render()`, 그리고 `componentDidUpdate()` 는 호출되지 않습니다. 나중에는 `shouldComponentUpdate()` 를 엄격한 지시자가 아닌 힌트로서 다루게 될 것이고, `false` 의 반환을 반환하더라도 컴포넌트가 계속해서 다시 렌더링을 수행할 것입니다.

`static getDerivedStateFromProps()` [\[#static-getderivedstatefromprops\]](#)

`static getDerivedStateFromProps(props, state)`

`getDerivedStateFromProps` 는 최초 마운트 시와 갱신 시 모두에서 `render` 메서드를 호출하기 직전에 호출됩니다. `state` 를 갱신하기 위한 객체를 반환하거나, `null` 을 반환하여 아무 것도 갱신하지 않을 수 있습니다.

이 메서드는 시간이 흐름에 따라 변하는 props에 state가 의존하는 [아주 드문 사용례](#)를 위하여 존재합니다. 예를 들어, 무엇을 움직이도록 만들지 결정하기 위하여 이전과 현재의 자식 엘리먼트를 비교하는 `<Transition>` 와 같은 컴포넌트를 구현할 때에 편리하게 사용할 수 있습니다.

`state` 를 끌어오면 코드가 장황해지고, 이로 인하여 컴포넌트를 이해하기 어려워집니다. [보다 간단한 다른 대안들에 익숙해지는 것을 권장합니다.](#)

- props 변화에 대응한 [부수 효과를 발생시켜야](#) 한다면 (예를 들어, 데이터 가져오기 또는 애니메이션), `componentDidUpdate` 생명주기를 대신해서 사용하세요.

- **props**가 변화했을 때에만 일부 데이터를 다시 계산 하고 싶다면, [Memoization Helper](#)를 대신해서 사용하세요.
- **props**가 변화할 때에 일부 **state**를 재설정 하고 싶다면, [완전 제어 컴포넌트](#) 또는 `key` 를 사용하는 [완전 비제어 컴포넌트](#)로 만들어서 사용하세요.

이 메서드는 컴포넌트 인스턴스에 접근할 수 없습니다. 인스턴스 접근이 필요하다면, class 정의 외부에서 컴포넌트의 props와 state에 대한 순수 함수를 추출하여 `getDerivedStateFromProps()` 와 다른 클래스 메서드 간에 코드를 공유 및 재사용할 수 있습니다.

이 메서드는 이유와 상관없이 렌더링 때마다 [매번 실행되므로 주의하세요](#). 이는 `UNSAFE_componentWillReceiveProps` 와는 다른데, 이 메서드의 경우 부모 컴포넌트가 다시 렌더링을 발생시켰을 때에만 실행되고, 해당 컴포넌트 내에서 지역적인 `setState` 가 발생한 경우에는 실행되지 않습니다.

`getSnapshotBeforeUpdate()` [\[#getsnapshotbeforeupdate\]](#)

`getSnapshotBeforeUpdate(prevProps, prevState)`

`getSnapshotBeforeUpdate()` 는 가장 마지막으로 렌더링된 결과가 DOM 등에 반영되기 전에 호출됩니다. 이 메서드를 사용하면 컴포넌트가 DOM으로부터 스크롤 위치 등과 같은 정보를 이후 변경되기 전에 얻을 수 있습니다. 이 생명주기 메서드가 반환하는 값은 `componentDidUpdate()` 에 인자로 전달됩니다.

이 메서드에 대한 사용례는 흔하지 않지만, 채팅 화면처럼 스크롤 위치를 따로 처리하는 작업이 필요한 UI 등을 생각해볼 수 있습니다.

스냅샷 값을 반환하거나 `null` 을 반환합니다.

사용하는 예시는 아래와 같습니다.

`embed:react-component-reference/get-snapshot-before-update.js`

위의 예시에서는 `getSnapshotBeforeUpdate` 의 `scrollHeight` 프로퍼티 값을 아는 것이 중요한데, `render` 와 같은 "렌더링" 단계의 생명주기와 `getSnapshotBeforeUpdate` 와 `componentDidUpdate` 와 같은 "커밋" 단계의 생명주기 간에 자연 시간이 발생할 수 있기 때문입니다.

Error Boundary

[Error boundary](#)는 자식 컴포넌트 트리 내의 자바스크립트 오류를 감지하고, 해당 오류를 기록하며, 충돌이 발생한 컴포넌트 트리를 대신하여 대체 UI를 표시하는 React 컴포넌트입니다. Error boundary의 하위 트리에 존재하는 렌더링 과정, 생명주기 메서드, 모든 생성자에 대하여 오류를 감지해냅니다.

클래스 컴포넌트에 `static getDerivedStateFromError()` 또는 `componentDidCatch()` 를 정의할 경우 해당 컴포넌트는 Error boundary가 됩니다. 두 생명주기 내에서 `state`를 갱신하게 되면 하위 트리 내의 처리되지 않은 자바스크립트 오류를 발생시키고, 대체 UI를 표시합니다.

반드시 Error boundary는 예측하지 않은 예외를 처리하여 복구하는 경우에만 사용하기 바랍니다. 제어 흐름을 조작하는 데에는 사용하지 마세요.

자세한 정보는 [React 16에서 오류 처리하기](#) 문서에서 확인할 수 있습니다.

주의

Error boundary는 트리 내에서 자신보다 하위에 존재하는 컴포넌트에 대한 오류만을 감지해냅니다. 즉, Error boundary는 자기 자신에 대한 오류를 감지할 수 없습니다.

static getDerivedStateFromError() {#static-getderivedstatefromerror}

static getDerivedStateFromError(error)

이 생명주기 메서드는 하위의 자손 컴포넌트에서 오류가 발생했을 때 호출됩니다. 이 메서드는 매개변수로 오류를 전달받고, 갱신된 state 값을 반드시 반환해야 합니다.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // state를 갱신하여 다음 렌더링에서 대체 UI를 표시합니다.
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      // 별도로 작성한 대체 UI를 렌더링할 수도 있습니다.
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

주의

`getDerivedStateFromError()`는 "render" 단계에서 호출되므로, 부수 효과를 발생시키면 안 됩니다. 해당 경우에는 `componentDidCatch()`를 대신 사용하세요.

componentDidCatch() {#componentdidcatch}

componentDidCatch(error, info)

이 생명주기 메서드는 자손 컴포넌트에서 오류가 발생했을 때에 호출되며, 2개의 매개변수를 전달받습니다.

1. `error` - 발생한 오류
2. `info` - 어떤 컴포넌트가 오류를 발생시켰는지에 대한 정보를 포함한 `componentStack` 키를 갖고 있는 객체

`componentDidCatch()`는 "커밋" 단계에서 호출되므로, 부수 효과를 발생시켜도 됩니다. 아래와 같이 오류 로그 기록 등을 위하여 사용하면 됩니다.

```

class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // state를 갱신하여 다음 렌더링에서 대체 UI를 표시합니다.
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    // Example "componentStack":
    //   in ComponentThatThrows (created by App)
    //   in ErrorBoundary (created by App)
    //   in div (created by App)
    //   in App
    logComponentStackToMyService(info.componentStack);
  }

  render() {
    if (this.state.hasError) {
      // 별도로 작성한 대체 UI를 렌더링할 수도 있습니다.
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}

```

React의 `componentDidCatch()` 가 오류를 처리하는 방식은 프로덕션과 개발 빌드가 약간 다릅니다.

개발 빌드에서, 오류는 `window` 까지 전파됩니다. 이는 `window.onerror` 나 `window.addEventListener('error', callback)` 가 `componentDidCatch()` 에서 잡은 오류를 인터셉트하는 것을 의미합니다.

그러나 프로덕션 빌드에서 오류는 전파되지 않습니다. 즉 상위 오류 핸들러는 `componentDidCatch()` 에 의해 명시적으로 잡히지 않은 오류만 받습니다.

주의

오류 이벤트 내에서는 `setState()` 의 호출을 통하여 `componentDidCatch()` 로 구현된 대체 UI를 렌더링할 수 있습니다. 하지만 이런 방식은 이후의 릴리즈에서는 사용할 수 없게 될 것입니다. 대체 UI 렌더링 제어를 하려면 `static getDerivedStateFromError()` 를 대신 사용하세요.

레거시 생명주기 메서드

아래의 생명주기 메서드들은 "레거시"로 분류됩니다. 즉 여전히 작동하지만, 새로 작성하는 코드에서는 사용하지 않을 것을 권장합니다. 레거시 생명주기 메서드를 최신 문법으로 전환(migration)하는 방법에 대한 자세한 정보는 [이 블로그 문서](#)에서 확인할 수 있습니다.

UNSAFE_componentWillMount() {#unsafe_componentwillmount}

1 UNSAFE_componentWillMount()

주의

이 생명주기 메서드의 기존 이름은 `componentWillMount`입니다. 이 이름은 버전 17까지 그대로 유지될 것입니다. `rename-unsafe-lifecycles codemod`를 사용하면 컴포넌트를 최신 문법으로 자동 갱신시킬 수 있습니다.

`UNSAFE_componentWillMount()`는 마운트가 발생하기 전에 호출됩니다. `render()`가 실행되기 전에 호출되므로, 이 메서드 내에서 `setState()`를 동기적으로 호출하더라도 추가적인 렌더링이 발생하지 않습니다. `state`를 초기화하는 경우라면, 보통은 `constructor()`를 사용하는 것이 좋습니다.

이 메서드 내에서 부수 효과를 발생시키거나 구독 작업(subscription)을 수행하면 안 됩니다. 해당 경우에는 `componentDidMount()`를 대신 사용하세요.

이 메서드는 서버 렌더링에서 호출되는 유일한 생명주기 메서드입니다.

UNSAFE_componentWillReceiveProps() {#unsafe_componentwillreceiveprops}

`UNSAFE_componentWillReceiveProps(nextProps)`

주의

이 생명주기 메서드의 기존 이름은 `componentWillReceiveProps`입니다. 이 이름은 버전 17까지 그대로 유지될 것입니다. `rename-unsafe-lifecycles codemod`를 사용하면 컴포넌트를 최신 문법으로 자동 갱신시킬 수 있습니다.

주의

이 생명주기 메서드를 사용하면 버그를 만들거나, 일관성을 해칠 수 있습니다.

- `props` 변화에 대응한 부수 효과를 발생시켜야 한다면 (예를 들어, 데이터 가져오기 또는 애니메이션), `componentDidUpdate` 생명주기를 대신해서 사용하세요.
- `props`가 변화할 때에 일부 데이터를 다시 계산하기 위하여 `componentWillReceiveProps`를 사용하였다면, `Memoization Helper`를 대신해서 사용하세요.
- `props`가 변화할 때에 일부 `state`를 재설정하기 위하여 `componentWillReceiveProps`를 사용하였다면, `완전 제어 컴포넌트` 또는 `key`를 사용하는 `완전 비제어 컴포넌트`로 만들어서 사용하세요.

다른 사용 사례의 경우는 [가져온 state에 대하여 다른 블로그 글에서 추천하는 방법을 따르세요](#).

`UNSAFE_componentWillReceiveProps()`는 마운트된 컴포넌트가 새로운 `props`를 전달받기 전에 호출됩니다. `props` 변화에 대응하여 `state`를 갱신해야 한다면(예를 들어, `state`를 재설정하기), `this.props`와 `nextProps`를 비교하고 이 메서드 내에서 `this.setState()`를 사용하여 `state`를 바꾸면 됩니다.

부모 컴포넌트가 해당 컴포넌트가 다시 렌더링하게 만든 경우, `props`가 변화하지 않았더라도 이 메서드가 호출된다는 점에 주의하세요. 변화가 발생했을 때만 메서드를 실행시키려면 반드시 `props`의 현재값과 다음값을 비교해야 합니다.

React는 마운팅할 때에서는 `UNSAFE_componentWillReceiveProps()` 를 호출하지 않으며, 초기 `props`를 가지지 않습니다. 이 메서드가 호출되는 경우는 컴포넌트의 `props`가 변화했을 때입니다. `this.setState()` 를 호출하면, 대부분의 경우 `UNSAFE_componentWillReceiveProps()` 를 발생시키지 않습니다.

`UNSAFE_componentWillUpdate()` {#unsafe_componentwillupdate}

`UNSAFE_componentWillUpdate(nextProps, nextState)`

주의

이 생명주기 메서드의 기존 이름은 `componentWillUpdate` 입니다. 이 이름은 버전 17까지 그대로 유지될 것입니다. `rename-unsafe-lifecycles codemod`를 사용하면 컴포넌트를 최신 문법으로 자동 갱신시킬 수 있습니다.

`UNSAFE_componentWillUpdate()` 는 새로운 `props` 또는 `state`가 전달되어서 렌더링이 이루어지기 전에 호출됩니다. 이 메서드 내에서 갱신이 이루어지기 전에 미리 준비할 수 있습니다. 이 메서드는 초기 렌더링에서는 호출되지 않습니다.

이 메서드 내에서는 `this.setState()` 를 호출할 수 없다는 점에 주의하세요. 또한 React 컴포넌트가 갱신되도록 만드는 그 어떤 작업(예를 들어, Redux Action을 Dispatch하기)도 `UNSAFE_componentWillUpdate()` 가 결과값을 반환하기 전에는 이루어지면 안 됩니다.

통상적으로 이 메서드는 `componentDidUpdate()` 로 대체할 수 있습니다. 이 메서드 내에서 DOM에 대한 정보를 얻는다면(예를 들어, 스크롤 위치 저장하기), 해당 코드를 `getSnapshotBeforeUpdate()` 로 이전하는 것이 가능합니다.

Note

`UNSAFE_componentWillUpdate()` 는 `shouldComponentUpdate()` 이 `false` 를 반환한다면 호출되지 않습니다.

기타 API

위에서 설명한 생명주기 메서드들과 달리 아래의 메서드들은 사용자가 컴포넌트 내에서 직접 호출할 수 있습니다.

단 2개의 메서드, `setState()` 와 `forceUpdate()` 만이 존재합니다.

`setState()` {#setstate}

`setState(updater[, callback])`

`setState()` 는 컴포넌트 `state`의 변경 사항을 대기열에 집어넣고, React에게 해당 컴포넌트와 그 자식들이 갱신된 `state`를 사용하여 다시 렌더링되어야 한다고 알립니다. 이 메서드는 이벤트 핸들러와 서버 응답 등에 따라 UI를 갱신할 때에 가장 많이 사용하는 메서드입니다.

`setState()` 는 컴포넌트를 갱신하는 데에 있어 즉각적인 명령이 아니라 요청이라고 생각하시기 바랍니다. 인지 성능(Perceived Performance)의 향상을 위하여 React는 이 메서드의 실행을 지연시키고 여러 컴포넌트를 한번에 갱신할 수도 있습니다. In the rare case that you need to force the DOM update to be applied synchronously, you may wrap it in `flushSync`, but this may hurt performance.

`setState()` 는 컴포넌트를 항상 즉각적으로 갱신하지는 않습니다. 오히려 여러 변경 사항과 함께 일괄적으로 갱신하거나, 나중으로 미룰 수도 있습니다. 이로 인하여 `setState()` 를 호출하자마자 `this.state` 에 접근하는 것이 잠재적인 문제가 될 수 있습니다. 그 대신에 `componentDidUpdate` 또는 `setState` 의 콜백(`setState(updater, callback)`)을 사용하세요. 둘 다 갱신이 적용된 뒤에 실행되는 것이 보장됩니다. 이전 `state` 값을 기준으로 `state` 값을 설정해야 한다면, 아래에 설명된 `updater` 인자에 대한 내용을 읽어보세요.

`shouldComponentUpdate()` 가 `false` 를 반환하지 않는다면 `setState()` 는 항상 렌더링이 다시 발생하도록 만듭니다. 가변 객체의 사용으로 인하여 `shouldComponentUpdate()` 내에서 조건부 렌더링을 구현할 수 없다면, 새로운 `state` 가 이전의 `state` 와 다를 때에만 `setState()` 를 호출하세요. 그래야 불필요하게 다시 렌더링이 발생하지 않습니다.

첫번째 인자 `updater` 는 다음과 같은 형태를 가지는 함수입니다.

`(state, props) => stateChange`

`state` 는 변경 사항이 적용되는 시점에 컴포넌트가 가지는 `state`에 대한 참조입니다. `state` 는 직접 변경하면 안 됩니다. 대신, 전달된 `state` 와 `props` 를 기반으로 새로운 객체를 만들어서 변경 사항을 표현해야 합니다. 예를 들어, `props.step` 만큼 `state`의 어떤 값을 증가시키고 싶은 상황을 가정해봅시다.

```
this.setState((state, props) => {
  return {counter: state.counter + props.step};
});
```

`updater` 함수로 전달된 `state` 와 `props` 는 최신값임이 보장됩니다. `updater` 의 결과는 `state` 에 얹게 병합됩니다.

`setState()` 로 전달되는 두번째 매개변수는 `setState` 의 실행이 완료되고 컴포넌트가 다시 렌더링된 뒤에 실행될 함수에 대한 콜백으로, 생략할 수 있습니다. 보통 이러한 방식의 실행에는 `componentDidUpdate()` 의 사용을 권장합니다.

`setState()` 에 전달하는 첫번째 인자로 아래와 같이 객체를 전달하는 것도 가능합니다.

`setState(stateChange[, callback])`

이렇게 하면 `stateChange` 객체는 새로운 `state`에 얹게 병합됩니다. 예를 들어, 쇼핑 카트의 상품 수량(`quantity`)을 조정하려면 아래와 같이 작성하면 됩니다.

`this.setState({quantity: 2})`

이러한 형태의 `setState()` 도 마찬가지로 비동기적으로 수행되며, 같은 주기 동안 여러번 호출된다면 일괄적으로 처리될 수 있습니다. 예를 들어, 같은 주기 동안 상품 수량을 한번 이상 증가시키게 되면, 아래의 코드와 동일한 결과를 만들게 됩니다.

```
Object.assign(
  previousState,
  {quantity: state.quantity + 1},
  {quantity: state.quantity + 1},
  ...
)
```

`setState`의 연이은 호출은 같은 주기 내의 바로 직전 호출 결과를 덮어쓰고, 따라서 수량값이 한번만 증가될 것입니다. 다음 state의 값이 이전 state의 값에 기반한다면, 아래와 같이 `updater`의 함수 형태를 대신 사용하는 것이 좋습니다.

```
this.setState((state) => {
  return {quantity: state.quantity + 1};
});
```

자세한 정보는 아래의 문서들에서 확인할 수 있습니다.

- [State와 생명주기 안내](#)
- [자세히 알아보기: `setState\(\)` 호출은 언제, 그리고 왜 일괄 처리되는가?](#)
- [자세히 알아보기: `this.state`는 왜 즉시 갱신되지 않는가?](#)

forceUpdate() {#forceupdate}

```
component.forceUpdate(callback)
```

컴포넌트의 state 또는 props가 변경되면, 컴포넌트가 다시 렌더링되는 것이 기본 동작입니다. 어떤 컴포넌트의 `render()` 메서드가 다른 데이터값에 의존하는 경우, React로 하여금 `forceUpdate()`를 호출하여 렌더링을 다시 수행하도록 만들 수 있습니다.

`forceUpdate()`를 호출하면 컴포넌트의 `render()` 가 호출되는데, 이때 `shouldComponentUpdate()` 는 무시하고 건너뜁니다. 그러면 자식 컴포넌트들에 대하여 통상적인 생명주기 메서드가 실행되는데, 여기에는 개별 자식들의 `shouldComponentUpdate()` 도 포함됩니다. React는 마크업이 변화했을 때에만 DOM을 갱신할 것입니다.

보통 `render()` 내에서는 `forceUpdate()`를 사용하지 말아야 하며, 오직 `this.props` 와 `this.state`의 값만을 사용하여야 합니다.

class 프로퍼티

defaultProps {#defaultprops}

`defaultProps` 는 컴포넌트 클래스 자체 내에서 프로퍼티로서 정의될 수 있고, 이를 통하여 해당 class의 기본 props 값을 설정할 수 있습니다. 아래 예시와 같이, `null` 이 아닌 아직 정의되지 않은 `undefined` 인 props를 다룰 때 사용됩니다.

```
class CustomButton extends React.Component {
  // ...
}

CustomButton.defaultProps = {
  color: 'blue'
};
```

`props.color`에 값이 제공되지 않았다면, 기본값인 `'blue'`로 설정될 것입니다.

```
render() {
  return <CustomButton />; // props.color는 blue로 설정됩니다
}
```

`props.color`에 `null` 값이 제공된다면, 해당 값은 `null`로 유지됩니다.

```
render() {
  return <CustomButton color={null} /> ; // props.color는 null으로 유지됩니다
}
```

displayName {#displayname}

`displayName` 문자열은 디버깅 메시지 표시에 사용됩니다. 대부분의 경우 이 값을 설정하지 않아도 되는데, 왜냐하면 해당 컴포넌트를 정의하는 함수 또는 `class`의 이름으로부터 추론되기 때문입니다. 디버깅을 위하여 다른 이름을 표시하거나 고차 컴포넌트 생성을 위하여 명시적으로 이 값을 설정하고 싶다면, [쉬운 디버깅을 위한 Display Name 래핑하기](#)를 통하여 자세한 정보를 확인하세요.

인스턴스 프로퍼티

props {#props}

`this.props` 는 해당 컴포넌트가 호출된 곳에서 정의한 `props`를 포함하고 있습니다. `props`에 대하여 더 알고 싶다면 [컴포넌트와 props](#) 문서를 확인하세요.

특히 `this.props.children` 은 특별한 `prop`으로, 일반적인 태그가 아닌 `JSX` 표현으로 작성된 자식 태그로 정의되는 경우가 많습니다.

state {#state}

`state`는 어떤 컴포넌트에만 한정하여 사용되는 데이터를 포함하며, 해당 데이터는 시간이 지남에 따라 변경될 수 있습니다. `state`는 사용자가 자유롭게 정의할 수 있으며, 일반적인 자바스크립트 객체이어야 합니다.

어떤 값이 렌더링 또는 데이터 흐름 상에서 (예를 들어, 타이머의 ID) 사용되지 않는다면, 해당 값을 `state`에 넣지 않아도 됩니다. 그러한 값은 컴포넌트 인스턴스의 필드로 정의할 수 있습니다.

자세한 정보는 [State와 생명주기](#) 문서에서 확인할 수 있습니다.

`this.state` 를 직접 변경하면 안 됩니다. 왜냐하면 이후 호출되는 `setState()` 가 이전에 적용된 변경 사항을 덮어쓰기 때문입니다. `this.state` 를 불변적(Immutable)인 데이터로 취급하세요.

[Go to TOC](#)

ReactDOMClient

The `react-dom/client` package provides client-specific methods used for initializing an app on the client. Most of your components should not need to use this module.

```
import * as ReactDOM from 'react-dom/client';
```

If you use ES5 with npm, you can write:

```
var ReactDOM = require('react-dom/client');
```

Overview

The following methods can be used in client environments:

- `createRoot()`
- `hydrateRoot()`

Browser Support

React supports all modern browsers, although [some polyfills are required](#) for older versions.

Note

We do not support older browsers that don't support ES5 methods or microtasks such as Internet Explorer. You may find that your apps do work in older browsers if polyfills such as [es5-shim](#) and [es5-sham](#) are included in the page, but you're on your own if you choose to take this path.

Reference

`createRoot()` [\[#createroot\]](#)

```
createRoot(container[, options]);
```

Create a React root for the supplied `container` and return the root. The root can be used to render a React element into the DOM with `render`:

```
const root = createRoot(container);
root.render(element);
```

`createRoot` accepts two options:

- `onRecoverableError` : optional callback called when React automatically recovers from errors.
- `identifierPrefix` : optional prefix React uses for ids generated by `React.useId`. Useful to avoid conflicts when using multiple roots on the same page. Must be the same prefix used on the server.

The root can also be unmounted with `unmount`:

```
root.unmount();
```

Note:

`createRoot()` controls the contents of the container node you pass in. Any existing DOM elements inside are replaced when `render` is called. Later calls use React's DOM diffing algorithm for efficient updates.

`createRoot()` does not modify the container node (only modifies the children of the container). It may be possible to insert a component to an existing DOM node without overwriting the existing children.

Using `createRoot()` to hydrate a server-rendered container is not supported. Use `hydrateRoot()` instead.

hydrateRoot() {#hydrateroot}

```
hydrateRoot(container, element[, options])
```

Same as `createRoot()`, but is used to hydrate a container whose HTML contents were rendered by `ReactDOMServer`. React will attempt to attach event listeners to the existing markup.

`hydrateRoot` accepts two options:

- `onRecoverableError`: optional callback called when React automatically recovers from errors.
- `identifierPrefix`: optional prefix React uses for ids generated by `React.useId`. Useful to avoid conflicts when using multiple roots on the same page. Must be the same prefix used on the server.

Note

React expects that the rendered content is identical between the server and the client. It can patch up differences in text content, but you should treat mismatches as bugs and fix them. In development mode, React warns about mismatches during hydration. There are no guarantees that attribute differences will be patched up in case of mismatches. This is important for performance reasons because in most apps, mismatches are rare, and so validating all markup would be prohibitively expensive.

[Go to TOC](#)

ReactDOMServer

`ReactDOMServer` 객체를 통해 컴포넌트를 정적 마크업으로 렌더링할 수 있습니다. 대체로 이것은 Node 서버에서 사용됩니다.

```
// ES modules
import * as ReactDOMServer from 'react-dom/server';
// CommonJS
var ReactDOMServer = require('react-dom/server');
```

개요

These methods are only available in the **environments with Node.js Streams**:

- `renderToPipeableStream()`
- `renderToNodeStream()` (Deprecated)
- `renderToStaticNodeStream()`

These methods are only available in the **environments with Web Streams** (this includes browsers, Deno, and some modern edge runtimes):

- `renderToReadableStream()`

The following methods can be used in the environments that don't support streams:

- `renderToString()`
- `renderToStaticMarkup()`

Reference

`renderToPipeableStream()` {#rendertopipeablestream}

`ReactDOMServer.renderToPipeableStream(element, options)`

Render a React element to its initial HTML. Returns a stream with a `pipe(res)` method to pipe the output and `abort()` to abort the request. Fully supports Suspense and streaming of HTML with "delayed" content blocks "popping in" via inline `<script>` tags later. [Read more](#)

If you call `ReactDOM.hydrateRoot()` on a node that already has this server-rendered markup, React will preserve it and only attach event handlers, allowing you to have a very performant first-load experience.

```
let didError = false;
const stream = renderToPipeableStream(
  <App />,
  {
    onShellReady() {
      // The content above all Suspense boundaries is ready.
      // If something errored before we started streaming, we set the error code
      // appropriately.
      res.statusCode = didError ? 500 : 200;
      res.setHeader('Content-type', 'text/html');
```

```

        stream.pipe(res);
    },
    onShellError(error) {
      // Something errored before we could complete the shell so we emit an
      // alternative shell.
      res.statusCode = 500;
      res.send(
        '<!doctype html><p>Loading...</p><script src="clientrender.js"></script>'
      );
    },
    onAllReady() {
      // If you don't want streaming, use this instead of onShellReady.
      // This will fire after the entire page content is ready.
      // You can use this for crawlers or static generation.

      // res.statusCode = didError ? 500 : 200;
      // res.setHeader('Content-type', 'text/html');
      // stream.pipe(res);
    },
    onError(err) {
      didError = true;
      console.error(err);
    },
  },
);

```

See the [full list of options](#).

Note:

This is a Node.js-specific API. Environments with [Web Streams](#), like Deno and modern edge runtimes, should use `renderToReadableStream` instead.

`renderToReadableStream()` [#rendertoreadablestream](#)

`ReactDOMServer.renderToReadableStream(element, options);`

Streams a React element to its initial HTML. Returns a Promise that resolves to a [Readable Stream](#). Fully supports Suspense and streaming of HTML. [Read more](#)

If you call `ReactDOM.hydrateRoot()` on a node that already has this server-rendered markup, React will preserve it and only attach event handlers, allowing you to have a very performant first-load experience.

```

let controller = new AbortController();
let didError = false;
try {
  let stream = await renderToReadableStream(
    <html>
      <body>Success</body>
    </html>,
  {
    signal: controller.signal,
    onError(error) {
      didError = true;
    }
  }
);

```

```

        console.error(error);
    }
};

// This is to wait for all Suspense boundaries to be ready. You can uncomment
// this line if you want to buffer the entire HTML instead of streaming it.
// You can use this for crawlers or static generation:

// await stream.allReady;

return new Response(stream, {
    status: didError ? 500 : 200,
    headers: {'Content-Type': 'text/html'},
});
} catch (error) {
    return new Response(
        '<!doctype html><p>Loading...</p><script src="clientrender.js"></script>',
    {
        status: 500,
        headers: {'Content-Type': 'text/html'},
    }
);
}
}

```

See the [full list of options](#).

Note:

This API depends on [Web Streams](#). For Node.js, use `renderToPipeableStream` instead.

`renderToString()` (Deprecated)

`ReactDOMServer.renderToString(element)`

Render a React element to its initial HTML. Returns a [Node.js Readable stream](#) that outputs an HTML string. The HTML output by this stream is exactly equal to what `ReactDOMServer.renderToString` would return. You can use this method to generate HTML on the server and send the markup down on the initial request for faster page loads and to allow search engines to crawl your pages for SEO purposes.

If you call `ReactDOM.hydrateRoot()` on a node that already has this server-rendered markup, React will preserve it and only attach event handlers, allowing you to have a very performant first-load experience.

주의

서버에서만 사용할 수 있습니다. 이 API는 브라우저에서 작동하지 않습니다.

반환되는 스트림은 utf-8로 인코딩된 바이트 스트림을 반환합니다. 다른 방법으로 인코딩된 스트림이 필요할 경우, 텍스트 트랜스 코딩을 위한 Transform 스트림을 제공하는 [iconv-lite](#)와 같은 프로젝트를 살펴보세요.

`renderToStaticNodeStream()` {#rendertostaticnodestream}

ReactDOMServer.`renderToStaticNodeStream(element)`

`renderToNodeStream` 과 비슷하지만 `data-reactroot` 와 같이 React에서 내부적으로 사용하는 추가적인 DOM 어트리뷰트를 만들지 않습니다. 여러분의 어트리뷰트를 제거함으로써 약간의 바이트를 절약할 수 있으므로 React를 간단한 정적 페이지 생성기로 사용하고 싶은 경우에 유용합니다.

이 스트림이 출력하는 HTML 문자열은 `ReactDOMServer.renderToStaticMarkup` 이 반환하는 값과 정확히 일치합니다.

If you plan to use React on the client to make the markup interactive, do not use this method. Instead, use `renderToNodeStream` on the server and `ReactDOM.hydrateRoot()` on the client.

주의

서버에서만 사용할 수 있습니다. 이 API는 브라우저에서 작동하지 않습니다.

The stream returned from this method will return a byte stream encoded in utf-8. If you need a stream in another encoding, take a look at a project like [iconv-lite](#), which provides transform streams for transcoding text.

`renderToString()` {#rendertostring}

ReactDOMServer.`renderToString(element)`

Render a React element to its initial HTML. React will return an HTML string. You can use this method to generate HTML on the server and send the markup down on the initial request for faster page loads and to allow search engines to crawl your pages for SEO purposes.

If you call `ReactDOM.hydrateRoot()` on a node that already has this server-rendered markup, React will preserve it and only attach event handlers, allowing you to have a very performant first-load experience.

Note

This API has limited Suspense support and does not support streaming.

On the server, it is recommended to use either `renderToPipeableStream` (for Node.js) or `renderToReadableStream` (for Web Streams) instead.

renderToStaticMarkup() {#rendertostaticmarkup}

ReactDOMServer.renderToStaticMarkup(element)

Similar to `renderToString`, except this doesn't create extra DOM attributes that React uses internally, such as `data-reactroot`. This is useful if you want to use React as a simple static page generator, as stripping away the extra attributes can save some bytes.

If you plan to use React on the client to make the markup interactive, do not use this method. Instead, use `renderToString` on the server and `ReactDOM.hydrateRoot()` on the client.

ReactDOM

The `react-dom` package provides DOM-specific methods that can be used at the top level of your app and as an escape hatch to get outside the React model if you need to.

```
import * as ReactDOM from 'react-dom';
```

If you use ES5 with npm, you can write:

```
var ReactDOM = require('react-dom');
```

The `react-dom` package also provides modules specific to client and server apps:

- `react-dom/client`
- `react-dom/server`

개요

The `react-dom` package exports these methods:

- `createPortal()`
- `flushSync()`

These `react-dom` methods are also exported, but are considered legacy:

- `render()`
- `hydrate()`
- `findDOMNode()`
- `unmountComponentAtNode()`

Note:

Both `render` and `hydrate` have been replaced with new [client methods](#) in React 18. These methods will warn that your app will behave as if it's running React 17 (learn more [here](#)).

브라우저 지원

React supports all modern browsers, although [some polyfills are required](#) for older versions.

주의

We do not support older browsers that don't support ES5 methods or microtasks such as Internet Explorer. You may find that your apps do work in older browsers if polyfills such as [es5-shim](#) and [es5-sham](#) are included in the page, but you're on your own if you choose to take this path.

참조

`createPortal()` [\[#createportal\]](#)

```
createPortal(child, container)
```

Creates a portal. Portals provide a way to [render children into a DOM node that exists outside the hierarchy of the DOM component](#).

`flushSync()` [\[#flushsync\]](#)

```
flushSync(callback)
```

Force React to flush any updates inside the provided callback synchronously. This ensures that the DOM is updated immediately.

```
// Force this state update to be synchronous.
flushSync(() => {
  setCount(count + 1);
});
// By this point, DOM is updated.
```

Note:

`flushSync` can significantly hurt performance. Use sparingly.

`flushSync` may force pending Suspense boundaries to show their `fallback` state.

`flushSync` may also run pending effects and synchronously apply any updates they contain before returning.

`flushSync` may also flush updates outside the callback when necessary to flush the updates inside the callback. For example, if there are pending updates from a click, React may flush those before flushing the updates inside the callback.

Legacy Reference

`render()` [\[#render\]](#)

```
render(element, container[, callback])
```

Note:

`render` has been replaced with `createRoot` in React 18. See [createRoot](#) for more info.

Render a React element into the DOM in the supplied `container` and return a [reference](#) to the component (or returns `null` for [stateless components](#)).

주의

`render()` 는 전달한 컨테이너 노드의 콘텐츠를 제어합니다. 처음 호출할 때 기존의 DOM 엘리먼트를 교체하며 이후의 호출은 React의 DOM diffing 알고리즘을 사용하여 더욱 효율적으로 업데이트합니다.

`render()` 는 컨테이너 노드를 수정하지 않고 컨테이너의 하위 노드만 수정합니다. 그렇기 때문에 자식 노드를 덮어쓸 필요 없이 기존의 DOM 노드에 컴포넌트를 추가할 수 있습니다.

`render()` 는 현재 `ReactComponent` 루트(root) 인스턴스에 대한 참조를 반환합니다. 그러나 이 반환 값을 사용하는 것은 레거시이며 React 신규 버전이 컴포넌트를 비동기로 렌더링하는 경우가 있기 때문에 피해야 합니다. `ReactComponent` 인스턴스의 참조가 필요하다면 권장하는 해결책은 루트 엘리먼트에 [콜백 ref](#)를 붙이는 것입니다.

`render()` 를 사용해 서버에서 렌더링한 컨테이너에 이벤트를 보충하는 것은 권장되지 않으며 React 17 버전에서 삭제될 예정입니다. `hydrateRoot()` 를 사용해주세요.

hydrate() {#hydrate}

`hydrate(element, container[, callback])`

주의

`hydrate` 는 React 18에서 `hydrateRoot` 로 대체되었습니다. 자세한 내용은 [hydrateRoot](#)를 참조하십시오.

`render()` 와 동일하지만 HTML 콘텐츠가 `ReactDOMServer` 로 렌더링 된 컨테이너에 이벤트를 보충하기 위해 사용됩니다. React는 기존 마크업에 이벤트 리스너를 연결합니다.

React는 렌더링 된 콘텐츠가 서버와 클라이언트 간에 같을 것으로 예상합니다. React가 텍스트 콘텐츠의 차이를 고칠 수는 있지만 이러한 불일치를 버그로 취급하여 고쳐야 합니다. 개발 모드에서 React는 이벤트 보충 중 발생하는 불일치에 대해 경고합니다. 불일치가 발생하는 경우에 어트리뷰트 차이를 고친다는 보장이 없습니다. 대다수의 애플리케이션에서 불일치가 발생하는 경우는 많지 않으며 발생하는 경우 모든 마크업을 검증하는 것이 매우 큰 비용을 수반하기 때문에 성능상의 이유로 중요한 문제입니다.

서버와 클라이언트 사이에서 단일 엘리먼트의 어트리뷰트나 텍스트가 불가피하게 다르다면(예를 들어 timestamp의 경우) 그 엘리먼트에 `suppressHydrationWarning={true}` 를 추가하는 것으로 경고를 끌 수 있습니다. 이는 한 단계까지만 작동하며 의도된 해결책입니다. 절대 남용하지 마세요. 텍스트가 아니라면 React는 해당 엘리먼트를 고치지 않을 것이며 이후의 업데이트까지 일치하지 않은 채로 남아있을 것입니다.

서버와 클라이언트 간의 차이를 의도한다면 2단계 렌더링을 사용할 수 있습니다. 클라이언트에서 다르게 렌더링 되는 컴포넌트는 `componentDidMount()`에서 `true`로 설정할 수 있는 `this.state.isClient`와 같은 상태 변수를 읽을 수 있습니다. 이 방식으로 초기 렌더 단계는 서버와 같은 콘텐츠를 렌더링하여 불일치를 방지하지만, 이벤트 보충 직후에 추가적인 단계가 동기적으로 발생합니다. 이 방식은 컴포넌트를 두 번 렌더링하게 만들어 속도를 느리게 할 수 있기 때문에 주의를 기울여야 합니다.

느린 연결에서의 사용자 경험에 유의해야 합니다. JavaScript 코드는 최초 HTML 렌더링보다 매우 늦게 로드될 수 있으며 클라이언트 전용 단계에서 다른 무언가를 렌더링한다면 그 전환 과정에서 방해를 받을 수 있습니다. 그러나 정상적으로 실행된다면 서버에 애플리케이션 "shell"을 렌더링하고 클라이언트에서 일부 추가 위젯만 표시하는 것이 효과적일 수 있습니다. 마크업 불일치 문제없이 이 방식을 사용하기 원한다면 이전 단락의 설명을 참고해주세요.

unmountComponentAtNode() {#unmountcomponentatnode}

`unmountComponentAtNode(container)`

Note:

`unmountComponentAtNode` has been replaced with `root.unmount()` in React 18. See [createRoot](#) for more info.

마운트된 React 컴포넌트를 DOM에서 제거하고 컴포넌트의 이벤트 핸들러와 `state`를 정리합니다. 컨테이너에 아무런 컴포넌트도 마운트 되지 않았다면 해당 함수를 호출하더라도 아무런 동작을 하지 않습니다. 컴포넌트가 마운트 해제되었다면 `true`, 마운트 해제할 컴포넌트가 존재하지 않는다면 `false`를 반환합니다.

findDOMNode() {#finddomnode}

주의

`findDOMNode` 는 기본 DOM 노드를 이용하는 데에 사용되는 해결책입니다. 대부분의 경우에서 이 해결책을 사용하는 것은 컴포넌트 추상화를 위반하기 때문에 권장하지 않습니다. 이 메서드는 `StrictMode`에서 권장되지 않습니다.

`findDOMNode(component)`

매개 변수로 전달받은 컴포넌트가 DOM 내부에 마운트되었다면 컴포넌트에 해당하는 네이티브 브라우저의 DOM 엘리먼트를 반환합니다. 해당 메서드는 `form` 필드 값이나 DOM 성능 측정과 같이 DOM에서 값을 읽을 때 유용합니다. 대부분의 경우에서 **DOM 노드에 ref를 붙일 수 있으며 `findDOMNode`를 사용할 필요가 전혀 없습니다.**

컴포넌트가 `null`이나 `false`를 렌더링하는 경우 `findDOMNode`은 `null`을 반환합니다. 컴포넌트가 문자를 반환하는 경우 `findDOMNode`은 문자 값을 포함하고 있는 텍스트 DOM 노드를 반환합니다. React 16부터 컴포넌트는 여러 개의 자식을 가진 `fragment`를 반환할 수 있으며 이 경우에 `findDOMNode`는 비어있지 않은 첫 번째 자식에 해당하는 DOM 노드를 반환합니다.

주의

`findDOMNode` 는 마운트된 컴포넌트에만 동작합니다(즉 컴포넌트는 DOM 내부에 존재해야 합니다). 아직 마운트되지 않은 컴포넌트를 대상으로 해당 메서드를 호출한다면(예를 들어 아직 생성되지 않은 컴포넌트의 `render()`에서 `findDOMNode()`를 호출하는 것) exception이 발생합니다.

`findDOMNode` 은 함수 컴포넌트에서는 사용할 수 없습니다.

React 최상위 API

`React` 는 React 라이브러리의 진입점입니다. `<script>` 태그를 사용하여 React를 불러오게 되면 전역 객체 `React` 를 통하여 최상위 API를 사용할 수 있습니다. npm에서 ES6를 사용하는 경우, `import React from 'react'` 를 작성하면 됩니다. npm에서 ES5를 사용하는 경우, `var React = require('react')` 를 작성하면 됩니다.

개요

컴포넌트

React 컴포넌트를 사용하면 UI를 독립적이고 재사용할 수 있는 부분으로 나누고 각 부분을 분리하여 생각할 수 있습니다. React 컴포넌트는 `React.Component` 또는 `React.PureComponent` 로 세부적으로 나누어 정의할 수 있습니다.

- `React.Component`
- `React.PureComponent`

ES6 class를 사용하지 않는다면, `create-react-class` 모듈을 대신 사용해도 됩니다. 자세한 정보는 [ES6 없이 React를 사용하기](#) 문서에서 확인할 수 있습니다.

React 컴포넌트를 정의할 때 래핑될 수 있는 함수의 형태로 할 수도 있습니다.

- `React.memo`

React 엘리먼트 생성하기

UI의 형태를 설명하는 데에 [JSX를 사용할 것을 권장합니다.](#) 각 JSX 엘리먼트는 단지 `React.createElement()` 를 호출하는 편리한 문법에 불과합니다. JSX를 사용할 경우 아래의 메서드들을 직접 호출하는 일은 거의 없습니다.

- `createElement()`
- `createFactory()`

자세한 정보는 [JSX 없이 React 사용하기](#) 문서에서 확인할 수 있습니다.

엘리먼트 변환하기

`React` 는 엘리먼트를 조작하는 API들을 제공합니다.

- `cloneElement()`
- `isValidElement()`
- `React.Children`

Fragments

또한 `React` 는 래퍼 없이 여러 엘리먼트를 렌더링할 수 있는 컴포넌트를 제공합니다.

- `React.Fragment`

Refs

- `React.createRef`
- `React.forwardRef`

Suspense

`Suspense`를 사용하면 컴포넌트가 렌더링하기 전에 다른 작업이 먼저 이루어지도록 "대기합니다". 현재 `Suspense`는 단 하나의 사용 사례 `React.lazy`를 사용하여 컴포넌트를 동적으로 불러오기만 지원합니다. 나중에는 데이터 불러오기와 같은 사용 사례를 지원할 계획입니다.

- `React.lazy`
- `React.Suspense`

Transitions

`Transitions` are a new concurrent feature introduced in React 18. They allow you to mark updates as transitions, which tells React that they can be interrupted and avoid going back to Suspense fallbacks for already visible content.

- `React.startTransition`
- `React.useTransition`

Hooks

`Hooks`는 React 16.8에 새로 추가되었습니다. Hooks를 사용하면 class를 사용하지 않아도 state와 React 기능들을 사용할 수 있도록 해줍니다. Hooks만을 다루는 문서와 API 문서가 존재합니다.

- [기본적인 Hooks](#)
 - `useState`
 - `useEffect`
 - `useContext`
 - [추가적인 Hooks](#)
 - `useReducer`
 - `useCallback`
 - `useMemo`
 - `useRef`
 - `useImperativeHandle`
 - `useLayoutEffect`
 - `useDebugValue`
 - `useDeferredValue`
 - `useTransition`
 - `useId`
 - [Library Hooks](#)
 - `useSyncExternalStore`
 - `useInsertionEffect`
-

Reference

React.Component {#reactcomponent}

`React.Component` 는 [ES6 class](#)를 사용하여 React 컴포넌트를 정의할 때에 기초가 되는 `class`입니다.

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

`React.Component` 기초 `class`와 관련된 메서드와 속성 목록은 [React.Component API Reference](#)에서 확인할 수 있습니다.

React.PureComponent {#reactpurecomponent}

`React.PureComponent` 는 `React.Component` 와 비슷합니다. `React.Component` 는 `shouldComponentUpdate()` 를 구현하지 않지만, `React.PureComponent` 는 `props`와 `state`를 이용한 얇은 비교를 구현한다는 차이점만이 존재합니다.

React 컴포넌트의 `render()` 함수가 동일한 `props`와 `state`에 대하여 동일한 결과를 렌더링한다면, `React.PureComponent` 를 사용하여 경우에 따라 성능 향상을 누릴 수 있습니다.

주의

`React.PureComponent` 의 `shouldComponentUpdate()` 는 컴포넌트에 대하여 얇은 비교만을 수행합니다. 따라서 컴포넌트에 복잡한 자료 구조가 포함되어있다면, 깊은 차이가 존재함에도 불구하고 차이가 없다고 판단하는 잘못된 결과를 만들어낼 수 있습니다. `props`와 `state`의 구조가 간단할 것으로 예상될 때에만 `PureComponent` 를 상속하고, 깊은 자료 구조의 변화가 있다면 `forceUpdate()` 를 사용하세요. 또는 중첩된 데이터들을 빠르게 비교할 수 있도록 하려면 [불변 객체](#)의 사용을 검토해보세요.

더 나아가 `React.PureComponent` 의 `shouldComponentUpdate()` 는 컴포넌트의 하위 트리에 대한 `props` 갱신 작업을 수행하지 않습니다. 자식 컴포넌트들이 "순수"한지 꼭 확인하기 바랍니다.

React.memo {#reactmemo}

```
const MyComponent = React.memo(function MyComponent(props) {
  /* props를 사용하여 렌더링 */
});
```

`React.memo` 는 고차 컴포넌트(Higher Order Component)입니다.

컴포넌트가 동일한 `props`로 동일한 결과를 렌더링해낸다면, `React.memo` 를 호출하고 결과를 메모이징(Memoizing)하도록 래핑하여 경우에 따라 성능 향상을 누릴 수 있습니다. 즉, React는 컴포넌트를 렌더링하지 않고 마지막으로 렌더링된 결과를 재사용합니다.

`React.memo` 는 `props` 변화에만 영향을 줍니다. `React.memo` 로 감싸진 함수 컴포넌트 구현에 `useState`, `useReducer` 또는 `useContext` 흑을 사용한다면, 여전히 `state`나 `context`가 변할 때 다시 렌더링됩니다.

`props`가 갖는 복잡한 객체에 대하여 같은 비교만을 수행하는 것이 기본 동작입니다. 다른 비교 동작을 원한다면, 두 번째 인자로 별도의 비교 함수를 제공하면 됩니다.

```
function MyComponent(props) {
  /* props를 사용하여 렌더링 */
}

function areEqual(prevProps, nextProps) {
  /*
    nextProps가 prevProps와 동일한 값을 가지면 true를 반환하고, 그렇지 않다면 false를 반환
  */
}

export default React.memo(MyComponent, areEqual);
```

이 메서드는 오직 [성능 최적화](#)를 위하여 사용됩니다. 렌더링을 "방지"하기 위하여 사용하지 마세요. 버그를 만들 수 있습니다.

주의

`class` 컴포넌트의 `shouldComponentUpdate()` 메서드와 달리, `areEqual` 함수는 `props`들이 서로 같으면 `true`를 반환하고, `props`들이 서로 다르면 `false`를 반환합니다. 이것은 `shouldComponentUpdate` 와 정반대의 동작입니다.

createElement() [\[#createelement\]](#)

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

인자로 주어지는 태입에 따라 새로운 `React` 엘리먼트를 생성하여 반환합니다. `type` 인자로는 태그 이름 문자열(`'div'` 또는 `'span'` 등), `React` 컴포넌트 태입, 또는 `React Fragment` 태입 중 하나가 올 수 있습니다.

JSX로 작성된 코드는 `React.createElement()`를 사용하는 형태로 변환됩니다. JSX를 사용할 경우 `React.createElement()`를 직접 호출하는 일은 거의 없습니다. 자세한 정보는 [JSX 없이 React 사용하기](#) 문서에서 확인할 수 있습니다.

cloneElement() [\[#cloneelement\]](#)

```
React.cloneElement(
  element,
  [config],
  [...children]
)
```

`element`를 기준으로 새로운 `React` 엘리먼트를 복사하고 반환합니다. `config` 는 `key` 와 `ref` 그리고 모든 새로운 `props`를 포함합니다. 새로운 엘리먼트에는 원본 엘리먼트가 가졌던 `props`가 새로운 `props`와 얹게 합쳐진 뒤 주어집니다. 새로운 자식들은 기존의 자식들을 대체합니다. `config`에 `key` 와 `ref`가 없다면 원본 엘리먼트의 `key` 와 `ref`는 그대로 유지됩니다.

`React.cloneElement()` 는 아래의 구문과 거의 동등합니다.

```
<element.type {...element.props} {...props}>{children}</element.type>
```

그렇지만 `ref` 들이 유지된다는 점이 다릅니다. 즉 조상이 가지고 있을 `ref` 를 사용하여 자식 엘리먼트에 접근하는 것이 허용됩니다. 새로운 엘리먼트에 덧붙여지는 것과 동일한 `ref` 를 얻을 수 있습니다. 새로운 `ref` 또는 `key` 가 있다면 이전 값을 대체합니다.

이 API는 더 이상 사용되지 않는 `React.addons.cloneWithProps()` 를 대체합니다.

`createFactory()` **{#createfactory}**

```
React.createFactory(type)
```

주어진 타입의 React 엘리먼트를 만들어내는 함수를 반환합니다. `React.createElement()` 와 마찬가지로 `type` 인자는 태그 이름 문자열(`'div'` 또는 `'span'` 등), `React` 컴포넌트 타입, 또는 `React Fragment` 타입 중 하나가 올 수 있습니다.

이 헬퍼 함수는 레거시 기능으로 간주되며, 대신 JSX 문법을 사용하거나 `React.createElement()` 를 직접 사용하는 것이 좋습니다.

JSX를 사용할 경우 `React.createFactory()` 를 직접 호출하는 일은 거의 없습니다. 자세한 정보는 [JSX 없이 React 사용하기](#) 문서에서 확인할 수 있습니다.

`isValidElement()` **{#isvalidelement}**

```
React.isValidElement(object)
```

객체가 React 엘리먼트인지 확인합니다. `true` 또는 `false` 를 반환합니다.

`React.Children` **{#reactchildren}**

`React.Children` 는 불투명(Opaque) 자료 구조인 `this.props.children` 를 다루는 유ти리티 함수들을 제공합니다.

`React.Children.map` **{#reactchildrenmap}**

```
React.Children.map(children, function[(thisArg)])
```

`children` 에 포함된 각 자식에 대하여 `this` 를 `thisArg` 의 값으로 설정한 함수를 호출합니다. `children` 이 배열일 경우, 이 배열의 각 자식에 대하여 함수가 호출됩니다. `children` 이 `null` 또는 `undefined` 일 경우, 이 메서드는 배열이 아니라 `null` 또는 `undefined` 를 반환합니다.

주의

`children`이 `Fragment` 일 경우, `children` 은 단일 자식으로 취급되어 순회하지 않습니다.

`React.Children.forEach` **{#reactchildrenforeach}**

```
React.Children.forEach(children, function[(thisArg)])
```

`React.Children.map()` 와 비슷하지만, 배열을 반환하지 않습니다.

`React.Children.count` {#reactchildrencount}

`React.Children.count(children)`

`children`에 포함된 컴포넌트의 개수를 반환합니다. `map` 또는 `forEach`로 전달된 콜백이 호출된 횟수와 동일한 값입니다.

`React.Children.only` {#reactchildrenonly}

`React.Children.only(children)`

`children`이 단 하나의 자식(React 엘리먼트)를 갖는지 확인하고 해당 자식 엘리먼트를 반환합니다. 그렇지 않을 경우 오류를 발생시킵니다.

주의

`React.Children.only()`는 `React.Children.map()`의 반환값을 허용하지 않는데, 왜냐하면 반환값이 React 엘리먼트가 아니라 배열이기 때문입니다.

`React.Children.toArray` {#reactchildrentoarray}

`React.Children.toArray(children)`

각 자식에 `key`가 할당된 배열을 `children` 불투명(opaque) 자료 구조로 반환합니다. `render()` 메서드에서 `children`의 집합을 다루고 싶을 때, 특히 `this.props.children`을 하부로 전달하기 전에 다시 정렬하거나 일부만 잘라내고 싶을 때에 유용합니다.

주의

`React.Children.toArray()`는 `children`을 평평하게(Flatten) 만들 때, 중첩된 배열들의 의미를 보존하기 위하여 `key`를 변경합니다. 즉, `toArray`는 반환되는 배열에 `key` 값을 덧붙여서 각 엘리먼트가 갖는 `key`가 평평해진 배열 내에서만 유효한 범위를 형성하도록 해줍니다.

`React.Fragment` {#reactfragment}

`React.Fragment` 컴포넌트를 사용하면 `render()` 메서드 안에서 추가적인 DOM 엘리먼트를 생성하지 않아도 여러 엘리먼트를 반환할 수 있습니다.

```
render() {
  return (
    <React.Fragment>
      Some text.
      <h2>A heading</h2>
```

```
</React.Fragment>
);
```

축약형인 `<></>` 문법으로도 동일하게 사용할 수 있습니다. 자세한 정보는 [React v16.2.0: Fragment에 대한 향상된 지원](#)에서 확인할 수 있습니다.

React.createRef {#reactcreateref}

`React.createRef` 는 React 엘리먼트에 `ref` 어트리뷰트로 붙일 수 있는 `ref`를 생성합니다. [embed:16-3-release-blog-post/create-ref-example.js](#)

React.forwardRef {#reactforwardref}

`React.forwardRef` 는 전달받은 `ref` 어트리뷰트를 하부 트리 내의 다른 컴포넌트로 전달하는 React 컴포넌트를 생성합니다. 이 기법은 잘 사용되지 않지만, 아래의 두 시나리오에서는 특히 유용합니다.

- DOM 엘리먼트로 `ref` 전달하기
- 고차 컴포넌트(Higher Order Component)로 `ref` 전달하기

`React.forwardRef` 는 렌더링에 사용될 함수를 인자로 받을 수 있습니다. React는 이 함수를 두 개 인자 `props` 와 `ref`를 사용하여 호출하고, 이 함수는 React 노드를 반환합니다.

[embed:reference-react-forward-ref.js](#)

위의 예시에서 React는 `<FancyButton ref={ref}>` 엘리먼트에 주어진 `ref` 를 `React.forwardRef` 호출시 렌더링 함수에 2번째 인자로 전달합니다. 이 렌더링 함수는 `ref` 를 `<button ref={ref}>` 엘리먼트에 전달합니다.

따라서 React가 해당 `ref`를 붙이고 난 뒤, `ref.current` 는 `<button>` DOM 엘리먼트 인스턴스를 직접 가리키게 됩니다.

자세한 정보는 [ref 전달하기](#)에서 확인할 수 있습니다.

React.lazy {#reactlazy}

`React.lazy()` 를 사용하면 동적으로 불러오는 컴포넌트를 정의할 수 있습니다. 그러면 번들의 크기를 줄이고, 초기 렌더링에서 사용되지 않는 컴포넌트를 불러오는 작업을 지연시킬 수 있습니다.

사용 방법은 [Code Splitting](#) 문서에서 익힐 수 있습니다. 또한 사용 방법을 자세히 다룬 [이 글](#)을 읽어보는 것도 좋습니다.

```
// 이 컴포넌트는 동적으로 불러옵니다
const SomeComponent = React.lazy(() => import('./SomeComponent'));
```

`lazy` 한 컴포넌트를 렌더링하려면 렌더링 트리 상위에 `<React.Suspense>` 컴포넌트가 존재해야 한다는 점에 유의하세요. 이를 활용하여 로딩 지시기>Loading indicator를 나타낼 수 있습니다.

React.Suspense {#reactsuspense}

`React.Suspense` lets you specify the loading indicator in case some components in the tree below it are not yet ready to render. In the future we plan to let `Suspense` handle more scenarios such as data fetching. You can read about this in [our roadmap](#).

Today, lazy loading components is the **only** use case supported by `<React.Suspense>`:

```
// 이 컴포넌트는 동적으로 불러옵니다
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    // Displays <Spinner> until OtherComponent loads
    <React.Suspense fallback={<Spinner />}>
      <div>
        <OtherComponent />
      </div>
    </React.Suspense>
  );
}
```

관련된 내용을 [Code Splitting 가이드](#) 문서에서 설명하고 있습니다. `lazy` 한 컴포넌트는 `Suspense` 트리 내의 깊숙한 곳에 위치할 수 있다는 점에 유의하세요. 즉, `Suspense` 가 모든 컴포넌트를 감쌀 필요는 없다는 것입니다. 가장 좋은 사용법은 로딩 지시기를 보여주고 싶은 지점에 `<Suspense>` 를 작성하는 것이지만, Code Splitting을 하고자 하는 지점 어디서든지 `lazy()` 를 써야 할 것입니다.

Note

For content that is already shown to the user, switching back to a loading indicator can be disorienting. It is sometimes better to show the "old" UI while the new UI is being prepared. To do this, you can use the new transition APIs `startTransition` and `useTransition` to mark updates as transitions and avoid unexpected fallbacks.

`React.Suspense` in Server Side Rendering

During server side rendering Suspense Boundaries allow you to flush your application in smaller chunks by suspending. When a component suspends we schedule a low priority task to render the closest Suspense boundary's fallback. If the component unsuspends before we flush the fallback then we send down the actual content and throw away the fallback.

`React.Suspense` during hydration

Suspense boundaries depend on their parent boundaries being hydrated before they can hydrate, but they can hydrate independently from sibling boundaries. Events on a boundary before its hydrated will cause the boundary to hydrate at a higher priority than neighboring boundaries. [Read more](#)

`React.startTransition` `{#starttransition}`

`React.startTransition(callback)`

`React.startTransition` lets you mark updates inside the provided callback as transitions. This method is designed to be used when `React.useTransition` is not available.

Note:

Updates in a transition yield to more urgent updates such as clicks.

Updates in a transition will not show a fallback for re-suspended content, allowing the user to continue interacting while rendering the update.

`React.startTransition` does not provide an `isPending` flag. To track the pending status of a transition see `React.useTransition`.

테스트 렌더러

불러오기

```
import TestRenderer from 'react-test-renderer'; // ES6
const TestRenderer = require('react-test-renderer'); // npm에서 ES5를 사용하는 경우
```

개요

이 패키지는 DOM 혹은 네이티브 모바일 환경의 제약 없이, React 컴포넌트를 순수한 JavaScript 객체로 렌더링하는데 사용할 수 있는 React 렌더러를 제공합니다.

기본적으로, 이 패키지를 사용하면 브라우저나 `jsdom` 없이 React DOM 또는 React Native 컴포넌트에 의해 렌더링된 플랫폼 뷰 계층의 스냅샷을 쉽게 찍을 수 있도록 도와줍니다.

예시

```
import TestRenderer from 'react-test-renderer';

function Link(props) {
  return <a href={props.page}>{props.children}</a>;
}

const testRenderer = TestRenderer.create(
  <Link page="https://www.facebook.com/">Facebook</Link>
);

console.log(testRenderer.toJSON());
// { type: 'a',
//   props: { href: 'https://www.facebook.com/' },
//   children: [ 'Facebook' ] }
```

Jest의 스냅샷 테스트 기능으로 JSON 트리의 복사본을 파일로 자동 저장하여 테스트 내에서 변경되지 않았는지 확인할 수 있습니다. [자세히 알아보기](#)

또한, 결과물을 순회하며 특정 노드를 찾아 원하는 값을 가지고 있는지 검증하는 데 사용할 수 있습니다.

```
import TestRenderer from 'react-test-renderer';

function MyComponent() {
  return (
    <div>
      <SubComponent foo="bar" />
      <p className="my">Hello</p>
    </div>
  )
}

function SubComponent() {
  return (
    <p className="sub">Sub</p>
  );
}
```

```
const testRenderer = TestRenderer.create(<MyComponent />);
const testInstance = testRenderer.root;

expect(testInstance.findByName(SubComponent).props.foo).toBe('bar');
expect(testInstance.findByName({className: "sub"}).children).toEqual(['Sub']);
```

TestRenderer

- `TestRenderer.create()`
- `TestRenderer.act()`

TestRenderer instance

- `testRenderer.toJSON()`
- `testRenderer.toTree()`
- `testRenderer.update()`
- `testRenderer.unmount()`
- `testRenderer.getInstance()`
- `testRenderer.root`

TestInstance

- `testInstance.find()`
- `testInstance.findByName()`
- `testInstance.findByProps()`
- `testInstance.findAll()`
- `testInstance.findAllByName()`
- `testInstance.findAllByProps()`
- `testInstance.instance`
- `testInstance.type`
- `testInstance.props`
- `testInstance.parent`
- `testInstance.children`

참조

TestRenderer.create() `{#testrenderercreate}`

```
TestRenderer.create(element, options);
```

전달된 React 엘리먼트로 `TestRenderer` 인스턴스를 생성합니다. 실제 DOM을 사용하지 않지만, 컴포넌트 트리 전체를 메모리상에 렌더링하기 때문에 원하는 값을 가졌는지 검증할 수 있습니다. 반환된 인스턴스는 다음과 같은 함수와 속성을 가지고 있습니다.

TestRenderer.act() `{#testrendereract}`

```
TestRenderer.act(callback);
```

`react-dom/test-utils` 或 `act()` 와 비슷하게, `TestRenderer.act` 는 검증을 위한 컴포넌트들을 준비합니다. `TestRenderer.create` 와 `TestRenderer.update` 의 호출을 이 버전의 `act()` 를 사용해서 감싸주세요.

```

import {create, act} from 'react-test-renderer';
import App from './app.js'; // The component being tested

// 컴포넌트를 렌더링합니다.
let root;
act(() => {
  root = create(<App value={1}>/>)
});

// root를 검증합니다.
expect(root.toJSON()).toMatchSnapshot();

// 몇몇의 다른 props를 업데이트합니다.
act(() => {
  root.update(<App value={2}>/>);
})

// root를 검증합니다.
expect(root.toJSON()).toMatchSnapshot();

```

testRenderer.toJSON() **{#testrenderertojson}**

`testRenderer.toJSON()`

렌더링 된 트리를 나타내는 객체를 반환합니다. 이 트리는 같은 플랫폼 고유의 노드(예: `<div>`, `<View>`)와, 그러한 노드의 속성만을 가지고 있습니다. 사용자가 작성한 컴포넌트는 나타나지 않습니다. 이 함수는 [스냅샷 테스팅](#) 시 유용하게 사용할 수 있습니다.

testRenderer.toTree() **{#testrenderertotree}**

`testRenderer.toTree()`

렌더링 된 트리를 나타내는 객체를 반환합니다. `toJSON()`에서 반환되는 값보다 더욱 자세한 정보가 반환되며, 반환 값에는 사용자가 작성한 컴포넌트 역시 포함되어있습니다. 테스트 렌더러 위에 별도의 검증(assertion) 라이브러리를 만드는 것이 아니라면 이 함수는 필요하지 않을 것입니다.

testRenderer.update() **{#testrendererupdate}**

`testRenderer.update(element)`

메모리 내의 트리를 새로운 최상위 엘리먼트로 다시 렌더링합니다. 이 함수를 사용해 최상위 엘리먼트에서의 React 업데이트를 시뮬레이션할 수 있습니다. 새로운 엘리먼트가 이전 엘리먼트와 같은 태입과 키(key)를 가지고 있다면 트리를 업데이트합니다. 그렇지 않다면, 새로운 트리를 새로 마운트합니다.

testRenderer.unmount() **{#testrendererunmount}**

`testRenderer.unmount()`

메모리 내의 트리를 마운트 해제하고 적절한 생명주기 이벤트를 발생시킵니다.

testRenderer.getInstance() **{#testrenderergetinstance}**

`testRenderer.getInstance()`

최상위 엘리먼트에 대응하는 인스턴스가 존재하면 값을 반환합니다. 최상위 엘리먼트가 함수 컴포넌트일 경우, 함수 컴포넌트에는 인스턴스가 없기 때문에 작동하지 않습니다.

`testRenderer.root {#testrendererroot}`

`testRenderer.root`

트리 내에서 특정 노드를 검증할 때 유용한 최상위 "테스트 인스턴스" 객체를 반환합니다. 이것을 사용하여 더욱 깊이 있는 다른 "테스트 인스턴스"를 찾을 수 있습니다.

`testInstance.find() {#testinstancefind}`

`testInstance.find(test)`

`test(testInstance)` 에 대해 `true` 를 반환하는 단 하나의 자식 테스트 인스턴스를 찾아 반환합니다. 해당되는 인스턴스가 하나가 아니라면 오류를 반환합니다.

`testInstance.findByType() {#testinstancefindbytype}`

`testInstance.findByType(type)`

주어진 타입(type)에 해당하는 단 하나의 자식 테스트 인스턴스를 찾아 반환합니다. 해당되는 인스턴스가 하나가 아니라면 오류를 반환합니다.

`testInstance.findByProps() {#testinstancefindbyprops}`

`testInstance.findByProps(props)`

주어진 `props` 들에 해당하는 단 하나의 자식 테스트 인스턴스를 찾아 반환합니다. 해당되는 인스턴스가 하나가 아니라면 오류를 반환합니다.

`testInstance.findAll() {#testinstancefindall}`

`testInstance.findAll(test)`

`test(testInstance)` 에 대해 `true` 를 반환하는 모든 자식 테스트 인스턴스들을 찾아 반환합니다.

`testInstance.findAllByType() {#testinstancefindallbytype}`

`testInstance.findAllByType(type)`

주어진 타입(type)에 해당하는 모든 자식 테스트 인스턴스들을 찾아 반환합니다.

`testInstance.findAllByProps() {#testinstancefindallbyprops}`

`testInstance.findAllByProps(props)`

주어진 `props` 들에 해당하는 모든 자식 테스트 인스턴스들을 찾아 반환합니다.

testInstance.instance {#testinstanceinstance}

testInstance.instance

이 테스트 인스턴스에 해당하는 컴포넌트 인스턴스입니다. 함수 컴포넌트에는 인스턴스가 없기 때문에 클래스 컴포넌트에서만 사용할 수 있습니다. 주어진 컴포넌트 내부의 `this` 와 같습니다.

testInstance.type {#testinstancetype}

testInstance.type

이 테스트 인스턴스에 해당하는 컴포넌트의 타입입니다. 예를 들어, `<Button />` 컴포넌트의 타입은 `Button` 입니다.

testInstance.props {#testinstanceprops}

testInstance.props

이 테스트 인스턴스에 해당하는 컴포넌트의 props들입니다. 예를 들어, `<Button size="small" />` 컴포넌트는 `{size: 'small'}` 이라는 props들을 가지고 있습니다.

testInstance.parent {#testinstanceparent}

testInstance.parent

이 테스트 인스턴스의 부모입니다.

testInstance.children {#testinstancechildren}

testInstance.children

이 테스트 인스턴스의 자식들입니다.

Ideas

커스텀 모의 `ref`를 만들어주는 `createNodeMock` 함수를 `TestRenderer.create`에 추가로 넘길 수 있습니다. `createNodeMock` 은 현재의 엘리먼트를 받아 모의 `ref` 객체를 반환할 것입니다. 이것은 `ref`에 의존하는 컴포넌트를 테스트할 때 유용합니다.

```
import TestRenderer from 'react-test-renderer';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.input = null;
  }
  componentDidMount() {
    this.input.focus();
  }
  render() {
    return <input type="text" ref={el => this.input = el} />
  }
}
```

```
let focused = false;
TestRenderer.create(
  <MyComponent />,
{
  createNodeMock: (element) => {
    if (element.type === 'input') {
      // 모의 focus 함수 생성
      return {
        focus: () => {
          focused = true;
        }
      };
    }
    return null;
  }
);
expect(focused).toBe(true);
```

Ref와 DOM

Ref는 render 메서드에서 생성된 DOM 노드나 React 엘리먼트에 접근하는 방법을 제공합니다.

일반적인 React의 데이터 플로우에서 `props`는 부모 컴포넌트가 자식과 상호작용할 수 있는 유일한 수단입니다. 자식을 수정하려면 새로운 `props`를 전달하여 자식을 다시 렌더링해야 합니다. 그러나, 일반적인 데이터 플로우에서 벗어나 직접적으로 자식을 수정해야 하는 경우도 가끔씩 있습니다. 수정할 자식은 React 컴포넌트의 인스턴스일 수도 있고, DOM 엘리먼트일 수도 있습니다. React는 두 경우 모두를 위한 해결책을 제공합니다.

Ref를 사용해야 할 때

Ref의 바람직한 사용 사례는 다음과 같습니다.

- 포커스, 텍스트 선택영역, 혹은 미디어의 재생을 관리할 때.
- 애니메이션을 직접적으로 실행시킬 때.
- 서드 파티 DOM 라이브러리를 React와 같이 사용할 때.

선언적으로 해결될 수 있는 문제에서는 ref 사용을 지양하세요.

예를 들어, `Dialog` 컴포넌트에서 `open()` 과 `close()` 메서드를 두는 대신, `isOpen`이라는 prop을 넘겨주세요.

Ref를 남용하지 마세요

`ref`는 애플리케이션에 "어떤 일이 일어나게" 할 때 사용될 수도 있습니다. 그럴 때는 잠시 멈추고 어느 컴포넌트 계층에서 상태를 소유해야 하는지 신중하게 생각해보세요. 대부분의 경우, 상태를 소유해야 하는 적절한 장소가 더 높은 계층이라는 결론이 날 겁니다. 상태를 상위 계층으로 올리는 것에 대한 예시는 [상태 끌어올리기](#) 가이드에서 확인하실 수 있으십니다.

주의

아래에 있는 예시는 React 16.3에서 추가된 `React.createRef()` API를 사용하도록 수정되었습니다. 이전 버전의 React를 사용하고 계신다면, [콜백 ref](#)를 대신 사용하시는 편이 좋습니다.

Ref 생성하기

Ref는 `React.createRef()`를 통해 생성되고 `ref` 어트리뷰트를 통해 React 엘리먼트에 부착됩니다. 보통, 컴포넌트의 인스턴스가 생성될 때 Ref를 프로퍼티로서 추가하고, 그럼으로서 컴포넌트의 인스턴스의 어느 곳에서도 Ref에 접근할 수 있게 합니다.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
    return <div ref={this.myRef} />;
  }
}
```

Ref에 접근하기

`render` 메서드 안에서 `ref`가 엘리먼트에게 전달되었을 때, 그 노드를 향한 참조는 `ref`의 `current` 어트리뷰트에 담기게 됩니다.

```
const node = this.myRef.current;
```

`ref`의 값은 노드의 유형에 따라 다릅니다.

- `ref` 어트리뷰트가 HTML 엘리먼트에 쓰였다면, 생성자에서 `React.createRef()`로 생성된 `ref`는 자신을 전달받은 DOM 엘리먼트를 `current` 프로퍼티의 값으로서 받습니다.
- `ref` 어트리뷰트가 커스텀 클래스 컴포넌트에 쓰였다면, `ref` 객체는 마운트된 컴포넌트의 인스턴스를 `current` 프로퍼티의 값으로서 받습니다.
- 함수 컴포넌트는 인스턴스가 없기 때문에 함수 컴포넌트에 `ref` 어트리뷰트를 사용할 수 없습니다.

아래의 예시들은 위에서 언급한 차이점들을 보여줍니다.

DOM 엘리먼트에 Ref 사용하기

아래의 코드는 DOM 노드에 대한 참조를 저장하기 위해 `ref`를 사용합니다.

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // TextInput DOM 엘리먼트를 저장하기 위한 ref를 생성합니다.
    this.textInput = React.createRef();
    this.focusTextInput = this.focusTextInput.bind(this);
  }

  focusTextInput() {
    // DOM API를 사용하여 명시적으로 text 태입의 input 엘리먼트를 포커스합니다.
    // 주의: 우리는 지금 DOM 노드를 얻기 위해 "current" 프로퍼티에 접근하고 있습니다.
    this.textInput.current.focus();
  }

  render() {
    // React에게 우리가 text 태입의 input 엘리먼트를
    // 우리가 생성자에서 생성한 `textInput` ref와 연결하고 싶다고 이야기합니다.
    return (
      <div>
        <input
          type="text"
          ref={this.textInput} />
        <input
          type="button"
          value="Focus the text input"
          onClick={this.focusTextInput}
        />
      </div>
    );
  }
}
```

컴포넌트가 마운트될 때 React는 `current` 프로퍼티에 DOM 엘리먼트를 대입하고, 컴포넌트의 마운트가 해제될 때 `current` 프로퍼티를 다시 `null`로 돌려 놓습니다. `ref`를 수정하는 작업은 `componentDidMount` 또는 `componentDidUpdate` 생명주기 메서드가 호출되기 전에 이루어집니다.

클래스 컴포넌트에 `ref` 사용하기

아래에 있는 `CustomTextInput` 컴포넌트의 인스턴스가 마운트 된 이후에 즉시 클릭되는 걸 흉내내기 위해 `CustomTextInput` 컴포넌트를 감싸는 걸 원한다면, `ref`를 사용하여 `CustomTextInput` 컴포넌트의 인스턴스에 접근하고 직접 `focusTextInput` 메서드를 호출할 수 있습니다.

```
class AutoFocusTextInput extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }

  componentDidMount() {
    this.textInput.current.focusTextInput();
  }

  render() {
    return (
      <CustomTextInput ref={this.textInput} />
    );
  }
}
```

위 코드는 `CustomTextInput` 가 클래스 컴포넌트일 때에만 작동한다는 점을 기억하세요.

```
class CustomTextInput extends React.Component {
  // ...
}
```

Ref와 함수 컴포넌트

- 함수 컴포넌트는 인스턴스가 없기 때문에 **함수 컴포넌트에 `ref` 어트리뷰트를 사용할 수 없습니다.**

```
function MyFunctionComponent() {
  return <input />;
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }
  render() {
    // 이 코드는 동작하지 않습니다.
    return (
      <MyFunctionComponent ref={this.textInput} />
    );
  }
}
```

함수 컴포넌트에 `ref` 를 사용할 수 있도록 하려면, `forwardRef` (높은 확률로 `useImperativeHandle` 와 함께)를 사용하거나 클래스 컴포넌트로 변경할 수 있습니다.

다만, DOM 엘리먼트나 클래스 컴포넌트의 인스턴스에 접근하기 위해 `ref` 어트리뷰트를 함수 컴포넌트에서 사용하는 것은 됩니다.

```

function CustomTextInput(props) {
  // TextInput은 ref 어트리뷰트를 통해 전달되기 위해서
  // 이곳에서 정의되어야만 합니다.
  consttextInput = useRef(null);

  function handleClick() {
    textInput.current.focus();
  }

  return (
    <div>
      <input
        type="text"
        ref={textInput} />
      <input
        type="button"
        value="Focus the text input"
        onClick={handleClick}
      />
    </div>
  );
}

```

부모 컴포넌트에게 DOM ref를 공개하기

보기 드문 경우지만, 부모 컴포넌트에서 자식 컴포넌트의 DOM 노드에 접근하려 하는 경우도 있습니다. 자식 컴포넌트의 DOM 노드에 접근하는 것은 컴포넌트의 캡슐화를 파괴하기 때문에 권장되지 않습니다. 그렇지만 가끔마다 자식 컴포넌트의 DOM 노드를 포커스하는 일이나, 크기 또는 위치를 계산하는 일 등을 할 때에는 효과적인 방법이 될 수 있습니다.

[자식 컴포넌트에 ref를 사용할 수 있지만](#), 이 방법은 자식 컴포넌트의 인스턴스의 DOM 노드가 아닌 자식 컴포넌트의 인스턴스를 가져온다는 점에서, 자식 컴포넌트가 함수 컴포넌트인 경우에는 동작하지 않는다는 점에서, 좋은 방법이 아닙니다.

React 16.3 이후 버전의 React를 사용하신다면 위와 같은 경우에서 [ref 전달하기\(ref forwarding\)](#)을 사용하는 것이 권장됩니다.

[Ref 전달하기는 컴포넌트가 자식 컴포넌트의 ref를 자신의 ref로서 외부에 노출시키게 합니다.](#) 자식 컴포넌트의 DOM 노드를 부모 컴포넌트에게 공개하는 방법에 대한 자세한 예시는 [ref 넘겨주기 문서에서 볼 수 있습니다.](#)

React 16.2 이전 버전을 사용하시거나 ref 전달하기보다 더 유연한 방법을 원한다면 [이런 대안을 사용할 수 있습니다.](#)

가능하다면 DOM 노드를 외부에 공개하는 일을 지양해야 합니다만 DOM 노드를 외부에 공개하는 일은 유용한 해결책이 될 수 있습니다. 또한 이 방법들은 자식 컴포넌트의 코드 수정을 요한다는 점을 기억하세요. 자식 컴포넌트의 코드를 수정할 수 없다면 최후의 방법인 `findDOMNode()`를 사용하는 방법이 있지만 `findDOMNode()`는 좋지 못한 방법일 뿐더러 `StrictMode`에서 사용할 수 없습니다.

콜백 ref

React는 ref가 설정되고 해제되는 상황을 세세하게 다룰 수 있는 "콜백 ref" 이라 불리는 ref를 설정하기 위한 또 다른 방법을 제공합니다.

콜백 ref를 사용할 때에는 `ref` 어트리뷰트에 `React.createRef()`를 통해 생성된 `ref` 를 전달하는 대신, 함수를 전달합니다. 전달된 함수는 다른 곳에 저장되고 접근될 수 있는 React 컴포넌트의 인스턴스나 DOM 엘리먼트를 인자로서 받습니다.

아래의 예시는 DOM 노드의 참조를 인스턴스의 프로퍼티에 저장하기 위해 `ref` 콜백을 사용하는 흔한 패턴을 보여줍니다.

```

class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);

    this.textInput = null;

    this.setTextInputRef = element => {
      this.textInput = element;
    };

    this.focusTextInput = () => {
      // DOM API를 사용하여 text 태입의 input 엘리먼트를 포커스합니다.
      if (this.textInput) this.textInput.focus();
    };
  }

  componentDidMount() {
    // 마운트 되었을 때 자동으로 text 태입의 input 엘리먼트를 포커스합니다.
    this.focusTextInput();
  }

  render() {
    // text 태입의 input 엘리먼트의 참조를 인스턴스의 프로퍼티
    // (예를 들어 `this.textInput`)에 저장하기 위해 `ref` 콜백을 사용합니다.
    return (
      <div>
        <input
          type="text"
          ref={this.setTextInputRef}
        />
        <input
          type="button"
          value="Focus the text input"
          onClick={this.focusTextInput}
        />
      </div>
    );
  }
}

```

컴포넌트의 인스턴스가 마운트 될 때 React는 `ref` 콜백을 DOM 엘리먼트와 함께 호출합니다. 그리고 컴포넌트의 인스턴스의 마운트가 해제될 때, `ref` 콜백을 `null`과 함께 호출합니다. `ref` 콜백들은 `componentDidMount` 또는 `componentDidUpdate`가 호출되기 전에 호출됩니다.

콜백 `ref` 또한 `React.createRef()`를 통해 생성했던 객체 `ref`와 같이 다른 컴포넌트에게 전달할 수 있습니다.

```

function CustomTextInput(props) {
  return (
    <div>
      <input ref={props.inputRef} />
    </div>
  );
}

class Parent extends React.Component {
  render() {
    return (
      <CustomTextInput
        inputRef={el => this.inputElement = el}
      />
    );
  }
}

```

```
    );
}
```

위의 예시에서 `Parent`는 자신의 콜백 `ref`를 `inputRef` prop으로서 `CustomTextInput`에게 전달합니다. 그리고 `CustomTextInput`은 전달받은 함수를 `<input>`에게 `ref` 어트리뷰트로서 전달합니다. 결과적으로 `Parent`에 있는 `thisInputElement`는 `CustomTextInput`의 `<input>` 엘리먼트에 대응하는 DOM 노드가 됩니다.

레거시 API: 문자열 ref

React를 이전에 사용해 보셨다면 `ref` 어트리뷰트의 값이 `"textInput"`처럼 문자열이고, DOM 노드를 `this.refs.textInput`와 같이 접근하는 구식 API를 아시고 계실지도 모릅니다. 문자열 `ref`는 [몇몇 문제](#)를 가지고 있고, 레거시로 여겨지며, [차후 배포에서 삭제될 것으로 예상되기 때문에 권장되지 않습니다](#).

주의

`ref`에 접근하기 위해 `this.refs.textInput`를 사용하고 계신다면, [콜백 ref](#)이나 [createRef API](#)를 대신 사용하는 것을 권해 드립니다.

콜백 ref에 관한 주의사항

`ref` 콜백이 인라인 함수로 선언되었다면 `ref` 콜백은 업데이트 과정 중에 처음에는 `null`로, 그 다음에는 DOM 엘리먼트로, 총 두 번 호출됩니다. 이러한 현상은 매 렌더링마다 `ref` 콜백의 새 인스턴스가 생성되므로 React가 이전에 사용된 `ref`를 제거하고 새 `ref`를 설정해야 하기 때문에 일어납니다. 이러한 현상은 `ref` 콜백을 클래스에 바인딩된 메서드로 선언함으로써 해결할 수 있습니다. 하지만 많은 경우 이러한 현상은 문제가 되지 않는다는 점을 기억하세요.

배포 채널

React는 버그 신고, 풀 리퀘스트 요청, [RFCs에 제출](#)하기 위해서 번성하고 있는 오픈 소스 커뮤니티에 의존한다. 피드백을 장려하기 위해서 우리는 때때로 배포되지 않은 기능을 포함하고 있는 특정 React 빌드를 공유한다.

이 문서는 프레임워크, 라이브러리, 개발자 도구에 관해 작업하는 개발자들과 가장 관련이 있습니다. 주로 React를 사용해서 사용자용 애플리케이션을 빌드하는 개발자는 prerelease 채널에 대해서 걱정할 필요가 없습니다.

React의 각 배포 채널은 하나의 고유한 사용 경우를 위해 설계되었습니다.

- **Latest** 는 안정적이고, 유의적인 React 배포입니다. npm에서 React를 설치할 때 얻는 것입니다. 이것은 이미 여러분이 사용하고 있는 채널입니다. 모든 사용자용 애플리케이션을 위해서는 이것을 사용해주세요.
- **Next** 는 React 소스 코드 저장소의 main branch를 추적합니다. 이것을 다음 minor 유의적인 배포를 위한 배포 후보자라고 생각하세요. 이것을 React와 타사 프로젝트 간의 통합 테스트에 사용해주세요.
- **Experimental** 는 실험용 API 및 stable 배포에서는 사용할 수 없는 기능이 포함됩니다. 이것은 또한 main branch를 추적하지만, 추가 기능 플래그가 켜져 있습니다. 배포하기 전에 배포가 예정된 기능들을 실험하는데 사용해주세요.

모든 배포는 npm에 게시되지만 오직 Latest만 의미론적 버전 관리를 사용합니다. Prereleases는 (Next와 Experimental 채널에 있는 것) 내용의 hash와 커밋 날짜로부터 생성된 버전들을 가집니다, 예: Next를 위한 0.0.0-68053d940-20210623 와 Experimental을 위한 0.0.0-experimental-68053d940-20210623 .

사용자용 애플리케이션에 대해 공식적으로 지원되는 배포 채널은 **Latest**입니다. Next와 Experimental 배포는 테스트 목적으로만 제공되며 배포 간에 동작이 변경되지 않는다는 보장을 제공하지 않습니다. 그것들은 Latest의 배포에 사용하는 유의적 버전원칙 프로토콜을 따르지 않습니다.

안정적인 배포에 사용하는 것과 동일한 레지스트리에 prereleases를 게시함으로써, 우리는 [unpkg](#), [CodeSandbox](#)와 같은 npm workflow를 지원하는 많은 도구를 이용할 수 있습니다.

Latest 채널

Latest는 stable React 배포에 사용되는 채널입니다. npm의 `latest` tag에 해당합니다. 실제 사용자들에게 제공되는 모든 React app에 권장되는 채널입니다.

어떤 채널을 사용해야 할지 잘 모르겠다면 **Latest**를 사용해야 합니다. React 개발자라면, 이미 이것을 사용하고 있을 것입니다.

Latest로 업데이트가 매우 안정적이라고 기대할 수 있습니다. 버전은 의미론적 버전 정책 체계를 따릅니다. [버전 정책](#)에서 안정성 및 incremental migration에 대한 우리의 노력에 대해 배워 보세요.

Next 채널

Next 채널은 React 저장소의 main branch를 추적하는 prerelease 채널입니다. 우리는 Latest 채널의 배포 후보로서 Next 채널의 prerelease를 사용합니다. Next 채널을 더 자주 업데이트되는 Latest 채널의 상위 집합으로 생각하면 됩니다.

가장 최신 Next 배포와 가장 최신 Latest 배포 사이의 변화 정도는 두 개의 부 유의적 버전 배포사이의 변화 정도와 거의 같습니다. 그러나, **Next** 채널은 유의적 버전 원칙을 따르지 않습니다. Next 채널에서 연속적인 배포 사이의 간헐적 주요 변경사항을 예상해야 합니다.

사용자용 애플리케이션에서 **prereleases**를 사용하지 마세요.

npm에서 Next에서 배포는 `next` tag와 함께 게시됩니다. 버전은 빌드의 내용의 해시와 커밋 날짜로부터 생성됩니다. 예: `0.0.0-68053d940-20210623`.

통합 테스트를 위해 Next 채널을 사용

Next 채널은 React와 다른 프로젝트 간 통합 테스트를 지원하기 위해 설계되었습니다.

React의 모든 변경사항들은 배포 전에 광범위한 내부 테스트를 거칩니다. 그러나 React 생태계 전체에서 사용되는 무수히 많은 환경과 구성이 있고 우리가 일일이 이 모든 것에 관해 테스트하는 것은 불가능합니다.

타사 React 프레임워크, 라이브러리, 개발자 도구 또는 유사 인프라 구조 타입 프로젝트의 작성자라면 가장 최근 변화들에 대한 테스트 suite들을 주기적으로 실행하여 사용자들과 React 커뮤니티에 도움을 줄 수 있습니다. 흥미가 있다면, 이 단계를 따르세요.

- 선호하는 지속적 통합 플랫폼을 사용해서 cron job을 설정하세요. Cron jobs는 [CircleCI](#) 와 [Travis CI](#) 모두에서 지원됩니다.
- cron job에서, npm의 `next` 태그를 사용해서 React 패키지를 Next 채널에서 가장 최신 React 배포로 업데이트해주세요. npm cli를 사용해서

```
npm update react@next react-dom@next
```

또는 yarn을 사용해서

```
yarn upgrade react@next react-dom@next
```

- 업데이트된 패키지에 대해 테스트 suite를 실행해주세요.
- 모든 것이 통과된다면 최고입니다! 프로젝트가 다음 minor React 배포와 함께 작동할 것으로 예상 할 수 있습니다.
- 예기치 않은 문제가 발생한 경우 [이슈를 생성](#)해서 알려주세요.

이 workflow를 사용하는 프로젝트는 Next.js입니다. (말장난이 아닙니다! 진지하게!) [CircleCI configuration](#)을 예를 들어 참조할 수 있습니다.

Experimental 채널

Next와 마찬가지로 Experimental 채널은 React 저장소의 main branch를 추적하는 prerelease 채널입니다. Next와 달리, Experimental 배포는 광범위한 배포를 위해 준비되지 않은 추가 기능과 API를 포함합니다.

일반적으로, Next에 대한 업데이트는 Experimental에 대한 해당 업데이트와 함께 동반됩니다. 그것들은 동일한 소스 수정을 기반으로 하지만 다른 기능 플래그 세트를 사용하여 빌드됩니다.

Experimental 배포는 Next 및 Latest 배포와 크게 다를 수 있습니다. 사용자용 애플리케이션에서는 **Experimental** 배포를 사용하지 마세요. Experimental 채널에서 배포 사이에 빈번한 주요 변경사항을 예상해야 합니다.

Experimental에서 배포는 npm에서 `experimental` tag와 함께 게시됩니다. 버전은 빌드의 내용과 커밋 날짜로부터 생성됩니다. 예: `0.0.0-experimental-68053d940-20210623`.

Experimental 배포에는 무엇이 포함됩니까?

Experimental 기능은 더 공적으로 배포될 준비가 되지 않은 기능이며 최종적으로 배포되기 전에 크게 변경될 수 있습니다. 일부 실험은 최종 배포가 절대 되지 않는 경우도 있습니다. — 실험을 하는 이유는 제안된 변경사항의 실행 가능성을 테스트하기 위한 것입니다.

예를 들어서, Hooks를 발표할 때 Experimental 채널이 존재했다면, Latest 채널에서 사용할 수 있기 때문에 몇 주 전에 Hooks를 Experimental 채널에 배포했을 것입니다.

Experimental에 대해 통합 테스트를 실행하는 것이 유용하다는 것을 알 수도 있습니다. 그것은 여러분에게 달려 있습니다. 그러나 Experimental은 Next보다 안정성이 훨씬 떨어집니다. 우리는 **Experimental 배포 간의 어떤 안정성을 보장하지 않습니다.**

Experimental 기능에 대해 더 배우고 싶다면 어떻게 해야 합니까?

Experimental 기능은 문서화될 수도 있고 되지 않을 수도 있습니다. 일반적으로 실험은 Next 나 Latest로 이동에 가까워지기 전까지 문서화되지 않습니다.

기능이 문서화되지 않는 경우 [RFC](#)와 함께 동반될 수 있습니다.

우리는 새로운 실험을 발표할 준비가 되면 [React blog](#)에 게시물을 포스팅할 예정이지만, 모든 실험이 포스팅된다는 말은 아닙니다.

언제든지 우리의 public GitHub 저장소에서 전반적인 변경 사항 목록[history](#)를 참조 할 수 있습니다.

[Go to TOC](#)

Render Props

"render prop"란, React 컴포넌트 간에 코드를 공유하기 위해 함수 props를 이용하는 간단한 테크닉입니다.

render props 패턴으로 구현된 컴포넌트는 자체적으로 렌더링 로직을 구현하는 대신, react 엘리먼트 요소를 반환하고 이를 호출하는 함수를 사용합니다.

```
<DataProvider render={data => (
  <h1>Hello {data.target}</h1>
)}>
```

render props를 사용하는 라이브러리는 [React Router](#), [Downshift](#), [Formik](#)이 있습니다.

이 문서에서는 render props가 왜 유용하고, 어떻게 여러분의 프로젝트에 적용할 수 있을지에 관해 이야기 하겠습니다.

횡단 관심사(Cross-Cutting Concerns)를 위한 render props 사용법

컴포넌트는 React에서 코드의 재사용성을 위해 사용하는 주요 단위입니다. 하지만 컴포넌트에서 캡슐화된 상태나 동작을 같은 상태를 가진 다른 컴포넌트와 공유하는 방법이 항상 명확하지는 않습니다.

예를 들면, 아래 컴포넌트는 웹 애플리케이션에서 마우스 위치를 추적하는 로직입니다.

```
class MouseTracker extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        <h1>Move the mouse around!</h1>
        <p>The current mouse position is ({this.state.x}, {this.state.y})</p>
      </div>
    );
  }
}
```

스크린 주위로 마우스 커서를 움직이면, 컴포넌트가 마우스의 (x, y) 좌표를 `<p>`에 나타냅니다.

여기서 질문입니다. 다른 컴포넌트에서 이 행위를 재사용하려면 어떻게 해야 할까요? 즉, 다른 컴포넌트에서 커서(cursor) 위치에 대해 알아야 할 경우, 쉽게 공유할 수 있도록 캡슐화할 수 있습니까?

React에서 컴포넌트는 코드 재사용의 기본 단위이므로, 우리가 필요로 하는 마우스 커서 트래킹 행위를 `<Mouse>` 컴포넌트로 캡슐화하여 어디서든 사용할 수 있게 리팩토링 해보겠습니다.

```
// <Mouse> 컴포넌트는 우리가 원하는 행위를 캡슐화 합니다...
class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        {/* ...하지만 <p>가 아닌 다른것을 렌더링하려면 어떻게 해야 할까요? */}
        <p>The current mouse position is ({this.state.x}, {this.state.y})</p>
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <>
        <h1>Move the mouse around!</h1>
        <Mouse />
      </>
    );
  }
}
```

이제 `<Mouse>` 컴포넌트는 마우스 움직임 이벤트를 감지하고, 마우스 커서의 `(x, y)` 위치를 저장하는 행위를 캡슐화했습니다. 그러나 아직 완벽하게 재사용할 수 있는 건 아닙니다.

예를 들어, 마우스 주위에 고양이 그림을 보여주는 `<Cat>` 컴포넌트를 생각해 보겠습니다. 우리는 `<Cat mouse={{x, y}}>` prop을 통해 Cat 컴포넌트에 마우스 좌표를 전달해주고 화면에 어떤 위치에 이미지를 보여줄지 알려 주고자 합니다.

첫 번째 방법으로는, 다음과 같이 `<Mouse>` 컴포넌트의 `render` 메서드안에 `<Cat>` 컴포넌트를 넣어 렌더링하는 방법이 있습니다.

```
class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      
    );
  }
}

class MouseWithCat extends React.Component {
```

```

constructor(props) {
  super(props);
  this.handleMouseMove = this.handleMouseMove.bind(this);
  this.state = { x: 0, y: 0 };
}

handleMouseMove(event) {
  this.setState({
    x: event.clientX,
    y: event.clientY
  });
}

render() {
  return (
    <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
      /*
        여기서 <p>를 <Cat>으로 바꿀 수 있습니다. ... 그러나 이 경우
        Mouse 컴포넌트를 사용할 때 마다 별도의 <MouseWithSomethingElse>
        컴포넌트를 만들어야 합니다, 그러므로 <MouseWithCat>는
        아직 정말로 재사용이 가능한게 아닙니다.
      */
      <Cat mouse={this.state} />
    </div>
  );
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <MouseWithCat />
      </div>
    );
  }
}

```

이러한 접근 방법은 특정 사례에서는 적용할 수 있지만, 우리가 원하는 행위의 캡슐화(마우스 트래킹)라는 목표는 달성하지 못했습니다. 이제 우리는 다른 사용 예시에서도 언제든지 마우스 위치를 추적할 수 있는 새로운 component(`<MouseWithCat>`과 근본적으로 다른)를 만들어야 합니다.

여기서 render prop를 사용할 수 있습니다. `<Mouse>` 컴포넌트 안에 `<Cat>` 컴포넌트를 하드 코딩(hard-coding)해서 결과물을 바꾸는 대신에, `<Mouse>`에게 동적으로 렌더링할 수 있도록 해주는 함수 prop을 제공하는 것입니다.—이것이 render prop의 개념입니다.

```

class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      
    );
  }
}

class Mouse extends React.Component {

```

```

constructor(props) {
  super(props);
  this.handleMouseMove = this.handleMouseMove.bind(this);
  this.state = { x: 0, y: 0 };
}

handleMouseMove(event) {
  this.setState({
    x: event.clientX,
    y: event.clientY
  });
}

render() {
  return (
    <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
      /*
        <Mouse>가 무엇을 렌더링하는지에 대해 명확히 코드로 표기하는 대신,
        `render` prop을 사용하여 무엇을 렌더링할지 동적으로 결정할 수 있습니다.
      */
      {this.props.render(this.state)}
    </div>
  );
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <Mouse render={mouse => (
          <Cat mouse={mouse} />
        )}/>
      </div>
    );
  }
}

```

이제 컴포넌트의 행위를 복제하기 위해 하드 코딩할 필요 없이 `render` 함수에 prop으로 전달해줌으로써, `<Mouse>` 컴포넌트는 동적으로 트래킹 기능을 가진 컴포넌트들을 렌더링할 수 있습니다.

정리하자면, **render prop은 무엇을 렌더링할지 컴포넌트에 알려주는 함수입니다.**

이 테크닉은 행위(마우스 트래킹 같은)를 매우 쉽게 공유할 수 있도록 만들어 줍니다. 해당 행위를 적용하려면, `<Mouse>` 를 그리고 현재 (x , y) 커서 위치에 무엇을 그릴지에 대한 정보를 prop을 통해 넘겨주기만 하면 됩니다.

`render props`에 대해 한가지 흥미로운 점은 대부분의 **higher-order components**(HOC)에 `render props pattern`을 이식할 수 있습니다. 예를 들면, `<Mouse>` 컴포넌트보다 `withMouse HOC`를 더 선호한다면 `render prop`을 이용해서 다음과 같이 쉽게 HOC를 만들 수 있습니다.

```

// 어떤 이유 때문에 HOC를 만들기 원한다면, 쉽게 구현할 수 있습니다.
// render prop을 이용해서 일반적인 컴포넌트를 만드세요!
function withMouse(Component) {
  return class extends React.Component {
    render() {
      return (
        <Mouse render={mouse => (

```

```

        <Component {...this.props} mouse={mouse} />
      )}>
    );
}
}

```

따라서 render props를 사용하면 두 가지 패턴 모두 사용이 가능합니다.

render 이외의 Props 사용법

여기서 중요하게 기억해야 할 것은, “render props pattern”으로 불리는 이유로 꼭 prop name으로 render를 사용할 필요는 없습니다. 사실, 어떤 함수형 prop이든 render prop이 될 수 있습니다.

위 예시에서는 render를 사용했지만, 우리는 children prop을 더 쉽게 사용할 수 있습니다.

```

<Mouse children={mouse => (
  <p>The mouse position is {mouse.x}, {mouse.y}</p>
)}>

```

실제로 JSX element의 “어트리뷰트”목록에 하위 어트리뷰트 이름(예를들면 render)을 지정할 필요는 없습니다. 대신에, element 안에 직접 꽂아넣을 수 있습니다!

```

<Mouse>
  {mouse => (
    <p>The mouse position is {mouse.x}, {mouse.y}</p>
  )}
</Mouse>

```

이 테크닉은 react-motion API에서 실제로 사용된 것을 볼 수 있습니다.

이 테크닉은 자주 사용되지 않기 때문에, API를 디자인할 때 children은 함수 타입을 가지도록 propTypes를 지정하는 것이 좋습니다.

```

Mouse.propTypes = {
  children: PropTypes.func.isRequired
};

```

주의사항

React.PureComponent에서 render props pattern을 사용할 땐 주의해주세요.

render props 패턴을 사용하면 React.PureComponent를 사용할 때 발생하는 이점이 사라질 수 있습니다. 같은 prop 비교는 새로운 prop에 대해 항상 false를 반환합니다. 이 경우 render마다 render prop으로 넘어온 값을 항상 새로 생성합니다.

위에서 사용했던 <Mouse> 컴포넌트를 이용해서 예를 들어보겠습니다. mouse에 React.Component 대신에 React.PureComponent를 사용하면 다음과 같은 코드가 됩니다.

```

class Mouse extends React.PureComponent {
  // 위와 같은 구현체...
}

class MouseTracker extends React.Component {

```

```

render() {
  return (
    <div>
      <h1>Move the mouse around!</h1>

      {/* 이것은 좋지 않습니다! `render` prop이 가지고 있는 값은
         각각 다른 컴포넌트를 렌더링 할 것입니다.
      */}
      <Mouse render={mouse => (
        <Cat mouse={mouse} />
      )}>
    </div>
  );
}
}

```

이 예시에서 `<MouseTracker>` 가 `render` 될때마다, `<Mouse render>` 의 `prop`으로 넘어가는 함수가 계속 새로 생성됩니다. 따라서 `React.PureComponent` 를 상속받은 `<Mouse>` 컴포넌트 효과가 사라지게 됩니다.

이 문제를 해결하기 위해서, 다음과 같이 인스턴스 메서드를 사용해서 `prop`을 정의합니다.

```

class MouseTracker extends React.Component {
  // `this.renderTheCat`를 항상 생성하는 매서드를 정의합니다.
  // 이것은 render를 사용할 때 마다 *같은* 함수를 참조합니다.
  renderTheCat(mouse) {
    return <Cat mouse={mouse} />;
  }

  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <Mouse render={this.renderTheCat} />
      </div>
    );
  }
}

```

`prop`을 정적으로 정의할 수 없는 경우에는 `<Mouse>` 컴포넌트는 `React.Component` 를 상속받아야 합니다.

엘리먼트 렌더링

엘리먼트는 React 앱의 가장 작은 단위입니다.

엘리먼트는 화면에 표시할 내용을 기술합니다.

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

브라우저 DOM 엘리먼트와 달리 React 엘리먼트는 일반 객체이며(plain object) 쉽게 생성할 수 있습니다. React DOM은 React 엘리먼트와 일치하도록 DOM을 업데이트합니다.

주의

더 널리 알려진 개념인 "컴포넌트"와 엘리먼트를 혼동할 수 있습니다. [다음 장](#)에서 컴포넌트에 대해 소개할 예정입니다. 엘리먼트는 컴포넌트의 "구성 요소"이므로 이번 장을 읽고 나서 다음 장으로 넘어갈 것을 권합니다.

DOM에 엘리먼트 렌더링하기

HTML 파일 어딘가에 `<div>` 가 있다고 가정해 봅시다.

```
<div id="root"></div>
```

이 안에 들어가는 모든 엘리먼트를 React DOM에서 관리하기 때문에 이것을 "루트(root)" DOM 노드라고 부릅니다.

React로 구현된 애플리케이션은 일반적으로 하나의 루트 DOM 노드가 있습니다. React를 기존 앱에 통합하려는 경우 원하는 만큼 많은 수의 독립된 루트 DOM 노드가 있을 수 있습니다.

React 엘리먼트를 렌더링 하기 위해서는 우선 DOM 엘리먼트를 `ReactDOM.createRoot()`에 전달한 다음, React 엘리먼트를 `root.render()`에 전달해야 합니다.

```
embed:rendering-elements/render-an-element.js
```

CodePen에서 실행하기

위 코드를 실행하면 화면에 "Hello, world"가 보일 겁니다.

렌더링 된 엘리먼트 업데이트하기

React 엘리먼트는 [불변객체](#)입니다. 엘리먼트를 생성한 이후에는 해당 엘리먼트의 자식이나 속성을 변경할 수 없습니다. 엘리먼트는 영화에서 하나의 프레임과 같이 특정 시점의 UI를 보여줍니다.

지금까지 소개한 내용을 바탕으로 하면 UI를 업데이트하는 유일한 방법은 새로운 엘리먼트를 생성하고 이를 `root.render()`로 전달하는 것입니다.

예시로 똑딱거리는 시계를 살펴보겠습니다.

```
embed:rendering-elements/update-rendered-element.js
```

[CodePen에서 실행하기](#)

위 함수는 `setInterval()` 콜백을 이용해 초마다 `root.render()` 를 호출합니다.

주의

실제로 대부분의 React 앱은 `root.render()` 를 한 번만 호출합니다. 다음 장에서는 이와 같은 코드가 [유상태 컴포넌트](#)에 어떻게 캡슐화되는지 설명합니다.

각 주제가 서로 연관이 있기 때문에 건너뛰지 않는 것을 추천합니다.

변경된 부분만 업데이트하기

React DOM은 해당 엘리먼트와 그 자식 엘리먼트를 이전의 엘리먼트와 비교하고 DOM을 원하는 상태로 만드는데 필요한 경우에만 DOM을 업데이트합니다.

개발자 도구를 이용해 [마지막 예시](#)를 살펴보면 이를 확인할 수 있습니다.

Hello, world!

It is 12:26:46 PM.

```

Console Sources Network Timeline
▼<div id="root">
  ▼<div data-reactroot>
    <h1>Hello, world!</h1>
    ▼<h2>
      <!-- react-text: 4 -->
      "It is "
      <!-- /react-text -->
      <!-- react-text: 5 -->
      "12:26:46 PM"
      <!-- /react-text -->
      <!-- react-text: 6 -->
      "."
      <!-- /react-text -->
    </h2>
  </div>
</div>

```

DOM inspector showing granular updates

매초 전체 UI를 다시 그리도록 엘리먼트를 만들었지만 React DOM은 내용이 변경된 텍스트 노드만 업데이트했습니다.

경험에 비추어 볼 때 특정 시점에 UI가 어떻게 보일지 고민하는 이런 접근법은 시간의 변화에 따라 UI가 어떻게 변화할지 고민하는 것보다 더 많은 수의 버그를 없앨 수 있습니다.

[Go to TOC](#)

State and Lifecycle

이 페이지는 React 컴포넌트 안의 state와 생명주기에 대한 개념을 소개해 줍니다. [자세한 컴포넌트 API 레퍼런스는 여기](#)에서 찾을 수 있습니다.

[이전 섹션](#)에서 다뤄본 째깍거리는 시계 예시를 다시 살펴보겠습니다. [엘리먼트 렌더링](#)에서는 UI를 업데이트하는 한 가지 방법만 배웠으며, 렌더링 된 출력값을 변경하기 위해 `root.render()` 를 호출했습니다.

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

function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  root.render(element);
}

setInterval(tick, 1000);
```

[CodePen에서 시도해보기](#)

이 섹션에서는 `Clock` 컴포넌트를 완전히 재사용하고 캡슐화하는 방법을 배울 것입니다. 이 컴포넌트는 스스로 타이머를 설정할 것이고 매초 스스로 업데이트할 것입니다.

시계가 생긴 것에 따라 캡슐화하는 것으로 시작할 수 있습니다.

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

function Clock(props) {
  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  root.render(<Clock date={new Date()} />);
}

setInterval(tick, 1000);
```

[CodePen에서 시도해보기](#)

그러나 여기에는 중요한 요건이 누락되어 있습니다. `Clock` 이 타이머를 설정하고 매초 UI를 업데이트하는 것이 `Clock` 의 구현 세부사항이 되어야 합니다.

이상적으로 한 번만 코드를 작성하고 `Clock` 이 스스로 업데이트하도록 만들려고 합니다.

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

이것을 구현하기 위해서 `Clock` 컴포넌트에 "state"를 추가해야 합니다.

`State`는 `props`와 유사하지만, 비공개이며 컴포넌트에 의해 완전히 제어됩니다.

함수에서 클래스로 변환하기

다섯 단계로 `Clock`과 같은 함수 컴포넌트를 클래스로 변환할 수 있습니다.

1. `React.Component`를 확장하는 동일한 이름의 `ES6 class`를 생성합니다.
2. `render()`라고 불리는 빈 메서드를 추가합니다.
3. 함수의 내용을 `render()` 메서드 안으로 옮깁니다.
4. `render()` 내용 안에 있는 `props`를 `this.props`로 변경합니다.
5. 남아있는 빈 함수 선언을 삭제합니다.

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.props.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

CodePen에서 시도해보기

`Clock`은 이제 함수가 아닌 클래스로 정의됩니다.

`render` 메서드는 업데이트가 발생할 때마다 호출되지만, 같은 DOM 노드로 `<Clock />`을 렌더링하는 경우 `Clock` 클래스의 단일 인스턴스만 사용됩니다. 이것은 로컬 `state`와 생명주기 메서드와 같은 부가적인 기능을 사용할 수 있게 해줍니다.

클래스에 로컬 `State` 추가하기

세 번째 단계에 걸쳐서 `date`를 `props`에서 `state`로 이동해 보겠습니다.

1. `render()` 메서드 안에 있는 `this.props.date`를 `this.state.date`로 변경합니다.

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

2. 초기 `this.state`를 지정하는 `class constructor`를 추가합니다.

```

class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

```

여기서 어떻게 `props` 를 기본 `constructor`에 전달하는지 유의해주세요.

```

constructor(props) {
  super(props);
  this.state = {date: new Date()};
}

```

클래스 컴포넌트는 항상 `props` 로 기본 `constructor`를 호출해야 합니다.

3. `<Clock />` 요소에서 `date` prop을 삭제합니다.

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

타이머 코드는 나중에 다시 컴포넌트로 추가하도록 하겠습니다.

결과는 다음과 같습니다.

```

class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

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

```

[CodePen에서 시도해보기](#)

다음으로 `Clock` 이 스스로 타이머를 설정하고 매초 스스로 업데이트하도록 만들어 보겠습니다.

생명주기 메서드를 클래스에 추가하기

많은 컴포넌트가 있는 애플리케이션에서 컴포넌트가 삭제될 때 해당 컴포넌트가 사용 중이던 리소스를 확보하는 것이 중요합니다.

`Clock` 이 처음 DOM에 렌더링 될 때마다 [타이머를 설정](#)하려고 합니다. 이것은 React에서 "마운팅"이라고 합니다.

또한 `Clock` 에 의해 생성된 DOM이 삭제될 때마다 [타이머를 해제](#)하려고 합니다. 이것은 React에서 "언마운팅"이라고 합니다.

컴포넌트 클래스에서 특별한 메서드를 선언하여 컴포넌트가 마운트되거나 언마운트 될 때 일부 코드를 작동할 수 있습니다.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {}

  componentWillUnmount() {}

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

이러한 메서드들은 "생명주기 메서드"라고 불립니다.

`componentDidMount()` 메서드는 컴포넌트 출력물이 DOM에 렌더링 된 후에 실행됩니다. 이 장소가 타이머를 설정하기에 좋은 장소입니다.

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

`this (this.timerID)`에서 어떻게 타이머 ID를 제대로 저장하는지 주의해주세요.

`this.props` 가 React에 의해 스스로 설정되고 `this.state` 가 특수한 의미가 있지만, 타이머 ID와 같이 데이터 흐름 안에 포함되지 않는 어떤 항목을 보관할 필요가 있다면 자유롭게 클래스에 수동으로 부가적인 필드를 추가해도 됩니다.

`componentWillUnmount()` 생명주기 메서드 안에 있는 타이머를 분해해 보겠습니다.

```
componentWillUnmount() {
  clearInterval(this.timerID);
}
```

마지막으로 `Clock` 컴포넌트가 매초 작동하도록 하는 `tick()`이라는 메서드를 구현해 보겠습니다.

이것은 컴포넌트 로컬 state를 업데이트하기 위해 `this.setState()`를 사용합니다.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

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

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

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

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}

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

CodePen에서 시도해보기

이제 시계는 매초 짜깍거립니다.

현재 어떤 상황이고 메서드가 어떻게 호출되는지 순서대로 빠르게 요약해 보겠습니다.

- `<Clock />` 가 `root.render()`로 전달되었을 때 React는 `Clock` 컴포넌트의 `constructor`를 호출합니다. `Clock`이 현재 시각을 표시해야 하기 때문에 현재 시각이 포함된 객체로 `this.state`를 초기화합니다. 나중에 이 `state`를 업데이트할 것입니다.
- React는 `Clock` 컴포넌트의 `render()` 메서드를 호출합니다. 이를 통해 React는 화면에 표시되어야 할 내용을 알게 됩니다. 그 다음 React는 `Clock`의 렌더링 출력값을 일치시키기 위해 DOM을 업데이트합니다.
- `Clock` 출력값이 DOM에 삽입되면, React는 `componentDidMount()` 생명주기 메서드를 호출합니다. 그 안에서 `Clock` 컴포넌트는 매초 컴포넌트의 `tick()` 메서드를 호출하기 위한 타이머를 설정하도록 브라우저에 요청합니다.

4. 매초 브라우저가 `tick()` 메서드를 호출합니다. 그 안에서 `Clock` 컴포넌트는 `setState()`에 현재 시각을 포함하는 객체를 호출하면서 UI 업데이트를 진행합니다. `setState()` 호출 덕분에 React는 state가 변경된 것을 인지하고 화면에 표시될 내용을 알아내기 위해 `render()` 메서드를 다시 호출합니다. 이 때 `render()` 메서드 안의 `this.state.date` 가 달라지고 렌더링 출력값은 업데이트된 시각을 포함합니다. React는 이에 따라 DOM을 업데이트합니다.

5. `Clock` 컴포넌트가 DOM으로부터 한 번이라도 삭제된 적이 있다면 React는 타이머를 멈추기 위해 `componentWillUnmount()` 생명주기 메서드를 호출합니다.

State를 올바르게 사용하기

`setState()`에 대해서 알아야 할 세 가지가 있습니다.

직접 State를 수정하지 마세요

예를 들어, 이 코드는 컴포넌트를 다시 렌더링하지 않습니다.

```
// Wrong
this.state.comment = 'Hello';
```

대신에 `setState()`를 사용합니다.

```
// Correct
this.setState({comment: 'Hello'});
```

`this.state`를 지정할 수 있는 유일한 공간은 바로 constructor입니다.

State 업데이트는 비동기적일 수도 있습니다.

React는 성능을 위해 여러 `setState()` 호출을 단일 업데이트로 한꺼번에 처리할 수 있습니다.

`this.props` 와 `this.state` 가 비동기적으로 업데이트될 수 있기 때문에 다음 state를 계산할 때 해당 값에 의존해서는 안 됩니다.

예를 들어, 다음 코드는 카운터 업데이트에 실패할 수 있습니다.

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

이를 수정하기 위해 객체보다는 함수를 인자로 사용하는 다른 형태의 `setState()`를 사용합니다. 그 함수는 이전 state를 첫 번째 인자로 받아들일 것이고, 업데이트가 적용된 시점의 `props`를 두 번째 인자로 받아들일 것입니다.

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

위에서는 화살표 함수를 사용했지만, 일반적인 함수에서도 정상적으로 작동합니다.

```
// Correct
this.setState(function(state, props) {
  return {
    counter: state.counter + props.increment
  };
});
```

State 업데이트는 병합됩니다

`setState()` 를 호출할 때 React는 제공한 객체를 현재 state로 병합합니다.

예를 들어, state는 다양한 독립적인 변수를 포함할 수 있습니다.

```
constructor(props) {
  super(props);
  this.state = {
    posts: [],
    comments: []
  };
}
```

별도의 `setState()` 호출로 이러한 변수를 독립적으로 업데이트할 수 있습니다.

```
componentDidMount() {
  fetchPosts().then(response => {
    this.setState({
      posts: response.posts
    });
  });

  fetchComments().then(response => {
    this.setState({
      comments: response.comments
    });
  });
}
```

병합은 얇게 이루어지기 때문에 `this.setState({comments})` 는 `this.state.posts`에 영향을 주진 않지만 `this.state.comments` 는 완전히 대체됩니다.

데이터는 아래로 흐릅니다

부모 컴포넌트나 자식 컴포넌트 모두 특정 컴포넌트가 유상태인지 또는 무상태인지 알 수 없고, 그들이 함수나 클래스로 정의되었는지에 대해서 관심을 가질 필요가 없습니다.

이 때문에 state는 종종 로컬 또는 캡슐화라고 불립니다. state가 소유하고 설정한 컴포넌트 이외에는 어떠한 컴포넌트에도 접근할 수 없습니다.

컴포넌트는 자신의 state를 자식 컴포넌트에 props로 전달할 수 있습니다.

```
<FormattedDate date={this.state.date} />
```

`FormattedDate` 컴포넌트는 `date` 를 자신의 props로 받을 것이고 이것이 `Clock` 의 state로부터 왔는지, `Clock` 의 props에서 왔는지, 수동으로 입력한 것인지 알지 못합니다.

```
function FormattedDate(props) {
  return <h2>It is {props.date.toLocaleTimeString()}</h2>;
}
```

[CodePen에서 시도해보기](#)

일반적으로 이를 "하향식(top-down)" 또는 "단방향식" 데이터 흐름이라고 합니다. 모든 state는 항상 특정한 컴포넌트가 소유하고 있으며 그 state로부터 파생된 UI 또는 데이터는 오직 트리구조에서 자신의 "아래"에 있는 컴포넌트에만 영향을 미칩니다.

트리구조가 props들의 폭포라고 상상하면 각 컴포넌트의 state는 임의의 점에서 만나지만 동시에 아래로 흐르는 부가적인 수원(water source)이라고 할 수 있습니다.

모든 컴포넌트가 완전히 독립적이라는 것을 보여주기 위해 App 렌더링하는 세 개의 <Clock> 을 만들었습니다.

```
function App() {
  return (
    <div>
      <Clock />
      <Clock />
      <Clock />
    </div>
  );
}
```

[CodePen에서 시도해보기](#)

각 Clock 은 자신만의 타이머를 설정하고 독립적으로 업데이트를 합니다.

React 앱에서 컴포넌트가 유상태 또는 무상태에 대한 것은 시간이 지남에 따라 변경될 수 있는 구현 세부 사항으로 간주합니다. 유상태 컴포넌트 안에서 무상태 컴포넌트를 사용할 수 있으며, 그 반대 경우도 마찬가지로 사용할 수 있습니다.

[Go to TOC](#)

Static Type Checking

[Flow](#), [TypeScript](#)와 같은 정적 타입 체커들은 코드 실행 전에 특정한 타입 문제를 찾아냅니다. 또한 자동완성과 같은 기능을 추가하여 개발자의 작업 흐름을 개선하기도 합니다. 이러한 이유로 큰 코드 베이스에서는 `PropTypes`를 사용하는 대신 Flow 혹은 TypeScript를 사용하는 것을 추천해 드립니다.

Flow

Flow는 JavaScript 코드를 위한 정적 타입 체커입니다. 페이스북에서 개발했으며, 보통 React와 함께 사용합니다. 특별한 타입 문법을 사용하여 변수, 함수 및 React 컴포넌트에 주석을 달 수 있고, 에러를 조기에 발견할 수 있습니다. 기초적인 부분을 더 알아보고 싶다면 [introduction to Flow](#)에서 확인할 수 있습니다.

Flow를 사용하기 위해서는 아래 요구 사항을 만족 해야 합니다.

- Flow를 프로젝트 의존성에 추가합니다.
- 컴파일된 코드에서 Flow 문법이 제거되었는지 확인합니다.
- 타입 주석을 추가하고, 타입을 체크하기 위해 Flow를 실행합니다.

이 단계들에 대해 아래에서 자세히 설명해드리겠습니다.

프로젝트에 Flow 추가하기

우선, 터미널을 통해 프로젝트 디렉토리로 들어간 뒤 다음 명령어를 실행해주세요.

[Yarn](#)을 사용한다면

```
yarn add --dev flow-bin
```

[npm](#)을 사용한다면

```
npm install --save-dev flow-bin
```

이 명령어를 입력하면 프로젝트에 최신 버전 Flow가 설치됩니다.

이제 터미널을 통해 Flow를 사용하기 위해서 `package.json` 파일의 `"scripts"` 부분에 `"flow"`라고 추가해줍니다.

```
{
  // ...
  "scripts": {
    "flow": "flow",
    // ...
  },
  // ...
}
```

마지막으로 다음 명령어를 통해 실행시켜줍니다.

[Yarn](#)을 사용한다면

```
yarn run flow init
```

[npm](#)을 사용한다면

```
npm run flow init
```

이 명령어는 Flow 환경설정 파일을 만들어줍니다.

컴파일된 코드에서 Flow 문법 제거하기

Flow는 코드 주석을 위한 특별한 문법과 함께 JavaScript 언어를 확장합니다. 하지만 브라우저는 이 문법을 알아차리지 못하기 때문에 컴파일된 JavaScript 번들을 브라우저에 보내기만 하고 끝내서는 안됩니다.

이 작업을 수행하기 위한 방법은 JavaScript를 컴파일하는 데 사용하는 도구에 따라 달라집니다.

Create React App

프로젝트를 [Create React App](#)을 통해 세팅하셨다면, 축하합니다! Flow 주석은 이미 기본적으로 제거되기 때문에 어떠한 작업을 별도로 해줄 필요가 없습니다.

Babel

주의

아래 설명은 Create React App 사용자들을 위한 것이 아닙니다. Create React App이 Babel을 사용하기는 하지만, 이미 Flow를 이해하도록 설정되어 있습니다. Create React App을 사용하지 않은 분들만 이 단계를 따라와 주세요.

직접 Babel 설정을 한다면 Flow를 위해 특별한 프리셋을 설치해야합니다.

[Yarn](#)을 사용한다면

```
yarn add --dev @babel/preset-flow
```

[npm](#)을 사용한다면

```
npm install --save-dev @babel/preset-flow
```

설치가 됐다면, `flow` 프리셋을 [Babel configuration](#)에 추가합니다. 예를 들어 Babel을 `.babelrc` 파일에서 설정한다면, 다음과 같이 할 수 있습니다.

```
{
  "presets": [
    "@babel/preset-flow",
    "react"
  ]
}
```

이 부분은 코드에 Flow 문법을 사용할 수 있도록 도와줍니다.

주의

Flow는 `react` 프리셋을 필수적으로 요구하지는 않지만 자주 함께 사용합니다. Flow 자체가 JSX 구문을 이해할 수 있습니다.

다른 빌드 설정들

Create React App과 Babel을 사용하지 않는다면 [flow-remove-types](#)를 사용해서 주석을 제거할 수 있습니다.

Flow 실행하기

위의 설명을 잘 따라왔다면, Flow를 바로 사용할 수 있습니다.

```
yarn flow
```

npm을 사용한다면

```
npm run flow
```

다음 같은 메세지가 보여야 합니다.

```
No errors!
★ Done in 0.17s.
```

Flow 타입 주석 추가하기

기본적으로 Flow는 다음 주석이 포함된 파일만 체크합니다.

```
// @flow
```

대체적으로 위 주석은 파일 최상단에 둡니다. 프로젝트의 몇몇 파일에 주석을 추가하고 `yarn flow` 나 `npm run flow` 명령어를 실행하여 Flow가 어떤 문제를 찾아냈는지 확인해보세요.

주석에 상관없이 모든 파일들을 체크하는 [옵션](#)도 있습니다. 이미 존재하는 프로젝트에 적용하는 것은 어렵겠지만 모든 타입을 체크하고자 하는 새로운 프로젝트에는 적합합니다.

이제 모든 준비를 마쳤습니다! Flow에 대해 더 알아보고 싶으시다면 다음 리소스들을 확인해보세요.

- [Flow Documentation: Type Annotations](#)
- [Flow Documentation: Editors](#)
- [Flow Documentation: React](#)
- [Linting in Flow](#)

TypeScript

TypeScript는 Microsoft가 개발한 프로그래밍 언어입니다. JavaScript의 타입 슈퍼셋이며 자체 컴파일러를 가지고 있습니다. 타입 언어이기 때문에 빌드 에러와 버그를 잡을 수 있으며, 이는 앱이 실행되기 훨씬 전입니다. TypeScript를 React와 함께 사용하는 방법에 대해 더 알아보고 싶다면 [여기](#)에 들어가보세요.

TypeScript를 사용하기 위해서는 아래 요구 사항을 만족해야합니다.

- 프로젝트 의존성에 TypeScript를 추가합니다.

- TypeScript 컴파일러 옵션을 설정합니다.
- 올바른 파일 확장을 사용합니다.
- 사용하는 라이브러리의 정의를 추가합니다.

좀 더 자세히 알아보겠습니다.

Create React App과 함께 타입스크립트 사용하기

Create React App은 타입스크립트를 별도의 설정 없이 사용할 수 있도록 지원해줍니다.

다음 명령어를 실행하면 TypeScript를 지원하는 **새로운 프로젝트**를 생성할 수 있습니다.

```
npx create-react-app my-app --template typescript
```

또한 이미 존재하는 **Create React App** 프로젝트에도 추가할 수 있습니다. [이 문서](#)에서 확인해보세요.

주의

Create React App을 사용한다면, 이 페이지의 남은 부분을 넘기셔도 좋습니다. 아래는 Create React App을 사용하지 않는 사용자들을 위한 설명입니다.

프로젝트에 TypeScript 추가하기

터미널에 다음 명령어를 입력하는 것으로 시작합니다.

[Yarn](#)을 사용한다면

```
yarn add --dev typescript
```

[npm](#)을 사용한다면

```
npm install --save-dev typescript
```

축하합니다! 프로젝트에 가장 최신 버전 TypeScript가 설치되었습니다. TypeScript를 설치하면 `tsc` 명령어에 접근할 수 있습니다. 설정을 하기 전에 `package.json` 파일 "script" 부분에 "`tsc`"를 추가해주세요.

```
{
  // ...
  "scripts": {
    "build": "tsc",
    // ...
  },
  // ...
}
```

TypeScript 컴파일러 설정하기

컴파일러는 무엇을 해야할지 설정해주지 않는다면 아무 도움도 주지 않습니다. TypeScript는 `tsconfig.json`이라는 특별한 파일에 설정을 해야 합니다. 이 파일을 생성하려면 아래의 명령어를 입력해야합니다.

[Yarn](#)을 사용한다면

```
yarn run tsc --init
```

[npm](#)을 사용한다면

```
npx tsc --init
```

생성된 `tsconfig.json`에서 사용할 수 있는 수많은 컴파일러 옵션들을 볼 수 있습니다. 모든 옵션에 대한 자세한 설명은 [여기](#)에서 확인해보세요.

많은 옵션 중에서 `rootDir` 와 `outDir` 를 살펴보려고 합니다. 실제로 컴파일러는 TypeScript 파일을 통해 JavaScript 파일을 생성합니다. 여기서 소스 파일과 생성된 파일간의 혼동을 야기할 수 있습니다.

이를 해결하기 위해 두 단계를 거칩니다.

- 우선 프로젝트 구조를 아래와 같이 정리합니다. 모든 소스코드는 `src` 디렉토리에 위치시킬 것입니다.

```
└── package.json
└── src
  └── index.ts
└── tsconfig.json
```

- 그 다음, 소스코드가 어디 있는지, 컴파일을 통해 생성된 코드를 어디에 위치시켜야 하는지 컴파일러에 서술합니다.

```
// tsconfig.json
{
  "compilerOptions": {
    // ...
    "rootDir": "src",
    "outDir": "build"
    // ...
  },
}
```

좋습니다! 이제 빌드 스크립트를 실행하면 컴파일러가 생성된 JavaScript를 `build` 폴더에 위치시킬 것입니다. [TypeScript React Starter](#)는 시작하기에 좋은 규칙들을 정의한 `tsconfig.json` 파일을 제공합니다.

일반적으로 생성된 JavaScript 코드를 소스 관리에 두고 싶어 하지 않습니다. 때문에 `build` 폴더를 `.gitignore` 파일에 추가하도록 합니다.

파일 확장자

React에서는 대부분 컴포넌트를 `.js` 파일에 작성합니다. TypeScript에는 두 가지 확장자가 있습니다.

`.ts` 는 TypeScript 파일 확장자 기본값입니다. 반면에 `.tsx` 는 `JSX` 문법이 포함된 코드를 위한 특별한 확장자입니다.

TypeScript 실행하기

위 설명을 잘 따라왔다면 TypeScript를 즉시 실행할 수 있습니다.

```
yarn build
```

npm을 사용한다면

```
npm run build
```

터미널에 아무런 출력이 없다면 컴파일이 성공적으로 완료됨을 의미합니다.

타입 정의

다른 패키지의 오류와 힌트를 출력하기 위해 컴파일러는 선언 파일에 의존합니다. 선언 파일은 라이브러리에 대한 모든 타입 정보를 제공합니다. 프로젝트의 npm에 라이브러리에 대한 선언파일이 있다면 해당하는 JavaScript 라이브러리를 사용할 수 있습니다.

라이브러리에 대한 선언을 가져올 수 있는 방법은 두가지가 있습니다.

Bundled - 라이브러리가 자신의 선언 파일을 번들합니다. 이 후 해야할 일은 그저 라이브러리를 다운받고 올바르게 사용하는 것 밖에 할 일이 없기 때문에 사용자에게 좋습니다. 라이브러리가 번들된 타입을 가지고있는지 확인하려면 프로젝트 내에 `index.d.ts` 파일이 존재하는지 찾아보세요. 어떤 라이브러리는 `package.json` 파일의 `typings` 혹은 `types` 필드 아래에 정의되어 있습니다.

DefinitelyTyped - DefinitelyTyped는 선언 파일을 번들하지 않은 라이브러리를 위한 거대 저장소입니다. 이 저장소의 선언은 Microsoft와 오픈소스 기여자들에 의해 관리되는 크라우드 소스입니다. 예를 들어 React는 자체 선언 파일을 번들하지 않습니다. 대신 DefinitelyTyped를 통해 다운받을 수 있습니다. 다음 명령어를 터미널에 입력해주세요.

```
# yarn
yarn add --dev @types/react

# npm
npm i --save-dev @types/react
```

Local Declarations 때때로 사용하고 싶은 패키지가 타입 선언 파일을 번들하지도 않고 DefinitelyTyped에서 제공하지도 않을 수 있습니다. 이러한 경우 로컬 타입 선언 파일을 가질 수 있습니다. 이 방법을 사용하려면 `declarations.d.ts` 파일을 source 디렉토리의 루트에 생성합니다. 간단한 선언은 다음과 같이 할 수 있습니다.

```
declare module 'querystring' {
  export function stringify(val: object): string;
  export function parse(val: string): object;
}
```

이제 코드를 작성할 준비를 마쳤습니다! TypeScript에 대해 좀 더 알아보고 싶다면 다음 리소스들을 확인해보세요.

- [TypeScript Documentation: Everyday Types](#)
- [TypeScript Documentation: Migrating from JavaScript](#)
- [TypeScript Documentation: React and Webpack](#)

ReScript

ReScript는 JavaScript로 컴파일되는 타입이 있는 언어입니다. 핵심 기능 중 몇 가지는 100% 타입 커버리지가 보장된다는 점과 first-class로 JSX를 지원하며, JS와 TS로 작성된 React 코드베이스와의 통합을 허용하기 위한 [React 전용 바인딩](#)이 있습니다.

이미 작성된 JS와 React 코드베이스에 ReScript를 통합하는 방법에 대한 더 자세한 정보는 [여기](#)에서 찾을 수 있습니다.

Kotlin

[Kotlin](#)은 JetBrains이 개발한 정적 타입 언어입니다. Kotlin의 타깃 플랫폼은 JVM, Android, LLVM, JavaScript입니다.

JetBrains은 React 커뮤니티를 위해 [React bindings](#)나 [Create React Kotlin App](#)와 같은 몇몇 도구를 개발, 유지하고 있습니다. Create React Kotlin App은 별다른 빌드 설정 없이 Kotlin으로 React 앱을 개발할 수 있도록 도와줍니다.

다른 언어들

JavaScript로 컴파일 할 수 있다면 다른 정적 타입 언어들도 React와 호환할 수 있습니다. 예를 들면 [F#/Fable](#)를 기반으로한 [elmish-react](#)가 있습니다. 자세한 내용은 각 사이트를 참고하세요. 또한 이 페이지에 React에서 사용할 수 있는 정적 타입 언어들을 자유롭게 추가해주세요.

Strict 모드

`StrictMode` 는 애플리케이션 내의 잠재적인 문제를 알아내기 위한 도구입니다. `Fragment` 와 같이 UI를 렌더링하지 않으며, 자손들에 대한 부가적인 검사와 경고를 활성화합니다.

주의

Strict 모드는 개발 모드에서만 활성화되기 때문에, 프로덕션 빌드에는 영향을 끼치지 않습니다.

애플리케이션 내 어디서든지 아래와 같이 strict 모드를 활성화할 수 있습니다. `embed:strict-mode/enabling-strict-mode.js`

위의 예시에서, `Header` 와 `Footer` 컴포넌트는 Strict 모드 검사가 이루어지지 않습니다. 하지만, `ComponentOne` 과 `ComponentTwo` 는 각각의 자손까지 검사가 이루어집니다.

`StrictMode` 는 아래와 같은 부분에서 도움이 됩니다.

- 안전하지 않은 생명주기를 사용하는 컴포넌트 발견
- 레거시 문자열 `ref` 사용에 대한 경고
- 권장되지 않는 `findDOMNode` 사용에 대한 경고
- 예상치 못한 부작용 검사
- 레거시 `context API` 검사
- [Ensuring reusable state](#)

React의 향후 릴리즈에서 더 많은 기능이 더해질 예정입니다.

안전하지 않은 생명주기를 사용하는 컴포넌트 발견

[블로그](#) 글에서 설명하였듯, 비동기 React 애플리케이션에서 특정 생명주기 메서드들은 안전하지 않습니다. 하지만 애플리케이션이 서드 파티 라이브러리를 사용한다면, 해당 생명주기 메서드가 사용되지 않는다고 장담하기 어렵습니다. Strict 모드는 이러한 경우에 도움이 됩니다!

Strict 모드가 활성화되면, React는 안전하지 않은 생명주기 메서드를 사용하는 모든 클래스 컴포넌트 목록을 정리해 다음과 같이 컴포넌트에 대한 정보가 담긴 경고 로그를 출력합니다.

```
▶ Warning: Unsafe lifecycle methods were found within a strict-mode tree:
  in div (created by ExampleApplication)
  in ExampleApplication

componentWillMount: Please update the following components to use componentDidMount instead: ThirdPartyComponent

Learn more about this warning here:
https://fb.me/react-strict-mode-warnings
```

Strict 모드에 의해 발견된 문제들을 해결한다면, 향후 릴리즈되는 React에서 concurrent 렌더링의 이점을 얻을 수 있을 것입니다.

레거시 문자열 ref 사용에 대한 경고

이전의 React에서 레거시 문자열 ref API와 콜백 API라는, ref를 관리하는 두 가지 방법을 제공하였습니다. 문자열 ref가 사용하기 더 편리했지만 [몇몇 단점들이](#) 있었습니다. 그래서 공식적으로는 [콜백 형태를 사용](#)하는 것을 권장하였습니다.

React 16.3에서는 여러 단점 없이 문자열 ref의 편리함을 제공하는 세 번째 방법을 추가하였습니다. [embed:16-3-release-blog-post/create-ref-example.js](#)

이제는 객체 ref가 문자열 ref를 교체하는 용도로 널리 더해졌기 때문에, Strict 모드는 문자열 ref의 사용에 대해 경고합니다.

주의

콜백 ref는 새로운 `createRef` API와 별개로 지속해서 지원될 예정입니다.

컴포넌트의 콜백 ref를 교체할 필요는 없습니다. 콜백 ref는 조금 더 유연하기 때문에, 고급 기능으로서 계속 지원할 예정입니다.

[createRef API에 대해서 알아보기](#)

권장되지 않는 `findDOMNode` 사용에 대한 경고

이전의 React에서 주어진 클래스 인스턴스를 바탕으로 트리를 탐색해 DOM 노드를 찾을 수 있는 `findDOMNode`를 지원하였습니다. [DOM 노드에 바로 ref를 지정](#)할 수 있기 때문에 보통은 필요하지 않습니다.

`findDOMNode` 는 클래스 컴포넌트에서도 사용할 수 있었지만, 부모가 특정 자식이 렌더링되는 것을 요구하는 상황이 허용되어, 추상화 레벨이 무너지게 되었습니다. 이로 인해 부모가 자식의 DOM 노드에까지 닿을 가능성이 있어 컴포넌트의 세세한 구현을 변경할 수 없게 되어 리팩토링이 어려워지는 상황을 만들고 말았습니다. `findDOMNode` 는 항상 첫 번째 자식을 반환하지만, Fragment와 함께 사용 할 경우 컴포넌트에서 여러 DOM 노드를 렌더링하게 됩니다. `findDOMNode` 는 일회성, 읽기 전용 API입니다. 물어보았을 때만 값을 반환합니다. 자식 컴포넌트가 다른 노드를 렌더링할 경우, 변경 사항에 대응할 방법이 없습니다. 그러므로, `findDOMNode` 는 항상 변하지 않는, 단일 DOM 노드를 반환하는 컴포넌트에서만 정상적으로 작동해왔습니다.

[ref를 넘겨주는 방식](#)을 사용해 커스텀 컴포넌트에 ref를 넘겨 DOM까지 닿게 하는 것으로, 이를 분명하게 만들 수 있습니다.

DOM 노드를 감싸는 래퍼를 만들어 ref를 바로 붙이는 것 역시 가능합니다.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.wrapper = React.createRef();
  }
  render() {
    return <div ref={this.wrapper}>{this.props.children}</div>;
  }
}
```

주의

노드가 레이아웃 바깥의 요소가 되는 것을 막고자 한다면, CSS에서 `display: contents` 속성을 사용할 수 있습니다.

예상치 못한 부작용 검사

개념적으로 React는 두 단계로 동작합니다.

- 렌더링 단계는 특정 환경(예를 들어, DOM과 같이)에 어떤 변화가 필요한지 결정하는 단계입니다. 이 과정에서 React는 `render` 를 호출하여 이전 렌더와 결과값을 비교합니다.
- 커밋 단계는 React가 변경 사항을 반영하는 단계입니다(React DOM의 경우 React가 DOM 노드를 추가, 변경 및 제거하는 단계를 말합니다). 이 단계에서 React는 `componentDidMount` 나 `componentDidUpdate` 와 같은 생명주기 메서드를 호출합니다.

커밋 단계는 일반적으로 매우 빠르지만, 렌더링 단계는 느릴 수 있습니다. 이로 인해, 곧 추가될 concurrent 모드(아직 기본적으로는 비활성화됨)는 렌더링 작업을 더 작은 단위로 나누고, 작업을 중지했다 재개하는 방식으로 브라우저가 멈추는 것을 피합니다. 즉, React는 커밋하기 전에 렌더링 단계의 생명주기 메서드를 여러 번 호출하거나 아예 커밋을 하지 않을 수도(예로 혹은 우선순위에 따른 작업 중단) 있습니다.

렌더링 단계 생명주기 메서드는 클래스 컴포넌트의 메서드를 포함해 다음과 같습니다.

- `constructor`
- `componentWillMount` (or `UNSAFE_componentWillMount`)
- `componentWillReceiveProps` (or `UNSAFE_componentWillReceiveProps`)
- `componentWillUpdate` (or `UNSAFE_componentWillUpdate`)
- `getDerivedStateFromProps`
- `shouldComponentUpdate`
- `render`
- `setState` 업데이트 함수 (첫 번째 인자)

위의 메서드들은 여러 번 호출될 수 있기 때문에, 부작용을 포함하지 않는 것이 중요합니다. 이 규칙을 무시할 경우, 메모리 누수 혹은 잘 못된 애플리케이션 상태 등 다양한 문제를 일으킬 가능성이 있습니다. 불행히도, 보통 이러한 문제들은 예측한 대로 동작하지 않기 때문에 발견하는 것이 어려울 수 있습니다.

Strict 모드가 자동으로 부작용을 찾아주는 것은 불가능합니다. 하지만, 조금 더 예측할 수 있게끔 만들어서 문제가 되는 부분을 발견할 수 있게 도와줍니다. 이는 아래의 함수를 의도적으로 이중으로 호출하여 찾을 수 있습니다.

- 클래스 컴포넌트의 `constructor`, `render` 그리고 `shouldComponentUpdate` 메서드
- 클래스 컴포넌트의 `getDerivedStateFromProps` static 메서드
- 함수 컴포넌트 바디
- State updater 함수 (`setState`의 첫 번째 인자)
- `useState`, `useMemo` 그리고 `useReducer`에 전달되는 함수

주의

개발 모드에서만 적용됩니다. 생명주기 메서드들은 프로덕션 모드에서 이중으로 호출되지 않습니다.

예를 들어, 아래의 코드를 생각해봅시다. `embed:strict-mode/side-effects-in-constructor.js`

얼핏 보면 이 코드에는 문제가 없어 보입니다. 하지만, `SharedApplicationState.recordEvent`의 연산 결과가 계속 달라진다면, 이 컴포넌트를 여러 번 인스턴스화했을 때 애플리케이션의 상태를 잘못된 방향으로 이끌 수 있습니다. 이와 같은 이해하기 어려운 버그들은 개발 중에 나타나지 않을 수도 있고, 일관성이 없어 발견하지 못할 수도 있습니다.

컴포넌트의 `constructor`와 같은 메서드를 의도적으로 두 번 호출하면 strict mode가 이와 같은 패턴을 쉽게 찾을 수 있도록 합니다.

주의

In React 17, React automatically modifies the console methods like `console.log()` to silence the logs in the second call to lifecycle functions. However, it may cause undesired behavior in certain cases where a workaround can be used.

Starting from React 18, React does not suppress any logs. However, if you have React DevTools installed, the logs from the second call will appear slightly dimmed. React DevTools also offers a setting (off by default) to suppress them completely.

레거시 context API 검사

레거시 context API는 오류가 발생하기 쉬워 이후 릴리즈에서 삭제될 예정입니다. 모든 16.x 버전에서 여전히 돌아가지만, Strict 모드에서는 아래와 같은 경고 메시지를 노출합니다.

```
✖ ▶Warning: Legacy context API has been detected within a strict-mode tree:  
  in div (at App.js:32)  
  in App (at index.js:7)  
  
Please update the following components: LegacyContextConsumer, LegacyContextProvider  
  
Learn more about this warning here:  
https://fb.me/react-strict-mode-warnings
```

새로운 context API 문서를 참조하여 새로운 버전으로 마이그레이션하시길 바랍니다.

Ensuring reusable state

In the future, we'd like to add a feature that allows React to add and remove sections of the UI while preserving state. For example, when a user tabs away from a screen and back, React should be able to immediately show the previous screen. To do this, React supports remounting trees using the same component state used before unmounting.

This feature will give React better performance out-of-the-box, but requires components to be resilient to effects being mounted and destroyed multiple times. Most effects will work without any changes, but some effects do not properly clean up subscriptions in the destroy callback, or implicitly assume they are only mounted or destroyed once.

To help surface these issues, React 18 introduces a new development-only check to Strict Mode. This new check will automatically unmount and remount every component, whenever a component mounts for the first time, restoring the previous state on the second mount.

To demonstrate the development behavior you'll see in Strict Mode with this feature, consider what happens when React mounts a new component. Without this change, when a component mounts, React creates the effects:

- * React mounts the component.
- * Layout effects are created.
- * Effects are created.

With Strict Mode starting in React 18, whenever a component mounts in development, React will simulate immediately unmounting and remounting the component:

- * React mounts the component.
 - * Layout effects are created.
 - * Effect effects are created.
- * React simulates effects being destroyed on a mounted component.
 - * Layout effects are destroyed.
 - * Effects are destroyed.
- * React simulates effects being re-created on a mounted component.
 - * Layout effects are created
 - * Effect setup code runs

On the second mount, React will restore the state from the first mount. This feature simulates user behavior such as a user tabbing away from a screen and back, ensuring that code will properly handle state restoration.

When the component unmounts, effects are destroyed as normal:

- * React unmounts the component.
- * Layout effects are destroyed.
- * Effect effects are destroyed.

Unmounting and remounting includes:

- `componentDidMount`
- `componentWillUnmount`
- `useEffect`
- `useLayoutEffect`
- `useInsertionEffect`

Note:

This only applies to development mode, *production behavior is unchanged*.

For help supporting common issues, see:

- [How to support Reusable State in Effects](#)

[Go to TOC](#)

테스트 환경

이 문서는 환경에 영향을 줄 수 있는 요소와 일부 시나리오에 대한 권장 사항을 살펴봅니다.

테스트 러너

[Jest](#), [Mocha](#), [ava](#)와 같은 테스트 러너는 테스트 스위트를 일반 자바 스크립트로 작성하고, 개발 프로세스의 일부로 실행할 수 있도록 합니다. 추가적으로, 테스트 스위트는 지속적 통합의 일부로 실행됩니다.

- Jest는 모의 [모듈 및 타이머](#), 그리고 [jsdom](#) 지원 등 여러 기능을 지원하는 React 프로젝트와 광범위하게 호환됩니다. [Create React App을 사용한다면, Jest는 이미 유용한 기본값과 함께 포함되어 있습니다.](#)
- [mocha](#)같은 라이브러리는 실제 브라우저 환경에서도 잘 작동하며, 이는 분명히 필요한 테스트에 도움이 될 수 있습니다.
- 앤드 투 앤드 테스트는 여러 페이지에 걸친 긴 흐름을 테스트하기 위해 사용되며, [다른 설정](#)이 필요합니다.

렌더링 표면에 대한 모의하기

테스트는 종종 브라우저와 같은 실제 렌더링 표면에 접근하지 않은 환경에서도 진행됩니다. 이런 환경에서는, Node.js 내에서 실행되는 가벼운 브라우저인 [jsdom](#)을 사용하여 브라우저를 시뮬레이션하는 것을 권장합니다.

대체로, jsdom은 일반 브라우저처럼 동작하지만 [레이아웃이나 탐색](#)과 같은 기능은 가지고 있지 않습니다. 이는 여전히 대부분의 웹 기반 컴포넌트 테스트에 유용합니다. 왜냐하면 테스트를 위해 브라우저를 시작하는 것보다 빨리 실행되기 때문입니다. 또한 테스트와 동일한 프로세스에서 실행되므로, 렌더링된 DOM을 검토하고 검증할 코드를 작성할 수 있습니다.

실제 브라우저와 마찬가지로, jsdom은 사용자 상호작용을 모델링할 수 있도록 합니다. 테스트는 DOM 노드에서 이벤트를 발송한 다음 이러한 동작의 부작용을 관찰하고 검증할 수 있습니다. ([예시](#))

UI 테스트의 많은 부분은 위의 설정으로 작성할 수 있습니다. jsdom에게 렌더링하는 테스트 러너로서, 브라우저 이벤트 시퀀스로 지정된 사용자 상호작용과 함께 Jest를 사용하는 것은 [act\(\)](#) 도우미에 의해 작동됩니다. ([예시](#)) 예를 들어, 많은 React 자체 테스트는 이런 조합으로 작성됩니다.

대부분의 브라우저별 동작을 테스트하고 레이아웃이나 실제 입력과 같은 네이티브 브라우저 동작을 요구하는 라이브러리를 작성하는 경우 [mocha](#)와 같은 프레임워크를 사용할 수 있습니다.

DOM을 시뮬레이션할 수 없는 환경에서 (예를 들면, Node.js에서 React Native 컴포넌트 테스트), 엘리먼트와의 상호작용을 시뮬레이션하기 위해 [이벤트 시뮬레이션 헬퍼](#)를 사용할 수 있습니다. 다른 대안으로, [@testing-library/react-native](#)의 [fireEvent](#) 헬퍼를 사용할 수 있습니다.

[Cypress](#), [puppeteer](#), [webdriver](#) 같은 프레임워크들은 [end-to-end 테스트](#)를 진행하기에 유용합니다.

함수 모의하기

테스트를 작성할 때, 우리는 테스트 환경 내부에서 동등성이 없는 우리의 코드 중 일부를 목아웃하고 싶어합니다(예를 들어, [navigator.onLine](#) 상태를 Node.js 내부에서 확인하는 것처럼). 테스트는 또한 일부 함수를 감시할 수 있으며 테스트의 다른 부분이 함수들과 어떻게 상호작용하는지를 관찰할 수 있습니다. 이는 이러한 함수들을 선택적으로 시험 친화적인 버전으로 모의할 수 있다는 점에서 유용합니다.

모의 함수는 특히 데이터를 불러올 때 유용합니다. 실제 API 종단점으로부터 발생하는 느려짐과 손상을 방지하기 위해 테스트에 "가짜"데이터를 사용하는 것이 바람직한 방법입니다 ([예시](#)). 이는 테스트를 예측 가능하게 만들어줍니다. Jest와 sinon과 같은 라이브러리들은 모의 함수들을 지원합니다. 앤드 투 앤드 테스트의 경우 네트워크를 모사하는 것은 어려울 수 있지만, 실제 API 앤드포인트를 테스트하기를 원할 수도 있습니다.

모듈 모의하기

일부 컴포넌트는 테스트 환경에서 잘 작동하지 않거나 테스트에 필수적이지 않은 모듈에 대한 의존성을 가지고 있습니다. 적절한 교체를 통해 이러한 모듈을 선택적으로 모의하는 것이 유용할 수 있습니다 ([예시](#)).

Node.js에서 Jest같은 러너는 [모의 모듈을 지원합니다](#). 또한 `mock-require` 라이브러리도 사용할 수 있습니다.

타이머 모의하기

컴포넌트는 `setTimeout`, `setInterval`, `Data.now` 와 같은 시간을 기반으로한 함수를 사용할 수 있습니다. 테스트 환경에서, 이러한 함수들을 수동으로 발전할 수 있는 대체품으로 모의하는 것이 유용할 수 있습니다. 이것은 테스트가 빨리 진행되도록 하는 데 좋다! 타이머에 의존하는 테스트는 여전히 순서대로 해결되지만 더 빨리 해결됩니다 ([예시](#)). Jest, sinon, lolex를 포함한 대부분의 프레임워크는 테스트에서 타이머를 모의할 수 있게 해줍니다.

가끔, 모의 타이머를 원하지 않는 경우가 있을 수 있기도 합니다. 예를 들어, 애니메이션을 테스트하거나, 또는 (API 속도 제한 장치와 같은) 타이밍에 민감한 종단점과의 상호작용을 하는 경우가 있습니다. 타이머 모의가 있는 라이브러리는 테스트/묶음별로 활성화 및 비활성화할 수 있으므로 이러한 테스트 실행 방법을 명시적으로 선택할 수 있습니다.

엔드 투 엔드 테스트

엔드 투 엔드 테스트는 더 긴 작업흐름을 테스트하는 데 유용하며, 특히 비즈니스에 중요한 작업흐름(결제 또는 회원가입 같이)을 테스트하는 데 유용합니다. 이러한 경우, 브라우저가 실제 앱 전체를 렌더링하고, 실제 API 종단점에서 데이터를 가져오고, 세션과 쿠키를 사용하며, 다른 링크 사이를 이동하는 방법을 모두 테스트 하기를 원할 것입니다. 또한 DOM 상태뿐만 아니라 백업 데이터(예를 들어, 업데이트가 데이터베이스에 유지되었는지 확인하기 위해)에 대해서도 검증하기를 원할 수 있습니다.

이러한 시나리오에서는 [Cypress](#), [Playwright](#)와 같은 프레임워크나 [Puppeteer](#) 같은 라이브러리를 사용하여 여러 경로를 탐색하고 브라우저뿐만 아니라 잠재적으로 백엔드에서도 부작용에 대해 주장할 수 있습니다.

테스팅 방안

React 컴포넌트를 위한 공통 테스트 패턴입니다.

주의

이 페이지는 테스트 러너로 Jest를 사용한다고 가정합니다. 다른 테스트 러너를 사용한다면, 아마도 API를 조정해야 할 수도 있지 만, 전체적인 형태는 거의 비슷할 겁니다. 테스트 환경에 대한 설정에 대해 더 알고 싶다면 [Testing Environments](#)를 참고해 주세요.

이 페이지에서는 함수 컴포넌트를 주로 사용합니다. 하지만, 아래 테스트 전략들은 구현 형태에 의존적이지 않으며 클래스 컴포넌트에서도 잘 동작합니다.

- [설정/해제](#)
- [act\(\)](#)
- [렌더링](#)
- [데이터 가져오기](#)
- [모듈 모의하기](#)
- [이벤트](#)
- [타이머](#)
- [스냅샷 테스트](#)
- [다수의 렌더러](#)
- [뭔가 부족하다면?](#)

설정/해제

테스트마다 일반적으로 React 트리를 `document`의 DOM 엘리먼트에 렌더링하는데, 이는 DOM 이벤트를 수신하기 위해 중요합니다. 테스트가 끝날 때는, 테스트와 관련된 설정 및 값에 대해 정리(clean up)를 하고 `document` 트리에서 마운트 해제합니다.

이를 처리하는 일반적인 방법은 `beforeEach` 와 `afterEach` 블록 쌍을 사용해서 항상 실행되며 테스트의 영향을 자체적으로 분리하도록 하는 것입니다.

```
import { unmountComponentAtNode } from "react-dom";

let container = null;
beforeEach(() => {
  // DOM 엘리먼트를 렌더링 대상으로 설정
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // 종료시 정리
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});
```

다른 패턴을 사용할 수도 있지만, 테스트가 실패하더라도 정리(clean up)는 해야 합니다. 정리하지 않으면 테스트에 완전히 격리가 되지 않은 '빈틈'이 생기기 되고, 하나의 테스트는 다른 테스트의 동작에 영향을 줄 수 있습니다. 이는 디버깅을 어렵게 만듭니다.

act() {#act}

UI 테스트를 작성할 때, 렌더링과 같은 작업, 유저 이벤트, 데이터 가져오기는 유저 인터페이스와의 상호작용하는 "단위"로 간주 됩니다.

`react-dom/test-utils` 는 `act()` 라 불리는 함수를 제공하는데, 이 함수는 "단위"와 관련된 모든 업데이트가 단언이 실행되기 전에 처리되고 DOM에 적용되도록 돕습니다.

```
act(() => {
  // 컴포넌트를 렌더링 한다.
});
// 단언을 추가
```

위의 함수를 통해 실제 사용자가 프로그램을 사용할 때 겪을 경험에 근접하게 테스트를 실행할 수 있습니다. 이후 예시에서는 `act()` 를 사용하여 이를 보장합니다.

`act()` 를 직접 사용하다 보면, 코드가 길어질 때가 있습니다. 이를 간결하게 하고 싶을 때는 `act()` 를 감싼 여러 도우미 함수를 제공하는 [React Testing Library](#)를 사용할 수 있습니다.

주의

`act`라는 이름은 [Arrange-Act-Assert](#) 패턴에서 유래되었습니다.

렌더링

일반적으로 주어진 `props`에 따라 컴포넌트 렌더링이 제대로 되었는지 테스트하고 싶을 때가 있습니다. 이때, `prop`을 기반으로 메시지를 렌더링하는 간단한 컴포넌트가 있다고 생각해봅시다.

```
// hello.js

import React from "react";

export default function Hello(props) {
  if (props.name) {
    return <h1>Hello, {props.name}!</h1>;
  } else {
    return <span>Hey, stranger</span>;
  }
}
```

위 컴포넌트의 테스트를 아래와 같이 작성할 수 있습니다.

```
// hello.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Hello from "./hello";

let container = null;
```

```

beforeEach(() => {
  // 렌더링 대상으로 DOM 엘리먼트를 설정합니다.
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // 기존의 테스트 환경을 정리합니다.
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("renders with or without a name", () => {
  act(() => {
    render(<Hello />, container);
  });
  expect(container.textContent).toBe("Hey, stranger");

  act(() => {
    render(<Hello name="Jenny" />, container);
  });
  expect(container.textContent).toBe("Hello, Jenny!");

  act(() => {
    render(<Hello name="Margaret" />, container);
  });
  expect(container.textContent).toBe("Hello, Margaret!");
});

```

데이터 가져오기

모든 테스트에서 실제 API를 호출하는 대신에 요청을 모의해서 더미 데이터를 가져올 수 있습니다. "가짜" 데이터를 사용하여 모의 데이터를 가져오면 사용할 수 없는 백엔드로 인해 테스트가 쉽게 망가지는 것을 방지하고 더 빠르게 실행할 수 있습니다. 주의: 일부 테스트에서는 "end-to-end" 프레임워크를 사용하여 전체 애플리케이션의 모든 부분이 함께 잘 동작하는지 살펴볼 수 있습니다.

```

// user.js

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

export default function User(props) {
  const [user, setUser] = useState(null);

  async function fetchUserData(id) {
    const response = await fetch(`/${id}`);
    setUser(await response.json());
  }

  useEffect(() => {
    fetchUserData(props.id);
  }, [props.id]);

  if (!user) {
    return "loading...";
  }

  return (
    <details>
      <summary>{user.name}</summary>
      <strong>{user.age}</strong> years old
    </details>
  );
}

```

```

    <br />
    lives in {user.address}
  </details>
);
}

```

위 컴포넌트의 테스트를 아래와 같이 작성할 수 있습니다.

```

// user.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";
import User from "./user";

let container = null;
beforeEach(() => {
  // 렌더링 대상으로 DOM 엘리먼트를 설정합니다.
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // 기존의 테스트 환경을 정리합니다.
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("renders user data", async () => {
  const fakeUser = {
    name: "Joni Baez",
    age: "32",
    address: "123, Charming Avenue"
  };

  jest.spyOn(global, "fetch").mockImplementation(() =>
    Promise.resolve({
      json: () => Promise.resolve(fakeUser)
    })
  );

  // resolved promises를 적용하려면 `act()`의 비동기 버전을 사용하세요.
  await act(async () => {
    render(<User id="123" />, container);
  });

  expect(container.querySelector("summary").textContent).toBe(fakeUser.name);
  expect(container.querySelector("strong").textContent).toBe(fakeUser.age);
  expect(container.textContent).toContain(fakeUser.address);

  // 테스트가 완전히 격리되도록 mock을 제거하세요.
  global.fetch.mockRestore();
});

```

모듈 모의하기

일부 모듈은 테스트 환경에서 제대로 작동하지 않거나 테스트 자체에 필수적이지 않을 수 있습니다. 이러한 모듈을 더미 모듈로 대체하는 방식으로 모의하여 코드에 대한 테스트를 더욱 쉽게 작성할 수 있습니다.

서드 파티인 `GoogleMap` 컴포넌트를 내장하는 `Contact` 컴포넌트를 살펴보세요

```
// map.js
import React from "react";
import { LoadScript, GoogleMap } from "react-google-maps";
export default function Map(props) {
  return (
    <LoadScript id="script-loader" googleMapsApiKey="YOUR_API_KEY">
      <GoogleMap id="example-map" center={props.center} />
    </LoadScript>
  );
}

// contact.js
import React from "react";
import Map from "./map";

export default function Contact(props) {
  return (
    <div>
      <address>
        Contact {props.name} via{" "}
        <a data-testid="email" href={"mailto:" + props.email}>
          email
        </a>
        or on their <a data-testid="site" href={props.site}>
          website
        </a>.
      </address>
      <Map center={props.center} />
    </div>
  );
}
```

테스트에서 컴포넌트를 로드하지 않는다면, 더미 컴포넌트에 대한 종속성을 모의 처리하고 테스트를 실행할 수 있습니다.

```
// contact.test.js
import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Contact from "./contact";
import MockedMap from "./map";

jest.mock("./map", () => {
  return function DummyMap(props) {
    return (
      <div data-testid="map">
        {props.center.lat}:{props.center.long}
      </div>
    );
  };
});

let container = null;
beforeEach(() => {
  // 렌더링 대상으로 DOM 엘리먼트를 설정합니다.
```

```

    container = document.createElement("div");
    document.body.appendChild(container);
});

afterEach(() => {
  // 기존의 테스트 환경을 정리합니다.
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("should render contact information", () => {
  const center = { lat: 0, long: 0 };
  act(() => {
    render(
      <Contact
        name="Joni Baez"
        email="test@example.com"
        site="http://test.com"
        center={center}
      />,
      container
    );
  });

  expect(
    container.querySelector("[data-testid='email']").getAttribute("href")
  ).toEqual("mailto:test@example.com");

  expect(
    container.querySelector('[data-testid="site"]').getAttribute("href")
  ).toEqual("http://test.com");

  expect(container.querySelector('[data-testid="map"]').textContent).toEqual(
    "0:0"
  );
});
});

```

이벤트

DOM 요소에 실제 DOM 이벤트를 전달한 다음 결과를 검증하는 게 좋습니다. `Toggle` 컴포넌트를 살펴보세요.

```

// toggle.js

import React, { useState } from "react";

export default function Toggle(props) {
  const [state, setState] = useState(false);
  return (
    <button
      onClick={() => {
        setState(previousState => !previousState);
        props.onChange(!state);
      }}
      data-testid="toggle"
    >
      {state === true ? "Turn off" : "Turn on"}
    </button>
  );
}

```

위 컴포넌트의 테스트를 아래와 같이 작성할 수 있습니다.

```
// toggle.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Toggle from "./toggle";

let container = null;
beforeEach(() => {
  // 렌더링 대상으로 DOM 엘리먼트를 설정합니다.
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // 기존의 테스트 환경을 정리합니다.
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("changes value when clicked", () => {
  const onChange = jest.fn();
  act(() => {
    render(<Toggle onChange={onChange} />, container);
  });

  // 버튼 엘리먼트를 가져와서 클릭 이벤트를 트리거 하세요.

  const button = document.querySelector("[data-testid=toggle]");
  expect(button.innerHTML).toBe("Turn on");

  act(() => {
    button.dispatchEvent(new MouseEvent("click", { bubbles: true }));
  });

  expect(onChange).toHaveBeenCalledTimes(1);
  expect(button.innerHTML).toBe("Turn off");

  act(() => {
    for (let i = 0; i < 5; i++) {
      button.dispatchEvent(new MouseEvent("click", { bubbles: true }));
    }
  });

  expect(onChange).toHaveBeenCalledTimes(6);
  expect(button.innerHTML).toBe("Turn on");
});
```

다른 DOM 이벤트와 프로퍼티는 [MDN](#)에 설명되어 있습니다. React는 자동으로 이벤트를 루트로 위임하기 때문에 이벤트마다 `{ bubbles: true }` 를 전달하여 이벤트가 React 이벤트 리스너에 도달할 수 있도록 해야 한다는 것에 주의해주세요.

주의

React Testing Library는 이벤트를 발생시키기 위한 [더욱 간결한 함수](#)를 제공합니다.

타이머

코드는 `setTimeout` 과 같은 타이머 기반 함수를 사용하여 향후 더 많은 작업을 예약 할 수 있습니다. 이 예시에서 다중 선택 패널은 선택을 기다렸다가 5초 이내에 선택하지 않으면 시간이 초과합니다.

```
// card.js

import React, { useEffect } from "react";

export default function Card(props) {
  useEffect(() => {
    const timeoutID = setTimeout(() => {
      props.onSelect(null);
    }, 5000);
    return () => {
      clearTimeout(timeoutID);
    };
  }, [props.onSelect]);
}

return [1, 2, 3, 4].map(choice => (
  <button
    key={choice}
    data-testid={choice}
    onClick={() => props.onSelect(choice)}
  >
    {choice}
  </button>
));
}
```

[Jest's timer mocks](#)를 활용하고 컴포넌트의 다양한 상태를 테스트하여 테스트 코드를 작성할 수 있습니다.

```
// card.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Card from "./card";

let container = null;
beforeEach(() => {
  // 렌더링 대상으로 DOM 엘리먼트를 세팅 합니다.
  container = document.createElement("div");
  document.body.appendChild(container);
  jest.useFakeTimers();
});

afterEach(() => {
  // 기존의 테스트 환경을 정리합니다.
  unmountComponentAtNode(container);
  container.remove();
  container = null;
  jest.useRealTimers();
});
```

```

it("should select null after timing out", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });

  // 시간을 100ms만큼 앞당긴다.
  act(() => {
    jest.advanceTimersByTime(100);
  });
  expect(onSelect).not.toHaveBeenCalled();

  // 그리고 5초만큼 앞당긴다.
  act(() => {
    jest.advanceTimersByTime(5000);
  });
  expect(onSelect).toHaveBeenCalledWith(null);
});

it("should cleanup on being removed", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });

  act(() => {
    jest.advanceTimersByTime(100);
  });
  expect(onSelect).not.toHaveBeenCalled();

  // 마운트 해제한다.
  act(() => {
    render(null, container);
  });

  act(() => {
    jest.advanceTimersByTime(5000);
  });
  expect(onSelect).not.toHaveBeenCalled();
});

it("should accept selections", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });

  act(() => {
    container
      .querySelector("[data-testid='2']")
      .dispatchEvent(new MouseEvent("click", { bubbles: true }));
  });

  expect(onSelect).toHaveBeenCalledWith(2);
});

```

일부 테스트에서만 가짜 타이머를 사용할 수 있습니다. 위에서 `jest.useFakeTimers()` 를 호출해서 활성화했습니다. 주요 장점은 테스트가 실제로 5초 동안 실행될 필요가 없으며 테스트를 위해 컴포넌트 코드를 더 복잡하게 만들 필요가 없다는 점입니다.

스냅샷 테스트

Jest와 같은 프레임워크를 사용하면 `toMatchSnapshot` / `toMatchInlineSnapshot`을 사용하여 데이터의 "스냅샷"을 저장할 수 있습니다. 이를 통해 렌더링 된 컴포넌트 출력을 "저장"하고 컴포넌트 출력의 변경이 스냅샷 변경 사항으로 명시적으로 커밋되도록 할 수 있습니다.

이 예시에서는 인라인 스냅샷으로 저장하기 전에 컴포넌트를 렌더링하고 렌더링 된 HTML을 `pretty` 패키지를 사용해서 포맷을 변환합니다.

```
// hello.test.js, again

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";
import pretty from "pretty";

import Hello from "./hello";

let container = null;
beforeEach(() => {
  // 렌더링 대상으로 DOM 엘리먼트를 설정합니다.
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // 기존의 테스트 환경을 정리합니다.
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("should render a greeting", () => {
  act(() => {
    render(<Hello />, container);
  });

  expect(
    pretty(container.innerHTML)
  ).toMatchSnapshot(); /* ... jest에 의해 자동으로 채워집니다 ... */

  act(() => {
    render(<Hello name="Jenny" />, container);
  });

  expect(
    pretty(container.innerHTML)
  ).toMatchSnapshot(); /* ... jest에 의해 자동으로 채워집니다 ... */

  act(() => {
    render(<Hello name="Margaret" />, container);
  });

  expect(
    pretty(container.innerHTML)
  ).toMatchSnapshot(); /* ... jest에 의해 자동으로 채워집니다 ... */
});
```

일반적으로 스냅샷을 사용하는 것보다 더 구체적인 단언을 만드는 것이 좋습니다. 이러한 종류의 테스트에는 구현 세부 정보가 포함되어 있어 쉽게 중단 할 수 있으며 팀은 스냅샷 손상에 민감하지 않을 수 있습니다. 선택적으로 일부 자식 컴포넌트를 모의하면 스냅샷 크기를 줄이고 코드 리뷰를 위한 가독성을 유지할 수 있습니다.

다수의 렌더러

드문 경우이지만 여러 렌더러를 사용하는 컴포넌트에서 테스트를 실행할 때가 있을 수 있습니다. 예를 들어, 자식 컴포넌트 내에서 react-dom의 render를 내부적으로 사용하여 일부 콘텐츠를 렌더링하는 react-test-renderer가 있는 컴포넌트에서 스냅샷 테스트를 실행할 수 있습니다. 이 시나리오에서는 렌더러에 해당하는 act()로 업데이트를 래핑 할 수 있습니다.

```
import { act as domAct } from "react-dom/test-utils";
import { act as testAct, create } from "react-test-renderer";
// ...
let root;
domAct(() => {
  testAct(() => {
    root = create(<App />);
  });
});
expect(root).toMatchSnapshot();
```

뭔가 부족하다면?

일반적인 시나리오가 다루어지지 않은 경우 issue tracker에 알려주세요.

테스팅 개요

- "community/testing.html"

다른 JavaScript 코드와 비슷한 방법으로 React 컴포넌트를 테스트할 수 있습니다.

React 컴포넌트를 테스트할 수 있는 몇 가지 방법이 있는데 크게는 2가지 카테고리로 나누어집니다.

- **컴포넌트 트리 렌더링** : 간략화된 테스팅 환경 및 출력값이 확실한 경우.
- **완성된 앱에서의 테스트** : 현실적 브라우저 환경("엔드 투 엔드" 테스트라고 알려져 있습니다.)

이 문서 섹션은 아주 기본 테스팅 전략에 초점을 맞추고 있습니다. 풀 엔드투엔드 테스트는 중요한 워크플로의 퇴행을 방지하기 위해 매우 유용하지만 이러한 테스트들은 특정한 React 컴포넌트와 관련되어 있지 않습니다. 그리고 이들은 이 세션의 범위를 벗어납니다.

트레이드오프

테스팅 도구를 선택할 때 몇 가지 트레이드오프를 생각하는 것은 의미 있는 일입니다.

- **반복작업 속도 vs 현실적인 개발 환경** : 몇몇 도구들은 변경사항이 생기고 결과 값이 출력되는 과정에서 매우 빠른 피드백 루프를 제공하지만 브라우저 동작을 정확히 구현하지 않습니다. 다른 도구들이 현실적 브라우저 환경에서 사용되지만 반복 작업 속도를 저하시키고 지속적 통합 서버 환경에서 더 연약합니다.
- **얼마나 다양한 방법을 통해 실제 테스트 환경의 동작을 테스트할 것인가?** 컴포넌트 안에서는 '유닛'테스트와 '통합'테스트의 차이는 명확하지 않습니다. 하나의 폼을 테스팅한다고 할 때 테스트는 또한 버튼의 안의 부분에서 이루어지나요? 또는 버튼 컴포넌트에 알맞은 테스트를 할 수 있나요? 리팩토링된 버튼이 폼 테스트에 오류를 일으키나요?

각각의 팀과 제품에 따라 다른 답이 적용될 수 있습니다.

추천 도구

Jest는 JavaScript 테스트 러너입니다. DOM에 접근하게 하는 `jsdom`을 통해서 jsdom은 단지 어떻게 브라우저가 작동하는지에 대한 대략적 개요입니다. 하지만 이는 종종 React 컴포넌트를 테스팅하기에 충분합니다. Jest는 mocking `modules`과 `timers` 같은 파워풀한 특징과 결합되어 훌륭한 반복속도를 제공합니다. 그래서 더 많은 코드 제어를 가집니다.

React Testing Library는 실행 디테일을 가지지 않는 React 컴포넌트를 테스트하게 하는 도구 모음입니다. 이러한 접근은 리팩토링을 수월하게 하며 접근성에 대한 가장 좋은 연습을 가능하게 합니다. 자식 컴포넌트를 가지지 않는 컴포넌트에 대한 얇은 렌더링 방법을 제공하지 않더라도 Jest와 같은 테스트 러너는 `mocking`에 의해 위를 가능하게 합니다.

더 학습하기

두 개의 페이지로 나뉘어 있습니다.

- [Recipes](#): React 컴포넌트에 대한 테스트를 작성할 때 흔한 패턴
- [Environments](#): React 컴포넌트에 대한 테스팅 환경을 설정할 때 고려해야 할 것

React로 사고하기

- 'blog/2013/11/05/thinking-in-react.html'
- 'docs/thinking-in-react-zh-CN.html'

React는 JavaScript로 규모가 크고 빠른 웹 애플리케이션을 만드는 가장 좋은 방법입니다. React는 Facebook과 Instagram을 통해 확장성을 입증했습니다.

React의 가장 멋진 점 중 하나는 앱을 설계하는 방식입니다. 이 문서를 통해 React로 상품들을 검색할 수 있는 데이터 테이블을 만드는 과정을 함께 생각해 봅시다.

목업으로 시작하기

JSON API와 목업을 디자이너로부터 받았다고 가정해 봅시다. 목업은 다음과 같을 것입니다.

<input type="text" value="Search..."/>
<input type="checkbox"/> Only show products in stock
Name Price
Sporting Goods
Football \$49.99
Baseball \$9.99
Basketball \$29.99
Electronics
iPod Touch \$99.99
iPhone 5 \$399.99
Nexus 7 \$199.99

목업

JSON API는 아래와 같은 데이터를 반환합니다.

```
[ ];
```

1단계: UI를 컴포넌트 계층 구조로 나누기

우리가 할 첫 번째 일은 모든 컴포넌트(와 하위 컴포넌트)의 주변에 박스를 그리고 그 각각에 이름을 붙이는 것입니다. 디자이너와 함께 일한다면, 이것들을 이미 정해두었을 수 있으니 한번 대화해보세요! 디자이너의 Photoshop 레이어 이름이 React 컴포넌트의 이름이 될 수 있습니다.

하지만 어떤 것이 컴포넌트가 되어야 할지 어떻게 알 수 있을까요? 우리가 새로운 함수나 객체를 만들 때처럼 만드시면 됩니다. 한 가지 테크닉은 [단일 책임 원칙](#)입니다. 이는 하나의 컴포넌트는 한 가지 일을 하는게 이상적이라는 원칙입니다. 하나의 컴포넌트가 커지게 된다면 이는 보다 작은 하위 컴포넌트로 분리되어야 합니다.

주로 JSON 데이터를 유저에게 보여주기 때문에, 데이터 모델이 적절하게 만들어졌다면, UI(컴포넌트 구조)가 잘 연결될 것입니다. 이는 UI와 데이터 모델이 같은 [인포메이션 아키텍처](#)(information architecture)를 가지는 경향이 있기 때문입니다. 각 컴포넌트가 데이터 모델의 한 조각을 나타내도록 분리해주세요.



Diagram showing nesting of components

다섯개의 컴포넌트로 이루어진 앱을 한번 봅시다. 각각의 컴포넌트에 들어간 데이터는 이탤릭체로 표기했습니다. 이미지의 숫자는 아래 숫자에 해당됩니다.

1. `FilterableProductTable` (노란색): 예시 전체를 포함합니다.
2. `SearchBar` (파란색): 모든 유저의 입력(*user input*)을 받습니다.
3. `ProductTable` (연두색): 유저의 입력(*user input*)을 기반으로 데이터 컬렉션(*data collection*)을 필터링 해서 보여줍니다.
4. `ProductCategoryRow` (하늘색): 각 카테고리(*category*)의 헤더를 보여줍니다.
5. `ProductRow` (빨강색): 각각의 제품(*product*)에 해당하는 행을 보여줍니다.

`ProductTable`을 보면 “Name”과 “Price” 레이블을 포함한 테이블 헤더만을 가진 컴포넌트는 없습니다. 이 같은 경우, 데이터를 위한 독립된 컴포넌트를 생성할지 생성하지 않을지는 선택입니다. 이 예시에서는 `ProductTable`의 책임인 데이터 컬렉션(*data collection*)이 렌더링의 일부이기 때문에 `ProductTable`을 남겨두었습니다. 그러나 이 헤더가 복잡해지면 (즉 정렬을 위한 기능을 추가하는 등) `ProductTableHeader` 컴포넌트를 만드는 것이 더 합리적일 것입니다.

이제 목업에서 컴포넌트를 확인하였으므로 이를 계층 구조로 나열해봅시다. 모형의 다른 컴포넌트 내부에 나타나는 컴포넌트는 계층 구조의 자식으로 나타냅니다.

- `FilterableProductTable`
 - `SearchBar`
 - `ProductTable`
 - `ProductCategoryRow`
 - `ProductRow`

데이터 모델을 렌더링하는 앱의 정적 버전을 만들기 위해 다른 컴포넌트를 재사용하는 컴포넌트를 만들고 `props`를 이용해 데이터를 전달해줍니다. `props`는 부모가 자식에게 데이터를 넘겨줄 때 사용할 수 있는 방법입니다. 정적 버전을 만들기 위해 **state**를 사용하지 마세요. `state`는 오직 상호작용을 위해, 즉 시간이 지남에 따라 데이터가 바뀌는 것에 사용합니다. 우리는 앱의 정적 버전을 만들고 있기 때문에 지금은 필요하지 않습니다.

앱을 만들 때 하향식(top-down)이나 상향식(bottom-up)으로 만들 수 있습니다. 다시 말해 계층 구조의 상층부에 있는 컴포넌트 (즉 `FilterableProductTable` 부터 시작하는 것)부터 만들거나 하층부에 있는 컴포넌트 (`ProductRow`) 부터 만들 수도 있습니다. 간단한 예시에서는 보통 하향식으로 만드는 게 쉽지만 프로젝트가 커지면 상향식으로 만들고 테스트를 작성하면서 개발하기가 더 쉽습니다.

이 단계가 끝나면 데이터 렌더링을 위해 만들어진 재사용 가능한 컴포넌트들의 라이브러리를 가지게 됩니다. 현재는 앱의 정적 버전이기 때문에 컴포넌트는 `render()` 메서드만 가지고 있을 것입니다. 계층구조의 최상단 컴포넌트 (`FilterableProductTable`)는 `prop`으로 데이터 모델을 받습니다. 데이터 모델이 변경되면 `root.render()`를 다시 호출해서 UI가 업데이트 됩니다. UI가 어떻게 업데이트되고 어디에서 변경해야하는지 알 수 있습니다. React의 단방향 데이터 흐름(**one-way data flow**) (또는 단방향 바인딩(**one-way binding**))은 모든 것을 모듈화 하고 빠르게 만들어줍니다.

이 단계를 실행하는 데 도움이 필요하다면 [공식 React 문서](#)를 참고하세요.

짧은 소개: Props vs State

React에는 두 가지 데이터 "모델"인 `props`와 `state`가 있습니다. 이 둘 사이의 차이점을 이해하는 것이 중요합니다. 차이점이 제대로 기억나지 않는다면 [공식 React 문서](#)와 [자주 묻는 질문: state와 props의 차이점은 무엇인가요?](#)까지 살펴보세요.

3단계: UI state에 대한 최소한의 (하지만 완전한) 표현 찾아내기

UI를 상호작용하게 만들려면 기반 데이터 모델을 변경할 수 있는 방법이 있어야 합니다. 이를 React는 **state**를 통해 변경합니다.

애플리케이션을 올바르게 만들기 위해서는 애플리케이션에서 필요로 하는 변경 가능한 `state`의 최소 집합을 생각해보아야 합니다. 여기서 핵심은 [중복배제](#) 원칙입니다. 애플리케이션이 필요로 하는 가장 최소한의 `state`를 찾고 이를 통해 나머지 모든 것들이 필요에 따라 그때그때 계산되도록 만드세요. 예를 들어 TODO 리스트를 만든다고 하면, TODO 아이템을 저장하는 배열만 유지하고 TODO 아이템의 개수를 표현하는 `state`를 별도로 만들지 마세요. TODO 갯수를 렌더링해야한다면 TODO 아이템 배열의 길이를 가져오면 됩니다.

예시 애플리케이션 내 데이터들을 생각해봅시다. 애플리케이션은 다음과 같은 데이터를 가지고 있습니다.

- 제품의 원본 목록
- 유저가 입력한 검색어
- 체크박스의 값
- 필터링 된 제품들의 목록

각각 살펴보고 어떤 게 `state`가 되어야 하는지 살펴봅시다. 이는 각 데이터에 대해 아래의 세 가지 질문을 통해 결정할 수 있습니다.

1. 부모로부터 `props`를 통해 전달됩니까? 그러면 확실히 `state`가 아닙니다.
2. 시간이 지나도 변하지 않나요? 그러면 확실히 `state`가 아닙니다.
3. 컴포넌트 안의 다른 `state`나 `props`를 가지고 계산 가능한가요? 그렇다면 `state`가 아닙니다.

제품의 원본 목록은 `props`를 통해 전달되므로 `state`가 아닙니다. 검색어와 체크박스는 `state`로 볼 수 있는데 시간이 지남에 따라 변하기도 하면서 다른 것들로부터 계산될 수 없기 때문입니다. 그리고 마지막으로 필터링된 목록은 `state`가 아닙니다. 제품의 원본 목록과 검색어, 체크박스의 값을 조합해서 계산해낼 수 있기 때문입니다.

결과적으로 애플리케이션은 다음과 같은 state를 가집니다.

- 유저가 입력한 검색어
- 체크박스의 값

4단계: State가 어디에 있어야 할지 찾기

```

class ProductCategoryRow extends React.Component {
  render() {
    const category = this.props.category;
    return (
      <tr>
        <th colSpan="2">
          {category}
        </th>
      </tr>
    );
  }
}

class ProductRow extends React.Component {
  render() {
    const product = this.props.product;
    const name = product.stocked ?
      product.name :
      <span style={{color: 'red'}}>
        {product.name}
      </span>;
    return (
      <tr>
        <td>{name}</td>
        <td>{product.price}</td>
      </tr>
    );
  }
}

```

Resources View Compiled

좋습니다. 이제 앱에서 최소한으로 필요한 state가 뭔지 찾아냈습니다. 다음으로는 어떤 컴포넌트가 state를 변경하거나 소유할지 찾아야 합니다.

기억하세요: React는 항상 컴포넌트 계층구조를 따라 아래로 내려가는 단방향 데이터 흐름을 따릅니다. 어떤 컴포넌트가 어떤 state를 가져야 하는지 바로 결정하기 어려울 수 있습니다. 많은 초보자가 이 부분을 가장 어려워합니다. 아래 과정을 따라 결정해 보세요.

애플리케이션이 가지는 각각의 state에 대해서

- state를 기반으로 렌더링하는 모든 컴포넌트를 찾으세요.
- 공통 소유 컴포넌트 (common owner component)를 찾으세요. (계층 구조 내에서 특정 state가 있어야 하는 모든 컴포넌트들의 상위에 있는 하나의 컴포넌트).
- 공통 혹은 더 상위에 있는 컴포넌트가 state를 가져야 합니다.
- 4단계에서 state를 소유할 적절한 컴포넌트를 찾지 못하였다면, state를 소유하는 컴포넌트를 하나 만들어서 공통 소유 컴포넌트의 상위 계층에 추가하세요.

이 저력을 애플리케이션에 적용해봅시다.

4단계의 예시에서 체크하거나 키보드를 타이핑할 경우 React가 입력을 무시하는 것을 확인할 수 있습니다. 이는 `input` 태그의 `value` 속성이 항상 `FilterableProductTable`에서 전달된 `state`와 동일하도록 설정했기 때문입니다.

우리가 원하는 것이 무엇인지를 한번 생각해봅시다. 우리는 사용자가 품을 변경할 때마다 사용자의 입력을 반영할 수 있도록 `state`를 업데이트하기를 원합니다. 컴포넌트는 그 자신의 `state`만 변경할 수 있기 때문에 `FilterableProductTable`는 `SearchBar`에 콜백을 넘겨서 `state`가 업데이트되어야 할 때마다 호출되도록 할 것입니다. 우리는 `input`에 `onChange` 이벤트를 사용해서 알림을 받을 수 있습니다. `FilterableProductTable`에서 전달된 콜백은 `setState()`를 호출하고 앱이 업데이트될 것입니다.

이게 전부입니다.

이 글을 통해 React를 가지고 애플리케이션과 컴포넌트를 만드는 데에 대한 사고방식을 얻어갈 수 있기를 바랍니다. 이전보다 더 많은 타이핑을 해야 할 수 있지만, 코드를 쓸 일보다 읽을 일이 더 많다는 사실을 기억하세요. 모듈화되고 명시적인 코드는 읽을 때 조금 덜 어렵습니다. 큰 컴포넌트 라이브러리를 만들게 되면 이 명시성과 모듈성에 감사할 것이며 코드 재사용성을 통해 코드 라인이 줄어들기 시작할 것입니다. :)

PropTypes와 함께 하는 타입 검사

주의

`React.PropTypes` 는 React v15.5부터 다른 패키지로 이동하였습니다. 대신 `prop-types` 라이브러리를 사용하시길 바랍니다.

우리는 변환을 자동화하기 위하여 [codemod 스크립트](#)를 제공하고 있습니다.

여러분의 앱이 커짐에 따라 타입 검사를 활용하면 많은 버그(bug) 잡을 수 있습니다. 특정 애플리케이션에서는 전체 애플리케이션의 타입 검사를 위해 [Flow](#) 또는 [TypeScript](#)와 같은 JavaScript 도구(Extensions)를 사용할 수 있습니다. 이러한 것들을 사용하지 않더라도 React는 내장된 타입 검사 기능들을 가지고 있습니다. 컴포넌트의 `props`에 타입 검사를 하려면 다음과 같이 특별한 프로퍼티인 `propTypes`를 선언할 수 있습니다.

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

class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}

Greeting.propTypes = {
  name: PropTypes.string
};
```

예시에서 클래스 컴포넌트를 사용하고 있지만 함수 컴포넌트와 `React.memo`, `React.forwardRef` 를 통해 만들어진 컴포넌트에도 동일하게 적용될 수 있습니다.

`PropTypes` 는 전달받은 데이터의 유효성을 검증하기 위해서 다양한 유효성 검사기(Validator)를 내보냅니다. 아래 예시에서는 `PropTypes.string` 을 사용하게 될 것입니다. `prop`에 유효하지 않은 값이 전달 되었을 때, 경고문이 JavaScript 콘솔을 통해 보일 것입니다. `propTypes` 는 성능상의 이유로 개발 모드(Development mode) 에서만 확인될 것입니다.

PropTypes

아래는 제공된 서로 다른 유효성 검사기들을 보여주는 예시입니다.

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

MyComponent.propTypes = {
  // prop가 특정 JS 형식임을 선언할 수 있습니다.
  // 이것들은 기본적으로 모두 선택 사항입니다.
  optionalArray: PropTypes.array,
  optionalBool: PropTypes.bool,
  optionalFunc: PropTypes.func,
  optionalNumber: PropTypes.number,
  optionalObject: PropTypes.object,
  optionalString: PropTypes.string,
```

```

optionalSymbol: PropTypes.symbol,
// 랜더링 될 수 있는 것들은 다음과 같습니다.
// 숫자(numbers), 문자(strings), 엘리먼트(elements), 또는 이러한 타입들(types)을 포함하고 있는 배열(array) (혹은 배열의 fragment)
optionalNode: PropTypes.node,
// React 엘리먼트.
optionalElement: PropTypes.element,
// React 엘리먼트 타입 (ie. MyComponent)
optionalElementType: PropTypes.elementType,
// prop가 클래스의 인스턴스임을 선언할 수 있습니다.
// 이 경우 JavaScript의 instanceof 연산자를 사용합니다.
optionalMessage: PropTypes.instanceOf(Message),
// 열거형(enum)으로 처리하여 prop가 특정 값들로 제한되도록 할 수 있습니다.
optionalEnum: PropTypes.oneOf(['News', 'Photos']),
// 여러 종류중 하나의 종류가 될 수 있는 객체
optionalUnion: PropTypes.oneOfType([
  PropTypes.string,
  PropTypes.number,
  PropTypes.instanceOf(Message)
]),
// 특정 타입의 행렬
optionalArrayOf: PropTypes.arrayOf(PropTypes.number),
// 특정 타입의 프로퍼티 값들을 갖는 객체
optionalObjectOf: PropTypes.objectOf(PropTypes.number),
// 특정 형태를 갖는 객체
optionalObjectWithShape: PropTypes.shape({
  color: PropTypes.string,
  fontSize: PropTypes.number
}),
// 추가 프로퍼티에 대한 경고가 있는 객체
optionalObjectWithStrictShape: PropTypes.exact({
  name: PropTypes.string,
  quantity: PropTypes.number
}),
// 위에 있는 것 모두 `isRequired`와 연결하여 prop가 제공되지 않았을 때
// 경고가 보이도록 할 수 있습니다.
requiredFunc: PropTypes.func.isRequired,
// 모든 데이터 타입이 가능한 필수값
requiredAny: PropTypes.any.isRequired,
// 사용자 정의 유효성 검사를 지정할 수도 있습니다.
// 검사 실패 시에는 에러(Error) 객체를 반환해야 합니다.
// `oneOfType`안에서는 작동하지 않으므로 `console.warn` 혹은 `throw` 하지 마세요.
customProp: function(props, propName, componentName) {
  if (!/matchme/.test(props[propName])) {
    return new Error(
      'Invalid prop `' + propName + '` supplied to' +
      ' `' + componentName + '`. Validation failed.'
    );
}

```

```

},

// `arrayOf` 와 `objectOf`에 사용자 정의 유효성 검사를 적용할 수 있습니다.
// 검사 실패 시에는 에러(Error) 객체를 반환해야 합니다.
// 유효성 검사기는 배열(array) 혹은 객체의 각 키(key)에 대하여 호출될 것입니다.
// 유효성 검사기의 첫 두 개의 변수는 배열 혹은 객체 자신과 현재 아이템의 키입니다.

customArrayProp: PropTypes.arrayOf(function(propValue, key, componentName,
location, propFullName) {
  if (!/matchme/.test(propValue[key])) {
    return new Error(
      'Invalid prop `' + propFullName + '` supplied to' +
      ' `' + componentName + '`. Validation failed.'
    );
  }
});

```

하나의 자식만 요구하기

`PropTypes.element`를 이용하여 컴포넌트의 자식들(Children)에 단 하나의 자식(Child)만이 전달될 수 있도록 명시할 수 있습니다.

```

import PropTypes from 'prop-types';

class MyComponent extends React.Component {
  render() {
    // 이것은 반드시 하나의 엘리먼트여야 합니다. 아니라면, 경고(warn)가 일어날 것입니다.
    const children = this.props.children;
    return (
      <div>
        {children}
      </div>
    );
  }
}

MyComponent.propTypes = {
  children: PropTypes.element.isRequired
};

```

초기 Prop 값

`defaultProps` 프로퍼티를 할당함으로써 `props`의 초기값을 정의할 수 있습니다.

```

class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}

// props의 초기값을 정의합니다.
Greeting.defaultProps = {
  name: 'Stranger'
};

```

```
// Renders "Hello, Stranger":
const root = ReactDOM.createRoot(document.getElementById('example'));
root.render(<Greeting />);
```

Since ES2022 you can also declare `defaultProps` as static property within a React component class. For more information, see the [class public static fields](#). This modern syntax will require a compilation step to work within older browsers.

```
class Greeting extends React.Component {
  static defaultProps = {
    name: 'stranger'
  }

  render() {
    return (
      <div>Hello, {this.props.name}</div>
    )
  }
}
```

`defaultProps` 는 `this.props.name`의 값이 부모 컴포넌트에 의해 명시되지 않았을 때 값을 갖도록 할 것입니다. `propTypes` 의 타입 검사는 `defaultProps` 에도 적용되게 하기 위하여 `defaultProps` 가 처리된 뒤에 일어날 것입니다.

Function Components

함수 컴포넌트를 사용해서 개발한다면, `PropTypes`를 적절히 적용할 수 있도록 몇 가지 작은 변경사항을 만들어낼 수도 있습니다.

아래와 같은 컴포넌트가 있다고 생각해봅시다.

```
export default function HelloWorldComponent({ name }) {
  return (
    <div>Hello, {name}</div>
  )
}
```

`PropTypes`를 추가하려면 아래처럼 컴포넌트를 외부에 노출시키기 전에 별도의 함수로 컴포넌트를 선언할 수 있습니다.

```
function HelloWorldComponent({ name }) {
  return (
    <div>Hello, {name}</div>
  )
}

export default HelloWorldComponent
```

그러면, `HelloWorldComponent`에 직접 `PropTypes`를 추가할 수 있습니다.

```
import PropTypes from 'prop-types'

function HelloWorldComponent({ name }) {
  return (
    <div>Hello, {name}</div>
  )
}

HelloWorldComponent.propTypes = {
```

```
    name: PropTypes.string  
}  
  
export default HelloWorldComponent
```

[Go to TOC](#)

비제어 컴포넌트

대부분 경우에 폼을 구현하는데 [제어 컴포넌트](#)를 사용하는 것이 좋습니다. 제어 컴포넌트에서 폼 데이터는 React 컴포넌트에서 다루어집니다. 대안인 비제어 컴포넌트는 DOM 자체에서 폼 데이터가 다루어집니다.

모든 state 업데이트에 대한 이벤트 핸들러를 작성하는 대신 비제어 컴포넌트를 만들려면 [ref를 사용](#)하여 DOM에서 폼 값을 가져올 수 있습니다.

예를 들어 아래 코드는 비제어 컴포넌트에 단일 이름을 허용합니다.

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.input = React.createRef();
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.input.current.value);
    event.preventDefault();
  }

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

[CodePen에서 실행하기](#)

비제어 컴포넌트는 DOM에 신뢰 가능한 출처를 유지하므로 비제어 컴포넌트를 사용할 때 React와 non-React 코드를 통합하는 것이 쉬울 수 있습니다. 빠르고 간편하게 적은 코드를 작성할 수 있지만, 그 외에는 일반적으로 제어된 컴포넌트를 사용해야 합니다.

특정 상황에서 사용해야 하는 컴포넌트의 타입이 명확하지 않은 경우, [제어 입력과 비제어 입력에 대한 글](#)이 도움이 될 것입니다.

기본 값

React 렌더링 생명주기에서 폼 엘리먼트의 `value` 어트리뷰트는 DOM의 `value`를 대체합니다. 비제어 컴포넌트를 사용하면 React 초기값을 지정하지만, 그 이후의 업데이트는 제어하지 않는 것이 좋습니다. 이러한 경우에 `value` 어트리뷰트 대신 `defaultValue`를 지정할 수 있습니다. 컴포넌트가 마운트된 후에 `defaultValue` 어트리뷰트를 변경해도 DOM의 값이 업데이트되지 않습니다.

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <label>
        Name:
        <input
```

```

        defaultValue="Bob"
        type="text"
        ref={this.input} />
    </label>
    <input type="submit" value="Submit" />
</form>
);
}

```

또한 `<input type="checkbox">` 와 `<input type="radio">` 는 `defaultChecked` 를 지원하고 `<select>` 와 `<textarea>` 는 `defaultValue` 를 지원합니다.

파일 입력 태그

HTML에서 `<input type="file">` 은 사용자가 장치 저장소에서 하나 이상의 파일을 선택하여 서버에 업로드하거나 [파일 API](#)를 사용하여 JavaScript로 조작할 수 있습니다.

```
<input type="file" />
```

React에서 `<input type="file" />` 은 프로그래밍적으로 값을 설정 할 수 없고 사용자만이 값을 설정할 수 있기 때문에 항상 비제어 컴포넌트입니다.

파일 API를 사용하여 파일과 상호작용해야 합니다. 아래 예시에서는 제출 핸들러에서 파일에 접근하기 위해서 DOM 노드의 `ref`를 만드는 방법을 보여주고 있습니다.

`embed:uncontrolled-components/input-type-file.js`

[CodePen에서 실행하기](#)

[Go to TOC](#)

웹 컴포넌트

React와 웹 컴포넌트는 서로 다른 문제를 해결하기 위해 만들어졌습니다. 웹 컴포넌트는 재사용할 수 있는 컴포넌트에 강한 캡슐화를 제공하는 반면, React는 데이터와 DOM을 동기화하는 선언적 라이브러리를 제공합니다.

두 목표는 상호보완적입니다. React에서 웹 컴포넌트를 사용할지, 웹 컴포넌트에서 React를 사용할지, 둘 다 사용할지는 자유롭게 정하실 수 있습니다.

React에서 웹 컴포넌트 사용하기

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

주의

웹 컴포넌트는 종종 강제성을 띠는 API를 열어놓고 있습니다. 예를 들어, `video`라는 웹 컴포넌트는 `play()`나 `pause()`라는 함수를 열어놓고 있을 것입니다. 이러한 웹 컴포넌트의 강제성을 띠는 API에 접근하기 위해서, DOM 노드에 직접 `ref`를 지정하는 것이 필요할 수 있습니다. 서드 파티 웹 컴포넌트를 사용 중이라면, 가장 좋은 해결방법은 웹 컴포넌트의 래퍼로서 동작하는 React 컴포넌트를 작성하는 것입니다.

웹 컴포넌트에서 나온 이벤트들은 React 렌더링 트리에 올바르게 전파되지 않을 수 있습니다. 이를 해결하기 위해 이벤트를 다루기 위한 핸들러를 React 컴포넌트 내에 각각 만들어야합니다.

많은 사람이 공통으로 착각하는 부분 중 하나로, 웹 컴포넌트는 “`className`”이 아닌 “`class`”를 사용합니다.

```
function BrickFlipbox() {
  return (
    <brick-flipbox class="demo">
      <div>front</div>
      <div>back</div>
    </brick-flipbox>
  );
}
```

웹 컴포넌트에서 React 사용하기

```
class XSearch extends HTMLElement {
  connectedCallback() {
    const mountPoint = document.createElement('span');
    this.attachShadow({ mode: 'open' }).appendChild(mountPoint);

    const name = this.getAttribute('name');
    const url = 'https://www.google.com/search?q=' + encodeURIComponent(name);
    const root = ReactDOM.createRoot(mountPoint);
    root.render(<a href={url}>{name}</a>);
  }
}
```

```
    }
}
customElements.define('x-search', XSearch);
```

주의

Babel로 클래스를 변환하면 이 코드가 작동하지 않을 것입니다. 해당 문제를 참조해주시기 바랍니다. 이 문제를 해결하려면 웹 컴포넌트를 불러오기 전에 [custom-elements-es5-adapter](#)를 추가하기 바랍니다.

[Go to TOC](#)

Colophon

This book is created by using the following sources:

- React - 한국어
- GitHub source: reactjs/ko.reactjs.org/content
- Created: 2022-11-28
- Bash v5.2.2
- Vivliostyle, <https://vivliostyle.org/>
- By: @shinokada
- Viewer: <https://read-html-download-pdf.vercel.app/>
- GitHub repo: <https://github.com/shinokada/markdown-docs-as-pdf>
- Viewer repo: <https://github.com/shinokada/read-html-download-pdf>