Untitled

리액트

페이스북에서 개발하고 관리하는 UI 라이브러리. UI 기능만 제공(전역 상태관리, 라우팅, 빌드 시스템등을 각자 직접 구축)

create-react-app

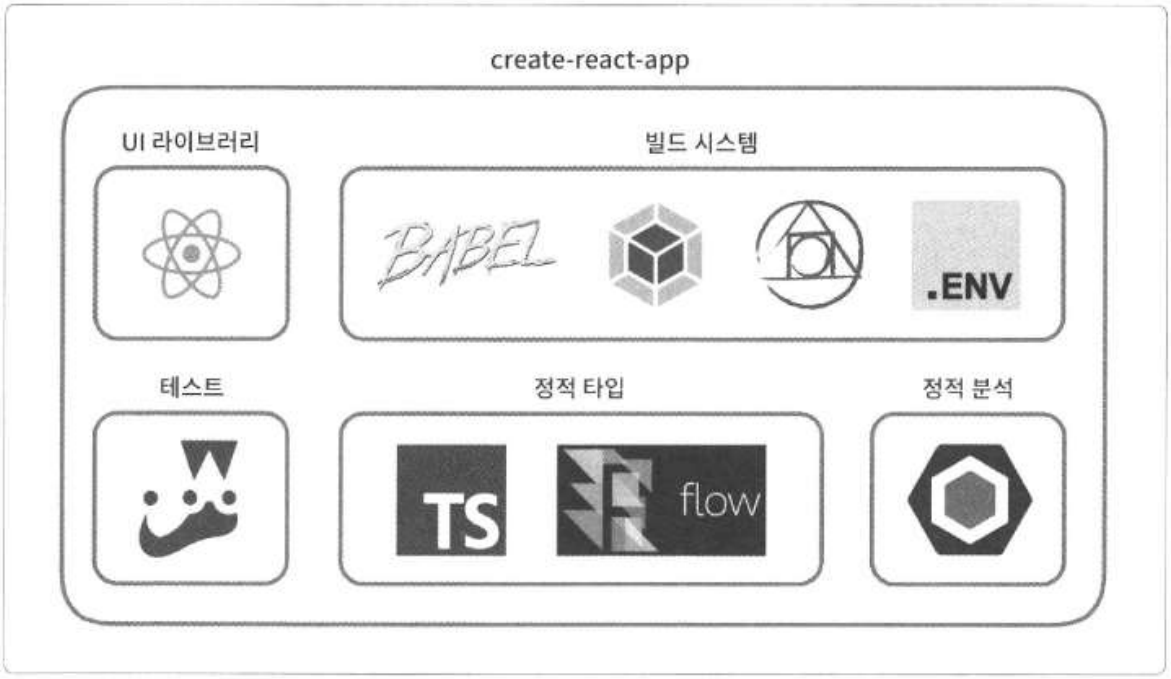


그림 1-1 create-react-app은 여러 패키지를 조합해서 리액트 개발 환경을 구축한다

리액트 사용 이유

UI를 자동으로 업데이트해 준다

UI = render(state)

sample

코드 1-1 **simple1.html**

```
<html>
  <body>
    <h2>안녕하세요. 이 프로젝트가 마음에 드시면 좋아요 버튼을 눌러 주세요.</h2>
    <div id="react-root"></div> ❶
    <script src="react.development.js"/></script>
    <script src="react-dom.development.js"></script> ❷
    <script src="simple1.js"></script> ❸
  </body>
</html>
```

코드 1-2 **simple1.js**

```
class LikeButton extends React.Component { ❶
  constructor(props) {
    super(props);
    this.state = { liked: false }; ❷
  }
  render() {
    const text = this.state.liked ? '좋아요 취소' : '좋아요'; ❸
    return React.createElement( ❹
      'button',
      { onClick: () => this.setState({ liked: true }) }, ❺
      text,
    );
  }
}
const domContainer = document.querySelector('#react-root'); ❻
ReactDOM.render(React.createElement(LikeButton), domContainer); ❼
```

createElement 이해하기

createElement 함수의 구조는 다음과 같다.

```
React.createElement(component, props, ...children) => ReactElement
```

첫 번째 매개변수 component는 일반적으로 문자열이나 리액트 컴포넌트다. component의 인수가 문자열이면 HTML 태그에 해당하는 돔 요소가 생성된다. 예를 들어, 문자열 p를 입력하면 HTML p 태그가 생성된다.

두 번째 매개변수 props는 컴포넌트가 사용하는 데이터를 나타낸다. 돔 요소의 경우 style, className 등의 데이터가 사용될 수 있다.

세 번째 매개변수 children은 해당 컴포넌트가 감싸고 있는 내부의 컴포넌트를 가리킨다. div 태그가 두 개의 p 태그를 감싸고 있는 경우에 다음과 같이 작성할 수 있다.

코드 1-3 **createElement 사용법**

```
<div>
  <p>hello</p>
  <p>world</p>
</div>
```

❶

```
createElement(
  'div',
  null,
  createElement('p', null, 'hello'),
  createElement('p', null, 'world'),
)
```

❷

❶ 일반적인 HTML 코드다. ❷ 같은 코드를 createElement 함수를 사용해서 작성했다. 대부분의 리액트 개발자는 createElement를 직접 작성하지 않는다. 일반적으로 바벨(babel)의 도움을 받아서 JSX 문법을 사용한다. 이는 createElement 함수보다는 JSX 문법으로 작성하는 리액트 코드가 훨씬 가독성이 좋기 때문이다. 바벨을 통한 JSX 문법의 사용은 잠시 후 설명한다.

Sample2

코드 1-4 **simple2.html**

```
<html>
  <body>
    <h2>안녕하세요. 이 프로젝트가 마음에 드시면 좋아요 버튼을 눌러 주세요.</h2>
    <div id="react-root1"></div>
    <!-- ... -->
    <div id="react-root2"></div>
    <!-- ... -->
    <div id="react-root3"></div>
    <script src="react.development.js"/></script>
    <script src="react-dom.development.js"></script>
    <script src="simple2.js"></script> ❷
  </body>
</html>
```

❶

코드 1-5 **simple2.js**

```
// ... ❶
ReactDOM.render(
  React.createElement(LikeButton),
  document.querySelector('#react-root1'),
);
ReactDOM.render(
  React.createElement(LikeButton),
  document.querySelector('#react-root2'),
);
ReactDOM.render(
  React.createElement(LikeButton),
  document.querySelector('#react-root3'),
);
```

Sample3

코드 1-6 **simple3.js**

```
class LikeButton extends React.Component {
  // 기존 코드와 같음
}
class Container extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  render() {
    return React.createElement(
      'div',
      null,
      React.createElement(LikeButton),
      React.createElement(
        'div',
        { style: { marginTop: 20 } },
        React.createElement('span', null, '현재 카운트: '),
        React.createElement('span', null, this.state.count),
        React.createElement(
          'button',
          { onClick: () => this.setState({ count: this.state.count + 1 }) },
          '증가',
        ),
        React.createElement(
          'button',
          { onClick: () => this.setState({ count: this.state.count - 1 }) },
          '감소',
        ),
      ),
    );
  }
}
const domContainer = document.querySelector('#react-root');
ReactDOM.render(React.createElement(Container), domContainer); ❷
```

❶

바벨

자바스크립트 코드를 변환해 주는 컴파일러

리액트에서는 JSX 문법을 사용하기 위해 바벨을 사용한다. 바벨이 JSX문법으로 작성된 코드를 createElement 함수를 호출하는 코드로 변환해 준다.

Sample4

코드 1-7 **simple4.js**

```
class Container extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }
  render() {
    return (
      <div> ❶
        <LikeButton />
        <div style={{ marginTop: 20 }}>
          <span>현재 카운트: </span>
          <span>{this.state.count}</span>
          <button
            onClick={() => this.setState({ count: this.state.count + 1 })}
          >
            증가
          </button>
          <button
            onClick={() => this.setState({ count: this.state.count - 1 })}
          >
            감소
          </button>
        </div>
      </div>
    );
  }
}
```

JSX 문법

JSX 문법 알아보기

JSX는 HTML에서 태그를 사용하는 방식과 유사하다. createElement 함수를 사용해서 렌더 함수를 작성하는 것보다는 JSX 문법을 사용하는 게 간결하고 가독성도 좋다. HTML 태그와의 가장 큰 차이는 속성값을 작성하는 방법에 있다.

코드 1-8 **JSX 문법 사용 예**

```
<div className="box"> ❶
  <Title text="hello world"  width={200} /> ❷
  <button onClick={() => {}}>좋아요</button> ❸
  <a href="/home" style={{ marginTop: '10px', color: 'red' }}> ❹
    홈으로 이동
  </a>
</div>
```

❶ HTML에서 돔 요소에 CSS 클래스 이름을 부여할 때 class 키워드를 사용했다면, JSX에서는 className 키워드를 사용한다. 이는 class라는 이름이 자바스크립트의 class 키워드와 같기 때문이다.

❷ Title은 리액트 컴포넌트다. JSX에서는 돔 요소와 리액트 컴포넌트를 같이 사용할 수 있다. Title 컴포넌트는 text, width라는 두 개의 속성값을 입력받는다. width처럼 문자열 리터럴이 아닌 속성값은 중괄호를 사용해서 입력한다. Title 컴포넌트가 두 개의 속성값을 이용해서 어떤 돔 요소로 치환될지는 Title 컴포넌트의 렌더 함수가 결정한다.

❸ 이벤트 처리 함수는 브라우저마다 다르게 동작할 수 있기 때문에 리액트와 같은 라이브러리를 사용하지 않을 때는 주의해야 한다. 다행히 리액트에서는 이벤트 처리 함수를 호출할 때 브라우저에 상관없이 통일된 이벤트 객체(SyntheticEvent)를 전달해 준다.

❹ HTML에서 돔 요소에 직접 스타일을 적용하는 것과 같이 JSX에서도 스타일을 적용할 수 있다. 다만 자바스크립트에서는 속성 이름에 대시(-)로 연결되는 이름을 사용하는 게 힘들기 때문에 카멜 케이스(camel case)를 사용한다.

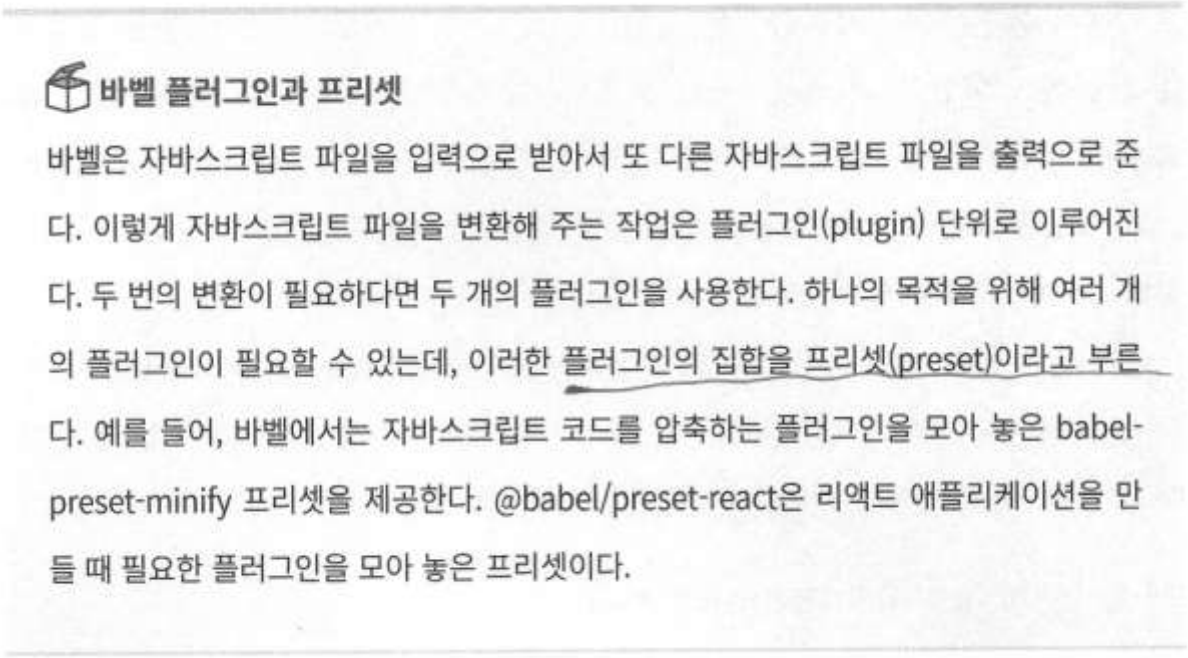
바벨로 컴파일 하기

```shell
npm install @babel/core @babel/cli @babel/preset-react
```

Shell

@babel/cli - 커맨드 라인에서 바벨 실행할 수 있는 바이너리 파일

@babel/preset-react - JSX로 작성된 코드를 createElement 함수를 이용한 코드로 변환해 주는 바벨 플러그인



바벨 플러그인과 프리셋

바벨은 자바스크립트 파일을 입력으로 받아서 또 다른 자바스크립트 파일을 출력으로 준다. 이렇게 자바스크립트 파일을 변환해 주는 작업은 플러그인(plugin) 단위로 이루어진다. 두 번의 변환이 필요하다면 두 개의 플러그인을 사용한다. 하나의 목적을 위해 여러 개의 플러그인이 필요할 수 있는데, 이러한 플러그인의 집합을 프리셋(preset)이라고 부른다. 예를 들어, 바벨에서는 자바스크립트 코드를 압축하는 플러그인을 모아 놓은 babel-preset-minify 프리셋을 제공한다. @babel/preset-react은 리액트 애플리케이션을 만들 때 필요한 플러그인을 모아 놓은 프리셋이다.

```shell
npx babel —watch src —out-dir . —presets @babel/preset-react
```

npx - 외부 패키지에 포함된 실행 파일을 실행할 때 사용, 외부 패키지의 실행 파일은 ./node_module/.bin/ 밑에 저장된다.

watch - src 폴더의 자바스크립트 파일을 수정할 때마다 자동으로 변환 후 저장

웹팩(webpack)

웹팩(webpakc)은 자바스크립트로 만든 프로그램을 배포하기 좋은 형태로 묵어 주는 툴

웹페이지가 SAP로 가면서 자바스크립트 파일의 관리가 어려워짐(파일 간 의존성, 데이터 오염등)

모듈

ES6부터 모듈 시스템이 언어 차원에서 지원

ESM(ES6)

ESM 문법 익히기

ESM 문법을 익히기 위해 모듈을 내보내고 가져오는 코드를 작성해 보자. 다음 코드는 세 파일의 내용을 보여 준다. file1.js 파일은 코드를 내보내는 쪽이고 file2.js, file3.js 파일은 코드를 사용하는 쪽이다.

코드 1-10 **ESM 예제 코드**

```
// file1.js 파일
export default function func1() {} ❶
export function func2() {}
export const variable1 = 123;
export let variable2 = 'hello';
                                    ❷

// file2.js 파일
import myFunc1, { func2, variable1, variable2 } from './file1.js'; ❸

// file3.js 파일
import { func2 as myFunc2 } from './file1.js'; ❹
```

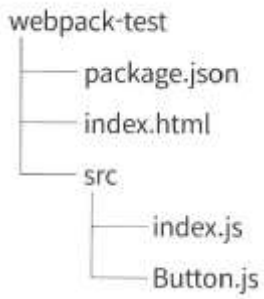
❶, ❷ 코드를 내보낼 때는 export 키워드를 사용한다. ❸ 코드를 사용하는 쪽에서는 import, from 키워드를 사용한다. ❶ default 키워드는 한 파일에서 한 번만 사용할 수 있다. ❸ default 키워드로 내보내진 코드는 괄호 없이 가져올 수 있고, 이름은 원하는 대로 정할 수 있다. ❶번 코드에서 내보낸 func1 함수는 ❸번 코드에서 myFunc1이라는 이름으로 가져왔다. ❸ default 키워드 없이 내보내진 코드는 괄호를 사용해서 가져온다. 가져올 때 이름은 내보낼 때 사용된 이름 그대로 가져와야 한다. ❹ 원한다면 as 키워드를 이용해서 이름을 변경해서 사용할 수 있다.

commonJS(nodejs)

웹팩 사용하기

```shell
npm init-y
```

simple1.html 파일을 복사해서 wehpack-test 폴더 밑에 index.html 파일을 만들고 , index. html 에 있는 simp lel. j 5 문자열을 dist/main.js로 변경하자. 그 다음 react. development.
js, react-dom.development.js 파일을 포함하고 있는 script 태그를 지운다. 이 두 리액트 파일은 모률 시스템을 이용해서 main.js 파일에 포함될 예정이다.
webpack-test 폴더 밑 에 src 폴더를 만들자. src 폴더 밑에 내용이 없는 index.js, Button.js 파일을 만든다. 여기까지 잘 따라 했다면 파일 구조는 다음과 같을 것 이다.




```shell
npm install webpack webpack-cli react react-dom
```

코드 1-11 **index.js**

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

function Container() { ❷
  return React.createElement(
    'div',
    null,
    React.createElement('p', null, '버튼을 클릭해 주세요.'),
    React.createElement(Button, { label: '좋아요' }),
    React.createElement(Button, { label: '싫어요' }),
  );
}
const domContainer = document.querySelector('#react-root');
ReactDOM.render(React.createElement(Container), domContainer);
```

❶

코드 1-12 **Button.js**

```
import React from 'react';

function Button(props) {
  return React.createElement('button', null, props.label);
}
export default Button;
```

❶

```
npx webpack
```

Shell

create-react-app

```
npx create-react-app cra-test npm install -g create-react-app create-react-app cra-test cd cra-test npm start
```

Shell

리액트

컴포넌트란?

리액트 엘리먼트를 반환하는 함수 또는 클래스

상탯값과 속성값으로 관리하는 UI 데이터

UI 데이터 = 상탯값(컴포넌트 내부에서 관되는 데이터) + 속성값(부모 컴포넌트에서 내려주는 데이터)

UI 데이터가 변경되면 리액트가 랜더 함수를 이용해서 화면을 자동으로 갱신해 준다.

state 사용

코드 3-3 컴포넌트의 상탯값을 사용하지 않은 코드

```
class MyComponent extends React.Component {
  color = 'red'; ❶
  onClick = () => {
    this.color = 'blue'; ❷
  };
  render() {
    return (
      <button style={{ backgroundColor: this.color }} onClick={this.onClick}>
        좋아요
      </button>
    );
  }
}
```

코드 3-4 컴포넌트의 상탯값을 사용하는 코드

```
class MyComponent extends React.Component {
  state = { color: 'red' }; ❶
  onClick = () => {
    this.setState({ color: 'blue' }); ❷
  };
  render() {
    return (
      <button style={{ backgroundColor: this.color }} onClick={this.onClick}>
        좋아요
      </button>
    );
  }
}
```

props 사용

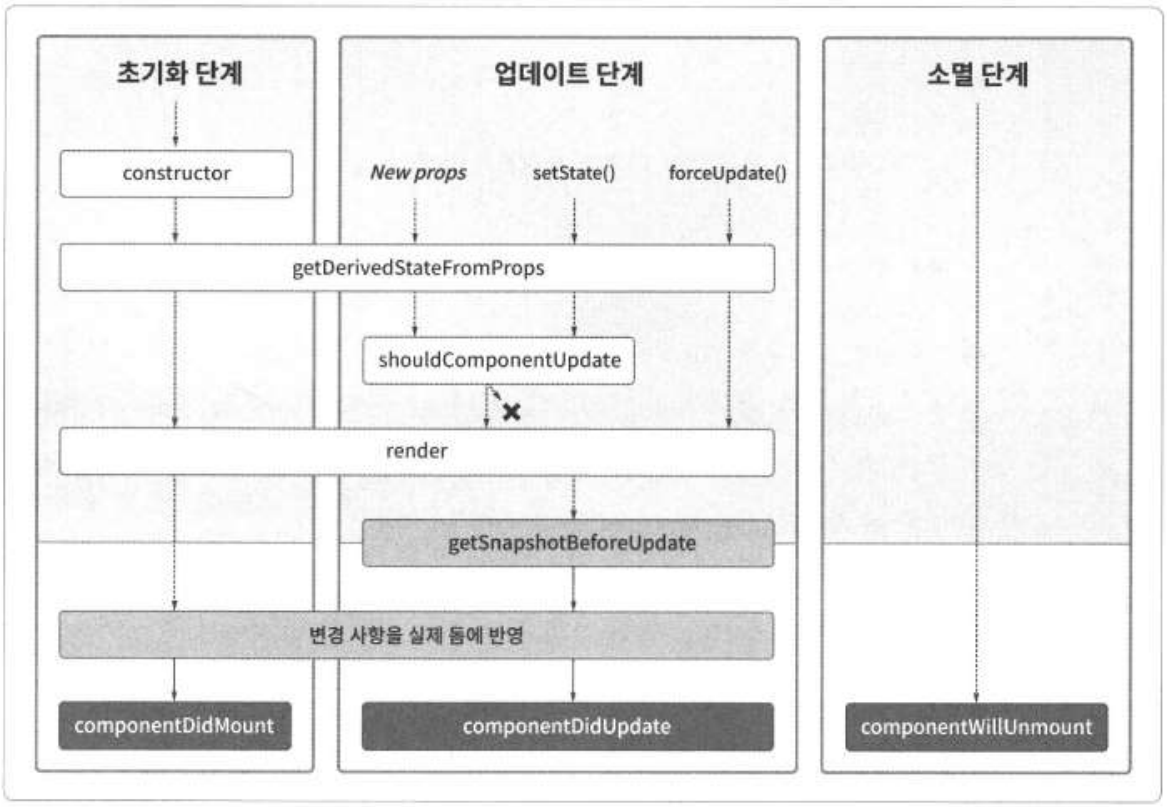
코드 3-5 속성값을 사용한 코드

```
function Title(props) {
  return <p>{props.title}</p>;
}
```

코드 3-6 부모 컴포넌트에서 속성값을 내려 주는 코드

```
class Todo extends React.Component {
  state = { count: 0 };
  onClick = () => {
    const { count } = this.state;
    this.setState({ count: count + 1 }); ❶
  };
  render() {
    const { count } = this.state;
    return (
      <div>
        <Title title={`현재 카운트: ${count}`} /> ❷
        <button onClick={this.onClick}>증가</button>
      </div>
    );
  }
}
```

생명주기



[1. 초기화 단계]

constructor()
static getDerivedStateFromProps()
render( )
componentDidMount()

[2. 업데이트 단계] - 초기화 단계와 소멸 단계 사이에서 반복해서 수행

static getDerivedStateFromProps()
shouldComponentUpdate()
render()
getSnapshotBeforeUpdate()
componentDidUpdate()

[3. 소멸 단계]

componentWillUnmount()

[4. 예외 발생시]

static getDerivedStateFromError()
componentDidCatch()

1.1 constructor(props)

초기 속성값으로부터 상탯값을 만드는 경우

상탯값 직적 할당하는 것은 constructor 메서드에서만 허용 된다.(다른 곳에서는 setState() 사용)

반드시 super() 호출해야 한다

setState 메서드 호출은 컴포넌트가 마운트된 이후에만 유효하다.

코드 3-28 초기 속성값으로부터 상탯값을 만드는 코드

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { ❶
      currentMovie: props.age < 10 ? '뽀로로' : '어벤져스', ❷
    };
  }
}
```

1.2 static getDerivedStateFromProps(props, state)

속성값을 이용해서 새로운 상탯값을 만들때 사용

render 메소드 호출 직전에 호출된다.

정적 메소드이기 때문에 this 객체에 접근할 수 없다. 오로지 속성값과 상탯값을 기반으로 새로운 상탯값을 만든다.

코드 3-33 **getDerivedStateFromProps 메서드에서 이전 속성값 이용하기**

```
class MyComponent extends React.Component {
  state = {
    // ...
    prevSpeed: this.props.speed, ❶
  };
  static getDerivedStateFromProps(props, state) {
    if (props.speed !== state.prevSpeed) { ❷
      // ...
      return {
        // ...
        prevSpeed: props.speed,
      };
    }
    return null; ❸
  }
}
```

이전 속성값과 이후 속성값 모두에 의존적인 상탯값이 필요한 경우에만 유의미

나머지 대부분의 경우 render(), componentDidUpdate()에서 처리할수 있다.

코드 3-40 **상탯값이 전후 속성값에 의존적인 경우**

```
class MyComponent extends React.Component {
  state = {
    // ...
    prevSpeed: this.props.speed,
    isMovingFaster: false,
  };
  static getDerivedStateFromProps(props, state) {
    if (props.speed !== state.prevSpeed) {
      return {
        isMovingFaster: state.prevSpeed < props.speed, ❶
        prevSpeed: props.speed,
      };
    }
    return null;
  }
}
```

1.3 render()

코드 3-41 **render 메서드가 반환할 수 있는 값**

```
return <MyComponent title="안녕하세요" />;
return <p>안녕하세요</p>;
return '안녕하세요';
return 123;
return [<p key="a">안녕하세요</p>, <p key="b">반갑습니다</p>];
return (
  <React.Fragment>
    <p>안녕하세요</p>
    <p>반갑습니다</p>
  </React.Fragment>
);
return null;
return false;
return ReactDOM.createPortal(<p>안녕하세요</p>, domNode);
```

❶ 우리가 작성한 컴포넌트와 HTML에 정의된 거의 모든 태그를 사용할 수 있다. ❷ 문자열과 숫자를 반환할 수 있다. ❸ 배열을 반환할 수 있다. 이때 각 리액트 요소는 key 속성값을 갖고 있어야 한다. ❹ 리액트 프래그먼트(fragment)를 사용하면 내부의 리액트 요소에 key 속성값을 부여하지 않아도 된다. ❺ null 또는 불(bool)을 반환하면 아무것도 렌더링하지 않는다. ❻ 리액트 포털(portal)을 사용하면 컴포넌트의 현재 위치와는 상관없이 특정 돔 요소에 렌더링할 수 있다.

- 랜더 함수 내부에서 setState를 호출하면 안된다.
- 랜더 함수의 반환값을 속성값과 상탯값만으로 결정되어야 한다.
- 부수 효과를 발생시키면 안 된다.(api 호출, 브라우저의 쿠키에 저장등은 다른 생명주기에서 처리)

1.4 componentDidMount()

render()메소드의 반환 값이 실제 돔에 반영된 직후 호출

componentDidMount 메서드는 API 호출을 통해 데이터를 가져올 때 적합하다. setState 메서드가 마운트 이후에만 동작하기 때문이다. constructor 메서드에서 API 호출 후 setState 메서드를 호출하면 데이터가 저장되지 않을 수 있으므로 주의하자. 하지만 constructor 메서 드가 componentDidMount 메서드보다 먼저 호출되기 때문에 API 호출 결과를 더 빨리 받아올 수 있다는 사실에는 변함이 없다 constructor 메서드에서 API 호출을 하고 componentDidMount 메서드에서 set5tate 메서드를 호출할 수는 없을까?

코드 3-45 **constructor 메서드에서 API 요청을 보내는 코드**

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.dataPromise = requestData(); ❶
  }
  componentDidMount() {
    this.dataPromise.then(data => this.setState({ data })); ❷
  }
}
```

2.1 shouldComponentUpdate(nextProps, nextState)

성능 최적화를 위해 존재한다.

false 리턴시 render() 호출 안함

별도로 구현하지 않았다면 항상 참을 반환하여 render() 호출됨(실제 돔이 변경되지 않더라도 가상 돔을 비교하게 된다.)

코드 3-46 **shouldComponentUpdate 메서드의 기본 구조**

```
class MyComponent extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    const { price } = this.state;
    return price !== nextState.price; ❶
  }
}
```

2.2 getSnapshotBeforeUpdate()

렌더링 결과가 실제 돔에 반영되기 직전에 호출

이전 돔 요소의 상태값을 가져오기 좋다.

getSnapshotBeforeUpdate() — (가상돔 반영) —> componentDidUpdate()

```
getSnapshotBeforeUpdate(prevProps, prevState) => snapshot
```

JavaScript

getSnapshotBeforeUpdate 메서드가 반환한 값은 componentDidUpdate 메서드의 세 번째 인자로 들어간다. 따라서 getSnapshotBeforeUpdate 메서드에서 이전 돔의 상뱃값을 반환하면, componentDidUpdate 메서드에서는 돔의 이전 상랫값과 이후 상뱃값을 모두 알기 때문에 돔의 상탯값 변화를 알 수 있다.

코드 3-47 돔 요소의 높잇값이 변경됐는지 검사하는 코드

```
class MyComponent extends React.Component {
  state = {
    items: [],
  };
  divRef = React.createRef(); ❶
  getSnapshotBeforeUpdate(prevProps, prevState) {
    const { items } = this.state;
    if (prevState.items.length < items.length) { ❷
      const rect = this.divRef.current.getBoundingClientRect();
      return rect.height;
    }
    return null;
  }
  componentDidUpdate(prevProps, prevState, snapshot) { ❸
    if (snapshot !== null) {
      const rect = this.divRef.current.getBoundingClientRect();
      if (rect.height !== snapshot) { ❹
        alert('새로운 줄이 추가되었습니다.');
      }
    }
  }
  onClick = () => {
    const { items } = this.state;
    this.setState({ items: [...items, '아이템'] });
  };
  render() {
    const { items } = this.state;
    return (
      <React.Fragment>
        <button onClick={this.onClick}>추가하기</button>
        <div ref={this.divRef} style={{ width: '100%' }}>
          {items.map(item => <span style={{ height: 50 }}>{item}</span>)} ❺
        </div>
      </React.Fragment>
    );
  }
}
```

2.3 componentDidUpdate(prevProps, prevState, snapshot)

가상 돔이 실제 돔에 반영된 후 호출된다. 새로 반영된 돔의 상탯값을 가장 빠르게 가져올수 있는 생명 주기 메서드

속성값이나 상탯값이 변경된 경우 API를 호출하는 용도로 사용되기도 한다. 이전, 이후의 상탯값과 속성값을 모두 알 수 있기 때문에 이와 같은 코드를 자주 작성하게 된다.

코드 3-49 **componentDidUpdate 메서드에서 API를 호출하는 코드**

```
class UserInfo extends React.Component {
  componentDidUpdate(prevProps) {
    const { user } = this.props;
    if (prevProps.user.id !== user.id) { ❶
      requestFriends(user).then(friends => this.setState({ friends }));
    }
  }
}
```

3.1 componentWi llUnmount()

코드 3-51 **componentWillUnmount 메서드에서 이벤트 처리 메서드 해제하기**

```
class MyComponent extends React.Component {
  componentDidMount() {
    const domNode = document.getElementById('someNode');
    domNode.addEventListener('change', this.onChange);
    domNode.addEventListener('dragstart', this.onDragStart);
  }
  componentWillUnmount() {
    const domNode = document.getElementById('someNode');
    domNode.removeEventListener('change', this.onChange);
    domNode.removeEventListener('dragstart', this.onDragStart);
```

4.1 getDerivedStateFromError(error), componentDidCatch(error, info)

생명 주기 메서드에서 예외가 발생하면 getDerivedStateFromError, componentDidCatch 메서드를 구현한 가장 가까운 부모 컴포넌트를 찾는다.

getDerivedStateFromError - 에러 정보를 상탯값에 저장해서 화면에 나타내는 용도

componentDidCatch -에러 정보를 서버로 전송하는 용도(추후 setState 호출 막힐 것으로 보임)

componentDidCatch에서 에러 정보를 서버로 전송하는 이유

이는 리액트 버전 16.9부터 도입될 것으로 보이는 비동기 렌더링 때문이다.

리액트에서 데이터 변경에 의한 화면 업데이트는 렌더 단계(render phase)와 커밋 단계(commit phase)를 거친다. 렌더 단계에서는 실제 돔에 반영할 변경 사항을 파악하고, 커밋 단계에서는 파악된 변경 사항을 실제 돔에 반영한다. 비동기 렌더링에서는 렌더 단계에서 실행을 멈췄다가 나중에 다시 실행하는 과정에서 같은 생명 주기 메서드를 여러 번 호출할 수 있다.

다음은 커밋 단계에서 호출되는 생명 주기 메서드다.

- getSnapshotBeforeUpdate
- componentDidMount
- componentDidUpdate
- componentDidCatch

이 메서드를 제외한 나머지 생명 주기 메서드는 렌더 단계에서 호출된다.

getDerivedStateFromError 메서드는 렌더 단계에서 호출되고, componentDidCatch 메서드는 커밋 단계에서 호출된다. 만약 getDerivedStateFromError 메서드에서 에러 정보를 서버로 전송한다면 같은 에러 정보가 여러 번 전송될 수 있다. 커밋 단계의 생명 주기 메서드는 비동기 렌더링에서도 한 번만 호출되기 때문에 에러 정보는 componentDidCatch 메서드에서 전송하는 게 좋다.

ErrorBoundary 컴포넌트(자식 컴포넌트 예외만 처리)

코드 3-52 **ErrorBoundary 컴포넌트**

```
class ErrorBoundary extends React.Component {
  state = { error: null };
  static getDerivedStateFromError(error) {
    return { error }; ❶
  }
  componentDidCatch(error, info) {
    sendErrorToServer(error, info); ❷
  }
  render() {
    const { error } = this.state;
    if (error) { ❸
      return <div>{error.toString()}</div>;
    }


    return this.props.children; ❹
  }
}
```

코드 3-53 **ErrorBoundary 컴포넌트를 사용한 코드**

```
class Counter extends React.Component {
  state = { count: 0 };
  onClick = () => {
    const { count } = this.state;
    this.setState({ count: count + 1 });
  };
  render() {
    const { count } = this.state;
    if (count >= 3) { ❶
      throw new Error('에러 발생!!!');
    }
    return <div onClick={this.onClick}>{`클릭하세요(${count})`}</div>;
  }
}

function App() {
  return (
    <ErrorBoundary>
      <Counter />
    </ErrorBoundary>
  );
}
```

❷

이벤트 처리 메서드는 생명주기 메서드가 아니기 때문에 ErrorBoundary 컴포넌트로 처리할 수 없다.

코드 3-55 **이벤트 처리 메서드에서 예외를 처리하는 코드**

```
onClick = () => {
  try { ❶
    this.setState({ name: 'mike' });
    throw new Error('some error');
    this.setState({ age: 23 });
  } catch (e) {
    // ... ❷
```

바벨과 웹팩

바벨(babel)

> 바벨이란?
>
> 바벨은 입력과 출력이 모두 자바스크립트 코드인 컴파일러다. 이는 보통의 컴파일러가 고수준의 언어를 저수준의 언어로 변환하는 것과 비교된다.
>
> 초기의 바벨은 ES6 코드를 ES5 코드로 변환해 주는 컴파일러였다. 현재는 바벨을 이용해서 리액트의 JSX 문법, 타입스크립트와 같은 정적 타입 언어, 코드 압축, 제안(proposal) 단계에 있는 문법 등을 사용할 수 있다.

바벨은 다음과 같이 실행될 수 있다.

1. @babe1/cli로 실행하기
2. 웹팩에서 babel-loader로 실행하기
3. @babe1/core를 직접 실행하기
4. @babe1/register로 실행 하기(생략)

1. @babe1/cli로 실행하기

실습 프르젝트

```shell
mkdir test-babel-how cd test babel-how npm init -y
```

Shell

패키지 설치

```
npm install @babel/core @babel/cli @babel/plugin-transform-arrow-functions
@babel/plugin-transfo nm-template-literals @babel/preset-react
```

Shell

샘플 코드 - src/code.js

```
const element = <div>babel test</div>; const text `element type is
${element.type}` const add = (a, b) => a + b;
```

JavaScript

@babel/cli로 실행

```
npx babel src/code.js --presets=@babel/preset-react --plugins=@babel/
plugin-transform-template-literals ,@babel/plugin-transform-arrow-functions
```

Shell

babel.config.js

```
const presets = ['@babel/preset-react']; const plugins = [ '@babel/plugin-
transform-template-literals', '@babel/plugin-transform-arrow-functions', ];
module.exports = { presets, plugins };
```

JavaScript

경로 지정

```
npx babel src/code.js npx babel src/code.js --out-file dist.js npx babel
src --out-dir dist
```

Shell

2. 웹팩에서 babel-loader로 실행하기

웹팩 설치

```shell
npm install webpack webpack-cli babel-loader
```

babel-loader를 설정하는 webpack.config.js

```javascript
const path = require('path'); module.exports = { entry : './src/code.js',
output: { path: path.resolve( __ dirname, 'dist'), filename:
'code.bundle.js' }, module: { rules: [{ test: /\.js$/, use:'babel-
loader'}], }, optization: {minimizer: []}, //압축기능 잠시 해제 };
```

babel-loader는 바벨의 설정 파일을 이요하므로 이전에 만들어 놓은 babel.confing.js 파일 내용이 설정 값으로 사용된다.

babel-loader는 바벨의 설정 파일을 이요하므로 이전에 만들어 놓은 babel.confing.js 파일 내용이 설정 값으로 사용된다.

3. @babe1/core를 직접 실행하기

먼저 프로젝트 루트에 runBabel.js 파일을 만들고, 다음 코드를 입력한다.

코드 7-6 **@babel/core로 바벨을 직접 실행하기**

```
const babel = require('@babel/core'); ❶
const fs = require('fs');

const filename = './src/code.js';
const source = fs.readFileSync(filename, 'utf8'); ❷
const presets = ['@babel/preset-react'];
const plugins = [
  '@babel/plugin-transform-template-literals',    ❸
  '@babel/plugin-transform-arrow-functions',
];
const { code } = babel.transformSync(source, { ❹
  filename,
  presets,
  plugins,
  configFile: false, ❺
});
console.log(code); ❻
```

컴파일 효율 높이기

코드 7-7 **같은 프리셋을 사용하는 두 가지 설정**

```
// 설정 1
const presets = ['@babel/preset-react'];
const plugins = ['@babel/plugin-transform-template-literals'];

// 설정 2
const presets = ['@babel/preset-react'];
const plugins = ['@babel/plugin-transform-arrow-functions'];
```

1. 파싱 (parse) 단계 : 입 력 된 코드로부터 AST(abstract syntax tree)를 생 성한다 .
2. 변환(transfonn) 단계 :AST를 원하는 형태로 변환한다.
3. 생성 (generate) 단계: AST를 코드로 출력한다.

코드 7-8 **AST를 활용해서 효율적으로 바벨을 실행하는 코드**

```
const babel = require('@babel/core');
const fs = require('fs');

const filename = './src/code.js';
const source = fs.readFileSync(filename, 'utf8');
const presets = ['@babel/preset-react'];

const { ast } = babel.transformSync(source, {
  filename,
  ast: true,
  code: false,
  presets,
  configFile: false,
});

const { code: code1 } = babel.transformFromAstSync(ast, source, {
  filename,
  plugins: ['@babel/plugin-transform-template-literals'],
  configFile: false,
});
const { code: code2 } = babel.transformFromAstSync(ast, source, {
  filename,
  plugins: ['@babel/plugin-transform-arrow-functions'],
  configFile: false,
});
console.log('code1:\n', code1);
console.log('code2:\n', code2);
```

❶ ❷ ❸

웹팩(webpack)

웹팩은 모듈(module) 번들러(bundler)

1. webpack-cli로 실행

webpack-di를 이용하면 CU(corrun때d line intelface) 에서 웹팩을 실행할 수 있다

```shell
mkdir webpack-init cd webpack-init npm init -y npm install webpack webpack-cli
```

> webpack-cli를 이용하지 않고 코드를 통해 직접 실행할 수도 있다. create-react-app 이나 next.Js 같은프레임워크에서는 세밀하게 웹팩을 다뤄야 하므로 webpack-cli를 이용하지 않고 코드에서 직접 실행한다.

util.js

```javascript
export function sayHello(name) { console.log('hello', name) ; }
```

index.js

```javascript
import { sayHello } from './util'; function myFunc() { sayHello('mike'); console.log('myFunc'); } myFunc() ;
```

웹팩 실행

```shell
npx webpack
```

설정 파일 이용하기

설정 파일을 이용하는 방법을 알아보자. 프로젝트 루트에 webpack.config.js 파일을 만들고, 다음 코드를 입력한다.

코드 7-49 **webpack.config.js 파일**

```
const path = require('path');

module.exports = {
  entry: './src/index.js', ❶
  output: {
    filename: 'main.js',
    path: path.resolve(__dirname, 'dist'),
  },
  mode: 'production', ❸
  optimization: { minimizer: [] }, ❹
};
```

❷

2. 로더 사용하기

로더(loader)는 모듈을 입력으로 받아서 원하는 형태로 변환한 후 새로운 모듈을 출력해 주는 함수다.

자바스크립트 파일뿐만 아니라 이미지 파일, CSS 파일, CSV 파일 등 모든 파일은 모듈이 될 수 있다.

```
mkdir webpack-loader cd webpack-loader npm init -y npm install webpack
webpack-cli
```

Shell

자바스크립트 파일 처리하기

```
npm install babel-loader @babel/core @babel/preset-react react react-dom
```

Shell

JSX 문법을 사용한 자바스크립트 코드

```
import React from 'react'’ ; import ReactDOM from 'react-dom'; function App() { return ( <div className="container"> <h3 className="title">webpack example</h3> </div> ); } ReactDOM.render(<App />, document.getElementByld('root'));
```

JavaScript

@babel/preset-react을 사용하도록 설정하기

```
const presets = ['@babel/preset-react']; module.exports = { presets };
```

JavaScript

코드 7-53 babel-loader 설정하기

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'main.js',
    path: path.resolve(__dirname, 'dist'),
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: 'babel-loader',
      },
    ],
  },
  mode: 'production',
};
```

❶

index.html

```
<html> <body> <div id="root" /> <script src="./main.js"></script> </body> </html>
```

HTML

3. 플러그인 사용하기

로더는 특정 모듈에 대한 처리만 담당하지만 플러그인은 웹팩이 실행되는 전체 과정에 개입할 수 있다.

```
mkdir webpack-plugin cd webpack-plugin npm init -y npm install webpack webpack-cli
```

Shell

간단한 리액트 코드

```
import React from 'react' ; import ReactDom from 'react-dom'; function App() { return ( <div> <h3>안녕하세요,  웹팩 플러그인 예제입니다. </h3> <p>html-webpack-plug i n 플러그인을 사용합니다. </p> </div> ) } ReactDom.render(<App />, document.getElementByld('root'));
```

JavaScript

컴파일하기 위한 패키지 설지

```
npm install @babel/core @babel/preset-react babel-loader react react-dom
```

Shell

코드 7-64 **babel-loader를 사용하도록 설정하기**

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: '[name].[chunkhash].js', ❶
    path: path.resolve(__dirname, 'dist'),
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-react'],
          },
        },
      },
    ],
  },
  mode: 'production',
};
```

❶ chunkhash를 사용하면 파일의 내용이 수정될 때마다 파일 이름이 변경되도록 할 수 있다. ❷ 자바스크립트 모듈을 처리하도록 babel-loader를 설정한다.

babel.config.js 파일로 바벨 설정할 수도 있지만 여기처럼 babel-loader에서 직접 바벨 설정을 할 수도 있다.

html-webpack-plugin

코드 7-65 **html-webpack-plugin을 사용하도록 설정하기**

```
// ...
const CleanWebpackPlugin = require('clean-webpack-plugin');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  // ...
  plugins: [
    new CleanWebpackPlugin(), ❶
    new HtmlWebpackPlugin({
      template: './template/index.html',
    }), ❷
  ],
  //...
```

코드 7-66 **template/index.html**

```
<html>
<head>
  <title>웹팩 플러그인 예제</title> ❶
</head>
<body>
  <div id="root" /> ❷
</body>
</html>
```

DefinePlugin

코드 7-68 **DefinePlugin으로 대체할 문자열**

```
// ...
    <div>
      // ...
      <p>{`앱 버전은 ${APP_VERSION}입니다.`}</p>
      <p>{`10 * 10 = ${TEN * TEN}`}</p>
    </div>
// ...
```

❶

❶ APP_VERSION, TEN 문자열을 우리가 원하는 문자열로 대체한다.

webpack.config.js 파일에 다음 코드를 추가해 보자.

코드 7-69 **DefinePlugin을 사용하도록 설정하기**

```
// ...
const webpack = require('webpack');

module.exports = {
  // ...
  plugins: [
    // ...
    new webpack.DefinePlugin({ ❶
      APP_VERSION: '"1.2.3"', // 또는 JSON.stringify('1.2.3'), ❷
      TEN: '10', ❸
    }),
    // ...
```

❶ DefinePlugin은 웹팩 모듈에 포함되어 있다. ❷ APP_VERSION 문자열을 '1.2.3'으로 대체한다. ❸ TEN 문자열을 10으로 대체한다.