1. 컴포넌트 스타일링(1)

HTML에서의 스타일 적용 방법

- 인라인 스타일
  - Hello
  - HTML에서는 동일한 스타일의 요소가 여러개 만들어질 경우 중복, 유지관리가 힘듬.
- style 태그
  - 페이지 단위 <style> 태그를 작성하여 참조하는 방법
- 외부 CSS 파일 참조
  - .css 파일을 작성하고 여러 페이지에서 <link> 태그로 참조하는 방법

1.1 React 스타일링

React 에서의 스타일

- 전역 css 참조
  - 설치 yarn add bootstrap (4버전은 yarn add bootstrap@4.x.x)
  - 사용 : import 'bootstrap/dist/css/bootstrap.css'
  - 3장에서 이미 다루어 보았음
  - import한 css 파일의 클래스는 모든 컴포넌트에서 사용할 수 있음.
- 인라인 스타일링
  - JS 객체를 style Attribute에 지정하여 CSS 생성
  - HTML에서는 권장하지 않지만 React(특히 React-native)에서는 사용하는 경우가 있음
- 컴포넌트 단위로 CSS 모듈 사용
  - 사용 : import appstyle from './appstyle.module.css';
  - CSS 클래스 기반
  - 클래스 이름이 난독화되어 이름 충돌을 피함

1.2 인라인 스타일링(1)

React 에서의 스타일(이어서)

- 인라인 스타일
  - React의 컴포넌트 단위로 JS 객체를 이용해 스타일을 적용함.
  - HTML의 인라인 스타일은 권장하지 않지만 React는 자주 사용함.
  - 컴포넌트 단위로 마크업 + 로직 + 스타일을 묶어 하나의 컴포넌트로 캡슐화!!
  - 인라인 스타일 정보는 JS 객체로 지정함
  - kebob casing --> camelCasing

```
let styles = {
    color:"yellow", backgroundColor:"purple"
}
```

  - style 특성에 객체 부여

```
<div style={styles}>Hello</div>
```

1.2 인라인 스타일링(2)

인라인 스타일 적용

- 3장의 helloapp 프로젝트에 적용해보자.
- src/styles.js 파일 추가

```
const styles = {
    listItemStyle : {
       fontStyle:"italic", textDecoration:"underline"
    },
    dashStyle: {
       backgroundColor: "#fff",
       borderTop: "2px dashed gray"
    }
}
export default styles;
```

1.2 인라인 스타일링(3)

- src/App.js 변경

```
……
import styles from './styles';
……
class App extends Component {
    ……
    render() {
        return (
            <div className="container">
                <h1>Hello {this.state.msg}!!</h1>
                <hr style={styles.dashStyle} />
                { this.createString(4,5) }
                <CountryList countries={this.state.list} />
            </div>
        );
    }
}
……
```

1.2 인라인 스타일링(4)

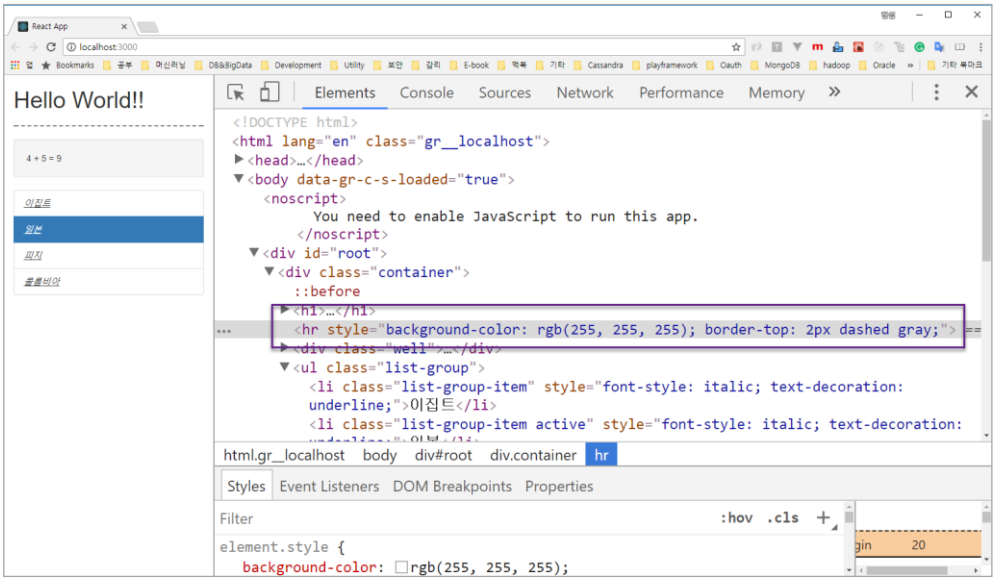
src/CountryItem.js 변경

```
import React, { Component } from 'react';
import styles from './styles';

class CountryItem extends Component {
    render() {
        return (
            <li style={styles.listItemStyle}
              className={this.props.visited ? 'list-group-item active' : 'list-group-item' }>
                  {this.props.country}
            </li>
        );
    }
}

export default CountryItem;
```

1.2 인라인 스타일링(5)

실행후 결과 확인

- yarn start



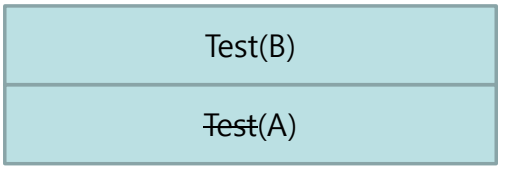
■ 실행 결과를 살펴보면 styles.js 파일에 객체로 지정한 스타일 정보가 hr 요소의 style 특성에 부여된 것을 확인할 수 있다.

1.3 CSS 모듈(1)

전역 CSS 사용의 단점

- 여러 컴포넌트에서 import한 CSS에 동일한 클래스명의 스타일이 존재하면 충돌이 발생함.
- CSS는 먼저 impor한 것이 Base에 깔리고 나중에 import한 CSS가 위에 포개져서 기존 클래스를 그림자 아래로 숨기는 개념

```
//두 CSS에 모두 Test 클래스가 존재한다면?
import A from './A.css';
.....
import B from './B.css';
.....
```



CSS 모듈

- 클래스 명을 해시를 이용해 충돌나지 않는 이름으로 변경
- .module.css로 끝나는 이름을 사용해야 함.

1.3 CSS 모듈(2)

CSS 모듈 적용

- css 모듈은 클래스 기반으로 작성해야 함.
- App.module.css

```
.test { color:blue; background-color: bisque; }
```

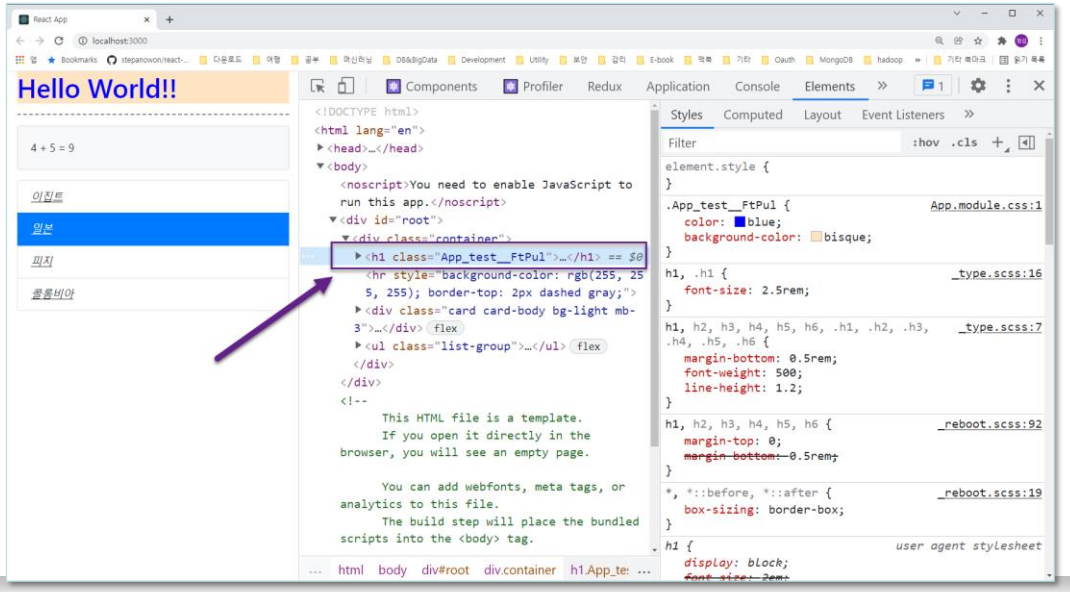
- App.js

```
……(생략)
import AppCssModule from './App.module.css';

class App extends Component {
    ……(생략)
    render() {
        return (
            <div className="container">
                <h1 className={AppCssModule.test}>Hello {this.state.msg}!!</h1>
                ……(생략)
            </div>
        );
    }
}
……
```

1.3 CSS 모듈(3)

CSS 모듈 적용 결과

- h1 태그의 class 명 확인



10

1.4 styled-components(1)

기존 CSS의 문제점

- CSS는 배우기는 쉽지만 스타일을 정교하게 조작하기가 쉽지 않음.
- 프로그래밍 언어적인 특성이 부족함.
  - 변수, 반복문, 함수 등의 표현이 쉽지 않음.

styled-component

- ES6의 Tagged Template Literal 문법을 사용하여 컴포넌트에 동적인 CSS 코드를 작성할 수 있도록 하는 라이브러리
- React와 React-Native에 모두 호환됨.
- 주요 제공 기능
  - CSS 스타일의 문법 사용
  - 전달된 속성에 따라 스타일의 동적 적용
  - 기존 스타일을 확장할 수 있는 Extending Style 제공

설치

- yarn add styled-components

11

1.4 styled-components(2)

전달된 속성에 따라 동적 스타일 적용하기

- src/Footer.js 작성

```
import React, { Component } from 'react';
import styled from 'styled-components'

class Footer extends Component {
    render() {
        const FooterBox = styled.div`
            position: absolute;
            right: 0; bottom: 0;  left: 0;
            padding: 1rem;
            background-color: ${
                (props)=> props.theme === 'basic' ? 'skyblue' : 'yellow'
            };
            text-align: center;
        `;
        return (
            <FooterBox theme={this.props.theme}>React styled-component Test!!</FooterBox>
        );
    }
}
export default Footer;
```

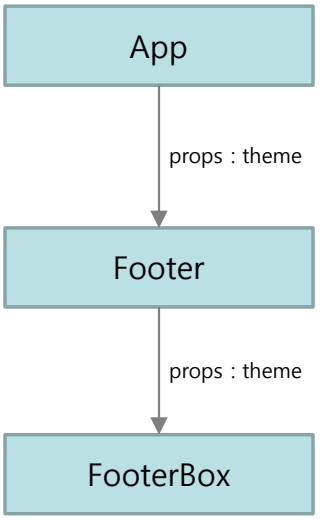
12

■ JSX 구문의 theme를 basic 또는 다른 문자열로 변경했을 때의 실행 결과를 비교한다.

1.4 styled-components(3)

Footer를 사용하도록 App.js 변경

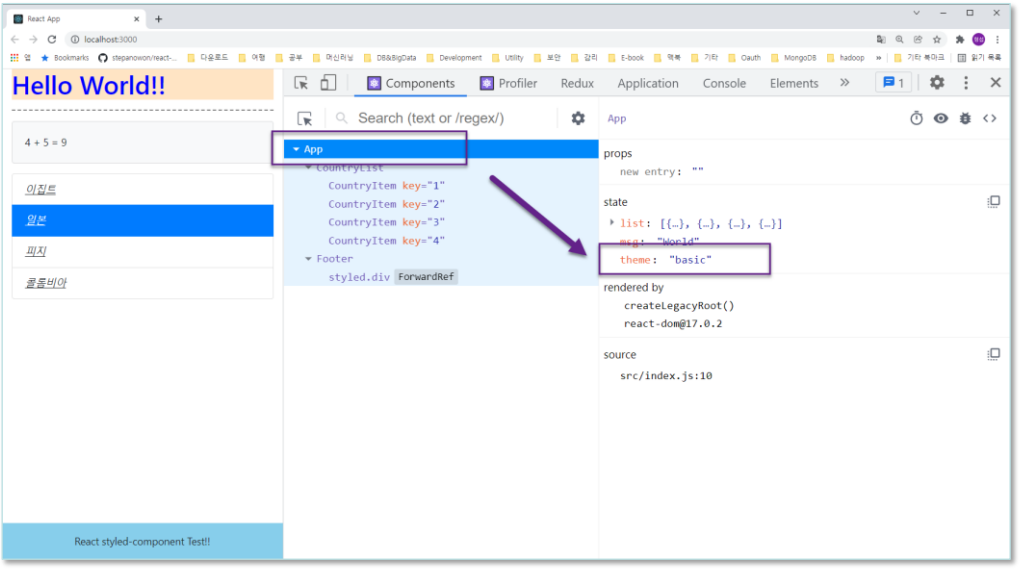
```
……
import Footer from './Footer';

class App extends Component {
    constructor(props) {
        super(props);
        this.state = {
            theme : "basic",
            ……
        }
    }
    ……
    render() {
        return (
            <div className="container">
                ……
                <Footer theme={this.state.theme} />
            </div>
        );
    }
}
export default App;
```



1.4 styled-components(4)

styled-components 적용 결과 확인

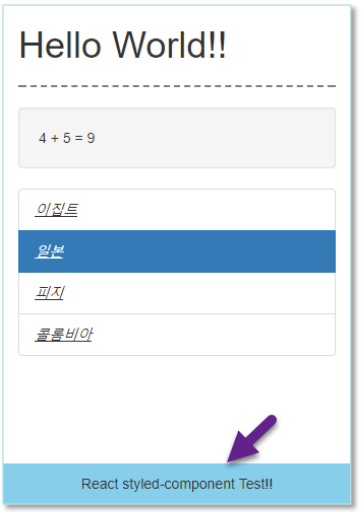
- App 컴포넌트의 theme 상태를 basic 또는 다른 값으로 변경해 봄



14

1.4 styled-components(5)

- 실행 결과
  - theme를 basic으로 설정했을 때



  - theme를 다른 문자열로 설정했을 때

15

2. 속성 심화(1)

속성의 유효성 검증

- 컴포넌트 기반으로 개발할 때 컴포넌트의 속성은 다음을 쉽게 식별할 수 있어야 함.
  - 컴포넌트에서 사용가능한 속성이 무엇인지...
  - 필수 속성은 무엇인지...
  - 속성에 전달할 수 있는 값의 타입은 무엇인지...
- 이를 위해 속성의 유효성 검사 기능이 필요함.

유효성 검증 방법

- PropTypes
- Flow
- Typescript

16

2. 속성 심화(2)

PropTypes를 이용한 유효성 검증

- ES6 클래스 컴포넌트, 함수형 컴포넌트의 static 멤버로 검증 정보를 부여

간단한 예제

- 초기 설정
  - npx create-react-app proptypes-test
  - cd proptypes-test
  - src 디렉터리의 다음 파일 삭제 : App.js, App.test.js, App.css
- src/Calc.js 새롭게 작성
  - 간단한 연산 기능을 가진 컴포넌트 작성
  - 다음 페이지

17

2. 속성 심화(3)

- src/Calc.js : 덧셈, 곱셈만 지원하도록 작성

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

class Calc extends Component {
    render() {
        let result = 0;
        switch(this.props.oper) {
            case "+" :
                result = parseFloat(this.props.x) + parseFloat(this.props.y)                break;
            case "*" :
                result = parseFloat(this.props.x) * parseFloat(this.props.y);                break;
            default :
                result = 0;
        }

        return (
            <div>
                <h3>연산 방식 : { this.props.oper }</h3><hr />
                <div>
                    {this.props.x} {this.props.oper} {this.props.y} = {result}
                </div>
            </div>
        );
    }
}

export default Calc;
```


18

2. 속성 심화(4)

- src/index.js 변경

```
……
import Calc from './Calc';
……

let values = { x:4, y:5, oper:"+" };
ReactDOM.render(
  <React.StrictMode><Calc {...values} /></React.StrictMode>,
  document.getElementById('root')
);
serviceWorker.unregister();
```

- 코드는 정상적으로 실행되지만 속성을 전달하지 않거나 잘못된 값을 전달한다면? 잘못된 결과!! 경고도 없음.

```
let values = { x:"aaa", y:"bbb", oper:"/" };
ReactDOM.render(
  <React.StrictMode><Calc {...values} /></React.StrictMode>,
  document.getElementById('root')
);
```

연산 방식 : /

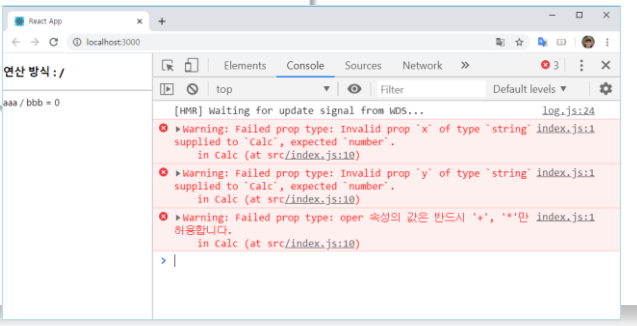
aaa / bbb= 0

2. 속성 심화(5)

유효성 검사 기능 추가

- src/Calc.js 변경

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';

class Calc extends Component { ..... }

Calc.propTypes = {
    x : PropTypes.number.isRequired,
    y : PropTypes.number.isRequired,
    oper : (props, propName, componentName) => {
        if (props[propName] !== "+" && props[propName] !== "*") {
            return new Error(propName + " 속성의 값은 반드시 '+', '*'만 허용합니다.");
        }
    }
}
export default Calc;
```



2. 속성 심화(6)

지정 가능한 유효성 검증 타입

- 단순 타입
  - PropTypes.array
  - PropTypes.bool
  - PropTypes.func
  - PropTypes.number
  - PropTypes.object
  - PropTypes.string
  - PropTypes.node
  - PropTypes.element
- 복잡한 객체, 배열 속성
  - PropTypes.instanceOf(Customer)
  - PropTypes.oneOf(['+', '*', '*', '/'])
  - PropTypes.oneOfType([PropTypes.number, PropTypes.string])
  - PropTypes.arrayOf(PropTypes.object)

21

2. 속성 심화(7)

지정 가능한 유효성 검증 타입(이어서)

- 복잡한 객체 속성

```
PropTypes.shape({
    name: PropTypes.string.isRequired,
    age: PropTypes.number
})
```

- 함수를 이용한 커스텀 유효성 검증
  - props : 속성
  - propName : 속성명
  - componentName : 컴포넌트명

```
(props, propName, componentName) => {
        if (props[propName] !== "+" && props[propName] !== "*") {
            return new Error(`${propName} 속성의 값은 반드시 '+', '*'만
허용합니다.(${componentName} 컴포넌트)`);
        }
}
```


22

2. 속성 심화(8)

속성의 기본값 지정

- ES6 클래스의 static 멤버로 defaultProps 객체 지정
- src/Calc.js 변경

```
.....
class Calc extends Component { ..... }

Calc.propTypes = {
    x : PropTypes.number,
    y : PropTypes.number,
    oper : (props, propName, componentName) => { ..... }
}

Calc.defaultProps = {
    x : 100,
    y : 200,
    oper : "+"
}

export default Calc;
```

23

■ PropTypes의 isRequired는 제거해야 함.

■ defaultProps로 기본값을 부여

■ src/index.js에서 속성 부여하던 것을 삭제함.

```
ReactDOM.render(
  <React.StrictMode>
    <Calc />
  </React.StrictMode>,
  document.getElementById('root')
);
```

3. 이벤트 기초(1)

React 이벤트

- HTML DOM 이벤트를 추상화하여 여러 브라우저에서 동일한 Attribute를 이용할 수 있도록 이벤트를 정규하였음
- 성능을 위해 이벤트를 가상 DOM Root에 연결하고 이벤트를 위임(delegation) 처리함.
  - 이벤트가 발생하면 React가 DOM Root에서 적절한 컴포넌트 요소로 바인딩함.
- camel Casing 규칙을 준수해야 함.
  - HTML DOM Event가 아니라 React Event임을 생각해야 함.
  - 예) onclick --> onClick
  - <button onClick={ ()=> alert('hello') }> OK </button>

3. 이벤트 기초(2)

이벤트 핸들러 메서드 사용하기

- this에 주의!!
  - 화살표 함수일 때는 바깥쪽 스코프의 this가 함수 내부로 전달되므로 컴포넌트 객체가 this임.
  - 일반적인 함수일 때는 바깥쪽 스코프의 this가 함수 안쪽으로 전달되지 않음. 따라서 this가 컴포넌트 객체가 되게 하려면 bind 메서드를 이용해 this를 강제 지정해야 함.

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

class App extends Component {
  helloClick() {
     alert('Hello ' + this.props.name);
  }
  render() {
    return (
      <button onClick={ this.helloClick.bind(this) }>OK</button>
    )
  }
}

export default Hello
```

25

■ 위의 예제와 같이 렌더링할 때 bind(this)하는 것보다는 생성자를 이용해 미리 bind(this)한 후에 렌더링하거나 화살표 함수를 사용할 것을 권장함.

■ 화살표 함수일 때는 아래쪽과 같이...

```
helloClick = () => {
    alert('Hello ' + this.props.name);
}


render() {
  return (
     <button onClick={ this.helloClick }>OK</button>
  )
}
```

3. 이벤트 기초(3)

이벤트와 상태 변경

- 이미 3장에서 다루었지만 한번 더 정리
  - 상태의 변경은 상태를 보유한 컴포넌트 객체에서만 가능함.
  - 상태의 변경은 반드시 setState() 메서드를 이용해서만 가능함. this.state 를 직접 변경하는 것은 허용하지 않음.

카운터 예제로 이벤트와 상태 변경 처리 테스트

- 프로젝트 초기화
  - npx create-react-app event-basic
  - cd event-basic
  - src 디렉터리의 App.js, App.css, App.test.js 파일 삭제

3. 이벤트 기초(4)

- src/App.js 작성

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

class App extends Component {
    constructor(props) {
        super(props);
        this.state = { num:0 };
    }
    add = ()=> { this.setState({ num: this.state.num+1 });  }
    subtract = ()=> { this.setState({ num : this.state.num-1 }); }
    render() {
        return (
            <div className="container">
                <button onClick={this.add}> + </button>
                <button onClick={this.subtract}> - </button>
                <hr />
                <input type="text" value={this.state.num} />
            </div>
        );
    }
}

export default App;
```

27

■ 화살표 함수인 subtract는 그대로 사용

■ 일반 함수는 add는 bind(this)를 이용해 컴포넌트 객체를 this로 강제로 지정

■ 상태를 변경하기 위해 컴포넌트 객체의 setState 메서드 이용

- this.state.num++ 과 같이 코드를 작성하면 UI에 반영되지 않고 경고가 나타남.

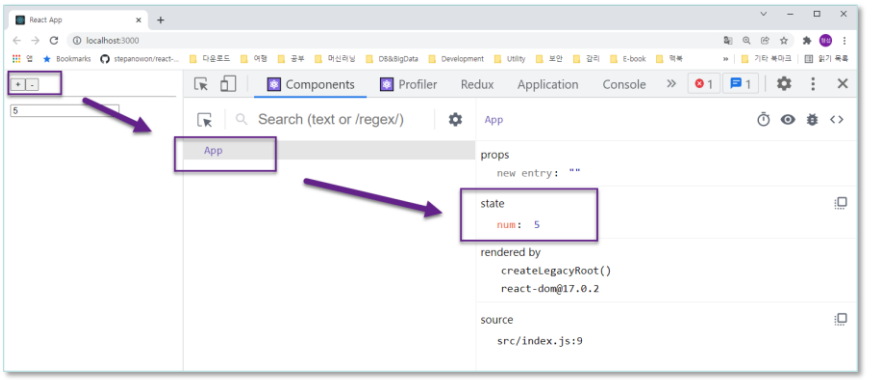
3. 이벤트 기초(5)

- src/index.css에 스타일 추가

```
……
.container { padding:10px; }
```

실행 결과

- yarn start



28

■ 이 코드를 실행해보면 Console에 오류가 발생함을 알 수 있다. 오류의 원인은 무엇일까?

- 다음 절에서 오류의 원인을 알아보자.

4. 제어컴포넌트와 비제어 컴포넌트(1)

제어 컴포넌트와 비제어 컴포넌트

- 제어 컴포넌트(Controlled Component)
  - 입력필드의 값이 state나 props에 의해 제어되는 컴포넌트
  - state가 바뀌지 않는 한 입력값을 변경할 수 없다. 변경하려면 상태를 바꾸도록 이벤트 처리가 요구됨.
- 비제어 컴포넌트(Uncontrolled Component)
  - 입력필드의 값이 state나 props에 의해 제어되지 않는 컴포넌트
  - 사용자가 쉽게 변경할 수 있지만 입력값을 알아내려면 실제 HTML DOM에 접근해야 하는 단점이 있다.
- 둘 중 제어 컴포넌트를 더 권장함.
  - 비제어 컴포넌트는 실제 DOM을 접근해야 하므로 고비용임.

4. 제어컴포넌트와 비제어 컴포넌트(2)

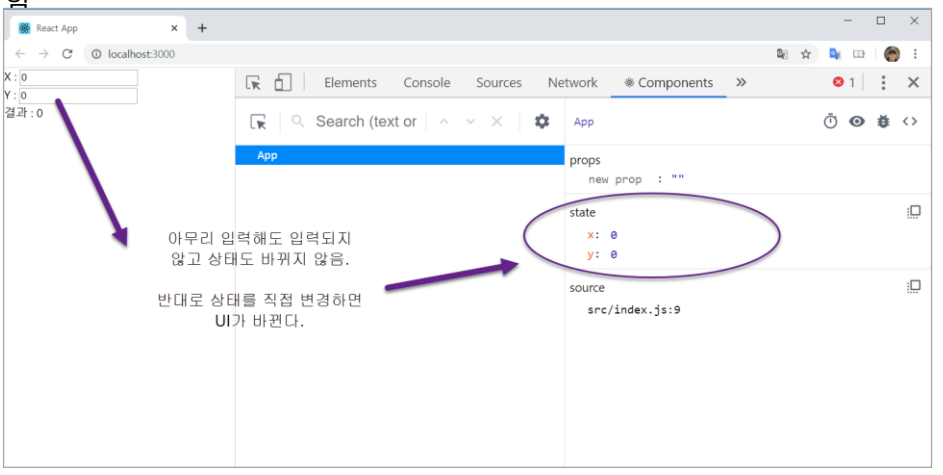
제어 컴포넌트 기능 확인

- 프로젝트 초기화
  - npx create-react-app controlled-component
  - src/App.js, App.test.js, App.css 파일 삭제
- src/App.js 새롭게 작성

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

class App extends Component {
  constructor() {
    super()
    this.state = { x:0, y:0 };
  }
  render() {
    return (
      <div>
        X : <input type='text' value={this.state.x} /><br />
        Y : <input type='text' value={this.state.y} /><br />
        결과 : <span>{this.state.x + this.state.y}</span>
      </div>
    )
  }
}

export default App;
```

4. 제어컴포넌트와 비제어 컴포넌트(3)

- 실행 후 화면 기능 확인
  - 제어 컴포넌트는 UI 요소의 값이 상태나 속성에 강하게 연결되어 있으므로 변경이 불가능함.
  - 상태를 변경하면 UI 요소의 값이 바뀜. 따라서 이벤트를 이용해 상태를 변경해야 함

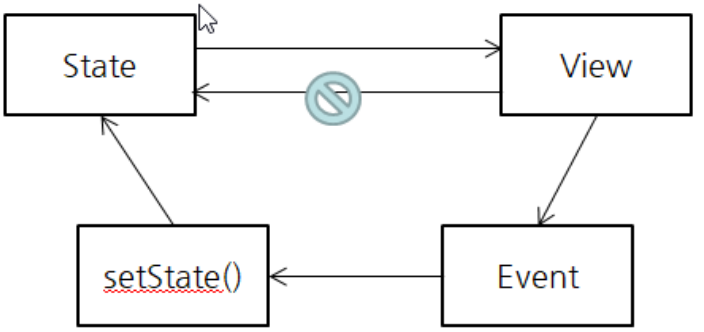


31

4. 제어컴포넌트와 비제어 컴포넌트(4)

어떻게 개선해야 할까?

- 양방향 바인딩을 제공하지 않음. 이벤트 핸들러를 통해 상태를 변경하도록 해야 함.



32

4. 제어컴포넌트와 비제어 컴포넌트(5)

- 제어가 가능하도록 이벤트 처리 적용

```
class App extends Component {
  constructor() {
    super()
    this.state = { x:0, y:0 };
  }
  change = (e) => {
    let newValue = parseInt(e.target.value);
    if (isNaN(newValue)) newValue = 0;
    if (e.target.id === "x")
      this.setState({ x: newValue })
    else
      this.setState({ y: newValue })
  }
  render() {
    return (
      <div>
        X : <input id="x" type='text' value={this.state.x} onChange={this.change}/><br />
        Y : <input id="y" type='text' value={this.state.y} onChange={this.change}/><br />
        결과 : <span>{this.state.x + this.state.y}</span>
      </div>
    )
  }
}
export default App;
```

33

■ 이벤트를 이용해 상태를 변경하면 UI가 갱신되도록 작성함.

- UI 에서 이벤트 발생
- 이벤트 처리를 통해 setState() 메서드로 상태 변경
- 변경된 상태가 다시 UI로 렌더링됨.

4. 제어컴포넌트와 비제어 컴포넌트(6)

비제어 컴포넌트

- state나 props에 종속되지 않기 때문에 이벤트를 이용해 처리하지 않아도 사용자가 값을 수정할 수 있음.
- 비제어 컴포넌트지만 초기값을 부여하고자 할 때
  - default~ 로 시작하는 Attribute를 이용함.
  - ex) value --> defaultValue
  - ex) checked --> defaultChecked
- 사용자가 HTML DOM 요소에서 입력한 값을 획득하기 위해 ref 특성을 사용할 수 있음.
  - 실제 HTML DOM에 접근하는 것은 고비용의 작업임. 꼭 필요한 경우가 아니라면 권장하지 않음
  - state나 props로 해결하기 어려운 경우에는 한가지 대안이 될 수 있음.

34

4. 제어컴포넌트와 비제어 컴포넌트(7)

- 비제어 컴포넌트 사용 컴포넌트 작성
- src/App2.js

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

class App extends Component {
    constructor() {
        super()
        this.state = { x:0, y:0, result:0 };
    }

    add() {
        let x = parseInt(this.elemX.value);
        let y = parseInt(this.elemY.value);
        if (isNaN(x)) x = 0;
        if (isNaN(y)) y = 0;
        this.setState({ x:x, y:y, result:x+y });
    }
```

```
    render() {
        return (
            <div className="container">
                X : <input id="x" type="text"
                        defaultValue={this.state.x}
                        ref={(input) => { this.elemX = input; }} /><br />
                Y : <input id="y" type="text"
                        defaultValue={this.state.y}
                        ref={(input) => { this.elemY = input; }} /><br />
                <button onClick={this.add.bind(this)}>
                    덧셈 계산</button><br/>
                결과 : <span>{this.state.result}</span>
            </div>
        )
    }
}

export default App;
```

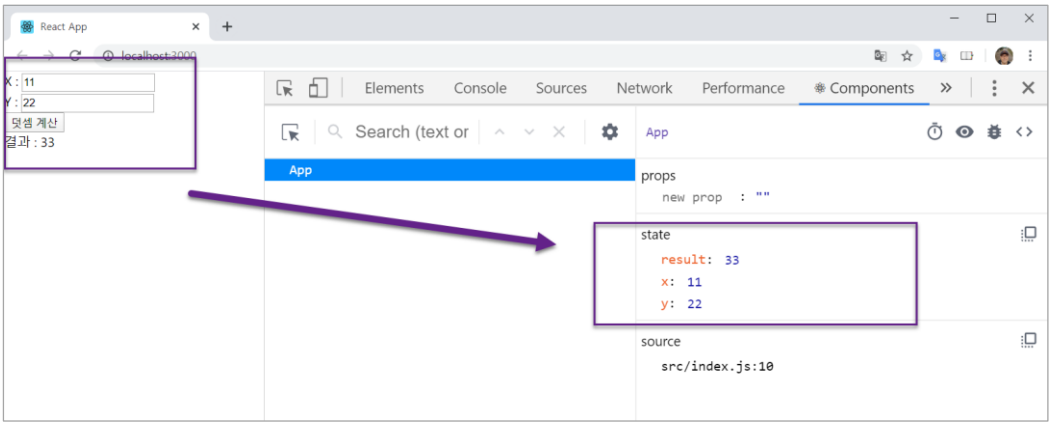
35

■ src/index.js에서 다음과 같이 변경한후 실행해본다.

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
//import App from './App';
import App from './App2';
.....
```

4. 제어컴포넌트와 비제어 컴포넌트(8)

- 실행 결과



■ '덧셈 계산' 버튼을 클릭했을 때 X, Y 입력필드의 값을 전달받을 수 있는 직접적인 방법이 없음.

■ 따라서 이러한 경우에는 ref 를 이용하는 것이 현실적인 방법임.

■ 예제 분석

- <input id="x" type="text" defaultValue={this.state.x} ref={(input) => { this.elemX = input; }} />
- ref 특성에 주어진 화살표 함수에 의해 전달된 input는 DOM 요소 자기자신을 참조한다. 즉 HTML DOM 요소 자신을 컴포넌트 객체 자기자신의 elemX 속성에 할당하는 것이다.
- '덧셈 계산' 버튼을 클릭했을 때 this.elemX.value와 같은 코드를 이용해 실제 DOM 요소의 특성, 속성을 모두 이용할 수 있게 된다.

5. 상태 심화(1)

상태와 불변성

- this.state를 직접 변경하는 것은 React의 상태 관리 기능을 우회하는 것이기 때문에 허용하지 않는다.
  - this.state가 객체인 경우 this.state.obj.x = 100; 과 같이 수정할 수는 있지만 상태 변경을 탐지해내지 못하므로 UI가 렌더링되지 않음.
- this.setState() 메서드를 호출하면 기본적으로 re-render를 수행함.
  - rendering을 최적화하기 위해서 값을 비교하여 render를 수행하지 않도록 해야 함.
  - shouldComponentUpdate 이벤트 생명주기 메서드를 이용함.

37

5. 상태 심화(2)

상태와 불변성(이어서)

- shouldComponentUpdate 생명주기 메서드
  - 현재의 컴포넌트 상태, 속성과 인자로 전달받은 새로운 상태, 속성을 비교하여 렌더링 할지 여부를 결정할 수 있는 메서드. 리턴값이 true일 때만 re-render를 수행함.
  - 이 생명주기 메서드를 이용해 UI 렌더링을 최적화할 수 있음
  - 이 메서드에서의 기존 상태, 속성과 새로운 상태, 속성의 비교를 빠르게 수행하려면?
    - Shallow compare!!!
  - shallow compare 만으로 상태의 변경여부를 확인할 수 없다면 객체의 트리를 추적해들어가면서 변경된 값이 있는지 여부를 확인해야만 함. --> 고비용의 작업
- 불변성을 확보하기 위한 방법
  - deepcopy! (npm install --save deepcopy)
    - 하지만 이 방법 또한 고비용의 작업이다. 객체 트리를 타고 내려가면서 매번 새로운 배열,객체를 복제함.
  - spread operator :
    - 깊이가 깊은 객체 트리를 효과적으로 처리할 수 없다.
  - 좀더 효율적인 방법이 필요함 : immer

■ shallow compare란?

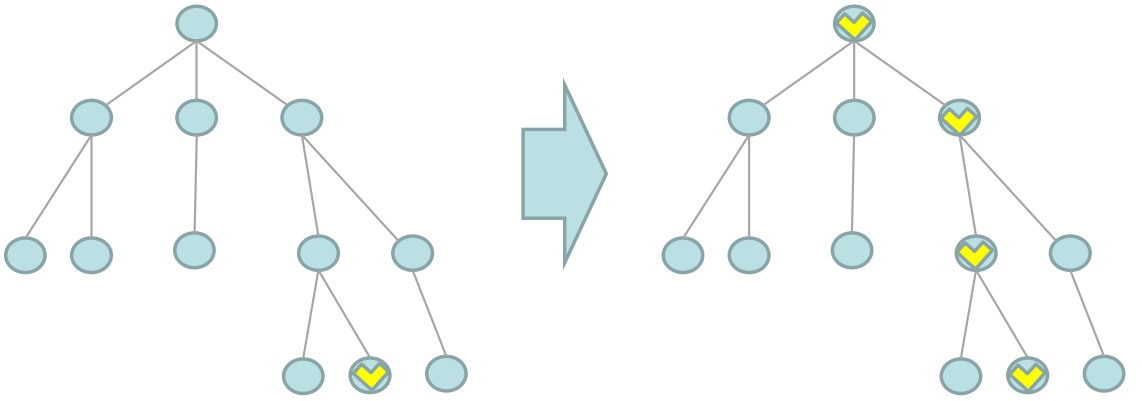
- 얕은 비교
- 두개의 객체를 비교할 때 객체의 메모리 주소가 같은지만을 비교하는 것을 말함.
- 이전 버전의 React에서는 shallowComapre 라이브러리를 이용했지만 React v16부터는 PureComponent의 사용하거나 직접 구현할 것을 권장함

■ 객체 내부의 트리 구조가 복잡할 경우 트리를 탐색하여 값의 변경여부를 확인하는 것은 성능에 나쁜 영향을 끼친다. 그렇기 때문에 shallow compare를 수행하는 것임

5. 상태 심화(3)

불변성 라이브러리(예:immer)를 사용하면...

- 객체 트리에서 변경된 노드로부터 상위로 거슬러 올라가는 경로상의 노드를 모두 변경함.



39

■ 간단한 객체나 배열인 경우는 spread operator를 사용할 수 있음.

- 하지만 복잡한 객체 트리인 경우에는 불변성을 확보할 수 없음

5. 상태 심화(4)

샘플용 객체 트리

- immutable-state-test 프로젝트 생성
  - npx create-react-app immutable-state-test
  - cd immutable-state-test
  - yarn add immer
- 객체 예

```
let quiz = {
    "students" : ["홍길동", "성춘향", "박문수", "변학도" ],
    "description" : "기본상식을 물어보는 테스트",
    "quizlist": [
     {
       "question": "한국 프로야구 팀이 아닌것은?",
       "options": [
          { "no":1, "option":"삼성라이온스" },
          { "no":2, "option":"기아타이거스" },
          { "no":3, "option":"두산베어스" },
          { "no":4, "option":"LA다져스" }
       ],
       "answer": 4
     },
```

```
 {
       "question": "2018년 크리스마스는 무슨 요일인가?",
       "options": [
          { "no":1, "option":"월" },
          { "no":2, "option":"화" },
          { "no":3, "option":"수" },
          { "no":4, "option":"목" }
       ],
       "answer": 2
     }
   ]
}
```

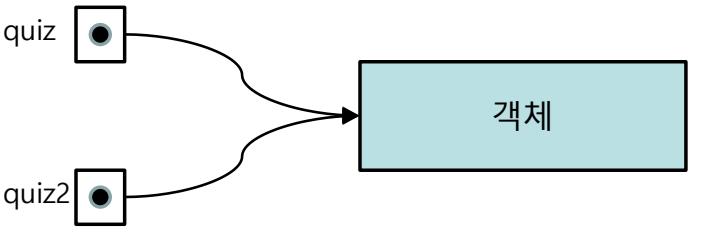

40

5. 상태 심화(5)

불변성을 갖추지 못한 경우

- index.js 의 가장 마지막에 다음 코드 작성

```
let quiz = { ...(생략-앞 페이지의 코드) }

let quiz2 = quiz;
quiz2.quizlist[0].options[0].option = "LG트윈스";

//true, true, true, true, true, true  ---> 동일한 객체
console.log(quiz === quiz2)
console.log(quiz.quizlist === quiz2.quizlist)
console.log(quiz.quizlist[0] === quiz2.quizlist[0])
console.log(quiz.quizlist[0].options[0] === quiz2.quizlist[0].options[0])
console.log(quiz.quizlist[0].options[0].option === quiz2.quizlist[0].options[0].option)
console.log(quiz.students === quiz2.students)
```



41

■ let quiz2 = quiz; 이 코드느 shallow copy를 수행한 것이다. 객체의 메모리 주소만을 복사하므로 같은 객체를 참조한다.

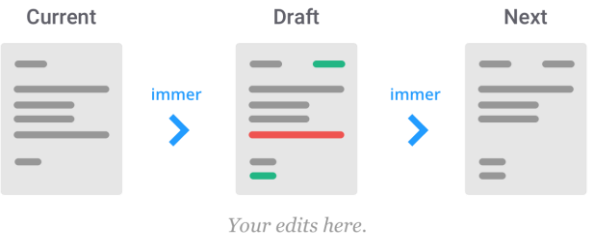
- 따라서 quiz2로 변경한 속성은 quiz로도 동일하게 확인이 가능하다.

5. 상태 심화(6)

immer

- 손쉬운 불변성 확보
- 자바스크립트 객체, 배열의 접근 방식을 그대로 사용함.

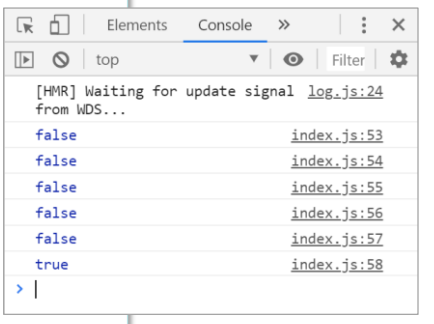
```
import produce from "immer"

const current = [
    {
        todo: "Learn es6",
        done: true
    },
    {
        todo: "Try immer",
        done: false
    }
]

const nextState = produce(current, (draft) => {
    draft.push({todo: "Tweet about it"})
    draft[1].done = true
})
```



5. 상태 심화(7)

immer.js

```
……
import produce from "immer";
……
let quiz = {  ……  };

const quiz2 = produce(quiz, draft => {
  draft.quizlist[0].options[0].option = "LG트윈스";
});

console.log(quiz.quizlist[0].options[0].option);

//false, false, false, false, false, true
console.log(quiz === quiz2)
console.log(quiz.quizlist === quiz2.quizlist)
console.log(quiz.quizlist[0] === quiz2.quizlist[0])
console.log(quiz.quizlist[0].options[0] === quiz2.quizlist[0].options[0])
console.log(quiz.quizlist[0].options[0].option === quiz2.quizlist[0].options[0].option)
console.log(quiz.students === quiz2.students)
```

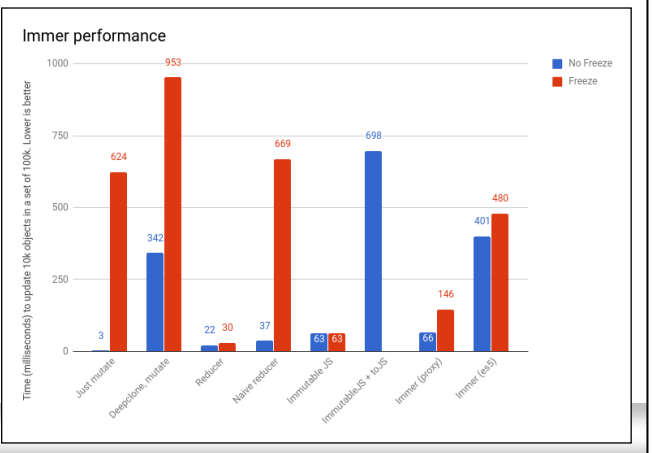


43

5. 상태 심화(8)

immer.js 특징

- 불변성을 확보하기 위해 표준 자바스크립트 데이터 구조와 API를 그대로 사용할 수 있다.
  - 상태 데이터가 특정 타입의 객체인 경우에 그 타입이 제공하는 기능을 그대로 활용할 수 있다.
- 기존의 불변성 헬퍼와 비교해서 더 간략한 코드를 사용한다.
- 나쁘지 않은 성능



5. 상태 심화(9)

- **불변성 관련 라이브러리를 반드시 이용해야 하는가?**
  - 그렇지 않다. 하지만 바람직하다.
    - 간단한 객체는 ES6의 Spread 연산자(...)을 이용할 수 있다.
  - 특히 UI 렌더링 성능 최적화를 위해서는 반드시 필요하다.
- **이 책에서는 immer를 사용하여 예제를 작성함.**
  - 상태 데이터에 대해 불변성을 확보하는 것이 중요하다는 점을 인지시키는 것이 이 절의 목적임.

45

6. 상태 컴포넌트 vs 비상태 컴포넌트(1)

상태의 보유 여부에 따른 컴포넌트의 유형

- 상태 컴포넌트 : stateful component
  - 상태(state)와 상태를 변경하는 기능을 보유하는 컴포넌트
- 비상태 컴포넌트 : stateless component
  - 상태가 없으며 부모 컴포넌트로부터 props를 전달받아 UI를 렌더링할 목적의 컴포넌트

비즈니스 로직 기능과 표현 기능으로 구분하는 방법

- 표현 컴포넌트 : presentational component
  - 컨테이너로부터 props를 전달받아 UI를 렌더링하는 기능을 수행함
  - 높은 재사용성, 행동 로직과의 분리
- 컨테이너 컴포넌트 : container component
  - UI와 스타일 정보를 포함하지 않음
  - 상태와 상태 변경 로직만을 가짐.

46

- 컨테이너 컴포넌트는 대부분 상태 컴포넌트이다.
- 표현 컴포넌트는 대부분이 비상태 컴포넌트이다.
- 컴포넌트의 재사용성은 비상태 컴포넌트, 표현 컴포넌트들이 뛰어나다.
- 따라서 컨테이너 컴포넌트가 부모 컴포넌트로써 자식컴포넌들에게 속성으로 표현할 데이터를 전달하는 방식을 자주 사용한다.
- 하지만 대규모의 애플리케이션인 경우는 컴포넌트들의 포함관계가 복잡해지면 속성 전달만으로 힘들어진다. 이러한 경우 Flux나 Redux와 같은 상태 관리 기능이 필요하다.

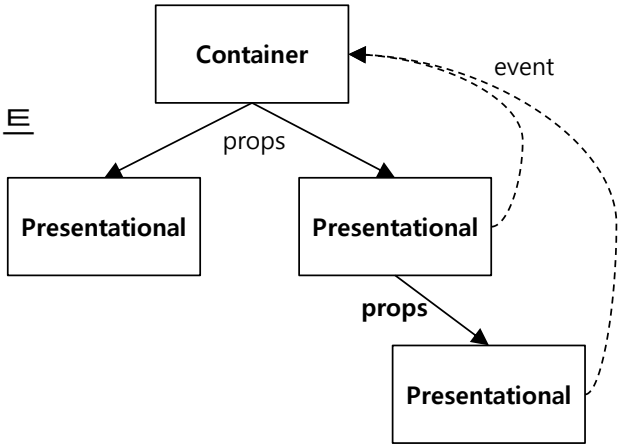
6. 상태 컴포넌트 vs 비상태 컴포넌트(2)

컴포넌트 각각 상태를 보유한다면?

- 애플리케이션의 상태 관리가 복잡해짐.
- 디버깅이 어려워짐

그렇다면 권장 사항은?

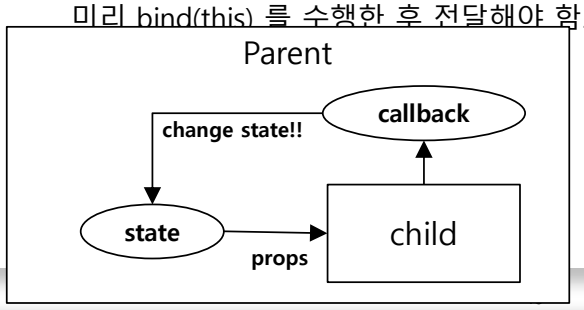
- 하나의 컨테이너 컴포넌트
- 여러개의 자식 표현 컴포넌트
- 그리고 속성으로 전달

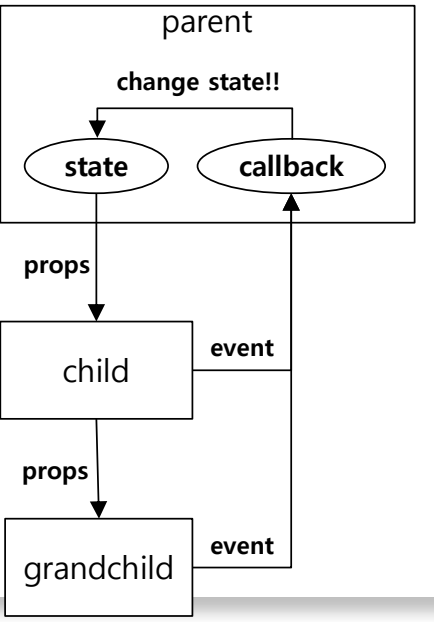


7. 부모-자식 컴포넌트간의 통신(1)

전체 개요

- 부모 -> 자식 으로의 정보 전달 방법
  - props를 이용해 전달함.
  - 이미 이전 예제에서 충분히 살펴보았음.
- 자식 -> 부모 로의 정보 전달 방법
  - 부모 컴포넌트의 콜백 함수(메서드)를 props를 이용해 자식 컴포넌트로 전달함.
  - 이 때 this는 부모 컴포넌트가 되도록 미리 bind(this) 를 수행한 후 전달해야 함.

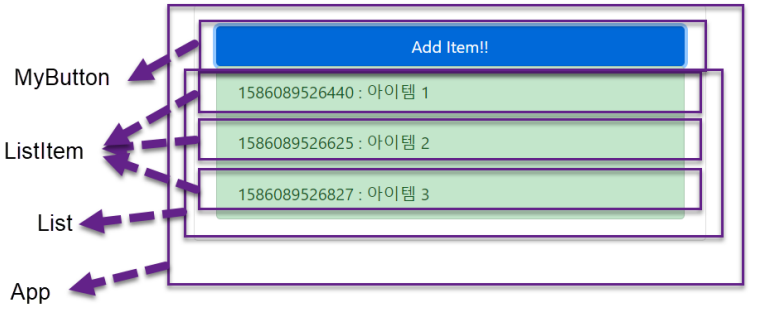




7. 부모-자식 컴포넌트간의 통신(2)

간단한 예제 작성

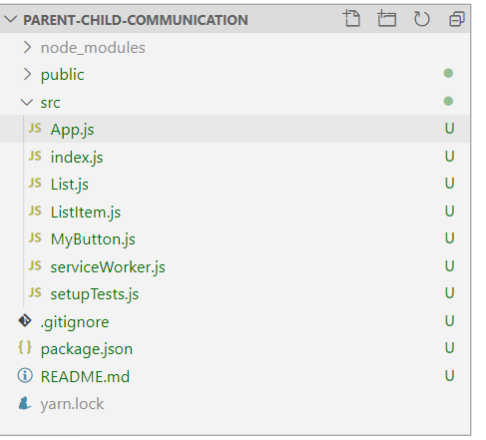
- 프로젝트 초기화
  - npx create-react-app parent-child-communication
  - cd parent-child-communication
  - yarn add bootstrap@4.x.x immer
- 전체 구조



7. 부모-자식 컴포넌트간의 통신(3)

- src 디렉터리의 App.js, App.test.js, App.css 삭제
- 오른쪽 그림 처럼 Component 파일 생성
  - App.js, List.js, ListItem.js, MyButton.js
- src/index.js 변경
  - bootstrap css 파일 import

```
import React from 'react';
import ReactDOM from 'react-dom';
import 'bootstrap/dist/css/bootstrap.css';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

reportWebVitals();
```



50

7. 부모-자식 컴포넌트간의 통신(4)

- src/App.js 새롭게 작성
  - 컨테이너 컴포넌트이며 상태와 상태 변경 로직을 보유함.

```
import React, { Component } from 'react';
import produce from 'immer';
import MyButton from './MyButton';
import List from './List';

class App extends Component {
    constructor() {
        super()
        this.state = {  itemlist : [ ] }
    }

    addItem() {
        if (!this.num) this.num = 0;
        this.num++;
        let newItemList = produce(this.state.itemlist,
            (draft)=> {
                draft.push({ no: new Date().getTime(),
                itemname: "아이템 " + this.num});
            }
        )
        this.setState({ itemlist : newItemList });
    }

    render() {
        return (
            <div className="container">
                <div className="card card-body bg-gray-300 m-2">
                    <MyButton addItem={this.addItem.bind(this)} />
                    <List itemlist={this.state.itemlist} />
                </div>
            </div>
        );
    }
}
export default App;
```


■ addItem 메서드를 ES6의 화살표 함수(Arrow Function)가 아닌 전통적인 함수로 작성하였음.

- 화살표 함수가 권장되지만 렌더링 최적화와 관련된 내용의 설명을 위해 일부러 전통적인 함수로 작성하였음

7. 부모-자식 컴포넌트간의 통신(5)

- src/MyButton.js
  - 버튼 클릭 이벤트 발생시 호출할 콜백 메서드를 속성으로 전달받음

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';

class MyButton extends Component {
    render() {
        return (
            <button className="btn btn-primary" onClick={() => this.props.addItem() }>
                Add Item!!
            </button>
        )
    }
}

MyButton.propTypes = {
    addItem: PropTypes.func.isRequired
};

export default MyButton;
```


52

7. 부모-자식 컴포넌트간의 통신(6)

- src/List.js
  - 렌더링할 itemlist 배열 정보를 속성으로 전달받음

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';
import ListItem from './ListItem';

class List extends Component {
    render() {
        let items = this.props.itemlist.map((item) => {
            return (<ListItem key={item.no} item={item} />)
        });

        return (
            <ul className="list-group">
                {items}
            </ul>
        )
    }
}

List.propTypes = {
    itemlist : PropTypes.arrayOf(PropTypes.object)
};

export default List;
```


53

7. 부모-자식 컴포넌트간의 통신(7)

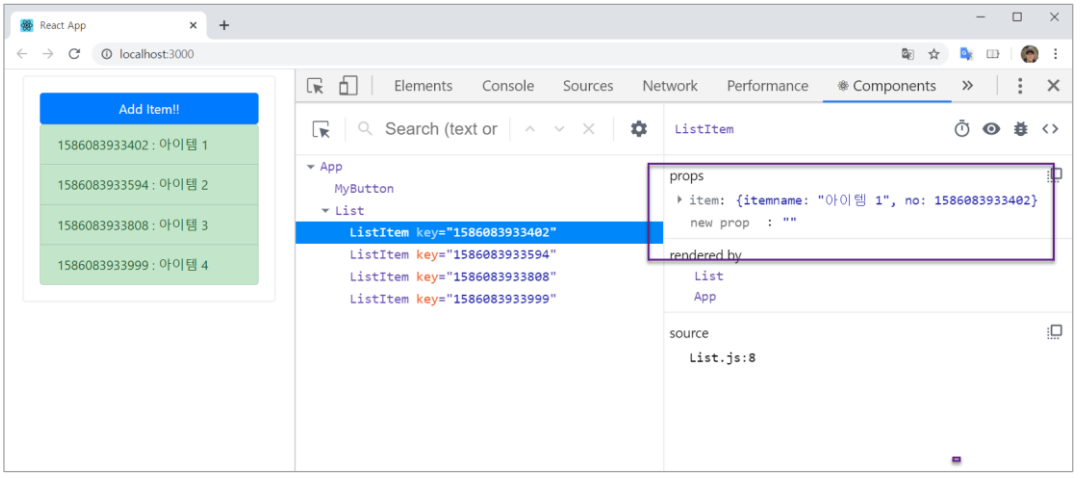
- src/ListItem.js
  - 아이템 하나를 렌더링할 때 필요한 정보를 List 컴포넌트로부터 전달받음

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';

class ListItem extends Component {
    render() {
        return (
            <li className="list-group-item list-group-item-success">
                {this.props.item.no} : {this.props.item.itemname}
            </li>
        )
    }
}

ListItem.propTypes = {
    item : PropTypes.object.isRequired
};

export default ListItem;
```


54

7. 부모-자식 컴포넌트간의 통신(8)

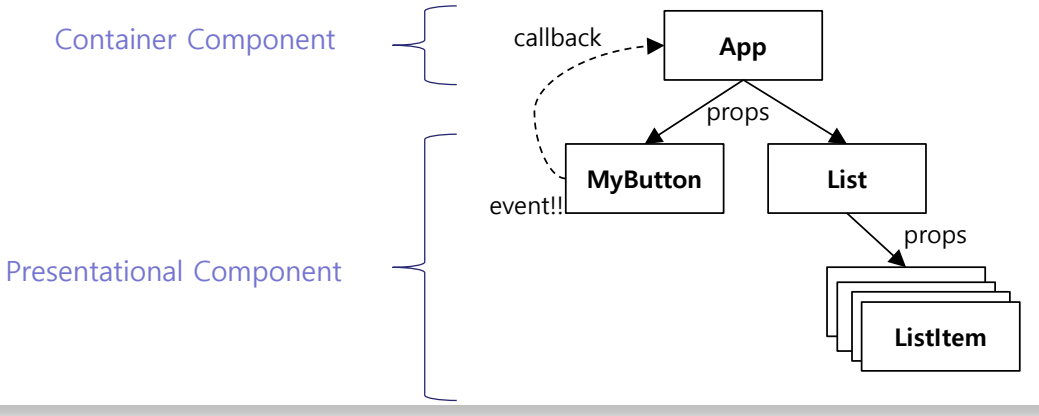
- 실행 결과



55

7. 부모-자식 컴포넌트간의 통신(9)

- 실행 결과 정리
  - App 컴포넌트의 addItem 메서드가 속성을 통해 MyButton 컴포넌트로 전달됨
  - MyButton 컴포넌트에서 버튼이 클릭되면 event를 통해서 addItem 메서드가 콜백됨
  - addItem 메서드 실행을 통해서 새로운 item 추가후 상태 갱신(setState)
    - 불변성 헬퍼를 이용해서 변경함
  - 변경된 상태는 List, ListItem으로 계층적으로 전달되어 re-render 함.



7. 부모-자식 컴포넌트간의 통신(10)

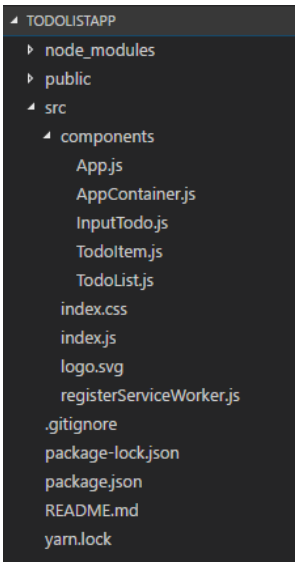
- 실행 결과 정리(이어서)
  - 대규모 애플리케이션의 경우 컴포넌트들이 복잡하게 계층 구조를 구성함
  - 이 경우 속성을 통해서 콜백메서드와 상태의 전달을 반복하는 것은 바람직하지 않음
    - Leaf Node의 컴포넌트가 사용할 속성과 콜백 메서드가 추가되어야 한다면 경로상의 모든 컴포넌트에도 속성이 추가되어야 함.
    - 따라서 유지보수가 어려워짐
  - 그렇기 때문에 Flux 아키텍처, ContextAPI와 같이 상태를 효과적으로 관리할 수 있는 기능이 필요함.

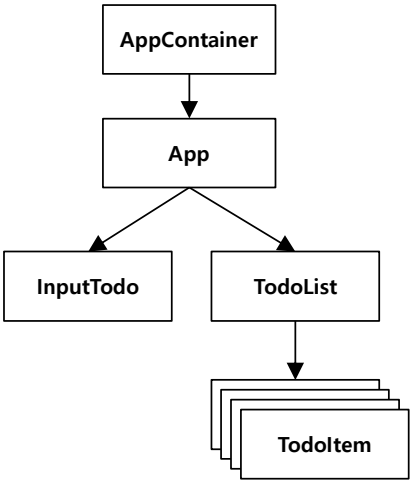
57

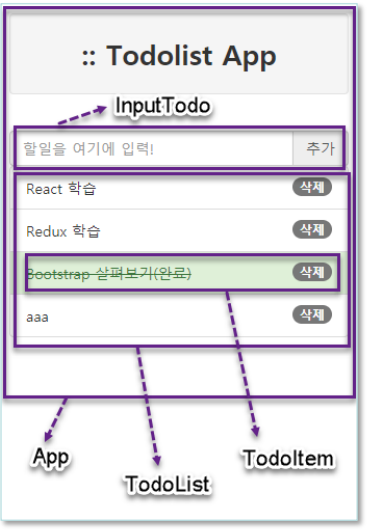
8. 예제 실습(1)

Todolist 앱 작성

- 전체 컴포넌트 구조







58

■ App, InputTodo, TodoList, TodoItem 은 표현 컴포넌트이자 비상태 컴포넌트이다.

■ AppContainer는 상태와 상태를 변경할 수 있는 액션(메서드)로 구성된 컨테이너 컴포넌트(상태 컴포넌트)이다.

■ 프로젝트 초기화

- npx create-react-app todolistapp
- cd todolistapp
- yarn add bootstrap immer
- src 디렉터리의 App.js, App.test.js, App.css 파일 삭제
- components 디렉토리 생성하고 App.js, AppContainer.js, InputTodo.js, TodoList.js, TodoItem.js 파일 생성

  – 화면 왼쪽 그림 참조

8. 예제 실습(2)

- 관리해야 할 상태
  - todolist

```
this.state = {
    todolist : [
        { no:1, todo:"React 학습", done:false },
        { no:2, todo:"Redux 학습", done:false },
        { no:3, todo:"Bootstrap 살펴보기", done:true }
    ]
}
```

- 상태를 변경 시키는 액션
  - addTodo : TodoItem 추가 기능
  - deleteTodo : no 값을 이용해 TodoItem 삭제 기능
  - toggleDone : no 값을 이용해 done 값을 toggle 처리(true->false, false->true)
  - 상태를 변경하기 위해 불변성 헬퍼 immer 사용

8. 예제 실습(3)

- 프로젝트 초기화
  - npx create-react-app todolistapp
  - cd todolistapp
  - yarn add bootstrap@4.x.x immer
  - src 디렉터리의 App.js, App.test.js, App.css 파일 삭제
- src/index.js 변경

```
import React from 'react';
import ReactDOM from 'react-dom';
import 'bootstrap/dist/css/bootstrap.css';
import './index.css';
import AppContainer from './components/AppContainer';
import reportWebVitals from './reportWebVitals';

ReactDOM.render(
  <React.StrictMode>
    <AppContainer />
  </React.StrictMode>,
  document.getElementById('root')
);

reportWebVitals();
```

8. 예제 실습(4)

- src/index.css 변경

```
body { margin: 0; padding: 0; font-family: sans-serif; }
.title { text-align: center; font-weight:bold; font-size:20pt; }
.todo-done { text-decoration: line-through; }
.container { padding:10px 10px 10px 10px; }
.panel-borderless { border: 0; box-shadow: none; }
.pointer { cursor:pointer; }
```

- src/AppContainer.js 작성

```
import React, { Component } from 'react';
import App from './App';
import produce from 'immer';

class AppContainer extends Component {
    constructor(props) {
        super(props);
        this.state = {
            todolist : [
                { no:1, todo:"React학습1", done:false },
                { no:2, todo:"React학습2", done:false },
                { no:3, todo:"React학습3", done:true },
                { no:4, todo:"React학습4", done:false },
            ]
        }
    }

(다음 페이지에 이어서...)
```

8. 예제 실습(5)

- src/AppContainer.js(이어서)

```
    addTodo = (todo) => {
        let newTodolist = produce(this.state.todolist, (draft)=> {
            draft.push({ no:new Date().getTime(), todo:todo, done:false })
        })
        this.setState({ todolist : newTodolist });
    }

    deleteTodo = (no) => {
        let index = this.state.todolist.findIndex((todo)=> todo.no === no);
        let newTodolist = produce(this.state.todolist, (draft)=> {
            draft.splice(index,1);
        })
        this.setState({ todolist : newTodolist });
    }
    toggleDone = (no) => {
        let index = this.state.todolist.findIndex((todo)=> todo.no === no);
        let newTodolist = produce(this.state.todolist, (draft)=> {
            draft[index].done = !draft[index].done;
        })
        this.setState({ todolist : newTodolist });
    }
```

(다음 페이지에 이어서...)

62

8. 예제 실습(6)

- src/AppContainer.js 작성(이어서)

```
    render() {
        return (
            <App
                todolist={this.state.todolist}
                addTodo={this.addTodo}
                deleteTodo={this.deleteTodo}
                toggleDone={this.toggleDone} />
        );
    }
}

export default AppContainer;
```

- AppContainer.js 는 컨테이너 컴포넌트로써 App 컴포넌트를 render() 한다.
- 상태와 상태를 변경하는 메서드를 보유하고 있으며 이것들을 속성을 통해 App 컴포넌트로 전달함.
- 메서드를 전달할 때 constructor에서 bind(this)를 수행함. 이렇게 함으로써 자식 컴포넌트에서 이 메서드를 호출하더라도 this는 AppContainer 컴포넌트를 가리킬 수 있도록 함.

63

8. 예제 실습(7)

- src/App.js 작성

```
import React, { Component } from 'react';
import InputTodo from './InputTodo';
import TodoList from './TodoList';
import PropTypes from 'prop-types';

class App extends Component {
   render() {
      return (
         <div className="container">
            <div className="card card-body bg-light"><div className="title">:: Todolist App</div></div>
            <div className="card card-default card-borderless">
               <div className="card-body">
                  <InputTodo addTodo={this.props.addTodo} />
                  <TodoList todolist={this.props.todolist}
                     toggleDone={this.props.toggleDone} deleteTodo={this.props.deleteTodo} />
               </div>
            </div>
         </div>
      );
   }
}
App.propTypes = {
   todolist : PropTypes.arrayOf(PropTypes.object),  addTodo : PropTypes.func.isRequired,
   deleteTodo : PropTypes.func.isRequired,  toggleDone : PropTypes.func.isRequired
}
export default App;
```

■ App 컴포넌트는 전달받은 속성을 다음과 같이 자식 컴포넌트로 전달한다.

- InputTodo 컴포넌트로 : addTodo 함수
- TodoList 컴포넌트로 : deleteTodo 함수, toggleDone 함수, todolist 배열

8. 예제 실습(8)

- src/InputTodo.js 작성

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';

class InputTodo extends Component {
    constructor() {
        super()
        this.state = {
            todo : ""
        }
    }

    addHandler = () => {
        this.props.addTodo(this.state.todo);
        this.setState({ todo: "" })
    }

    enterInput = (e) => {
        if (e.keyCode === 13)  {
            this.addHandler();
        }
    }

    changeTodo = (e) => {
        this.setState({ todo : e.target.value });
    }
```

```
    render() {
        return (
            <div className="row">
            <div className="col">
              <div className="input-group">
                <input id="msg" type="text"
                    className="form-control" name="msg"
                    placeholder="할일을 여기에 입력!"
                    value={this.state.todo}
                    onChange={this.changeTodo}
                    onKeyUp={this.enterInput} />
                <span
                 className="btn btn-primary input-group-addon"
                    onClick={this.addHandler}>추가</span>
              </div>
            </div>
            </div>
        );
    }
}

InputTodo.propTypes = {
    addTodo : PropTypes.func.isRequired
};

export default InputTodo;
```

65

■ InputTodo 컴포넌트는 로컬 상태를 가지는 컴포넌트이다. 상태는 컨테이너 컴포넌트에서 속성으로 전달하기를 권장하지만 InputTodo 컴포넌트와 같이 이 컴포넌트 내부에서만 사용되며 다른 컴포넌트에서는 이용되지 않는 로컬 데이터가 있다면 상태로 사용할 수 있다.

■ InputTodo는 제어컴포넌트로써 사용자가 입력하는 문자열을 처리하기 위해 로컬 상태를 사용한다.

■ 추가 버튼을 클릭하면 속성으로 전달받은 addTodo 함수를 호출하여 컨테이너 컴포넌트의 상태에 새로운 todolist 배열 요소를 추가한다.

8. 예제 실습(9)

- src/TodoList.js 추가

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';
import TodoItem from './TodoItem';

class TodoList extends Component {
    render() {
        let todoItems = this.props.todolist.map((item) => {
            return (
              <TodoItem key={item.no}  todoitem={item}  deleteTodo={this.props.deleteTodo}
                  toggleDone={this.props.toggleDone} />
            )
        })
        return (
            <div className="row"> <div className="col">
                <ul className="list-group">
                    {todoItems}
                </ul>
            </div></div>
        );
    }
}
TodoList.propTypes = {
    todolist : PropTypes.arrayOf(PropTypes.object),
    toggleDone : PropTypes.func.isRequired,
    deleteTodo : PropTypes.func.isRequired
}
export default TodoList;
```

■ TodoList에서 TodoItem 하나하나를 표현하기 위해 TodoItem 컴포넌트를 참조하여 사용한다.

■ TodoItem 컴포넌트는 item 각각과 deleteTodo 함수, toggleDone 함수를 속성으로 전달받는다.

■ Array의 map 메서드를 이용해 컴포넌트 객체 배열을 렌더링할 때는 반드시 key Attribute 값을 전달해야 하며 이 값은 Unique 해야 한다. key Attribute의 의미에 대해서는 Virtual DOM을 다룰 때 자세하게 다룰 것이다.

8. 예제 실습(10)

- src/TodoItem.js 추가

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';

class TodoItem extends Component {
    render() {
        let itemClassName = "list-group-item";
        if (this.props.todoitem.done) itemClassName +=" list-group-item-success";
        return (
            <li className={itemClassName}>
                <span className={this.props.todoitem.done ? "todo-done pointer": "pointer"}
                    onClick={ ()=>this.props.toggleDone(this.props.todoitem.no) }>
                    {this.props.todoitem.todo}{ this.props.todoitem.done ? "(완료)" : "" }
                </span>
                <span className="float-right badge badge-secondary pointer"
                    onClick={ ()=>this.props.deleteTodo(this.props.todoitem.no) }>삭제</span>
            </li>
        );
    }
}
TodoItem.propTypes = {
    todoitem : PropTypes.object.isRequired,
    toggleDone : PropTypes.func.isRequired,
    deleteTodo : PropTypes.func.isRequired,
};
export default TodoItem;
```

67

■ TodoItem 컴포넌트는 컨테이너 컴포넌트의 상태 데이터를 계층적으로 속성을 통해 전달받았다.

■ 속성으로 전달받은 배열의 한건의 값을 이용해 UI를 생성한다.

- done 값이 true, false일 때 각각 다른 스타일이 적용될 수 있도록 3항 조건 연산식과 if문을 사용 적용할 스타일 클래스명을 다르게 지정하도록 하였다.

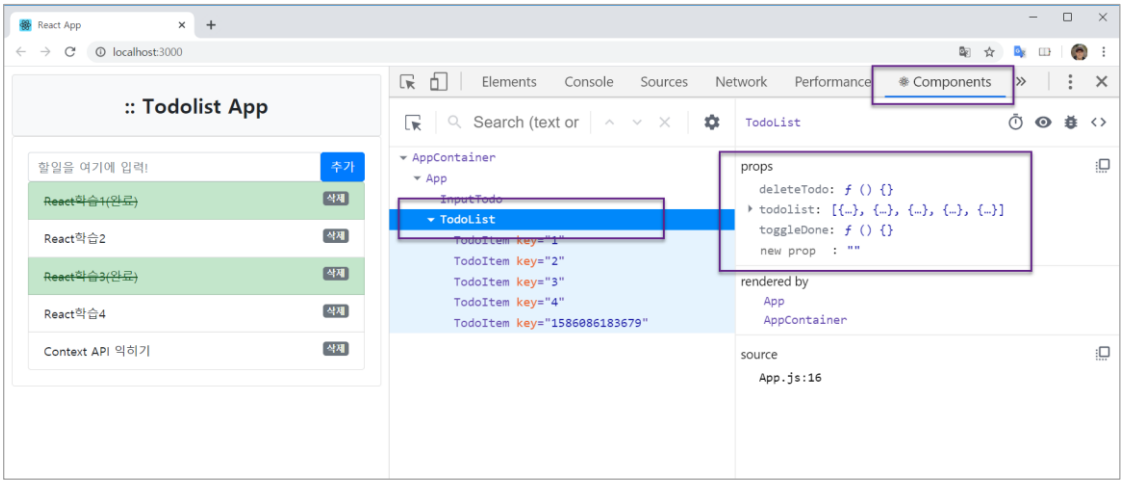
■ 화면에서 TodoItem 명을 클릭하면 속성으로 전달받은 toggleDone 함수가 호출된다. 이때 컨테이너 컴포넌트로 전달할 아이템의 키는 no 필드이므로 this.props.no를 인자로 전달하며 호출한다.

■ 별도의 삭제버튼을 클릭할 경우에는 deleteTodo 함수가 호출된다. this.props.todoitem.no 값을 전달하는 것은 toggleDone과 동일한 이유에서이다.

8. 예제 실습(11)

실행 결과

- yarn start



68