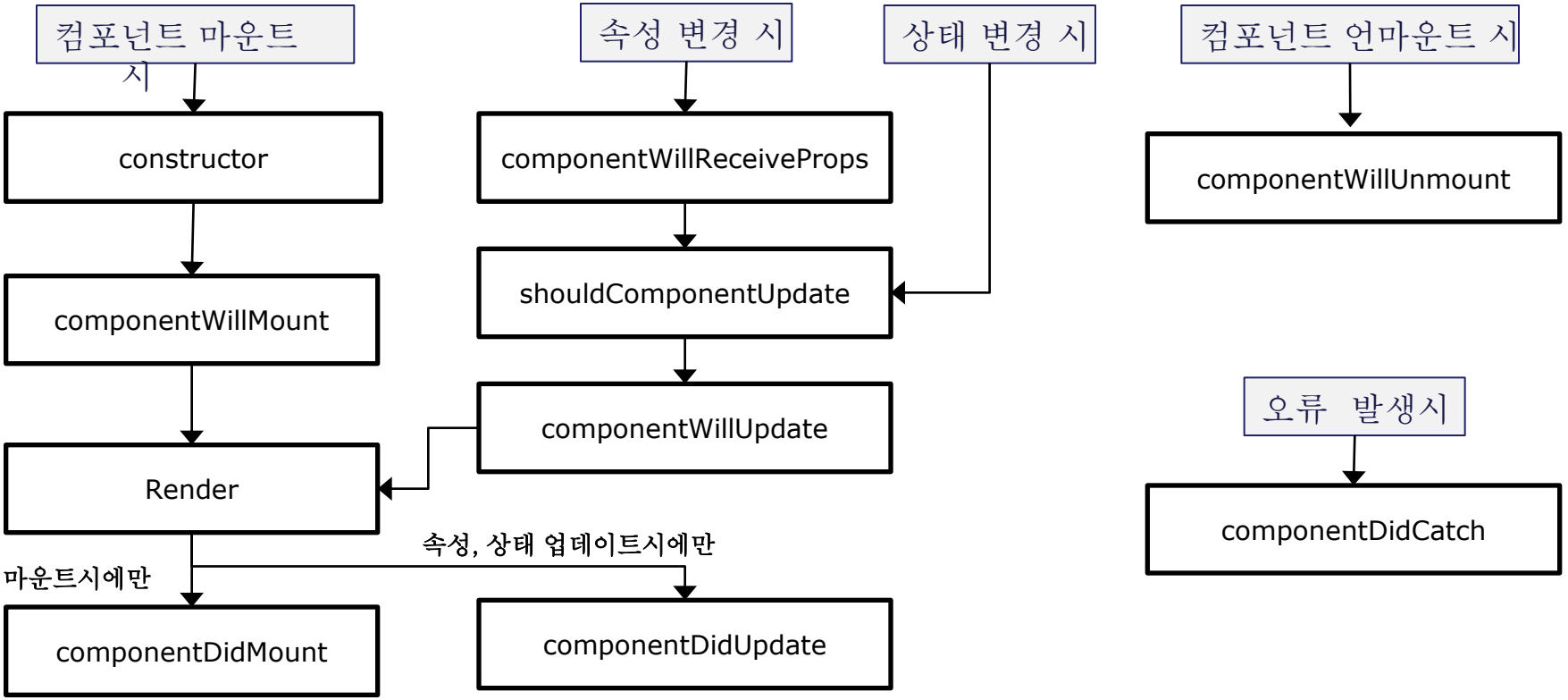
1. 컴포넌트 생명주기 메서드(1)

컴포넌트 생명주기 메서드란?

- 컴포넌트가 생성되거나 상태나 속성이 업데이트되거나 할 때 자동으로 호출되는 메서드



1. 컴포넌트 생명주기 메서드(2)

컴포넌트 마운트시의 흐름

- 컴포넌트의 인스턴스가 만들어지고 DOM에 추가될 때 호출됨
- constructor(props) : 생성자
  - 마운트 되기 전에 호출되며 상태를 초기화하기 위한 최적의 시점
  - 인자는 속성이며 constructor 내부의 첫 줄에 반드시 super(props)가 포함되어야 함.
  - 상태가 없다면 생성자를 구현할 필요가 없음
  - 부모 컴포넌트로부터 속성을 전달받아 상태를 초기화할 수 있으나 부모 컴포넌트로부터 전달되는 속성이 변경되면 이를 이용해 상태를 매번 변경해줘야 하는 불편함이 있으므로 권장하지 않음. --> 속성이 변경되더라도 초기 상태를 유지해야 하는 경우라면 사용가능
- componentWillMount()
  - 마운팅이 일어나기 직전에 호출됨.
  - 이 단계에서 setState() 를 이용해 상태를 변경하더라도 render()가 추가로 호출되지 않음
  - 서버사이드 렌더링을 사용할 때 유일하게 호출되는 생명주기 메서드

1. 컴포넌트 생명주기 메서드(3)

- componentDidMount()
  - 컴포넌트의 마운트가 완료된 후에 호출. DOM에 대한 초기화를 하기에 적합한 시점.
  - 원격 서버로부터 데이터를 로드하여 초기화하거나 이벤트 구독(subscription)을 설정하기 적절한 시점
    - 서버와 소켓을 연결하거나 이벤트 구독을 수행했다면 componentWillUnmount 단계에서 반드시 해제해야 함.
  - 이 단계에서 setState()가 호출되면 re-render가 일어나지만 실제 브라우저의 HTML DOM에 반영되기 전에 호출되는 것임
    - 따라서 render() 가 두번 호출되지만 사용자는 최종 상태만 조회하게 됨.

속성, 상태 변경 시의 흐름

- componentWillReceiveProps(nextProps)
  - 전달 인자 : 새로운 속성
  - 이미 마운트된 컴포넌트가 새로운 속성을 전달받을 때 호출됨. 마운트 단계에서는 이 메서드가 호출되지 않음.
  - 상태 변경시에는 이 메서드는 호출되지 않음
  - 사용 예) 부모 컴포넌트로부터 전달받은 속성으로 로컬 상태를 변경할 때
    - 기존 속성과 변경된 속성을 비교하여 다를 경우에만 로컬 상태를 setState()를 이용해 변경함

1. 컴포넌트 생명주기 메서드(4)

- shouldComponentUpdate(nextProps, nextState)
  - 전달 인자 : 새로운 속성, 새로운 상태
  - 리턴값이 중요함
    - 리턴값이 true : 이후 단계의 메서드를 실행함(componentWillUpdate, render, componentDidUpdate)
    - 리턴값이 false : 이후 단계의 메서드를 실행하지 않음.
  - 이 메서드를 작성하지 않으면 기본적으로 true를 리턴함
    - PureComponent 인 경우 shallowCompare 한후 다른 경우에만 true를 리턴하도록 이미 구현되어 있으므로 이 메서드 작성이 불가능하다.
  - 이 단계에서 속성과 상태를 비교할 때 deepCompare는 권장하지 않음
    - 고비용의 작업이므로 오히려 성능을 저하시킨다.
    - 따라서 5장에서 다룬 불변성 헬퍼를 사용하고 shallowCompare하는 편이 바람직하다.
- componentWillUpdate(nextProps, nextState)
  - 마운트 단계에서는 실행되지 않음
  - render()가 호출되기 직전에 실행됨.
    - render() 실행 전에 준비 작업을 하고 싶을 때 사용할 수 있음
  - 이 단계에서는 setState() 메서드로 상태를 변경할 수 없음

1. 컴포넌트 생명주기 메서드(5)

- componentDidUpdate(prevProps, prevState)
  - DOM 업데이트가 일어난 직후에 실행됨.
  - 마운트 시에는 실행되지 않음.
  - 실제 HTML DOM이 업데이트된 후이므로 실제 DOM을 이용한 작업을 하기에 적절함
  - 현재 속성과 이전 속성을 비교하여 다른 경우에만 지정된 작업을 실행하도록 할 수 있음
    - 예) 현재의 속성과 이전 속성이 다르면 서버측으로 데이터를 요청하고자 할 때

컴포넌트 언마운트 시의 흐름

- componentWillUnmount
  - 컴포넌트가 언마운트될 때 실행됨(예: 화면이 완전히 전환될 때)
  - 소켓 서버로의 네트워크 연결 해제, 이벤트 구독 해제 등의 작업을 수행할 수 있음

1. 컴포넌트 생명주기 메서드(6)

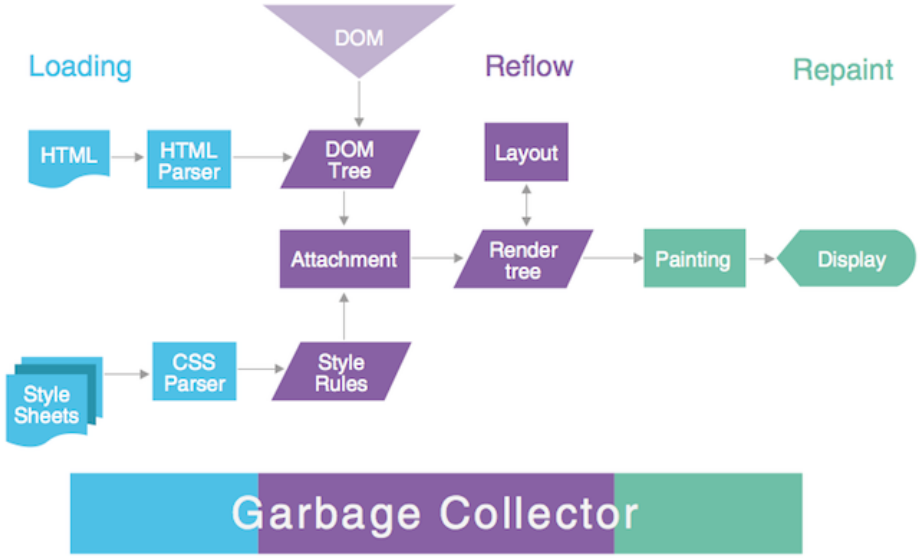
오류 발생시의 흐름

- componentDidCatch(error, info)
  - 자신과 자식 컴포넌트 트리에서 오류가 발생할 경우 호출됨.
  - 이후 fallback UI 가 지정되어 있다면 fallback UI를 표시할 수 있음
  - 자세한 내용은 다음 내용을 참조
    - https://reactjs.org/blog/2017/07/26/error-handling-in-react-16.html
    - 에러처리 전용 컴포넌트를 작성하고 에러처리를 하고 싶은 범위의 컴포넌트 트리를 래핑함.
    - 에러 처리 전용 컴포넌트는 자신의 자식 트리상의 컴포넌트에서의 오류만 처리함. 자신의 오류는 처리하지 못함.

2. 가상DOM과 조정 작업(1)

HTML DOM이 느리다?

- DOM 조작은 빠르지만 브라우저에서 reflow, repaint 과정이 느림
  - Reflow : layout이라고도 부름. 렌더링할 DOM Tree를 새로이 만들고 HTML Element 각각의 위치를 계산하고 배치함.
  - Repaint : HTML Element에 스타일을 요소에 입히고 그려냄



https://mobidev.biz/blog/how_to_optimize_the_performance_of_phonegap_apps

2. 가상DOM과 조정 작업(2)

React는?

- Always re-render on update!!
- 개발자가 원하는 출력물만을 선언적으로 작성하기 때문에 컴포넌트 전체를 re-render 하듯이 개발할 수 밖에 없음
- 따라서 UI 성능을 위해서 Virtual DOM이 반드시 필요함
- Virtual DOM 트리를 비교하면서 차이가 나는 부분만을 업데이트함.
  - 이것을 조정(Reconciliation) 작업이라 부름
  - 브라우저 DOM의 업데이트 로직은 개발자가 신경쓰지 않아도 됨.

2. 가상DOM과 조정 작업(3)

Virtual DOM

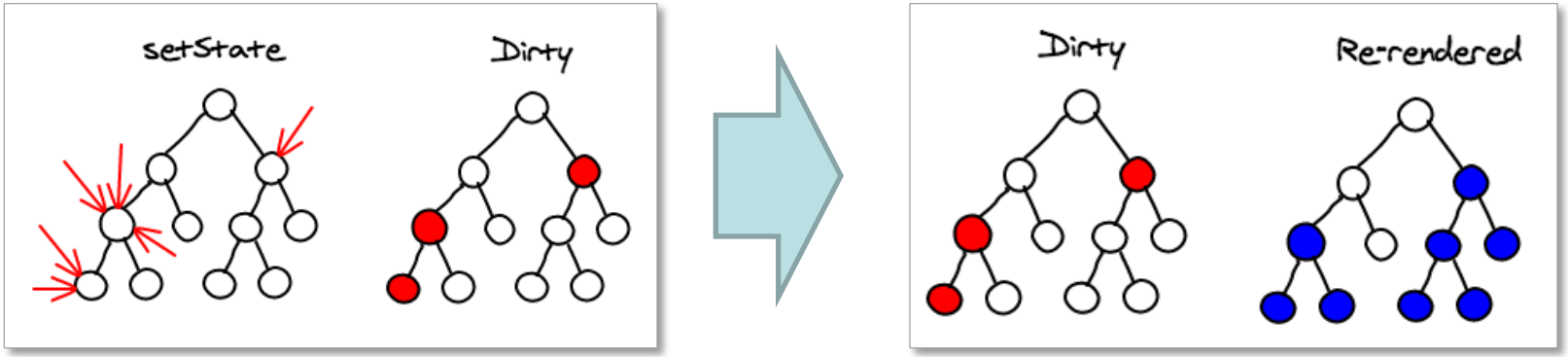
- DOM에 대한 추상화된 객체
- 업데이트해야 할 정보를 메모리에 저장함.
- Virtual DOM의 업데이트 비용은 작음
  - 실제 HTML DOM을 업데이트하는 것이 아니므로 Reflow, Repaint가 일어나지 않음
- 이전 스냅샷과 Virtual DOM 변경 후의 스냅샷을 비교해 차이가 발생한 부분에 대해서만 실제 HTML DOM을 업데이트함.

https://github.com/drborges/react-in-a-nutshell

2. 가상DOM과 조정 작업(4)

Virtual DOM의 Diff 알고리즘

- DOM 트리의 노드들을 비교하면서 노드가 다른 유형일 경우 기존 노드를 버리고 새로운 노드로 교체
- 노드가 같은 유형인 경우
  - Attribute와 Style을 비교하여 변경함.
  - 부모 컴포넌트에서 re-render가 실행되면 자식 컴포넌트로 속성을 전달해 자식 컴포넌트도 re-render를 수행함.
  - 선택적 re-render를 위해 shouldComponentUpdate() 생명주기 메서드를 이용할 수 있음.

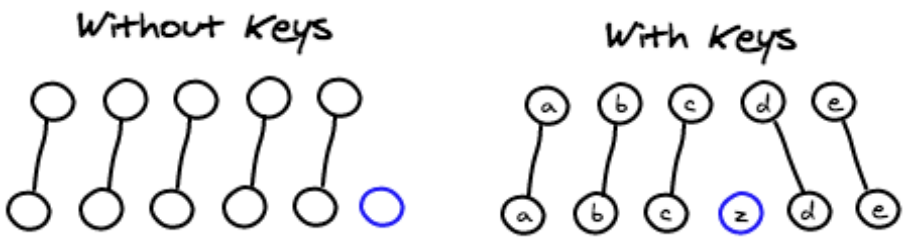


http://calendar.perfplanet.com/2013/diff/

2. 가상DOM과 조정 작업(5)

key 특성

- 컴포넌트 내부에서 반복적으로 자식 컴포넌트, 요소를 렌더링할 때 사용
- 반복적인 리스트의 변경 사항을 추적하기 힘듬
  - 새로운 요소가 추가, 삽입되는 경우
  - 요소들의 순서가 변경되는 경우
  - 특정 요소가 삭제되는 경우
- 반복적으로 렌더링할 때 key를 부여하지 않으면 React는 경고를 일으킴.
- ke 야 함.(index번호(X))

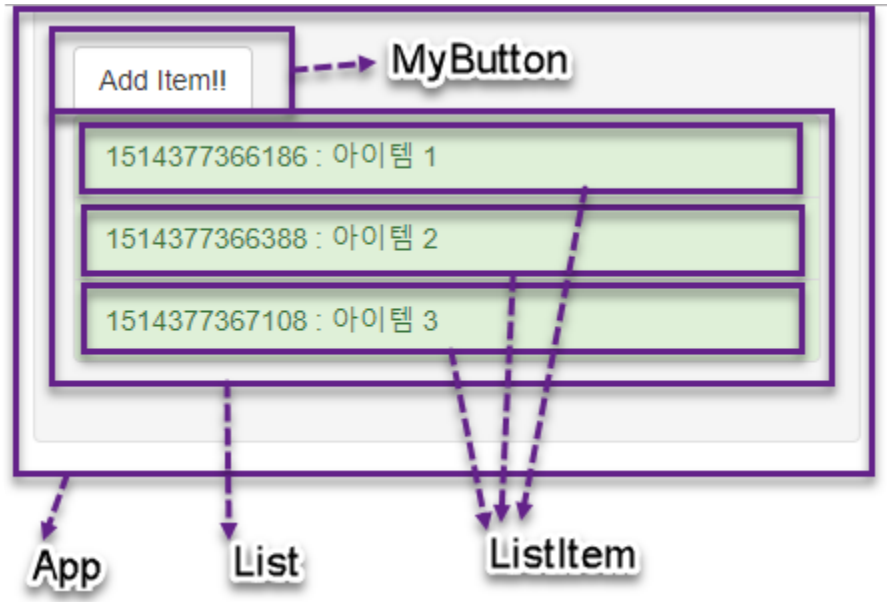


https://calendar.perfplanet.com/2013/diff/

2. 가상DOM과 조정 작업(6)

조정 과정 확인을 위한 예제

- 프로젝트 초기화
  - 5장에서 작성한 parent_child 프로젝트 디렉터리를 다른 곳에 복사한 후 디렉터리명을 reconcil로 변경하자.
- 전체 컴포넌트 구조 리뷰



2. 가상DOM과 조정 작업(7)

- 콘솔 로그에 출력하는 코드 추가
  - ListItem.js, List.js, MyButton.js 컴포넌트에 추가
  - 단순히 render() 메서드가 실행되고 있음을 확인하는 코드!!

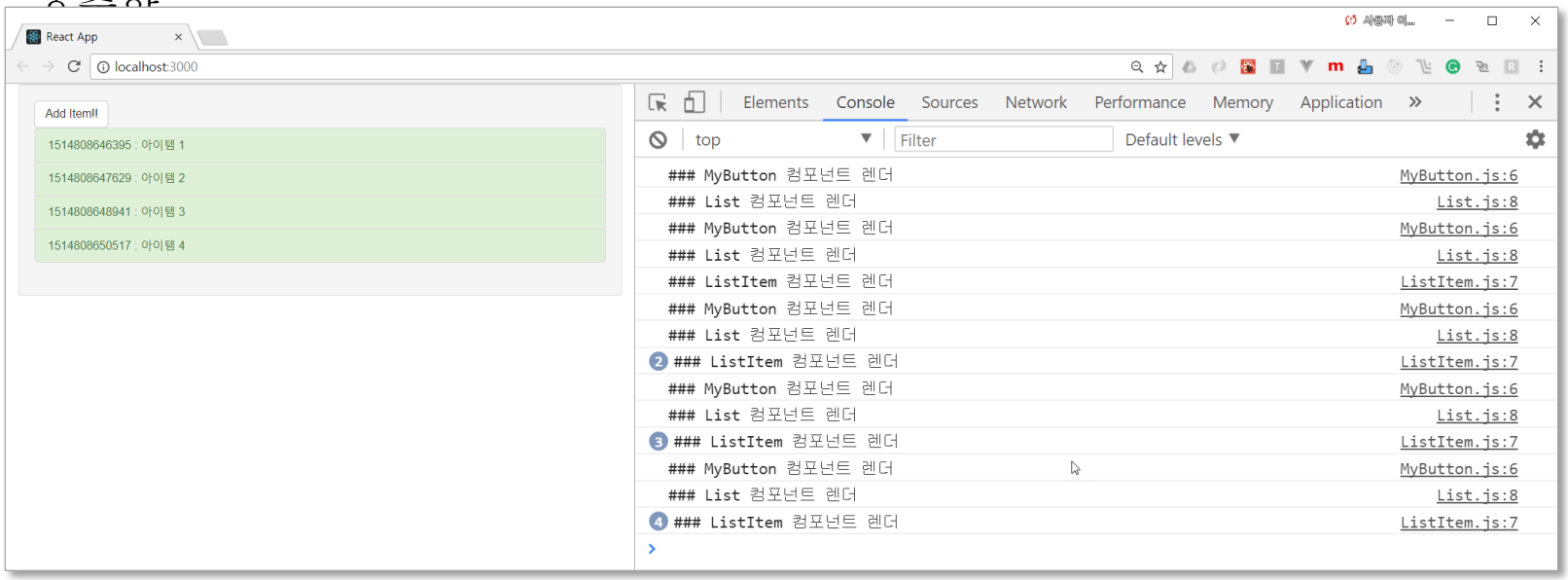
```
class ListItem extends Component {
    render() {
        console.log("### ListItem 컴포넌트 렌더")
        return ( ..... )
    }
}
```

```
class List extends Component {
    render() {
        console.log("### List 컴포넌트 렌더")
        ......
    }
}
```

```
class MyButton extends Component {
    render() {
        console.log("### MyButton 컴포넌트 렌더")
        ......
    }
}
```

2. 가상DOM과 조정 작업(8)

- 실행 결과 확인
  - 새로운 아이템이 추가되면서 setState() 가 호출되고 App.js에서 render()가 호출되면?
    - 자식 컴포넌트들이 모두 render()!!
    - 하나의 아이템을 추가하지만 ListItem 컴포넌트 모두에서 render()가 호출됨.
    - 기존에 마운트된 ListItem 컴포넌트는 Update될 사항이 없다면 re-render할 이유가 없음.
  - 심지어 MyButton 컴포넌트는 전혀 업데이트될 필요가 없지만 re-render를 호출함



2. 가상DOM과 조정 작업(9)

- shouldComponentUpdate 메서드로 최적화
  - ListItem.js 최적화

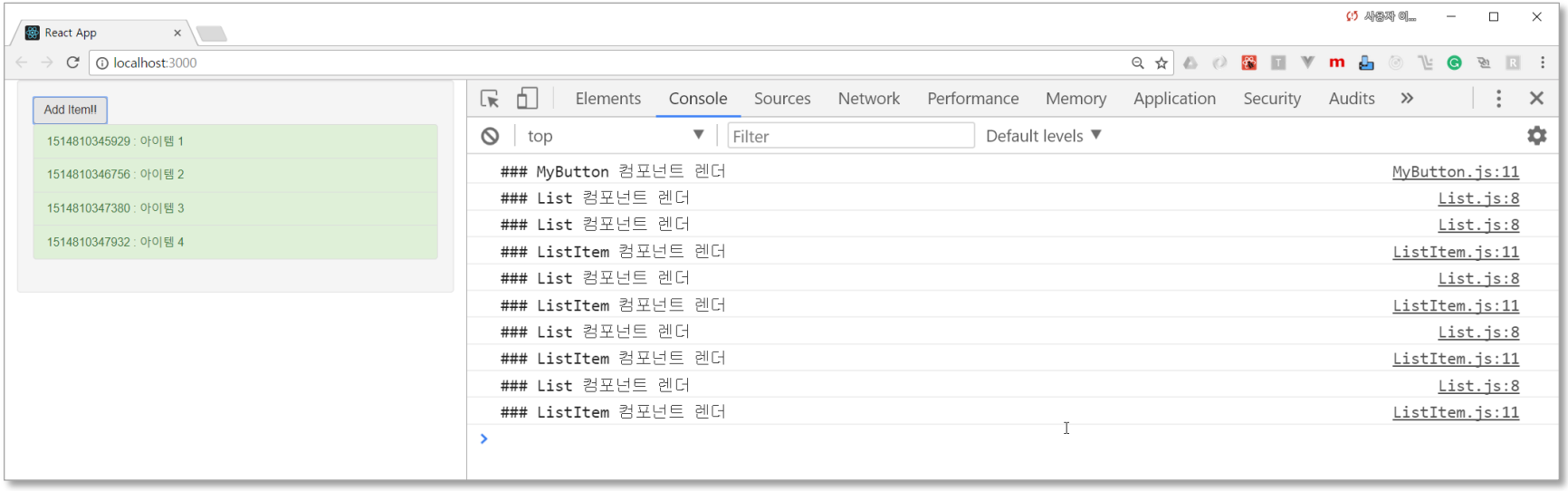
```
class ListItem extends Component {
    shouldComponentUpdate(nextProps, nextState) {
        return this.props.no !== nextProps.no || this.props.item !== nextProps.item;
    }
    render() { ..... }
}
```

  - MyButton.js 최적화
    - 이 컴포넌트는 속성을 이용해 렌더링하지 않으므로 부모 컴포넌트에서 전달된 속성이 변경되더라도 re-render할 필요가 없음

```
class MyButton extends Component {
    shouldComponentUpdate(nextProps, nextState) {
        return false;
    }
    render() {
        ......
    }
}
```

2. 가상DOM과 조정 작업(10)

- 다시 실행하여 브라우저 화면에서 아이템 추가
  - 가장 마지막에 추가된 ListItem 컴포넌트만 렌더링!!
  - MyButton.js는 마운트 될때만 렌더링!!



3. PureComponent(1)

React.PureComponent

- React.Component
  - shouldComponentUpdate() 메서드가 구현되어 있지 않기 때문에 setState()가 호출되면 무조건 render()를 호출함.
  - Rendering 과정을 최적화하기 위해
    - 개발자가 직접 shouldComponentUpdate() 메서드를 작성하여 비교하도록 작성해야 함.
- React.PureComponent
  - shouldComponentUpdate()가 shallowCompare 하도록 이미 구현되어 있음
    - 객체의 메모리 주소가 같을 경우 render()를 호출하지 않음.
    - 참조 타입이 아닌 값 타입인 경우는 값이 같으면 render()를 호출하지 않음
    - 명시적으로 shouldComponentUpdate를 작성할 수 없음
  - 표현 컴포넌트에서 사용하기에 적합함.
    - 새롭게 전달받은 props와 현재의 props를 shallowCompare 하여 일치한다면 render()를 수행하지 않으므로 최적화하기가 용이함.
    - 불변성 헬퍼(immutability-helper)를 사용하는 경우에 더욱 효과적임.

3. PureComponent(2)

PureComponent 적용

- reconcil 프로젝트 코드 변경
  - ListItem.js 컴포넌트를 PureComponent로 변경

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

class ListItem extends PureComponent {

  //shouldComponentUpdate 메서드는 삭제

   render() {
      console.log("### ListItem 컴포넌트 렌더")
      return (
         <li className="list-group-item list-group-item-success">
            {this.props.no} : {this.props.item}
         </li>
      )
   }
}
……
```

  - 실행 결과는 이전과 동일함.

3. PureComponent(3)

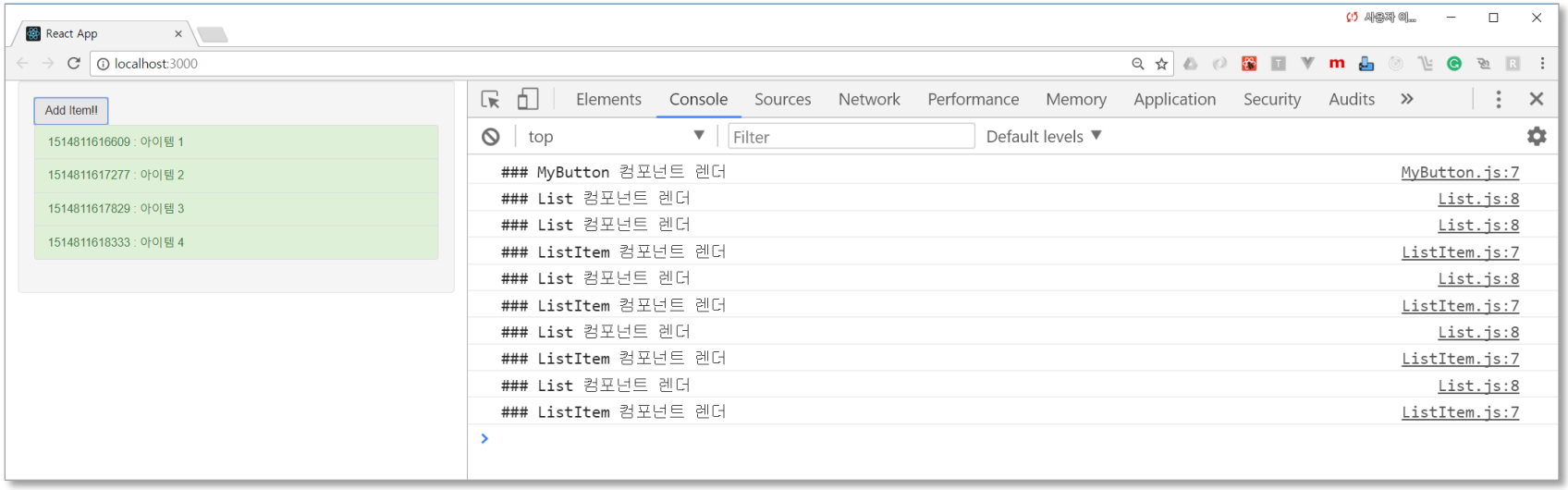
- MyButton 컴포넌트를 PureComponent로 변경하면?

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

class MyButton extends PureComponent {
   ......
}
......
```

- 매번 렌더링을 수행함. 원인은 App.js에서 렌더링할 때마다 bind(this)를 매번 호출한 후 속성으로 전달하기 때문임.
- App.js의 생성자에서 한번만 bind(this)를 수행한 후 속성으로 전달하도록 해야 함.
- App.js 코드는 아래 참조

3. PureComponent(4)

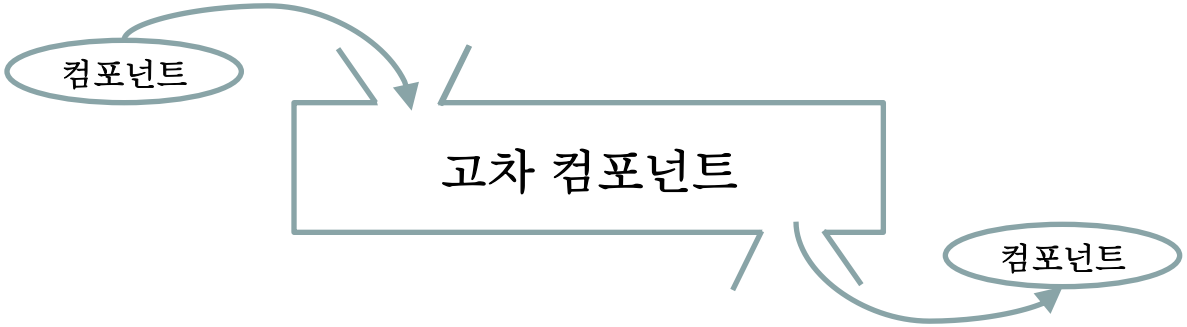
- 실행 결과



4. 고차 컴포넌트(1)

HOC : Higher Order Component

- 컴포넌트를 입력값으로 받아 새로운 기능을 추가하여 다시 리턴하는 컴포넌트
- 고차 함수 : Higher Order Function
  - 다른 함수를 인자로 받거나 그 결과로 함수를 반환하는 함수다.
- 컴포넌트들 사이의 공통 로직을 분리하고 재사용할 수 있음
  - 사용자 로그인 여부, 권한 상태 확인 기능 추가
  - 에러 발생시 에러 페이지 보여주기
  - 로깅 기능 추가



4. 고차 컴포넌트(2)

HOC 기능 테스트

- 프로젝트 초기화
  - 이전 절까지 작성했던 reconcil 프로젝트를 복사한 후 디렉터리명을 hoc로 변경함.
  - MyButton, List, ListItem 컴포넌트의 render() 메서드 내부에 작성했던 console.log() 코드를 모두 주석처리함.
- 로깅 기능을 추가하는 고차 컴포넌트 작성
  - console에 로깅하는 기능 추가
  - 컴포넌트가 마운트 될 때의 render() 시간 측정
    - componentWillMount -> render -> componentDidMount
  - 컴포넌트가 업데이트될 때의 render() 시간 측정
    - componentWillUpdate -> render -> componentDidUpdate
  - 어느 컴포넌트인지도 로깅해야 함.
    - component의 name값을 받아내야 함.
  - 속성을 통해서 로깅할지 여부를 결정할 수 있도록...
  - src/Logger.js 참조

4. 고차 컴포넌트(3)

- src/Logger.js

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

let Logger = LoggingComponent => class Logger extends Component {
    componentWillMount() {
        if (this.props.isLog) {
            this.start = new Date();
        }
    }
    componentDidMount() {
        if (this.props.isLog) {
            let ts = new Date().getTime() - this.start.getTime();
            console.log(`### ${this.componentName} mounted : ${ts}ms`);
        }
    }
    componentWillUpdate(nextProps, nextState) {
        if (this.props.isLog) {
            this.start = new Date();
        }
    }
    componentDidUpdate(prevProps, prevState) {
        if (this.props.isLog) {
            let ts = new Date().getTime() - this.start.getTime();
            console.log(`### ${this.componentName} updated : ${ts}ms`);
        }
    }
```

4. 고차 컴포넌트(4)

- src/Logger.js (이어서)

```
    render() {
        this.componentName = LoggingComponent.name;
        return <LoggingComponent {...this.props} />;
    }
};

Logger.propTypes = {
    isLog : PropTypes.bool
};

Logger.defaultProps = {
    isLog: false
};

export default Logger;
```

4. 고차 컴포넌트(5)

- src/List.js
  - Logger 고차 컴포넌트를 거쳐서 로깅 기능을 추가하도록 변경

```
……
import Logger from './Logger';

class ListItem extends Component {
   ……
}
……
export default Logger(ListItem);
```

- src/List.js

```
……
import Logger from './Logger';

class List extends Component {
    render() {
        let items = this.props.itemlist.map((item) => {
            return (<ListItem  isLog="true" key={item.no} {…item} />)
        });
        ……
    }
}
……
export default Logger(List);
```

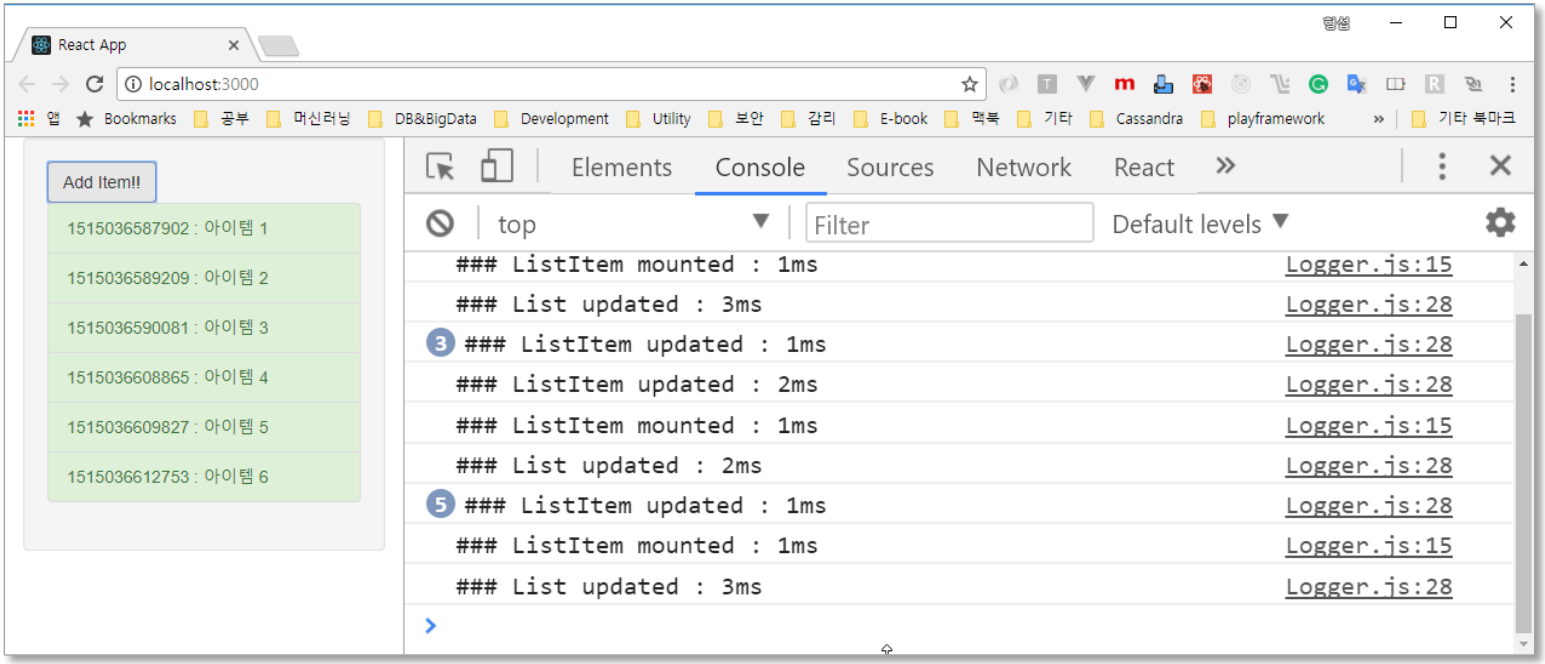
4. 고차 컴포넌트(6)

- src/App.js

```
……
class App extends Component {
    ……
    render() {
        return (
            <div className="container">
                <div className="well">
                    <MyButton addItem={this.addItem} />
                    <List itemlist={this.state.itemlist} isLog="true" />
                </div>
            </div>
        );
    }
}

export default App;
```

4. 고차 컴포넌트(7)

- 실행 결과



5. Portal(1)

Portal 이란?

- render할 때 자식 요소를 부모 컴포넌트의 DOM 트리 밖에 존재하는 DOM 요소 내에 추가할 수 있는 기능
  - vue.js의 slot과 유사한 개념
- 언제 사용할까?
  - Modal, Hover, Tooltip과 같이 메인화면과 독립적인 UI를 구성하고자 할 때
- 이벤트 버블링
  - 부모 요소의 범위 밖의 다른 요소에 표현되지만 컴포넌트 계층 구조는 부모 컴포넌트 내부에 있는 것으로 간주하므로 이벤트 버블링을 활용할 수 있다.
- 반드시 사용해야만 하는 것은 아니지만 적절히 활용하면 편리함.

5. Portal(2)

간단한 예제

- 개요
  - 모달 다이얼로그박스(Modal Dialogbox)
  - 다이얼로그 박스를 닫아야 메인화면이 활성화됨.
- 프로젝트 초기화
  - create-react-app portaltest
  - cd portaltest
  - yarn add bootstrap
  - src 디렉터리의 App.js, App.test.js, App.css 파일 삭제

5. Portal(3)

- src/Modal.js 작성
  - 모달 다이얼로그박스로 보여줄 컴포넌트

```
import React, { Component } from 'react'
import './Modal.css'

class Modal extends Component {
    constructor(props) {
        super(props)
        this.closeSelf = this.closeSelf.bind(this);
    }

    closeSelf() {
        this.props.toggleModalBox();
    }

    render() {
        const { showModal } = this.props;
        let modalBox = null;
        if (showModal) {
            modalBox = (
                <div className="modal">
                <div className="form">
                    <h3 className="heading">
                        :: {this.props.header}
                    </h3>
                    <div className="form-group">
                        {this.props.children}
                    </div>
                    <div>
                        <button className="button"
                            onClick={this.closeSelf}>닫기</button>
                    </div>
                </div>
                </div>
            )
        }
        return modalBox;
    }
}

export default Modal;
```

5. Portal(4)

- src/Modal.css

```
.modal {
    display: block; position: fixed; z-index: 1;
    left: 0; top: 0; width: 100%; height: 100%;
    overflow: auto; background-color: rgb(0,0,0);
    background-color: rgba(0,0,0,0.4);
}
.form {
    background-color: white; margin:100px auto;
    max-width: 400px; min-width: 200px; font: 13px "verdana";
    padding: 10px 10px 10px 10px;
}
.form div {
    padding: 0;  display: block;  margin: 10px 0 0 0;
}
.form .heading {
    background: #33A17F; font-weight: 300;
    text-align: left; padding : 20px; color: #fff;
    margin:5px 0 30px 0; padding: 10px; min-width:200px;
    max-width:400px;
}
.form .button {
    background: #2B798D; padding: 8px 15px 8px 15px;
    border: none; color: #fff; font-size:8pt;
}
```

5. Portal(5)

- src/Portal.js
  - HTML DOM에서 modal-area 라는 id를 가진 요소를 찾아 children 속성으로 전달된 자식 컴포넌트를 내부에 추가함.

```
import React from 'react'
import ReactDOM from 'react-dom'

class Portal extends React.Component {
  constructor(props) {
    super(props)

    this.modalArea =
        document.getElementById('modal-area')
    this.container =
        document.createElement('div')
  }

  componentDidMount() {
    this.modalArea.appendChild(this.container)
  }

  componentWillUnmount() {
    this.modalArea.removeChild(this.container)
  }
```

```
  render() {
    return ReactDOM.createPortal(
        this.props.children,
        this.container
    )
  }
}

export default Portal;
```

5. Portal(6)

- public/index.html 변경
  - App 컴포넌트가 나타날 div#root 이외에 모달 창을 보여줄 div#modal-area를 추가함

```
<!DOCTYPE html>
<html lang="en">
  <head>
    ……
    <title>React App</title>
  </head>
  <body>
    ……
    <div id="root"></div>
    <div id="modal-area"></div>
  </body>
</html>
```

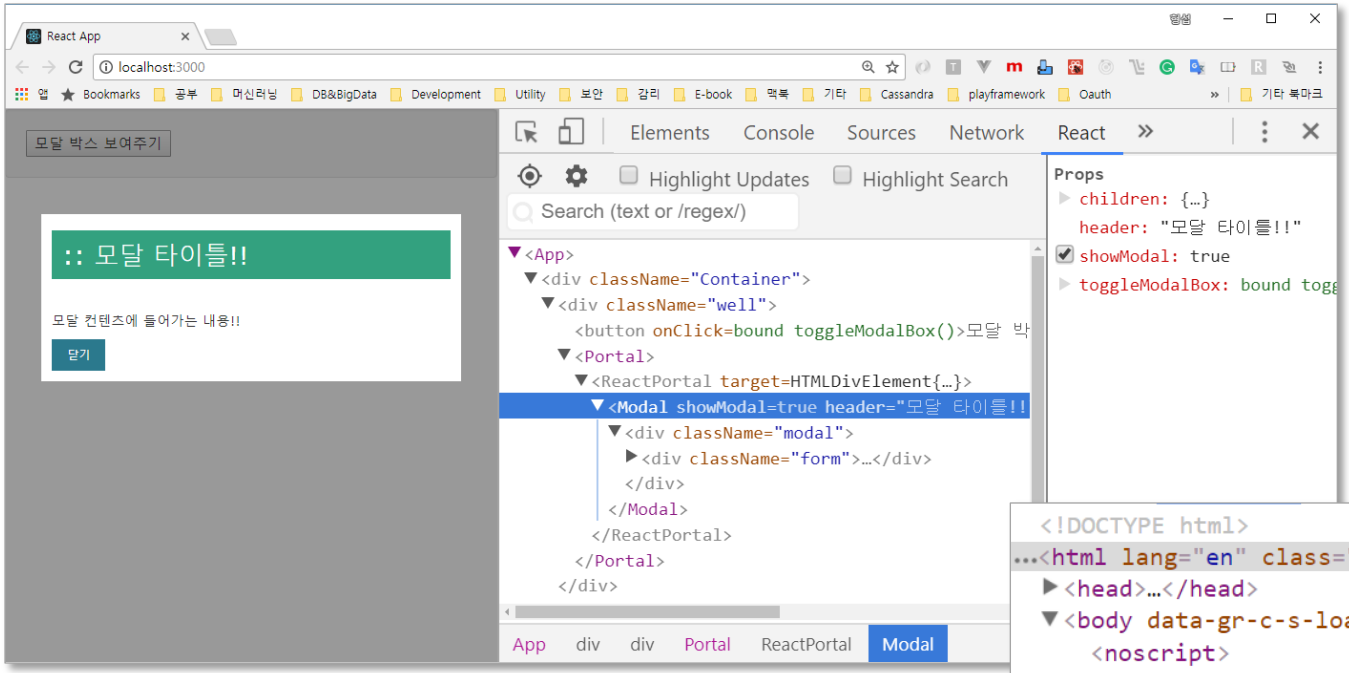
5. Portal(7)

- src/App.js 작성

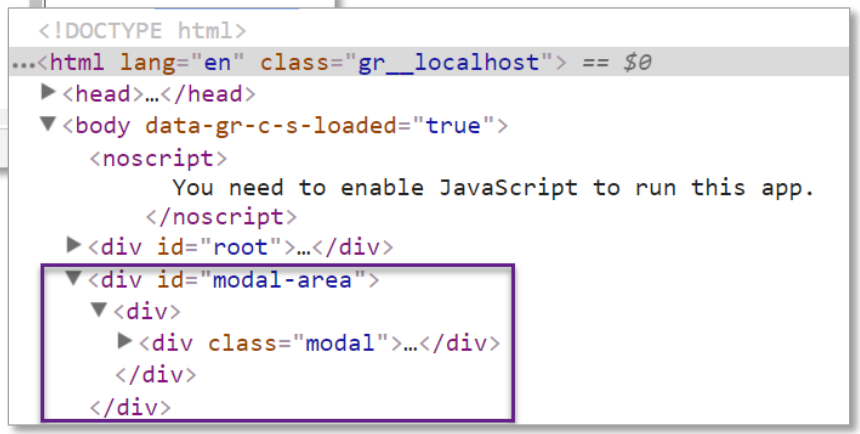
```
import React, { Component } from 'react'
import Portal from './Portal'
import Modal from './Modal'

class App extends Component {
    constructor(props) {
        super(props)
        this.state = { showModal: false }
        this.toggleModalBox =
              this.toggleModalBox.bind(this);
    }

    toggleModalBox() {
        this.setState({
            showModal: !this.state.showModal })
    }

    render() {
        const { showModal } = this.state

        return (
            <div className="Container">
                <div className="well">
                    <button onClick={this.toggleModalBox}>
                        모달 박스 보여주기
                    </button>
                    <Portal>
                        <Modal
                            showModal={showModal}
                            header="모달 타이틀!!"
                            toggleModalBox={this.toggleModalBox}>
                                <p>모달 컨텐츠에 들어가는 내용!!</p>
                        </Modal>
                    </Portal>
                </div>
            </div>
        )
    }
}

export default App;
```

5. Portal(8)

- 이제까지 실행 결과



컴포넌트 계층 구조



Element 계층 구조

5. Portal(9)

- 이벤트 버블링을 활용하도록 코드 변경
  - toggleModalBox 메서드를 Modal 컴포넌트로 전달하지 않고 이벤트 버블링을 이용하도록 App.js를 변경함.
- src/App.js 변경

```
……
class App extends Component {
……
    render() {
        const { showModal } = this.state
        return (
            <div className="Container">
                <div className="well" onClick={this.toggleModalBox}>
                    <button onClick={this.toggleModalBox}>모달 박스 보여주기</button>
                    <Portal>
                        <Modal showModal={showModal} header="모달 타이틀!!">
                            <p>모달 컨텐츠에 들어가는 내용!!</p>
                        </Modal>
                    </Portal>
                </div>
            </div>
        )
    }
}
export default App
```

5. Portal(10)

- src/Modal.js 변경
  - constructor와 closeSelf 메서드 삭제
  - 닫기 버튼에 있던 onClick 이벤트도 삭제

```
……
class Modal extends Component {

    render() {
        const { showModal} = this.props;
        let modalBox = null;
        if (showModal) {
            modalBox = (
                <div className="modal">
                <div className="form">
                    ……(생략)
                    <div>
                        <button className="button">닫기</button>
                    </div>
                </div>
                </div>
            )
        }
        return modalBox;
    }
}
……
```

5. Portal(11)

- 실행 결과 화면은 이전과 동일
  - Modal 컴포넌트는 Element 계층 구조로는 div#modal-area 내부에 렌더링되었지만 컴포넌트 계층 구조로는 여전히 App 컴포넌트 내부임.
  - Modal 컴포넌트의 닫기 버튼을 클릭하면 이벤트 버블링에 의해 App 컴포넌트 내부의 div#well 에 설정된 이벤트가 호출된다.

5. Portal(12)

react-portal 패키지를 이용하여 코드 변경

- src/Portal.js 와 같은 것을 별도로 만드는 것이 불편함
  - 이런 이유로 React v16에서 react-portal이 추가되었음
- 패키지 추가
  - yarn add react-portal (또는) npm install --save react-portal
- src/Portal.js가 더이상 필요치 않음
  - 삭제 또는 파일명 변경(예:Portal2.js와 같이...)

5. Portal(13)

- src/App.js 코드 변경

```
……
//import Portal from './Portal'
import {Portal} from 'react-portal'

class App extends Component {
    ……
    render() {
        const { showModal } = this.state
        return (
            ……
            <Portal node={document && document.getElementById('modal-area')}>
                ……
            </Portal>
            ……
        )
    }
}
……
```

- 더욱 자세한 내용은 아래에서
  - https://github.com/tajo/react-portal

6. 컴포넌트의 설계(1)

컴포넌트를 설계할 때 고려할 점

- 재사용성
  - 독립적인 요소, 스타일을 가지며 재사용 가능한 수준에서 분할
  - 독립성, 재사용성을 높이려면 속성을 이용한 순수 컴포넌트(표현 컴포넌트, 비상태 컴포넌트)로 작성하는 것이 바람직함.
- 관리성
  - 컴포넌트 단위로 관리, 조정 가능하도록 분할
  - 예) reconcile 프로젝트에서 List.js 단위로만 재사용된다고 하더라도 ListItem 컴포넌트를 작성하여 세분화시키는 것이 바람직함 → 조정 작업 처리!!
  - 한 컴포넌트 내부에서 지나치게 복잡한 작업을 수정하지 않도록 컴포넌트를 분할함.

6. 컴포넌트의 설계(2)

컴포넌트를 설계할 때 고려할 점(이어서)

- 상속보다는 조합 방식을 사용
  - React.Component를 상속받아 컴포넌트를 작성하지만 계층 구조로 상속받아 전체 UI를 구성하지 않고 컴포넌트들을 조합하도록 구성하는 것이 권장됨.
  - 웹화면 = 요소들의 조합 === 컴포넌트들의 조합
- 여러 컴포넌트에서 공통적으로 사용하는 기능은?
  - 상속보다는 고차 컴포넌트(HOC:Higher Order Component)!!
  - 공통 로직의 분리가 핵심
    - 로직을 구체화시키고 공통 로직을 찾아내 분리함.

7. TodoList 앱 예제(1)

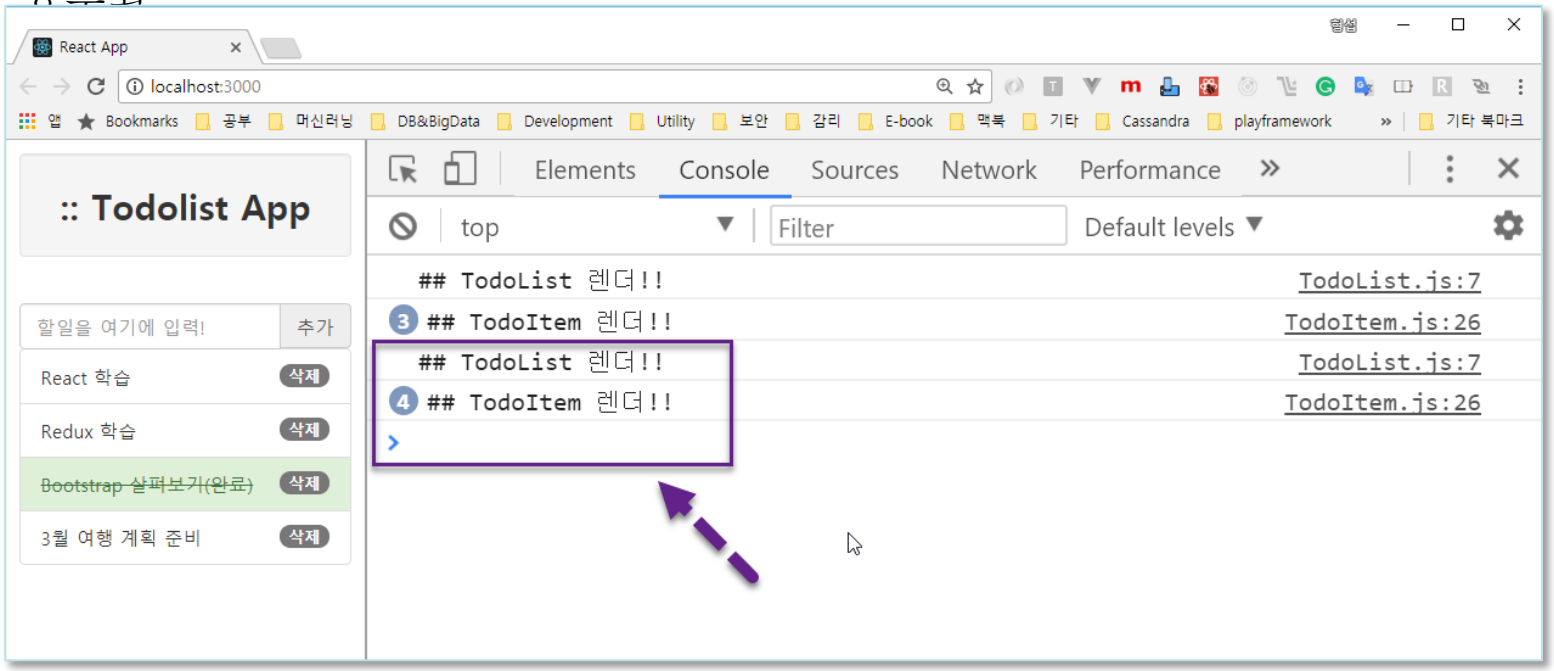
- 5장에서 작성했던 Todolist 앱 예제에 최적화 기능을 적용함.
  - 기존 예제 문제점 확인
    - 그에 출력하는 코드

```
.....
class TodoList extends Component {
    render() {
        console.log("## TodoList 렌더!!")
        .....
    }
}
.....
```

```
.....
class TodoItem extends Component {
    .....
    render() {
        console.log("## TodoItem 렌더!!")
        .....
    }
}
.....
```

7. TodoList 앱 예제(2)

- 실행 결과 확인
  - 하나의 TodoItem만 추가되었을 뿐이지만 모든 TodoItem에서 다시 render()가 호출됨

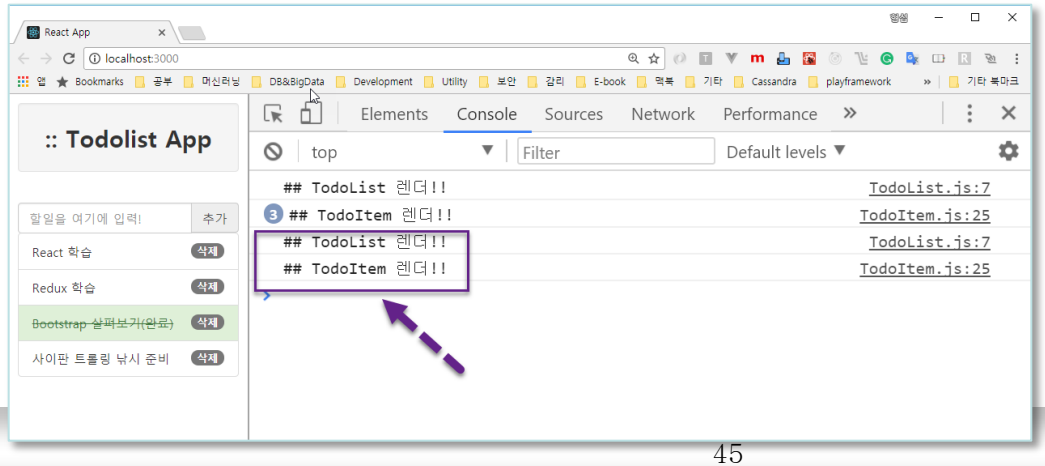


7. TodoList 앱 예제(3)

- shouldComponentUpdate() 메서드로 최적화
  - TodoItem 컴포넌트에 shouldComponentUpdate 메서드 작성

```
……
class TodoItem extends Component {
    ……
    shouldComponentUpdate(nextProps, nextState) {
        return nextProps.no !== this.props.no || nextProps.done !== this.props.done;
    }
    ……
}
……
```

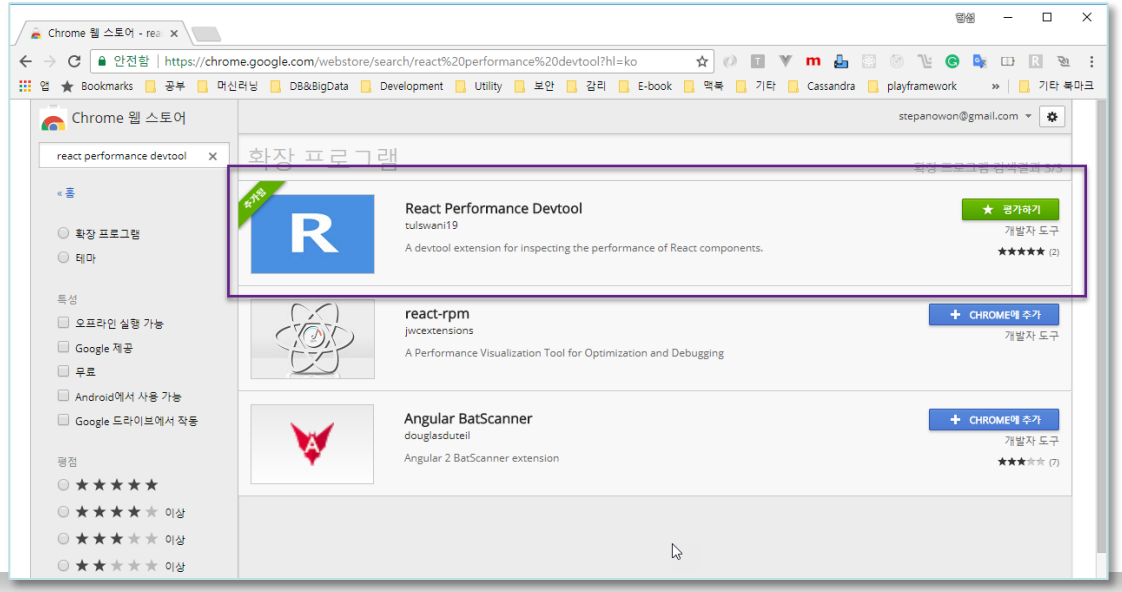
- 재실행 결과



8. React Performance Devtool(1)

React Performance Devtool

- 개념
  - React 컴포넌트의 성능을 조사하기 위한 브라우저 확장 도구
- 사용 방법
  - Chrome 확장 프로그램 + 약간의 코드
- 크롬 웹스토어에서 확장 프로그램 추가



8. React Performance Devtool(2)

- 약간의 코드
  - react-perf-devtool 패키지를 추가함.
    - yarn add -D react-perf-devtool
    - npm install --save-dev react-perf-devtool
  - src/index.js 에 코드 추가

```
……
import registerObserver from 'react-perf-devtool'

registerObserver();
ReactDOM.render(<App />, document.getElementById('root'));
registerServiceWorker();
```

성능 측정을 위한 앱 작성

- 프로젝트 초기화
  - create-react-app perftest
  - cd perftest
  - yarn add immutability-helper
  - yarn add -D react-perf-devtool

8. React Performance Devtool(3)

- src/App.js

```
import React, { Component } from 'react';
import List from './List';
import update from 'immutability-helper';

class App extends Component {
    constructor(props) {
        super(props);
        this.state = {  items : [ ] };
        this.no = 0;
    }
    componentDidMount() {
        this.handleId = setInterval(()=> {
            this.addItem()
        }, 50)
    }
    componentWillUnmount() {
        clearInterval(this.handleId);
    }
    addItem() {
        this.no++;
        let newItems = update(this.state.items, {
            $push : [ { no:this.no, name:'아이템 '+this.no } ]
        })
        this.setState({ items : newItems })
        if (this.no % 200 === 0)
             this.setState({ items : [] })
    }
    render() {
        let styles = { border:"solid 1px gray",
             float:"left", width:"33%" };
        return (
            <div>
                <div style={styles}>
                    <h2>#1</h2>
                    <List items={this.state.items} />
                </div>
                <div style={styles}>
                    <h2>#2</h2>
                    <List items={this.state.items} />
                </div>
                <div style={styles}>
                    <h2>#3</h2>
                    <List items={this.state.items} />
                </div>
            </div>
        );
    }
}

export default App;
```

8. React Performance Devtool(4)

- src/List.js

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

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

        return (
            <div>
                <ul>{listitems}</ul>
            </div>
        );
    }
}

List.propTypes = {
    items : PropTypes.arrayOf(Object)
};

export default List;
```

8. React Performance Devtool(5)

- src/ListItem.js

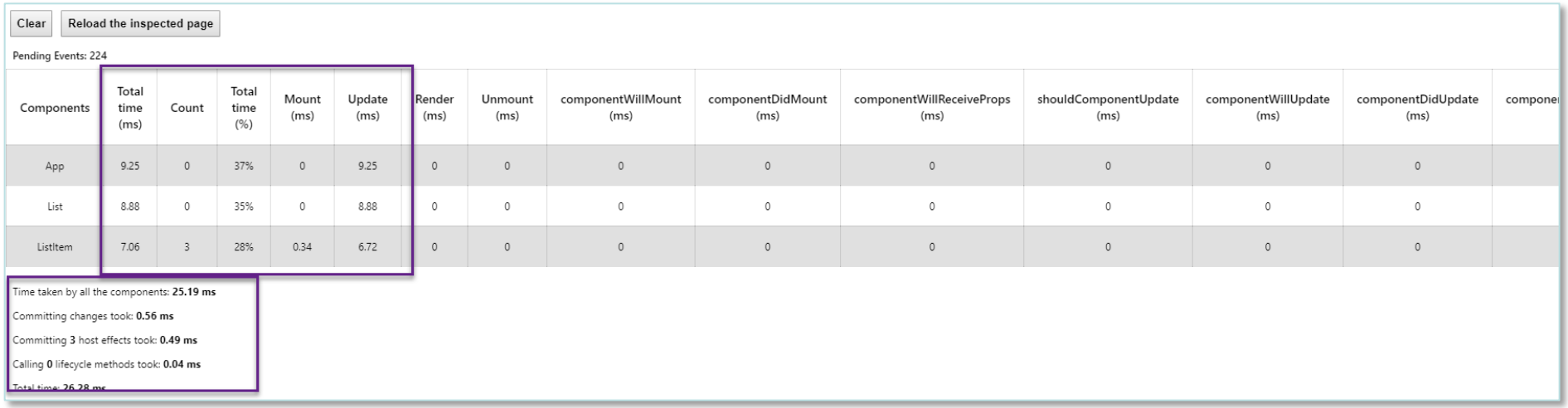
```
import React, { Component } from 'react';
import PropTypes from 'prop-types';

class ListItem extends Component {
    render() {
        return (
            <li>{this.props.no} : {this.props.name}</li>
        );
    }
}
ListItem.propTypes = {
    no : PropTypes.number.isRequired,
    name : PropTypes.string.isRequired
};
export default ListItem;
```

- src/index.js 변경

```
……(생략)
import registerObserver from 'react-perf-devtool'

registerObserver();
ReactDOM.render(<App />, document.getElementById('root'));
registerServiceWorker();
```

8. React Performance Devtool(6)

- 최적화 하지 않은 상태로 실행
  - 실행 후 브라우저의 개발자 도구에서 React Performance Devtool 탭을 열어서 확인
  - Reload ther instpected Page 버튼을 클릭하여 소요 시간 측정
  - 결과
    - ListItem 컴포넌트를 렌더링하는데 상당한 시간이 소요되었음
    - Update가 일어날 필요가 없는 경우임에도 update 를 수행하고 있음.



8. React Performance Devtool(7)

- ListItem 컴포넌트 수정
  - shouldComponentUpdate 를 이용해 최적화 후 실행

```
……
class ListItem extends Component {
    shouldComponentUpdate(nextProps, nextState) {
        return this.props.no !== nextProps.no || this.props.name !== nextProps.name;
    }
    ……
}
……
```

Clear | Reload the inspected page

Pending Events: 104

Components	Total time (ms)	Count	Total time (%)	Mount (ms)	Update (ms)	Render (ms)	Unmount (ms)	componentWillMount (ms)	componentDidMount (ms)	componentWillReceiveProps (ms)	shouldComponentUpdate (ms)	componentWillUpdate (ms)	componentDidUpdate (ms)	componen
App	4.55	0	48%	0	4.55	0	0	0	0	0	0	0	0	
List	4.27	0	45%	0	4.27	0	0	0	0	0	0	0	0	
ListItem	0.58	3	6%	0.33	0	0	0	0	0	0	0.25	0	0	

Time taken by all the components: **9.40 ms**
Committing changes took: **0.44 ms**
Committing **3** host effects took: **0.38 ms**
Calling **0** lifecycle methods took: **0.04 ms**
Total time: **10.26 ms**