



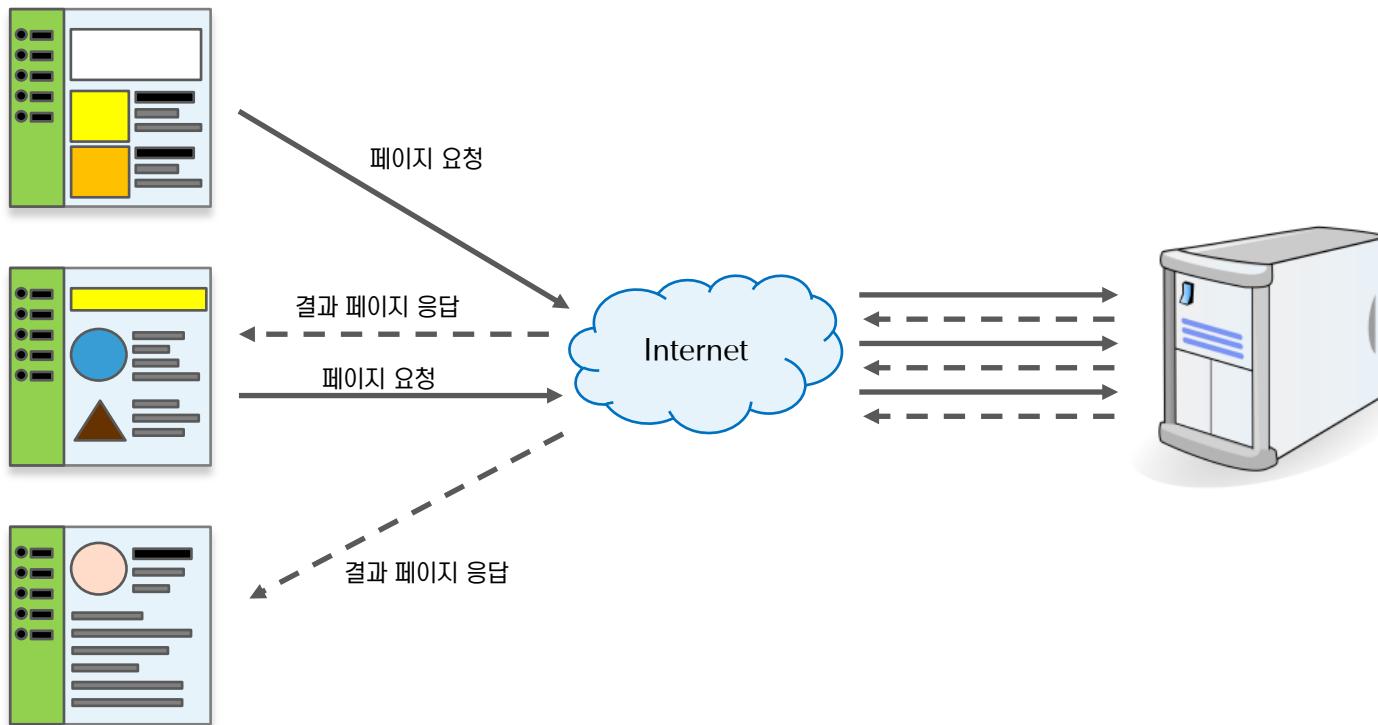
# 1. Intro to React

---

- 1.1 MPA vs SPA
- 1.2 React 소개
- 1.3 React Application 개발 구성 요소
- 1.4 개발 환경 구성

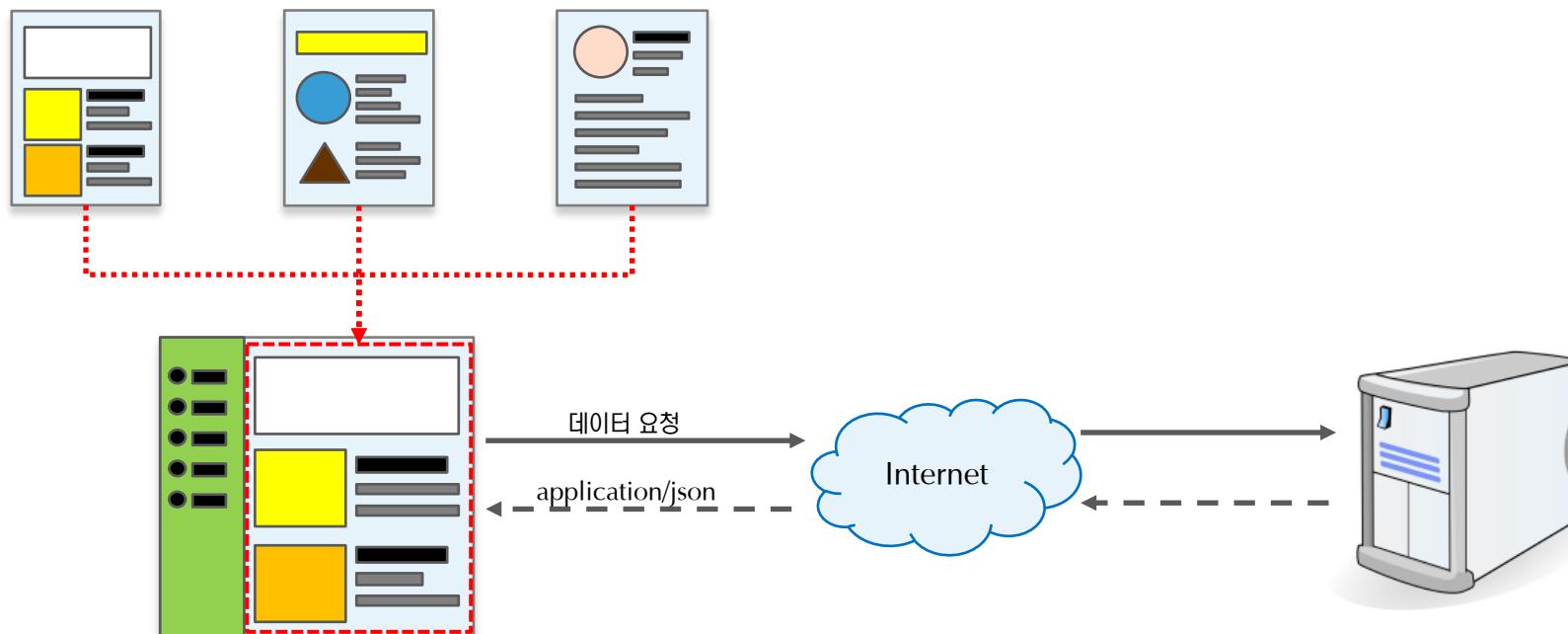
## 1.1 MPA vs SPA[1/2]

- ✓ MPA(Multi-Page Application)는 클라이언트의 요청에 따라 서버에서 페이지를 생성하고 이를 반환하는 형태입니다.
- ✓ 클라이언트의 입장에서는 서버에 요청하는 모든 페이지가 존재하기 때문에 사용자 요구에 따라 요청만 진행합니다.
- ✓ MPA 방식에서 서버는 모든 클라이언트들의 요청에 각각 대응하며 모든 페이지를 생성하고 응답해야 합니다.
- ✓ MPA 방식은 페이지의 작은 변화에도 서버에 전체 페이지를 요청하고 화면을 갱신합니다.



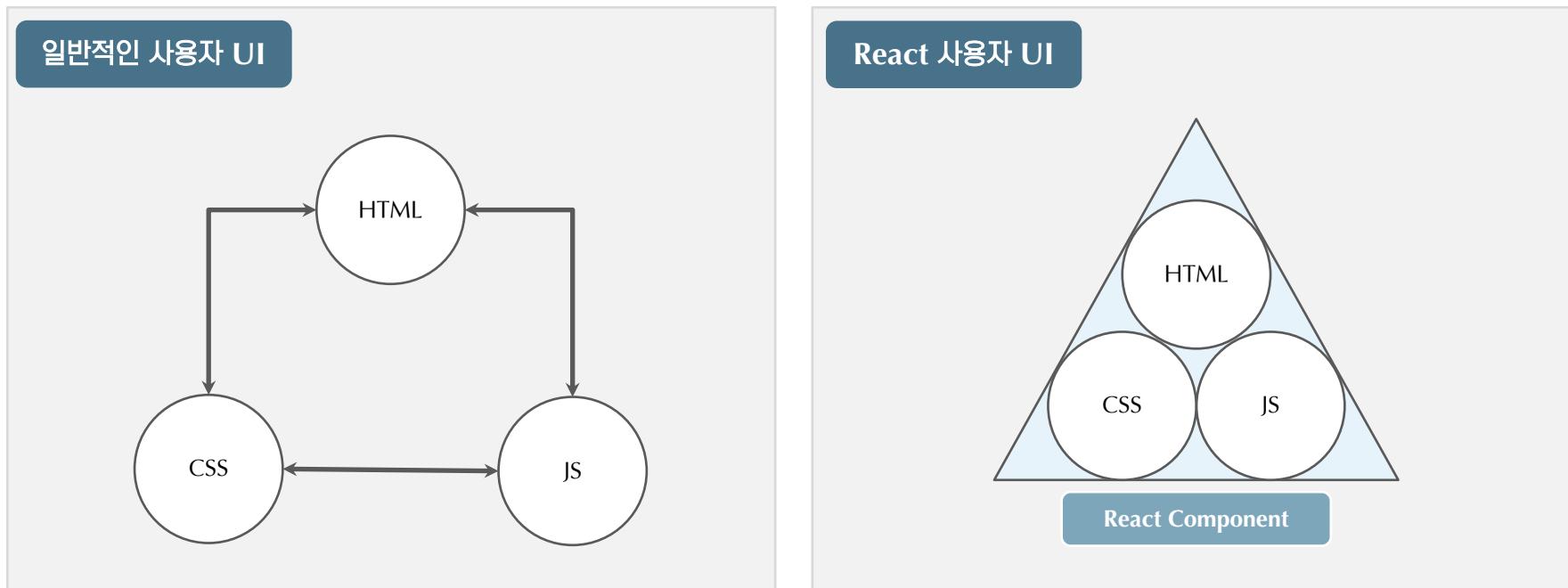
## 1.1 MPA vs SPA[2/2]

- ✓ SPA(Single-Page Application)는 서버로 부터 단일 페이지를 응답받고 클라이언트에서 화면을 구성하는 방식입니다.
- ✓ 클라이언트는 서버에 데이터(JSON)를 요청하고 응답 받은 데이터를 이용해 화면을 갱신합니다.
- ✓ 서버의 입장에서 다수의 클라이언트로부터 요청이 발생해도 페이지를 생성하지 않고 요청 데이터만 반환하기 때문에 부하가 적습니다.



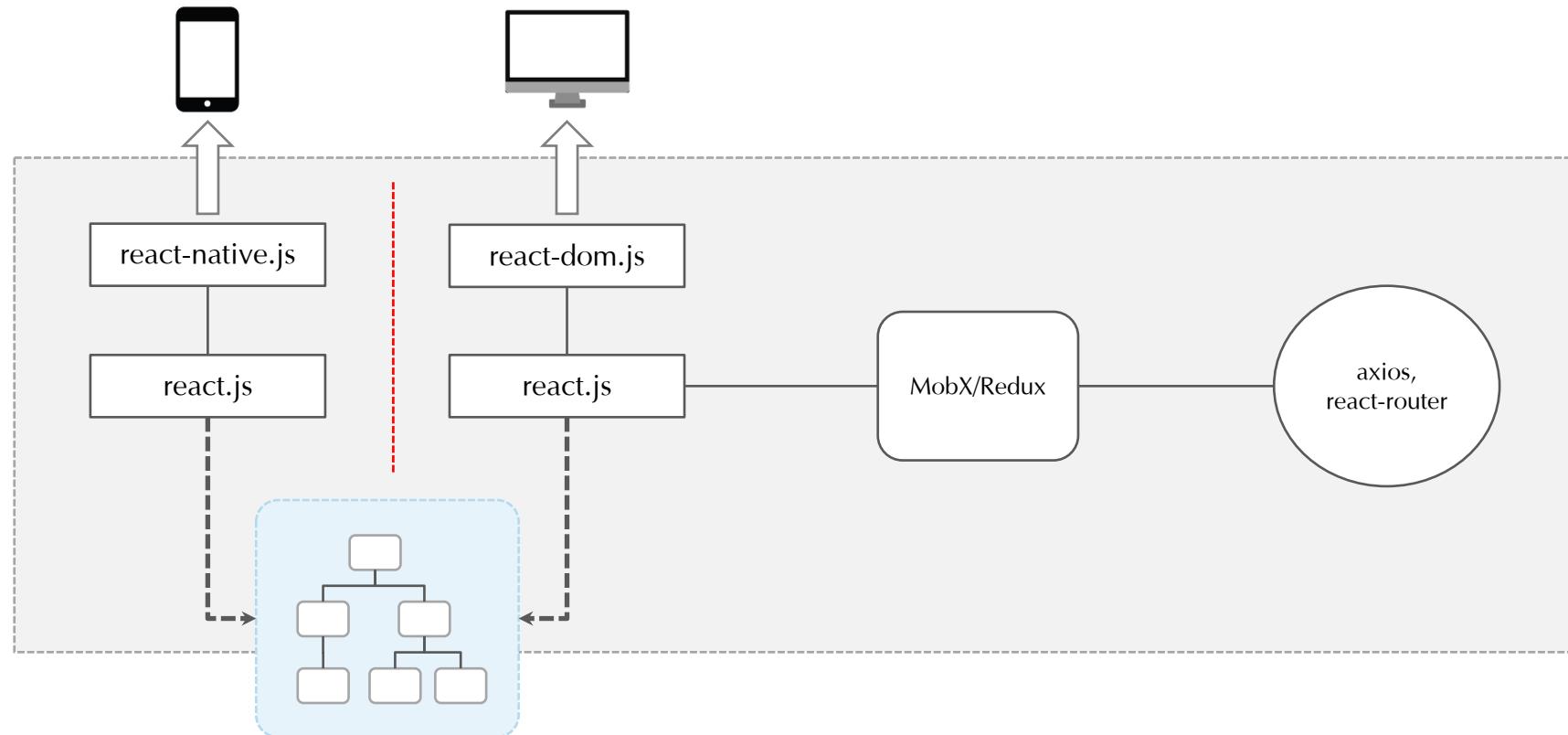
## 1.2 React 소개

- ✓ React는 SPA 기반 웹 사용자 UI(User Interface) 구성을 위한 JavaScript 라이브러리(Library)입니다.
- ✓ React는 독립적이며 재사용이 가능한 UI 컴포넌트를 손쉬운 방법으로 생성할 수 있도록 합니다.
- ✓ 컴포넌트를 만든다는 것은 기존에 구분해서 관리하던 HTML, CSS, JS를 하나의 요소로 묶는 것을 의미합니다.
- ✓ React UI를 구성하는 컴포넌트를 만드는 것의 핵심은 구성하고자 하는 화면을 어떻게 컴포넌트로 분리하느냐 입니다.



## 1.3 React Application 개발 구성 요소

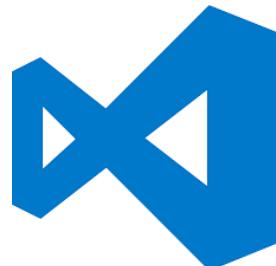
- ✓ React 어플리케이션을 구성하는 기본 단위는 캡슐화 된 컴포넌트 입니다.
- ✓ React 컴포넌트를 정의하고 활용하기 위해 사용하는 라이브러리는 react.js 코어 라이브러리 입니다.
- ✓ react-dom.js, react-native.js 라이브러리는 UI 요소들을 화면에 렌더링하는 라이브러로 react.js와 함께 사용합니다.
- ✓ React는 UI 요소들을 화면에 표현하는 것에만 집중한 라이브러리 입니다. 따라서, 다양한 종류의 Third-party 라이브러리와 함께 어플리케이션을 구성합니다.



## 1.4 개발 환경 구축 (1/5)

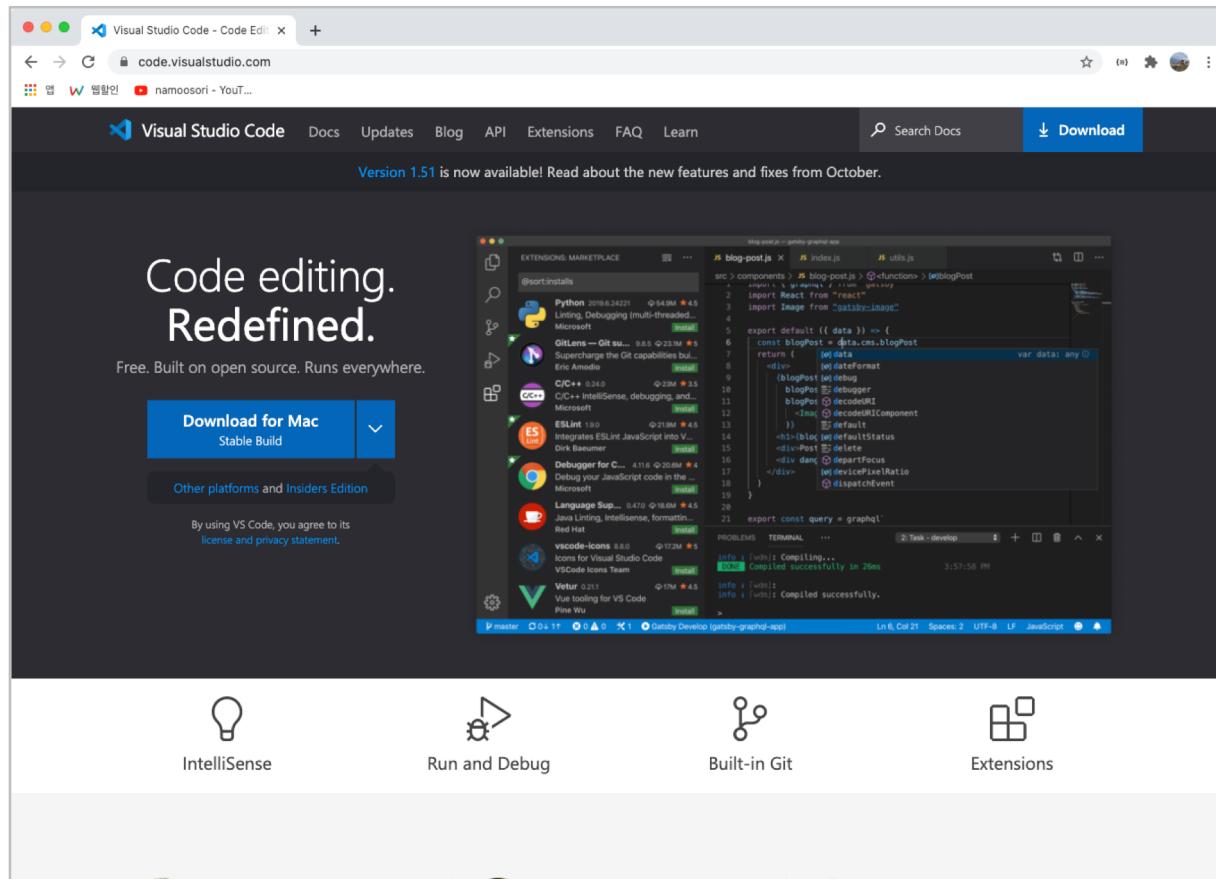
✓ React 개발에는 다음의 환경이 필요합니다.

- Code Editor: 실제 코드를 작성하기 위한 편집 도구입니다. 대표적인 도구로는 Atom, Bracket, VS code(Visual Studio Code) 등이 있습니다.
- node.js & npm: node.js는 JavaScript가 동작 할 수 있는 실행 환경입니다. npm(node package manager)은 다양한 JavaScript 패키지를 관리하기 위한 도구이며 패키지는 node.js에서 사용 가능한 모듈들의 집합을 뜻합니다.
- Web Browser: React를 이용한 UI 개발에 적합한 일반적인 웹브라우저는 일반적으로 Google의 Chrome을 사용합니다.
- git: 특정 패키지나 모듈의 경우 git 설치를 필요로 합니다.



# 1.4 개발 환경 구축 (2/5)

- ✓ Visual Studio Code는 MS에서 개발한 코드 편집 도구입니다.
- ✓ 다음의 URL을 통해 해당 사이트에서 개발 컴퓨터에 적합한 버전을 다운로드 합니다.
  - <https://code.visualstudio.com>

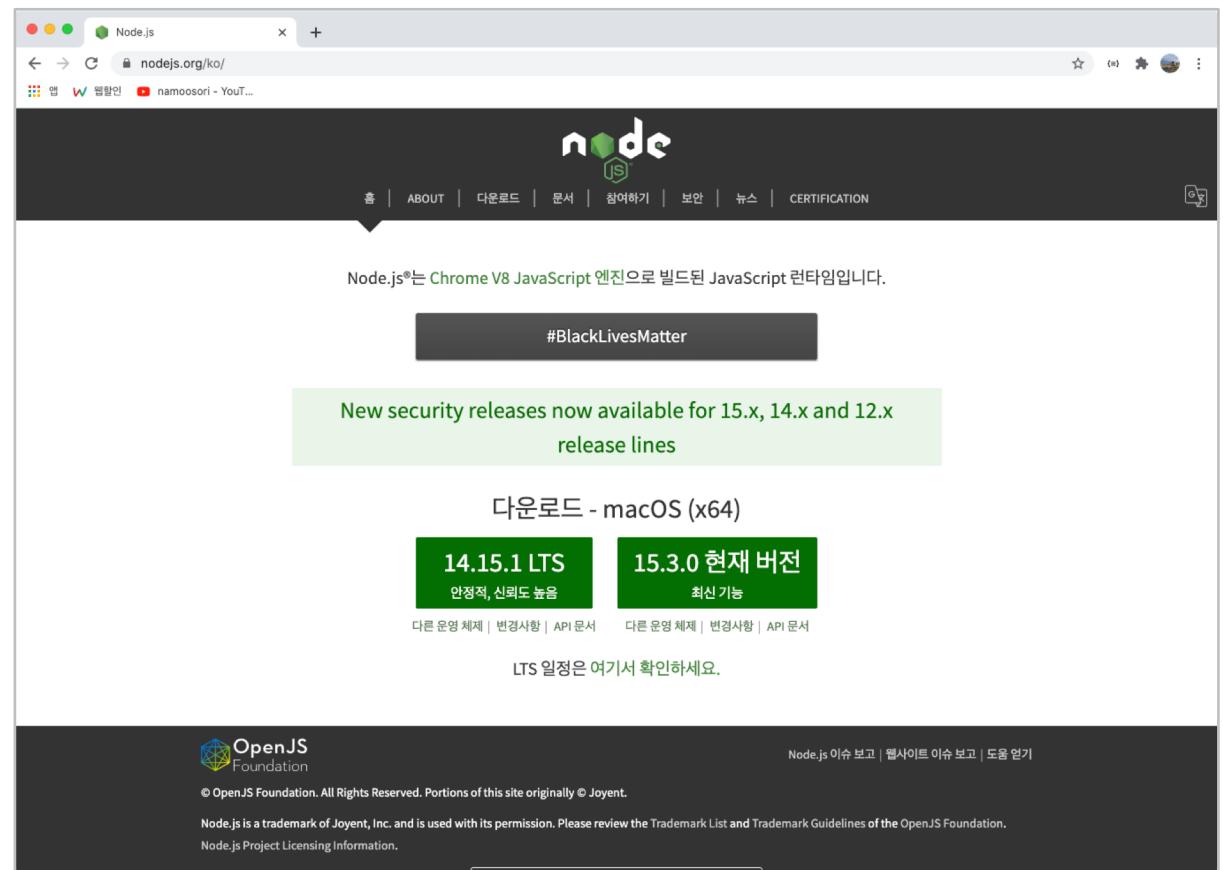
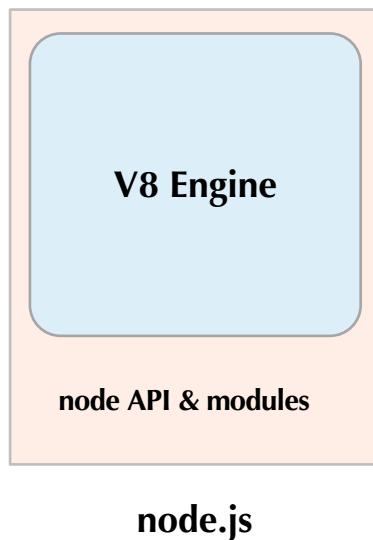


The screenshot shows the Visual Studio Code interface with a React project named 'react\_workspace'. The Explorer sidebar on the left lists files and folders such as 'App.js', 'index.js', 'utils.js', 'components', 'node\_modules', and 'public'. The 'App.js' file is open in the editor, displaying React code for a blog post application. The code includes imports for React, Semantic UI React, and GraphQL, and defines a functional component 'App' that renders a search bar and employee list. The bottom status bar indicates the file is in 'master' branch, has 0 changes, and is initializing JS/TS language features.

```
JS App.js
emp_exam > src > JS App.js > ...
1 import React, { Component } from 'react';
2 import { Grid, Segment } from 'semantic-ui-react';
3
4 import SearchBar from './components/SearchBar';
5 import EmployeeList from './components/EmployeeList';
6 import EmployeeDetail from './components/EmployeeDetail';
7
8 import Employees from './components/Employees';
9
10 class App extends Component {
11
12   constructor(props) {
13     super(props);
14
15     this.state = {
16       employees: Employees,
17       selectedEmployee: Employees[0]
18     }
19
20   }
21
22   searchByName(name) {
23     let updateList = Employees;
24     updateList = updateList.filter(employee => {
25       return employee.name.toLowerCase().search(name) !== -1;
26     });
27     this.setState(
28       { employees: updateList }
29     )
30   }
31
32   render() {
33     return (
34       <div className='App'>
35         <Segment>
36           <SearchBar onSearchByName={this.searchByName.bind(this)} />
37         </Segment>
38         <Grid columns={2} stackable>
39           <Grid.Column>
40             <Segment>
41               <EmployeeList
42                 onEmployeeSelect={selectedEmployee => this.setState({ selectedEmp
43                   employee=selectedEmployee ) />
44               </Segment>
45             </Grid.Column>
46             <Grid.Column>
47               <EmployeeDetail employee={this.state.selectedEmployee} />
48             </Grid.Column>
49           </Grid>
50         </Grid>
51       </div>
52     );
53   }
54 }
55
56 export default App;
```

## 1.4 개발 환경 구축 (3/5)

- ✓ JavaScript는 기본적으로 Web 기반의 언어로 개발되었습니다. 따라서, 실행 범위가 Web Browser로 국한되었습니다.
- ✓ node.js는 V8 엔진 기반의 JavaScript 실행 환경을 제공합니다. 즉, Web Browser 이외의 환경에서도 JavaScript를 실행할 수 있습니다.
- ✓ 다음의 URL을 통해 개발 환경에 맞는 node.js를 설치합니다.
  - <https://nodejs.org/>



## 1.4 개발 환경 구축 (4/5)

- ✓ npm(Node Package Module)은 JavaScript 패키지와 모듈(modules)을 관리하는 도구입니다.
- ✓ npm은 node.js를 설치하면 기본적으로 설치 됩니다. 설치의 확인은 다음 명령을 통해 확인할 수 있습니다.
  - node -v : node 버전 확인.
  - npm -v : npm 버전 확인.
  - yarn -v : yarn 버전 확인.

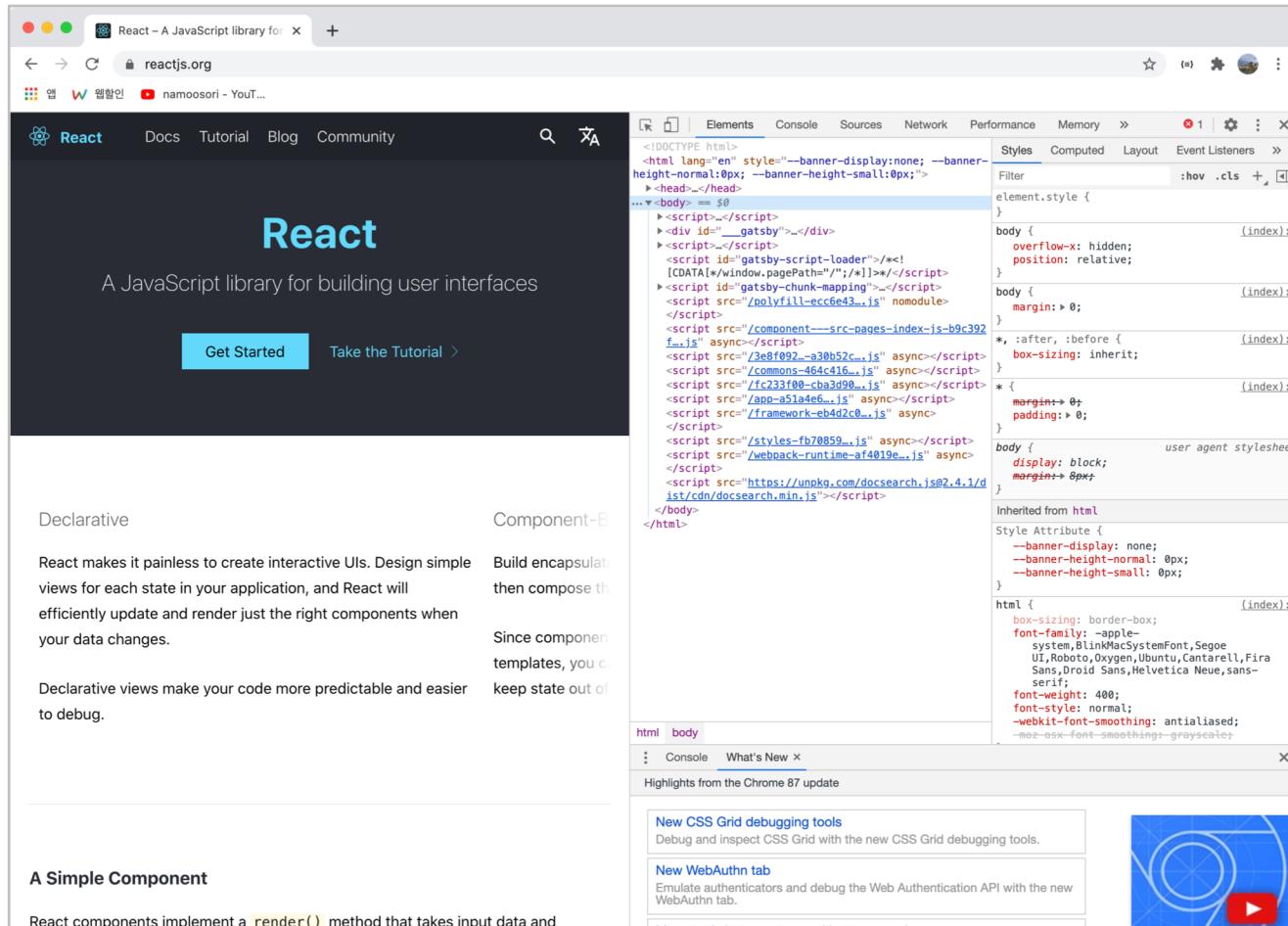


A screenshot of a macOS terminal window titled "kwon-kijin — kwon-kijin@Kwon-Kiui-MacBook-Pro — ~ — zsh — 80x20". The window shows the following command history:

```
[~] ~ node -v
v10.16.3
[~] ~ npm -v
6.9.0
[~] ~ yarn -v
1.19.0
[~]
```

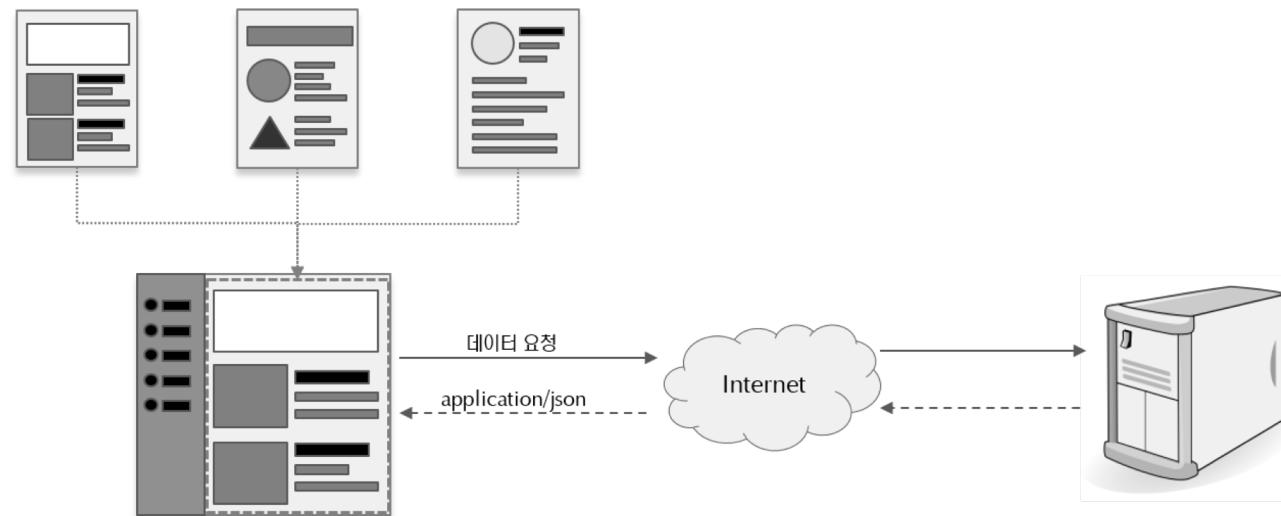
# 1.4 개발 환경 구축 (5/5)

- ✓ 웹브라우저는 IE, Chrome, Firefox 등 별도의 설치 없이도 사용 가능합니다.
- ✓ 개발에 사용할 웹 브라우저는 반드시 개발자 도구를 사용할 수 있어야 합니다.
- ✓ 대부분의 웹브라우저는 개발자 도구를 지원하고 있습니다.



# 요약

- ✓ React는 JavaScript로 만들어진 UI 컴포넌트 라이브러리이며 독립적인 이 컴포넌트들을 이용해 UI를 구성합니다.
- ✓ MPA 방식은 클라이언트의 요청이 있을 때 서버로 부터 새 페이지에 대한 전체 정보를 받아 표현합니다.
- ✓ SPA는 첫 페이지를 제외하고 서버로부터 필요한 데이터만 받습니다.





## 2. React Basic

---

2.1 React 동작 방식

2.2 React Element

2.3 Virtual DOM

2.4 JSX

## 2.1 React 동작 방식

- ✓ 기존 웹 애플리케이션은 클라이언트의 요청이 있으면 서버에서 UI(화면 구성 + 데이터)를 생성하여 응답합니다.
- ✓ React는 서버로부터 필요한 데이터를 요청하고 서버로부터 응답 받은 데이터를 이용해 사용자 UI를 구성합니다.
- ✓ 서버를 통해서는 필요 데이터만 받고, 클라이언트는 사용자 UI 전체를 관리합니다.
- ✓ 클라이언트에서 사용자 UI를 표현하는 것을 Client Side Rendering(CSR)이라고 합니다.



## 2.2 React Element

- ✓ React Element는 브라우저 DOM 엘리먼트에 대한 정보를 담고 있는 객체입니다.
- ✓ 실제 브라우저 DOM이 어떻게 만들어져야 하는지에 대한 설명을 가지고 있는 객체 리터럴입니다.
- ✓ 실제 DOM이 아닌 단순 객체이기 때문에 생성이 쉽습니다.
- ✓ ReactElement는 포함하는 값을 변경할 수 없으며 변경이 필요한 경우 새롭게 생성합니다.

```
let h1 = React.createElement('h1', null, 'Hello React!');  
// React Element 생성  
ReactDOM.render(  
  h1,  
  document.getElementById('root')  
);  
// 브라우저 DOM에 랜더링  
console.log(h1);  
// React Element 정보 출력
```

Hello React!

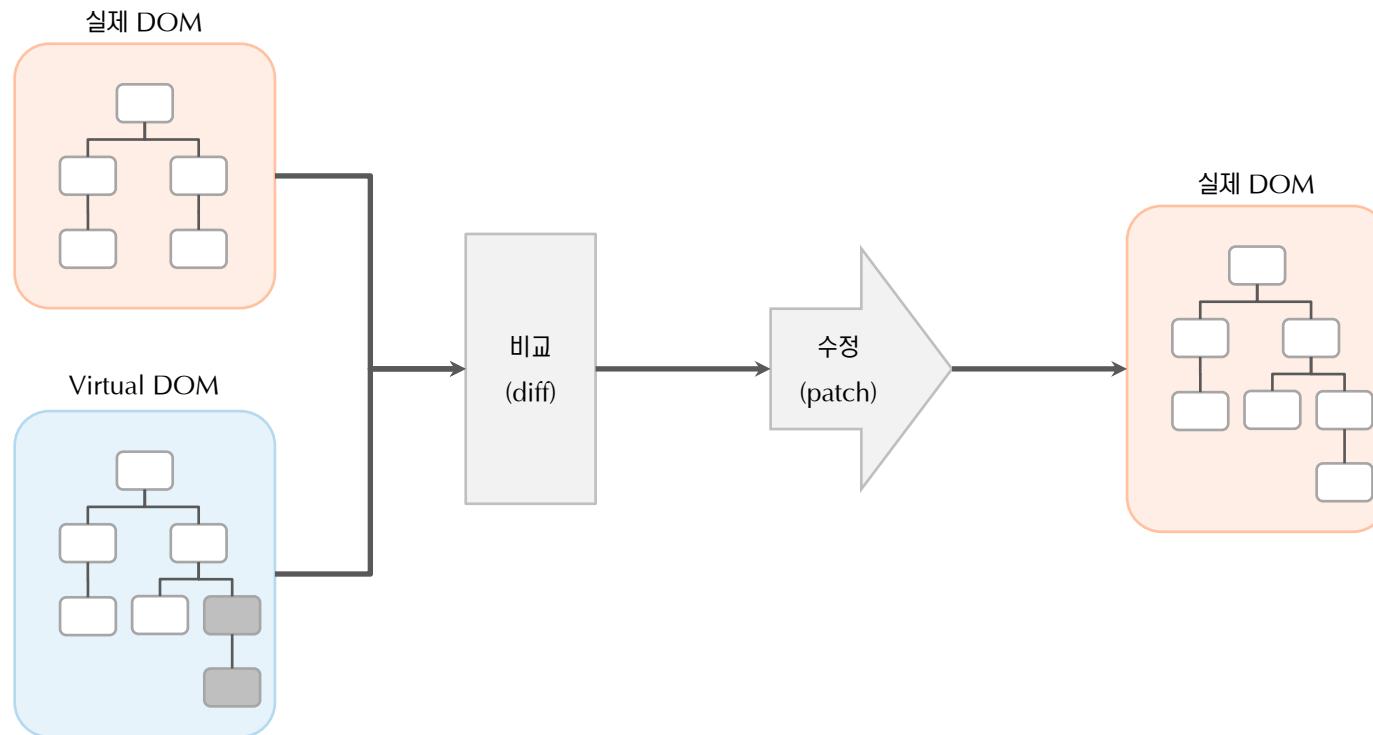
```
function tick() {  
  let element = React.createElement('h1', null, new Date().toLocaleTimeString());  
  ReactDOM.render(element, document.getElementById('root'));  
}  
setInterval(tick, 1000);
```

오후 1:45:54

```
1. {$$typeof: Symbol(react.element), type: "h1",  
key: null, ref: null, props: {...}, ...}  
1. $$typeof: Symbol(react.element)  
2. key: null  
3. props: {children: "Hello React!"}  
4. ref: null  
5. type: "h1"  
6. _owner: null  
7. _store: {validated: false}  
8. _self: null  
9. _source: null  
10. __proto__: Object
```

## 2.3 Virtual DOM

- ✓ Virtual DOM은 브라우저의 실제 DOM을 반영하고 있는 데이터의 모음으로 ReactElement로 구성됩니다.
- ✓ 웹 브라우저에서 화면의 변경은 곧 실제 DOM 구조의 변화를 뜻하며 기존 방식은 직접 접근을 통한 변경 방식입니다.
- ✓ React는 Virtual DOM이라는 실제 DOM과 비교 가능한 데이터 구조를 통해 변경이 필요한 요소를 변경합니다.
- ✓ 실제 DOM과 Virtual DOM의 차이를 파악하고 변경하는 과정은 비교(diff)하고 수정(patch)을 거칩니다.

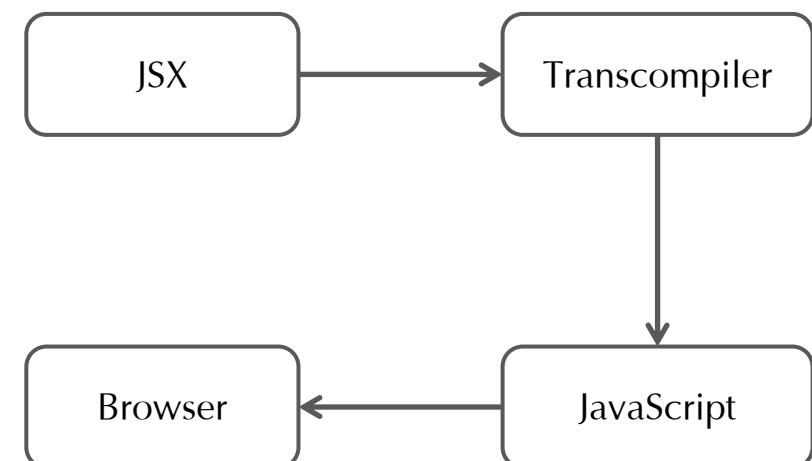


## 2.4 JSX (1/2)

- ✓ JSX(JavaScript XML)는 JavaScript의 확장으로 문법적 편의를 위해 만들어졌습니다.
- ✓ JSX로 작성한 코드는 트랜스컴파일러(Transcompiler)를 통해 JavaScript 코드로 변환 합니다.
- ✓ JSX를 이용하면 React.createElement() 함수 대신 HTML과 유사한 XML 형식의 코드를 작성할 수 있습니다.
- ✓ 중첩된 React 엘리먼트를 정의하려고 할 때 JavaScript 코드로 구현하는 것은 한계가 있으며 이런 경우 JSX를 이용하면 간결한 구현이 가능합니다.

```
//JSX
ReactDOM.render(
  <h1 className='heading'>Hello World</h1>,
  document.getElementById('content')
)
```

```
//JavaScript
ReactDOM.render(
  React.createElement(
    'h1',
    { className: 'heading' },
    'Hello World'
  )
)
```



## 2.4 JSX [2/2]

✓ JSX는 HTML과 유사한 형태이기 때문에 컴포넌트의 구조를 익숙한 방법으로 선언할 수 있습니다.

✓ JSX를 이용해 컴포넌트를 정의할 때 고려해야 할 몇가지 사항이 존재합니다.

- 대소문자 구분 : React 컴포넌트와 기본 HTML 요소와 대소문자로 구분하며, 컴포넌트는 Pascal Case로 작성합니다.  
예시) 컴포넌트-<List />, <Button/>      HTML 요소 : <h2 />, <input />, <a />
- {}의 사용 : JSX 코드상에서 JavaScript 코드를 반영하기 위해서는 {중괄호} 내부에 JavaScript 코드를 작성합니다.
- 예약어의 사용 : 표준 HTML의 사용에서 class, for 같은 예약어는 JSX에서 className, htmlFor를 대신하여 사용합니다.

```
class DateTimeNow extends React.Component {
  render(){
    let dateTimeNow = new Date().toLocaleDateString();
    return React.createElement(
      'span',
      null,
      '현재 시간 : ${dateTimeNow}'. // ECMAScript 6
    );
  }
}
```



```
class DateTimeNow extends React.Component {
  render(){
    let dateTimeNow = new Date().toLocaleDateString();
    return <span> 현재 시간 ${dateTimeNow}.</span>
  }
}
```

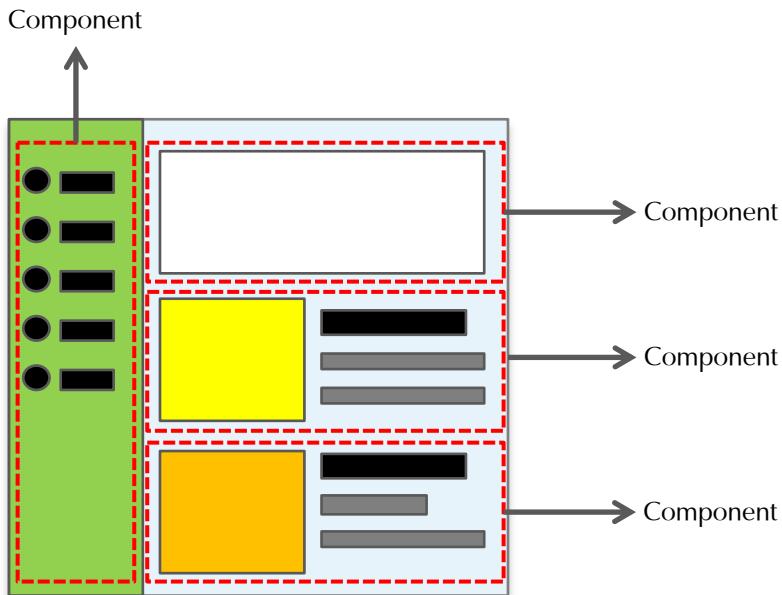


## 3. React Component

- 
- 3.1 React Component 개요
  - 3.2 Props 개요
  - 3.3 Props의 검증 - PropTypes
  - 3.4 Props의 적용

## 3.1 React Component 개요(1/2)

- ✓ 컴포넌트는 React UI 구성의 기본 단위이며, React는 컴포넌트를 생성하는 다양한 방법을 제공합니다.
- ✓ 캡슐화 된 컴포넌트는 재사용 및 재구성이 가능하며 컴포넌트들의 집합으로 사용자 UI를 구성합니다.
- ✓ React UI는 컴포넌트들의 집합으로 구성하며 컴포넌트는 클래스 기반의 컴포넌트와 함수 기반의 컴포넌트로 나뉩니다.
- ✓ 클래스 기반으로 생성하는 컴포넌트는 반드시 `render()` 메서드를 재정의 해야 하며 `render()` 메서드는 하나의 React Element를 반환합니다.



```
let element = React.createElement('h1', null, "Component Test");

class MyComponent extends React.Component {
  render(){
    return React.createElement('div', null, element, element);
  }
}

ReactDOM.render(
  React.createElement(MyComponent, null), document.getElementById('root')
)
```

Component Test  
Component Test

## 3.1 React Component 개요(2/2)

- ✓ 컴포넌트는 화면을 구성하는 다양한 요소(기능, 스타일, 마크업 등)를 그룹화 하는 방법입니다.
- ✓ 사용자 UI를 여러 부분(part)로 나누는 경계가 되며, 컴포넌트는 다른 컴포넌트를 포함할 수 있습니다.
- ✓ 컴포넌트는 독립적이면서도 재사용이 가능하기 때문에 필요한 기능들을 독자적으로 구성할 수 있습니다.
- ✓ 컴포넌트는 클래스 기반의 상태가 있는(stateful) 컴포넌트와 함수 기반 상태가 없는(stateless) 컴포넌트로 구분합니다.

```
class Customer extends React.Component {
  render() {
    const { id, name, orders } = this.props;

    return (
      <div className="customer">
        <h2>{id}</h2>
        <p>
          <span>이름 : {name}</span><br/>
          <span>주문 수량 : {orders.length} 개</span>
        </p>
      </div>
    )
  }
}
```

```
function App(){
  return <HelloMessage />
}

function HelloMessage(){
  const message = "Hello~~";
  return <h1> {message} </h1>
}

const App = () => {
  return <HelloMessage />
}

const HelloMessage = () => {
  const message = 'Hello~~';
  return <h1>{message}</h1>
}
```

## 3.2 Props 개요

- ✓ React.Component 클래스를 확장(extends)해 정의하는 클래스 기반의 컴포넌트는 props 객체를 갖습니다.
- ✓ props는 해당 컴포넌트의 생성 시점에 상위 컴포넌트로 부터 전달되는 데이터가 할당되는 객체입니다.
- ✓ props를 통해 상위 컴포넌트로부터 전달되는 데이터의 종류는 크게 2가지 형태입니다.
  - Html 요소의 속성 : href, title, style, class 등
  - 상위 컴포넌트로부터 할당 받는 임의의 데이터 : this.props.속성명
- ✓ props를 통해 전달 되는 데이터는 해당 컴포넌트에서 값을 변경할 수 없습니다.

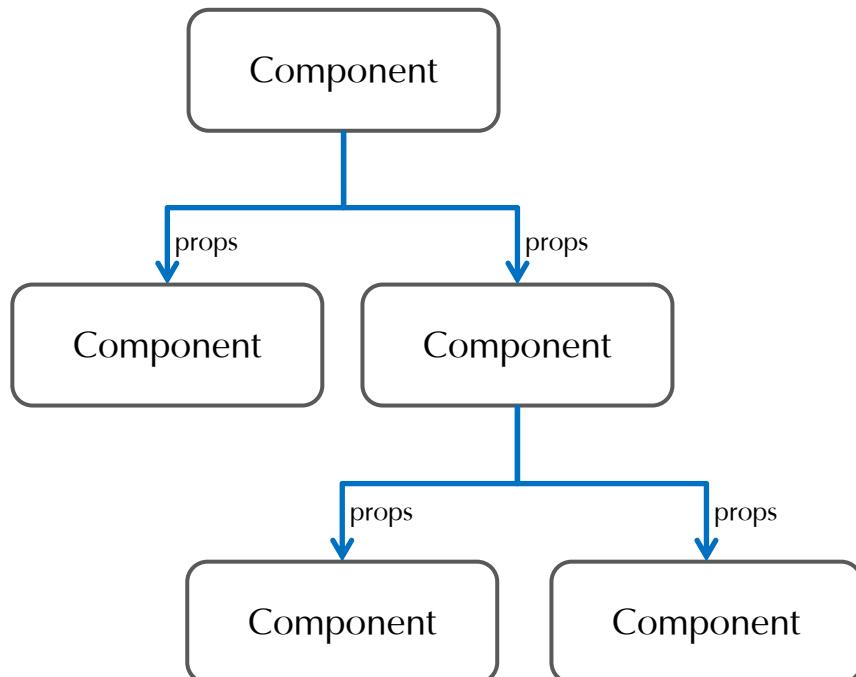
```
<Customer id='kim@namoosori.io' name='Kim' orders={['A0001', 'B0002', 'C0003']} />
```

```
class Customer extends React.Component {  
  render() {  
    const { id, name, orders } = this.props;  
    return (  
      <div className="customer">  
        <h2>{id}</h2>  
        <p>  
          <span>이름 : {name}</span><br/>  
          <span>주문 수량 : {orders.length} 개</span>  
        </p>  
      </div>  
    )  
  }  
}
```

kim@namoosori.io  
이름 : Kim  
주문 수량 : 3 개

### 3.3 Props의 검증 – PropTypes[1/2]

- ✓ JavaScript는 타입 검증이 느슨한 언어이며 이는 변수에 할당되는 데이터 타입의 제한이 느슨한(loosely) 것을 뜻합니다.
- ✓ Props로 전달 받는 데이터 타입의 검증은 데이터를 받는 해당 컴포넌트가 담당합니다.
- ✓ React.PropTypes는 상위 컴포넌트로 부터 전달받는 props의 타입에 대한 검증을 수행합니다.
- ✓ prop-types 라이브러리를 이용해서 해당 컴포넌트의 props에 할당된 데이터 유효성 검사를 진행합니다.



Type	Verification
Array	React.PropTypes.array
Boolean	React.PropTypes.bool
Function	React.PropTypes.func
Number	React.PropTypes.number
Object	React.PropTypes.object
String	React.PropTypes.string

### 3.3 Props의 검증 – PropTypes[2/2]

- ✓ PropTypes를 통해 상위 컴포넌트로부터 전달받는 props를 검증합니다.
- ✓ PropTypes는 props에 대한 타입 검증이며 타입이 다를 경우 개발자 콘솔(console)을 통해 경고가 표시됩니다.
- ✓ PropTypes를 반드시 적용해야 하는 것은 아니지만 버그의 예방과 디버깅을 위해 사용합니다.
- ✓ DefaultProps를 이용하면 props에 기본값을 설정하는 것도 가능합니다.

```
<Customer id='kim@namoosori.io' name='Kim' orders='A0001, B0002, C0003' />
```

```
class Customer extends Component {
  static propTypes = {
    id : PropTypes.string.isRequired,
    name : PropTypes.string.isRequired,
    orders : PropTypes.array.isRequired
  }

  render() {
    const { id, name, orders } = this.props;
    return (
      <div className="customer">
        <h2>{id}</h2>
        <p>
          <span>이름 : {name}</span><br/>
          <span>주문 수량 : {orders.length} 개</span>
        </p>
      </div>
    )
  }
}
```

kim@namoosori.io

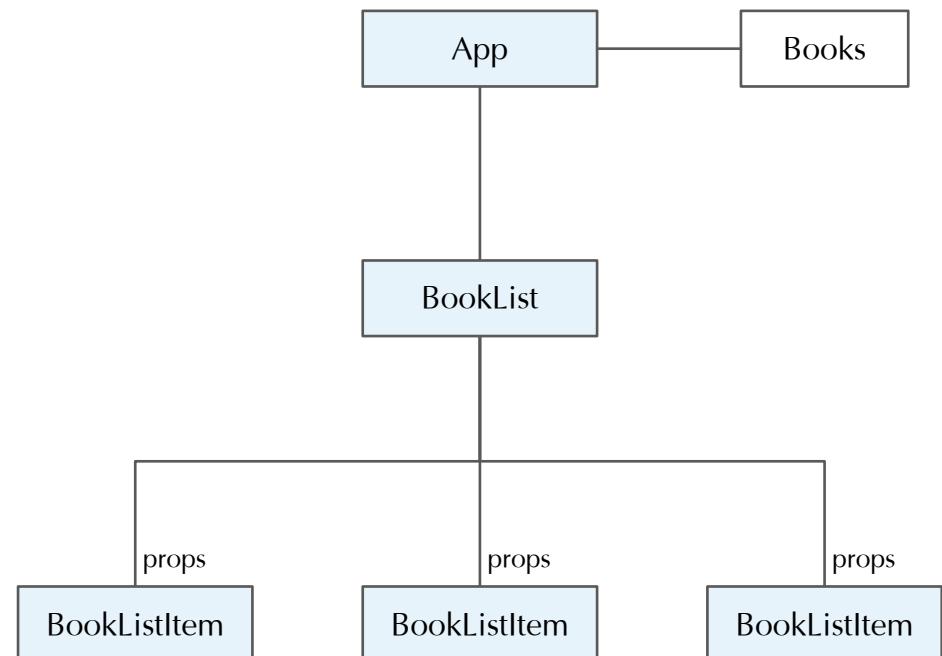
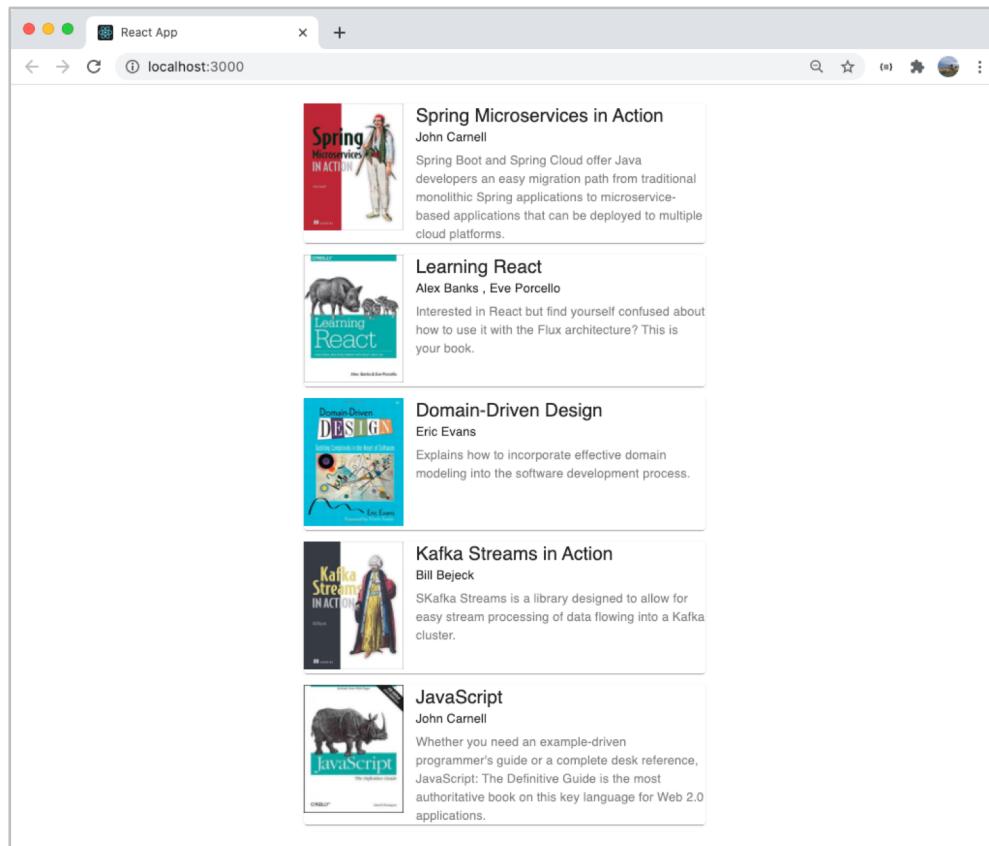
이름 : Kim

주문 수량 : 19 개

```
✖ ▶ Warning: Failed prop type: Invalid prop `orders` of
  type `string` supplied to `Customer`, expected `array`.
  in Customer (at App.js:14)
  in App (at src/index.js:7)
```

## 3.4 Props의 적용

- ✓ 다음은 props를 통해 포함 계층 구조에서 하위 컴포넌트에 값을 전달하는 예제입니다.
- ✓ App 컴포넌트는 Books.js를 통해 리스트에 표현할 값을 참조합니다.
- ✓ BookListItem 컴포넌트는 BookList로부터 props를 통해 표현할 데이터를 전달 받습니다.
- ✓ 컴포넌트의 표현을 위해 Material-UI React 를 사용합니다.





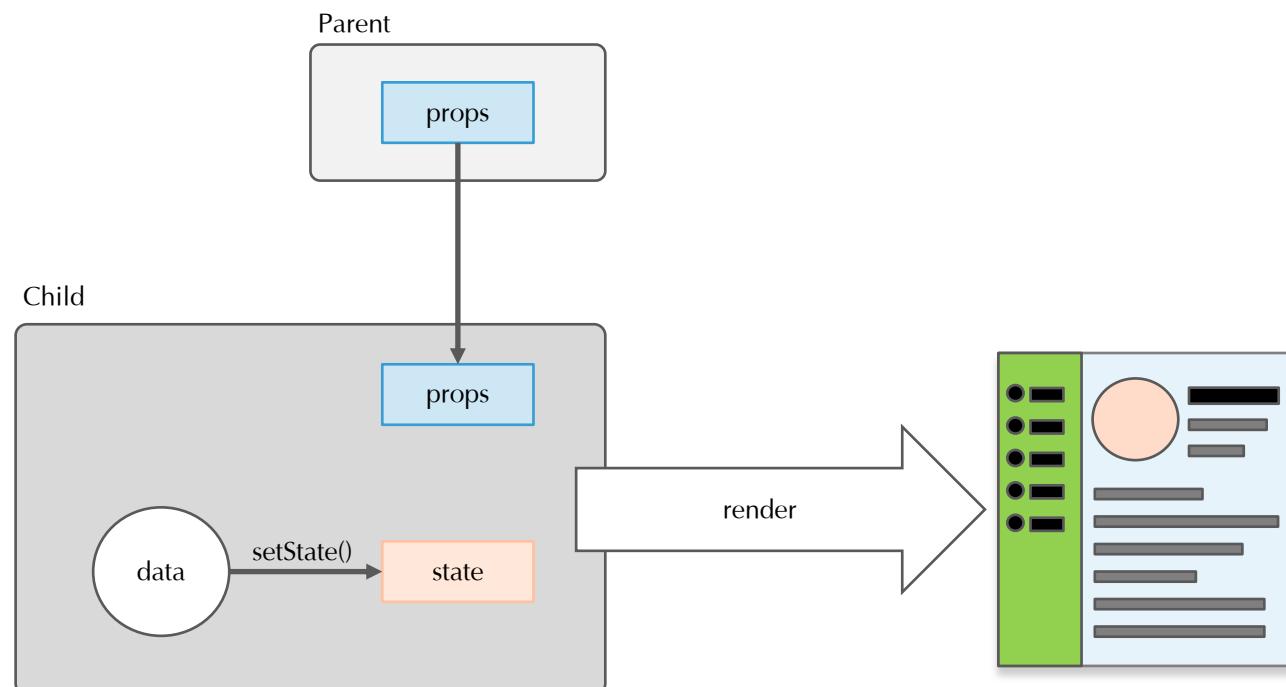
## 4. State

---

- 4.1 State 개요
- 4.2 State의 초기화와 변경
- 4.3 State의 적용

## 4.1 State 개요

- ✓ props는 해당 컴포넌트의 입장에서 불변(immutable) 데이터입니다.
- ✓ 컴포넌트에서 변경 가능한 데이터를 관리하기 위해 사용하는 객체는 state입니다.
- ✓ State는 React.Component 클래스를 상속한 클래스 기반의 컴포넌트에만 존재합니다.
- ✓ State 값의 초기화는 객체 필드의 선언부 혹은 생성자에서 구현하며, state 값의 변경은 setState() 메서드를 이용합니다.



## 4.2 State 초기화와 변경

- ✓ 생성자(constructor)는 클래스가 인스턴스화 될 때 한번 호출되는 특수한 메서드입니다.
- ✓ state 객체에 값을 대입(assign)하여 초기화 하는 것은 해당 컴포넌트 객체의 필드 선언부 혹은 생성자에서 진행합니다.
- ✓ state 초기화를 위해 생성자를 정의할 때에는 super 생성자를 호출하여 부모 클래스에 props를 전달합니다. 이는 this 참조를 사용하기 위한 코드입니다.
- ✓ state 객체에 값을 변경하기 위해서는 반드시 setState() 메서드를 호출하여 변경합니다.

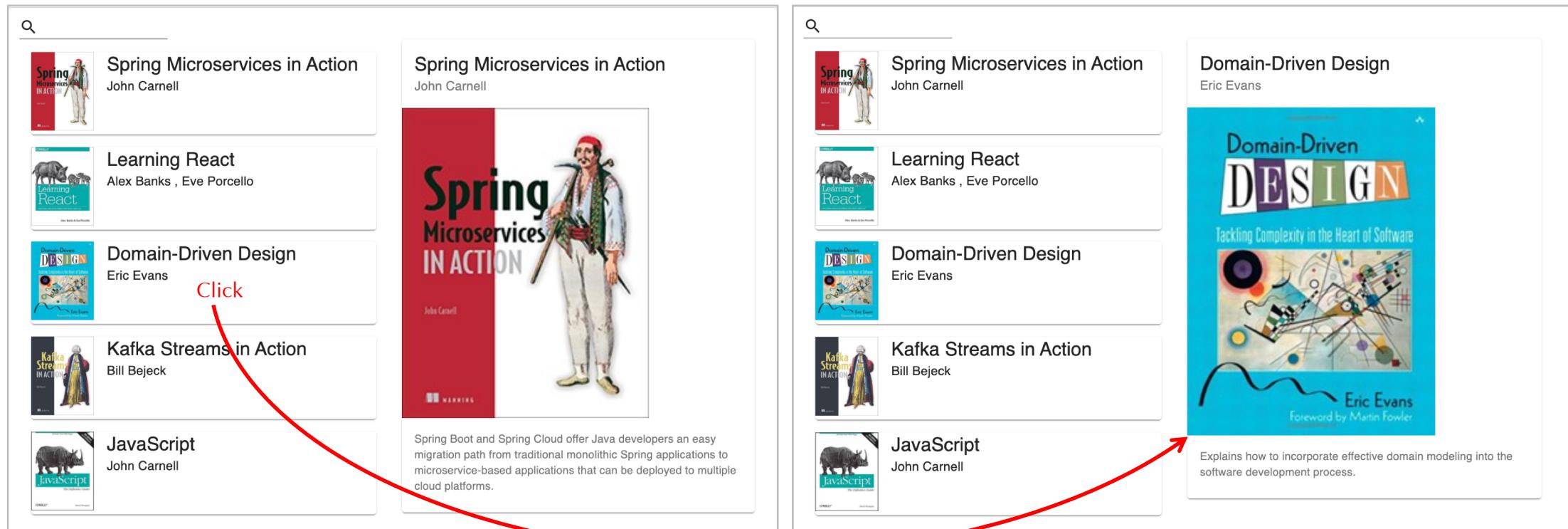
```
class StateApp extends Component {  
    constructor(props) {  
        super(props);  
        this.state = {  
            books: Books,  
            selectedBook: Books[0]  
        }  
    }  
    searchByTitle(title) {  
        let updateList = Books;  
        updateList = updateList.filter(book => {  
            return book.title.toLowerCase().search(title.toLowerCase()) !== -1;  
        });  
        this.setState({  
            books: updateList  
        })  
    }  
}
```

state 초기화

state 변경

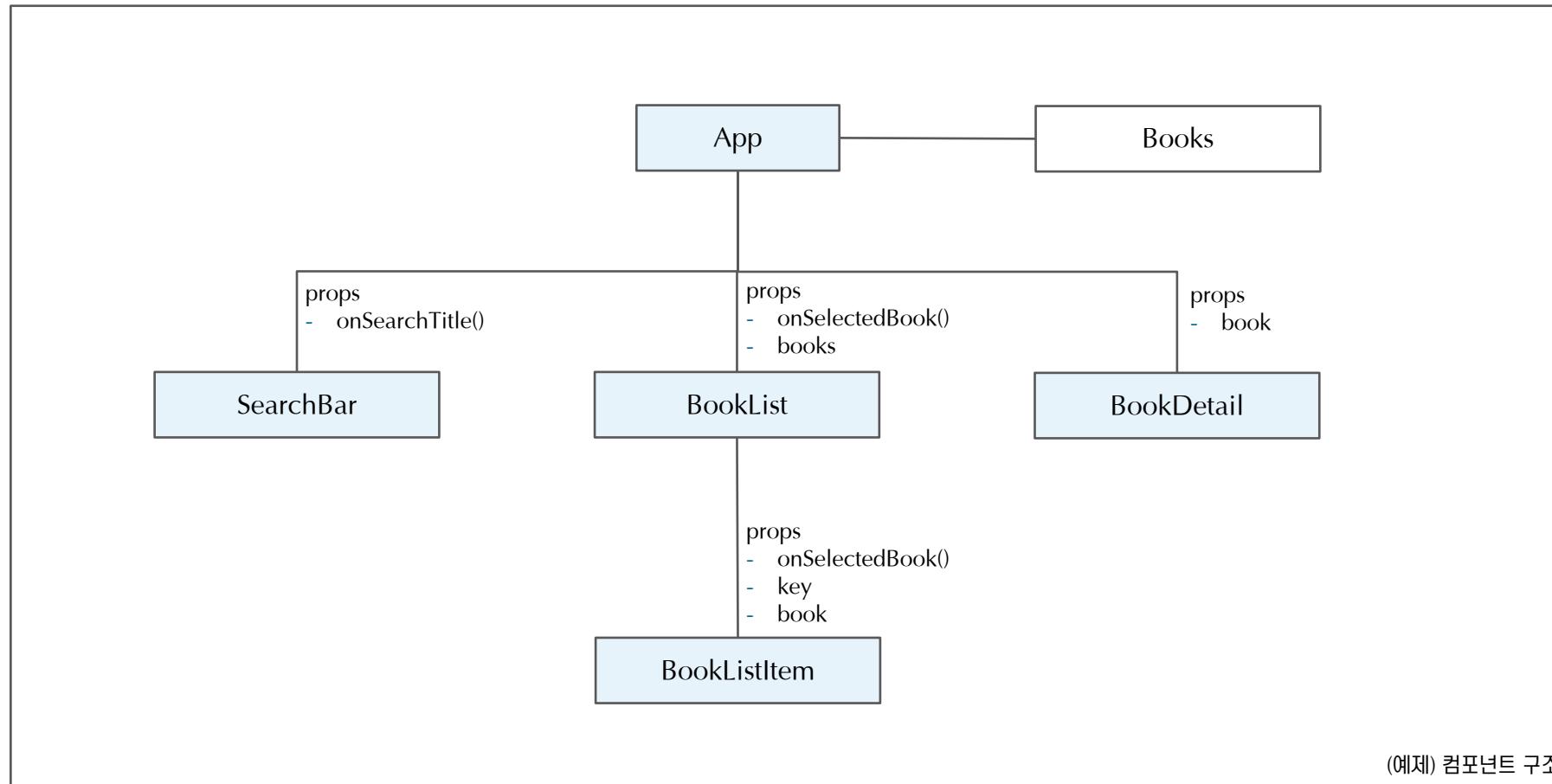
## 4.3 State의 적용 [1/2]

- ✓ state 객체는 해당 컴포넌트가 갖는 데이터를 관리하며, 이 데이터의 변경을 통해 화면을 갱신합니다.
- ✓ **React.Component** 클래스를 확장하는 모든 클래스 기반 컴포넌트는 state 객체를 갖습니다.
- ✓ 하나의 UI를 구성하는 다양한 컴포넌트가 각각 별도의 state를 갖고 관리하는 것은 프로그램의 복잡도를 올립니다.
- ✓ 루트(root)가 되는 컴포넌트가 전체 state를 갖게 하고 하위 컴포넌트들은 props를 통해 불변의 데이터를 받습니다.



## 4.3 State의 적용 [2/2]

- ✓ state는 JavaScript 객체인 만큼 다양한 종류의 데이터를 담을 수 있습니다.
- ✓ App 컴포넌트는 books 데이터를 가지며 state 객체를 통해 관리 합니다.
- ✓ SearchBar, BookList, BookDetail 컴포넌트는 App 컴포넌트로부터 props 객체 데이터로 책의 정보를 받습니다.





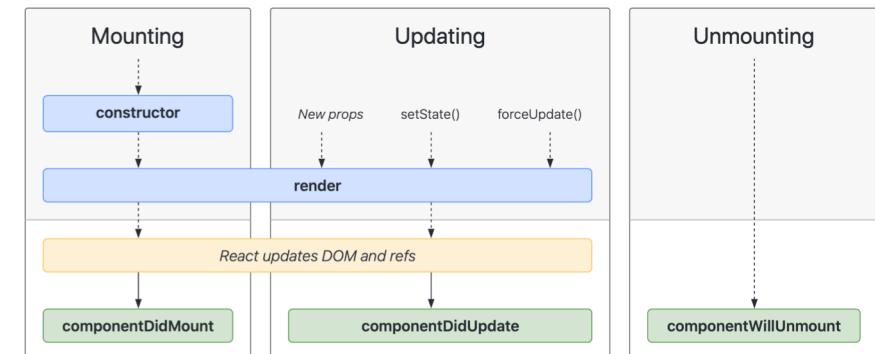
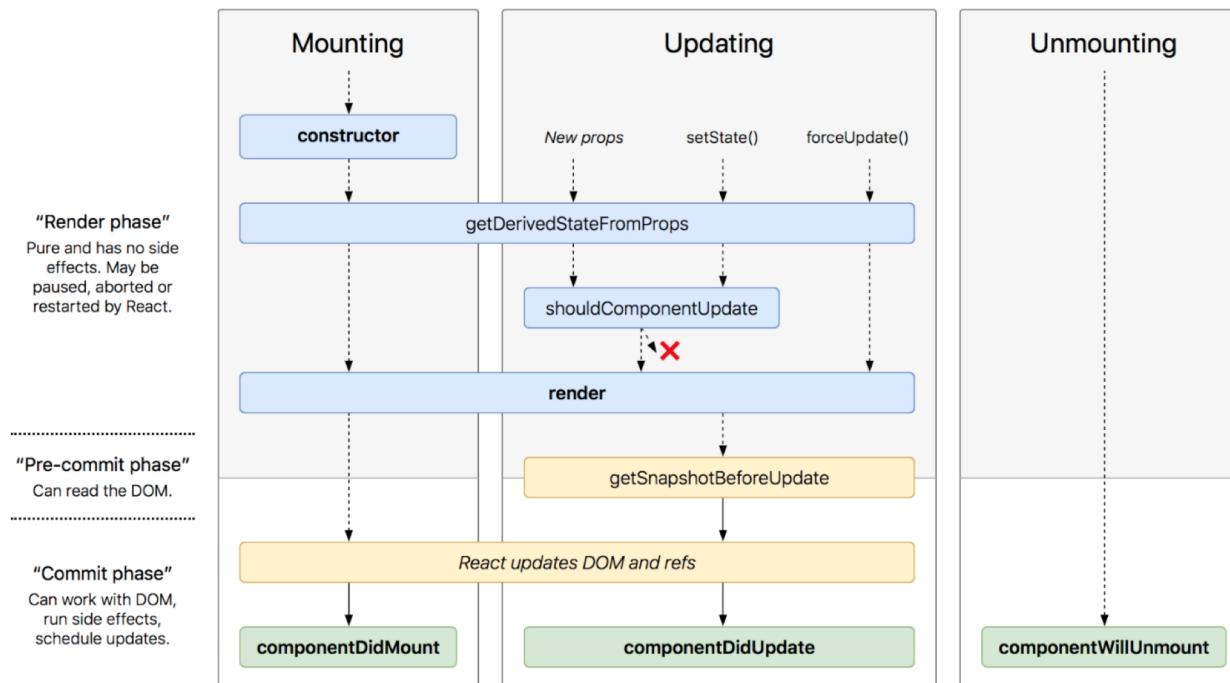
## 5. Component Lifecycle

5.1 Component Lifecycle 개요

5.2 주요 Component Lifecycle 메서드

# 5.1 Component Lifecycle 개요

- ✓ 생명주기(Lifecycle) 메서드는 클래스 기반 컴포넌트에 존재하는 메서드로 특정 시점에 호출되는 메서드입니다.
- ✓ 생명주기 메서드는 컴포넌트의 마운트(mounting), 업데이트(updating), 마운트 해제(unmounting) 시점에 따라 구분 하며 모두 8개의 메서드가 있습니다.
- ✓ 컴포넌트가 생성되고 `render()` 메서드를 통해 `ReactElement`를 반환 실제 DOM에 반영되는 것이 마운트입니다.
- ✓ `props, state`의 데이터가 변경되거나 `forceUpdate()` 메서드 호출을 통해 컴포넌트가 변경되는 것이 업데이트입니다.



출처: <http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/>

## 5.2 주요 Component Lifecycle 메서드 (1/3)

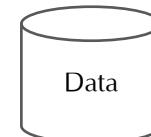
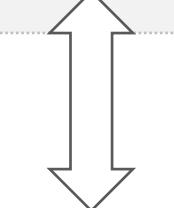
- ✓ 마운트 과정에서 호출되는 주요 메서드는 `constructor()` 메서드와 `componentDidMount()` 메서드입니다.
- ✓ 마운트가 된다는 것은 React 컴포넌트가 실제 DOM에 삽입 된다는 것을 의미 합니다.
- ✓ `constructor()` 메서드는 해당 컴포넌트의 인스턴스가 생성될 때 호출되는 메서드로 state 객체를 초기화하거나 특정 메서드를 바인딩(binding) 하는 작업을 진행합니다.
- ✓ `componentDidMount()` 메서드는 컴포넌트가 마운트 된 직후 호출되며 데이터 로딩과 같은 작업 등에 재정의 합니다.

```
constructor(props) {
  super(props); // 필수 코드
  this.state = {
    counter : 0
  }
  this.onButtonClick = this.onButtonClick.bind(this);
  // 이벤트 메소드 바인딩
}
```

[참고]

- `super()` 생성자 호출 이전에는 `this`가 할당되지 않습니다.
- `super()` 생성자 파라미터로 `props`의 전달의 목적은 생성자 내부에서 `this.props` 접근이 가능하도록 하기 위해서입니다.

```
componentDidMount(){
  // 데이터 로딩
}
```



## 5.2 주요 Component Lifecycle 메서드 (2/3)

- ✓ `render()` 메서드는 클래스 기반 컴포넌트에서 반드시 재정의 해야 하는 메서드로 React element를 반환합니다.
- ✓ React 16.x 이전 버전까지는 `render()` 메서드가 하나의 element만 반환해야 했지만 16.x 버전부터는 배열을 이용해 다수의 React element를 반환할 수 있습니다.
- ✓ `getDerivedStateFromProps()` 메서드는 `nextProps`, `prevState` 두 개의 전달인자를 받으며 새로운 `props` 값으로 state 값을 동기화 하고자 할 때 사용합니다.

```
import React, { Component } from 'react';
class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
  }
  onAddButtonHandler() {
    this.setState({
      count: this.state.count + 1,
    });
  }
  shouldComponentUpdate() {
    console.log('Lifecycle : shouldComponentUpdate()');
    return true;
  }
  getSnapshotBeforeUpdate() {
    console.log('Lifecycle : getSnapshotBeforeUpdate()');
    return null;
  }
  componentDidMount() {
    console.log('Lifecycle : componentDidMount()');
  }
  componentDidUpdate() {
    console.log('Lifecycle : componentDidUpdate()');
  }
}
```

```
render() {
  console.log('Lifecycle : render()');
  return (
    <div>
      <label>{this.state.count}</label>
      <br />
      <button type="button" onClick={this.onAddButtonHandler.bind(this)}>
        +
      </button>
    </div>
  );
}
export default Counter;
```

Lifecycle : render()	Counter.js:30
Lifecycle : render()	Counter.js:30
Lifecycle : componentDidMount()	Counter.js:24
Lifecycle : shouldComponentUpdate()	Counter.js:16
Lifecycle : shouldComponentUpdate()	Counter.js:16
Lifecycle : render()	Counter.js:30
Lifecycle : render()	Counter.js:30
Lifecycle : getSnapshotBeforeUpdate()	Counter.js:20
Lifecycle : componentDidUpdate()	Counter.js:27

## 5.2 주요 Component Lifecycle 메서드 (3/3)

- ✓ `shouldComponentUpdate()` 메서드는 성능 최적화를 위해 사용하는 메서드입니다.
- ✓ React는 Virtual DOM과 실제 DOM의 상태를 비교하여 변경된 부분에 대해서만 새로운 렌더링을 진행합니다.
- ✓ `render()` 메서드는 Virtual DOM을 대상으로 렌더링을 진행하는 메서드입니다.
- ✓ `shouldComponentUpdate()`에서 이전 상태(state, props)를 비교하여 `render()` 호출 여부를 결정할 수 있습니다.
- ✓ `shouldComponentUpdate()`를 재정의 할 수도 있지만 `React.PureComponent`를 확장하여 사용합니다.

```
import React, { Component } from 'react';
class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
  }
  onAddButtonHandler() {
    this.setState({
      count: this.state.count + 1,
    });
  }
  shouldComponentUpdate() {
    console.log('Lifecycle : shouldComponentUpdate()');
    return false;
  }
  getSnapshotBeforeUpdate() {
    console.log('Lifecycle : getSnapshotBeforeUpdate()');
    return null;
  }
  componentDidMount() {
    console.log('Lifecycle : componentDidMount()');
  }
}
```

```
componentDidUpdate() {
  console.log('Lifecycle : componentDidUpdate()');
}
render() {
  console.log('Lifecycle : render()');
  return (
    <div>
      <label>{this.state.count}</label>
      <br />
      <button type="button" onClick={this.onAddButtonHandler.bind(this)}>
        +
      </button>
    </div>
  );
}
export default Counter;
```

Lifecycle : render()	<a href="#">Counter.js:31</a>
Lifecycle : render()	<a href="#">Counter.js:31</a>
Lifecycle : componentDidMount()	<a href="#">Counter.js:25</a>
Lifecycle : shouldComponentUpdate()	<a href="#">Counter.js:16</a>
Lifecycle : shouldComponentUpdate()	<a href="#">Counter.js:16</a>



## 6. MobX

---

**6.1 state 관리 개요**

**6.2 flux**

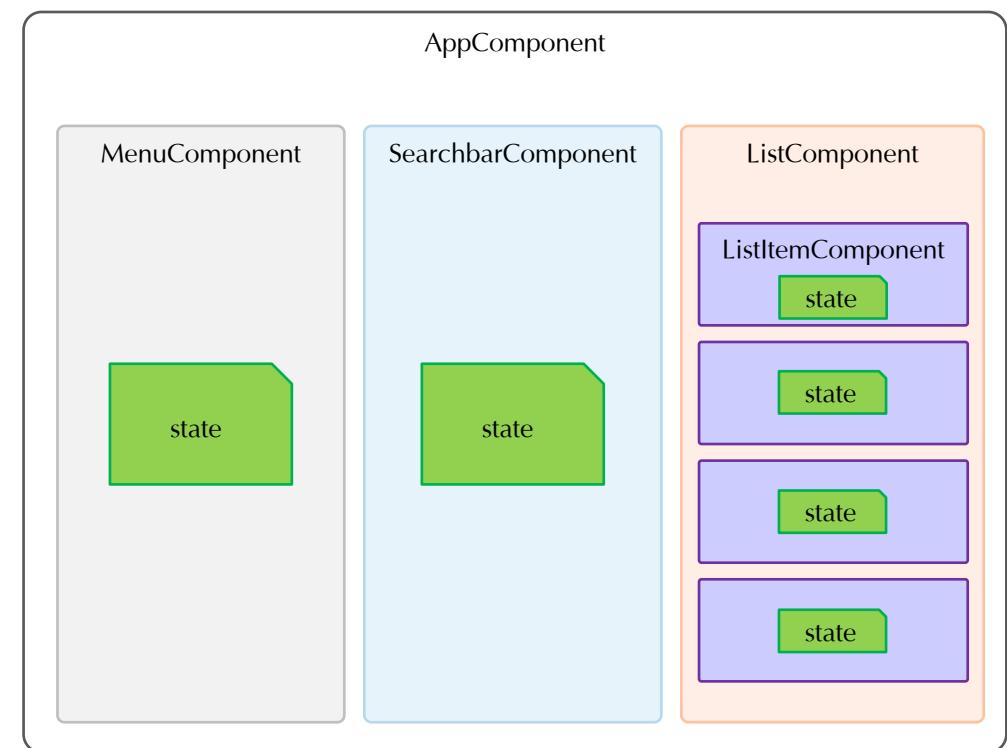
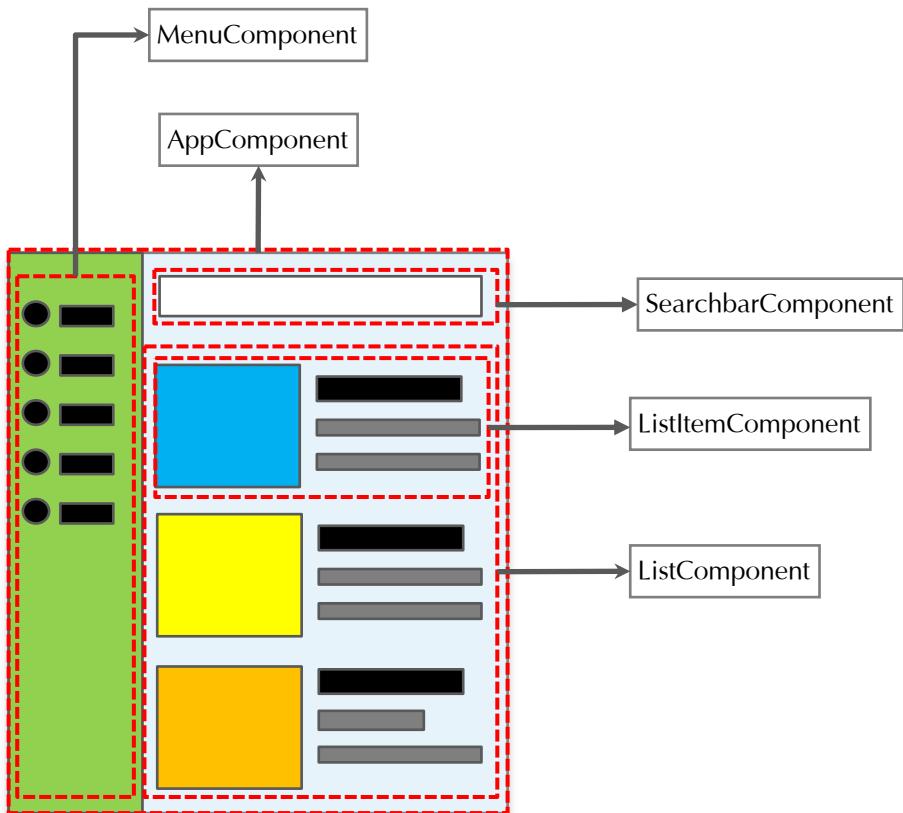
**6.3 MobX 개요**

**6.4 MobX의 적용**

**6.5 MobX 예제 실습**

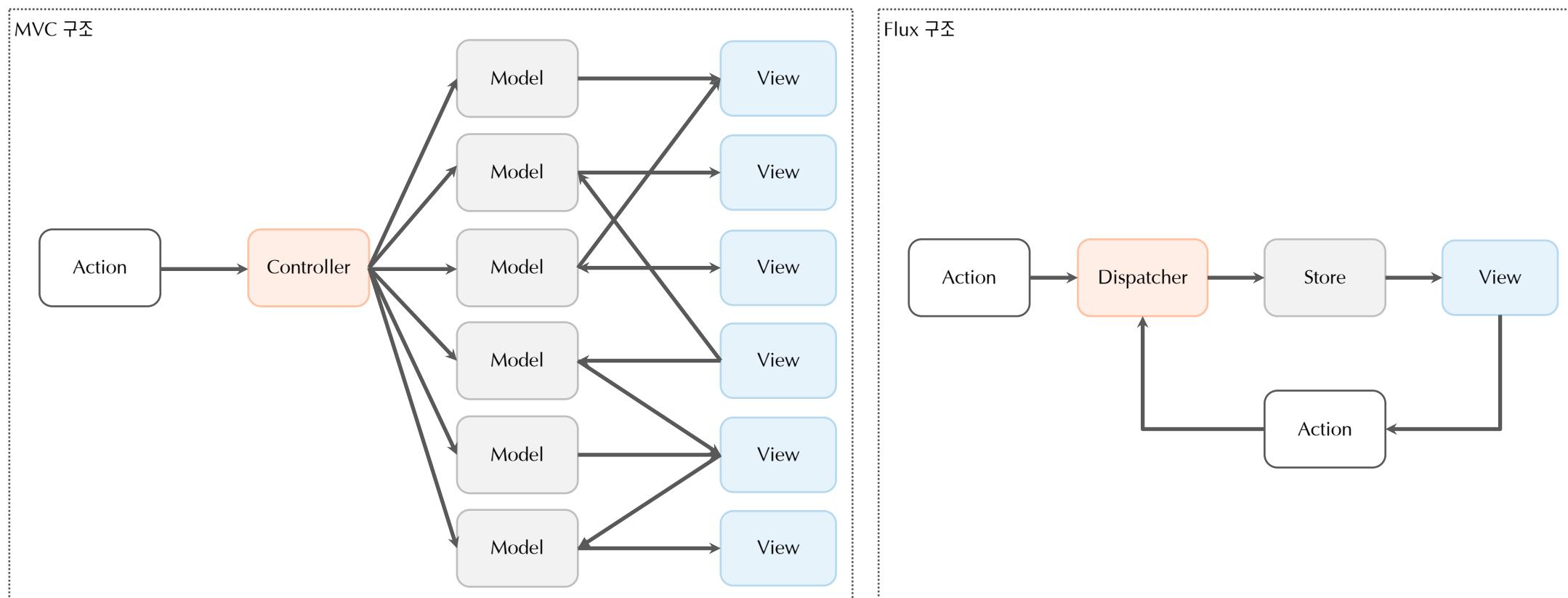
## 6.1 state 관리 개요

- ✓ 클래스 기반의 컴포넌트는 state 객체를 이용해 변경 가능한 데이터를 관리 할 수 있습니다.
- ✓ 사용자 UI는 다수의 컴포넌트를 통해 표현되며 각 컴포넌트가 상태를 관리할 경우 데이터 관리 및 제어가 어렵습니다.
- ✓ 각각의 컴포넌트에서 특정 데이터의 공유가 필요한 경우 이 데이터를 참조하거나 변경할 때 제약이 발생합니다.
- ✓ 공통의 데이터 관리 영역(store)을 두고 이를 통해 state를 관리하면 컴포넌트 간 데이터 공유가 가능합니다.



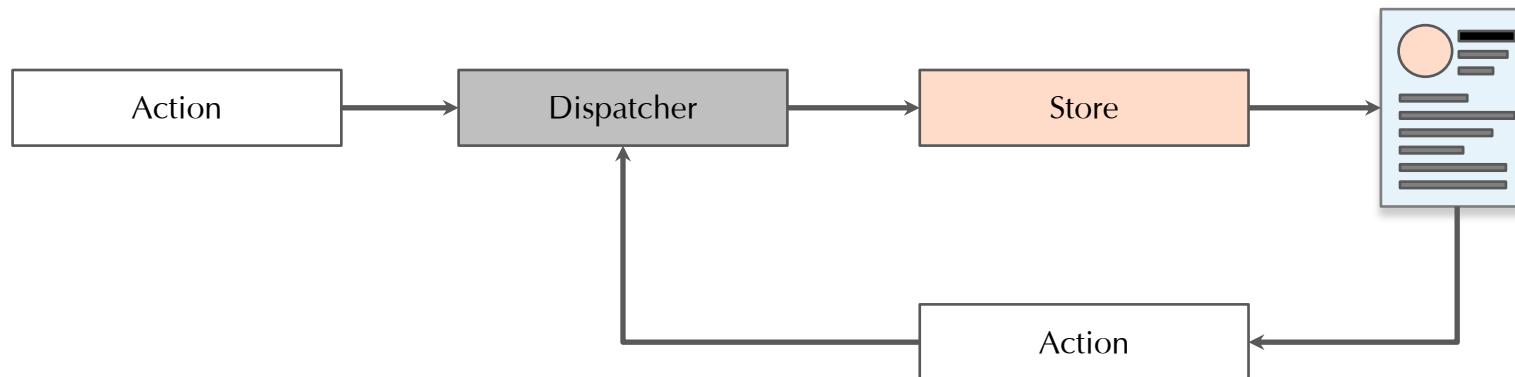
## 6.2 Flux[1/2]

- ✓ Flux는 React를 이용한 UI 구성에서 데이터의 흐름을 관리하는 어플리케이션 아키텍처입니다.
- ✓ MVC 구조의 데이터 흐름은 View와 Model 사이에 양방향 데이터 이동이 가능합니다.
- ✓ MVC 구조는 View가 많아 질수록 데이터의 흐름과 관리가 어렵습니다.
- ✓ React는 단방향 데이터 흐름(Model → View)만 가능하기 때문에 MVC 아키텍처는 적용하지 않습니다.
- ✓ 단방향 데이터 흐름은 구조를 단순화 할 수 있으며 데이터의 이동 또한 명확하게 확인할 수 있습니다.



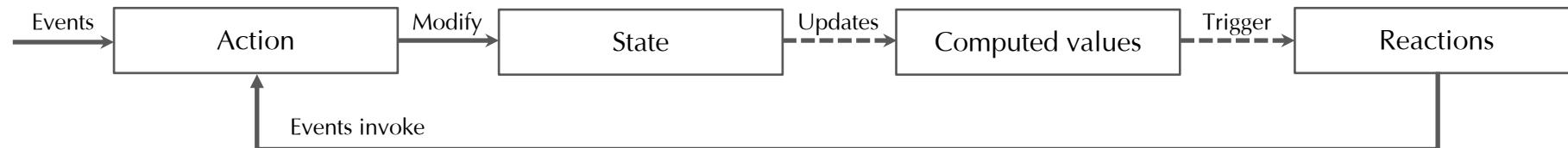
## 6.2 Flux[2/2]

- ✓ Flux는 단방향 데이터 흐름을 보완하기 위해 개발된 아키텍처이며 View로 분산된 state를 통합 관리합니다.
- ✓ Flux 아키텍처는 Action, Store, Dispatcher, View로 구성됩니다.
- ✓ View 각각의 state는 Store를 이용해 통합 관리되며, Store의 데이터는 Action을 이용해 제어합니다.
- ✓ Store에서 제어하는 state는 곧 연결된 View의 state와 다름없으며 Store의 state가 변경되면 View도 갱신됩니다.



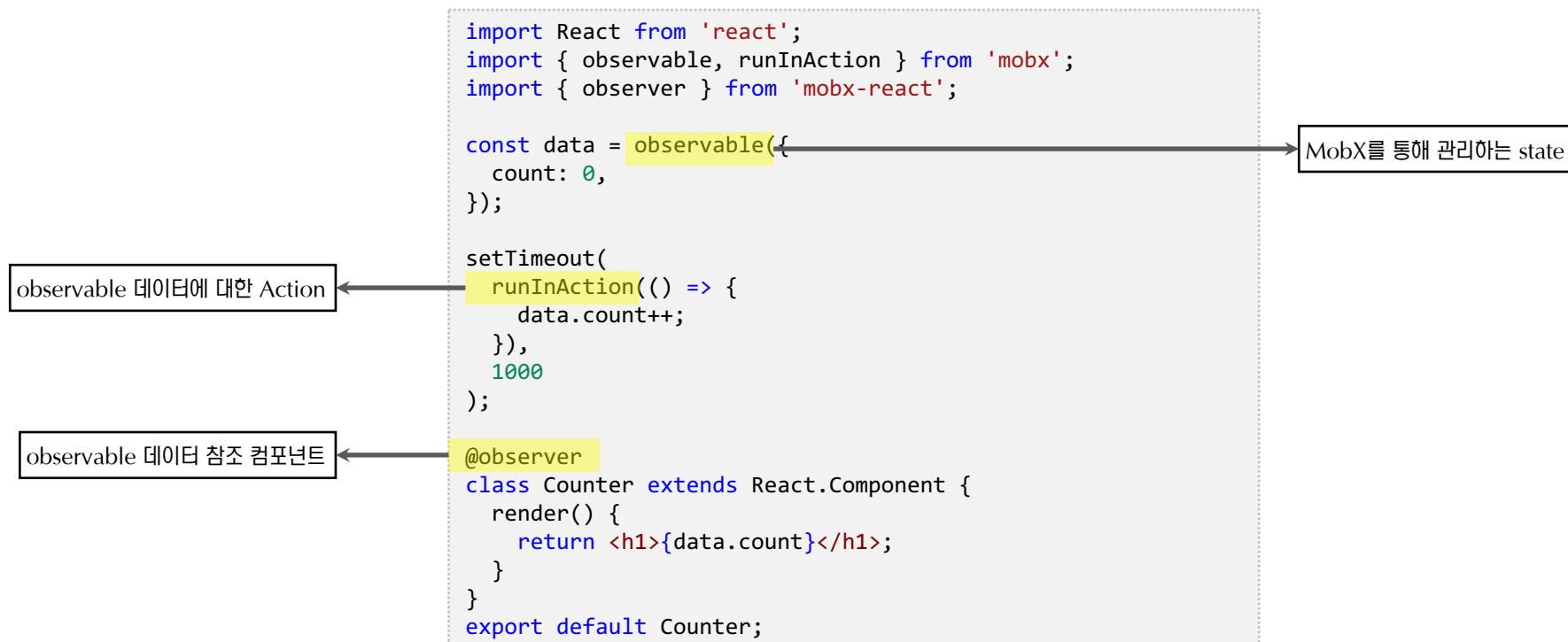
## 6.3 MobX 개요(1/4)

- ✓ MobX도 Flux와 마찬가지로 클라이언트 사이드에서 state를 관리하기 위해 사용하는 라이브러리입니다.
- ✓ MobX를 이용하면 컴포넌트의 state를 별도의 영역에서 관리하고 각 컴포넌트는 이에 접근할 수 있습니다.
- ✓ MobX를 적용하기 위해서는 mobx.js 라이브러리와 mobx-react.js 라이브러리가 필요합니다.
- ✓ MobX는 다수의 store를 관리할 수 있으며, 관리하는 데이터는 특정 데이터의 형태(Observable)로 관리 합니다.



## 6.3 MobX 개요(2/4)

- ✓ MobX가 제공하는 대표적인 API는 observable, action, observer, computed가 있습니다.
- ✓ MobX 라이브러리는 TypeScript가 적용되어 있으며 API의 사용은 데코레이터(@)를 이용하는 것이 일반적입니다.
- ✓ @observable API는 store에서 관리하고자 하는 state 데이터를 의미하며 Observable 객체를 통해 관리됩니다.
- ✓ @observer API는 @observable API로 관리되는 state를 참조하는 React 컴포넌트에 적용합니다.



## 6.3 MobX 개요(3/4)

- ✓ **@action**은 관찰 대상 데이터 즉, **observable state**의 값을 변경하는 메서드에 적용합니다.
- ✓ **state**에 대한 단순 조회와 같은 메서드에 적용하는 것은 의미가 없습니다.
- ✓ **@computed**는 **get** 메서드에 일반적으로 적용하거나 Model 객체간 전환 시점에 적용합니다.
- ✓ **@computed**가 적용된 메서드를 수행할 때 해당 **observable state**의 변화가 없을 경우 내부 로직을 생략합니다.

```
import { observable, computed, autorun } from 'mobx';

class TodoListStore {
  @observable todos = [];

  constructor(){
    autorun(() => {
      console.log("Number of unfinished : " + this.unfinishedTodoCount);
    });
  }

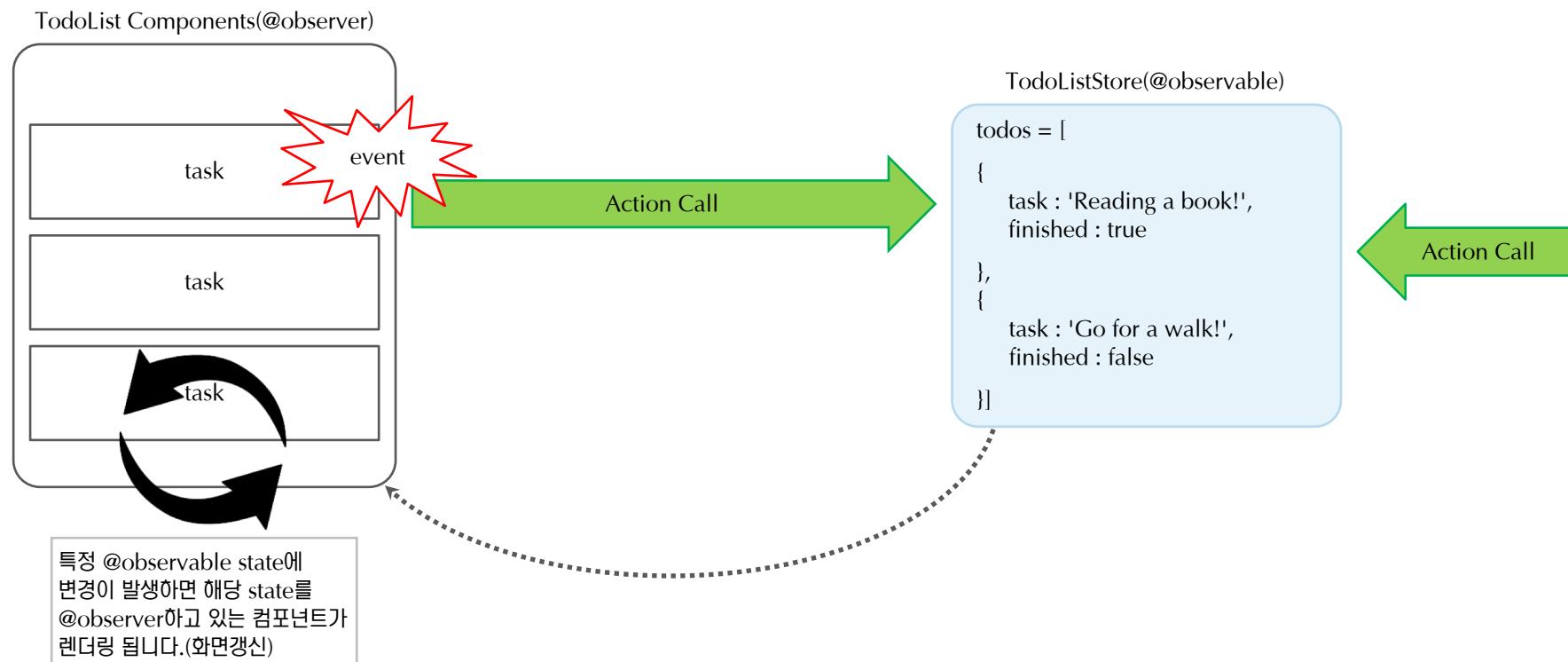
  @computed
  get unfinishedTodoCount(){
    return this.todos.filter(todo => !todo.finished).length;
  }

  @action
  addTodo(task){
    this.todos.push({
      task : task,
      finished: false
    });
  }
}

export default TodoListStore;
```

## 6.3 MobX 개요(4/4)

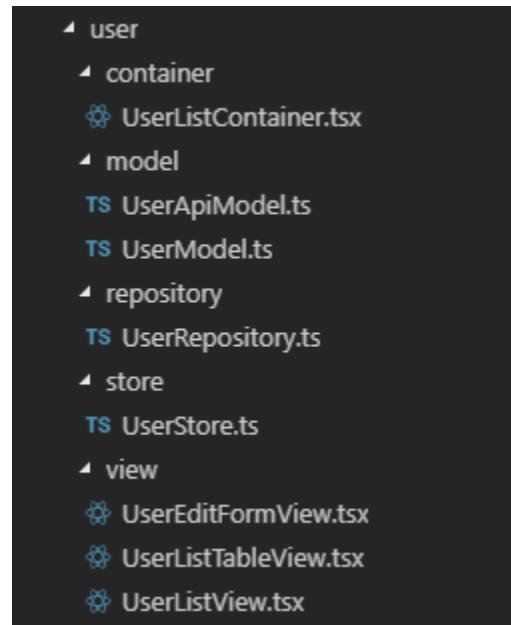
- ✓ `@observer`는 store를 통해서 state를 관리하는 `React.Component`에 적용합니다.
- ✓ `@observer`가 적용된 `React.Component`는 관련 observable state가 변경되면 렌더링을 수행합니다.
- ✓ MobX는 다수의 store를 구성하는 것이 가능하며 `@inject`를 이용해 해당 `@observer` 컴포넌트의 store를 주입합니다.
- ✓ 이외에도 MobX 라이브러리는 `@autorun`, `@transaction`과 같은 다양한 API를 제공합니다.



## 6.4 MobX의 적용[1/3]

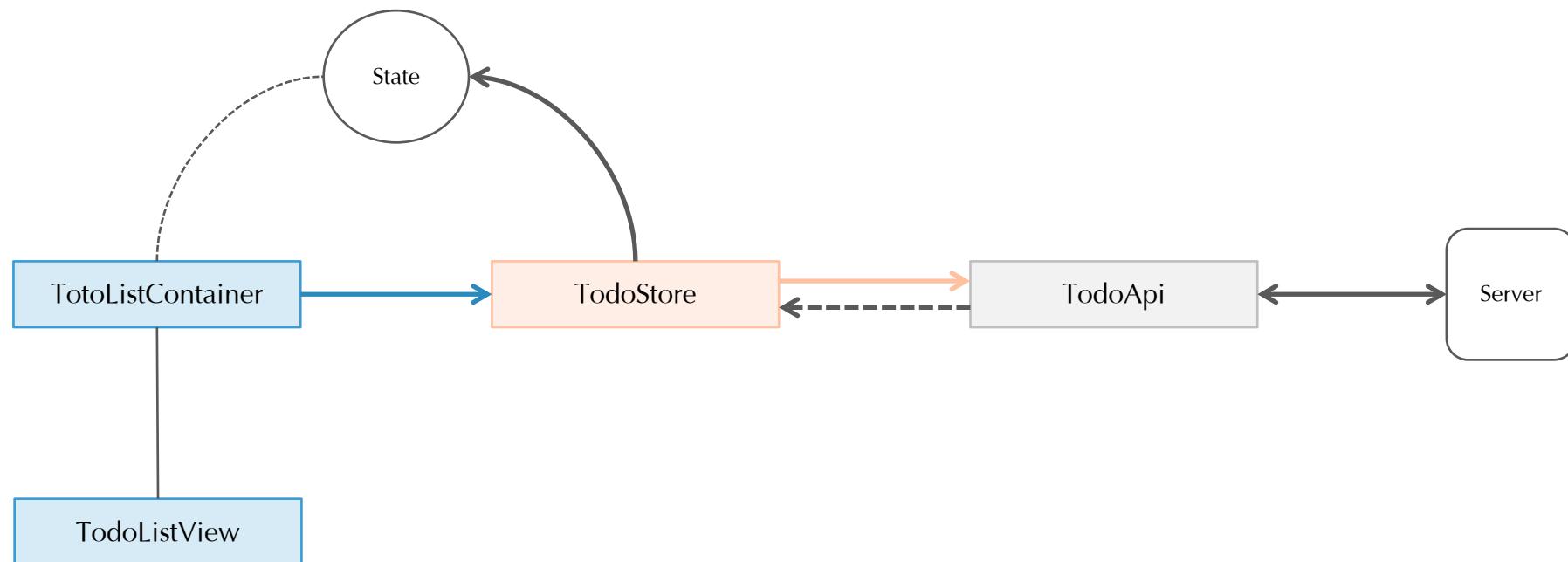
✓ React와 MobX를 통한 UI 구성은 다음과 같은 패키지의 구성을 갖습니다.

- container : React Component로 구성하며 store와 React Component를 연결하는 역할을 담당합니다.
- view : 순수 React Component로 구성하며 container에 포함됩니다.
- repository(or api) : 서버와 통신을 담당하는 클래스로 구성합니다.
- store : 전역 state를 관리하는 Store 클래스로 구성합니다.
- model : 서버의 model과 view model의 전환을 담당합니다.



## 6.4 MobX의 적용[2/3]

- ✓ container로 정의한 React 컴포넌트의 state는 store를 통해 관리합니다.
- ✓ store는 @action 메소드를 포함하며 이 메소드 호출을 통해 state를 제어합니다.
- ✓ store의 state 데이터에 변경이 일어나면 해당 state와 연결된 React Component는 다시 렌더링 됩니다.
- ✓ api는 서버와 통신을 담당하며 axios.js와 같은 서드-파티 라이브러리(Third-party Lib.)를 사용합니다.



## 6.4 MobX의 적용[3/3]

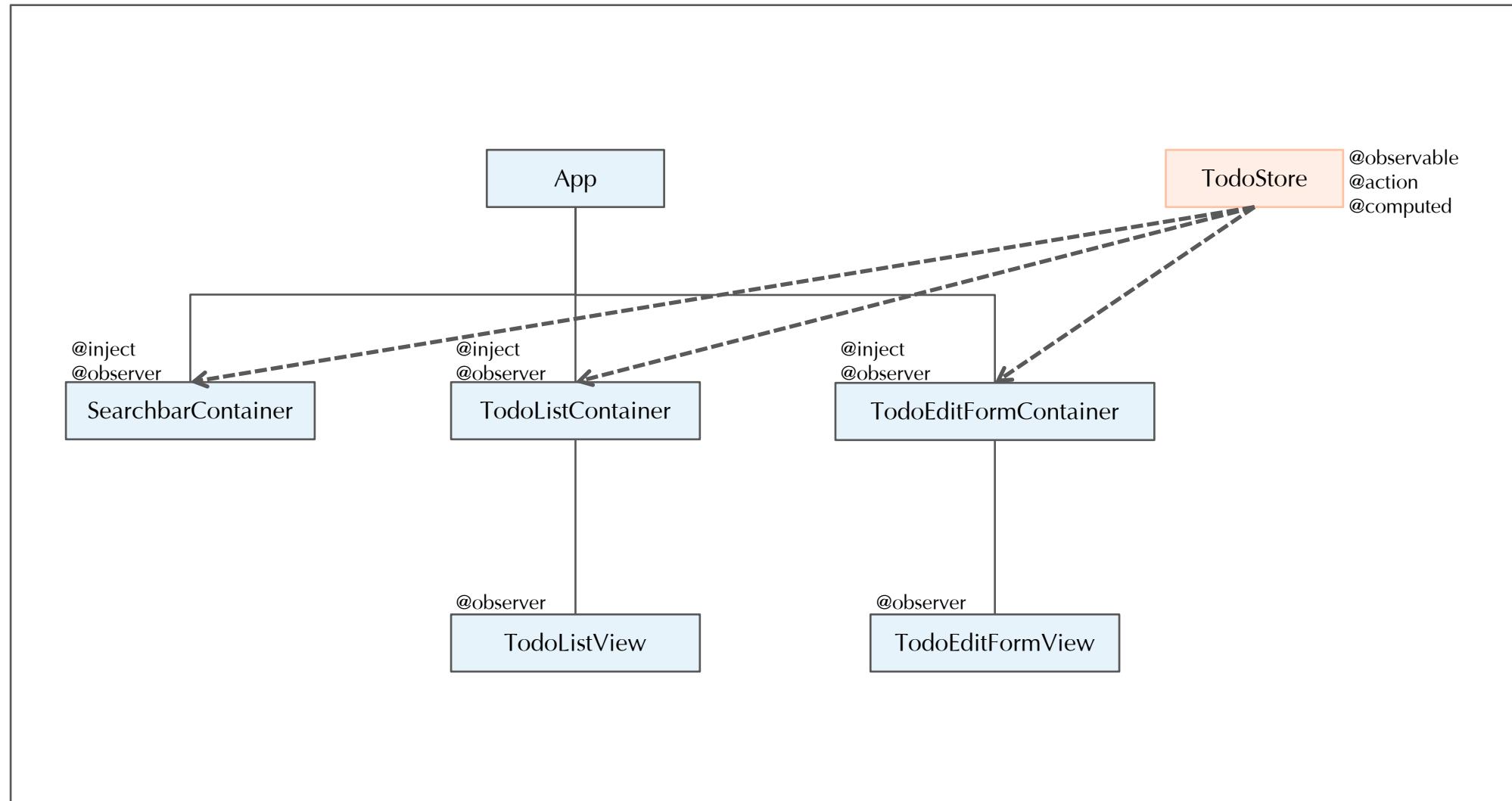
- ✓ MobX는 ES7의 스펙인 데코레이터(Decorator)를 지원하며 함수 형태의 사용도 가능합니다.
- ✓ 전역 형태의 Store를 구성하기 위해서는 Store 클래스를 정의 합니다.
- ✓ 정의한 Store 클래스에 사용하고자 하는 state를 정의하고 초기화 합니다.

```
class UserStore {  
    @observable  
    users = [];  
  
    @observable  
    user = {};  
  
    constructor(){  
        autoBind(this);  
    }  
}
```

```
class UserStore {  
    @observable  
    users: UserModel[];  
    @observable  
    user: UserModel;  
  
    constructor() {  
        autoBind(this)  
    }  
}
```

## 6.5 MobX 예제 실습





## 7. Router

---

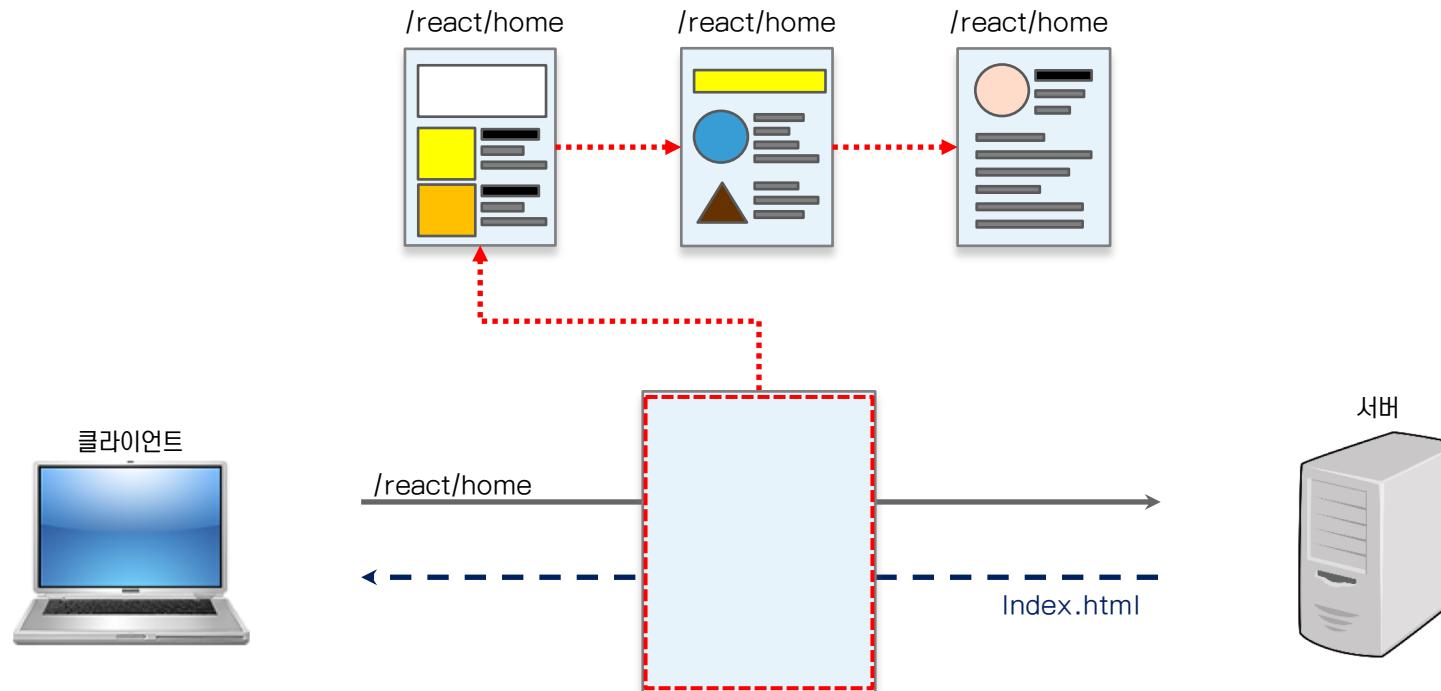
7.1 Router 개요

7.2 React-Router 동작 방식

7.3 React-Router 주요 컴포넌트

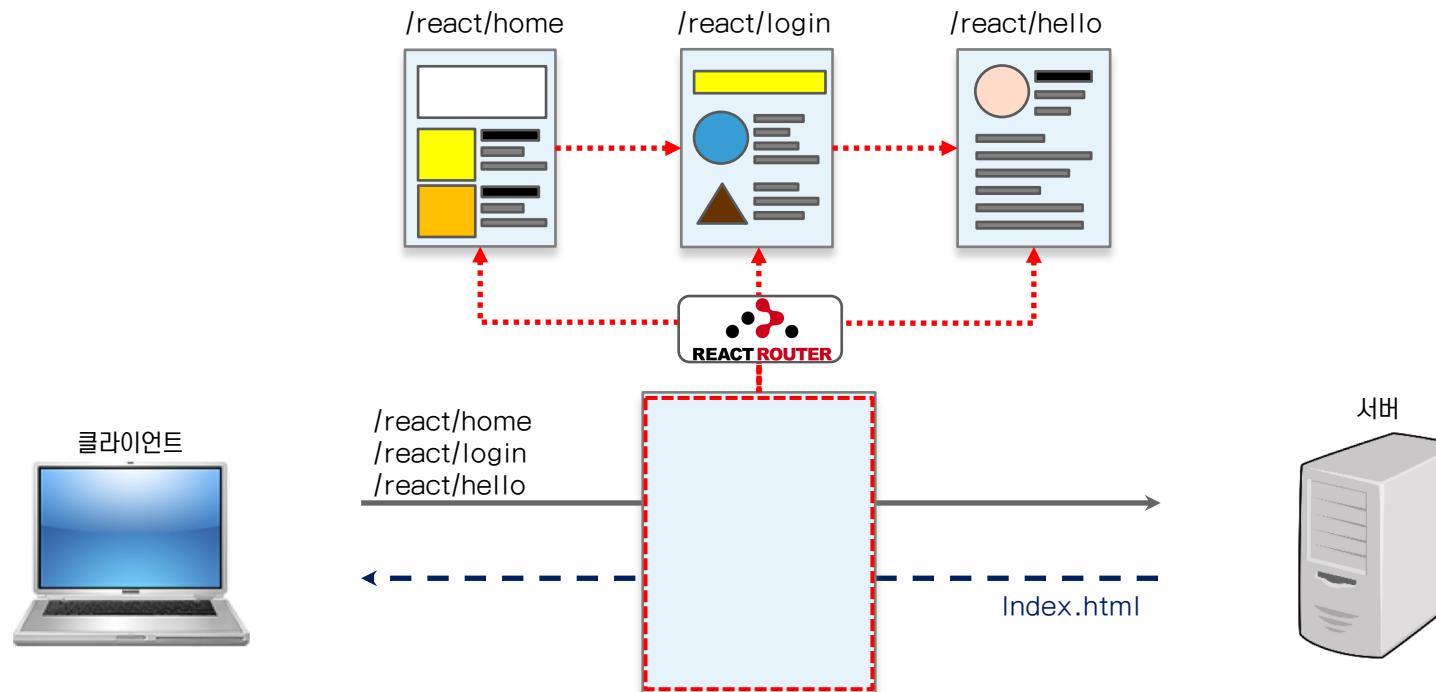
## 7.1 Router 개요 [1/2]

- ✓ SPA는 단일 페이지이므로 사용자가 브라우저에 URL을 입력하여 요청하면 서버는 같은 페이지를 반환합니다.
- ✓ JavaScript에서 페이지를 구성하고 이동하면 사용자가 브라우저를 새로 고침 하는 순간 처음 화면으로 돌아갈 것입니다.
- ✓ 페이지마다 주소가 없으므로 특정 페이지에 접속하기 힘들고 북마크를 할 수 없다는 단점이 있습니다.
- ✓ 이를 개선하기 위해 JavaScript에서 URL과 페이지를 유기적으로 구성하고, 브라우저의 history와 동기화 해야 합니다.



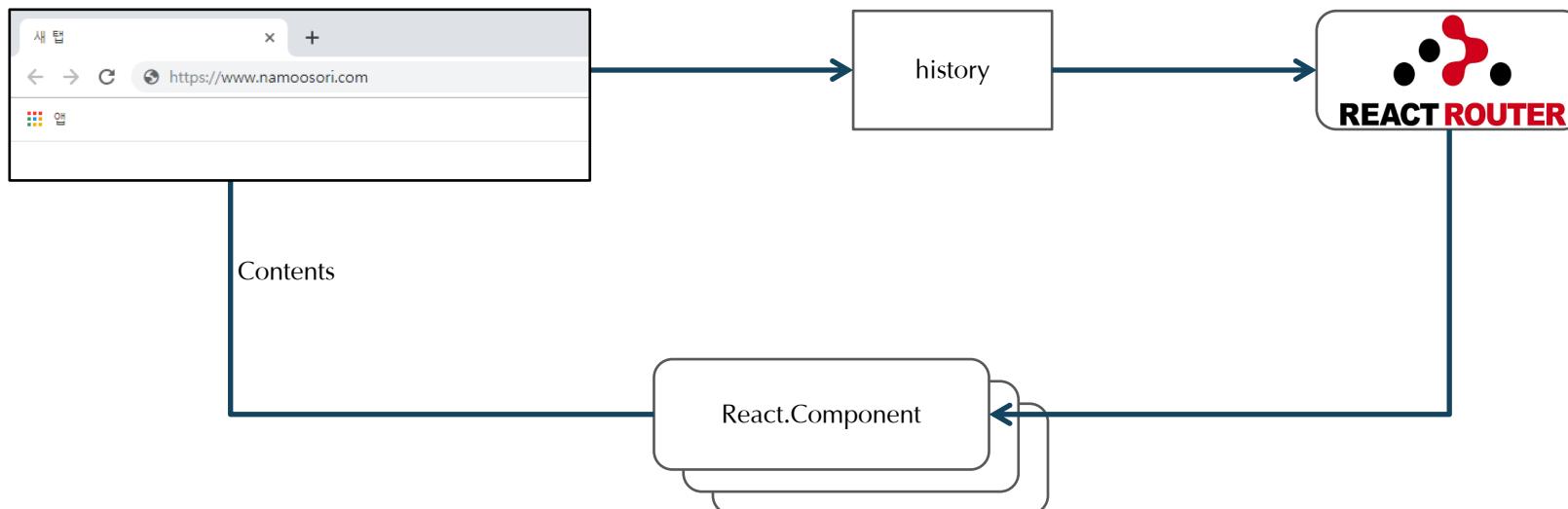
## 7.1 Router 개요 [2/2]

- ✓ URL의 구성, URL과 리액트 컴포넌트의 맵핑 및 라우팅 하기 위해 React-Router를 사용합니다.
- ✓ React는 다른 UI프레임워크와 다르게 view만을 담당하기 때문에 라우팅 기능이 존재하지 않습니다.
- ✓ 페이스북 공식 라우팅 라이브러리는 존재하지 않으므로 써드파티 라이브러리를 사용합니다.
- ✓ React Router는 JSX를 사용하기 때문에 좀 더 손쉬운 사용이 가능합니다.



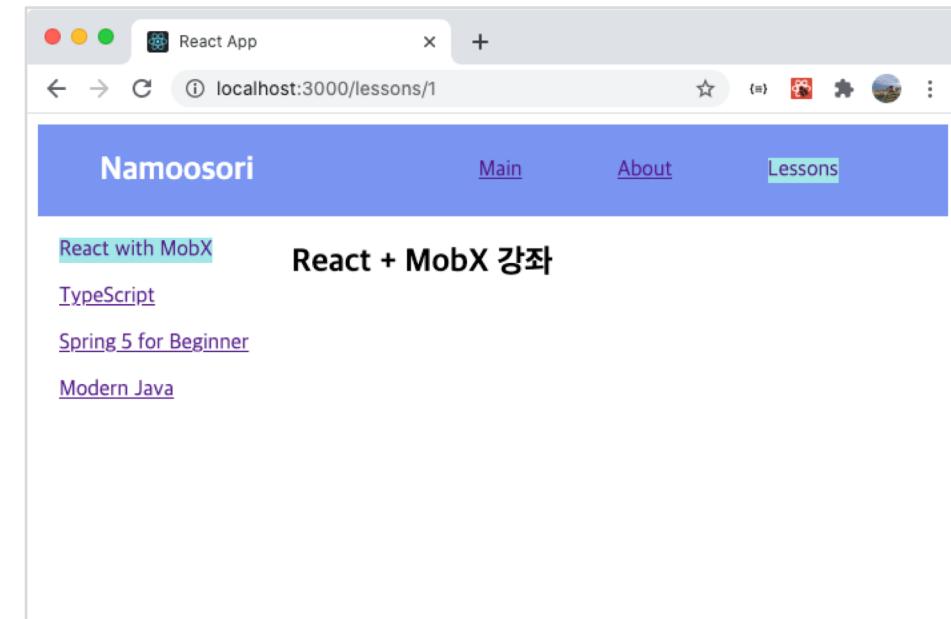
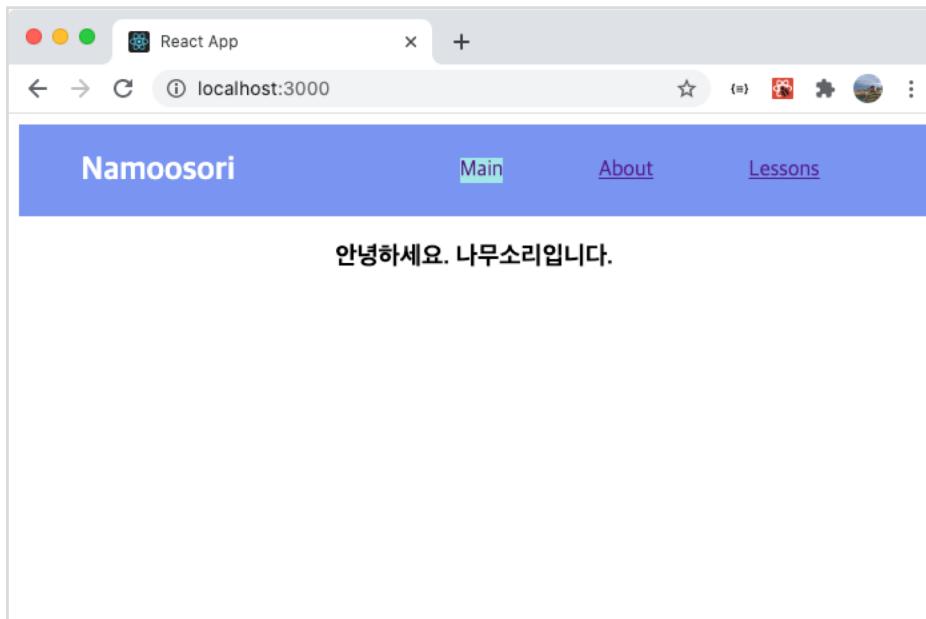
## 7.2 React-Router 동작 방식

- ✓ React-Router 라이브러리는 history 라이브러리를 포함합니다.
- ✓ 사용자에 의해 URL이 변경되면 history는 변경 URL을 감지하여 React-Router로 전달합니다.
- ✓ React-Router는 변경된 URL에 해당하는 컴포넌트에 대해 렌더링을 요청합니다.
- ✓ 사용자는 URL의 변경으로 새로운 컴포넌트의 컨텐츠를 확인합니다.



## 7.3 React-Router 주요 컴포넌트

- ✓ react-router가 제공하는 주요 컴포넌트는 <BrowserRouter>, <Route>, <Link>, <Switch> 등이 있습니다.
- ✓ <BrowserRouter> 컴포넌트는 <Router> 컴포넌트를 기반으로하며 HTML5 history 객체를 이용해 포함한 컴포넌트들 간의 라우팅을 지원합니다.
- ✓ <Route> 컴포넌트는 경로(path)에 따라 해당 컴포넌트를 랜더링 하는 가장 기본이 되는 컴포넌트입니다.
- ✓ <Link> 컴포넌트는 html의 <a>와 같은 페이지 이동이 아닌 브라우저의 URL을 변경합니다.



✓ 토론

## 감사합니다...

- ❖ 넥스트리(주)
- ❖ [www.nextree.io](http://www.nextree.io)