ReactJS 입문과 활용

백명숙

과정개요

■ React의 개념과 Architecture를 이해하고, React 컴포넌트의 생성 방법을 살펴본다. React에서 컴포넌트를 작성할 때 사용하는 JSX 문법과 React의 데이터 흐름, 이벤트 핸들링, state와 props 관련 함수를 살펴본다. State를 관리를 효율적으로 해주는 Redux를 활용해 본다.

Learning Object(학습모듈) 및 커리큘럼

LO	커리큘럼
React 소개	- React 소개 - JSX 및 ES6 소개 - 개발환경 구축
React 컴포넌트	- React 컴포넌트 소개 - 컴포넌트간의 통신 (Props / State)
React 이벤트 및 DOM	- 이벤트 바인딩 - ref:DOM - 컴포넌트 반복 : map() - Todo-List App 만들기
React Redux	- React Redux - Counter App 만들기
Redux와 Axios	- Axios 개념 - Todo-List App에 redux, redux-thunk, axios 적용하기
React Router	- SPA 와 Routing 개념 - React Router 사용하기

ReactJS

15.0.2

3.5.2

2.4.0

1.13

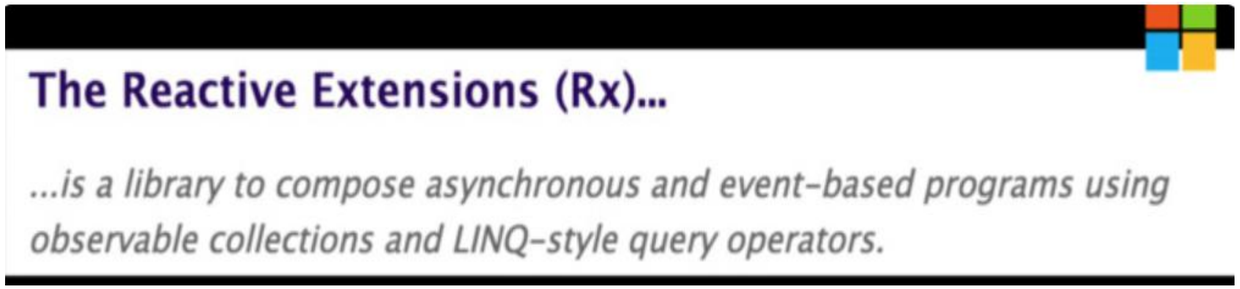
6.*

백명숙(ms.baek@hucloud.co.kr)

React & Reactive

- 리액트(React)는 페이스북이 웹 개발을 쉽게 하기 위해 만든 기술입니다. 커스텀 컴포넌트를 만들고 쉽게 조합해서 뷰를 손쉽게 만들 수 있습니다.

- 리액티브(Reactive)는 마이크로소프트가 창안한 개념으로 스트림과 비동기 처리등을 LINQ에서 영향을 받은 방법으로 비동기 처리를 할 수 있게 한 패러다임입니다. 리액티브는 ReactiveX(Rx)를 중심으로 여러 언어에 맞게 다양하게 구현이 되어 있습니다.



React Native

- 리액트 네이티브(React Native) 리액트의 접근 방법을 모바일로 확장하는 페이스북의 오픈소스 프로젝트 입니다.
  기존의 모바일 자바스크립트 Tool들이 WebView를 통해 인터페이스를 구축하는 하이브리드 방식이었다면 리액트 네이티브는 자바스크립트로 작업 하지만 인터페이스는 네이티브 위젯으로 표시하는 방식입니다.
  리액트 네이티브는 네이티브 UI를 통해서 이질감 없고 쾌적한 사용자 경험을 제공합니다.

프론트엔드 라이브러리 / 프레임워크

- 프론트엔드 라이브러리가 필요한 이유

요즘의 웹은 단순히 웹 페이지가 아니라, 웹 애플리케이션이다. 유저인터페이스를 동적으로 나타내기 위해서는 정말 많은 상태를 관리 해주어야 합니다.

```
<div>
  <h1>Counter</h1>
  <h2 id="number">0</h2>
  <button id="increase">+</button>
</div>
```

```
var number = 0;
var elNumber = document.getElementById('number');
var btnIncrease = document.getElementById('increase');

btnIncrease.onclick = function() {
  number++;
  elNumber.innerText = number;
}
```

Javascript로 직접 구현하는 것도 가능은 하지만 프로젝트가 규모가 커지고, 다양한 유저 인터페이스와 인터랙션을 제공하게 된다면, 많은 DOM 요소들을 직접 관리하고 코드를 정리 하는 것은 힘든 일입니다. 웹 개발을 하게 될 때, DOM 관리와 상태 값 업데이트 관리를 최소화 하고, 오직 기능 개발, 그리고 사용자 인터페이스를 구현하는 것에 집중 할 수 있도록 하기 위해서 여러 라이브러리들 혹은 프레임워크들이 만들어졌습니다. 대표적으로 Angular, Vue, React , Ember, Backbone 등이 있습니다.

프론트엔드 라이브러리 / 프레임워크 선택

- Angular(https://angular.io/)

Router, HTTP 클라이언트 등 웹 프로젝트에서 필요한 대부분의 도구들이 프레임워크 안에 내장되어 있으며, 컴포넌트와 템플릿이 완벽하게 분리되어 있습니다.
AngularJS는 javascript와 함께 사용되며, 기업에서 많이 사용 되어, 유지보수하고 있는 프로젝트가 많은 편입니다. Angular2 부터는 주로 typescript와 함께 사용 됩니다. 현재 Angular8 버전

- React(https://reactjs.org/)

"컴포넌트" 라는 개념에 집중이 되어 있는 라이브러리입니다.
생태계가 엄청 넓고, 사용하는 곳도 많습니다. HTTP 클라이언트, Router, 상태 관리 등의 기능들은 내장되어 있지 않습니다. 따로 공식 라이브러리가 있는 것이 아니므로, 개발자가 원하는 스택을 맘대로 골라서 사용 할 수 있습니다.

- Vue(https://vuejs.org/)

입문자가 사용하기 쉽다. Webpack 같은 모듈 번들러를 사용하여 프로젝트를 구성 해야하는 Angular와 React와 달리, 단순히 CDN 에 있는 파일을 로딩 하는 형태로 스크립트를 불러와서 사용하기도 편합니다. HTML을 템플릿 처럼 그대로 사용 할 수도 있어서 마크업을 만들어 주는 디자이너/퍼블리셔가 있는 경우 작업 흐름이 매우 매끄럽습니다. 공식 Vue-Router, Vuex 상태 관리 라이브러리가 존재합니다.

React 소개

We built React to solve one problem: building large applications with data that changes over time.

우리는 지속해서 데이터가 변화하는 대규모 애플리케이션을 구축하기 위해 React를 만들었습니다.

- 변화(Mutation)

변화(Mutation)라는 것은 상당히 복잡한 작업입니다. 특정 이벤트가 발생 했을 때, 모델에 변화를 일으키고, 변화를 일으킴에 따라 어떤 DOM 을 가져와서 어떠한 방식으로 뷰를 업데이트 해줄 지 로직을 정해줘야 합니다.
Facebook에서는 “그냥 Mutation을 하지 말자. 대신에, 데이터가 바뀌면 그냥 View를 날려 버리고 새로 만들어 버리면 어떨까?”

- Virtual DOM (https://www.youtube.com/watch?v=BYbgopx44vo)

Virtual DOM 은 가상의 DOM 입니다. 변화가 일어나면, 실제로 브라우저의 DOM에 새로운 것을 넣지 않고, 자바스크립트로 이루어진 가상 DOM 에 한번 렌더링을 하고, 기존의 DOM과 비교를 한 다음에 변화가 필요한 곳에만 업데이트를 해줍니다.
이 Virtual DOM 을 사용함으로서, 데이터가 바뀌었을 때 업데이트 할 지를 고려하는게 아니라, 그냥 일단 바뀐 데이터로 일단 그려놓고 비교를 한 다음에, 바뀐 부분만 찾아서 변경 시켜 줍니다.

React 소개

- 리액트는 라이브러리입니다.

프론트엔드 프레임워크가 MVC 또는 MVW 등 구조를 지향하는 것과는 다르게 오직 View 만을 담당합니다.

그러므로 리액트는 프레임워크가 아니라 라이브러리 입니다.

Angular와 Vue가 Ajax, 데이터모델링, 라우팅 기능을 내장하고 반면에, React는 외부 라이브러리를 사용해야 합니다.

- Routing : react-router,
- 상태관리 : redux, MobX
- Ajax 처리 : axios나 fetch

React 프로젝트 환경설치

- 1.Node.js 설치하기

Node.js 를 현재 기준 LTS 버전인 v10 버전을 설치하세요.
윈도우의 경우에 노드 공식 홈페이지 다운로드 페이지 에서 설치를 하면 됩니다.

- 2.Yarn 설치하기 : npm을 대체하는 패키지 매니저

Yarn 설치는 Yarn Installation 페이지에서 여러분의 운영체제에 맞는 방식에 따라 설치하시면 됩니다.

- 3.Visual Studio Code 설치 및 Plug In 설치하기

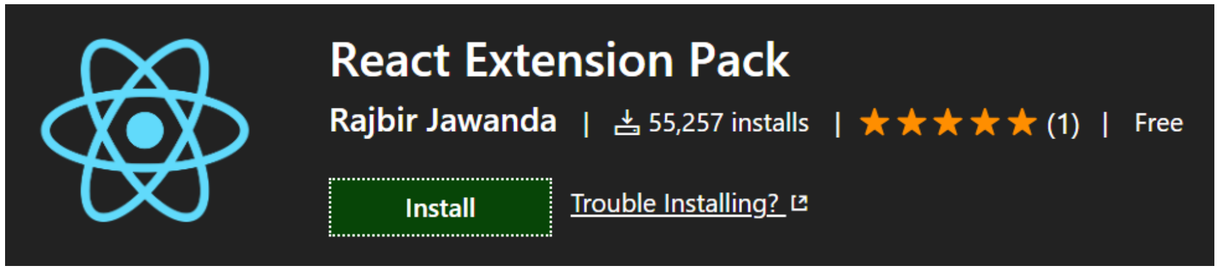
VS Code 설치는 Visual Studio Code 에서 하실 수 있습니다.

VS Code의 확장 프로그램(플러그인) 설치

1. Relative Path (상대 경로에 있는 파일 경로를 편하게 작성할 수 있는 도구입니다)
2. Guides (들여쓰기 가이드라인을 그려 줍니다)
3. vscode-js-import (import 기능, 제작자가 wangtao0101)
4. CSS Formatter (Martin Aesch)
5. IntelliSense for CSS class names

React 프로젝트 환경설치

- 4. React Extension Pack



Extensions Included

- Reactjs code snippets - Adds code snippets for React development in ES6
- ES Lint - Integrates ESLint into VS Code.
- npm - Run npm scripts from the command palatte and validate the installed modules defined in `package.json`.
- JavaScript (ES6) Snippets - Adds code snippets for JavaScript development in ES6 syntax.
- Search node_modules - Quickly search for node modules in your project.
- NPM IntelliSense - Adds IntelliSense for npm modules in your code.
- Path IntelliSense - Autocompletes filenames in your code.

1. ESLint (자바 스크립트 문법을 체크합니다)
2. Reactjs code snippets

React 프로젝트 환경설치

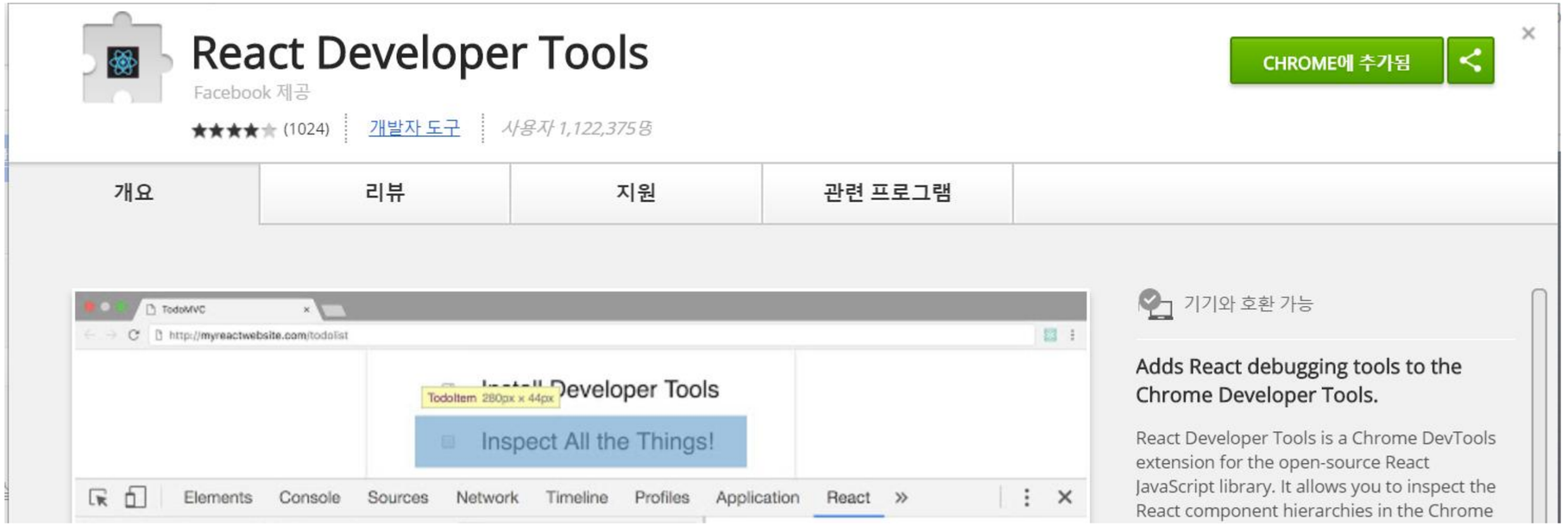
- 5.create-react-app 설치하기

create-react-app은 React 앱을 만들어주는 도구입니다.

```
npm install -g create-react-app
```

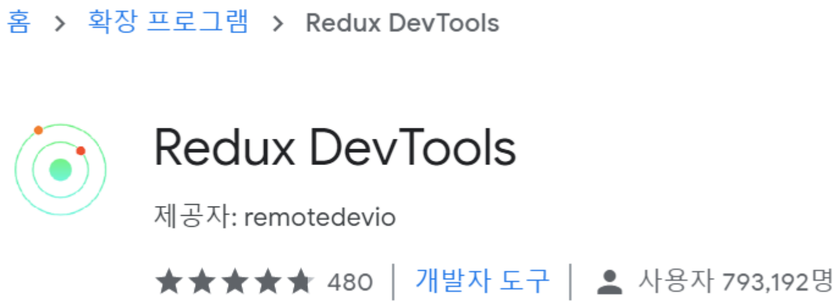
```
yarn global add create-react-app
```

- 6.Chrome에 React Developer Tool 설치



React 프로젝트 환경설치

- 7.Chrome에 Redux DevTools 설치



ECMAScript 6 소개

- ECMA(European Computer Manufacturers Association) Script는 JavaScript 프로그래밍 언어를 정의하는 국제 표준의 이름입니다.
- ECMA의 Technical Committee39(TC39)에서 논의 되었습니다.
- 현재 사용하는 대부분의 JavaScript는 2009년에 처음 제정되어 2011년에 개정된 ECMAScript 5.1 표준에 기반하고 있습니다.
- 이후 클래스 기반 상속, 데이터 바인딩(Object.observe), Promise 등 다양한 요구사항들이 도출 되었고 그 결과 2015년 6월에 대대적으로 업데이트된 ECMAScript6 가 발표되었고, 매년 표준을 업데이트하는 정책에 따라 2016년 6월에 ECMAScript7 까지 발표되었습니다.
- ES6 = ECMAScript6 = ECMAScript 2015 = ES2015

ES6

- var / let,const
  - const는 ES6에 도입된 키워드로서, 한번 선언하고 바뀌지 않는 값을 설정 할 때 사용됩니다.
    변경 될 수 있는 값은 let 을 사용하여 선언합니다.
  - var는 scope이 함수 단위이고, 반면 const와 let은 scope이 블록 단위 입니다.
  - ES6 에서는 var 를 사용하지 않고, 값을 선언 후 바꿔야 할 땐 let, 그리고 바꾸지 않을 땐 const를
    사용하면 됩니다.

src/App.js

```
function foo() {
  var a = 'hello';
  if (true) {
    var a = 'bye';
    console.log(a); // bye
  }
  console.log(a); // bye
}
```

src/App.js

```
function foo() {
  let a = 'hello';
  if (true) {
    let a = 'bye';
    console.log(a); // bye
  }
  console.log(a); // hello
}
```

ES6 축약코딩기법

- 1.삼항 조건 연산자 (The Ternary Operator)

기존

```
const x = 20;
let answer;
if (x > 10) {
    answer = 'greater than 10';
} else {
    answer = 'less than 10';
}
```

축약기법

```
const answer = x > 10 ? 'greater than 10' : 'less than 10';
```

React에서 사용

```
//react에서 특정 버튼을 state 값에 따라 보여지게 할 경우에 이렇게 사용할 수 있음
{editable ? (
    <a onClick={() => this.save(record.key)}> </a>
) : (
    <a onClick={() => this.edit(record.key)}> </a>
)}
```

ES6 축약코딩기법

• 2.간략 계산법 (Short-circuit Evaluation)

- 기존의 변수를 다른 변수에 할당하고 싶은 경우, 기존 변수가 null, undefined 또는 empty 값이 아닌 것을 확인 해야 합니다. (위 세가지 일 경우 에러가 뜹니다) 이를 해결 하기 위해서 긴 if문을 작성 하거나 축약 코딩으로 한 줄에 끝낼 수 있습니다

기존

```
if (variable1 != null || variable1 !== undefined || variable1 !== '') {
    let variable2 = variable1;
}
```

축약기법

```
const variable2 = variable1 || 'new';
```

Console에서 확인

```
let variable1;
let variable2 = variable1 || '';
console.log(variable2 === '');  //print true

let variable3 = 'foo';
let variable4 = variable3 || 'foo';
console.log(variable4 === 'foo');  //print true
```

• 3. 변수 선언

- 함수를 시작하기 전 먼저 필요한 변수들을 선언하는 것은 현명한 코딩 방법입니다. 축약 기법을 사용하면 여러 개의 변수를 더 효과적으로 선언함으로 시간과 코딩 스페이스를 줄일 수 있습니다.

기존

```
let x;
let y;
let z = 3;
```

축약기법

```
let x,y,z = 3;
```

ES6 축약코딩기법

- 4.For 루프

기존

```
for (let i=0; i < msgs.length; i++)
```

축약기법

```
for (let value of msgs)
```

Array.forEach 축약기법

```
function logArrayElements(element,index, array) {
    console.log('a[' + index + '] = ' + element);
}
[2,5,9].forEach(logArrayElements);
//a[0] = 2
//a[1] = 5
//a[2] = 9
```

• 5.간략 계산법 (Short-circuit Evaluation)

- 기본 값을 부여하기 위해 파라미터의 null 또는 undefined 여부를 파악할 때 short-circuit evaluation 방법을 이용해서 한줄로 작성하는 방법이 있습니다.
- Short-circuit evaluation 이란?

  두가지의 변수를 비교할 때, 앞에 있는 변수가 false 일 경우 결과는 무조건 false 이기 때문에 뒤의 변수는 확인하지 않고 return 시키는 방법.
- 아래의 예제에서는 process.env.DB_HOST 값이 있을 경우 dbHost 변수에 할당하지만, 없으면 localhost를 할당합니다.

기존

```
let dbHost;
if (process.env.DB_HOST) {
    dbHost = process.env.DB_HOST;
} else {
    dbHost = 'localhost';
}
```

축약기법

```
Const dbHost = process.env.DB_HOT || 'localhost';
```

ES6 축약코딩기법

• 6.객체 프로퍼티

- 객체 리터럴 표기법은 자바스크립트 코딩을 훨씬 쉽게 만들어 줍니다. 하지만 ES6는 더 쉬운 방법을 제안합니다. 만일 프로퍼티 이름이 key 이름과 같을 경우, 축약 기법을 활용할 수 있습니다.

기존

```
const obj = {x:x, y:y}
```

축약기법

```
const obj = {x, y}
```

ES6 축약코딩기법

• 7.Arrow(화살표) 함수

- Arrows 는 => 함수를 짧게 표현하는 방식(람다식)을 말합니다.
- 이는 C#, Java8 이나 CoffeeScript와 문법적으로 유사합니다.

기존

```
function sayHello(name) {
  console.log('Hello', name);
}

setTimeout(function() {
  console.log('Loaded')
}, 2000);

list.forEach(function(item) {
  console.log(item);
});
```

축약기법

```
sayHello = name => console.log('Hello', name);

setTimeout(() => console.log('Loaded'), 2000);

list.forEach(item => console.log(item));
```

• 8.묵시적 반환(Implicit Return)

- return 은 함수 결과를 반환 하는데 사용되는 명령어 입니다.
- 한 줄로만 작성된 arrow 함수는 별도의 return 명령어가 없어도 자동으로 반환 하도록 되어 있습니다.
- 다만, 중괄호({})를 생략한 함수여야 return 명령어도 생략 할 수 있습니다
- 한 줄 이상의 문장(객체 리터럴)을 반환 하려면 중괄호({})대신 괄호(())를 사용해서 함수를 묶어야 합니다.
  이렇게 하면 함수가 한 문장으로 작성 되었음을 나타낼 수 있습니다.

기존

```
function calcCircumference(diameter) {
  return Math.PI * diameter
}
```

축약기법

```
calcCircumference = diameter => (
  Math.PI * diameter;
)
```

• 9.파라미터 기본 값 지정하기(Default Parameter Values)

- 기존에는 if 문을 통해서 함수의 파라미터 값에 기본 값을 지정해 줘야 했습니다. ES6에서는 함수 선언문 자체에서 기본값을 지정해 줄 수 있습니다.

기존

```
function volume(l, w, h) {
  if (w === undefined)
    w = 3;
  if (h === undefined)
    h = 4;
  return l * w * h;
}
```

축약기법

```
volume = (l, w = 3, h = 4 ) => (l * w * h);
volume(2) //output: 24
```

ES6 축약코딩기법

• 10.템플릿 리터럴 (Template Literals)

- 백틱(backtick) 을 사용해서 스트링을 감싸고, ${}를 사용해서 변수를 담아 주면 됩니다.

기존

```
const welcome = 'You have logged in as ' + first + ' ' + last + '.'

const db = 'http://' + host + ':' + port + '/' + database;
```

축약기법

```
const welcome = `You have logged in as ${first} ${last}`;

const db = `http://${host}:${port}/${database}`;
```

ES6 축약코딩기법

- 11.비구조화 할당 (Destructuring Assignment)

- 데이터 객체가 컴포넌트에 들어가게 되면, unpack 이 필요합니다.

기존

```
const observable = require('mobx/observable');
const action = require('mobx/action');
const runInAction = require('mobx/runInAction');

const store = this.props.store;
const form = this.props.form;
const loading = this.props.loading;
const errors = this.props.errors;
const entity = this.props.entity;
```

축약기법

```
import { observable, action, runInAction } from 'mobx';

const { store, form, loading, errors, entity } = this.props;
```

축약기법으로 커스텀 변수명 지정

```
const { store, form, loading, errors, entity:contact } = this.props;
```

• 12.전개연산자 (Spread Operator) #1

- ES6에서 소개된 전개 연산자는 자바스크립트 코드를 더 효율적으로 사용 할 수 있는 방법을 제시합니다. 간단히는 배열의 값을 변환하는데 사용할 수 있습니다. 전개 연산자를 사용하는 방법은 점 세개(...)를 붙이면 됩니다.

기존

```
// joining arrays
const odd = [1, 3, 5];
const nums = [2 ,4 , 6].concat(odd);

// cloning arrays
const arr = [1, 2, 3, 4];
const arr2 = arr.slice();
```

축약기법

```
// joining arrays
const odd = [1, 3, 5 ];
const nums = [2 ,4 , 6, ...odd];
console.log(nums); // [ 2, 4, 6, 1, 3, 5 ]

// cloning arrays
const arr = [1, 2, 3, 4];
const arr2 = [...arr];
```

• 12.전개연산자 (Spread Operator) #2

- concat()함수와는 다르게 전개 연산자를 이용하면 하나의 배열을 다른 배열의 아무 곳에나 추가할 수 있습니다.

축약기법

```
const odd = [1, 3, 5 ];
const nums = [2, ...odd, 4 , 6];
```

- 전개 연산자는 ES6의 구조화 대입법(destructuring notation)와 함께 사용할 수도 있습니다.

축약기법

```
const { a, b, ...z } = { a: 1, b: 2, c: 3, d: 4 };
console.log(a) // 1
console.log(b) // 2
console.log(z) // { c: 3, d: 4 }
```

• 13.필수(기본) 파라미터 (Mandatory Parameter)

- 기본적으로 자바스크립트는 함수의 파라미터 값을 받지 않았을 경우, undefined로 지정합니다. 다른 언어들은 경고나 에러 메시지를 나타내기도 하죠. 이런 기본 파라미터 값을 강제로 지정하는 방법은 if 문을 사용해서 undefined일 경우 에러가 나도록 하거나, ‘Mandatory parameter shorthand’을 사용하는 방법이 있습니다.

기존

```
function foo(bar) {
  if(bar === undefined) {
    throw new Error('Missing parameter!');
  }
  return bar;
}
```

축약기법

```
let mandatory = () => {
  throw new Error('Missing parameter!');
}

let foo = (bar = mandatory()) => {
  return bar;
}
```

• 14 Object.assign() 함수

- Object.assign() 함수는 첫 번째 Object에 그 다음 Object (들)을 병합해 줍니다.

```
Object.assign(target, ...sources)
```

- object.assign 메서드의 첫 번째 인자는 target 객체입니다. 두 번째 인자부터 마지막 인자까지는 source 객체입니다. source 객체는 target 객체에 병합됩니다. 그리고 리턴값으로 타겟오브젝트를 반환합니다.

Object.assign()

```
//target에 빈 객체를 주고 source 객체를 한 개만 주면 해당 source 객체를 복제하는 것이 됩니다.
var obj = {a:1};
var copy = Object.assign({}, obj);
console.log(copy); // {a: 1}

//obj1, obj2, obj3를 각각 {}안에 병합합니다.
var obj1 = {a:1};
var obj2 = {b:2};
var obj3 = {c:3};
var newObj = Object.assign({}, obj1, obj2, obj3);
console.log(newObj); // {a: 1, b: 2, c: 3}
```

15 Promise 객체

- “A promise is an object that may produce a single value some time in the future”
- Promise는 자바스크립트 비동기 처리에 사용되는 객체입니다. 여기서 자바스크립트의 비동기 처리란 ‘특정 코드의 실행이 완료될 때까지 기다리지 않고 다음 코드를 먼저 수행하는 자바스크립트의 특성’을 의미합니다.
- new Promise(function(resolve, reject) { ... });
- Promise 객체를 생성하면 resolve와 reject의 함수를 전달받는다.
- 작업이 성공하면 resolve함수를 호출하여 resolve의 인자값을 then으로 받게 되고
- 작업에 실패하면 reject 함수를 호출하여 reject의 인자값을 catch로 받게 된다.
- 성공, 실패 여부에 관계없이 항상 처리 되게 하려면 finally로 받아서 처리할 수도 있다.

Promise

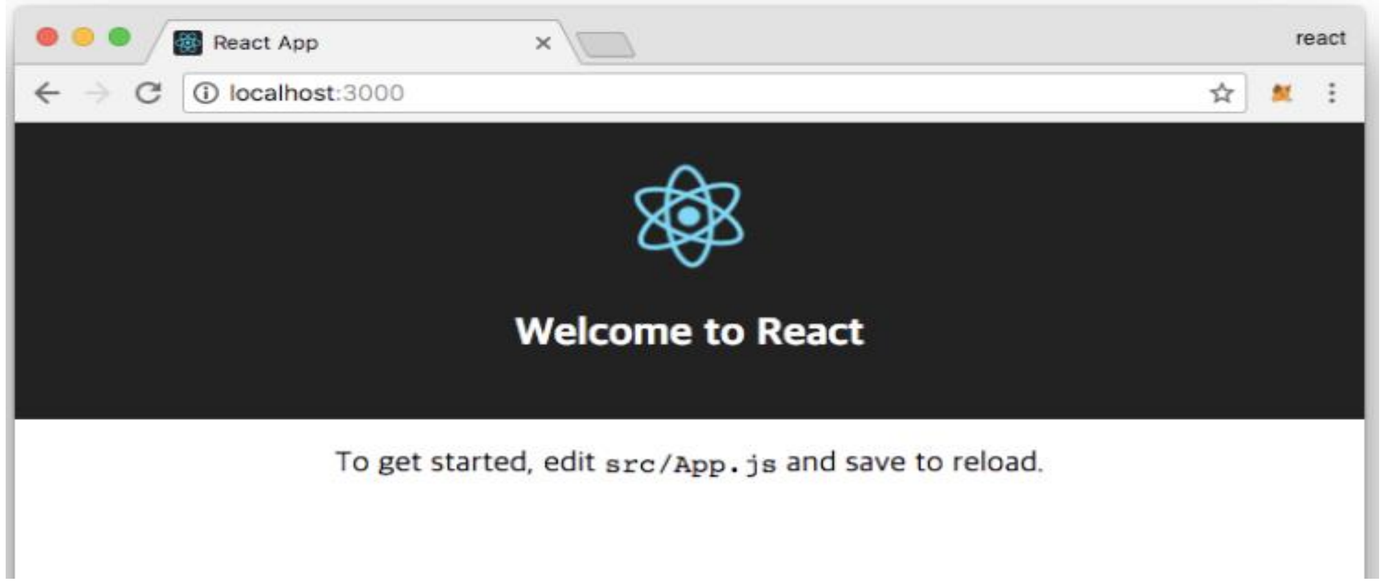
```
new Promise(function (resolve, reject) {
}).then(function (resolve) {
    //resolve 값 처리
}).catch(function (reject) {
    //reject 값 처리
}).finally(function(){
    //항상 처리
});
```

React 프로젝트 시작하기

React 프로젝트 시작하기

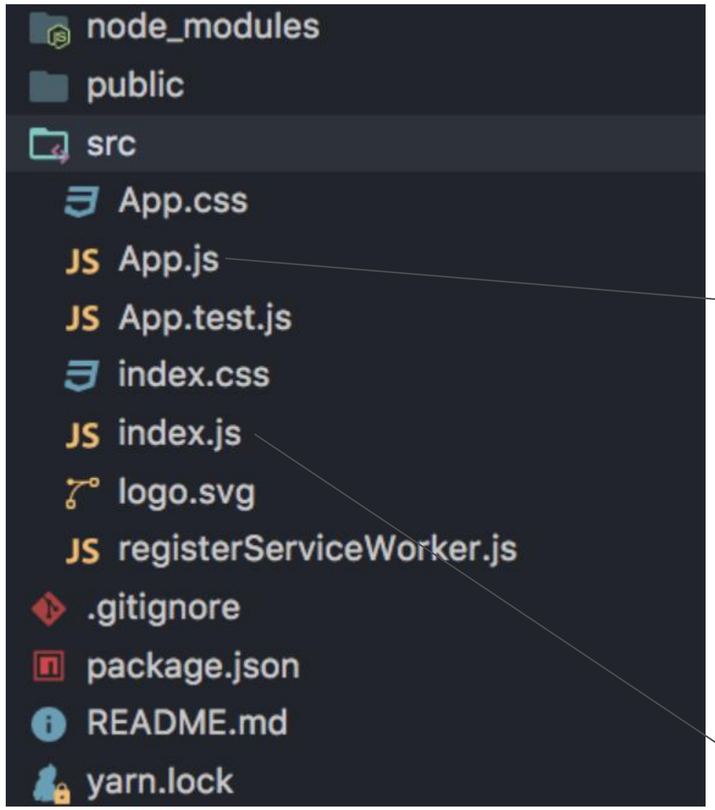
- 1.react 프로젝트 생성하기

```
create-react-app hello-react
cd hello-react
npm start
```



React 프로젝트 구조

- hello-react 프로젝트는 다음과 같이 프로젝트가 구성되어 있을 것입니다.



src/App.js

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1 className="App-title">Welcome to React</h1>
        </header>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save t
        </p>
      </div>
    );
  }
}

export default App;
```

src/index.js

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

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

public/index.html

```
<div id="root"></div>
```

Webpack : 번들링 도구

- 번들링 도구
  - 번들링 도구는 Browserify, RequireJS, Webpack 등이 대표적이다.
  - 리액트 프로젝트에서는 주로 Webpack을 사용한다.
  - 번들링 도구를 사용하면 require (또는 import)로 모듈을 불러 왔을 때 번들링 되면서 모듈들을 파일 하나로 합쳐 줍니다.
  - src/index.js를 시작으로 필요한 파일을 다 불러온 후에 파일 하나로 합쳐 주는 것이다.
  - css 파일도 Webpack의 CSS-loader가 불러온다.
  - File-loader는 웹폰트나 미디어 파일 등을 불러온다.
  - Babel-loader는 js 파일들을 불러 오면서 ES6로 작성된 코드를 ES5로 문법으로 변환해 줍니다.

JSX (XML-LIKE SYNTAX EXTENSION TO ECMASCRIPT)

- JSX 특징
  - React.js는 일반 JavaScript 문법이 아닌 JSX 문법을 사용하여 UI를 템플릿 화 합니다
  - JSX는 컴파일링 되면서 최적화 되므로 빠르다
  - Type-safe 하며 컴파일링 과정에서 에러를 감지 할 수 있다.
  - HTML과 비슷한 문법으로 작성을 하면 이를 React.createElement를 사용하는 자바스크립트 형태로 변환시켜 줍니다. (http://bit.ly/2FJsJmo)
  - JSX를 사용했을 때와 사용하지 않았을 때를 비교하고 싶다면 다음 링크들을 참고.

    JSX 사용 : https://jsfiddle.net/reactjs/69z2wepo/ , JS 사용 : https://jsfiddle.net/reactjs/5vjqabv3/
- Nested Element
  - 컴포넌트에서 여러 Element를 렌더링 해야 할 때, 그 Element들은 필수적으로 container element 안에 포함 시켜 줘야 합니다.

src/App.js

```
return  (
            <h1> Hello React </h1>
             <h2> Welcome </h2>
        );}
```

src/App.js

```
return  (
            <div>
                <h1> Hello React </h1>
                <h2> Welcome </h2>
            </div>
        );}
```

JSX (XML-LIKE SYNTAX EXTENSION TO ECMASCRIPT)

- Nested Element
  - 단순히 감싸기 위해서 새로운 div를 사용 하고 싶지 않다면
  - 오른쪽과 같이 Fragment 라는 것을 사용하면 됩니다. (이 기능은 v16.2 에 도입 되었습니다.)

src/App.js

```
import React, { Component } from 'react';
class App extends Component {
  render() {
    return (
      <div>
        <div>Hello</div>
        <div>Bye</div>
      </div>
    );
  }
}
export default App;
```

src/App.js

```
import React, { Component } from 'react';
class App extends Component {
  render() {
    return (
      <React.Fragment>
        <div>Hello</div>
        <div>Bye</div>
      </React.Fragment>
    );
  }
}
export default App;
```

JSX (XML-LIKE SYNTAX EXTENSION TO ECMASCRIPT)

- Javascript Expression

JSX 안에서, JavaScript 표현을 사용하는 방법은 매우 간단합니다. { } 로 wrapping 하면 됩니다.

src/App.js

```
    sayHey(){
       console.log("hey");
    }
    render(){
        let text = "Dev-Server"
        return  (
            <div>
                <h1> Hello React </h1>
                <h2> Welcome to {text}</h2>
                <button onClick={this.sayHey}>Click Me</button>
            </div>
        );
    }
```

- 꼭 닫아야 하는 태그

src/App.js

```
<form>
   First Name : <br/>
   <input type="text"  name="firstname" /> <br/>
</form>
```

JSX (XML-LIKE SYNTAX EXTENSION TO ECMASCRIPT)

- If-Else 문 사용 불가

JSX 안에서 사용되는 JavaScript 표현에는 If-Else 문이 사용 불가 합니다.
이에 대한 대안은 삼항연산자 (condition ? true : false) 표현을 사용하는 것입니다.

src/App.js

```
class App extends Component {
  render() {
    return (
      <div>
        {
          1 + 1 === 2 ? (<div>맞아요!</div>):(<div>틀려요!</div>)
        }
      </div>
    );
  }
}
export default App;
```

반면 AND 연산자의 경우 조건식이 true 일 때만 보여주고 false 경우 아무것도 출력하지 않을 때 사용함.

src/App.js

```
<div>
        {
          1 + 1 === 2 && (<div>맞아요!</div>)
        }
</div>
```

React의 Component

컴포넌트에서 다루는 데이터

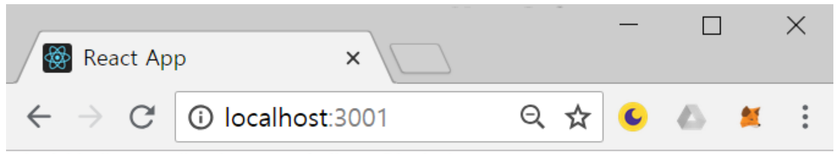
- props와 state
  - 리액트 컴포넌트에서 다루는 데이터는 props 와 state로 나누어 진다.
  - props는 부모 컴포넌트가 자식 컴포넌트에게 주는 값입니다.
  - 자식 컴포넌트에서는 props를 받아 오기 만 하고 props를 수정 할 수 없습니다.
  - 반면에 state는 컴포넌트 내부에서 선언하며 내부에서 값을 변경 할 수 있습니다.

src/MyComponent.js

```
import React, { Component } from 'react';
class MyComponent extends Component {
  render() {
    return (
      <div>
        안녕하세요! 제 이름은 <b>{this.props.name}</b> 입니다.
      </div>
    );
  }
}
export default MyComponent;
```

src/App.js

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

class App extends Component {
  render() {
    return (
      <MyComponent name="리액트" />
    );
  }
}
export default App;
```



컴포넌트에서 다루는 데이터

• defaultProps

- defaultProps는 props의 기본값을 설정할 때 사용한다.

src/MyComponent.js

```
import React, { Component } from 'react';
class MyComponent extends Component {
   (...)
}
MyComponent.defaultProps = {
  name: '기본이름'
};
export default MyComponent;
```

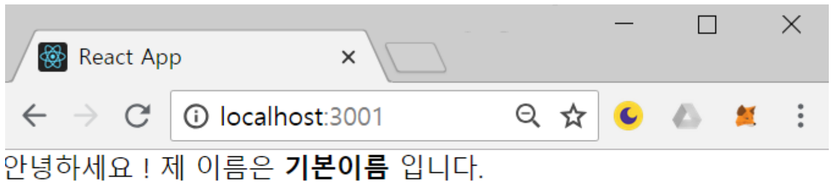
src/App.js

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

class App extends Component {
  render() {
    return (
      <MyComponent />
    );
  }
}
export default App;
```

- transform-class-properties 문법

src/MyComponent.js

```
import React, { Component } from 'react';
class MyComponent extends Component {
   static defaultProps = {
      name: '기본이름'
   }
   (...)
}

export default MyComponent;
```



컴포넌트에서 다루는 데이터

- propTypes
  - props의 type을 지정할 때는 propTypes를 사용합니다.

src/MyComponent.js

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

class MyComponent extends Component {
  render() {
    return (
      <div>
        안녕하세요! 제 이름은 <b>{this.props.name}</b> 입니다.
      </div>
    );
  }
}
MyComponent.defaultProps = {
  name: '기본이름'
};
MyComponent.propTypes = {
  name: PropTypes.string //name props 타입을 문자열로 설정함
}

export default MyComponent;
```

src/App.js

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

class App extends Component {
  render() {
    return (
      <MyComponent name={3} /> //문자열 종류
    );
  }
}
export default App;
```

문자열 종류 외의 값을 컴포넌트에 전달할 때는 { } 로 감싸야 합니다.

컴포넌트에서 다루는 데이터

- defaultProps와 propTypes
  - class 내부에서 transform-class-properties 문법을 사용하여 설정 할 수도 있음

src/MyComponent.js

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

class MyComponent extends Component {
  static defaultProps = {
      name: '기본이름'
  };
  static propTypes = {
      name: PropTypes.string
  }

  render() {
    return (
      <div>
        안녕하세요! 제 이름은 <b>{this.props.name}</b> 입니다.
      </div>
    );
  }
}
export default MyComponent;
```

src/App.js

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

class App extends Component {
  render() {
    return (
      <MyComponent name={3} /> //문자열 종류
    );
  }
}
export default App;
```

문자열 종류 외의 값을 컴포넌트에 전달할 때는 { } 로 감싸야 합니다.

컴포넌트에서 다루는 데이터

- 필수 propTypes 설정
  - propTypes를 설정할 때 뒤에 isRequired를 붙여 주면 됩니다.

src/MyComponent.js

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

class MyComponent extends Component {
  static defaultProps = {
      name: '기본이름'
  };
  static propTypes = {
      name: PropTypes.string,
      age: PropTypes.number.isRequired
  }

  render() {
    return (
     <div>
       <p>안녕하세요! 제 이름은<b>{this.props.name}</b> 입니다</p>
       <p>저는 {this.props.age} 살 입니다</p>
      </div>
    );
  }
}
export default MyComponent;
```

src/App.js

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

class App extends Component {
  render() {
    return (
       <MyComponent name="React" age={10} />
    );
  }
}
export default App;
```

문자열 종류 외의 값을 컴포넌트에 전달할 때는 { } 로 감싸야 합니다.

컴포넌트에서 다루는 데이터

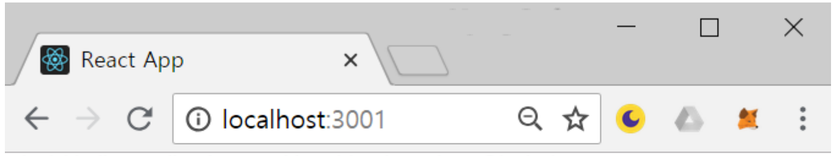
- 함수형 컴포넌트
  - props만 받아와서 보여 주기만 하는 컴포넌트의 경우에는 함수 형태로 컴포넌트를 작성할 수 있다.

src/MyComponentFunc.js

```
import React from 'react';

const MyComponentFunc = ({ name }) => {
  return (
    <div>
      안녕하세요! 제 이름은 {name} 입니다.
    </div>
  );
};

export default MyComponentFunc;
```

src/App.js

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

class App extends Component {
  render() {
    return (
      <MyComponentFunc name="함수형 컴포넌트"/>
    );
  }
}
export default App;
```



컴포넌트에서 다루는 데이터

- state
  - 컴포넌트 내부에서 읽거나 변경할 수 있는 값을 사용 하려면 state를 사용 해야 합니다.
  - state는 언제나 기본값을 미리 설정 가능하며, this.setState() 메서드를 통해서만 값을 변경할 수 있음
- state 초기값 설정
  - constructor 메서드 안에서 state의 초기값을 설정할 수 있다.

src/MyComponent.js

```
import React, { Component } from 'react';
class MyComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {
      number: 0
    }
  }
  render() {
    return (
      <div>
        <p>안녕하세요! 제 이름은 <b>{this.props.name}</b> 입니다.</p>
        <p>number 값은 : {this.state.number} </p>
      </div>
    );
  }
}
export default MyComponent;
```

컴포넌트에서 다루는 데이터

- state 값 변경
  - state 값을 업데이트 할 때는 this.setState() 메서드를 사용합니다.

```
this.setState({
    수정할 필드 이름: 값,
    수정할 또 다른 필드 이름 : 값
});
```

src/MyComponent.js

```
import React, { Component } from 'react';
class MyComponent extends Component {
  render() {
    return (
      <div>
        <p>안녕하세요! 제 이름은 <b>{this.props.name}</b> 입니다.</p>
        <p>number 값은 : {this.state.number} </p>
        <button onClick={() => {
             this.setState({
                number: this.state.number + 1
             })
          }
        }>더하기</button>
      </div>
    );
  }
}
export default MyComponent;
```

컴포넌트에서 다루는 데이터

- state를 constructor에서 꺼내기

src/MyComponent.js

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

class MyComponent extends Component {
  static defaultProps = {
      name: '기본이름'
  };
  static propTypes = {
      name: PropTypes.string,
      age: PropTypes.number.isRequired
  }
  state = {
     number: 0
  }

  render() {
    return (..);
  }
}

export default MyComponent;
```

컴포넌트에서 다루는 데이터

- Counter 예제

src/Counter.js

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

class Counter extends Component {
  state = { number: 0}

  handleIncrease = () => {
    this.setState({ number: this.state.number + 1 });
  }
  handleDecrease = () => {
    this.setState({ number: this.state.number - 1 });
  }
  render() {
    return (
      <div>
        <h1>카운터</h1>
        <div>값: {this.state.number}</div>
        <button onClick={this.handleIncrease}>+</button>
        <button onClick={this.handleDecrease}>-</button>
      </div>
    );
  }
}
export default Counter;
```

src/App.js

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

class App extends Component {
  render() {
    return (
      <Counter />
    );
  }
}
export default App;
```

React의 Event Handling

Event

- Event를 사용할 때 주의사항
  1.이벤트 이름은 camelCase로 작성합니다. => html의 onclick은 리액트에서는 onClick
  2.이벤트에 실행 할 자바스크립트 코드를 전달하는 것이 아니라, 함수 형태의 객체를 전달합니다.
  3.DOM 요소에만 이벤트를 설정할 수 있습니다.
  : div, button, input, form, span 등 DOM 요소에는 이벤트를 설정할 수 있지만, 직접 만든 컴포넌트에는 이벤트를 설정할 수 없습니다

src/MyComponent.js – render 메서드의 button 요소

```
import React, { Component } from 'react';
class MyComponent extends Component {
  render() {
    return (
      <div>
         <button onClick={() => {
             this.setState({
                number: this.state.number + 1
             })
          }
        }>더하기</button>
      </div>
    );
  }
}
export default MyComponent;
```

Event

- 컴포넌트 생성 및 불러오기

  1.EventPractice.js 생성과 App.js에서 EventPractice 불러오기

src/EventPractice.js

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

class EventPractice extends Component {
   render() {
      return (
         <div>
           <h1>이벤트 연습</h1>
         </div>
      );
   }
}

export default EventPractice;
```

src/App.js

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

class App extends Component {
  render() {
    return (
      <EventPractice />
    );
  }
}

export default App;
```

이벤트 연습

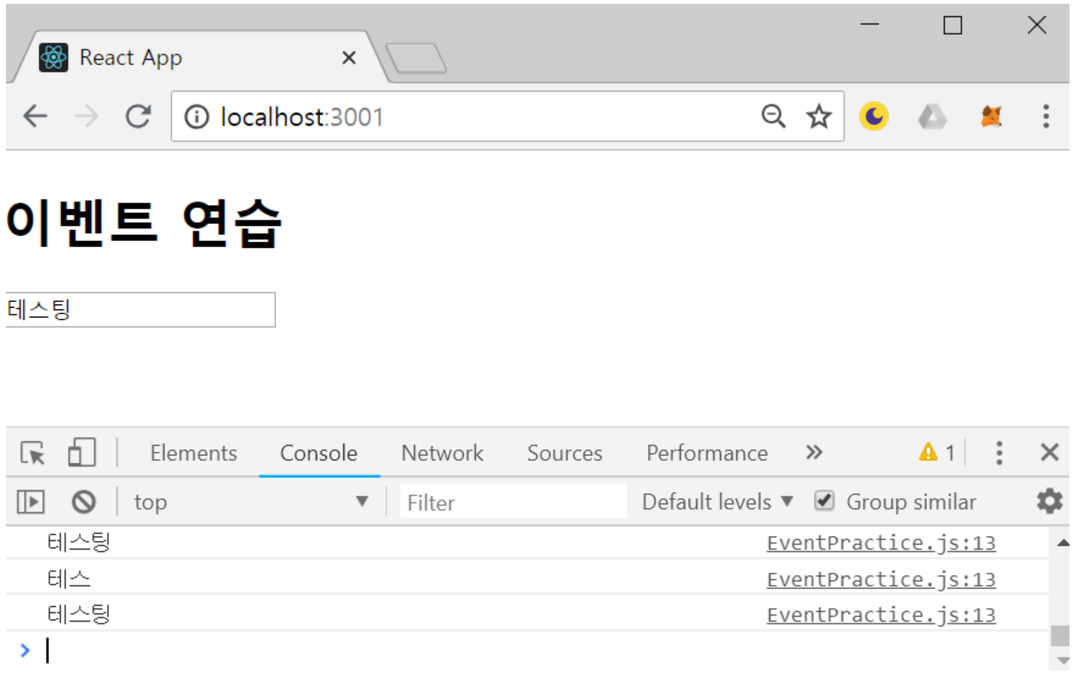
Event

- 이벤트 핸들링

  2.onChange 이벤트 설정

src/EventPractice.js

```
import React, { Component } from 'react';
class EventPractice extends Component {
  render() {
    return (
      <div>
        <h1>이벤트 연습</h1>
        <input
         type="text"
         name="message"
         placeholder="아무거나 입력해보세요"
         onChange={
           (e) => {
             console.log(e.target.value);
           }
         }
        />
      </div>
    );
  }
}
export default EventPractice;
```



Event

- 이벤트 핸들링

3. input에 입력한 값을 state에 저장

src/EventPractice.js

```
import React, { Component } from 'react';
class EventPractice extends Component {
  state = {  message: '' }
  render() {
    return (
      <div>
         <h1>이벤트 연습</h1>
         <input
          type="text"
          name="message"
          placeholder="아무거나 입력해보세요"
          value = {this.state.message}
          onChange={
            (e) => {
               this.setState({
                  message: e.target.value
               })
            }
          }
        />
      </div>
    );
  }
}
export default EventPractice;
```

4. 입력한 값이 state에 저장되었는지 확인

src/EventPractice.js

```
import React, { Component } from 'react';
class EventPractice extends Component {
  state = {  message: '' }
  render() {
    return (
      <div>
         <h1>이벤트 연습</h1>
         <input
           (...)
         />
         <button onClick={
                () => {
                    alert(this.state.message);
                    this.setState({
                        message:''
                    });
                }
            }>확인 </button>
      </div>
    );
  }
}
export default EventPractice;
```

Event

- 이벤트 핸들링

5.handleChange(), handleClick() 메서드 추가 하기

: 컴포넌트에 임의 메서드를 만들면 기본적으로 this에 접근할 수 없기 때문에 constructor에서 각 메서드를 this와 binding 해 주어야 합니다. 즉 메서드에서 this를 사용할 수 있도록 메서드에 this를 binding 해주는 것이다.

src/EventPractice.js

```
import React, { Component } from 'react';
class EventPractice extends Component {
    state = {  message: '' }
    constructor(props) {
        super(props);
        this.handleChange = this.handleChange.bind(this);
        this.handleClick = this.handleClick.bind(this);
    }
    handleChange(e){ }
    handleClick() { }
    render() {..}
}
```

Event

- 이벤트 핸들링

6.handleChange(), handleClick() 메서드 구현 및 handler method 연결

src/EventPractice.js

```
import React, { Component } from 'react';
class EventPractice extends Component {
     constructor(props) {
        this.handleChange = this.handleChange.bind(this);
        this.handleClick = this.handleClick.bind(this);
     }
     handleChange(e){
        this.setState({
            message:e.target.value
        });
     }
     handleClick() {
        alert(this.state.message);
        this.setState({
            message:''
        });
    }
    <input ... onChange={this.handleChange} />
    <button onClick={this.handleClick}>확인</button>
}
```

Event

- 이벤트 핸들링

7.transform-class-properties 문법 사용 : handleChange(), handleClick() 메서드 구현
: 바벨의 transform-class-properties 문법을 사용하여 화살표 함수 형태로 메서드 정의

src/EventPractice.js

```
import React, { Component } from 'react';
class EventPractice extends Component {

     handleChange = (e) => {
        this.setState({
            message:e.target.value
        });
     }
     handleClick = () => {
        alert(this.state.message);
        this.setState({
            message:''
        });
    }
    <input ... onChange={this.handleChange} />
    <button onClick={this.handleClick}>확인</button>
}
```

Event

- 이벤트 핸들링

8.여러 개의 input 과 state 관리
=> [e.target.name]: e.target.value

src/EventPractice.js

```
import React, { Component } from 'react';
class EventPractice extends Component {
     state = {  message: '', username: '' }

     handleChange = (e) => {
        this.setState({
            [e.target.name]:e.target.value
        });
     }
     handleClick = () => {
        alert(this.state.username + ':' + this.state.message);
        this.setState({
            message:'',
            username:''
        });
    }
    <input name="message" value={this.state.username} onChange={this.handleChange} />
    <input name="username" value={this.state.message} onChange={this.handleChange} />
    <button onClick={this.handleClick}>확인</button>
}
```

Event

- 이벤트 핸들링

9.onKeyPress 이벤트 핸들링
: input에서 엔터키를 눌렀을 때 handleClick 메서드를 호출한다.

src/EventPractice.js

```
import React, { Component } from 'react';
class EventPractice extends Component {
     state = {  message: '', username: '' }

     handleKeyPress = (e) => {
        if (e.key === 'Enter') {
            this.handleClick();
        }
     }
    <input value={this.state.username} onChange={this.handleChange} />
    <input value={this.state.message} onChange={this.handleChange} onKeyPress={this.handleKeyPress}  />
    <button onClick={this.handleClick}>확인</button>
}
```

ref:DOM에 이름 달기

DOM

- DOM에 접근하기
  - HTML에서는 id를 부여하여 DOM에 접근한다.

    document.getElementById('root')

  - 리액트에서는 ref를 사용하여 DOM에 접근한다.
    <input ref={ (ref) => { this.input=ref } />

- DOM에 접근하는 예제
  - HTML DOM에 ref 사용 : 특정 input에 focus 주기

    <input ref={ (ref) => { this.input=ref } />

    this.input.focus()

  - 컴포넌트에 ref 사용 : 스크롤 박스 조작하기

    <ScrollBox ref={ (ref) => { this.scrollBox=ref }}

    this.scrollBox.scrollToBottom()

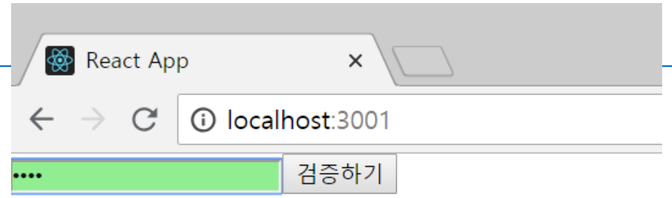
DOM

- 컴포넌트 생성 및 불러오기

1.ValidationPractice.js 와 ValidationPractice.css 생성

src/ValidationPractice.js

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

class ValidationPractice extends Component {
  state = {
    password: '',
    clicked: false,
    validated: false
  }
  handleChange = (e) => {
    this.setState({
      password: e.target.value
    });
  }
  handleButtonClick = () => {
    this.setState({
      clicked: true,
      validated: this.state.password === '0000'
    })
  }
```



src/ValidationPractice.js

```
render() {
    return (
      <div>
        <input
          type="password"
          value={this.state.password}
          onChange={this.handleChange}
          className={this.state.clicked &&
  this.state.validated ? 'success' : 'failure'}
        />
        <button onClick={this.handleButtonClick}>검증하기
        </button>
      </div>
    );
  }
}
export default ValidationPractice;
```

src/ValidationPractice.css

```
.success { background-color: lightgreen;}

.failure { background-color: lightcoral;}
```

DOM

- HTML DOM에 ref 속성 추가하기

  2.input 에 focus를 주기 위해서 ref 속성 추가

src/ValidationPractice.js

```
import React, { Component } from 'react';
class ValidationPractice extends Component {
  state = {
    password: '',
    clicked: false,
    validated: false
  }
  handleChange = (e) => {
    this.setState({
      password: e.target.value
    });
  }
  handleButtonClick = () => {
    this.setState({
      clicked: true,
      validated: this.state.password === '0000'
    })
    this.input.focus();
  }
```

src/ValidationPractice.js

```
render() {
    return (
      <div>
        <input
          ref={(ref) => this.input=ref}
          type="password"
          value={this.state.password}
          onChange={this.handleChange}
          className={this.state.clicked &&
(this.state.validated ? 'success' : 'failure')}
        />
        <button onClick={this.handleButtonClick}>검증하기
</button>
      </div>
    );
  }
}
export default ValidationPractice;
```

DOM

- 사용자 정의 컴포넌트에 ref 속성 추가하기

1.ScrollBox 컴포넌트 생성 App.js에서 ScrollBox 불러오기

src/ScrollBox.js

```
import React, { Component } from 'react';
class ScrollBox extends Component {
  render() {
    const style = {
      border: '1px solid black',
      height: '300px',
      width: '300px',
      overflow: 'auto',
      position: 'relative'
    };
    const innerStyle = {
      width: '100%',
      height: '650px',
      background: 'linear-gradient(white, black)'
    }
   return (
         <div
           style={style}
           ref={(ref) => { this.box=ref }}>
           <div style={innerStyle}/>
         </div>
       );
     }
   }
export default ScrollBox;
```

src/App.js

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

class App extends Component {
  render() {
    return (
      <div>
        <ScrollBox />
      </div>
    );
  }
}

export default App;
```

DOM

- 사용자 정의 컴포넌트에 ref 속성 추가하기 - 메서드 생성

1.ScrollBox 컴포넌트에 스크롤바를 맨 아래쪽으로 내리는 메서드 생성

- scrollTop : 세로 스크롤바 위치 (0~350)
- scrollHeight : 스크롤 박스 내부의 높이 (650)
- clientHeight : 스크롤 박스 외부의 높이 (300)

src/ScrollBox.js

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

class ScrollBox extends Component {
  scrollToBottom = () => {
    const { scrollHeight, clientHeight } = this.box;
    /* 위 코드에는 비구조화 할당(destructuring assignment) 문법 사용
      const scrollHeight = this.box.scrollHeight;
      const clientHeight = this.box.clientHeight;
    */
    this.box.scrollTop = scrollHeight - clientHeight;
  }

  render() {
    ...
  }
}
export default ScrollBox;
```

DOM

- 사용자 정의 컴포넌트에 ref 속성 추가하기 - 메서드 호출

  1.App 컴포넌트에서 ScrollBox 컴포넌트 호출

  : App 컴포넌트에서 ScrollBox에 ref를 추가하고 버튼을 만들어 클릭하면 ScrollBox 컴포넌트의 scrollToBottom 메서드를 실행한다.

src/App.js

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

class App extends Component {
  render() {
    return (
      <div>
        <ScrollBox ref={(ref) => this.scrollBox = ref}/>
        <button onClick={() => this.scrollBox.scrollToBottom()}>
          맨 밑으로
        </button>
      </div>
    );
  }
}

export default App;
```

컴포넌트 반복

컴포넌트 반복 : map() 함수

- map() 함수
  - arr.map(callback, [thisArg])

    1) callback : 새로운 배열의 요소를 생성하는 함수이며, 파라미터는 다음 3가지

    - currentValue : 현재 처리하고 있는 요소
    - index : 현재 처리하고 있는 요소의 index 값
    - array : 현재 처리하고 있는 원본 배열

    2) thisArg(선택항목) : callback 함수 내부에서 사용할 this 레퍼런스

src/App.js

```
var numbers = [1,2,3,4,5]
var processed = numbers.map(function(num){
    return num * num;
});
console.log(processed);
```

src/App.js

```
const numbers = [1,2,3,4,5]
const processed = numbers.map(num =>
        num * num;
);
console.log(processed);
```

컴포넌트 반복 : map() 함수

- 컴포넌트 생성 및 불러오기

  1.IterationPractice.js 생성 App.js에서 IterationPractice 불러오기

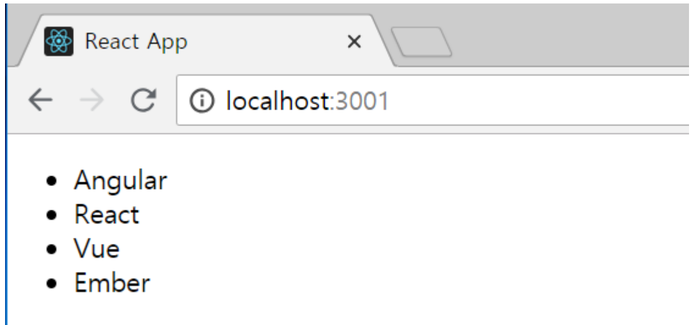
src/IterationPractice.js

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

class IterationPractice extends Component {
    render() {
        const names =
           ['Angular','React','Vue','Ember'];
        const nameList = names.map(
           (name) => (<li>{name}</li>)
        );

        return(
            <ul>
                {nameList}
            </ul>
        );
    }
}
export default IterationPractice;
```

src/App.js

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

class App extends Component {
  render() {
    return (
      <div>
        <IterationPractice />
      </div>
    );
  }
}

export default App;
```

```
▶Warning: Each child in an array or iterator should have a unique "key" prop.

Check the render method of `IterationPractice`. See https://fb.me/react-warning-
    in li (at IterationPractice.js:6)
    in IterationPractice (at App.js:21)
    in App (at index.js:7)
```



컴포넌트 반복 : map() 함수

- key
  - 리액트에서 key는 배열을 렌더링 했을 때 어떤 엘리먼트에 변동이 있었는지 알아낼 때 사용합니다.
    1) key가 없으면
       - 가상 DOM을 비교하는 과정에서 리스트를 순차적으로 모두 비교하여 변화를 감지합니다.
    2) key가 있으면
       - key값을 사용하여 어떤 변화가 일어 났는지 빠르게 감지할 수 있다.

src/IterationPractice.js

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

class IterationPractice extends Component {
    render() {
        const names =
           ['Angular','React','Vue','Ember'];
        const nameList = names.map(
           (name,index) => (<li key={index}>{name}</li>)
        );

        return(
            <ul>
                {nameList}
            </ul>
        );
    }
}
export default IterationPractice;
```

컴포넌트 반복 : map() 함수

- 배열에 데이터 추가 기능 구현
  - 1. 초기 state 설정하기 : const로 정의 했던 배열을 state에 저장한다.
    2. 데이터 추가를 위한 input과 button을 렌더링 하고, 이벤트 핸들러 메서드 구현한다.

src/IterationPractice.js

```
import React, {Component} from 'react';
class IterationPractice extends Component {
    state = {
        names: ['Angular','React','Vue','Ember'],
        name: ''
    };

    handleChange = (e) => {
        this.setState({
          name: e.target.value
        });
    }

    handleInsert = () => {
        // names 배열에 값을 추가하고, name 값을 초기화합니다
        this.setState({
          names: this.state.names.concat(this.state.name),
          name: ''
        });
    }
```

src/IterationPractice.js

```
    render() {
            const nameList = this.state.names.map(
            (name, index) => (
                <li key={index}>
                    {name}
                </li>)
            );

            return (
                <div>
                    <input
                    onChange={this.handleChange}
                    value={this.state.name}/>
                    <button onClick={this.handleInsert}>
                    추가</button>
                    <ul>
                       {nameList}
                    </ul>
                </div>
            );
      }
}
export default IterationPractice;
```

컴포넌트 반복 : map() 함수

- 배열에 데이터 삭제 기능 구현 #1 – spread operator 사용
- 1. 아이템을 두 번 클릭(onDoubleClick)하면 제거 하도록 기능 구현
  2. handleRemove메서드에서 전개연산자(spread operator) … 를 사용하여 뒤에 위치한 배열값을 그대로 꺼내서 현재 배열에 복사한다.

src/IterationPractice.js

```
import React, {Component} from 'react';
class IterationPractice extends Component {
    state = {
        names: ['Angular','React','Vue','Ember'],
        name: ''
    };

    (...)

    handleRemove = (index) => {
        const {names} = this.state;
        this.setState({
            names:[
            ...names.slice(0,index),
            ...names.slice(index+1,names.length)
            ]
        });
    }
```

src/IterationPractice.js

```
    render() {
            const nameList = this.state.names.map(
            (name, index) => (
                <li key={index}
                    onDoubleClick={() => this.handleRemove(index)}>
                    {name}
                </li>)
            );

            return (
                <div>
                    (...)
                </div>
            );
    }
}
export default IterationPractice;
```

컴포넌트 반복 : map() 함수

- 배열에 데이터 삭제 기능 구현 #2 – filter 함수
- 1. 아이템을 두 번 클릭(onDoubleClick)하면 제거 하도록 기능 구현
  2. handleRemove 핸들러 메서드에서는 클릭한 아이템과 일치하지 않는 것들만 filtering 해서 보여준다.

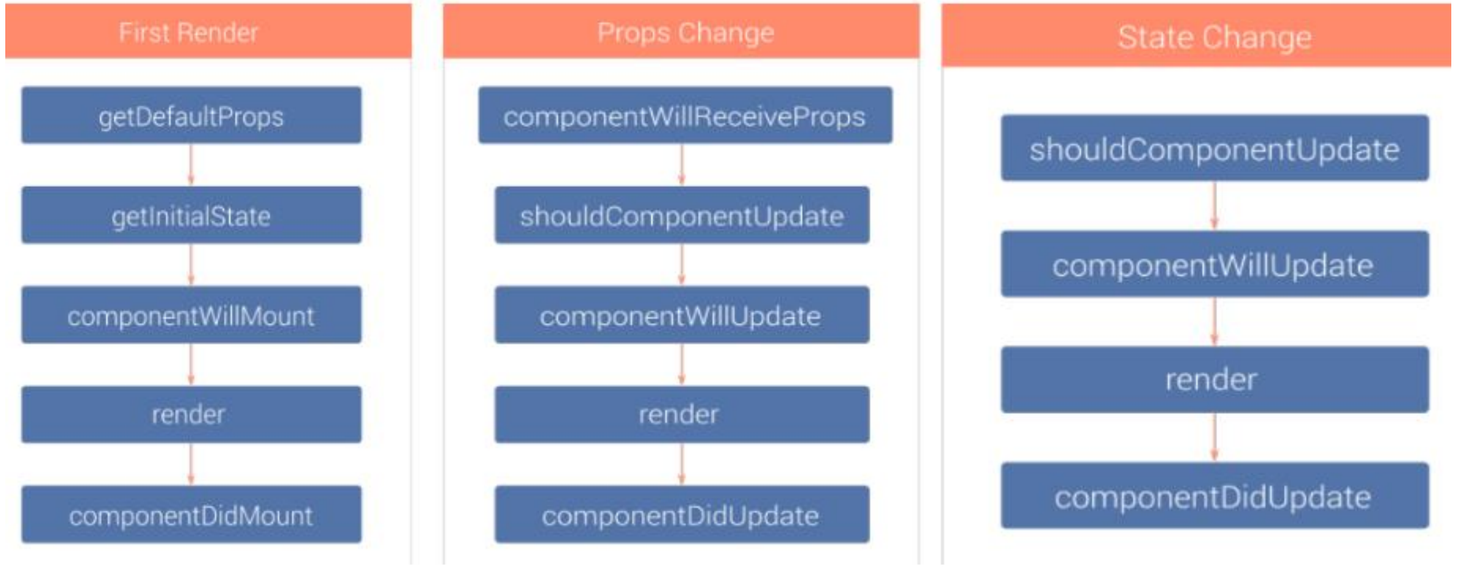
src/IterationPractice.js

```
import React, {Component} from 'react';
class IterationPractice extends Component {
    state = {
        names: ['Angular','React','Vue','Ember'],
        name: ''
    };

    (...)

  handleRemove = (index) => {
    // 편의상 names에 대한 레퍼런스를 미리 만듭니다
    const { names } = this.state;
    this.setState({
      // filter를 통해 index 번째를 제외한 원소만 있는 새 배열 생성
          names: names.filter((item, i) => i !== index)
    });
  }
```

src/IterationPractice.js

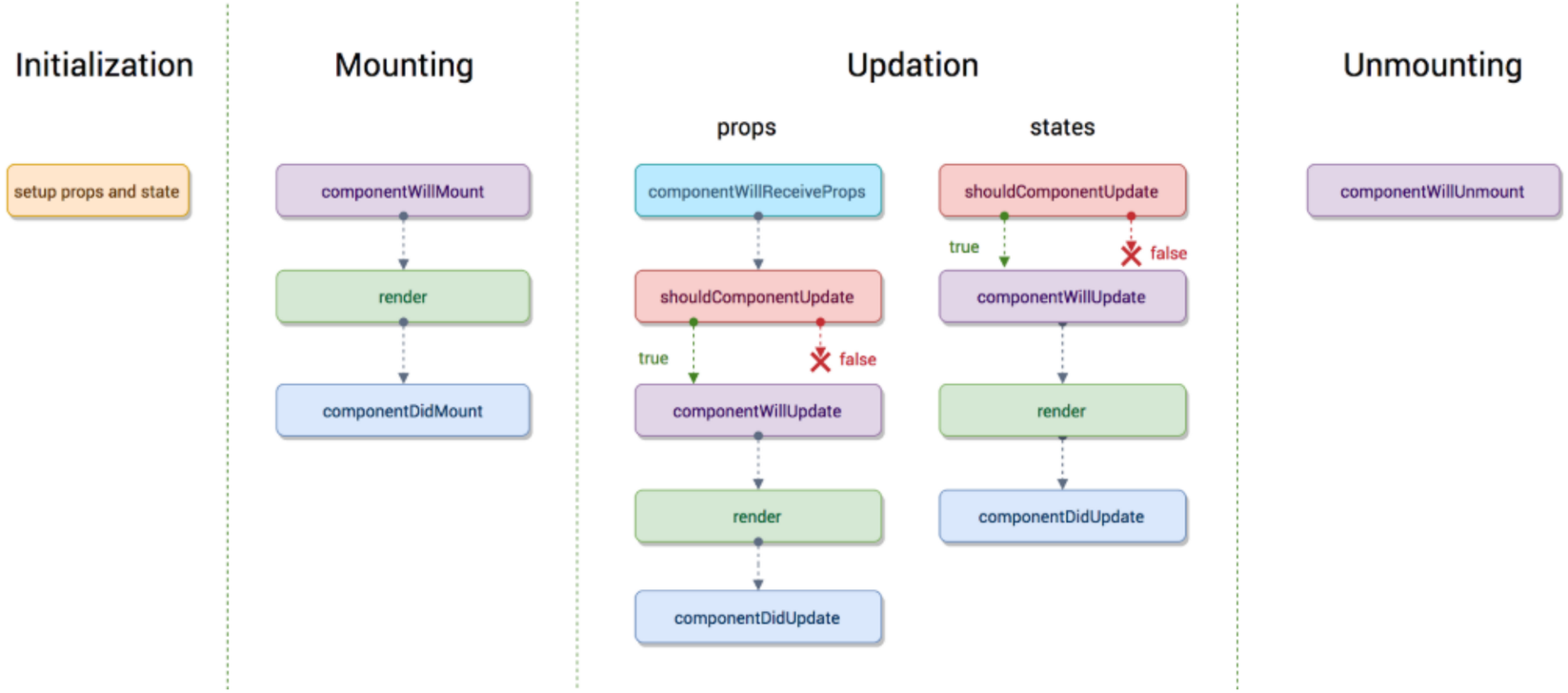
```
    render() {
            const nameList = this.state.names.map(
            (name, index) => (
                <li key={index}
                    onDoubleClick={() =>
                        this.handleRemove(index)}>
                    {name}
                </li>)
            );

            return (
                <div>
                    (...)
                </div>
            );
      }
}
export default IterationPractice;
```

React - Life Cycle



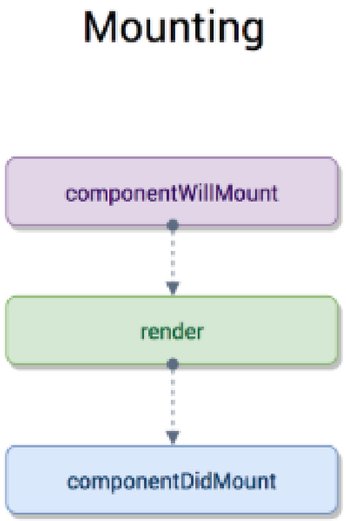
Life Cycle

- Life Cycle 메서드
  - 리액트 컴포넌트를 만들 때 컴포넌트 생명주기의 특정 시점에 자동으로 호출될 메서드를 선언할 수 있다. 각 생명주기 API를 이해하고 있으면 컴포넌트가 생성 또는 삭제될 때 특정한 작업을 수행할 수 있다.
  - Life Cycle 메서드의 종류는 10가지이며, Will 접두사가 붙은 메서드는 어떤 작업을 작동하기 전에 실행되는 메서드이고, Did 접두사가 붙은 메서드는 어떤 작업을 작동한 후에 실행되는 메서드이다.



Life Cycle

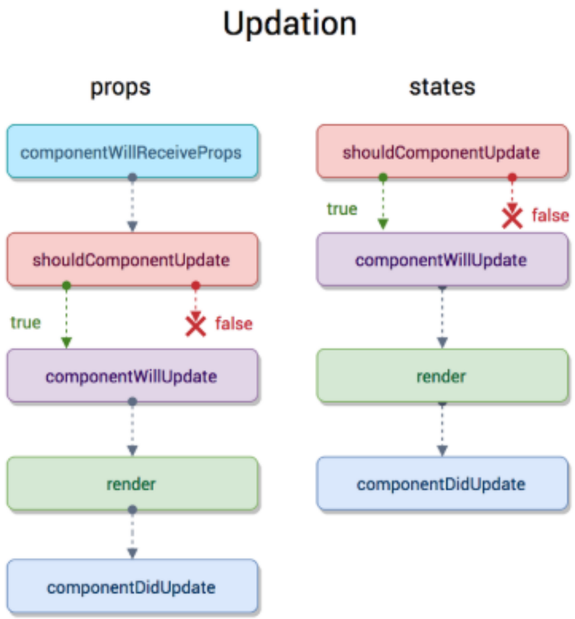
- Mounting
  - DOM이 생성되고 웹 브라우저상에 나타나는 것을 마운트(mount) 라고 합니다. .



- **componentWillMount**: 렌더링을 수행하기 전에 호출된다. 이 단계에서 상태를 설정하더라도 렌더링이 다시 트리거 되지 않는다.
- **render** : UI를 렌더링하는 메서드입니다.
- **componentDidMount**: 렌더링을 수행한 후에 호출된다. 이 시점에서 컴포넌트에 대한 DOM 표현이 생성되기 때문에 데이터 가져오기 등의 작업을 할 수 있다.

Life Cycle

- Update
  - props나 state가 바뀔 때, 부모 컴포넌트가 Re렌더링 될 때 컴포넌트를 업데이트 한다.



1) 속성 변경(Updating - prop)

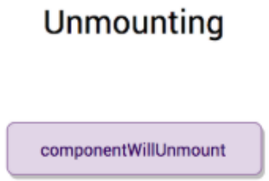
- **componentWillReceiveProps**: 컴포넌트가 새 속성을 받을 때 호출된다. 이 함수 안에 this.setState()를 호출해도 렌더링이 트리거 되지 않는다.
- **shouldComponentUpdate**: render 함수보다 먼저 호출되는 함수이며, 해당 컴포넌트의 렌더링을 생략할 수 있는 기회를 제공한다.
- **componentWillUpdate**: 새로운 속성이나 상태를 수신하고 렌더링하기 전에 호출된다. 등록된 업데이트에만 이용해야 하며, 업데이트 자체를 트리거 하지 않아야 하므로 this.setState를 통한 상태 변경은 허용되지 않는다.
- **componentDidUpdate**: 컴포넌트 업데이트가 DOM으로 플러시 된 후에 호출된다.

2) 상태 변경(Updating - state)

속성 변경과 거의 동일한 생명주기를 가지고 있지만, componentWillReceiveProps에 해당하는 메서드는 없다. 따라서 shouldComponentUpdate 부터 시작된다.

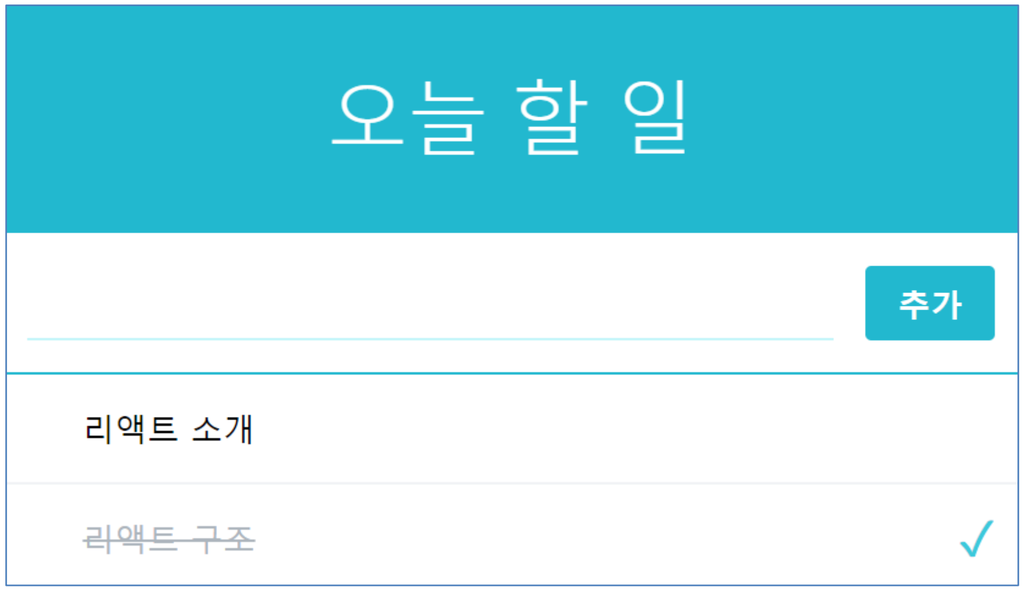
Life Cycle

- Unmounting
  - 마운트의 반대 과정으로 컴포넌트를 DOM에서 제거하는 것을 언마운트 라고 합니다. .



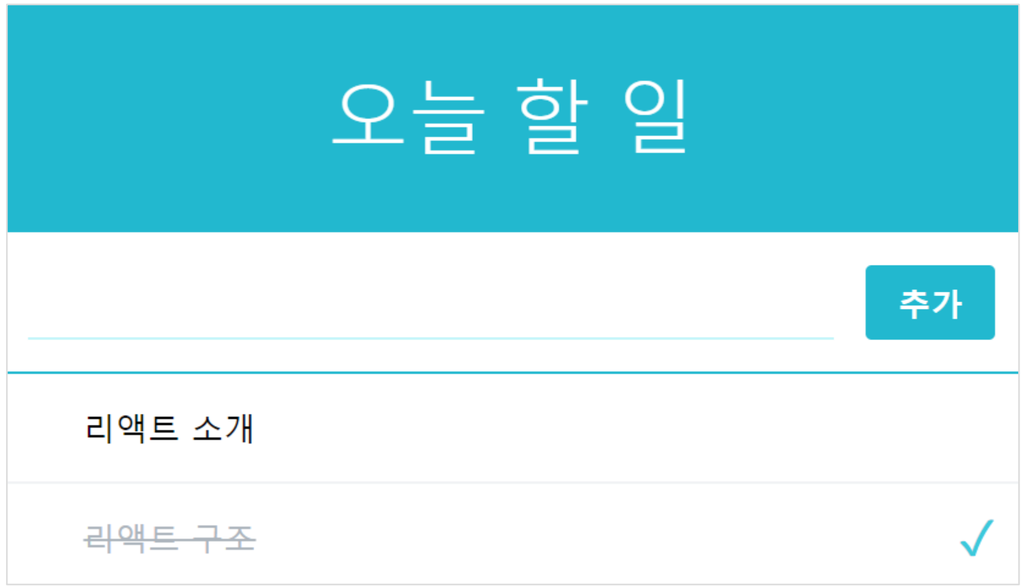
- **componentWillUnmount:** 컴포넌트가 DOM에서 언마운트 되기 전에 호출된다.

일정관리(Todo-List) App

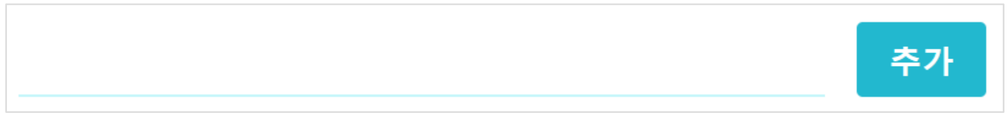


Todo-List

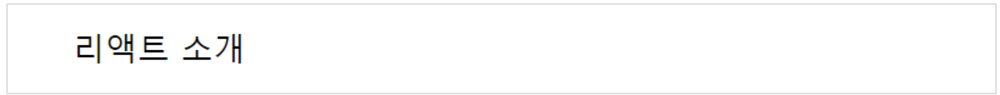
1. TodoListTemplate : 전체적인 틀을 설정



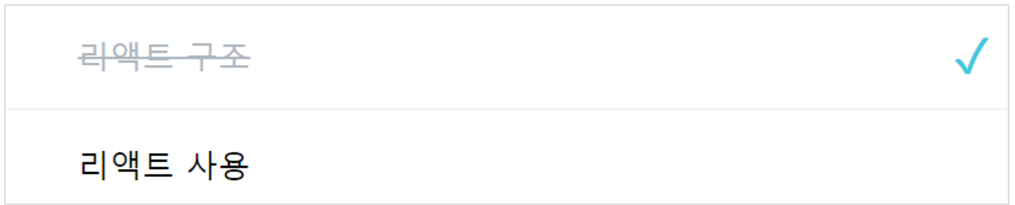
2. Form : 일정을 추가할 때 사용하는 input 컴포넌트



3. TodoItem : 각 일정을 렌더링하는 컴포넌트 , 클릭하면 체크되면서 줄을 긋고, 지우기 버튼을 누르면 일정을 화면에서 제거합니다.



4. TodoItemList : 일정 데이터가 담긴 배열을 TodoItem 컴포넌트로 구성된 배열로 변환해서 렌더링하는 컴포넌트입니다.



Todo-List : 프로젝트 생성

1. 새로운 프로젝트 생성 : todo-list

```
create-react-app  todo-list
```

2. 프로젝트 환경설정

```
cd todo-list

npm start
```

Todo-List : 프로젝트 초기화

1. App.js 수정

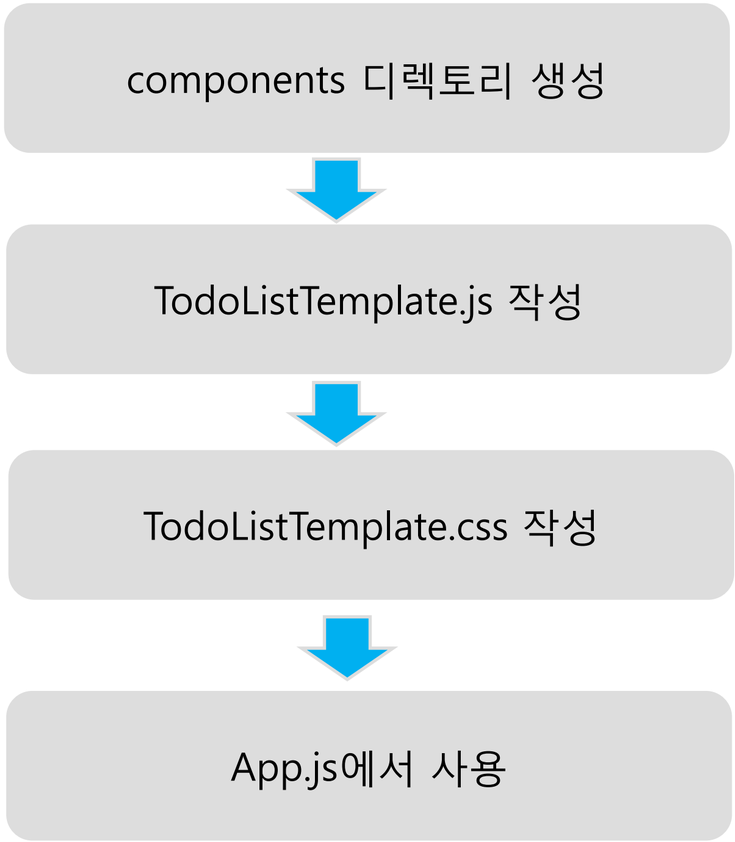
src/App.js

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

class App extends Component {
  render() {
    return (
      <div>
        App
      </div>
    );
  }
}

export default App;
```

App.css, App.test.js, logo.svg 파일도 제거 하세요

Todo-List : 첫번째 컴포넌트, TodoListTemplate

컴포넌트 작성 순서



첫번째 컴포넌트 , TodoListTemplate

components 디렉토리에 다음 파일을 생성하세요:

: src/components/TodoListTemplate.js

: src/components/TodoListTemplate.css

이 컴포넌트는 템플릿의 역할을 합니다. 중앙에 흰색 박스가 있고, 타이틀이 보여지고, 그 아래에는 폼과 리스트가 있습니다.

Todo-List : 첫번째 컴포넌트, TodoListTemplate

1. TodoListTemplate 컴포넌트

src/components/TodoListTemplate.js

```
import React from 'react';
import './TodoListTemplate.css';

const TodoListTemplate = ({form, children}) => {
  return (
    <main className="todo-list-template">
      <div className="title">
        오늘 할 일
      </div>
      <section className="form-wrapper">
        {form}
      </section>
      <section className="todos-wrapper">
        { children }
      </section>
    </main>
  );
};
export default TodoListTemplate;
```

이 컴포넌트는 함수형 컴포넌트 입니다.

파라미터로 받는 것은 props 이며, 이를 비구조화 할당 (destructuring assignment) 하여 원래 (props) => { ... } 를 해야 하는 것을 ({form, children}) => { ... } 형태로 작성 하였습니다.

form은, input과 button이 있는 컴포넌트를 렌더링 할 때 사용하며, 이것도 children을 사용 하듯이 JSX 형태로 전달 합니다.

```
<TodoListTemplate form={<div>이렇게 말이죠.</div>}>
    <div>여기엔 children 자리구요.</div>
</TodoListTemplate>
```

Todo-List : 첫번째 컴포넌트, TodoListTemplate

2. TodoListTemplate Style

src/components/TodoListTemplate.css

```
.todo-list-template {
  background: white;
  width: 512px;
  box-shadow: 0 3px 6px rgba(0,0,0,0.16), 0 3px 6px rgba(0,0,0,0.23);
  margin: 0 auto; /* 페이지 중앙 정렬 */
  margin-top: 4rem;
}
.title {
  padding: 2rem;
  font-size: 2.5rem;
  text-align: center;
  font-weight: 100;
  background: #22b8cf;;
  color: white;
}
.form-wrapper {
  padding: 1rem;
  border-bottom: 1px solid #22b8cf;
}
.todos-wrapper {
  min-height: 5rem;
}
```

src/index.css

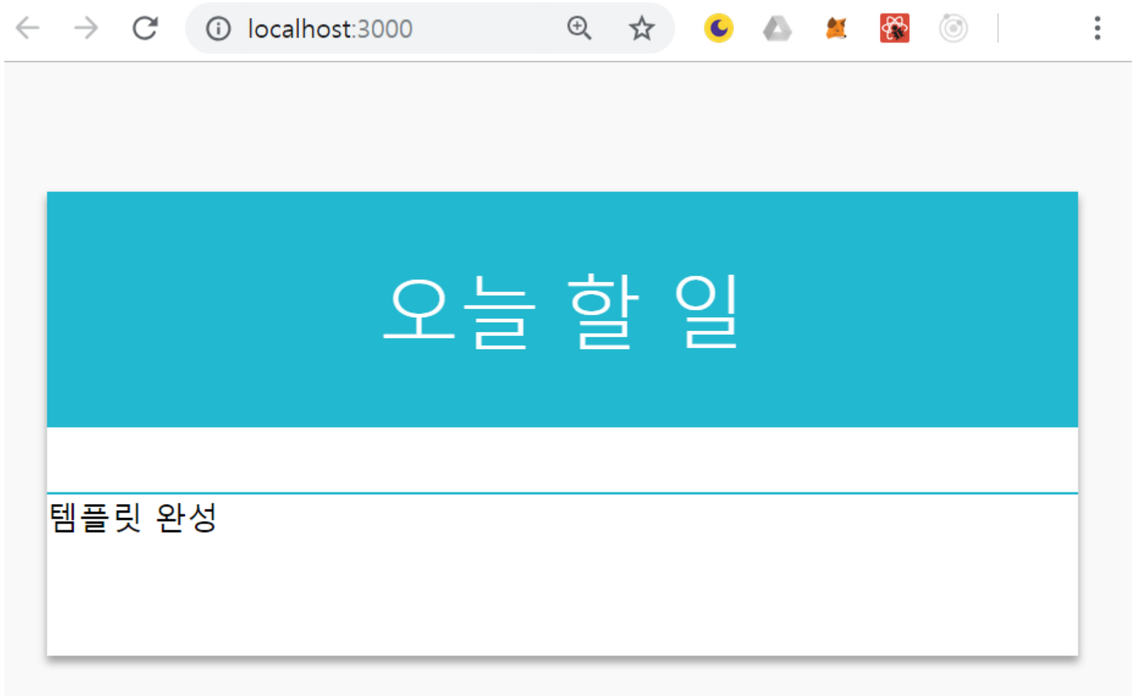
```
body {
  margin: 0;
  padding: 0;
  font-family: sans-serif;
  background: #f9f9f9;
}
```

Todo-List : 첫번째 컴포넌트, TodoListTemplate

3. App.js 에서 TodoListTemplate 컴포넌트 사용하기

src/App.js

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

class App extends Component {
  render() {
    return (
      <TodoListTemplate>
        템플릿 완성
      </TodoListTemplate>
    );
  }
}

export default App;
```



Todo-List : 두번째 컴포넌트, Form

컴포넌트 작성 순서

Form.js 작성

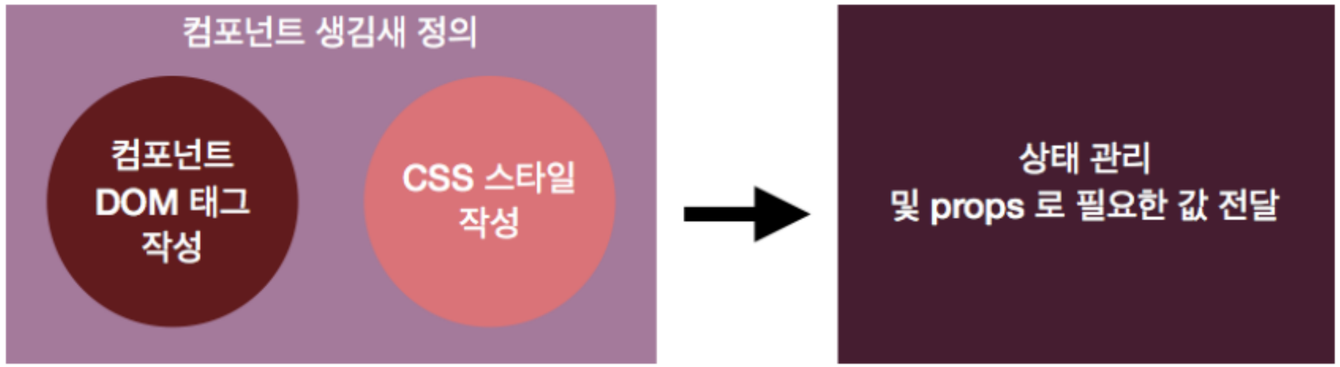
↓

Form.css 작성

↓

App.js에서 사용

Form 컴포넌트는 input과 button을 포함하고 있는 컴포넌트 입니다. React 컴포넌트를 구현하게 될 때는, 다음과 같은 흐름으로 개발하게 됩니다.



Todo-List : 두번째 컴포넌트, Form

1. Form 컴포넌트

src/components/Form.js

```
import React from 'react';
import './Form.css';

const Form = ({value, onChange, onCreate, onKeyPress}) => {
  return (
    <div className="form">
      <input value={value} onChange={onChange}
            onKeyPress={onKeyPress}/>
      <div className="create-button" onClick={onCreate}>
        추가
      </div>
    </div>
  );
};

export default Form;
```

이 컴포넌트는 총 4가지의 props 를 받아옵니다.

- value: input의 내용
- onChange: input 내용이 변경 될 때 실행 되는 함수
- onCreate: button이 클릭 될 때 실행 될 함수
- onKeyPress: input에서 키를 입력 할 때 실행 되는 함수

이 함수는 나중에 Enter 가 눌렸을 때 onCreate 를 한 것과 동일한 작업을 하기 위해서 사용합니다.

Todo-List : 두번째 컴포넌트, Form

2. Form Style

src/components/Form.css

```
.form {
  display: flex;
}
.form input {
  flex: 1; /* 버튼을 뺀 빈 공간을 모두 채워줍니다 */
  font-size: 1.25rem;
  outline: none;
  border: none;
  border-bottom: 1px solid #c5f6fa;
}
.create-button {
  padding-top: 0.5rem;
  padding-bottom: 0.5rem;
  padding-left: 1rem;
  padding-right: 1rem;
  margin-left: 1rem;
  background: #22b8cf;
  border-radius: 3px;
  color: white;
  font-weight: 600;
  cursor: pointer;
}
.create-button:hover {
  background: #3bc9db;
}
```

CSS에서 flex box라는 개념이 생겼습니다.

요소들을 자유 자제로 위치 시키는 flex 속성

Flexbox 모델을 사용 하려면 부모 엘리먼트를 flex container로 만들어야 한다.

display: flex 또는

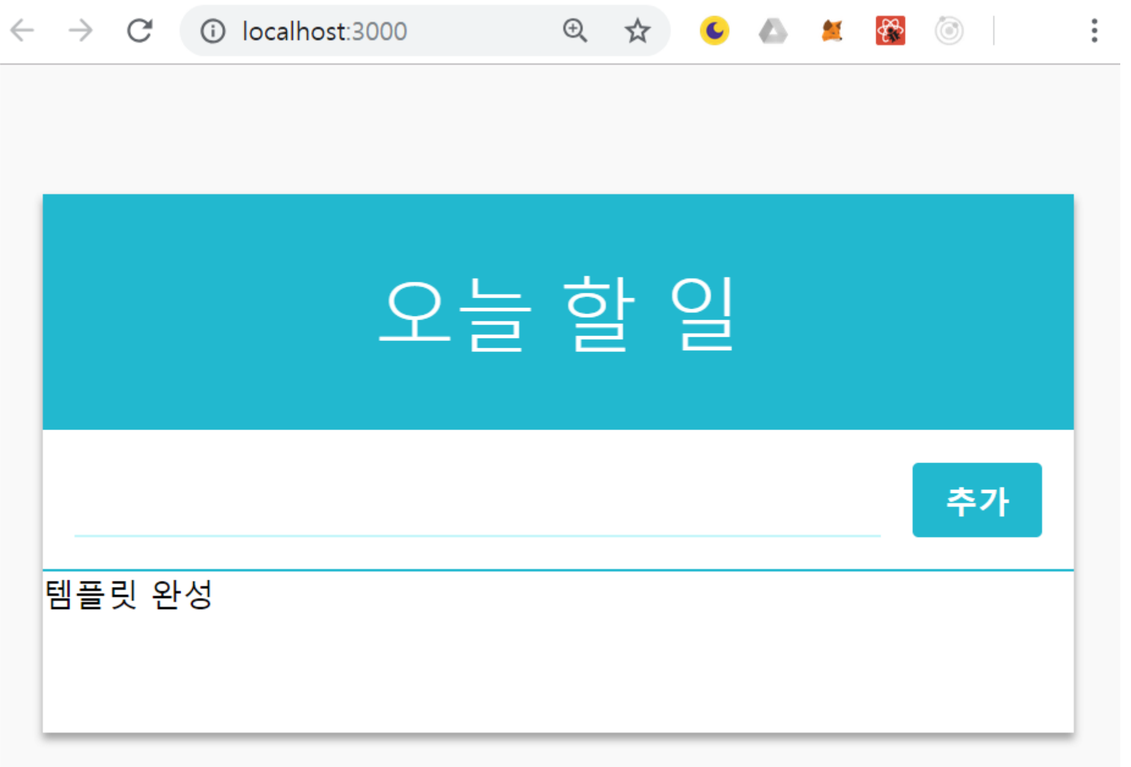
display: inline-flex를 설정하면 된다.

Todo-List : 두번째 컴포넌트, Form

3. App.js 에서 TodoListTemplate, Form 컴포넌트 사용하기

src/App.js

```
import React, { Component } from 'react';
import TodoListTemplate from './components/TodoListTemplate';
import Form from './components/Form';

class App extends Component {
  render() {
    return (
      <TodoListTemplate form={<Form />}>
        템플릿 완성
      </TodoListTemplate>
    );
  }
}

export default App;
```



Todo-List : 세번째 컴포넌트, TodoItemList

1. TodoItemList 컴포넌트

TodoItemList.js 작성

TodoItemList 컴포넌트는 TodoItem 컴포넌트 여러 개를 렌더링 해주는 역할을 합니다.
보여 주는 리스트가 동적인 경우에는 함수형이 아닌 클래스형 컴포넌트로 작성 해야 합니다.
클래스형 컴포넌트로 작성해야 나중에 컴포넌트 성능 최적화를 할 수 있기 때문입니다.
3개의 props 를 받는다.

- todos: todo 객체들이 들어 있는 배열
- onToggle: 체크박스를 키고 끄는 함수
- onRemove: 아이템을 삭제시키는 함수

src/components/TodoItemList.js

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

class TodoItemList extends Component {
  render() {
    const { todos, onToggle, onRemove } = this.props;

    return (
      <div>

      </div>
    );
  }
}

export default TodoItemList;
```

Todo-List : 네번째 컴포넌트, TodoItem

컴포넌트 작성 순서

TodoItem.js 작성

↓

TodoItem.css 작성

↓

TodoItemList.js 수정

↓

App.js 수정

네번째 컴포넌트 , TodoItem

이 컴포넌트는, 체크 값이 활성화 되어 있으면 우측에 체크마크 (✓ ✓)를 보여주고, 마우스가 위에 있을 때는 좌측에 엑스마크 (× ×)를 보여 줍니다. 이 컴포넌트의 영역이 클릭 되면 체크박스가 활성화 되며 중간 줄이 그어지고, 좌측의 엑스 마크가 클릭 되면 todoItem이 삭제 됩니다.

components 디렉토리에 다음 파일을 생성하세요:

: src/components/TodoItem.js

: src/components/TodoItem.css

Todo-List : 네번째 컴포넌트, TodoItem

1. TodoItem 컴포넌트

src/components/TodoItem.js

```
import React, { Component } from 'react';
import './TodoItem.css';
class TodoItem extends Component {
  render() {
    const { text, checked, id, onToggle, onRemove } = this.props;
    return (
      <div className="todo-item" onClick={() => onToggle(id)}>
        <div className="remove" onClick={(e) => {
          e.stopPropagation(); // onToggle 이 실행되지 않도록 함
          onRemove(id)}
        }>&times;</div>
        <div className={`todo-text ${checked && 'checked'}`}>
          <div>{text}</div>
        </div>
        {
          checked && (<div className="check-mark">✓</div>)
        }
      </div>
    );
  }
}
export default TodoItem;
```

이 컴포넌트는 총 5가지의 props 를 전달받게 됩니다.

- text: todo 내용
- checked: 체크박스 상태
- id: todo 의 고유 아이디
- onToggle: 체크박스를 키고 끄는 함수
- onRemove: 아이템을 삭제 시키는 함수

checked 값에 따라 className 에 checked 라는 css 클래스 이름을 동적으로 넣었습니다. CSS 클래스를 유동적으로 설정하기 위해 템플릿 리터럴을 사용 하였음.

```
`todo-text ${checked && 'checked'}`
// 아래와 동일합니다.
"todo-text " + checked && 'checked'
```

Todo-List : 네번째 컴포넌트, TodoItem

2. TodoItem Style #1

src/components/TodoItem.css

```
.todo-item {
  padding: 1rem;
  display: flex;
  align-items: center; /* 세로 가운데 정렬 */
  cursor: pointer;
  transition: all 0.15s;
  user-select: none;
}
.todo-item:hover {
  background: #e3fafc;
}
/* todo-item 에 마우스가 있을 때만 .remove 보이기 */
.todo-item:hover .remove {
  opacity: 1;
}
/* todo-item 사이에 윗 테두리 */
.todo-item + .todo-item {
  border-top: 1px solid #f1f3f5;
}
```

TodoItem Style #2

src/components/TodoItem.css

```
.remove {
  margin-right: 1rem;
  color: #e64980;
  font-weight: 600;
  opacity: 0;
}
.todo-text {
  flex: 1; /* 체크, 엑스를 제외한 공간 다 채우기 */
  word-break: break-all;
}
.checked {
  text-decoration: line-through;
  color: #adb5bd;
}
.check-mark {
  font-size: 1.5rem;
  line-height: 1rem;
  margin-left: 1rem;
  color: #3bc9db;
  font-weight: 800;
}
```

Todo-List : 네번째 컴포넌트, TodoItem

3. TodoItemList 컴포넌트 수정

src/components/TodoItemList.js

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

class TodoItemList extends Component {
  render() {
    const { todos, onToggle, onRemove } = this.props;

    return (
      <div>
        <TodoItem text="안녕"/>
        <TodoItem text="리액트"/>
        <TodoItem text="반가워"/>
      </div>
    );
  }
}

export default TodoItemList;
```

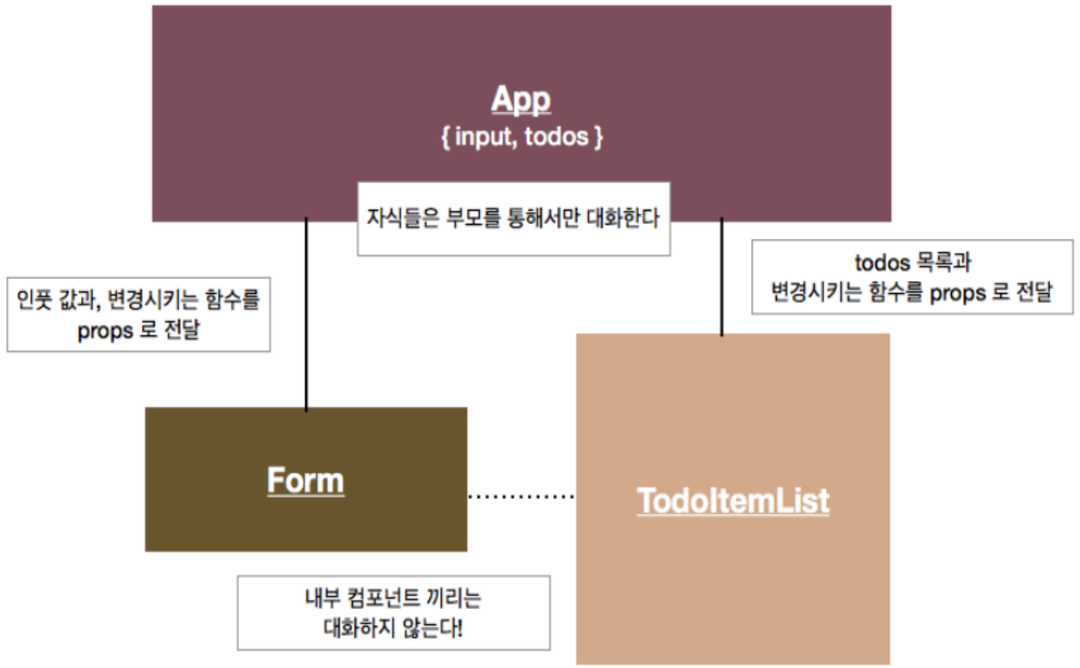
4. App 컴포넌트 수정

src/App.js

```
import React, { Component } from 'react';
import TodoListTemplate from './components/TodoListTemplate';
import Form from './components/Form';
import TodoItemList from './components/TodoItemList';


class App extends Component {
  render() {
    return (
      <TodoListTemplate form={<Form/>}>
        <TodoItemList/>
      </TodoListTemplate>
    );
  }
}

export default App;
```

Todo-List : 상태 관리 하기

상태가 필요한 컴포넌트는 Form과 TodoItemList 입니다. 자식 컴포넌트끼리 직접 데이터를 전달하는 것은 ref를 사용 할 수 있지만 정말 비효율적인 방법입니다. react에서는 안티패턴 입니다. 자식 컴포넌트들은 부모를 통하여 대화를 해야 합니다.



App 은 Form과 TodoItemList의 부모 컴포넌트이며, 해당 컴포넌트에 input, todos 상태를 넣어주고 해당 값들을 업데이트 하는 함수들을 각각 컴포넌트에 props로 전달해 주어서 기능을 구현하게 됩니다.

Todo-List : 초기 state 정의하기

1. App 컴포넌트에 state 정의하기

src/App.js

```
import React, { Component } from 'react';
import TodoListTemplate from './components/TodoListTemplate';
import Form from './components/Form';
import TodoItemList from './components/TodoItemList';

class App extends Component {

  id = 3 // 이미 0,1,2 가 존재하므로 3으로 설정
  state = {
    input: '',
    todos: [
      { id: 0, text: '리액트 소개', checked: false },
      { id: 1, text: '리액트 구조', checked: true },
      { id: 2, text: '리액트 사용', checked: false }
    ]
  }
  render() {
    return (
      <TodoListTemplate form={<Form/>}>
        <TodoItemList/>
      </TodoListTemplate>
    );
  }
}
export default App;
```

초기 state 에는 input의 값과, todos 배열에 기본 Item 3개를 넣어 주었음.

Todo 객체들을 구분 하기 위하여 id 값을 지정한다.

데이터가 추가 될 때마다 this.id 값이 1씩 올라가도록 설정

Todo-List : Form 기능 구현하기

1. App 컴포넌트에 Form 기능 구현하기 #1

src/App.js

```
class App extends Component {
  .....
  handleChange = (e) => {
    this.setState({
      input: e.target.value // input 의 다음 바뀔 값
    });
  }

  handleCreate = () => {
    const { input, todos } = this.state;
    this.setState({
      input: '',  // input 초기화
      // concat 을 사용하여 배열에 추가
      todos: todos.concat({
        id: this.id++,
        text: input,
        checked: false
      })
    });
  }
  handleKeyPress = (e) => {
    // 눌려진 키가 Enter 이면 handleCreate 호출
    if(e.key === 'Enter') {
      this.handleCreate();
    }
  }
```

우선 Form 컴포넌트에서 필요한 기능

- 텍스트 내용 바뀌면 input state 속성 업데이트
- 버튼이 클릭 되면 새로운 todo 생성 후 todos 배열 업데이트
- input에서 Enter 누르면 버튼 클릭한 것과 동일한 작업 진행하기

App 컴포넌트에서 handleChange, handleCreate, handleKeyPress 메소드를 구현하고, 이를 state의 input 값과 함께 Form 컴포넌트로 전달한다.

Todo-List : Form 기능 구현하기

2. App 컴포넌트에 Form 기능 구현하기 #2

src/App.js

```
class App extends Component {
  ....
  render() {
    const { input } = this.state;
    const {
      handleChange,
      handleCreate,
      handleKeyPress
    } = this;

    return (
      <TodoListTemplate form={(
        <Form
          value={input}
          onKeyPress={handleKeyPress}
          onChange={handleChange}
          onCreate={handleCreate}
        />
      )}>
        <TodoItemList/>
      </TodoListTemplate>
    );
  }
}
export default App;
```

render() 메소드에서 이벤트 메서드들을 전달할 때

```
const {
  handleChange,
  handleCreate,
  handleKeyPress
} = this;
```

비구조화 할당(destructuring assignment)을 했습니다.

이렇게 함으로서 this.handleChange, this.handleCreate, this.handleKeyPress

this를 붙여 줘야 하는 작업을 생략 할 수 있습니다.

Todo-List : 배열을 TodoItem 배열로 변환

1. todos 배열을 전달하기 : App -> TodoItemList

src/App.js

```
class App extends Component {
  ....
  render() {
    const { input, todos } = this.state;
    const {
      handleChange,
      handleCreate,
      handleKeyPress
    } = this;

    return (
      <TodoListTemplate form={(
        <Form
          value={input}
          onKeyPress={handleKeyPress}
          onChange={handleChange}
          onCreate={handleCreate}
        />
      )}>
        <TodoItemList todos={todos} />
      </TodoListTemplate>
    );
  }
}
export default App;
```

src/components/TodoItemList.js

```
import React, { Component } from 'react';
import TodoItem from './TodoItem';
class TodoItemList extends Component {
  render() {
    const { todos, onToggle, onRemove } = this.props;
    const todoList = todos.map(
      ({id, text, checked}) => (
        <TodoItem
          id={id}
          text={text}
          checked={checked}
          onToggle={onToggle}
          onRemove={onRemove}
          key={id}
        />
      )
    );
    return (
      <div>
        {todoList}
      </div>
    );
  }
}
export default TodoItemList;
```

Todo-List : Checkbox 이벤트 처리

1. Checkbox 이벤트 처리하기 #1

src/App.js

```
import React, { Component } from 'react';
import TodoListTemplate from './components/TodoListTemplate';
import Form from './components/Form';
import TodoItemList from './components/TodoItemList';

class App extends Component {
  ......
  handleToggle = (id) => {
    const { todos } = this.state;

    // 파라미터로 받은 id 를 가지고 몇번째 Item인지 찾습니다.
    const index = todos.findIndex(todo => todo.id === id);
    const selected = todos[index]; // 선택한 객체

    const copyTodos = [...todos]; // 배열을 복사

    // 기존의 값들을 복사하고, checked 값을 덮어쓰기
    copyTodos[index] = {
      ...selected,
      checked: !selected.checked
    };
    this.setState({
      todos: copyTodos
    });
  }
```

배열을 업데이트 할 때는 배열의 값을 직접 수정하면 않됩니다.

전개 연산자(spread operator)를 통하여 업데이트 해야 할 배열이나 객체의 내용을 복사해 주어야 합니다.

Todo-List : Checkbox 이벤트 처리

2. Checkbox 이벤트 처리하기 #2

src/App.js

```
class App extends Component {
  ......
  render() {
    const { input, todos } = this.state;
    const {
      handleChange,
      handleCreate,
      handleKeyPress,
      handleToggle
    } = this;

    return (
      <TodoListTemplate form={(
        <Form
          value={input}
          onKeyPress={handleKeyPress}
          onChange={handleChange}
          onCreate={handleCreate}
        />
      )}>
        <TodoItemList todos={todos} onToggle={handleToggle} />
      </TodoListTemplate>
    );
  }
```

Checkbox를 업데이트 하는 handleToggle 함수들을 onToggle이라는 key로 TodoItemList 컴포넌트에 props로 전달한다.

Todo-List : Item 제거하기

1. Item 삭제 이벤트 처리하기

src/App.js

```
class App extends Component {
  ......
  handleRemove = (id) => {
    const { todos } = this.state;
    this.setState({
      todos: todos.filter(todo => todo.id !== id)
    });
  }
  render() {
    const { input, todos } = this.state;
    const {
      ...
       handleRemove
    } = this;
    return (
      .....
        <TodoItemList todos={todos} onToggle={handleToggle}
                onRemove={handleRemove} />
      </TodoListTemplate>
    );
  }
  }
```

handleRemove에서는 배열의 내장함수인 filter를 사용하여 파라미터로 받아온 id를 배제한 배열을 새로 생성한것이다.

새로 생성한 배열을 todos로 설정하면, 원하는 데이터가 없어진다.

Todo-List : 컴포넌트 최적화 하기

1. TodoItemList 최적화

src/App.js

```
const initialTodos = new Array(500).fill(0).map(
  (item, idx) => ({ id: idx, text: `일정 ${idx}`, checked: true })
);
class App extends Component {
      state = {
          todos: initialTodos
      }
}
export default App;
```

src/components/TodoItemList.js

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

class TodoItemList extends Component {

  shouldComponentUpdate(nextProps, nextState) {
    return this.props.todos !== nextProps.todos;
  }

  render() {
  }
}
export default TodoItemList;
```

1. 현재는 값을 입력할 때 마다 render 함수가 실행되어 자원이 미세하게 낭비가 되고 있음.
2. 리스트의 갯수가 많아질 수 있다면 최적화를 해줘야 버퍼링이 걸리지 않습니다.
3. 컴포넌트 라이프 사이클 메소드 중 shouldComponentUpdate는 컴포넌트가 Re렌더링을 여부를 정해줍니다.
4. 이 메서드는 true 를 반환하지만, 구현하는 경우에는 업데이트에 영향을 주는 조건을 return 하면 됩니다.
5. Todos 값이 바뀔 때 Re렌더링 하면 되므로 this.props.todos 와 nextProps.todos를 비교해서 값이 다를 때만 Re렌더링 하게 설정하면 됩니다.

Todo-List : 컴포넌트 최적화 하기

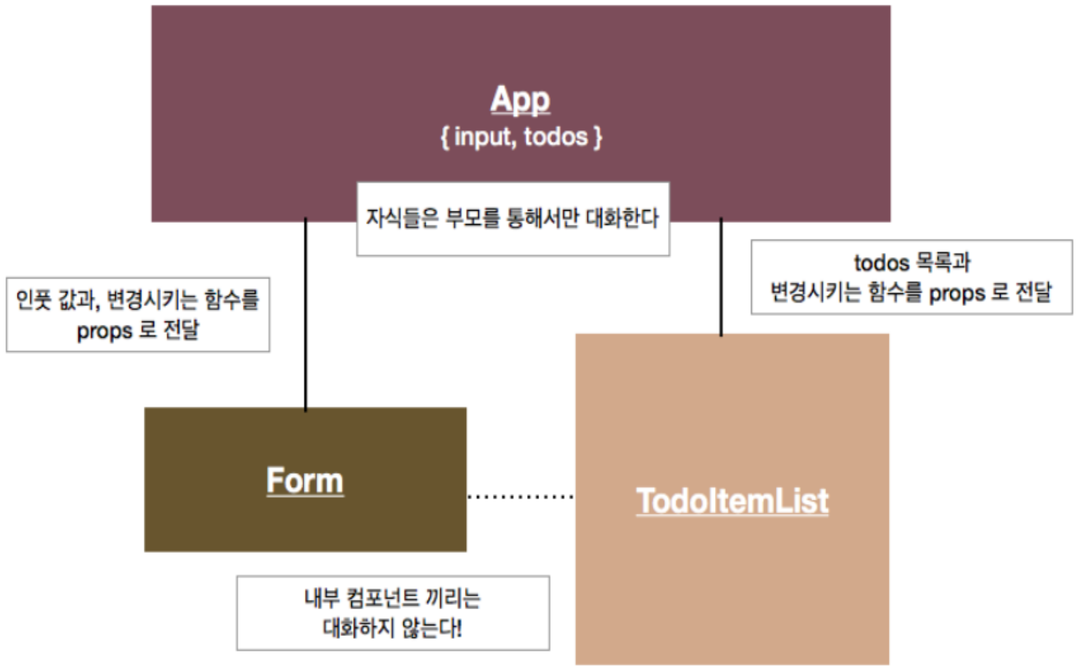
2. TodoItem 최적화

src/components/TodoItem.js

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

class TodoItem extends Component {

  shouldComponentUpdate(nextProps, nextState) {
    return this.props.checked !== nextProps.checked;
  }

  render() {
    (...)
  }
}

export default TodoItem;
```

1. Item의 checkbox를 클릭하거나, 새로운 todo를 입력,삭제를 하면 하나의 TodoItem 컴포넌트 만 업데이트 되지 않고 모든 컴포넌트가 렌더링 되고 있다.
2. TodoItem 컴포넌트가 업데이트 되는 경우는 checked 값이 바뀔 때 shouldComponentUpdate 를 구현하면 됩니다.

Todo-List : 요약정리

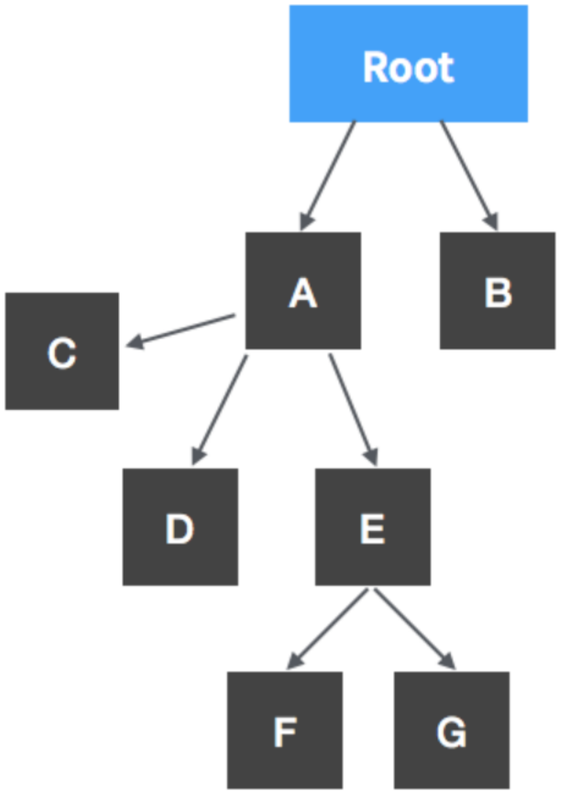


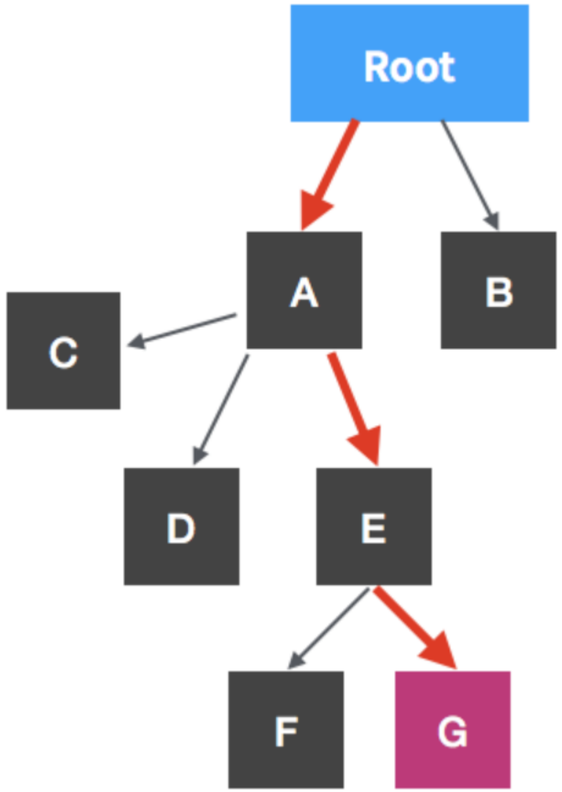
1. TodoList 프로젝트에서는 부모(App) 컴포넌트가 중간자 역할을 합니다.
2. App에서는 input값과, 이를 변경하는 onChange 함수와 Item을 생성하는 onCreate 함수를 props로 Form에게 전달해줍니다.
3. Form은 해당 함수와 값을 받아서 화면에 보여주고, 변경 이벤트가 일어나면 App에서 받은 onChange를 호출하여 input 값을 업데이트 합니다.
4. input 값을 입력하여 추가 버튼을 누르면, onCreate를 호출하여 todos 배열을 업데이트 합니다.
   todos 배열이 업데이트 되면, 해당 배열이 TodoItemList 컴포넌트 한테 전달이 되어 화면에 렌더링 됩니다.
5. 부모 컴포넌트에서 모든걸 관리하고 아래로 내려주기 때문에, 매우 직관적이기도 하고, 관리하기 편하지만 앱의 규모가 커지면 App 의 코드가 엄청 나게 길어지고, 유지보수 하는 것도 힘들어 집니다.

리덕스

Todo-List : 기존 App 구조의 문제점

Root 컴포넌트에서 G 컴포넌트에게 어떠한 값을 전달해 줘야 하는 상황에는 어떻게 해야 할까요?





A 를 거치고 E 를 거치고 G 를 거쳐야 합니다.

// App.js 에서 A 렌더링

<A value={5}>

// A.js 에서 E 렌더링

<E value={this.props.value} />

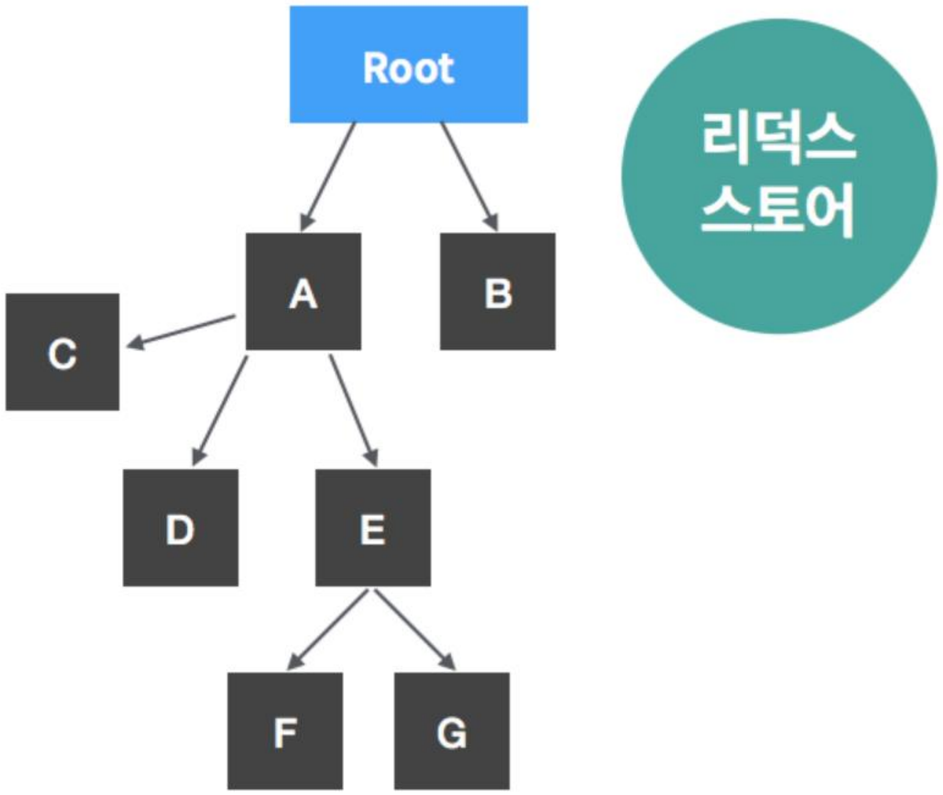
// B.js 에서 G 렌더링

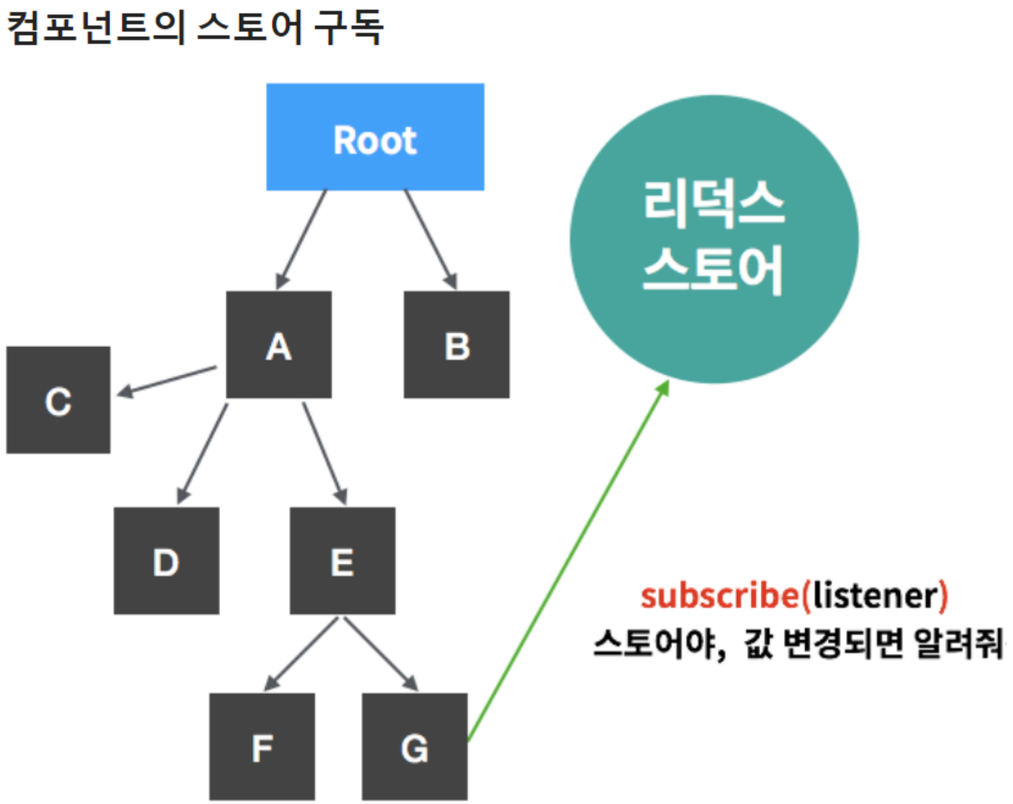
<G value={this.props.value} />

Todo-List : Redux 사용

* 컴포넌트의 Store 구독

Redux를 프로젝트에 적용하게 되면 Store가 생성 됩니다. Store 안에는 프로젝트의 state에 관한 데이터들이 담겨 있습니다. G 컴포넌트는 스토어에 구독을 합니다. 구독을 하는 과정에서, 특정 함수가 Store 한테 전달이 됩니다. 나중에 Store의 상태값에 변동이 생긴다면 전달 받았던 함수를 호출해 줍니다.

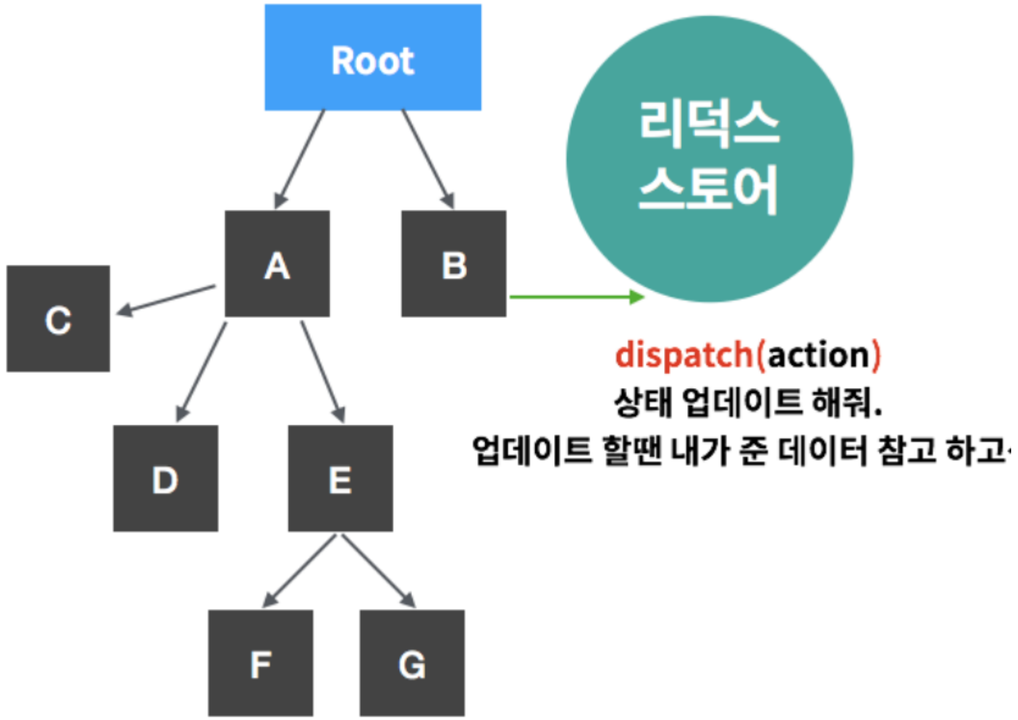




Todo-List : Redux 사용

* Store에 상태 변경 하라고 알려주기

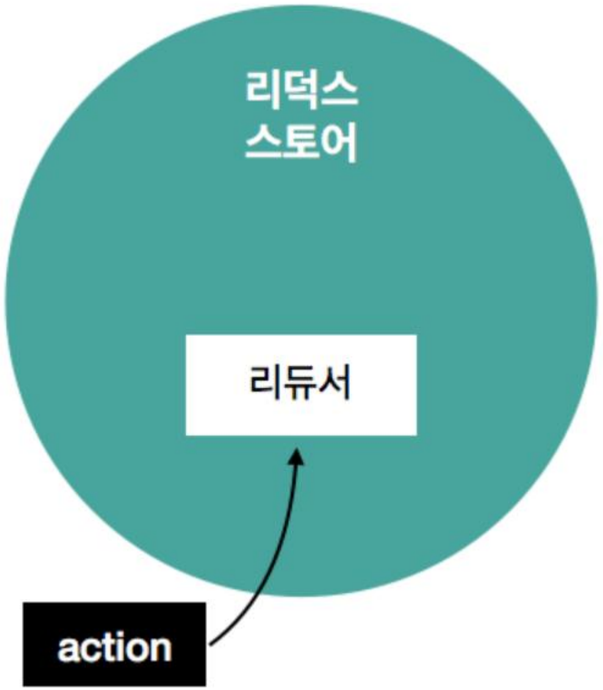
B 컴포넌트에서 어떤 이벤트가 생겨서, 상태가 변경될 때 dispatch라는 함수를 통하여 Action을 Store한테 던져 줍니다. Action은 상태에 변화를 일으킬 때 참조 할 수 있는 객체입니다. 객체는 필수적으로 type 이라는 값을 가지고 있어야 합니다. 예를 들어 { type: ' INCREMENT ' } 이런 객체를 전달 받게 된다면, redux 스토어는 ' state에 값을 더해야 하는구나 ' 하고 Action을 참조하게 됩니다.



Todo-List : Redux 사용

* Reducer를 통하여 상태를 변화 시키기

Action 객체를 받으면 전달 받은 Action의 타입에 따라 어떻게 상태를 업데이트 해야 할지 정의를 해야 합니다. 이러한 업데이트 로직을 정의하는 함수를 Reducer라고 부릅니다. Reducer 함수는 직접 구현하게 됩니다. 예를 들어 type 이 INCREMENT 라는 액션이 들어오면 숫자를 더해주고, DECREMENT 라는 Action이 들어오면 숫자를 감소 시키는 그런 작업을 Reducer 함수에서 하면 됩니다.



Reducer 함수는 두가지의 파라미터를 받습니다.

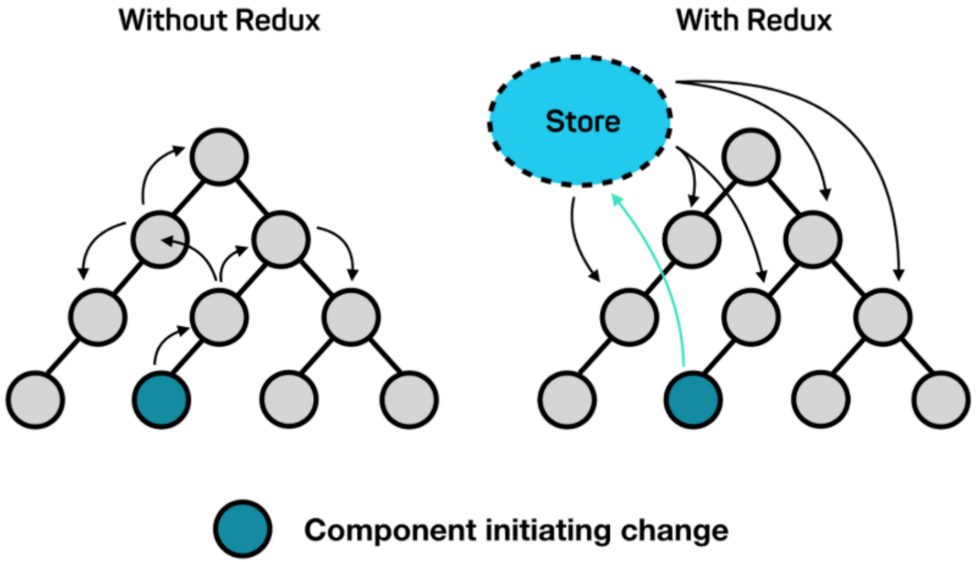
- state: 현재 상태,
- action: 액션 객체

이 파라미터들을 참조하여, 새로운 상태 객체를 만들어서 이를 반환 합니다.

Redux

- Redux란?

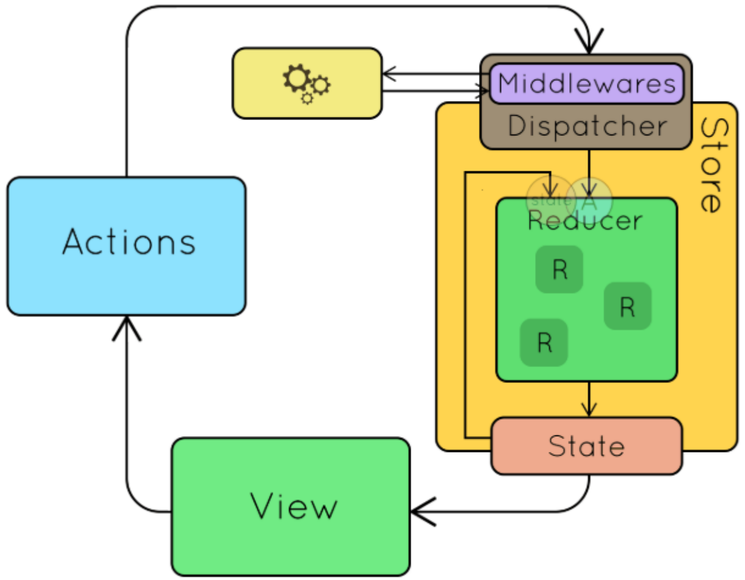
1. Redux는 리액트에서 상태를 더 효율적으로 관리 하는데 사용하는 상태 관리 라이브러리이다.
2. Redux는 상태 관리의 로직을 컴포넌트 밖에서 처리하는 것이다.
3. Redux를 사용하면 스토어(Store)라는 객체에 상태를 저장한다.
4. 모든 상태 관리가 스토어에서 일어난다. 상태가 변경 되면 액션(Action)이라는 것을 스토어에 전달된다.



Redux

- Redux 구조

1. View에서 상태가 변경되면 Action을 Store에 전달합니다. 액션을 전달하는 과정을 디스패치 라고 합니다.
2. Store가 Action을 받으면 Reducer가 전달 받은 Action을 기반으로 상태를 어떻게 변경시켜야 할지 정한다.
3. Action을 처리하면 새로운 상태를 Store에 저장합니다.
4. Store 안에 있는 상태가 바뀌면 Store를 Subscribe하고 있는 컴포넌트에 바로 전달합니다.



https://github.com/LambdaSchool/Redux-Counter

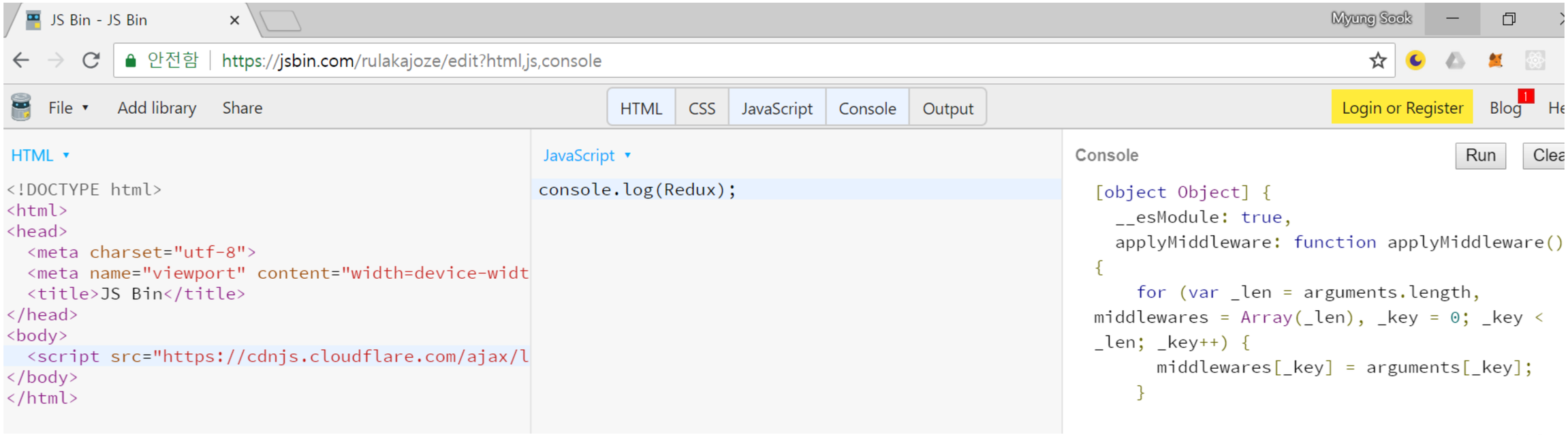
- Store : 애플리케이션의 상태 값들을 내장하고 있다.
- Action : 상태 변화를 일으킬 때 참조하는 객체입니다.
- Dispatch : 액션을 스토어에 전달하는 것을 의미합니다.
- Reducer : 상태를 변화시키는 로직이 있는 함수입니다.
- Subscribe : 스토어 값이 필요한 컴포넌트는 스토어를 구독합니다.

Redux

- Redux 사용

1. 리덕스는 리액트에서 사용하려고 만든 상태 관리 라이브러리 이지만, 리액트에 의존하지는 않습니다.
2. 리액트를 사용하지 않아도 리덕스를 사용할 수 있습니다.
3. JSBin(https://jsbin.com)에서 리덕스를 사용해 보기

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/redux/3.6.0/redux.js"></script>
```

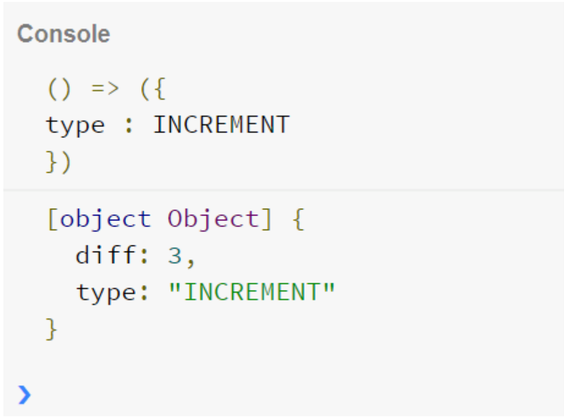


• Action과 Action생성 함수

1. 액션은 type 값을 반드시 가지고 있어야 합니다. 액션 타입은 해당 액션이 어떤 작업을 하는 액션인지 정의하며, 대문자와 밑줄을 조합하여 만듭니다.

액션과 액션생성함수 정의하기

```
const INCREMENT = 'INCREMENT';
const DECREMENT = 'DECREMENT';

//값을 증가시키는 액션
const increment = () => ({
  type : INCREMENT
});
//가변적인 값이 들어가야 하면 파라미터를 넣어서 액션을 만든다.
const increment2 = (diff) => ({
  type : INCREMENT,
  diff : diff
});
//값을 감소시키는 액션
const decrement = (diff) => ({
  type : DECREMENT,
  diff : diff
});

console.log(increment);
console.log(increment2(3));
```



Redux

- 변화를 일으키는 함수 : Reducer – 상태에 값이 하나만 있는 경우

1. 상태에 변화를 일으키는 함수인 리듀서는 파라미터를 두개 받아야 하는데, 첫번째 파라미터는 현재 상태이고, 두번째 파라미터는 액션 객체입니다.
2. 함수 내부에서는 switch 문을 사용하여 action.type에 따라 새로운 상태를 만들어서 반환해야 합니다.

리듀서 정의하기 (상태에 값이 하나만 있는 경우)

```
//값을 증가시키는 액션
const increment = (diff) => ({
  type : INCREMENT, diff : diff
});
//값을 감소시키는 액션
const decrement = (diff) => ({  ..});
const initialState = { number : 0 }
function counter(state = initialState, action) {
  switch(action.type) {
    case INCREMENT:
      return {
        number:state.number + action.diff
      };
    case DECREMENT:
      return { number:state.number - action.diff};
    default:
      return state;
  }
}
console.log(counter(undefined,increment(2)));
```

Console

```
[object Object] {
  number: 2
}
```

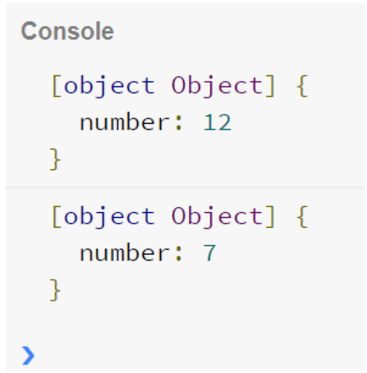
Redux

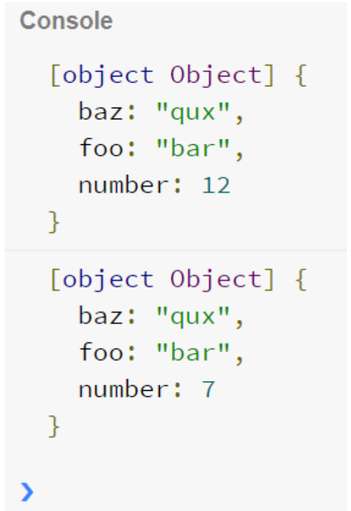
- 변화를 일으키는 함수 : Reducer – 상태에 값이 여러 개 있는 경우

1. State에 속성 값이 여러 개 있는 경우에는 **전개 연산자(Spread Operator)** 를 사용하여 기존의 속성값도 유지가 되도록 해주어야 합니다.

리듀서 정의하기 (상태에 값이 여러 개 있는 경우)

```
const initialState = {
      number : 10,
      foo:'bar',
      baz:'qux'
}
function counter(state = initialState, action) {
  switch(action.type) {
    case INCREMENT:
      return {
        ...state,
        number:state.number + action.diff
      };
    case DECREMENT:
      return {
        ...state,
        number:state.number - action.diff};
    default:
      return state;
  }
}
console.log(counter(undefined,increment(2)));
console.log(counter(undefined,decrement(3)));
```





Redux

- Redux Store 생성

1. 액션과 리듀서가 준비되면 리덕스 스토어를 만들 수 있습니다.
2. 스토어를 생성할 때는 createStore 함수를 사용합니다.
   파라미터로는 리듀서 함수가 들어가고, 두 번째 파라미터를 설정하면 해당 값을 스토어의 기본 값으로 사용합니다.

스토어 생성

```
/*
  실제로 프로젝트에서 불러올 때는
  import { createStore } from 'redux';
*/
const { createStore } = Redux;

const store = createStore(counter);
```

Redux

- 구독

1. 리액트 컴포넌트에서 리덕스 스토어를 구독(subscribe)하는 작업은 나중에 react-redux의 connect 함수가 대신합니다.
2. 리덕스의 내장 함수 subscribe를 직접적으로 사용하는 경우는 없지만, 리덕스 개념을 이해하기 위해 직접 실행해 보는 것입니다.
3. 리덕스 스토어를 구독한다는 것은 리덕스 스토어의 상태가 바뀔 때마다 특정 함수를 실행시킨다는 의미입니다.
4. getState() 함수는 현재 스토어 상태를 반환합니다.

구독 - subscribe() 호출

```
const unsubscribe = store.subscribe(() => {
  console.log(store.getState());
});
```

Redux

- dispatch로 액션 전달

1. 스토어에 액션을 넣을 때는 store.dispatch 함수를 사용합니다.
2. 액션들이 dispatch 될 때마다 등록했던 함수가 실행되어 진다.

dispatch() 호출

```
const { createStore } = Redux;

const store = createStore(counter);

const unsubscribe = store.subscribe(() => {
  console.log('스토어 상태값 출력');
  console.log(store.getState());
});

store.dispatch(increment(1));
store.dispatch(increment(5));
store.dispatch(increment(10));
```

Console

```
"스토어 상태값 출력"
[object Object] {
  baz: "qux",
  foo: "bar",
  number: 11
}
"스토어 상태값 출력"
[object Object] {
  baz: "qux",
  foo: "bar",
  number: 16
}
"스토어 상태값 출력"
[object Object] {
  baz: "qux",
  foo: "bar",
  number: 26
}
```

Redux 규칙

- 리덕스 사용시 알아야 할 3가지 규칙

1. 하나의 애플리케이션 안에는 하나의 Store가 있습니다.

   : 스토어는 언제나 단 한 개입니다. 스토어를 여러 개 생성해서 상태를 관리하면 안 됩니다.

   : 리듀서는 여러 개 만들어서 관리할 수 있습니다.

2. 상태는 읽기전용 입니다.

   : 리덕스의 상태, state 값은 읽기 전용입니다. State 값을 직접 수정 하면 리덕스의 구독 함수를 제대로 실행하지 않거나, 컴포넌트의 Re렌더링 되지 않을 수도 있습니다.

   : Immutable.js 를 사용하면 불변성을 유지하며 상태를 쉽게 관리하게 됩니다.

3. 변화를 일으키는 함수, Reducer는 순수 함수(pure function)로 구성 해야 합니다.

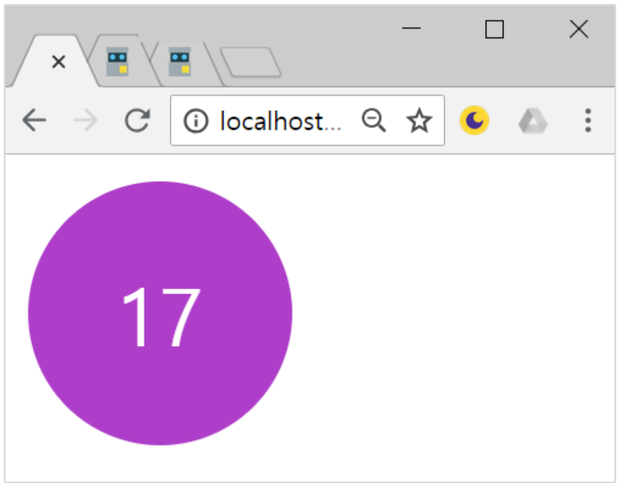
   : 순수 함수란?

   리듀서 함수는 이전 상태와, 액션 객체를 파라미터로 받습니다.

   이전의 상태는 절대로 변경하지 않고, 변화를 일으킨 새로운 상태 객체를 만들어서 반환합니다.

   똑같은 파라미터로 호출된 리듀서 함수는 언제나 똑같은 결과값을 반환 해야만 합니다.

Counter App #1 (redux 사용)



Counter

1. 새로운 프로젝트 생성 : redux-counter

```
create-react-app redux-counter
```

2. redux와 react-redux 설치

```
cd redux-counter

npm i --save redux react-redux
```

3. dev server start

```
npm start
```

Counter

4. src 디렉토리에 필요 없는 파일들 삭제
   - App.css
   - App.js
   - App.test.js
   - logo.svg

5. src 디렉토리에 sub directory 생성
   - actions : 액션 타입과 액션 생성자 파일을 저장합니다.
   - components : 컴포넌트의 뷰에 해당하는 presentational 컴포넌트를 저장합니다.
   - containers : store에 있는 상태를 props로 받아 오는 container 컴포넌트들을 저장합니다.
   - reducers : 스토어의 기본 상태 값과 상태의 업데이트를 담당하는 reducer 파일들을 저장합니다.
   - utils : 일부 컴포넌트에서 함께 사용하는 파일을 저장합니다.

Counter

- Presentational 컴포넌트와 Container 컴포넌트

1. Presentational 컴포넌트

: View 기능만 담당한다. (DOM 엘리먼트와 스타일이 있다)

: 리덕스 스토어에 직접 접근하지 않고, 오직 props로만 데이터를 가져올 수 있다.

: 주로 함수형 컴포넌트로 작성되며 state 를 가지고 있어야 하거나, 최적화를 위해 LifeCycle 이 필요하면 클래스형 컴포넌트로 작성됩니다.

2. Container 컴포넌트

: 상태를 가지고 있으며, 리덕스에 직접 접근할 수 있다.

: DOM 엘리먼트나 스타일을 직접적으로 사용할 수 없다.

3. 이 구조의 장점

: UI 쪽과 Data 쪽이 분리 되어 프로젝트를 이해 하기가 쉬워지며, 컴포넌트의 재 사용률도 높여 줍니다.

4. 리덕스의 창시자 댄 아브라모프(Dan Abramov)가 더 나은 설계를 하기 위해 공유한 구조이다.

https://twitter.com/dan_abramov/status/802569801906475008

Counter

- 1.기본적인 틀 생성

src/containers/App.js

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

class App extends Component {
    render() {
        return (
            <div>
                Counter
            </div>
        );
    }
}
export default App;
```

src/index.js

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

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

Counter

- 2.Counter presentational 컴포넌트 생성

src/components/Counter.js

```
import React from 'react';
import PropTypes from 'prop-types';
import './Counter.css';

const Counter = ({number, color, onIncrement,
onDecrement, onSetColor}) => {
    return (
        <div
            className="Counter"
            onClick={onIncrement}
            onContextMenu={(e) => {
                e.preventDefault();
                onDecrement();
            }}
            onDoubleClick={onSetColor}
            style={{backgroundColor: color}}>
            {number}
        </div>
    );
};
```

src/components/Counter.js

```
Counter.propTypes = {
    number: PropTypes.number,
    color: PropTypes.string,
    onIncrement: PropTypes.func,
    onDecrement: PropTypes.func,
    onSetColor: PropTypes.func
};

Counter.defaultProps = {
    number: 0,
    color: 'black',
    onIncrement: () => console.warn('onIncrement not defined'),
    onDecrement: () => console.warn('onDecrement not defined'),
    onSetColor: () => console.warn('onSetColor not defined')
};

export default Counter;
```

Counter

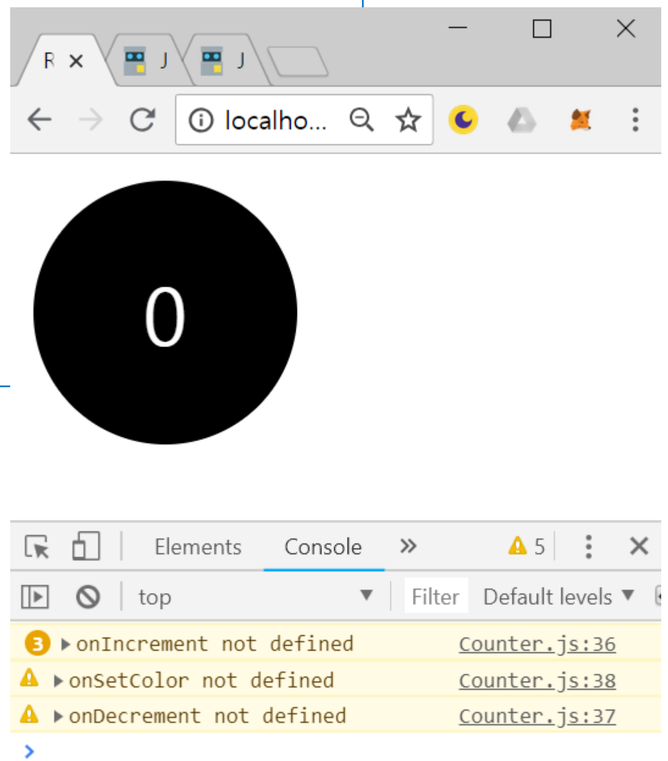
- 2-1.Counter Style 과 App 컴포넌트

src/components/Counter.css

```
.Counter {
  /* 레이아웃 */
  width: 10rem;
  height: 10rem;
  display: flex;
  align-items: center;
  justify-content: center;
  margin: 1rem;
  /* 색상 */
  color: white;
  /* 폰트 */
  font-size: 3rem;
  /* 기타 */
  border-radius: 100%;
  cursor: pointer;
  user-select: none;
  transition: background-color 0.75s;
}
```

src/containers/App.js

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

class App extends Component {
    render() {
        return (
            <div>
                <Counter />
            </div>
        );
    }
}
export default App;
```



컴포넌트를 동그라미 모양으로, 숫자는 가운데에 위치 시키고 흰색으로 설정함.

Counter

- 3.액션 생성

(ActionTypes와 Action 생성함수)

1. actions 디렉토리에 ActionTypes.js를 작성하여 다음과 같이 상수들을 선언하세요.
2. 액션 만드는 함수를 작성합니다. 이를 액션 생성자라고 부릅니다. actions 디렉토리에 index.js 파일을 만들어서 액션 생성 함수를 작성합니다.

src/actions/ActionTypes.js

```
/*
 Action의 종류들을 선언합니다.
 앞에 export를 붙이면 나중에 이것들을 불러올 때,
 import * as types from './ActionTypes'를 할 수 있음
*/

export const INCREMENT = 'INCREMENT';
export const DECREMENT = 'DECREMENT';
export const SET_COLOR = 'SET_COLOR';
```

src/actions/index.js

```
/*
 action 객체를 만드는 액션 생성 함수들을 선언합니다
 (action creators).
 여기서 () => ({})은 function() { return { } }와
 동일한 의미입니다.
*/
import * as types from './ActionTypes';

export const increment = () => ({
    type: types.INCREMENT
});

export const decrement = () => ({
    type: types.DECREMENT
});

export const setColor = (color) => ({
    type: types.SET_COLOR,
    color
});
```

Counter

- 4.리듀서 생성

: 리듀서는 액션의 type에 따라 변화를 일으키는 함수이다.

1. 리듀서 파일에는 객체의 초기 상태가 정의 되어야 합니다.

src/reducers/index.js

```
import * as types from '../actions/ActionTypes';

// 초기 상태를 정의합니다.
const initialState = {
            color: 'black',
            number: 0
};
```

1) 리듀서 함수는 state와 action을 파라미터로 받으며, 내부에서 switch 문을 통하여 action.type 에 따른 상태에 다른 변화를 일으키면 됩니다.

2) 주의 할 점은 state를 직접 수정하면 않 되고, 기존 상태 값에 원하는 값을 덮어쓴 새로운 객체를 만들어서 반환해야 합니다.

src/reducers/index.js

```
function counter(state = initialState, action) {
    switch (action.type) {
        case types.INCREMENT:
            return {
                ...state,
                number:state.number + 1
            };
        case types.DECREMENT:
            return {
                ...state,
                number:state.number - 1
            };
        case types.SET_COLOR:
            return {
                ...state,
                color:action.color
            };
        default:
            return state;
    }
};
export default counter;
```

Counter

- 5.스토어 생성

: Store는 리덕스에서 가장 핵심적인 인스턴스입니다. 이 안에 현재 상태를 내장하고 있고, 구독(subscribe)중인 함수들이 상태가 업데이트 될 때 마다 다시 실행되게 해줍니다.

: 리덕스에서 createStore를 불러와 앞에서 만든 리듀서를 전달하여 store 생성한다.

src/index.js

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

//리덕스 관련 불러오기
import {createStore} from 'redux';
import reducers from './reducers';

const store = createStore(reducers);

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

Counter

- 6.Provider 컴포넌트로 리액트 앱에 스토어 연동

: Provider는 react-redux 라이브러리에 내장된 컴포넌트이며, 리액트 애플리케이션에서 손쉽게 스토어를 연동할 수 있도록 도와주는 컴포넌트이다.

: Provider 컴포넌트를 불러온 후에 연동할 프로젝트의 최상위 App 컴포넌트를 감싸고,props로 store 값을 설정하면 됩니다.

src/index.js

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

//리덕스 관련 불러오기
import {createStore} from 'redux';
import reducers from './reducers';
import {Provider} from 'react-redux';

const store = createStore(reducers);

ReactDOM.render(
    <Provider store={store}>
        <App />
    </Provider>,
    document.getElementById('root'));
```

Counter

- 7.CounterContainer 컴포넌트 생성-1

: 컨테이너 컴포넌트를 store 에 연결 하려면 react-redux의 connect 함수를 사용해야 한다.

: connect([mapStateProps],[mapDispatchToProps],[mergeProps])

1) mapStateToProps : store.getState() 결과 값인 state를 파라미터로 받아 컴포넌트의 props로 사용할 객체를 반환합니다.

2) mapDispatchToProps : dispatch를 파라미터로 받아 액션을 디스패치하는 함수들을 객체 안에 넣어서 반환합니다.

: 상태를 연결시키는 함수는 mapStateToProps로, 액션함수를 연결시키는 함수는 mapDispatchToProps 로 만들어서, 이를 connect에 전달해주고, 그렇게 전달받은 함수에 앞에서 만든 Presentational Counter 컴포넌트를 파라미터로 전달해 주면 리덕스 스토어에 연결된 새로운 컴포넌트가 만들어진다.

src/containers/CounterContainer.js

```
import Counter from '../components/Counter';
import * as actions from '../actions';
import { connect } from 'react-redux';

 // store 안의 state 값을 props로 연결해줍니다.
    const mapStateToProps = (state) => ({
        color: state.color,
        number: state.number
    });
```

Counter

- 7.CounterContainer 컴포넌트 생성-2

src/containers/CounterContainer.js

```
/* 액션 생성 함수를 사용하여 액션을 생성하고,해당 액션을 dispatch하는 함수를 만든 후, 이를 props로 연결해줍니다.*/
 const mapDispatchToProps = (dispatch) => ({
     onIncrement: () => dispatch(actions.increment()),
     onDecrement: () => dispatch(actions.decrement()),
     onSetColor: () => {
         const color = 'black'  // 임시 getRandomColor()를 작성하여 변경할 예정
         dispatch(actions.setColor(color));
     }
 });
 /*Counter 컴포넌트를 애플리케이션의 데이터 레이어와 묶는 역할을 합니다.*/
 const CounterContainer = connect(
     mapStateToProps,
     mapDispatchToProps
 )(Counter);

 export default CounterContainer;
```

Counter

- 8.랜덤 색상을 만드는 함수 만들기

  : 랜덤 색상을 만드는 함수를 utils 디렉토리에 index.js 안에 정의 합니다.
  mapDispatchToProps에서 임시로 const color = 'black'을 getRandomColor()로 변경하세요.

src/utils/index.js

```
export function getRandomColor() {
    const colors = [
        '#495057',
        '#f03e3e',
        '#d6336c',
        '#ae3ec9',
        '#7048e8',
        '#4263eb',
        '#1c7cd6',
        '#1098ad',
        '#0ca678',
        '#37b24d',
        '#74b816',
        '#f59f00',
        '#f76707'
    ];
    // 0 부터 12까지 랜덤 숫자
    const random = Math.floor(Math.random() * 13);
    // 랜덤 색상 반환
    return colors[random];
}
```

src/containers/CounterContainer.js

```
import { getRandomColor } from '../utils';

const mapDispatchToProps = (dispatch) => ({
    onIncrement: () =>
dispatch(actions.increment()),
    onDecrement: () =>
dispatch(actions.decrement()),
    onSetColor: () => {
        const color = getRandomColor();
        dispatch(actions.setColor(color));
    }
});
```

Counter

- 9.Counter 대신 CounterContainer 컴포넌트 사용하기

  : App컴포넌트에서 Counter 대신 CounterContainer 컴포넌트를 호출하여 렌더링하기.

src/containers/App.js

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

class App extends Component {
    render() {
        return (
            <div>
                <CounterContainer />
            </div>
        );
    }
}
export default App;
```

Counter

- Redux Dev Tools 연결하기

  : 리덕스에서 액션이 dispatch 될 때마다 로그를 보고, 또 그 전 상태로 돌아갈 수도 있게 해주는 리덕스 개발자도구 Redux DevTool이 있습니다.

  개발자도구를 열었을 때, 개발자도구의 상단 탭의 우측에 Redux가 나타나게 됩니다.

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './containers/App';
import './index.css';
// Redux 관련 불러오기
import { createStore } from 'redux'
import reducers from './reducers';
import { Provider } from 'react-redux';

// 스토어 생성
const store = createStore(reducers, window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__());
ReactDOM.render(
    <Provider store={store}>
        <App />
    </Provider>,
    document.getElementById('root')
);
```

Counter

- 10.서브 리듀서 작성하기 #1

  : 기존 리듀서를 색상 리듀서, 숫자 리듀서로 분리를 시킨 다음에, combineReducers를 통해 이를 합쳐서 루트 리듀서로 만드는 방법을 알아보겠습니다.

src/reducers/color.js

```
import * as types from '../actions/ActionTypes';

const initialState = {
    color: 'black'
};

const color = (state = initialState, action) => {
    switch(action.type) {
        case types.SET_COLOR:
            return {
                color: action.color
            };
        default:
            return state;
    }
}

export default color;
```

src/reducers/number.js

```
import * as types from '../actions/ActionTypes';

const initialState = {
    number: 0
};

const number = (state = initialState, action) => {
    switch(action.type) {
        case types.INCREMENT:
            return {
                number: state.number + 1
            };
        case types.DECREMENT:
            return {
                number: state.number - 1
            };
        default:
            return state;
    }
}
export default number;
```

Counter

- 10.서브 리듀서 작성하기 #2

  : reducers 디렉토리의 index.js 에서 redux의 combineReducers 를 불러온 뒤 다음과 같이 호출합니다.

src/reducers/index.js

```
import number from './number';
import color from './color';
import { combineReducers } from 'redux';
/*
    서브 리듀서들을 하나로 합칩니다.
    combineReducers를 실행하고 나면, 나중에 store의 형태가 파라미터로 전달한 객체의 모양대로 만들어집니다.
    지금의 경우:
    {
        numberData: {
            number: 0
        },
        colorData: {
            color: 'black'
        }
    }
    로 만들어집니다.
*/
const reducers = combineReducers({
    numberData: number,
    colorData: color
});

export default reducers;
```

Counter

- 10.서브 리듀서 작성하기 #3

  : CounterContainer 컴포넌트의 mapStateToProps 함수를 아래와 수정합니다.

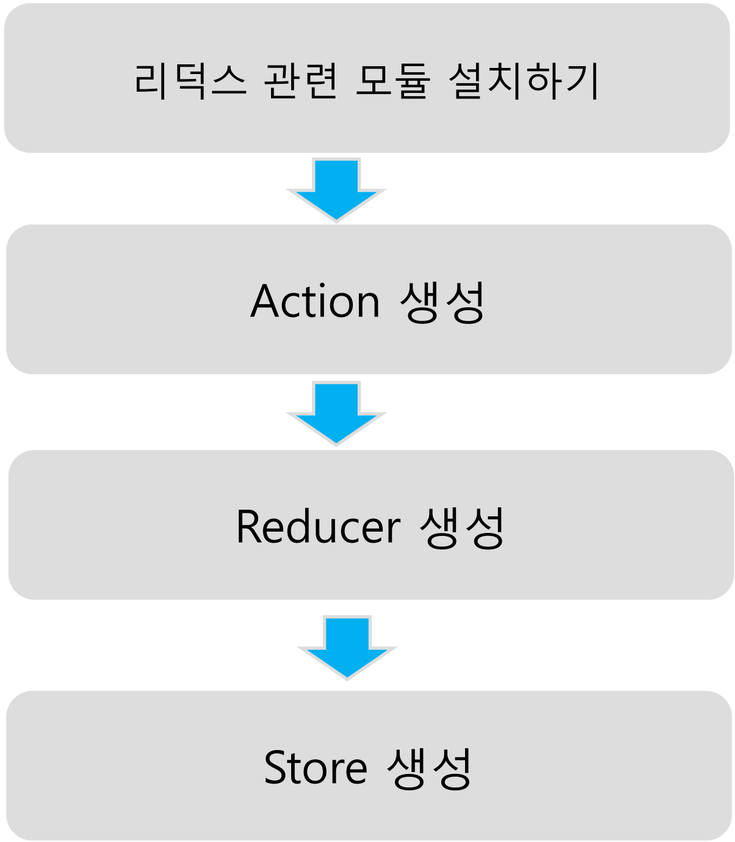
src/containers/CounterContainer.js

```
import Counter from '../components/Counter';
import * as actions from '../actions';
import { connect } from 'react-redux';
import { getRandomColor } from '../utils';

// store 안의 state 값을 props 로 연결해줍니다.
const mapStateToProps = (state) => ({
    color: state.colorData.color,
    number: state.numberData.number
});
// 액션 생성자를 사용하여 액션을 생성하고, 해당 액션을 dispatch 하는 함수를 만들은 후, 이를 props 로 연결해줍니다.
const mapDispatchToProps = (dispatch) => ({
    onIncrement: () => dispatch(actions.increment()),
    onDecrement: () => dispatch(actions.decrement()),
    onSetColor: () => {
        const color = getRandomColor();
        dispatch(actions.setColor(color));
    }
});
// Counter 컴포넌트를 어플리케이션의 데이터 레이어와 묶는 역할을 합니다.
const CounterContainer = connect(mapStateToProps, mapDispatchToProps)(Counter);

export default CounterContainer;
```

일정관리(Todo-List) App
(redux, redux-thunk, axios 사용)
: Server 연동

Todo-List : Redux 적용

컴포넌트 작성 순서



1. redux와 react-redux 설치

```
npm install --save redux react-redux
```

2. src 디렉토리에 sub directory 생성

- actions

: ActionTypes와 Action 생성 함수 코드

- reducers

: Action의 type에 따라 변화를 일으키는 리듀서 함수 코드

Todo-List : Axios

- Axios (https://github.com/axios/axios)

Axios는 현재 가장 성공적인 HTTP 클라이언트 라이브러리 중 하나입니다.

데이터를 외부에서(API 등) 에서 가져오는 방법은 JQuery Ajax 도 있지만, 요청 시 취소와 TypeScript를 지원하는 Axios를 주로 사용합니다.

```
npm install --save axios
```

```
// GET
axios.get('/api/todos')
  .then(res => {
    console.log(res.data)
  })
//axios 요청 메소드의 두 번째 인자로 config 객체를 넘길 수 있습니다
axios.get('/api/todos', {
  params: { // query string
    title: 'React JS'
  },
  headers: { // 요청 헤더
    'X-Api-Key': 'my-api-key'
  },
  timeout: 3000 // 3초 이내에 응답이 오지 않으면 에러로 간주
}).then(res => { console.log(res.data) })
```

Todo-List : Axios

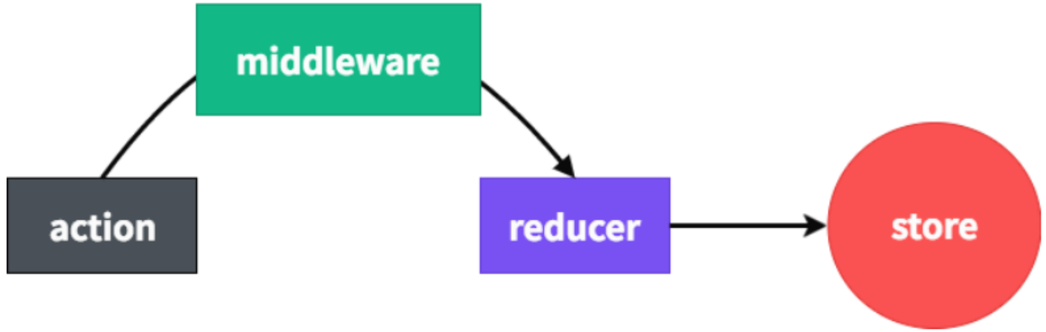
- Axios (https://github.com/axios/axios)

```
// POST
axios.post('/api/todos', {title: "ajax 공부"})
  .then(res => {
    console.log(res.data);
  })

// UPDATE
axios.put('/api/todos/3', {title: "axios 공부"})
  .then(res => {
    console.log(res.data);
  })

// DELETE
axios.delete('/api/todos/3')
  .then(res => {
     console.log(res.data);
  })
```

Todo-List : redux-thunk

- redux-thunk 미들웨어 (https://github.com/reduxjs/redux-thunk)

Redux를 사용 하면서 비동기 작업(예: 네트워크 요청)을 다룰 때는 미들웨어가 있어야 더욱 손쉽게 상태를 관리 할 수 있습니다. 미들웨어는 액션이 디스패치(dispatch) 되어 리듀서에서 이를 처리하기 전에 사전에 지정된 작업들을 설정합니다. 미들웨어를 액션과 리듀서 사이의 중간자로 이해하면 됩니다.



thunk 란?

특정 작업을 나중에 하도록 미루기 위해서 함수 형태로 감싼 것을 말합니다.

첫번째 코드가 실행되면 1 + 2 의 연산이 바로 진행됩니다. 두번째 코드는 1 + 2 의 연산이 코드가 실행 될 때 바로 진행되지 않고 나중에 foo() 함수가 호출 되어야만 연산이 진행됩니다.

```
const x = 1 + 2;
```

```
const foo = () => 1 + 2;
```

Todo-List : redux-thunk

- redux-thunk 미들웨어 (https://github.com/reduxjs/redux-thunk)

```
npm install --save redux-thunk
```

- redux-thunk 미들웨어는 객체 대신 함수를 생성하는 액션 생성 함수를 작성 할 수 있게 해줍니다. 리덕스에서는 기본적으로는 action 객체를 dispatch 합니다.
- 특정 액션이 몇 초 뒤에 실행되게 하거나, 현재 상태에 따라 액션이 무시되게 하려면, 일반 액션 생성자로는 할 수가 없습니다. 하지만 redux-thunk 는 이를 가능하게 합니다.

```
const INCREMENT_COUNTER = 'INCREMENT_COUNTER';

function increment() {
  return {
    type: INCREMENT_COUNTER
  };
}
function incrementAsync() {
  return dispatch => { // dispatch를 파라미터로 가지는 함수를 리턴합니다.
    setTimeout(() => {
      // 1초 뒤 dispatch 합니다
      dispatch(increment());
    }, 1000);
  };
}
```

Todo-List : Express

- Express (https://www.npmjs.com/package/express)

Express 는 Node.js를 위한 빠르고 개방적이며 간결한 웹 프레임워크입니다.

Express 프레임워크를 사용한 Node 웹서버에서 간단한 REST API 를 구현하고, React.js 어플리케이션에서는 axios 라이브러리를 사용하여 비동기적으로 통신합니다.

```
//app.js

var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

```
$ node app.js
```

Todo-List : CORS

- CORS (https://www.npmjs.com/package/cors)

보통 보안 상의 이슈 때문에 동일 출처(Single Origin Policy)를 기본적으로 웹에서는 준수하게 됩니다.

- SOP(Single Origin Policy)

: 같은 Origin에만 요청을 보낼 수 있다.

- CORS(Cross-Origin Resource Sharing)

: Single Origin Policy를 우회하기 위한 기법

: 서로 다른 Origin 간에 resource를 share 하기 위한 방법

- Origin 이란?

: URI 스키마 (http, https) + hostname (localhost) + 포트 (8080, 18080)

Todo-List : Action 생성함수

①. Action type과 Action 생성 함수 : Todo 목록 가져오기

src/actions/index.js

```
import axios from 'axios';

//Action type 정의
export const FETCH_TODOS = "FETCH_TODOS";

const apiUrl = 'http://localhost:4500/api/todos';

export const fetchAllTodos = () => {
    return (dispatch) => {
        axios.get(apiUrl)
             .then(res => {
                dispatch({
                     // 요청이 성공하면, 서버 응답내용을 payload로 설정하여
                     // FETCH_TODOS 액션을 디스패치 합니다.
                    type: FETCH_TODOS,
                    payload: res.data
                })
            })
            .catch(error => {
                console.log(error);
                throw (error);
            })
    }
}
```

① Todo 목록 : Action

Todo-List : Reducer 함수

②. Reducer 함수 : Todo 목록 가져오기

src/reducers/index.js

```
import { FETCH_TODOS  } from '../actions';

const initialState = {
    todos: [
        {
            id: 0,
            text: '',
            checked: false,
        }
    ]
}

export const toDoReducer = (state = initialState, action) => {
    switch (action.type) {
        case FETCH_TODOS:
            return Object.assign({}, state, { todos: action.payload });
        default:
            return state;
    }
}
```

② Todo 목록 : Reducer

Todo-List : Store생성, Middleware적용

③. Store 생성 및 redux-thunk 미들웨어 적용

Store를 생성할 때 redux-thunk 미들웨어를 적용합니다.

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';

import App from './App';
import { toDoReducer } from './reducers';

const store = createStore(toDoReducer, applyMiddleware(thunk));

ReactDOM.render(
<Provider store={store}>
    <App />
</Provider>, document.getElementById('root'));
```

③ index.js 수정

Todo-List : Store생성, redux-devtools 적용

③-1. redux-devtools 를 적용하기

Store를 생성할 때 redux-devtools 적용합니다.

```
npm install --save-dev redux-devtools-extension
```

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import { composeWithDevTools } from 'redux-devtools-extension';

import App from './App';
import { toDoReducer } from './reducers';

const store = createStore(toDoReducer, composeWithDevTools(applyMiddleware(thunk)));

ReactDOM.render(
<Provider store={store}>
    <App />
</Provider>, document.getElementById('root'));
```

③ index.js 수정

Todo-List : Root 컴포넌트 수정

④. App.js : Todo 목록 가져오기

Store에 todos 상태 변수를 저장하기 때문에 <TodoItemList /> 하위 컴포넌트를 호출할 때 props로 todos를 전달할 필요가 없습니다.

src/App.js

```
class App extends Component {
render() {

    return (
      <TodoListTemplate form={<Form
        value={input}
        onKeyPress={handleKeyPress}
        onChange={handleChange}
        onCreate={handleCreate}
      />}>
        <TodoItemList onToggle={handleToggle} onRemove={handleRemove} />
      </TodoListTemplate>

    );
  }
}
```

④ Todo 목록 : App.js 수정

Todo-List : TodoItemList

⑤. TodoItemList.js : Todo 목록 가져오기

componentDidMount() 메서드에서 Action 생성 함수 fetchAllTodos()를 호출한다.

컴포넌트를 store 에 연결 하려면 react-redux의 connect 함수를 사용한다.

상태를 연결시키는 mapStateToProps와, 액션함수를 connect() 인자로 전달하고, TodoItemList 컴포넌트를 파라미터로 전달해 주면 redux store에 연결된 새로운 컴포넌트가 만들어진다.

src/components/TodoItemList.js

```
import { connect } from 'react-redux'
import { fetchAllTodos } from '../actions'

class TodoItemList extends Component {
  componentDidMount() {
    this.props.fetchAllTodos();
  }
}

const mapStateToProps = state => {
  return {
      todos: state.todos
  }
}
export default connect(mapStateToProps, { fetchAllTodos })(TodoItemList);
```

⑤ Todo 목록 : TodoItemList.js 수정

Todo-List : Action 생성함수

①-1. Action type과 Action 생성 함수 : Todo 추가하기

src/actions/index.js

```
import axios from 'axios';

//Action type 정의
export const ADD_TODO = "ADD_TODO";

export const addTodo = (todo) => {
    return (dispatch) => {
        axios.post(apiUrl,todo)
        .then(res => {
            dispatch({
                type: ADD_TODO,
                payload: res.data
            })
        })
        .catch(error => {
            console.log(error);
            throw (error);
        })
    }
}
```

① Todo 추가 : Action

Todo-List : Reducer 함수

②-1. Reducer 함수 : Todo 추가하기

src/reducers/index.js

```
import { FETCH_TODOS, ADD_TODO  } from '../actions';


export const toDoReducer = (state = initialState, action) => {
    switch (action.type) {
        case FETCH_TODOS:
            return Object.assign({}, state, { todos: action.payload });
       case ADD_TODO:
            return Object.assign({}, state, { todos: action.payload });
        default:
            return state;
    }
}
```

② Todo 추가 : Reducer

Todo-List : Root 컴포넌트 수정

④-1. App.js : Todo 추가하기

App.js 에서 관리하던 input 상태변수를 Form.js 로 이동 시키기 때문에, <Form /> 하위 컴포넌트를 호출할 때 props로 input, handleKeyPress, handleChange, handleCreate를 전달하지 않는다.

src/App.js

```
class App extends Component {
render() {

    return (
      <TodoListTemplate form={<Form />}>
        <TodoItemList onToggle={handleToggle} onRemove={handleRemove} />
      </TodoListTemplate>

    );
  }
}
```

④ Todo 추가 : App.js 수정

input 상태변수를 삭제한다.

handleChange(), handleCreate(), handleKeyPress() 메서드를 삭제한다.

render() 메서드 안에 선언된 상수들도 삭제한다.

Todo-List : Form 컴포넌트 수정

⑤-1. Form.js : Todo 추가하기 #1

함수형 컴포넌트를 클래스형 컴포넌트로 변경한다. input state 변수를 정의한다.

handleCreate() 메서드에서 Action 생성 함수 addTodo()를 호출한다.

src/components/Form.js

```
import { connect } from 'react-redux';
import { addTodo } from '../actions';

class Form extends React.Component {
  state = {
    input: ''
  }
  handleChange = (e) => {
    this.setState({
      input: e.target.value // input 의 다음 바뀔 값
    });
  }
  handleCreate = () => {
    this.props.addTodo(
    {
      text: this.state.input,
      checked: false,
    });
    this.setState({ input: '' });
  }
```

⑤ Todo 추가 :
Form.js 수정 #1

Todo-List : Form 컴포넌트 수정

⑤-1. Form.js : Todo 추가하기 #2

handleKeyPress() 메서드를 구현한다.

render() 메서드에서 상태변수와 이벤트 핸들러 메서드 설정을 수정한다. react-redux의 connect() 메서드를 호출한다.

src/components/Form.js

```
  handleKeyPress = (e) => {
    // 눌려진 키가 Enter 이면 handleCreate 호출
    if (e.key === 'Enter') {
      this.handleCreate();
    }
  }

  render() {
    return (
      <div className="form">
        <input value={this.state.input} onChange={this.handleChange}
          onKeyPress={this.handleKeyPress} />
        <div className="create-button" onClick={this.handleCreate}>
          추가
      </div>
      </div>
    );
  }
};
export default connect(null, { addTodo })(Form);
```

⑤ Todo 추가 :
Form.js 수정 #2

Todo-List : Action 생성함수

①-2. Action type과 Action 생성 함수 : Todo 삭제하기

src/actions/index.js

```
import axios from 'axios';

//Action type 정의
export const REMOVE_TODO = "REMOVE_TODO";

export const removeTodo = id => {
    return (dispatch) => {
        axios.delete(`${apiUrl}/${id}`)
        .then(res => {
            dispatch({
                type: REMOVE_TODO,
                payload: res.data
            })
        })
        .catch(error => {
            console.log(error);
            throw (error);
        })
    }
}
```

① Todo 삭제 : Action

Todo-List : Reducer 함수

②-2. Reducer 함수 : Todo 삭제하기

src/reducers/index.js

```
import { FETCH_TODOS, ADD_TODO, REMOVE_TODO  } from '../actions';


export const toDoReducer = (state = initialState, action) => {
    switch (action.type) {
        case FETCH_TODOS:
            return Object.assign({}, state, { todos: action.payload });
        case ADD_TODO:
            return Object.assign({}, state, { todos: action.payload });
        case REMOVE_TODO:
            return Object.assign({}, state, { todos: action.payload });
        default:
            return state;
    }
}
```

② Todo 삭제 : Reducer

Todo-List : Root 컴포넌트 수정

④-2. App.js : Todo 삭제하기

<TodoItemList /> 하위 컴포넌트를 호출할 때 props로 handleRemove를 전달하지 않는다.

src/App.js

```
class App extends Component {
render() {

    return (
      <TodoListTemplate form={<Form />}>
        <TodoItemList onToggle={handleToggle} />
      </TodoListTemplate>

    );
  }
}
```

④ Todo 삭제 : App.js 수정

handleRemove() 메서드를 삭제한다.

render() 메서드 안에 선언된 상수들도 삭제한다.

Todo-List : TodoItemList

⑤-2. TodoItemList.js : Todo 삭제하기

render() 메서드 안에 선언된 onRemove 상수를 삭제한다.

<TodoItem /> 하위 컴포넌트를 호출할 때 onRemove 를 전달하지 않는다.

src/components/TodoItemList.js

```
class TodoItemList extends Component {
  render() {
    const { todos, onToggle } = this.props;
    const todoList = todos.map(
      ({id, text, checked}) => (
        <TodoItem
          id={id}
          text={text}
          checked={checked}
          onToggle={onToggle}
          key={id}
        />
      )
    );
    return (
      <div>
        {todoList}
      </div>
    );
  }
}
```

⑤ Todo 삭제 : TodoItemList.js 수정

Todo-List : TodoItem

⑥-2. TodoItem.js : Todo 삭제하기

handleRemove() 메서드에서 Action 생성 함수 removeTodo()를 호출한다.

src/components/TodoItem.js

```
import { connect } from 'react-redux';
import { removeTodo } from '../actions';

class TodoItem extends Component {
  handleRemove = (id) => {
    this.props.removeTodo(id);
  }
  render() {
    const { text, checked, id, onToggle } = this.props;
    return (
      <div className="todo-item" onClick={() => onToggle(id)}>
        <div className="remove" onClick={(e) => {
          e.stopPropagation(); // onToggle 이 실행되지 않도록 함
          this.handleRemove(id)}
        }>&times;</div>
        <div className={`todo-text ${checked && 'checked'}`}>
          <div>{text}</div>
        </div>
        { checked && (<div className="check-mark">✓</div>) }
      </div>
    );
  }
export default connect(null, { removeTodo })(TodoItem);
```

⑥ Todo 삭제 :
TodoItem.js 수정

Todo-List : Action 생성함수

①-3. Action type과 Action 생성 함수 : Todo 토글링하기

src/actions/index.js

```
import axios from 'axios';

//Action type 정의
export const TOGGLE_TODO = "TOGGLE_TODO";

export const toggleTodo = todo => {
    return (dispatch) => {
        axios.put(`${apiUrl}/${todo.id}`, todo)
        .then(res => {
            dispatch({
                type: TOGGLE_TODO,
                payload: res.data
            })
        })
        .catch(error => {
            console.log(error);
            throw (error);
        })
    }
}
```

① Todo 토글링 : Action

Todo-List : Reducer 함수

②-3. Reducer 함수 : Todo 토글링하기

src/reducers/index.js

```
import { FETCH_TODOS, ADD_TODO, REMOVE_TODO, TOGGLE_TODO  } from '../actions';


export const toDoReducer = (state = initialState, action) => {
    switch (action.type) {
        case FETCH_TODOS:
            return Object.assign({}, state, { todos: action.payload });
        case ADD_TODO:
            return Object.assign({}, state, { todos: action.payload });
        case REMOVE_TODO:
            return Object.assign({}, state, { todos: action.payload });
        case TOGGLE_TODO:
            return Object.assign({}, state, { todos: action.payload });
        default:
            return state;
    }
}
```

② Todo 토글링 : Reducer

Todo-List : Root 컴포넌트 수정

④-3. App.js : Todo 토글링하기

<TodoItemList /> 하위 컴포넌트를 호출할 때 props로 handleToggle를 전달하지 않는다.

src/App.js

```
class App extends Component {
render() {

    return (
      <TodoListTemplate form={<Form />}>
        <TodoItemList />
      </TodoListTemplate>

    );
  }
}
```

④ Todo 토글링 : App.js 수정

state 상태 변수들도 삭제한다.

handleToogle() 메서드를 삭제한다.

render() 메서드 안에 선언된 상수들도 삭제한다.

Todo-List : TodoItemList

⑤-3. TodoItemList.js : Todo 토글링하기

render() 메서드 안에 선언된 onToggle 상수를 삭제한다.

<TodoItem /> 하위 컴포넌트를 호출할 때 onToggle 전달하지 않는다.

id, text, checked만 <TodoItem />에게 전달한다.

src/components/TodoItemList.js

```
class TodoItemList extends Component {
  render() {
    const { todos } = this.props;
    const todoList = todos.map(
      ({ id, text, checked }) =>
      (<TodoItem id={id} text={text} checked={checked} key={id} />)
    );
    return (
      <div>
        {todoList}
      </div>
    );
  }
}
```

⑤ Todo 토글링 :
TodoItemList.js 수정

Todo-List : TodoItem

⑥-3. TodoItem.js : Todo 토글링하기

handleToggle() 메서드에서 Action 생성 함수 toggleTodo()를 호출한다.

src/components/TodoItem.js

```
import { connect } from 'react-redux';
import { removeTodo, toggleTodo } from '../actions';

class TodoItem extends Component {
  handleToggle = (todo) => {
    this.props.toggleTodo(todo)
  }
  render() {
    const { id, text, checked } = this.props;
    return (
      <div className="todo-item" onClick={() => {
        const todo = { id, text, checked };
        todo.checked = !todo.checked;
        this.handleToggle(todo)
      }}>
        ....
      </div>
    );
  }
export default connect(null, { removeTodo,toggleTodo })(TodoItem);
```

⑥ Todo 토글링 :
TodoItem.js 수정

React-router

SPA

SPA(Single Page Application)이란?

- 단일 페이지 애플리케이션(Single Page Application, SPA)는 모던 웹의 패러다임이다. SPA는 기본적으로 단일 페이지로 구성되며 기존의 서버 사이드 렌더링과 비교할 때, 배포가 간단하며 네이티브 앱과 유사한 사용자 경험을 제공할 수 있다는 장점이 있다.

- SPA는 기본적으로 웹 애플리케이션에 필요한 모든 정적 리소스를 최초에 한번 다운로드 한다. 이후 새로운 페이지 요청 시, 페이지 갱신에 필요한 데이터만을 전달받아 페이지를 갱신하므로 전체적인 트래픽이 감소할 수 있고, 전체 페이지를 다시 렌더링하지 않고 변경되는 부분만을 갱신하므로 새로고침이 발생하지 않아 네이티브 앱과 유사한 사용자 경험을 제공할 수 있다.

- SPA의 핵심 가치는 사용자 경험(UX) 향상에 있으며 부가적으로 애플리케이션 속도의 향상도 기대할 수 있어서 모바일 퍼스트(Mobile First) 전략에 부합한다.

- SPA 단점
  SPA는 서버 렌더링 방식이 아닌 자바스크립트 기반 비동기 모델(클라이언트 렌더링 방식)이다. 따라서 SEO(검색엔진 최적화) 문제는 언제나 단점으로 부각되어 왔던 이슈이다. Angular 또는 React 등의 SPA 프레임워크는 서버 렌더링을 지원하는 SEO 대응 기술이 이미 존재하고 있어 SEO 대응이 필요한 페이지에 대해서는 선별적 SEO 대응이 가능하다.

Routing

Routing이란?

- Routing이란 출발지에서 목적지까지의 경로를 결정하는 기능이다. 애플리케이션의 라우팅은 사용자가 태스크를 수행하기 위해 어떤 화면(view)에서 다른 화면으로 화면을 전환하는 내비게이션을 관리하기 위한 기능을 의미한다. 일반적으로 사용자가 요청한 URL 또는 이벤트를 해석하고 새로운 페이지로 전환하기 위한 데이터를 취득하기 위해 서버에 필요 데이터를 요청하고 화면을 전환하는 위한 일련의 행위를 말한다.

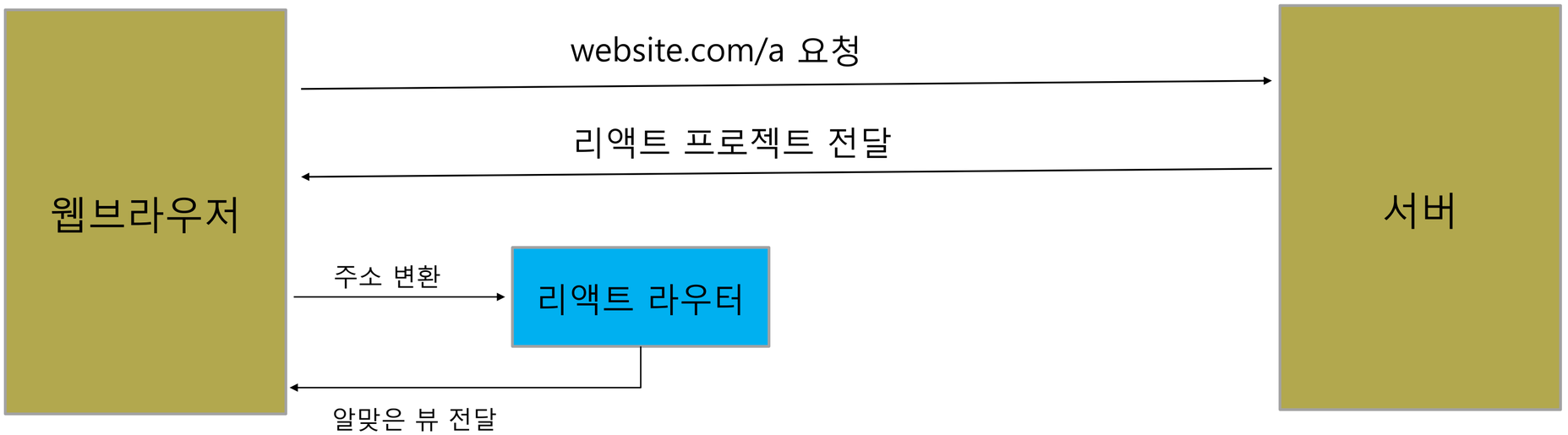
- 브라우저가 화면을 전환하는 경우는 아래와 같다.
  1) 브라우저의 주소창에 URL을 입력하면 해당 페이지로 이동한다.
  2) 웹페이지의 링크를 클릭하면 해당 페이지로 이동한다.
  3) 브라우저의 뒤로 가기 또는 앞으로 가기 버튼을 클릭하면 사용자가 방문한 웹페이지의 기록(history)의 뒤 또는 앞으로 이동한다.

- AJAX 요청에 의해 서버로부터 데이터를 응답 받아 화면을 생성하는 경우, 브라우저의 주소 창의 URL은 변경되지 않는다. 이는 사용자의 방문 history를 관리할 수 없음을 의미하며, SEO(검색엔진 최적화) 이슈의 발생 원인이기도 하다. history 관리를 위해서는 각 페이지는 브라우저의 주소 창에서 구별할 수 있는 유일한 URL을 소유 하여야 한다.

react-router

react-router란?

- react-router는 서드 파티 라이브러리이며, 리액트 애플리케이션에서 라우팅 용도로 가장 많이 사용합니다.
- react-router는 클라이언트 사이드에서 이루어 지는 라우팅을 간단하게 해줍니다. 서버 사이드 렌더링도 도와 주는 도구들이 함께 있습니다.
- react-router는 react-native 에서도 사용 될 수 있습니다.
- 여러 화면으로 구성된 웹 어플리케이션을 만들게 된다면, react-router는 필수 라이브러리 입니다.



react-router

react-router 라이브러리를 이용한 SPA개발

리액트 라우터 적용하기

라우터 생성 및 파라미터 사용하기

라우터 이동하기

라우터 관련 객체 알아보기

```
create-react-app react-router-tutorial
```

```
npm i --save react-router-dom
```

react-router

1. 프로젝트 초기화 및 구조 설정

1-1. src 디렉토리에 sub directory 생성

- components : 컴포넌트들이 위치하는 디렉토리입니다.
- pages : 각 라우터들이 위치하는 디렉토리입니다.

1-2. NODE_PATH 설정

컴포넌트나 모듈을 import 할 때 상대 경로가 아닌 프로젝트의 루트 경로를 지정하여 파일을 절대경로로 import 하기 위한 설정

import MyComponent from '../components/MyComponent' 대신에

import MyComponent from 'components/MyComponent'로 불러 올 수 있다.

react-router

1-2. NODE_PATH 설정

package.json

```
"start": "NODE_PATH=src react-scripts start",
"build": "NODE_PATH=src react-scripts build",
```

windows 에서는 이 코드가 제대로 작동하지 않으므로 cross-env를 설치해야 한다.

```
npm i cross-env --save-dev
```

package.json

```
"start": "cross-env NODE_PATH=src react-scripts start",
"build": "cross-env NODE_PATH=src react-scripts build",
```

위의 설정은 webpack의 resolve 기능을 사용하는데, create-react-app으로 만든 프로젝트에서는 NODE_PATH로 설정된 경로를 자동으로 resolve하기 때문에 따로 webpack 설정을 변경할 필요가 없다.

react-router

2. 컴포넌트 준비

2-1. 메인 컴포넌트

src/App.js

```
import React from 'react';
const App = () => {
  return (
     <div>
      리액트 라우터를 배워봅시다.
     </div>
  );
};
export default App;
```

2-2 Root 컴포넌트

: BrowserRouter를 적용합니다. BrowserRouter는 HTML5의 history API를 사용하여 새로 고침 하지 않고도 페이지 주소를 교체할 수 있다.

src/Root.js

```
import React from 'react';
import { BrowserRouter } from 'react-router-dom';
import App from './App';

const Root = () => {
  return (
    <BrowserRouter>
      <App/>
    </BrowserRouter>
  );
};

export default Root;
```

react-router

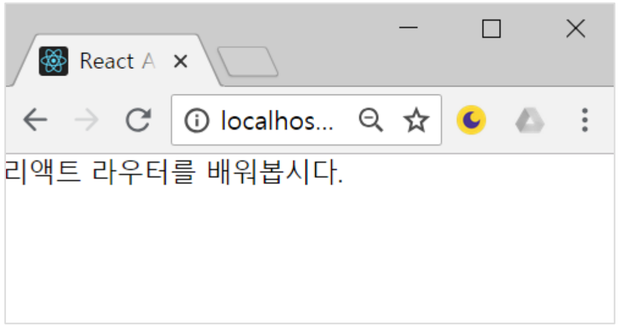
2. 컴포넌트 준비

2-3 index.js 수정

: index.js에서 App아닌 Root를 렌더링 하도록 수정하세요

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import Root from './Root';
import registerServiceWorker from './registerServiceWorker';

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




```
npm start
```

react-router

3. Route와 파라미터

3-1. 기본 라우트 준비

src/pages/Home.js

```
import React from 'react';

const Home = () => {
  return (
    <div>
      <h2>홈</h2>
    </div>
  );
};

export default Home;
```

src/pages/About.js

```
import React from 'react';

const About = () => {
  return (
    <div>
      <h2>소개</h2>
      <p>
        안녕하세요, 저는 리액트 라우터입니다.
      </p>
    </div>
  );
};
export default About;
```

src/pages/index.js

```
/* 다음 코드는 컴포넌트를 불러온 다음에
   동일한 이름으로 내보내 줍니다. */
export { default as Home } from './Home';
export { default as About } from './About';
```

react-router

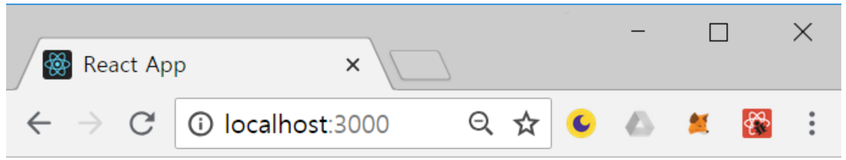
3. Route와 파라미터

3-2 라우트 설정 (App.js 수정)

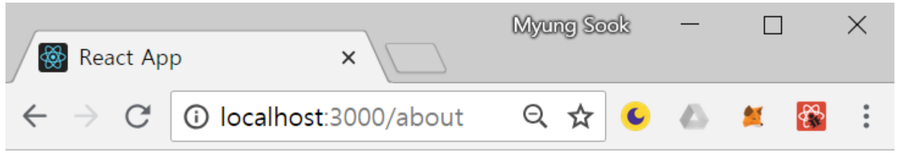
: Route 컴포넌트에서 경로는 path에서 설정하고, 컴포넌트는 component 값으로 설정한다.
: 첫번째 라우트 Home은 주소가 / 와 일치하고, 두번째 라우트는 주소가 /about과 일치할 때만 보이도록 설정함.

src/App.js

```
import React from 'react';
import { Route } from 'react-router-dom';
import { Home, About } from 'pages';

const App = () => {
  return (
    <div>
      <Route exact path="/" component={Home}/>
      <Route path="/about" component={About}/>
    </div>
  );
};

export default App;
```



홈



소개

안녕하세요, 저는 리액트 라우터입니다.

react-router

3. Route와 파라미터

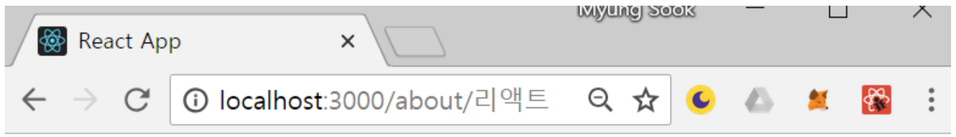
3-3 라우트 파라미터와 쿼리 읽기

: 라우트의 경로에 특정 값을 넣은 방법은
params 사용, Query String 사용

3-3-1) URL params 사용

src/App.js

```
import React from 'react';
import { Route } from 'react-router-dom';
import {
  Home,
  About,
} from 'pages';

const App = () => {
  return (
    <div>
      <Route exact path="/" component={Home}/>
      <Route path="/about" component={About}/>
      <Route path="/about/:name" component={About}/>
    </div>
  );
};
export default App;
```



소개

안녕하세요, 저는 입니다.

소개

안녕하세요, 저는 리액트입니다.

src/About.js

```
import React from 'react';

const About = ({match}) => {
    return (
        <div>
            <h2>소개</h2>
            <p>
                안녕하세요, 저는 {match.params.name}입니다.
            </p>
        </div>
    );
};
export default About;
```

react-router

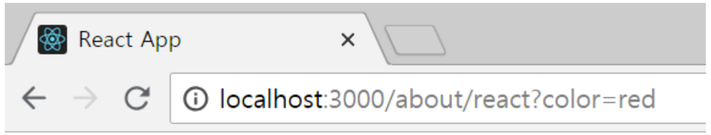
3. Route와 파라미터

3-3 라우트 파라미터와 쿼리 읽기

3-3-2) Query String 사용

http://localhost:3000/about/react?color=red

```
npm i --save query-string
```



src/App.js

```
import React from 'react';
import { Route } from 'react-router-dom';
import {
  Home,
  About,
} from 'pages';

const App = () => {
  return (
    <div>
      <Route exact path="/" component={Home}/>
      <Route path="/about/:name?" component={About}/>
    </div>
  );
};
export default App;
```

src/About.js

```
import React from 'react';
import queryString from 'query-string';

const About = ({location, match}) => {
    const query = queryString.parse(location.search);
    const { color } = query;
    return (
        <div>
            <h2 style={{color}}>소개</h2>
            <p>
                안녕하세요, 저는 {match.params.name}입니다.
            </p>
        </div>
    );
};
export default About;
```

react-router

4. 라우트 이동

4-1. Link 컴포넌트

1) a 태그를 클릭해서 이동시키면 페이지를 새로 고침 하면서 로딩 하기 때문에 라우터에 있는 Link 컴포넌트를 사용해야 합니다.
2) Link 컴포넌트를 사용하면 페이지를 새로 고침 하여 불러 오지 않고, 주소 창 상태를 변경하고 원하는 라우터로 화면을 전환합니다.

src/components/Menu.js

```
import React from 'react';
import {Link} from 'react-router-dom';

const Menu = () => {
  return (
      <div>
         <ul>
           <li><Link to="">홈</Link></li>
           <li><Link to="/about">소개</Link></li>
           <li><Link to="/about/react">React소개</Link></li>
         </ul>
      </div>
  );
};
export default Menu;
```

src/App.js

```
import React from 'react';
import { Route } from 'react-router-dom';
import { Home, About} from 'pages';
import Menu from 'components/Menu';

const App = () => {
    return (
        <div>
            <Menu/>
            <Route exact path="/" component={Home}/>
            <Route path="/about/:name?" component={About}/>
        </div>
    );
};
export default App;
```

react-router

4. 라우트 이동

4-2. NavLink 컴포넌트

1) NavLink 컴포넌트는 현재 주소와 해당 컴포넌트의 목적지 주소가 일치한다면 특정 스타일 또는 클래스를 지정할 수 있다.
2) NavLink 컴포넌트를 사용하면 해당 링크를 활성화 했을 때 activeStyle로 스타일을 지정할 수 있다. CSS 클래스를 적용하려면 activeClassName 값을 지정하면 됩니다.
3) 첫번째와 두번째에 exact가 포함되어 있지 않다면 /about/react 페이지에 들어갈 때 / 경로와 /about 경로를 둘 다 일치하는 것으로 간주하여 모든 링크에 스타일을 적용합니다.

src/components/Menu.js

```
import React from 'react';
import { NavLink } from 'react-router-dom';
const Menu = () => {
    const activeStyle = {
        color: 'green',
        fontSize: '2rem'
    };
    return (
        <div>
            <ul>
                <li><NavLink exact to="/" activeStyle={activeStyle}>홈</NavLink></li>
                <li><NavLink exact to="/about" activeStyle={activeStyle}>소개</NavLink></li>
                <li><NavLink to="/about/react" activeStyle={activeStyle}>React 소개</NavLink></li>
            </ul>
        </div>
    );
};
export default Menu;
```

react-router

4. 라우트 이동

4-3. 자바 스크립트에서 라우팅

1) Link 컴포넌트는 단순히 이동 시키는 기능만 있으므로 자바스크립트 라우팅을 하려면 라우터로 사용된 컴포넌트가 받아 오는 props 중 하나인 history 객체의 push 함수를 활용할 수 있습니다.

src/pages/Home.js

```
import React from 'react';

const Home = ({ history }) => {
    return (
        <div>
            <h2>홈</h2>
            <button onClick={() => {
                history.push('/about/javascript')
            }}>자바스크립트를 사용하여 이동</button>
        </div>
    );
};
export default Home;
```

react-router

5. 라우터 안의 라우터

5-1. Post 컴포넌트

: Post 컴포넌트에서는 params.id 값을 받아 와서 렌더링합니다.

src/pages/Post.js

```
import React from 'react';

const Post = ({match}) => {
    return (
        <p>
            포스트 #{match.params.id}
        </p>
    );
};
export default Post;
```

Post 페이지를 만든 후 pages 디렉토리의 index에 등록합니다.

src/pages/index.js

```
/* 다음 코드는 컴포넌트를 불러온 다음에
   동일한 이름으로 내보내 줍니다. */
export { default as Home } from './Home';
export { default as About } from './About';
export { default as Post } from './Post';
```

react-router

5. 라우터 안의 라우터

5-2. Posts 컴포넌트

: Post 목록을 보여 줄 Posts 컴포넌트는 Link를 만들 때는 현재 주소 뒤에 id를 붙여서 목적지 주소를 설정한다.
: match.url은 현재 라우트에 설정된 경로 "/posts" 를 알려 줍니다.
: Link 아래쪽에는 Route를 사용하여 조건에 따라 원하는 결과를 보여 주도록 설정한다.

src/pages/Posts.js

```
import React from 'react';
import { Post } from 'pages';
import { Link, Route } from 'react-router-dom';

const Posts = ({ match }) => {
  return (
    <div>
      <h3>포스트 목록</h3>
      <ul>
        <li><Link to={`${match.url}/1`}>포스트 #1</Link></li>
        <li><Link to={`${match.url}/2`}>포스트 #2</Link></li>
        <li><Link to={`${match.url}/3`}>포스트 #3</Link></li>
      </ul>
      <Route exact path={match.url} render={() => (<p>포스트를 선택하세요</p>)}/>
      <Route exact path={`${match.url}/:id`} component={Post}/>
    </div>
  );
};
export default Posts;
```

src/pages/index.js

```
/* 다음 코드는 컴포넌트를 불러온 다음에
   동일한 이름으로 내보내 줍니다. */
export { default as Home } from './Home';
export { default as About } from './About';
export { default as Post } from './Post';
export { default as Posts } from './Posts';
```

react-router

5. 라우터 안의 라우터

5-3. App 컴포넌트

: App 컴포넌트에 라우터를 등록한다.

src/App.js

```
import React from 'react';
import { Route } from 'react-router-dom';
import {
    Home,
    About,
    Posts
} from 'pages';
import Menu from 'components/Menu';

const App = () => {
    return (
        <div>
            <Menu/>
            <Route exact path="/" component={Home}/>
            <Route path="/about/:name?" component={About}/>
            <Route path="/posts" component={Posts} />
        </div>
    );
};
export default App;
```

react-router

5. 라우터 안의 라우터

5-4. Menu 컴포넌트

: Menu 컴포넌트에 링크를 추가한다.

src/components/Menu.js

```
import React from 'react';
import { NavLink } from 'react-router-dom';

const Menu = () => {
    const activeStyle = {
        color: 'green',
        fontSize: '2rem'
    };
    return (
        <div>
            <ul>
                <li><NavLink exact to="/" activeStyle={activeStyle}>홈</NavLink></li>
                <li><NavLink exact to="/about" activeStyle={activeStyle}>소개</NavLink></li>
                <li><NavLink to="/about/react" activeStyle={activeStyle}>React 소개</NavLink></li>
                <li><NavLink to="/posts" activeStyle={activeStyle}>포스트 목록</NavLink></li>
            </ul>
        </div>
    );
};
export default Menu;
```

react-router

6. 라우터로 사용된 컴포넌트가 전달 받는 props

6-1. location

: location은 현재 페이지의 주소 상태를 알려 줍니다.

src/pages/Post.js

```
import React from 'react';

const Post = ({location,match}) => {
    console.log('post :',location);
    (...)
};
export default Post;
```

src/pages/Posts.js

```
import React from 'react';

const Posts = ({location,match}) => {
    console.log('posts :',location);
    (...)
};
export default Posts;
```

```
Posts  ▼{pathname: "/posts/2", search: "", hash: "", state: undefined, key: "spa60f"}
          hash: ""
          key: "spa60f"
          pathname: "/posts/2"
          search: ""
          state: undefined
        ▶ __proto__: Object
post : ▼{pathname: "/posts/2", search: "", hash: "", state: undefined, key: "spa60f"}
          hash: ""
          key: "spa60f"
          pathname: "/posts/2"
          search: ""
          state: undefined
        ▶ __proto__: Object
```

react-router

6. 라우터로 사용된 컴포넌트가 전달 받는 props

6-2. match

: match는 <Route> 컴포넌트에서 설정한 path와 관련된 데이터들을 조회할 때 사용합니다.
: 현재 URL이 같은 다른 라우트에서 사용된 match는 다른 정보를 알려 줍니다.
: match 객체는 주로 params를 조회하거나 서브 라우트를 만들 때 현재 path를 참조하는데 사용합니다.

src/pages/Post.js

```
import React from 'react';

const Post = ({location,match}) => {
    console.log('post :',match);
    (...)
};
export default Post;
```

src/pages/Posts.js

```
import React from 'react';

const Posts = ({location,match}) => {
    console.log('posts :',match);
    (...)
};
export default Posts;
```

```
Posts match  ▼{path: "/posts", url: "/posts", isExact: false, params: {…}}
               isExact: false
             ▶ params: {}
               path: "/posts"
               url: "/posts"
             ▶ __proto__: Object
post match : ▼{path: "/posts/:id", url: "/posts/2", isExact: true, params: {…}}
               isExact: true
             ▶ params: {id: "2"}
               path: "/posts/:id"
               url: "/posts/2"
             ▶ __proto__: Object
```

react-router

6. 라우터로 사용된 컴포넌트가 전달 받는 props

6-3. history

: history는 현재 라우터를 조작할 때 사용합니다.
: 예를 들어 뒤쪽 페이지로 넘어가거나, 다시 앞쪽 페이지로 가거나, 새로운 주소로 이동해야 할 때 history가 가진 함수들을 호출합니다.