React 실전 심화 스터디

송동욱 김후정 배민근

목차

CRA 3.0 update

CRA 3.0 update

Hooks를 지원합니다.

React Hooks이라는 최신 기술이 있습니다. Hooks는 컴포넌트를 작성하는 데 있어 class component가 아닌 functional component를 사용하면서, 여러 stateful logic을 지원합니다. 이것은 컴포넌트를 pure한 방향으로 작성할 수 있게 도울 뿐만 아니라 어플리케이션의 속도를 향상시킬 수 있습니다.

CRA 3.0에서는 Hooks를 정식 지원하면서, App.js가 functional component로 작성되어 있습니다.

CRA 3.0 update

Typescript linting을 지원합니다.

CRA 3.0에서는 프로젝트를 Typescript로 작성할 때 linting을 지원합니다. Visual Studio Code에서는 제한된 법칙을 사용하지 않는 경우에 error message를 보여줍니다.

CRA 3.0 update

browserslist를 지원합니다.

package.json에서의 browserlist config는 production과 development에서 사용되는 JS 파일들을 컨트롤할 수 있습니다.

baseUrl에 대해 Absolute Path를 지원합니다.

PostCSS Normalize를 지원합니다.

HoC(High-order-Component)

코드를 작성하면서 반복되는 부분들을 재사용할 수 있는 단위로 만들어 사용할 수 있습니다. 리액트 컴포넌트를 작성하면서 반복될 수 있는 코드들은 HOC를 만들어서 해결할 수 있습니다. HOC는 하나의 함수로, 함수를 통하여 특정 기능을 부여합니다.

다음은 HOC의 기본적인 포맷입니다. HOC도 컴포넌트입니다!

`withRequest.js`

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

const withRequest = (url) => (WrappedComponent) => {
  return class extends Component {
    render() {
      return (
        <WrappedComponent {...this.props}/>
      )
    }
  }
}

export default withRequest;
```

HoC(High-order-Component)

이것을 사용하게 된다면 포맷은 다음과 같습니다. 컴포넌트를 내보내는 부분에서 Post를 withRequest로 감싸고 있습니다.

Post.js

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

class Post extends Component {
  render() {
    const { data } = this.props;
    if (!data) return null;
    return (
      <div>
        { JSON.stringify(this.props.data) }
      </div>
    );
  }
}
export default withRequest('https://jsonplaceholder.typicode.com/posts/1')
(Post);
```

HoC(High-order-Component)

만약에 Comment.js처럼 유사한 기능을 하는 컴포넌트의 경우 withRequest의 api 주소 부분만 바꿔주면 됩니다.

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

class Comments extends Component {
  render() {
    const { data } = this.props;
    if (!data) return null;
    return (
      <div>
        {JSON.stringify(data)}
      </div>
    );
  }
}
export default withRequest('https://jsonplaceholder.typicode.com/comments?
postId=1')(Comments);
```

이렇게 HOC를 사용하면 반복되는 코드들을 많이 줄일 수 있습니다.

Hooks

Hooks는 React v16.8에 새로 도입된 기능으로, 앞서 CRA 3.0버전에 정식으로 탑재된 기능힙니다. 기존의 클래스형 컴포넌트에서 컴포넌트 라이프 사이클을 이용하여 처리하던 다양한 기능들을 함수형 컴포넌트에서도 사용할 수 있게 하는 기능입니다. 함수형 컴포넌트만을 사용하면서 어플리케이션의 속도를 높힐 수 있다는 장점이 있습니다. (클래스형 컴포넌트 대비 함수형 컴포넌트는 최대 20% 정도 빠릅니다.) 이번 스터디 발표에서는 가장 대중적으로 사용되는 useState와 useEffect에 대해 간단히 설명드리도록 하겠습니다.

Hooks - useState

useState는 가장 기본적인 Hook입니다. 함수형 컴포넌트에서도 State를 가질 수 있게 해 주는데요, 클래스의 State와 유사하게 이해하실 수 있습니다. 다음은 예제 코드입니다.

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

const Counter = () => {
  const [value, setValue] = useState(0);
  return (
    <div>
      <p>
        현재 카운터 값은 <b>{value}</b> 입니다.
      </p>
      <button onClick={() => setValue(value + 1)}>+1</button>
      <button onClick={() => setValue(value - 1)}>-1</button>
    </div>
  );
};
```

Hooks - useState

useState은 다음과 같은 방법으로 사용합니다.


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

const Counter = () => {
  const [value, setValue] = useState(0);
  return (
    <div>
      <p>
        현재 카운터 값은 <b>{value}</b> 입니다.
      </p>
      <button onClick={() => setValue(value + 1)}>+1</button>
      <button onClick={() => setValue(value - 1)}>-1</button>
    </div>
  );
};
```

Hooks - useState

useState은 다음과 같은 방법으로 사용합니다.

`const [value, setValue] = useState(0);`

이 문법은 배열 비구조화 할당 문법인데요, 예제는 다음과 같습니다.

```
const array = ['dog', 'cat', 'sheep'];
const [first, second] = array;
console.log(first, second); // dog cat
```

useState에 대해 살펴보면, value 는 상태값이고, setValue는 'value'라는 상태값을 바꿀 때 사용되는 함수입니다. useState(0)에서 0 은 value의 초기값입니다.

useState를 이용하면 클래스형 컴포넌트를 사용하지 않고도 상태 관리를 할 수 있다는 장점이 있습니다.

Hooks - useEffect

useEffect는 리액트 컴포넌트가 렌더링 될 때마다 특정 작업을 수행하도록 설정할 수 있는 Hook입니다. 클래스형 컴포넌트의 componentDidMount 와 componentDidUpdate, 그리고 componentWillUnmount를 합친 형태와 유사하지만, **분명히 다른 점도 존재합니다.** 기본적인 형태는 다음과 같습니다.

```
useEffect(() => {
    console.log('useEffect의 기본 포맷');
  });
```

Hooks - useEffect(마운트될 때만)

useEffect를 이용하면 함수형 컴포넌트에서 마운트 될 때 이벤트를 처리할 수 있습니다. 즉, Class형 컴포넌트에서의 **componentDidMount와 유사한 상황** 과 유사한 상황을 다음과 같이 사용할 수 있습니다.

함수의 두 번째 파라미터로 비어있는 배열을 넣어줍니다.

```
useEffect(() => {
    console.log('마운트 될 때만 실행됩니다.');
  }, []);
```

Hooks - useEffect(업데이트될 때만)

useEffect를 이용하면 함수형 컴포넌트에서 어떤 값이 업데이트 될 때 이벤트를 처리할 수 있습니다. 즉, Class형 컴포넌트에서의 **componentDidUpdate와 유사한 상황** 과 유사한 상황을 다음과 같이 사용할 수 있습니다.

```
componentDidUpdate(prevProps, prevState) {
  if (prevProps.value !== this.props.value) {
    doSomething();
  }
}
```

위 코드는 props안에 들어 있는 value 값이 바뀔 때만 특정 작업을 수행하도록 만든 코드입니다. 이러한 작업은 useEffect를 통해 작업한다면 다음과 같습니다. 값이 바뀌는지 감시하고자 하는 값을 함수의 두 번째 파라미터의 배열 안에 넣어줍니다.

```
 useEffect(() => {
    console.log('value가 바뀔 때만 실행됩니다.');
  }, [value]);
```

Hooks - useEffect : 조심할 점!

useEffect는 componentDidMount + componentDidUpdate + componentWillUnmount가 아닙니다. useEffect 훅을 사용하기 전에 고려해야 하는 점이 있습니다. 이 훅은 조금 특별하고 다르고 멋지기 때문인데요. 클래스 컴포넌트에서 훅으로 리팩토링할 때에 componentDidMount, componentDidUpdate, componentWillUnmount를 하나 이상의 useEffect 콜백 함수로 바꾸게 될 겁니다.

리팩토링이 아니고 useEffect 콜백은 이후에 구동하도록 예정 되었습니다. componentDidMount나 componentDidUpdate와 다르게 useEffect로 예정한 효과는 브라우저가 화면을 업데이트하는 것을 막지 않습니다. 앱이 더 높은 응답성을 제공할 수 있도록 이렇게 동작합니다. 대부분의 효과는 동기적으로 구동될 필요가 없습니다. 동기적으로 동작하는 경우는 레이아웃을 측정해야 한다거나 하는 등 특수한 경우에 해당합니다. 이런 코드는 useLayoutEffect에 넣을 수 있으며 useEffect와 동일하게 동작하는 API입니다.

Hooks - 그 외

Hooks에는 이 두가지 함수 이외에도 useReducer, useContext 등 기존의 클래스형 컴포넌트에서 할 수 있는 기능들을 함수형 컴포넌트에서도 동작하게 하도록 하는 많은 Hook들이 있습니다. 심지어 useMemo나 useCallback을 사용하면 연산을 최적화하거나, 렌더링 성능을 최적화할 수도 있습니다. 만약 여러 컴포넌트에서 비슷한 기능을 공유하게 된다면 로직 재사용성을 위한 Custom Hook도 지원합니다.

Hook은 클래스형 컴포넌트가 필요없이 어플리케이션을 작성할 수 있으나, 클래스형 컴포넌트의 개념들을 모른다면 러닝 커브가 상당히 높습니다. 그렇기에 컴포넌트 라이프 사이클, 리덕스, 콘텍스트 API, ref등 클래스형 컴포넌트의 개념을 익힌 후에 Hooks를 학습하시는 것을 추천드립니다.

Hooks - 그 외

1. 리액트 함수의 제일 상단에 작성해야합니다.

- 조건문, 내부 함수, 반복문 안에 사용하지 않습니다.

2. 리액트 함수 안에서만 사용합니다.

- custom 함수 안에서는 예외적으로 호출 할 수 있습니다.

React-Router

SPA

React Router이 무엇인지 알기 전에 SPA에 대해 알아야 합니다. SPA는 single-page-application으로, 페이지가 하나 존재하는 어플리케이션입니다. 옛날 방식의 어플리케이션은 여러 페이지로 이루어져 있습니다. 유저의 요청 때마다 페이지가 새로고침되고, 페이지의 로딩 때 마다 서버가 렌더링까지 처리해서 보내 주었습니다.

React의 경우 렌더링을 유저 단에서 처리해줌으로써 서버 자원을 아끼고 불필요한 트레픽을 줄여 주었습니다. 유저 어플리케이션(SPA)을 로드 한뒤에 필요한 정보만 선택적으로 보여줍니다. 클라이언트 사이드에서 필요한 정보만 보여지도록 라우팅을 한다는 의미인데요, 이를 react-router가 도와줍니다.

여기서 SPA의 단점이 드러나는데요, 일단 어플리케이션을 모두 로드해 오다 보니 앱의 규모가 커지면 파일 사이즈가 너무 커져서, 유저가 사용하지 않는 페이지 관련 렌더링 파일도 불러옵니다. 하지만 이런 문제는 code splitting을 통해 해결할 수 있으니 문제 없다고 생각해도 될 것 같습니다 :)

React-Router : Route 컴포넌트

```
import React, { Component } from 'react';
import { Route } from 'react-router-dom';
import { Home, About } from 'pages';
class App extends Component {
    render() {
        return (
            <div>
                <Route exact path="/" component={Home}/>
                <Route path="/about" component={About}/>
            </div>
        );
    }
}
export default App;
```

라우트를 설정할때는 Route 컴포넌트를 사용합니다. 경로는 path를 통해 설정합니다. 예를 들어 localhost:3000/about 을 가게 되면 About component을 보여 주게 됩니다. 첫번째 라우트의 경우엔 exact 가 붙어있는데, 이게 붙어있으면 주어진 경로와 정확히 맞아 떨어져야만 설정한 컴포넌트를 보여줍니다. 라우트의 경로에 params 혹은 query를 통하여 특정 값을 넣는 것도 가능합니다.

React-Router : Switch 컴포넌트

```
import React, { Component } from 'react';
import { Route, Switch } from 'react-router-dom';
import { Home, About } from 'pages';
class App extends Component {
    render() {
        return (
            <div>
                <Route exact path="/" component={Home}/>
                <Switch>
                    <Route path="/about/:name" component={About}/>
                    <Route path="/about" component={About}/>
                </Switch>
            </div>
        );
    }
}
export default App;
```

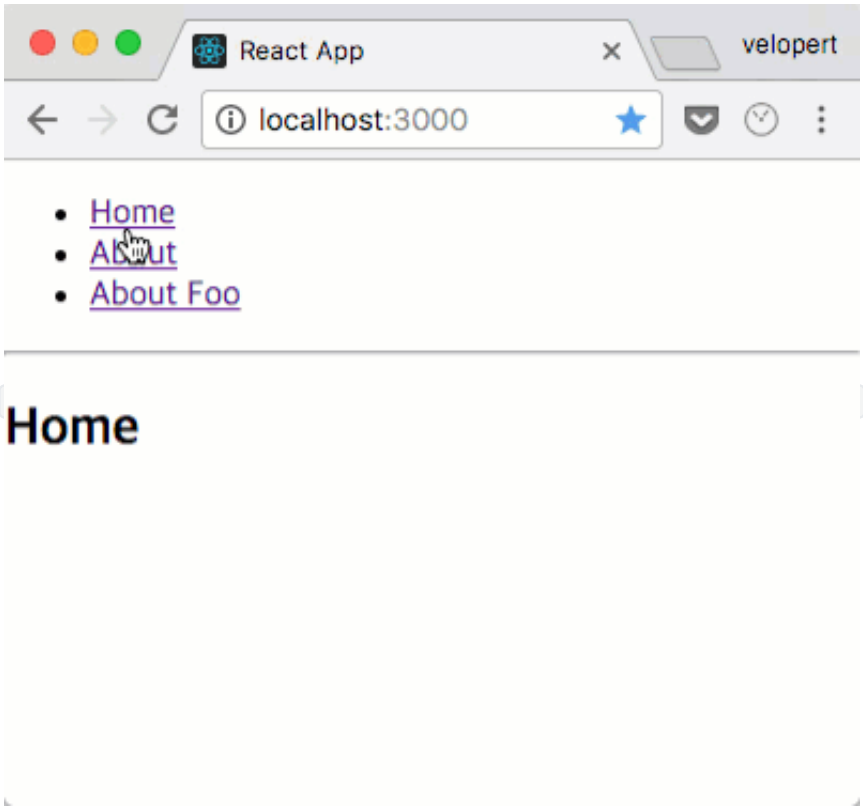
exact를 넣지 않으면 컴포넌트들이 중복되어서 모두 나타납니다. 이를 해결하는 다른 방법은 Switch 컴포넌트를 사용하는 것입니다. 라우트들을 이 컴포넌트에 감까면 매칭되는 첫번째 라우트만 보여주고 나머지는 보여주지 않습니다.

React-Router : Link 컴포넌트

```
import React from 'react';
import { Link } from 'react-router-dom';
const Menu = () => {
    return (
        <div>
            <ul>
                <li><Link to="/">Home</Link></li>
                <li><Link to="/about">About</Link></li>
                <li><Link to="/about/foo">About Foo</Link></li>
            </ul>
            <hr/>
        </div>
    );
};
export default Menu;
```

앱 내에서 다른 라우트로 이동할 때는 a 태그가 아닌 link 태그를 이용해야 합니다. a태그를 이용하면 새로고침을 해서 기존의 데이터가 없어져 버리기 때문입니다. Link 태그를 이용하면 새로고침 없이 원하는 라우트로 새로고침됩니다.

React-Router : Link 컴포넌트



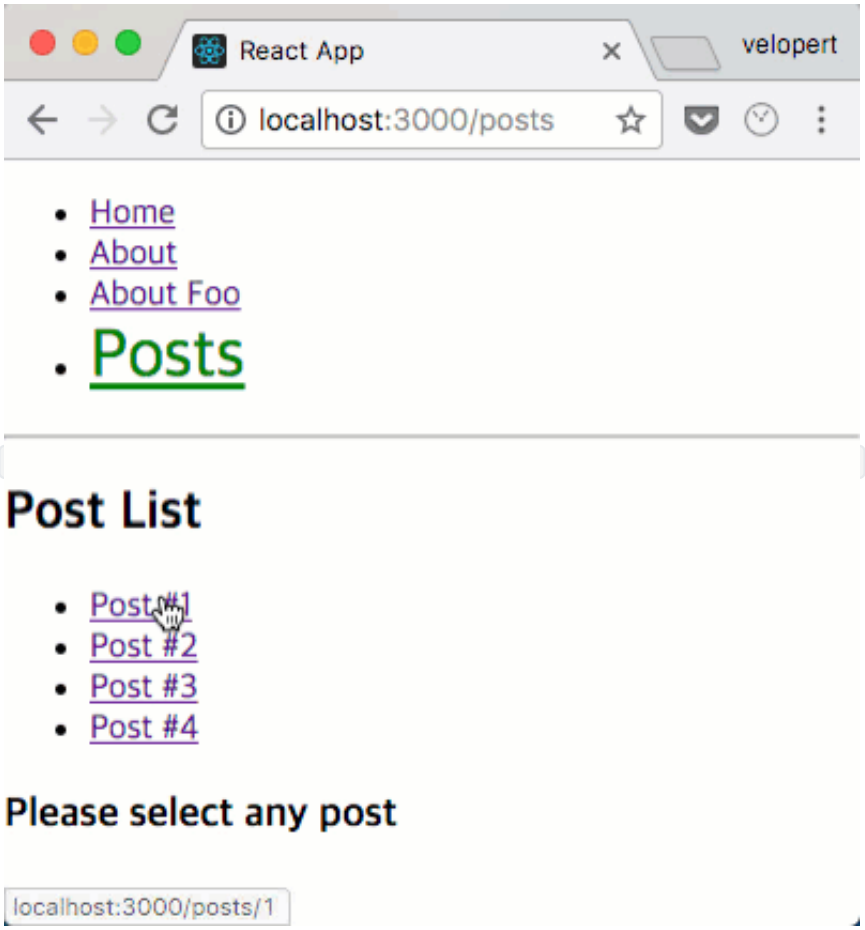
React-Router : 라우트 속의 라우트

react-router v4에서는 라우트에서 보여주는 컴포넌트 내부에 또 Route 컴포넌트를 사용할 수 있습니다.

```
import React from 'react';
import { Link, Route } from 'react-router-dom';
import { Post } from 'pages';
const Posts = ({match}) => {
    return (
        <div>
            <h2>Post List</h2>
            <ul>
                <li><Link to={`${match.url}/1`}>Post #1</Link></li>
                <li><Link to={`${match.url}/2`}>Post #2</Link></li>
                <li><Link to={`${match.url}/3`}>Post #3</Link></li>
            </ul>
            <Route exact path={match.url} render={()=>(<h3>Please select any post</h3>)}/>
            <Route path={`${match.url}/:id`} component={Post}/>
        </div>
    );
};
export default Posts;
```

match.url은 현재 라우트의 경로를 알려줍니다. 첫 번째 Route 컴포넌트는 포스트의 아이디가 주어지지 않았을 때 컴포넌트 대신 그냥 렌더링을 진행합니다. Localhost:3000/posts 의 경우입니다. 그 아래 있는 Route는 포스트 목록에서 포스트를 클릭 했을 때 특정 포스트 컴포넌트를 보여줍니다. Localhost:3000/posts/1 이나 Localhost:3000/posts/2와 같은 경우입니다.

React-Router : 라우트 속의 라우트



위 코드는 다음과 같이 동작하게 됩니다.

코드 스플리팅

웹팩 설정을 통하여 코드를 여러개의 파일로 분리시키는 방법입니다. 이를 통하여 특정 라우트에서 필요한 어플리케이션 데이터만 불러와서 사용할 수 있습니다. 트래픽과 로딩 속도 개선을 기대할 수 있습니다. 기존의 SPA의 단점이었던 큰 자바스크립트 파일을 해결할 수 있습니다.

SSR(ServerSide Rendering)

- 서버사이드 렌더링을 통해 얻을 수 있는 이점
  - 검색엔진 최적화 : 클라이언트 렌더링만 이루어 지는 경우 검색엔진 크롤러가 어플리케이션의 데이터를 제대로 수집하지 못해서 원하는 정보가 검색엔진에 표현되지 않습니다.
  - 성능 개선 : 서버사이드 렌더링을 하게 되면 첫 렌더링된 html을 클라이언트에게 전달해주기 때문에

초기 로딩 속도를 줄일 수 있습니다.

- 서버사이드 렌더링의 단점
  - 프로젝트의 복잡도 : 프로젝트 구조가 많이 복잡해 집니다.
  - 성능 악화 가능성 : 서버사이드 렌더링을 하게 되면서 서버의 부담이 커지고, 서버사이드 렌더링에서 사용하게 되는 renderToString이라는 함수가 동기적으로 작동하여 문제가 될 수 있습니다. 하지만 여러 대안이 있으므로 고려해 볼 수 있습니다.

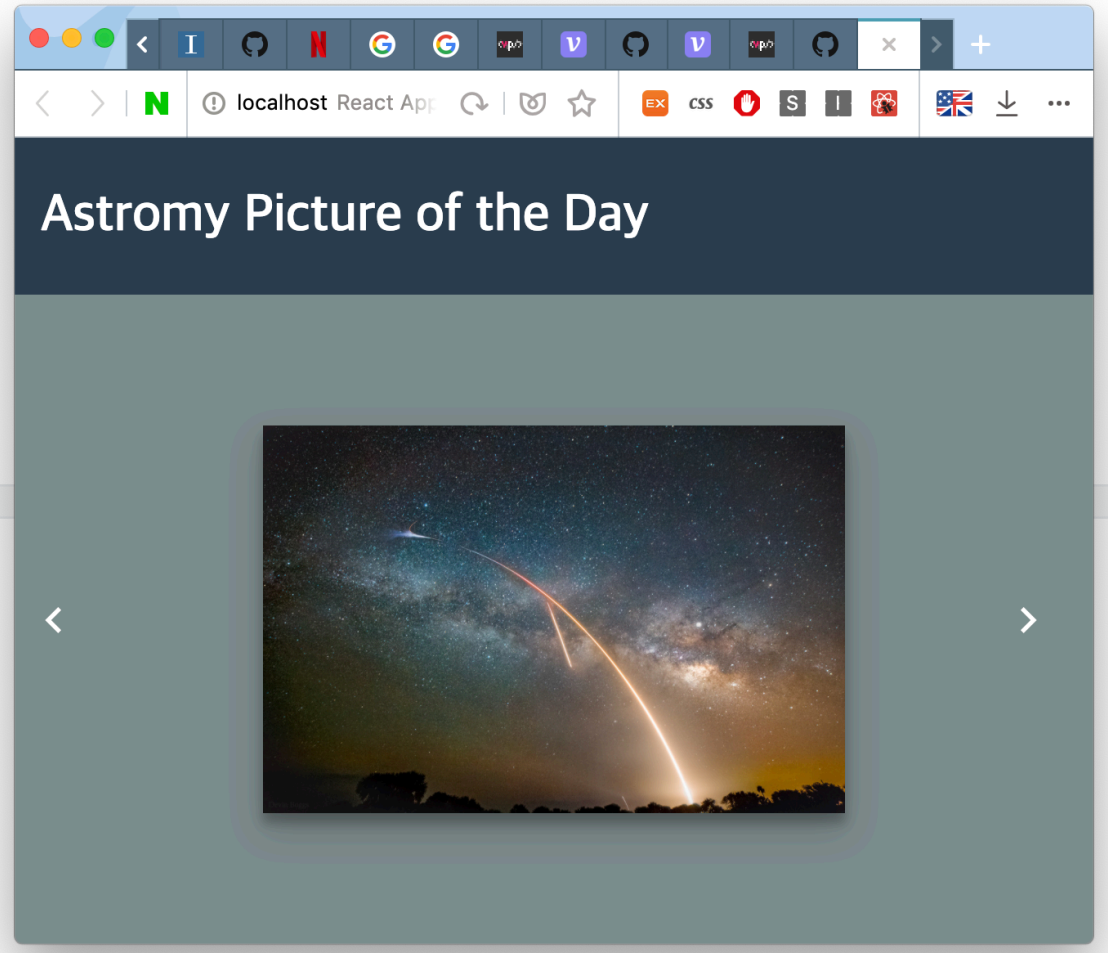
CSR (ClientSide Rendering)

반면에 CSR은 서버에서 View를 렌더링하지 않습니다. HTML과 JS파일, 그리고 여타 서비스에 필요한 리소스를 다운로드한 후 브라우저에서 렌더링합니다. 그렇기에 SSR대비 초기 View를 보기까지의 시간이 오래 걸립니다.

NASA API with Hooks + Styled Component

https://velopert.com/3503 의 Sass를 이용한 프로젝트를 Hooks와 Styled Component를 통하여 진행하였습니다.

https://github.com/baemingun/service-study-2019-spring/tree/master/sdw/nasa-apod 에서 전체 코드를 확인할 수 있습니다.



Hooks를 이용하여 코드를 작성해 본 후기

1. 코드가 굉장히 간결해졌다. Javascript다워졌다.
2. React의 기본적인 개념을 알고 나면 러닝 커브가 그렇게 높지 않다.

MEMOPAD 2019 버전

현재 https://velopert.com/tag/reactcodelab 를 기반으로 새 버전의 MEMOPAD 어플리케이션을 제작하고 있습니다. 위 블로그에서 작성한 MEMOPAD는 2016년에 작성된 것으로, React, Redux, CSS, Express를 이용합니다. 작성 중인 새로운 버전은 Hooks만을 이용한 React, Styled-Component, Express를 사용하여 작업 중입니다. 양이 너무 방대하여 이틀간의 스터디에도 불구하고 미완성 중이며, 완성 시에 블로그를 작성할 예정입니다.