[2022-11-24] 2반 실습 파일

학습 목표

- React에서 비동기 처리를 진행하는 방법은, useEffect 훅 및 axios 라이브러리를 활용하여 학습
- Single Page Application 직접 구현해보고 React에서 페이지를 이동하는 라우팅에 대해 학습

Styled-components 실습

저번 실습 때 못해보았던 실습을 이어서 해보겠습니다.

Step1. 설치

> npm i styled-components
> vscode-styled-components extension 설치 (선택)

++**Auto Import - ES6, TS, JSX, TSX** extension 설치 (탬플릿 리터럴 쓸 때 스타일링 태그 보기 쉽게 컬러링, 스타일링 컴포넌트 이름만 치면 자동 임포트된다던가! 도와줍니다)

Step 2. 실습

```
// Button.styles.js

import styled from 'styled-components'

// props 받아오기
export const CustomButton = styled.button`
    background-color: ${(props) => (props.succeed ? "green": "tomato")};
    color: white;
    outline: none;
    border: none;
    // pseudo class
    &:hover{
        opacity: 0.6;
    }
`
// 상속 - 확장
export const FancyButton = styled(CustomButton)`
    // 기존 background-color 바꿔줘 나머지는 그대로 상속... Polymorphism?다형성?! 이런 개념?! 생각하면 느낌이 오실 것 같아요
    background-image: linear-gradient(to right, tomato , green);
`

// 타입 지정 가능
export const SubmitButton = styled(CustomButton).attrs({
    type: "submit"
```

```
})`
    background-color: blue;
`
`

// App.js

import './App.css';
import { CustomButton, FancyButton, SubmitButton } from './components/Button.styles';
// import * as B from "..."
// 후에 <B.CustomButton></B.CustomButton>로도 사용 가능

function App() {
  // 그냥 succeed 넘겨주면 succeed = true가 넘겨짐
  // as로 태그 종류 바꿔주기. submit button/ a 링크 같은 디자인으로 만들고 싶을 수 있으니
  return (
    <div className="App">
      <CustomButton succeed>Custom succeeded</CustomButton>
      <br></br>
      <CustomButton>Custom failed</CustomButton>
      <br></br>
      <FancyButton as="a">Fancy</FancyButton>
      <br></br>
      <CustomButton type="submit">Custom succeeded</CustomButton>
      <br></br>
      <SubmitButton>Custom succeeded</SubmitButton>
    </div>
  );
}

export default App;
```

- 애니메이션 추가?

```
import styled, {keyframes} from 'styled-components'

// animation
export const rotate = keyframes`
  0%   {background-color: red;}
  25%  {background-color: yellow;}
  50%  {background-color: blue;}
  100% {background-color: green;}
  /* from {
    transform: rotate(0deg);
  } to {
    transform: rotate(360deg)
  } */
`

// props 받아오기
export const FancyButton = styled.button`
    ...
    animation: ${rotate} infinite 2s linear;
`
```

- Theming - 글로벌 변수~

```
// App.js

import {ThemeProvider} from 'styled-components'

// 컴포넌트 레벨에 상관없이 접근 가능한 값. var(--) 이거 생각하면 편리. props.theme으로 꺼냄
const theme = {
  citypop:{
      purple: "#881689",
      pink: "#F628C6",
  },
}


// Button.styles.js
// 타입 지정 가능
export const SubmitButton = styled(CustomButton).attrs({
    type: "submit"
})`
    /* background-color: blue; */
    background-color: ${(props)=>(props.theme.citypop.purple)};
`
```

- Global styling - 글로벌 스타일링~

```
import {ThemeProvider, createGlobalStyle} from 'styled-components'

const GlobalStyle = createGlobalStyle`
  button {
    font-family: '  Times New Roman';
  }
`

function App() {
  <ThemeProvider theme={theme}>
      <GlobalStyle/>
  ...
  </ThemeProvider>

}
```

▼ —save, —save-dev

> `-save-dev` 개발 목적으로만 쓰이는 패키지를 다운받을 때. (유닛테스트, minification 용도…)
>
> `-save` 앱을 실행하는데 꼭 필요한 패키지를 다운받을 때

- framer-motion 리액트 내 애니메이션을 쉽게 줄 수 있는 패키지 (https://www.youtube.com/watch?v=2V1WK-3HQNk&list=PL4cUxeGkcC9iHDnQfTHEVVceOEBsOf07i&index=1)

SSR & CSR

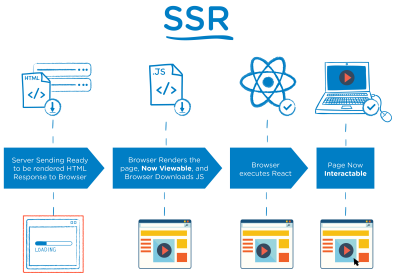
이 두 방식의 차이는 단순히 “렌더링 방식”의 차이에요.

SSR (Server-Side-Rendering)이란?

탭 이동할 때마다 서버로부터 새로운 html 파일을 받아오는 그런 웹사이트

예시> https://news.naver.com/ 언론사별 → 정치 (리로딩)

서버로 요청 → ***이미 보여줄 페이지를 모두 구성한 상태로 페이지 먼저 보여줌*** → 그 사이 자바스크립트 파일 다운 (단, 그동안 유저가 컨텐츠를 볼 수 있지만 조작하지 못하는 상태) → 다운받은 자바스크립트 파일이 실행되며 상호작용이 가능해짐

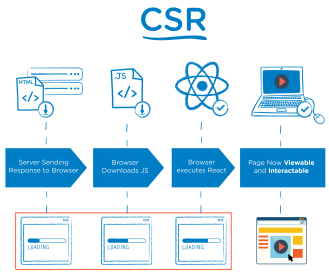


CSR (Client-Side-Rendering)이란? (화면 리로딩 필요 X)

헤더는 고정 부분만 바뀌는 그런 웹사이트

예시> https://www.glowpick.com/?tab=casts&cast-main-cate=2147483647 어워드 → 컬렉션 탭 이동

서버로 요청 → **클라이언트가 컨텐츠 (비어있는 html)파일을 다운 받음, 동시에 모든 자바스크립트 파일도 다운 받아옴 (그 기간 동안 유저는 아무것도 볼 수 없음. 왜냐? 이 방식은 하나의 html을 두고 컴포넌트만 변경시켜 주기 때문)** → 이후 다운 완료된 자바스크립트 파일 실행 → 해당 파일이 dom을 빈 html위에 그려주기 시작.



장단점 비교~!

	SSR	CSR

첫 페이지 로딩	필요한 부분 스크립트만 불러옴. 빠름	모든 스크립트 한번에 불러옴. 느림.
나머지 페이지 로딩 (페이지 이동 시)	다시 로딩 과정 반복. 좀 더 느림.	이미 다 받아와서 빠름. 스무스~한 페이지 이동. (서버에 해당 데이터만 요청하면 되기에)
서버 자원	매 페이지 이동할 때마다 요청. 더 많은 자원 사용.	조금 사용. (빈 뼈대 html 받아올 때 & 데이터 요청할 때만)
SEO	굿굿	불리함
언제 쓰면 좋은가?	네트워크 느릴 때. SEO가 필요할 때. 최초 로딩이 빨라야할 때. 스크립트가 매우 커서 로딩 속도가 느릴 것 같을 때. 웹 사이트 상호작용이 별로 없을 때.	네트워크 굿. 사용자와 상호작용할 것들이 많을 때.

**SEO == 웹사이트, 컨텐츠를 검색엔진에서 더 상위에 노출시킬 수 있에 최적화시키는 것.

웹 크롤러 봇이라는 게 웹상에 있는 모든 웹페이지가 어떤 것인지 파악을 해요

이제 그걸 파악하는데 저희 html 파일에 있는 미리 정의된 meta!를 보게됩니다.

CSR의 경우 맨 처음 요청을 하고 js파일이 실행되기 전까지 빈 뼈대 html만 받아온 상태이기 때문에 검색엔진 최적화가 안될 수 있겠죠?! 그럴 해결하기 위해 react-helmet이라는 패키지도 나왔으니 한번 사용해보시길…

위 렌더링 방식과는 다른 용어!로 아래 MPA, SPA라는 단어도 많이 마주치게 될 건데요, 결론은 위 SSR, CSR은 단순히 렌더링 방식! MPA, SPA는 페이지가 몇개냐 이 차이입니다.

MPA (Multi page application) - 페이지 여러개. SSR 방식을 많이 차용한다…CSR 방식을 차용할 수 있게 도와주는 프레임워크들도 있긴 있습니다.

SPA (Single page application) - 페이지 한개. CSR 방식을 많이 차용한다…

문제 풀이

1. React로 싱글 페이지 웹 서비스 만들기

▼ 문제 정답

```
// App.js

import React from "react";
import { BrowserRouter as Router, Switch, Route, Link } from "react-router-dom";
import "./index.css"
import Home from "./components/Home.js"
import Coffee from "./components/Coffee.js"
import Car from "./components/Car.js"
import Youtube from "./components/Youtube.js"
```

```
function App() {
  return (
    <Router>
      <div className="navbar">
        <nav>
          <ul>
            <li>
              <Link to="/">Home</Link> // <a> 태그 대신 쓰임. 클릭 시 이
// 페이지 리로딩 없이 해당 url로 바뀌고
            </li>
            <li>
              <Link to="/coffee">커피</Link>
            </li>
            <li>
              <Link to="/car">자동차</Link>
            </li>
            <li>
              <Link to="/youtube">유튜브</Link>
            </li>
          </ul>
        </nav>
      </div>

      <div className="box-container">
        <Switch> // 최신 문법에서 <Routes>와 동일한 역할을 해요
          <Route path="/youtube"> // 위 url이 바뀐 걸 감지해서 <Youtube/> 컴포넌트 보여줌
            <Youtube />
          </Route>
          <Route path="/coffee">
            <Coffee />
          </Route>
          <Route path="/car">
            <Car />
          </Route>
          <Route path="/">
            <Home />
          </Route>
        </Switch>
      </div>

    </Router>
  );
}


export default App
```

```
// Coffee.js

import React, { useEffect } from 'react'
import "../index.css"

function Coffee(){
    useEffect(() => {
        document.title = "커피 페이지"
    },[])

    return(
      <div className="box box-coffee">
```

```
        <h2> 커피 관련 페이지입니다. </h2>
      </div>
    );
}


export default Coffee
```

```
// Youtube.js

import React, { useEffect } from 'react'
import "../index.css"

function Youtube(){
    useEffect(() => {
        document.title = "유튜브 페이지"
    },[])

    return(
      <div className="box box-youtube">
        <h2> 유튜브 관련 페이지입니다. </h2>
      </div>
    );
}


export default Youtube
```

▼ 문제 확장

**실습파일 참고~

https://codesandbox.io/s/react-router-vpyd3y?file=/src/components/routes/NotFound.js

2. POST

Axios vs Fetch?

둘 다 프론트에서 서버로 데이터 요청할 때 쓰입니다.

둘 다 특정 url로 요청을 보내게 되면 promise라는 응답보따리를 반환해주는데요, 그건 뒤에서 곧 확인해보겠습니다.

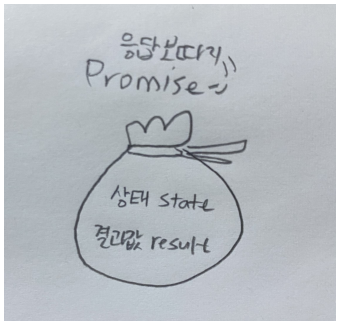
차이! axios는 별도의 설치를 필요로 합니다. 단 fetch 보다 많은 기능을 지원 + 간소화된 문법을 사용하는 장점이 있습니다. // axios는 get요청으로 데이터 받아올 때 뭐… json으로 변환 안해줘도 되고 그렇습니다

(아래 코드 외에 많은 옵션이 있을 수 있으나…자주 쓰는 옵션들 관해서 잘 정리된 링크가 있어 Axios vs Fetch 공유드립니다)

```
 // fetch 사용법?

fetch("url") // 디폴트로 get 방식
  .then((response) => response.json()) // 응답 객체를 json형태로 변환해 받아올 수 있음
  // 아니면 .json(), .text(), .blob()형태...일 수 있어 <- 얘네들은 나중에 마주치면 알아봐주세요
  .then((data) => console.log(data))


fetch("url", {
  method: "POST", // post 방식
  headers: { // 헤더 조작
    "Content-Type": "application/json",
  },
  // 헤더 넣어줌으로써 보내는 데이터가 json이라는 걸 알려줘서 바디에서 json형태로 파싱해서 불러올 수 있음
  body: JSON.stringify({ // 자바스크립트 객체(서버가 이해할 수)를 json(브라우저가 이해)텍스트화 한다.
    // 데이터를 넣어줄 때는 body라는 곳에 문자열로 직렬화를 해서 보내줘야함 JSON.stringify
    title: "Test",
    body: "I am testing!",
    userId: 1,
  }),
})
  .then((response) => response.json()) // 응답 객체를 json형태로 변환해 받아올 수 있음
  .then((data) => console.log(data))
```

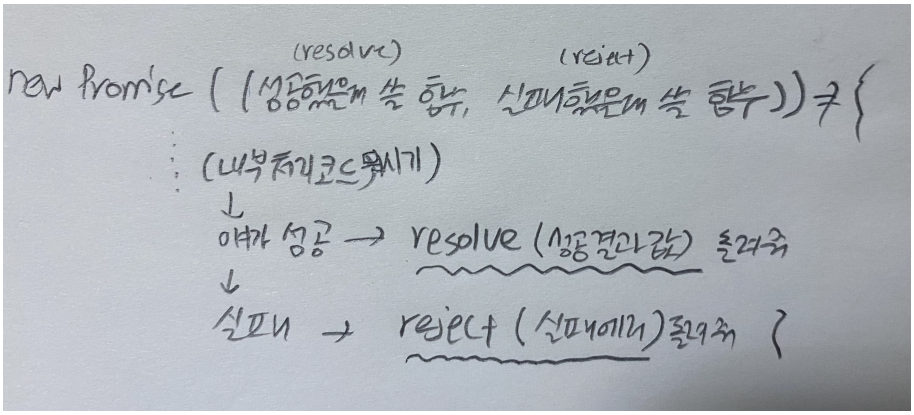
```
// axios 사용법?

axios.get("url")
  .then(function (response) {
    // 성공했을 때
    console.log(response);
  })
  .catch(function (error) {
    // 에러가 났을 때
    console.log(error);
  })
  .finally(function () {
    // 항상 실행되는 함수
  });


axios.post("url", {
    // .. 보낼 데이터
    })
    .then(function (response) {
        // response
    }).catch(function (error) {
        // 오류발생시 실행
    })
```

Promise란?!



응답 보따리 입니다. axios로 요청을 보냈을 때 보따리를 피면 보낸 요청에 대한 상태 state, 결과에 대한 값 result을 확인할 수 있습니다.

Promise 간단한 원리 >



이 객체 내에서 resolve(value)라는 함수가 호출이 되면서 상태가 pending → fulfilled(이행, 성공)으로 바뀜과 동시에 result인 resolve 함수 안에 담겼던 **value**를 돌려주고,

reject(error)라는 함수가 호출이 되면 상태가 pending → rejected(실패!)로 바뀌면서 result에 reject 함수 안에 담겼던 **error**를 돌려줍니다.

Promise 다루는 방법?!! 바로바로 then >

```
프로미스.then(
  (res)=>{}  // fufilled 되었을 때 실행할 함수
).catch(
  (err)=>{} // rejected 되었을 때 실행할 함수
).finally(
  ()=>{} // fulfilled이던 rejected던 완료 후 항상 실행
)
```

▼ 문제 정답

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

function Users() {
  let [result, setResult] = useState('');
  // 삽입할 데이터 객체를 선언하세요.
  console.log('result', result);
  const url = 'https://reqres.in/api/login';
  const login = { email: 'eve.holt@reqres.in', password: 'cityslicka' };

  // axios.post를 호출하고 result에 반환되는 토큰 값을 저장하세요.
  // 방법 1. axios
  axios.post(url, login).then(res => {
    // console.log(res); // 응답보따리
    // console.log(res.data); //  만약 반환되는 데이터
    // console.log(res.status); //  상태코드 확인
    setResult(res.data.token);
  });

  // 방법 2. axios + async - await
  //   const postData = async () => {
  //     try {
  //       const res = await axios.post(url, login);
  //       setResult(res.data.token);
  //     } catch (err) {
  //       console.log(err);
  //     }
  //   };
  //   postData();

  // 방법 3.
  //   fetch(url, {
  //     method: 'POST',
  //     headers: {
  //       'Content-Type': 'application/json',
  //     },
  //     body: JSON.stringify(login),
  //   })
  //     .then(res => res.json())
  //     .then(data => setResult(data.token));

  return (
    <div>
      <h4>React Axios로 HTTP POST 요청하기</h4>
      <div>Token: {result}</div>
    </div>
  );
}

export default Users;
```

▼ 문제 확장

**async/await VS then??!!!

둘다 같은 역할을 해요. 바로바로 비동기 처리 (뭐 하나가 실행이 끝나야 그 다음 걸 하겠다)

실습 예제에서 비교를 해보았듯이 asyn/await을 쓰면 가독성이 좋고 이해하기도 편해보여요.

그렇다고 꼭 async/await을 써야하냐? 그건 상황에 따라 다를 수 있습니다.

비동기 처리가 **연속적으로** 일어나야하고 + 각 promise마다 **서로 다른 로직을** 적용한다고 생각하면 then이 더 유용하게 사용될 수도 있어요. 아래 예제를 async/await으로 바꾼다고 생각해보세요…끔찍합니다끔찍

```
// then을 쓰는 경우...
const multiplyWithPromise = () => {
    const promise = new Promise((resolve, reject) => {
      setTimeout(() => resolve(1), 1000); // (*)
    }).then((result) => { // (**)
      console.log(result); // 1
      return result * 2;
    }).then((result) => { // (***)
      console.log(result); // 2
      return result * 2;
    }).then((result) => {
      console.log(result); // 4
      return result * 2;
    });
    return promise;
}

multiplyWithPromise();
```

3. GET

▼ 문제 정답

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

function Users() {
  let [result, setResult] = useState('');
  // useEffect를 사용하지 않고 아래를 출력해보세요.
  console.log('result', result);
  // 엄청나게 많이 호출됨을 볼 수 있습니다. == 무한루프 why?
  // axios 요청을 보낼 당시에는 실행이 끝나지 않은 상태로 rendering이 된 상태였을 것임
  // axios 요청이 끝나면 setResult함수가 result를 업데이트 시켰을테고,
  // state에 업데이트되면 리렌더링이 일어나 밑에 axios코드가 또 한번 실행이 되면서 계속 무한루프
  // 그걸 막아주기 위하여 useEffect를 사용할 수 있습니다.

  // axios.get을 호출하고 result에 반환되는 데이터를 저장하세요.
  useEffect(() => {
```

```
    // 방법 1. axios + then
    // axios
    //   .get('https://reqres.in/api/users/2')
    //   .then(res => setResult(res.data.data))
    //   .catch(err => console.log(err)); // 동기적으로 처리를 해주죠?

    // 방법 2. axios + async - await
    const fetchData = async () => {
      try {
        const res = await axios.get('https://reqres.in/api/users/2');
        setResult(res.data.data);
      } catch (err) {
        console.log(err);
      }
    };
    fetchData();

    // 방법 3. fetch
    // fetch('https://reqres.in/api/users/2')
    //   .then(res => res.json())
    //   .then(res => setResult(res.data));
  }, []);

  return (
    <div>
      <h4>React Axios로 HTTP GET 요청하기</h4>
      <div>
        Name: {result.first_name + ' ' + result.last_name}
        <br />
        Email:{result.email}
        <br />
      </div>
    </div>
  );
}

export default Users;
```

▼ 문제 확장

**useFetch 훅 만들어보기~ [시간 남으면 진행]

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

const useFetch = url => {
  const [result, setResult] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
  // axios.get을 호출하고 result에 반환되는 데이터를 저장하세요.
  useEffect(() => {
    fetch(url)
      .then(res => res.json())
      .then(res => {
        setResult(res.data);
      })
      .catch(err => {
        setError(err);
```

```
      })
      .finally(() => {
        setLoading(false);
      });
  }, [url]);

  return [result, loading, error];
};

function Users() {
  // axios.get을 호출하고 result에 반환되는 데이터를 저장하세요.
  const [result, loading, error] = useFetch('https://reqres.in/api/users/2');

  return (
    <div>
      <h4>React Axios로 HTTP GET 요청하기</h4>
      <div>
        Name: {loading ? null : result.first_name + ' ' + result.last_name}
        <br />
        Email:{loading ? null : result.email}
        <br />
      </div>
    </div>
  );
}

export default Users
```