개발 일지

day1.

휴대폰에서도 편하게 접속할 수 있게 하기 위해 시작부터 pwa로 개발하기로 함.

css 레이아웃을 잡는데 시간이 좀 걸렸고(

https://www.daleseo.com/css-holy-grail-layout/,

https://opentutorials.org/course/2418/13526,

https://itskeleton.tistory.com/entry/CSS-FlexBox-사용하기

),

아이콘을 넣기위해 fontawesome을 사용했음(

https://fontawesome.com/icons,

https://hymndev.tistory.com/47)

⇒ 링크들을 참고하여 해결

react-router-dom에서 routes, route, link를 불러와 페이지를 여러개로 나누고 경로 설정을 처음 시도했는데 나름 쉽고 잘 작동해서 재미있었음.

아직 전체적인 틀만 만들어 놓았음. 손과 머리에 익지않아서 생각대로 뚝딱 만들어 지지않고, 예전에 들었던 강의들을 참고해서 천천히 만들어 가야함.

---

day2.

클라이언트 사이드 랜더링를 이용할지 서버 사이드 랜더링을 이용할지 아직 정하지 않았기 때문에

일단 데이터를 받아와서 가공하는 것은 뒤로 미루고, Modal을 띄워서 유저 개인 설정 기능을 먼저 만들기로 했다.

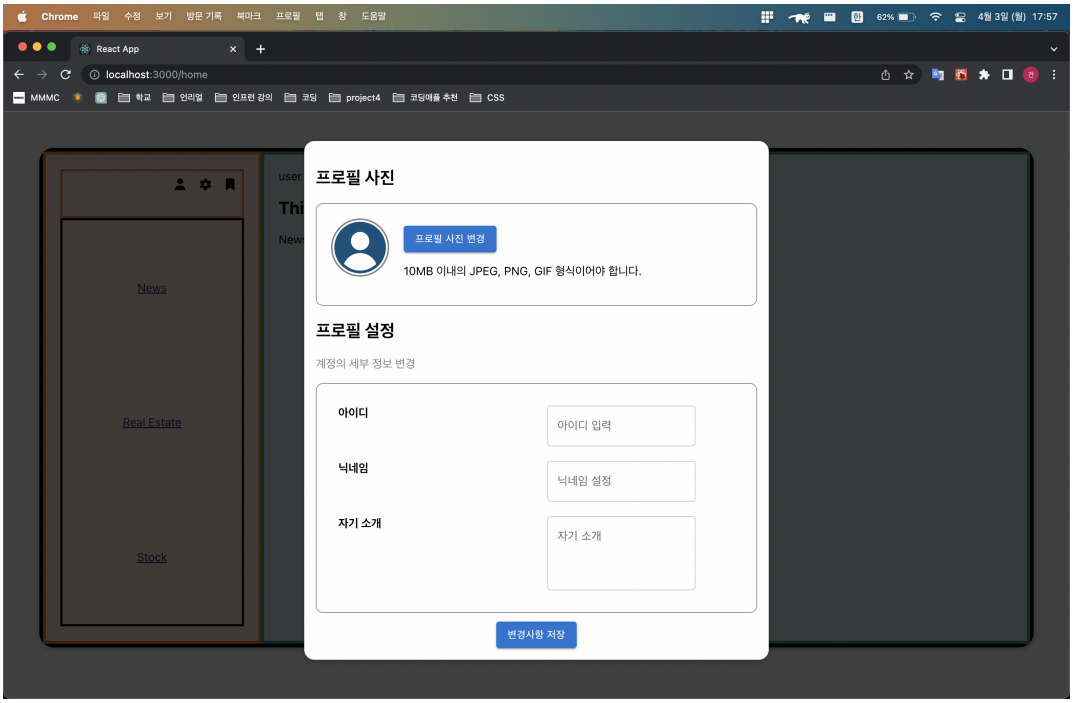
Twitch.com의 유저 설정 UI를 참고하여 만드는 중이다.

좌상단의 아이콘을 클릭하면 모달을 띄워 프로필 설정, 다크모드, 알림 UI가 뜨도록 만들었음.

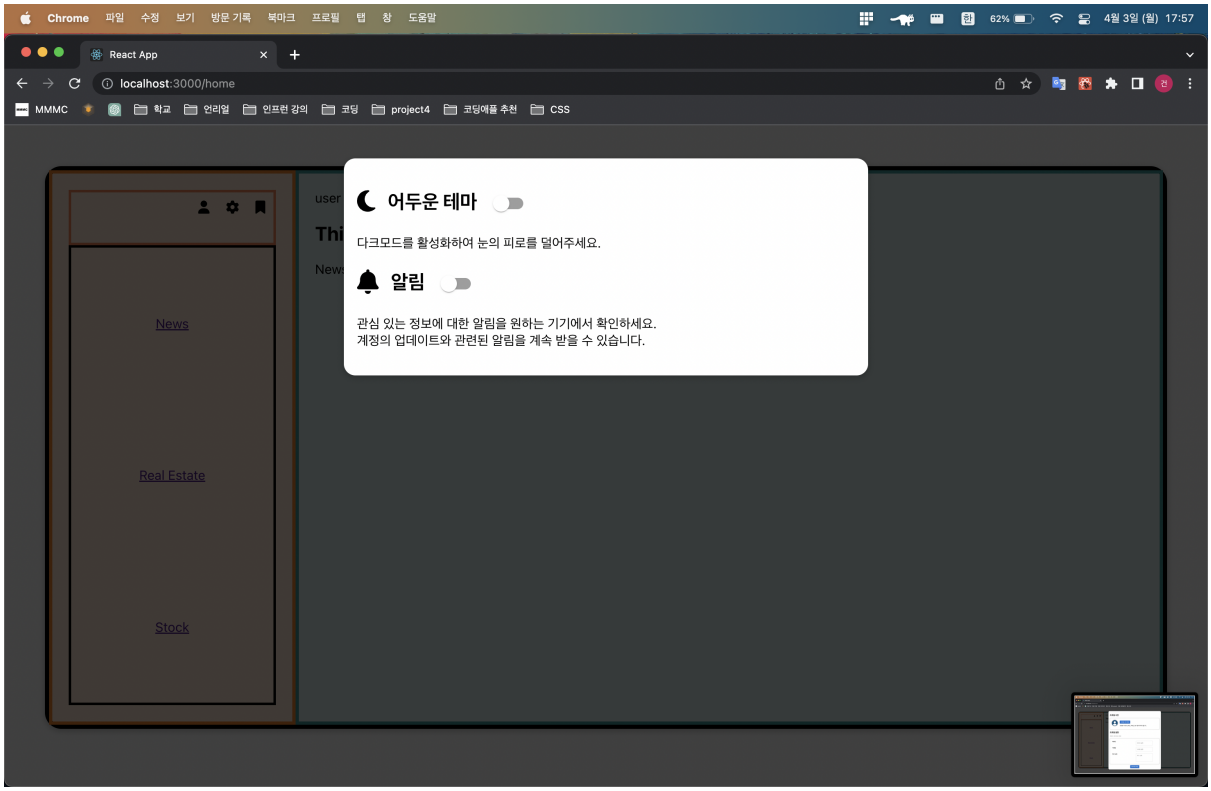
기존에 배웠던 createPotal()기능 사용.

```
(App.jsx)
function App() {
  const [profile, setProfile] = useState(false);

  const onHide = () => {
    setProfile(false)
  };

  return (
  (생략)
    <div className="wrapper">
      {profile === true ? <ProfileModal onHide={onHide} /> : null}
    <div>
```

```
(ProfileModal.jsx)
const ProfileModal = (props) => {
  return (
    <>
      {ReactDOM.createPortal(<BackDrop onHide={props.onHide} />, protalElement)}
      {ReactDOM.createPortal(<Profile />, protalElement)}
    </>
  );
};
```



프로필(유저)버튼을 눌렀을때



설정버튼을 눌렀을때

mui(https://mui.com/)에서 다수의 컴포넌트를 가져왔다.

처음으로 styled-component 사용했음

```
npm install @mui/material @emotion/react @emotion/styled
```

```
import Avatar from "@mui/material/Avatar";
import Button from "@mui/material/Button";
import TextField from "@mui/material/TextField";
import Switch from "@mui/material/Switch";
```

```
import styled from "styled-components";
const ProfileItem = styled.div`
  padding: 20px;
  border: 1px solid gray;
  border-radius: 10px;
`;

const ProfileDetail = styled.div`
  display: flex;
  padding: 10px;
  font-weight: bolder;
`;
```

💡   로 html태그 간에 띄어쓰기를 할 수 있는걸 구글링을 통해 처음 알았다.

브라우저에 설정된 기본 문자들 ⇒ 이런거 101개 있음 https://sensechef.com/957

우리가 HTML 웹 페이지를 만들면 화일 내에 수 많은 공백이 존재한다.

웹 브라우저나 스크립트 엔진 등이 이를 처리하기 위해서는 중간에 여러 개의 공백이 있는 경우 1개만 남겨 두고 모두 제거한다.

따라서 웹 페이지에 여러 개의 공백을 넣어야 한다면   를 반복 입력하면 된다.

특수 문자의 운영 필요성에 대한 이유이다.

& = &

" = &quot

' = '

< = <

> = >

```
<FontAwesomeIcon icon={faMoon} size="lg" />
             어두운 테마  
          <Switch
            onClick={() => {
              onSwitch === "off"
                ? setOnSwitch("on") : setOnSwitch("off");
              console.log(onSwitch);
            }}
```

간단한 UI와 Modal만 만들었고, 상세한 기능와 유저 데이터 저장은 Redux 라이브러리를 이용해 만들것이다.

---

day3.

Loggin 컴포넌트가 App 컴포넌트와 isLoggin 컴포넌트에 둘다 state를 가지고 있어서 상태 변경이 이상하게 변해버림.

내일 Redux를 사용해서 두 컴포넌트 모두 하나의 state를 공유, 관리해야함.

Loggin 컴포넌트는 다른 Modal 컴포넌트들 처럼 화면에 띄워서 표시할 예정.

회원가입 기능은 아직 만들지 않음.

로그인되면 유저 아이콘을 눌렀을때 프로필 설정 가능, 우상단에 로그인 버튼을 프로필로 변경, 로그아웃 버튼 생성.

---

day4.

store.js에 state를 만들어서 LoginModal을 띄우는 것까지 성공. useState를 다룰때랑 너무 달라서 어렵고 복잡하다.

```
import { configureStore, createSlice } from "@reduxjs/toolkit";
```

```
let loginHandler = createSlice({
  name: "login",
  initialState: {
    showLoginModal: false,
  },
  reducers: {
    onShowLogin(state) {
      state.showLoginModal = true;
    },
  },
});

export let { onShowLogin } = loginHandler.actions;

export default configureStore({
  reducer: {
    loginHandler: loginHandler.reducer,
  },
});
```

project2를 참고해서 아이디와 비밀번호 유효성 검사를 추가했다.

useEffect를 이용해서 유저가 입력할 때 마다 유효성을 검사할 수 있게했다.

```
const Login = () => {
  const [enteredId, setEnteredId] = useState(""); // 초기값에 ""를 지정해 주어야 input에 value속성을 줄때 에러가 나지 않
는다.
  const [enteredPassward, setEnteredPassward] = useState("");
  const [idIsValid, setIdIsVaild] = useState();
  const [passwardIsValid, setPasswardIsVaild] = useState();
  const [formIsVaild, setFormIsValid] = useState(true);

  useEffect(() => {
    setFormIsValid(
      enteredId.includes("@") && enteredPassward.trim().length > 7 // 아이디가 @를 포함하는지와 비밀번호가 최소 8자리 이
상인지 검사
    );
  }, [enteredId, enteredPassward]);

  const validateIdHandler = () => {
    setIdIsVaild(enteredId.includes("@"));
  };

  const validatePasswardHandler = () => {
    setPasswardIsVaild(enteredPassward.trim().length > 7);
  };
return (
    <div className={style.modal}>
      <form onSubmit={submitHandler}>
        <div>
          <div className={`${idIsValid == false ? classes.invalid : ""}`}>
            <TextField
              fullWidth
              type="email"
              id="standard-basic"
              label="아이디 입력"
              variant="standard"
              value={enteredId}
              onChange={(e) => {
                setEnteredId(e.target.value);
              }}
              onBlur={validateIdHandler} // onBlur는 유저가 input태그를 떠날 때 실행될 함수를 만들 수 있다.
            />
          </div>
            // element 사이에 띄어쓰기
          <div className={`${passwardIsValid == false ? classes.invalid : ""}`}>
            <TextField
              fullWidth
              type="password"
              id="standard-basic"
```

```
              label="비밀번호 입력"
              variant="standard"
              value={enteredPassward}
              onChange={(e) => {
                setEnteredPassward(e.target.value);
              }}
              onBlur={validatePasswardHandler}
            />
          </div>
        </div>
        <div className={classes.save}>
          <Button type="submit" variant="contained" disabled={!formIsVaild}>
            로그인
          </Button>
        </div>
      </form>
    </div>
  );
};
```

LoginModal를 띄우고 BackDrop을 눌렀을 때 모달창이 사라지지 않는 문제도

store에 있는 onShowLogin()을 조작해서 해결했다.

```
(store.js)
reducers: {
    onShowLogin(state) {
      if (state.showLoginModal == false) {
        state.showLoginModal = true;
      } else {
        state.showLoginModal = false;
      }
    },
  },
```

로그인 버튼을 누르면 localStorage에 아이디와 비밀번호가 추가되도록 해야한다.

그렇게 추가한 후에는 Isloggin.jsx가 로그인, 회원가입 버튼 대신 Avatar만 보여야하고, 유저버튼을 눌렀을 때 회원정보 수정이 보여야함.

아직 로그인 전이라면 로그인 모달띄우기

Avatar를 눌렀을때 로그아웃 버튼이 보이게 만들 예정

로그아웃되면 로그인된 정보를 모두 삭제하고, UI를 다시 원래대로 되돌려야함.

원래 계획보다 훨씬 복잡한 것 같고, 처음부터 계획을 잘 짜놓고 개발해야겠다는 생각이 들었다.

```
(App.js)
import { useDispatch } from "react-redux";
import { onShowLogin } from "./Store/store";

function App() {

  const onHide = () => {
    setProfile(false);
    setSetting(false);
    setBookMark(false);
    dispatch(onShowLogin());
    };

  return (
    (생략)
    <header>
      <h1 className="title">{content} tab</h1>
      <IsLoggin className="login" onHide={onHide} />
```

```
    </header>
  );
```

---

day4.

state와 state변경 함수, useEffect에 대해 좀 더 확실히 공부가 된 것 같다.

sessionStorage에 유저 아이디 비밀번호를 객체 형태로 추가.

sessionStorage는 그냥 한번 사용해 봄

```
const [userArr, setUserArr] = useState([]);

useEffect(() => {
    const set = new Set(userArr);
    const setString = JSON.stringify([...set]);
    sessionStorage.setItem("userLoginInfo", setString);
  }, [userArr]);

<form
    onSubmit={(e) => {
      e.preventDefault();
      let copy = userArr;
      copy = [
        ...userArr,
        { id: e.target[0].value, pass: e.target[1].value },
      ];
      setUserArr(copy);
    }}
>
```

setUserArr를 통해 userArr를 업데이트 해야만 useEffect가 이를 감지하고 호출된다.

여기서 useEffect의 의존성 배열에는 state나 props만 허용 된다.

처음에는 밑에 onSumit에서 copy도 변경 되었으니 의존성 배열에 copy를 넣으면 useEffect가 trigger될 줄 알았다.

```
let copy;
  useEffect(() => {
    setUserArr(copy);
    const set = new Set(userArr);
    const setString = JSON.stringify([...set]);
    sessionStorage.setItem("userLoginInfo", setString);
  }, [copy, userArr]);

<form
    onSubmit={(e) => {
      e.preventDefault();
      console.log("id:" + e.target[0].value);
      console.log("pass:" + e.target[1].value);
      copy = [
        ...userArr,
        { id: e.target[0].value, pass: e.target[1].value },
      ];
    }}
>
```

하지만 copy는 일반 변수이고, state함수로 변경되지도 않았기 때문에 useEffect를 호출하지 못한다.

일반 변수를 의존성 배열에 포함시키는 것은 가능하지만, 주의가 필요하다.

일반 변수는 React의 상태 관리 기능을 사용하지 않는 외부 변수이기 때문에, `useEffect`가 해당 변수의 변경을 감지할 수 없다.

따라서 일반 변수를 의존성 배열에 포함시키는 경우, 해당 변수의 변경 여부와 상관없이 `useEffect`가 실행되지 않는다.

로그인하면 더 이상 로그인 버튼이 보이지 않고, 유저를 눌러도 로그인 모달이 뜨지않는 것을 해결했다.

```
(App.js)
useEffect(() => {
  let geted = sessionStorage.getItem("userLoginInfo");
  geted = JSON.parse(geted);
  if (geted) {
    setIsLogined(true);
    dispatch(onLogined());
    console.log(geted);
  }
}, []);

<FontAwesomeIcon
  className="icon"
  icon={faUser}
  onClick={() => {
    if (loginCheck.alreadyLogin === false) {
      dispatch(onShowLogin());
    } else {
      setProfile(true);
    }
  }}
/>
```

```
(store.js)
initialState: {
    showLoginModal: false,
    alreadyLogin: false,
  },
reducers: {
  onShowLogin(state) {
    if (state.showLoginModal === false) {
      state.showLoginModal = true;
    } else {
      state.showLoginModal = false;
    }
  },
  onLogined(state) {
      state.alreadyLogin = true;
  },
```

`alreadyLogin` 이라는 `state`를 하나 더 만들어서

`showLoginModal state`와 겹치는 것을 막았다.

유저 버튼을 클릭 했을때 `alreadyLogin`이 `false`라면 로그인 모달을 띄우고,

아니면 프로필 수정 모달을 띄운다.

커스텀 훅을 이용해서 컴포넌트가 처음 마운트될 때는 `sessionStorage`에 값이 저장되지 않는 기능만들어야 함

```
(useDidMountEffect.js)
import { useEffect, useRef } from "react";

const useDidMountEffect = (func, deps) => {
  const didMount = useRef(false);

  useEffect(() => {
    if (didMount.current) func();
    else didMount.current = true;
  }, deps);
};

export default useDidMountEffect;
```

```
(LogginModal.jsx)
useDidMountEffect(() => {
    const set = new Set(userArr);
    const setString = JSON.stringify([...set]);
    sessionStorage.setItem("userLoginInfo", setString);
  }, [userArr]);
```

구글링을 통해 해결하려 했지만 작동 안됨.

https://velog.io/@dlruddms5619/React-useEffect-첫-렌더링-시-함수-실행-막기

이제 Avatar를 눌렀을때 로그아웃 버튼이 보이게 만들 예정.

로그아웃되면 로그인된 정보를 모두 삭제하고, UI를 다시 원래대로 되돌려야함.

LogginModal.jsx에 useDidMountEffect 첫 렌더링 시 실행 막기.

state가 너무 많아서 헷갈린다.

---

day5.

useEffect의 첫 랜더링을 막으려고 useDidMountEffect를 사용했지만, 내 코드에서는 먹히지 않았다.

그래서 대신 useRef를 사용했다.

일반변수를 사용하면 재랜더링 시 값이 초기화되기 때문에 useRef를 사용해서 재랜더링 시에도 변수값을 그대로 남겨두었다.

```
(LogginModal.jsx)

let count = useRef(0);

useEffect(() => {
  if (count.current !== 0) {
    console.log(count);
    const set = new Set(userArr);
    const setString = JSON.stringify([...set]);
    sessionStorage.setItem("userLoginInfo", setString);
  }
}, [count.current]);

return(
  <form
    onSubmit={(e) => {
      // e.preventDefault();
      count.current += 1;
      let copy = userArr;
      copy = [
```

```
        ...userArr,
        { id: e.target[0].value, pass: e.target[1].value },
      ];
      setUserArr(copy);
    }}
>
)
```

먼저 count.current 초기값을 0을 준다.

useEffect 내부에 조건문으로 count.current가 0일 때는 내부코드가 실행되지 않도록 하고, 의존성 배열에 count.current를 추가하여 count.current가 변경될 때 마다 useEffect가 실행되게 한다.

form이 submit 될 때 마다 count.current += 1을 해주어서 useEffect 내부가 실행되게 한다.

이제 로그인하면 로그인버튼 대신 Avatar 아이콘이 보이고, 페이지를 새로고침해도 로그아웃(세션 스토리지에서 userLoginInfo를 삭제)하지 않는 이상 다시 로그인할 필요 없다.

```
(store.js)

let loginHandler = createSlice({
  name: "login",
  initialState: {
    showLoginModal: false,
    alreadyLogin: false,
    showProfileModal: false,
  },
  reducers: {
    onShowLogin(state) {
      if (state.showLoginModal === false) {
        state.showLoginModal = true;
      } else {
        state.showLoginModal = false;
      }
    },
    onLogined(state) {
      state.alreadyLogin = true;
    },
    onShowProfile(state) {
      if (state.showProfileModal === false) {
        state.showProfileModal = true;
      } else {
        state.showProfileModal = false;
      }
    },
  },
});
```

ProfileModal.jsx를 LogginModal.jsx처럼 여러곳에서 쓰기 위해

App.js 내부에서 쓰던 기존의 setProfile state를 삭제하고,

store.js에 onShowProfile state를 추가했다.

```
import Button from "@mui/material/Button";
import Menu from "@mui/material/Menu";

const MenuBtn = style.button`
  display: block;
  border:none;
  background-color: transparent;
  font-size:16px;
```

```
  padding:5px 18px;
  width: full-width;
  &:hover{
    cursor:pointer;
    background: cornflowerblue;
    color: white;
    transition: 0.3s;
  }
`;

const IsLoggin = (props) => {
  let loginHandler = useSelector((state) => state.loginHandler);
  const dispatch = useDispatch();

  const [anchorEl, setAnchorEl] = React.useState(null);
  const open = Boolean(anchorEl);
  const handleClick = (event) => {
    setAnchorEl(event.currentTarget);
  };

  const onClose = () => {
    setAnchorEl(null);
  };

  return (
    <div>
      {loginHandler.showLoginModal && <LogginModal onHide={props.onHide} />}
      {props.isLogined === true ? (
        <div>
          <form
            id="logout"
            onSubmit={() => {
              setAnchorEl(null);
              dispatch(onLogined());
              sessionStorage.removeItem("userLoginInfo");
            }}
          >
            <Button id="basic-button" onClick={handleClick}>
              <Avatar
                className={classes.avatar}
                alt=""
                src="/img/project5Profile.png"
              />
            </Button>
            <Menu
              id="basic-menu"
              anchorEl={anchorEl}
              open={open}
              onClose={onClose}
              MenuListProps={{
                "aria-labelledby": "basic-button",
              }}
            >
              <MenuBtn
                className={classes.menuBtn}
                onClick={() => (props.onHideProfile(), setAnchorEl(null))}
              >
                 내 계정 
              </MenuBtn>
              <MenuBtn className={classes.menuBtn} type="submit" form="logout">
                로그아웃
              </MenuBtn>
            </Menu>
          </form>
        </div>

...생략
```

이제 Avatar를 눌렀을때 로그아웃 버튼이 보이고,

로그아웃되면 로그인된 정보를 모두 삭제하고, UI가 로그인 전으로 돌아간다.

💡 로그인 끝

로그인하나 만드는데 일주일 정도 걸린 것 같다.

아직 다크모드 같은 기능이나 프로필 설정 상세 기능들은 만들지 못했다.

처음이라 서툴긴하지만 좀 더 만족할 만한 퀄리티가 나올 수 있도록 더 노력해야겠다.

---

day6.

이제 본격적으로 메인 부분에 보이는 컨텐츠들을 만들려고 뉴스, 부동산, 주식 관련 api들을 찾아 봤는데, 부동산과 주식 관련 api가 구하기 쉽지않다.

공공데이터는 대부분이 xml데이터라 지금 내 수준에서 가공할 수 없고, 받아온다고 해도 단순히 거래량만 보여주거나 데이터의 양이 너무 적다.

네이버 뉴스 api를 써서 뉴스탭 먼저 만들어 보아야겠다.

뉴스 api를 쓰려다가 api 사용가이드 문서를 보는데 http관련 지식이 필요하다는 걸 알게 되었다.

관련 지식이 없으면 프로젝트를 더 진행하는게 무리일 것 같아서 일단 강의로 공부를 좀 더한 다음 프로젝트를 계속하기로 했다.

---

day7.

약 이틀간 HTTP기초를 학습하고, 다시 시작

이제 HTTP가 어떤 기술이고, 어떻게 읽어야하는지, 어떻게 사용하는지, 헤더에 무엇이 들어가고, 그 의미가 무엇인지 대략적으로 알게되었다.

하지만 여전히 HTTP API를 이용할 때 헤더에 내가 원하는 메서드를 넣는 방법은 잘 모르겠어서 좀 더 학습하기로 했다.

원하는 메서드를 넣고 아이디와 시크릿도 넣었지만

그 유명한 CORS에러를 만났다.

네이버 검색 API 예제는 Node.js를 이용해서 요청을 주고 받는것으로 되어있다.

나는 아직 서버를 만들고 데이터를 받을 줄 모르기 때문에

proxy-middleware를 학습 후에 사용하기로했다.

https://www.datoybi.com/http-proxy-middleware/

임시로 사용하고, node.js와 next.js를 따로 공부할 계획이다.

⇒ **실패**

그냥 간단히 서버르 만들어서 사용하기로했다.

5월 시작전에 project5를 마무리 짓고 싶었지만,

생각보다 걸려 넘어지는 부분이나 공부해야할 부분이 많아서 오래 걸린다.

---

day8.

proxy-middleware만들기 실패

⇒ 간단히 Node.js를 사용해서 서버를 만들어 사용했다.

```
(News.js)
import React, { useEffect, useState } from "reac
t";
import NewsItem from "./NewsItem";

import axios from "axios";

const News = () => {
  let [fetchedData, setFetchedData] = useState
([]);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await axios.get("http://l
ocalhost:8070/");
        // console.log(response.data);
        setFetchedData(response.data.items);
      } catch (error) {
        console.error("Error fetching data from se
rver:", error);
      }
    };
    fetchData();
  }, []);

  return (
    <div>
      <div>
        {fetchedData.map((items, i) => (
          <NewsItem
            key={i}
            title={items.title}
            originallink={items.originallink}
            description={items.description}
            pubDate={items.pubDate}
          />
        ))}
      </div>
    </div>
  );
};

export default News;
```

```
(app.js) => server
const express = require("express");
const axios = require("axios");
const cors = require("cors");

const app = express();
app.use(cors());

app.get("/", async (req, res) => {
  const response = await axios.get(
    `https://openapi.naver.com/v1/search/news?quer
y=${encodeURI("tcp")}`,
    {
      headers: {
        "X-Naver-Client-Id": "PHGhD4sYi3g0KUcXkYy
Z",
        "X-Naver-Client-Secret": "Fha83JC9TJ",
      },
    }
  );
  res.send(response.data);
  console.log(response.data);
});

app.listen(8070, function () {
  console.log("listening on 8070");
});
```

setFetchedData(response.data.items);

fetchedData.map((items, i))

라고 하면 되고,

setFetchedData(response.data);

fetchedData.items.map((items, i))

라고하면 안되는 이유가 궁금하다.

❗ **http와 https 구분할 것 !!!**

서버에서 get요청으로 네이버 api로 데이터를 요청하고, localhost:8070포트를 열어둔다.

클라이언트에서 localhost:8070으로 get요청을 보내서 데이터를 받아와서 화면에 표시한다.

딱 두줄을 실행하는데 하루종일 걸렸다.

내일은 fetch해온 데이터를 bookmark에 저장해 봐야겠다.

---

day9.

bookmark 저장 기능을 만들기 전에 무한 스크롤 기능 먼저 만들기로 했다.

News.jsx에 `overflow: scroll;` 를 사용해서 받아온 데이터가 div박스 밖으로 넘어가지 않고, 스크롤 할 수 있게 했다.

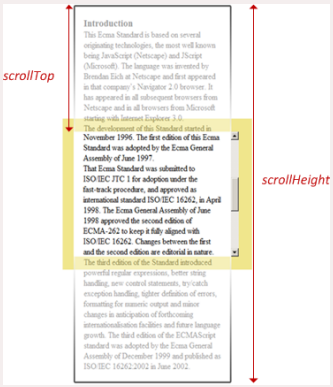
여기서 스크롤이 끝에 다다르면 무한 스크롤링 할 수 있게 구현할 예정이다.

> 💡 무한 스크롤은 말 그대로 무한히 스크롤링을 기능하는 기능을 말한다.
>
> 페이지를 클릭하면 다음 페이지 주소로 이동하는 pagenation과 달리 페이지 하단에 도달하면 새로운 컨텐츠가 한 화면에 추가로 로드된다.
>
> 첫 스크롤 화면에는 적은양의 데이터만 렌더되게끔 만들어주고 사용자가 스크롤의 하단에 근접했을때 다음 데이터를 뿌려주는 기법(?)중에 하나.

스크롤 박스를 움직이려면 DOM을 건드려야해서 useRef를 사용했다.

> 💡 **ref 속성 값에 대해**
>
> 특정 input에 focus 주기, 스크롤 박스 조작, Canvas 요소에 그림 그리기 등
>
> React에서 state로만 해결할 수 없고, DOM을 반드시 직접 건드려야 할 때 사용하게 된다.
>
> `const info = useRef();` 를 통해
>
> `info` 에 ref 객체를 반환합니다.
>
> 그리고 class명이 `box`인 div 요소에 `ref={info}` 값을 부여한다.
>
> **이후 `info.current`를 하면, 위에서 부여한 div 요소에 접근 가능한 id처럼 사용할 수 있다. (`info.current`는 이 div 요소를 가리킨다.)**
>
> 위 용도 외에도 `useRef`는 `값이 변해도 리렌더링하지 않도록 하는 변수`를 관리하는 용도로 사용할 수도 있다.

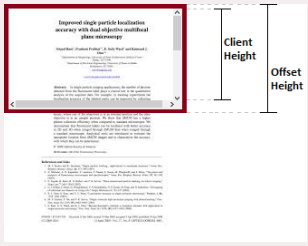
scrollHeight, scrollTop, clientHeight



scrollTop = 원글 맨 처음부터 ~ 현재 화면에 보이는 부분까지의 길이

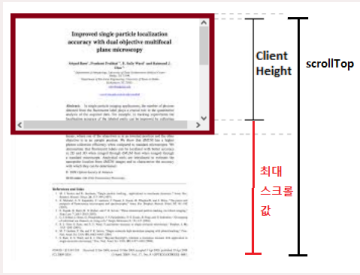
글의 맨 처음 부터 얼마나 내려왔는지 현재 스크롤 위치를 알수있다.

scrollHeight = 화면 바깥으로 삐져나간 부분까지 포함해서 전체 글의 길이



clientHeight = 현재 화면에서 보이는 height 만 말한다. (스크롤 사이즈, 보더 사이즈는 포함 X)

포함한 것은 **OffsetHeight**



스크롤을 끝까지 내렸을 때,

(현재 화면 이후 ~ 맨 아래까지의) 스크롤할 수 있는 화면 길이가 최대 얼마인지를 의미하는 것 같다.

예를 들어 원글의 길이는 0 - 1000px 이다.

현재 화면 범위가 0 - 100px 이라면, 101px - 1000px 까지가 마우스휠로 끝까지 스크롤을 내릴 수 있는 길이인 것이다.

최대 스크롤값은 (1000 - 101) px 인 899px 가 될 것이다.

각 요소에 접근해서 ref의 current값을 얻은 다음 스크롤 영역이 하단에 도달했을 때 데이터를 불러오는 함수를 실행해야한다.

다행이 div의 속성중에 onScroll안에 event객체를 담고있어 scroll에 대한 정보로 쉽게 기능을 구현할 수 있다.

⇒ 스크롤을 진행하는 순간마다 이벤트가 호출되어 메인 스레드 성능에 좋지 않고, 쓰로틀(Throttle)과 같은 최적화 작업이 늘 동반되어야 한다.

하지만 IntersectionObserver는 메인 스레드와 별개로 비동기적으로 실행되기 때문에,

별도의 최적화가 없더라도 기본적으로 훨씬 빠른 퍼포먼스를 보여준다.

그리고 onScroll이벤트를 가지고 하기에는 너무 어려움..

⇒ 따라서 IntersectionObserver를 사용하기로 했다.

참고 : https://nohack.tistory.com/124, https://pks2974.medium.com/intersection-observer-간단-정리하기-fc24789799a3

```
(News.jsx)

import React, { useEffect, useState } from "reac
```

왼쪽 코드
useEffect 내부에서 fetchData함수를 실행해서 데이터를 가져온 후에

```
t";
import NewsItem from "./NewsItem";

import classes from "./News.module.css";

import axios from "axios";

const News = () => {
  let [fetchedData, setFetchedData] = useState
([]);

  useEffect(() => {
    fetchData();
    let timer = setTimeout(() => {
      const observeLastItem = (observer, items) =>
{
        const lastItem = items[items.length - 1];
        observer.observe(lastItem);
      };

      const observerCallback = (entries, observer)
=> {
        entries.forEach((entry) => {
          console.log(entry);
          if (entry.isIntersecting) {
            observer.unobserve(entry.target);
            // 여기서 추가 요청 보내기
            timer = setTimeout(() => {
              observeLastItem(observer, [
                ...document.querySelectorAll(".my-
element"),
              ]);
            }, 1000);
          }
        });
      };

      let items = [...document.querySelectorAll(".
my-element")];

      const observer = new IntersectionObserver(ob
serverCallback, {
        threshold: 0.9,
      });
      observeLastItem(observer, items);
    }, 2000);

    return () => {
      clearTimeout(timer);
    };
  }, []);

  const fetchData = async () => {
    try {
      const response = await axios.get("http://loc
alhost:8070/", {
        params: { display: 10, sort: "sim" },
      });
      // console.log(response.data);
      setFetchedData(response.data.items);
    } catch (error) {
      console.error("Error fetching data from serv
er:", error);
    }
  };

  return (
    <div className={classes.articles}>
      {fetchedData.length === 0 ? (
        <h2>로딩중...!</h2>
      ) : (
        fetchedData.map((items, i) => (
          <NewsItem
```

```
  useEffect(() => {
    fetchData();
    let timer = setTimeout(() => {
      // 4. observeLastItem함수로 마지막 아이템을 찾을
 때까지 관찰한다.
      const observeLastItem = (observer, items) =>
{
        const lastItem = items[items.length - 1];
        observer.observe(lastItem);
      };
      // 5. options.threshold로 정의한 Percent(%) 만
큼 화면에 노출 혹은 제외 되면, entries 배열에 추가하고, Cal
lback Function 을 호출한다.
      const observerCallback = (entries, observer)
=> {
        entries.forEach((entry) => {
          console.log(entry);
          if (entry.isIntersecting) {
          // 6. 노출여부 확인하고 추가 요청 보내고, 이전의
 마지막 아이템 관찰 중지
              // 여기서 추가 요청 보내기
            observer.unobserve(entry.target);
            timer = setTimeout(() => {
              // 7. 새로운 마지막 아이템 찾기
              observeLastItem(observer, [
                ...document.querySelectorAll(".my-
element"),
              ]);
            }, 1000);
          }
        });
      };

      let items = [...document.querySelectorAll(".
my-element")];
        // 1. .map으로 랜더링 된 NewItem들을 모두 가져온
다.
      const observer = new IntersectionObserver(ob
serverCallback, {
        threshold: 0.9,
      });
        // 2. observer를 생성해주고
      observeLastItem(observer, items);
        // 3. observeLastItem에 생성된 items들과 obse
rver를 넘겨 실행한다.
    }, 2000);

    return () => {
      clearTimeout(timer);
    };
  }, []);
```

```
            key={i}
            title={items.title}
            originallink={items.originallink}
            description={items.description}
            pubDate={items.pubDate}
          />
        ))
      )}
    </div>
  );
};

export default News;
```

❗ 하나의 **Intersection Observer Entry** 에 들어있는 속성값

- `target` : Target Element
- `time` : 노출되거나 비노출된 시간
- `isIntersecting` : 노출 여부
- `intersectionRatio` : 노출된 비율
- `intersectionRect` : 노출된 영역
- `boundingClientRect` : TargetElement.getBoundingClie 값
- `rootBounds` : RootElement의 bound 값 만약 옵션에서 지정하지 않았다면, 면 크기이다.

내일 다시 해봐야한다.

재요청을 보냈을 때 받는 데이터도 읽어 와야함, 똑같은 데이터를 불러오면 안된다.

useState를 활용해서 useEffect를 다시 트리거 해야할 것 같다.

⇒ start 파라미터를 get요청에 추가해서 해결 가능

참고: https://swpfun.tistory.com/628

---

day10.

start 파라미터를 get요청에 추가해서 불러오기

불러온 다음 reqCount state를 변경해서 start파라마터 값도 변경해주기

```
(News.jsx)
import React, { useEffect, useRef, useState } from "react";
import NewsItem from "./NewsItem";

import classes from "./News.module.css";

import LinearProgress from "@mui/material/LinearProgress";

import axios from "axios";

const News = () => {
  let [fetchedData, setFetchedData] = useState([]);
  let reqCount = useRef(1);
```

`let reqCount = useRef(1);`를 사용해서 start 파라미터 값을 10씩 더해주어 새로운 데이터를 10개씩 더 받아오게 했다.

처음 fetch해 올 때

`reqCount.current`값을 반드시 넣어주어야

start 파라미터를 넣어줄때 undefind가 생기지 않는다.

```
  let [progress, setPregress] = useState(false);

  useEffect(() => {
    fetchData(reqCount.current); // 초기값 반드시 필요, 없으면 첫
 start파라미터가 없음
  }, []);

  useEffect(() => {
    let timer = setTimeout(() => {
      let items = [...document.querySelectorAll(".my-elemen
t")];

      const observeLastItem = (observer, items) => {
        const lastItem = items[items.length - 1];
        observer.observe(lastItem);
        console.log(lastItem);
      };

      const observerCallback = (entries, observer) => {
        entries.forEach((entry) => {
          console.log(entry);
          if (entry.isIntersecting) {
            observer.unobserve(entry.target);
            setPregress(true);
            if (reqCount.current <= 100) {
              reqCount.current += 10;
              console.log(reqCount.current);
              // 추가 요청 보내기
              fetchData(reqCount.current);
            } else {
                setPregress(false);
              console.log("No More Data");
              return;
            }

            timer = setTimeout(() => {
              setPregress(false);
              observeLastItem(observer, [
                ...document.querySelectorAll(".my-element"),
              ]);
            }, 2000);
          }
        });
      };

      const observer = new IntersectionObserver(observerCall
back, {
        threshold: 0.9,
      });
      observeLastItem(observer, items);
    }, 2000);

    return () => {
      clearTimeout(timer);
    };
  }, []);

  const fetchData = async (reqCount) => {
    console.dir(reqCount);
    try {
      const response = await axios.get("http://localhost:807
0/", {
        params: { display: 5, start: reqCount, sort: "date"
 },
      });
      let copy = [...response.data.items];
      // 현재 상태 값을 업데이트할 때 이전 상태 값을 유지
      setFetchedData((prevData) => [...prevData, ...copy]);
    } catch (error) {
      console.error("Error fetching data from server:", erro
r);
    }
  };
```

reqCount값이 100이 넘으면 그만 받아오게 했다.(데이터가 100개가 넘지 않게 했다. 1000개까지 가능하기는 함)

fetchData함수 내부에서

`setFetchedData((prevData) => [...prevData, ...copy]);`를 사용해 새로운 데이터를 받아올때 이전 데이터를 유지하도록 했다.

처음 데이터를 받아오거나, 추가 데이터를 받아오는 동안 시간적 공백을 채우기 위해 progress state를 하나 만들어서

로딩중 UI를 볼 수 있게 했다.

`let [progress, setPregress] = useState(false);`

```
 . . .
{progress && (
        <LinearProgress style=
{{ padding: "10px", margin:
"20px" }} />
      )}
```

받아올 데이터가 더이상 없다면 로딩중UI는 보이지 않는다.

```
  return (
    <div className={classes.articles}>
      {fetchedData.length === 0 ? (
        <LinearProgress style={{ padding: "10px", margin: "2
0px" }} />
      ) : (
        fetchedData.map((items, i) => (
          <NewsItem
            key={i}
            title={items.title}
            originallink={items.originallink}
            description={items.description}
            pubDate={items.pubDate}
          />
        ))
      )}
      {progress && (
        <LinearProgress style={{ padding: "10px", margin: "2
0px" }} />
      )}
    </div>
  );
};

export default News;
```

이제 fetch해온 데이터를 bookmark에 저장하는 기능과 삭제하는 기능을 만들어 봐야겠다.

저장하기 버튼이 눌려진 해당 NewsItem을 찾아보려고 했지만, 오늘은 실패

각각의 아이템들을 구별해서 저장할지, 저장한 다음 구별할지 선택해야겠다.

처음에 보낼 때 구별해야할 것 같은데 방법을 찾지 못했다.

앞에 강의를 들으면서 했던 미니 프로젝트들 중에 cart에 아이템들을 저장했던 방법을 다시 살펴 봐야겠다.

---

day11.

처음 데이터를 받아왔을때,

NewsItem에서 함수 하나를 만들어서 전달할 props값들을 객체로 만들어 놓고,

어떤 작업(저장하기 버튼 누르기, 상품 추가하기 등)으로 인해 해당 함수가 트리거 되면 그 정보를 가지고 bookMark에서 bookMarkItem으로 만든다.

> ❗ **중요! ⇒ 2시간 삽질**
>
> Redux를 사용해서 state를 나눌 때는 store파일 자체를 하나 더 만드는게 아니라,
>
> slice파일을 하나 더 만들어서 그 state를 store파일에서 import 해온 다음 등록해서 사용한다.

북마크에 저장하는 것까지 완료하긴 했는데

문제는

- 같은 기사를 저장해도 중복 저장되며(id로 구별해서 중복 막고, 삭제 기능도 해결 가능 할 것 같음, 장바구니 만들기 숙제 때 했었음.)

- bookMarkModal이 화면 밑으로 무한정 길어짐 ⇒ 해결!!

---

day12.

로그인하지 않으면 BookMarkModal이 뜨지 않고, LogginModal이 떠서 로그인을 해야해야만 저장기능을 이용할 수 있게 만들었다.

LogginModal은 IsLoggin.jsx파일에서 동작한다!(App.js의 하단 부분 line:123)

```
(bookMarkSlice.js)

import { createSlice } from "@reduxjs/toolkit";

let bookMark = createSlice({
  name: "bookMark",
  initialState: {
    storedArticle: [], // storedArticle 프로퍼티의 초기값을 빈 배
열로 설정
    isExist: false,
  },
  reducers: {
    onStore(state, action) {
      // console.log(action.payload);
      let exist = state.storedArticle.find((origin) => {
        // console.log(origin.id);
        // console.log(action.payload.id);
        return origin.id === action.payload.id;
      });
      if (exist) {
        state.isExist = true;
        // console.log("이미 존재하는 게시물");
      } else {
        state.isExist = false;
        state.storedArticle = [...state.storedArticle, actio
n.payload];
      }
    },
    onDeleteStore(state, action) {
      // console.log(action.payload);
      state.storedArticle = state.storedArticle.filter(
        (origin) => origin.id !== action.payload
        // filter를 사용해서 payload된 id와 같지 않은 item들만 다시
 state에 담음
      );
    },
  },
});

export let { onStore, onDeleteStore } = bookMark.actions;

export default bookMark;
```

1. 기존 state배열에 아무것도 없으면 payload된 객체를 배열에 추가하고,
2. 같은 객체를 배열에 추가하려고 하면

find로 객체를 하나하나 꺼내서

기존 id와 payload된 id를 비교하고

id가 같은 객체가 존재한다면 아무런 동작을 취하지 않고, isExist의 값만 true로 변경한다.

3. 같은 id를 가진 객체가 없다면 기존 배열에 객체를 추가 한다.

삭제할때는

filter를 사용해서 payload된 id와 같지 않은 item들만 다시 새로운 state에 반환한다.

❗ 여러번 하는 실수

onClick이벤트를 붙일 때 바로 실행하고 싶으면

onClick(()⇒{ 실행할 함수 })이렇게 해야한다.

onClick(실행할 함수) 이렇게 하지말 것!

onClick(실행할 함수이름) 이렇게는 가능

+++

`find()` 메서드는 배열 요소를 순서대로 탐색하며, 콜백 함수를 호출하며 콜백 함수가 참을 반환하는 첫 번째 요소를 반환한다. 만약 콜백 함수가 모든 요소에서 거짓을 반환하면 `undefined`를 반환한다.

`findIndex()` 메서드는 array에 있던 자료를 다 꺼내서 콜백함수에 대입해보는데 콜백함수가 참을 반환하는 요소의 index를 알려준다.

완성까지 12일이 걸렸다.

코딩을 한 기간만 12일고, 총 프로젝트 기간은 대략 3주 정도 걸렸다.

활용하고 싶은 기능 중 4/5를 활용했고, 추가하고 싶은 기능 중 1/5을 구현했다.

생각했던 것보다 해야할 일이 굉장히 많았고, 어이없는 실수도 아직하고, 중간중간 부족한 지식들도 너무 많았다.

하지만 처음 코딩을 시작 했을 때는 엄두도 내지 못했던 것들을 하나씩 완성해 나가는 재미가 있다.

---

day13.

검색 기능을 추가로 만들었다.

input으로 유저 입력을 받아서 변수에 저장한 다음 News.jsx로 넘겨주었다.

News.jsx에서 .include()함수를 사용해서, 받아온 데이터의 기사 제목에 유저가 입력한 string이 포함되어있는지 검사한 다음

포함되어 있으면 해당 NewsItem을 렌더링 하게 코드를 작성했다.

❗ 여러번 하는 실수

`filter` 메서드의 콜백 함수에서 반환값을 명시하지 않아서 `searched` 배열에 아무것도 담기지 않게 된다.

따라서 `filter` 메서드의 콜백 함수에서 참/거짓 값을 반환해주어야 한다.

```
const searched = fetchedData.filter((items) => {
  return items.title.includes(userSearchInput);
});

또는 더 간략히 return과 {}를 생략해서

const searched = fetchedData.filter((items) => items.title.includes(userSearchInput));
```

어떤 콜백 함수든 그 값을 어딘가에 담거나 사용하려면 {return 값}에 담아야한다. or return과 {}를 생략해서 적을 수도 있다.

https://splopsky.tistory.com/48를 참고해서

.filter와 .map을 연달아 사용해서 렌더링 해봐야겠다.

따로 만들어서 조건문으로 랜더링하려니 난관이 너무 많다.

---

day14.

```
(News.jsx line.79)

useEffect(() => {
    let timer = setTimeout(() => {
      const observeLastItem = (observer, items) => {
        const lastItem = items[items.length - 1];
        observer.observe(lastItem);
        console.log(lastItem);
      };

      const observerCallback = (entries, observer) => {
        entries.forEach((entry) => {
          if (entry.isIntersecting && userSearchInput ==
= "") {
            observer.unobserve(entry.target);
            if (reqCount.current <= 1000) {
              reqCount.current += 10;
              console.log(reqCount.current);
              // 추가 요청 보내기
              fetchData(reqCount.current);
            } else {
              setProgress(false);
              console.log("No More Data");
              return;
            }

            timer = setTimeout(() => {
              setProgress(false);
              observeLastItem(observer, [
                ...document.querySelectorAll(".my-elemen
t"),

              ]);
            }, 1000);
          }
        });
```

불러온 데이터를 조건에 따라 둘다 렌더링 하는 것이 아니라

애초에 조건문으로 렌더링할 데이터를 필터링해서 한번만 그리는 방법을 알아 냈다.

그리고 검색 완료 후 input에 있는 문자열을 지워서

다시 userSearchInput === ""가 되었을 때

observeLastItem함수가 실행되지 않는 문제가 있었다.

이 문제는

userSearchInput을 의존성 배열에 추가하고,

observerCallback 함수에

&& userSearchInput === "" 조건을 추가해서

input창에 유저의 입력이 없을 때 데이터를 추가 요청할 수 있게 했다.

fetchData 함수도 기존 데이터와 중복된 데이터를 제외한 새로운 데이터만 추가해주기

```
      };

      const observer = new IntersectionObserver(observer
Callback, {
        threshold: 1,
      });
      observeLastItem(observer, [...document.querySelect
orAll(".my-element")]);
    }, 1000);

    return () => {
      clearTimeout(timer);
    };
  }, [userSearchInput]);

let displayItems = fetchedData
    .filter((data) => {
      if (userSearchInput === "") {
        return data;
      } else if (data.title.includes(userSearchInput)) {
        return data;
      }
    })
    .map((data, i) => (
      <NewsItem
        key={i}
        id={data.pubDate}
        title={data.title}
        originallink={data.originallink}
        description={data.description}
        pubDate={data.pubDate}
      />
    ));

  return (
    <div className={classes.articles}>
      {fetchedData.length === 0 ? null : displayItems}
      {progress && (
        <LinearProgress style={{ padding: "10px", margi
n: "20px" }} />
      )}
    </div>
  );
```

위해 !prevData.some을 사용했다.

```
const fetchData = async (startCount) => {
    setProgress(true);
    try {
      const response = await axios.get("htt
p://localhost:8070/", {
        params: { display: 10, start: startC
ount, sort: "date" },
      });
      let newItems = [...response.data.item
s];
      setFetchedData((prevData) => {
        // 기존 데이터와 중복된 데이터를 제외한 새로운
데이터만 추가
        const newItemsFiltered = newItems.fi
lter(
          (item) =>
            !prevData.some((prevItem) => pre
vItem.pubDate === item.pubDate)
        );
        return [...prevData, ...newItemsFilt
ered];
      });
    } catch (error) {
      console.error("Error fetching data fro
m server:", error);
    } finally {
      setProgress(false);
    }
  };이
```

💡 !prevData.some()

`!prevData.some`은 이전 상태인 `prevData` 배열에 존재하지 않는 데이터가 있다면, `true`를 반환한다.

이 구문은 이전 상태 배열에 존재하지 않는 데이터가 있다면, 해당 데이터를 `fetchedData` 배열에 추가하도록 한다.

`Array.some()` 메서드는 배열 요소 중 하나라도 주어진 조건을 만족하는 것이 있는지 판별하여, 만족하는 요소가 있다면 `true`를 반환하고, 그렇지 않으면 `false`를 반환한다.

따라서 `!prevData.some`은 "이전 상태 배열에 존재하지 않는 데이터가 있다면"을 의미한다.

`!` 연산자는 불리언 값의 반대값을 반환하기 때문에, `some()` 메서드의 반환값을 반대로 뒤집어주어야 한다.

추가하고 싶었던 기능 중 2/5 완료.

기능 3개 추가하고 만드는데 2시간..삽질은 멀고도 험하다. 그래도 생각하는대로 만들어져서 기쁨.

css로 색이나 레이아웃까지 좀 더 손봤다.

나름 만족함. 처음 시작했을 때는 엄두도 내지 못했던 것들을 하나씩 구현하는 재미가 있다.

나만의 작은 토이 프로젝트 완성!

💡 +++

`target="_blank" rel="noopener noreferrer"`를 이용해서 링크를 클릭했을 때 새탭에서 열리게 했다.

새 탭에서 링크를 열도록 하려면, a태그의 `target`속성을 `_blank`로 설정하면 된다.

`rel`속성은 현 페이지와 링크된 URL 사이의 관계를 설정한다.

이를 `noopener norefererer`로 설정하면 탭내빙으로 알려진 피싱 공격을 막을 수 있다.

탭내빙?

'역 탭내빙(reverse tabnabbing)'이라고도 하는 '탭내빙(tabnabbing)'은

`window.object` API를 통해 당신의 웹 페이지에 부분적으로 접근하기 위해서

`target="_blank"`와 브라우저의 기본 동작을 이용하는 취약점 공격(exploit)이다.

탭내빙을 사용하면, 연결한 페이지로 인해 당신의 페이지가 가짜 로그인 페이지로 재접속될 수 있다.

일반적으로 원래 탭이 아니라 방금 열린 탭에 집중하기 때문에 대부분의 사용자가 이 문제를 알아차리지 못한다.

그런 다음 사용자가 원래 페이지로 다시 돌아오면 유저의 원래 페이지 대신 가짜 로그인 페이지를 보게 될 것이고,

사용자는 가짜 로그인 페이지에 로그인 정보를 입력할지도 모른다.