리액트(React)

TypeScript Migration

김경민

마이그레이션(Migration)

- 기존 시스템, 코드, 데이터, 환경 등을 새로운 것으로 전환하는 작업 또는 과정
  - 유지보수성 향상
    - 코드 구조 정리, 재사용성 증가
  - 생산성 향상
    - 자동완성, 타입 추론 등 개발자 편의 기능 활용
  - 보안성 및 안정성 강화
    - 최신 기술은 보안 업데이트와 문서가 잘 되어 있음
  - 협업 효율 개선
    - 명확한 규약, 자동화 도구 활용 가능

React + TypeScript 마이그레이션

- 파일 확장자 변경
  - .jsx → .tsx 변경
  - JSX에 타입스크립트 문법 적용 가능하도록 확장자 통일
- 타입 명시 작업
  - Props, state, event 등 주요 요소에 명확한 타입 선언 필요
- 외부 라이브러리 타입 처리
  - @types/패키지명 형식으로 타입 패키지 설치
  - 타입이 없으면 직접 타입 선언하여 사용
- 점진적 리팩토링
  - 전체 코드를 한 번에 바꾸기보다는 컴포넌트 단위로 점진적 마이그레이션 수행

타입스크립트 프로젝트 생성

```
npm create vite@latest ./mylogints

> npx
> cva ./mylogints

|
*  Select a framework:
|    Vanilla
|    Vue
|  > React          리액트 선택
|    Preact
|    Lit
|    Svelte
|    Solid
|    Qwik
|    Angular
|    Others
*  Select a variant:
|  > TypeScript     타입스크립트 선택
|    TypeScript + SWC
|    JavaScript
|    JavaScript + SWC
|    React Router v7 ↗
—
```

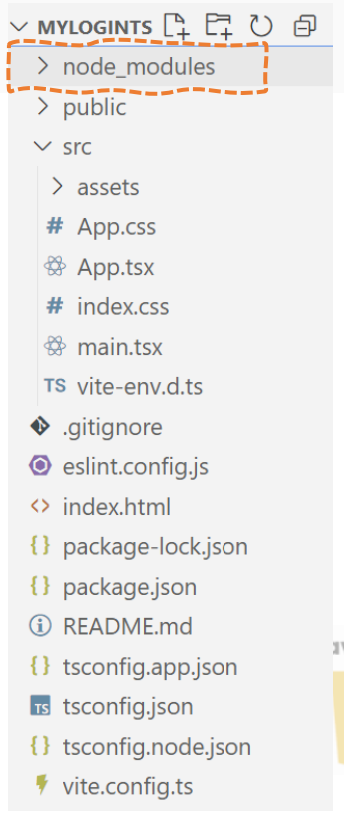
프로젝트에 필요한
라이브러리와 의존성들을 설치

```
mylogints>npm install

ges, and audited 182 packages in 34s

43 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```



타입스크립트 프로젝트 생성

1. Tailwindcss 설치
   - npm install tailwindcss @tailwindcss/vite
2. vite.config.js 수정

```
1  import { defineConfig } from 'vite'
2  import react from '@vitejs/plugin-react'
3  import tailwindcss from '@tailwindcss/vite'
4
5  // https://vite.dev/config/
6  export default defineConfig({
7    plugins: [
8      react(),
9      tailwindcss(),
10   ],
11 })
```

3. index.css 수정

src > # index.css

```
1  @import "tailwindcss";
```

타입스크립트 프로젝트 생성

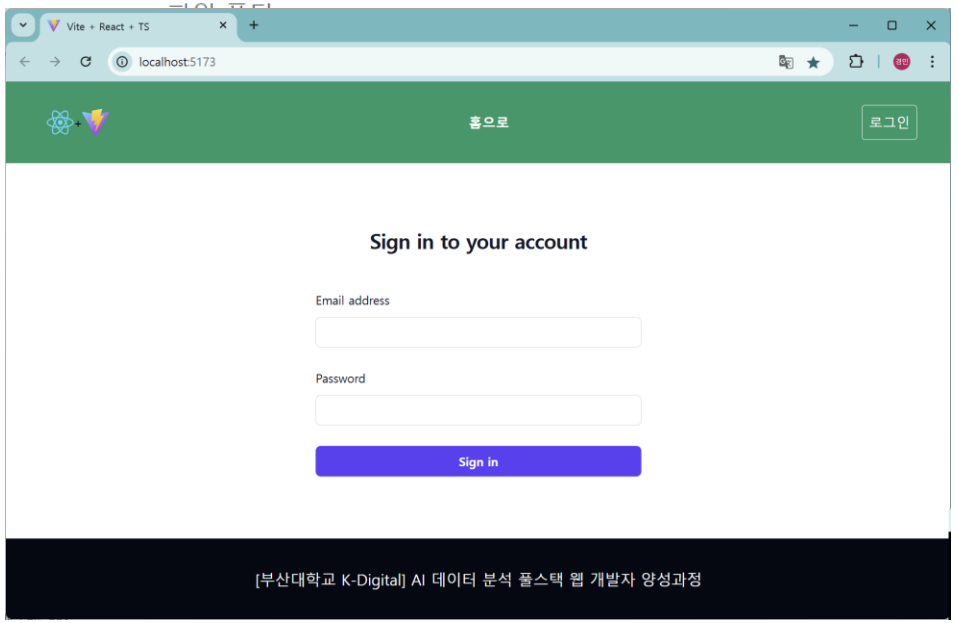
- 리액트 아이콘 설치
  - npm install react-icons -save
- 환경변수 복사 및 .gitignore에 추가
- 라우터 설치
  - npm install react-router-dom
- jotai 설치
  - npm install jotai

파일 확장명 변경

이름	유형
assets	파일 폴더
atoms	파일 폴더
componets	파일 폴더
db	파일 폴더
ui	파일 폴더
App.css	
App.jsx	
index.css	
main.jsx	JavaScript 원본 파...

마이그레이션 파일을 복사하여
jsx-> tsx로 변경

이름	유형
assets	파일 폴더
atoms	파일 폴더
componets	
db	
ui	
App.css	
App.tsx	
index.css	
main.tsx	
vite-env.d.ts	



타입스크립트 선언

- 변수 타입 선언
  - let, const 뒤에 : 타입 형식으로 선언
    - let name: string = "Alice";
    - let age: number = 30;
    - let isStudent: boolean = true;

타입	설명	예시
string	문자열	"hello"
number	숫자 (정수, 소수 등)	42, 3.14
boolean	참/거짓	true, false
null	값이 없음 (JS의 null)	null
undefined	정의되지 않음	undefined
bigint	매우 큰 정수	123n
symbol	고유한 식별자	Symbol("id")

타입스크립트 선언

- 배열 타입 선언
  - number[] , Array<number> 표현식 모두 가능
    - let numbers: number[] = [1, 2, 3];
    - let fruits: Array<string> = ["apple", "banana"];

- 튜플 (Tuple)
  - 배열이지만 요소의 수와 순서, 타입이 고정되어 있음
    - let user: [string, number] = ["Kim", 26];

타입스크립트 선언

- 객체 타입 선언
  - let person: { name: string; age: number } = {
    name: "Lee",
    age: 22,
    };
  - interface나 type으로 분리해서 선언
    - type Person = {
      name: string;
      age: number;
      };
      const p: Person = { name: "Min", age: 25 };

- type
  - 객체, 유니언, 튜플, 기본 타입 등 다양한 타입 정의 가능
  - extends 가능하나, 병합은 불가
- interface
  - 주로 객체의 구조를 정의할 때 사용
  - extends 또는 선언 병합으로 쉽게 확장 가능
  - 여러 번 선언하면 자동으로 병합됨
  - 유니언/교차 타입 지원하지 않음
  - 튜플/기본 타입 조합 불가능

타입스크립트 선언

- 함수 타입 선언
  - 직접 지정
    - function greet(name: string) : string {
      return "Hello, " + name;
      }
  - 화살표 함수
    - const add = (a: number, b: number) : number => a + b;
  - 함수 타입 별도로 정의
    - type MathFunc = (x: number, y: number) => number;
      const multiply: MathFunc = (x, y) => x * y;

타입스크립트 선언

- 리터럴 타입
  - 하나의 구체적인 값만 허용하는 타입
    - let direction: "left";

- 유니언 타입 (Union Type)
  - | 기호를 사용하여 여러 타입 중 하나를 허용
    - let value: string | number;
    - type Direction = "left" | "right" | "up" | "down";

컴포넌트 Props 타입 정의

컴포넌트가 받아야 할 props의 타입을 정의한 인터페이스

```
interface TailButtonProps {
  caption : string,
  color: string,
  onClick : () => void
}
```

```
export default function TailButton({caption, color, onClick}:TailButtonProps)
```

컴포넌트 Props 타입 정의

```
interface TailButtonProps {
  caption : string,
  color: string,
  onClick : () => void
}
```

- React의 마우스 이벤트 타입
  onClick, onMouseEnter 같은 이벤트 핸들러의 매개변수 타입을 명확하게 지정

```
import { MouseEvent } from "react";
```

- 세 가지 리터럴 값만 허용되는 유니언 타입

```
type ButtonColor = "blue" | "orange" | "lime" ;

interface TailButtonProps {
  caption : string,
  color: ButtonColor,
  onClick? : (e:MouseEvent<HTMLButtonElement>) => void
```

- ?는 선택적(Optional) 프로퍼티를 의미하며, onClick이 전달되지 않아도 컴파일 오류가 발생하지 않음

- 이 이벤트가 발생하는 DOM 요소의 타입

컴포넌트 Props 타입 정의

```tsx
import { ChangeEvent, RefObject } from "react" ;

interface TailSelectProps {
  id : string,
  refSel : RefObject<HTMLSelectElement | null>,
  ops : string[],
  handleChange? : (e:ChangeEvent<HTMLSelectElement>) => void
}

export default function TailSelect({id, refSel, ops, handleChange}:TailSelectProps)
```

- ref는 DOM을 참조하므로 useRef<HTMLSelectElement>(null) 등으로 전달되는 타입

- onChange 이벤트 타입

atom 타입 확인

```tsx
import { atom } from "jotai";

export const isLogin = atom<boolean>(false) ;
```

atom 정의

```tsx
declare function atom<T>(initialValue: T): Atom<T>
```

제네릭(Generic)

- <T>는 타입 변수. 사용할 때 T에 실제 타입이 대입
- 타입을 변수처럼 사용하는 문법
- 함수, 클래스, 인터페이스 등에서 동적인 타입을 안전하게 사용할 수 있게 해주는 강력한 도구
- 같은 로직이 다양한 타입에서 동작하되, 타입 안정성은 유지하고 싶은 경우 사용

useRef 타입

```
import { useRef } from "react" ;
import { useNavigate } from "react-router-dom";
import { useAtom } from "jotai";
import { isLogin } from "../atoms/IsLoginAtom";

export default function Login() {
  const emailRef = useRef<HTMLInputElement>(null) ;
  const pwdRef = useRef<HTMLInputElement>(null) ;
```

- TypeScript에서 HTML 요소에 안전하게 접근하기 위한 표준적인 방법
- 초기값은 항상 null : DOM은 마운트 이후에 생기므로 초기에는 null로 설정

```
const handleOk = () => {
  if (emailRef.current?.value == '') {
    alert("Email을 입
    emailRef.current.focus() ;
    return ;
  }
  if (pwdRef.current?.value == '') {
    alert("비밀번호를 입력하세요.") ;
    pwdRef.current.focus();
    return ;
  }
```

- 접근 시에는 ?.(optional chaining) 사용

useState 타입

```tsx
interface TDate {
  [key : string] : string ;
}

const [areaIndex, setAreaIndex] = useState<string | undefined>() ;
const [tdata, setTdata] = useState<TDate | undefined>() ;
const [tags, setTags] = useState<React.ReactNode[]>([]) ;
```

- React에서 화면에 렌더링 가능한 모든 타입을 아우르는 타입
- JSX, 문자열, 숫자, null, undefined, fragment 등 모든 렌더링 가능한 React 요소의 배열

JSON 타입

```
[
  {
      "코드": "201193",
      "측정소": "서면역1호선승강장"
  },
    ...
  {
      "코드": "201191",
      "측정소": "서면역1호선대합실"
  }
]
```

```
import sarea from "../db/sarea.json" ;

interface SArea {
  코드 : string;
  측정소 : string;
}

let ops = (sarea as SArea[]).map(item => item["측정소"]) ;
ops = ['--- 측정소
```

타입 단언 (Type Assertion)

- 이미 선언된 sarea 변수의 타입을 "강제로" 특정 타입으로 단언(assert) 할 때 사용
- "표현식"에 사용하는 문법
- sarea가 어떤 타입인지 확신할 때 컴파일러에게 알려주는 용도

JSON 타입

```json
{
  "pm10": {
    "name": "미세먼지",
    "unit": "㎍/㎥",
    "description": "미세먼지 측정값"
  },
    ...
  "fad": {
    "name": "폼알데하이드",
    "unit": "㎍/㎥",
    "description": "폼알데하이드 측정값"
  }
}
```

```tsx
import scode from "../db/scode.json" ;

interface SCode {
  [key : string] : {
    name : string;
    unit : string;
    description : string;
  }
}

const scodeMap = scode as SCode ;
let tm = Object.keys(scodeMap).map(item =>
        <div key={item}>
          <div className="bg-emerald-600 pt-2
          border-r
          text-white font-bold">{scodeMap[item]["name"]}</div>
          <div className="bg-eme
          border-r text-white fo
          <div className="p-2 font-bold border border-emerald-200">{tdata[item]}
            {tdata[item] == '-' ? '' : scodeMap[item]["unit"]}
          </div>
        </div>);
```

- JSON 객체의 타입을 명시
- 객체의 모든 키(문자열 배열)를 반환
- 해당 key의 name 속성 사용 (타입 안전)

함수 인수 타입 설정

```
const getFetchData = async (idx : string): Promise<void>  => { ···
}

useEffect(() => {
  setTags([]) ;
}, []);

useEffect(() => {
  if (!areaIndex) return ;

  console.log("areaIndex", areaIndex)
  getFetchData(areaIndex) ;
}, [areaIndex]) ;
```

- 매개변수 타입 선언
- 리턴 타입 선언
- async 함수는 Promise 반환