리액트(React)

TypeScript Migration

김경민

마이그레이션(Migration)

- 기존 시스템, 코드, 데이터, 환경 등을 새로운 것으로 전환하는 작업 또는 과정
- React 프로젝트 TypeScript 도입
  - 유지보수성 향상
    - 정적 타입 검사: 코드를 실행하기 전(컴파일 단계)에 타입 관련 에러를 미리 잡아낼 수 있음
    - 코드의 명확성: props, state 등의 데이터 구조가 코드로 명시되어 주석 없이도 코드 이해하기 쉬움
  - 생산성 향상
    - 강력한 자동완성: VSCode와 같은 에디터에서 객체의 속성, 함수 등을 정확하게 자동완성
    - 리팩토링 편의성: 변수명이나 객체 구조를 변경할 때, 타입스크립트가 연관된 모든 파일의 오류를 즉시 알려주어 수정이 용이
  - 협업 효율 개선
    - 컴포넌트의 props 타입을 보면, 해당 컴포넌트를 어떻게 사용해야 하는지 명확한 가이드(규약)가 됨

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 ↗
—
```

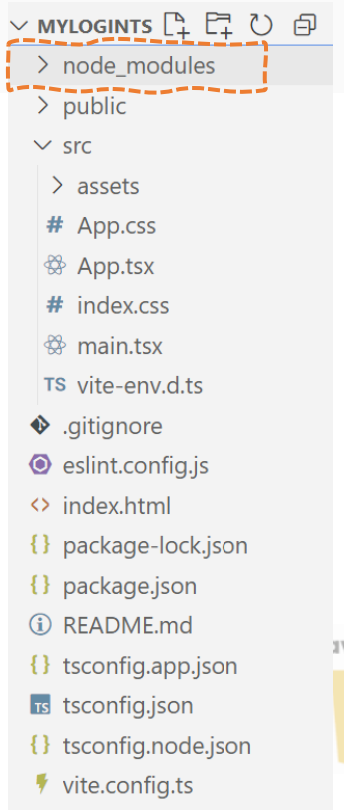
프로젝트에 필요한
라이브러리와 의존성들을 설치

```
mylogints>npm install

ges, and audited 182 packages in 34s

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

found 0 vulnerabilities
```



Tailwindcss 설치

https://tailwindcss.com/docs/installation/using-vite

1. Tailwind css 설치

npm install -D tailwindcss @tailwindcss/vite @tailwindcss/postcss postcss autoprefixer

```
4 moderate severity vulnerabilities

To address all issues possible (including breaking changes), run:
  npm audit fix --force

Some issues need review, and may require choosing
a different dependency.

Run `npm audit` for details.
```

2. postcss.config.cjs생성하여 다음 내용 추가

```
const tailwind = require('@tailwindcss/postcss');
const autoprefixer = require('autoprefixer');

module.exports = {
  plugins: [
    tailwind(),
    autoprefixer(),
  ],
};
```

3. vite.config.js 수정

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

4. index.css 첫줄에 추가

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

타입스크립트 프로젝트 생성

- 리액트 아이콘 설치
  - npm install react-icons
- 환경변수 복사 및 .gitignore에 추가
- 라우터 설치
  - npm install react-router-dom
- jotai 설치
  - npm install jotai
- Supabase 라이브러리 설치
  - npm install @supabase/supabase-js

React + TypeScript 마이그레이션

- tsconfig.json
  - TypeScript 프로젝트에서 타입 검사, 코드 변환 방식, 파일 범위 등을 설정하는 핵심 구성 파일이며, 없으면 TS 프로젝트가 제대로 동작하지 않음

역할	설명
컴파일 옵션 설정	어떤 JS 버전으로 변환할지, 모듈 방식, JSX 방식
타입 검사 규칙	strict 모드, noUnusedLocals 등 타입 검사 세부 설정
포함/제외 파일 지정	어떤 파일을 타입 체크할지
프로젝트 레퍼런스	여러 tsconfig를 연결해 대규모 구조 가능
환경 타입 정의	DOM, Node 등에서 어떤 타입을 사용할지 지정

React + TypeScript 마이그레이션

- 점진적 리팩토링
  - 전체 코드를 한 번에 바꾸기보다는 컴포넌트 단위로 점진적 마이그레이션 수행
- 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

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
  - 대괄호 표기법(type[])과 제네릭 표기법(Array<type>) 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 예제

```tsx
type UserTuple = [string, number] ;
type PersonObj = {
  name : string;
  age : number
};
export default function TsxTest() {
  let name:string = "PNU" ;
  let nums:number[] = [1,2,3] ;
  let user:[string, number] = ['Kim', 26];
  let user2 : UserTuple = ['Lee', 30] ;
  let person : { name : string, age : number} = {
      name : 'KimObj' ,
      age: 26
  }
  let person2 : PersonObj = {
    name : 'LeeObj' ,
    age: 30
}
```

배열과 오브젝트는 type으로 선언하여 사용

```tsx
  return (
    <div>
      <ul>
        <li>이름 : {name}</li>
        {nums.map((item : number, idx:number)=>
                        <li key={`num-${idx}`}>{item}</li>)}
        <li>{user.map((item : string | number, idx:number) =>
                        <span key={`user-${idx}`}>{item} </span>)}</li>
        <li>{user2.map((item : string | number, idx:number) =>
                        <span key={`user2-${idx}`}>{item} </span>)}</li>
        <li>{person.name} {person.age}</li>
        <li>{person2.name} {person2.age}</li>
      </ul>
    </div>
  )
}
```

타입스크립트 예제

- keyof
  - 객체 타입의 키(key)들을 타입으로 추출
  - 객체 타입의 모든 속성 이름을 유니언 타입으로 만들어 주는 연산자
    - 유니언 타입(Union Type) : 여러 타입 중 하나일 수 있음을 명시할 때 사용하는 타입 (|)

```
let p3 : person2 = {name : "pnu3", age : 20} ;
Object.keys(p3).map(item => console.log(item, p3[item]));
```

```
Element implicitly has an 'any' type because
expression of type 'string' can't be used to index
type 'person2'.
No index signature with a parameter of type 'string'
was found on type 'person2'.ts(7053)
(parameter) item: string
```

Object.keys(p3)는 string[]을 반환하기 때문에,이를 (keyof person2)[]로 명시적 타입 단언을 해주어야 TypeScript가 안전하다고 판단
keyof person2는 "name" | "age"타입이므로, 정확히 해당 키만 허용

```
let p3 : person2 = {name : "pnu3", age : 20} ;
(Object.keys(p3) as (keyof person2)[]).map(item => console.log(item, p3[item]))
```

타입스크립트 선언

- 함수 타입 선언
  - 직접 지정

```
function add(x:number, y:number): number {
  return x + y ;
}
console.log(add(10, 20))
```

  - 화살표 함수

```
const add2 = (x:number, y:number): number => {
  return x + y ;
}
console.log(add2(10, 20))
```

  - 함수 타입 별도로 정의

```
type Add = (x:number , y:number) => number ;
const add3 : Add = (x, y) => {
  return x + y
}
console.log(add3(10, 20))
```

타입스크립트 예제

```
//매개 변수와 리턴값이 없는 경우
const handleHello = () : void => {
  console.log('안녕하세요') ;
}
handleHello();

//매개 변수는 있고 리턴값이 없는 경우
const handleHello2 = (name : string) : void => {
  console.log(`${name} 안녕하세요`) ;
}
handleHello2('PNU');

//매개 변수는 있고 리턴값이 있는 경우
const handleHello3 = (name : string) : string => {
  return `${name} 안녕하세요` ;
}
console.log(handleHello3('PNU'));
```

- **매개변수와 리턴 값이 없는 경우**
  리턴 값이 없는 경우는 void를 사용하지 않아도 되지만 의도를 명확히 할 경우는 void를 사용하는 것이 좋음

```
//매개 변수가 오브젝트인 경우
interface User {
  name : string,
  age : number
}
const handleHello4 = (u : User) : string => {
  return `${u.name} 안녕하세요. (${u.age})` ;
}

let user4 : User = {name : 'pnu4', age : 20} ;
console.log(handleHello4(user4))
```

- **매개변수가 오브젝트인 경우** type이나 interface로 정의된 타입을 사용

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";

타입스크립트 선언

- 제너릭 타입
  - 함수나 클래스, 인터페이스, 타입을 정의할 때 타입을 고정하지 않고, 나중에 사용하는 시점에 타입을 지정할 수 있도록 하는 문법
    - 타입을 변수처럼 사용하는 문법
    - 타입을 변수처럼 받아 동적으로 적용하되 타입 안정성을 확보

```
interface Gen<T> {
  value : T
}

const stringGen : Gen<string> = {value: 'PNU'} ;
const numberGen : Gen<number> = {value: 30 } ;

console.log(stringGen.value) ;
console.log(numberGen.value) ;
```

- value 속성의 타입을 나중에 지정