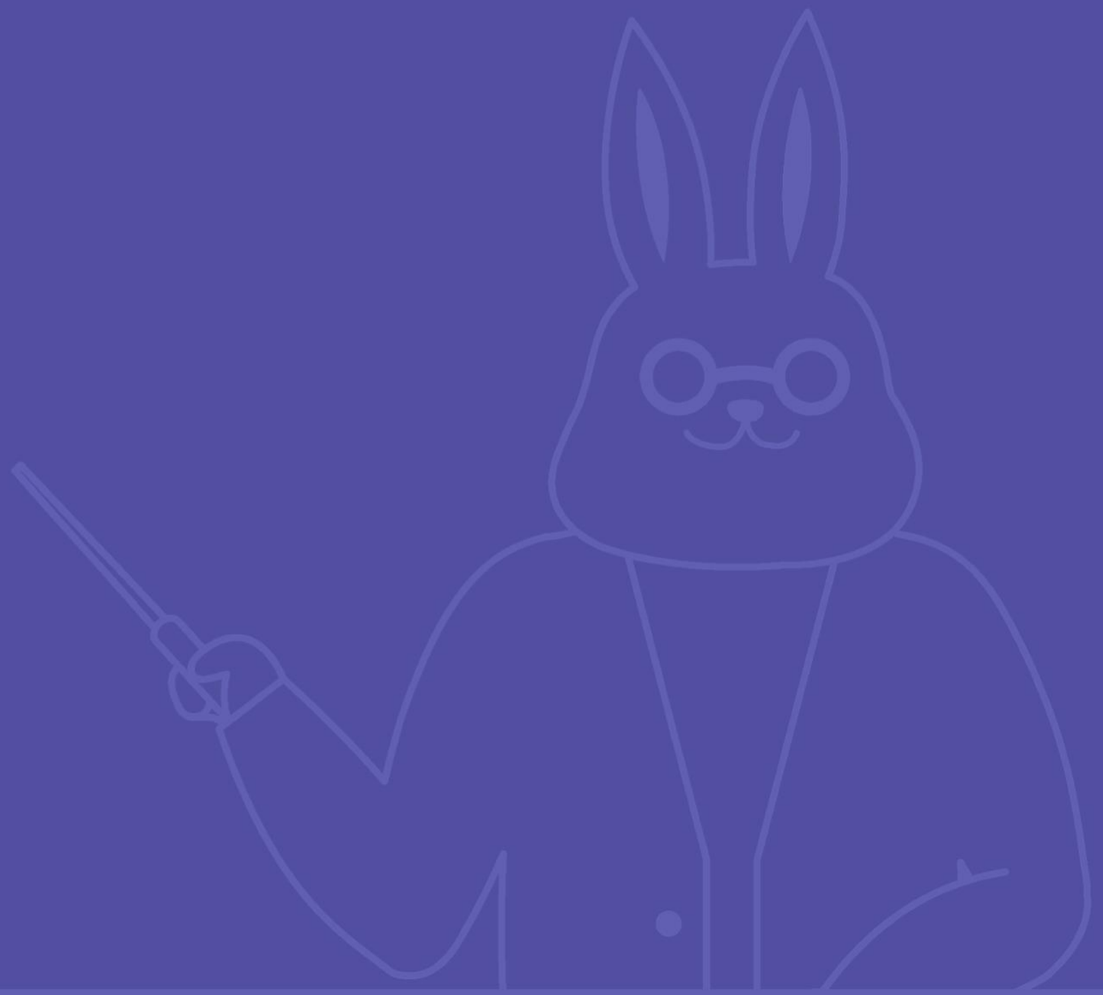
타입스크립트 II

04 React에서의 타입스크립트



목차

🎯 수강목표

1. 타입스크립트 시작하기

마이크로소프트의 타입스크립트 마이그레이션 가이드를 통해 기존 자바스크립트 프로젝트를 타입스크립트로 전환할 수 있게 되고, React 프레임워크를 통해 아주 간단하게 타입스크립트 + React 프로젝트를 시작할 수 있습니다.

2. 예제를 통한 props, hooks, 상태 관리에 타입 적용

예제로 타입스크립트 능력 고사 사이트를 개발해보면서 props, state, hooks, context API, 상태 관리 등 React의 전반적인 부분에 대해 타입을 적용하는 방법을 배웁니다.

3. React가 쉬워지는 타입스크립트 팁

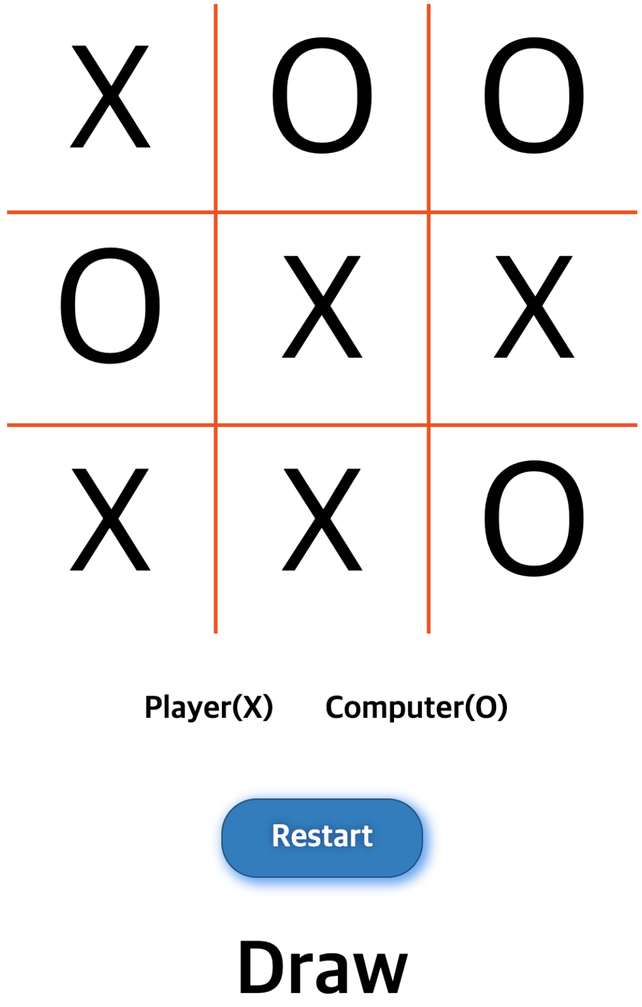
지금까지 배운 타입스크립트 심화 기능을 응용하여 다형성을 지닌 컴포넌트, 템플릿 리터럴 문법을 이용한 타입 생성 등 React를 더 프로답게 이용할 수 있는 방법을 배웁니다.

01

React 앱 타입스크립트로 마이그레이션하기

틱택토 React 예제를 타입스크립트로 변환하기

마이크로소프트의 틱택토 자바스크립트 리액트 예제를 타입스크립트로 변환



마이크로소프트 틱택토 자바스크립트 리액트 예제 : https://github.com/microsoft/TypeScript-React-Conversion-Guide/tree/master/TicTacToe_JS

틱택토 React 예제 프로젝트 구조

구조

```
TicTacToe_JS /
  |---- public/
  |---- src/
    |---- app.jsx
    |---- board.jsx
    |---- constants.js
    |---- gameStateBar.jsx
    |---- restartBtn.jsx
  |---- package.json
  |---- webpack.config.js
```

- public : html, css static files 폴더
- src: 로직이 있는 jsx 파일 폴더
- webpack.config.js: css-loader, babel-loader 등 컴파일, 번들링에 필요한 설정

타입스크립트 마이그레이션

1. 타입스크립트 마이그레이션을 위한 라이브러리 설치
2. tsconfig.json 설정
3. 사용하는 라이브러리들 중 @types 패키지 추가
4. webpack 설정 변경
5. .jsx -> .tsx로 확장자 변환

타입스크립트 마이그레이션을 위한 라이브러리 설치

```
npm i -D typescript esbuild-loader @types/react @types/react-dom
```

```
yarn add -D typescript esbuild-loader @types/react @types/react-dom
```

- typescript: tsc 컴파일러, typescript 문법 지원을 위해 필요한 라이브러리
- @types/react: react 라이브러리를 위한 타입 패키지
- @types/react-dom: react에서 dom element와 관련된 타입들을 모아놓은 패키지
- esbuild-loader: 타입스크립트 트랜스파일링을 위한 패키지(속도가 매우 빠릅니다!)

tsconfig.json 설정

tsconfig.json

```
{
   "compilerOptions": {
      "outDir": "./dist/",
      "sourceMap": true,
      "strictNullChecks": true,
      "module": "es6",
      "jsx": "react",
      "target": "es5",
      "allowJs": true,
   "allowSyntheticDefaultImports": true
   },
   "include": [
      "./src/"
   ]
}
```

tsconfig.json 파일 생성

`tsc --init` 또는 `npx tsc --init`

tsconfig.json 파일 설정

- target: 익스플로러를 지원할 경우 es5 타깃
- sourceMap: 디버깅을 위한 소스 맵 추가
- outDir: 컴파일된 결과가 위치할 경로
- jsx: jsx 파일을 js 파일로 변환하도록 하는 설정
- module: module 코드를 esm 방식으로 변환

사용하는 라이브러리들 중 @types 패키지 추가

DefinitelyTyped 오픈소스에 등록된 타입 선언 파일 설치

```
npm i -D @types/<library name>
```

git repo에 index.d.ts가 있는 라이브러리면 설치 안 해도 됨
npm에서 @types/패키지명 검색해보고 있으면 @types/패키지 설치
@types/패키지가 없다면 직접 모듈에 대한 타입 선언

DefinitelyTyped 오픈소스 주소: https://github.com/DefinitelyTyped/DefinitelyTyped

webpack 설정 변경

webpack.config.js

```
module.exports = {
  entry: "./src/app.tsx",
  output: {
    filename: "./bundle.js",
  },
  resolve: {
    extensions: [".js", ".jsx", ".ts", ".tsx"],
  },
  module: {
    rules: [
      {
        test: /\.(t|j)sx?$/,
        loader: "esbuild-loader",
        options: {
          loader: "tsx", // Or 'ts' if you don't need tsx
          target: "es2015",
        },
      },
      {
        test: /\.css$/,
        use: ["style-loader", "css-loader"],
      },
    ],
  },
  externals: {
    react: "React",
    "react-dom": "ReactDOM",
  },
};
```

- entry: 앱을 시작할 파일
- output: 웹팩 번들링 결과에 대한 옵션. 기본 경로는 dist
- resolve: 번들링할 확장자 설정
- module: 번들링할 때 플러그인 설정 가능
- esbuild-loader: 타입스크립트 변환을 위한 로더
- css-loader: .css 확장자의 css파일을 읽기 위한 로더
- style-loader: style태그를 삽입해 dom에 css 추가
- externals: 번들링 결과에서 제외할 라이브러리들

jsx -> tsx 확장자 변환

jsx 파일에서 tsx 파일로 확장자 변경

이 과정에서 생기는 타입 오류들을 해결해줘야 함

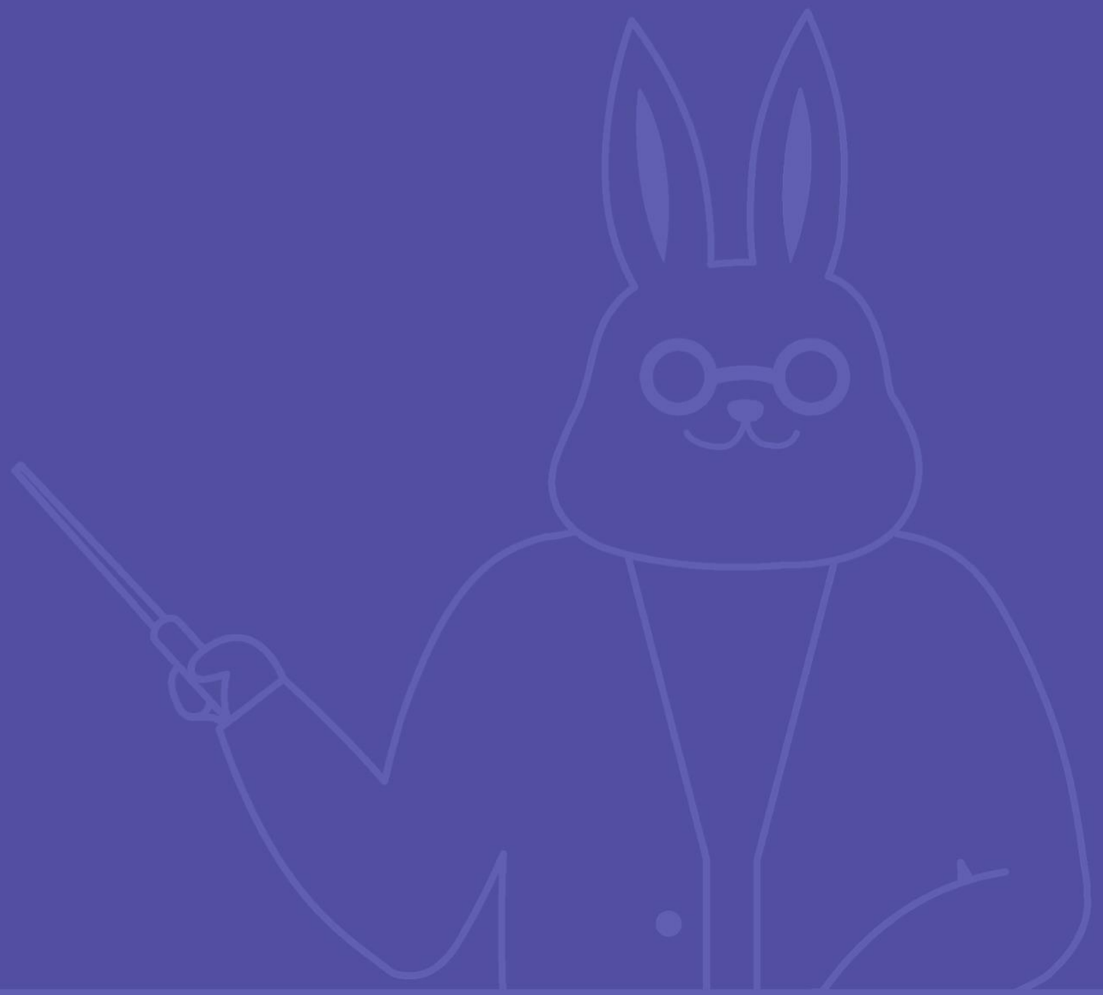
타입스크립트 프로젝트 시작하기

```
npx create-react-app <프로젝트 이름> -template typescript
```

react 프레임워크인 create-react-app로
typescript가 적용된 react 프로젝트를 바로 시작 가능

02

Props 타입 적용



함수형 컴포넌트에 props 타입 설정하기

component props에 타입 설정

```
export const Button = (props: React.PropsWithChildren<ButtonProps>) => {
  return <button {...props} }>{props.children}</button>;
};
```

- {…props}: 스프레드 연산자로 props를 button의 props에 모두 전달
- React.PropsWithChildren: 제네릭에 전달한 props와 props.children을 인터섹션
- props.children: react에서 기본적으로 전달해주는 children props. 자식 노드들이 전달됨

함수형 컴포넌트에 props 타입 설정하기

React.FC 사용해 더 간단히 표현

```
export const Button: React.FC<ButtonProps> = (props) => {
  return <button style={props.buttonStyles}>{props.children}</button>;
};

type FC<P = {}> = FunctionComponent<P>;

interface FunctionComponent<P = {}> {
  (props: PropsWithChildren<P>, context?: any): ReactElement<any, any> | null;
}
```

React.FC 타입을 사용하면 내부적으로 PropsWithChildren을 사용하여 제네릭의 Props 타입과children 타입을 인터섹션. props에 React PropsWithChildren을 선언하는 것과 같은 효과

style props에 타입 적용하기

App.css

```
button {
  padding: 20px;
  border-radius: 5px;
  background-color: #61dafb;
  color: #fff;
  font-size: 25px;
  font-weight: 700;
  border: none;
  cursor: pointer;
}
```

- 모든 button에 적용됨
- class로 만들어도 background-color, color, font-size, font-weight 등 자주 변경되는 것들을 위해 여러 개의 class를 조합해야 함
- ex) className="button bg-black size-16 weight-700"
- props로 전달하면 어떨까?

style props에 타입 적용하기

Button.tsx

```tsx
const createButtonSylte = (
  styles?: React.CSSProperties
): React.CSSProperties => ({
  padding: 20,
  borderRadius: 5,
  border: "none",
  cursor: "pointer",
  fontSize: 25,
  fontWeight: 700,
  backgroundColor: "#61dafb",
  color: "#fff",
  ...styles,
});

interface ButtonProps {
  styles?: React.CSSProperties;
}

export const Button: React.FC<ButtonProps> = (props) => {
  const buttonStyles = createButtonSylte(props.styles);
  return <button style={buttonStyles}>{props.children}</button>;
};
```

- React.CSSProperties: button 태그의 style props 타입
- createButtonStyle: 반환 타입을 React.CSSProperties로 하는 style 객체 팩토리 함수 생성
- '...styles'로 다른 style 속성도 받을 수 있게 추가

event props에 타입 적용하기

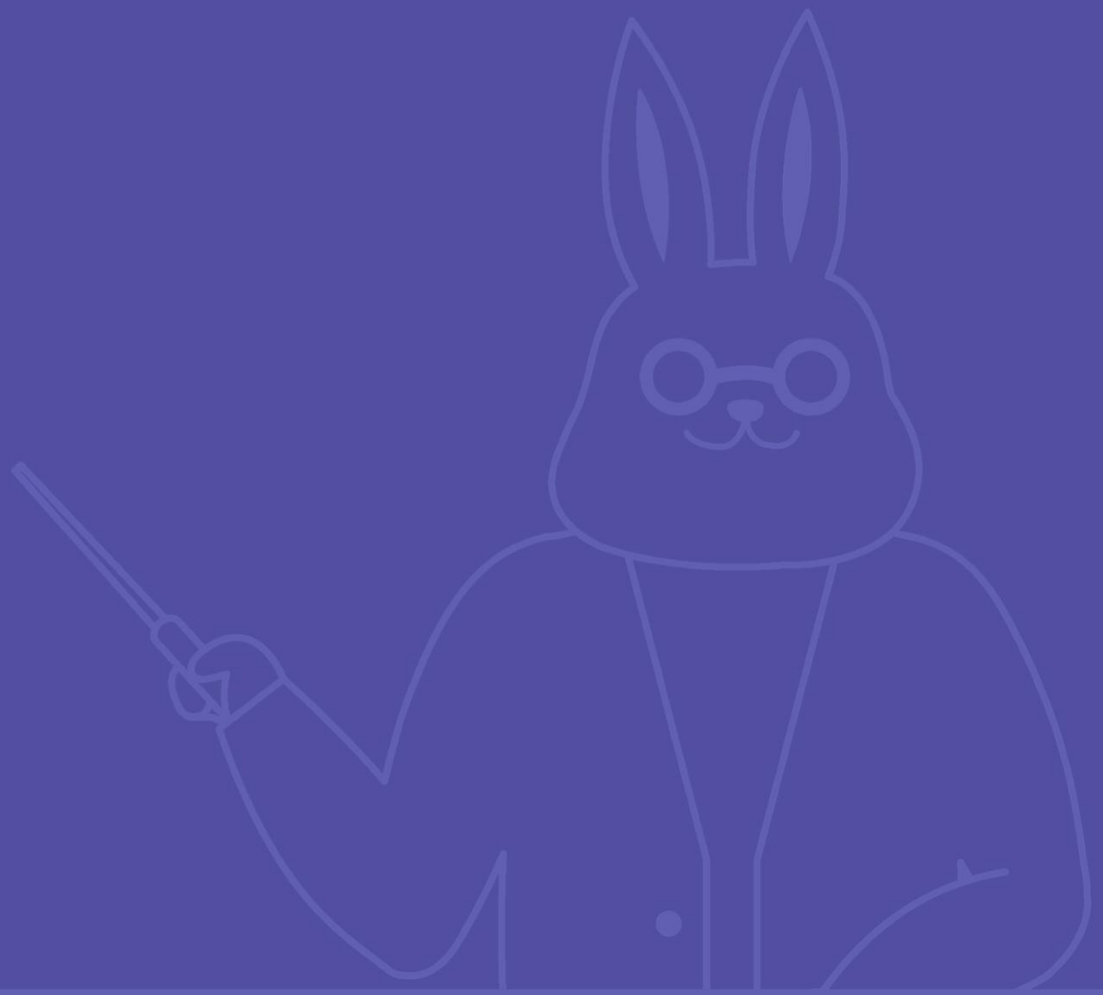
Button - onClick props

```
interface ButtonProps {
  styles?: React.CSSProperties;
  handleClick: (e: React.MouseEvent<HTMLButtonElement,
MouseEvent>) => void;
}

export const Button: React.FC<ButtonProps> = (props)
=> {
  const buttonStyles = createButtonSylte(props.styles);
  return (
    <button style={buttonStyles}
onClick={props.handleClick}>
      {props.children}
    </button>
  );
};
```

- handleClick: button의 onClick props에 전달할 props
- event가 인자로 들어옴
- React.MouseEvent<Element, Event>:
  <button onClick={e => console.log(e)} />
  상태에서 e에 마우스 호버
- e(event)의 타입을 복사해서 ButtonProps 인터페이스의 handleClick event 타입으로 붙여넣기
- 혹은 BaseSyntheticEvent 사용

03

Hook에 타입 적용



useState에 타입 적용하기

useState 타입

```
function useState<S>(initialState: S | (() => S)): [S, Dispatch<SetStateAction<S>>];
function useState<S = undefined>(): [S | undefined, Dispatch<SetStateAction<S | undefined>>];
```

```
const [name, setName] = useState(null)
```

초기값으로 state 타입을 결정

```
const [name, setName] = useState()
```

초기값이 없다면 undefined로 설정

useState에 타입 적용하기

코드

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

export const TestScreen = () => {
  const [name, setName] = useState(null);

  const handleChange = (e:
React.ChangeEvent<HTMLInputElement>) => {
    setName(e.target.value);
            ^^^^^^^^^^^^^^^ 타입 에러
  };
  return (
    <div>
      <input value="입력" onChange={(e) =>
handleChange(e)} />
      <h1>Hello {name} </h1>
    </div>
  );
};
```

- 초깃값 설정 시 초깃값의 타입을 추론해서 state와 setState의 타입을 결정
- 초깃값과 다른 타입의 데이터를 setState의 인자로 넘길 경우 에러
- 이런 경우 useState의 제네릭 타입 설정

```
useState<string | null>(null)
```

useReducer에 타입 적용하기

타입 적용이 안 된 reducer 함수

```
import React from "react";

const reducer = (state, action) => {
  switch (action.type) {
    case "INCREMENT":
      return state + 1;
    case "DECREMENT":
      return state - 1;
    default:
      return state;
  }
};

export const ScoreCounter = () => {
  const [score, dispatch] = React.useReducer(reducer,
{ score: 0 });
  return <div>{score}</div>;
};
```

문제점

- 의도한 타입과 다른 타입의 데이터를 case로 추가해도 에러가 나지 않음
- state가 어떤 타입인지 알 수 없음
- action에 어떤 프로퍼티가 있는지 알 수 없음
- reducer의 제네릭 타입은 Reducer<any, any>를 확장하므로 타입을 지정하지 않으면 state와 dispatch도 any 타입이 됨

useReducer에 타입 적용하기

타입 적용이 된 reducer 함수

```
import React from "react";

type ScoreState = {
  score: number;
};

type ScoreAction = {
  type: string;
  score: number;
};

const reducer = (state: ScoreState, action: ScoreAction):
ScoreState => {
  switch (action.type) {
    case "INCREMENT":
      return { score: state.score + action.score };
    case "DECREMENT":
      const result = state.score - action.score;
      return { score: result < 0 ? 0 : result };
    default:
      return state;
  }
};

export const ScoreCounter = () => {
  const [score, dispatch] = React.useReducer(reducer, { score:
0 });
  return <div>{score}</div>;
};
```

1. state와 action type 선언
2. Action type은 action을 구분할 type 외에는 자유롭게 구성
3. score, dispatch가 각각 ScoreState, React.Dispatch<ScoreAction>로 type 결정

문제점

action.type에서 increment, decrement 외에 다른 action.type이 오지 못하게 해야 함

useReducer에 strict type 적용하기

타입 적용이 된 reducer 함수

```
import React from "react";

type ScoreState = {
  score: number;
};

type ScoreAction = {
  type: string;
  score: number;
};

const reducer = (state: ScoreState, action: ScoreAction):
ScoreState => {
  switch (action.type) {
    case "INCREMENT":
      return { score: state.score + action.score };
    case "DECREMENT":
      const result = state.score - action.score;
      return { score: result < 0 ? 0 : result };
    default:
      return state;
  }
};

export const ScoreCounter = () => {
  const [score, dispatch] = React.useReducer(reducer, { score:
0 });
  return <div>{score}</div>;
};
```

1. state와 action type 선언
2. Action type은 action을 구분할 type 외에는 자유롭게 구성
3. score, dispatch가 각각 ScoreState, React.Dispatch<ScoreAction>로 type 결정

문제점

action.type에서 increment, decrement 외에도 string 타입이라면 추가 가능

useReducer에 strict type 적용하기

타입 적용이 된 reducer 함수

```
type ScoreAction = {
  type: "INCREMENT" | "DECREMENT" | "RESET";
  score: number;
};

const reducer = (state: ScoreState, action:
ScoreAction): ScoreState => {
  switch (action.type) {
    case "INCREMENT":
      return { score: state.score + action.score };
    case "DECREMENT":
      const result = state.score - action.score;
      return { score: result < 0 ? 0 : result };
    case "RESET":
      return { score: 0 };
    default:
      return state;
  }
};
```

해결 방법

- ScoreAction의 type을 string union type으로 선언
- reducer에서는 literal type guard로 타입마다 다른 로직 실행

useReducer에 strict type 적용하기

타입 적용이 된 reducer 함수

```
export const ScoreCounter = () => {
  const [score, dispatch] = React.useReducer(reducer,
{ score: 0 });
  return (
    <div>
      <h3>Score: {score}</h3>
      <Button handleClick={() => dispatch({ type:
"INCREMENT", score: 10 })}>
        정답
      </Button>
      <Button handleClick={() => dispatch({ type:
"DECREMENT", score: 10 })}>
        오답
      </Button>
      <Button handleClick={() => dispatch({ type:
"RESET" })}>초기화</Button>
    </div>
  );
};
```

문제점

- RESET 액션의 경우 score를 따로 받지 않아도 됨
- 그러나 Action 타입에 의해 score를 넣어줘야 함

useReducer에 strict type 적용하기

타입 적용이 된 reducer 함수

```
type CounterAction = {
  type: "INCREMENT" | "DECREMENT";
  score: number;
};

type ResetAction = {
  type: "RESET";
};

type ScoreAction = CounterAction | ResetAction;

const reducer = (state: ScoreState, action: ScoreAction):
ScoreState => {
  switch (action.type) {
    case "INCREMENT":
      return { score: state.score + action.score };
    case "DECREMENT":
      const result = state.score - action.score;
      return { score: result < 0 ? 0 : result };
    case "RESET":
      return { score: 0 };
    default:
      return state;
  }
};
```

해결 방법

1. ScoreAction의 score를 optional로 바꾸면 아까의 문제는 해결된다
2. 하지만 reducer의 case 문에서 action.score를 사용하는 쪽에서는 action.score가 undefined가 될 수 있다는 경고를 내뿜는다
3. 따라서 구별된 유니온을 응용하여 type을 단서로 score 필드가 들어갈지 안 들어갈지 switch 문에서 type guard 하도록 만든다

04

Context API에 타입 적용하기

Context에 타입 적용 전

ScoreContext.ts

```
import React from "react";

export const ScoreContext =
React.createContext({
  score: 0,
  dispatch: () => {},
  ^^^^^^^^^^^^^^^^^^^ 에러 유발
});
```

App.tsx

```
import "./App.css";
import React from "react";
import { AppRouter } from "./Router";
import { reducer } from "./reducers/ScoreCounterReducer";
import { ScoreContext } from "./contexts/ScoreContext";

function App() {
  const [counter, dispatch] = React.useReducer(reducer, { score:
0 });

  return (
    <div className="App">
      <header className="App-header">
        <ScoreContext.Provider value={{ score: counter.score,
dispatch }}>
^^^^^^^^ 타입 에러
          <AppRouter />
        </ScoreContext.Provider>
      </header>
    </div>
  );
}

export default App;
```

Context에 타입 적용 전

ScoreContext.ts

```ts
import React, { Dispatch } from "react";
import { ScoreAction } from
"../reducers/ScoreCounterReducer";

interface ScoreContextValue {
  score: number;
  dispatch: Dispatch<ScoreAction>;
}

export const ScoreContext =
React.createContext<ScoreContextValue>({
  score: 0,
  dispatch: () => {},
});
```

- createContext는 초깃값에 대한 타입을 제네릭으로 받음
- Context의 value에 대한 타입을 선언한 뒤 제네릭의 타입 파라미터에 넣어주면 됨

05

타입스크립트와 리액트 상태 관리: Redux

React Redux와 타입스크립트

```
yarn add redux @reduxjs/toolkit react-redux @types/react-redux
```

react, typescript와 함께 쓰기 위한 redux 라이브러리들을 설치
@reduxjs/toolkit은 RootState와 Dispatch 타입을 추출하는 데 사용
react-redux의 경우 타입 선언 파일이 없어 @types 패키지를 따로 설치해야 함

```
npx create-react-app my-app --template redux-typescript
```

또는 CRA의 redux-typescript 템플릿을 사용해도 됨

React Redux와 타입스크립트

Reducres/store.ts

```
import { configureStore, ThunkAction, Action }
from "@reduxjs/toolkit";
import { scoreCounter } from "../reducers/index";

export const store = configureStore({
  reducer: { scoreCounter },
});

export type AppDispatch = typeof store.dispatch;
export type RootState = ReturnType<typeof
store.getState>;
export type AppThunk<ReturnType = void> =
ThunkAction<
  ReturnType,
  RootState,
  unknown,
  Action<string>
>;
```

- configureStore: redux의 createStore를 사용성 높게 한 번 더 추상화한 것
- redux의 combineReducers를 쓰는 것보다 RootState, AppDispatch, AppThunk 등 타입 추론이 더 쉬워짐

React Redux와 타입스크립트

src/index.tsx

```
import { TypedUseSelectorHook, useDispatch,
useSelector } from "react-redux";
import type { RootState, AppDispatch }
from "./store";

export const useAppDispatch = () =>
useDispatch<AppDispatch>();
export const useAppSelector:
TypedUseSelectorHook<RootState> =
useSelector;
```

- selector는 reducer의 state를 추출
- dispatch는 reducer의 dispatch를 추출
- useAppDispatch, useAppSelector: app의 모든 dispatch, selector 값을 사용할 수 있게 하는 hook
- TypedUseSelectorHook을 사용해 특정 State에 대한 타이핑이 된 useSelector를 생성할 수 있음

React Redux와 타입스크립트

src/index.tsx

```
// redux store 적용을 위한 provider 사용
import { Provider } from "react-redux";
import { store } from "./reducers/store";

ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById("root")
);
```

App의 하위컴포넌트에 store를 적용하기 위해 Provider로 감싼다

React Redux와 타입스크립트

hooks 사용

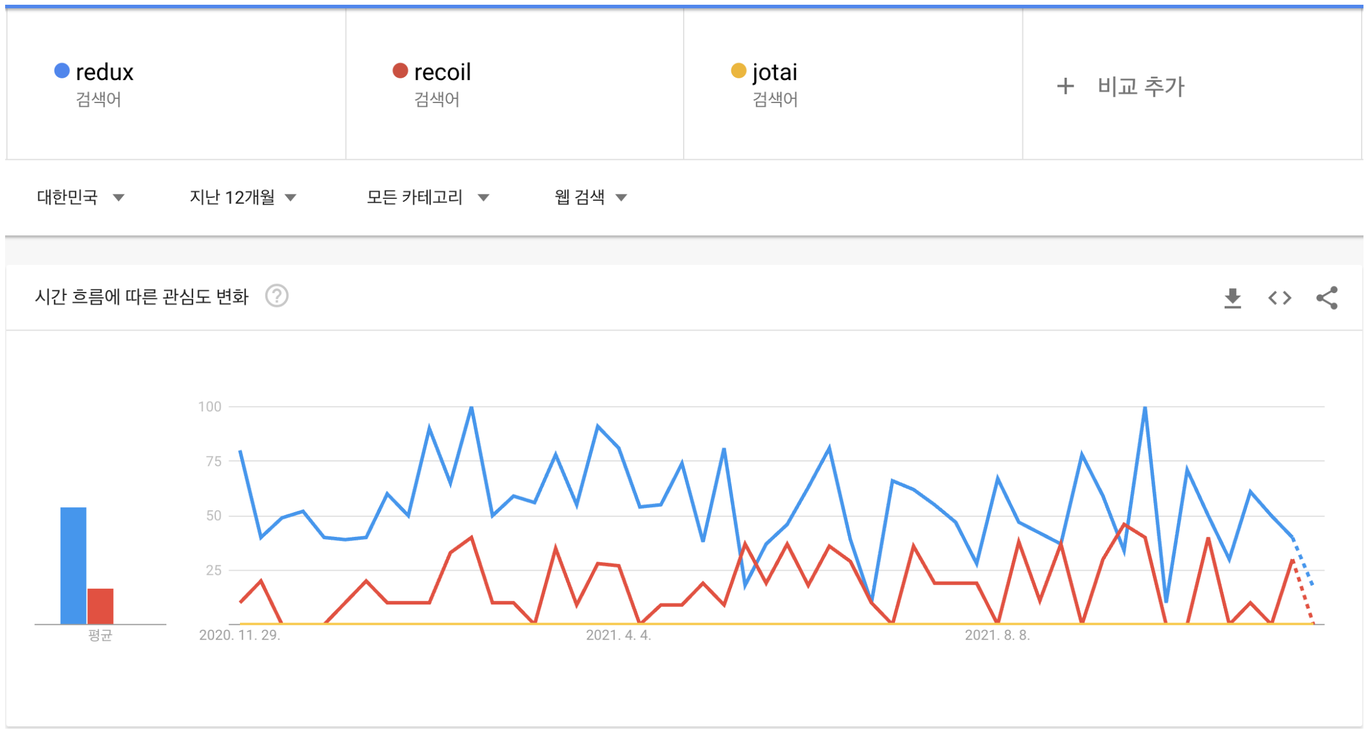
```
import { useNavigate } from "react-router-dom";
import { Button } from "../../components/Button";
import { useAppSelector, useAppDispatch } from
"../../reducers";

export const ResultScreen = () => {
  const score = useAppSelector((state) =>
state.scoreCounter.score);
  const dispatch = useAppDispatch();

  const navigate = useNavigate();
  const handleReset = () => {
    dispatch({ type: "scoreCounter/RESET" });
    navigate("/");
  };
  return (
    <div>
      <h1>
        당신의 점수는 {score}점 입니다 {score >= 80 ? "🎉" :
"ㅋ"}
      </h1>
      <Button handleClick={handleReset}>처음으로</Button>
    </div>
  );
};
```

- useAppSelector: TypedUseSelectorHook 덕분에 RootState에서 자동 완성을 통해 특정 reducer에 대한 state 추출 가능
- useAppDispatch: 전역에서 관리되는 dispatch 사용.
- dispatch({ type: “scoreCounter/RESET” }): **다른 reducer와 함께 쓰이기 때문에 reducer의 이름을 앞에 붙여서 구분해주는 것이 관습**

06

타입스크립트와 리액트 상태 관리: Jotai

상태 관리 하나만 더 알아보자! 뭘 사용할까?



redux: 자바스크립트 앱에서 사용 가능하고 리액트 생태계에서 가장 많이 쓰이는 상태 관리 라이브러리
recoil: 리액트 팀에서 만든 상태 관리 라이브러리
Jotai: recoil의 atomic model 기반의 상향식 접근에 영감을 받아 만든 상태 관리 라이브러리

jotai을 사용해봅시다

recoil과 jotai는 매우 유사하며 같은 문제를 해결한다. 하지만 내 경험상 jotai를 선호한다.
(Recoil and jotai are very similar (and solve the same types of problems). But based on my (limited) experience with them, I prefer jotai.)

https://kentcdodds.com/blog/application-state-management-with-react Kent.d.dodds 블로그 글 인용

useState나 useReducer같은 기존 리액트의 사용성을 그대로 가져오면서
core api는 minimalistic 하게 디자인

http://blog.hwahae.co.kr/all/tech/tech-tech/6099/ 화해 블로그 글 인용

Recoil은 잘못된 memo()의 사용으로 인해 CPU 점유율 상승과 메모리 누수가 발생

https://www.facebook.com/groups/react.ko/permalink/2835701603356660/ 리액트 코리아 페이스북 그룹 글 인용

https://issueexplorer.com/issue/facebookexperimental/Recoil/1220 개선되었다는 0.4.1에서도 비슷한 이슈 발생

jotai에 사용되는 개념

Atom: 상태를 나타내는 단위

recoil과 달리 atom을 생성할 때 사용되는 문자열 key가 필요 없다

config: atom을 생성할 때 넣어주는 초깃값

Provider: Atom이 쓰이는 범위(scope)를 나눌 때 사용

atom, hook 생성

atom 생성

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

export const scoreAtom = atom(0);
```

- atom을 import 해서 config 값을 0으로 설정
- scoreAtom은 state처럼 사용됨

atom, hook 생성

useAtom을 사용한 custom hook

```
import { useAtom } from "jotai";
import { scoreAtom } from "../atoms/scoreAtom";

export type UpdateAction = {
  type: "INCREMENT" | "DECREMENT";
  score: number;
};
export type ResetAction = {
  type: "RESET";
};
export type ButtonAction = UpdateAction | ResetAction;
export type ScoreActionType = "INCREMENT" | "DECREMENT" | "RESET";

export function useScoreHook() {
  const [score, setScore] = useAtom(scoreAtom);
 const dispatch = (action: ButtonAction): void => {
    switch (action.type) {
      case "INCREMENT":
        return setScore(score + action.score);
      case "DECREMENT":
        const newScore = score - action.score;
        return setScore(newScore < 0 ? 0 : newScore);
      case "RESET":
        return setScore(0);
      default:
        return setScore(score);
    }
  };
  return { score, dispatch };
}
```

- useAtom을 사용해 useState처럼 사용 가능
- setScore를 consumer(state를 가져다 쓰는 쪽)에서 타입에 따라 setScore로 매번 같은
  코드를 반복해야 함
- custom hook을 만들어서 dispatcher처럼 사용
- Custom hook에서 **use<hook이름>**은 hook 네이밍 컨벤션이므로 따라야 함

useScoreHook 사용

screens/ResultScreen

```tsx
import React from "react";
import { useNavigate } from "react-router-dom";
import { Button } from "../../components/Button";
import { useScoreHook } from "../../hooks/useScoreHook";

interface ResultScreenProps {}

export const ResultScreen: React.FC<ResultScreenProps> = () => {
  const navigate = useNavigate();
  const { score, dispatch } = useScoreHook();
  const handleReset = () => {
    dispatch({ type: "RESET" });
    navigate("/");
  };
  return (
    <div>
      <h1>
        당신의 점수는 {score}점 입니다 {score >= 80 ? "😆" : "ㅋ"}
      </h1>
      <Button handleClick={handleReset}>처음으로</Button>
    </div>
  );
};
```

생성한 custom hook을 import 해서 사용

07

React에서의 타입스크립트 팁

1. generic props

Comment

```
interface CommentListProps {
 items: Comment[];
 handleClick: (e:
React.MouseEvent) => void;
}

export const CommentList:
React.FC<CommentListProps> = ({
 items,
 handleClick,
}) => {
 return (
  <ul>
   {items.map((item, index) => (
    <li key={index}
onClick={handleClick}>
     {item}
    </li>
   ))}
  </ul>
 );
};
```

Post

```
interface PostListProps {
 items: Post[];
 handleClick: (e:
React.MouseEvent) => void;
}

export const PostList:
React.FC<PostListProps> =
({ items, handleClick }) => {
 return (
  <ul>
   {items.map((item, index) => (
    <li key={index}
onClick={handleClick}>
     {item}
    </li>
   ))}
  </ul>
 );
};
```

User

```
interface UserListProps {
 items: User[];
 handleClick: (e:
React.MouseEvent) => void;
}

export const UserList:
React.FC<UserListProps> =
({ items, handleClick }) => {
 return (
  <ul>
   {items.map((item, index) => (
    <li key={index}
onClick={handleClick}>
     {item}
    </li>
   ))}
  </ul>
 );
};
```

1. generic props

Generic Component

```
interface ListProps<T> {
  items: T[];
  handleClick: () => void;
}

export const List = <T extends {}>({ items,
handleClick }: ListProps<T>) => {
  return (
    <ul>
      {items.map((item, index) => (
        <li key={index} onClick={handleClick}>
          {item}
        </li>
      ))}
    </ul>
  );
};
```

- props를 제네릭 인터페이스로 선언
- 함수 시그니처에서 괄호 앞에 타입 파라미터 T 선언
- interface가 객체의 형태기 때문에 {} 객체 리터럴을 extends하는 제약 조건 추가

1. generic props

Generic Component 사용

```
import { List } from "./List";

export const MainScreen = () => {
  return (
    <div>
      <List
        items={[{ id: "1", name: "elice" }]}
        handleClick={() =>
console.log("hello")}
      ></List>
    </div>
  );
};
```

사용 시에 props만 전달하면 타입이 알아서 추론됨

2. template literal type

2020년 11월에 릴리즈된 typescript 4.1 버전에서 추가된 기능

기존 TypeScript의 String Literal Type을 기반으로 새로운 타입을 만들 수 있는 기능.

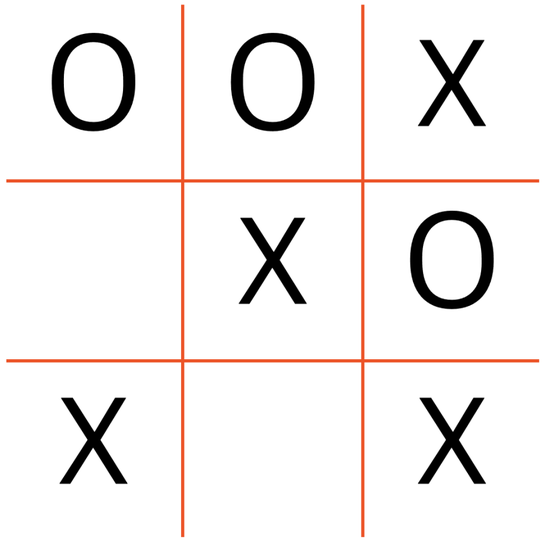
String literal type에 템플릿 문법을 사용 가능

2. template literal type

string literal type

```
type TicTacToeClassName =
  | "left-top"
  | "center-top"
  | "right-top"
  | "left-center"
  | "center-center"
  | "right-center"
  | "left-bottom"
  | "center-bottom"
  | "right-bottom";
```

틱택토의 className을 string literal type으로 만들 경우 일일이 포지션을 써줘야 함



2. template literal type

Template literal type

```
type XPosition = "left" | "center" | "right";
type YPosition = "top" | "center" | "bottom";

type TicTacToeClassName = `${XPosition}-${YPosition}`;
```

예전에는 9가지 string literal type 직접 만들었다면
String 타입으로 className에 대한 string type을
Template literal type으로 간단하게 나타낼 수 있음

3. 다른 컴포넌트의 props 추출

코드

```
import React from "react";
import { Button } from "./Button";

export const CustomButton:
React.FC<React.ComponentProps<typeof
Button>> = (
  props
) => {
  return (
    <button style={props.styles}
onClick={props.handleClick}>
      {props.children}
    </button>
  );
};
```

기존 컴포넌트를 확장할 때 컴포넌트의 props를 그대로 사용한다면 똑같은 props 인터페이스를 만들지 않고 **React.ComponentProps<typeof 컴포넌트>**를 사용

크레딧

/* elice */

코스 매니저
이재성

콘텐츠 제작자
송현지

강사
송현지

감수자
이재성

디자이너
강혜정

연락처

TEL
070-4633-2015

WEB
https://elice.io

E-MAIL
contact@elice.io