Lekcja 33: Zmapowane modyfikatory typu

Nasza aplikacja wideo umożliwia zalogowanym użytkownikom. Po zalogowaniu się użytkownik może określić preferencje dotyczące sposobu korzystania z jego treści wideo. Preferencje użytkownika modelowania prostego typu mogą wyglądać następująco:

type UserPreferences = {

format: keyof VideoFormatURLs

subtitles: {

active: boolean,

language: keyof SubtitleURLs

},

theme: 'dark' | 'light'

}

Odniesienia do VideoFormatURL i SubtitleURL sprawiają, że nie musimy utrzymywać większej liczby typów niż to konieczne. Aktualizacja jednego z tych typów dodaje kolejną część do unii kluczy w format i subtitles.language. Ponadto, zamiast pozwalać, aby każdy ciąg był prawidłowym motywem, ograniczamy tę właściwość, aby była dark lub light.

Częściowe

Jak można wyczytać z typu UserPreferences, żadna właściwość nie jest opcjonalna. Wszystkie właściwości są wymagane, aby zapewnić użytkownikom dobre wrażenia, więc nie chcemy niczego pomijać. Aby upewnić się, że wszystkie klucze są ustawione, zapewniamy zestaw domyślnych preferencji użytkownika:

const defaultUP: UserPreferences = {

format: 'format1080p',

subtitles: {

active: false,

language: 'english'

},

theme: 'light'

}

Używamy tutaj adnotacji typu. Zwykle staramy się wywnioskować jak najwięcej, ale wartości domyślne należą do kategorii obiektów obsługiwanych. defaultUP może się zmienić, a w momencie, gdy go zmienimy, chcemy zweryfikować go z UserPreferences. Dla naszych użytkowników przechowujemy tylko delty. Jeśli użytkownik zmieni preferowany format wideo na inny, będzie to tylko nowy, który przechowujemy.

const userPreferences = {

format: 'format720p'

}

Aby uzyskać pełny zestaw preferencji, łączymy nasze domyślne preferencje z preferencjami użytkownika w funkcji:

function combinePreferences(defaultP, userP) {

return { ...defaultP, ...userP }

}

Używając składni rozszerzania obiektów, tworzymy obiekt, który jest kopią defaultP i nadpisujemy lub rozszerzamy o wszystkie właściwości z userP. Wynikowy obiekt to pełne preferencje użytkownika z zastosowaną różnicą. Teraz dodajmy typy do tej funkcji. defaultP jest łatwy do wpisania:

function combinePreferences(

defaultP: UserPreferences,

userP: unknown

) {

return { ...defaultP, ...userP }

}

Ale jak wpisujemy userP? Potrzebowalibyśmy typu, w którym każdy klucz może być opcjonalny, coś takiego:

type OptionalUserPreferences = {

format?: keyof VideoFormatURLs

subtitles?: {

active?: boolean,

language?: keyof SubtitleURLs

},

theme?: 'dark' | 'light'

}

Ale oczywiście nie chcemy sami utrzymywać tego typu. Stwórzmy pomocniczy typ Optioal, który robi to za nas. Jest to typ mapowany, w którym modyfikujemy właściwości właściwości, aby każdy klucz stał się opcjonalny:

type Optional<Obj> = {

[Key in keyof Obj]?: Obj[Key]

}

Zwróć uwagę na mały znak zapytania obok zmapowanego argumentu, w którym iterujemy przez wszystkie klucze. Nazywa się to modyfikatorem właściwości. Dzięki temu tworzymy kopię parametru

typu Obj, w którym wszystkie klucze są opcjonalne. Zanotujmy naszą funkcję connectPreferences za pomocą tego typu pomocnika.

```
function combinePreferences(

defaultP: UserPreferences,

userP: Optional<UserPreferences>

) {

return { ...defaultP, ...userP }

}
```

Teraz otrzymujemy dodatkowe autouzupełnianie i bezpieczeństwo typów podczas korzystania z connectPreferences.

```
// OK!

const prefs = combinePreferences(

defaultUP,

{ format: 'format720p' }

)

// boom!

const prefs = combinePreferences(

defaultUP,

{ format: 'format720p' }

)
```

Optional<Obj> to typ wbudowany w TypeScript o nazwie Partial<Obj>. Ma również odwróconą operację Required<Obj>, która powoduje, że wszystkie klucze są wymagane przez usunięcie opcjonalnego modyfikatora właściwości. Jest definiowany jako:

```
type Required<Obj> = {

[Key in Obj]-?: Obj[Key]

}
```

Tylko do odczytu

Jedną rzeczą, którą chcemy się upewnić, jest to, że defaultUP nie może zostać zmieniony z innych części naszego oprogramowania. Powinien być utrzymany w kodzie, a nie przez efekt uboczny. Z punktu widzenia narzędzi potrzebujemy typu, który zapewni, że każda właściwość będzie tylko do odczytu.

```
type Const<Obj> = {

readonly [Key in Obj]: Obj[Key]

}
```

Widzisz, że dodajemy modyfikator właściwości: tylko do odczytu. Dzięki temu defaultUP nie zostanie zaktualizowany bez narzekania TypeScript.

```
const defaultUP: Const<UserPreferences> = {

format: 'format1080p',

subtitles: {

active: false,

language: 'english'

},

theme: 'light'

}

defaultUP.format = 'format720p'
```

Const <Obj> jest dostępny w języku TypeScript jako Readonly <Obj>. W JavaScript nadal moglibyśmy modyfikować ten obiekt. Dlatego używamy Object.freeze, aby upewnić się, że nie możemy niczego zmienić w czasie wykonywania. Typ zwracanej wartości Object.freez jest Readonly<Obj>

```
function genDefaults(obj: UserPreferences) {

return Object.freeze(obj)

}

const defaultUP = genDefaults({

format: 'format1080p',

subtitles: {

active: false,

language: 'english'

},

theme: 'light'

})

// defaultUP is Readonly<UserPreferences>

defaultUP.format = 'format720p'
```

Powoduje to błąd zarówno w języku TypeScript, jak i JavaScript.

Głębokie modyfikacje

W przypadku opcji Readonly i Partial należy pamiętać o jednej rzeczy: naszej zagnieżdżonej strukturze danych. Na przykład to wywołanie spowoduje błędy w TypeScript:

```
const prefs = combinePreferences(
```

defaultUP,

{ subtitles: { language: 'german' } }

)

TypeScript oczekuje, że udostępnimy pełny obiekt dla subtitles, jako Partial właśnie ustawił pierwszy poziom właściwości jako opcjonalny. Przy rodzaju zadania, które wykonujemy, jest to właściwie oczekiwane zachowanie. Powyższe wywołanie spowodowałoby przesłonięcie naszej właściwości subtitles i usunięcie subtitles.active. Musielibyśmy tworzyć bardziej wyrafinowane zadania, a także bardziej wyrafinowane typy. Podobny problem pojawia się, gdy spojrzymy na nasze domyślne preferencje. Tylko do odczytu modyfikuje tylko pierwszy poziom właściwości, co oznacza, że to wywołanie nie powoduje błędu w TypeScript, natomiast przerywa działanie po uruchomieniu w przeglądarce:

defaultUP.subtitles.language = 'german'

Aby upewnić się, że nasze typy są tym, czego od nich oczekujemy, potrzebujemy typów pomocniczych, które sięgają głębiej niż jeden poziom. Na szczęście TypeScript pozwala na typy rekurencyjne. Możemy zdefiniować typ, który odwołuje się do siebie i idzie o jeden poziom głębiej

type DeepReadonly<Obj> = {

readonly [Key in Obj]: DeepReadonly<Obj[Key]>

}

TypeScript wie, aby zatrzymać rekursję, jeśli Obj [Key] zwraca typ pierwotny lub wartościowy albo sumę typów pierwotnych lub wartościowych. Zastosujmy nowy typ pomocnika do naszej funkcji genDefaults jako typ zwracany:

function genDefaults(

obj: UserPreferences

): DeepreadOnly<UserPreferences> {

return Object.freeze(obj)

}

Ponieważ Readonly jest podtypem DeepReadonly, węższy zwracany typ Object.freeze jest zgodny z szerszym zdefiniowanym przez nas typem zwracania. to samo można zrobić dla podobieństw:

type DeepPartial<T> = {

[P in keyof T]?: DeepPartial<T[P]>

}

Ale szczegóły nowej implementacji zależą od Ciebie!