Lekcja 40: Tworzenie typów pomocniczych

Szybkie spojrzenie na poprzednią lekcję. Oto jak daleko zaszliśmy:

declare function createMedium<

Kin extends MediaKinds

>(

kind: Kin, info

): SelectBranch<AllMedia, Kin>

Wybieramy określony rodzaj i wiemy, jakiego typu zwrotu się spodziewać.

createMedium ('lp', {/ * tbd * /}) // Zwraca LP!

createMedium ('cd', {/ * tbd * /}) // Zwraca CD!

Teraz chcemy skupić się na brakujących informacjach. Pamiętaj, że chcemy dodać wszystko, co jest potrzebne do stworzenia pełnego medium, z wyjątkiem id, który jest generowany automatycznie przez createMedium lub kind, z, które już zdefiniowaliśmy.

Exclude

Oznacza to, że musimy przekazywać obiekty, które wyglądają tak:

type CDInfo = {

title: string,

description: string,

tracks: number,

duration: number

}

type LPInfo = {

title: string,

description: string,

sides: {

a: {

tracks: number,

duration: number

},

b: {

tracks: number,

duration: number

}

}

}

Ale nie chcemy, aby te typy były utrzymywane - chcemy, aby były generowane automatycznie. Pierwszą rzeczą, o którą chcemy się zająć, jest wiedza, których kluczy naszego obiektu faktycznie potrzebujemy. Najlepszym sposobem na to jest wiedza, których kluczy nie potrzebujemy: kind i id.

type Removable = „kind” | 'ID'

Dobrze. Teraz musimy odfiltrować wszystkie klucze właściwości, których nie ma w tym zestawie kluczy. W tym celu tworzymy kolejny dystrybucyjny typ warunkowy. Wygląda bardzo podobnie do Extract, ale działa inaczej.

type Remove<A, B> = A extends B ? never : A

Odczytuje, że jeśli typ A jest częścią B, usuń go (never); w przeciwnym razie zachowaj. Zobaczmy, co się stanie, jeśli użyjemy wszystkich kluczy płyty CD i rozprowadzimy sumę nad typem Remove. Pamiętaj, warunek związku jest jak związek warunków.

// First our keys

type CDKeys = keyof CD

// Equal to

type CDKeys = ‘id’ | ‘description’ |

‘title’ | ‘kind’ | ‘tracks’ | ‘duration’

// Now for the keys we actually want

type CDInfoKeys = Remove<CDKeys, Removable>

// Equal to

type CDInfoKeys =

Remove<’id’ | ‘description’ | ‘title’ |

‘kind’ | ‘tracks’ | ‘duration’, ‘id’ | ‘kind’>

// A conditional of a union

// is a union of conditionals

type CDInfoKeys =

Remove<’id’, ‘id’ | ‘kind’> |

Remove<’description’, ‘id’ | ‘kind’> |

Remove<’title’, ‘id’ | ‘kind’> |

Remove<’kind’, ‘id’ | ‘kind’> |

Remove<’tracks’, ‘id’ | ‘kind’> |

```
Remove<’duration’, ‘id’ | ‘kind’>

// Substitute

type CDInfoKeys =

(‘id’ extends ‘id’ | ‘kind’ ?

never : ‘id’) |

(‘description’ extends ‘id’ | ‘kind’

? never : ‘description’) |

(‘title’ extends ‘id’ | ‘kind’

? never : ‘title’) |

(‘kind’ extends ‘id’ | ‘kind’ ? never : ‘kind’) |

(‘tracks’ extends ‘id’ | ‘kind’ ? never : ‘tracks’) |

(‘duration’ extends ‘id’ | ‘kind’ ? never : ‘duration’)

// Evaluate

type CDInfoKeys =

never | ‘description’ | ‘title’ | never |

‘tracks’ | ‘duration’

// Remove impossible types from the union

type CDInfoKeys =

‘description’ | ‘title’ | ‘tracks’ | ‘duration’'
```

Wow, co za proces! Ale zbliżamy się o krok do oczekiwanego wyniku. Typ Remove jest wbudowany w TypeScript i nazywa się Exclude. Definicja jest dokładnie taka sama, a jej opis mówi, że wyklucza typy z A, które są w B. Tak właśnie się stało.

Omit

Musimy teraz wziąć ten nowy zestaw kluczy - podzbiór oryginalnego zestawu kluczy - i utworzyć typ obiektu z nowymi kluczami, które muszą być typu oryginalnego obiektu. To bardzo przypomina typ mapowany, prawda? Pamiętasz Pick? Pick przechodzi przez zestaw kluczy i wybiera typ z oryginalnego typu właściwości. Właśnie tego szukamy.

```
type CDInfo = Pick<

CD,

Exclude<keyof CD, ‘kind’ | ‘id’>

>
```

Jak czytamy ten nowy typ? Wybieramy z CD wszystkie klucze CD, ale nie uwzględniamy kind i id. Rezultatem jest typ, jaki pierwotnie wyobrażaliśmy. Ponownie typy ogólne zachowują się jak funkcje. Mają parametry i dane wyjściowe i można je komponować. Czytanie tego typu może wydawać się

trochę skręcone językiem. Dlatego TypeScript ma wbudowany typ dla dokładnie tej kombinacji Pick i Exlude, zwany Omit.

type CDInfo = Omit <CD, ‘kind’ | „Id”>

Przeszliśmy długą drogę z naszymi typami. Ostatnim krokiem jest skomponowanie wszystkiego w naszej funkcji createMedium. Aby skutecznie pominąć kind i id w naszych typach mediów, musimy przekazać wybraną gałąź do Omit. Inny typ pomocnika sprawia, że jest to nieco bardziej czytelne.

type RemovableKeys = ‘kind’ | ‘id’

type GetInfo<Med> = Omit<Med, RemovableKeys>

declare function createMedium<

Kin extends MediaKinds

>(

kind: Kin,

info: GetInfo<SelectedBranch<AllMedia, Kin>>

): SelectBranch<AllMedia, Kin>

I to wszystko! Teraz TypeScript pyta nas tylko o brakujące właściwości. Nie musimy podawać zbędnych informacji, a podczas korzystania z funkcji createMedium uzyskujemy autouzupełnianie i bezpieczeństwo typów.

Zestaw typów pomocniczych

Możliwość komponowania typów ogólnych i dystrybucyjnych typów warunkowych pozwala na zestaw mniejszych typów pomocniczych jednofunkcyjnych, które można zestawiać dla różnych scenariuszy. To pozwala nam zdefiniować zachowanie typów bez utrzymywania zbyt wielu typów. Skoncentruj się na modelu, opisz zachowanie z pomocnikami.