Lekcja 32: Ogólne typy mapowane

TypeScript ma kilka typów pomocniczych, których można użyć do tego, co zrobiliśmy ręcznie. Mogą się przydać, gdy zaczniemy tworzyć typy zaawansowane. Spójrzmy na Record and Pick. Oba są mapowane na typy generyczne.

Pick

Pick <O, K> tworzy nowy obiekt z wybranymi kluczami właściwości K obiektu O. Jest zdefiniowany jako

```
type Pick<

O,

K extends keyof O

> = {

[P in K]: O[P];

}
```

[P in K] obejmuje wszystkie typy wartości w unii K, która obejmuje wszystkie klucze typu O. O [P] jest indeksowanym typem dostępu. To jak indeksowanie obiektu, ale pobieranie typu. To pozwala nam zdefiniować sumę kluczy, które są częścią oryginalnego typu obiektu i wybrać te klucze i ich typy z oryginalnego obiektu. Na przykład byłby to typ ze wszystkimi filmami HD

```
type HD = Pick<

VideoFormatURLs,

'format1080p' | 'format720p'

>

// Equivalent to

type HD = {

format1080p: URL,

format720p: URL

}
```

Najbardziej oczywistym zastosowaniem typu pomocnika Pick jest funkcja narzędzia pick dostępna w bibliotekach takich jak Lodash. Ale może być również pomocny w innych scenariuszach. Przyjrzymy się niektórym z nich w dalszych rozdziałach.

Rekord

Record <K, T> tworzy typ obiektu, w którym wszystkie typy w T otrzymują typ K. Podobnie jak słownik. Jest zdefiniowany jako

```
type Record<

K extends string | number | symbol,

T
```

```
> = {
[P in K]: T
}
```

Zauważ, że K jest podtypem string | number | symbol. Spotkaliśmy się z tym trio wcześniej, ponieważ są to dozwolone typy kluczy obiektów. URLObject z poprzedniej lekcji zostałby zdefiniowany jako

```
type URLObject = Record<string, URL>
```

Rekord jest fajnym skrótem, jeśli musimy utworzyć typ obiektu w locie.

Mapowane i indeksowane typy dostępu

Załóżmy, że nasza platforma wideo umożliwia przesyłanie wszystkich czterech rodzajów rozdzielczości wideo, ale nie wymaga ich wszystkich. Wymagamy co najmniej jednego formatu. Modelowanie tej sytuacji można łatwo wykonać za pomocą typów związków:

```
type Format360 = {
format360p: URL
}
type Format480 = {
format480p: URL
}
type Format720 = {
format720p: URL
}
type Format1080 = {
format1080p: URL
}
type AvailableFormats =
Format360 | Format480 | Format720 | Format1080
const hq: AvailableFormats = {
format720p: new URL('...'),
format1080p: new URL('...')
} // OK!
const lofi: AvailableFormats = {
format360p: new URL('...'),
format480p: new URL('...')
```

} // OK!

W przypadku typów związków musimy wypełnić umowę tylko jednego członka związku. To sprawia, że jest to świetne, jeśli potrzebujemy co najmniej jednego losowego zestawu właściwości, a wszystkie inne są opcjonalne. Ale - zgadłeś - wymagałoby to od nas utrzymania drugiego zestawu typów. Nie chcemy na nowo definiować VideoFormatURL, ponieważ ten typ jest niezbędny do niektórych funkcji naszej aplikacji. Chcemy tylko, aby adresy URL VideoFormatURL były podzielone na związki. Zbudujmy pomocnika o nazwie Split. Celem jest utworzenie typu związku. Aby było to łatwiejsze, zaczynamy od konkretnego typu i pracujemy nad zastąpieniem później. Więc co już wiemy? Po pierwsze, wiemy, że keyof VideoFormatURLs tworzy unię wszystkich kluczy z VideoFormatURL.

type Split = keyof VideoFormatURLs

// Equivalent to

type Split =

“format360p” | “format480p” |

“format720p” | “format1080p”

Wiemy również, że zmapowany typ działa na wszystkie klucze i tworzy nowy obiekt z tymi kluczami. Poniższy przykład tworzy ten sam typ co VideoFormatURLs, ale kluczem jest również wartość:

type Split = {

[P in keyof VideoFormatURLs]: P

}

// Equivalent to

type Split = {

format360p: “format360p”,

format480p: "format480p”,

format720p: “format720p”,

format1080p: “format1080p”

}

Teraz możemy ponownie uzyskać dostęp do wartości tego typu za pomocą indeksowanego operatora dostępu. Jeśli uzyskamy dostęp za pomocą unii kluczy VideoFormatURL, otrzymamy sumę wartości.

type Split = {

[P in keyof VideoFormatURLs]: P

}[keyof VideoFormatURLs]

// Equivalent to

type Split =

“format360p” | “format480p” |

“format720p” | “format1080p”

Wygląda to dokładnie tak, jak pierwszy krok, ale jest zasadniczo inny. Zamiast pobierać lewą stronę typu obiektu - klucze właściwości -w unii, otrzymujemy prawą stronę typu obiektu - typy właściwości - w unii. Więc jedyne, co musimy zrobić, to uzyskać właściwe wartości i mamy związek, jaki sobie wyobrażaliśmy. Wprowadź Record. Record <P, VideoFormatURLs [P] daje nam obiekt z właściwością P, którą otrzymujemy z unii kluczy i uzyskujemy dostęp do odpowiedniego typu z klucza właściwości.

type Split = {

[P in keyof VideoFormatURLs]

: Record<P, VideoFormatURLs[P]>

}[keyof VideoFormatURLs]

// Equivalent to

type Split =

Record<”format360p”, URL> |

Record<”format480p”, URL> |

Record<”format720p”, URL> |

Record<”format1080p”, URL>

// Equivalent to

type Split =

{ format360p: URL } |

{ format480p: URL } |

{ format720p: URL } |

{ format1080p: URL }

Na koniec, zbudujmy z tego ogólną wersję.

wpisz Split <Obj> = {

[Prop in keyof Obj]: Record <Prop, Obj [P]>

} [klucz przedmiotu]

type AvailableFormats = Split <VideoFormatURLs>

W momencie, gdy coś zmienimy w VideoFormatURLs, zaktualizujemy również AvailableFormats. A TypeScript wrzeszczy na nas cudownymi czerwonymi zawijasami, jeśli ustawiliśmy właściwość, która już nie istnieje.