Lekcja 26: Klucze obiektów i predykaty typów

Nasza witryna nie tylko zawiera listę wydarzeń różnego rodzaju, ale także umożliwia użytkownikom prowadzenie list wydarzeń, którymi są zainteresowani. Dla użytkowników zdarzenia mogą mieć różne stany:

1. Użytkownicy mogą oglądać wydarzenia, które ich interesują. Mogą być na bieżąco z zapowiedziami prelegentów i nie tylko.

2. Użytkownicy mogą aktywnie zapisywać się na wydarzenia, co oznacza, że planują uczestniczyć lub już wnieśli opłatę. W tym celu odpowiedzieli na wydarzenie.

3. Użytkownicy mogą uczestniczyć w minionych wydarzeniach. Chcą śledzić nagrania wideo, opinie i slajdy.

4. Użytkownicy mogli wylogować się z wydarzeń, co oznacza, że albo zasubskrybowali wydarzenie, ale zmienili zdanie, albo po prostu nie chcą już widzieć tego wydarzenia na swoich listach. Nasza aplikacja śledzi również te zdarzenia.

Jak zawsze, najpierw chcemy modelować nasze dane. Ponieważ nie chcemy zmieniać naszych istniejących typów, ale chcemy mieć szybki dostęp do wszystkich czterech kategorii, tworzymy kolejny obiekt, który służy jako mapa dla każdej kategorii. Typ tego obiektu wygląda następująco:

```
type UserEvents = {

watching: TechEvent[],

rvsp: TechEvent[],

attended: TechEvent[],

signedout: TechEvent[],

}
```

Teraz trochę operacji na tym obiekcie.

keyof

Chcemy dać użytkownikom możliwość filtrowania swoich wydarzeń. Pierwsza według kategorii: watching, rsvp, attended i signedout; po drugie - i opcjonalnie - według rodzaju wydarzenia: konferencja, meetup, webinar czy hackathon. Funkcja, którą chcemy utworzyć, przyjmuje trzy argumenty:

1. UserEventList, którą chcemy filtrować.

2. category, którą chcemy wybrać. Odpowiada to jednemu z kluczy obiektu userEventList.

3. Opcjonalnie ciąg z zestawu EventKind, który umożliwia nam dalsze filtrowanie.

Pierwsza operacja filtra jest dość prosta. Chcemy uzyskać dostęp do jednej z list za pośrednictwem operatora dostępu do indeksu; na przykład userEventList ['watching']. Dlatego dla typu kategorii tworzymy typ unii, który zawiera wszystkie klucze userEventList.

```
type UserEventCategory =

'watching' | 'rsvp' | 'attended' | 'signedoff'
```

```
function filterUserEvent(
userEventList: UserEvents,
category: UserEventCategory,
filterKind?: EventKind
) {
const filteredList = userEventList[category]
if (filterKind) {
return filteredList.filter(event =>
event.kind === filterKind)
}
return filteredList
}
```

To działa, ale napotykamy te same problemy, co w poprzedniej Lekcji: ręcznie obsługujemy typy, co jest podatne na błędy i literówki. Problemy tego typu, które są trudne do wyłapania. Być może nie zauważyłeś, że popełniłem błąd, używając typu wartości signedoff w UserEventCategory, która nie jest kluczem w UserEvents. To byłoby signedout. Chcemy tworzyć takie typy dynamicznie, a TypeScript ma do tego operator. Dzięki keyof możemy uzyskać klucze obiektów każdego zdefiniowanego przez nas typu. I mam na myśli każdy. Możemy użyć keyof nawet z typami wartości zestawu łańcuchowego i uzyskać wszystkie funkcje łańcuchowe. Lub z tablicą i pobrać wszystkie operatory tablicowe:

```
// 'speaker' | 'title' | 'abstract'
type TalkProperties = keyof Talk
// number | 'toString' | 'charAt' | ...
type StringKeys = keyof 'speaker'
// number | 'length' | 'pop' | 'push' | ...
type ArrayKeys = keyof []
```

Wynikiem jest typ unii typów wartości. Chcemy kluczy naszych zdarzeń użytkownika, więc to właśnie robimy:

```
function filterUserEvent(
userEventList: UserEvents,
category: keyof UserEvents,
filterKind?: EventKind
) {
const filteredList = userEventList[category]
if (filterKind) {
```

return filteredList.filter(event =>

event.kind === filterKind)

}

return filteredList

}

W momencie aktualizacji naszego typu UserEvent wiemy również, jakich kluczy możemy się spodziewać. Jeśli więc coś usuniemy, wystąpienia, w których używany jest usunięty klucz, otrzymają czerwone faliste linie. Jeśli dodamy kolejny klucz, TypeScript da nam odpowiednie autouzupełnianie.

Predykaty typu

Załóżmy, że filterUserEvents jest nie tylko w naszej aplikacji, ale także poza nią. Inne zespoły programistów w naszej organizacji mogą uzyskać dostęp do tej funkcji i mogą nie używać języka TypeScript do wykonania swojej pracy. Dla nich chcemy wychwycić niektóre możliwe błędy z góry, zachowując jednocześnie nasze bezpieczeństwo typu. W przypadku obu operacji filtrowania filtr category jest problematyczny, ponieważ może uzyskać dostęp do klucza, który nie jest dostępny w userEventList. Aby zapewnić nam bezpieczeństwo typu i większą elastyczność na zewnątrz, akceptujemy, że category nie jest podzbiorem ciągów, ale całym zestawem ciągów:

function filterUserEvent (

lista: Zdarzenia użytkownika,

kategoria: ciąg,

filterKind ?: EventKind

) {

// ... tbd

}

Ale zanim uzyskamy dostęp do kategorii, chcemy sprawdzić, czy jest to prawidłowy klucz na naszej liście. W tym celu tworzymy funkcję pomocniczą o nazwie isUserEventListCategory:

function isUserEventListCategory(

list: UserEvents,

category: string

) {

return Object.keys(list).includes(category)

}

i zastosuj to sprawdzenie do naszej funkcji:

function filterUserEvent(

list: UserEvents,

```
category: string,
filterKind?: EventKind
) {
if(isUserEventListCategory(list, category)) {
const filteredList = list[category]
if (filterKind) {
return filteredList.filter(event =>
event.kind === filterKind)
}
return filteredList
}
return list
}
```

Jest to wystarczające bezpieczeństwo, aby nie zawiesić programu, jeśli otrzymamy dane wejściowe, które nie działają dla nas. Ale TypeScript (zwłaszcza w trybie ścisłym) nie jest z tego zadowolony. Tracimy wszystkie połączenia z UserEvents, a category nadal jest ciągiem. Na poziomie typu, jak możemy być pewni, że mamy dostęp do odpowiednich właściwości? W tym miejscu pojawiają się predykaty typów. Predykaty typów są sposobem na dodanie większej ilości informacji do sterowania analizą przepływu. Możemy rozszerzyć możliwości zawężenia, mówiąc TypeScript, że jeśli wykonamy określone sprawdzenie, możemy być pewni, że nasze zmienne są określonego typu:

```
function isUserEventListCategory(
list: UserEvents,
category: string
): category is keyof UserEvents { // The type predicate
return Object.keys(list).includes(category)
}
```

Predykaty typów działają z funkcjami, które zwracają wartość logiczną. Jeśli ta funkcja ma wartość true, możemy być pewni, że category jest kluczem USerEvents. Oznacza to, że w prawdziwej gałęzi instrukcji if TypeScript lepiej zna typ. Zawęziliśmy ziór stringów do mniejszego zbioru kluczy UserEvents.