Lekcja 25: Związki dynamiczne

Rozważ następującą funkcję. Otrzymujemy listę wydarzeń technicznych i chcemy je przefiltrować według określonego typu zdarzenia:

```
type EventKind =

'conference' | 'webinar' | 'meetup'

function filterByKind(

list: TechEvent[],

kind: EventKind

): TechEvent[] {

return list.filter(el => el.kind === kind)

}
```

Ta funkcja przyjmuje dwa argumenty: list, oryginalna lista zdarzeń; i kind, według którego chcemy filtrować. Zwracamy nową listę wydarzeń technicznych. Używamy dwóch typów, aby poprawić bezpieczeństwo czcionek. Jednym z nich jest TechEvent, z którego często korzystaliśmy podczas ostatnich lekcji. Drugi to EventKind, czyli suma wszystkich dostępnych typów wartości dla rodzaju właściwości. Po wprowadzeniu tej unii możemy filtrować tylko zdarzenia wymienione w tej unii:

```
// A list of tech events we get from a back end

declare const eventList: TechEvent[]

filterByKind(eventList, 'conference') // OK!

filterByKind(eventList, 'webinar') // OK!

filterByKind(eventList, 'meetup') // OK!

// 'concert' is not part of EventKind

filterByKind(eventList, 'concert') // Bang!
```

Jest to ogromna poprawa dla programistów, ale wiąże się z pewnymi pułapkami, gdy zmieniają się nasze dane.

Typy wyszukiwania

Co jeśli dodamy inny typ wydarzenia do istniejącej listy typów wydarzeń, zwany Hackathonem? Wydarzenie z kodowaniem na żywo, które może coś kosztować, ale nie wymaga żadnych rozmów. Zdefiniujmy nowy typ:

```
type Hackathon = TechEventBase & {

location: string,

price?: number,

kind: 'hackathon'

}
```

I dodaj Hackathon do unii TechEvents:

type TechEvent =

Conference | Webinar | Meetup | Hackathon

Natychmiast uzyskujemy rozłączenie między EventKind a TechEvent. Nie możemy filtrować według „hackathonu”, mimo że powinno to być możliwe.

// To powinno być możliwe

filterByKind (eventList, 'hackathon') // Błąd

Jednym ze sposobów zmiany tego byłoby dostosowanie EventKind za każdym razem, gdy zmieniamy TechEvent. Ale to dużo wysiłku, zwłaszcza w przypadku powiększania lub zmieniania list danych. A co, jeśli nagle osobiste konferencje przestają istnieć? Chcemy, aby zmiany, które wprowadzamy w naszych typach, były jak najmniejsze. W tym celu musimy utworzyć połączenie między EventKind i TechEvent. Być może zauważyłeś, że typy obiektów mają podobną strukturę do obiektów JavaScript. Okazuje się, że mamy również podobne operatory na typach obiektów. Podobnie jak możemy uzyskać dostęp do właściwości obiektu przez indeksowanie go, możemy uzyskać dostęp do typu właściwości za pomocą odpowiedniego indeksu:

declare const event: TechEvent

// Accessing the kind property via the index

// operator

console.log(event['kind'])

// Doing the same thing on a type level

type EventKind = TechEvent['kind']

// EventKind is now

// 'conference' | 'webinar' | 'meetup' | 'hackathon'

Ponieważ połączenie TechEvent łączy już wszystkie możliwe wartości typów właściwości w związki, nie musimy już samodzielnie definiować EventKind. Tego typu typy nazywane są typami dostępu do indeksów lub typami wyszukiwania. Dzięki typom wyszukiwania tworzymy własny system połączonych typów, które tworzą czerwone faliste linie wszędzie, gdzie się ich nie spodziewaliśmy, działając jako zabezpieczenie naszej własnej, ciągle zmieniającej się pracy.

Mapowane typy

Mówiąc o typach generowanych dynamicznie, przyjrzyjmy się funkcji grupującej zdarzenia według ich rodzaju.

type GroupedEvents = {

conference: TechEvent[],

meetup: TechEvent[],

webinar: TechEvent[],

hackathon: TechEvent[]

```
}

function groupEvents(

events: TechEvent[]

): GroupedEvents {

const grouped = {

conference: [],

meetup: [],

webinar: [],

hackathon: []

};

events.forEach(el => {

grouped[el.kind].push(el)

})

return grouped

}
```

Funkcja tworzy mapę, a następnie przechowuje oryginalną listę wydarzeń technicznych w nowej kolejności, w oparciu o rodzaj zdarzenia. Ponownie mamy do czynienia z podobnym problemem jak poprzednio. Typ GroupedEvents jest obsługiwany ręcznie. Widzimy, że mamy cztery różne klucze oparte na zdarzeniach, z którymi pracujemy, a w momencie zmiany oryginalnej unii TechEvent musielibyśmy również zachować ten typ. Na szczęście TypeScript ma narzędzie do takich sytuacji. Za pomocą języka TypeScript możemy tworzyć typy obiektów, uruchamiając zestaw typów wartości w celu wygenerowania kluczy właściwości i przypisując im określony typ. W naszym przypadku chcemy, aby hackathon , webinar, meetup i conference były generowane automatycznie i mapowane na listę TechEvent, uruchamiając EventKind:

```
type GroupedEvents = {

[Kind in EventKind]: TechEvent[]

}
```

Nazywamy ten rodzaj mapowaniem typu. Zamiast mieć wyraźne nazwy właściwości, używają nawiasów do wskazania symbolu zastępczego dla ostatecznych kluczy właściwości. W naszym przykładzie klucze właściwości są generowane przez zapętlenie typu unii EventKind. Aby zwizualizować, jak to działa, rozwińmy zamapowany typ w kilku krokach:

```
// 1. The original declaration

type GroupedEvents = {

[Kind in EventKind]: TechEvent[]

}
```

```
// 2. Resolving the type alias.
// We suddenly get a connection to tech event
type GroupedEvents = {
[Kind in TechEvent['kind']]: TechEvent[]
}
// 3. Resolving the union
type GroupedEvents = {
[Kind in 'webinar' | 'conference'
| 'meetup' | 'hackathon']: TechEvent[]
}
// 4. Extrapolating keys
type GroupedEvents = {
webinar: TechEvent[],
conference: TechEvent[],
meetup: TechEvent[],
hackathon: TechEvent[],
}
```

Tak jak w przypadku naszego oryginalnego typu! Odwzorowane typy to nie tylko wygoda, która pozwala nam pisać dużo mniej i korzystać z tego samego rodzaju narzędzi. Tworzymy również rozbudowaną sieć połączonych informacji, która pozwala nam wychwycić błędy już w momencie zmiany danych. W momencie, gdy dodamy inny rodzaj zdarzenia do naszej listy wydarzeń technicznych, EventKind otrzyma automatyczną aktualizację i otrzymamy więcej informacji dla filterByKind. Wiemy również, że mamy inny wpis w GroupedEvents, a funkcja groupEvents nie zostanie skompilowana, ponieważ w zwracanej wartości brakuje klucza. Wszystkie te korzyści otrzymujemy bez dodatkowych kosztów. Musimy tylko jasno określić nasze typy i stworzyć niezbędne połączenia. Pamiętaj, że konserwacja typu jest potencjalnym źródłem błędów. Pomaga dynamiczne aktualizowanie typów.