Lekcja 42: Praca z wartością null

Dowiedzieliśmy się, że undefined i null są częściami każdego zestawu w przestrzeni typów, chyba że ustawimy flagę strictNullChecks. To usuwa undefined i null i traktuje je jako własne jednostki. To skłania TypeScript do rzucania w nas czerwonych zawijasów, gdy zapomnimy obsłużyć wartości zerowe. Ma to ogromny wpływ na kod, który sami piszemy. Jeśli używamy typów jako kontraktów do przekazywania danych między funkcjami, możemy być pewni, że mamy już do czynienia z wartościami null i undefined. Gorzka prawda jest taka, że wartości zerowe mogą i będą się zdarzać, przynajmniej w punkcie, w którym nasze oprogramowanie będzie musiało współpracować z zewnętrznymi danymi wejściowymi:

1. element z DOM, który chcemy wybrać

2. dane wejściowe użytkownika z pola wejściowego

3. dane, które pobieramy asynchronicznie z zaplecza

Przyjrzyjmy się bardzo uproszczonej funkcji fetchOrderList, która robi mniej więcej to samo, co wcześniej w tym rozdziale, ale jest wyłącznie asynchroniczna.

```
declare function fetchOrderList(

input: Customer | Product

): Promise<Order[]>
```

Umowa tej funkcji mówi nam, że otrzymujemy obietnicę, która zdecydowanie zwraca listę zamówień. Niejawny komunikat jest taki, że pobieranie zamówień również się powiodło i że obsługa błędów lub wartości zerowych już się zdarzyła. Jeśli zaimplementujemy tę funkcję z fetch, tak jak to zrobiliśmy w lekcji 37, zobaczymy, że mamy problem: fetch zwraca obietnicę any (najwyższy typ, który obejmuje wszystko i przyjmuje wszystko, w tym null i undefined - i never, jeśli weźmiemy możliwość uwzględnienia błędu). Oznacza to, że wewnątrz tej funkcji tracimy informacje, jeśli wartość zwracana jest faktycznie zdefiniowana. Musimy być bardziej szczegółowi co do zestawu możliwych wartości zwracanych, zwłaszcza że strictNullChecks mówi, że nie bierzemy wartości nullish do naszych zbiorów. Prawdziwa nagłówek funkcji fetchOrderList jest znacznie bardziej podobny do tego:

```
declare function fetchOrderList(

input: Customer | Product

): Promise<Order[] | null>
```

To jest dobre. Dodajemy wartości zerowe z powrotem do naszych zestawów i wyraźnie o tym mówimy. Oznacza to, że jesteśmy również zmuszeni sprawdzić, czy wartości mogą być zerowe. To sprawia, że nasz kod jest znacznie bezpieczniejszy niż wcześniej.

NonNullable

Aby obsłużyć wartość null, mamy dwie możliwości, o czym świadczy funkcja listOrders, która wyświetla wszystkie zamówienia na konsoli. Pierwsza opcja polega na dodaniu wartości null do argumentów wejściowych.

```
declare function listOrders(Order[] | null): void
```

Dzięki temu funkcja listOrders jest zgodna z danymi wyjściowymi funkcji fetchOrderList. Wewnątrz listOrders należy przeprowadzić kontrolę zerową. Inną opcją jest upewnienie się, że nigdy nie przekazujemy wartości zerowych do listOrders. Oznacza to, że przed:

declare function listOrders(Order[]): void

W każdym razie będziemy musieli sprawdzić, czy nie ma wartości null. I najprawdopodobniej nie tylko w przypadku list zamówień, ale także list produktów, list klientów itp. Wywołuje to ogólną funkcję pomocniczą, która sprawdza, czy obiekt jest rzeczywiście dostępny.

declare function isAvailable<Obj>(

obj: Obj

): obj is NonNullable<Obj>

Ta funkcja ogólna wiąże się ze zmienną typu Obj. Jak dotąd Obj może być wszystkim. Nie mamy żadnych ograniczeń typu i nie chcemy żadnych ograniczeń typu. isAvailable powinno działać ze wszystkim. Ale wynik powinien zapewnić, że nie mamy żadnych zerowych wartości. Dlatego używamy wbudowanego narzędzia typu NonNullable, które usuwa wartości null i undefined z naszego zestawu wartości. Wartość NonNullable jest zdefiniowana w następujący sposób:

type NonNullable<T> =

T extends null | undefined ? never : T

NonNullable jest dystrybucyjnym typem warunkowym. Jeśli przekażemy unię, w której jawnie ustawimy wartość null lub undefined, możemy upewnić się, że ponownie usuniemy te typy wartości i będziemy kontynuować zawężony zestaw. Oto realizacja:

function isAvaialble<Obj>(

obj: Obj

): obj is NonNullable<Obj> {

return typeof obj !== ‘undefined’ &&

obj !== null

}

To jest funkcja pomocnicza w akcji:

// orders is Order[] | null

const orders = await fetchOrderList(customer)

if(isAvailable(orders)) {

//orders is Order[]

listOrders(orders)

}

Zaleca się wczesne rozważenie wartości zerowych. Utrzymuj rdzeń aplikacji bez wartości zerowej i spróbuj jak najszybciej wyłapać wszelkie możliwe skutki uboczne.

Narzędzia niskiego poziomu

Wbudowane typy warunkowe języka TypeScript bardzo pomagają, jeśli pracujesz nad funkcjami narzędziowymi niskiego poziomu, których możesz ponownie użyć w swojej aplikacji. To samo dotyczy naszych własnych typów użytkowych, które zadeklarowaliśmy w poprzedniej lekcji. funkcja fetchOrderList jest bardzo szczegółowa; teraz pomyśl o znacznie bardziej ogólnej funkcji i niektórych możliwych procesach.

Najpierw pobieranie z bazy danych.

type FetchDBKind =

'orders' | 'products' | 'customers'

type FetchDBReturn<T> =

T extends 'orders' ? Order[] :

T extends 'products' ? Products[] :

T extends 'customers' ? Customers[] : never

declare function fetchFromDatabase<

Kin extends FetchKind

>(

kind: Kin

function process<T extends Promise<any>>(): Promise<FetchDbReturn<Kin>| null>

Przetwarzanie wszystkiego, co pobraliśmy i upewnienie się, że otrzymujemy do tego odpowiednią funkcję procesu.

promise: T,

cb: (res: Unpack<NonNullable<T>>) => void

): void {}

promise.then(res =>

if(isAvailable(res)) {

cb(res)

}

)

}

To pozwala nam bezpiecznie listOrder do funkcji, której wyniki mogą być niejednoznaczne.

process(

fetchFromDatabase('orders'),

listOrders

)

Podsumowanie

Typy warunkowe są prawdopodobnie najbardziej unikalną cechą systemu typów TypeScript. Wynikają one bezpośrednio z ogromnej elastyczności, jaką JavaScript ma do zaoferowania programistom. Tego się nauczyliśmy.

1. Typy warunkowe są doskonałym narzędziem do tworzenia bezpośrednich połączeń między typami wejściowymi i wyjściowymi, coś, co ginie w przeciążeniach funkcji bardzo wcześnie.

2. Nadal potrzebujemy dobrej i zdrowej kombinacji typów warunkowych i przeciążeń funkcji, aby mieć pewność, że funkcje z niejednoznacznymi wynikami będą zrozumiałe w sposób bezpieczny dla typów.

3. Dowiedzieliśmy się o dystrybucyjnej naturze typów warunkowych, gdy są używane z parametrami typu nagiego. Typ warunkowy typów unii jest podobny do typu unii typów warunkowych.

4. To pozwala nam filtrować za pomocą typu never, redukując liczbę związków i bardziej precyzyjnie określając, czego oczekujemy.

5. Dzięki typom pomocniczym, takim jak Pick, Extract i Exclude, możemy modelować zachowanie do zestawu danych i upewnić się, że bez względu na to, jak zmieniają się dane, nasze procesy są przygotowane na zmianę. Mniej konserwacji, większe bezpieczeństwo typów.

6. Dowiedzieliśmy się o słowie kluczowym infer i o tym, jak można go użyć do wyodrębnienia typów ze znacznie bardziej złożonego typu ogólnego.

7. Dowiedzieliśmy się o pracy z typami null i Non Nullable i zdaliśmy sobie sprawę, jak dobry zestaw prymitywów niskiego poziomu może uczynić nasz własny kod bardziej ogólnym i elastycznym bez utraty bezpieczeństwa typów.

Typy warunkowe powinny stać się nieocenionymi narzędziami na pasku TypeScript. Będziesz mógł tworzyć mocne pisma za pomocą zaledwie kilku wierszy kodu i możesz skupić się wyłącznie na pisaniu kodu JavaScript