Lekcja 41: Słowo kluczowe infer

Podczas wydajnej pracy z TypeScript chcemy, aby obsługa typów była jak najmniejsza. W końcu typy powinny pomóc nam być produktywnym i nie przeszkadzać w tym, co próbujemy osiągnąć. Do tej pory trzymaliśmy się przejrzystego przepływu pracy: modeluj dane, opisuj zachowanie. Nie chcemy spędzać zbyt wiele czasu na utrzymaniu typów, jeśli możemy dynamicznie tworzyć typy z innych typów. Są jednak chwile, kiedy nie jesteśmy pewni, jak będzie wyglądał nasz model. Zwłaszcza w trakcie rozwoju sytuacja może się zmienić. Dane można dodawać i usuwać, a ogólny kształt obiektu zmienia się. To jest w porządku. To elastyczność, z której słynie JavaScript! Pomyśl o rozszerzeniu naszej aplikacji administracyjnej e-commerce o funkcję, która tworzy użytkowników, którzy mogą czytać i modyfikować zamówienia, produkty i klientów. Funkcja może wyglądać mniej więcej tak:

```
// A userId variable counting up... not safe

// but we are in development mode

let userId = 0

function createUser(name, roles) {

return {

userId: userId++,

name,

roles,

createdAt: new Date()

}

}
```

Całkiem proste: dwie wygenerowane właściwości, a pozostałe są po prostu dodawane do obiektu. Zwróć uwagę na brak typów! Później nasza funkcja może stać się bardziej konkretna. Role są podzielone

• admin: może czytać i modyfikować wszystko.

• konserwacja: wolno modyfikować produkty.

• wysyłka: możliwość czytania zamówień w celu uzyskania informacji niezbędnych do ich wysłania.

```
function createUser(

name: string,

role: ‘admin’ | ‘maintenance’ | ‘shipping’,

isActive: boolean

) {

return {

userId: userId++,

name,
```

```
role,

isActive,

createdAt: new Date()

}

}
```

Ale to tylko kolejna mutacja. Typy danych wejściowych stają się bardziej konkretne, a typ zwracany dostosowuje się do zmian. Zawsze musielibyśmy utrzymywać typ User, jeśli chcemy kontynuować pracę z użytkownikami w środowisku bezpiecznym dla typów.

Typ zwracany infer

Bardzo by pomogło, gdybyśmy mogli nazwać typ zwracany przez createUser. TypeScript może wywnioskować typy na podstawie przypisań. Typ zmiennej przyjmuje kształt zwracanej przez funkcję.

```
// The type of user is the shape returned by createuser

const user = createUser('Stefan', 'shipping', true)
```

Możemy nadać temu nazwę za pomocą operatora typeof:

```
/*

type User = {

userId: number,

name: string,

role: 'admin' | 'maintenace' | 'shipping',

isActive: boolean,

createdAt: Date

}

*/

type User = typeof user
```

To daje nam typ User, ale za bardzo wysoką cenę. Zawsze musimy wywołać funkcję, aby uzyskać kształt zwracanego typu. Co się stanie, jeśli to wywołanie funkcji spowoduje mutację krytycznych danych, na przykład w bazie danych? Czy transakcje bazy danych są uzasadnione tylko w celu pobrania typu obiektu? Chcemy pobrać typ zwracany z sygnatury funkcji. W takich sytuacjach TypeScript pozwala nam wywnioskować zmienne typu w klauzuli extends typu warunkowego. Utwórzmy typ GetReturn, który przyjmuje funkcję. Dowolną funkcję. Na razie chcemy sprawdzić, czy przekazany typ jest prawidłową funkcją.

```
type GetReturn<Fun> =

Fun extends (...args: any[]) => any ? Fun : never
```

Łączymy wszystkie możliwe argumenty w krotkę argumentów. Wiemy, że mamy dowolny typ zwrotu. Jeśli przekażemy naszą funkcję, otrzymamy w zamian jej typ:

type Fun = GetReturn<typeof createUser>

Teraz mamy możliwość wnioskowania o typach, które znajdują się w tej klauzuli extends. Dzieje się tak ze słowem kluczowym infer. Możemy wybrać zmienną typu i zwrócić zmienną tego typu.

type GetReturn<Fun> =

Fun extends (...args: any[]) => infer R ? R : never

Dzięki infer R mówimy, że niezależnie od typu zwracanego tej funkcji, przechowujemy ją w zmiennej typu R. Jeśli mamy prawidłową funkcję, zwracamy R jako typ.

/*

type User = {

userId: number,

name: string,

role: 'admin' | 'maintenance' | 'shipping',

isActive: boolean,

createdAt: Date

}

*/

type User = GetReturn<typeof createUser>

Zero konserwacji; zawsze poprawne typy. Ten typ pomocnika jest dostępny w języku TypeScript jako ReturnType <Fun>.

Rodzaje pomocników

Typy pomocnicze, takie jak ReturnType, są niezbędne, jeśli konstruujemy funkcje i biblioteki, w których bardziej zależy nam na zachowaniu i wzajemnych połączeniach części niż na samych typach. Przechowywanie i pobieranie obiektów z bazy danych, tworzenie obiektów na podstawie schematu, tego typu rzeczy. Dzięki słowu kluczowemu infer uzyskujemy dużą elastyczność w uzyskiwaniu typów, nawet jeśli nie wiemy jeszcze, z czym mamy do czynienia. Na przykład prosty typ, który pozwala nam pobrać ustaloną wartość obietnicy:

type Unpack<T> =

T extends Promise<infer Res> ? Res : never

type A = Unpack<Promise<number>> // A is number

Lub typ, który spłaszcza tablicę, więc otrzymujemy typ wartości tablicy.

type Flatten<T> =

T extends Array<infer Vals> ? Vals : never

type A = Flatten<Customer[]> // A is Customer

TypeScript ma kilka wbudowanych typów warunkowych, które używają wnioskowania. Jednym z nich są Parameters, które gromadzą wszystkie argumenty z funkcji w krotce.

type Parameters<T> =

T extends (...args: infer Param) => any

? Param

: never

/* A is [

string,

“admin” | “maintenace” | “shipping”,

boolean

]

*/

type A = Parameters<typeof createUser>

Inni to:

• InstanceType. Pobiera typ utworzonego wystąpienia funkcji konstruktora klasy.

• ThisParameterType. Jeśli używasz funkcji wywołania zwrotnego, które to wiążą, możesz w zamian otrzymać powiązany typ.

• OmitThisParameterType. Używa wnioskowania, aby zwrócić sygnaturę funkcji bez tego typu. Jest to przydatne, jeśli Twoja aplikacja nie dba o powiązanie tego typu i musi być bardziej elastyczna w przekazywaniu funkcji.

Typy ze słowem kluczowym infer mają jedną wspólną cechę: są to niskopoziomowe typy narzędziowe, które pomagają w łatwym sklejaniu części kodu. Jest to zachowanie zdefiniowane w typie i pozwala na bardzo ogólne scenariusze, w których kod musi być wystarczająco elastyczny, aby sprostać nieznanym oczekiwaniom.