Lekcja 15: Funkcja wyszukiwania

W JavaScript funkcje są wszędzie. W ostatnich kilku przykładach widzieliśmy kilka funkcji. Funkcje przyczyniają się do tworzenia wartości w przestrzeni TypeScript. Ich parametry mają typy, które są dowolne i można je wywnioskować z wartości domyślnych. Zwracane wartości mają typy; można je wywnioskować z rzeczywistej wartości, którą zwracamy w treści funkcji. Zamierzamy utworzyć pole wyszukiwania dla naszej witryny. W momencie, gdy wprowadzamy zapytanie, wywołujemy zaplecze, które dostarcza nam wyniki w formacie JSON. W naszym programie tworzymy funkcję wyszukiwania, która pobiera parametry zapytania, wywołuje interfejs API wyszukiwania zaplecza i zwraca poprawne wyniki.

Typowanie nagłówków funkcji

Spójrzmy na nagłówek funkcji naszej funkcji wyszukiwania. Używamy słowa kluczowego declare, aby udostępnić funkcję bez implementowania jej treści. To świetny sposób na zastanowienie się nad typami i interfejsem funkcji przed przejściem do szczegółów. Później usuniemy słowo kluczowe declare i upewnimy się, że otrzymamy właściwą implementację.

```
/ A helper type with the results we expect

// from calling the back end

type Result = {

title: string,

url: string,

abstract: string

}

/**

* Funkcja wyszukiwania przyjmuje zapytanie, które wysyła

* do końca, a także kilka tagów

* jako tablicę ciągów, aby uzyskać przefiltrowane wyniki

*/

declare function search(

query: string,

tags: string[]

): Result[]
```

Jawnie wpisaliśmy parametry i wartości zwracane przez nagłówek funkcji, co jest dobrą praktyką, ponieważ pozwala upewnić się, że nie tylko otrzymujesz właściwe wartości zwracane i przetwarzasz je dalej, ale możesz również zweryfikować swoją implementację pod kątem jawnych typów, których oczekujesz. Szef funkcji ma kilka problemów:

1. Filtry tagów są wymagane. Nie byłoby możliwe wywołanie funkcji wyszukiwania bez dostarczonych żadnych tagów, nawet jeśli jest to tylko pusta tablica.

2. Funkcja wyszukiwania jest niejawnie synchroniczna. Wywołujemy wyszukiwanie i oczekujemy w zamian szeregu wyników. Wywołanie zaplecza zwykle obejmuje operację asynchroniczną. Cóż, przynajmniej powinno! W naszym przypadku chcemy, aby funkcja wyszukiwania działała asynchronicznie.

Rozwiążmy każdy problem.

Parametry opcjonalne

Tagi są fajne do filtrowania wyników na podstawie preferencji. Pomyśl o stronie internetowej, która oferuje artykuły na temat JavaScript, CSS, projektowania, kierownictwa graficznego i UX. Czy możesz coś wymyślić? Na przykład w momencie wyszukiwania hasła „Ember" należy odróżnić środowisko JavaScript od cyfrowego albumu. Dlatego wybierasz tagi.

Ale tagi są opcjonalne, przynajmniej w naszym przypadku. Nasza funkcja jeszcze nie wie:

search ('Ember', ['JavaScript']) // Działa

search ('Ember') // Błędy! Brakuje tagów

search ('Ember', []) // Paskudne obejście

Przekazywanie pustych wartości nie przypomina języka JavaScript. Podobnie jak obiekty, funkcje mogą przyjmować parametry opcjonalne, oznaczone znakiem zapytania:

declare function search(

query: string,

tags?: string[]

): Result[]

search('Ember') // Yes!

search('Ember', ['JavaScript']) // Also yes!

search('Ember', ['JavaScript', 'CSS']) // Yes yes!

Podobnie jak opcjonalne właściwości w obiektach, po napisaniu treści funkcji musisz sprawdzić, czy są one dostępne.

Asynchroniczne wywołania zaplecza

Aby nasza funkcja była poprawnie asynchroniczna, będziemy pracować z obietnicami. Obietnice to główna konstrukcja JavaScript do radzenia sobie z zadaniami asynchronicznymi. Nazywa się je obietnicami, ponieważ obiecują osiągnąć jakąś wartość… w końcu. Po prostu nie wiemy, kiedy. Jednym z najpopularniejszych zadań opartych na obietnicach jest pobieranie, wygodny sposób wywoływania zaplecza i odbierania danych. Użyjmy pobierania, aby zaimplementować naszą funkcję wyszukiwania. Usuń słowo kluczowe declare z nagłówka funkcji. Nie deklarujemy już funkcji, my ją wdrażamy. Na razie usuńmy również zwracany typ wartości z naszej deklaracji funkcji. Dodamy go później.

function search(query: string, tags?: string[]) {

// Based on our input parameters, we build a query

// string

```
let queryString = `?query=${query}`

// tags can be undefined as it's optional.

// let's check if they exist

if(tags && tags.length) {

// and add all tags in that array to the

// query string

queryString += `&tags=${tags.join()}`

}

// Ready? Fetch from our search API

return fetch(`/search${queryString}`)

// When we get a response, we call the

// .json method to get the actual results

.then(response => response.json())

}
```

Kilka rzeczy, na które należy zwrócić uwagę:

1. Musimy sprawdzić, czy tags istnieją. Gdy najedziesz kursorem na opcjonalny parametr tags w treści funkcji, zobaczysz, że może to być tablica ciągów, ale może być również niezdefiniowany. Tylko jeśli sprawdzimy, czy wartość istnieje, TypeScript pozwoli nam użyć metod tablicowych i dołączyć do ciągu zapytania.

2. fetch zwraca obietnicę. Obietnice są wtedy możliwe, co oznacza, że gdy wartość jest tutaj, możemy uzyskać dostęp do wywołania zwrotnego funkcji then. W tym momencie interesujące jest oglądanie typów. Najedź kursorem na fetch aby zobaczyć, że funkcja fetch zwraca Promise <Response>, a parametr response wewnątrz wywołania zwrotnego jest typu Response. Jeśli kiedykolwiek zobaczysz typ ze znakami < i >, pamiętaj o nazwie ogólnej. Obietnice mogą dotyczyć wielu rzeczy. Jeśli chcemy określić typ zwracanej wartości, używamy metody generycznej, aby ustawić ją na, na przykład Response. W dalszych Lekcjach zobaczymy wiele ogółó.

Jedną fajną rzeczą w TypeScript jest to, że nie musisz znać na pamięć wszystkich interfejsów API. Po wywołaniu funkcji fetch edytor podpowiada, co można przekazać jako argumenty. Kiedy jesteś w wtedy w wywołaniu zwrotnym then, nie musisz wiedzieć, że response ma funkcję .json (). Możesz przeglądać listę sugestii w swoim edytorze i wybrać ten, który Twoim zdaniem jest najbardziej odpowiedni. Najprawdopodobniej zatrzymasz się na .json () i pomyślisz: „Ach tak, właśnie tego szukałem”. fetch zwraca wartość typu Promise <Response>. Następnie zwraca wartość zwracaną przez response.json, która również staje się typem wartości zwracanej naszej funkcji wyszukiwania. Wnioskowanie o typie!

Jednak typ wartości zwracanej przez response.json to Promise<any>. I słusznie! Skąd TypeScript powinien wiedzieć, co otrzymujesz po wywołaniu zaplecza? To, czego chcemy, to w rzeczywistości to, co otrzymujemy: obietnica wyników. Lub w typach: Promise <Result []>

Tutaj musimy być jawni, albo poprzez rzutowanie typu:

```
function search(query: string, tags?: string[]) {
...
return fetch(`/search${queryString}`)
.then(response =>
response.json() as Promise<Result[]>)
}
```

co jest w porządku, ponieważ dokładniej określamy typ w miejscu, w którym go otrzymujemy. Lub inną możliwością jest nagłówek funkcji:

```
function search(
query: string,
tags?: string[]): Promise<Result[]> {
...
return fetch(`/search${queryString}`)
.then(response => response.json())
}
```

Obie wersje działają tak samo: każda jest kompatybilna z każdym innym typem. To symbol wieloznaczny, typ beztroski, w którym wszystko może się zdarzyć. Mówimy wprost, że oczekujemy tablicy Result zamiast any. A TypeScript to akceptuje! To, której wersji używasz, zależy od Ciebie. Rzuty typu są szybkie i dostępne tam, gdzie są potrzebne, ale czasami mogą zostać przeoczone. Wolę jawne nagłówki funkcji i pozostawienie JavaScript wewnątrz ciała funkcji w takiej postaci, w jakiej jest.