Lekcja 16: Funkcje wywołania zwrotnego

W poprzedniej Lekcji zajmowaliśmy się typami zwracanych wartości i parametrów w funkcjach. Ale funkcje też mają typy! Rzućmy okiem na naszą funkcję wyszukiwania z poprzedniej lekcji:

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

Najłatwiej możemy uzyskać typ funkcji, wywołując typeof:

```
type SearchFn = typeof search
```

Po najechaniu kursorem na SearchFn zobaczysz rozszerzoną definicję typu funkcji:

```
type SearchFn = (
query: string, tags?: string[] | undefined
) => Promise<Result[]>
```

Jest to bardzo podobne do notacji strzałek JavaScript dla funkcji. Widzimy: otwarcie nagłówka funkcji, zadeklarowanie zapytania parametrycznego, które jest łańcuchem; tagi, które mogą być tablicą ciągów lub nieokreślone, ponieważ są opcjonalne; potem pojawia się strzałka, a funkcja zwraca obietnicę wyników. Oczywiście możemy wpisać typ funkcji zgodnie z definicją tam, bez używania typeof. Ale typeof jest przydatny!

Typy funkcji w obiektach

Więc co możemy zrobić z typami funkcji? Sporo! Możemy zdefiniować złożone typy obiektów, które zawierają funkcje:

```
type Query = {
query: string,
tags?: string[],
assemble: (includeTags: boolean) => string
}
```

Definiuje obiekt zapytania, który zawiera nie tylko samo zapytanie, ale także opcjonalne znaczniki i funkcję, która zestawia ciąg zapytania w celu ewentualnego wyszukiwania. Taki przedmiot byłby zgodny z umową:

```
const query: Query = {

query: 'Ember',

tags: ['javascript'],

assemble(includeTags = false) {

let query = `?query=${this.query}`

if(includeTags && typeof this.tags !==

'undefined') {

query += `&${this.tags.join(',')}`

}

return query

}

}
```

Typy są komponowalne, więc możemy zdefiniować typ funkcji dla assemble w innej pozycji:

```
type AssembleFn = (includeTags: boolean) => string

type Query = {

query: string,

tags?: string[],

assemble: AssembleFn

}
```

Zachowuje się podobnie jak JavaScript, w którym możemy również definiować funkcje poza głównym obiektem.

Typy funkcji w funkcjach

Powiedzmy, że chcemy, aby nasza funkcja wyszukiwania była podłączana i chcemy tworzyć różne funkcje wyszukiwania dla różnych scenariuszy. Może takie, które działają inaczej, ale zawsze słuchają tych samych parametrów: zapytania, listy tagów. I odwzajemniają obietnicę z wynikami. Zdefiniowaliśmy już funkcję wyszukiwania w następujący sposób:

```
type SearchFn = (

query: string, tags?: string[] | undefined

) => Promise<Result[]>
```

Możemy użyć tego typu funkcji do tworzenia różnych funkcji, które mają ten sam podpis. Jest to niezwykle pomocne w przypadku funkcji wywołania zwrotnego. Funkcje są obywatelami pierwszej

klasy w JavaScript i mogą być używane jako wartości, tak jak każdy ciąg lub liczba. Możemy więc przekazywać funkcje jako argumenty do innej funkcji. Zwykle ten typ funkcji nazywany jest funkcją zwrotną, ponieważ powinien wywoływać z powrotem miejsce, w którym funkcja została wywołana. Napiszmy funkcję, która łączy kilka przepływów pracy dla naszego wyszukiwania:

1. wybór elementu, w którym użytkownik wprowadza zapytanie

2. wywołanie wyszukiwania

3. pokazanie wyników

Wymaga trzech argumentów:

1. identyfikator elementu wejściowego

2. identyfikator elementu, w którym mają być prezentowane wyniki

3. funkcja wyszukiwania

Rodzaje tych dwóch parametrów są dość proste. Oba identyfikatory są ciągami. Funkcja wyszukiwania używa wspomnianego typu funkcji. Ponownie, pomyśl najpierw o nagłówku funkcji , zanim zaczniesz rozwijać ciało. Napiszmy nagłówek funkcji, który pobiera dwa ciągi dla identyfikatorów naszych elementów, a także funkcję wyszukiwania:

```
declare function displaySearch(

inputId: string,

outputId: string,

search: SearchFn

): void
```

Oczywiście moglibyśmy jawnie wpisać typ funkcji. Ale czasami bardziej czytelny jest osobny typ do tego. Używamy void jako wartości zwracanej, ponieważ nie spodziewamy się, że coś zwróci.

Funkcje anonimowe

Później zajmiemy się znaczeniem void i implementacją displaySearch. Na razie skupmy się na możliwych funkcjach wyszukiwania, które możemy przekazać; na przykład oryginalna funkcja wyszukiwania:

```
displaySearch ('searchField', 'result', search)
```

Albo zupełnie nowy, który właśnie wymyśliliśmy. Spójrz na tę funkcję testową:

```
displaySearch(

'searchField',

'result',

function(query, tags) {

return Promise.resolve([{

title: `The ${query} test book`,
```

```
url: `/${query}-design-patterns`,

abstract: `A practical book on ${query}`

}])

}

)
```

Zauważamy tutaj kilka rzeczy:

1. Funkcja jest anonimowa: nie ma nazwy. Właśnie został przekazany jako argument do displaySearch.

2. Ale nadal jest zgodne z umową SearchFn. Oba parametry są tutaj i zwraca obietnicę z tablicą wyników, nawet jeśli jest tylko jeden wpis.

3. Nie potrzebujemy adnotacji typu. Wszystkie adnotacje typu są definiowane za pomocą SearchFn, a TypeScript wnioskuje prawidłowe typy: query staje się ciągiem, tags stają się tablicą ciągów. I otrzymamy czerwone faliste linie, jeśli nie zwrócimy obietnicy z wartościami kształtu Result [].

Ten sam typ wnioskujemy, jeśli wyodrębnimy funkcję do jej własnej funkcji anonimowej i przypiszemy ją do const, która jest jawnie typowana za pomocą SearchFn:

```
const testSearch: SearchFn = function(query, tags) {

// All types still intact

return Promise.resolve([{

title: `The ${query} test book`,

url: `/${query}-design-patterns`,

abstract: `A practical book on ${query}`

}])

}
```

TypeScript ma strukturalny system typów. Dotyczy to również funkcji: kształt musi być nienaruszony. Jednak kształty funkcji działają nieco inaczej: struktura i kształt nie są definiowane przez nazwy argumentów, jak w obiektach, ale przez kolejność argumentów. Oznacza to, że możemy zmienić nazwy naszych parametrów i nadal zachować typy:

```
const testSearch: SearchFn = function(term, options) {

// term is a string (as defined by query)

// options is an optional string array

return Promise.resolve([{

title: `The ${term} test book`,

url: `/${term}-design-patterns`,

abstract: `A practical book on ${term}`

}])
```

}

Jest to szczególnie ważne, jeśli różne nazwy mają więcej sensu. Co się stanie, jeśli nasz zaplecze nie działa z tagami, ale z różnymi częściami większego portalu, takimi jak dokumentacja, witryna marketingowa lub forum? Nazwy takie jak sections lub subdomain byłyby bardziej sensowne niż bardzo ogólna nazwa tags. Możemy również całkowicie usunąć opcjonalne parametry tags (tablicę ciągów), jeśli nie mamy z tego żadnego pożytku:

```
const testSearch: SearchFn = function(term) {

// All types still intact

return Promise.resolve([{

title: `The ${term} test book`,

url: `/${term}-design-patterns`,

abstract: `A practical book on ${term}`

}])

}
```

To jest siła systemów typu strukturalnego dla funkcji. Ale wiesz co? Możemy zmienić jeszcze więcej.