Lekcja 19: Pasek narzędziowy typu funkcji

W JavaScript nie zajdziesz daleko bez funkcji. Jak widzieliśmy, typy funkcji mogą być bardzo wszechstronne, obejmując wiele przypadków użycia, które pojawiają się w zwykłym JavaScript. Mantra jest zawsze taka sama: najpierw JavaScript i tylko tyle typów, aby dać kompilatorowi coś do pracy. W tej lekcji przyjrzymy się kilku scenariuszom funkcji, w których język TypeScript pomaga uzyskać dodatkowe informacje o typie.

Oznaczone literały szablonów

Wśród prawdopodobnie najfajniejszych funkcji w ostatnim JavaScript są literały szablonów. Łączenie ciągów zawsze było czymś, co wydawało się zbyt nużące dla współczesnego języka programowania. Dzięki literałom szablonów jest to po prostu eleganckie:

const term = 'Ember'

const results = 12

const result text =

`You searched for ${term}, and got ${no} results`

Możemy wykonać dowolne wyrażenie w ciągu i połączyć wynik z resztą ciągu. Rozszerzeniem literałów szablonów są oznaczone literały szablonów. Pod względem składniowym są one bardzo podobne: ciągi znaków podświetlenia z wyrażeniami JavaScript, ale przed nimi znajduje się niestandardowy tag. Pomyślmy o znaczniku podświetlenia, który pozwala nam zastąpić określony symbol w ciągu znaków niektórymi elementami HTML; na przykład:

const result = {

title: 'A guide to @@starthl@@Ember@@endhl@@.js',

url: '/a-guide-to-ember',

description: 'The framework @@starthl@@Ember@@endhl@@.js

in a nutshell'

}

Chcemy zastąpić każdy @@ starthl @@ tagami <mark>, a @@ endhl @@ tagami zamykającymi </mark>. Scenariusz, w którym chcemy zamienić te ciągi, znajduje się w budowaniu znaczników listy wyników: lista nieuporządkowana (<ul>) z dużą ilością elementów listy (<li>). Pozycja listy zawiera tytuł wyniku, ale z zastąpionymi znacznikami podświetlenia. Aby to osiągnąć, tworzymy tag o nazwie highlight, który działa w ten sposób:

let markup = highlight` <li> $ {result.title} </li> `

Znacznik dla otagowanego literału szablonu to nic innego jak funkcja, która ma zdefiniowany zestaw parametrów.

1. Pierwszym parametrem jest TemplateStringsArray, tablica zawierająca wszystkie ciągi znaków wokół wyrażeń. W naszym przypadku <li> i

2. Druga to tablica łańcuchowa zawierająca rzeczywiste wyrażenia. W naszym przypadku cokolwiek daje nam ${result.title}.

Naszym zadaniem jest ponowne połączenie wszystkich części strun i dokonywanie modyfikacji, gdziekolwiek zechcemy; na przykład: zamień wszystkie znaczniki na rzeczywiste elementy. TypeScript dostarcza nam do tego odpowiednie typy. Typ TemplateStringsArray różni się od innych tablic ciągów, ponieważ jest tylko do odczytu i ma wskaźnik do nieprzetworzonej tablicy. Odzwierciedla to rzeczywistą implementację w JavaScript. Implementacja może wyglądać mniej więcej tak:

```
function highlight(
strings: TemplateStringsArray,
...values: string[]
) {
let str = '' // The result string
strings.forEach((templ, i) => {
// Fetch the expression from the same position
// or assign an empty string
let expr = values[i]?
.replace('@@start@@', '<em>')
.replace('@@end@@', '</em>') ?? ''
// Merge template and expression
str += templ + expr
});
return str
}
```

Z takim nagłówkiem funkcji, TypeScript rozpoznaje highlight jako znacznik szablonu. W użyciu literał szablonu ze znacznikami highlight wygląda mniej więcej tak:

```
function createResultTemplate(results: Result[]): string {
return `<ul>
${results.map(result =>
highlight`<li>${result.title}</li>`)}
</ul>`
}
```

Poręczna i, przy użyciu specjalnej głowicy funkcyjnej, bardzo celowa. Tego właśnie chcemy!

Parametry odpoczynku

Właśnie zobaczyłeś bardzo dziwną notację dla drugiej tablicy ciągów w podświetleniu. A co z tymi kropkami?

```
declare function highlight(

strings: TemplateStringsArray,

...values: string[]

): string
```

Kropki mówią nam, że value to parametr spoczynkowy. Jak ustaliliśmy na innych lekcjach, funkcje w JavaScript można wywoływać z praktycznie nieograniczoną liczbą parametrów. Implementacja funkcji dba o liczbę faktycznie używanych parametrów. Ale co, jeśli chcemy użyć wszystkich parametrów, bez względu na ich liczbę? Tutaj właśnie wchodzą w grę parametry odpoczynku. Spójrzmy na inne zastosowanie naszej funkcji wyszukiwania:

```
// I want to search for a term

search('Ember')

// I want to add a tag

search('Ember', 'JavaScript')

// and a second tag

search('Ember', 'JavaScript', 'Web Development')

// and so on...

search('Ember', 'JavaScript', 'Web Development', 'Code')

// and so on...

search('Ember', 'JavaScript', 'Web Development',

'Code', 'Guides')
```

Jako programiści widzimy jedno wyszukiwane hasło i tyle tagów, ile chcemy. Jak deklarujemy typy do tego w TypeScript? Z tablicami!

```
declare function

search (termin: string, ... tags: string []): Promise <Result []>
```

Wystarczy dodać trzy kropki, aby użyć funkcji wyszukiwania, podczas gdy sposób, w jaki funkcja wyszukiwania działa wewnętrznie, pozostaje dokładnie taki sam. Zwróć uwagę, że pozostałe parametry są zawsze opcjonalne. Więc nawet jeśli term i tags są tego samego typu, chcesz wyodrębnić tyle parametrów, ile potrzebujesz, aby funkcja działała.

Funkcje asynchroniczne

Funkcje zwracające obietnice to funkcje, których możemy używać w kontekście asynchronicznym, co oznacza, że możemy await na ich wynik:

```
const results = await search('Ember') // Yass!
```

Co oznacza, że możemy zadeklarować funkcje asynchroniczne. Użycie słowa kluczowego async wpływa na treść funkcji i implementację. Twoje wartości zwrotne są automatycznie pakowane w typ zwrotu obietnicy:

```
async function search(
query: string,
tags?: string[]
) {
let queryString = `?query=${query}`
if(tags && tags.length) {
queryString += `&tags=${tags.join()}`
}
// Instead of thenable promise calls
// we await results
const response
= await fetch(`/search${queryString}`)
const results = await response.json()
// The return type becomes Promise<Result[]>
return results as Result[]
}
```

Zauważ, że TypeScript obsługuje async najwyższego poziomu. Oznacza to, że dopóki jesteś w kontekście modułu, możesz wywoływać funkcje async tak długo, jak chcesz. Jeden ważny szczegół: kiedy declare asynchroniczny typ funkcji, nie możesz użyć słowa kluczowego async. Powodem jest to, że nagłówek funkcji pozostaje dokładnie taki sam. Zwracamy obietnicę wyników - nic się nie zmieniło w porównaniu z poprzednią wersją.

```
declare function
search(term: string, tags?: string[]): Promise<Result[]>
```

Po zadeklarowaniu takiego typu funkcji można używać tej funkcji w kontekście asynchronicznym.