Lekcja 21: Funkcje generatorów

W JavaScript istnieje specjalny rodzaj funkcji zwany funkcją generatora. Funkcje generatora można opuścić, a następnie ponownie wprowadzić. Chodzi o to, że taka funkcja generuje wartości w czasie, stąd jej nazwa. Brzmi to niewiarygodnie skomplikowanie i, prawdę mówiąc, tak jest. Jednak informacje dotyczące pisania w języku TypeScript mogą bardzo pomóc w ich opracowaniu. A wnioskowanie o typie jest całkiem niezłe! Podczas pracy z generatorami należy pamiętać o dwóch rzeczach.

1. Gwiazdka wokół funkcji generatora informuje, że nie jest to zwykła funkcja.

2. Jest nowe słowo kluczowe: yield. Działa jak przejście, które przekazuje wyniki na zewnątrz, ale także pozwala nam wprowadzić wartości do następnej iteracji. Jeśli kiedykolwiek widziałeś funkcję jako czarne pudełko, potraktuj ustąpienie jako właz, przez który możemy zajrzeć do środka.

Bardzo prosta funkcja generatora może wyglądać następująco:

```
// Nonsensical, but it illustrates the way they work

function *generateStuff() {

yield 1

yield 2

let proceed = yield 3

if(proceed) {

yield 4

}

return 'done'

}

// In use:

const generator = generateStuff()

console.log(generator.next().value) // logs 1

console.log(generator.next().value) // logs 2

console.log(generator.next().value) // logs 3

// The door is open, we pass true through and...

console.log(generator.next(true).value) // logs 4

console.log(generator.next().value) // 'done'
```

yield najpierw otwiera drzwi i zwraca wartość. Otwarte drzwi przyjmują wartość przekazaną w następnej funkcji. Dlatego w kroku czwartym byliśmy w stanie przejść true, ponieważ drzwi były otwarte, gdy yield 3, ale przed uzyskaniem 4. Wpisy do tego są już bardzo skomplikowane, ale TypeScript może już wiele wywnioskować. Jeśli najedziesz kursorem na funkcję, zobaczysz, że TypeScript wywnioskuje następujący typ zwracanej wartości:

```
Generator<1 | 2 | 3 | 4, string, unknown>
```

co oznacza, że podajemy liczby 1, 2, 3 lub 4. W końcu zwrócimy ciąg znaków i nie mamy pojęcia, co wchodzi do środka, gdy otworzymy drzwi. Prawdę mówiąc, ta funkcja generatora prawdopodobnie nie jest najbardziej użyteczna, a typy to odzwierciedlają. W bardziej realistycznym przykładzie zobaczymy, że typy funkcji generatora mogą nam wiele powiedzieć, jeśli zrobimy je dobrze.

Wyszukiwanie z odpytywaniem

W naszym przykładzie chcemy przeprowadzić wyszukiwanie z wykorzystaniem pollingu. Wyobraź sobie zaplecze, które reaguje na określone zapytanie wyszukiwania, a następnie zwraca wyniki w ciągu milisekund. Wyniki nie są bynajmniej kompletne, tylko kilka, które udało się pobrać z bazy danych. Informuje również, czy zapytanie zostało zakończone. Następnie mamy możliwość ponownego zapytania i uzyskania większej liczby wyników. Ciągle sondujemy zaplecze, aby uzyskać więcej. To już brzmi jak typ:

```
type PollingResults = {

results: Result[],

done: boolean

};
```

Tak może wyglądać bardzo uproszczona implementacja funkcji odpytywania. Usuwamy takie rzeczy, jak resetowanie zapytania wyszukiwania lub łączenie wyników z użytkownikiem:

```
async function polling(

term: string

): Promise<PollingResults> {

return fetch(`/pollingSearch?query=${term}`)

.then(res => res.json())

}
```

Powodem, dla którego pobieramy wyniki partiami, jest to, że chcemy je wyświetlać jak najszybciej, dołączając wyniki na bieżąco:

```
function append(result: Result) {

const node = document.createElement('li')

node.innerHTML = `

<a href="${result.url}">${result.title}</a>

`

document.querySelector('#results')?.append(node)

}
```

Tutaj przydaje się funkcja generatora. Stale sondujemy nasze zaplecze i zwracamy wyniki, ale kończymy funkcję tylko wtedy, gdy zaplecze mówi nam, że zostało wykonane.

```
async function *getResults(term: string) {
```

```
let state

do {

// state is a PollingResult

state = await polling(term)

// yield the current result array

yield state.results

} while(!state.done)

// Nothing more to do

}
```

I to jest nasz bardzo prosty, bardzo podstawowy generator. Zauważ, że na razie musieliśmy tylko ustawić typ wartości wejściowej. Jak dotąd wszystkie inne rzeczy zostały wywnioskowane. Podnieśliśmy też poziom gry: pracujemy teraz z generatorami asynchronicznymi. Nie powinno nas to zbytnio martwić, słowo kluczowe asynchroniczne ukrywa większość złożoności. Typ generatora jest następujący:

AsyncGenerator <Result [], void, unknown>

1. Otrzymujemy tablice Result [].

2. Nic nie zwracamy, a więc void.

3. Nic nie przekazujemy. Przez drzwi pojawiają się nieznane wartości.

Teraz do wykorzystania naszego generatora. Każdy generator zwraca iterator, czyli sposób na zapętlenie wartości. Każde wywołanie next zwraca wynik iteratora z możliwą wartością i statusem (jeśli done). Tak więc przeglądanie naszych stale pobieranych wyników wygląda mniej więcej tak:

```
// Adding an event listener, we've been there

document.getElementById('searchField')?.

addEventListener('change', handleChange)

// The actual event handler

async function handleChange(this: HTMLElement, ev: Event) {

if (this instanceof HTMLInputElement) {

// Search for a term,

// call the generator, get an iterator

let resultsGen = getResults(this.value);

let next

do {

// Get the next iterator result
```

```
next = await resultsGen.next()

// The value can be a Result[] or void

// because that's what the generator function

// returns

if(typeof next.value !== 'undefined') {

next.value.map(append)

}

} while(!next.done) // As long as we are not done

}

}
```

W przypadku tej bardzo podstawowej iteracji, w której nie umieszczamy niczego z powrotem przez drzwi yield, możemy użyć pętli for await:

```
async function handleChange(this: HTMLElement, ev: Event) {

if (this instanceof HTMLInputElement) {

let resultsGen = getResults(this.value);

for await(results of resultsGen) {

results.map(append)

}

}

}
```

Dużo jaśniej! Ale zgadnij co? Chcemy włożyć coś z powrotem przez drzwi yield.

Poddając się

Wspaniałą rzeczą w posiadaniu funkcji generatora jest to, że możemy kontrolować wyjście w połowie. Oczywiście możemy poczekać, aż nasz zaplecze będzie kompletne ze wszystkimi wynikami wyszukiwania i wyśle nam flagę done. Lub zapobiegawczo mówimy przestań sondować, jeśli osiągniemy określoną liczbę wyników, które chcemy pokazać. Na szczęście funkcja next pozwala nam przekazywać wyniki. Wywołanie, czy odpytywanie powinno być kontynuowane, byłoby miłe, najlepiej, gdy pokażemy więcej niż pięć wyników. Funkcja handleChange jest dostosowywana szybko poprzez wprowadzenie zmiennej licznika.

```
async function handleChange(this: HTMLElement, ev: Event) {

if (this instanceof HTMLInputElement) {

let resultsGen = getResults(this.value)

let next

+ let count = 0
```

```
do {
- next = await resultsGen.next()
+ next = await resultsGen.next(count >= 5)
if(typeof next.value !== 'undefined') {
next.value.map(append)
+ count += next.value.length
}
} while(!next.done)
}
}
```

TypeScript jest w porządku z tą zmianą, ponieważ wszystko, co możemy przekazać przez drzwi yield, to unknown. Musimy więc uczynić to bardziej konkretnym. Wróćmy do naszej funkcji generatora.

```
async function *getResults(term: string) {
let state
+ let stop
do {
state = await polling(term)
- yield state.results
+ stop = yield state.results
- } while(!state.done)
+ } while(!state.done || stop)
}
```

To działa, ale teraz zdarzył się najgorszy przypadek: przeszliśmy od unknown (dobrego) do any (bardzo zły). Bądźmy bardziej konkretni. Niech TypeScript wywnioskuje to przez przypisanie wartości domyślnej:

```
async function *getResults(term: string) {
let state
+ let stop = false
do {
state = await polling(term)
stop = yield state.results
} while(!state.done || stop)
```

}

To już zmienia typ na

AsyncGenerator <Result [], void, boolean>

A to gwarantuje, że przekazujemy poprawne typy. Lub jesteśmy bardzo dokładni co do naszego typu zwracanego:

async function *getResults(

term: string

): AsyncGenerator<Result[], void, boolean> {

let state, stop

do {

state = await polling(term)

stop = yield state.results

// from here on, stop is boolean

} while(state.done && stop)

}

Jak to często bywa, wybór polega na mimowolnym posuwaniu się naprzód podczas kodowania lub bardzo wyraźnym określeniu kontraktów poprzez zdefiniowanie głów funkcji tak konkretnie, jak to tylko możliwe.

Podsumowanie

Funkcje w JavaScript są duże. Tak też jest w TypeScript. Dowiedzieliśmy się wiele o funkcjach:

1. Najpierw dowiedzieliśmy się o typach funkcji, typach zwracanych i typach parametrów.

2. Zagłębiliśmy się w wywołania zwrotne, koncepcję w JavaScript, która się pojawia wszędzie. Dowiedzieliśmy się, że funkcje mają swoje własne typy i że kolejność argumentów jest ważna, a nie ich nazwy.

3. Poznaliśmy pojęcie substytucyjności. Funkcje mogą mieć inny kształt niż ich typy, jeśli pozwala na to kontekst.

4. Czasami pisząc JavaScript, mam ochotę krzyczeć: „To śmieszne!” ale nigdy nie wiem, do czego to się odnosi. Cóż, dzięki Bogu, TypeScript może nam w tym pomóc! te typy argumentów pomagają nam zapobiegać błędom i uzyskać więcej informacji o obiekcie, z którym łączymy naszą funkcję.

5. Dowiedzieliśmy się, jak TypeScript wnioskuje o typy zwracane przez funkcję asynchroniczną i działa z parametrami reszt.

6. Zauważyliśmy również, że język TypeScript wymaga specjalnych głowic funkcyjnych dla oznaczonych literałów szablonów.

7. Przeciążenia funkcji pomagają nam zdefiniować wiele typów funkcji dla jednej funkcji, co jest zgodne z elastycznością funkcji JavaScript, ale także uwidacznia złożoność bardzo elastycznych funkcji.

8. Wreszcie, sięgnęliśmy po specjalny rodzaj funkcji: funkcje z gwiazdką, funkcje generatora!

Przyjrzeliśmy się również bardziej zaawansowanym koncepcjom, takim jak typy związków, typy ogólne i wiele więcej! Rzeczy do rozwikłania w przyszłych lekcjach.