Lekcja 47: Silnik DOM JSX, część 2

To, co stworzyliśmy w poprzedniej lekcji, jest już bardzo potężne. Możemy zajść naprawdę daleko i mieć fajne API do pisania w DOM bez żadnej biblioteki czy frameworka! Ale my, ludzie z TypeScript, tęsknimy za jedną ważną rzeczą: poprawnym pisaniem. Wnioskowanie o typie wiele dla nas robi, ale przy wyłączonej opcji noImplicitAny tracimy wiele ważnych informacji. Włączając go ponownie, wszędzie widzimy czerwone zawijasy. Bądźmy wierni sobie i stwórzmy odpowiednie typy dla naszych funkcji. Dodatkową korzyścią jest bardzo dobre wnioskowanie o typie!

Typowanie Factory

Zacznijmy od trzech funkcji, które mamy. TypeScript w tym momencie naprawdę narzeka głównie na niejawne anys. Więc lepiej dorzućmy kilka konkretnych typów. Dla zwięzłości skupimy się tylko na głowicach funkcji. Funkcja pomocnicza nonNull jest łatwa do wpisania. Bierzemy dwa typy generyczne, które możemy powiązać, gdy używamy funkcji.

function nonNull<T, U>(val: T, fallback: U) {

return Boolean(val) ? val : fallback;

}

Typ zwracany jest wywnioskowany jako T | U.

Następnie pracujemy nad DOMparseChildren, ponieważ ma on najprostszy zestaw argumentów. Istnieje tylko kilka typów, które mogą być elementami potomnymi naszego drzewa DOM:

1. HTMLElement, podstawowa klasa wszystkich elementów HTML.

2. string, jeśli przekażemy zwykły ciąg, który powinien zostać zamieniony na węzeł tekstowy.

3. Text, jeśli utworzyliśmy węzeł tekstowy na zewnątrz, który chcemy dołączyć.

Tworzymy związek pomocnika PossibleChildren i używamy go jako argumentu DOMparseChildren.

type PossibleChildren =

HTMLElement | Text | string

function DOMparseChildren(

children: PossibleChildren[]

) {

// ...

}

Zwracany typ jest poprawnie wywnioskowany jako HTMLElement | Text, ponieważ pozbywamy się ciągów i konwertujemy je na Text. Następną funkcją jest DOMparseNode, ponieważ ma taką samą sygnaturę jak DOMcreateElement. Przyjrzyjmy się możliwym argumentom wejściowym.

1. element może być ciągiem znaków lub funkcją. Chcemy użyć generycznego, aby powiązać konkretną wartość elementu z typem.

2. properties mogą być argumentami funkcji komponentu lub zbiorem właściwości odpowiedniego elementu HTML.

3. children to zbiór możliwych dzieci. Mamy już na to typ.

Aby to działało poprawnie, potrzebujemy kilku typów pomocników. Zabawa to znacznie luźniejsza interpretacja funkcji. Potrzebujemy tego, aby wywnioskować parametry.

type Fun = (...args: any[]) => any;

Musimy wiedzieć, który element HTML jest tworzony, gdy przekazujemy określony ciąg. TypeScript udostępnia interfejs o nazwie HTMLElementTagNameMap. Jest to tak zwana mapa typów, co oznacza, że jest to lista klucz-wartość identyfikatorów (tagów) i odpowiadających im typów (podtypy HTMLElement). Listę tę można znaleźć w witrynie ib.dom.ts.

interface HTMLElementTagNameMap {

"a": HTMLAnchorElement;

"abbr": HTMLElement;

"address": HTMLElement;

"applet": HTMLAppletElement;

// and so on ...

}

Chcemy stworzyć typ, CreatedElement, który zwraca element zgodnie z przekazanym przez nas ciągiem. Jeśli ten element nie istnieje, zwracamy HTMLElement, typ podstawowy.

type AllElementsKeys = keyof HTMLElementTagNameMap

type CreatedElement<T> =

T extends AllElementsKeys ? HTMLElementTagNameMap[T] :

HTMLElement

Używamy tego typu pomocnika do prawidłowego definiowania rekwizytów. Jeśli przekazujemy funkcję, chcemy uzyskać parametry tej funkcji. Jeśli przekażemy ciąg, możliwe właściwości są częścią odpowiedniego elementu HTML. Bez częściowej musielibyśmy zdefiniować wszystkie właściwości. A jest ich dużo!

type Props<T> =

T extends Fun ? Parameters<T>[0] :

T extends string ? Partial<CreatedElement<T>> :

never;

Zauważ, że indeksujemy pierwszy parametr Parameters. Dzieje się tak, ponieważ funkcja JSX ma tylko jeden argument - właściwości. Musimy dokonać destrukcji z krotki na rzeczywisty typ. To wszystko, czego teraz potrzebujemy. Przejdźmy do DOMparseNode. Ta funkcja działa tylko wtedy, gdy podajemy ciągi znaków.

function DOMparseNode<

T extends string

```
>(
element: T,
properties: Props<T>,
children: PossibleElements[]
)
```

Funkcja poprawnie wnioskuje właściwości HTMLElement jako typ zwracany. Wreszcie funkcja DOMcreateElement. Może to być trochę trudne, ponieważ rozdzielenie między właściwościami funkcji i właściwościami elementu HTML nie jest tak łatwe, jak na pierwszy rzut oka. Naszą najlepszą opcją są przeciążenia funkcji, ponieważ mamy tylko dwa warianty. Ponadto typ Rekwizyty pomaga nam w poprawnym połączeniu między rodzajem elementu a odpowiednimi właściwościami.

```
function DOMcreateElement<
T extends string
>(
element: T, properties: Props<T>,
...children: PossibleElements[]
): HTMLElement
function DOMcreateElement<
F extends Fun
>(
element: F, properties: Props<F>,
...children: PossibleElements[]
): HTMLElement
function DOMcreateElement(
element: any, properties: any,
...children: PossibleElements[]
): HTMLElement {
// ...
}
```

Teraz możemy bezpośrednio korzystać z funkcji fabrycznej!

Wpisując JSX

Nadal otrzymujemy błędy typu, gdy używamy JSX zamiast funkcji fabrycznych. Dzieje się tak, ponieważ TypeScript ma własny parser JSX i chce wcześnie wychwycić problemy JSX. W tej chwili TypeScript nie wie, których elementów się spodziewać. Dlatego domyślnie jest to dowolny dla każdego elementu. Aby to zmienić, musimy rozszerzyć własną przestrzeń nazw JSX TypeScript i zdefiniować typ zwracania

tworzonych elementów. Przestrzenie nazw są sposobem w języku TypeScript do organizowania kodu. Zostały stworzone w czasach przed modułami ECMAScript i dlatego nie są już tak często używane. Mimo to podczas definiowania wewnętrznych interfejsów, które powinny być grupowane, przestrzeń nazw jest drogą do zrobienia.

Podobnie jak w przypadku interfejsów, przestrzenie nazw umożliwiają scalanie deklaracji. Otwieramy przestrzenie nazw JSX i definiujemy dwie rzeczy:

1. Element zwrotny, który rozszerza interfejs Element.

2. Wszystkie dostępne elementy HTML, więc TypeScript może dać nam autouzupełnianie w JSX. Są zdefiniowane w IntrinsicElements. Używamy zmapowanego typu, w którym kopiujemy HTMLElementTagNameMap, aby być mapą częściowych. Następnie rozszerzamy z niego IntrinsicElements. Chcemy użyć interfejsu zamiast typu, ponieważ chcemy, aby scalanie deklaracji było otwarte.

```
// Open the namespace

declare namespace JSX {

// Our helper type, a mapped type

type OurIntrinsicElements = {

[P in keyof HTMLElementTagNameMap]:

Partial<HTMLElementTagNameMap[P]>

}

// Keep it open for declaration merging

interface IntrinsicElements

extends OurIntrinsicElements {}

// JSX returns HTML elements. Keep this also

// open for declaration merging

interface Element extends HTMLElement {}

}
```

Dzięki temu otrzymujemy autouzupełnianie dla elementów HTML i komponentów funkcji. A nasz mały silnik DOM JSX oparty na TypeScript jest gotowy na najlepszy czas!