Lekcja 44: JSONify

Po zobaczeniu mocy i potencjału systemu czcionek, w tym momencie zostałem fanem TypeScript. JSON.parse i JSON.stringify to przydatne funkcje do serializacji i deserializacji obiektów JavaScript. JSON to podzbiór obiektów JavaScript, brakująyh tylko funkcji i undefined. Ten podzbiór sprawia, że analizowanie ciągów JSON jest nawet szybsze niż analizowanie zwykłych obiektów. Klasa, która serializuje i deserializuje obiekty JavaScript, które mogą być powiązane z różnymi typami, może wyglądać następująco:

```
class Serializer<T> {

serialize(inp: T): string {

return JSON.stringify(inp)

}

deserialize(inp: string): JSONified<T> {

return JSON.parse(inp)

}

}
```

Z JSONified do zdefiniowania. Zadeklarujmy typ, który obejmuje wszystkie możliwe sposoby zapisywania wartości w JavaScript: liczby, łańcuchy, wartości logiczne, funkcje. Zagnieżdżone i w tablicach. I typ, który ma funkcję toJSON. Jeśli obiekt ma funkcję toJSON, JSON.stringify użyje obiektu zwróconego z toJSON do serializacji, a nie rzeczywistych właściwości.

```
// toJSON returns this object for

// serialization, no matter which other

// properties this type has.

type Widget = {

toJSON(): {

kind: “Widget”, date: Date

}

}

type Item = {

// Regular primitive types

text: string;

count: number;

// Options get preserved

choice: “yes” | “no” | null;

// Functions get dropped.
```

```
func: () => void;

// Nested elements need to be parsed

// as well

nested: {

isSaved: boolean;

data: [1, undefined, 2];

}

// A pointer to another type

widget: Widget;

// The same object referenced again

children?: Item[];

}
```

Istnieje różnica między obiektami JSON i JavaScript i możemy modelować tę różnicę za pomocą zaledwie kilku wierszy typów warunkowych. Zaimplementujmy JSONified <T>.

Wartości JSONified

Najpierw tworzymy typ JSONified i robimy jedną konkretną kontrolę: czy to jest obiekt z funkcją toJSON? Jeśli tak, wywnioskujemy typ zwrotu i używamy go. W przeciwnym razie sam obiekt JSONify. toJSON również zwraca obiekt, więc przekazujemy go również do naszego następnego kroku.

```
type JSONified<T> =

JSONifiedValue<

T extends { toJSON(): infer U } ? U : T

>;
```

Następnie przyjrzyjmy się faktycznym wartościom i temu, co się stanie, gdy je serializujemy. Typy pierwotne można łatwo przenosić. Funkcje powinny zostać usunięte. Tablice i obiekty zagnieżdżone powinny być obsługiwane osobno. Umieśćmy to w typie:

```
type JSONifiedValue<T> =

T extends string | number | boolean | null ? T :

T extends Function ? never :

T extends object ? JSONifiedObject<T> :

T extends Array<infer U> ? JSONifiedArray<U> :

never;
```

JSONifiedObject to mapowany typ, w którym przeglądamy wszystkie właściwości i ponownie stosujemy JSONified.

```
type JSONifiedObject<T> = {

[P in keyof T]: JSONified<T[P]>

}
```

Jest to pierwsza rekurencyjna sytuacja. TypeScript zezwala na pewien poziom rekurencji, o ile nie ma odwołań cyklicznych. Wywołanie JSONified ponownie działa w dół drzewa. Podobnie jest z tablicą JSONifiedArray, która staje się tablicą wartości JSONified. Jeśli istnieje niezdefiniowany element, JSON.stringify zamapuje go na null. Dlatego potrzebujemy innego typu pomocnika.

```
type UndefinedAsNull<T> =

T extends undefined ? null : T;

type JSONifiedArray<T> =

Array<UndefinedAsNull<JSONified<T>>>
```

I to wszystko, czego potrzebujemy. W kilku wierszach kodu opisaliśmy całe zachowanie JSON.parse po JSON. stringify. Nie tylko na poziomie pisma, ale także ładnie opakowane w klasę:

```
const itemSerializer = new Serializer<Item>()

const serialization =

itemSerializer.serialize(anItem)

const obj =

itemSerializer.deserializer(serialization)
```

Zachęcam do wypróbowania tego na placu zabaw lub we własnej aplikacji. Niezwykłe jest zobaczenie, jak działają typy rekurencyjne i jak głęboko mogą się one zagłębić w zagnieżdżonej strukturze obiektów