Lekcja 45: Definicje usług

To jest drugi przykład, który sprawił, że pokochałem Type-Script i uznaję ogromną moc, jaka tkwi w systemie czcionek.

Definicje dynamiczne

Chcemy zapewnić naszym współpracownikom funkcję pomocniczą, aby mogli definiować usługę i jej metody usług za pomocą obiektu JavaScript, który wygląda trochę jak typ; na przykład definicja usługi do otwierania, zamykania i zapisywania w plikach.

```
const serviceDefinition = {

open: { filename: String },

insert: { pos: Number, text: String },

delete: { pos: Number, len: Number },

close: {},

};
```

Obiekt definicji usługi opisuje listę definicji metod. Każda właściwość definicji usługi jest metodą. Każda definicja metody definiuje ładunek i towarzyszący mu typ. Używamy tutaj dużej liczby Number i dużego String, funkcji konstruktora w JavaScript, które tworzą wartości odpowiednio typu liczba lub łańcuch. To nie są typy TypeScript! Ale wyglądają strasznie podobnie. Celem jest posiadanie funkcji createService, w której przekazujemy dwie rzeczy:

1. Definicja usługi w formacie, który opisaliśmy powyżej.

2.Osoba obsługująca wnioski. Jest to funkcja, która odbiera komunikaty i ładunki i jest implementowana przez użytkowników tej funkcji; na przykład dla powyższej definicji usługi spodziewamy się, że zostanie odebrany komunikat open z nazwą pliku playload, gdzie nazwą pliku jest ciąg. W zamian otrzymujemy obiekt usługi. Ten obiekt usługi udostępnia metody (otwieranie, wstawianie, usuwanie i zamykanie, zgodnie z definicją usługi), które pozwalają na określony ładunek (zdefiniowany w definicji usługi). Po wywołaniu metody konfigurujemy obiekt żądania, który jest obsługiwany przez moduł obsługi żądania.

Typowanie definicji usługi

Jak zawsze, zanim coś zaimplementujemy, popracujmy nad nagłowkiem funkcji createService. Zgodnie z powyższą specyfikacją, definicja zwykłej funkcji wygląda następująco:

```
declare function createService<

S extends ServiceDefinition

>(

serviceDef: S,

handler: RequestHandler<S>,

): ServiceObject<S>
```

Zauważ, że potrzebujemy tylko jednej powiązanej zmiennej typu ogólnego: definicji usługi. Zdefiniujmy typy definicji usług.

```
// A service definition has keys we don't know

// yet and lots of method definitions

type ServiceDefinition = {

[x: string]: MethodDefinition;

};

// This is the payload of each method:

// a key we don't know, and either a string or

// a number constructor (the capital String, Number)

type MethodDefinition = {

[x: string]:

StringConstructor | NumberConstructor;

};
```

Pozwala to na obiekty z każdym możliwym kluczem ciągu. W momencie, gdy powiążemy konkretną definicję usługi ze zmienną ogólną, klucze zostaną zdefiniowane i pracujemy z typem zawężonym. Następnie pracujemy nad drugim argumentem, modułem obsługi żądań. Procedura obsługi żądania to funkcja z jednym argumentem, obiektem żądania. Zwraca wartość logiczną, jeśli wykonanie się powiodło.

```
type RequestHandler<

T extends ServiceDefinition

> = (req: RequestObject<T>) => boolean;
```

Obiekt Request

Obiekt żądania jest zdefiniowany przez przekazaną przez nas definicję usługi. Jest to obiekt, w którym każdy klucz definicji usługi staje się komunikatem. Obiekt po kluczu definicji usługi staje się ładunkiem.

```
type RequestObject<T extends ServiceDefinition> = {

[P in keyof T]: {

message: P;

payload: RequestPayload<T[P]>;

}

}[keyof T];
```

Z typem dostępu indeksu do keyof T, tworzymy unię z obiektu, który zawierałby każdy klucz. Ładunek żądania jest definiowany przez obiekt każdego klucza w definicji usługi:

```
type RequestPayload<T extends MethodDefinition> =
```

```
// Is it an empty object?
{} extends T ?
// Then we don't have a payload
undefined :
// Otherwise we have the same keys, and get the
// type from the constructor function
{ [P in keyof T]: TypeFromConstructor<T[P]> };
type TypeFromConstructor<T> =
T extends StringConstructor ?
string :
T extends NumberConstructor ? number : any;
```

Dla opisanej wcześniej definicji usługi wygenerowany typ wygląda następująco:

```
{
req: {
message: “open”;
payload: {
filename: string;
};
} | {
message: “insert”;
payload: {
pos: number;
text: string;
};
} | {
message: “delete”;
payload: {
pos: number;
len: number;
};
} | {
```

message: “close”;

payload: undefined;

}

}

Obiekt usługi

Na koniec wpisujemy obiekt usługi, wartość zwracaną. Dla każdego wpisu w definicji usługi tworzy pewne metody usług.

type ServiceObject<T extends ServiceDefinition> = {

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

};

Każda metoda usługi pobiera ładunek zdefiniowany w obiekcie po każdym kluczu w definicji usługi.

type ServiceMethod<T extends MethodDefinition> =

// The empty object?

{} extends T ?

// No arguments!

() => boolean :

// Otherwise, it's the payload we already

// defined

(payload: RequestPayload<T>) => boolean;

To wszystko, czego potrzebujemy!

Wdrażanie createService

Teraz zaimplementujmy samą funkcję. Tworzymy pusty obiekt i dodajemy do niego nowe klucze. Każdy klucz jest metodą, która pobiera ładunek i wywołuje procedurę obsługi.

function createService<S extends ServiceDefinition>(

serviceDef: S,

handler: RequestHandler<S>,

): ServiceObject<S> {

const service: Record<string, Function> = {};

for (const name in serviceDef) {

service[name] =

(payload: any) => handler({

message: name,

```
payload
});
}
return service as ServiceObject<S>;
}
```

Zauważ, że na końcu pozwalamy sobie na małe rzutowanie typów, aby upewnić się, że bardzo ogólne tworzenie nowego obiektu jest bezpieczne dla typów. Oto jak wygląda nasza definicja usługi w akcji:

```
const service = createService(
serviceDefinition,
req => {
// req is now perfectly typed and we know
// which messages we are able to get
switch (req.message) {
case 'open':
// Do something
break;
case 'insert':
// Do something
break;
default:
// Due to control flow anaysis, this
// message now can only be close or
// delete.
// We can narrow this down until we
// reach never
break;
}
return true;
}
);
// We get full autocomplete for all available
```

```
// methods, and know which payload to

// expect

service.open({ filename: 'text.txt' });

// Even if we don't have a payload

service.close();
```

Napisaliśmy około 25 wierszy definicji typów i kilka szczegółów implementacji, a nasi koledzy będą mogli pisać niestandardowe usługi, które są całkowicie bezpieczne dla typów. Weź inną definicję usługi i baw się dobrze!