08

Facilitando a composição de funções

Transcrição

Podemos simplificar a composição de funções criando uma função utilitária que se encarregará de compor as funções para nós. Antes de partirmos para sua implementação, vamos vislumbrar seu uso em nosso código:

```
// app/nota/service.js

// exemplo de uso, não entra ainda em nossa aplicação

sumItems(code) {

    const filterItems = partialize(filterItemsByCode, code);
    const sumItems = compose(sumItemsValue, filterItems, getItemsFromNotas);

    return this.listAll().then(sumItems);
}
```

Que tal? Muito mais enxuto e legível do que a versão anterior.

Point-free style

Um olhar atento também reparará que não precisamos, durante a composição, referenciar qualquer parâmetro, diferente da abordagem anterior na qual o parâmetro `notas` era referenciado. Essa estética do nosso código possui um nome, **point-free style**.

Agora que já sabemos até onde queremos chegar, vamos implementar a função `compose` no novo módulo `app/utils/operators.js`.

Ela receberá como parâmetro um número indeterminado de funções e retornará uma função que terá apenas um argumento. No contexto do problema que estamos resolvendo, essa função representará a extinta função `sumItemsWithCode`. Esta função, quando chamada, passará seu parâmetro para a última função da lista de funções passada para `compose` e seu resultado será passado para a função anterior sucessivamente até chegarmos ao resultado final.

Vamos ao código da função `compose`:

```
// app/utils/operators.js

// código anterior omitido
export const compose = (...fns) => value =>
    fns.reduceRight((previousValue, fn) =>
        fn(previousValue), value);
```

Como queremos reduzir a lista de funções a um valor apenas no final, nada mais justo do que usarmos a função `reduce`. No entanto, como queremos aplicar as funções da direita para a esquerda, usamos `reduceRight`. Quando usamos `reduce` ou `reduceRight`, podemos indicar qual será o valor inicial utilizado na redução, no caso, usamos `value` que será o array recebido por `sumItemsWithCode`. Esse valor será o `previousValue` na primeira chamada de `fn(previousValue)`. Em nosso

contexto, na primeira iteração de `reduceRight`, `fn` será a função `getItemsFromNotas`. Seu resultado será o `previousValue` passado para a função anterior e assim sucessivamente.

Agora que temos nossa função pronta, vamos importá-la e utilizá-la em `app/app.js`:

```
// app/nota/service.js

import { handleStatus } from '../utils/promise-helpers.js';
// importou compose
import { partialize, compose } from '../utils/operators.js';

const API = `http://localhost:3000/notas`;

const getItemsFromNotas = notas => notas.$flatMap(nota => nota.itens);
const filterItemsByCode = (code, items) => items.filter(item => item.codigo === code);
const sumItemsValue = items => items.reduce((total, item) => total + item.valor, 0);

export const notasService = {

    listAll() {
        return fetch(API)
        .then(handleStatus)
        .catch(err => {
            console.log(err);
            return Promise.reject('Não foi possível obter as notas fiscais');
        });
    },

    sumItems(code) {

        // realizando a composição
        const sumItems = compose(
            sumItemsValue,
            partialize(filterItemsByCode, code),
            getItemsFromNotas
        );

        return this.listAll().then(sumItems);
    }
};
```

Ótimo! Alteramos a estrutura do nosso código em prol da legibilidade e manutenção sem alterar seu comportamento. Podemos fazer mais por ele, assunto da próxima seção.