04

Partial application

Transcrição

Para que possamos compreender mais facilmente a *partial application*, inclusive sua implementação, trabalharemos com um problema de escopo menor para então transpor a solução para o problema que estamos tentando resolver.

Temos a função `ehDivisivel` que recebe dois parâmetros. O primeiro é o divisor e o segundo o número que será testado. Verificaremos se três números são divisíveis por dois:

```
// exemplo apenas, não entra na aplicação
const ehDivisivel = (divisor, numero) => !(numero % divisor);
ehDivisivel(2, 10); // true
ehDivisivel(2, 15); // false
ehDivisivel(2, 20); // true
```

Todavia, com *partial application* podemos memorizar o divisor evitando de passá-lo como argumento para a função toda vez que for utilizada. Conseguimos isso facilmente através da função `bind` :

A função bind

Podemos realizar a *partial application* de funções facilmente através da função `Function.bind()` :

```
// exemplo apenas, não entra na aplicação
const ehDivisivel = (divisor, numero) => !(numero % divisor);
// criou uma função parcial
const ehDivisivelPorDois = ehDivisivel.bind(null, 2);
ehDivisivelPorDois(10); // true
ehDivisivelPorDois(15); // false
ehDivisivelPorDois(20); // true
```

A função `Function.bind` cria uma nova função. Seu primeiro argumento é o valor de `this` que desejamos que a nova função utilize como contexto. Porém, como declaramos a função através de *arrow function* que não aceita a modificação do seu contexto, simplesmente passamos `null` . Mesmo que tivéssemos passado outro valor ele seria ignorado.

Os demais parâmetros são todos aqueles que desejamos assumir como argumentos já fornecidos toda vez que a função resultante de `bind()` for chamada. No caso, estamos indicando que o primeiro parâmetro será sempre `2` .

Podemos passar quantos parâmetros desejarmos. Vejamos outro exemplo:

```
// exemplo apenas, não entra na aplicação
const ehDivisivel = (divisor, numero) => !(numero % divisor);
// assume como parâmetros 2 e 5, nesta ordem
const fn = ehDivisivel.bind(null, 2, 5);
fn(); // false
```

Agora já podemos aplicar a solução que vimos com a função `filterItemsByCode` .Todavia, essa aplicação deve ser feita dentro do método `sumItens` do nosso serviço, porque é nele que temos acesso ao código:

```
// app/nota/service.js
// código anterior omitido

// função continua com a estrutura original
const filterItemsByCode = (code, items) => items.filter(item => item.codigo
=== code);

// código anterior omitido

sumItems(code) {
    // criando uma função parcial
    const filterItems = filterItemsByCode.bind(null, code);
    // ainda falta realizar a composição dessa função com as
    // outras para criar novamente sumItems, que não existe
    return this.listAll().then(sumItems(code));
}
// código posterior omitido
```

Excelente, mas pode ficar ainda melhor! Que tal isolarmos a lógica que realiza a aplicação parcial em uma função utilitária para que possamos utilizar sempre que necessário? Vamos criar a função **partialize** no **novo** módulo `app/utils/operators.js` :

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

export const partialize = (fn, ...params) =>
    fn.bind(null, ...params);
```

Utilizando a função em `app/nota/service.js` , que ainda continuará incompleto.

```
import { handleStatus } from '../utils/promise-helpers.js';
import { partialize } 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) {
```

```
            const filterItems = partialize(filterItemsByCode, code);
            // ainda falta realizar a composição dessa função com as
            // outras para criar novamente sumItems, que não existe
            return this.listAll().then(sumItems(code));
        }
    };
```

Agora que já resolvemos a questão dos parâmetros da função `filterItemsByCode`, podemos partir para realização da composição das funções que criamos.