01

Separando responsabilidades

Transcrição

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/772-js-padrao-funcional/stages/03-project.zip) completo do projeto anterior e continuar seus estudos a partir daqui.

Vamos voltar nossa atenção para a função `sumItems`:

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

// código anterior omitido
const sumItems = code => notas =>
    notas.$flatMap(nota => nota.itens)
    .filter(item => item.codigo == code)
    .reduce((total, item) => total + item.valor, 0)
// código posterior omitido
```

Não precisamos meditar muito para chegarmos a conclusão que a função faz muita coisa. Ela:

1. Obtém uma lista de itens de uma lista de notas fiscais
2. Filtra a lista de itens por um código
3. Totaliza o valor dos itens

Para que nosso código fique ainda mais claro e mais fácil de mantar, vamos extrair cada responsabilidade que acabamos de lista em sua respectiva função:

```
// app/app.js

// código anterior omitido

// não existe mais a função `sumItemsWithCode`. Ela foi substituída por três funções
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);
// código posterior omitido
```

Agora que temos três funções com responsabilidades bem definidas, precisamos combiná-las para criarmos a extinta função `sumItem`. Em outras palavras, queremos realizar a **composição de funções**.

Na matemática, composição de funções consiste na aplicação de uma função ao resultado de outra função, sucessivamente. Todavia, nossas funções precisam receber um parâmetro apenas e para atrapalhar um pouco nossa tarefa, a função `filterItemsByCode` recebe duas. E agora?

Transformando uma função que recebe dois parâmetro para um parâmetro apenas

Podemos fazer com que a função `filterItemsByCode` recebe um parâmetro apenas, no caso, o código que será utilizado durante o filtro dos itens. Seu retorno será uma função que receberá a lista de notas fiscais que precisa operar. Esta última função mesmo retornada lembrará do parâmetro do código recebido pela primeira função através de closure:

```
// exemplo apenas, não entra em nosso código

const getItemsFromNotas = notas => notas.$flatMap(nota => nota.itens);
// alterando a função
const filterItemsByCode = code => items => items.filter(item => item.codigo === code);
const sumItemsValue = items => items.reduce((total, item) => total + item.valor, 0);
// código posterior omitido
```

Simulando o uso da função, ela funcionará dessa forma:

```
// exemplo apenas, não entra em nosso código
const filterItems = filterItemsByCode('2143');
// função pronta para filtrar itens
```

Excelente, mas não utilizaremos essa solução. O motivo? Se tivéssemos outros lugares da nossa aplicação que fizesse uso de `filterItemsByCode` eles quebrariam! A ideia é criarmos uma nova função baseada em `filterItemsByCode` que já aplique **parcialmente** o primeiro parâmetro da função, no caso, o código. Com essa abordagem, a função receberá apenas um parâmetro, no caso, a lista de notas fiscais. No fim, o que queremos é realizar **partial application** com a função `filterItemsByCode` .