06

Implementando a função flatMap

Transcrição

Criamos um mecanismo para realizar o log entre chamadas encadeadas à `then` centralizando o código em um único lugar. Porém, nosso código ainda deixa a desejar.

Vamos voltar nosso olhos para a seguinte chamada:

```
// código anterior omitido
.then(notas => notas.reduce((array, nota) => array.concat(nota.itens), []))
// código posterior omitido
```

Apesar de funcionar, essa chamada não deixa clara nossa intenção. Nosso código pode ficar ainda melhor simplesmente pelo fato do `Array` ser um **functor** em JavaScript.

Functor

No jargão da programação funcional um *Functor* é simplesmente algo mapeável, ou seja, que suporta a operação `map`. Vimos que a função `map` do Functor `Array` não resolve nosso problema pois retorna a cada iteração um dado multidimensional. No entanto, existe um `map` especial chamado `flatMap` garante que os dados tenham uma dimensão apenas.

A palavra `flat` remete à plano, única dimensão. A má notícia é que o ECMASCRIPT ainda não adicionou o suporte à função `flatMap`, apesar ter sido proposta. Nesse sentido, que tal criarmos nossa própria implementação de `flatMap` e adicioná-la no functor Array?

Implementando a função flatMap

Queremos que a função funcione da seguinte maneira

O código

```
notas.reduce((array, nota) => array.concat(nota.itens), [])
```

Se tornará:

```
notas.$flatMap(notas => notas.itens)
```

Adicionaremos a função `$flatMap` no `prototype` de `Array`, permitindo assim que todos os arrays criados em nossa aplicação tenham acesso à função. Vamos criá-la no módulo `app/utils/array-helpers.js`:

```
// só adicionada no prototype se não existir
if(!Array.prototype.$flatMap) {
```

```
    Array.prototype.$flatMap = function(cb) {
        return this.map(cb).reduce((destArray, array) =>
            destArray.concat(array), []);
    }
}
```

Tivemos que usarmos uma `function` no lugar de uma `arrow function` pois precisamos que o `this` seja dinâmico, referênciado a instância do array que esta executando a função em dado momento. Ela recebe como parâmetro a lógica do `map` que será empregada. Em nosso caso, passaremos a lógica `notas => notas.itens` porque queremos um array multidimensional de itens. Por fim, o `reduce` garantirá que o array terá uma dimensão apenas concatenando cada item do array em um novo array.

Um ponto curioso é que nosso módulo não exportará nada. Mas como assim? Queremos apenas que nosso módulo seja carregado para que a operação de adição da função `$flatMap` seja executada.

Vamos voltar para `app/app.js` e importar o módulo da seguinte maneira:

```
// app/app.js
import { handleStatus, log } from './utils/promise-helpers.js';
// novo import!
import './utils/array-helpers.js';
```

Reparem que apenas importamos o módulo passando seu caminho.

- A biblioteca RxJS utiliza essa estratégia para adicionar no prototype do Observer os `operators` que carregados.

Agora podemos modificar nosso código que ficará assim:

```
app/app.js
import { handleStatus, log } from './utils/promise-helpers.js';
import './utils/array-helpers.js';

document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notas')
    .then(handleStatus)
    .then(notas => notas.$flatMap(nota => nota.itens))
    .then(log)
    .then(itens => itens.filter(item => item.codigo == '2143'))
    .then(log)
    .then(itens => itens.reduce((total, item) => total + item.valor, 0))
    .then(console.log)
    .catch(console.log);
```

Como já temos uma visão geral do código que escrevemos, não é mais necessário as chamadas `.then(log)` . Vamos removê-las:

```
// app/app.js
```

```
import { handleStatus, log } from './utils/promise-helpers.js';
import './utils/array-helpers.js';

document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notas')
    .then(handleStatus)

    .then(notas => notas.$flatMap(nota => nota.itens))
    .then(itens => itens.filter(item => item.codigo == '2143'))
    .then(itens => itens.reduce((total, item) => total + item.valor, 0))
    .then(console.log)
    .catch(console.log);
```

Como não estamos mais verificando a passagem de dados de um `then` para outro, podemos rescrever nosso código da seguinte maneira:

```
// app/app.js
import { handleStatus, log } from './utils/promise-helpers.js';
import './utils/array-helpers.js';

document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notas')
    .then(handleStatus)
    .then(notas => notas
        .$flatMap(nota => nota.itens)
        .filter(item => item.codigo == '2143')
        .reduce((total, item) => total + item.valor, 0))
    .then(console.log)
    .catch(console.log);
```

Ainda mais enxuto, não? Todavia nosso código ainda deixa a desejar e veremos a razão disso no próximo vídeo.