08

# Implementando retry em Promises

### Transcrição

A função `retry` receberá uma função que ao ser chamada, deve retornar uma nova Promise com a operação que desejamos realizar, o número de tentativas e o intervalo de tempo entre essas tentativas.

Vamos criar seu esqueleto em `app/utils/promise-helpers.js` :

```
// app/utils/promise-helpers.js
// código anterior omitido

export const retry = (retries, milliseconds, fn) => // falta implementar
```

A primeira coisa que faremos é chamar a função fn e programar uma resposta no caso de sua rejeição, isto é, caso algum erro aconteça durante sua execução. Sabemos que é através da função `catch` que lidamos com exceções.

Como precisaremos repetir a operação até atingirmos o limite definido pelo parâmetro `retries` , faz bastante sentido lançarmos mão de recursão. Em outras palavras, nossa função `retry` será uma função recursiva, que nada mais é do que uma função que chama a si mesma até que um *leave event* ocorra, caso contrário cairemos em uma recursão infinita que travará nossa aplicação:

```
// app/utils/promise-helpers.js
// código anterior omitido

export const retry = (retries, milliseconds, fn) =>

    fn().catch(err => {
        console.log(retries);
        return retries > 1
            ? retry(retries - 1, milliseconds, fn)
            : Promise.reject(err));
    });
```

Na cláusula `catch` , através de um `if` ternário, testamos se o número de tentativas ainda é maior do que um (esta certo, porque já gastamos uma tentativa na chamada da promise), se for, temos direito a mais uma tentativa e chamamos recursivamente retry(fn, retries - 1, time). Se o número máximo de tentativas for excedido, retornamos uma rejeição com Promise.reject(err) que recebe a causa do último erro.

O que as chamadas recursivas farão é encadear uma sucessão de chamadas artificiais à `then()` , repetindo a operação. Faz sentido, pois precisamos de uma nova promise a cada tentativa.

Todavia, é preciso haver um intervalo entre as tentativas. Já temos a função delay e só nos resta combiná-la com retry:

```
// app/utils/promise-helpers.js
// código anterior omitido

export const retry = (retries, milliseconds, fn) =>
```

```
    fn().catch(err => {
        console.log(retries);
        return delay(milliseconds)().then(() =>
            retries > 1
                ? retry(retries - 1, milliseconds, fn)
                : Promise.reject(err))
    });
```

Foi necessário realizar `delay(time)()`, porque delay retorna uma função que ao ser chamada devolve uma Promise e como já vimos, a chamada encadeada à `then()` só será feita depois do tempo do delay ter expirado.

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

```
// importação de delay removida e em seu lugar importamos retry
import { log, timeoutPromise, retry} from './utils/promise-helpers.js';
import './utils/array-helpers.js';
import { notasService as service } from './nota/service.js';
import { takeUntil, debounceTime, partialize, pipe } from './utils/operators.js';

const operations = pipe(
    partialize(takeUntil, 3),
    partialize(debounceTime, 500),
);

const action = operations(() =>
    retry(3, 3000, () => timeoutPromise(200, service.sumItems('2143')))
    .then(console.log)
    .catch(console.log)
);

document
.querySelector('#myButton')
.onclick = action;
```

Para testarmos basta carregar a página no navegador, parar o servidor e clicar no botão. Veremos através do console as mensagens das tentativas que serão realizadas dentro de um intervalo de cinco segundos. Na segunda tentativa, podemos subir rapidamente o servidor para que a operação seja realizada com sucesso.