10

Consumindo endpoint REST com a API Fetch

Transcrição

Agora que já temos as coisas no lugar, vamos buscar uma lista de notas fiscais através de uma requisição Ajax com auxílio da da API **Fetch**. Caso o aluno queira saber mais sobre essa fantástica adição na especificação ECMASCRIPT, ele pode conferir o curso JavaScript Avançado - Parte 3 (https://cursos.alura.com.br/course/javascript-es6-orientacao-a-objetos-parte-3). O que faremos ao longo deste curso é levar ao limite o uso do API Fetch para resolvermos problemas do dia a dia.

Vamos direto para implementação do código:

```
document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notas')
    .then(res => res.json())
    .then(notas => console.log(notas))
    .catch(err => console.log(err));
```

Recordar é viver e como utilizaremos a Fetch API até o final deste curso, vamos recordar alguns de seus pontos chaves.

Ela é acessível globalmente através da função `fetch` que recebe como parâmetro o endereço web de algum recurso a ser consumido. Seu retorno é uma **Promise**, uma maneira elegante de lidarmos com o resultado futuro de uma ação. Aliás, Promises também fazem parte da especificação ES2015 (ES6).

Toda Promise possui as funções `then` e `catch`. Na primeira obtemos o resultado da operação. Já na segunda, tratamos possíveis erros que possam ocorrer durante a resolução da nossa Promise. Porém, como podemos ver, há duas chamadas para função `then`.

Um dos *pulos do gato* da Promise é entender que se a função `then` retorna um valor qualquer, esse valor é acessível através de uma nova chamada **encadeada** a `then`.

Vejamos o seguinte trecho de código:

```
// Revisando
// código anterior omitido
fetch('http://localhost:3000/notas')
.then(res => res.json())
// código posterior omitido
```

O resultado da busca é um objeto `Response`. Podemos obter a resposta no formato texto através da função `res.text` ou seu valor já convertido para JSON através de `res.json`. Essa pequena função nos poupa de utilizarmos a função `JSON.parse`.

Como estamos usando uma *arrow function* sem bloco, implicitamente é adicionada a instrução `return`. Sendo assim, na próxima chamada à `then` teremos acesso aos dados retornados no formato JSON, dados que exibimos no console do navegador.

Por fim, na cláusula `catch`, passamos `console.log`. É uma forma mais enxuta do que fazermos `.catch(() => console.log(err))`.

```
// código anterior omitido
// Revisando
document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notas')
    .then(res => res.json())
    // notas aqui é o resultado de res.json()
    .then(notas => console.log(notas))
    .catch(console.log);
    // código posterior omitido
```

Tudo muito lindo se não fosse **uma pegadinha do malandro aqui**. O que acontecerá se acessarmos um endereço que não existe, por exemplo, `http://localhost:3000/notasx`?

Utilizando o endereço proposto acima, veremos a seguinte mensagem de erro no console do Chrome:

```
Unexpected end of JSON input
```

Olhando a linha do erro, vemos que ela aconteceu na função `then` e não na função `catch`, função que esperámos que fosse chamada. Por que isso aconteceu?

Para a API Fetch, qualquer resposta vinda do servidor é uma resposta válida, inclusive a `statusMessage` de status `404` (não encontrado). Nesse sentido, não basta executarmos a operação, precisamos verificar na cláusula `then` o código de status da operação. A boa notícia é que a API Fetch nos traz o atalho `res.ok` para sabermos se a requisição é válida ou não. Ela nos poupa o trabalho de testar cada código de status das faixas 400 e 500, por exemplo.

Vamos alterar nosso código:

```
document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notasx')
    .then(res => {
        if(res.ok) return res.json();
        return Promise.reject(res.statusText);
    })
    .then(notas => console.log(notas))
    .catch(console.log);
```

Usamos a seguinte estratégia. Se o `res.ok` é válido, retornamos o resultado de `res.json()` para a próxima chamada encadeada à `then`. Caso contrário, rejeitamos a Promise passando a mensagem de erro do status vindo do servidor. Isso fará com que a função `catch` seja acionada. Agora, a mensagem de erro recebida no console será:

```
/notasx not found
```

Excelente, porém pode ficar ainda melhor. Na maioria das vezes utilkizaremos a mesma estratégia para lidar com o status da requisição. Sendo assim, que tal realizarmos seu isolamento em um módulo? Que tal o nome `app/utils/promise-helpers.js` ?

Vamos criar o novo arquivo e nele vamos declarar a função `handleStatus` . Aliás, vamos lançar mão de um `if` ternário para evitar o uso de um bloco em nossa arrow function:

```
// app/utils/promise-helpers.js

const handleStatus = res =>
    res.ok ? res.json() : Promise.reject(res.statusText);
```

Todavia, do jeito que declaramos, a função não poderá ser importada por nenhum outro módulo. Precisamos explicitar isso através da instrução `export` :

```
// app/utils/promise-helpers.js

// agora a função pode ser exportada por outros módulos
export const handleStatus = res =>
    res.ok ? res.json() : Promise.reject(res.statusText);
```

Agora, vamos importar a função e utilizá-la em `app/app.js` . Faremos isso através da instrução `import` . No entanto, a implementação do sistema nativo de módulos do Chrome exige que a extensão `.js` seja indicada, caso contrário não conseguiremos importar o módulo.

> *Frameworks como Angular ou React que utilizam Webpack por debaixo dos panos não precisam adicionar a extensão, pois o módulo bundler se encarregará de resolvê-la.*

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

// ainda estamos usando o endereço errado
document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notasx')
    .then(handleStatus)
    .then(notas => console.log(notas))
    .catch(console.log);
```

Muito mais legível não? Não esqueça depois de realizar alguns testes e corrigir o endereço utilizado para que possamos continuar.

Se olharmos a aba *network* do Chrome veremos que o módulo `promise-helpers.js` foi carregado inteligentemente pelo navegador que detectou a dependência no módulo principal da aplicação `app.js` .

Agora que já temos a espinha dorsal do nosso projeto podemos começar, no próximo capítulo, a dificultar as coisas e a exigir mais do desenvolvedor. Preparado?