01

xmlHttpRequest: será que existe algo de mais alto nível?

Transcrição

Continuaremos com as melhorias no código e a seguir, veremos algo mais avançado. Nós criamos a classe `HttpService`, depois, escondemos a complexidade de trabalhar com o `XMLHttpRequest()`. Nós fizemos método `get` e `post` devolverem uma *Promise*, e assim, escondemos a complexidade de trabalhar com tal objeto.

Neste curso, estamos usando o ECMAScript 2015. Não usamos mais o termo "ES 6", porque a cada ano, o JavaScript ganha novos recursos. No ES 2016, foi incluída uma API com o objetivo de simplificar a criação de requisições Ajax: **Fetch API**, uma API de busca do JS. O que veremos aqui, vai além do ECMAScript 2015.

Talvez, você fique preocupado se o seu código funcionará em outros navegadores, mas temos uma solução para a questão de compatibilidade. Mas, por enquanto, pedimos que você realize os testes no Chrome ou no Firefox, deixando os outros browsers de lado por enquanto.

Atualmente, o método `get()` está assim:

```
class HttpService

    get(url) {

        return new Promise((resolve, reject) => {

          let xhr = new XMLHttpRequest();

          xhr.open('GET', url);

          xhr.onreadystatechange = () => {

              if(xhr.readyState == 4) {

                  if(xhr.status == 200) {

                    resolve(JSON.parse(xhr.responseText));
                  } else {

                    reject(xhr.responseText);
                  }
              }
          };

          xhr.send();

          });
    }
```

Nós iremos apagar este trecho e reescreveremos o `get()`. No escopo global, nós iremos adicionar a variável `fetch`, no `HttpService.js`. O resultado dela está no `then()`, isto significa que o retorno será uma *Promise* por padrão.

```
class HttpService {

    get(url) {

      return fetch(url)
          .then(res => console.log(res));
    }
}
```

Pedimos que ela busque por uma resposta (`res`). Observe a diferença entre o código anterior com o atual. Conseguimos simplificá-lo bastante... No entanto, quando recebemos a resposta, ela está bruta - não sendo um texto ou JSON. Pediremos que a resposta seja convertida para o formato que desejamos. No caso, definiremos que ela seja `json` , mas poderíamos pedir em `texto` também.

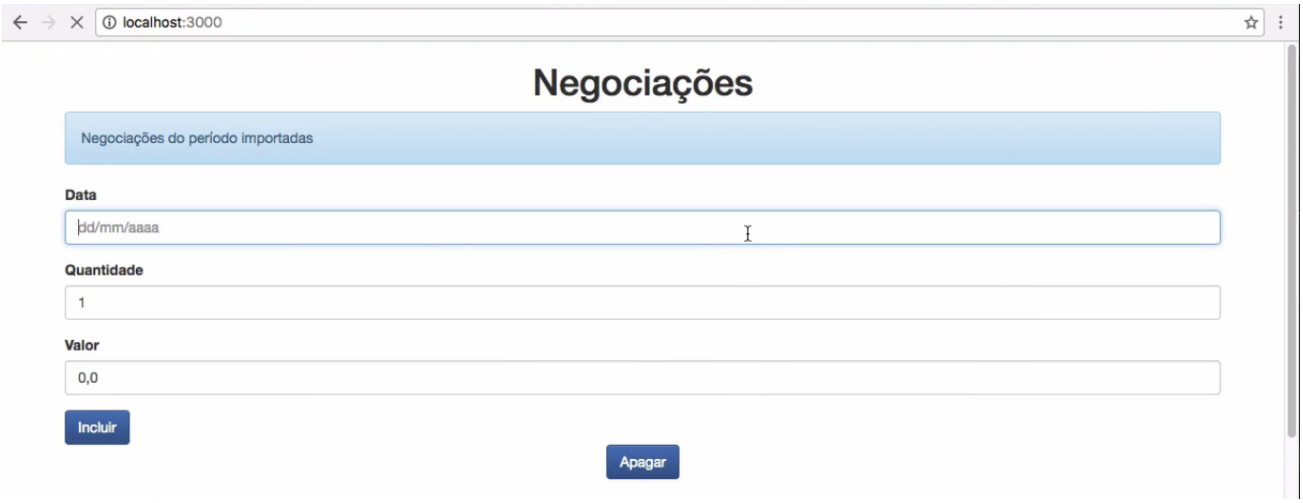
```
class HttpService {

    get(url) {

      return fetch(url)
          .then(res => res.json());
    }
}
```

Com as alterações, o `.then(res => res.json())` substituiu o `JSON.parse` do `post()` . Nós pediremos para o próprio objeto da resposta, vindo do Back-end, será o responsável pela conversão do formato. Como estamos trabalhando com uma *Promise*, também faremos o retorno.

Até aqui, nosso código já poderia estar funcionando. Você pode estar surpreso com o tamanho enxuto, mas conseguimos isto porque não precisamos trabalhar com o `onreadystatechange` . No entanto, temos a desvantagem de, ao trabalhar com a Fetch API, por não trabalharmos com estado, também não conseguiremos cancelar um requisição Ajax no meio. Com o `readyState` , quando mandamos a requisição e ela demora muito, temos a opção de cancelá-la. Porém, como são raros os casos em que queremos cancelar a requisição, a Fetch API é uma boa escolha.



No entanto, faltou tratar os casos de erro na nossa Fetch API. Como o código identificava se tínhamos um erro? Testávamos com o `readyState` se a requisição estava completa e verificávamos se o estado era `200` ou com um valor próximo. Neste caso, nós usaremos o `res.ok` para fazermos testes com o status e nos indicará se é falso ou verdadeiro. Vamos ver como tratar o erro:

```
class HttpService {

    _handleErrors(res) {
        if(res.ok) {
            return res;
        } else {
          throw new Error(res.statusText);
        }
    }

    get(url) {

        return fetch(url)
            .then(res => this._handleErrors(res))
            .then(res => res.json());
    }
//...
```

Para manter a organização do código, criamos o método privado `_handleErrors()`. O `.then` no `fetch` devolverá a própria requisição `this._handleErrors` que será acessível no próximo `.then` e será convertido para `json`. Com o `if` identificamos se tudo funcionou bem com o `res.ok`, caso contrário, cairemos no `else` e exibiremos a mensagem de erro (`statusText`).

Mas vamos simplificar o código, reescrevendo o `if` :

```
_handleErrors(res) {
    if(!res.ok) throw new Error(res.statusText);
    return res;
}
```

Se tivermos problema, retornaremos o `throw`. Mas se tudo correr bem, retornaremos o `res`. A mensagem de erro antes era exibida com o `responseText`, e agora, usamos o `res.statusText`. Quando a exceção for lançada, a Promise não irá para o `.then` do `get()`. Ela seguirá para o `catch`.

Se atualizarmos a página no navegador, em alguns instantes receberemos a mensagem de que as negociações do período foram importadas corretamente.