08

Relembrando Promises e a API Fetch

Este curso pressupõe que o aluno tenha conhecimento prévio de `Promises` e que tenha tido um contato, ainda que ínfimo, com a API Fetch. Como diz o ditado, "recordar é viver", vamos recapitular os conceitos fundamentais antes que você assista o próximo vídeo. Se por algum motivo, as explicações aqui não serem suficientes, o aluno deve se dirigir imediatamente para o curso Curso JavaScript Avançado III: ES6, orientação a objetos e padrões de projetos (https://cursos.alura.com.br/course/javascript-es6-orientacao-a-objetos-parte-3) deste mesmo autor, aliás, pré-requisito deste treinamento. Lá os assuntos aqui apresentados serão abordados detalhadamente.

Promises retornam o resultado futuro de uma ação

O padrão `Promise` adicionado à especificação ES2015 nos auxilia a lidar com o resultado futuro de uma ação, em outras palavras, essa especificação brilha no tratamento de operações assíncronas substituindo callbacks que podem dificultar a leitura e manutenção do código. Esse problema decorrente de callbacks aninhados é chamado de *callback HELL*.

Uma Promise possui estados. Os mais importantes são o `resolved` e o `rejected`. O primeiro é quando a Promise realiza seu objetivo, já o segundo ocorre quando, por algum motivo, ela é impedida de ser resolvida. Não é permitido que uma Promise no estado `resolved` vá para o estado `rejected` e vice-versa. Nesse sentido, uma Promise é imutável. No entanto, `Promises` podem retornar novas `Promises` a partir de suas operações.

API Fetch adere à espeficação Promise

A API Fetch, que veio simplificar em muito a realização de requisições assíncronas (Ajax) adere à especificação `Promise`. Vejamos um exemplo:

```
fetch('http://endereco-de-uma-api')
    .then(res => console.log(res));
```

A chamada `fetch('http://endereco-de-uma-api')` retorna uma `Promise`. Toda `Promise` possui o método `then` que será chamado apenas quando a operação for resolvida. No caso da API Fetch, recebemos como resposta um objeto com o fluxo de resposta do servidor. Geralmente, precisamos realizar o parser da resposta através de `JSON.parse`, contudo o fluxo de resposta vindo do servidor representado como um objeto possui o método `json()` que realiza esse parser para nós:

```
fetch('http://endereco-de-uma-api')
    .then(res => res.json());
    .then(retorno => console.log(retorno));
```

A instrução `then(res => res.json());` retorna o resultado de `res.json()`, pois uma *arrow function* sem bloco possui um retorno implícito. Todo dado retornado por `then()` é encapsulado em uma nova `Promise`, por isso podemos encadear uma nova chamada à `then()` para termos acesso aos dados retornados. Podemos simplificar nosso código ainda desta forma:

```
fetch('http://endereco-de-uma-api')
    .then(res => res.json());
    .then(console.log);
```

Mas se algo der errado? É através do método `catch` que lidamos com erros de rejeição da `Promise` :

```
fetch('http://endereco-de-uma-api')
    .then(res => res.json());
    .then(console.log)
    .catch(err => console.log(err));
```

Qualquer erro lançado pela `Promise` será capturado no método `catch` .

Para terminarmos, é importante consultarmos o status da requisição, porque qualquer resposta vinda do servidor será uma resposta válida, mesmo as de erro como 404, 500 e assim por diante. A API Fetch nos dá um atalho para verificar o status da requisição:

```
fetch('http://endereco-de-uma-api')
    .then(res => {
        // se true, retorna os dados parseados
        if(res.ok) return res.json();
        // retorna uma Promise rejeitada com a informação vinda do servidor
        return Promise.reject(res.statusText);
    })
    .then(res => res.json());
    .then(console.log)
    .catch(err => console.log(err));
    // ou .catch(console.log);
```

Ao longo deste treinamento extrapolaremos o conhecimento que temos sobre Promises, por isso estar seguro nesta tecnologia é importante.