05

Só acredito vendo: listando objetos de uma store

Transcrição

Aprendemos a incluir e precisamos aprender como listar, as duas operações que precisaremos para nossa aplicação.

Vamos criar a função `listaTodos` :

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
</head>
<body>

    <script src="js/app/models/Negociacao.js"></script>
    <script>

        var connection;

        // código anterior que abre a conexão com o banco omitido

        function adiciona() {

            // código omitido
        }

        function listaTodos() {

            let transaction = connection.transaction(['negociacoes'],'readwrite');
            let store = transaction.objectStore('negociacoes');
            let negociacoes = [];
            }
    </script>
</body>
```

Veja que o procedimento é parecido com a função `adiciona()` . Primeiro, abrimos uma transação para uma lista de *object stores* e depois solicitamos uma transação específica para a *store* que desejamos trabalhar. Em seguida, veja que já é declarado um array vazio de negociações. Precisamos populá-lo com as negociações do nosso banco. Para isso, precisaremos solicitar à *store* um cursor.

Lidando com cursores

O `cursor` é um objeto especial que nos permitirá iterar sobre todas as negociações armazenadas em nossa *store*. À medida que formos iterando com o cursor, vamos adicionando as negociações na lista:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
```

```
</head>
<body>

    <script src="js/app/models/Negociacao.js"></script>
    <script>

        var connection;

        // código anterior que abre a conexão com o banco omitido

        function adiciona() {

            // código omitido
        }

        function listaTodos() {

            let transaction = connection.transaction(['negociacoes'],'readwrite');
            let store = transaction.objectStore('negociacoes');
            let negociacoes = [];

            let cursor = store.openCursor();

            //
            cursor.onsuccess = e => {

            };

            cursor.onerror = e => {
                console.log('Error:' + e.target.error.name);
            };
        }
    </script>
</body>
```

O evento `onsuccess` do nosso cursor será chamado o número de vezes correspondente à quantidade de negociações armazenadas em nossa *object store*. Na primeira chamada, teremos acesso a um ponteiro para a primeira negociação, na segunda chamada teremos um ponteiro para a segunda, e assim por diante...

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
</head>
<body>

    <script src="js/app/models/Negociacao.js"></script>
    <script>

        var connection;

        // código anterior que abre a conexão com o banco omitido

        function adiciona() {

            // código omitido
```

```
        }

        function listaTodos() {

            let transaction = connection.transaction(['negociacoes'],'readwrite');
            let store = transaction.objectStore('negociacoes');
            let negociacoes = [];

            let cursor = store.openCursor();

            cursor.onsuccess = e => {

                let atual = e.target.result;

                if(atual) {

                    negociacoes.push(atual.value);
                    atual.continue();

                } else {

                    // quando não há mais objects em nossa store.
                    // Isso significa que já terminados de popular negociacoes

                    console.log(negociacoes);
                }
            };

            cursor.onerror = e => {
                console.log('Error:' + e.target.error.name);
            };
        }
    </script>
</body>
```

Vamos fazer um teste, executando a nossa função `listaTodos()` no console do `Chrome`:

```
listaTodos()
```

Será impresso no console um array com todas as negociações. Contudo, se examinarmos cada item do array, veremos que é apenas um objeto com as propriedades `_data`, `_quantidade` e `_valor`. Isso acontece porque quando gravamos um objeto em uma *store*, apenas suas propriedades que não forem funções serão gravadas. Sendo assim, antes de adicionarmos no array de negociações, precisaremos criar uma nova instância de `Negociacao` com base nos dados:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
</head>
<body>

    <script src="js/app/models/Negociacao.js"></script>
    <script>
```

```
        var connection;

        // código anterior que abre a conexão com o banco omitido

        function adiciona() {

            // código omitido
        }

        function listaTodos() {

            let transaction = connection.transaction(['negociacoes'],'readwrite');
            let store = transaction.objectStore("negociacoes");
            let negociacoes = [];

            let cursor = store.openCursor();

            cursor.onsuccess = e => {

                let atual = e.target.result;

                if(atual) {

                    let dado = atual.value;

                    negociacoes.push(new Negociacao(dado._data, dado._quantidade, dado._valor)

                    atual.continue();

                } else {

                    console.log(negociacoes);
                }
            };

            cursor.onerror = e => {
                console.log('Error:' + e.target.error.name);
            };
        }
    </script>
</body>
```

Perfeito, se olharmos a saída no console veremos que temos um array de instâncias da classe `Negociacao`.

Com todo conteúdo visto, temos o pré-requisito para integrar nossa aplicação com o `IndexedDB`, contudo você deve ter visto que nosso código não é um dos melhores. No próximo capítulo, veremos como organizá-lo, recorrendo a algumas *patterns* do mercado.