01

O padrão de projeto DAO

Transcrição

Começando deste ponto? Você pode fazer o DOWNLOAD (https://github.com/alura-cursos/javascript-avancado-iii/archive/aula2.zip) completo do projeto até aqui e continuar seus estudos.

Vamos continuar... Nós já temos a nossa `ConnectionFactory`, e sempre que quisermos usá-la, iremos chamá-la da seguinte forma:

```
ConnectionFactory
    .getConnection()
    .then(connection => {
    });
```

Pelo fato de termos uma conexão, abstraímos muita coisa. Nós já conseguimos organizar muita coisa do código de `aprendendo_indexddb.html`, um arquivo de difícil manutenção. Mas ainda temos as partes referentes à adição e à listagem...Vamos aplicar outro padrão de projeto para organizarmos a parte de persistência: usaremos o padrão **DAO** (Data Access Object). Normalmente, quando este é utilizado, ele abstrairá os detalhes de lidar com o banco.

```
ConnectionFactory
    .getConnection()
    .then(connection => {

        let dao = new NegociacaoDAO(connection);
        let negociacao = new Negociacao(new Date(), 1, 100);
        dao
            .adiciona(negociacao);
            .then() =>
          //*
});
```

Utilizamos uma convenção do padrão DAO na qual, se estamos fazendo uma persistência de `Negociacao`, usaremos `NegociacaoDAO`. Por sua vez, ele dependerá de uma `connection` para funcionar. Então, se observarmos o código, perceberemos que `NegociacaoDAO` foi criado para criar conexão e fazer a persistência da negociação. Isto significa que conseguimos resumir os métodos `adiciona()` e `ListaTodos()` neste trecho. Porém, o `dao.adiciona` também é uma *Promise* e podemos aproveitar o o resultado do `then()`, exibindo alguma mensagem para o usuário.

Esta será a lógica do padrão DAO que utilizaremos. Vamos começar a aplicá-la, criando a pasta `dao` e um novo arquivo: `NegociacaoDao.js`. A classe irá receber uma conexão no construtor.

```
class NegociacaoDao {

    constructor(connection) {

        this._connection = connection;
        this._store = 'negociacoes';
```

```
    }
}
```

Teremos duas propriedades privadas: `_connection` e `_store`. Elas só poderão ser utilizadas pelo DAO, que irá operar sobre a store `negociacoes`. A seguir, implementaremos o método `adiciona()`.

```
adiciona() {

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

    });
}
```

O método retornará uma Promise. Depois, começaremos a elaborar o processo de inclusão aproveitando o código de `adiciona()` gerando no `aprendendo_indexddb.html` no `NegociacaoDAO.js`:

```
adiciona(negociacao) {

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

          let transaction = this._connection.transaction([this._store], 'readwrite');

          let store = transaction.objectStore(this._store);

          let request = store.add(negociacao);

      });
  }
```

Quando adicionarmos uma negociação, ela será passada para o DAO, a `_connection` abrirá uma transação para a `_store`: `readwrite`. No fim, pediremos para dar `add`. No entanto, temos a opção de encadear as chamadas das funções. Com as alterações, o trecho ficará da seguinte forma:

```
adiciona(negociacao) {

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

        let request = this._connection
            .transaction([this._store],'readwrite')
            .objectStore(this._store)
            .add(negociacao);

    });
}
```

Ao chamarmos o `add()`, ele nos retornará o `request`. Em seguida, o `request` lidará com dois eventos: `onsuccess` e `onerror`. Quando tivermos sucesso, passaremos o `resolve()`, nos casos de erro, teremos o `console.log`:

```
class NegociacaoDao {
```

```
        constructor(connection) {

            this._connection = connection;
            this._store = 'negociacoes';
        }

        adiciona(negociacao) {

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

                let request = this
                    ._connection
                    .transaction([this._store],"readwrite")
                    .objectStore(this._store)
                    .add(negociacao);

                request.onsuccess = (e) => {

                    resolve();
                };

                request.onerror = e => {

                    console.log(e.target.error);
                    reject('Não foi possível adicionar a negociação');
                };
            });
        }
    }
```

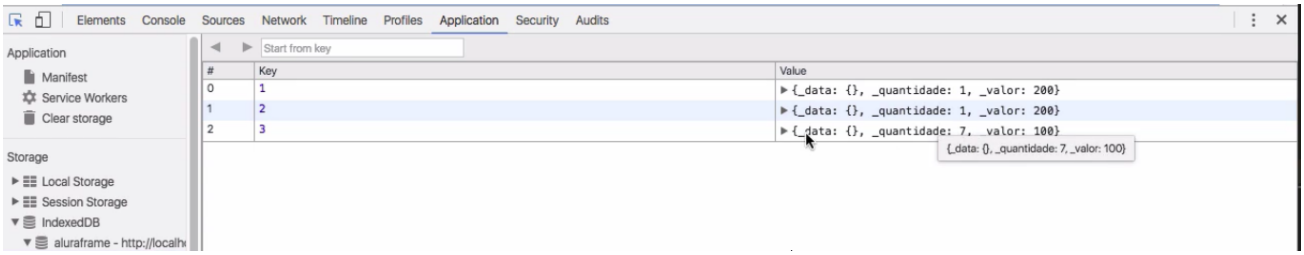
Depois, importaremos o `NegociacaoDAO` .

```
<script src="js/app/models/Negociacao.js"></script>
<script src="js/app/models/ListaNegociacoes.js"></script>
<script src="js/app/models/Mensagem.js"></script>
<script src="js/app/controllers/NegociacaoController.js"></script>
<script src="js/app/helpers/DateHelper.js"></script>
<script src="js/app/views/View.js"></script>
<script src="js/app/views/NegociacoesView.js"></script>
<script src="js/app/views/MensagemView.js"></script>
<script src="js/app/services/ProxyFactory.js"></script>
<script src="js/app/helpers/Bind.js"></script>
<script src="js/app/services/NegociacaoService.js"></script>
<script src="js/app/services/HttpService.js"></script>
<script src="js/app/services/ConnectionFactory.js"></script>
<script src="js/app/dao/NegociacaoDao.js"></script>
<script>
    var negociacaoController = new NegociacaoController();
</script>
```

Agora, vamos fazer um teste no navegador. No Console, digitaremos:

```
ConnectionFactory.getConnection().then(connection => new NegociacaoDao(connection).adiciona(new
```

Veremos que ele gravou perfeitamente as negociações.



O Chrome tem um bug que não exibe a data da negociação, mas ela foi gravada corretamente.

Após criarmos o `NegociacaoDao` , iremos integrá-lo com a aplicação. É o que faremos mais adiante.