02

Construindo um template dinâmico com a função map

Transcrição

A função `update()` para atualização da View está funcionando e a tabela já pode ser visualizada abaixo do formulário. Porém, os dados do modelo ainda não são levados em consideração na construção dinâmica da tabela. Primeiramente, passaremos a `ListaNegociacoes` como parâmetro do método `update()`. Ou seja, quando o modelo for alterado, a lista deverá ser atualizada da tabela.

```
class NegociacaoController {

    constructor() {

        let $ = document.querySelector.bind(document);
        this._inputData = $('#data');
        this._inputQuantidade = $('#quantidade');
        this._inputValor = $('#valor');

        this._listaNegociacoes = new ListaNegociacoes();
        this._negociacoesView = new NegociacoesView($('#negociacoesView'));
        this._negociacoesView.update(this._listaNegociacoes)

    }
}
```

A ação não será realizada apenas quando a `controller` for carregada, mas também quando o `adiciona()` for chamado. Porque atualizamos a lista, assim que acabamos de adicionar uma nova negociação, temos que solicitar para View que esta se renderize com o novo modelo. Em `NegociacoesView.js`, faremos com que o método `update()` recebe o `model`.

```
update(model) {

    this._elemento.innerHTML = this._template(model);
}
```

Passamos o `model` como parâmetro do `_template()`.

```
_template(model)

    return `
    <table class="table table-hover table-bordered">
                <thead>
                    <tr>
                        <th>DATA</th>
                        <th>QUANTIDADE</th>
                        <th>VALOR</th>
                        <th>VOLUME</th>
                    </tr>
                </thead>

                <tbody>
```

```
            </tbody>
        </table>
        `;
    }
```

Dentro da tag `<tbody>`, adicionaremos tags `<tr>` com base em cada negociação do `ListaNegociacoes`. Para isto, usaremos uma expressão que conterá o `map()` - podemos usar, inclusive, uma *arrow function*.

```
<tbody>
    ${model.negociacoes.map(n => {

    })}
</tbody>
```

Se adicionarmos um `console.log(n)` e executarmos o código, a negociação será impressa no Console. Com o `return n`, será gerada uma nova lista, com base na modificação. O código ficaria assim:

```
<tbody>
    ${model.negociacoes.map(n => {
        console.log(n);
        return n;
    })}
</tbody>
```

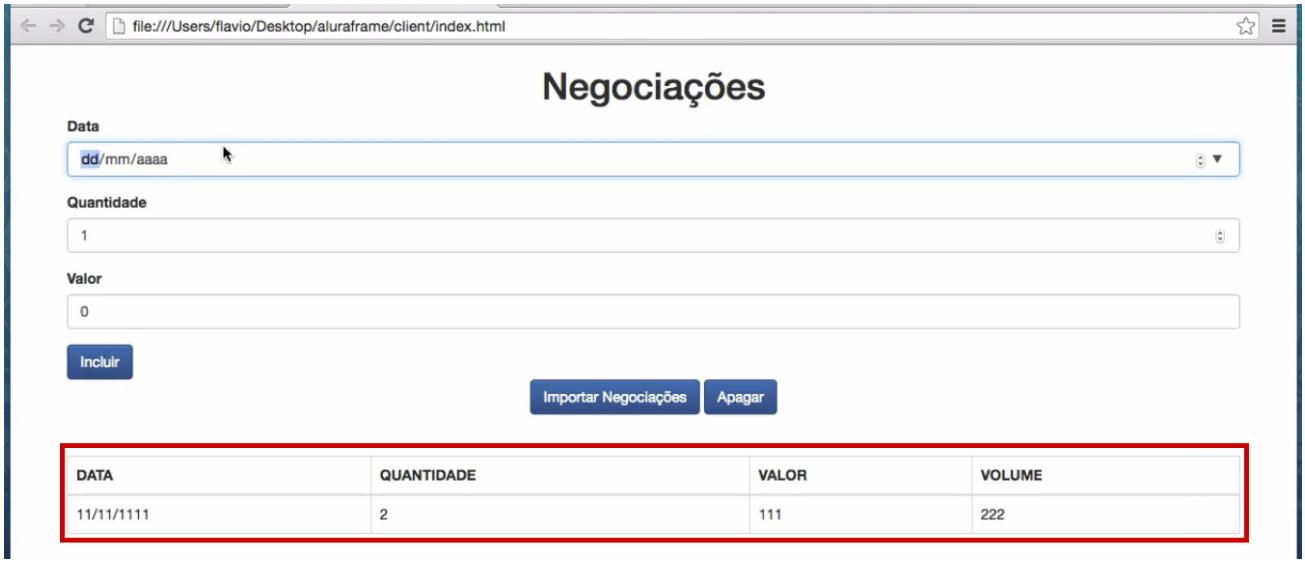
Porém, se selecionarmos esta opção, teremos um problema: a expressão precisa nos devolver uma *string*, que seja enxertada no template. Seguiremos outro caminho: varreremos cada negociação e usaremos o `return` de outra *template string*.

```
<tbody>
    ${model.negociacoes.map(n => {

        return `
            <tr>
                <td>${DateHelper.dataParaTexto(n.data)}</td>
                <td>${n.quantidade}</td>
                <td>${n.valor}</td>
                <td>${n.volume}</td>
            </tr>
        `
    })}
</tbody>
```

Dentro da *template string*, adicionamos as tags `<tr>` e `<td>`, e usamos várias expressões para definirmos a exibição de `data`, `quantidade`, `valor` e `volume`. Quando o `_template()` for retornar a string, terá que processar o trecho do `return` primeiramente, e depois retornar a *template string*. Para cada negociação será criada uma lista - cada uma com as tags `<tr>` e os dados cadastrados. Estamos varrendo a lista e para um objeto `Negociacao`, estamos criando um *array*, mas o novo elemento será uma *string* com os dados. No entanto, por enquanto, o retorno será um *array*. Por isso, adicionaremos o `join()`.

```
<tbody>
    ${model.negociacoes.map(n => {

      return `
        <tr>
            <td>${DateHelper.dataParaTexto(n.data)}</td>
            <td>${n.quantidade}</td>
            <td>${n.valor}</td>
            <td>${n.volume}</td>
        </tr>
      `
      }).join('')}
</tbody>
```

Ao utilizarmos o `join()`, usamos como critério de junção uma *string* em branco. Agora, teremos uma *string* com todos os dados do *array* concatenados. Vamos ver o que será exibido no navegador, após o preenchimento do formulário:



Em seguida, adicionaremos uma nova negociação e os dados também serão exibidos na tabela.

DATA	QUANTIDADE	VALOR	VOLUME
11/11/1111	2	111	222
22/2/1111	2	200	400

Se completarmos os dados do formulário novamente, a tabela terá dados das três negociações. Observe que não manipulamos o DOM de maneira imperativa, em vez disso, fizemos de maneira **declarativa**. Nós declaramos o template, ele recebeu um modelo e com base nos dados do modelo, usamos a *template string*.

Conseguimos de maneira elegante, utilizando apenas recursos do JavaScript, fazer um template render. Porém, faltam algumas ações para que nossa tabela fique completa.