01

# Classes que representam nossas Views

### Transcrição

Começando deste ponto? Você pode fazer o DOWNLOAD (https://github.com/alura-cursos/javascript-avancado-i/archive/aula5.zip) completo do projeto até aqui e continuar seus estudos.

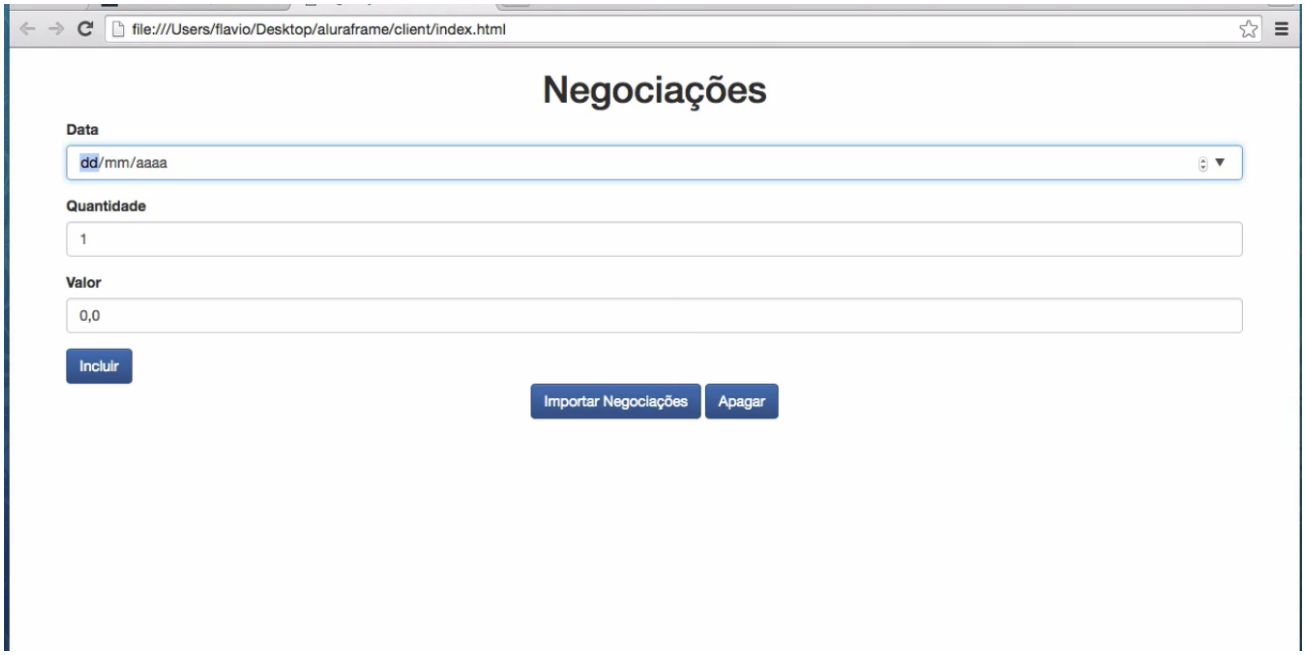
Temos um modelo de `Negociacao`, outro de `ListaNegociacoes`, e temos uma `controller` que orquestra o acesso aos modelos de acordo com as ações do usuário. No entanto, ainda não conseguimos refletir o estado do modelo para a tela. A tabela ainda não exibe os dados cadastrados. Agora temos que atacar a View, do MVC, já temos o model e a controller. Para que possamos aplicar vários conceitos e conhecer novos recursos da linguagem JavaScript, minha proposta é que em cada parte da View que sincronizarmos com o modelo, não será feita no arquivo HTML. Em vez disso, criaremos a classe `NegociacoesView` dentro da pasta `views` que irá encapsular a apresentação que exibiremos para o usuário. É o arquivo `NegociacoesView.js` que terá as definições de como será a tabela. Começaremos com ela assim:

```
<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>

    <tfoot>
    </tfoot>
</table>
```

Como retiramos o trecho do código referente à tabela, no `index.html`, ela já não será mais exibida abaixo do formulário.




Ao incluirmos uma negociação na lista, queremos que ela seja incluída e exibida na tabela. Para isto, em `NegociacoesView.js`, adicionaremos a classe `NegociacoesView`, e dentro dela, criaremos a função `template()` - que retornará uma *template string*. Depois, jogaremos o conteúdo da tabela dentro do `return` da função.

```
class NegociacoesView {

    template() {

        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>
            `;
    }
}
```

Se o retorno não fosse uma *template string*, não poderíamos "identar" o código desta forma. Se fosse uma *string*, não poderíamos quebrar o código em linhas desta maneira. Ele não funcionaria. Nós teríamos que concatenar todas as linhas.

Em seguida, no `NegociacaoController`, adicionaremos a propriedade `this._negociacoesView`:

```
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();
    }
//...
```

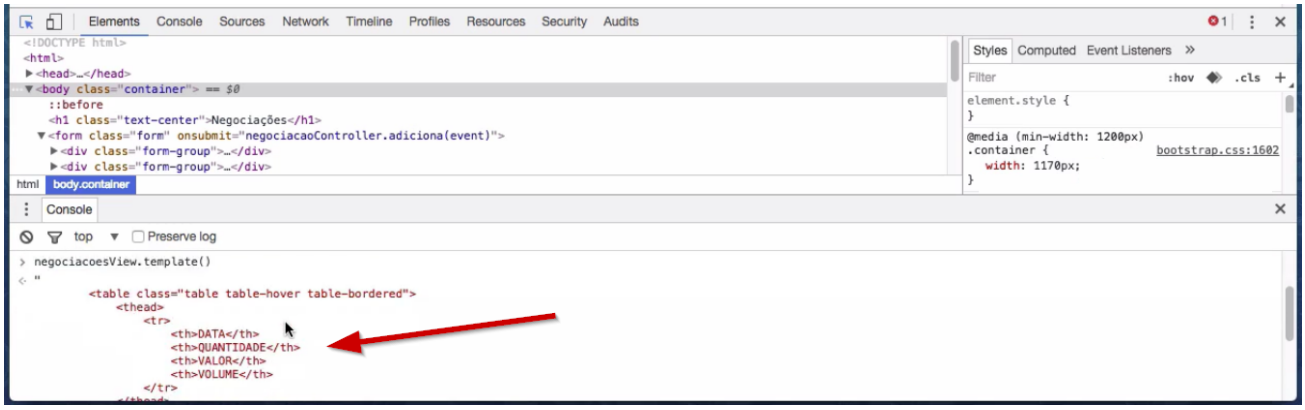
Precisaremos importar o arquivo também no `index.html` :

```
    <script src="js/app/models/Negociacao.js"></script>
    <script src="js/app/controllers/NegociacaoController.js"></script>
    <script src="js/app/helpers/DateHelper.js"></script>
    <script src="js/app/models/ListaNegociacoes.js"></script>
    <script src="js/app/views/NegociacoesView.js"></script>
    <script>
        let negociacaoController = new NegociacaoController();
</script>
```

Após recarregarmos a página vamos digitar a seguinte linha no Console:

```
let negociacoesView = new NegociacoesView()
```

Teremos uma instância de `NegociacoesView` . Se chamamos `negociacoesView.template()` , recebemos uma mensagem de erro:




No Console, será exibida a string da tabela. Então, qual será nosso próximo objetivo? O template que está no `NegociacoesView.js` tem que aparecer no `index.html` , onde estava a marcação do HTML da tabela. Para isto, sinalizaremos o local em que o template será renderizado, adicionando a tag `<div>` e dentro, o `id` .

```
<div id="negociacoesView"></div>
```

Mas o `NegociacoesView` precisa saber que construiremos a tabela nesta `<div>` . Precisamos de alguma forma associar o elemento do DOM com a `NegociacoesView` . Por isso, vamos gerar um `constructor()` que recebe um elemento, responsável por receber o template:

```
class NegociacoesView {

    constructor(elemento) {

        this._elemento = elemento;
    }

    template() {

        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>
                `;
    }
}
```

Em `NegociacaoController`, teremos que buscar o `#negociacoesView`:

```
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();
    }
//...
```

Assim que a negociação for criada, pediremos para o `negociacoesView` fazer um `update`, então, a tabela aparecerá dentro da View. Depois, adicionaremos a função `update` dentro de `NegociacoesView`. Também vamos inserir o `_` ao `template`, sinalizando que será uma função privada.

```
_template() {

    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>
        `;
    }
```

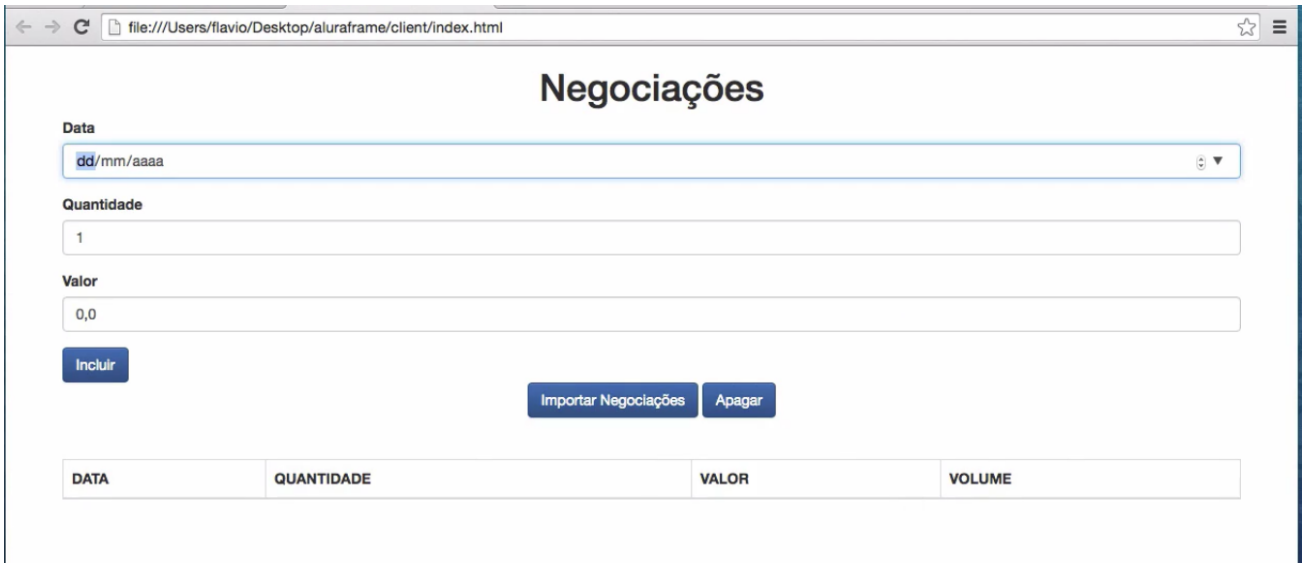
O método `update()` que será criado, pegará o elemento do DOM e acessará a propriedade `innerHTML` . Ela será o retorno da função `_template()` :

```
update() {

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

O `innerHTML` será responsável por converter as *strings* em elementos do DOM. Isto será inserido com filho da `<div>` .

Após as últimas alterações, quando recarregarmos a página no navegador, a tabela já será visualizada.




O problema é que ao chamarmos a função `update()` , a View que estamos renderizando no HTML, deveria refletir a lista de negociações. Se tivermos cinco negociações, todas deverão ser exibidas. Veremos mais adiante como atualizar a View com os dados do modelo.

Nosso objetivo era chamar na View a função `update()` , quando o modelo fosse atualizado, passando como parâmetro o `model` - que será enviado para o template da View. Depois, este será processado e usará como base os dados do `model` . A *string* final será colocada no atributo do elemento que a View associada no DOM. Toda *string* colocada na propriedade `innerHTML` será convertida em elementos do DOM.

Desta forma, conseguimos cadastrar as negociações, sendo atualizada no modelo e este notificará a View que deverá ser renderizada.