04

Apresentando o modelo para o usuário

Transcrição

De nada adianta termos uma lista de negociações se não somos capazes de exibí-la para o usuário. Precisamos construir dinamicamente uma tabela que represente os dados mais atualizados. Temos duas formas de fazer isso. A primeira, imperativa, é realizarmos o passo a passo de cada mutação que desejamos realizar no DOM, já a segunda, declarativa, podemos utilizar um template que pode ser compilado resultado em todas as mutações necessárias que precisamos realizar. Seguiremos a segunda forma.

Vamos declarar nosso template no mundo JavaScript. Se você já trabalhou com React do Facebook, ele também declara templates no mundo JavaScript com JSX. Utilizaremos uma solução que envolve apenas JavaScript.

Vamos criar o arquivo classe `app/ts/views/NegociacoesView.ts`. Nele, temos a classe `NegociacoesView` que possuirá apenas o método `template()`, que retornará uma string com a estrutura da tabela que desejamos construir:

```
// app/ts/views/NegociacoesView.ts

class NegociacoesView {


    template(): string {

        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>

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

Utilizamos template string pela comodidade de quebras de linha, além de permitir interpolações sem o uso do operador `+`.

Antes de continuarmos, vamos importar o JavaScript resultante da compilação do nosso arquivo.

```
<!-- app/index.html -->
<!-- código anterior omitido -->
```

```
    <script src="js/models/Negociacao.js"></script>
    <script src="js/controllers/NegociacaoController.js"></script>
    <script src="js/models/Negociacoes.js"></script>
    <script src="js/views/NegociacoesView.js"></script>
    <script src="js/app.js"></script>
<!-- código posterior omitido -->
```

Recarregando nossa página, vamos realizar um teste no Console do navegador:

```
// NO CONSOLE DO NAVEGADOR
view = new NegociacoesView();
view.template(); // exibe o template retornado
```

Excelente, mas agora precisamos indicar onde em nosso `app/index.html` nosso template será renderizado. Faremos isso adicionando uma `<div>` com o ID `negociacoesView`:

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

    <script src="js/models/Negociacao.js"></script>
    <script src="js/controllers/NegociacaoController.js"></script>
    <script src="js/models/Negociacoes.js"></script>
    <script src="js/views/NegociacoesView.js"></script>
    <script src="js/app.js"></script>
```

Agora, vamos criar a propriedade `_negociacoesView` em `NegociacaoController`. Ela guardará uma instância de `NegociacoesView`:

```
class NegociacaoController {

    private _inputData: HTMLInputElement;
    private _inputQuantidade: HTMLInputElement;
    private _inputValor: HTMLInputElement;
    private _negociacoes = new Negociacoes();

    // vai dar um erro de compilação, pois a classe não recebe parâmetro ainda
    private _negociacoesView = new NegociacoesView('#negociacoesView');
```

Estamos passando para o `constructor` de `NegociacoesView` um seletor CSS que indica o elemento do DOM que receberá a compilação do nosso template. No entanto, a classe ainda não esta preparada para receber esse parâmetro.

A ideia é seguinte. Com base no seletor recebido, `NegociacoesView` guardará internamente uma referência para o elemento do DOM referente ao seletor. Como teremos uma referência para o elemento, podemos atualizá-la com o resultado do template. O tipo da propriedade será `Element`, um tipo bem genérico, aceitando receber tipos mais especializados do DOM. Ele é suficiente porque disponibiliza o setter `innerHTML` que ao receber uma string, internamente a converte para elementos do DOM.

```
class NegociacoesView {

    private _elemento: Element;
```

```
    constructor(seletor: string) {

        this._elemento = document.querySelector(seletor);
    }

    template(): string {

        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>

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

O método `template()` é aquele no qual definimos o template da view, já o novo método `update()` que adicionaremos é aquele que ao ser chamado, atribuirá o resultado de `template()` à propriedade `innerHTML` do elemento do DOM:

```
class NegociacoesView {

    private _elemento: Element;

    constructor(seletor: string) {

        this._elemento = document.querySelector(seletor);
    }

    update(): void {

        this._elemento.innerHTML = this.template();
    }

    template(): string {

        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>

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

Já podemos testar nosso código no terminal.

```
// NO CONSOLE DO NAVEGADOR
view = new NegociacoesView('#negociacoesView');
view.update(); // atualiza o elemento do DOM com os dados do template
```