07

Criando nosso próprio DOM Injector e Lazy loading

Transcrição

Nossa aplicação é funcional, mas podemos organizar ainda melhor nosso código. Percebam que no `constructor()` de `NegociacaoController` procuramos os elementos do DOM para que possamos extrair seus valores ao adicionarmos uma negociação. Essa busca é feita assim que nossa classe é instanciada. Mas se o usuário abre a aplicação e desiste de adicionar negociações? Nós teríamos varrido o DOM desnecessariamente. Além disso, se nossa aplicação crescer e tiver que interagir com mais elementos, já estaríamos buscando todos de uma vez. Podemos melhorar isso com auxílio de decorators de propriedades.

Antes de criarmos nosso decorator, queremos algo assim:

```
import { NegociacoesView, MensagemView } from '../views/index';
import { Negociacoes, Negociacao } from '../models/index';

export class NegociacaoController {

    @domInject('#data')
    private _inputData: JQuery;

    @domInject('#quantidade')
    private _inputQuantidade: JQuery;

    @domInject('#valor')
    private _inputValor: JQuery;
```

Nosso código não compila, claro, mas a ideia é injetarmos os elementos do DOM diretamente na propriedade da classe. Além disso, usaremos a estratégia de *lazy loading*. Por debaixo dos panos, vamos substituir cada propriedade por um getter. Sendo um getter, podemos escrever um bloco de código que ainda assim para o JavaScript ele será considerado uma propriedade. Nesse bloco de código, só buscaremos o elemento do DOM quando o getter for acessado pela primeira vez. Novos acessos retornarão o mesmo elemento!

```
// app/ts/helpers/decorators/domInject.ts

export function domInject(seletor: string) {

    return function(target: any, key: string) {

        let elemento: JQuery;

        const getter = function() {

            if(!elemento) {
                console.log(`buscando  ${seletor} para injetar em ${key}`);
                elemento = $(seletor);
            }

            return elemento;
        }
```

```
    }
}
```

Criamos uma função que será nosso getter, mas como faremos a substituição da propriedade alvo do decorator pelo getter que criamos? Faremos isso com auxílio de `Object.defineProperty`:

```
// app/ts/helpers/decorators/domInject.ts

export function domInject(seletor: string) {

    return function(target: any, key: string) {

        let elemento: JQuery;

        const getter = function() {

            if(!elemento) {
                console.log(`buscando  ${seletor} para injetar em ${key}`);
                elemento = $(seletor);
            }

            return elemento;
        }

       Object.defineProperty(target, key, {
           get: getter
       });
    }
}
```

Não podemos nos esquecer de exportar o decorator através de `app/ts/helpers/decorators/index.ts`.

Por fim, vamos importá-lo em `NegociacaoController` e utilizá-los nas propriedades da classe, não esquecendo de remover a busca manual dos elementos do seu `constructor`:

```
import { NegociacoesView, MensagemView } from '../views/index';
import { Negociacoes, Negociacao } from '../models/index';
import { domInject } from '../helpers/decorators/index';

export class NegociacaoController {

    @domInject('#data')
    private _inputData: JQuery;

    @domInject('#quantidade')
    private _inputQuantidade: JQuery;

    @domInject('#valor')
    private _inputValor: JQuery;

    private _negociacoes = new Negociacoes();
    private _negociacoesView = new NegociacoesView('#negociacoesView');
    private _mensagemView = new MensagemView('#mensagemView');
```

```
        constructor() {
            // removeu a busca manual dos elementos
            this._negociacoesView.update(this._negociacoes);
        }
    // código posterior omitido
```

Quando recarregamos nossa página, nada é exibido no console, perfeito, isso significa que nossos decorators não entraram em ação. Agora, quando adicionarmos uma nova negociação, os elementos serão buscados do DOM uma única vez e reutilizados sem serem buscados novamente. Criamos um decorator que realiza injeção de elementos do DOM com o padrão lazy loading.