03

Isolando a complexidade de associar o modelo com a view na classe Bind

Transcrição

Aplicamos vários conceitos de boas práticas e refatoramos o nosso código... No entanto, quando criamos um Proxy da `_listaNegociacoes` e `_mensagem`, nosso objetivo é realizar um *Data binding* (que traduzido para o português, significa "ligação de dados"). Nós queremos fazer uma associação entre o modelo e a View, ou seja, sempre que alterarmos o modelo, queremos disparar a atualização da View. Damos o nome disso de **Data binding unidirecional**.

Mesmo criando o Proxy, ainda precisaremos chamar o `update()` para fazer a primeira renderização.

```
class NegociacaoController {

    constructor() {

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

        this._listaNegociacoes = ProxyFactory.create(
            new ListaNegociacoes(),
            ['adiciona', 'esvazia'], model =>
                this._negociacoesView.update(model));

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

        this._mensagem = ProxyFactory.create(
            new Mensagem(), ['texto'], model =>
                this._mensagemView.update(model));

        this._mensagemView = new MensagemView($('#mensagemView'));
        this._mensagemView.update(this._mensagem);
    }
//...
```

Mas queremos nos livrar do `update()` do `constructor` - ele deve continuar sendo realizado na estratégia de atualização que passamos do `model` com a `view`. Com as alterações o código ficou assim:

```
class NegociacaoController {

    constructor() {

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

```
        this._listaNegociacoes = ProxyFactory.create(
            new ListaNegociacoes(),
            ['adiciona', 'esvazia'], model =>
                this._negociacoesView.update(model));

        this._negocicoesView = new NegociacoesView($('#negociacoesView'));


        this._mensagem = ProxyFactory.create(
            new Mensagem(), ['texto'], model =>
                this._mensagemView.update(model));

        this._mensagemView = new MensagemView($('#mensagemView'));
    }
//...
```

Mas se recarregarmos a página do formulário, a tabela abaixo já não será visualizada inicialmente. Só será vista, ao preenchermos os dados. O que acha de explicitarmos o trecho referente ao *Data binding*? Na pasta `helpers`, adicione um novo arquivo: `Bind.js`.

Em `NegociacaoController`, vamos esconder o `ProxyFactory`:

```
    constructor() {

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

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

        this._listaNegociacoes = new Bind (
                new ListaNegociacoes(),
                this._negociacoesView,
                ['adiciona', 'esvazia']);


        this._mensagemView = new MensagemView($('#mensagemView'));
        this._mensagem = new Bind(
            new Mensagem(),
            this._mensagemView,
            ['texto']);
    }
```

Queremos criar um `new Bind` da `ListaNegociacoes()` com a View - que só será atualizada quando os métodos `adiciona` e `esvazia` forem atualizadas. Não estamos mais fazendo o `View.update()` no `_mensagem` também. Observe que removemos a parte do `ProxyFactory`. Mas `new Bind` retornará uma instância da classe `Bind` e nós queremos que ele nos dê o Proxy configurado. Em seguida, começaremos a trabalhar com a classe `Bind`:

```
class Bind {

    constructor(model, view, props) {
```

```
        let proxy = ProxyFactory.create(model, props, model => {
            view.update(model)
        });

        view.update(model);
        return proxy;
    }
}
```

Na classe `Bind`, receberemos o `constructor()`, receberemos o `model`, a `view` e a `props`. Depois, criaremos uma Proxy, que chamará o `ProxyFactory`. Praticamente, reaproveitaremos o código que removemos anteriormente.

Estamos renderizando pela primeira vez. A grande novidade do JS é que um construtor pode dar um retorno - não apenas uma instância de sua classe. Em linguagens como Java e C#, o construtor não pode dar um retorno. A seguir, vamos importar o `Bind` no `index.html`.

```
<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>
    let negociacaoController = new NegociacaoController();
</script>
```

Se preenchermos os campos do formulário, conseguiremos incluir os dados normalmente. Nós conseguimos reduzir as responsabilidades do desenvolvedor ao criarmos a classe `Bind`, esclarecendo que queremos fazer a associação entre **dado** e **View**, e que ela será feita quando as propriedades especificadas forem acessadas. E usamos internamente o `ProxyFactory` no `Bind`, para criarmos a Proxy. Em seguida, `view.update()` foi chamado - lembrando que, no fim, retornaremos uma instância **diferente** do `Bind`.

Nós criamos ainda um mecanismo de *data binding*, semelhante aos frameworks como AngularJS e AureliaJS - ainda que estes usem recursos mais sofisticados. Durante a nossa jornada para resolver todos os problemas encontrados até aqui, nós conhecemos vários padrões de projetos e muitos recursos da linguagem JavaScript.