**Existe modelo mentiroso? O padrão de projeto Proxy!**

Começando deste ponto? Você pode fazer o [DOWNLOAD](https://s3.amazonaws.com/caelum-online-public/js-avancado/stages/08-aluraframe.zip) completo do projeto do capítulo anterior e continuar seus estudos a partir deste capítulo.

No capítulo anterior vimos que a atualização da view de negociações era realizada em dois momentos. O primeiro na inclusão de novas negociações e o segundo quando a lista era esvaziada. Nessas duas situações, o desenvolvedor precisava intervir explicitamente no código para atualizar a view.

Para livrarmos o desenvolvedor da responsabilidade dele mesmo atualizar a view, adicionamos armadilhas em ListaNegociacoes. Essas armadilhas nada mais eram do que funções passadas para o construtor do nosso modelo, que continham a lógica de atualização da view, sendo chamadas nos métodos adiciona e esvazia. Contudo, essa solução deixa a desejar. Por qual motivo?

Veja que se quisermos aplicar a mesma estratégia em outras classes (e vamos querer fazer isso) precisaremos alterar cada classe do nosso modelo, colocando um código de infraestrutura que não deveria estar lá. Por que não deveria?

**Modelo e reutilização em projetos**

O modelo é a parte do nosso código com maior índice de reutilização. Por exemplo, podemos usar nossa classe ListaNegociacoes em outras aplicações, inclusive com outros frameworks.

Se ela conter qualquer código que não seja do domínio de negociação, seremos obrigados a lidar com esse código. Imagine utilizarmos nossa classe Negociacao com Angular 2? Com certeza teríamos problema, por causa das nossas armadilhas.

Precisamos de alguma forma continuar com a solução que já temos, mas sem poluir nossas classes do modelo. Para isso, aplicaremos o padrão de projeto **Proxy**.

**O padrão de projeto Proxy**

Um Proxy nada mais é do que um cara mentiroso, ele substitui o objeto real que temos a intenção de interagir. O objeto real só é acessado através do proxy, que pode ou não executar um código arbitrário se assim definirmos. Esse código arbitrário é chamado de armadilha. Aham! Já usamos esse termo antes!

Então, a ideia é criarmos proxies a partir de instâncias dos nossos modelos e transferir as armadilhas que adicionamos neles para os proxies. Dessa forma, manteremos nossos modelos intactos, sem qualquer dependência de um código de infraestrutura.

A boa notícia é que o ES2015 possui um recurso nativo para criação de Proxy, dispensando a necessidade de implementarmos esse padrão manualmente. Para que possamos entender como ele funciona, realizaremos alguns testes diretamente na tag <script> de index.html, que instanciamos nosso controller:

<!-- aluraframe/client/index.html -->

<!-- código anterior omitido -->

<script>

let negociacaoController = new NegociacaoController();

// instância real

let negociacao = new Negociacao(new Date(), 2, 100);

// instância mentirosa

let negociacaoProxy = new Proxy(negociacao, {});

// acessamos como se fosse a real

console.log(negociacao.volume);

</script>

Instanciamos um Proxy da mesma maneira que instanciamos uma classe. Em seu construtor, passamos o objeto no qual queremos imitar como primeiro parâmetro, e como segundo um objeto no formato literal {}, que contém as armadilhas (*traps*) que queremos executar para a interações com o objeto que definirmos. Veja que por enquanto não passamos qualquer configuração.

Veja que acessamos o Proxy da mesma maneira que acessamos uma instância de ListaNegociacao. Para escondermos totalmente a instância de Negociacao, vamos remover a declaração da variávelnegociacao, instanciando diretamente uma Negociacao no construtor do Proxy:

<!-- aluraframe/client/index.html -->

<!-- código anterior omitido -->

<script>

let negociacaoController = new NegociacaoController();

let negociacao = new Proxy(new Negociacao(new Date(), 2, 100), {});

// acessamos como se fosse a real

console.log(negociacao.volume);

</script>

Veja que até mudamos o nome da nossa variável de negociacaoProxy para negociacao. Ninguém precisa saber que nossa instância é um Proxy.

**Construindo armadilhas de leitura**

Excelente, mas para que vejamos o poder do Proxy em ação, precisamos configurar algumas armadilhas em nosso proxy. Para que possamos fazer isso, precisamos ter em mente que ao acessarmos uma propriedade do nosso proxy, estaremos realizando um acesso do tipo get.

Vamos adicionar a propriedade get no objeto passado como segundo parâmetro para Proxy e nele passar uma função que recebe três parâmetros: target (objeto real encapsulado pelo proxy), prop(propriedade que está sendo interceptada) e receiver (referência para nosso proxy):

<!-- aluraframe/client/index.html -->

<!-- código anterior omitido -->

<script>

let negociacaoController = new NegociacaoController();

let negociacao = new Proxy(new Negociacao(new Date(), 2, 100), {

get: function(target, prop, receiver) {

// executa antes da leitura da propriedade

console.log(`Interceptou ${prop}`);

}

});

console.log(`Volume: ${negociacao.volume}`);

</script>

Excelente, quando olhamos no console do navegador, vemos que a mensagem "Interceptou volume" é exibida, indicando que nossa armadilha foi disparada, no entanto, o valor da propriedade no console éundefined. Por quê?

O retorno de negociacao.volume é undefined porque logo depois de termos executado nossa armadilha não explicitamos que a propriedade deve seguir seu curso normal de acesso, ou seja, que deve retornar seu valor. Para isso, precisamos efetivamente retornar o valor da propriedade em nosso proxy da seguinte maneira:

<!-- aluraframe/client/index.html -->

<!-- código anterior omitido -->

<script>

let negociacaoController = new NegociacaoController();

let negociacao = new Proxy(new Negociacao(new Date(), 2, 100), {

get: function(target, prop, receiver) {

// executa antes da leitura da propriedade

console.log(`Interceptou ${prop}`);

// retorna para quem acessou a propriedade seu valor

return Reflect.get(target, prop, receiver);

}

});

console.log(`Volume: ${negociacao.volume}`);

</script>

Com o auxílio da classe Reflect, solicitamos que a operação de leitura seja executada e seu valor retornado. O método Reflect.get recebe como parâmetro o contexto no qual uma propriedade será executada, como segundo a propriedade em si e por último uma referência ao nosso proxy.

Podemos ainda perverter o valor, retornando quando acessamos a propriedade negociacao.volume. Vamos retornar o valor calculado acrescido de 200!

<!-- aluraframe/client/index.html -->

<!-- código anterior omitido -->

<script>

let negociacaoController = new NegociacaoController();

let negociacao = new Proxy(new Negociacao(new Date(), 2, 100), {

get: function(target, prop, receiver) {

// executa antes da leitura da propriedade

console.log(`Interceptou ${prop}`);

return target.volume + 200;

}

});

console.log(`Volume: ${negociacao.volume}`);

</script>

Vamos alterar mais uma vez o nosso teste, voltando com o retorno padrão da propriedade volume:

<!-- aluraframe/client/index.html -->

<!-- código anterior omitido -->

<script>

let negociacaoController = new NegociacaoController();

let negociacao = new Proxy(new Negociacao(new Date(), 2, 100), {

get: function(target, prop, receiver) {

console.log(`Interceptou ${prop}`);

return Reflect.get(target, prop, receiver);

}

});

console.log(`Volume: ${negociacao.volume}`);

</script>

Excelente. Que tal aproveitarmos essa situação para aprender mais uma vantagem do ES2015? Quando declaramos um objeto literal em JavaScript com {}, podemos declarar propriedades que recebem funções da mesma maneira que declaramos em nossas classes, sendo assim temos:

<!-- aluraframe/client/index.html -->

<!-- código anterior omitido -->

<script>

let negociacaoController = new NegociacaoController();

let negociacao = new Proxy(new Negociacao(new Date(), 2, 100), {

get(target, prop, receiver) {

console.log(`Interceptou ${prop}`);

return Reflect.get(target, prop, receiver);

}

});

console.log(`Volume: ${negociacao.volume}`);

</script>

Menos verbosidade, não?

**Construindo armadilhas de escrita**

Bom, aprendemos a executar um código arbitrário toda vez que uma propriedade for lida, mas para solucionar o problema da atualização automática de nossa view, precisamos disparar nossa armadilha só quando o modelo for alterado. Leituras de propriedades não devem ser pegas pelas nossas armadilhas.

Para isso, vamos remover o código anterior e criar um Proxy da classe ListaNegociacoes, nosso modelo para representar uma lista de negociações:

<!-- aluraframe/client/index.html -->

<!-- código anterior omitido -->

<script>

let negociacaoController = new NegociacaoController();

let lista = new Proxy(new ListaNegociacoes(), {

set(target, prop, value, receiver) {

console.log(`A propriedade ${prop} receberá ${value}`);

return Reflect.set(target, prop, value, receiver);

}

});

</script>

Diferente do get, a propriedade set recebe um parâmetro extra, que é o valor que está sendo atribuído à propriedade. Atenção para o retorno, que agora usa Reflect.set.

Que tal adicionarmos uma nova negociação na lista e ver se nossa armadilha está sendo executada corretamente?

<!-- aluraframe/client/index.html -->

<!-- código anterior omitido -->

<script>

let negociacaoController = new NegociacaoController();

let lista = new Proxy(new ListaNegociacoes(), {

set(target, prop, value, receiver) {

console.log(`A propriedade ${prop} receberá ${value}`);

return Reflect.set(target, prop, value, receiver);

}

});

// não dispara nosso interceptador!

lista.adiciona(new Negociacao(new Date(), 1, 100));

</script>

Bizarro, nada é exibido no console quando adicionamos uma nova negociação. Deveria, porque o método adiciona altera \_negociacoes.

O problema é que o método não altera a propriedade. Ele não faz this.\_negociacoes = AlgumaCoisa, ele acessa o array e adiciona novos elementos, fazendo com que a alteração passe desapercebida pelo nosso proxy.

Uma solução, não muito performática é alterar a classe ListaNegociacoes, criar um array novo e atribuí-lo a this\_negociacoes. Assim:

// aluraframe/client/js/app/models/ListaNegociacoes.js

class ListaNegociacoes {

constructor() {

this.\_negociacoes = [];

}

adiciona(negociacao) {

// cria um novo array com a concatenação da this com o novo elemento

let lista = [].concat(this.\_negociacoes, negociacao);

this.\_negociacoes = lista;

// this.\_negociacoes.push(negociacao);

}

// código posterior omitido

}

Recarregando nossa página, temos exibido no console:

A propriedade \_negociacoes receberá [object Object]

Nem preciso dizer que essa solução deixa a desejar, pois para cada inclusão estaremos criando uma nova lista. Agora, imagine se fosse possível dispararmos uma armadilha quando o método x ou y for chamado? Isso resolveria nosso problema no que diz respeito a adição de novas negociações. Nesse contexto, são as chamadas de adiciona e esvazia da nossa classe que devem ser interceptadas.

Primeiro, vamos voltar o método adiciona de ListaNegociacoes para o jeito que era antes:

// aluraframe/client/js/app/models/ListaNegociacoes.js

class ListaNegociacoes {

constructor() {

this.\_negociacoes = [];

}

adiciona(negociacao) {

this.\_negociacoes.push(negociacao);

}

// código posterior omitido

}

Por mais estranho que isso possa parecer, precisaremos voltar a interceptar ações get, de leitura. O motivo disso é que chamadas de métodos são consideradas chamadas de leitura para o nosso proxy, mesmo que ele altere alguma propriedade do objeto.

**Construindo armadilhas para métodos**

**Além do que deve ser feito no parágrafo anterior, precisamos ter certeza que a propriedade acessada é um método ou função. A razão disso é porque iremos substituir o método correpondente em nosso proxy por outro que contenha nossa armadilha e a chamada do método original no objeto alvo do proxy.**

Para sabermos se a propriedade é um método/função, acessamos o conteúdo da propriedade através detarget[prop] e verificamos seu tipo com a função typeof. Se for igual à string function, significa que é uma função. Contudo, veja que na comparação não usamos a string, mas o resultado detypeof(Function). É uma boa prática sempre dar preferência para tipos do que trabalhar com strings diretamente:

<!-- aluraframe/client/index.html -->

<!-- código anterior omitido -->

<script>

let negociacaoController = new NegociacaoController();

let lista = new Proxy(new ListaNegociacoes(), {

get(target, prop, receiver) {

if(['adiciona', 'esvazia'].includes(prop) && typeof(target[prop]) === typeof(Function)) {

return function() {

console.log(`Método chamado: ${prop}`);

Reflect.apply(target[prop], target, arguments);

}

}

// só executa se não for função

return Reflect.get(target, prop, receiver);

}

});

lista.adiciona(new Negociacao(new Date(), 1, 100));

console.log(lista.negociacoes)

</script>

Funciona! Agora já podemos aplicar a mesma solução para assim que instanciarmos o modeloListaNegociacoes, já indicarmos a atualização da view toda vez que adiciona e esvazia forem chamados:

// aluraframe/client/js/app/controllers/NegociacoesController.js

class NegociacaoController {

constructor() {

let $ = document.querySelector.bind(document);

this.\_inputData = $('#data');

this.\_inputQuantidade = $('#quantidade');

this.\_inputValor = $('#valor');

this.\_listaNegociacoes = new Proxy(new ListaNegociacoes(), {

get(target, prop, receiver) {

if(['adiciona', 'esvazia'].includes(prop) && typeof(target[prop]) === typeof(Function)) {

return function() {

console.log(`método '${prop}' interceptado`);

Reflect.apply(target[prop], target, arguments);

// como acessar negociações view?

}

}

return Reflect.get(target, prop, receiver);

}

});

}

}

Mas como teremos acesso à this.\_negociacoesView? Nem com this nem mesmo usando *arrow function* garantiremos que o this dentro da nossa armadilha é a nossa view. Para resolver esse problema, vamos declarar uma variável chamada self e nela atribuir this. Nesse ponto temos certeza absoluta que self é uma instância de ListaNegociacoes. Daí em diante, pasta usar a variável em nossa armadilha:

// aluraframe/client/js/app/controllers/NegociacoesController.js

class NegociacaoController {

constructor() {

let $ = document.querySelector.bind(document);

this.\_inputData = $('#data');

this.\_inputQuantidade = $('#quantidade');

this.\_inputValor = $('#valor');

let self = this; // guarda em uma variável o valor de this

this.\_listaNegociacoes = new Proxy(new ListaNegociacoes(), {

get(target, prop, receiver) {

if(['adiciona', 'esvazia'].includes(prop) && typeof(target[prop]) === typeof(Function)) {

return function() {

console.log(`método '${prop}' interceptado`);

Reflect.apply(target[prop], target, arguments);

// acessa o self que a instância de NegociacoesController

self.\_negociacoesView.update(target);

}

}

return Reflect.get(target, prop, receiver);

}

});

}

}

Nosso código funciona e não precisamos mais poluir nossas classes de modelo com código que não lhe dizem respeito. No entanto, se formos aplicar a mesma solução para o modelo Mensagem, teremos que escrever um código complexo. Resolveremos essa complexidade no próximo capítulo. Aliás, nem tente implementar a nossa solução com o modelo Mensagem, pois há um problema que ainda precisaremos resolver, exercite apenas com ListaNegociacoes.