Nossa aplicação já funciona e totaliza o volume no fim da tabela. Mas ainda falta implementar uma regra de negócio no modelo: vamos adicionar a funcionalidade que nos permite esvaziar uma lista.

Atualmente, o arquivo ListaNegociacoes.js está assim:

class ListaNegociacoes {  
  
    constructor() {  
  
        this.\_negociacoes = [];  
    }  
  
    adiciona(negociacoes) {  
  
        this.\_negociacoes.push(negociacao);  
    }  
  
    get negociacoes() {  
  
        return [].concat(this.\_negociacoes);  
    }  
}

Adicionaremos o método esvazia() abaixo da linha do return():

esvazia()   {  
  
    this.\_negociacoes = [];  
}

O *array* de negociações receberá uma nova lista e com isso, apagará todos os itens da anterior. Ao clicarmos no botão "Apagar" do formulário, queremos que as informações sejam apagadas da lista de negociações. Para isto, sabemos que quem atua no modelo é a controller mediante às ações do usuário. Em seguida, no arquivo NegociacaoController.js, adicionaremos o método apaga():

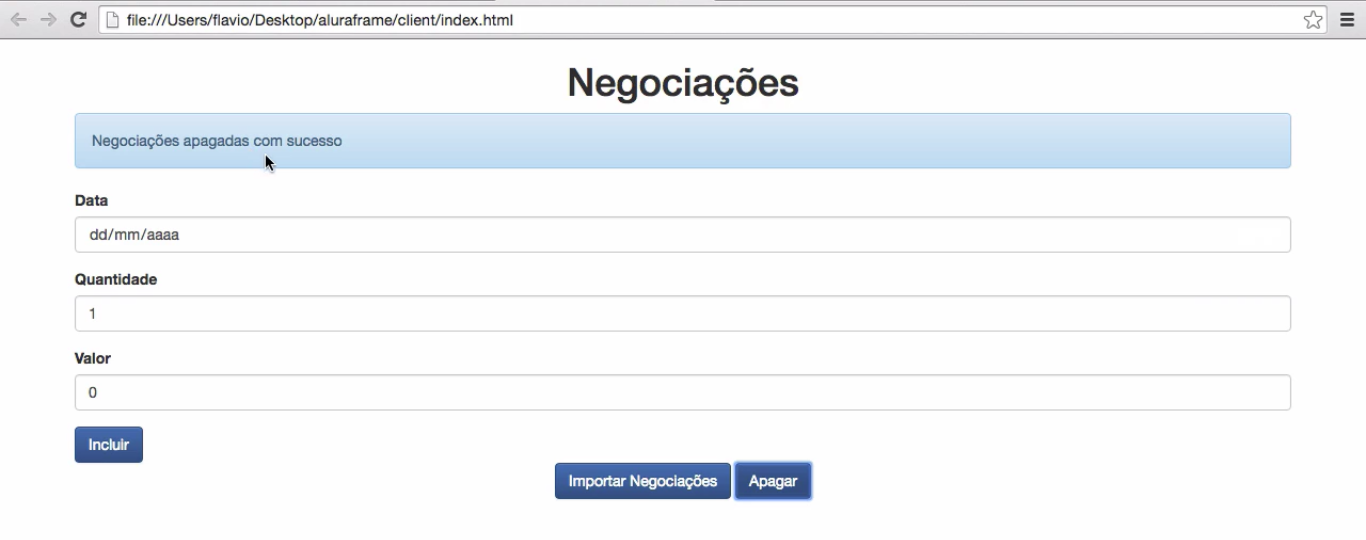
apaga() {  
  
    this.\_listaNegociacoes.esvazia();  
    this.\_negociacoesView.update(this.\_listaNegociacoes);  
  
    this.\_mensagem.texto = 'Negociações apagadas com sucesso';  
    this.\_mensagemView.update(this.\_mensagem);  
}

O método solicitará o modelo \_listaNegociacoes.esvazia(). Observe que adicionamos o update(). Quando atualizarmos a View, esta será recarregada automaticamente e a tabela ficará limpa.

Temos que associar a página a partir do evento de clique, e chamar o apaga(). Para isto, no index.html, adicionaremos o onclick() dentro da tag <button> de "Apagar":

<div class="text-center">  
        <button class="btn btn-primary text-center" type="button">  
            Importar Negociações  
        </button>  
        <button onclick="negociacaoController.apaga()" class="btn btn-primary text-center" type="button">  
            Apagar  
        </button>  
    </div>

Se recarregarmos a página no navegador, nenhum problema será apontado no console e conseguiremos cadastrar uma negociação normalmente. Quando clicarmos em "Apagar", a tabela ficará vazia. A View foi atualizada com os dados do modelo da lista de negociações atualizada e a mensagem foi exibida.



Sempre que atualizarmos o modelo, é possível esquecer de chamar o modelo da View? Sim. Nós temos duas Views: MensagemView e NegociacoesView. É preciso atualizar o modelo nas duas e, depois, chamar o update(). Mas um sistema maior pode conter um número muito superior de Views e corremos o risco de esquecer de adicionar o método. Esta é uma grande responsabilidade para o desenvolvedor.

Será que existe alguma maneira de automatizarmos a atualização da View? Vamos alterar a lista de negociações nos métodos adiciona() e apaga(), quando criarmos a instância de negociações no constructor(). Tentaremos encontrar um solução para que a View seja atualizada automaticamente assim que o modelo for alterado.

Nós queremos alterar o modelo e chamar a atualização da view automaticamente quando ela for recarregada. Para resolver a questão, imagine que somos caçadores. Nós vamos colocar uma armadilha que será disparada quando alguém pisar no código do ListaNegociacoes.js.

Em qual dos métodos seria melhor colocarmos a armadilha, considerando que o nosso objetivo é disparar a atualização da View? A melhor resposta é nos métodos adiciona() e esvazia() porque são elas que modificam as propriedades da View. Ao invocarmos o método adiciona() simultaneamente chamaremos uma armadilha que atualizará a View.

Em seguida, adicionaremos a armadilha como parâmetro de constructor(), também vamos incluir uma propriedade chamada \_armadilha.

class ListaNegociacoes {  
  
    constructor(armadilha) {  
  
        this.\_negociacoes = [];  
        this.\_armadilha = armadilha;  
    }  
//...

A armadilha é uma função, que guardaremos no constructor() para chamá-la posteriormente - ou seja, quando chamarmos o adiciona() e esvazia().

Em NegociacaoController, vamos adicionar o function() dentro de new ListaNegociacao. Após passarmos uma função anônima como parâmetro, ela vai executar a seguinte linha de código:

this.\_negociacoesView.update(this.\_listaNegociacoes);

O trecho do código ficará assim:

class NegociacaoController {  
  
    constructor() {  
  
        let $ = document.querySelector.bind(document);  
        this.\_inputData = $('#data');  
        this.\_inputQuantidade = $('#quantidade');  
        this.\_inputValor = $('#valor');  
        this.\_listaNegociacoes = new ListaNegociacoes(function() {  
  
            this.\_negociacoesView.update(this.\_listaNegociacoes);  
        });  
       //...

Fique tranquilo sobre o NegociacoesView ser declarado depois. O trecho acima só será executado quando os métodos adiciona() e esvazia() forem chamados.

Vamos manter o negociacoesView.update() mais abaixo, para que seja feita a primeira renderização da lista. Porém, vamos apagar a mesma linha do adiciona() e apaga().

adiciona(event) {  
  
    event.preventDefault();  
  
    this.\_listaNegociacoes.adiciona(this.\_criaNegociacao());  
    this.\_mensagem.texto = 'Negociação adicionada com sucesso';  
    this.\_mensagemView.update(this.\_mensagem);  
  
    this.\_limpaFormulario();  
}  
  
// código posterior omitido  
  
apaga() {  
  
    this.\_listaNegociacoes.esvazia();  
    // Linha abaixo comentada, não precisamos mais dela  
    // this.\_negociacoesView.update(this.\_listaNegociacoes);  
  
    this.\_mensagem.texto = "Negociações removidas com sucesso";  
    this.\_mensagemView.update(this.\_mensagem);  
}

Depois, faremos alterações no ListaNegociacoes.js:

adiciona(negociacao) {  
  
    this.\_negociacoes.push(negociacao);  
    this.armadilha(this);  
  
}

No this.armadilha() passamos o *model* como parâmetro, que será acessado com o this. No NegociacaoController.js, adicionaremos o model em \_listaNegociacoes e também no update().

class NegociacaoController {  
  
    constructor() {  
  
        let $ = document.querySelector.bind(document);  
        this.\_inputData = $('#data');  
        this.\_inputQuantidade = $('#quantidade');  
        this.\_inputValor = $('#valor');  
        this.\_listaNegociacoes = new ListaNegociacoes(function(model) {  
  
            this.\_negociacoesView.update(model);  
        });

Criamos o \_listaNegociacoes, passei a função que será chamada quando usarmos o adiciona() e o esvazia(), os dois métodos passaram o modelo como parâmetro. Depois, voltaremos para o ListaNegociacoes.js, e adicionaremos o this.armadilha(this) no método esvazia():

esvazia() {  
  
    this.\_negociacoes = [];  
    this.armadilha(this);  
}

Vamos recarregar a página e testar se nossa armadilha funcionará. Mas ela não vai... No Console, veremos uma mensagem de erro:



Ele nos diz que o this.armadilha não é uma função dentro de ListaNegociacoes.

Teremos que fazer alguns ajustes, primeiramente, adicionaremos o prefixo \_ ao armadilha:

class ListaNegociacoes {  
  
    constructor(armadilha) {  
        this.\_negociacoes = [];  
        this.\_armadilha = armadilha;  
    }  
  
    adiciona(negociacao) {  
        this.\_negociacoes.push(negociacao);  
        this.\_armadilha(this);  
    }  
  
    get negociacoes() {  
        return [].concat(this.\_negociacoes);  
    }  
  
    esvazia() {  
        this.\_negociacoes = [];  
        this.\_armadilha(this);  
    }  
}

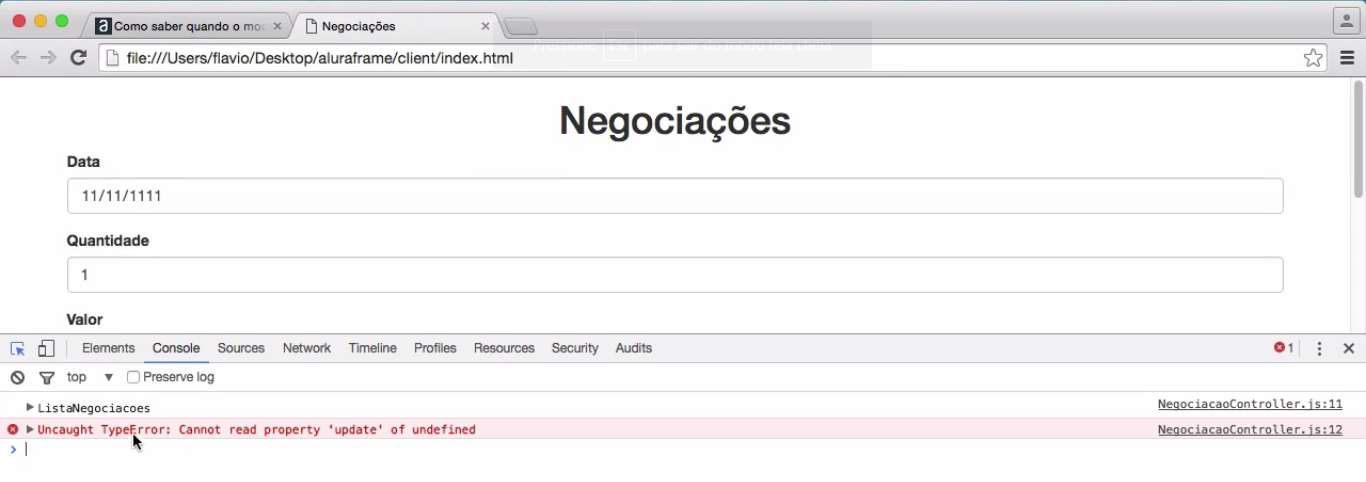
Depois, temos que resolver um problema no constructor de NegociacaoController().

this.\_listaNegociacoes = new ListaNegociacoes(function(model) {  
           this.\_negociacoesView.update(model);

Quando o \_armadilha é executado, o this.\_negociacoesView não existe. Isto ocorre, porque a função é executada em um contexto dinâmico de ListaNegociacoes(). O this dentro de uma função, para ser avaliado, depende do contexto no qual ela foi executada - no caso, o contexto será de ListaNegociacoes. Então, this é a \_listaNegociacoes, porém, esta não tem \_negociacoesView. Para resolver, precisamos que o this seja NegociacaoController, porque toda função JavaScript tem o escopo this dinâmico, que varia de acordo com o contexto. Vamos fazer um teste, adicionando o console.log() em \_listaNegociacoes:

this.\_listaNegociacoes = new ListaNegociacoes(function(model) {  
          console.log(this);  
          this.\_negociacoesView.update(model);

Testaremos preencher o formulário, no Console, veremos outra mensagem de erro:



Ele mostra que this é o ListaNegociacoes. É assim, porque a função está sendo executada no contexto de \_listaNegociacoes. Tem como fazer com o contexto de this seja o NegociacaoController? É o que veremos mais adiante.

Temos uma função com o escopo dinâmico:

class NegociacaoController {

constructor() {

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

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

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

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

this.\_listaNegociacoes = new ListaNegociacoes(function(model) {

this.\_negociacoesView.update(model);

});

//...

O this irá variar de acordo com o contexto da execução. Como a função será chamada dentro da classe ListaNegociacoes, ele será usado como contexto do this da função.

this.\_listaNegociacoes = new ListaNegociacoes(function(model) {

this.\_negociacoesView.update(model);

});

Mas para que o código funcione, queremos que o this tenha como contexto o NegociacaoController. Da mesma maneira que o this é dinâmico, nós programaticamente podemos modificá-lo. Como faremos isso? Primeiramente, no construtor de \_listaNegociacoes, adicionaremos o primeiro parâmetro: this - referente ao NegociacaoController.

class NegociacaoController {

constructor() {

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

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

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

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

this.\_listaNegociacoes = new ListaNegociacoes(this, function(model) {

this.\_negociacoesView.update(model);

});

Este que será recebido depois em ListaNegociacoes.js, como contexto:

class ListaNegociacoes {

constructor(contexto, armadilha) {

this.\_negociacoes = [];

this.\_armadilha = armadilha;

this.\_contexto = contexto;

}

//...

Observe que criamos o atributo \_contexto.

Agora o construtor do model recebe o contexto, no qual queremos que ele execute uma função. Mas para que o thisseja realmente o NegociacaoController, teremos que mudar a maneira de chamar a função utilizada. Pediremos uma ajuda para a **API de Reflexão** do JavaScript, adicionando em ListaNegociacoes.js o Reflect.apply. Com isto, chamaremos o método estático da classe:

adiciona(negociacao) {

this.\_negociacoes.push(negociacao);

//this.\_armadilha(this);

Reflect.apply(this.\_armadilha, this.\_contexto, [this]);

}

get negociacoes() {

return [].concat(this.\_negociacoes);

}

esvazia() {

this.\_negociacoes = [];

//this.\_armadilha(this);

Reflect.apply(this.\_armadilha, this.\_contexto, [this]);

}

Observe que fizemos algumas alterações nos métodos adiciona() e esvazia(). O Reflect.apply recebeu o this.\_armadilha como primeiro parâmetro e o segundo é this.\_contexto. O terceiro parâmetro é o [this], que será a própria ListaNegociacoes. Depois, adicionamos o Reflect.apply() também no esvazia().

Se executarmos o código, o formulário continua funcionando normalmente.



Nós conseguimos redefinir o contexto em que queremos executar a função de \_listaNegociacoes, utilizando o Reflect.apply(). O método apply() recebeu a função executada, depois o contexto e os parâmetros que serão passados para a função. Fizemos isto com o adiciona() e o esvazia().

adiciona(negociacao) {

this.\_negociacoes.push(negociacao);

//this.\_armadilha(this);

Reflect.apply(this.\_armadilha, this.\_contexto, [this]);

}

A função espera receber um modelo no NegociacaoController, dentro do constructor:

this.\_listaNegociacoes = new ListaNegociacoes(this, function(model) {

this.\_negociacoesView.update(model);

});

No método adiciona(), passaremos o [this] que será o ListaNegociacoes. O nosso código está funcionando corretamente, mas será que existe outra maneira de conseguirmos o mesmo efeito, sem ter que passar o contexto para o model? Veremos mais adiante.

Toda vez que um modelo for atualizado, tanto por nós quanto pela aplicação, você se lembrará de chamar o update()na View? É improvável.

Vimos como tirar a responsabilidade do desenvolvedor e colocar a chamada da atualização da View no código, quando o modelo for alterado.

Seguindo o exemplo do listaNegociacoes, em que os métodos adiciona() e esvazia() eram chamados, será esse o momento no qual vamos disparar a atualização da View. Optamos por colocar "armadilhas", funções passadas para o construtor da classe que são chamadas sempre que os métodos adiciona() ou esvazia() forem usados. Estas funções recebem um código que acessa a View da controller e executará o método update().

Mas a solução fracassou, porque a function() era enviada para o construtor do modelo que é a armadilha. Quando isso acontecia, o this era dinâmico, ou seja, não pertencia a controller, e sim, ao model. Então, tentamos acessar no this a nossa View. Vimos como fazer isto por meio da API de reflexão do JavaScript, **Reflection API**, usando reflect.apply(). Ela recebe o nome do método, o contexto em que queremos executar o método, além dos parâmetros que serão passados para o método para corrigir o this no momento da execução da função. Mas vimos que este processo não era necessário. Em vez disso, usamos uma arrow function.

As arrow functions possuem um escopo léxico, enquanto as funções padrões têm um escopo dinâmico. Isto significa que, se temos uma função em JavaScript que varia de acordo com o contexto no qual é chamada, o this léxico de uma arrow function manterá o mesmo this em todas as chamadas da função. Isso torna o código mais enxuto, porque não precisarmos passar o contexto do construtor do modelo.

Nos exercícios, você terá a chance de praticar o conteúdo trabalhado, mas vale ressaltar que poluir o modelo com esse código de infraestrutura não é uma boa ideia. Atacaremos tal questão na próxima seção do curso, e descobriremos como disparar atualizações na View sem colocar "armadilhas". No entanto, os conceitos abordados aqui já devem estar bem consolidados. Vamos praticar!

**Infraestrutura 1 - Chrome!**

Neste treinamento é necessário usar o **Google Chrome versão 50** ou superior. **É importante essa escolha do navegador**, porque usaremos muitos recursos do ES6, inclusive alguns que estão sendo consolidados. Se você está inseguro, não fique. Todos os browsers hoje possuem atualização automática, e desde o IE 10 a atualização do browser na plataforma Windows ficou independente do sistema operacional, isso significa que em pouquíssimas semanas todos os usuários de internet acabam recebendo o browser mais novo.

**ATENÇÃO USUÁRIOS MAC**: nem sempre o Chrome é instalado na língua **português-brasil**. Se o seu navegador exibe todas as opções de menu em inglês, ele exibirá a data da tag <input type="date"> no formato mês/dia/ano. Não há problema nenhum ir até o final do curso neste formato, mas se você quiser que o input exiba a data no formato dia/mês/ano, precisará usar a versão em **português-brasil** do Chrome. Você não precisa baixar novamente o navegador, basta abrir seu terminal e executar o comando:

exec defaults write com.google.Chrome AppleLanguages '(pt-BR)'

Veja que essa solução só é necessária se você deseja a data no formato bonitinho. Durante o treinamento, eu preferi pegar carona neste tipo de input em vez de escrever um código de formatação. Existem zilhões de soluções aí fora para formatação, mas a ideia é usarmos tudo o que o browser oferece.

**DICA: Como saber se determinada funcionalidade do ES2015+ (ES6) é suportada por cada navegador?**

Durante o treinamento, evitarei dizer que determinado aspecto só funciona nesse ou naquele browser, porque pode ser que na semana seguinte o browser X passe a suportá-la. Contudo, para que o aluno tenha a informação mais recente dos recursos que utilizarei neste treinamento, ele pode consultar sempre que desejar: <https://kangax.github.io/compat-table/es6/>.

Não ache que os recursos que lhe mostrarei são experimentais, muito pelo contrário, eles já fazem parte da especificação ES6. Inclusive o conhecimento aqui adquirido pode ser aplicado na plataforma Node.js v6.0 ou superior, exceto a parte de manipulação de DOM. Além disso, se você desenvolve usando algum tipo de *transpiler*, como Babel ou até mesmo o famoso TypeScript, também será beneficiado. Com essas tecnologias, escrevemos um código em ES6 e no final entregamos para o navegador o mesmo código convertido para ES5, para garantir máxima compatibilidade. Sendo assim, tudo o que você aprender aqui pode ser usado.

**Infraestrutura 2 - Node.js**

Em um certo ponto do nosso treinamento, precisaremos um servidor web que disponibilize URL's para serem consumidas pela nossa aplicação. Já disponibilizamos um para você dentro do projeto. Para que ele funcione, você precisa ter no mínimo o Node.js **v4.0** instalado.

Você pode baixar o Node.js da sua plataforma preferida (Windows, MAC ou Linux) em [https://nodejs.org](https://nodejs.org/). Depois de instalá-lo, para saber se ele está funcionando, basta abrir seu terminal preferido da sua plataforma e executar o comando (logo a seguir há dicas para o caso de o comando não funcionar):

node --version

Este comando deve exibir a versão do Node instalada no terminal. Se por acaso o comando node não for um comando válidos tente o seguinte:

1 - Windows: não mude o diretório padrão da instalação do Node.js. Há relatos que em algumas versões do Windows a pasta do Node não é colocada no *PATH* do Windows, sendo necessário adicioná-la manualmente. Não sabe como? Temos um treinamento de [prompt no Windows](https://cursos.alura.com.br/course/prompt) que pode ajudá-lo nesta tarefa.

2 - Linux: algumas distribuições Linux já possui um binário chamado **node**, que não tem nenhuma relação com o Node.js. Nestas distribuições, o binário passa a se chamar **nodejs**. Sendo assim, em todo lugar que eu referenciar o comando **node** ele deve ser trocado para **nodejs**.

A ideia desse exercício é deixar ainda mais clara a diferença do this de uma *arrow function* do this de uma função tradicional em JavaScript. Sugiro fortemente que você crie os arquivos em um projeto separado, para poder ver o que acontece além da teoria.

Vamos começar por um exemplo clássico. Temos três elementos distintos em nossa página e queremos exibir o conteúdo de cada um deles.

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Document</title>

</head>

<body>

<h1>TITULO</h1>

<p>PARAGRAFO</p>

<div>DADOS</div>

<script>

console.log(this); // é window

let exibeConteudo = function() {

console.log(this);

alert(this.textContent);

};

$ = document.querySelector.bind(document);

$('h1').addEventListener('click', exibeConteudo);

$('p').addEventListener('click', exibeConteudo);

$('div').addEventListener('click', exibeConteudo);

</script>

</body>

</html>

Perfeito, quando clicamos em cada um deles, exibimos no console o valor de this, inclusive exibimos um alerta com conteúdo de cada elemento. Repare que o this é dinâmico, ou seja, **seu valor é definido no momento em que a função é chamada, jamais no momento em que é declarada**. Quando clicamos no h1, o this será este elemento, quando clicamos em p, o this será o elemento. Ainda bem que isso acontece, pois se o this não fosse dinâmico, não conseguiríamos escrever uma função genérica como a nossa.

Que tal declararmos nossa função como uma *arrow function*, que é menos verbosa? Alterando nosso código:

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Document</title>

</head>

<body>

<h1>TITULO</h1>

<p>PARAGRAFO</p>

<div>DADOS</div>

<script>

console.log(this); // é window

// arrow function agora!

let exibeConteudo = () => {

console.log(this);

alert(this.textContent);

};

$ = document.querySelector.bind(document);

$('h1').addEventListener('click', exibeConteudo);

$('p').addEventListener('click', exibeConteudo);

$('div').addEventListener('click', exibeConteudo);

</script>

</body>

</html>

Um teste demonstra que nosso código deixa de funcionar. Primeiro, independente do elemento que eu clique, o this que é impresso no console é window e não aquele elemento do DOM. Segundo, como this é window e ele não possui a propriedade textContent, é exibido undefined para o usuário. Esse problema serve para demonstrar que uma *arrow function* vai além de uma sintaxe mais enxuta para declararmos funções.

Diferente de uma função, que possui um this dinâmico, uma *arrow function* possui um this estático, ou seja, que nunca muda e que é determinado no momento em que é declarado! Veja que quando declararmos nossa *arrow function*, ela vai considerar o this do local onde é declarada. Sendo assim, como o this dentro da tag <script> é window, ela adotará window.

Resumindo:

* O this de uma função é dinâmico, isto é, seu valor é determinado no momento em que a função é chamada. Como o this é dinâmico, é possível usar artifícios da linguagem, como a API Reflect, para alterá-lo se assim desejarmos.
* O this de uma *arrow function* é léxico, isto é, seu valor é determinado no local onde a *arrow function* for definida, ela não cria um novo this. O this de uma *arrow function* não pode ser alterado, mesmo se usarmos recursos da linguagem, como a API Reflect.

No contexto que vimos acima, a *arrow function* atrapalhou mais do que ajudou. Mas vejamos um exemplo em que seu escopo léxico torna-se **muito** interessante:

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Document</title>

</head>

<body>

<script>

class SistemaImpressao {

constructor() {

this.\_codigo = 2;

}

imprime(nomes) {

nomes.forEach(function(nome) {

console.log(this);

console.log(`${this.\_codigo}: ${nome}`);

});

}

}

let nomes = ['Flávio', 'Nico', 'Douglas'];

let si = new SistemaImpressao();

si.imprime(nomes);

</script>

</body>

</html>

Temos a seguinte classe SistemaImpressao, que possui o método imprime. O método recebe uma lista e para cada item da lista imprime primeiro a versão do sistema, seguido do item. O problema é que o this.\_codigo acessado em nosso forEach não é de uma instância da classe SistemaImpressao, aliás, ele é undefined. Contudo, se usarmos *arrow function*, o this usado no forEach usará o this do contexto no qual foi declarado.

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Document</title>

</head>

<body>

<script>

class SistemaImpressao {

constructor() {

this.\_codigo = 2;

}

imprime(nomes) {

// usando arrow function.

nomes.forEach(nome => {

console.log(this);

console.log(`${this.\_codigo}: ${nome}`);

});

}

}

let nomes = ['Flávio', 'Nico', 'Douglas'];

let si = new SistemaImpressao();

si.imprime(nomes);

</script>

</body>

</html>

Agora nosso código funciona!

Arrow functions ajudam bastante quando aplicamos o paradigma orientado a objetos, pois o this passa a se comportar como o esperado, especialmente em outras linguagens de programação, como Java e C#.

Um programador de Javascript decidiu criar uma abstração de um relógio em seu código, através da classe Relogio. A ideia é a seguinte: assim que ele instanciar essa classe, deve ser exibida no console a quantidade de segundos a partir da sua criação. Aprendemos que o constructor é chamado toda vez que instanciamos objetos de uma classe:

class Relogio {  
  
    constructor() {  
        this.\_segundos = 0;  
  
        setInterval(function () {  
            console.log(++this.\_segundos);  
          }, 1000);  
    }  
}  
  
var relogio = new Relogio();

O problema é que a mensagem exibida no console é NaN *(****N****ot* ***a******N****umber)* e não os segundos desde que o relógio foi instanciado.

Encontre o erro deste código e pense em uma solução, utilizando o que aprendemos neste capítulo, para que o relógio se comporte como o esperado.

O principal erro deste código está na função passada para setInterval, aonde fazemos o ++this.\_segundos. Como setInterval é global e acessível em qualquer canto do nosso código, ele pertence ao objeto global window, logo seu this aponta para window e não para nossa classe Relogio. Como window não possui a propriedade \_segundos, o incremento resultará em NaN, pois não podemos incrementar uma variável que é undefined.

Uma maneira de resolver esse problema é guardando o this do constructor em uma variável, por exemplo, self e acessá-la quando necessário. Vejamos:

class Relogio {  
  
    constructor() {  
  
        let self = this; // guardando o this que é a instância da classe `Relogio`  
        this.\_segundos = 0;  
  
        setInterval(function () {  
            console.log(++self.\_segundos); // acessando a variável self, que é a instância de `Relogio`   
          }, 1000);  
  
    }  
}

Contudo, aprendemos a usar *arrow functions* nesse capítulo, que além de nos fornecer uma sintaxe bastante enxuta, ainda possui escopo léxico, isto é, seu this é estático e não muda.

class Relogio {  
  
    constructor() {  
        this.\_segundos = 0;  
        setInterval(() => console.log(++this.\_segundos), 1000); // usando arrow function. O this é o this de `Relogio`, e não `window`.   
    }  
}  
  
var relogio = new Relogio();

Assim, ao alterar o valor de this.\_segundos, estamos ainda no contexto de Relogio e o valor é acrescido corretamente. E deste modo o código ficou ainda mais sucinto!

Neste capítulo você aprendeu gradativamente como resolver o problema do contexto de invocação da nossa armadilha passada para nosso modelo. A primeira solução foi passar o contexto de invocação como parâmetro e usar a mágica de Reflect.apply para alterar o contexto de invocação:

class NegociacaoController {  
  
    constructor() {  
  
        // código anterior omitido  
  
        this.\_listaNegociacoes = new ListaNegociacoes(this, function(model) {   
  
            this.\_negociacoesView.update(model);  
        });  
    }  
  
    // código posterior omitido  
}

Depois, aprendemos que o uso de uma arrow function resolvia essa questão, porque o valor de seu this é definido no local onde ela é declarada em nosso código e seu valor não muda (diferente do this de uma function que é dinâmico). Sendo assim, o this da nossa armadilha será a instância de NegociacaoController:

class NegociacaoController {  
  
    constructor() {  
  
        // código anterior omitido  
  
        this.\_listaNegociacoes = new ListaNegociacoes(model =>   
            this.\_negociacoesView.update(model));  
    }  
  
    // código posterior omitido  
}

Essa solução evitou passarmos mais um parâmetro em nosso modelo, o contexto de invocação e ficou mais compacta e elegante.

Contudo, há ainda uma terceira solução, mas envolve a declaração de uma variável extra. Podemos guardar uma referência para a instância de NegociacaoController em uma variável. Geralmente essa variável é chamada de self:

class NegociacaoController {  
  
    constructor() {  
  
        // a variável self guarda uma referência para this, instância de NegociacaoController  
  
        let self = this;  
  
        // aqui usei uma function tradicional, mas poderia ser uma arrow function também  
  
        this.\_listaNegociacoes = new ListaNegociacoes(function(model) {   
            self.\_negociacoesView.update(model);  
        });  
    }  
  
    // código posterior omitido  
}

Quando nossa armadilha for executada, o self será nosso NegociacaoController.

Esse é mais um exercício de revisão. Não é necessário respondê-lo, no entanto praticar esse código vale a pena.

Temos dois objetos criados de maneira literal por uma questão de brevidade, mas que poderiam ser instâncias de uma classe:

let objeto1 = {  
  
    nome: 'Bob'  
};  
  
let objeto2 = {  
  
    nome: 'Leo'  
}

Temos a seguinte função:

function exibeNome() {  
  
    alert(this.nome);  
}

O que acontecerá se chamarmos exibeNome? O resultado será undefined, porque o this da função, ou seja, seu contexto não possui a propriedade nome.

Agora, que tal chamarmos a função exibeNome, mas indicando que seu contexto de execução será objeto1? Vejamos:

Reflect.apply(exibeNome, objeto1, []); // exibe 'Bob'

O resultado será o alerta sendo exibido com o texto Bob. Podemos executar a função agora fazendo com que o seu this (contexto) seja objeto2:

Reflect.apply(exibeNome, objeto2, []); // exibe 'Leo'

Como Reflect.apply funciona? O primeiro parâmetro é o método ou função que desejamos invocar. O segundo parâmetro é o contexto que o método ou função adotará, ou seja, o valor que será assumido pelo this. Por fim, o último parâmetro é um array que contém todos os parâmetros que o método passado como primeiro parâmetro receberá. Como ele não recebe parâmetro nenhum, passamos um array vazio.

Vamos alterar nossa função para receber dois parâmetros. O primeiro será um prefixo que será adicionando no nome e o último um sufixo:

function exibeNome(prefixo, sufixo) {  
  
    alert(prefixo + this.nome + sufixo);  
}

Agora, vamos chamar o método através de Reflect.apply:

Reflect.apply(exibeNome, objeto1, ['(', ')']); // exibe '(Bob)'

Veja que agora estamos passando dois parâmetros para o método.

Esse exercício é de apenas reflexão. Você pode executar seu código se assim desejar.

Temos as seguintes classes:

<!-- troca-troca.html -->  
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="UTF-8">  
    <title></title>  
</head>  
<body>  
    <p class="info">Era uma vez...</p>  
    <script>  
  
        class Aviao {  
  
            constructor(nome) {  
                this.\_nome = nome;  
            }  
  
            voa() {  
                alert(`${this.\_nome} está voando`);  
            }  
  
            ligaMotor() {  
                console.log('liga o motor');  
            }  
  
            fechaPortas() {  
                console.log('Portas sendo fechadas');  
            }  
        }  
  
        class Passarinho {  
  
            constructor(nome) {  
                this.\_nome = nome;  
            }  
  
            voa() {  
                // hum..precisamos implementar esse método também!  
            }  
        }  
  
    </script>  
</body>  
</html>

Veja que o método voa de Passarinho não esta completo. Podemos até usar herança e herdar de Aviao, mas com certeza um passarinho não ligaMotor nem fechaPortas. Não podemos usar herança porque Passarinho não é um Aviao.

## Reutilização de código através de composição

Uma maneira de solucionar esse problema é usar composição no lugar de herança. Na composição, a classe que deseja usar o método de outra possui uma instância dessa classe. Por mais que a instância tenha vários métodos, só chamamos aqueles que nos interessa:

Alterando nosso código para usar composição:

<!-- troca-troca.html -->  
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="UTF-8">  
    <title></title>  
</head>  
<body>  
    <p class="info">Era uma vez...</p>  
    <script>  
  
        class Aviao {  
  
            constructor(nome) {  
                this.\_nome = nome;  
            }  
  
            voa() {  
                alert(`${this.\_nome} está voando`);  
            }  
  
            ligaMotor() {  
                console.log('liga o motor');  
            }  
  
            fechaPortas() {  
                console.log('Portas sendo fechadas');  
            }  
        }  
  
        class Passarinho {  
  
            constructor(nome) {  
                this.\_nome = nome;  
                // guarda uma instância de avião  
                this.\_aviao = new Aviao(nome);  
            }  
  
            voa() {  
                // usa o método voa de Aviao  
                this.\_aviao.voa();  
            }  
        }  
  
    </script>  
</body>  
</html>

Nessa solução, quem usa a instância da classe Passarinho nem sabe que o método voausa por debaixo dos panos uma instância de Aviao para funcionar. Veja que a composição tem a vantagem de podermos escolher quais métodos queremos reaproveitar, diferente da herança que é tudo ou nada. Contudo, veja que com composição precisamos escrever um pouco mais, pois temos que delegar as chamadas dos métodos voa de Passarinho para o voa de avião.

Ainda há outra forma de resolver este problema sem usar herança nem composição, mas usando **mixin**!

## Reutilização de código através de mixin!

Com mixin podemos "pegar emprestado" o método de outra classe sem termos que ter uma instância dessa classe como é o caso de composição.

Vamos alterar a classe Passarinho removendo a instância de Aviao:

<!-- troca-troca.html -->  
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="UTF-8">  
    <title></title>  
</head>  
<body>  
    <p class="info">Era uma vez...</p>  
    <script>  
  
        class Aviao {  
  
            constructor(nome) {  
                this.\_nome = nome;  
            }  
  
            voa() {  
                alert(`${this.\_nome} está voando`);  
            }  
  
            ligaMotor() {  
                console.log('liga o motor');  
            }  
  
            fechaPortas() {  
                console.log('Portas sendo fechadas');  
            }  
        }  
  
        class Passarinho {  
  
            constructor(nome) {  
                this.\_nome = nome;  
            }  
  
            voa() {  
                // executa o método `voa` de `Avião` usando como contexto a instância de `Passarinho`  
                Reflect.apply(Aviao.prototype.voa, this, []);  
            }  
        }  
  
    </script>  
</body>  
</html>

Olha ai o Reflect.apply novamente! Nesta linha de código estamos querendo executar o método voa da classe Aviao, mas usando como contexto o this da instância de Passarinho. O último parâmetro é um array que contém os parâmetros do método. Como voa não recebe parâmetro algum, passamos um array vazio.

Um detalhe: foi necessário fazer Aviao.**prototype**.voa porque métodos criados usando ES6 são adicionados no prototype. Qualquer método adicionando em prototype estará disponível para todas as instâncias.

Reflect é um artefato novo do ES2015! Contudo, poderíamos conseguir o mesmo resultado dessa forma:

// No ES5  
voar() {  
    Aviao.prototype.voa.apply(this, []);  
}

Essa solução é menos verbosa, mas a ideia do ES2015 foi centralizar operações como essa em Reflect e com o tempo esse deve se tornar o método mais indicado para realizar operações como essa.

Aprendemos a alterar o contexto de execução de uma função/método através de Reflect.apply. Vejamos um trecho de código que faz isso:

class ListaNegociacoes {  
  
    constructor(contexto, armadilha) {  
        this.\_negociacoes = [];  
        this.\_armadilha = armadilha;  
        this.\_contexto = contexto;  
    }  
  
    adiciona(negociacao) {  
        this.\_negociacoes.push(negociacao);  
        Reflect.apply(this.\_armadilha, this.\_contexto, [this]);  
    }  
  
    get negociacoes() {  
        return [].concat(this.\_negociacoes);  
    }  
  
    esvazia() {  
        this.\_negociacoes = [];  
        Reflect.apply(this.\_armadilha, this.\_contexto, [this]);  
    }  
}

O artefato Reflect é algo novo do ES6, contudo poderíamos conseguir com ES5 o mesmo resultado, assim:

class ListaNegociacoes {  
  
    constructor(contexto, armadilha) {  
        this.\_negociacoes = [];  
        this.\_armadilha = armadilha;  
        this.\_contexto = contexto;  
    }  
  
    adiciona(negociacao) {  
        this.\_negociacoes.push(negociacao);  
        this.\_armadilha.apply(this.\_contexto, [this]);  
    }  
  
    get negociacoes() {  
        return [].concat(this.\_negociacoes);  
    }  
  
    esvazia() {  
        this.\_negociacoes = [];  
        this.\_armadilha.apply(this.\_contexto, [this]);  
    }  
}

Veja que estamos chamando o método apply, mas diretamente na função atribuída à propriedade this.\_armadilha. O resultado é o mesmo quando usamos Reflect.apply.

Apesar de um pouco menos verbosa, a ideia é que operações como essa sejam feitas através de Reflect, pois é uma primeira tentativa da linguagem JavaScript de centralizar funcionalidades como vimos em um único lugar.

Achei interessante mostrar como esse procedimento é feito no ES5 para que o aluno entenda que o ES6, além de trazer muitas novidades, procura padronizar bastante coisa da própria linguagem.

ES6 não veio criar apenas novas funcionalidades, ele veio também padronizar o que já existia.

No final deste capítulo implementamos uma solução para automatizar o processo de atualização da view toda vez que o modelo mudar. O que fizemos na verdade foi implementar o padrão de projeto **Observer**.

## O padrão de projeto Observer

Usamos o padrão de projeto **Observer** sempre que queremos notificar partes do sistema interessadas quando um evento importante for disparado em nosso sistema.

No contexto da nossa aplicação, entendemos um evento como o ato de adicionar ou esvaziar nossa lista de negociações. É a view que está interessada em observar esse evento e tomar uma ação, no caso, se atualizar com base no estado mais atual do modelo.

Para liberarmos o desenvolvedor da responsabilidade de atualizar programaticamente a View sempre que o modelo fosse atualizado, nós colocamos "armadilhas": funções que eram chamadas quando métodos específicos eram executados. Desta forma, chamávamos automaticamente a atualização da View. Nós declaramos o modelo no inicio, definimos as ações que deveriam acontecer quando ocorria a modificação, e assim liberamos o desenvolvedor da responsabilidade.

No entanto, esta solução deixa a desejar porque coloca código de infraestrutura - ou seja, de atualização da View - no modelo. Geralmente, a parte mais reutilizada de um sistema é o modelo. Então, ao acessarmos um modelo de negociação e encontrarmos um atributo chamado \_armadilha, por exemplo:

class ListaNegociacoes {  
  
    constructor(armadilha) {  
  
        this.\_negociacoes = [];  
        this.\_armadilha = armadilha;  
    }  
//...

O que \_armadilha tem a ver com a lista de negociação? Ela foi usada apenas para aplicar o artifício que chama a View automaticamente. E se tivéssemos outros métodos que quiséssemos monitorar e executar uma armadilha? Teríamos que alterar a classe do modelo. Então, o modelo é a parte mais reutilizável. Se agora não quisermos mais utilizar um sistema baseado em MVC, podemos optar em usar o AngularJS ou outro framework.

Mas se começamos a incluir diversos itens de infraestrutura, de recursos para que ela gere benefícios - como a atualização de View - começamos a não reutilizar continuamente o modelo. Encontraremos uma forma de manter o modelo intacto, sem utilizarmos armadilhas e ainda assim, conseguir executar um código arbitrário quando algum método for chamado. A seguir, encontraremos uma solução para a questão.

Veremos qual é a solução que nos permite manter o modelo... Começaremos retirando o \_armadilha de ListaNegociacoes:

class ListaNegociacoes {  
  
  constructor() {  
  
        this.\_negociacoes = [];  
  }  
  adiciona(negociacao) {  
      this.\_negociacoes.push(negociacao);  
  }  
  
  get negociacoes() {  
      return [].concat(this.\_negociacoes);  
  }  
  
  esvazia() {  
      this.\_negociacoes = [];  
  }  
}

Como removemos o \_armadilha, o construtor de NegociacaoController deixará de funcionar e descobriremos uma forma de resolver problema da View. Temos ainda outro problema com a solução que usa o \_armadilha: se quisermos monitorar os models Mensagem e Negociacoes, teremos que abrir a classe para alterar e colocar a armadilha - mas, não faremos isto.

Existe um famoso padrão de projeto chamado **Proxy**, que de forma resumida, é "um cara mentiroso". Vimos que não é bom inserirmos armadilhas na classe, porque estaremos perdendo a reutilização do modelo e teremos que repetir em todos os modelos do sistema. No entanto, o Proxy é idêntico ao objeto que queremos trabalhar, e teremos bastante dificuldade de diferenciá-los. Nós acessamos o Proxy como se ele fosse o objeto real, este último ficará escondido dentro do outro. Nós substituímos o objeto real, que só poderá ser acessado por meio do Proxy - que pode ou não ser executado em um código arbitrário se assim definirmos.

Observe que ListaNegociacoes tem o métodos adiciona() e negociacoes(), que também estarão presentes no Proxy.

class ListaNegociacoes {  
  
  constructor(armadilha) {  
  
        this.\_negociacoes = [];  
        this.\_armadilha = armadilha;  
  }  
  adiciona(negociacao) {  
      this.\_negociacoes.push(negociacao);  
      this.\_armadilha(this);  
  }  
  
  get negociacoes() {  
      return [].concat(this.\_negociacoes);  
  }  
  
  esvazia() {  
      this.\_negociacoes = [];  
      this.\_armadilha(this);  
  }

A diferença está em que quando chamarmos o adiciona(), o Proxy delegará a chamada do método para o objeto encapsulado por ele. Mas ainda não temos benefícios com esta mudança. A vantagem está que colocaremos as armadilhas **entre** a chamada do Proxy e o objeto real. Toda vez que acessamos o Proxy, executaremos um código antes de chamarmos um método ou propriedade correspondente ao objeto real.

A boa notícia é que não precisamos implementar esse padrão de projeto.

A partir da versão 2015 do ECMAScript, a própria linguagem já possui um recurso de Proxy. Então, implementaremos o padrão de projeto Proxy usando o ES6.

Vamos aprender a trabalhar com o Proxy. Começaremos comentando o código de NegociacaoController, porque ele não está válido devido a remoção da armadilha passada para o construtor.

class NegociacaoController {  
  
    constructor() {  
  
        let $ = document.querySelector.bind(document);  
        this.\_inputData = $('#data');  
        this.\_inputQuantidade = $('#quantidade');  
        this.\_inputValor = $('#valor');  
  
        /\*  
        this.\_listaNegociacoes =   
            new ListaNegociacoes(model => this.\_negociacoesView.update(model));  
            \*/  
//...

Depois, abriremos o Console no navegador. Veremos uma mensagem de erro, porque a negociação está com problema. Começaremos criando um negociacao:

let negociacao = new Negociacao(new Date(), 1, 100);  
undefined  
let negociacaoProxy = new Proxy(negociacao, {});

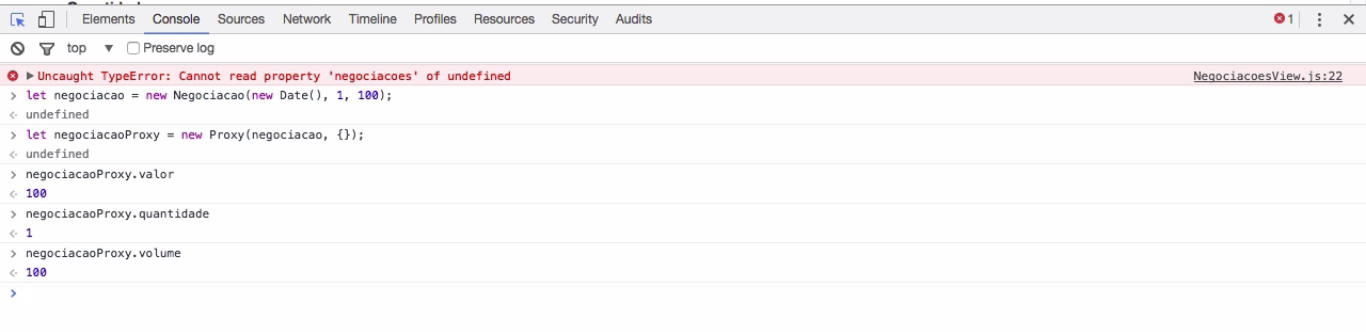
Criaremos um Proxy de negociacao. O segundo parâmetro é um objeto no formato literal {}, em que iremos configurar nossas armadilhas. Em seguida, digitaremos a seguinte linha:

negocicacaoProxy.valor  
100

Ao acessarmos valor, o retorno será 100. Conseguimos ter acesso a quantidade e volume:

negocicacaoProxy.valor  
100  
negociacaoProxy.quantidade  
1  
negocicacaoProxy.volume  
100

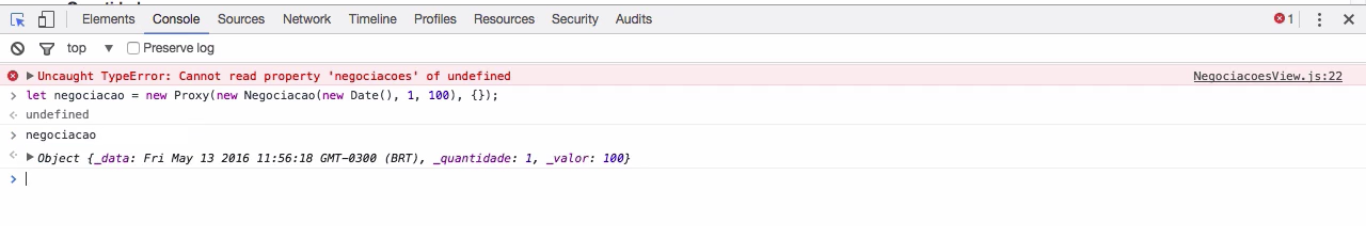
O Proxy terá exatamente o mesmo comportamento do objeto.



No entanto, não queremos que ninguém tenha acesso ao objeto real (negociacao), caso contrário, ninguém cairá nas armadilhas. Para isto, executaremos a seguinte linha no Console:

let negociacao = new Proxy(new Negociacao(new Date(), 1, 100), {});

Neste caso, negociacao é o Proxy:



Agora, a única maneira de lidar com a instância de negociação criada é por intermédio do Proxy. Precisamos aprender como passamos o *handlers* ({}) para colocarmos nossas armadilhas. Adicionaremos outra tag <script> no index.html e depois, colocaremos o código que executamos no Console.

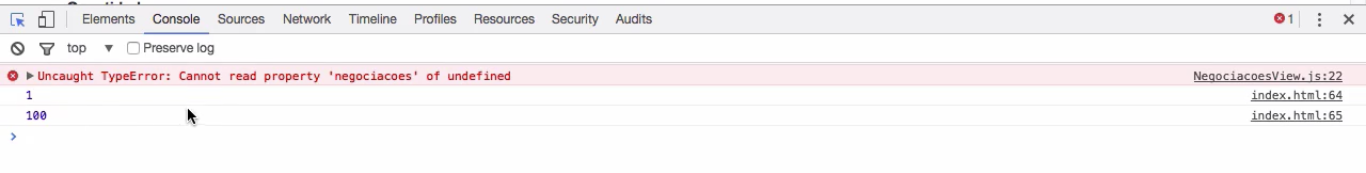
<script>  
  
    let negociacao = new Proxy(new Negociacao(new Date(), 2, 100), {  
  
      });  
  
</script>

Mais adiante, definiremos o *handler*.

Nós já criamos uma Proxy. Em seguida, adicionaremos o console.log:

<script>  
    let negociacao = new Proxy(new Negociacao(new Date(), 1, 100), {});  
  
    console.log(negociacao.quantidade);  
    console.log(negociacao.valor);  
  
</script>

Se executarmos o código, veremos os valores 1 e 100 impressos no Console.

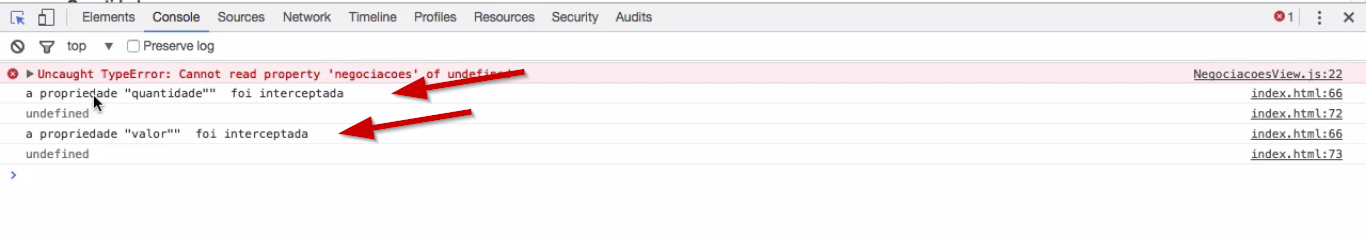


No entanto, queremos executar um código antes de exibirmos o valor de quantidade, queremos que seja visualizado um texto informando que a quantidade foi acessada. Faremos com que o texto também seja exibido antes da propriedade valor. A seguir, adicionaremos uma função que receberá três parâmetros: target, prop e receive.

<script>  
    let negociacao = new Proxy(new Negociacao(new Date(), 2, 100), {  
  
        get: function(target, prop, receiver) {  
  
              console.log(`a propriedade "${prop}" foi inteceptada`);  
        }  
    });  
  
    console.log(negociacao.quantidade);  
    console.log(negociacao.valor);  
  
  </script>

O getserá chamado sempre que tentarmos ler uma das propriedades do objeto. Ao ser chamado, ele tem o target(uma referência ao objeto original que está encapsulado pelo Proxy), a propriedade (prop) que está sendo acessada, e uma referência (receiver) para o Proxy. Agora, antes de ser exibidos os valores das propriedades, os textos devem ser exibidos.

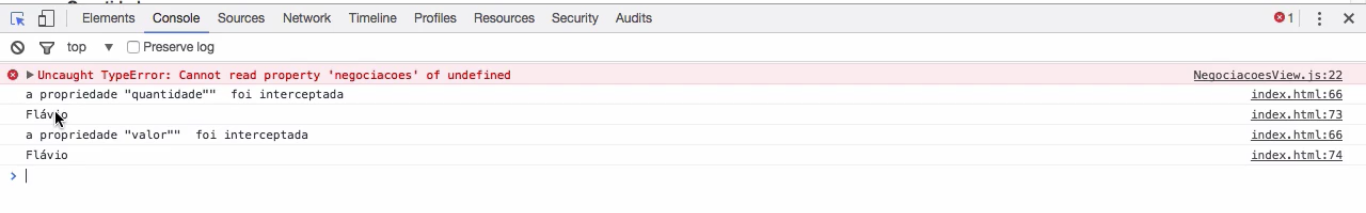
No navegador, vamos obter o resultado esperado:



Porém, os valores das propriedades resultaram em undefined. Isto ocorreu, porque quando executamos uma armadilha (*trap*, traduzido para o inglês), é necessário informar qual será o valor retornado após a interceptação da propriedade de leitura. Adicionaremos o return:

 <script>  
  
       let negociacao = new Proxy(new Negociacao(new Date(), 1, 100), {  
  
           get: function(target, prop, receiver) {  
  
                 console.log(`a propriedade "${prop}" foi interceptada`);  
                 return 'Flávio';  
           }  
       });  
       console.log(negociacao.quantidade);  
       console.log(negociacao.valor);  
 </script>

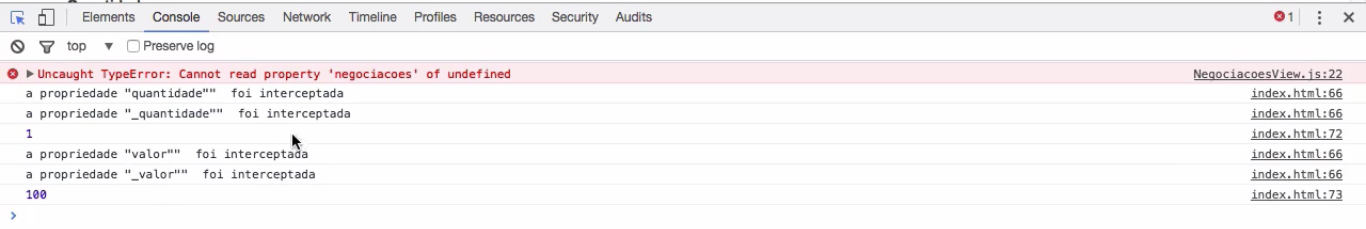
Veja o que será exibido no Console:



Quando acessamos negociacao.valor, ele retorna Flávio. Mas não é o que queremos... Queremos o valor verdadeiramente guardado. Para isto, vamos pedir auxilio para a API de Reflect.get() e os três parâmetros. Nós queremos executar uma operação de leitura.

get: function(target, prop, receiver) {  
  
     console.log(`a propriedade "${prop}" foi interceptada`);  
     return Reflect.get(target, prop, receiver);  
}

No navegador, ele imprimirá o texto e os valores das propriedades.



Mas por que a mensagem foi impressa duas vezes, com um pequena diferença. Isto ocorre, porque no arquivo Negociacao.js, ele irá interceptar para quantidade e \_quantidade.

get quantidade() {  
  
    return this.\_quantidade;  
}  
  
get valor() {  
  return this.\_valor;  
}

O mesmo acontecerá com valor e \_valor. Então nosso código funciona.

Nós vimos como executar um código, antes da leitura das propriedades do objeto. Mas para resolvermos o problema da atualização automática da View - lembrando que não queremos atualizá-la enquanto estivermos lendo um dado. Mas este não é o nosso foco. O objetivo é encontrar uma forma de executar o código quando uma propriedade é modificada. Veremos mais adiante.

Não estamos interessados em executar um código quando ocorrer a leitura, e sim, quando acontecer a modificação de alguma propriedade. Em seguida, faremos pequenas alterações no arquivo index.html. Atualmente, o get está assim:

<script>  
  
    let negociacao = new Proxy(new Negociacao(new Date(), 1, 100), {  
    get: function(target, prop, receiver) {  
  
         console.log(`a propriedade "${prop}" foi interceptada`);  
         return Reflect.get(target, prop, receiver);  
    }  
    });  
    negociacao.quantidade = 10;  
    negociacao.valor = 100;  
</script>

Mas esse código ainda não funcionará, porque quantidade e valor em Negociacao.js são getters. Por isso, não podemos fazer uma atribuição, considerando que são apenas leitura e não podem ser alterados. De volta ao index.html, faremos uma "licença poética" e acessaremos diretamente as propriedades *privates*.

negociacao.\_quantidade = 10;  
negociacao.\_valor = 100;

Vamos desrespeitar a convenção da nossa propriedade, mas com isso, poderemos disparar a nossa armadilha.

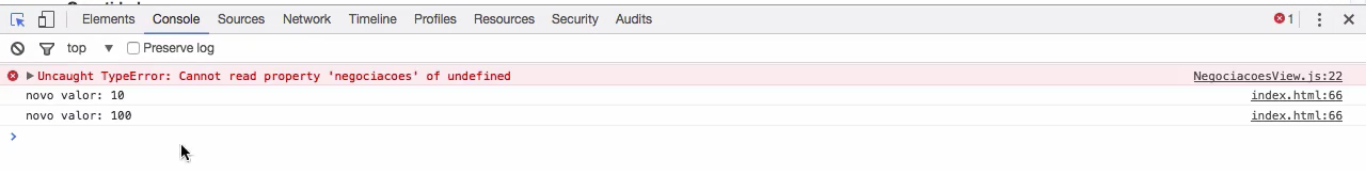
A primeira coisa que devemos fazer para executar uma armadilha quando estou atribuindo, é alterarmos de get para set. Também adicionaremos outro parâmetro: value.

<script>  
  
    let negociacao = new Proxy(new Negociacao(new Date(), 1, 100), {  
    set: function(target, prop, value, receiver) {  
  
         console.log(`a propriedade "${prop}" foi interceptada`);  
         return Reflect.set(target, prop, value, receiver);  
    });  
    negociacao.\_quantidade = 10;  
    negociacao.\_valor = 100;  
</script>

Quando atribuímos 10 para \_quantidade, a função no set será chamada e retornará os quatro parâmetros. No console.log() queremos exibir o valor atual e o que será exibido depois.

<script>  
  
    let negociacao = new Proxy(new Negociacao(new Date(), 1, 100), {  
    set: function(target, prop, value, receiver) {  
  
         console.log(`novo valor: ${value}`);  
         return Reflect.set(target, prop, value, receiver);  
}  
    });  
    negociacao.\_quantidade = 10;  
    negociacao.\_valor = 100;  
</script>

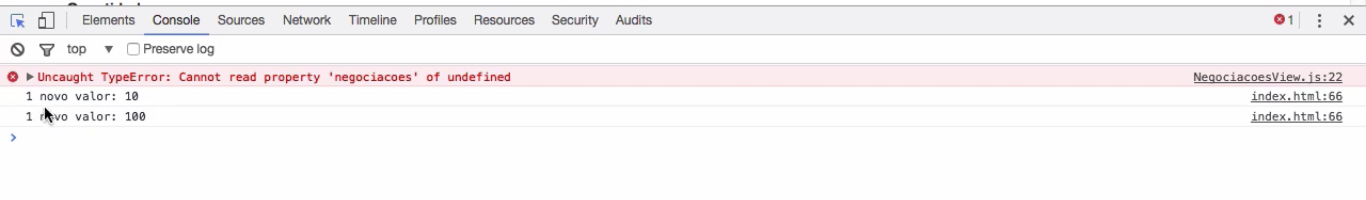
Com o value no console.log exibiremos o valor que queremos atribuir.



A armadilha foi executada e os valores exibidos estão corretos. Mas também queremos mostrar o valor antigo, para isto, só teremos acesso ao target que é o objeto Negociacao. Vamos tentar usar o seguinte código no console.log:

console.log(`${target.quantidade} novo valor: ${value}`);  
return Reflect.set(target, prop, value, receiver);

No Console. veremos os seguintes valores:



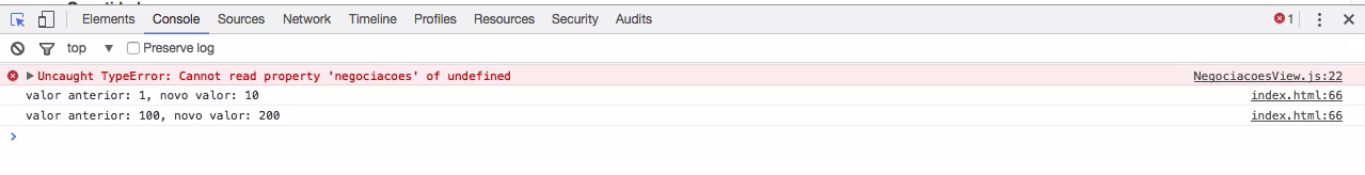
Para quantidade, o valor antigo ficou igual a 1, e o mesmo para o valor. A explicação é que a linha do console.logestá sendo executada também quando tentamos descobrir o negociacao.\_valor. Então, o valor de quantidade deve ser igual ao valor de prop. Mas como conseguiremos fazer a *property* dinâmica? Não temos a opção de usar ${target.prop}, mas podemos usar um recurso do JS, podemos passar ${target[prop]}. O trecho do código ficará assim:

<script>  
  
  let negociacao = new Proxy(new Negociacao(new Date(), 1, 100), {  
    set: function(target, prop, value, receiver) {  
  
         console.log(`${target[prop]} novo valor: ${value}`);  
         return Reflect.set(target, prop, value, receiver);  
}  
    });  
    negociacao.\_quantidade = 10;  
    negociacao.\_valor = 100;  
</script>

O prop junto com o target será o \_quantidade e conseguirá ler o valor. Em seguida, adicionaremos valor anterior:à linha.

<script>  
  
  let negociacao = new Proxy(new Negociacao(new Date(), 2, 100), {  
    set: function(target, prop, value, receiver) {  
  
         console.log(`valor anterior: ${target[prop]} novo valor: ${value}`);  
         return Reflect.set(target, prop, value, receiver);  
}  
    });  
    negociacao.\_quantidade = 10;  
    negociacao.\_valor = 200;  
</script>

No Console, veremos os seguintes valores:



Com isso, resolvemos o problema da execução de um código que só atualizará a View quando \_quantidade e \_valorforem alterados.

Agora que ensaiamos com o negociacao, vamos nos focar em criar a Proxy no modelo do ListaNegociacoes. No index.html, vamos substituir no código negociacao por ListaNegociacoes. Mudaremos também o console.log pelo lista.adiciona().

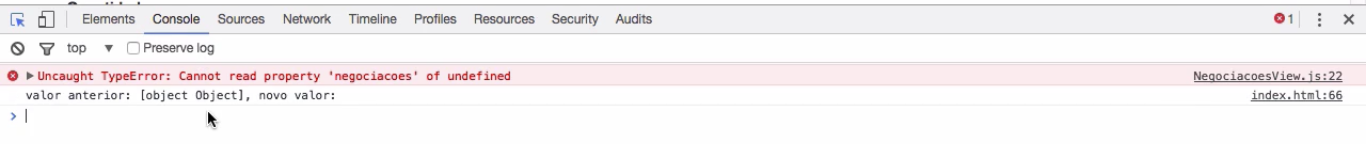
<script>  
  
  let lista = new Proxy(new ListaNegociacoes(), {  
    set: function(target, prop, value, receiver) {  
  
         console.log(`valor anterior: ${target[prop]} novo valor: ${value}`);  
         return Reflect.set(target, prop, value, receiver);  
    });  
    lista.adiciona(new Negociacao(new Date(), 1, 100));  
</script>

Se tentarmos executar o código como está, nada acontecerá. Nós já entendemos que devemos usar o get quando estamos olhando uma propriedade e o set quando queremos modificar uma propriedade. No entanto, a armadilha não foi disparada no lista.adiciona(). Sabe por que isto aconteceu? O adiciona() é um método, que no arquivo ListaNegociacoes.js, pede internamente para negociacoes fazer o push().

adiciona(negociacao) {  
  
    this.\_negociacoes.push(negociacao);  
  
}

Porém, nós não atribuímos valores para uma propriedade, sendo assim, o set não será chamado. Teremos que encontrar uma solução, porque tanto o método adiciona() como o esvazia() terá este tipo de problema - ainda que o segundo, atribua um valor para negociacoes. Vamos fazer um teste, adicionando o esvazia(), no index.html:

<script>  
  
  let lista = new Proxy(new ListaNegociacoes(), {  
    set: function(target, prop, value, receiver) {  
  
         console.log(`valor anterior: ${target[prop]} novo valor: ${value}`);  
         return Reflect.set(target, prop, value, receiver);  
    });  
    lista.adiciona(new Negociacao(new Date(), 1, 100));  
    lista.esvazia();  
</script>



O esvazia() executou o código, porque o valor anterior: era um objeto e o novo valor: ficou vazio. Nós queremos que o interceptador dispare com o lista.adiciona. No entanto, por padrão, não conseguiremos fazer isto usando o Proxy do ES6.

Mas não vamos ignorar o que vimos até aqui.

Nós não vamos nos abdicar de utilizar o Proxy. Mas a primeira tentativa de solucionar o assunto será com uma "gambiarra". Em ListaNegociacoes, vamos forçar uma atribuição em this.\_negociacoes.

class ListaNegociacoes {  
  
    constructor() {  
        this.\_negociacoes = [];  
    }  
  
    adiciona(negociacao) {  
  
        this.\_negociacoes = [].concat(this.\_negociacoes, negociacao);  
        // this.\_negociacoes.push(negociacao);  
    }

Com o concat(), podemos passar tanto o *array* quanto o elemento, além de outros parâmetros - e todos serão concatenados. Como forçamos uma atribuição, quando recarregamos a página, veremos no Console:

\_negociacoes => valor anterior: , novo valor: [object Object]

\_negociacoes => valor anterior: está saindo em branco, porque o valor anterior é um array vazio. O novo valortem o object porque se trata do novo array.

É possível resolver com esta gambiarra, mas não será esta a solução que utilizaremos. Imagine que fazemos um loop de 100 negociações, se nosso código ficar assim, a cada negociação adicionada, teremos que criar um nova lista e reatribuir para ListaNegociacoes. Teríamos problema com a performance. Apesar de termos feito o mesmo no get:

get negociacoes() {  
  
    return [].concat(this.\_negociacoes);  
}

Mas neste caso, quem pede a lista de negociação não fará solicitação infinitas vezes seguidas. É possível varrer a lista diversas vezes, sem chamar get sempre. O mesmo não ocorre com adiciona(). Se queremos fazer uma operação em lote, usaremos o adiciona(). Precisamos descobrir uma outra estratégia - mais de "cangaceiro" do que de "ninja". Ao identificarmos o adiciona() como um método no index.html, queremos executar a armadilha.

<script>  
  
  let lista = new Proxy(new ListaNegociacoes(), {  
    set: function(target, prop, value, receiver) {  
  
         console.log(`valor anterior: ${target[prop]} novo valor: ${value}`);  
         return Reflect.set(target, prop, value, receiver);  
    });  
    lista.adiciona(new Negociacao(new Negociacao(new Date(), 1, 100));  
</script>

Não há uma maneira de interceptarmos o método com o Proxy. Quando chamamos um método no JavaScript, por exemplo o adiciona(), ele fará um get na função e depois, executará o Reflect.apply. Então, será passado o valor para a função dentro do parênteses. Mas o set não está incluso no método.

A seguir, resolveremos o assunto.

Vamos procurar a solução "cangaceira" para resolvermos o nosso problema. Lembra que anteriormente falamos que quando chamamos uma função, o JavaScript fará um getter e após a leitura, será enviada um apply. Teremos que substituir o set para o get:

<script>  
  
  let lista = new Proxy(new ListaNegociacoes(), {  
    get: function(target, prop, receiver) {  
  
    }  
  });  
    lista.adiciona(new Negociacao(new Date(), 1, 100));  
</script

Observe que o get não receberá o value. A questão é: quando o getter for executado, queremos perguntar se ele está na lista de métodos que queremos interceptar. Para isto, adicionaremos um if para o get.

get: function(target, prop, receiver) {  
    if(['adiciona', 'esvazia'].includes(prop) && type(target[prop]) == typeof(Function)) {  
  
    }  
}

Criaremos uma condição que testará se o método incluído é o adiciona() ou o esvazia(), que tem ou não props (uma novidade do ES6 é podermos fazer este tipo de pergunta para o array) e se é uma função. Para testarmos esta última parte, usamos o typeof[], que recebeu a propriedade do target. Se isso é uma função ou método, o typeof será o parâmetro. Vamos verificar se isso é o typeof de Function.

Se viermos no Console e digitarmos Function, veremos que existe uma função que existe function em JavaScript:

Function  
function Function() { [native code] }

Se adiciona() e esvazia() tem a propriedade e é uma função, faremos algo a respeito. Caso contrário, é a leitura de um get padrão.

<script>  
  
  let lista = new Proxy(new ListaNegociacoes(), {  
    get: function(target, prop, receiver) {  
        if(['adiciona', 'esvazia'].includes(prop) && type(target[prop]) == typeof(Function)) {  
  
        }  
        return Reflect.get(target, prop, receiver);  
    }  
    });  
      lista.adiciona(new Negociacao(new Date(), 1, 100));  
 </script>

Se está ou não na nossa lista, teremos o retorno do valor, considerando que estamos fazendo um get. Adicionaremos o lista.\_negociacoes:

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

Com isso, já faremos um get. E o que iremos devolver? Se a propriedade está na lista (de adiciona() ou esvazia()) e é uma função, em vez de retornarmos um valor direto, retornaremos uma função.

Tem que ser function para ter o this dinâmico. Não pode ser *arrow function* que possui escopo léxico.

A função não pode ser arrow function, porque ela deve ter um contexto dinâmico, e dentro, substituiremos o método por outro que tem a armadilha - porém, a substituição será feita no Proxy. No entanto, este não nos permite colocar uma armadilha para método, encontraremos um forma de que ao cairmos no método substituiremos por outro do Proxy. Falamos do return.

<script>  
  
  let lista = new Proxy(new ListaNegociacoes(), {  
  
    get: function(target, prop, receiver) {  
        if(['adiciona', 'esvazia'].includes(prop) && typeof(target[prop]) == typeof(Function)) {  
        return function() {  
console.log(`a propriedade "${prop}" foi interceptada`);  
            }  
        }  
        return Reflect.get(target, prop, receiver);  
    });  
      lista.adiciona(new Negociacao(new Date(), 1, 100));  
    lista.\_negociacoes  
</script>

Agora, teremos que fazer o método receber os parâmetros que ele está recebendo aqui. Como estamos substituindo e retornando a função, substituiremos o adiciona(), quando este método for chamado estaremos chamando na verdade, o console.log. No entanto, precisamos fazer com que o adiciona() receba o parâmetro original dele:

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

Como a função no if, substituirá o método adiciona(), existe um objeto implícito chamado arguments que dá acesso a todos os parâmetros passados para a função. A seguir, usaremos o Reflect.apply(), e chamaremos uma função.

if(['adiciona', 'esvazia'].includes(prop) && typeof(target[prop]) == typeof(Function)) {  
   return function() {  
  
        console.log(`a propriedade "${prop}" foi interceptada`);  
        Reflect.apply(target[prop], target, arguments);  
 }

Vamos entender o que foi feito até aqui... Ao fazemos métodos e funções o Proxy sempre entende que é um get, quando fazemos o lista.adiciona(). No get, perguntaremos "você está na lista de itens que quero interceptar? E você é uma função?" Caso a resposta seja positiva, iremos substituir o adiciona()ou o esvazia() no Proxy. A substituição será feita por uma nova função. Então, ao ser chamada, a função imprimirá o conteúdo do console.log, porque a função lembrará do contexto de execução e quem é prop. Com o Reflect.apply(), faremos a função receber os parâmetros dela. O arguments é uma variável implícita que dá acesso a todos os parâmetros da função quando esta é chamada. Foi uma maneira de via get ter acesso aos parâmetros da função.

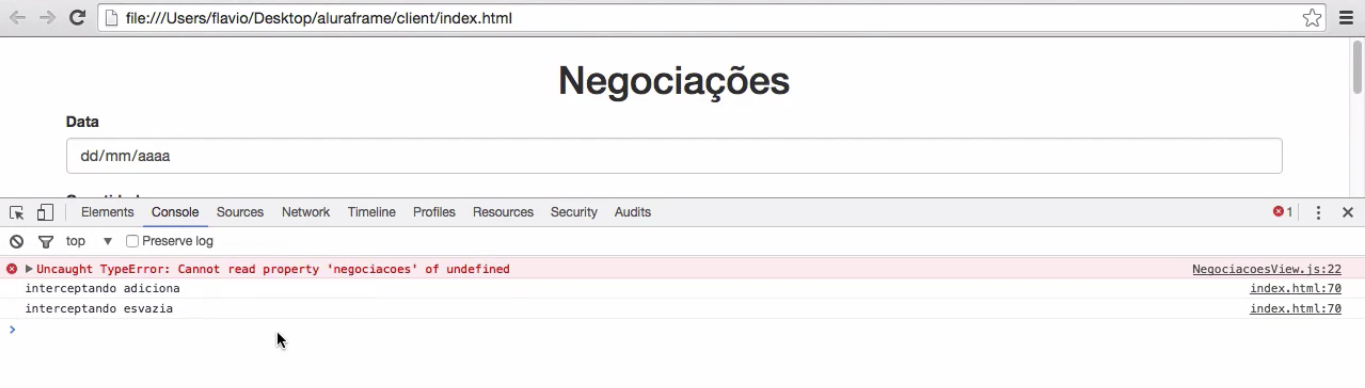
Ao recarregarmos a página no navegador, veremos a confirmação de que foi interceptado adiciona.



O mesmo ocorrerá se chamarmos o lista.esvazia():

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

No Console, veremos o interceptado esvazia.



Desta forma, encontramos uma maneira de escolher qual método queremos interceptar e executar o código. Antes de continuarmos, faremos um pequeno ajuste no get:

get(target, prop, receiver) {  
    if(['adiciona', 'esvazia'].includes(prop) && typeof(target[prop]) == typeof(Function)) {  
  
        return function() {  
  
          console.log(`a propriedade "${prop}" foi interceptada`);  
          Reflect.apply(target[prop], target, arguments);  
        }  
  
    }  
    return Reflect.get(target, prop, receiver);  
  
  }

Agora, precisamos substituir o trecho de código no Controller. Vamos aplicar efetivamente no nosso sistema.

Você pode estar pensando "será que existe uma maneira mais fácil de fazer o que queremos?". A resposta é sim. Conheceremos a forma mais simples, depois. Primeiramente, vamos levar a estratégia criada no index.html, para o NegociacoesController.

Abaixo dos inputs do constructor(), adicionaremos o seguinte trecho do código:

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);  
  
              this.\_negociacoesView.update(target);  
  
            }  
     }  
  
     return Reflect.get(target, prop, receiver);  
  }  
});

Passamos o model como parâmetro do update(), ao adicionarmos o target. No entanto, o nosso código terá problema porque o this não será o controller, nem se usássemos uma *arrow function*. Por isso, vamos inserir uma variável chamada self. que receberá this.

let self = 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);  
  
              self.\_negociacoesView.update(target);  
  
            }  
     }  
  
     return Reflect.get(target, prop, receiver);  
  }  
});

Faremos um teste a seguir, mas faltou um detalhe: só podemos chamar a atualização da View, depois de aplicar o valor depois do target. Vamos testar no navegador, e veremos que o método foi interceptado.



Então, o código está funcionando. Nós tivemos que aprender a lidar com as particularidades do Proxy, e assim, não "sujamos" o nosso modelo. Mais adiante veremos como esconder a complexidade de criação do Proxy. O nosso código ficou muito verboso. Inclusive, não vamos inserir um Proxy para MensagemView, mas seria um erro repetir todo este código novamente.

Por enquanto, só brincaremos com o listaNegociacoes. A seguir, vamos aplicar um outro padrão de projeto que nos ajudará bastante na criação de um Proxy. Mas a forma de se trabalhar internamente já foi apresentada.

Sabemos que podemos declarar um objeto JavaScript com propriedades que podem armazenar funções (uma função que pertence ao objeto podemos chamar de método). Por exemplo:

var objeto = {  
    exibeMensagem : function(){  
        console.log("Olá");  
    }  
};

Neste capítulo, aprendemos um modo de reescrever este mesmo código acima, só que de modo mais sucinto. Você consegue lembrar como era?

Com o ES2015 , podemos reescrever aquele código da seguinte maneira:

var objeto = {  
    exibeMensagem() {  
        console.log("Olá");  
    }  
}

Assim conseguimos enxugar um pouco mais o código, aproveitando essa *feature* do ES2015.

## Vejamos o seguinte exemplo:

## let pessoa = {    nome: 'Flávio'     } let pessoaProxy = new Proxy(pessoa, {    get(target, prop, receiver) {         //...    } });

## Como segundo argumento de um proxy, passamos um *handler*, que é um objeto JavaScript que contém as armadilhas (*traps*) do nosso Proxy. Neste objeto, podemos criar uma propriedade get e passar para ela uma função com 3 parâmetros.

## get(target, prop, receiver) {    //... }

## Sobre estes 3 parâmetros, qual das afirmativas abaixo é completamente verdadeira?

O target é o objeto real, que é encapsulado pelo proxy.

O prop é a propriedade que está sendo lida.

O receiver é uma referência ao próprio proxy.

O ***target*** é o objeto real que é encapsulado pela proxy. É este objeto que não queremos "sujar" com armadilhas ou qualquer código que não diga respeito ao modelo.

O ***prop*** é a propriedade em si, que está sendo lida naquele momento.

O ***receiver*** é a referência ao próprio proxy. É na configuração do handler do Proxy que colocamos armadilhas.

Aprendemos neste capítulo a criar proxies, que nada mais são do que objetos mentirosos que encapsulam outros, mais notadamente objetos do nosso modelo. Pense em proxies como "cascas" que envolvem objetos. Dentro desse contexto, só podemos "tocar" os objetos encapsulados passando pelo proxy. É justamente essa característica que torna o uso desse padrão de projeto tão poderoso.

Temos o seguinte objeto literal (aquele criado com chaves):

let funcionario = {email: 'abc@abc.com'};

Crie um proxy para este objeto, exibindo no console a mensagem "Armadilha aqui", toda vez que a propriedade email for lida.

Uma implementação possível é:

let funcionario = {email: 'abc@abc.com'};  
let funcionarioProxy = new Proxy(funcionario,  {  
  
    get(target, prop, receiver) {  
        console.log('Armadilha aqui!');  
        return Reflect.get(target, prop, receiver);  
    }  
  
});  
console.log(funcionarioProxy.email);

Veja que a instrução a seguinte é muito importante: return Reflect.get(target, prop, receiver). É ela que efetivamente realiza a operação no objeto real. Aliás, poderíamos ter conseguido o mesmo resultado da seguinte maneira:

let funcionario = {email: 'abc@abc.com'};  
let funcionarioProxy = new Proxy(funcionario,  {  
  
    get(target, prop, receiver) {  
        console.log('Armadilha aqui!');  
        return target[prop];  
    }  
  
});  
console.log(funcionarioProxy.email);

Há uma ligeira diferença entre as duas. Na primeira, get retorna uma função que é invocada para obter o valor da propriedade original, na segunda, o valor é retornado diretamente. Entenda que Reflect.get cria algo semelhante a uma getter. Você ainda lembra que um getter é uma espécie de função?

Temos mais uma vez o seguinte objeto literal:

let funcionario = {email: 'abc@abc.com'};

Desta vez, imprima a mensagem "Armadilha aqui!", além disso, modifique o retorno do *getter*, fazendo-o retornar o email, começando e terminando com dois asteriscos. Por exemplo, se fizermos:

console.log(funcionario.email);

Queremos que saia no console:

\*\*abc@abc.com\*\*

Temos a seguinte implementação. Veja que ele concatena os asteriscos com o retorno de Reflect.get, aquele que efetivamente executa a operação de leitura no objeto real. Ou seja, nosso proxy executa a leitura no objeto original e devolve para quem chamou o próprio proxy o valor concatenado com asteriscos:

let funcionario = {email: 'abc@abc.com'};  
let funcionarioProxy = new Proxy(funcionario,  {  
  
    get(target, prop, receiver) {  
        console.log('Armadilha aqui!');  
        return '\*\*' + Reflect.get(target, prop, receiver) + '\*\*' ;          
    }  
  
});  
console.log(funcionarioProxy.email);

Agora, podemos evitar a declaração da variável funcionarioProxy desta forma:

funcionario = new Proxy({email: 'abc@abc.com'},  {  
  
    get(target, prop, receiver) {  
        console.log('Armadilha aqui!');  
        return '\*\*' + Reflect.get(target, prop, receiver) + '\*\*' ;          
    }  
  
});  
console.log(funcionario.email);

Temos a seguinte classe:

class Funcionario {

constructor(email) {

this.\_email = email;

}

get email() {

return this.\_email;

}

set email(email) {

this.\_email = email;

}

}

Crie um proxy a partir dessa classe, que exiba no console "Armadilha aqui" toda vez que qualquer propriedade da classe for lida.

Segue uma implementação:

let funcionario = new Proxy(new Funcionario('abc@abc.com'), {

get(target, prop, receiver) {

console.log('Armadilha aqui!');

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

}

});

console.log(funcionario.email);

Veja que teremos três mensagens de log no console. As duas primeiras Armadilha aqui!e logo depois o email do usuário. Mas qual a razão dele escrever a primeira mensagem duas vezes?

É que email é um *getter*. Nosso proxy irá executar seu código quando o *getter* for chamado e também para a propriedade \_email, que é acessada pelo *getter*. Inclusive podemos deixar isso ainda mais claro, exibindo em nosso proxy o nome da propriedade:

let funcionario = new Proxy(new Funcionario('abc@abc.com'), {

get(target, prop, receiver) {

console.log('Armadilha aqui!');

console.log(prop);

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

}

});

console.log(funcionario.email);

Olha o funcionário no formato literal novamente aí gente:

let funcionario = {email: 'abc@abc.com'};

Crie um proxy que exibe no console o valor da propriedade antes dela ser alterada e o valor novo.

Uma implementação possível:

let funcionario = {email: 'abc@abc.com'};

let funcionarioProxy = new Proxy(funcionario, {

set(target, prop, value, receiver) {

console.log(`Valor antigo ${target[prop]}, valor atual: ${value}`);

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

}

});

funcionarioProxy.email = 'aaa@aaa.com';

Veja que usamos set no *handler* passado para o proxy. Além disso, muito cuidado que quando usamos set, a função deve receber quatro parâmetros e não três, como no caso do get.

Poderíamos conseguir o mesmo resultado desta forma:

let funcionario = {email: 'abc@abc.com'};

let funcionarioProxy = new Proxy(funcionario, {

set(target, prop, value, receiver) {

console.log(`Valor antigo ${target[prop]}, valor atual: ${value}`);

target[prop] = value;

}

});

funcionarioProxy.email = 'aaa@aaa.com';

A diferença é que essa última solução altera diretamente o valor o objeto encapsulado. A primeira forma, com Reflect.set sempre deve retornar um valor, que é uma função que será chamada para realizar a operação de atribuição. Por enquanto, podemos usar uma ou outra forma que o resultado será o mesmo.

Durante o vídeo, para explicar como executar uma armadilha quando uma propriedade for modificada, eu usei como exemplo uma instância da classe Negociacao. Contudo, nossa classe era imutável porque usamos Object.freeze. Sendo assim, o seguinte código não teria efeito:

n1.\_quantidade = 10;

n1.\_valor = 10;

E realmente não tem, mas ainda sim a armadilha definida no proxy será executada! Resumindo a moral da história: armadilhas serão disparadas mesmo se tentarmos modificar uma propriedade congelada de um objeto, ainda que ele não seja modificado.

Aprendemos lá no jardim de infância em JavaScript a passar parâmetros para funções e métodos. Vejamos um exemplo:

function exibeNomeCompleto(nome, sobrenome) {

alert(`${nome} ${sobrenome}`);

}

exibeNomeCompleto('Flávio', 'Almeida');

Contudo, podemos conseguir o mesmo resultado sem passar parâmetros para a função:

function exibeNomeCompleto() {

alert(`${arguments[0]} ${arguments[1]}`);

}

exibeNomeCompleto('Flávio', 'Almeida');

Por mais que nossa função não receba parâmetros, podemos ter acesso aos parâmetros passados com arguments. É uma variável implícita que nos dá acesso a todos os parâmetros passados para a função ou método. É claro que a primeira forma, nomear os parâmetros da função, é menos verbosa e mais legível. Mas há muitos hacks em JavaScript que podem fazer uso de arguments.

Temos a seguinte declaração de classe:

class Pessoa {

constructor(nome) {

this.\_nome = nome;

}

get nome() {

return this.\_nome;

}

set nome(nome) {

this.\_nome = nome;

}

grita(frase) {

return `${this.\_nome} grita ${frase}`;

}

}

Criando uma instância e chamando o método grita:

let pessoa = new Pessoa('Barney');

pessoa.grita('Olá');

E se quisermos interceptar a chamada do método grita? A má notícia é que toda proxy criada, por padrão, não esta preparada para interceptar métodos (getters e setters são exceções a este problema). Essa limitação ocorre porque sempre que um método de um objeto (que não deixa de ser uma propriedade que armazena uma função) é chamado, primeiro é realizado uma operação de leitura (get, do nosso handler da proxy) e depois os parâmetros são passados através de Reflect.apply. O problema é que, como o método é interceptado pelo get do handler passado para a proxy, não temos acesso aos seus parâmetros. E agora?

Uma solução é implementar o seguinte código:

let pessoa = new Proxy(new Pessoa('Barney'), {

get(target, prop, receiver) {

if(prop == 'grita' && typeof(target[prop]) == typeof(Function)) {

// essa função retornada irá substituir o método 'grita' no proxy!!! Ou seja, estamos usando o handler do proxy para modificar o próprio proxy, que loucura!

return function() {

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

// Quando usarmos Reflect.apply, Reflect.get e Reflect.set precisamos retornar o resultado da operação com return

// arguments é uma variável implícita que dá acesso à todos os parâmetros recebidos pelo método/função

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

}

}

// só executa se não for função

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

}

});

pessoa.grita('Olá');

No código acima, verificarmos se a propriedade que está sendo acessada é uma função através de typeof(target[prop]) == typeof(Function)). Se for, trocaremos o valor da propriedade (nosso método) por outra função, e, essa sim, executa nosso código antes de o método ser executado.

Sobre o código anterior, qual o papel de arguments?

A variável arguments é uma variável implícita que pode ser acessada em métodos ou funções. Ele se comporta como um array onde cada posição equivale ao parâmetro que foi passado para o método ou função. Existe desde o ES5!

Vamos voltar a trabalhar com a Proxy. Esta é uma solução interessante que não polui o modelo original, com o código para atualizar a View. Mas criamos uma forma bastante verbosa... Estamos trabalhando com dez propriedades da Controller com as quais queremos criar uma Proxy. Podemos criar um padrão de projeto chamado Factory que consiste em uma classe ser especializada em criar determinado tipo de objeto. Em seguida, vamos gerar o arquivo ProxyFactory.js, que ficará dentro da pasta services:

class ProxyFactory {

static createProxy(objeto, props, acao) {

}

}

Não é uma regra em si, mas podemos invocar um método estático da classe para não ter que instanciá-la. Depois, moveremos de NegociacaoController, o trecho referente ao Proxy, para a nova classe criada:

class ProxyFactory {

static create(objeto, props, acao) {

return new Proxy(objeto, {

get(target, prop, receiver) {

if(props.includes(prop) && typeof(target[prop]) == typeof(Function)) {

return function() {

console.log(`a propriedade "${prop}" foi interceptada`);

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

return acao(target);

}

}

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

}

})

}

}

Dentro de props, temos um array no qual está a propriedade que queremos verificar. A acao() nos devolverá um valor. O próximo passo será importar o ProxyFactory 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>

let negociacaoController = new NegociacaoController();

</script>

Agora, não precisaremos mais do código do Proxy no NegociacaoController.js. No lugar, vamos adicionar uma \_listaNegociacoes com o ProxyFactory:

this.\_listaNegociacoes = ProxyFactory.create (

new ListaNegociacoes(),

['adiciona', 'esvazia'], model =>

this.\_negociacoesView.update(model));

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

this.\_negociacoesView.update(this.\_listaNegociacoes);

Dentro do ProxyFactory.create(), incluímos um array com as propriedades com adiciona e esvazia. Já não será necessário usar a variável self e, como a arrow function tem escopo léxico, ela entenderá que o this é referente a controller. Se testarmos no navegador, veremos que conseguimos preencher o formulário corretamente.

Nós já melhoramos a manutenção e a legibilidade do código. Aproveitaremos para fazer o mesmo com o Mensagem.

this.\_mensagem = ProxyFactory.create(

new Mensagem(), ['texto'], model =>

this.\_mensagemView.update(model));

this.\_mensagemView = new MensagemView($('#mensagemView'));

No entanto, ainda teremos que chamar manualmente a View, usando o update() e chamando this.\_mensagem. Sempre que criamos o modelo autoatualizável, a View só será recarregada, quando o modelo for modificado. Mas vamos resolver isso.

Iremos para o método adiciona(), removeremos o update().

adiciona(event) {

event.preventDefault();

this.\_listaNegociacoes.adiciona(this.\_criaNegociacao());

this.\_mensagem.texto = 'Negociação adicionada com suecesso';

this.\_limpaFormulario();

}

Faremos ajustes em apaga():

apaga(){

this.\_listaNegociacoes.esvazia();

this.\_mensagem.texto = 'Negociação adicionada com sucesso';

}

Se recarregarmos a página no navegador, veremos que o formulário funciona corretamente, mas não atualizou a mensagem.

O que está acontecendo? O ProxyFactory está preparado para interceptar métodos, mas não está preparado para interceptar propriedades, como em Mensagem.js:

class Mensagem {

constructor(texto='') {

this.\_texto = texto;

}

get texto() {

return this.\_texto;

}

set texto(texto) {

this.\_texto = texto;

}

}

O get texto() é um getter. O ProxyFactory não entende que precisa interceptar com o get. Já que os getters e setters são acessados como propriedades, temos também que colocar no ProxyFactory.js, um código para lidarmos com as propriedades. Para isto, adicionaremos um set, logo abaixo do último return:

set(target, prop, value, receiver) {

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

acao(target);

}

Depois, usaremos o Reflect.set(), recebendo os quatro parâmetros. Logo após, chamaremos o acao(target). Assim, garantiremos que quando for executada a propriedade, depois, será a vez do interceptador.

Mas se preenchermos o formulário no navegador, veremos que ele não ficará limpo, e aparecerá uma mensagem de erro que dará 'set' on proxy. Isto ocorreu porque quando chamamos um Reflect.set() temos que chamar também um return. Também será necessário adicionar o if:

set(target, prop, value, receiver) {

if(props.includes(prop)) {

target[prop] = value;

acao(target);

}

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

}

Aplicamos os filtros nas propriedades que queremos. Podemos melhorar o nosso código... Começando pelo fato que não queremos retornar fixo o ListaNegociacoes() - nós queremos levar em consideração o objeto que estamos usando como parâmetro. Faremos um segundo ajuste: o código do typeof() quer identificar se estamos trabalhando com uma função. Para ficar mais específico, criaremos o método static \_ehFuncao(), em seguida, substituiremos o typeof por ele:

static \_ehFuncao(func) {

return typeof(func) == typeof(Function);

}

Depois, substituiremos o typeof localizado no if do get, por \_ehFuncao.

class ProxyFactory{

static create(objeto, props, acao) {

return new Proxy(objeto, {

get(target, prop, receiver) {

if(props.includes(prop) && ProxyFactory.\_ehFuncao(target[prop])) {

return function() {

console.log(`a propriedade "${prop}" foi interceptada`);

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

return acao(target);

};

}

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

},

set(target, prop, value, receiver) {

if(props.includes(prop)) {

target[prop] = value;

acao(target);

}

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

}

});

}

}

Atenção: como estamos trabalhando com um método estático, não se esqueça de adicionar o nome da classe ProxyFactory antes do \_ehFuncao(), no if.

Vamos recarregar a página no navegador e preencher os campos do formulário. Tudo estará funcionando no nosso sistema.

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 biding? 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 adicionae 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.

Nós queremos simplificar ainda mais o nosso código. Nós estamos passando as propriedades adiciona, esvazia e texto dentro de um array:

class NegociacaoController {

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($('#negociacoesView')),

['adiciona', 'esvazia']);

this.\_mensagemView = new MensagemView($('#mensagemView')),

this.\_mensagem = new Bind(

new Mensagem(),

this.\_mensagemView,

['texto']);

}

//...

Vamos retirar as propriedades do array.

this.\_listaNegociacoes = new Bind (

new ListaNegociacoes(),

this.\_negociacoesView($('#negociacoesView')),

'adiciona', 'esvazia');

Mas, então, teremos uma ListaNegociacoes com uma View e dois parâmetros? Na classe Bind, veremos que são aceitos apenas três parâmetros.

class Bind {

constructor(model, view, props) {

let proxy = ProxyFactory.create(model, props, model =>

view.update(model));

view.update(model);

return proxy;

}

}

No entanto, a ProxyFactory precisa receber um array, por isso, as propriedades eram passadas entre [] (colchetes). Contudo, quando o último parâmetro de um construtor, função ou método é variável, podemos usar o parâmetro **REST** operator (...):

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

let proxy = ProxyFactory.create(model, props, model => {

view.update(model)

});

view.update(model);

return proxy;

}

Vamos entender o que será feito, relembrando o Bind do NegociacaoController.js:

* O primeiro parâmetro recebido pelo Bind é o model, o segundo é a view, e a partir do terceiro, eles caem dentro do ...props - podendo ser diversos, como um array. No nosso caso, ...props é um array com duas posições (adiciona e esvazia). É isso que o create() do ProxyFactory espera receber. Com a pequena adição do REST. Isto também nos permite fazer uma associação com apenas um parâmetro, como no caso do texto de \_mensagem, sem colocá-lo em um array.

O rest operator (...) não deve ser adicionado no primeiro parâmetro, porque isso traria problemas para os seguintes. Por exemplo, se fizéssemos ...model:

constructor (...model, view, ...props){ }

Para gerarmos um array com *n* parâmetros, devemos usar o REST apenas no último, outro vantajoso recurso do ECMAScript.

Tem mais uma melhoria que podemos fazer no código... No início da aplicação, criamos uma instância de NegociacoesView, que era guardada na propriedade do \_negociacoesView:

class NegociacaoController {

constructor() {

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

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

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

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

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

//...

Também criamos o MensagemView() e guardamos na propriedade da controller. Mas se observarmos o código, veremos que não utilizamos a View novamente em NegociacoesController. Quem manipulará a View será a nossa associação, que mediante a atualização do modelo, recarregará a View. Então, não precisamos ter as propriedades this.\_negociacoesView e this.\_mensagemView, afinal, elas não são utilizadas. Passaremos as instâncias da View de forma mais direta:

class NegociacaoController {

constructor() {

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

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

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

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

this.\_listaNegociacoes = new Bind(

new ListaNegociacoes(),

new NegociacoesView($('#negociacoesView')),

'adiciona', 'esvazia')

this.\_mensagem = new Bind(

new Mensagem(), new MensagemView($('#mensagemView')),

'texto');

}

Não vamos mais manipular a View manualmente, ela será atualizada automaticamente quando o modelo for alterado. Mas precisaremos de um modelo para trabalhar. Quando usamos o new Bind(), associaremos o modelo com a View e o ListaNegociacoes será o Proxy. Então, agora, o código ficou mais enxuto.

Vamos misturar os conceitos um pouco, afinal isso é um treinamento avançado, certo?

Usamos o padrão **Factory** para isolar ou encapsular a complexidade da criação do proxy, mas uma fábrica pode fazer mais! Quero dizer que existem outros motivos para usar uma fábrica.

Não sou especialista em bolsa de valores, no entanto sei que, além de ações de uma empresa, existem também *opções* para comprar. Uma *opção* dá o direito de comprar ou vender uma determinada ação/índice na bolsa de valores. *Opções* também são negociadas! Ou seja, além de ações podemos negociar, comprar e vender *opções*.

Para representar isso no nosso modelo, poderíamos criar duas filhas da classe Negociacao: NegociacaoAcao e NegociacaoOpcao.



Agora precisamos decidir no nosso código qual das duas classes devemos instanciar! Não vai ter jeito, e em algum lugar precisará ter um if:

var negociacao = null;

let tipoNegociacao = "opcao"; //isso poderia vir de um formulário web

if(tipoNegociacao == "opcao") {

negociacao = new NegociacaoOpcao(/\*passando params aqui\*/);

} else {

negociacao = new NegociacaoAcao(/\*passando params aqui\*/);

}

Nesse exemplo, a decisão é relativamente simples, mas se tivesse mais parâmetros para avaliar? E se a gente precisasse desse if em mais de um lugar?

Onde vamos colocar essa decisão?

A resposta é: vamos colocar aquele if dentro de uma **factory**.

Podemos criar uma classe NegociacaoFactory, que possui um método de criação:

class NegociacaoFactory {

static create(tipoNegociacao, dados) {

if(tipoNegociacao == "opcao") {

return new NegociacaoOpcao(dados.data, dados.quantidade, dados.valor);

}

return new NegociacaoAcao(dados.data, dados.quantidade, dados.valor);

}

}

let n = NegociacaoFactory.create("acao", {'data': new Date(), 'quantidade': 2, 'valor': 34.3});

Repare que a **Factory** possui mais uma outra responsabilidade, instanciar NegociacaoAcao ou NegociacaoOpcao. A fábrica decide então qual implementação usar. Para quem chama, isso pouco importa, pois basta saber que recebemos uma Negociacao.

As fábricas não só fazem parte do *nosso* código, como também da *API do JavaScript*. Já existem várias classes que aproveitam esse padrão.

Por exemplo, a classe String possui um método (ou *factory method*) para transformar vários CharCode em uma string:

let abc = String.fromCharCode(65, 66, 67); // "ABC"

Outro exemplo é a classe Array, que pode receber uma string ou um iterável, como lista ou mapas, para criar um array:

let d = Array.from("abc");

["a", "b", "c"]

Teste os dois métodos de fábrica agora no seu navegador :)

Ou seja, um *factory method* nem sempre precisa estar dentro de uma classe dedicada. No exemplo desse exercício, não existe uma classe StringFactory ou ArrayFactory. O método pode fazer parte da classe em questão.

Além disso, o *factory method* não precisa se chamar *create* ou *constroi*. Outros nomes são válidos, como *from* ou *getInstance*.

Nós aplicamos diversos recursos da linguagem JavaScript, padrões de projetos e outros... Mas para a aplicação ficar completa, além de incluir negociações, queremos poder importar negociações de serviços na web. Alguns bancos oferecem a possibilidade de obter as negociações da semana atual e da anterior. Nós já disponibilizamos no projeto uma pasta chamada server. No [primeiro exercício obrigatório](https://cursos.alura.com.br/course/javascript-es6-orientacao-a-objetos-parte-2/task/23371) do curso, você encontrará toda a infraestrutura que você precisa para subir o servidor.

Com a infraestrutura instalada, entraremos na pasta Desktop. Para isto, você deve ter um conhecimento básico sobre o uso do Terminal para entrar nesta pasta.

No curso, usaremos o Terminal do Mac. Mas se você é usuário de Windows e não domina o prompt de comando, nossa recomendação é que faça o curso [Windows: Introdução ao Prompt](https://cursos.alura.com.br/course/prompt). Neste ponto do curso, assumiremos que você já sabe interagir com seu Terminal.

Para levantar o servidor, abriremos nosso Terminal e depois, entraremos na pasta aluraframe/server. É a primeira vez que interagiremos com essa pasta, mas basicamente, você só precisa rodar o comando no seu terminal:

> npm start

Caso esse comando de erro use esse:

> node server.js

Isto fará com que um servidor rode e seja acessível por meio do endereço http://localhost:3000. Acesse esse endereço e automaticamente a página index.htmlserá carregada. Se preferir, pode digitar http://localhost:3000/index.html.

A única mudança até agora é não acessar mais index.html do sistema de arquivos da plataforma, mas por intermédio do servidor. Certifique-se que o servidor esteja funcionando antes de continuar. O nosso servidor além de servir o index.html, possui também alguns serviços.

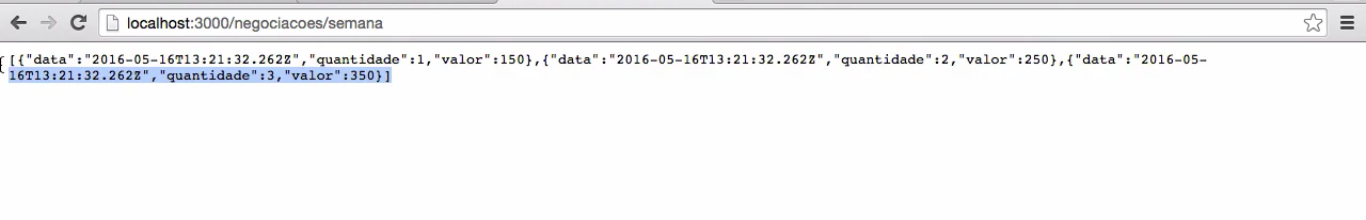
Geralmente, quem publica serviços na web disponibiliza uma URL de acesso. O nosso servidor publicará três:

negociacoes/semana

negociacoes/anterior

negociacoes/retrasada

Ao acessarmos o endereço http://localhost:3000/negociacoes/semana, sairemos da página index.html e, na tela, será exibida uma estrutura de dados no formato JSON (JavaScript Object Notation):



No entanto, não queremos exibir o arquivo JSON na tela e buscar os dados por meio de URL. Nosso objetivo é que ao clicarmos no botão Importar Negociações, seja feita a busca dos dados usando JavaScript e depois, os dados sejam inseridos na tabela de negociações. Desta forma, a tabela será automaticamente atualizada.

Nós temos todo mecanismo de Data binding, então, basta incluirmos os dados na tabela. A seguir, nós iremos relembrar como acessamos endereços na Web por meio do JavaSript, usando AJAX. Faremos uma revisão sobre o assunto e assim, entender o problema que desejamos resolver e qual recurso avançado pode nos ajudar em programação assíncrona .

O foco do curso é sobre como consumimos os dados publicados pelos servidores usando o JavaScript. Se você deseja aprender como servidores são criados e como eles disponibilizam esse tipo de dados, nossa sugestão é que você faça o curso de [**MEAN**](https://cursos.alura.com.br/course/mean-javascript).

Nosso objetivo é que ao clicarmos em "Importar Negociações", seja chamada uma ação na controller. Buscaremos o trecho referente no código do index.html.

<div class="text-center">

<button class="btn btn-primary text-center" type="button">

Importar Negociações

</button>

<button class="btn btn-primary text-center" type="button">

Apagar

</button>

</div>

Dentro da tag <button>, vamos adicionar o onclick.

<div class="text-center">

<button onclick="negociacaoController.importaNegociacoes()" class="btn btn-primary text-center" type="button">

Importar Negociações

</button>

<button class="btn btn-primary text-center" type="button">

Apagar

</button>

</div>

Nós chamamos o importaNegociacoes(), que será adicionado no NegociacaoController.js, logo abaixo do método adiciona().

importaNegociacoes() {

alert('Importando negociações');

}

Desta forma, poderemos ver se tudo está funcionando bem. Se recarregarmos a página no navegador e depois, clicarmos em "Importar Negociações", o alert será exibido.



Agora, realizaremos requisições assíncronas para o servidor usando o objeto especial existente no JavaScript. Substituiremos o alert pela variável xhr.

importaNegociacoes() {

let xhr = new XMLHttpRequest();

}

A variável é uma instância de new XMLHttpRequest(). É comum vermos desenvolvedores realizarem o AJAX utilizando apenas jQuery, mas nós faremos manualmente. Pediremos para xhr abrir um endereço e especificaremos qual método será utilizado:

importaNegociacoes() {

let xhr = new XMLHttpRequest();

xhr.open('GET', 'negociacoes/semana');

}

O método open() recebeu dois parâmetros: o primeiro especifica o tipo de requisição a ser realizada(GET), o segundo é o endereço (negociacoes/semana). Se trabalhássemos com outro endereço do serviço na Web, seria necessário colocar o endereço completo. Como estamos trabalhando localmente, só adicionamos negociacoes/semana.

Mas até aqui, a requisição não será executada. Para que a ação seja executada, usaremos o método send().

importaNegociacoes() {

let xhr = new XMLHttpRequest();

/\* configurações \*/

xhr.open('GET', 'negociacoes/semana');

/\* executa \*/

xhr.send();

}

A requisição ainda não está pronta. Será preciso fazer várias configuração antes de realizar o envio. É o que faremos a seguir: primeiramente, precisamos entender que toda requisição AJAX passa por estados - um deles nos dará os dados retornados do servidor. Por isso, precisamos interagir com esses estados e especificar que adicionaremos os dados de um deles no nosso model. O xhr tem a propriedade onreadystatechange, depois, passaremos uma arrow funtion que será chamada sempre que o estado do xhr for modificado.

Então, quais são os estados possíveis de um requisição AJAX? Listaremos abaixo os estados:

* 0: requisição ainda não iniciada
* 1: conexão com o servidor estabelecida
* 2: requisição recebida
* 3: processando requisição
* 4: requisição está concluída e a resposta está pronta

O estado **4** é o que mais nos interessa, porque é nele que temos acesso à resposta enviada pelo servidor. Precisamos encontrar um forma de descobrir em qual estado estamos quando o onreadystatechange é chamado. Para isto, adicionaremos um if:

importaNegociacoes() {

let xhr = new XMLHttpRequest();

xhr.open('GET', 'negociacoes/semana');

xhr.onreadystatechange = () => {

if(xhr.readyState == 4) {

}

};

xhr.send();

}

Se o estado for igual a 4, buscaremos os dados do servidor. Este é o estado que a requisição está concluída e a resposta está pronta, mas não podemos confiar no mesmo, porque pode ocorrer algum erro no servidor e ainda assim, a resposta será válida. É comum o servidor lidar com estados da requisição. Então, só teremos certeza de que os dados chegaram se o xhr.status for igual a 200 (status code genérico).

if(xhr.status == 200) {

}

Se fosse um status de erro, poderíamos colocar 400 ou 500... Usaremos os dois ifs para nos blindarmos de qualquer tipo de problema.

Com as modificações, o nosso código ficará assim:

importaNegociacoes() {

let xhr = new XMLHttpRequest();

xhr.open('GET', 'negociacoes/semana');

xhr.onreadystatechange = () => {

if(xhr.readyState == 4) {

if(xhr.status == 200) {

}

}

};

xhr.send();

}

A seguir, faremos um teste, adicionando o else e o console.log.

importaNegociacoes() {

let xhr = new XMLHttpRequest();

xhr.open('GET', 'negociacoes/semana');

xhr.onreadystatechange = () => {

if(xhr.readyState == 4) {

if(xhr.status == 200) {

console.log('Obtendo as negociações do servidor.')

} else {

console.log('Não foi possível obter as negociações do servidor.')

}

}

}

xhr.send();

}

Será exibida uma mensagem de erro quando o valor for diferente de 200. A questão é: como teremos acesso às requisições que vieram do servidor?

Para termos acesso às requisições que foram retornadas pelo servidor, vamos usar a propriedade xhr.responseText. Ela também vai capturar a mensagem de erro vinda do servidor:

//...

if(xhr.readyState == 4) {

if(xhr.status == 200) {

console.log('Obtendo as negociações do servidor.');

console.log(xhr.responseText);

}else{

console.log('Não foi possível obter as negociações do servidor.');

console.log(xhr.responseText);

}

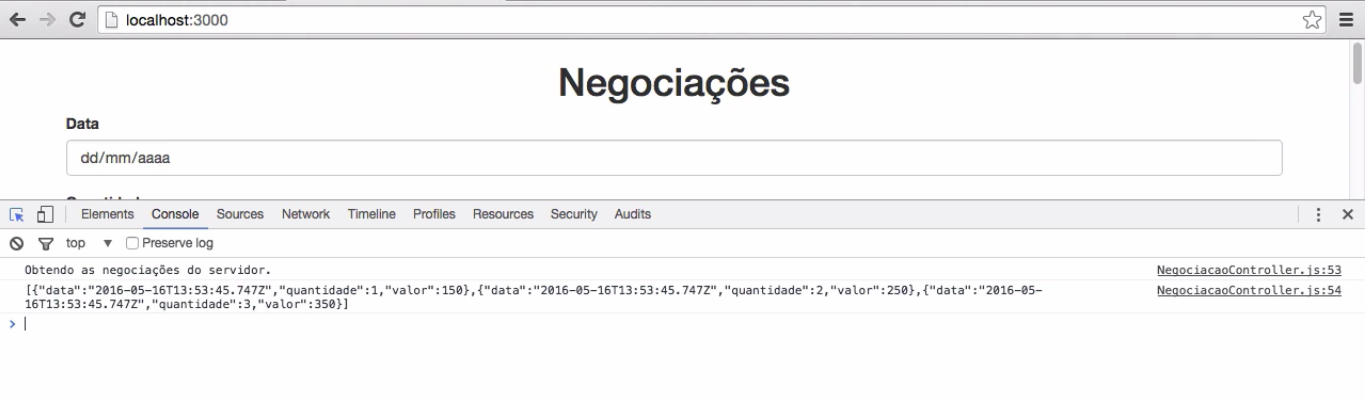
}

};

xhr.send();

}

Em seguida, recarregaremos a página e veremos o que será exibido no Console:



Vemos um texto sendo exibido e, não, efetivamente um *array*. Isto acontece porque o JSON tem um formato textual para que os dados possam trafegar pela internet. Para transformarmos o JSON (texto) em um array de objetos, usaremos a função JSON.parse(), existente desde a versão anterior do JavaScript. Cada item do *array* é um objeto JavaScript. Mas nós poderíamos fazer como no exemplo abaixo?

if(xhr.readyState == 4) {

if(xhr.status == 200) {

this.\_listaNegociacoes.adiciona(JSON.parse(xhr.responseText));

} else {

console.log('Não foi possível obter as negociações do servidor.');

console.log(xhr.responseText);

}

}

Não poderíamos fazer isto, porque o adiciona() recebe uma negociação por vez. Na verdade, ela recebe uma instância de negociação. Se observarmos o *array* exibido no Console, **não** veremos uma instância de negociação, na verdade, cada item é um *object*. Primeiramente teremos que converter o JSON, fazendo o parse() e para cada item do *array*, teremos que criar uma negociação. Para realizarmos esta ação, usaremos a função map() - que varrerá o array e criará um novo com modificações. Usaremos uma *arrow function*:

if(xhr.readyState == 4) {

if(xhr.status == 200) {

JSON.parse(xhr.responseText)

.map(objeto => new Negociacao(objeto.data, objeto.quantidade, objeto.valor));

} else {

console.log('Não foi possível obter as negociações do servidor.');

console.log(xhr.responseText);

}

}

Os itens serão chamados de objeto e para cada um deles, retornaremos uma Negociacao(). Como trabalhamos com uma *arrow function*, não precisaremos usar o return. Da lista que foi criada pelo map, vamos iterar com o forEach(), e para cada item teremos um negociação. A seguir, vamos incluir this.\_listaNegociacoes:

if(xhr.readyState == 4) {

if(xhr.status == 200) {

JSON.parse(xhr.responseText)

.map(objeto => new Negociacao(objeto.data, objeto.quantidade, objeto.valor))

.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

} else {

console.log(xhr.responseText);

this.\_mensagem.texto = 'Não foi possível obter as negociações da semana';

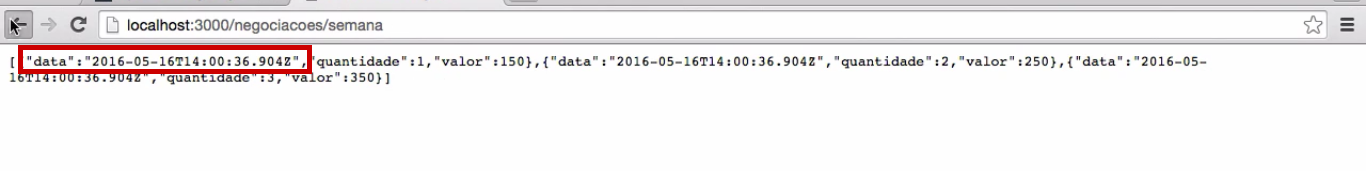
}

}

Vamos entender o que foi feito: xhr.responseText é um texto, o JSON() irá transforma-lo para o formato textual para objeto. Para cada um dos objetos listados, converteremos em uma instância de negociação, no fim, um novo array será gerado. Depois, com o forEach(), percorreremos cada item e adiciono na minha lista de negociações. Dentro do else alteramos as mensagens no caso de erro. No console, será exibido o xhr.responseText e para o usuário, será mostrada uma mensagem de alto nível.

Mas quando executarmos o código como está, teremos problemas.

Se acessarmos o endereço localhost:3000/negociacoes/semana, veremos que a data está no formato de uma string um pouco diferente.



Nós estamos tentando passar objeto.data (com formato de texto) diretamente para Negociacao. A seguir, vamos passar a string objeto data para um construtor de Date().

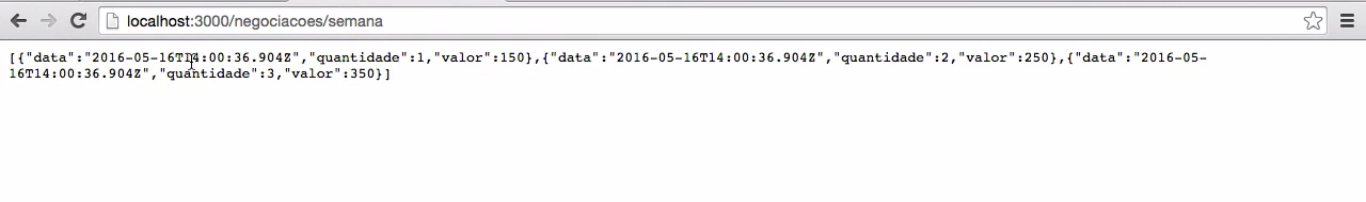
if(xhr.status == 200) {

JSON.parse(xhr.responseText)

.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor))

.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

}



Então, vamos instanciar uma nova data, baseada na string objeto.data. Recarregaremos a página do formulário e tudo funcionará corretamente. Para exibirmos uma mensagem de sucesso, adicionaremos o this.\_mensagem.texto. Faremos ajustes na mensagem de erro também.

if(xhr.status == 200) {

JSON.parse(xhr.responseText)

.map(objeto=> new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor))

.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao))

this.\_mensagem.texto = 'Negociações importadas com sucesso.';

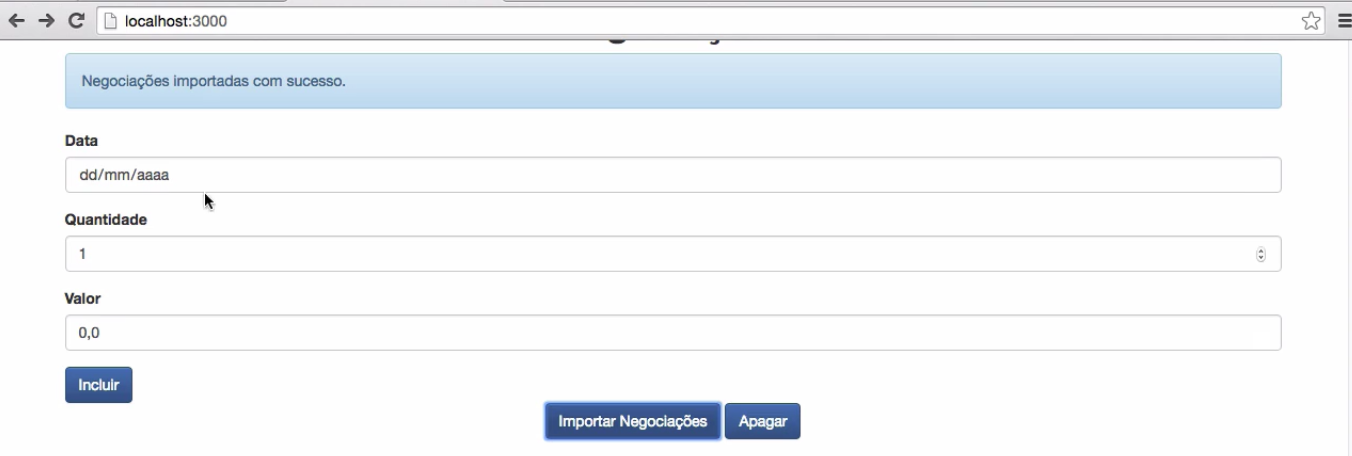
} else {

console.log(xhr.responseText);

this.\_mensagem.texto = 'Não foi possível obter as negociações.';

}

}



A mensagem será exibida corretamente e podemos considerar esta primeira parte finalizada. Depois, faremos uma pequena brincadeira para vermos o que acontece em um caso de erro. Mudaremos o endereço no importaNegociacoes() para negociacoes/xsemana:

importaNegociacoes() {

let xhr = new XMLHttpRequest();

xhr.open('GET', 'negociacoes/xsemana');

//...

Neste caso, quando tentarmos recarregar a página, o usuário verá a seguinte mensagem de erro:



No Console, veremos a mensagem de log:



Conseguimos implementar a nossa solução com AJAX, usando JavaScript "puro" e sem utilizar bibliotecas como jQuery.

Estamos conseguindo importar, mas a responsabilidade de buscar os dados do servidor não é da controller. Por isso, nós vamos isolar o código do importaNegociacoes(), numa classe que será especializada em obter as negociações do servidor e que será utilizada pela controller. Dentro da pasta services, vamos criar NegociacaoService.js.

importaNegociacoes() {

let xhr = new XMLHttpRequest();

xhr.open('GET', 'negociacoes/semana');

xhr.onreadystatechange = () => {

if(xhr.readyState == 4) {

if(xhr.status == 200) {

JSON.parse(xhr.responseText);

.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)

.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

this.\_mensagem.texto = 'Negociações importadas com sucesso.';

} else {

console.log(xhr.responseText);

this.\_mensagem.texto = 'Não foi possível obter as negociações.';

}

}

};

xhr.send();

}

}

A vantagem de isolarmos o código é que se tivermos outra parte do sistema que precisa obter a lista de negociações do servidor, não iremos cortar o código da controller. Nós reutilizaremos em uma nova classe NegociacaoService, que terá o método obterNegociacoesDaSemana() que retornará as negociações da semana:

class NegociacaoService {

obterNegociacoesDaSemana() {

let xhr = new XMLHttpRequest();

xhr.open('GET', 'negociacoes/semana');

xhr.onreadystatechange = () => {

if(xhr.readyState == 4) {

if(xhr.status == 200) {

JSON.parse(xhr.responseText)

.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)

.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

this.\_mensagem.texto = 'Negociações importadas com sucesso.';

} else {

console.log(xhr.responseText);

this.\_mensagem.texto = 'Não foi possível obter as negociações.';

}

}

};

xhr.send();

}

}

O service não terá acesso a View, porque ele não tem referência para os elementos da controller. Logo, removemos this.\_mensagem.texto a mensagem de erro para o usuário. E importaremos o arquivo em 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 src="js/app/services/NegociacaoService.js"></script>

<script>

let negociacaoController = new NegociacaoController();

</script>

Em seguida, no arquivo NegociacaoController.js, vamos adicionar a variável serviceno importaNegociacoes().

importaNegociacoes() {

let service = new NegociacaoService();

service.obterNegociacoesDaSemana();

}

No método obterNegociacoesDaSemana() temos que ter acesso ao retorno, porque será na controller que levantaremos os dados com os quais atualizaremos o model e a View ser renderizada. Para isto, o método receberá a função chamada cb (*callback*).

class NegociacaoService {

obterNegociacoesDaSemana(cb) {

let xhr = new XMLHttpRequest();

xhr.open('GET', 'negociacoes/semana');

xhr.onreadystatechange = () => {

if(xhr.readyState == 4) {

if(xhr.status == 200) {

JSON.parse(xhr.responseText)

.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor))

.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

} else {

console.log(xhr.responseText);

}

}

}

xhr.send();

}

}

Depois, em NegociacaoController.js, usaremos uma *arrow function* com dois valores:

importaNegociacoes() {

let service = new NegociacaoService();

service.obterNegociacoesDaSemana(() => {

});

}

Quando o nosso servidor, via AJAX, buscar a negociação e estiver tudo pronto, ele chamará a função que adicionamos. Agora, vamos inserir um ifpara o caso em que ocorrer um erro.

importaNegociacoes() {

let service = new NegociacaoService();

service.obterNegociacoesDaSemana((err, negociacoes) => {

if(err) {

this.\_mensagem.texto = err;

return;

}

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

this.\_mensagem.texto = 'Negociações importadas com sucesso';

});

}

Se o erro não retornar preenchido, o if não será executado. Com o forEach(), para cada negociação, vamos incorrer em this.\_listaNegociacoes.adiciona().

Também vamos adotar um convenção: em casos de erro, ele será descoberto sempre no primeiro parâmetro e o resultado da operação virá no segundo. Estamos aplicando um padrão que vem do mundo NodeJS, e que recebe o nome de ***Error-First-Callback***.

Então, se ocorrer um erro, exibiremos a mensagem e daremos o retorno. Desta forma, as linhas abaixo do return não serão executadas. Mas no caso em que venha uma negociação, faremos o forEach().

Agora, em NegociacaoService, vamos implementar o *callback*(cb):

class NegociacaoService {

obterNegociacoesDaSemana(cb) {

let xhr = new XMLHttpRequest();

xhr.open('GET', 'negociacoes/semana');

xhr.onreadystatechange = () => {

if(xhr.readyState == 4) {

if(xhr.status == 200) {

cb(null, JSON.parse(xhr.responseText)

.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

} else {

console.log(xhr.responseText);

cb('Não foi possível obter as negociações da semana', null);

}

}

}

xhr.send();

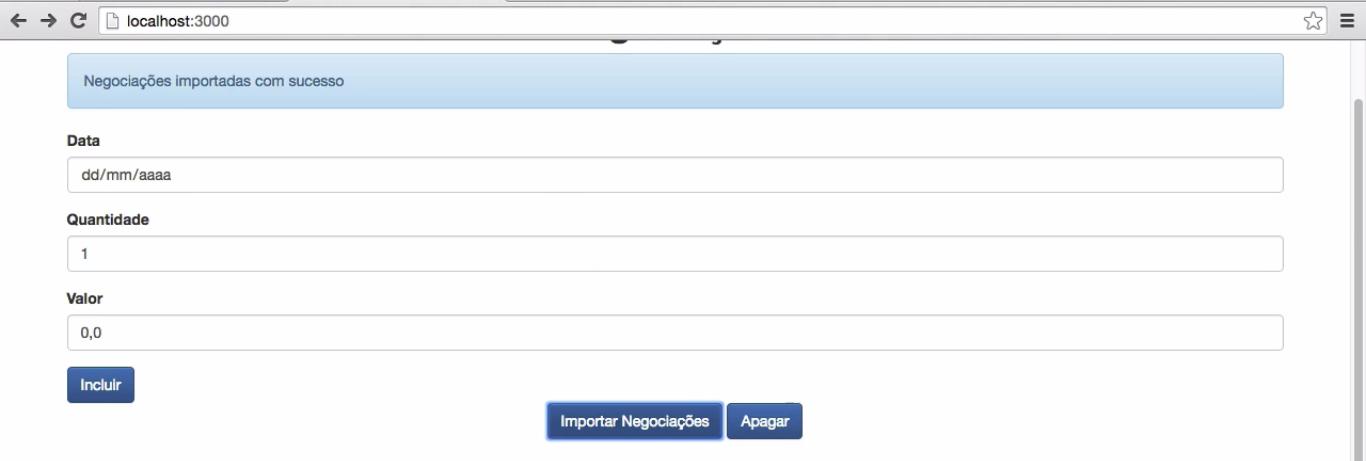
}

}

Se ocorrer um erro, executaremos o cb de alto nível, informando para o usuário que não foi possível obter as negociações.

Esta estratégia de *Error-First* significa que passaremos a função obterNegociacoesDaSemana(), se tiver sucesso receberá o primeiro parâmetro null, indicando que não teve o erro, e no segundo parâmetro, teremos o retorno. Em caso de erro, o primeiro parâmetro passarem será o erro, e o segundo, será o valor null. Temos a opção de deixar o segundo parâmetro em branco também.

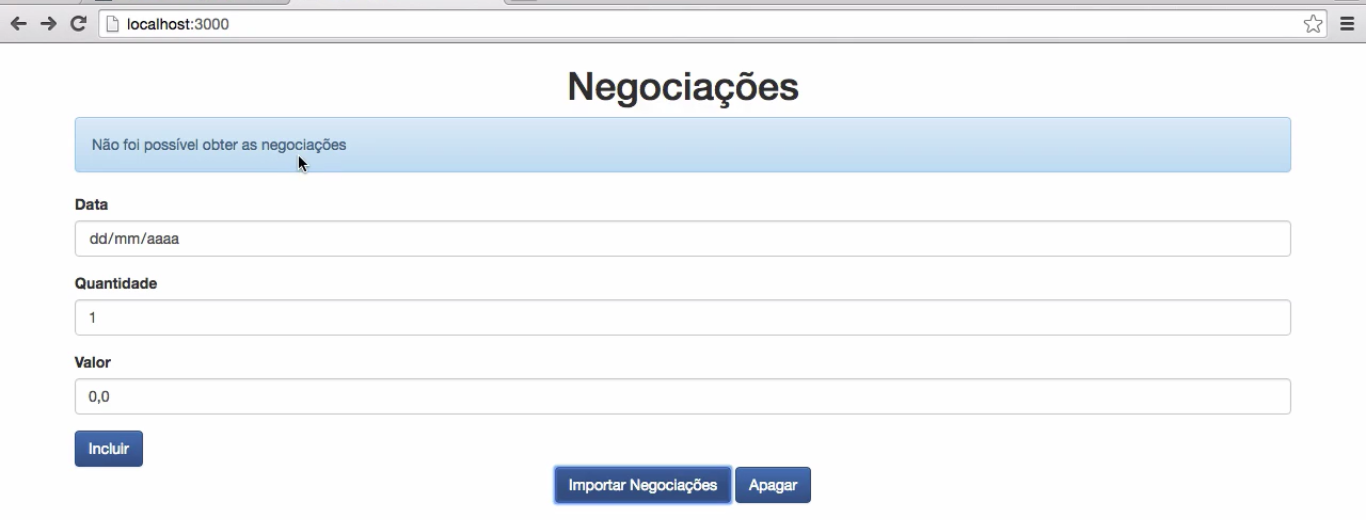
Ao recarregar a página e importar as negociações, veremos a mensagem de sucesso.



Nós isolamos a URL no serviço.

xhr.open('GET', 'negociacoes/semana');

Se tivermos diversas controllers utilizando o serviço, só precisaremos alterar a URL uma única vez. Caso a URL esteja errada, veremos a mensagem de erro:



O código de NegociacaoController ficou mais limpo. Faremos um pequeno ajuste substituindo o err por erro:

importaNegociacoes() {

let service = new NegociacaoService();

service.obterNegociacoesDaSemana((erro,negociacoes) => {

if(erro) {

this.\_mensagem.texto = erro;

return;

}

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

this.\_mensagem.texto = 'Negociações importadas com sucesso';

});

}

Fizemos uma pequena revisão de AJAX. Trabalhamos com requisições assíncronas, usando JavaScript puro. Aprendemos a isolar a lógica que geramos para realizar a requisição do servidor. Aprendemos a trabalhar com a ideia do callback e *Error-First*.

Vimos como usar o objeto XMLHttpRequest para enviar uma requisição AJAX. O estranho foi que nem usamos XML, e sim JSON! Para que então esse XML no nome?

A extensão XML foi um legado que ficou, mas hoje em dia é menos importante e até confuso. O XMLHttpRequest foi criado pela Microsoft (dentro da equipe Outlook), e o XML realmente era o mais usado na época . O Firefox implementou um objeto com o mesmo nome, mas que trabalhava com a estrutura de dados JSON e que acabou se tornando muito popular.

O *Error-first Callback*, ou *errorback*, é um padrão que foi adotado no mundo Node.js. Como você já aprendeu, o *callback* é uma função chamada quando uma tarefa for executada, como uma requisição Ajax ou o acesso ao banco de dados. No entanto, a qualquer momento pode acontecer um erro no processamento e aí vem a questão de como lidar com isso.

A convenção é que cada *callback* receba sempre o erro no primeiro parâmetro. Na função *callback*, basta então verificar esse parâmetro para saber se ocorreu um erro ou não!

Durante o treinamento, lidamos apenas com requisições do tipo GET, porque eu pressupunha que você já conhecia o XMLHttpRequest. Contudo, para dar um brilho ainda maior em sua experiência, vamos praticar um pouquinho mais. Desta vez, realizaremos uma requisição do tipo POST para enviar dados para o servidor. Só fique atento com a volatilidade dos dados cadastrados, isto é, se você reiniciar o servidor, perderá os dados enviados. Aliás, não há problema nenhum nisso, porque a ideia aqui é focar o código no lado do cliente, ou seja, código JavaScript e não código do back-end.

Para conseguir realizar uma requisição do tipo POST você precisa:

* HTTP POST para /negociacoes;
* Usar o cabeçalho Content-type sendo application/json;
* No body um JSON da negociação.

Já preparamos um HTML bem básico para começar o cadastro! Para usá-lo no projeto **crie uma nova página** post.html, dentro do diretório aluraframe/client:

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

<html>

<head>

<meta charset="UTF-8">

<title>Negociações</title>

<link rel="stylesheet" href="css/bootstrap.css">

<link rel="stylesheet" href="css/bootstrap-theme.css">

</head>

<body class="container">

<form class="form">

<div class="form-group">

<label for="data">Data</label>

<input type="date" id="data" class="form-control" required autofocus/>

</div>

<div class="form-group">

<label for="quantidade">Quantidade</label>

<input type="number" min="1" step="1" id="quantidade" class="form-control" value="1" required/>

</div>

<div class="form-group">

<label for="valor">Valor</label>

<input id="valor" type="number" class="form-control" min="0.01" step="0.01" value="0.0" required />

</div>

<button class="btn btn-primary" type="submit" onclick="sendPost(event)">Enviar dados para servidor</button>

</form>

<script>

function sendPost(event) {

event.preventDefault();

console.log("Enviando post");

//aqui você deve ler os dados do formulário

//construir o json

//enviar o XMLHttpRequest

}

</script>

</body>

</html>

Se você achou que eu daria a solução logo de cara, se enganou! Contudo, depois de quebrar a cabeça, veja a solução do problema na minha resposta :)

Obs: Não esqueça de subir o servidor com npm start. Para acessar a página post.html, basta digitar localhost:3000/post.html

Nosso código é bem simplório e não aproveita toda nossa estrutura MVC, mas vale para conhecer melhor o objeto XMLHttpRequest.

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

<!-- código anterior omitido -->

<script>

function sendPost(event) {

event.preventDefault();

console.log("Enviando post");

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

inputData = $('#data');

inputQuantidade = $('#quantidade');

inputValor = $('#valor');

let negociacao = {

data: inputData.value,

quantidade: inputQuantidade.value,

valor: inputValor.value

};

let xhr = new XMLHttpRequest();

xhr.open("POST", "/negociacoes", true);

xhr.setRequestHeader("Content-type", "application/json");

xhr.onreadystatechange = () => {

if (xhr.readyState == 4) {

if (xhr.status == 200) {

inputData.value = '';

inputQuantidade.value = 1;

inputValor.value = 0.0;

inputData.focus();

alert('Negociação enviada com sucesso');

} else {

alert(`Não foi possível enviar a negociação: ${xhr.responseText}`);

}

}

}

xhr.send(JSON.stringify(negociacao));

}

</script>

<!-- código posterior omitido -->

Um detalhe: não podemos enviar o objeto negociacao diretamente, precisamos convertê-lo para string, porque no protocolo HTTP os dados são transmitidos no formato texto. Veja que fizemos o contrário de quando recebemos os dados vindos do servidor. Lá, convertemos string em objeto.

O nosso código está funcionando, conseguimos recarregar e importar as negociações ao clicar no botão correspondente. No entanto, precisamos encontrar uma forma de importar as negociações da semana atual, da passada e da retrasada para popularmos a lista.

Para isto, copiaremos a lógica do método obterNegociacoesDaSemana(), no NegociacaoService, e modificaremos o nome para obterNegociacaoDaSemanaAnterior:

obterNegociacoesDaSemanaAnterior(cb) {

let xhr = new XMLHttpRequest();

xhr.open('GET', 'negociacoes/anterior');

xhr.onreadystatechange = () => {

if(xhr.readyState == 4) {

if(xhr.status == 200) {

cb(null, JSON.parse(xhr.responseText)

.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

} else {

console.log(xhr.responseText);

cb('Não foi possível obter as negociações da semana', null);

}

}

}

xhr.send();

}

}

Observe que alteramos o endereço para negociacoes/anterior. Faremos algo parecido com o trecho referente à semana **retrasada**:

obterNegociacoesDaSemanaRetrasada(cb) {

let xhr = new XMLHttpRequest();

xhr.open('GET', 'negociacoes/retrasada');

xhr.onreadystatechange = () => {

if(xhr.readyState == 4) {

if(xhr.status == 200) {

cb(null, JSON.parse(xhr.responseText)

.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

} else {

console.log(xhr.responseText);

cb('Não foi possível obter as negociações da semana', null);

}

}

}

xhr.send();

}

}

Nós criamos três métodos que irão obter os dados da semana atual, passada e retrasada. Agora, no arquivo NegociacaoController.js, já temos o obterNegociacoesDaSemana em importaNegociacoes:

importaNegociacoes() {

let service = new NegociacaoService();

service.obterNegociacoesDaSemana((erro, negociacoes) => {

if(erro) {

this.\_mensagem.texto = erro;

return;

}

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

this.\_mensagem.texto = 'Negociações importadas com sucesso';

});

}

Agora, adicionaremos obterNegociacoesDaSemanaAnterior e obterNegociacoesDaSemanaRetrasada :

service.obterNegociacoesDaSemanaAnterior((erro, negociacoes) => {

if(erro) {

this.\_mensagem.texto = erro;

return;

}

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

this.\_mensagem.texto = 'Negociações importadas com sucesso';

});

service.obterNegociacoesDaSemanaRetrasada((erro, negociacoes) => {

if(erro) {

this.\_mensagem.texto = erro;

return;

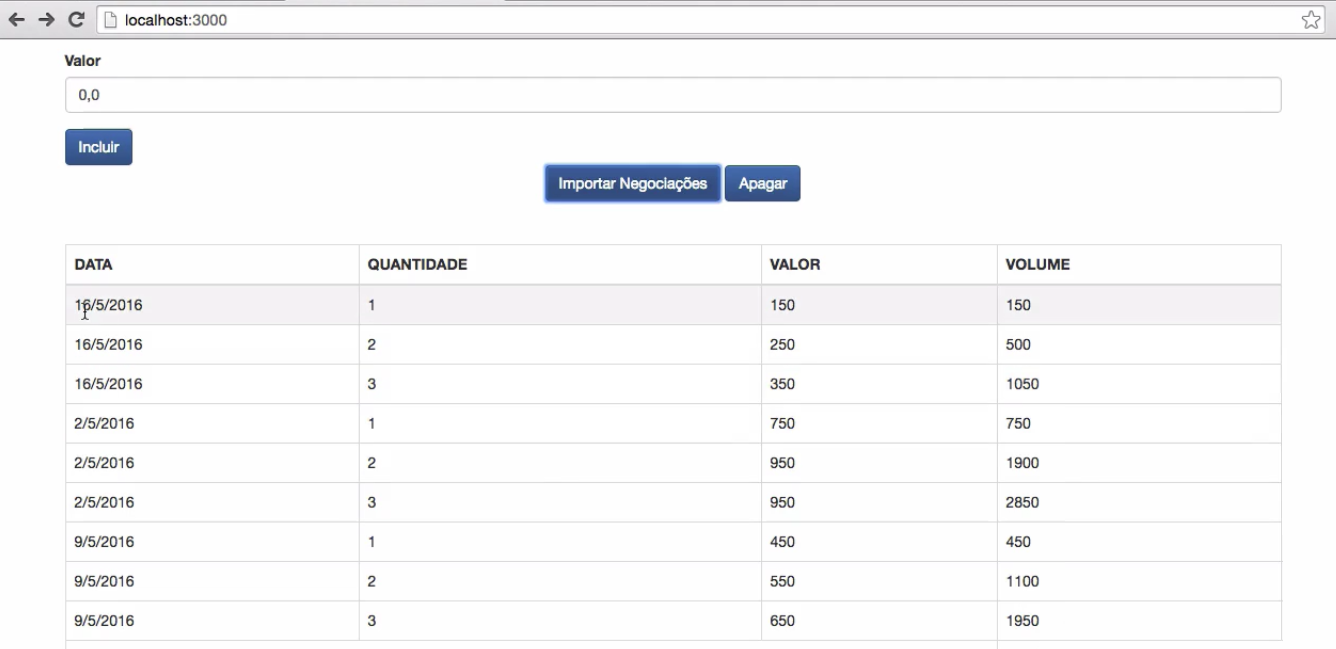
}

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

this.\_mensagem.texto = 'Negociações importadas com sucesso';

});

Vamos realizar as três chamadas do serviço. Depois, recarregaremos a página e conseguiremos que os dados sejam importados corretamente.



Mas temos um pequeno problema... Como as requisições são assíncronas, elas não esperam a operação terminar para só então começarem a executar a próxima.. Segundo uma regra de negócios, essas importações devem estar em ordem: primeiramente, a negociações desta semana, depois, a da semana anterior. Por último, teremos na semana retrasada. Mas se analisarmos as datas da tabela, veremos que elas não estão em ordem. Nós não temos os controles das datas, e como são assíncronas, cada requisição poderá ser executada em diferentes momentos. O que é preciso fazer para que todas sejam executadas em ordem? Começaremos executando a requisição da semana.

importaNegociacoes() {

let service = new NegociacaoService();

service.obterNegociacoesDaSemana((erro, negociacoes) => {

if(erro) {

this.\_mensagem.texto = erro;

return;

}

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

service.obterNegociacoesDaSemanaAnterior((erro, negociacoes) => {

if(erro) {

this.\_mensagem.texto = erro;

return;

}

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

service.obterNegociacoesDaSemanaRetrasada((erro, negociacoes) => {

if(erro) {

this.\_mensagem.texto = erro;

return;

}

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

this.\_mensagem.texto = 'Negociações importadas com sucesso';

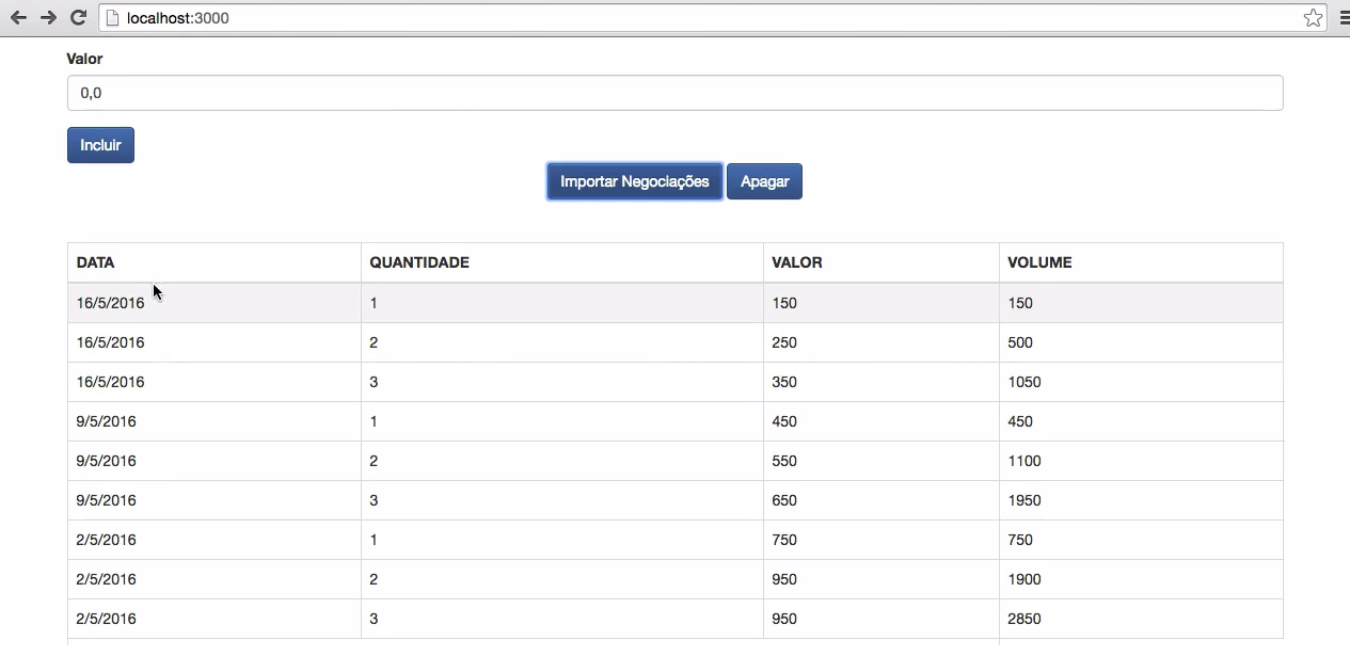
});

});

});

}

Quando a requisição de obterNegociacoesDaSemana terminar, as negociações serão adicionadas na lista. Então, teremos certeza de que a requisição está terminada e que podemos chamar a requisição, referente à semana anterior. E por último, virão as da semana retrasada. Vamos testar se as requisições são executadas na ordem correta.



Agora, as datas estão em ordem.

Quando você for fazer o teste na sua máquina, o servidor está gerando essas negociações aleatoriamente e podem aparecer outras datas.

Conseguimos resolver o nosso problema, mas se analisarmos o "desenho" do código, veremos que ele forma uma **pirâmide**. Costumamos dar o nome de ***Pyramid of Doom***(traduzida para o português, significa **pirâmide do destino**) em situações que isto ocorre e temos uma função aninhada dentro de outra. A pirâmide é um forte indício de que temos problemas de legitimidade do código, na verdade, é o sintoma de um problema maior, o ***Callback Hell***. Ocorre quando temos requisições assíncronas executadas em determinada ordem, que chama vários callbacks seguidos.

Se tivéssemos mais ações que precisassem ser executadas em ordem, teríamos um pirâmide com mais funções. Também vale ressaltar: em uma situação de erro - por exemplo, se a URL estivesse equivocada -, dentro do importaNegociacoes, executaríamos diversos trechos de código referentes ao erro. Testaríamos diversas vezes se ocorreu o erro, porque o código está repetido.

Agora, aplicaremos um padrão de projeto que nos ajudará a lidar com a complexidade da programação assíncrona, além de evitar que o erro seja tratado em diversos lugares.

Vamos melhorar a legibilidade do código, que está repetindo o tratamento de erro:

importaNegociacoes() {

let service = new NegociacaoService();

service.obterNegociacoesDaSemana((erro, negociacoes) => {

if(erro) {

this.\_mensagem.texto = erro;

return;

}

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

service.obterNegociacoesDaSemanaAnterior((erro, negociacoes) => {

if(erro) {

this.\_mensagem.texto = erro;

return;

}

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

service.obterNegociacoesDaSemanaRetrasada((erro, negociacoes) => {

if(erro) {

this.\_mensagem.texto = erro;

return;

}

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

this.\_mensagem.texto = 'Negociações importadas com sucesso';

});

});

});

}

Em NegociacaoService.js, faremos pequenos ajustes nas mensagens de erros dos métodos de obterNegociacoesDaSemana(), obterNegociacoesDaSemanaAnterior() e obterNegociacoesDaSemanaRetrasada, especificando qual semana está sendo trabalhada.

obterNegociacoesDaSemanaAnterior(cb) {

let xhr = new XMLHttpRequest();

xhr.open('GET', 'negociacoes/anterior');

xhr.onreadystatechange = () => {

if(xhr.readyState == 4) {

if(xhr.status == 200) {

cb(null, JSON.parse(xhr.responseText)

.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

} else {

console.log(xhr.responseText);

cb('Não foi possível obter as negociações da semana anterior', null);

}

}

}

xhr.send();

}

}

Desta forma, se tivermos problemas, saberemos que será na semana anterior. Queremos reduzir a complexidade do código de programação assíncrona. Para isto, aplicaremos um padrão de projeto chamado **Promessa** (*Promise*, em inglês).

Vamos comentar o trecho que criamos até agora, porque faremos um antes e depois. Enquanto trabalhamos com o padrão *Promise*, não implementaremos ainda o NegociacaoService. Mas vamos considerar que ele usará o padrão de projeto Promise. Veremos a seguir, como usaremos o NegociacaoService(), no NegociacaoController.js:

importaNegociacoes() {

let service = new NegociacaoService();

let retorno = service.obterNegociacoesDaSemana();

}

Se observarmos o método obterNegociacoesDaSemana(), veremos que este não recebe mais o callback - apenas nos devolverá um valor. Parecerá ser um método **síncrono**. No entanto, ele não é... Porque ele não devolverá a lista de negociações, mas, sim, uma *promise* - que não poderá encontrar o que busca. A promessa é o resultado futuro de uma operação. Quando pensamos no conceito de uma promessa, nos vem a ideia de que "se você cumprir a promessa, **então**algo irá acontecer...". Seguindo está relação com **então**, chamaremos o método then() na *promise*.

importaNegociacoes() {

let service = new NegociacaoService();

let promise = service.obterNegociacoesDaSemana();

promise

.then(negociacoes => {

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao))

this.\_mensagem.texto = 'Negociação da semana obtida com sucesso';

});

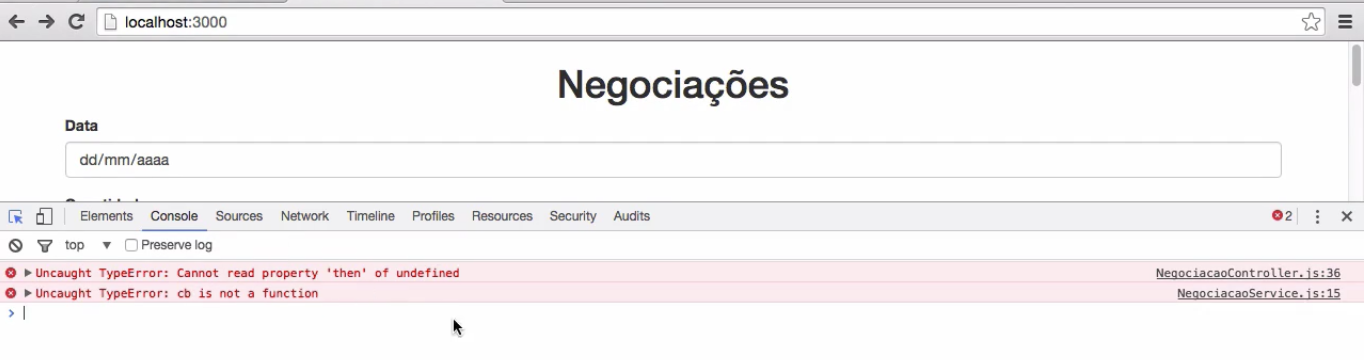
Se a promessa for cumprida, receberemos a lista de negociação e, com esta, poderemos fazer o forEach(). O método obterNegociacoesDaSemana() devolve uma promessa de que tentará obter os dados. Caso a promessa seja cumprida, receberemos uma lista de negociações e exibiremos a mensagem para o usuário.

Para o caso de ocorrer um erro, vamos encadear uma função catch(), na promise.

.catch(erro => this.mensagem.texto = erro);

Com ela, iremos capturar o erro da promise e exibiremos para o usuário.

No entanto, se tentarmos executar o nosso código como está, receberemos várias mensagens de erro no Console.



Ele nos informa que não recebe then(), porque obterNegociacoesDaSemana() não é uma promise. Precisamos fazer esta transformação. O ES6 suporta a promise nativamente, então, o método deverá retornar uma Promise(), que receberá dois parâmetros ( resolve e reject). Em que momento sabemos que os dados são retornados? É onde temos o cb, que não será mais necessário, por isso, vamos substitui-los por resolve:

class NegociacaoService {

obterNegociacoesDaSemana() {

return new Promise((resolve, reject) => {

let xhr = new XMLHttpRequest();

xhr.open('GET', 'negociacoes/semana');

xhr.onreadystatechange = () => {

if(xhr.readyState == 4) {

if(xhr.status == 200) {

resolve(JSON.parse(xhr.responseText)

.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

} else {

console.log(xhr.responseText);

reject('Não foi possível obter as negociações da semana');

}

}

};

xhr.send();

});

O resolve passará diretamente o resultado de JSON.parse(). Observe que não precisaremos mais utilizar o parâmetro (null) do *Error-First-Callback*. Nós passamos direto para o resolve o resultado que será disponibilizado para a função then. E se tivermos algum tipo de erro, chamaremos a função reject(). O que está no resolve, pegaremos dentro do método then() do arquivo NegociacaoController.js e o erro passado para o reject, pegaremos no catch.

importaNegociacoes() {

let service = new NegociacaoService();

let promise = service.obterNegociacoesDaSemana();

promise

.then(negociacoes => {

negociacoes.forEach(negociacao => {

this.\_listaNegociacoes.adiciona(negociacao);

});

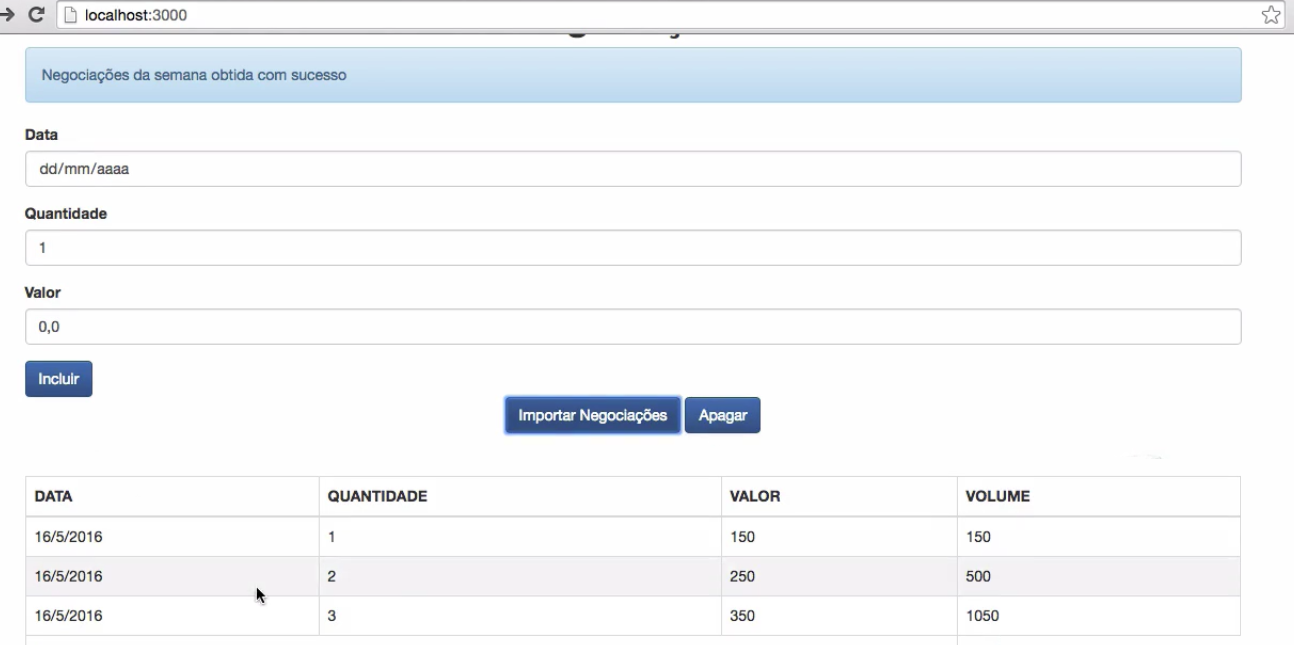
this.\_mensagem.texto = 'Negociações importadas com sucesso'

})

.catch(erro => this.\_mensagem.texto = erro);

}

Se testarmos no navegador, veremos que tudo está funcionando e conseguiremos importar as negociações corretamente.



Aplicaremos o padrão Promise nos outros métodos do NegociacaoService.js.

obterNegociacoesDaSemanaAnterior() {

return new Promise((resolve, reject) => {

let xhr = new XMLHttpRequest();

xhr.open('GET', 'negociacoes/anterior');

xhr.onreadystatechange = () => {

if(xhr.readyState == 4) {

if(xhr.status == 200) {

resolve(JSON.parse(xhr.responseText)

.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

} else {

console.log(xhr.responseText);

reject('Não foi possível obter as negociações da semana anterior');

}

}

}

xhr.send();

});

}

obterNegociacoesDaSemanaRetrasada() {

return new Promise((resolve, reject) => {

let xhr = new XMLHttpRequest();

xhr.open('GET', 'negociacoes/retrasada');

xhr.onreadystatechange = () => {

if(xhr.readyState == 4) {

if(xhr.status == 200) {

resolve(JSON.parse(xhr.responseText)

.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

} else {

console.log(xhr.responseText);

reject('Não foi possível obter as negociações da semana retrasada');

}

}

}

xhr.send();

});

}

Em seguida, no NegociacoesController.js, não iremos mais declarar a variável Promise:

importaNegociacoes() {

let service = new NegociacaoService();

service.obterNegociacoesDaSemana()

.then(negociacoes => {

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

this.\_mensagem.texto = 'Negociações da semana obtidas com sucesso';

})

.catch(erro => this.\_mensagem.texto = erro);

service.obterNegociacoesDaSemanaAnterior()

.then(negociacoes => {

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

this.\_mensagem.texto = 'Negociações da semana obtidas com sucesso';

})

.catch(erro => this.\_mensagem.texto = erro);

service.obterNegociacoesDaSemanaRetrasada()

.then(negociacoes => {

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

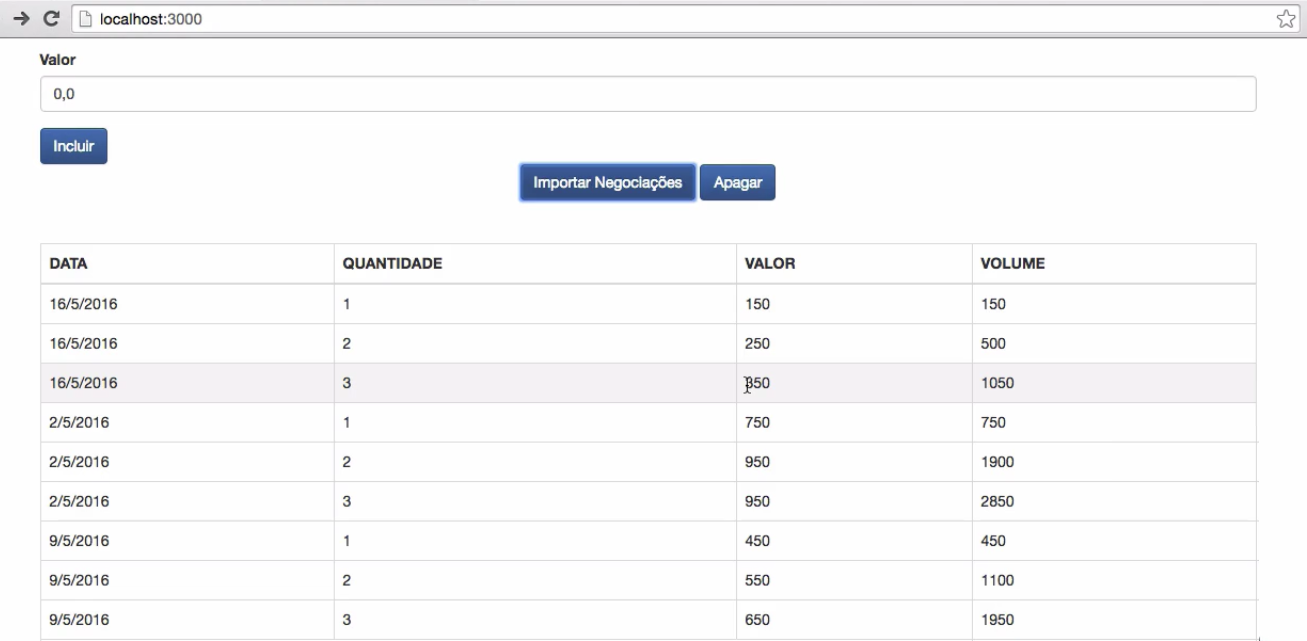
this.\_mensagem.texto = 'Negociações da semana obtidas com sucesso';

})

.catch(erro => this.\_mensagem.texto = erro);

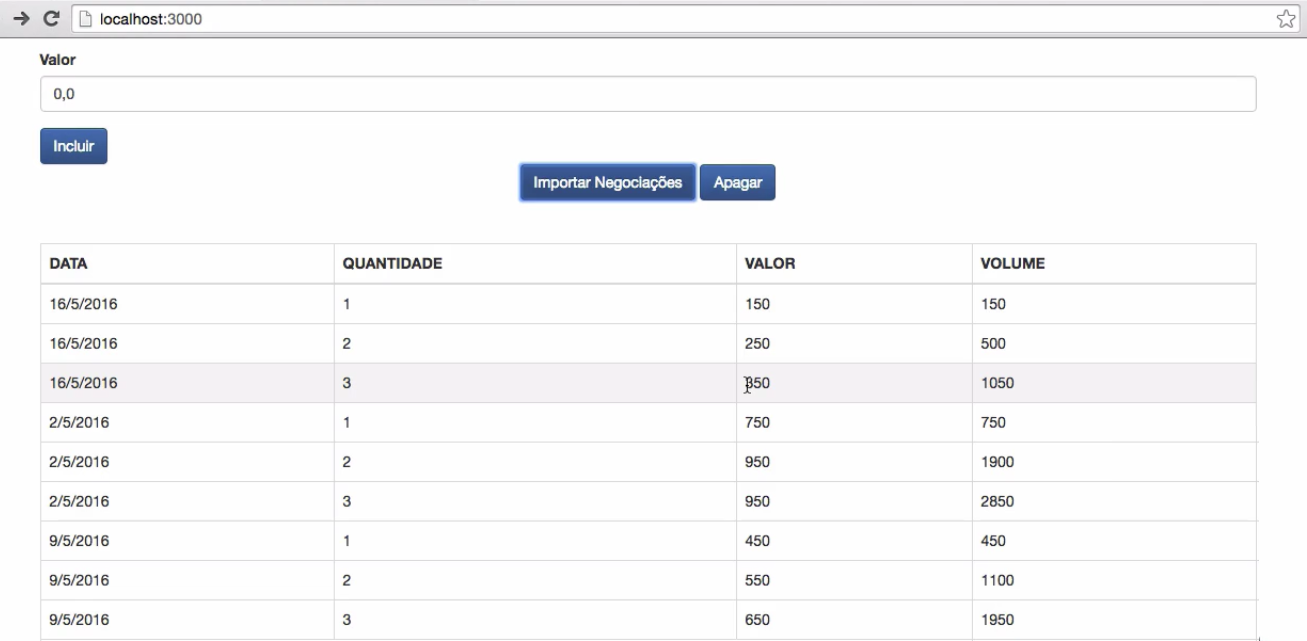
}

No navegador, dessa vez, teremos problemas com as negociações importadas. Novamente, as negociações estão fora da ordem.



A ordem de execução das Promises está incorreta. Mais adiante, entenderemos por que isso aconteceu e descobriremos a solução.

Caímos no mesmo problema enfrentado anteriormente: as negociações são exibidas fora da ordem das datas.



Isto ocorreu porque a Promise é assíncrona e as negociação são executadas de forma independente.

importaNegociacoes() {

let service = new NegociacaoService();

service.obterNegociacoesDaSemana()

.then(negociacoes => {

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

this.\_mensagem.texto = 'Negociações da semana obtidas com sucesso';

})

.catch(erro => this.\_mensagem.texto = erro);

service.obterNegociacoesDaSemanaAnterior()

.then(negociacoes => {

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

this.\_mensagem.texto = 'Negociações da semana obtidas com sucesso';

})

.catch(erro => this.\_mensagem.texto = erro);

service.obterNegociacoesDaSemanaRetrasada()

.then(negociacoes => {

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

this.\_mensagem.texto = 'Negociações da semana obtidas com sucesso';

})

.catch(erro => this.\_mensagem.texto = erro);

}

Outro problema é que estamos tratando a mensagem de erro em cada uma das promises. Como resolveremos isto? A promise possui um recurso com o qual temos uma sequência de operações assíncronas, que será executada em uma determinada ordem.

Uma maneira de executarmos todas as promises em ordem e obtermos todos os resultado de uma vez só é usar a função Promise.all, que receberá um array com as promises. Vamos fazer a segunda refatoração do nosso código:

importaNegociacoes() {

let service = new NegociacaoService();

Promise.all(

service.obterNegociacoesDaSemana(),

service.obterNegociacoesDaSemanaAnterior(),

service.obterNegociacoesDaSemanaRetrasada()

).then(negociacoes => {

console.log(negociacoes)

})

.catch(erro => this.\_mensagem.texto = erro);

}

Pedimos para que o Promise.all() resolvesse todas as promises na ordem indicada. Mas iremos obter os dados da Promise com o then(). Caso ocorra um erro, trataremos com o catch(). E se der uma mensagem de erro específica de obterNegociacoesDaSemana(), o catch() será chamado - sem precisar ser chamado diversas vezes.

No entanto, se executarmos nosso código no navegador, teremos uma mensagem de erro, porque o Promise.allreceberá a lista de promises dentro de um array, ou seja, elas deverão estar entre colchetes ([]). Com a pequena alteração, o trecho do código ficará assim:

importaNegociacoes() {

let service = new NegociacaoService();

Promise.all([

service.obterNegociacoesDaSemana(),

service.obterNegociacoesDaSemanaAnterior(),

service.obterNegociacoesDaSemanaRetrasada()]

).then(negociacoes => {

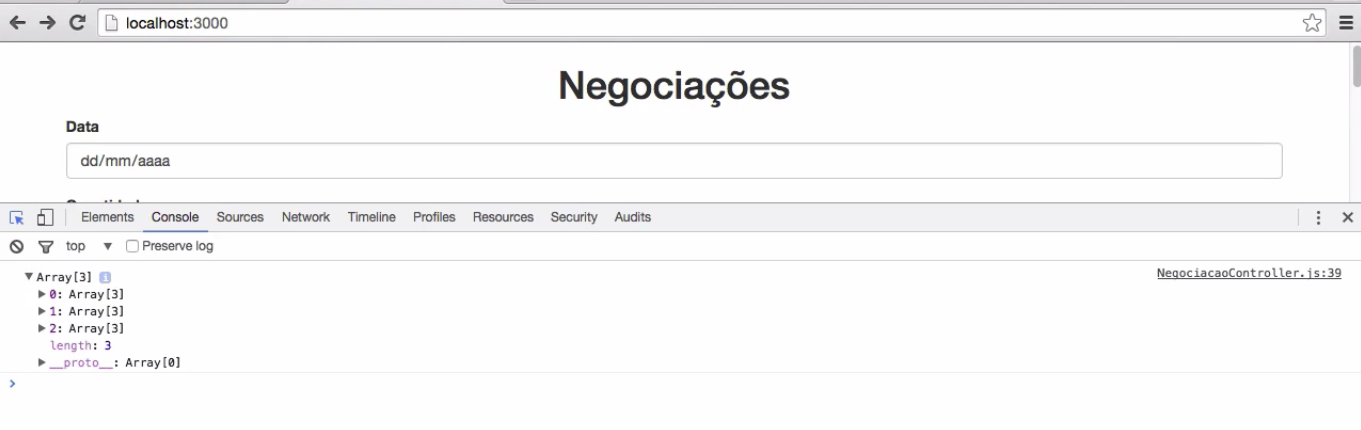
console.log(negociacoes)

})

.catch(erro => this.\_mensagem.texto = erro);

}

Agora, faremos um teste para verificar se a página está funcionando. Após clicarmos em "Importar Negociações", o array será exibido no Console.



A grande vantagem da função Promise.all() é que todas as promises do array serão exibidos na sequência e o resultado estará em negociacoes, e em caso de erro, ele será capturado uma única vez. No entanto, a negociacaoretornada não é equivalente à lista de negociações, mas sim, cada posição do array será uma lista de negociações. Para resolver a questão, usaremos o forEach(), mas teremos que pensar bem no que faremos.

importaNegociacoes() {

let service = new NegociacaoService();

Promise.all([

service.obterNegociacoesDaSemana(),

service.obterNegociacoesDaSemanaAnterior(),

service.obterNegociacoesDaSemanaRetrasada()]

).then(negociacoes => {

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

this.\_mensagem.texto = 'Negociações importadas com sucesso';

})

.catch(erro => this.\_mensagem.texto = erro);

Se usarmos esta opção, o negociacao dentro do forEach() será uma lista. Seguiremos por outro caminho... Antes de chegarmos até o forEach(), executaremos uma **transformação** do array que possui outros três dentro de si. Com o reduce(), criaremos um array que contem apenas um elemento, contendo todos as negociações. Nós faremos *flatten* - achatar - o array.

importaNegociacoes() {

let service = new NegociacaoService();

Promise.all([

service.obterNegociacoesDaSemana(),

service.obterNegociacoesDaSemanaAnterior(),

service.obterNegociacoesDaSemanaRetrasada()]

).then(negociacoes => {

negociacoes

.reduce((arrayAchatado, array) => arrayAchatado.concat(array), [])

.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

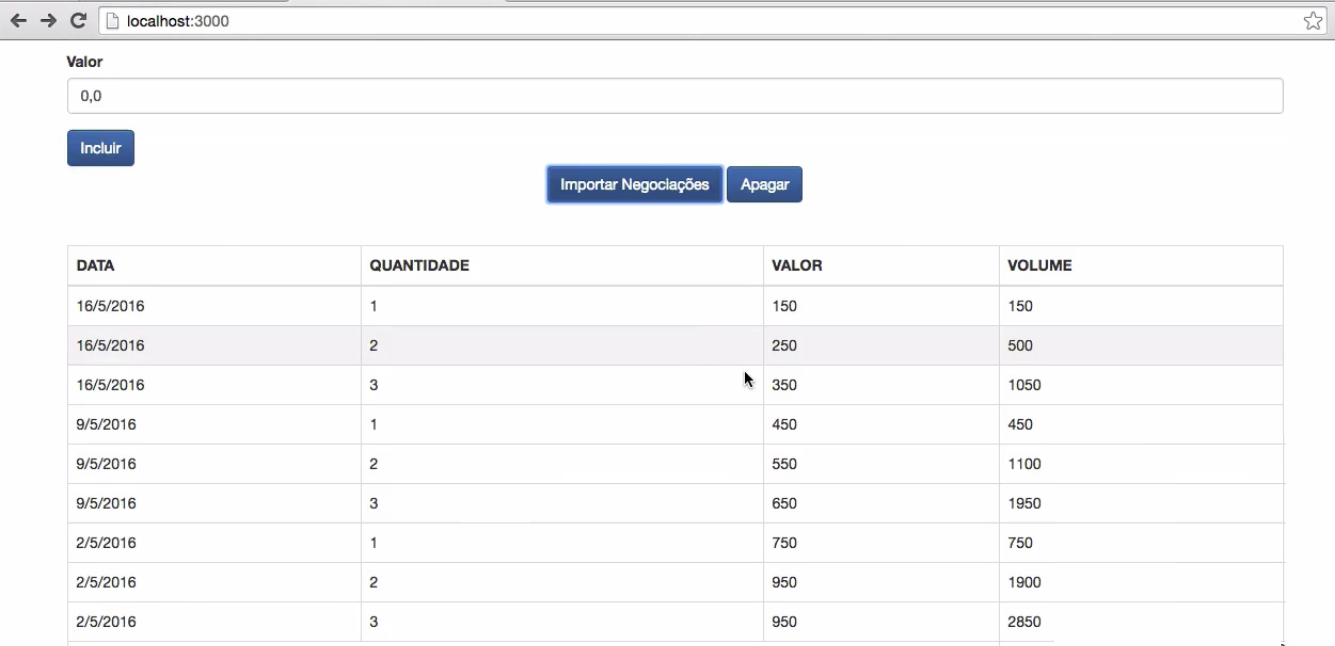
this.\_mensagem.texto = 'Negociações importadas com sucesso';

})

.catch(erro => this.\_mensagem.texto = erro);

}

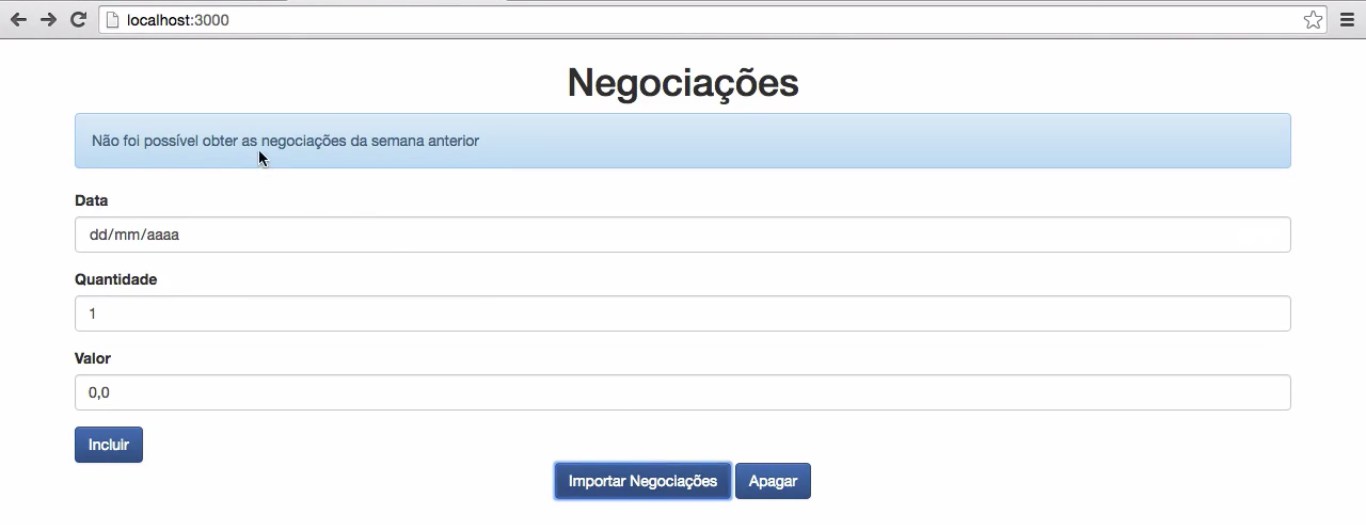
Observe que também usamos a função concat(), que concatenará o array da primeira posição de negociacoes. No fim, o reduce devolverá uma única lista cheia de negociações e o forEach() será executado sem problemas.



Primeiro ele resolveu as negociações da semana atual, depois da semana anterior e por último, da semana retrasada. E como só tratamos o erro em um único lugar, se provocarmos o erro modificando a URL da semana anterior, em NegociacaoService.js, por exemplo:

xhr.open('GET', 'negociacoes/anteriorx');

A mensagem de erro será exibida para o usuário.



O mesmo ocorreria se o problema estivesse relacionado com as demais semanas.

Vimos como solucionar questões assíncronas com o padrão de projeto Promise. Mas ainda podemos extrair do NegociacaoService, a parte que lida com XMLHttpRequest.

Se quisermos realizar um requisição do tipo GET, precisaremos repetir isso no código e preparar o onreadystatechange...

class NegociacaoService

obterNegociacoesDaSemanaAnterior() {

return new Promise((resolve, reject) => {

let xhr = new XMLHttpRequest();

xhr.open('GET', 'negociacoes/anterior');

xhr.onreadystatechange = () => {

if(xhr.readyState == 4) {

if(xhr.status == 200) {

resolve(JSON.parse(xhr.responseText)

.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor));

}

//...

Para facilitar o processo, criaremos um serviço que chamaremos de HttpService.js, que ficará na pasta Service. A nova classe não terá nenhum atributo ou método estático. Ele terá o método get(), com a URL que queremos conectar, e que retornará uma Promise.

class HttpService {

get(url) {

return new Promise((resolve, reject) => {

let xhr = new XMLHttpRequest();

xhr.open('GET', url);

xhr.onreadystatechange = () => {

if(xhr.readyState == 4) {

if(xhr.status == 200) {

resolve(JSON.parse(xhr.responseText));

} else {

console.log(xhr.responseText);

reject(xhr.responseText);

}

}

}

xhr.send();

});

}

}

Observe que aproveitamos o padrão Promise e, em caso de erro, repassaremos o que veio do servidor. Também vamos passar o que veio do servidor em caso de sucesso, JSON.parse(xhr.responseText). Se quisermos usar o HttpService, faremos o seguinte:

service = new HttpService();

service.get('negociacoes/semana').then(negociacoes => ????);

Com then(), teremos acesso à lista de negociações e poderemos fazer as negociações que desejamos. Deixaremos desta forma genérica, e você poderá fazer melhorias... Nossa intenção era mostrar o conceito para isolarmos este código.

Agora, NegociacaoService é dependente de HttpService, vamos declarar isto no construtor da classe.

class NegociacaoService {

constructor() {

this.http = new HttpService();

}

}

Em seguida, faremos grandes alterações no return do obterNegociacoesDaSemana():

class NegociacaoService {

constructor() {

this.http = new HttpService();

}

obterNegociacoesDaSemana() {

return new Promise((resolve, reject) => {

this.http

.get('negociacoes/semana')

.then(negociacoes => {

resolve(negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

})

.catch(erro => {

console.log(erro);

reject('Não foi possível obter as negociações da semana');

})

});

}

O reject será responsável pela mensagem que será exibida para o usuário.

Vamos rever o que fizemos até aqui... Pedimos para o serviço \_http buscar negociacoes/semana e, no retorno, já teremos objetos "parseados". Mas no caso do NegociacaoService que quando usamos este endereço, trata-se de uma lista de negociações com um objeto - que contem dado, quantidade e valor. Nós converteremos esta lista para outra em que teremos instâncias de negociações e passaremos para o resolve.

Agora, só falta importarmos o HttpService.js 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 src="js/app/services/NegociacaoService.js"></script>

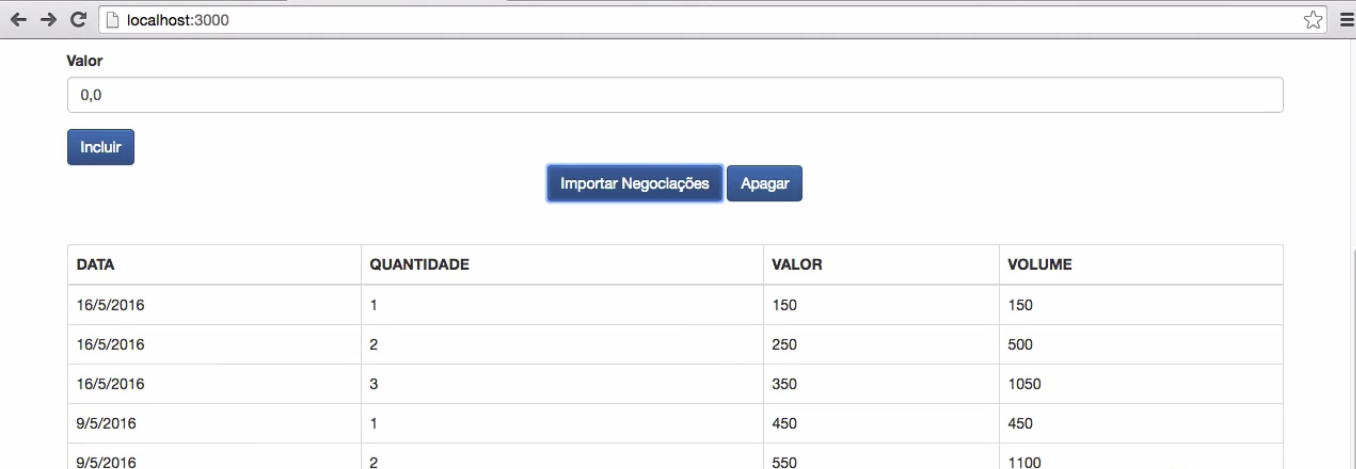
<script src="js/app/services/HttpService.js"></script>

<script>

let negociacaoController = new NegociacaoController();

</script>

Carregaremos novamente a página e clicaremos em "Importar Negociações".



Continuaremos o trabalho de conversão para usarmos o HttpService. As alterações serão feitas em obterNegociacoesDaSemanaAnterior e obterNegociacoesDaSemanaRetrasada:

obterNegociacoesDaSemanaAnterior() {

return new Promise((resolve, reject) => {

this.http

.get('negociacoes/anterior')

.then(negociacoes => {

console.log(negociacoes);

resolve(negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

})

.catch(erro => {

console.log(erro);

reject('Não foi possível obter as negociações da semana anterior');

})

});

obterNegociacoesDaSemanaRetrasada() {

return new Promise((resolve, reject) => {

this.http

.get('negociacoes/retrasada')

.then(negociacoes => {

console.log(negociacoes);

resolve(negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

})

.catch(erro => {

console.log(erro);

reject('Não foi possível obter as negociações da semana retrasada');

})

});

Depois de salvar as mudanças, podemos executar o código e a página funcionará corretamente.

Nós conseguimos desmembrar a parte de HttpService. Caso você queira criar um POST ou PUT ou DELETE, basta acrescentar estes métodos no HttpService, usando o padrão Promise.

Se quisermos realizar um requisição do tipo GET, precisaremos repetir isso no código e preparar o onreadystatechange...

class NegociacaoService

obterNegociacoesDaSemanaAnterior() {

return new Promise((resolve, reject) => {

let xhr = new XMLHttpRequest();

xhr.open('GET', 'negociacoes/anterior');

xhr.onreadystatechange = () => {

if(xhr.readyState == 4) {

if(xhr.status == 200) {

resolve(JSON.parse(xhr.responseText)

.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor));

}

//...

Para facilitar o processo, criaremos um serviço que chamaremos de HttpService.js, que ficará na pasta Service. A nova classe não terá nenhum atributo ou método estático. Ele terá o método get(), com a URL que queremos conectar, e que retornará uma Promise.

class HttpService {

get(url) {

return new Promise((resolve, reject) => {

let xhr = new XMLHttpRequest();

xhr.open('GET', url);

xhr.onreadystatechange = () => {

if(xhr.readyState == 4) {

if(xhr.status == 200) {

resolve(JSON.parse(xhr.responseText));

} else {

console.log(xhr.responseText);

reject(xhr.responseText);

}

}

}

xhr.send();

});

}

}

Observe que aproveitamos o padrão Promise e, em caso de erro, repassaremos o que veio do servidor. Também vamos passar o que veio do servidor em caso de sucesso, JSON.parse(xhr.responseText). Se quisermos usar o HttpService, faremos o seguinte:

service = new HttpService();

service.get('negociacoes/semana').then(negociacoes => ????);

Com then(), teremos acesso à lista de negociações e poderemos fazer as negociações que desejamos. Deixaremos desta forma genérica, e você poderá fazer melhorias... Nossa intenção era mostrar o conceito para isolarmos este código.

Agora, NegociacaoService é dependente de HttpService, vamos declarar isto no construtor da classe.

class NegociacaoService {

constructor() {

this.http = new HttpService();

}

}

Em seguida, faremos grandes alterações no return do obterNegociacoesDaSemana():

class NegociacaoService {

constructor() {

this.http = new HttpService();

}

obterNegociacoesDaSemana() {

return new Promise((resolve, reject) => {

this.http

.get('negociacoes/semana')

.then(negociacoes => {

resolve(negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

})

.catch(erro => {

console.log(erro);

reject('Não foi possível obter as negociações da semana');

})

});

}

O reject será responsável pela mensagem que será exibida para o usuário.

Vamos rever o que fizemos até aqui... Pedimos para o serviço \_http buscar negociacoes/semana e, no retorno, já teremos objetos "parseados". Mas no caso do NegociacaoService que quando usamos este endereço, trata-se de uma lista de negociações com um objeto - que contem dado, quantidade e valor. Nós converteremos esta lista para outra em que teremos instâncias de negociações e passaremos para o resolve.

Agora, só falta importarmos o HttpService.js 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 src="js/app/services/NegociacaoService.js"></script>

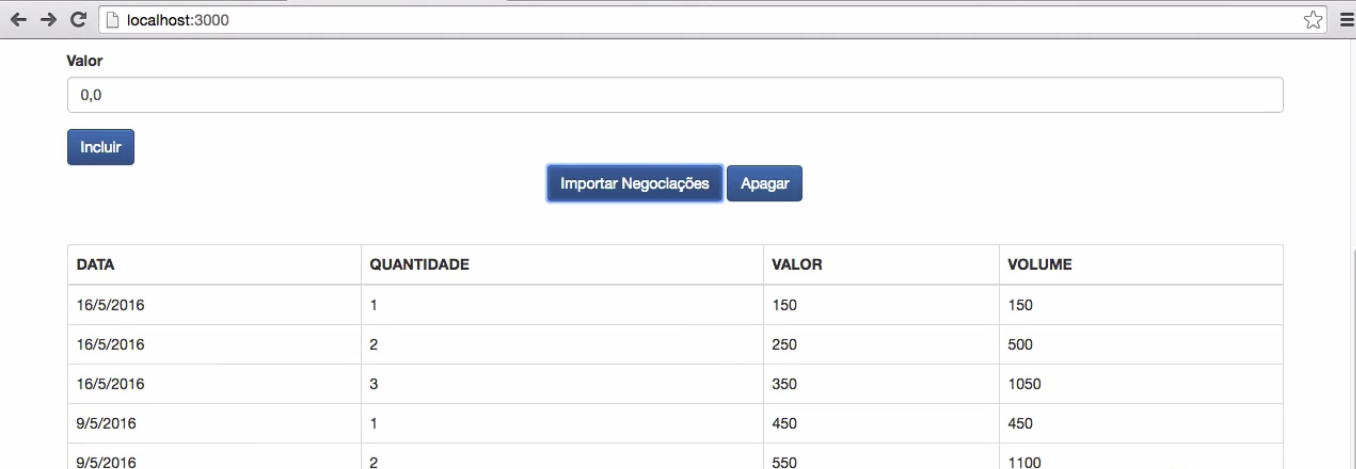
<script src="js/app/services/HttpService.js"></script>

<script>

let negociacaoController = new NegociacaoController();

</script>

Carregaremos novamente a página e clicaremos em "Importar Negociações".



Continuaremos o trabalho de conversão para usarmos o HttpService. As alterações serão feitas em obterNegociacoesDaSemanaAnterior e obterNegociacoesDaSemanaRetrasada:

obterNegociacoesDaSemanaAnterior() {

return new Promise((resolve, reject) => {

this.http

.get('negociacoes/anterior')

.then(negociacoes => {

console.log(negociacoes);

resolve(negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

})

.catch(erro => {

console.log(erro);

reject('Não foi possível obter as negociações da semana anterior');

})

});

obterNegociacoesDaSemanaRetrasada() {

return new Promise((resolve, reject) => {

this.http

.get('negociacoes/retrasada')

.then(negociacoes => {

console.log(negociacoes);

resolve(negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

})

.catch(erro => {

console.log(erro);

reject('Não foi possível obter as negociações da semana retrasada');

})

});

Depois de salvar as mudanças, podemos executar o código e a página funcionará corretamente.

Nós conseguimos desmembrar a parte de HttpService. Caso você queira criar um POST ou PUT ou DELETE, basta acrescentar estes métodos no HttpService, usando o padrão Promise.

## Revisão de Promise

Para ficarmos melhores do que já somos em promises, crie o arquivo dissecando-uma-promise.htmle cole o código abaixo:

<!-- dissecando-uma-promise.html -->

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Dissecando uma promise</title>

</head>

<body>

<script>

let promise = new Promise((resolve, reject) => {

setTimeout(() => resolve('PROMISE RESOLVIDA'), 5000);

});

promise.then(resultado => console.log(resultado));

</script>

</body>

</html>

Agora, abra a página no Chrome e verifique no console do navegador. Depois de 5 segundos será exibida a mensagem **PROMISE RESOLVIDA**. Mas o que aconteceu durante todo esse processo?

Bom, veja que a variável promise recebeu uma instância de Promise. O construtor de Promiserecebe uma função como parâmetro. É essa função passada como parâmetro que será chamada internamente pela Promise, quando for criada. Como é a própria Promise que chama essa função, ela passa sempre dois parâmetros para ela nesta ordem: a função na qual passamos o valor de sucesso e a função que passamos o valor de fracasso.

let promise = new Promise((resolve, reject) => {

// é aqui dentro que definimos o que será passado para `resolve` e o que será passado para `reject`.

});

Bom, criar uma Promise não é suficiente. Se olharmos o fragmento acima, em nenhum momento estamos dizendo o que acontecerá se a promessa for cumprida. Para efeito didático, colocarei um setTimeout de 5 segundos dentro do corpo da Promise. Só depois de 5 segundos passaremos o resultado da nossa operação para o resolve:

let promise = new Promise((resolve, reject) => {

setTimeout(() => resolve('PROMISE RESOLVIDA'), 5000);

});

## Obtendo o retorno da ação

Perfeito, temos a variável promise, que guarda uma instância de Promise, o resultado futuro de uma ação. Mas em que parte do código pegamos o resultado dessa ação quando concluída?

É por meio do método then, da instância de Promise que temos acesso ao resultado da ação. O método then recebe uma função e nela temos acesso sempre como primeiro parâmetro ao resultado da ação. Internamente em nossa Promise, é o valor passado para resolve que estará disponível para a função then. Sendo assim, em then, só depois de 5 segundos teremos acesso ao resultado a ação, que é uma string, mas poderia ser qualquer outro tipo de dado.

let promise = new Promise((resolve, reject) => {

setTimeout(() => resolve('PROMISE RESOLVIDA'), 5000);

});

// imprime no console "PROMISE RESOLVIDA"

promise.then(resultado => console.log(resultado));

É interessante saber que, como nosso código é assíncrono, não sabemos quando nossa promessa será cumprida (sabemos que são 5 segundos, mas se fosse uma conexão de rede não teríamos tanta certeza assim, certo?).

**Quero que vocês façam uma pequena alteração no código:**

<!-- dissecando-uma-promise.html -->

<script>

let promise = new Promise((resolve, reject) => {

setTimeout(() => resolve('PROMISSE RESOLVIDA'), 5000);

});

promise.then(resultado => console.log(resultado));

console.log('FIM'); // novidade aqui!

</script>

Como a promise é assíncrona e não bloqueia a execução do nosso código, veremos impresso no console as mensagens nesta ordem:

FIM

PROMISE CONCLUÍDA

## Lidando com erros

E se algo sair errado? Onde trataremos o erro? Quando há algum erro dentro do corpo da nossa Promise, cabe ao desenvolvedor capturar esse erro e passá-lo para a função reject:

<script>

let promise = new Promise((resolve, reject) => {

console.log(resolve);

setTimeout(() => reject('HOUVE PROBLEMAS'), 5000);

});

promise

.then(resultado => console.log(resultado));

</script>

Depois de 5 segundos, nossa promise será rejeitada, indicando que houve algum erro. Mas onde teremos acesso à causa da rejeição? Basta, depois de then, encadearmos uma chamada à função catch:

<script>

let promise = new Promise((resolve, reject) => {

console.log(resolve);

setTimeout(() => reject('HOUVE PROBLEMAS'), 5000);

});

promise

.then(resultado => console.log(resultado))

.catch(erro => console.log(erro)); // exibe no console HOUVE PROBLEMAS

</script>

Mas é claro que queremos que nossa promise esteja preparada para resolver ou rejeitar. Para efeito didático, vamos colocar um variável booleana. Se for true, resolvemos, se for false, rejeitamos. Dessa forma, você pode brincar e simular quando a promise é resolvida ou não:

<script>

let ok = false;

let promise = new Promise((resolve, reject) => {

// como temos mais de uma instrução, precisamos colocar um bloco em nossa arrow function! Lembrou?

setTimeout(() => {

if(ok) {

resolve('PROMISE CONCLUÍDA');

} else {

reject('HOUVE PROBLEMAS');

}

}, 5000);

});

promise

.then(resultado => console.log(resultado))

.catch(erro => console.log(erro));

</script>

VER

Promises (**promessas**) agora fazem parte da linguagem JavaScript a partir do ES6. Elas representam o resultado futuro de uma ação, que pode ser de sucesso ou fracasso. Elas visam tornar códigos assíncronos mais legíveis e fáceis de manter, evitando o Callback Hell. Uma ou outra chamada assíncrona não é problemática, o problema é quando temos uma sucessão de chamadas assíncronas e o modo tradicional de lidar com elas, aninhando callbacks, torna o código difícil de ler e manter, principalmente o tratamento de erros.

Durante esta seção criamos a classe NegociacaoService. Esta classe centraliza operações que realizamos com nosso back-end, mais notadamente aquelas que buscam negociações. Ela também serve para encapsular o uso outra classe que criamos, a HttpService. Esta última, encapsula a complexidade de se realizar requisições Ajax devolvendo uma promise para determinadas operações.

Vejamos o código de HttpService mais uma vez:

class HttpService {

get(url) {

return new Promise((resolve, reject) => {

let xhr = new XMLHttpRequest();

xhr.open('GET', url);

xhr.onreadystatechange = () => {

if(xhr.readyState == 4) {

if(xhr.status == 200) {

resolve(JSON.parse(xhr.responseText));

} else {

reject(xhr.responseText);

}

}

};

xhr.send();

});

}

post(url, dado) {

return new Promise((resolve, reject) => {

let xhr = new XMLHttpRequest();

xhr.open("POST", url, true);

xhr.setRequestHeader("Content-type", "application/json");

xhr.onreadystatechange = () => {

if (xhr.readyState == 4) {

if (xhr.status == 200) {

resolve(JSON.parse(xhr.responseText));

} else {

reject(xhr.responseText);

}

}

};

xhr.send(JSON.stringify(dado)); // usando JSON.stringifly para converter objeto em uma string no formato JSON.

});

}

}

Todos os métodos get e post retornam uma promise. Até ai tudo bem, nenhuma novidade.

Agora, vejamos o código da classe NegociacaoService que usa HttpService:

class NegociacaoService {

constructor() {

this.\_http = new HttpService();

}

obterNegociacoesDaSemana() {

return new Promise((resolve, reject) => {

this.\_http

.get('negociacoes/semana')

.then(negociacoes => {

console.log(negociacoes);

resolve(negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

})

.catch(erro => {

console.log(erro);

reject('Não foi possível obter as negociações da semana');

});

});

}

obterNegociacoesDaSemanaAnterior() {

return new Promise((resolve, reject) => {

this.\_http

.get('negociacoes/anterior')

.then(negociacoes => {

console.log(negociacoes);

resolve(negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

})

.catch(erro => {

console.log(erro);

reject('Não foi possível obter as negociações da semana anterior');

});

});

}

obterNegociacoesDaSemanaRetrasada() {

return new Promise((resolve, reject) => {

this.\_http

.get('negociacoes/retrasada')

.then(negociacoes => {

console.log(negociacoes);

resolve(negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

})

.catch(erro => {

console.log(erro);

reject('Não foi possível obter as negociações da semana retrasada');

});

});

}

}

Veja que seus métodos também devolvem uma promise. Mas espere um pouco! Se a classe NegociacaoService já recebe o resultado de HttpService que é uma promise, porque no lugar de criarmos uma nova promise, não fazemos com que os métodos de NegociacaoService retorne a promise de HttpService? Sim, isso é possível!

O primeiro passo, é removermos o new Promise((resolve, reject) => {}) de todos os métodos de NegociacaoService:

class NegociacaoService {

constructor() {

this.\_http = new HttpService();

}

obterNegociacoesDaSemana() {

this.\_http

.get('negociacoes/semana')

.then(negociacoes => {

console.log(negociacoes);

resolve(negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

})

.catch(erro => {

console.log(erro);

reject('Não foi possível obter as negociações da semana');

});

}

obterNegociacoesDaSemanaAnterior() {

this.\_http

.get('negociacoes/anterior')

.then(negociacoes => {

console.log(negociacoes);

resolve(negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

})

.catch(erro => {

console.log(erro);

reject('Não foi possível obter as negociações da semana anterior');

});

}

obterNegociacoesDaSemanaRetrasada() {

this.\_http

.get('negociacoes/retrasada')

.then(negociacoes => {

console.log(negociacoes);

resolve(negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

})

.catch(erro => {

console.log(erro);

reject('Não foi possível obter as negociações da semana retrasada');

});

}

}

O próximo passo é usarmos a instrução return na frente da chamada de this.\_http.get. Como o método retorna uma promise, o que estamos fazendo é retornar esta promise nos métodos de NegociacaoService:

class NegociacaoService {

constructor() {

this.\_http = new HttpService();

}

obterNegociacoesDaSemana() {

return this.\_http

.get('negociacoes/semana')

.then(negociacoes => {

console.log(negociacoes);

resolve(negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

})

.catch(erro => {

console.log(erro);

reject('Não foi possível obter as negociações da semana');

});

}

obterNegociacoesDaSemanaAnterior() {

return this.\_http

.get('negociacoes/anterior')

.then(negociacoes => {

console.log(negociacoes);

resolve(negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

})

.catch(erro => {

console.log(erro);

reject('Não foi possível obter as negociações da semana anterior');

});

}

obterNegociacoesDaSemanaRetrasada() {

return this.\_http

.get('negociacoes/retrasada')

.then(negociacoes => {

console.log(negociacoes);

resolve(negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor)));

})

.catch(erro => {

console.log(erro);

reject('Não foi possível obter as negociações da semana retrasada');

});

}

}

Como não estamos mais retornando return new Promise((resolve, reject) => {}), não temos mais as funções resolve e reject para passarmos o resultado e o erro, caso exista. E agora?

A ideia é a seguinte, se uma função then possui um retorno, este retorno é acessível para quem encadear uma nova chamada à função then. Sendo assim, onde há resolve trocaremos por um return. Mas cuidado, não esqueça de remover também os () do resolve!

class NegociacaoService {

constructor() {

this.\_http = new HttpService();

}

obterNegociacoesDaSemana() {

return this.\_http

.get('negociacoes/semana')

.then(negociacoes => {

console.log(negociacoes);

return negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor));

})

.catch(erro => {

console.log(erro);

reject('Não foi possível obter as negociações da semana');

});

}

obterNegociacoesDaSemanaAnterior() {

return this.\_http

.get('negociacoes/anterior')

.then(negociacoes => {

console.log(negociacoes);

return negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor));

})

.catch(erro => {

console.log(erro);

reject('Não foi possível obter as negociações da semana anterior');

});

}

obterNegociacoesDaSemanaRetrasada() {

return this.\_http

.get('negociacoes/retrasada')

.then(negociacoes => {

console.log(negociacoes);

return negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor));

})

.catch(erro => {

console.log(erro);

reject('Não foi possível obter as negociações da semana retrasada');

});

}

}

Do jeito que está, podemos fazer algo como:

let service = new NegociacaoService();

service

.obterNegociacoesDaSemana()

.then(negociacoes => /\* faz alguma coisa com as negociações \*/)

Veja que é exatamente a maneira que já utilizávamos antes. O que mudou foi apenas a criação de uma promise extra na definição dos métodos.

Mas ainda não acabou! E se um erro acontecer? No lugar de usarmos reject, lançamos uma exceção em seu lugar:

class NegociacaoService {

constructor() {

this.\_http = new HttpService();

}

obterNegociacoesDaSemana() {

return this.\_http

.get('negociacoes/semana')

.then(negociacoes => {

console.log(negociacoes);

return negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor));

})

.catch(erro => {

console.log(erro);

throw new Error('Não foi possível obter as negociações da semana');

});

}

obterNegociacoesDaSemanaAnterior() {

return this.\_http

.get('negociacoes/anterior')

.then(negociacoes => {

console.log(negociacoes);

return negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor));

})

.catch(erro => {

console.log(erro);

throw new Error('Não foi possível obter as negociações da semana anterior');

});

}

obterNegociacoesDaSemanaRetrasada() {

return this.\_http

.get('negociacoes/retrasada')

.then(negociacoes => {

console.log(negociacoes);

return negociacoes.map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor));

})

.catch(erro => {

console.log(erro);

throw new Error('Não foi possível obter as negociações da semana retrasada');

});

}

}

Veja que, com essa alteração, poupamos algumas linhas de código e tornamos o código da classe NegociacaoService mais legível. É claro, isso só funciona porque HttpService devolve uma promise. Se não devolvesse, NegociacaoService precisaria retornar uma promise, como havíamos feito.

Criamos 3 métodos na nossa classe NegociacaoService para obter as negociações da semana:

* obterNegociacoesDaSemana
* obterNegociacoesDaSemanaAnterior
* obterNegociacoesDaSemanaRetrasada

Perfeito, mas em nosso controller foi necessário chamar esses três métodos e ainda usar Promise.all para resolver todos em ordem, aliás, ainda tivemos que achatar o array retornado pelo Promise.all, porque ele possuia como valor de três arrays, cada um contendo as negociações da semana, anterior e retrasada respectivamente.

Que tal criarmos em NegociacaoService o método obterNegociacoes? Se fizermos isso, nosso controller ficará mais simples. Crie o método! Quando você clicar em **Ver opinião do instrutor** você verá o gabarito do instrutor!

Segue a implementação do método em NegociacaoService:

class NegociacaoService {

// código anterior omitido

obterNegociacoes() {

return Promise.all([

this.obterNegociacoesDaSemana(),

this.obterNegociacoesDaSemanaAnterior(),

this.obterNegociacoesDaSemanaRetrasada()

]).then(periodos => {

let negociacoes = periodos

.reduce((dados, periodo) => dados.concat(periodo), []);

return negociacoes;

}).catch(erro => {

throw new Error(erro);

});

}

}

Veja que no o método obterNegociacoes retorna uma Promise, mas claro, aquela que combina o resultado de outras. Veja também que no método then da nossa Promisse estamos retornando a lista de negociacoes. É este retorno que estará disponível para quem usar nossa Promise. Além disso, se um erro ocorrer, lançamos um Error. É a mensagem passada para o Error que estará disponível no catch de quem usar nossa Promise. Resumindo a história: tanto uma Promise quanto o retorno de then são "Thenables".

E agora, alterando NegociacaoController para usar o novo método:

class NegociacaoController {

// constructor e outros métodos omitidos

importaNegociacoes() {

let service = new NegociacaoService();

service

.obterNegociacoes()

.then(negociacoes => {

negociacoes.forEach(negociacao => this.\_listaNegociacoes.adiciona(negociacao));

this.\_mensagem.texto = 'Negociações do período importadas com sucesso';

})

.catch(error => this.\_mensagem.texto = error);

}

// código posterior omitido

}

## Melhorando a experiência do usuário

Nossa aplicação é capaz de exibir uma lista de negociações em uma tabela, que pode ser alimentada pelo usuário ou importada de serviços na web. Para deixarmos a aplicação ainda melhor, que tal permitir que o usuário ordene a tabela, clicando em cada coluna?

Por exemplo, se o usuário clicar na coluna "**QUANTIDADE**", ordenaremos pela quantidade, se ele clicar na coluna "**DATA**", ordenaremos pela data. Além disso, se ele clicar mais de uma vez na mesma coluna, ele ordenará a tabela ascendentemente ou descendentemente. Nesse exercício, mostrarei uma "receita" para resolver este problema.

A primeira coisa que faremos é criar um método em nosso NegociacaoController, que será o responsável em ordenar a lista de negociações de ListaNegociacoes. A ordenação da lista envolve sua alteração, logo, a view NegociacoesView precisará ser atualizada. Ainda bem que implementamos um mecanismo caseiro de data binding (associação de dados) entre o model e view, no qual a alteração no modelo automaticamente renderiza a view ao qual foi associado.

Altere aluraframe/client/js/app/controllers/NegociacaoController.js e adicione o método ordena:

// aluraframe/client/js/app/controllers/NegociacaoController.js

// código anterior omitido

ordena(coluna) {

// ainda vamos implementar o método!

}

Veja que o método ordena recebe como parâmetro a coluna que queremos ordenar. Lembre-se que ordenaremos nosso modelo pela coluna que o usuário clicar, sendo assim, nada mais justo do que associar o método ordena ao evento click do cabeçalho de cada coluna em NegociacoesView:

Alterando aluraframe/client/js/app/views/NegociacoesView.js:

<!-- aluraframe/client/js/app/views/NegociacoesView.js -->

<!-- código anterior omitido -->

<thead>

<tr>

<th onclick="negociacaoController.ordena('data')">DATA</th>

<th onclick="negociacaoController.ordena('quantidade')">QUANTIDADE</th>

<th onclick="negociacaoController.ordena('valor')">VALOR</th>

<th onclick="negociacaoController.ordena('volume')">VOLUME</th>

</tr>

</thead>

<!-- código posterior omitido -->

Quando associamos a chamada do método ao evento, na chamada do método passamos como parâmetro a coluna que desejamos ordenar. É importante que cada parâmetro passado exista como uma propriedade em nosso modelo Negociacao.

Você deve estar pensando "Mas Flávio, queremos é ordenar a lista de negociações que ListaNegociacoes guarda". Sim, mas cada negociação da lista é uma instância da classe Negociacao.

Primeiramente, vamos implementar a solução de ordenação sem nos preocupar em alternar ascendentemente ou descendentemente, resolveremos isso depois. Além disso, antes de partir para a solução, que tal entender como é feito o processo de ordenação de uma lista a partir de algum critério da lista.

## Entendendo primeiro: Array.sort

Um Array em Javascript possui o método sort. Este método recebe uma estratégia de ordenação, ou seja, essa estratégia deve ser passada pelo desenvolvedor, mas deve seguir algumas regras. Vejamos um exemplo com escopo menor:

let lista = [10,1, 5, 9, 8, 12, 15];

Queremos ordenar essa lista em ordem crescente:

let lista = [10,1, 5, 9, 8, 12, 15];

lista.sort();

console.log(lista); // exibe a lista na ordem crescente

E se quisermos em ordem decrescente? Ordenamos primeiro de maneira ascendente e depois invertemos a ordem do array com reverse:

let lista = [10,1, 5, 9, 8, 12, 15];

lista.sort();

lista.reverse();

console.log(lista); // exibe a lista ordenada em ordem decrescente

Na verdade, mesmo a ordenação numérica tem problemas no JavaScript. Faça o teste, o resultado é um pouco inesperado. Contudo temos uma [explicação detalhada sobre o ordenamento numérico do JavaScript em nosso blog](http://blog.alura.com.br/ordenacao-de-numeros-no-javascript-nao-funciona/).

Podemos até mesmo ordenar uma lista de strings, que o procedimento é o mesmo. A ordenação funcionou porque o padrão do sort é classificar os elementos em ordem crescente na ordem da tabela ASCII. Vamos para um exemplo mais complexo?

Agora temos uma lista de negociações:

let negociacoes = [

new Negociacao(new Date(), 7, 200),

new Negociacao(new Date(), 1, 300),

new Negociacao(new Date(), 8, 100)

]

Queremos que a lista seja ordenada pela propriedade quantidade. O que será que vai acontecer se chamarmos lista.sort?

let negociacoes = [

new Negociacao(new Date(), 7, 200),

new Negociacao(new Date(), 1, 300),

new Negociacao(new Date(), 8, 100)

]

negociacoes.sort();

negociacoes.forEach(negociacao => console.log(negociacao));

Pois é, o método sort não fez curso de "Mãe Diná" para saber qual critério deve usar para ordenar nossa lista. Além disso, a lista continua do jeito que está. O método sort não consegue aplicar a estratégia de ordenar de maneira crescente porque um objeto da classe Negociacao não tem representação na tabela ASCII. E agora?

Quando temos uma lista de objetos que não sejam strings, números ou boolean (com este tipo, false vem primeiro e depois true), precisamos passar o critério de ordenação para o método sort:

let negociacoes = [

new Negociacao(new Date(), 7, 200),

new Negociacao(new Date(), 1, 300),

new Negociacao(new Date(), 8, 100)

]

negociacoes.sort((a, b) => a.quantidade - b.quantidade);

negociacoes.forEach(negociacao => console.log(negociacao));

A função passada para sort recebe dois parâmetros que representam pares de elementos, isso porque toda comparação envolve um par de elementos. A regra é a seguinte: com o critério selecionado, se o valor retornado for 0 não há alteração a ser feita, se o valor retornado for positivo, b deve vir antes de a, se o valor for negativo, a deve vir antes de b.

Que tal ordenar pela data?

let negociacoes = [

new Negociacao(new Date(), 7, 200),

new Negociacao(new Date(), 1, 300),

new Negociacao(new Date(), 8, 100)

]

negociacoes.sort((a, b) => b.data - a.data); // agora é b menos a!

negociacoes.forEach(negociacao => console.log(negociacao));

Não fique chocado, quando subtraímos uma data pela outra é retornado um número que pode ser zero, positivo ou negativo, atendendo a regra do sort. Faça um teste no console do Chrome e veja você mesmo:

new Date(2016,4,12) - new Date(2016,5,1) // negativo

new Date(2016,5,1) - new Date(2016,4,12) // positivo

new Date(2016,5,1) - new Date(2016,5,1) // 0

E se quisermos uma ordem decrescente? Só inverter a subtração:

let negociacoes = [

new Negociacao(new Date(), 7, 200),

new Negociacao(new Date(), 1, 300),

new Negociacao(new Date(), 8, 100)

]

negociacoes.sort((a, b) => b.quantidade - a.quantidade); // agora é b menos a!

negociacoes.forEach(negociacao => console.log(negociacao));

Agora que você já sabe definir um critério de ordenação para Array.sort, vamos voltar para o método ordena de NegociacaoController.

## Implementando nossa solução

Já sabemos como ordenar um Array segundo um critério, mas o problema é que ao acessarmos this.\_listaNegociacoes.negociacoes nós recebemos uma cópia da lista original e qualquer alteração na lista não afeta a instância de ListaNegociacoes (ainda lembra da programação defensiva?). Para resolvermos isso, vamos criar o método ordena em ListaNegociacoes. Este método receberá o critério de ordenação, que será passado para a lista de negociações encapsulada pela classe:

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

class ListaNegociacoes {

// código anterior omitido

// novo método!

ordena(criterio) {

this.\_negociacoes.sort(criterio);

}

}

Agora, vamos voltar para NegociacaoController e alterar seu método ordena e implementá-lo:

class NegociacaoController {

// código anterior omitido

ordena(coluna) {

this.\_listaNegociacoes.ordena((a, b) => a[coluna] - b[coluna]);

}

}

Veja que interessante. Não podemos fazer a.quantidade ou a.data, porque a propriedade usada no critério de ordenação é escolhida pelo usuário. Sendo assim, usamos a sintaxe objeto[nomePropriedade] para acessar a propriedade do objeto. Essa forma mais verbosa é interessantíssima quando queremos acessar as propriedades de um objeto dinamicamente .

Apesar de termos feitos essas mudanças, nada acontecerá. Precisamos atualizar a view quando o método ordena do nosso modelo for chamado, para isso, precisamos adicioná-lo na lista de métodos ou propriedades que desejamos monitorar do nosso modelo. Alterando NegociacaoController:

class NegociacaoController {

constructor() {

// propriedades omitidas

this.\_listaNegociacoes = new Bind(

new ListaNegociacoes(),

new NegociacoesView($('# negociacoesView')),

'adiciona', 'esvazia', 'ordena');

// outras propriedades omitidas

}

}

Perfeito, faça um teste agora. Alterne cliques em algumas colunas e veja o resultado. Gostou? Contudo, nossa solução está incompleta. Precisamos efetuar uma ordenação ascendente ou descendente quando o usuário clicar na mesma coluna. Como implementar isso?

## Há sempre uma solução

A lógica é seguinte. Se a ordenação atual é X e ele clicou em outra coluna, trocando a ordenação para Y, não fazemos nada e deixamos a lista ser ordenada por Y. No entanto, se a ordenação atual é X e ele clica na coluna que solicita novamente uma ordenação por X, invertemos a ordem atual.

**Vamos criar como propriedade de NegociacaoController a propriedade this.\_ordemAtual**.

class NegociacaoController {

constructor() {

this.\_ordemAtual = ''; // quando a página for carregada, não tem critério. Só passa a ter quando ele começa a clicar nas colunas

}

// código posterior omitido

}

class NegociacaoController {

// código anterior omitido

ordena(coluna) {

if(this.\_ordemAtual == coluna) {

// inverte a ordem da lista!

} else {

this.\_listaNegociacoes.ordena((a, b) => a[coluna] - b[coluna]);

}

this.\_ordemAtual = coluna;

}

}

E para invertermos a lista? Precisamos criar em nosso modelo ListaNegociacoes o método inverteOrdem, que chama this.\_negociacoes.reverse() para nós:

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

class ListaNegociacoes {

// código anterior omitido

inverteOrdem() {

this.\_negociacoes.reverse();

}

}

Agora, podemos terminar nossa controller com esta última alteração:

class NegociacaoController {

// código anterior omitido

ordena(coluna) {

if(this.\_ordemAtual == coluna) {

this.\_listaNegociacoes.inverteOrdem();

} else {

this.\_listaNegociacoes.ordena((a, b) => a[coluna] - b[coluna]);

}

this.\_ordemAtual = coluna;

}

}

E claro, não podemos nos esquecer de adicionar o método inverteOrdem como um dos métodos que estamos monitorando para atualizar automaticamente a View:

class NegociacaoController {

constructor() {

// propriedades omitidas

this.\_listaNegociacoes = new Bind(

new ListaNegociacoes(),

new NegociacoesView($('#negociacoesView'),

'adiciona', 'esvazia', 'ordena', 'inverteOrdem');

// outras propriedades omitidas

}

}

Perfeito! Experimente agora brincar com os critérios de ordenação clicando "igual a um louco" nos cabeçalhos das colunas da nossa tabela.

Durante o treinamento criamos a classe HttpService para centralizar toda infraestrutura necessária para realizarmos requisições Ajax, inclusive utilizamos o padrão Promise para tornar nosso código mais fácil de manter e legível. Contudo, nosso serviço possui apenas o método get responsável em ler os dados do servidor. Que tal encapsularmos nesse serviço toda complexidade para realizarmos requisições do tipo POST?

Vamos alterar aluraframe/client/js/app/services/HttpService.js e adicionar o método POST:

class HttpService {

get(url) {

return new Promise((resolve, reject) => {

let xhr = new XMLHttpRequest();

xhr.open('GET', url);

xhr.onreadystatechange = () => {

if (xhr.readyState == 4) {

if (xhr.status == 200) {

resolve(JSON.parse(xhr.responseText));

} else {

reject(xhr.responseText);

}

}

};

xhr.send();

});

}

post(url, dado) {

return new Promise((resolve, reject) => {

let xhr = new XMLHttpRequest();

xhr.open("POST", url, true);

xhr.setRequestHeader("Content-Type", "application/json");

});

}

}

Nosso novo método recebe uma URL e o dado que desejamos enviar. Criamos uma instância de XMLHttpRequest e usamos o verbo POST no já conhecido método open de XMLHttpRequest. Contudo, quando estamos enviando um dado para o servidor, precisamos dizer **qual tipo de conteúdo** estamos enviando. A ideia é enviarmos um dado no tipo (formato) JSON. É por isso que precisamos adicionar ao cabeçalho da requisição que será realizada a informação Content-Typecom o valor application/JSON.

Os próximos passos vocês já conhecem. Precisamos saber quando a requisição foi realizada e para isso usamos o onreadystatechange:

class HttpService {

get(url) {

// código anterior omitido

});

}

post(url, dado) {

return new Promise((resolve, reject) => {

let xhr = new XMLHttpRequest();

xhr.open("POST", url, true);

xhr.setRequestHeader("Content-type", "application/json");

xhr.onreadystatechange = () => {

if (xhr.readyState == 4) {

if (xhr.status == 200) {

resolve(JSON.parse(xhr.responseText));

} else {

reject(xhr.responseText);

}

}

};

// falta enviar!!!!!!

});

}

}

Veja que usamos JSON.parse para converter o JSON (String) retornado pelo servidor em um objeto JavaScript. Mas perceba que ainda falta a chamada do método xhr.send que recebe com parâmetro os dados que queremos enviar. Mas atenção: como HTTP é um protocolo textual, não podemos enviar um objeto JavaScript diretamente, precisamos convertê-lo para uma string no formato JSON. Para isso, usamos JSON.stringify:

class HttpService {

get(url) {

// código omitido

}

post(url, dado) {

return new Promise((resolve, reject) => {

let xhr = new XMLHttpRequest();

xhr.open("POST", url, true);

xhr.setRequestHeader("Content-type", "application/json");

xhr.onreadystatechange = () => {

if (xhr.readyState == 4) {

if (xhr.status == 200) {

resolve(JSON.parse(xhr.responseText));

} else {

reject(xhr.responseText);

}

}

};

xhr.send(JSON.stringify(dado)); // usando JSON.stringify para converter objeto em uma string no formato JSON.

});

}

}

Excelente, o método POST está pronto. Agora, se você fez os exercícios do capítulo anterior, deve lembrar que já fizemos um exercício para enviar dados com o método POST para nosso servidor e criamos a página post.html. Se você não lembra, é porque pulou exercícios e deixou de aprender várias coisas extras durante o treinamento. Enfim, partindo do pressuposto que você fez o exercício do post.html, vamos alterá-lo para fazer uso do nosso serviço e seu método post.

Primeiro, vamos ver o código antes da nossa modificação:

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

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

<html>

<head>

<meta charset="UTF-8">

<title>Negociações</title>

<link rel="stylesheet" href="css/bootstrap.css">

<link rel="stylesheet" href="css/bootstrap-theme.css">

</head>

<body class="container">

<form class="form">

<div class="form-group">

<label for="data">Data</label>

<input type="date" id="data" class="form-control" required autofocus/>

</div>

<div class="form-group">

<label for="quantidade">Quantidade</label>

<input type="number" min="1" step="1" id="quantidade" class="form-control" value="1" required/>

</div>

<div class="form-group">

<label for="valor">Valor</label>

<input id="valor" type="number" class="form-control" min="0.01" step="0.01" value="0.0" required />

</div>

<button class="btn btn-primary" type="submit" onclick="sendPost(event)">Enviar dados para servidor</button>

</form>

<script>

function sendPost(event) {

event.preventDefault();

console.log("Enviando post");

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

inputData = $('#data');

inputQuantidade = $('#quantidade');

inputValor = $('#valor');

let negociacao = {

data: inputData.value,

quantidade: inputQuantidade.value,

valor: inputValor.value

};

let xhr = new XMLHttpRequest();

xhr.open("POST", "/negociacoes", true);

xhr.setRequestHeader("Content-type", "application/json");

xhr.onreadystatechange = () => {

if (xhr.readyState == 4) {

if (xhr.status == 200) {

inputData.value = '';

inputQuantidade.value = 1;

inputValor.value = 0.0;

inputData.focus();

alert('Negociação enviada com sucesso');

} else {

alert(`Não foi possível enviar a negociação: ${xhr.responseText}`);

}

}

}

xhr.send(JSON.stringify(negociacao));

}

</script>

</body>

</html>

Agora, importando o script aluraframe/client/js/app/services/HttpService.js:

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

<html>

<head>

<meta charset="UTF-8">

<title>Negociações</title>

<link rel="stylesheet" href="css/bootstrap.css">

<link rel="stylesheet" href="css/bootstrap-theme.css">

</head>

<body class="container">

<form class="form">

<div class="form-group">

<label for="data">Data</label>

<input type="date" id="data" class="form-control" required autofocus/>

</div>

<div class="form-group">

<label for="quantidade">Quantidade</label>

<input type="number" min="1" step="1" id="quantidade" class="form-control" value="1" required/>

</div>

<div class="form-group">

<label for="valor">Valor</label>

<input id="valor" type="number" class="form-control" min="0.01" step="0.01" value="0.0" required />

</div>

<button class="btn btn-primary" type="submit" onclick="sendPost(event)">Enviar dados para servidor</button>

</form>

<script src="js/app/services/HttpService.js"></script>

<script>

function sendPost(event) {

event.preventDefault();

console.log("Enviando post");

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

inputData = $('#data');

inputQuantidade = $('#quantidade');

inputValor = $('#valor');

let negociacao = {

data: inputData.value,

quantidade: inputQuantidade.value,

valor: inputValor.value

};

// usando nosso serviço. Veja que nem guardei em uma variável

new HttpService()

.post('/negociacoes', negociacao)

.then(() => {

inputData.value = '';

inputQuantidade.value = 1;

inputValor.value = 0.0;

inputData.focus();

alert('Negociação enviada com sucesso');

})

.catch(erro => alert(`Não foi possível enviar a negociação: ${erro}`));

}

</script>

</body>

</html>

Veja que agora não precisamos lidar com detalhes de XMLHttpRequest quando formos realizar requisições do tipo POST!

Desde o início do treinamento solicitei que vocês usassem o Google Chrome por ele suportar vários recursos do ES6. Contudo, o Firefox não fica atrás. Então, porque eu deixei esse navegador de fora? Simplesmente por eles não suportarem o input do tipo Date. Só por isso, Flávio? Sim!

Se eu tivesse começado o treinamento sem o auxílio do input do tipo Date muitos alunos focariam na validação do campo e perderiam o foco do treinamento. Contudo, com uma pequena alteração podemos fazer com que nosso código funcione no Firefox.

**ATENÇÃO**: o código foi testado no Firefox 45.0.2! Quer saber se versões mais antigas suportam os recursos que utilizamos? Acesse o módulo I deste curso, e veja a [dica no exercício obrigatório do capítulo](https://cursos.alura.com.br/course/javascript-es6-orientacao-a-objetos-parte-1/task/16495).

## Alterando a interface

O primeiro passo é deixarmos de usar o input do tipo date. Vamos usar um input padrão, do tipo texto e usar um placeholder para solicitar que o usuário digite seu código no formato dd/mm/aaaa:

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

<div class="form-group">

<label for="data">Data</label>

<!-- alterou para input text e ainda esta com um placeholder -->

<input type="text" placeholder="dd/mm/aaaa" id="data" class="form-control" required autofocus/>

</div>

## Alterando nosso helper de data

Precisamos alterar a classe DateHelper. Hoje ela espera receber um string no formato aaaa-mm-dd. Esse formato só era assim porque usávamos o input date, agora, vamos fazer com que o DateHelper funcione da seguinte maneira:

1 - validar uma data no formato dd/mm/yyyy

2 - extraia do formato dd/mm/yyyy um array com o ano, mes e dia.

Alterando a classe DateHelper:

// aluraframe/client/js/app/helpers/DateHelper.js

class DateHelper {

// código anterior omitido

static textoParaData(texto) {

// mudamos a validação para aceitar o novo formato!

if(!/\d{2}\/\d{2}\/\d{4}/.test(texto))

throw new Error('Deve estar no formato dd/mm/aaaa');

// veja que usamos no split '/' no lugar de '-'. Usamos `reverse` também para ficar ano/mes/dia.

return new Date(...texto.split('/').reverse().map((item, indice) => item - indice % 2));

}

}

Quando alteramos nossa expressão regular, trocamos - por /, contudo, como esse é um caractere especial, precisamos usar \/. O nosso processo de desmembrar a string continua o mesmo, mas como temos uma data no formato dd/mm/aaaa, precisamos realizar um split usando / como separador e aplicar um .reverse()! A inversão dos itens do array é importante, porque a função map espera encontrar um array com ano, mês e dia e não dia, mês e ano.

Realize um teste ainda no Chrome. Digite por enquanto apenas datas válidas. Assim que você verificar que esta funcionando, digite seu nome no campo que captura a data e clique em incluir. O que acontecerá? Nada! Abrindo o console vemos a mensagem:

DateHelper.js:16 Uncaught Error: Deve estar no formato dd/mm/aaaa

## Existe try e catch em JavaScript?

O usuário nem fica sabendo que sua data é inválida! Que tal exibirmos essa mensagem para o usuário para que ele saiba o que esta acontecendo? Lembre-se que no método DateHelper.textoParaData lançamos um erro com a instrução throw. Essa instrução indica que houve um erro e que o método onde ele ocorreu não vai tratá-lo, mas sim lançá-lo para quem chamou o método. Sendo assim, quem chamou DateHelper.textoParaData? O método \_criaNegociacao de NegociacaoController.

Quando temos uma área que pode resultar em um erro, envolvemos essa área com a instrução try. É como se aquela área fosse um campo minado e que em algum momento pode ocorrer um erro e precisamos estar preparados para lidar com ele. No bloco try quando ocorre um lançamento com throw, podemos capturar o erro lançado no bloco catch. Vejamos:

class NegociacaoController {

// código anterior omitido

adiciona(event) {

event.preventDefault();

try {

this.\_listaNegociacoes.adiciona(this.\_criaNegociacao());

this.\_mensagem.texto = 'Negociação adicionada com sucesso';

this.\_limpaFormulario();

} catch(erro) {

this.\_mensagem.texto = erro;

}

}

Veja que a chamada de this.\_criaNegociacao() está dentro do bloco try, mas não ela apenas, mas o código que exibe a mensagem de sucesso e o de limpeza do formulário. A ideia é a seguinte: quando um erro é lançado, como this\_criaNegociacao() não o trata, o erro sobe na pilha. Daí, o interpretador JavaScript perguntará se quem chamou this.\_criaNegociacao está preparado para tratá-lo. E sim, está!

Se um erro acontecer, nosso código será direcionado para a cláusula catch, e nela temos acesso ao erro lançado pela instrução throws lá do nosso DateHelper. Resumindo: se um erro acontecer, não exibiremos a mensagem de sucesso e não limparemos o formulário e exibiremos a mensagem de erro para o usuário. Se não houver erro, as três instruções bloco try serão executadas, ou seja, a negociação será adicionada, a mensagem de sucesso exibida e o formulário limpo.

Teste mais uma vez no Chrome. Funcionando? Agora abra o Firefox e verifique que tudo funciona.

O Safari, até sua versão 9 suporta muito pouco o ES6 (nem com reza forte vai funcionar nosso código sem o uso de transpiler!). Contudo uma nova versão está no forno e essa promete ter o maior suporte de todos os navegadores. Eu realizei um teste com o Safari Tecnology Preview (versão teste da nova versão que sairá) e o código funcionou perfeitamente, mas há um detalhe interessante que vale a pena colocar aqui até para engrandecer a discussão.

OBS.: você precisa ter feito o fix do Firefox (ele não suporta o input date) e ainda estar usando a versão teste do próximo navegador da Apple. Tudo bem?

Durante o treinamento, aprendemos a usar let ao invés de var justamente por let criar escopo de bloco. Contudo, vejamos a tag script que inicializa nossa controller:

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

<!-- código anterior omitido -->

<script>

let negociacaoController = new NegociacaoController();

</script>

<!-- código posterior omitido -->

No Chrome e no Firefox não há problema, mas na tecnologia Preview do Safari, apesar de suportar a sintaxe let apenas aqui o código não funciona. o Safari está errado? Vamos dizer que dependendo do ponto de vista não.

A sintaxe let criado foi para permitir um escopo de bloco de variáveis. Veja que o código anterior que declara nosso controller no escopo global não está dentro de nenhum bloco! O Safari entende que é uma declaração inválida, pois let deve ser usada apenas dentro de blocos. Para mim, faz sentido, apesar de o Chrome e o Firefox não serem tão chatos nesse ponto.

Sendo assim, para que nosso código funcione no Safari precisamos alterar simplesmente, apenas nesse ponto a declaração da variável para var:

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

<!-- código anterior omitido -->

<script>

var negociacaoController = new NegociacaoController();

</script>

<!-- código posterior omitido -->

Então, podemos usar a seguinte regra: usamos var quando queremos que a variável tenha escopo global ou de função, e usarmos let quando queremos que a variável tenha sempre escopo de bloco. Em nosso caso, como nosso controller está no escopo global, faz todo sentido usarmos var.

Até sua versão 13, o Microsoft Edge não possui o método includes de Array. Veja que isso nos causará problemas. Vejamos o código do nosso ProxyFactory:

class ProxyFactory {

static create(objeto, props, acao) {

return new Proxy(objeto, {

get(target, prop, receiver) {

// uso includes aqui!

if(props.includes(prop) && ProxyFactory.\_ehFuncao(target[prop])) {

return function() {

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

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

acao(target);

return retorno;

}

}

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

},

set(target, prop, value, receiver) {

let retorno = Reflect.set(target, prop, value, receiver);

// uso includes aqui!

if(props.includes(prop)) acao(target);

return retorno;

}

});

}

static \_ehFuncao(func) {

return typeof(func) == typeof(Function);

}

E agora? Bom, para resolver o problema do includes podemos criar um **polyfill**extreme go horse que será suficiente para resolver o problema. Mas o que é um polyfill?

## Criando um polyfill

Um polyfill é um script que emula o comportamento de um recurso quando esse não é suportado para garantir que nosso código funcione sem termos que abdicar do que é mais novo.

Crie o arquivo aluraframe/client/js/app/polyfill/es6.js. Nele, vamos adicionar no prototype de Array o método includes que usa por debaixo dos panos o já conhecido indexOf. Mas, Flávio, é assim que o includes oficial é implementado? Não faço ideia, o importante é que o resultado final seja o mesmo, e usar o indexOfpor debaixo dos panos resolve isso perfeitamente. Veja que o método só é adicionando se ele não existir:

// aluraframe/client/js/app/polyfill/es6.js

if(!Array.prototype.includes) {

// Se não existir, adiciona

console.log('Polyfill para Array.includes aplicado.');

Array.prototype.includes = function(elemento) {

return this.indexOf(elemento) != -1;

};

}

Quando adicionamos métodos no prototype de uma classe ou função construtora, todas as instâncias dessa função construtora ou classe terão o método.

Agora vamos importar esse script no head da nossa página. Isso é necessário porque ele deve alterar Array antes que ele seja usado pela nossa aplicação. Alterando aluraframe/client/index.html:

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

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Negociações</title>

<link rel="stylesheet" href="css/bootstrap.css">

<link rel="stylesheet" href="css/bootstrap-theme.css">

<!-- carregando nosso polyfill -->

<script src="js/app/polyfill/es6.js"></script>

</head>

Excelente, mas isso não é suficiente. Sabe por quê? Porque o Edge 13 não suporta parâmetros opcionais, apenas o Edge 14 suportará. E agora?

## Edge 13 não suporta parâmetros opcionais do ES6

Aqui não há como fazer um polyfill como o que fizemos sem termos que escrever muito código hack. Aqui eu deixo ao aluno decidir ou não se abdica do parâmetro opcional usando uma estratégia antiga de JavaScript. Que tal revisitarmos esse antigo truque?

Hoje nossa classe Mensagem é assim:

class Mensagem {

// o valor padrão quando o parâmetro não é chamado é ''

constructor(texto='') {

this.\_texto = texto;

}

get texto() {

return this.\_texto;

}

set texto(texto) {

this.\_texto = texto;

}

}

Precisamos alterá-la dessa forma para funcionar no Edge:

class Mensagem {

// DEIXOU DE RECEBER O PARÂMTRO OPCIONAL

constructor(texto) {

this.\_texto = texto || ''; // se texto for undefined, vai passar ''

}

get texto() {

return this.\_texto;

}

set texto(texto) {

this.\_texto = texto;

}

}

Pronto. Com essas alterações (e mais a solução que usamos para o Firefox do campo date) nosso código funcionará no Microsot Edge.

Todos os hacks que fizemos até agora é para você ficar sabendo ainda mais de JavaScript. Contudo, há uma solução mais genérica e poderosa.

É possível que todo o código escrito em ES6 funcione em navegadores que nem suportem a nova versão do JavaScript. Para isso é necessário o uso de um transpiler. Aliás, um ótimo assunto a ser abordado no terceiro módulo deste treinamento. Não é mesmo? ;)

Vale salientar que os grandes vilões de compatibilidade do código JavaScript são os browsers mobile. A solução com transpiler é uma solução que abarca também esses dispositivos.

Em um dos exercícios citei que uma solução definitiva para resolver problemas de compatibilidade do ES6 com todos os navegadores do mercado é usar um transpiler. Mas o que é isso?

Um transpiler é um compilador de código fonte para código fonte. Ou seja, em linguagens como C, compilamos o código fonte para um código binário para que esse rode em nosso sistema operacional. Já um transpiler poderia converter o código escrito em C para JavaScript, ou seja, é um compilador que traduz o código fonte de uma linguagem para outra.

No caso do JavaScript, o truque é compilar o código fonte (nossos arquivos JS) para JavaScript. Mas espera aí Flávio, não faz sentido algum converter código JavaScript para código JavaScript. Faz sim, porque no caso estaríamos convertendo um código em ES6 para ES5! Nesse contexto, o programador organiza seu código em ES6 para ajudar na manutenção e legibilidade, mas na hora que esse código for entrar em produção ele é todo convertido para ES5, ou seja, toda feature do ES6 é implementada (muitas vezes verbosamente) usando ES5! Com isso, qualquer navegador do mercado será capaz de compreender nosso código.

Vejamos um exemplo de código em ES6:

class Pessoa {

constructor(nome, sobrenome = '') {

this.nome = nome;

this.sobrenome = sobrenome;

}

obterNomeCompleto() {

return `${this.nome} ${this.sobrenome}`;

}

static imprimePessoas(pessoas) {

pessoas.forEach((pessoa) => console.log(pessoa.nome));

}

}

O mesmo código, convertido para ES5 através de um transpiler ficaria assim:

function Pessoa(nome, sobrenome) {

this.nome = nome;

this.sobrenome = sobrenome || ''

}

Pessoa.prototype.obterNomeCompleto = function() {

return this.nome + ' ' + this.sobrenome

}

Pessoa.imprimePessoa(pessoas) {

pessoas.forEach(function(pessoa) {

console.log(pessoa.nome);

});

}

O mais importante nessa abordagem, é nunca editarmos os arquivos em ES5, editamos sempre os que estão em ES6 que a cada modificação devem ser convertidos para ES5.

Tudo (arrow functions, proxy, classes...) o que aprendemos de ES6 pode ser aplicado na plataforma [Node.js](https://nodejs.org/en/), mas tem um porém: **você DEVE usar a versão 6 ou superior do Node.js**. Versões anteriores não suportam totalmente os recursos que aprendemos aqui.