

Cangaceiro JavaScript

Uma aventura no sertão da programação



ISBN

Impresso e PDF: 978-85-94188-00-7

EPUB: 978-85-94188-01-4

MOBI: 978-85-94188-02-1

Você pode discutir sobre este livro no Fórum da Casa do Código: <http://forum.casadocodigo.com.br/>.

Caso você deseje submeter alguma errata ou sugestão, acesse <http://erratas.casadocodigo.com.br>.

AGRADECIMENTOS

Dedico este livro à minha primeira filha, pois, sem ela, ele provavelmente não teria existido. A motivação de deixar materializado nesta obra boa parte do conhecimento que tenho veio da vontade de torná-lo facilmente acessível para ela, caso algum dia deseje trilhar o caminho do pai. Boa parte das explicações que vocês encontrarão aqui foram construídas enquanto vigiava seu sono, pensando em uma didática de pai para filha.

Por fim, o título "*Cangaceiro JavaScript*" não foi escolhido por acaso. Foi uma tentativa de homenagear parte da nossa cultura, seja na história fictícia escrita por mim que inicia cada parte do livro, como também nas citações de outras obras literárias.

PREFÁCIO POR MAURÍCIO LINHARES

O Sertão, para os forasteiros, parece uma terra de pouca vida, de vegetação rasteira e com poucos animais à vista, diferente das grandes florestas tropicais da Zona da Mata, cheias de barulho e vida. Mas o sertanejo não se engana, seja plantando na terra dura ou usando as folhas do juazeiro, ele transforma o ambiente difícil e tira o seu sustento desse lugar que parece impossível de se aproveitar.

O homem e a mulher do Sertão são frutos desse ambiente inóspito, e precisam lidar com a dificuldade de onde estão diariamente e se acostumaram a seguir em frente independente dos problemas que surgem. Mas não pense que são pessoas tristes e duras. Uma sanfona, um triângulo, uma rabeca e uma zabumba para inspirar a dança, junto de um tacho de canjica, queijo coalho e trouxinhas de pamonha enroladas na folha do milho e você vai ver festa pra dotô nenhum botar defeito.

Neste livro, você vai correr pelo Sertão do JavaScript, seguindo o cangaceiro Flávio Almeida (*@flaviohalmeida*) para aprender os detalhes e segredos desta linguagem, construindo uma aplicação de verdade de ponta a ponta. Com o seu editor de código favorito como sua confiável peixeira, falando Vixe Maria sempre que encontrar um bug e dançando aquele forró agarradinho quando o código vai pra produção, atravessar essa terra inóspita que é desenvolver software será uma grande aventura. Que Padim Ciço e Frei Damião alumeiem o seu caminho!

SOBRE O AUTOR



Figura 1: Flávio Almeida

Flávio Almeida é desenvolvedor e instrutor na Caelum. Também é instrutor na Alura, contribuindo com mais de 20 treinamentos para esta plataforma de ensino online. Autor do livro *Mean: Full stack JavaScript para aplicações web com MongoDB, Express, Angular e Node*, possui mais de 15 anos de experiência na área de desenvolvimento. Bacharel em Informática com MBA em Gestão de Negócios em TI, tem Psicologia como segunda graduação e procura aplicar o que aprendeu no desenvolvimento de software e na educação.

Atualmente, foca na plataforma Node.js e na linguagem JavaScript, tentando aproximar ainda mais o front-end do back-end. Já palestrou e realizou workshops em grandes conferências como QCON e MobileConf, e está sempre ávido por novos eventos.

INTRODUÇÃO

"Mestre não é quem sempre ensina, mas quem de repente aprende." - Grande Sertão: Veredas

Talvez nenhuma outra linguagem tenha conseguido invadir o coletivo imaginário dos desenvolvedores como JavaScript fez. Em sua história fabular em busca de identidade, foi a única que conseguiu se enraizar nos navegadores, e nos últimos anos passou a empoderar servidores de alta performance através da plataforma Node.js. Com tudo isso, tornou-se uma linguagem em que todo desenvolvedor precisa ter algum nível de conhecimento.

Com a complexidade cada vez maior dos problemas, soluções ninjas não são mais suficientes; é preciso soluções cangaceiras. É nesse contexto que este livro se encaixa, ajudando-o a transcender seu conhecimento, tornando-o um **Cangaceiro JavaScript**.

A quem se destina o livro

Este livro destina-se àqueles que desejam aprimorar a manutenção e legibilidade de seus códigos aplicando os paradigmas da Orientação a Objetos e da Programação Funcional, inclusive padrões de projetos. Por fim, ele ajudará também aqueles que desejam trabalhar com frameworks Single Page Application (SPA) conceituados do mercado, mas que carecem de um conhecimento mais aprofundado da linguagem JavaScript.

Procurei utilizar uma linguagem simples para tornar este livro acessível a um amplo espectro de desenvolvedores. No entanto, é necessário que o leitor tenha algum conhecimento, mesmo que básico, da linguagem JavaScript para que tenha um melhor aproveitamento.

Visão geral da nossa jornada

O projeto deste livro será um cadastro de negociações de bolsa de valores. Ele será construído através do paradigma funcional e orientado a objetos ao mesmo tempo, utilizando o melhor dos dois mundos.



A interface da aplicação, intitulada "Negociações", apresenta um formulário para cadastrar uma nova negociação. O formulário contém três campos de entrada: "Data" com o placeholder "dd/mm/aaaa", "Quantidade" com o placeholder "1", e "Valor" com o placeholder "0,0". Abaixo dos campos, há um botão azul "Incluir". À direita, há dois botões azuis: "Obter negociações" e "Apagar". Na base da interface, há uma barra de cabeçalho com quatro colunas: "DATA", "QUANTIDADE", "VALOR" e "VOLUME".

Figura 1: Preview da aplicação

Em um primeiro momento, o escopo da nossa aplicação pode parecer bem reduzido, mas é o suficiente para lançarmos mão de uma série de recursos acumulados na linguagem até a sua versão **ECMAScript 2017 (ES8)**.

Para que o leitor prepare suas expectativas sobre o que está por vir, segue um breve resumo de cada capítulo:

Capítulo 01: escreveremos um código rapidamente sem nos preocuparmos com boas práticas ou sua reutilização, sentindo na pele as consequências dessa abordagem. É a partir dele que toda a trama do livro dará início, inclusive a motivação para uma estrutura aplicando o modelo MVC.

Capítulo 02: materializar uma representação de algo do mundo real em código é uma das tarefas do desenvolvedor. Veremos como o paradigma orientado a objetos pode nos ajudar na construção de um modelo.

Capítulo 03: do que adianta um modelo se o usuário não pode interagir com ele? Criaremos um *controller*, a ponte entre as ações do usuário e o modelo.

Capítulo 04: lidar com data não é uma tarefa trivial em qualquer linguagem, mas se combinarmos recursos da própria linguagem JavaScript pode ser algo bem divertido.

Capítulo 05: não é apenas uma negociação que possui regras, uma lista de negociação também. Criaremos mais um modelo com suas respectivas regras.

Capítulo 06: do que adianta termos um modelo que é modificado pelo usuário se ele não consegue ver o resultado? Aprenderemos a ligar o modelo com a view.

Capítulo 07: criar um modelo, modificá-lo com as ações do usuário e apresentá-lo é uma tarefa corriqueira. Será que podemos isolar esse processo e reutilizá-lo? Com certeza!

Capítulo 08: jogar a responsabilidade de atualizar a view no colo do desenvolvedor toda vez que o modelo mudar está sujeito a

erros. Aprenderemos a automatizar esse processo.

Capítulo 09: aplicaremos o padrão de projeto Proxy para implementarmos nossa própria solução de *data binding*, que consiste na atualização da view toda vez que o modelo sofrer alterações.

Capítulo 10: isolaremos a responsabilidade da criação de proxies em uma classe, aplicando o padrão de projeto Factory e permitindo que ela seja reutilizada pela nossa aplicação.

Capítulo 11: aprenderemos a lidar com exceções, inclusive a criá-las.

Capítulo 12: trabalharemos com o velho conhecido *XMLHttpRequest* para nos integrarmos com a API fornecida com o projeto.

Capítulo 13: combateremos o *callback hell*, resultado da programação assíncrona com callbacks através do padrão de projeto Promise. Isolaremos a complexidade de se trabalhar com o *XMLHttpRequest* em classes de serviços.

Capítulo 14: negociações cadastradas são perdidas toda vez que a página é recarregada ou quando o navegador é fechado. Utilizaremos o IndexedDB, um banco presente em todo navegador para persistir nossas negociações.

Capítulo 15: não basta termos uma conexão com o IndexedDB. Aplicaremos um conjunto de boas práticas para lidar com a conexão, facilitando assim a manutenção e legibilidade do código.

Capítulo 16: aplicaremos o padrão de projeto DAO para

esconder do desenvolvedor os detalhes de acesso ao banco e tornar ainda mais legível o código escrito.

Capítulo 17: o tendão de Aquiles da linguagem JavaScript sempre foi o escopo global e as dependências entre scripts, jogando a responsabilidade da ordem de carregamento no colo do desenvolvedor. Aprenderemos a usar o sistema de módulos padrão do próprio JavaScript para atacar esses problemas. Durante esse processo, aprenderemos a lidar com Babel, o transcompilador mais famoso do mundo open source.

Capítulo 18: veremos como o uso de promises com `async/await` torna nosso código assíncrono mais fácil de ler e de manter. Aplicaremos mais padrões de projetos.

Capítulo 19: aplicaremos o padrão de projeto Decorator com auxílio do Babel, inclusive aprenderemos a realizar requisições assíncronas através da API Fetch, simplificando ainda mais nosso código. Nos aprofundaremos na metaprogramação com `reflect-metadata`.

Capítulo 20: utilizaremos Webpack para agrupar nossos módulos e aplicar boas práticas em tempo de desenvolvimento e de produção. Inclusive aprenderemos a fazer o deploy da aplicação no GitHub Pages.

Ao final do livro, o leitor terá um arsenal de recursos da linguagem e padrões de projetos que vão ajudá-lo a resolver problemas do dia a dia. A criação de um miniframework é apenas para deixar a aprendizagem mais divertida.

Por fim, o leitor poderá acompanhar o autor pelo twitter [@flaviohalmeida](https://twitter.com/flaviohalmeida) e pelo blog (<http://cangaceirojavascript.com.br>).

Agora que já temos uma visão geral de cada capítulo, veremos o download do projeto e a infraestrutura mínima necessária.

Infraestrutura necessária

A infraestrutura necessária para o projeto construído até o capítulo 10 é apenas o **Google Chrome**. É importante que esse navegador seja usado, pois só atacaremos questões de compatibilidade a partir do capítulo 11. Ao final do livro, sintam-se à vontade para utilizar qualquer navegador.

Do capítulo 12 em diante, precisaremos do **Node.js** (<https://nodejs.org/en/>) instalado. Durante a criação deste projeto, foi utilizada a versão 8.1.3. Mas não há problema baixar versões mais novas, contanto que sejam **versões pares**, as chamadas versões LTS (*long term support*).

Download do projeto

Há duas formas de baixar o projeto base deste livro. A primeira é clonar o repositório do <https://github.com/flaviohenriquealmeida/jscangaceiro>.

Você também pode baixar o seu zip no endereço <https://github.com/flaviohenriquealmeida/jscangaceiro/archive/master.zip>. Escolha aquela que for mais cômoda para você.

No entanto, caso tenham optado pela versão zip, depois de descompactá-la, renomeie a pasta `jscangaceiro-master` para `jscangaceiro`.

O projeto possui a seguinte estrutura:

```
|— client
|   |— app
|   |— css
|   |— index.html
|— server
```

Existem duas pastas, `client` e `server`. A primeira contém todo o código que rodará no navegador, inclusive é dentro da pasta `client/app` que ficará todo o código que criaremos até o fim do livro. As demais pastas dentro de `client` possuem os arquivos `css` do Bootstrap.

O Bootstrap nada mais é do que uma biblioteca CSS que permite aplicar um visual profissional em nossa aplicação com pouco esforço, apenas com o uso de classes. Não é necessário que o leitor tenha conhecimento prévio dele.

Ainda na pasta `client`, temos o arquivo `index.html`, a única página que utilizaremos ao longo do projeto, nosso ponto de partida:

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

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width">
  <title>Negociações</title>
  <link rel="stylesheet" href="css/bootstrap.css">
  <link rel="stylesheet" href="css/bootstrap-theme.css">
</head>
<body class="container">

  <h1 class="text-center">Negociações</h1>
```

```

<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">Incluir</button>
</form>

<div class="text-center">
    <button id="botao-importa" class="btn btn-primary text-center" type="button">
        Importar Negociações
    </button>
    <button id="botao-apaga" class="btn btn-primary text-center" type="button">
        Apagar
    </button>
</div>
<br>
<br>

<table class="table table-hover table-bordered">

    <thead>
        <tr>
            <th>DATA</th>
            <th>QUANTIDADE</th>
            <th>VALOR</th>

```

```
        <th>VOLUME</th>
    </tr>
</thead>

<tbody>
</tbody>

<tfoot>
</tfoot>
</table>

</body>
</html>
```

Por fim, a pasta `server` só será usada a partir do capítulo 12. Ela disponibiliza um servidor web com as APIs que serão consumidas pela nossa aplicação. Inclusive este servidor é feito em Node.js, um dos motivos que tornam essa plataforma uma dependência de infraestrutura. As instruções de como levantar o servidor serão apresentadas assim que o seu uso for necessário.

VISUAL STUDIO CODE (OPCIONAL)

Durante a criação do projeto, foi utilizado o Visual Studio Code (<https://code.visualstudio.com/download>), um editor de texto gratuito e multiplataforma disponível para Windows, Linux e MAC. Sinta-se livre para escolher aquele editor que você preferir.

Dando início à nossa jornada

Com o projeto baixado e os requisitos de infraestrutura atendidos, podemos começar nossa jornada através desse sertão

que é o JavaScript. Mas assim como em qualquer trama com início, meio e fim, no próximo capítulo teremos um prólogo que motiva os demais capítulos.

Sumário

Parte 1 - O caminho do cangaceiro	1
1 Prólogo: era uma vez no sertão	2
1.1 O problema do nosso código	11
1.2 O padrão MVC (Model-View-Controller)	12
2 Negociar com o cangaceiro, tem coragem?	14
2.1 O papel de um modelo	14
2.2 A classe Negociação	15
2.3 Construtor de classe	20
2.4 Métodos de classe	23
2.5 Encapsulamento	25
2.6 A sintaxe get	30
2.7 Objetos imutáveis	34
2.8 A instância é imutável mesmo?	37
2.9 Programação defensiva	40
2.10 Menos verbosidade no constructor com Object.assign	43
2.11 Atalho para propriedades de objetos literais	46
2.12 As surpresas de declarações com var	48

2.13 Declaração de variáveis com let	51
2.14 Temporal Dead Zone	53
3 No cangaço, é ação para todo lado	58
3.1 O papel de um controlador	58
3.2 A classe NegociacaoController	59
3.3 Associando métodos do controller às ações do usuário	62
3.4 Evitando percorrer desnecessariamente o DOM	67
3.5 Criando uma instância de Negociação	70
3.6 Criando um objeto Date a partir da entrada do usuário	73
3.7 Um desafio com datas	76
3.8 Resolvendo um problema com o paradigma funcional	77
3.9 Arrow functions: deixando o código ainda menos verboso	85
4 Dois pesos, duas medidas?	89
4.1 Isolando a responsabilidade de conversão de datas	93
4.2 Métodos estáticos	97
4.3 Template literal	101
4.4 A boa prática do fail-fast	103
5 O bando deve seguir uma regra	107
5.1 Criando um novo modelo	107
5.2 O tendão de Aquiles do JavaScript	110
5.3 Blindando o nosso modelo	116
6 A moda no cangaço	121
6.1 O papel da View	121
6.2 Nossa solução de View	122
6.3 Construindo um template dinâmico	129

6.4 Totalizando o volume de negociações	135
6.5 Totalizando com reduce	137
7 O plano	140
7.1 Parâmetro default	143
7.2 Criando a classe MensagemView	146
7.3 Herança e reutilização de código	151
7.4 Classes abstratas?	156
7.5 Para saber mais: super	157
7.6 Adquirindo um novo hábito com const	159
 Parte 2 - Força Volante	 162
8 Um cangaceiro sabe delegar tarefas	163
8.1 E se atualizarmos a View quando o modelo for alterado?	167
8.2 Driblando o this dinâmico	173
8.3 Arrow function e seu escopo léxico	177
9 Enganaram o cangaceiro, será?	181
9.1 O padrão de projeto Proxy	182
9.2 Aprendendo a trabalhar com Proxy	183
9.3 Construindo armadilhas de leitura	187
9.4 Construindo armadilhas de escrita	193
9.5 Reflect API	196
9.6 Um problema não esperado	197
9.7 Construindo armadilhas para métodos	200
9.8 Uma pitada do ES2016 (ES7)	202
9.9 Aplicando a solução em NegociacaoController	206

10 Cúmplice na emboscada	210
10.1 O padrão de projeto Factory	210
10.2 Nosso proxy ainda não está 100%!	216
10.3 Associando modelo e View através da classe Bind	218
10.4 Parâmetros REST	224
11 Data dos infernos!	226
11.1 O problema com o input date	226
11.2 Ajustando nosso converter	227
11.3 Lidando com exceções	229
11.4 Criando nossa própria exceção: primeira tentativa	233
11.5 Criando nossa própria exceção: segunda tentativa	235
12 Pilhando o que interessa!	239
12.1 Servidor e infraestrutura	239
12.2 Requisições Ajax com o objeto XMLHttpRequest	241
12.3 Realizando o parse da resposta	248
12.4 Separando responsabilidades	255
13 Lutando até o fim	263
13.1 Callback HELL	264
13.2 O padrão de projeto Promise	265
13.3 Criando Promises	267
13.4 Criando um serviço para isolar a complexidade do XMLHttpRequest	270
13.5 Resolvendo Promises sequencialmente	275
13.6 Resolvendo Promises paralelamente	280
13.7 Ordenando o período	284

13.8 Impedindo importações duplicadas	289
13.9 As funções filter() e some()	293
Parte 3 - A revelação	297
14 A algibeira está furada!	298
14.1 IndexedDB, o banco de dados do navegador	298
14.2 A conexão com o banco	304
14.3 Nossa primeira store	306
14.4 Atualização do banco	309
14.5 Transações e persistência	311
14.6 Cursores	316
15 Colocando a casa em ordem	322
15.1 A classe ConnectionFactory	322
15.2 Criando Stores	326
15.3 Garantindo uma conexão apenas por aplicação	329
15.4 O padrão de projeto Module Pattern	332
15.5 Monkey Patch: grandes poderes trazem grandes responsabilidades	338
16 Entrando na linha	345
16.1 O padrão de projeto DAO	346
16.2 Criando nosso DAO de negociações	347
16.3 Implementando a lógica de inclusão	349
16.4 Implementando a lógica da listagem	352
16.5 Criando uma DAOFactory	353
16.6 Combinando padrões de projeto	355

16.7 Exibindo todas as negociações	358
16.8 Removendo todas as negociações	360
16.9 Factory function	362
17 Dividir para conquistar	366
17.1 Módulos do ES2015 (ES6)	366
17.2 Instalando o loader	367
17.3 Transformando scripts em módulos	370
17.4 O papel de um transcompilador	376
17.5 Babel, instalação e build-step	377
17.6 Sourcemap	379
17.7 Compilando arquivos em tempo real	380
17.8 Barrel, simplificando a importação de módulos	382
18 Indo além	385
18.1 ES2017 (ES8) e o açúcar sintático async/await	387
18.2 Para saber mais: generators	389
18.3 Refatorando o projeto com async/wait	393
18.4 Garantindo a compatibilidade com ES2015	397
18.5 Lidando melhor com exceções	398
18.6 Debounce pattern: controlando a ansiedade	401
18.7 Implementando o Debounce pattern	402
19 Chegando ao limite	407
19.1 O padrão de projeto Decorator	408
19.2 Suportando Decorator através do Babel	410
19.3 Um problema não esperado com nosso Decorator	415
19.4 Elaborando um DOM Injector	417

19.5 Decorator de classe	418
19.6 Simplificando requisições Ajax com a API Fetch	424
19.7 Configurando uma requisição com API Fetch	428
19.8 Validação com parâmetro default	432
19.9 Reflect-metadata: avançando na metaprogramação	435
19.10 Adicionando metadados com Decorator de método	437
19.11 Extraindo metadados com um Decorator de classe	440
20 Enfrentamento final	445
20.1 Webpack, agrupador de módulos	446
20.2 Preparando o terreno para o Webpack	446
20.3 O temível webpack.config.js	448
20.4 Babel-loader, a ponte entre o Webpack e o Babel	450
20.5 Preparando o build de produção	453
20.6 Mudando o ambiente com cross-env	456
20.7 Webpack Dev Server e configuração	457
20.8 Tratando arquivos CSS como módulos	461
20.9 Resolvendo o FOUC (Flash of Unstyled Content)	469
20.10 Resolvemos um problema e criamos outro, mas tem solução!	472
20.11 Importando scripts	473
20.12 Lidando com dependências globais	475
20.13 Otimizando o build com Scope Hoisting	479
20.14 Separando nosso código das bibliotecas	480
20.15 Gerando a página principal automaticamente	483
20.16 Simplificando ainda mais a importação de módulos	486
20.17 Code splitting e Lazy loading	487
20.18 System.import vs import	492

20.19 Quais são os arquivos de distribuição?	493
20.20 Deploy do projeto no GitHub Pages	494
20.21 Alterando o endereço da API no build de produção	496
20.22 Considerações finais	500

Parte 1 - O caminho do cangaceiro

A vida talhou em sua carne tudo o que ele sabia. Mas durante uma noite a céu aberto, enquanto fixava seu olhar na fogueira que o aquecia, foi tomado de súbito por uma visão. Nela, Lampião aparecia apontando para a direção oposta àquela em que se encontrava. Por mais fugaz que a visão tenha sido, ele teve a certeza de que ainda faltava um caminho a ser trilhado, o caminho do cangaceiro. E então, ele juntou suas coisas e tomou seu rumo. Mesmo sabendo que havia cangaceiros melhores e mais bem preparados do que ele, não esmoreceu e ainda apertou o passo com a certeza de que estava no caminho certo. Na manhã seguinte, vendo o sol nascer e olhando o horizonte, entendeu que tudo só dependia de sua vontade.

PRÓLOGO: ERA UMA VEZ NO SERTÃO

"Quem desconfia fica sábio." - Grande Sertão: Veredas

Um cangaceiro não pode existir sem uma trama que o contextualize. Com o nosso projeto, não será diferente. Este capítulo é o prólogo que prepara o terreno do que está por vir. A trama gira em torno de uma aplicação que permite o cadastro de negociações de bolsa de valores. Apesar de o domínio da aplicação ser reduzido, é suficiente para aprimorarmos nosso conhecimento da linguagem JavaScript até o final do livro.

Para a nossa comunidade, o projeto que baixamos na introdução possui a página `client/index.html`, com um formulário pronto e já estilizado com Bootstrap, com três campos de entrada para capturarem respectivamente a **data**, a **quantidade** e o **valor** de uma negociação. Vejamos um dos campos:

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

<form class="form">
```

```

    <div class="form-group">
      <label for="data">Data</label>
      <input type="date" id="data" class="form-control" requi
red autofocus/>
    </div>
<!-- código posterior omitido -->

```

Como cada campo possui um `id`, fica fácil trazermos uma referência desses elementos para que possamos manipulá-los em nosso código JavaScript. Vamos escrever todo o código deste capítulo no novo arquivo `client/app/index.js`. Sem pestanejar, vamos importá-lo logo de uma vez em `client/index.html`, antes do fechamento da tag `</body>`:

```

<!-- client/index.html -->
<!-- código anterior omitido -->

</table>

<script src="app/index.js"></script>
</body>
</html>

```

Dentro do arquivo `client/js/index.js`, vamos criar o array `campos` que armazenará uma referência para cada um dos elementos de entrada do formulário. A busca será feita através de `document.querySelector()`, uma API do DOM que nos permite buscar elementos através de seletores CSS:

```

// client/app/index.js

var campos = [
  document.querySelector('#data'),
  document.querySelector('#valor'),
  document.querySelector('#quantidade')
];

console.log(campos); // verificando o conteúdo do array

```

Vamos recarregar a página no navegador e ver a saída no Console do navegador:

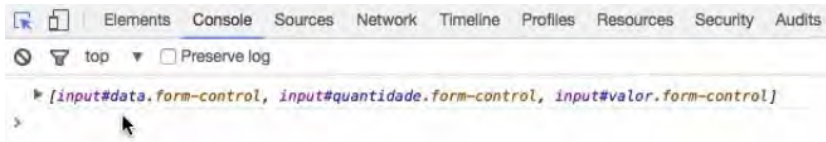


Figura 1.1: Formulário

Antes de mais nada, vamos buscar o elemento `<tbody>` da `<table>`, pois é nele que vamos inserir a `<tr>` que criaremos com os dados das negociações:

```
// client/app/index.js

var campos = [
  document.querySelector('#data'),
  document.querySelector('#valor'),
  document.querySelector('#quantidade')
];

// precisamos de tbody, pois ele receberá a tr que vamos construir

var tbody = document.querySelector('table tbody');
```

Agora que temos nosso array `campos` preenchido, precisamos lidar com a ação do usuário. Olhando na estrutura de `client/index.html`, encontramos o seguinte botão:

```
<button class="btn btn-primary" type="submit">Incluir</button>
```

Ele é do tipo `submit`; isso significa que, ao ser clicado, fará o formulário disparar o evento `submit` responsável pelo envio de seus dados. No entanto, não queremos enviar os dados para lugar algum, queremos apenas ler os valores inseridos pelo usuário em cada campo e construir uma nova `<tr>`. Aliás, como estamos

usando o atributo `required` em cada `input`, só conseguiremos disparar o evento `submit` caso todos sejam preenchidos. O navegador exibirá mensagens padrões de erro para cada campo não preenchido.

Agora, através de `document.querySelector()`, vamos buscar o formulário através da sua classe `.form`. De posse do elemento em nosso código, vamos associar a função ao evento `submit` através de `document.addEventListener`. É esta função que será chamada quando o usuário submeter o formulário:

```
// client/app/index.js

var campos = [
  document.querySelector('#data'),
  document.querySelector('#valor'),
  document.querySelector('#quantidade')
];

console.log(campos); // verificando o conteúdo do array

var tbody = document.querySelector('table tbody');

document.querySelector('.form').addEventListener('submit', function(event) {

  alert('oi');
});
```

Por enquanto, vamos exibir um simples pop-up através da função `alert` a cada submissão do formulário:

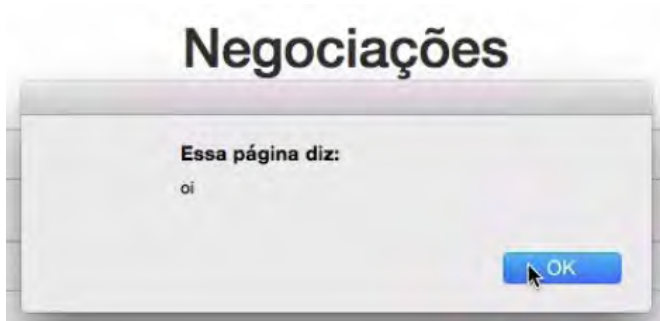


Figura 1.2: Formulário

Com a certeza de que nossa função está sendo chamada na submissão do formulário, já podemos dar início ao código que constrói dinamicamente uma nova `<tr>`, aquela que conterá os dados das negociações do formulário. Através de `document.createElement`, criamos um novo elemento:

```
// client/app/index.js

// código anterior omitido

var tbody = document.querySelector('table tbody');

document.querySelector('.form').addEventListener('submit', function(event) {

    // substituindo o alert pelo novo código
    var tr = document.createElement('tr');
});
```

Todo array possui a função `forEach` que nos permite iterar em cada um dos seus elementos. Para cada elemento iterado, vamos criar sua respectiva `td` que fará parte no final da `tr` que acabamos de criar:

```
// client/app/index.js
```

```
// código anterior omitido

var tbody = document.querySelector('table tbody');

document.querySelector('.form').addEventListener('submit', function(event) {

    var tr = document.createElement('tr');

    campos.forEach(function(campo) {

        // cria uma td sem informações
        var td = document.createElement('td');

        // atribui o valor do campo à td
        td.textContent = campo.value;

        // adiciona a td na tr
        tr.appendChild(td);

    });
});
```

O código anterior cria uma coluna com o valor de cada campo do nosso array `campos`, no entanto, ainda falta uma coluna para guardar volume. Toda negociação possui um volume, que nada mais é do que o valor da negociação multiplicado pela quantidade negociada no dia. Neste caso, a adição desta coluna precisará ficar fora do nosso `forEach`:

```
// client/app/index.js

// código anterior omitido

var tbody = document.querySelector('table tbody');

document.querySelector('.form').addEventListener('submit', function(event) {

    var tr = document.createElement('tr');

    campos.forEach(function(campo) {
```

```

    var td = document.createElement('td');
    td.textContent = campo.value;
    tr.appendChild(td);
  });

  // nova td que armazenará o volume da negociação
  var tdVolume = document.createElement('td');

  // as posições 1 e 2 do array armazenam os campos de quantidade
  e valor, respectivamente
  tdVolume.textContent = campos[1].value * campos[2].value;

  // adicionando a td que faltava à tr
  tr.appendChild(tdVolume);
});

```

Agora que já temos a `tr` completa, podemos adicioná-la em `tbody`. Nosso código até este ponto deve estar assim:

```

// client/app/index.js

var campos = [
  document.querySelector('#data'),
  document.querySelector('#valor'),
  document.querySelector('#quantidade')
];

console.log(campos); // verificando o conteúdo do array

var tbody = document.querySelector('table tbody');

document.querySelector('.form').addEventListener('submit', function(event) {

  var tr = document.createElement('tr');

  campos.forEach(function(campo) {

    var td = document.createElement('td');
    td.textContent = campo.value;
    tr.appendChild(td);
  });
});

```

```

var tdVolume = document.createElement('td');
tdVolume.textContent = campos[1].value * campos[2].value;
tr.appendChild(tdVolume);

// adicionando a tr
tbody.appendChild(tr);
});

```

Vamos realizar um teste cadastrando uma nova negociação. Para nossa surpresa, nada é exibido. Nossa `<tr>` é até inserida, mas como a submissão do formulário recarregou nossa página, ela volta para o estado inicial sem `<tr>`.

Lembre-se de que não queremos submeter o formulário, queremos apenas pegar os seus dados no evento `submit`. Sendo assim, basta cancelarmos o comportamento padrão do evento `submit` através da chamada de `event.preventDefault()`. Nosso código continuará a ser executado, mas o formulário não será submetido:

```

// client/app/index.js

var campos = [
  document.querySelector('#data'),
  document.querySelector('#valor'),
  document.querySelector('#quantidade')
];

console.log(campos); // verificando o conteúdo do array

var tbody = document.querySelector('table tbody');

document.querySelector('.form').addEventListener('submit', function(event) {

  // cancelando a submissão do formulário
  event.preventDefault();

  // código posterior omitido
});

```




Figura 1.3: Página não recarregada

Excelente, nossa negociação aparece na lista, no entanto, é uma boa prática limpar os campos do formulário, preparando-o para o próximo cadastro. Além disso, faremos com que o campo da data receba o foco através da chamada da função `focus`, permitindo que o usuário já comece o novo cadastro, sem ter de clicar no primeiro campo. Nosso código final fica assim:

```
// client/app/index.js

var campos = [
  document.querySelector('#data'),
  document.querySelector('#valor'),
  document.querySelector('#quantidade')
];

console.log(campos); // verificando o conteúdo do array

var tbody = document.querySelector('table tbody');

document.querySelector('.form').addEventListener('submit', function(event) {

  event.preventDefault();

  var tr = document.createElement('tr');
```

```
campos.forEach(function(campo) {

    var td = document.createElement('td');
    td.textContent = campo.value;
    tr.appendChild(td);

});

var tdVolume = document.createElement('td');
tdVolume.textContent = campos[1].value * campos[2].value;
tr.appendChild(tdVolume);

tbody.appendChild(tr);

// limpa o campo da data
campos[0].value = '';
// limpa o campo da quantidade
campos[1].value = 1;
// limpa o campo do valor
campos[2].value = 0;
// foca no campo da data
campos[0].focus();
});
```

Apesar de funcionar, nosso código deixa a desejar e entenderemos o motivo a seguir.

1.1 O PROBLEMA DO NOSSO CÓDIGO

Nenhuma aplicação nasce do nada, mas sim a partir de especificações criadas das necessidades do cliente. No caso da nossa aplicação, ainda é necessário totalizarmos o volume no rodapé da tabela, além de garantirmos que a data seja exibida no formato dia/mês/ano . Também, precisamos garantir que negociações, depois de criadas, não sejam modificadas. Por fim, a lista de negociações não pode ser alterada, ela só poderá receber novos itens e, se quisermos uma nova lista, precisaremos criá-la do zero.

Da maneira como nossa aplicação se encontra, não será nada fácil implementar com segurança as regras que vimos no parágrafo anterior. Nosso código mistura em um único lugar o código que representa uma negociação com sua apresentação. Para tornar ainda mais desanimador, não temos uma estrutura que pode ser reaproveitada entre projetos.

Será que um ninja se sairia bem dessa? Ou quem sabe um cangaceiro possa botar a casa em ordem?

1.2 O PADRÃO MVC (MODEL-VIEW-CONTROLLER)

Existem diversos padrões que podem ser aplicados para organizar o código da nossa aplicação, e talvez o mais acessível deles seja o padrão MVC (*Model-View-Controller*). Este padrão cria uma separação marcante entre os dados da aplicação (*model*) e sua apresentação (*view*). Ações do usuário são interceptadas por um controller responsável em atualizar o modelo e refletir seu estado mais atual através de sua respectiva view.

Além do padrão MVC, trabalharemos com dois paradigmas da programação que compõem a cartucheira de todo cangaceiro, o paradigma **orientado a objetos** e o paradigma **funcional**.

Caso o leitor nunca tenha aplicado o paradigma orientado a objetos, não deve se preocupar, pois seus conceitos-chaves serão ensinados ao longo do livro. No entanto, é necessário ter um conhecimento fundamental da linguagem JavaScript.

A ideia é transitarmos entre os dois paradigmas, como um espírito errante do cangaço que vive entre dois mundos, utilizando

o que cada um deles oferece de melhor para o problema a ser resolvido. No final, isso tudo resultará em um miniframework.

A construção de um miniframework não é a finalidade deste livro, mas consequência de aprendizado da arte de um cangaceiro na programação. **A ideia é que o leitor possa extrair de cada passagem desta obra recursos que são interessantes de serem utilizados em suas aplicações.**

Nosso trabalho de verdade começa no próximo capítulo no qual daremos início a criação do modelo de negociações.

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/01>

NEGOCIAR COM O CANGACEIRO, TEM CORAGEM?

"Vivendo se aprende; mas o que se aprende, mais, é só a fazer outras maiores perguntas." - Grande Sertão: Veredas

No capítulo anterior, vimos que nosso código não realizava uma separação clara entre modelo e sua apresentação, tornando-o difícil de compreender e manter. Também foi proposto o uso do modelo MVC (*Model-View-Controller*) para melhorar sua estrutura.

Da tríade do modelo MVC, começaremos construindo um modelo de negociação. Mas aliás, o que seria um modelo?

2.1 O PAPEL DE UM MODELO

Um modelo é uma abstração de algo do mundo real. Um exemplo de modelo é aquele criado por um analista de mercado,

permitindo-o simular operações e até prever seu comportamento. No contexto de nossa aplicação, criaremos um modelo de negociação.

Nosso modelo precisa materializar as mesmas regras de uma negociação; em outras palavras, deve seguir uma especificação de negociação. O paradigma orientado a objetos trata especificações através do uso de **classes**.

A partir do ES2015 (ES6), ficou ainda mais fácil materializar em nosso código a criação de classe com a introdução da sintaxe `class`.

Antes de continuar com nosso projeto, vamos remover o script `client/app/index.js` de `client/index.html`, pois não usaremos mais este código. Com o arquivo removido, podemos dar início a construção da nossa classe `Negociacao`.

2.2 A CLASSE NEGOCIAÇÃO

Vamos criar o script que declarará nossa classe `Negociacao` na pasta `client/app/domain/negociacao/Negociacao.js`. Este simples ato de criar um novo arquivo já evidencia algumas convenções que utilizaremos até o final do livro.

A pasta `client/domain` é aquela que armazenará todos os scripts que dizem respeito ao domínio da aplicação, no caso, tudo aquilo sobre negociações. A subpasta `client/app/domain/negociacao` não foi criada por engano, é dentro dela que o script da nossa classe residirá, inclusive classes de serviços que possuem forte relação com esse domínio. Dessa forma, basta olharmos uma única pasta para saber sobre

determinado domínio da aplicação.

Por fim, o nome do script segue o padrão *PascalCase*, caracterizado por arquivos que começam com letra maiúscula, assim como cada palavra que compõe o nome do arquivo. Essa convenção torna evidente para quem estuda a estrutura do projeto que este arquivo possui a declaração de uma classe, pois toda classe por convenção segue o mesmo padrão de escrita.

Assim, alteraremos nosso arquivo `client/app/domain/negociacao/Negociacao.js` que acabamos de criar:

```
// client/app/domain/negociacao/Negociacao.js

class Negociacao {

}
```

Uma classe no mundo da Orientação a Objetos define a estrutura que todo objeto criado a partir dela deve seguir. Sabemos que uma negociação precisa ter uma data, uma quantidade e um valor. É através da função `constructor()` que definimos as propriedades de uma classe:

```
// client/app/domain/negociacao/Negociacao.js

class Negociacao {

  constructor() {

    this.data = new Date(); // data atual
    this.quantidade = 1;
    this.valor = 0.0;
  }

}
```

Através de `this.nomeDaPropriedade`, especificamos que a

negociação terá: data , quantidade e valor , cada propriedade com seu valor padrão.

Em seguida, no arquivo `client/index.html` , vamos incluir uma nova tag `<script>` antes do fechamento da tag `</body>` . Nosso objetivo será criar duas instâncias de negociação, ou seja, dois objetos criados a partir da classe `Negociacao` . Usaremos as variáveis `n1` e `n2` .

```
<!-- client/index.html -->
<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script>

    var n1 = new Negociacao();
    console.log(n1);

    var n2 = new Negociacao();
    console.log(n2);

</script>
<!-- código posterior omitido -->
```

Vamos recarregar o navegador e abrir o console. Vejam que já aparecem duas `Negociacoes` :

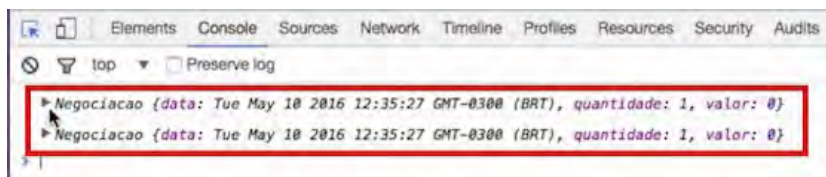


Figura 2.1: Duas negociações impressas

Cada instância criada possui as mesmas propriedades e valores, pois foram criadas a partir da mesma classe que atribui valores padrões para cada uma delas.

O uso do operador `new` não foi por acaso. Vejamos o que acontece caso ele seja omitido:

```
<!-- client/index.html -->
<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script>

    var n1 = new Negociacao();
    console.log(n1);

    var n2 = Negociacao(); // <- operador new removido
    console.log(n2);

</script>
<! código posterior omitido
```

No navegador, uma mensagem de erro é exibida no console:

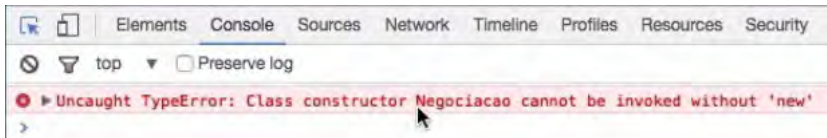


Figura 2.2: Mensagem de erro

Somos informados de que a classe `constructor()` não pode ser invocada sem operador `new`. O operador `new` é bastante importante por ser o responsável pela inicialização da variável implícita `this` de cada instância criada, ou seja, de cada objeto criado. Dessa forma, caso o programador esqueça de usar o operador `new` para criar a instância de uma classe, o que não faz sentido, ele será avisado com uma bela mensagem de erro no console.

Como cada instância criada possui seu próprio contexto em `this`, podemos alterar o valor das propriedades de cada objeto,

sem que isso interfira no outro:

```
<!-- client/index.html -->
<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script>

    var n1 = new Negociacao();
    n1.quantidade = 10;
    console.log(n1.quantidade);

    var n2 = Negociacao();
    n2.quantidade = 20;
    console.log(n2.quantidade);
    console.log(n1.quantidade); // continua 10

</script>
<!-- código posterior omitido -->
```

Agora, se executarmos o código, no console do navegador, veremos que cada `Negociacao` terá um valor diferente.

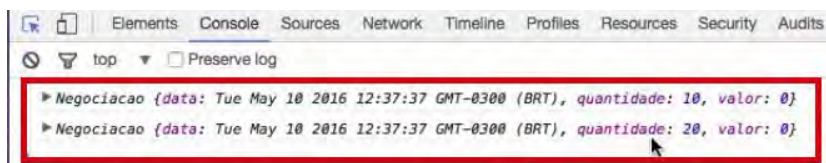


Figura 2.3: Negociações impressas 2

Realizamos o primeiro teste com duas instâncias que possuem seus próprios valores de propriedade, mas que compartilham uma mesma estrutura com `data`, `quantidade` e `valor`.

Antes de continuarmos, vamos apagar uma das negociações, pois uma já é suficiente para os novos testes que realizaremos a seguir:

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

```

<script src="app/domain/negociacao/Negociacao.js"></script>
<script>

    var n1 = new Negociacao();
    n1.quantidade = 10;
    console.log(n1.quantidade);
</script>
<!-- código posterior omitido -->

```

Veremos a seguir outra forma de criarmos instâncias de uma classe.

2.3 CONSTRUTOR DE CLASSE

Toda negociação precisa ter uma data, quantidade e valor ou não será uma negociação. Vejamos um exemplo que constrói uma instância de `Negociacao` com todos os seus dados atribuídos:

```

<!-- client/index.html -->
<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script>

    var n1 = new Negociacao();
    n1.data = new Date();
    n1.quantidade = 10;
    n1.valor = 200.50;
    console.log(n1);
</script>
<!-- código posterior omitido -->

```

Nosso código é funcional, mas se toda negociação precisa ter uma data, quantidade e valor, que tal definirmos esses valores na mesma instrução que cria uma nova instância? Além de nosso código ficar menos verboso, deixamos clara essa necessidade na especificação da classe.

Você deve lembrar que toda classe possui um

`constructor()` , uma função especial que será chamada toda vez que o operador `new` for usado para criar uma instância de uma classe. Sendo uma função, ela pode receber parâmetros.

Queremos poder fazer isso:

```
<!-- client/index.html -->
<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script>
    // basta uma instrução apenas em vez de 4!
    var n1 = new Negociacao(new Date(), 5, 700);
    console.log(n1);
</script>
<!-- código posterior omitido -->
```

Na classe `Negociacao` , adicionaremos os parâmetros de seu `constructor()` :

```
// client/app/domain/negociacao/Negociacao.js

class Negociacao {

    constructor(data, quantidade, valor) {

        // cada parâmetro recebido será atribuído às propriedades
        da classe
        this.data = data;
        this.quantidade = quantidade;
        this.valor = valor;
    }
}
```

Veja que a passagem dos parâmetros para a instância de `Negociacao` segue a mesma ordem de seus parâmetros definidos em seu `constructor()` . Cada parâmetro recebido é repassado para cada propriedade da classe.

Faremos um teste no console para ver se tudo funcionou

corretamente:

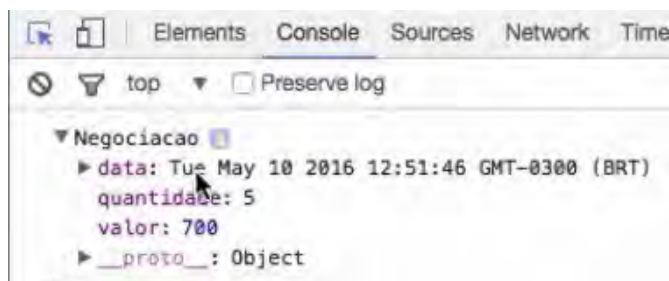


Figura 2.4: Teste no console

Os valores coincidem com os que especificamos no HTML. Porém, ainda não calculamos o volume - a quantidade multiplicada pelo valor. Faremos isto agora:

```
<!-- client/index.html -->
<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script>

    var n1 = new Negociacao(new Date(), 5, 700);
    var volume = n1.quantidade * n1.valor;
    console.log(volume);
</script>
<!-- código posterior omitido -->
```

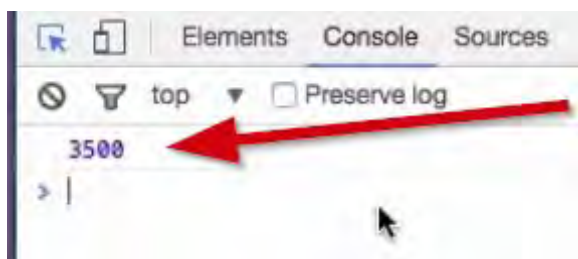


Figura 2.5: Volume

O valor 3500 é o resultado da operação de multiplicação. Mas estamos fazendo uma programação procedural, e sempre que o programador precisar do volume, ele mesmo terá de realizar este cálculo.

Além de ser algo tedioso ter de repetir o cálculo do volume, as chances de cada programador cometer algum erro ao calculá-lo são grandes, mesmo com um código simples como esse. Aliás, esse é um problema que o paradigma orientado a objetos vem resolver.

2.4 MÉTODOS DE CLASSE

No paradigma da Orientação a Objetos, há uma forte conexão entre dado e comportamento, diferente da programação procedural, na qual temos de um lado o dado e, do outro, procedimentos que operam sobre estes dados. Nossa classe possui apenas dados, chegou a hora de dotá-la do seu primeiro comportamento, o `obtemVolume`.

Vamos criar o método:

```
// client/app/domain/negociacao/Negociacao.js
```

```
class Negociacao {  
  
  constructor(data, quantidade, valor) {  
  
    this.data = data;  
    this.quantidade = quantidade;  
    this.valor = valor;  
  }  
  
  obterVolume() {  
  
    return this.quantidade * this.valor;  
  }  
}
```

```
}
```

Funções que definem um comportamento de uma classe são chamadas de **métodos** para alinhar o vocabulário com o paradigma da Orientação a Objetos. No entanto, não é raro alguns cangaceiros continuarem a utilizar o termo função.

Agora, basta perguntarmos para a própria instância da classe `Negociacao` para que obtenha seu volume para nós. Temos um único local que centraliza o cálculo do volume. Então, alteraremos nosso teste para usarmos o método que acabamos de criar:

```
<!-- client/index.html -->
<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script>

    var n1 = new Negociacao(new Date(), 5, 700);
    var volume = n1.obtemVolume();
    console.log(volume);

</script>
<!-- código posterior omitido -->
```

Agora, ao executarmos o código, o valor `3500` aparecerá novamente no console:

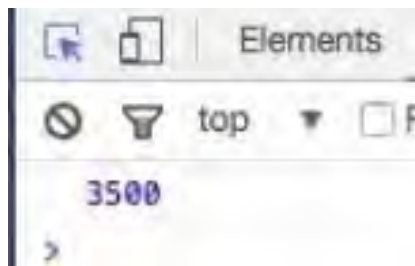


Figura 2.6: Volume 2

O valor foi calculado corretamente. O objeto sabe lidar com as propriedades trabalhadas. Logo, a regra de como obter o volume está no próprio objeto. Voltamos a ter uma conexão entre dado e comportamento.

2.5 ENCAPSULAMENTO

Nós já conseguimos trabalhar com a classe `Negociacao`, mas precisamos implementar outra regra de negócio além do cálculo do volume: após criada a negociação, esta não poderá ser alterada. No entanto, nosso código não está preparado para lidar com essa regra, como podemos verificar com um novo teste:

```
<!-- client/index.html -->
<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script>

    var n1 = new Negociacao(new Date(), 5, 700);
    n1.quantidade = 10;
    console.log(n1.quantidade);
</script>
<!-- código posterior omitido -->
```

No código anterior, estamos atribuindo um novo valor à propriedade `quantidade` da nossa instância, para em seguida imprimirmos seu valor. Qual valor será impresso no console, 5 ou 10 ?

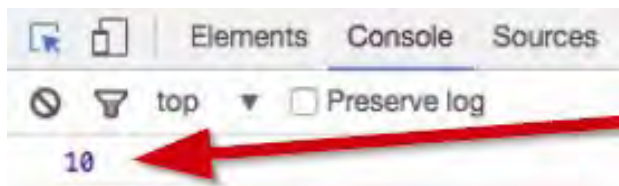


Figura 2.7: Regra alterada

Ele imprimiu o valor recém-adicionado 10 e isto pode causar problemas. Não é à toa a regra de uma negociação na qual ela não pode ser alterada depois de criada.

Imagine que acabamos de fazer uma negociação e combinamos um determinado valor, mas depois alguém decide alterá-la para benefício próprio. Nosso objetivo é que as propriedades de uma negociação sejam somente de leitura.

No entanto, a linguagem JavaScript - até a atual data - não nos permite usar modificadores de acesso, um recurso presente em outras linguagens orientadas a objeto. Não podemos fazer com que uma propriedade seja apenas leitura (ou gravação). O que podemos fazer é convencionar-mos que toda propriedade de uma classe prefixada com um **underline** (`_`) não deve ser acessada fora dos métodos da própria classe:

```
// client/app/domain/negociacao/Negociacao.js

class Negociacao {

  constructor(data, quantidade, valor) {

    this._data = data;
    this._quantidade = quantidade;
    this._valor = valor;
  }

  obterVolume() {

    return this._quantidade * this._valor;
  }
}
```

Esta será uma convenção que informará ao programador que as propriedades que contenham `_` (*underline*) só poderão ser acessadas pelos próprios métodos da classe. Isto significa que,

mesmo podendo imprimir a propriedade `_quantidade` com outro valor, não deveríamos mais poder acessá-la. O `_` funciona como um aviso dizendo: "programador, você não pode alterar esta propriedade!".

Então, se usamos a convenção de utilizar o prefixo, como faremos para imprimir o valor da quantidade de uma negociação, por exemplo? Será que precisaremos abandonar a convenção que acabamos de aplicar? Não será preciso, pois podemos criar **métodos acessadores**.

Métodos acessadores geralmente começam com o prefixo `get` em seus nomes. No caso da nossa classe `Negociacao`, adicionaremos o método `getQuantidade()`, que retornará o `_quantidade`. Usaremos também o `getData()` e o `getValor()` que terão finalidades semelhantes:

```
// client/app/domain/negociacao/Negociacao.js

class Negociacao {

  constructor(data, quantidade, valor) {

    this._data = data;
    this._quantidade = quantidade;
    this._valor = valor;
  }

  obterVolume() {

    return this._quantidade * this._valor;
  }

  getData() {

    return this._data;
  }
}
```

```

    getQuantidade() {

        return this._quantidade;
    }

    getValor() {

        return this._valor;
    }
}

```

Como são métodos da própria classe, seguindo nossa convenção, eles podem acessar às propriedades prefixadas com (`_`). E qualquer outro código que precisar ler as propriedades de um objeto do tipo `Negociacao` poderá fazê-lo através dos métodos acessadores que criamos.

Vamos atualizar nosso código em `index.html`, para refletir o estado atual do nosso código:

```

<!-- client/index.html -->
<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script>

    var n1 = new Negociacao(new Date(), 5, 700);
    console.log(n1.getQuantidade());
    console.log(n1.getData());
    console.log(n1.getValor());
</script>
<!-- código posterior omitido -->

```

Observe que estamos acessando os demais campos:

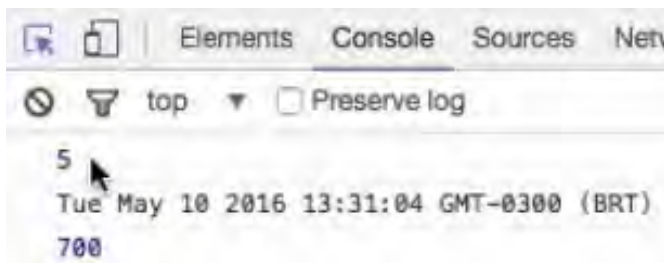


Figura 2.8: Valores no console

Os valores serão impressos corretamente no console. Em seguida, modificaremos o `obtemVolume()` para `getVolume` em `Negociacao.js`

```
// client/app/domain/negociacao/Negociacao.js

class Negociacao {

  constructor(data, quantidade, valor) {

    this._data = data;
    this._quantidade = quantidade;
    this._valor = valor;
  }

  // trocamos o nome de obtemVolume para getVolume
  getVolume() {

    return this._quantidade = this._valor;
  }

  // código posterior omitido
}
```

Vamos testar todos os métodos acessadores em `index.html` :

```
<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script>
```

```

var n1 = new Negociacao(new Date(), 5, 700);
console.log(n1.getQuantidade());
console.log(n1.getData());
console.log(n1.getValor());
console.log(n1.getVolume());
</script>
<!-- código posterior omitido -->

```

No navegador, veremos que os valores quantidade , data , valor e volume serão impressos corretamente.

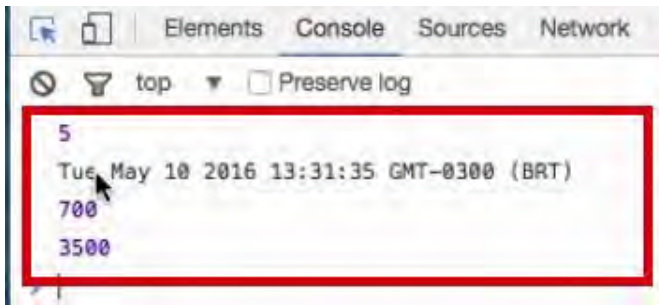


Figura 2.9: Quantidade e volume

2.6 A SINTAXE GET

Podemos enxugar nossos métodos acessadores através da sintaxe `get` que a linguagem nos oferece. Vamos alterar nosso código:

```

// client/app/domain/negociacao/Negociacao.js

class Negociacao {

  constructor(data, quantidade, valor) {

    this._data = data;
    this._quantidade = quantidade;
    this._valor = valor;
  }
}

```

```

get volume() {

    return this._quantidade * this._valor;
}

get data() {

    return this._data;
}

get quantidade() {

    return this._quantidade;
}

get valor() {

    return this._valor;
}
}

```

Vejam que a sintaxe `get` vem antes do nome do método, que não possui mais o prefixo `get` que usamos antes. Por fim, acessamos esses métodos como se fossem propriedades em nosso código.

Vamos alterar `index.html` para vermos em prática a alteração:

```

<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script>

    var n1 = new Negociacao(new Date(), 5, 700);

    console.log(n1.quantidade);
    console.log(n1.data);
    console.log(n1.valor);
    console.log(n1.volume);
</script>
<!-- código posterior omitido -->

```

Estamos criando uma propriedade *getter* de acesso à leitura. E mesmo sendo um método, precisamos acessá-lo como uma propriedade. Contudo, por debaixo dos panos, as propriedades que acessamos continuam sendo métodos.

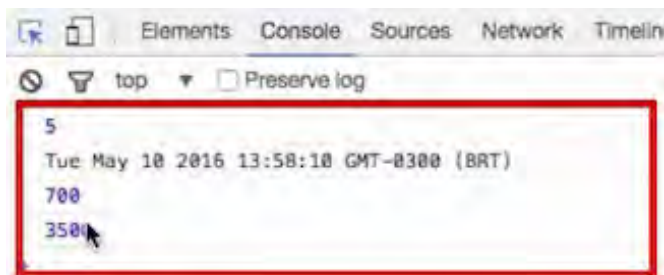


Figura 2.10: console

Outra característica de métodos criados com a sintaxe `get` é que, se tentarmos atribuir um novo valor à propriedade, ele será ignorado. Vejamos:

```
<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script>

    var n1 = new Negociacao(new Date(), 5, 700);
    n1.quantidade = 1000; // instrução é ignorada, sem mensagem d
e erro
    console.log(n1.quantidade);
    console.log(n1.data);
    console.log(n1.valor);
    console.log(n1.volume);
</script>
<!-- código posterior omitido -->
```

O valor `1000` não será repassado para o `_quantidade`. O valor continuará sendo `5`:

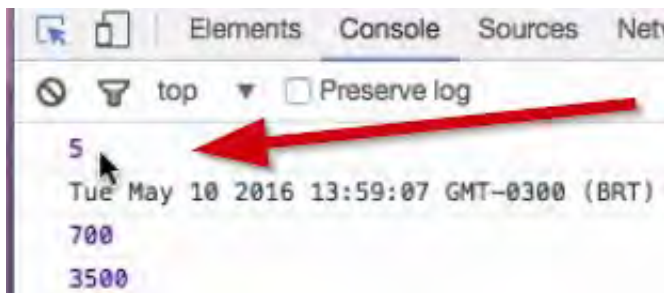


Figura 2.11: Valor de quantidade inalterados

Isto acontece quando a propriedade é um *getter* (ou seja, é de leitura), já que não podemos atribuir a ela um valor. Mas ainda podemos modificá-la com `n1._quantidade` :

```
<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script>

    var n1 = new Negociacao(new Date(), 5, 700);

    // instrução é ignorada, sem mensagem de erro
    n1.quantidade = 1000;

    // nada nos impede de fazermos isso, apenas nossa convenção
    n1._quantidade = 5000;

    console.log(n1.quantidade);
    console.log(n1.data);
    console.log(n1.valor);
    console.log(n1.volume);
</script>
<!-- código anterior omitido -->
```

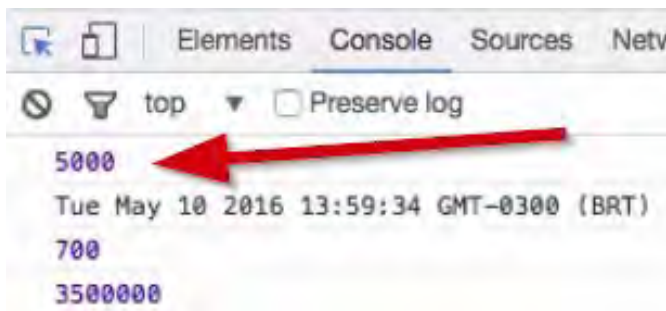



Figura 2.12: Quantidade alterada

Como vimos, isso não impede o programador desavisado de ainda acessar as propriedades prefixadas com *underline*. No entanto, ainda temos um artifício da linguagem que pode evitar que isso aconteça.

2.7 OBJETOS IMUTÁVEIS

Nós usaremos um artifício existente há algum tempo na linguagem JavaScript que permite "congelar" um objeto. Propriedades de objetos congeladas não podem receber novas atribuições. Conseguimos esse comportamento através do método `Object.freeze()`.

Vamos congelar a instância `n1` em nosso teste:

```
<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script>

    var n1 = new Negociacao(new Date(), 5, 700);

    Object.freeze(n1);
    n1._quantidade = 1000000000000; // não consegue modificar e ne
    num erro acontece
```

```

    console.log(n1._quantidade);
</script>
<!-- código posterior omitido -->

```

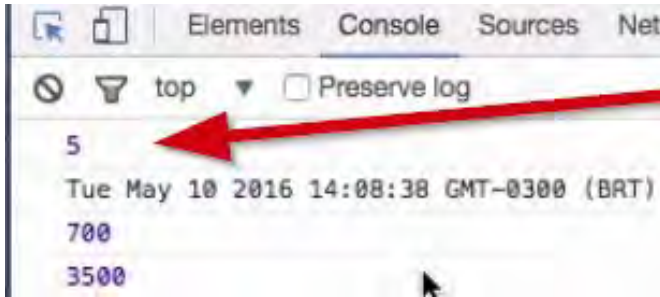


Figura 2.13: Objeto congelado

No console, veremos o valor 5 ; isto significa que não conseguimos mais alterar e o objeto foi congelado. Vale lembrar que quantidade é uma propriedade que chamamos em Negociacao.js , que por "debaixo dos panos" rodará como um método e retornará this._quantidade .

Mesmo colocando n1._quantidade = 1000000000000; no index.html , esse valor parece ter sido ignorado. Isso é bom, pois agora o desenvolvedor já não conseguirá alterar esta quantidade. Vamos fazer um pequeno teste que nos mostrará o que está congelado no console:

```

<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script>

    var n1 = new Negociacao(new Date(), 5, 700);

    console.log(Object.isFrozen(n1));
    Object.freeze(n1);
    console.log(Object.isFrozen(n1));
    n1._quantidade = 1000000000000;
</script>

```

No console, veremos o seguinte resultado:

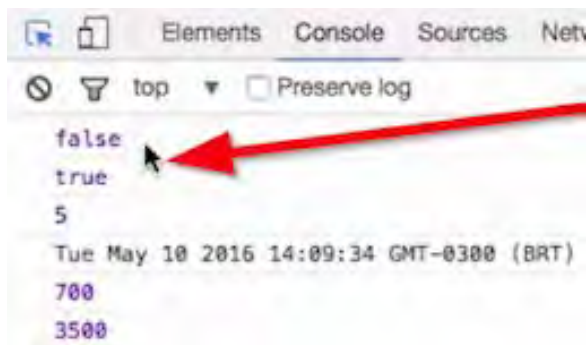


Figura 2.14: Console com false e true

O objeto antes não era congelado e, depois, foi. Por isso, não conseguimos alterar `_quantidade`, o atributo que definimos ser privado por convenção. Porém, essa solução é procedural, porque você terá sempre de lembrar de congelar a instância.

Entretanto, queremos que, ao usarmos o `new Negociacao`, uma instância congelada já seja devolvida. Nós podemos fazer isso congelando a instância no `constructor()`. Em `app/domain/negociacao/Negociacao.js`, adicionaremos `Object.freeze()` após o último `this`:

```
// client/app/domain/negociacao/Negociacao.js

class Negociacao {

  constructor(data, quantidade, valor) {

    this._data = data;
    this._quantidade = quantidade;
    this._valor = valor;
    Object.freeze(this);

  }
}
```

```
// código posterior omitido
```

Vocês lembram que o `this` é uma variável implícita? E quando algum método é chamado, temos acesso à instância trabalhada. O `this` do `Object.freeze()` será o `n1` no `index.html`.

```
<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script>

    var n1 = new Negociacao(new Date(), 5, 700);

    n1._quantidade = 1000000000000;

    console.log(n1.quantidade);
    console.log(n1.data);
    console.log(n1.valor);
    console.log(n1.volume);
</script>
<!-- código anterior omitido -->
```

Então, o resultado de `new Negociacao(/* parâmetros */)` já devolverá uma instância congelada. Aparentemente, resolvemos o nosso problema.

Mas será que nossa negociação é realmente imutável? Vamos verificar mais adiante se de fato a `Negociacao` não será alterada.

2.8 A INSTÂNCIA É IMUTÁVEL MESMO?

Pela regra de negócio, uma negociação deve ser imutável e não pode ser alterada depois de criada. Mas faremos um novo teste tentando alterar a data da negociação. Nós fizemos isto com a quantidade. Agora, criaremos a variável `outroDia`, que será referente à data diferente da atual:

```

<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script>

    var n1 = new Negociacao(new Date(), 5, 700);

    console.log(n1.data);

    var outroDia = new Date();

    // setDate permite alterar o dia de uma data
    outroDia.setDate(11);

    n1.data = outroDia;

    console.log(n1.data);
</script>
<!-- código posterior omitido -->

```

Para verificar se o código funciona corretamente, adicionamos o `console.log()`. Se `Negociacao` for realmente imutável, a data não poderá ser alterada.

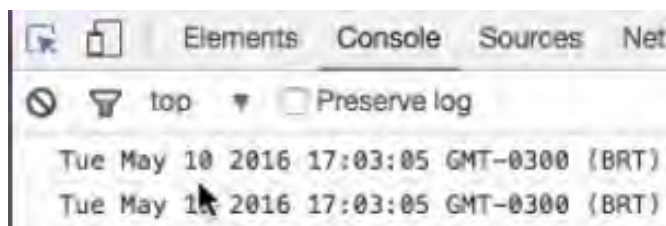


Figura 2.15: Negociação imutável

Vemos que ela continuou imutável.

Agora, faremos um teste diferente. Não vamos mais trabalhar com a variável `outroDia`, e substituiremos por `n1.data.setDate(11)`. Veja que estamos chamando o método `setDate` da propriedade `data` do próprio objeto:

```

<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script>

    var n1 = new Negociacao(new Date(), 5, 700);

    console.log(n1.data);

    n1.data.setDate(11);

    console.log(n1.data);
</script>
<!-- código posterior omitido -->

```

Será que a data foi alterada?

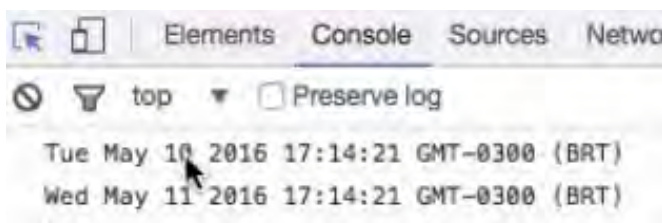


Figura 2.16: data alterada

Veja que a data foi alterada, pois a data exibida no console foi o dia 11. Nossa negociação é mutável! Nós não podemos deixar isso acontecer.

Mas se nós utilizamos o `Object.freeze()`, por que isso aconteceu? O `Object.freeze` somente evita que seja realizada uma nova atribuição à propriedade do objeto congelado. Não podemos fazer, por exemplo, `n1.data = new Date()`, pois estamos atribuindo um novo valor à propriedade. Porém, é perfeitamente possível fazermos `n1.data.setDate(11)`, pois as propriedades do objeto definido em `n1.data` não são congeladas.

Quando o programador chega a um ponto em que a linguagem não oferece uma solução nativa para o problema, ele pode apelar para sua criatividade; característica fundamental de todo cangaceiro para lidar com as adversidades que encontra.

2.9 PROGRAMAÇÃO DEFENSIVA

Uma maneira de tornarmos a `data` de uma negociação imutável é retornarmos um novo objeto `Date` toda vez que o *getter* da `data` for acessado. Dessa maneira, qualquer modificação através das funções que toda instância de `Date` possui só modificará a cópia, e não o objeto original.

Atualmente, o `get` do arquivo `Negociacao` está assim:

```
// app/domain/negociacao/Negociacao.js
// código anterior omitido

get data() {

    return this._data;
}
// código posterior omitido
```

Vamos modificar o retorno do `get data()` :

```
// app/domain/negociacao/Negociacao.js
// código anterior omitido

get data() {

    return new Date(this._data.getTime());
}
// código posterior omitido
```

O `getTime()` de uma `data` retornará um número *long* com uma representação da data. Vejamos no console:

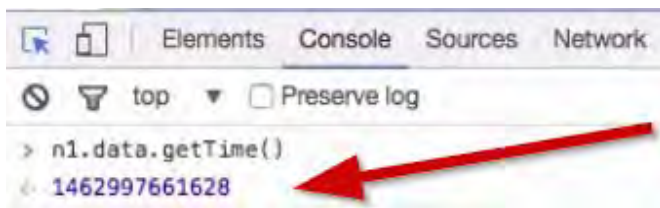


Figura 2.17: Retorno da data

No `constructor()` de `Date`, este número será aceito para a construção de uma nova data. Então, se tentarmos alterar a data através de `n1.data.setDate(11)`, não estaremos mudando a data que `n1` guarda, mas uma cópia. Isto é o que chamamos de **programação defensiva**.

Vamos testar o nosso código, após as alterações feitas no `getData()` da classe `Negociacao`. Depois de recarregarmos a página no navegador, veremos a seguinte data no console:

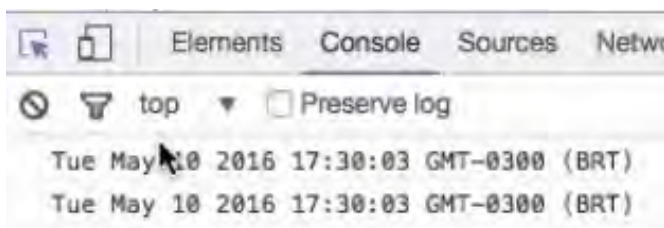


Figura 2.18: Datas no console

Devemos ter o mesmo cuidado com o `constructor()` também, porque se passamos uma data no `constructor()` de `Negociacao`, qualquer mudança na referência da data fora da classe modificará também sua data. Vejamos o problema acontecendo:

```
<!-- código anterior omitido -->
```



```

<script src="app/domain/negociacao/Negociacao.js"></script>
<script>

    var hoje = new Date();

    var n1 = new Negociacao(hoje, 5, 700);

    console.log(n1.data);

    // altera a data que foi armazenada em Negociacao também
    hoje.setDate(11);

    console.log(n1.data);
</script>

```

Observe que, em vez de passarmos o `new Date()` para o `constructor()` de `Negociacao`, passamos a variável `hoje` que referencia uma `Date`. Quando a `Negociacao` receber o objeto `hoje`, ele guardará uma referência para o mesmo objeto. Isto significa que, se alterarmos a variável `hoje`, modificaremos a data que está na `Negociacao`. Se executarmos o código, a data será alterada para o dia 11.

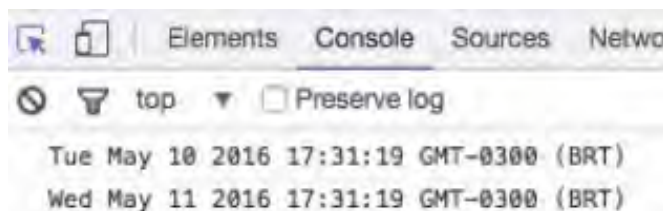


Figura 2.19: Data alterada 2

Por isso, também usaremos o `getTime()` no `constructor()`:

```

// app/domain/negociacao/Negociacao.js

class Negociacao {

```

```

constructor(data, quantidade, valor) {

    // criando uma nova data, uma nova referência
    this._data = new Date(data.getTime());

    this._quantidade = quantidade;
    this._valor = valor;
    Object.freeze(this);
}
// código posterior omitido

```

Ao fazermos isto, já não conseguiremos alterar a referência no `index.html`, porque o que estamos guardando no `Negociacao` não é mais uma referência para `hoje`. Então, nós usaremos um novo objeto. Quando recarregarmos a página no navegador, a data que aparecerá no console será `10`.

No momento da criação do design das classes, **seja cuidadoso** com a **mutabilidade**, tomando as medidas necessárias para garantir a imutabilidade quando for preciso.

Com isso, finalizamos a blindagem da nossa classe para garantir a sua imutabilidade. Existem outras soluções mais avançadas no JS para tentarmos emular o *privacy* - a privacidade - do seu código; entretanto, perdemos em legibilidade e performance. Então, a solução usada é a mais viável.

Podemos simplificar ainda mais nosso código com um pequeno truque que veremos a seguir.

2.10 MENOS VERBOSIDADE NO CONSTRUCTOR COM OBJECT.ASSIGN

Vamos lançar nosso olhar mais uma vez no `constructor` da classe `Negociacao`:

```
// app/domain/negociacao/Negociacao.js

class Negociacao {

  constructor(data, quantidade, valor) {

    this._data = new Date(data.getTime());
    this._quantidade = quantidade;
    this._valor = valor;
    Object.freeze(this);
  }
// código posterior omitido
```

Cada parâmetro recebido no `constructor` é atribuído a uma propriedade de instância da classe através de instruções que seguem a estrutura `this._nomeDaPropriedade = parametro`. Se tivéssemos uma quantidade de parâmetros muito grande, o código do nosso `constructor` cresceria bastante.

Podemos simplificar bastante esse processo de atribuição dos parâmetros recebidos pelo `constructor` nas propriedades da instância da classe com o método `Object.assign()`. Primeiro, vamos compreender como ele funciona.

O método `Object.assign()` é usado para copiar os valores de todas as propriedades próprias enumeráveis de um ou mais objetos de origem para um objeto destino. Ele retornará o objeto destino. Vejamos um exemplo:

```
// EXEMPLO APENAS, FAÇA NO CONSOLE DO CHROME

var origem1 = { nome: 'Cangaceiro' };
var origem2 = { idade: 20 };

var copia = Object.assign({}, origem1, origem2);

console.log(copia); // exibe { nome: 'Cangaceiro', idade: 20 };
```

O primeiro parâmetro de `Object.assign()` é o objeto

destino que receberá a cópia das propriedades dos objetos de origem, um objeto sem qualquer propriedade com a sintaxe `{}` . Podemos passar quantos objetos de origem desejarmos a partir do segundo parâmetro, em nosso caso passamos `origem1` e `origem2` . O retorno do método é um objeto que contém as propriedades de `origem1` e `origem2` .

Se mudarmos nosso exemplo e passarmos a referência de um objeto já existente como parâmetro de destino no método `Object.assign()` , esse objeto será modificado ganhando as propriedades de um ou mais objetos de origem passados como parâmetro:

```
// EXEMPLO APENAS, FAÇA NO CONSOLE DO CHROME

var origem = { idade: 20 };
var destino = { nome: 'Cangaceiro' };

Object.assign(destino, origem);
console.log(destino); // exibe { nome: 'Cangaceiro', idade: 20 };
```

O comportamento que acabamos de ver nos permite realizar a seguinte alteração no constructor de `Negociacao` :

```
// app/domain/negociacao/Negociacao.js

class Negociacao {

  constructor(data, quantidade, valor) {

    Object.assign(this, { _data: new Date(data.getTime()), _q
    uantidade: quantidade, _valor: valor});

    Object.freeze(this);
  }
  // código posterior omitido
```

Estamos copiando para `this` , que representa a instância da

classe `Negociacao` , as propriedades do objeto de origem criado com as propriedades `_data` , `_quantidade` e `_valor` e com os valores recebidos pelo constructor da classe. Temos apenas uma instrução de atribuição no lugar de três instruções.

Em um primeiro momento, não parece ser muita vantagem trocar três instruções por uma apenas com `Object.assign()` , pois a legibilidade não é lá uma das melhores. Mas se combinarmos este recurso com o próximo que veremos a seguir, teremos uma solução bem interessante.

2.11 ATALHO PARA PROPRIEDADES DE OBJETOS LITERAIS

O método `Object.assign()` recebeu como origem um objeto criado na forma `literal` , isto é, através de `{}` . Vejamos um exemplo isolado que cria um objeto através da forma `literal`:

```
// EXEMPLO APENAS, FAÇA NO CONSOLE DO CHROME

var perfil = 'Cangaceiro';

var objeto = { perfil: perfil };

console.log(objeto); // exhibe { perfil: "Cangaceiro" }
```

No exemplo anterior, a propriedade `perfil` recebe como valor uma variável de mesmo nome. Quando o nome da propriedade tem o mesmo nome da variável que será atribuída como seu valor, podemos declarar nosso objeto desta forma:

```
// EXEMPLO APENAS, FAÇA NO CONSOLE DO CHROME

var perfil = 'Cangaceiro';
```

```
var objeto = { perfil }; // passamos apenas perfil

console.log(objeto); // exibe { perfil: "Cangaceiro" }
```

Essa novidade introduzida no ES2016 (ES6) nos permite simplificar o uso de `Object.assign()` que vimos antes. Alterando o constructor de `Negociacao`:

```
// app/domain/negociacao/Negociacao.js

class Negociacao {

    // MUDANÇA NO NOME DOS PARÂMETROS, AGORA COM UNDERLINE
    constructor(_data, _quantidade, _valor) {

        // USANDO O ATALHO PARA DECLARAÇÃO DE PROPRIEDADES

        Object.assign(this, { _data: new Date(_data.getTime()), _
quantidade, _valor});
        Object.freeze(this);
    }
}
// código posterior omitido
```

Reparem que o nome dos parâmetros recebidos no constructor mudaram de `data`, `quantidade`, `valor` para `_data`, `_quantidade`, `_valor`. Isso não foi por acaso, a intenção é que eles tenham o mesmo nome das propriedades do objeto passado como origem para `Object.assign()`. Possuindo o mesmo nome, podemos usar o atalho de declaração de propriedades que vimos na seção anterior.

Para melhorarmos a legibilidade, vamos convencionar que toda propriedade recebida pelo constructor que precisa ser alterada antes da atribuição será atribuída individualmente, sem o auxílio de `Object.assign()`.

Alterando nosso código:

```
// app/domain/negociacao/Negociacao.js

class Negociacao {

  constructor(_data, _quantidade, _valor) {

    Object.assign(this, { _quantidade, _valor });
    this._data = new Date(_data.getTime());
    Object.freeze(this);
  }
}
// código posterior omitido
```

Simplificamos bastante a passagem dos parâmetros do constructor para as propriedades da classe.

Antes de continuarmos com o próximo capítulo, precisamos adquirir um novo hábito ao declararmos nossas variáveis.

2.12 AS SURPRESAS DE DECLARAÇÕES COM VAR

Temos nosso modelo de negociação pronto. Contudo, antes de continuarmos, vamos voltar nossa atenção para as variáveis declaradas com `var`.

Ausência de escopo de bloco

Declarações com `var` não criam escopo de bloco assim como em outras linguagens. Vejamos um exemplo:

```
<!-- CÓDIGO DE EXEMPLO, NÃO ENTRA EM NOSSA APLICAÇÃO -->
<script>
  for(var i = 1; i <= 100; i++) {
    console.log(i);
  }
  alert(i); // qual será o valor?
</script>
```

O código anterior imprimirá corretamente os números de 1 a 100 no console. Porém, a instrução `alert(i)` exibirá o valor 101 :

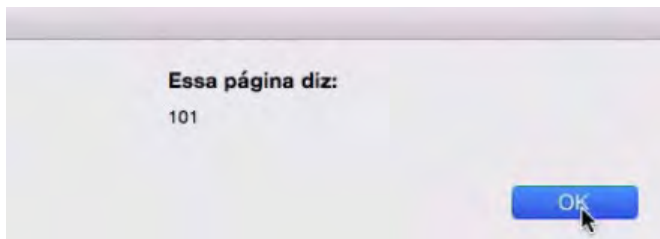


Figura 2.20: Alert

A variável `i` continua sendo acessível fora do bloco do `for` . A única maneira de termos um escopo para variáveis declaradas com `var` é ao declará-las dentro do bloco de uma função:

```
function exibeNome() {  
    var nome = 'Flávio Almeida';  
    alert(nome);  
}  
exibeNome(); // exibe Flávio Almeida  
alert(nome); // undefined
```

No exemplo que acabamos de ver, a variável `nome` não é acessível fora do bloco da função.

Declarações repetidas

Com `var` , podemos declarar variáveis com o mesmo nome sem que isso incomode o interpretador do JavaScript. Por exemplo:

```
function minhaFuncao() {
```



```
var nome = "Flávio";  
var nome = "Almeida"; // não precisa de var!  
}
```

Este comportamento não ajuda em nada o desenvolvedor a perceber, nas dezenas de linhas de código, que está declarando novamente uma variável que já havia sido declarada.

Acesso antes de sua declaração

Vejamos mais um exemplo:

```
function minhaFuncao() {  
  
    alert(nome);  
    var nome = 'Flávio';  
}  
  
minhaFuncao();
```

O código anterior exibe `undefined`, na função `alert()`, em vez de um erro. Isso acontece porque variáveis declaradas com `var` são içadas (*hoisting*) para o início do bloco da função na qual foram declaradas.

Por debaixo dos panos, o JavaScript executa nosso código da seguinte maneira:

```
// COMO O INTERPRETADOR LÊ NOSSO CÓDIGO  
function minhaFuncao() {  
  
    /*  
        Içou a declaração da variável para o topo do contexto, no  
        caso, início da função  
    */  
    var nome;  
    alert(nome);  
    nome = 'Flávio';  
}
```

```
minhaFuncao();
```

Esse comportamento permite que códigos como este funcionem:

```
function minhaFuncao() {  
    if(!nome) {  
        var nome = 'Flávio Almeida'  
    }  
    alert(nome);  
}  
  
minhaFuncao();
```

Por debaixo dos panos, nosso código será interpretado desta forma:

```
// COMO O INTERPRETADOR LÊ NOSSO CÓDIGO  
  
function minhaFuncao() {  
    var nome;  
    if(!nome) {  
        nome = 'Flávio Almeida'  
    }  
    alert(nome);  
}  
  
minhaFuncao();
```

Dessa maneira, a variável `nome` testada pelo `if` já foi declarada, porém com valor `undefined`. A boa notícia é que há outra maneira de declararmos variáveis em JavaScript, assunto da próxima seção.

2.13 DECLARAÇÃO DE VARIÁVEIS COM LET

A partir do ES2015 (ES6), foi introduzida uma nova sintaxe para declaração de variáveis, a sintaxe `let`. Vejamos as

características deste novo tipo de declaração.

Escopo de bloco

Vamos rever o exemplo da seção anterior, mas desta vez usando `let` para declarar a variável do bloco do `for` :

```
<!-- CÓDIGO DE EXEMPLO, NÃO ENTRA EM NOSSA APLICAÇÃO -->
<script>

    for(let i = 1; 1 <= 100; 1++) {
        console.log(i);
    }
    // undefined!
    alert(i);
</script>
```

A instrução `alert(i)` exibirá `undefined` , pois variáveis declaradas com `let` , dentro de um bloco, só são acessíveis dentro desse bloco.

Vejamos mais um teste:

```
<!-- CÓDIGO DE EXEMPLO, NÃO ENTRA EM NOSSA APLICAÇÃO -->
<script>

    for(let i = 1; i<= 100; i++) {
        // só existe no bloco for!
        let nome = "Flávio";
        console.log(i);
    }
    // undefined!
    alert(i);
    // undefined!
    alert(nome);
</script>
```

Não é possível haver declarações repetidas dentro de um mesmo bloco

Outro ponto interessante de `let` é que não podemos declarar duas variáveis com o mesmo nome dentro de um mesmo bloco `{}`. Vejamos:

```
function minhaFuncao() {  
    let nome = 'Flávio';  
    let nome = 'Almeida';  
}  
minhaFuncao();
```

A função anterior, ao ser chamada, resultará no erro:

Uncaught SyntaxError: Identifier 'nome' has already been declared

Contudo, o código seguinte é totalmente válido:

```
let nome = 'Flávio';  
  
function minhaFuncao() {  
    let nome = 'Almeida';  
    alert(nome);  
}  
minhaFuncao(); // exibe Almeida  
alert(nome); // exibe Flávio
```

Há mais um detalhe que precisamos entender a respeito do novo tipo de declaração que acabamos de aprender.

2.14 TEMPORAL DEAD ZONE

Vamos revisar comportamentos de variáveis declaradas com `var`. A função declara a variável `nome` por meio de `var`:

```
function minhaFuncao() {  
    alert(nome);  
    var nome = 'Almeida';  
}
```

```
minhaFuncao();
```

Sabemos que, através de *hoisting*, ela será interpretada desta forma:

```
// COMO O INTERPRETADOR LÊ NOSSO CÓDIGO
```

```
function minhaFuncao() {  
    var nome;  
    alert(nome);  
    nome = 'Almeida';  
}  
minhaFuncao();
```

Da maneira que é interpretada pelo JavaScript, temos a declaração `var nome;`, que não possui qualquer atribuição. Por isso o `alert(nome)` exibe *undefined* em vez de um erro alegando que a variável não existe. Contudo, quando usamos `let`, precisamos ficar atentos para o **Temporal Dead Zone**. O nome pode parecer um tanto assustador inicialmente.

Variáveis declaradas com `let` também são içadas (*hoisting*). Contudo, seu acesso só é permitido após receberem uma atribuição; caso contrário, teremos um erro. Vejamos o exemplo anterior, desta vez usando `let`:

```
function minhaFuncao() {  
    alert(nome);  
    let nome = 'Flávio';  
}  
  
minhaFuncao();
```

Quando executamos nosso código, receberemos a mensagem de erro:

```
Uncaught ReferenceError: nome is not defined
```

Como foi dito, variáveis declaradas com `let` também são içadas, então nosso código estará assim quando interpretado:

```
// COMO O INTERPRETADOR LÊ NOSSO CÓDIGO
```

```
function minhaFuncao() {  
    let nome;  
    alert(nome);  
    nome = 'Flávio';  
}  
  
minhaFuncao();
```

É entre a declaração da variável e sua atribuição que temos o *Temporal Dead Zone*. Qualquer acesso feito à variável nesse espaço de tempo resultará em `ReferenceError`. Contudo, é um erro benéfico, pois acessar uma variável antes de sua declaração é raramente intencional em uma aplicação.

Nada nos impede de deliberadamente fazermos isso:

```
function minhaFuncao() {  
    let nome = undefined;  
    alert(nome);  
    nome = 'Flávio';  
}  
  
minhaFuncao();
```

O resultado de `alert` será `undefined`. Faz todo sentido, porque acessamos a variável `nome` após receber uma atribuição, isto é, fora do *Temporal Dead Zone*.

Agora que já sabemos dos desdobramentos do uso de `let` em nossa aplicação, vamos alterar `client/index.html` para utilizarmos essa nova declaração:

```
<!-- código anterior omitido -->
```

```

<script src="app/domain/negociacao/Negociacao.js"></script>
<script>

    let hoje = new Date();

    let n1 = new Negociacao(hoje, 5, 700);

    console.log(n1.data);

    hoje.setDate(11);

    console.log(n1.data);
</script>

```

Se executarmos o código, tudo continuará funcionando normalmente.

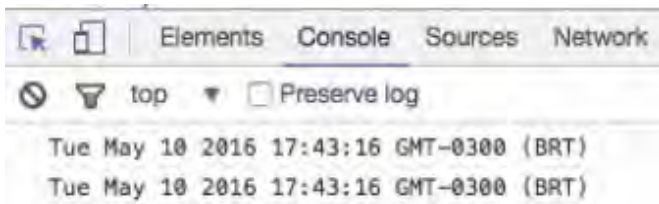


Figura 2.21: Data console

Neste capítulo, criamos um modelo de negociação aplicando o paradigma da Orientação a Objetos. Contudo, implementamos apenas o **M** do modelo MVC. Precisamos implementar as interações do usuário com o modelo, papel realizado pelo controller do modelo MVC, assunto do próximo capítulo.

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/02>

NO CANGAÇO, É AÇÃO PARA TODO LADO

"O real não está no início nem no fim, ele se mostra pra gente é no meio da travessia..." - Grande Sertão: Veredas

No capítulo anterior, criamos nosso modelo de negociação através da classe `Negociacao` e, através do console, realizamos diversos testes com instâncias dessa classe. Para que nossa aplicação seja útil para o usuário, essas instâncias devem considerar informações prestadas pelo usuário através do formulário definido `client/index.html`. Resolveremos essa questão com auxílio de um `Controller` do modelo MVC.

3.1 O PAPEL DE UM CONTROLADOR

No modelo MVC, o `Controller` faz a ponte de ligação entre as ações do usuário na `View` com o `Model`. Na `View`, temos nosso `client/index.html` (ainda nos aprofundaremos), e no `Model` temos instâncias de `Negociacao`. Dessa forma, mantendo a separação entre `Model` e `View`, alterações na `View` (como por

exemplo, trocar de `<table>` para ``) não afetam o `Model` . Isso é interessante, pois podemos ter várias representações visuais de um mesmo modelo sem que estas forcem qualquer alteração no modelo.

Chegou o momento de criarmos nosso primeiro controlador.

3.2 A CLASSE NEGOCIACAOCONTROLLER

Vamos criar o arquivo `client/app/controllers/NegociacaoController.js` , no qual declararemos a classe do nosso controller:

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

}
```

Mais uma vez, para não correremos o risco de esquecermos de importar o script que acabamos de criar, vamos importá-lo em `client/index.html` , como o último script. Aliás, vamos remover a tag `<script>` que usamos para realizar nossos testes com a classe `Negociacao` :

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

<script src="app/domian/negociacao/Negociacao.js"></script>

<!-- REMOUEU A TAG COM NOSSOS TESTES -->

<script src="app/controllers/NegociacaoController.js"></script>

<!-- código posterior omitido -->
```

Vamos agora criar um novo método chamado `adiciona` . Ele

será chamado toda vez que o usuário submeter o formulário clicando no botão "Incluir" .

Por enquanto, vamos apenas cancelar o evento padrão de submissão do formulário (não queremos que a página recarregue) e exibir um alerta:

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

  adiciona(event) {

    // cancelando a submissão do formulário
    event.preventDefault();

    alert('Chamei ação no controller');
  }
}
```

Como faremos a ligação do método `adiciona` com a submissão do formulário? Uma prática comum entre programadores JavaScript é realizar essa associação através do próprio JavaScript, buscando o elemento do DOM e associando-o com uma função ou método através da interface de eventos.

Vamos criar o arquivo `client/app/app.js` , no qual faremos a associação citada no parágrafo anterior. Aliás, já vamos importá-lo em `client/index.html` antes de escrever qualquer linha de código:

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

<script src="app/domain/negociacao/Negociacao.js"></script>
<script src="app/controllers/NegociacaoController.js"></script>
<script src="app/app.js"></script>

<!-- código posterior omitido -->
```

Não foi engano o nome do arquivo começar com letra minúscula, uma vez que ele não declara uma classe. Entendam `app.js` como o ponto de entrada da nossa aplicação, isto é, aquele que faz a inicialização de todos os objetos necessários para nossa aplicação. No caso da nossa aplicação, só precisamos de um controller por enquanto.

Em `client/app/app.js`, criaremos uma instância de `NegociacaoController`, para em seguida buscarmos o formulário da página. Por fim, vamos associar o método `adiciona` da instância criada ao evento `submit` do formulário:

```
// client/app/app.js

// criou a instância do controller
let controller = new NegociacaoController();

// associa o evento de submissão do formulário à chamada do método
// "adiciona"

document
  .querySelector('.form')
  .addEventListener('submit', function(event) {

    controller.adiciona(event);

  });
```

Com a alteração que fizemos, basta recarregarmos a página e preenchermos o formulário. Com os dados da negociação todos preenchidos, ao clicarmos no botão "Incluir" o alerta que programamos em nosso controller será exibido.

Apesar de funcional, este código pode ser reduzido. A função `addEventListener` espera receber dois parâmetros. O primeiro é o nome do evento que estamos tratando e o segundo a função que desejamos chamar. Mas, se o método `controller.adiciona`

também é uma função e funções podem ser passadas como parâmetro para outras funções, podemos simplificar desta forma:

```
// client/app/app.js

let controller = new NegociacaoController();

// passamos diretamente controller.adiciona
document
  .querySelector('.form')
  .addEventListener('submit', controller.adiciona);
```

Tudo continua funcionando como antes. Então, a ação do controller foi chamada. Mas o nosso objetivo não é exibir a mensagem, nós queremos criar uma negociação.

3.3 ASSOCIANDO MÉTODOS DO CONTROLLER ÀS AÇÕES DO USUÁRIO

Ao clicarmos no botão Incluir do formulário, já conseguimos executar as ações do controller. Agora, precisamos capturar os dados preenchidos. Mas já fizemos isso antes, no prólogo, através de `document.querySelector()`, que recebe como parâmetro um seletor CSS que permite identificar qual elemento desejamos buscar:

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

  adiciona(event) {

    event.preventDefault();

    // buscando os elementos
    let inputData = document.querySelector('#data');
    let inputQuantidade = document.querySelector('#quantidad
e');
```

```

    let inputValor = document.querySelector('#valor');

    console.log(inputData.value);
    console.log(inputQuantidade.value);
    console.log(inputValor.value);
  }
}

```

O `document.querySelector()` é o responsável pela busca no DOM dos elementos com id `#data`, `#quantidade` e `#valor` - observe que conseguimos utilizar os seletores CSS. Aliás, um cangaceiro JavaScript tem um bom conhecimento desses seletores.

Vamos executar o código para ver se todos os dados aparecem no console do navegador. Preencheremos o formulário com a data 11/12/2016, a quantidade 1 e o valor 100.

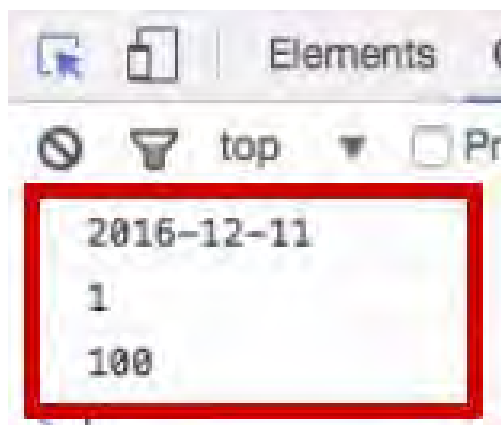


Figura 3.1: Dados do console

Os dados aparecem no console, no entanto, os valores das propriedades `_quantidade` e `_valor` são strings. Podemos verificar isso através da instrução `typeof`:

```
// client/app/controllers/NegociacaoController.js
```

```

class NegociacaoController {

  adiciona(event) {

    event.preventDefault();

    // buscando os elementos
    let inputData = document.querySelector('#data');
    let inputQuantidade = document.querySelector('#quantidad
e');
    let inputValor = document.querySelector('#valor');

    console.log(inputData.value);
    console.log(inputQuantidade.value);

    // exibe string
    console.log(typeof(inputQuantidade.value));

    console.log(inputValor.value);

    // exibe string
    console.log(typeof(inputValor.value));
  }
}

```

Isso acontece porque o retorno da propriedade `value` dos elementos dos DOM do tipo `input` retorna seus valores como string. Precisamos convertê-los para números com as funções `parseInt()` e `parseFloat()`, respectivamente:

```

// client/app/controllers/NegociacaoController.js

class NegociacaoController {

  adiciona(event) {

    event.preventDefault();

    // buscando os elementos
    let inputData = document.querySelector('#data');
    let inputQuantidade = document.querySelector('#quantidad
e');

```

```

        let inputValor = document.querySelector('#valor');

        console.log(inputData.value);
        console.log(parseInt(inputQuantidade.value));
        console.log(parseFloat(inputValor.value));
    }
}

```

Nosso formulário é pequeno, mas se tivesse mais de dez campos, teríamos de repetir a instrução `document.querySelector` dez vezes. Podemos enxugar essa sintaxe.

Vamos criar a variável `$` (uma homenagem ao jQuery!) que receberá como valor `document.querySelector`, algo totalmente possível em JavaScript, pois funções podem ser atribuídas a variáveis. Com isso, criamos um `alias`, um atalho para a instrução:

```

// client/app/controllers/NegociacaoController.js

class NegociacaoController {

    adiciona(event) {

        event.preventDefault();

        // a ideia é que $ seja o querySelector
        let $ = document.querySelector;

        let inputData = $('#data');
        let inputQuantidade = $('#quantidade');
        let inputValor = $('#valor');

        console.log(inputData.value);
        console.log(parseInt(inputQuantidade.value));
        console.log(parseFloat(inputValor.value));
    }
}

```


Desta forma, ficou menos trabalhoso escrever o código. Mas se o executarmos como está, veremos uma mensagem de erro no navegador:

```
Uncaught TypeError: Illegal invocation
    at HTMLFormElement.adiciona (NegociacaoController.js:9)
```

A atribuição do `document.querySelector` na variável `$` parece não ter funcionado. Por que não funcionou?

O `document.querySelector()` é uma função que pertence ao objeto `document` - chamaremos tal função de método. Internamente, o `querySelector` tem uma chamada para o `this`, que é o contexto no qual o método é chamado, no caso `document`.

No entanto, quando atribuímos o `document.querySelector` à variável `$`, ele passa a ser executado fora do contexto de `document`, e isto não funciona. O que devemos fazer, então? Precisamos tratar o `querySelector` como uma função separada, mas que ainda mantenha o `document` como seu contexto.

A boa notícia é que conseguimos isso facilmente através de `bind()`, uma função, por mais estranho que pareça, presente em qualquer função em JavaScript: Assim, alteraremos nosso código:

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

  adiciona(event) {

    event.preventDefault();

    // realizando o bind, $ mantém document como seu contexto
    this
    let $ = document.querySelector.bind(document);
```

```

    let inputData = $('#data');
    let inputQuantidade = $('#quantidade');
    let inputValor = $('#valor');

    console.log(inputData.value);
    console.log(parseInt(inputQuantidade.value));
    console.log(parseFloat(inputValor.value));
  }
}

```

Agora, estamos informando que o `querySelector` vai para a variável `$`, mas ainda manterá `document` como seu contexto. Desta vez, o código funcionará corretamente no navegador.

Temos "a moringa e a cartucheira" em mãos para podermos criar uma instância de negociação com os dados do formulário. No entanto, o código anterior, apesar de ninja, não é um código de cangaceiro. Podemos melhorá-lo ainda mais e faremos isso a seguir.

3.4 EVITANDO PERCORRER DESNECESSARIAMENTE O DOM

O código funciona, mas se adicionarmos dez negociações e clicarmos dez vezes em `Incluir`, o `querySelector` buscará os elementos identificados por `#data`, `#quantidade` e `#valor` dez vezes também. Contudo, devemos evitar ao máximo percorrer o DOM por questões de performance.

Nosso código está assim:

```

// client/app/controllers/NegociacaoController.js

class NegociacaoController {

```

```

adiciona(event) {

    event.preventDefault();

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

    let inputData = $('#data');
    let inputQuantidade = $('#quantidade');
    let inputValor = $('#valor');

    console.log(inputData.value);
    console.log(parseInt(inputQuantidade.value));
    console.log(parseFloat(inputValor.value));
}
}

```

Por mais que o impacto de performance buscando apenas três elementos pelo seu ID não seja tão drástico em nossa aplicação, vamos pensar como cangaceiros e aplicar boas práticas.

Para melhorar a performance, adicionaremos o `constructor` e moveremos os elementos de entrada para dentro dele. Mas em vez de criarmos variáveis, criaremos propriedades em `this`. Sendo assim, a busca pelos elementos do DOM só serão realizadas uma única vez no `constructor()`, e não mais a cada chamada do método `adiciona()`.

Como essas propriedades só fazem sentido ao serem acessadas pela própria classe `NegociacaoController`, usaremos o prefixo `_` (a convenção para propriedades privadas):

```

// client/app/controllers/NegociacaoController.js

class NegociacaoController {

    constructor() {

        let $ = document.querySelector.bind(document);
        this._inputData = $('#data');
    }
}

```

```

        this._inputQuantidade = $('#quantidade');
        this._inputValor = $('#valor');
    }

    adiciona(event) {

        event.preventDefault();

        // precisamos acessar as propriedades através de this
        console.log(this._inputData.value);
        console.log(parseInt(this._inputQuantidade.value));
        console.log(parseFloat(this._inputValor.value));
    }
}

```

Agora, mesmo que incluamos 300 negociações, os elementos do DOM serão buscados uma única vez, quando a classe `NegociacaoController` for instanciada. No entanto, **as coisas não saem como esperado.**

Depois de clicarmos no botão `Incluir` e verificarmos o console do Chrome, vemos a seguinte mensagem de erro:

```

Uncaught TypeError: Cannot read property 'value' of undefined
    at HTMLFormElement.adiciona (NegociacaoController.js:15)

```

Temos o mesmo problema quando associamos o `document.querySelector` à variável `$`, que perdeu a referência de `this` para o `document`. Vamos olhar o arquivo `client/app/app.js`:

```

// client/app/app.js

let controller = new NegociacaoController();
document
    .querySelector('.form')
    .addEventListener('submit', controller.adiciona);

```

Temos o mesmo problema aqui. Quando passamos `controller.adiciona` para o método `addEventListener`, seu

this deixou de ser o controller e passou a ser o próprio elemento do DOM, no caso, o formulário. Isso acontece porque toda função/método possui um this dinâmico que assume como valor o contexto no qual foi chamado. Mas já aprendemos a lidar com isso.

Vamos lançar mão mais uma vez da função bind :

```
// client/app/app.js

let controller = new NegociacaoController();

// bind aqui!
document
  .querySelector('.form')
  .addEventListener('submit', controller.adiciona.bind(controller));
```

Se executarmos o código, veremos que eles serão exibidos corretamente no console.

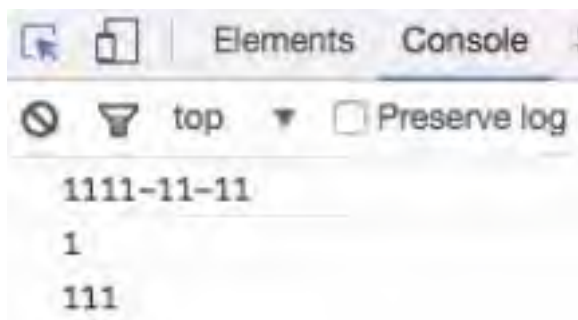


Figura 3.2: Dados no console

3.5 CRIANDO UMA INSTÂNCIA DE NEGOCIAÇÃO

Agora que o usuário pode chamar o método adiciona da

nossa classe `NegociacaoController` podemos construir uma negociação com base nos dados do formulário:

```
// client/app/controllers/NegociacaoController.js
```

```
class NegociacaoController {  
  
  // código anterior omitido  
  
  adiciona(event) {  
  
    event.preventDefault();  
  
    let negociacao = new Negociacao(  
      this._inputData.value,  
      parseInt(this._inputQuantidade.value),  
      parseFloat(this._inputValor.value)  
    );  
  
    console.log(negociacao);  
  }  
}
```

Passamos os parâmetros `data`, `quantidade` e `valor` para o constructor de `Negociacao`. Com os dados do formulário, nós já temos uma negociação. Mas será que temos mesmo?

Olhando o console do Chrome, vemos a seguinte mensagem de erro:

```
Negociacao.js:5 Uncaught TypeError: Cannot read property 'getTime'  
' of undefined  
    at new Negociacao (Negociacao.js:5)  
    at NegociacaoController.adiciona (NegociacaoController.js:15)
```

Recebemos uma mensagem de erro, mas não foi em `NegociacaoController`. O erro ocorreu na classe `Negociacao`. No console, vemos ainda em qual linha foi o problema, além de ele nos avisar que `data.getTime` não é uma função. Como isso é possível?

Vamos dar uma olhadinha na definição da nossa classe:

```
// client/app/domain/negociacao/Negociacao.js

class Negociacao {

  constructor(_data, _quantidade, _valor) {

    Object.assign(this, { _quantidade, _valor });
    this._data = new Date(_data.getTime());
    Object.freeze(this);
  }

  // código posterior omitido
```

Quer dizer então que a data que estamos passando para o constructor não é uma Date ? Que tipo de dado ela é então?

Para descobriremos o tipo da data capturada no formulário, podemos usar a função `typeof()` :

```
// client/app/controllers/NegociacaoController.js

adiciona(event) {

  event.preventDefault();

  // verificando o tipo
  console.log(typeof(this._inputData.value));

  let negociacao = new Negociacao(
    this._inputData.value,
    parseInt(this._inputQuantidade.value),
    parseFloat(this._inputValor.value)
  );

  console.log(negociacao);
}
```

Através do console, vemos que a valor da data do elemento é do tipo **string**:

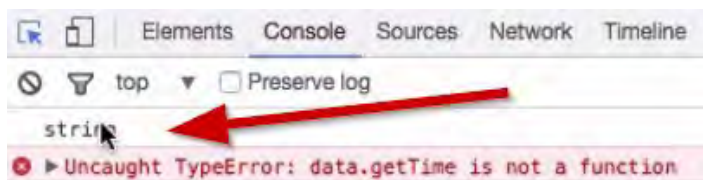


Figura 3.3: String

Encarar conversões de datas nas mais diversas linguagens requer determinação. É por isso que precisamos da nossa cartucheira e do nosso chapéu de cangaceiro a seguir.

3.6 CRIANDO UM OBJETO DATE A PARTIR DA ENTRADA DO USUÁRIO

Existem várias maneiras de construirmos uma data passando parâmetros para o constructor de `Date`. Uma das formas que já vimos é que, se já temos uma data, podemos passar o resultado do `getTime()` dela para o constructor de outra. Será que funciona se passarmos a string diretamente? Vamos tentar:

```
// client/app/controllers/NegociacaoController.js

// código anterior omitido

adiciona(event) {

    event.preventDefault();

    // o constructor está recebendo uma string. Será que funciona

    let data = new Date(this._inputData.value)
    console.log(data);
}

// código posterior omitido
```


Realizando um teste com a última alteração, não temos sucesso:

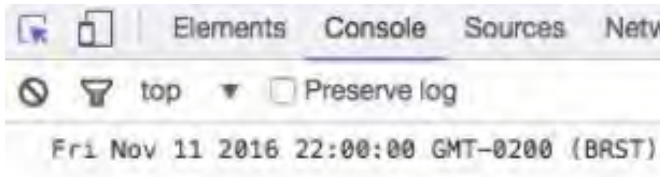


Figura 3.4: Data errada no Console

A data que apareceu no console não foi a mesma. Em vez disso, vemos que o dia é 11. Então, não funcionou. Temos de encontrar outro jeito de trabalhar com essa data.

Outra maneira de trabalharmos com `Date` é passarmos um *array* para seu construtor com os valores do ano, mês e dia, respectivamente. Se digitarmos no console:

```
new Date(['2016', '11', '12']);
```

O retorno será a data correta.

Esta é uma forma de resolvermos. Precisamos então encontrar uma maneira de transformar a string retornada por `this._inputData.value` em um array de strings. A boa notícia é que strings são objetos em JavaScript e, como todo objeto, sabem executar tarefas para nós.

É através do método `split` de uma string que podemos transformá-la em um array. No entanto, para que o método identifique cada elemento, precisamos informar um **separador***. Como a string retornada por `this._inputData.value` segue o padrão *ano-mês-dia*, utilizaremos o hífen como separador.

Que tal um teste antes no console do Chrome?

```
dataString = '2016-12-11';  
lista = dataString.split('-');  
console.log(lista); // exhibe ["2016", "12", "11"]
```

Ele nos retornou a data correta, perfeito. Então, no método adiciona da nossa classe `NegociacaoController`, faremos assim:

```
// client/app/controllers/NegociacaoController.js  
// código anterior omitido  
  
adiciona(event) {  
  
    event.preventDefault();  
  
    // agora o Date recebe um array  
    let data = new Date(this._inputData.value.split('-'));  
    console.log(data);  
}  
// código posterior omitido
```

Veja que, em uma única instrução, capturamos de `this._inputData.value`, realizamos o *split* e passamos o resultado final para `Date`. Resolvemos o problema da data! Vejamos o resultado no console!

```
Sat Nov 12 2016 00:00:00 GMT-0200 (-02)
```

Mas nossa solução não para por aqui, há outras formas de se resolver o mesmo problema. Por exemplo, quando um `Date` recebe um array, internamente ele reagrupa todos os elementos do Array em uma única string usando uma vírgula como separador.

O reagrupamento interno é feito através da função `join`, que recebe como parâmetro o separador utilizado. Podemos simular este comportamento no console:

```
dataString = '2016-12-11';
lista = dataString.split('-'); // separou os números
lista = lista.join(',') // juntou os números usando vírgula como
separador
console.log(lista);
```

No entanto, podemos em nosso código simplesmente trocar os hífens por vírgulas e passar a string resultante para o novo `Date`. Para isso, podemos pedir ajuda à função `replace`, presente em todas as strings. Passaremos como parâmetro uma expressão regular, pedindo que seja trocado o hífen de todas as ocorrências da string (ou seja, global) por vírgula: `replace(/-/g, ',')`.

```
// client/app/controllers/NegociacaoController.js
// código anterior omitido

adiciona(event) {

    event.preventDefault();

    // outra forma de resolver
    let data = new Date(this._inputData.value.replace(/-/g, ','));
;
    console.log(data);
}
```

E o resultado no console fica:

```
Sat Nov 12 2016 00:00:00 GMT-0200 (-02)
```

A data ficou correta. Vimos que existem várias formas de resolver a questão da data. Mas que tal escolhermos uma outra maneira para testar nossas habilidades de cangaceiro? É o que veremos a seguir.

3.7 UM DESAFIO COM DATAS

Há mais uma forma aceita pelo `Date` na construção de uma

nova data. No console do Chrome, escreveremos a seguinte linha:

```
data = new Date(2016, 11, 12)
console.log(data);
```

No entanto, se imprimirmos esta data, veremos o seguinte retorno:

```
Mon Dec 12 2016 00:00:00 GMT-0200 (-02)
```

A data impressa é Mon Dec 12 2016 , ou seja, 12/12/2016. Por que ele imprimiu **dezembro**, se passamos o mês 11 ? Porque nesta forma o mês deve ser passado de 0 (janeiro) a 11 (dezembro). Então, se queremos que a data seja em **novembro**, precisamos decrementar o valor do mês. Vamos fazer um novo teste no console, digitando:

```
data = new Date(2016,10,12)
console.log(data);
```

Agora, a data já aparece correta.

```
Sat Nov 12 2016 00:00:00 GMT-0200 (-02)
```

Todas as soluções que vimos anteriormente para se construir uma data foram soluções ninjas. No entanto, cangaceiros gostam de desafios, e por isso escolheremos esta última forma que vimos.

Então, voltando ao problema, como deve ser feita a conversão da string no formato 2016-11-12 , capturada de `this._inputData.value` para um `Date` , usando a forma que acabamos de aprender? É isso que veremos a seguir.

3.8 RESOLVENDO UM PROBLEMA COM O PARADIGMA FUNCIONAL

Nosso objetivo é que o `Date` receba em seu `constructor` um ano, mês e dia. Vamos alterar nosso código lançando mão da função `split()`:

```
// client/app/controllers/NegociacaoController.js
// código anterior omitido

adiciona(event) {

    event.preventDefault();

    let data = new Date(this._inputData.value.split('-'));
    console.log(data);
}

// código posterior omitido
```

Através do `split`, a string `2016-11-12` se tornará um *array* de três elementos. Sabemos que se passarmos da forma como já está, conseguiremos o resultado desejado. Porém, não queremos que o `Date` receba um *array*, queremos que ele receba cada item do array como cada parâmetro do seu `constructor`. Queremos que ele receba a primeira posição do *array* como a primeira posição do `constructor`, e que o processo se repita com o segundo e terceiro elemento do *array*.

A partir do ES2015 (ES6), podemos utilizar o **spread operator**, que permite tratar cada elemento do array como um elemento individualmente. Leia atentamente a alteração no código:

```
// client/app/controllers/NegociacaoController.js
// código anterior omitido

adiciona(event) {

    event.preventDefault();

    // usamos os famosos três pontinhos!
    let data = new Date(...this._inputData.value.split('-'));
}
```

```
    console.log(data);  
}  
  
// código posterior omitido
```

Adicionamos o `spread operator` `...` (reticências) antes do parâmetro recebido pelo `constructor` de `Date`. Com ele, no lugar de o `constructor` receber o array como único parâmetro, ele receberá cada elemento do array individualmente. Os elementos serão passados na mesma ordem para o `constructor` da classe.

Nosso código está sintaticamente correto e os três parâmetros para o `constructor` de `Date` serão passados. Porém, o mês ficará incorreto se não o ajustarmos, subtraindo `1` de seu valor. Vejamos seu resultado no console do Chrome:

```
Mon Dec 12 2016 00:00:00 GMT-0200 (-02)
```

O **spread operator** `operator` funcionou, mas ainda precisamos encontrar uma forma de, antes de reposicionarmos cada parâmetro do array para o `constructor` de `Date`, decrementarmos `1` do valor do mês. Para isto, trabalharemos com a função `map()`, bem conhecida no mundo JavaScript e que nos permitirá realizar o ajuste que necessitamos.

Primeiro, chamaremos a função `map()` sem realizar qualquer transformação, apenas para entendermos em qual lugar do nosso código ele deve entrar:

```
// client/app/controllers/NegociacaoController.js  
// código anterior omitido  
  
adiciona(event) {  
  
    event.preventDefault();
```

```

    let data = new Date(
      ...this._inputData.value
        .split('-')
        .map(function(item) {

            return item;

        })
    );
    console.log(data);
}

```

// código posterior omitido

A lógica passada para `map()` atuará em cada elemento do array, retornando um novo array no final. Mas do jeito que está, o array resultante será idêntico ao array que estamos querendo transformar. Precisamos agora aplicar a lógica que fará o ajuste do mês, subtraindo 1 de seu valor. Mas como conseguiremos identificar se estamos no primeiro, segundo ou terceiro elemento da lista? Essa informação é importante, porque queremos alterar apenas o segundo.

A boa notícia é que a função `map()` pode disponibilizar para nós a posição do elemento iterado no segundo parâmetro passado para ela:

// client/app/controllers/NegociacaoController.js
 // código anterior omitido

```

adiciona(event) {

    event.preventDefault();

    let data = new Date(...
      this._inputData.value
        .split('-')
        .map(function(item, indice) {

            // lembre-se de que arrays começam com índice 0,
            // logo, 1 é o segundo elemento!

```

```

        if(indice == 1) {
            // o resultado aqui será um número, conversão imp
lícita
            return item - 1;
        }
        return item;
    })
);
console.log(data);
}

// código posterior omitido

```

Quando `indice` for `1`, significa que estamos iterando no segundo elemento, aquele que contém o valor do mês e que precisa ser ajustado. Na condição `if`, realizamos facilmente esse ajuste. No entanto, sabemos que os elementos do array que estamos iterando são do tipo `string`, sendo assim, qual a razão da operação de subtração funcionar?

Quando o interpretador do JavaScript detecta operações de multiplicação, divisão e subtração envolvendo uma `string` e um número, por debaixo dos panos tentará converter implicitamente a `string` em um número. Caso a `string` não represente um número, o resultado será `NaN` (Not a Number).

Por fim, o resultado bem-sucedido da operação resultará em um novo número. Operações de soma não são levadas em consideração na conversão implícita, porque o interpretador não terá como saber se deve concatenar ou converter o número.

Veremos se o código vai funcionar. No formulário, preencheremos o campo da data com `12/11/2016`. Desta vez, a data que aparecerá no console estará correta.

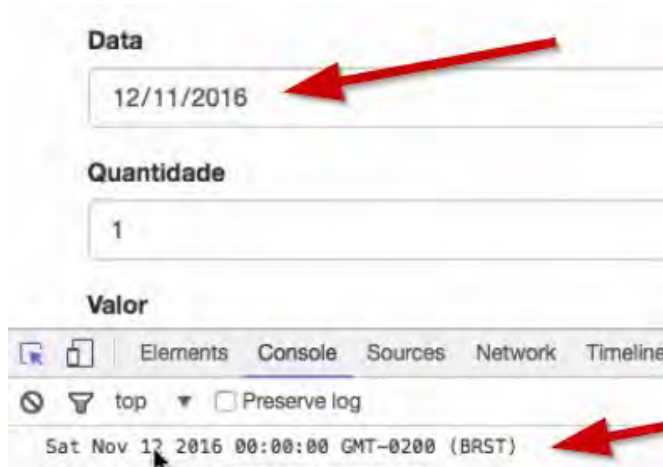


Figura 3.5: Datas correspondentes

Nosso código funciona perfeitamente e conseguimos resolver o problema. Porém, cangaceiros falam pouco, são homens de ação. Será que podemos escrever menos código e conseguir o mesmo resultado? Sim, através do **módulo do índice**.

Recordar é viver

Na matemática, módulo é o resto da divisão. No console do Chrome, vamos realizar um teste:

```
2 % 2 // o módulo é 0, pois este é o resto da divisão de 2 por 2
```

Vejamos outro exemplo:

```
3 % 2 // o módulo será 1, pois é o resto da divisão de 3 por 2
```

Os índices do nosso array são `0`, `1` e `2`, respectivamente. Sendo assim, se formos calcular o módulo de cada um deles por dois, temos os módulos:

```
0 % 2 // módulo é 0
1 % 2 // módulo é 1
2 % 2 // módulo é 0
```

Veja que é justamente índice do mês que possui o resultado 1, justamente o valor que precisamos subtrair. Por fim, nosso código ficará assim:

```
// código anterior omitido

adiciona(event) {

    event.preventDefault();

    let data = new Date(...this._inputData
        .value.split('-')
        .map(function(item, indice) {
            return item - indice % 2;
        }));

    console.log(data);
}

// código posterior omitido
```

Se estamos na primeira posição do array, o valor de `indice` é `0`. Por isso, o resultado de `indice % 2` será igual a `0` também. Se subtrairmos este valor de `item`, nada mudará. Mas quando estivermos na segunda posição do array, o `indice` será igual a `1`. Agora, quando calcularmos `1` módulo de `2`, o resultado será `1`. E quando estivermos na terceira posição do array, `2` módulo de `2`, também será igual a `0`. Não diminuiremos nada do valor do item. Dessa forma conseguimos evitar a criação de um `if`:

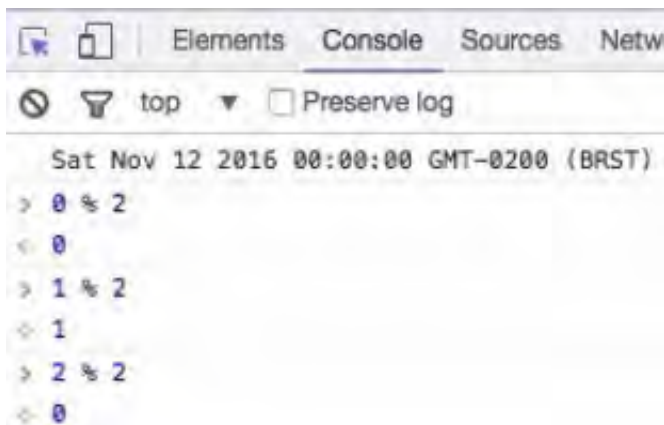


Figura 3.6: Valores dos módulos

Até aqui, o nosso código ficou assim:

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

  constructor() {

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

  }

  adiciona(event) {

    event.preventDefault();

    let data = new Date(...this._inputData
      .value.split('-')
      .map(function(item, indice) {
        return item - indice % 2;
      }));

    console.log(data);
```

```
}  
}
```

Se ele for executado, veremos que a data que surgirá no console será 12 de novembro de 2016.

3.9 ARROW FUNCTIONS: DEIXANDO O CÓDIGO AINDA MENOS VERBOSO

Podemos ainda deixar o nosso código menos verboso, usando uma forma diferente de se declarar funções, usada a partir da versão ES2015 (ES6). Falamos das *arrow functions*.

O termo *arrow* traduzido para o português significa **flecha**. Com estas "funções flecha", podemos eliminar a palavra `function` do código. Mas se simplesmente apagarmos a palavra e tentarmos executar o código, teremos um erro de sintaxe, que será apontado no navegador. Para emitirmos o termo, teremos de adicionar `=>` (olha a flecha aqui!). Ela sinalizará que temos uma *arrow function* que receberá dois parâmetros.

```
// client/app/controllers/NegociacaoController.js  
// código anterior omitido
```

```
adiciona(event) {  
  
    event.preventDefault();  
  
    let data = new Date(...  
        this._inputData.value  
        .split('-')  
        .map((item, indice) => {  
            return item - indice % 2;  
        })  
    );  
  
    console.log(data);  
}
```

```
}
```

```
// código posterior omitido
```

Quando usamos a sintaxe da flecha, já sabemos que se trata de uma *arrow function*. Se atualizar a página no navegador, veremos que tudo funciona.

Agora, uma pergunta: no bloco da *arrow functions*, quantas instruções nós temos? Apenas **uma**: `return item - indice % 2`. A linha está dentro da função `map()` :

```
// analisando um trecho do código
```

```
.map((item, indice) => {  
    return item - indice % 2;  
});
```

Quando nossa arrow function possui duas ou mais instruções, somos obrigados a utilizar o bloco `{}` , inclusive a adicionar um `return` . No entanto, como nosso código possui apenas uma instrução, é permitido remover o bloco `{}` , inclusive a instrução `return` . Isso pois ela é adicionada implicitamente pela arrow function. Nosso código simplificado fica assim:

```
// client/app/controllers/NegociacaoController.js  
// código anterior omitido
```

```
adiciona(event) {  
  
    event.preventDefault();  
  
    let data = new Date(...  
        this._inputData.value  
        .split('-')  
        .map((item, indice) => item - indice % 2)  
    );  
  
    console.log(data);  
}
```

```
// código posterior omitido
```

Vejam como o código ficou mais simples!

Se rodarmos o código no navegador, o formulário está funcionando normalmente, e a data aparecerá como desejamos no console. Nós conseguimos reduzir a "verbosidade" do código usando uma *arrow function*. Agora, só nos resta criar uma instância de `Negociacao` com os dados do formulário:

```
// client/app/controllers/NegociacaoController.js
// código anterior omitido
```

```
adiciona(event) {

    event.preventDefault();

    let data = new Date(...
        this._inputData.value
        .split('-')
        .map((item, indice) => item - indice % 2)
    );

    let negociacao = new Negociacao(
        data,
        parseInt(this._inputQuantidade.value),
        parseFloat(this._inputValor.value)
    );

    console.log(negociacao);
}
// código posterior omitido
```

Nossa saga em lidar com datas ainda não terminou. Somos capazes de criar um novo objeto `Date` a partir dos dados do formulário, mas como faremos se quisermos exibir um `Date` para o usuário no formato `dia/mês/ano` ? Preparados para o próximo capítulo?

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/03>

DOIS PESOS, DUAS MEDIDAS?

"Conto ao senhor é o que eu sei e o senhor não sabe; mas principal quero contar é o que eu não sei se sei, e que pode ser que o senhor saiba." - Grande Sertão: Veredas

Já somos capazes de instanciar uma negociação com base nos dados do formulário. Em breve, precisaremos exibir seus dados dentro da `<table>` de `client/index.html`. Mas será que a exibição da data será amigável?

Vamos alterar o método `adiciona()` de `NegociacaoController` para que seja impressa a data da negociação criada em vez do objeto inteiro:

```
// client/src/app/controllers/NegociacaoController.js
// código anterior omitido

adiciona(event) {

    // código anterior omitido

    // acessando o getter da data exibindo-a no console
```



```
    console.log(negociacao.data);  
}  
  
// código posterior omitido
```

Agora, para a data 12/11/2016 informada, sua apresentação no console fica neste formato:

```
Sat Nov 12 2016 00:00:00 GMT-0200 (BRST)
```

Apesar de ser uma informação válida, ficaria muito mais simples e direto exibirmos para o usuário a data no formato dia/mês/ano . É um problema que precisamos resolver, pois muito em breve incluiremos negociações em nossa tabela, e uma das colunas se refere à propriedade data de uma negociação. Faremos um experimento.

Ainda no método adiciona() , vamos imprimir a data desta maneira:

```
// client/src/app/controllers/NegociacaoController.js  
// código anterior omitido  
  
adiciona(event) {  
  
    // código anterior omitido  
  
    let diaMesAno = negociacao.data.getDate()  
    + '/' + negociacao.data.getMonth()  
    + '/' + negociacao.data.getFullYear();  
  
    console.log(diaMesAno);  
}  
  
// código posterior omitido
```

Com o getDate() retornaremos o dia e depois, com o getMonth() , retornaremos o mês e, com getFullYear() , o ano. Todos foram concatenados com a data.

Mas a data que aparecerá no console ainda não será a correta:

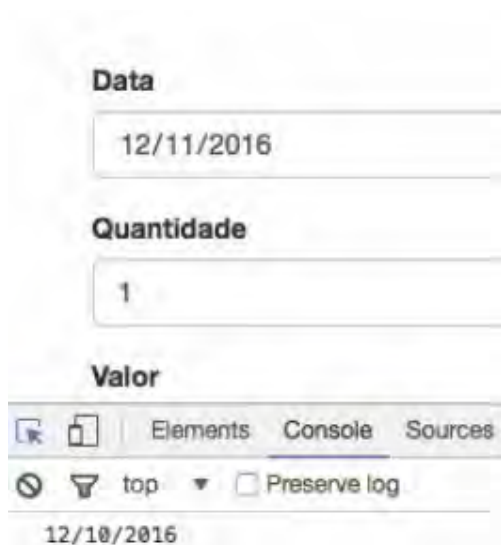


Figura 4.1: Data errada no console

No console, vemos o mês 10 . Isso acontece porque ele veio de um *array* que vai de 0 a 11 . Então, se a data gravada for no mês 11 , ele será impresso no mês 10 .

Uma tentativa para solucionarmos o problema é somando +1 ao retorno de `getMonth()` :

```
// client/src/app/controllers/NegociacaoController.js
// código anterior omitido

adiciona(event) {

    // código anterior omitido

    let diaMesAno = negociacao.data.getDate()
    + '/' + negociacao.data.getMonth() + 1
    + '/' + negociacao.data.getFullYear();
```

```

    console.log(diaMesAno);
}

// código posterior omitido

```

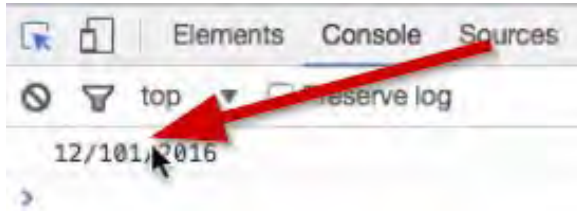


Figura 4.2: Data esquisita

Contudo, o resultado ainda não é o esperado e ainda exibe o mês como "101" , uma string! Aliás, esse é um problema clássico de concatenação com que todo programador JavaScript se depara quando está começando seu primeiro contato com a linguagem.

Para resolvermos o problema da precedência das concatenações, basta adicionarmos parênteses na operação que queremos que seja processada primeiro, antes de qualquer concatenação:

```

// client/src/app/controllers/NegociacaoController.js
// código anterior omitido

adiciona(event) {

    // código anterior omitido

    let diaMesAno = negociacao.data.getDate()
    + '/' + (negociacao.data.getMonth() + 1)
    + '/' + negociacao.data.getFullYear();

    console.log(diaMesAno);
}

// código posterior omitido

```

Básico de JavaScript. Um teste demonstra o texto correto de exibição da data. Podemos tornar ainda melhor nosso código, principalmente quando o assunto é lidar com datas.

4.1 ISOLANDO A RESPONSABILIDADE DE CONVERSÃO DE DATAS

Em poucas linhas, garantimos a conversão de uma string em objeto `Date`, inclusive o caminho inverso, um `Date` em uma string no formato `dia/mês/ano`. No entanto, em todo lugar da nossa aplicação que precisarmos realizar essas operações, teremos de repetir um código que já escrevemos.

Vamos criar o novo arquivo `client/app/ui/converters/DateConverter.js`. Como sempre, já vamos importar o arquivo em `client/index.html` para não correremos o risco de esquecermos de importá-lo:

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

<script src="app/domain/negociacao/Negociacao.js"></script>
<script src="app/controllers/NegociacaoController.js"></script>
<script src="app/app.js"></script>
<script src="app/ui/converters/DateConverter.js"></script>

<!-- código posterior omitido -->
```

No arquivo que acabamos de criar, vamos declarar a classe `DateConverter` com o esqueleto de dois métodos que realizarão a conversão:

```
// client/app/ui/converters/DateConverter.js

class DateConverter {
```

```

    paraTexto(data) {

    }

    paraData(texto) {

    }
}

```

Teremos pouco esforço para implementá-lo, pois podemos copiar a lógica de conversão espalhada pelo método `adiciona` de `NegociacaoController`, só faltando adequar o código ao nome dos parâmetros recebidos pelos métodos:

```

// client/app/ui/converters/DateConverter.js

class DateConverter {

    paraTexto(data) {

        return data.getDate()
            + '/' + (data.getMonth() + 1)
            + '/' + data.getFullYear();
    }

    paraData(texto) {

        return new Date(...texto.split('-').map((item, indice) =>
            item - indice % 2));
    }
}

```

Agora, precisamos alterar a classe `NegociacaoController`. No seu método `adiciona()`, usaremos a nova classe que acabamos de criar:

```

// client/app/controllers/NegociacaoController.js
// código anterior omitido

adiciona(event) {

```

```
event.preventDefault();

let converter = new DateConverter();

let data = converter.paraData(this._inputData.value);

let negociacao = new Negociacao(
    data,
    parseInt(this._inputQuantidade.value),
    parseFloat(this._inputValor.value)
);

let diaMesAno = converter.paraTexto(negociacao.data);

console.log(diaMesAno);
}

// código posterior omitido
```

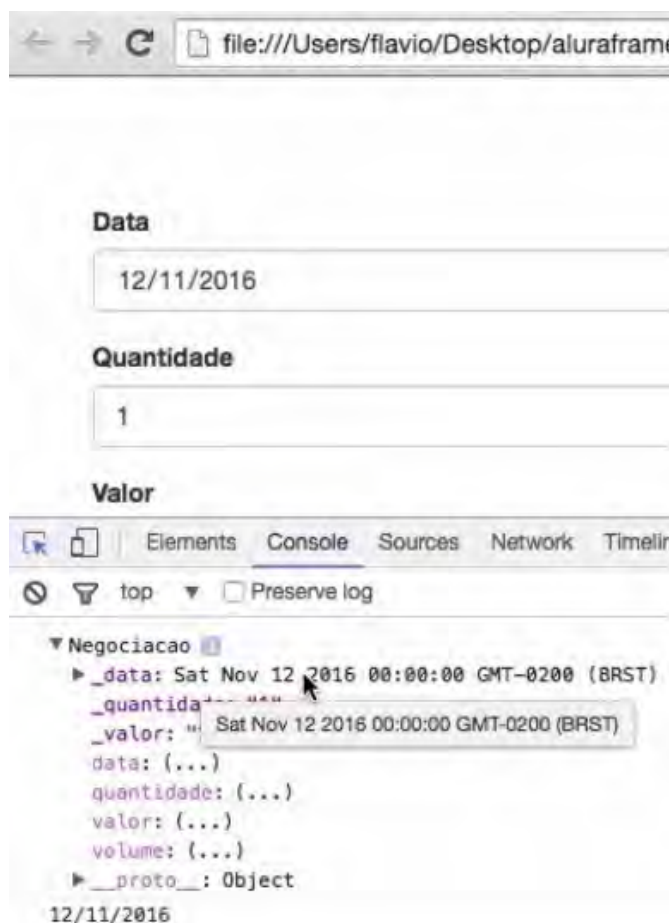


Figura 4.3: Negociação no controller

A data também é impressa corretamente nos dois formatos. No entanto, talvez vocês possam estar se perguntando como o operador `new` funcionou com `DateConverter`, uma vez que a classe não possui um `constructor`. Não aconteceu nenhum erro porque classes que não definem um `constructor` ganham um padrão.

Nós conseguimos isolar o código dentro do `Helper` . Mas será que podemos melhorá-lo ainda mais?

4.2 MÉTODOS ESTÁTICOS

Aprendemos que instâncias de uma classe podem ter propriedades e que cada instância possui seu próprio valor. Mas se procurarmos por propriedades de instância na classe `DateConverter` , não encontraremos nenhuma. Sendo assim, estamos criando uma instância de `DateConverter` apenas para podermos chamar seus métodos que não dependem de nenhum dado de instância.

Quando temos métodos que não fazem sentido serem chamados de uma instância, como no caso do método `paraTexto()` que criamos, podemos chamá-los diretamente da classe na qual foram declarados, adicionando o modificador `static` antes do nome do método:

```
// client/app/ui/converters/DateConverter.js

class DateConverter {

  static paraTexto(data) {

    return data.getDate()
      + '/' + (data.getMonth() + 1)
      + '/' + data.getFullYear();
  }

  static paraData(texto) {

    return new Date(...texto.split('-').map((item, indice) =>
item - indice % 2));
  }
}
```


Agora, podemos chamar os métodos diretamente da classe.
Alterando o código de `NegociacaoController` :

```
// client/app/controllers/NegociacaoController.js
// código anterior omitido

adiciona(event) {

    event.preventDefault();

    // CHAMANDO O MÉTODO ESTÁTICO

    let negociacao = new Negociacao(
        DateConverter.paraData(this._inputData.value),
        parseInt(this._inputQuantidade.value),
        parseFloat(this._inputValor.value)
    );

    console.log(negociacao.data);

    // CHAMANDO O MÉTODO ESTÁTICO

    let diaMesAno = DateConverter.paraTexto(negociacao.data);
    console.log(diaMesAno);
}

// código posterior omitido
```

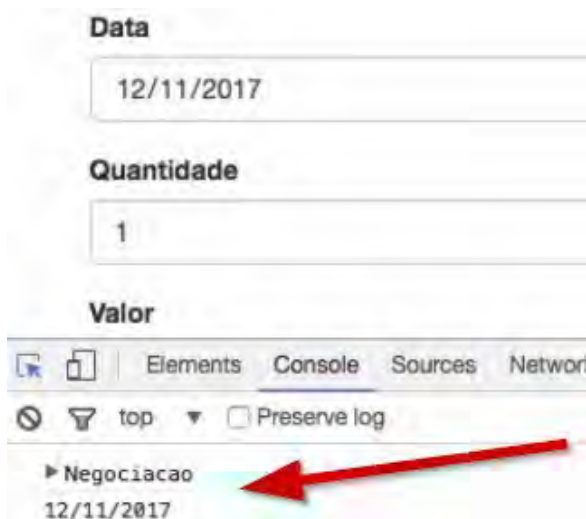


Figura 4.4: Negociação no console

Vimos uma novidade em termos de Orientação a Objeto: a classe `DateConverter` tem métodos estáticos, o que torna desnecessária a criação de uma instância.

Apesar de funcional, nada impede o programador desavisado de criar uma instância de `DateConverter` e, a partir dessa instância, tentar acessar algum método estático. Isso resultaria no erro de método ou função não definida.

A boa notícia é que podemos pelo menos avisar ao desenvolvedor que determinada classe não deve ser instanciada escrevendo um construtor padrão que lance uma exceção com uma mensagem clara de que ele não deveria fazer isso:

```
// client/app/ui/converters/DateConverter.js
```

```
class DateConverter {
```

```

constructor() {
    throw new Error('Esta classe não pode ser instanciada');
}

```

// código posterior omitido

No próprio console do Chrome podemos criar uma instância de `DateConverter` e vermos o resultado. No console, digitaremos:

```

// dá erro!
x = new DateConverter()

```

Veremos o seguinte retorno:

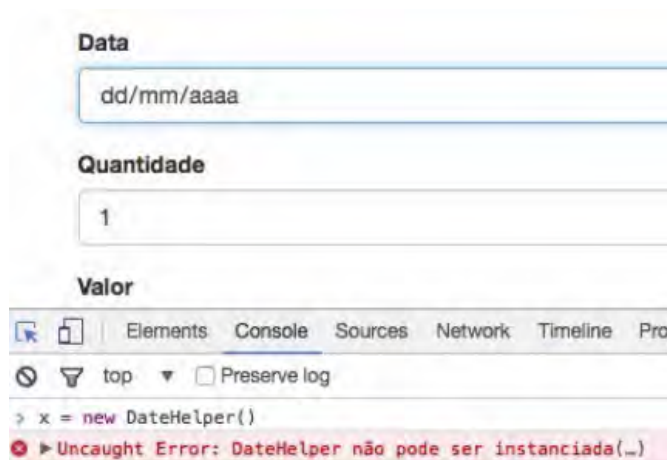


Figura 4.5: Mensagem de erro

Ao ver esta mensagem, o programador saberá que trabalhamos com métodos estáticos. Se clicarmos no erro, veremos qual é a classe e onde está o problema.

Ainda podemos melhorar um pouco mais a legibilidade da nossa classe com um recurso simples, porém poderoso, que

veremos a seguir.

4.3 TEMPLATE LITERAL

Vamos voltar ao arquivo `DateConverter.js` e analisar o método `paraTexto`. Nele, realizamos concatenações envolvendo *strings* e números:

```
// client/app/ui/converters/DateConverter.js

// código anterior omitido

static paraTexto(data) {

    return data.getDate()
        + '/' + (data.getMonth() + 1)
        + '/' + data.getFullYear();
}

// código posterior omitido
```

Inclusive tivemos de envolver a expressão `(data.getMonth() + 1)` entre parênteses, para que ela tivesse precedência. No entanto, a partir do ES2015 (ES6), foi introduzida na linguagem **template literal** que nada mais são do que uma forma diferente de se declarar strings e que evita o processo de concatenação.

Através do console, vamos relembrar a concatenação clássica. Primeiramente, digitaremos nele:

```
nome = 'Flávio';
idade = 18;
console.log('A idade de ' + nome + ' é ' + idade + '.');
```

O primeiro passo é alterarmos a linha do nosso último `console.log`, que ficará assim:

```
nome = 'Flávio';
```

```
idade = 18;
console.log(`A idade de nome é idade.`)
```

Observe que a string foi declarada entre ``` (backtick), mas se executarmos o código desta forma, será exibida a frase: A idade de nome é 18. . Ele não entendeu que o conteúdo da variável `nome` deve ser considerado na template literal. Para que isso aconteça, precisamos colocar todas as variáveis que desejamos dentro da expressão `${}` :

```
nome = 'Flávio';
idade = 18;
console.log(`A idade de ${nome} é ${idade}.`)
```

Com o uso de `${}` dentro da template literal, ele fará o mecanismo de **interpolação**. A expressão vai interpolar o conteúdo das variáveis `nome` e `idade` na string. Essa forma é menos sujeita a erro do que a anterior, que continha várias concatenações, principalmente se a quantidade de variáveis envolvidas fosse ainda maior.

Agora, podemos transpor o conhecimento aqui adquirido para nossa classe `DateConverter` . Seu método `paraTexto` ficará assim:

```
// client/app/ui/converters/DateConverter.js

class DateConverter {

  constructor() {

    throw new Error('Esta classe não pode ser instanciada');
  }

  static paraTexto(data) {

    return `${data.getDate()}/${data.getMonth()+1}/${data.getFullYear()}`;
  }
}
```

```
}
```

```
// código posterior omitido
```

Desta vez, nem foi necessário adicionar parênteses, porque cada expressão será avaliada individualmente primeiro, para logo em seguida seu resultado ser interpolado com a string.

A terminologia "Template Literal" substitui o termo "Template String" no final da implementação da especificação do ECMAScript 2015 (ES6).

Este foi o nosso primeiro contato com o template literal. Veremos que este é um recurso muito poderoso ao longo do nosso projeto, colocando-o ainda mais à prova. Porém, ainda não acabou.

4.4 A BOA PRÁTICA DO FAIL-FAST

Nossa classe `DateConverter` está cada vez melhor. Entretanto, o que acontecerá se passarmos para seu método `paraData` a string `11/12/2017`, que não segue o padrão com hífen estipulado? Vamos testar no console:

```
// testando no console
date = DateConverter.paraData('11/12/2017')
```

Será impresso:

```
Invalid Date
```

Como o método internamente depende da estrutura que

recebe uma data no formato `dia/mês/ano` , trocar o separador fará com que o `split` usado internamente por `paraData` retorne um valor inválido para o constructor de `Date` . Demos sorte, já pensou se ele devolvesse uma `Data` , mas diferente da qual desejamos?

Podemos usar a boa prática do **fail-fast**, em que validamos os parâmetros recebidos pelo método e já falhamos antecipadamente antes que eles sejam usados pela lógica do programa. Mas claro, fornecendo uma mensagem de erro clara para o desenvolvedor sobre o problema ocorrido.

Vamos apelar um pouquinho para as **expressões regulares** para verificar se o texto recebido pelo método `paraData` segue o padrão `dia/mês/ano` . A expressão que usaremos é a seguinte:

```
/ a expressão que utilizaremos
^\d{4}-\d{2}-\d{2}$
```

Nela, estamos interessados em qualquer texto que comece com quatro dígitos, seguido de um hífen e de um número com dois dígitos. Por fim, ela verifica se o texto termina com um número de dois dígitos.

```
// client/app/ui/converters/DateConverter.js

class DateConverter {

  // código anterior omitido

  static paraData(texto) {

    if(!/^\d{4}-\d{2}-\d{2}$/.test(texto)) {
      throw new Error('Deve estar no formato aaaa-mm-dd');
    }

    return new Date(...texto.split('-').map((item, indice) =>
```

```
item - indice % 2));  
}
```

// código posterior omitido

A linha com o `throw new` só será executada se o `if` for `false`. Por isso, usamos o sinal de `!`. Ainda podemos nos livrar do bloco do `if` desta forma:

// client/app/ui/converters/DateConverter.js

```
class DateConverter {  
  
  // código anterior omitido  
  
  static paraData(texto) {  
  
    if(!/^d{4}-d{2}-d{2}$/.test(texto))  
      throw new Error('Deve estar no formato aaaa-mm-dd');  
  
    return new Date(...texto.split('-').map((item, indice) =>  
item - indice % 2));  
  }  
}
```

// código posterior omitido

Apesar de o `if` não ter mais um bloco, o interpretador do JavaScript entende que a próxima instrução imediatamente após `if` será executada apenas se a condição for `true`. As demais instruções não são consideradas como fazendo parte do bloco do `if`. É por isso que a instrução `throw new ...` foi indentada, para dar uma pista visual para o programador de que ela pertence ao `if`.

Depois de recarregarmos nossa página, podemos realizar testes no console. Primeiro, com uma string válida:

```
DateConverter.paraData('2017-11-12')  
Sun Nov 12 2017 00:00:00 GMT-0200 (BRST)
```


A data exibida está correta. Agora, forçaremos o erro no console para vermos o que acontece.

```
DateConverter.paraData('2017/11/12')  
Uncaught Error: Deve estar no formato aaaa-mm-dd(..)
```

Perfeito, mas o que acontecerá se, no campo da data, digitarmos um ano com mais de quatro dígitos? A mensagem de erro continuará sendo impressa no console. Mais tarde, aprenderemos a lidar com este tipo de erro, exibindo uma mensagem mais amigável para o usuário. Por hora, o código que escrevemos é suficiente para podermos avançar com nossa aplicação.

Agora que já lidamos com as conversões de data, podemos atacar outro requisito da aplicação, a lista de negociações.

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/04>

O BANDO DEVE SEGUIR UMA REGRA

"Natureza da gente não cabe em nenhuma certeza." - Grande Sertão: Veredas

Todas as negociações cadastradas pelo usuário precisam ser armazenadas em uma lista que só deve permitir inclusões e nenhum outro tipo de operação além de leitura. Essa restrição não nos permite usar um simples array do JavaScript, devido à ampla gama de operações que podemos realizar com ele. No entanto, o array é uma excelente estrutura de dados para trabalharmos. E agora?

Uma solução é criarmos uma classe que encapsule o array de negociações. O acesso ao array só poderá ser feito através dos métodos dessa classe, pois estes implementam as regras que queremos seguir. Em suma, criaremos um **modelo de negociações**.

5.1 CRIANDO UM NOVO MODELO

Vamos criar o arquivo `client/app/domain/negociacao/Negociacoes.js`. A nova classe criada terá como propriedade um array de negociações sem qualquer item:

```
// client/app/domain/negociacao/Negociacoes.js
```

```
class Negociacoes {  
  
  constructor() {  
  
    this._negociacoes = [];  
  }  
  
  adiciona(negociacao) {  
  
    this._negociacoes.push(negociacao);  
  }  
}
```

Observe que usamos o prefixo `_` para indicar que a lista só pode ser acessada pelos métodos da própria classe. Devido a essa restrição, adicionamos o método `adiciona`, que receberá uma negociação. Precisamos também de um método que nos permita ler a lista de negociações.

Vamos criar o método `paraArray` que, ao ser chamado em instâncias de `Negociacoes`, retornará o array interno encapsulado pela classe:

```
class Negociacoes {  
  
  constructor() {  
  
    this._negociacoes = [];  
  }  
  
  adiciona(negociacao) {
```

```

        this._negociacoes.push(negociacao);
    }

    paraArray() {

        return this._negociacoes;
    }
}

```

Depois, no `index.html`, temos de importar a nossa classe.

```

<!-- código anterior omitido -->

<script src="app/domain/negociacao/Negociacao.js"></script>
<script src="app/controllers/NegociacaoController.js"></script>
<script src="app/app.js"></script>
<script src="app/ui/converters/DateConverter.js"></script>

<!-- novo script aqui! -->

<script src="app/domain/negociacao/Negociacoes.js"></script>

<!-- código posterior omitido -->

```

De volta ao `NegociacaoController`, adicionaremos a propriedade `negociacoes` que receberá uma instância de `Negociacoes`:

```

// client/app/controllers/NegociacaoController.js

class NegociacaoController {

    constructor() {

        let $ = document.querySelector.bind(document);
        this._inputData = $('#data');
        this._inputQuantidade = $('#quantidade');
        this._inputValor = $('#valor');
        this._negociacoes = new Negociacoes();
    }
}
// código posterior omitido

```

Agora, no método `adiciona` do nosso controller, vamos

adicionar a negociação criada com os dados do formulário em nossa lista através do método `this._negociacoes.adiciona()`. Vamos aproveitar e remover o código que exibia a data da negociação formatada, pois ele não é mais necessário:

```
// client/app/controllers/NegociacaoController.js

// código posterior omitido
adiciona(event) {

    event.preventDefault();

    let negociacao = new Negociacao(
        DateConverter.paraData(this._inputData.value),
        parseInt(this._inputQuantidade.value),
        parseFloat(this._inputValor.value)
    );

    // inclui a negociação
    this._negociacoes.adiciona(negociacao);

    // imprime a lista com o novo elemento
    console.log(this._negociacoes.toArray());
}
// código posterior omitido
```

Pfeito, tudo no lugar, já podemos testar.

5.2 O TENDÃO DE AQUILES DO JAVASCRIPT

A cada inclusão de uma nova negociação, exibiremos no console o estado atual da lista encapsulada por `Negociacoes`. Contudo, essa nova solução parece não ter funcionado, pois qualquer teste realizado exibirá no console a mensagem:

```
Uncaught ReferenceError: Negociacoes is not defined
    at new NegociacaoController (NegociacaoController.js:9)
    at app.js:1
```

O problema é que, em `app.js`, estamos criando uma instância de `NegociacaoController`. Esta classe, por sua vez, depende da classe `Negociacoes` que ainda não foi carregada! Faz sentido, pois se olharmos a ordem de importação de scripts em `client/index.html`, a importação de `Negociacoes.js` foi feita após `app.js`:

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

<script src="app/domain/negociacao/Negociacao.js"></script>
<script src="app/controllers/NegociacaoController.js"></script>
<script src="app/app.js"></script>
<script src="app/ui/converters/DateConverter.js"></script>

<!-- precisa vir antes de app.js -->
<script src="app/domain/negociacao/Negociacoes.js"></script>

<!-- código posterior omitido -->
```

Essa dependência entre scripts é um dos tendões de Aquiles do JavaScript. Isso faz com que o desenvolvedor assuma a responsabilidade de importar os scripts de sua aplicação na ordem correta. Aprenderemos a solucionar esse problema através dos módulos do ES2015 (ES6), mas ainda não é a hora de abordarmos esse assunto, pois temos muita coisa a aprender antes. Por enquanto, vamos nos preocupar com a ordem de importação dos scripts.

Vamos ajustar o `client/index.html`, movendo `app.js` como último script a ser importado:

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

<script src="app/domain/negociacao/Negociacao.js"></script>
<script src="app/controllers/NegociacaoController.js"></script>
<script src="app/ui/converters/DateConverter.js"></script>
```

```

<script src="app/domain/negociacao/Negociacoes.js"></script>
<!-- app.js agora sempre será o último script porque ele é o resp
onsável em instanciar NegociacaoController que por conseguinte de
pende de outras classes -->

<script src="app/app.js"></script>
<!-- código posterior omitido -->

```

Agora nosso teste exibe a lista no console assim que cadastramos uma nova negociação:

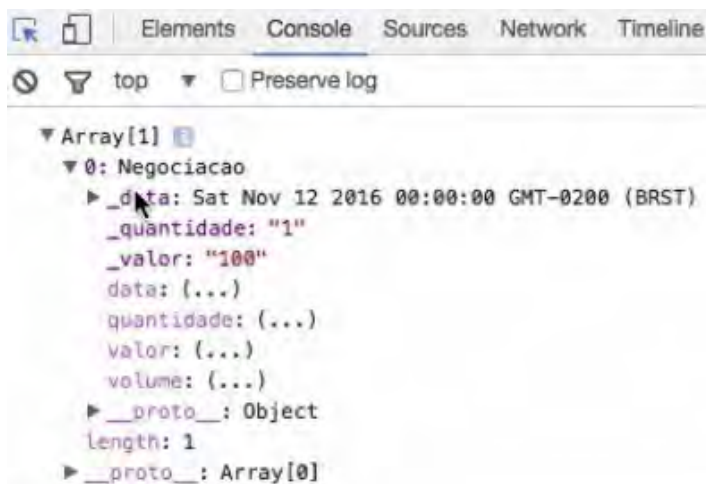


Figura 5.1: Negociação no console

Tudo funciona, mas os dados do formulário continuam preenchidos depois do cadastro. É uma boa prática limparmos o formulário preparando-o para a próxima inclusão.

Vamos criar o método privado `_limpaFormulario()`, que conterá a lógica que limpará nosso formulário.

```

// client/app/controllers/NegociacaoController.js

class NegociacaoController {

```

```

// código anterior omitido

_limpaFormulario() {

    this._inputData.value = '';
    this._inputQuantidade.value = 1;
    this._inputValor.value = 0.0
    this._inputData.focus();
}
}

```

Agora, vamos chamar o método depois de adicionarmos a negociação na lista:

```

// client/app/controllers/NegociacaoController.js

class NegociacaoController {

    // código anterior omitido

    adiciona(event) {

        // código anterior omitido

        // limpando o formulário
        this._limpaFormulario();
    }

    // código posterior omitido
}

```

Podemos incluir uma nova negociação, que o console exibirá corretamente a lista com seus dois itens:

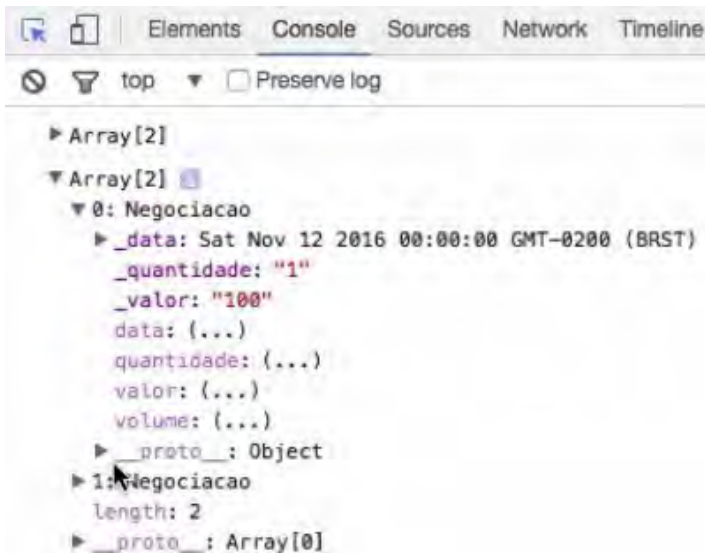


Figura 5.2: Exibições no array

Nosso código funciona, mas podemos torná-lo ainda melhor. Que tal isolarmos o código que instancia uma negociação com base nos dados do formulário em um método privado da classe? Com ele, deixaremos clara nossa intenção de criar uma nova negociação, já que seu nome deixa isso bastante explícito.

Nossa classe `NegociacaoController` no final ficará assim:

```
// client/app/controller/NegociacaoController.js

class NegociacaoController {

  constructor() {

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

```

    }

    adiciona(event) {

        event.preventDefault();
        this._negociacoes.adiciona(this._criaNegociacao()); // modificação!
        console.log(this._negociacoes.toArray());
        this._limpaFormulario();
    }

    _limpaFormulario() {

        this._inputData.value = '';
        this._inputQuantidade.value = 1;
        this._inputValor.value = 0.0;
        this._inputData.focus();
    }

    _criaNegociacao() {

        // retorna uma instância de negociação
        return new Negociacao(
            DateConverter.paraData(this._inputData.value),
            parseInt(this._inputQuantidade.value),
            parseFloat(this._inputValor.value)
        );
    }
}

```

Depois de concluída as alterações, um novo teste exibe as informações esperadas no console:

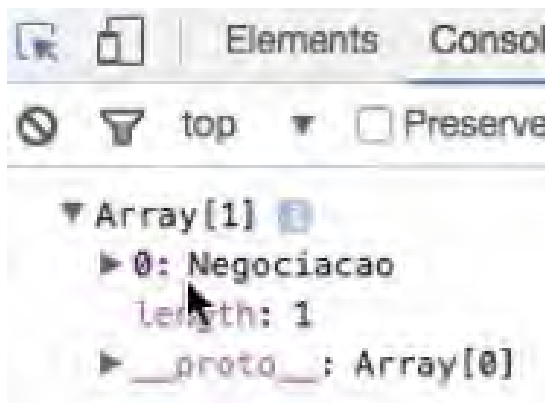


Figura 5.3: Negociações como esperado

Tudo está funcionando corretamente. Mas será que instâncias de `Negociacoes` realmente só permitem adicionar novas negociações e que nenhuma outra operação é permitida? Faremos um teste a seguir para descobrirmos.

5.3 BLINDANDO O NOSSO MODELO

Para sabermos se as negociações encapsuladas por `Negociacao` são imutáveis, podemos realizar um rápido teste, apagando todos os itens da lista. Para isso, vamos acessar o array retornado pelo método `paraArray()` de `Negociacoes`, atribuindo `0` à propriedade `length` que todo array possui. Isso será suficiente para esvaziá-la:

```
// client/app/controller/NegociacaoController.js
// código anterior omitido

adiciona(event) {

    event.preventDefault();
    this._negociacoes.adiciona(this._criaNegociacao());
```

```

// será que conseguimos apagar a lista?
this._negociacoes.toArray().length = 0;

// qual o resultado?
console.log(this._negociacoes.toArray());
this._limpaFormulario();
}

// código posterior omitido

```

Executando nosso teste:

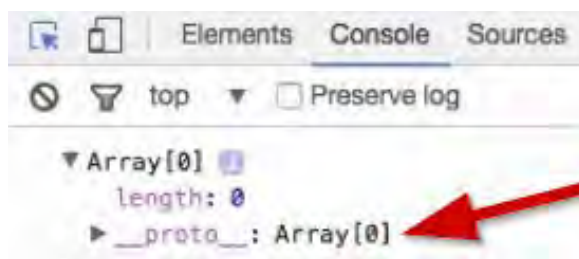


Figura 5.4: Array vazio

Conseguem identificar o problema? Qualquer trecho do nosso código que acessar o array de negociações devolvido através do método `toArray` terá uma referência para o mesmo array encapsulado de `Negociacoes`. Já vimos esse problema antes, quando trabalhamos com datas. Inclusive, aplicamos uma programação defensiva, uma estratégia que cai muito bem nesta situação.

A ideia é retornarmos um novo array, ou seja, uma nova referência com base nos itens do array de negociações encapsulado pela classe `Negociacoes`. Criaremos um novo array através de um truque: faremos `return` com *array* vazio, seguido pelo método `concat()` que receberá a lista cujos itens desejamos

copiar:

```
// client/app/domain/negociacoes/Negociacoes.js

class Negociacoes {

  constructor() {

    this._negociacoes = [];
  }

  adiciona(negociacao) {

    this._negociacoes.push(negociacao);
  }

  paraArray() {

    // retorna uma nova referência criada com os itens de thi
    s._negociacoes
    return [].concat(this._negociacoes);
  }
}
```

Ao passarmos o `this._negociacoes` dentro do `concat()`, o retorno será uma nova lista, um novo *array*. Agora se tentarmos utilizar a instrução `this._negociacoes.paraArray().length = 0`, nada acontecerá com a lista encapsulada pela instância de `Negociacoes`, porque estamos modificando uma cópia do array, e não o array original.

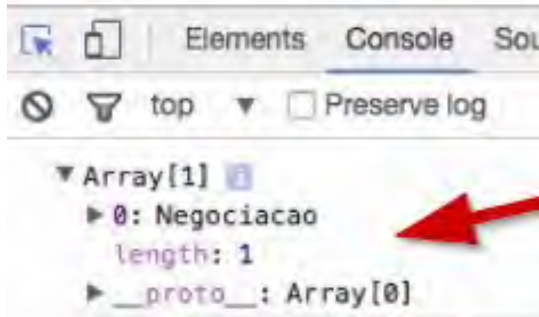


Figura 5.5: Array blindado

Agora que já temos tudo no lugar, vamos alterar `NegociacaoController` e remover do método `adiciona` o código que tenta esvaziar a lista e todas as chamadas ao console, deixando apenas as três instruções essenciais para seu funcionamento:

```
// client/app/controllers/NegociacaoController.js
// código anterior omitido

adiciona(event) {

    event.preventDefault();
    this._negociacoes.adiciona(this._criaNegociacao());
    this._limpaFormulario();
}

// código posterior omitido
```

Dessa forma, garantimos que nossa lista de negociação só poderá receber novas negociações e nenhuma outra operação poderá ser realizada.

Agora que temos nosso controller e nossos modelos, já estamos preparados para apresentar os dados do nosso modelo na página `index.html`. Este é o assunto do próximo capítulo.

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/05>

A MODA NO CANGAÇO

"Feito flecha, feito fogo, feito faca." - Grande Sertão: Veredas

Criamos as classes `Negociacao` e `Negociacoes` para modelar as regras de negócio solicitadas. Inclusive criamos `NegociacaoController` para que o usuário pudesse interagir com instâncias dessas classes. No entanto, ainda não apresentamos os estados desses modelos em `client/index.html`.

Precisamos criar uma solução que seja capaz de traduzir uma instância de `Negociacoes` em alguma apresentação que o usuário entenda, por exemplo, uma `<table>` do HTML. Estamos querendo implementar uma solução de *View*, o *V* do modelo MVC.

6.1 O PAPEL DA VIEW

No modelo MVC, a *View* é a estrutura na qual apresentamos modelos. Um modelo pode ter diferentes apresentações, e alterações nessas apresentações não afetam o modelo. Essa separação é vantajosa, porque a interface do usuário muda o tempo

todo e, quanto menos impacto as alterações na View causarem, mais fácil será a manutenção da aplicação. A questão agora é definirmos a estratégia de construção das nossas Views.

6.2 NOSSA SOLUÇÃO DE VIEW

A estratégia de criação de View que adotaremos será escrevermos a apresentação dos nossos modelos em HTML no próprio JavaScript.

Vamos criar o arquivo `client/app/ui/views/NegociacoesView.js`. Nele, vamos declarar a classe `NegociacoesView` com o único método, chamado `template()`:

```
// client/app/ui/views/NegociacoesView.js

class NegociacoesView {

  template() {

  }

}
```

A finalidade do método `template` será o de retornar uma string com a apresentação HTML que desejamos utilizar para representar em breve o modelo `Negociacoes`. É por isso que, agora, vamos **mover** a `<table>` declarada em `client/index.html` para dentro do método. Sabemos que, se simplesmente movermos o código, o runtime do JavaScript indicará uma série de erros. É por isso que `template()` retornará a marcação HTML como uma template literal:

```
// client/app/ui/views/NegociacoesView.js
```

```

class NegociacoesView {
    template() {
        return `
        <table class="table table-hover table-bordered">
            <thead>
                <tr>
                    <th>DATA</th>
                    <th>QUANTIDADE</th>
                    <th>VALOR</th>
                    <th>VOLUME</th>
                </tr>
            </thead>

            <tbody>
            </tbody>

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

```

Como retiramos o trecho do código referente à tabela, no `index.html`, ela já não será mais exibida abaixo do formulário.

Negociações

Data

Quantidade

Valor

Figura 6.1: Formulário sem tabela

Para não correremos o risco de esquecermos, vamos importar o novo arquivo `client/index.html` . Mas atenção, `NegociacaoController` dependerá da declaração de `NegociacoesView` , então o novo script deve ser importado antes do script `NegociacaoController.js` . Nosso `client/index.html` ficará assim:

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

<script src="app/domain/negociacao/Negociacao.js"></script>
<script src="app/controllers/NegociacaoController.js"></script>
<script src="app/ui/converters/DateConverter.js"></script>
<script src="app/domain/negociacao/Negociacoes.js"></script>

<!-- importou antes! -->
<script src="app/ui/views/NegociacoesView.js"></script>

<script src="app/app.js"></script>
<!-- código posterior omitido -->
```

Agora, em `NegociacaoController` , adicionaremos a propriedade `this._negociacoesView` , que receberá uma instância de `NegociacaoView` :

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

  constructor() {

    let $ = document.querySelector.bind(document);
    this._inputData = $('#data');
    this._inputQuantidade = $('#quantidade');
    this._inputValor = $('#valor');
    this._negociacoes = new Negociacoes();
    this._negociacoesView = new NegociacoesView();
  }
}

// código posterior omitido
```

Após recarregarmos a página, vamos digitar no console as instruções:

```
view = new NegociacoesView()  
view.template() // exibe o template da view
```

Vemos no console o template como string, definido em `NegociacoesView` :

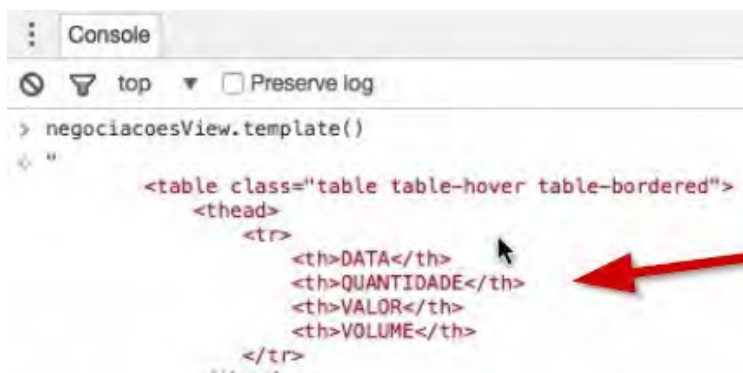


Figura 6.2: String no console

Com certeza não queremos exibir o template no console, queremos que ele seja exibido para o usuário quando ele visualizar `client/index.html` . Como faremos isso?

O primeiro passo é indicarmos o local em `client/index.html` , no qual o template da nossa view deve ser exibido. Indicaremos através de uma `<div>` com o `id="negociacoes"` :

```
<!-- código anterior omitido -->  
<br>  
<div id="negociacoes"></div> <!-- entrou no mesmo local onde tínhamos a <table> -->  
<script src="app/domain/negociacao/Negociacao.js"></script>  
<!-- código posterior omitido -->
```

Excelente! Porém, como uma instância de `NegociacaoView` saberá que deve ser associada ao elemento do DOM da `<div>` que acabamos de adicionar?

Adicionaremos um `constructor()` em `NegociacoesView` que receba um seletor CSS. A partir desse seletor, buscamos o elemento do DOM guardando-o na propriedade `this._elemento`:

```
// client/app/ui/views/NegociacoesView.js

class NegociacoesView {

  constructor(seletor) {

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

  template() {

    // código do template omitido
  }
}
```

Agora, em `NegociacaoController`, passaremos o seletor `#negociacoes` para o `constructor()` da instância de `NegociacoesView` que estamos criando:

```
// client/app/ui/views/NegociacoesView.js

class NegociacaoController {

  constructor() {

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

```

        // passamos para o construtor o seletor CSS de ID
        this._negociacoesView = new NegociacoesView('#negociacoes
    );
    }
    // código posterior omitido

```

O código que escrevemos ainda não soluciona a questão de como transformar o template `NegociacoesView` em elementos do DOM. Estes precisam ser inseridos como elementos filhos do elemento indicado pelo seletor CSS passado para o seu `constructor()`. A solução mora no método `update` que criaremos:

```

// client/app/ui/views/NegociacoesView.js

class NegociacoesView {
    constructor(seletor) {
        this._elemento = document.querySelector(seletor);
    }

    // novo método!
    update() {
        this._elemento.innerHTML = this.template();
    }

    template() {
        // código omitido
    }
}

```

O método `update`, quando chamado, atribuirá à propriedade `innerHTML` o script devolvido pelo método `template()`. O `innerHTML` na verdade é um `setter`, ou seja, quando ele recebe seu valor, por debaixo dos panos um método é chamado convertendo a string em elementos do DOM.

Agora, em `NegociacaoController` , precisamos chamar `this._negociacoesView.update()` para que o template da nossa view seja renderizado:

```
// client/app/controllers/NegociacoesController.js

class NegociacaoController {

  constructor() {

    let $ = document.querySelector.bind(document);
    this._inputData = $('#data');
    this._inputQuantidade = $('#quantidade');
    this._inputValor = $('#valor');
    this._negociacoes = new Negociacoes();
    this._negociacoesView = new NegociacoesView('#negociacoes');

    // atualizando a view
    this._negociacoesView.update();
  }
}

// código posterior omitido
```

Após as últimas alterações, quando recarregarmos a página no navegador, a tabela já será visualizada.



DATA	QUANTIDADE	VALOR	VOLUME
------	------------	-------	--------

Figura 6.3: Tabela na página

A função `update()` funciona e a tabela já pode ser visualizada

abaixo do formulário. Porém, os dados do modelo ainda não são levados em consideração na construção dinâmica da tabela. Podemos cadastrar quantas negociações forem necessárias, que a tabela criada não exibirá os dados da nossa lista. Precisamos de um template dinâmico!

6.3 CONSTRUINDO UM TEMPLATE DINÂMICO

Para tornarmos nosso template dinâmico, o método `update()` passará a receber como parâmetro o modelo no qual a função `template()` deve se basear:

```
// client/app/ui/views/NegociacoesView.js

class NegociacoesView {

  constructor(seletor) {

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

  // recebe o model
  update(model) {

    // repassa o model para this.template()
    this._elemento.innerHTML = this.template(model);
  }

  // PARÂMETRO AQUI!
  // deve retornar o template baseado no model
  template(model) {

    // código do template omitido
  }
}
```


Alteraremos `NegociacaoController` para passarmos `this._negociacoes` como parâmetro para `this._negociacoesView.update()` :

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

  constructor() {

    let $ = document.querySelector.bind(document);
    this._inputData = $('#data');
    this._inputQuantidade = $('#quantidade');
    this._inputValor = $('#valor');
    this._negociacoes = new Negociacoes();
    this._negociacoesView = new NegociacoesView('#negociacoes');
  };

  // recebe inicialmente o modelo que encapsula uma lista vazia
  this._negociacoesView.update(this._negociacoes);
}

// código posterior omitido
```

Só nos resta implementar a lógica do método `template()` de `NegociacoesView` . Cada objeto da nossa lista de negociações será transformado em uma string que corresponda a uma `<tr>` com suas `<td>` . Aliás, como o modelo recebido por `template()` possui o método `paraArray` que nos devolve o array encapsulado por ele, podemos utilizar a poderosa função `map` , que realizará um "de para", de objeto para string, com a lógica que definiremos.

A transformação deve ser feita dentro de `<tbody>` . Sendo assim, vamos acessar o modelo através da expressão `${}` . Para efeito de brevidade, temos apenas o trecho do `template` que modificaremos a seguir:

```
<tbody>
  ${model.paraArray().map(negociacao => {
```

```

        // precisa converter antes de retornar!
        return negociacao;
    }}
</tbody>

```

Não podemos simplesmente retornar a negociação que está sendo iterada pela função `map`, pois continuaremos com um array de negociações. Precisamos de um array de `string`, em que cada elemento seja uma string com a representação de uma `<tr>`.

Para resolvermos esta questão, para cada negociação iterada, vamos retornar uma nova template literal, representando uma `<tr>` e que interpola os dados da negociação:

```

<tbody>
  ${model.toArray().map(negociacao => {
    return `
      <tr>
        <td>${DateConverter.parse(negociacao.data)}</td>
        <td>${negociacao.quantidade}</td>
        <td>${negociacao.valor}</td>
        <td>${negociacao.volume}</td>
      </tr>
    `
  })}
</tbody>

```

Estamos quase lá! Agora que temos um novo array de string, precisamos concatenar cada elemento em uma única string. Esta, no final, será aplicada na propriedade `innerHTML` do elemento que associamos à view. Podemos fazer isso facilmente através da função `join()`, presente em todo array:

```

<tbody>
  ${model.toArray().map(negociacao => {
    return `
      <tr>
        <td>${DateConverter.parse(negociacao.data)}</td>
        <td>${negociacao.quantidade}</td>
        <td>${negociacao.valor}</td>

```

```

        <td>${negociacao.volume}</td>
      </tr>
    }
  }).join('')
</tbody>

```

A função `join()` aceita um separador, no entanto, não queremos separador algum, por isso passamos uma string em branco para ele. Por fim, não podemos nos esquecer de chamarmos

`this._negociacoesView.update(this._negociacoes)` logo após a inclusão da negociação na lista, para que nossa tabela seja renderizada levando em consideração os dados mais atuais.

```

// client/app/controllers/NegociacaoController.js
// código anterior omitido

adiciona(event) {

  event.preventDefault();
  this._negociacoes.adiciona(this._criaNegociacao());
  this._negociacoesView.update(this._negociacoes);
  this._limpaFormulario();
}

// código posterior omitido

```

Vamos ver o que será exibido no navegador, após o preenchimento do formulário:



DATA	QUANTIDADE	VALOR	VOLUME
11/10/2011	2	111	222

Figura 6.4: Tabela com valores

Em seguida, adicionaremos uma nova negociação e seus dados também serão exibidos na tabela.

DATA	QUANTIDADE	VALOR	VOLUME
11/11/1111	2	111	222
22/2/1111	2	200	400

Figura 6.5: Tabela com duas negociações

Se realizarmos novas inclusões, nossa tabela será renderizada novamente, representando o estado atual do modelo.

Atualmente, o innerHTML é bem performático. No entanto, se a quantidade de dados a ser renderizada for exorbitante, a solução de incluir cada <tr> individualmente se destacaria, apesar de mais verbosa. Porém, é uma boa prática utilizar paginação quando trabalhamos com uma massa de dados considerável, o que minimizaria possíveis problemas de performance do innerHTML.

De maneira elegante, usando apenas recursos do JavaScript, conseguimos fazer um *template engine*. Mas que tal enxugarmos um pouco mais o código do nosso template?

Toda arrow function que possui apenas uma instrução pode ter seu bloco `{}` removido, inclusive a instrução `return`, pois ela já assume como retorno o resultado da instrução. Alterando nosso código:

```
<tbody>
  ${model.paraArray().map(negociacao =>
    ,
    <tr>
      <td>${DateConverter.paraTexto(negociacao.data)}</td>
      <td>${negociacao.quantidade}</td>
```

```

        <td>${negociacao.valor}</td>
        <td>${negociacao.volume}</td>
    </tr>
  `).join('')}
</tbody>

```

Agora, com nossa alteração, a única instrução da arrow function é a criação da template literal, que é retornada automaticamente.

Por fim, nosso método `template` completo ficou assim:

```

// client/app/ui/views/NegociacoesView.js
// código anterior omitido

template(model) {

  return `
    <table class="table table-hover table-bordered">
      <thead>
        <tr>
          <th>DATA</th>
          <th>QUANTIDADE</th>
          <th>VALOR</th>
          <th>VOLUME</th>
        </tr>
      </thead>

      <tbody>
        ${model.paramArray().map(negociacao =>
          `
            <tr>
              <td>${DateConverter.paraTexto(negociacao.data
            </td>

              <td>${negociacao.quantidade}</td>
              <td>${negociacao.valor}</td>
              <td>${negociacao.volume}</td>
            </tr>
          `).join('')}
      </tbody>

      <tfoot>
      </tfoot>

```

```

    </table>
  },
}

// código posterior omitido

```

Nossa solução é uma caricatura da estratégia utilizada pela biblioteca React (<https://facebook.github.io/react/>) para renderização da apresentação de componentes através do JSX (<https://facebook.github.io/react/docs/jsx-in-depth.html>).
Vejam os um exemplo:

```

// EXEMPLO APENAS

let minhaLista = React.createClass({
  render: function() {
    return (
      <ul>
        {this.props.list.map(function(listValue){
          return <li>{listValue}</li>;
        })}
      </ul>
    )
  }
});

```

Voltando para a nossa aplicação, se você acha que terminamos com nosso template, se enganou. Ainda precisamos completá-lo com outra informação.

6.4 TOTALIZANDO O VOLUME DE NEGOCIAÇÕES

Precisamos totalizar o volume no rodapé da tabela do nosso template. Primeiro, na classe `Negociacoes`, adicionaremos uma propriedade `getter`, chamada `volumeTotal`. Ela percorrerá a

lista encapsulada pela classe, tornando o somatório dos volumes das negociações.

```
// client/app/domain/negociacao/Negociacoes.js

class Negociacoes {

  constructor() {

    this._negociacoes = []

  }

  // código anterior omitido

  get volumeTotal() {

    let total = 0;

    for(let i = 0; i < this._negociacoes.length; i++) {
      total+=this._negociacoes[i].volume;
    }

    return total;
  }
}
```

Agora, vamos adicionar a marcação completa do `<tfoot>` do nosso template e utilizar a expressão `${}` para interpolar o volume total:

```
<tfoot>
  <tr>
    <td colspan="3"></td>
    <td>${model.volumeTotal}</td>
  </tr>
</tfoot>
```

Um teste demonstra que nossa solução funcionou como esperado.

DATA	QUANTIDADE	VALOR	VOLUME
11/11/1111	2	111	222
11/11/1111	2	2	4
			226

Figura 6.6: Tabela com o volume total

Apesar de funcionar, a lógica do *getter* `volumeTotal` pode ser escrita de outra maneira.

6.5 TOTALIZANDO COM REDUCE

No mundo JavaScript, arrays são muito espertos. Podemos realizar conversões através de `map()`, iterar em seus elementos através de `forEach()`, e concatenar seus elementos com `join()`. Inclusive, podemos reduzir todos os seus elementos a um único valor com a função `reduce()`.

Em nossa aplicação, `reduce()` reduzirá todos os elementos do array a um único valor, no caso, o volume total. Este nada mais é do que a soma do volume de todas as negociações. Vejamos a nova implementação:

```
// client/app/domain/negociacao/Negociacoes.js
// código anterior omitido

get volumeTotal() {

    return this._negociacoes
        .reduce(function (total, negociacao) {
            return total + negociacao.volume
        }, 0);
}

// código posterior omitido
```

Antes de entrarmos nos detalhes da função `reduce()`, vamos

lançar mão de uma arrow function para simplificar ainda mais nosso código:

```
// client/app/domain/negociacao/Negociacoes.js
// código anterior omitido

get volumeTotal() {

    // agora com arrow functions, sem o uso de bloco
    return this._negociacoes
        .reduce((total, negociacao) =>
            total + negociacao.volume, 0);
}

// código posterior omitido
```

A função `reduce()` recebe dois parâmetros: a lógica de redução e o valor inicial da variável acumuladora, respectivamente. A função com a lógica de redução nos dá acesso ao acumulador, que chamamos de `total`, e ao item que estamos iterando, chamado de `negociacao`.

Para cada item iterado, será retornada a soma do total atual com o volume da negociação. O total acumulado será passado para a próxima chamada da função `reduce`, e esse processo se repetirá até o término da iteração. No final, `reduce()` retornará o valor final de `total`.

Ao executarmos o código, vemos que ele funciona perfeitamente:



DATA	QUANTIDADE	VALOR	VOLUME
11/11/1911	2	11	22
11/11/1111	5	5	25
11/11/1111	5	5	47

Figura 6.7: Tabela com total

Terminamos o template da tabela. A cada negociação incluída, a informação será exibida para o usuário com base nas informações da lista. Mas será que podemos generalizar nossa solução? É isso que veremos no próximo capítulo.

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/06>

O PLANO

"O correr da vida embrulha tudo, a vida é assim: esquenta e esfria, aperta e daí afrouxa, sossega e depois desinquieta. O que ela quer da gente é coragem." - Grande Sertão: Veredas

Já somos capazes de exibir novas negociações na tabela de negociações, no entanto, podemos melhorar a experiência do usuário exibindo uma mensagem indicando que a operação foi realizada com sucesso.

No lugar de usarmos uma string para esta finalidade, criaremos a classe `Mensagem`, que representará uma mensagem em nosso sistema. Ela não faz parte do domínio da aplicação que é o de negociações, por isso vamos criá-la em `client/app/ui/models`:

```
// client/app/ui/models/Mensagem.js
```

```
class Mensagem {  
  
  constructor() {  
  
    this._texto;  
  
  }  
}
```

```

    get texto() {

        return this._texto;
    }
}

```

Antes de continuarmos, importaremos o script em `client/index.html`. Não podemos nos esquecer de importá-lo antes de `NegociacaoController.js`, pois há uma ordem de dependência, algo que já vimos quando importamos `NegociacoesView`.

```

<!-- client/index.html -->
<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script src="app/controllers/NegociacaoController.js"></script>
<script src="app/ui/converters/DateConverter.js"></script>
<script src="app/domain/negociacao/Negociacoes.js"></script>
<script src="app/ui/views/NegociacoesView.js"></script>

<!-- importou aqui! -->
<script src="app/ui/models/Mensagem.js"></script>

<script src="app/app.js"></script>
<!-- código posterior omitido -->

```

Usamos a convenção do prefixo `_` para manter o `_texto` privado, e criamos um *getter* para a propriedade. Mas e se quisermos alterar o texto da mensagem? Para isso, criaremos nosso primeiro *setter*:

```

// client/app/ui/models/Mensagem.js

class Mensagem {

    constructor() {

        this._texto;
    }

    get texto() {

```

```

        return this._texto;
    }

    set texto(texto) {

        this._texto = texto;
    }
}

```

Da mesma maneira que acessamos um `getter` como se fosse uma propriedade, podemos atribuir um valor ao `setter` também da mesma maneira. Recarregando nossa página, podemos realizar o seguinte teste no console:

```

mensagem = new Mensagem();
mensagem.texto = 'x';

// NOSSO TESTE ALTEROU A PROPRIEDADE!
console.log(mensagem.texto);

```

Podemos ainda adicionar um parâmetro no `constructor()` da classe, contendo o texto da mensagem.

```

// client/app/ui/models/Mensagem.js

class Mensagem {

    constructor(texto) {

        this._texto = texto;
    }

    get texto() {

        return this._texto;
    }

    set texto(texto) {

        this._texto = texto;
    }
}

```

```
}
```

Vamos fazer um novo teste no console:

```
mensagem = new Mensagem('Flávio Almeida');  
  
// PASSOU CORRETAMENTE O TEXTO PELO CONSTRUCTOR()  
console.log(mensagem.texto);
```

No entanto, o que acontecerá se não passarmos um valor para o `constructor()` ?

```
mensagem = new Mensagem();  
  
// O RESULTADO É UNDEFINED!  
console log(mensagem texto)
```

Não podemos deixar que isso aconteça.

7.1 PARÂMETRO DEFAULT

Que tal assumirmos uma string em branco, caso o parâmetro não seja passado?

```
// client/app/ui/models/Mensagem.js  
  
class Mensagem {  
  
  constructor(texto) {  
  
    if(!texto) {  
      texto = '';  
    }  
  
    this._texto = texto;  
  }  
  
  get texto() {  
  
    return this._texto;  
  }  
}
```

```

    set texto(texto) {

        this._texto = texto;
    }
}

```

Podemos nos livrar do `if` com o seguinte truque:

```

// client/app/ui/models/Mensagem.js

class Mensagem {

    constructor(texto) {

        this._texto = texto || '';
    }

    // código posterior omitido

```

No código anterior, se o valor do parâmetro `texto` for diferente de `null`, `undefined`, `0` e não for uma string em branco, ele será atribuído a `this._texto`; caso contrário, receberá a string em branco.

Podemos enxugar ainda mais indicando no próprio parâmetro do `constructor()` seu valor padrão, algo possível apenas a partir do ES2015 (ES6) em diante:

```

// client/app/ui/models/Mensagem.js

class Mensagem {

    // se não for passado, será uma string em branco
    constructor(texto = '') {

        this._texto = texto
    }

    // código posterior omitido

```

Este é um recurso interessante, porque podemos definir um parâmetro default, tanto no `constructor()` como em métodos.

Com a classe criada, vamos adicionar a propriedade `this._mensagem` em `NegociacaoController.js`. Ela guardará uma instância de `Mensagem`:

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

  constructor() {

    let $ = document.querySelector.bind(document);
    this._inputData = $('#data');
    this._inputQuantidade = $('#quantidade');
    this._inputValor = $('#valor');
    this._negociacoes = new Negociacoes();
    this._negociacoesView = new NegociacoesView('#negociacoes');

    this._negociacoesView.update(this._negociacoes);

    // instanciando o modelo!
    this._mensagem = new Mensagem();
  }

  // código posterior omitido
```

Quando uma nova negociação for realizada, vamos atribuir uma nova mensagem à propriedade `this._mensagem.texto`.

```
// client/app/controllers/NegociacaoController.js

// código anterior omitido

adiciona(event) {

  event.preventDefault();
  this._negociacoes.adiciona(this._criaNegociacao());
  this._mensagem.texto = 'Negociação adicionada com sucesso';
  this._negociacoesView.update(this._negociacoes);
  this._limpaFormulario();
```



```
}  
// código posterior omitido
```

Se preencheremos o formulário, os dados serão inseridos na tabela, mas a mensagem não, porque ela ainda não possui uma apresentação, isto é, sua View. Chegou a hora de criá-la.

7.2 CRIANDO A CLASSE MENSAGEMVIEW

Vamos criar a View que representará nosso modelo Mensagem em `client/app/ui/views/MensagemView.js`:

```
class MensagemView {  
  
  constructor(seletor) {  
  
    this._elemento = document.querySelector(seletor);  
  }  
  
  template(model) {  
  
    return `

${model.texto}</p>`;  
  }  
}


```

Usaremos o `alert alert-info` do Bootstrap, seguido pela expressão `${model.texto}`. Logo abaixo, adicionaremos o método `update()` que receberá o `model`.

```
// client/app/ui/views/MensagemView.js  
  
class MensagemView {  
  
  constructor(seletor) {  
  
    this._elemento = document.querySelector(seletor);  
  }  
  
  update(model) {  
  
    this._elemento.innerHTML = this.template(model);  
  }  
}
```

```

template(model) {

    return `

${model.texto}</p>`;
}

// método update
update(model) {

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


```

Agora vamos importar o arquivo que acabamos de criar em `client/index.html` :

```

<!-- client/index.html -->
<!-- código anterior omitido -->

<script src="app/domain/negociacao/Negociacao.js"></script>
<script src="app/controllers/NegociacaoController.js"></script>
<script src="app/ui/converters/DateConverter.js"></script>
<script src="app/domain/negociacao/Negociacoes.js"></script>
<script src="app/ui/views/NegociacoesView.js"></script>
<script src="app/ui/models/Mensagem.js"></script>

<!-- importou! -->
<script src="app/ui/views/MensagemView.js"></script>

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

<!-- código posterior omitido -->

```

Ainda em `client/index.html` , vamos adicionar a `<div>` com ID `mensagemView` para indicar o local de renderização de `MensagemView` :

```

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

<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width">

```

```

<title>Negociações</title>
<link rel="stylesheet" href="css/bootstrap.css">
<link rel="stylesheet" href="css/bootstrap-theme.css">
</head>
<body class="container">

    <h1 class="text-center">Negociações</h1>

    <!-- nova tag! -->
    <div id="mensagemView"></div>

    <form class="form">
<!-- código anterior omitido -->

```

Em `NegociacoesController`, vamos adicionar a propriedade `_mensagemView` que receberá uma instância de `MensagemView`. Não podemos nos esquecer de passar o seletor CSS no construtor da classe, para que ela encontre o elemento do DOM onde seu template deve ser renderizado. Inclusive, já vamos chamar seu método `update`, passando o modelo da mensagem como parâmetro.

```

// client/app/controllers/NegociacaoController.js

class NegociacaoController {

    constructor() {

        let $ = document.querySelector.bind(document);
        this._inputData = $('#data');
        this._inputQuantidade = $('#quantidade');
        this._inputValor = $('#valor');
        this._negociacoes = new Negociacoes();
        this._negociacoesView = new NegociacoesView('#negociacoes');

        this._negociacoesView.update(this._negociacoes);
        this._mensagem = new Mensagem();

        // nova propriedade!
        this._mensagemView = new MensagemView('#mensagemView');
        this._mensagemView.update(this._mensagem);

    }
}

```

```
// código posterior omitido
```

Ainda em `NegociacaoController`, vamos alterar seu método `adiciona()` para que chame a atualização da View logo após receber a mensagem que definimos:

```
// client/app/controllers/NegociacaoController.js
```

```
// código anterior omitido
```

```
adiciona(event) {  
  
    event.preventDefault();  
    this._negociacoes.adiciona(this._criaNegociacao());  
    this._mensagem.texto = 'Negociação adicionada com sucesso';  
    this._negociacoesView.update(this._negociacoes);  
  
    // atualiza a view com o texto da mensagem que acabamos de at  
ribuir  
    this._mensagemView.update(this._mensagem);  
    this._limpaFormulario();  
}
```

```
// código posterior omitido
```

Vamos ver se algo já é exibido no navegador.

DATA	QUANTIDADE	VALOR	VOLUME
------	------------	-------	--------

Figura 7.1: Barra azul

Agora aparece uma barra com um fundo azul. Isto é uma mensagem do Bootstrap vazia, e vamos melhorar isso em breve. O cadastro de usuários passa a exibir mensagens para o usuário.



Figura 7.2: Cadastro com sucesso

Conseguimos adicionar os dados à tabela, e a mensagem de sucesso apareceu corretamente. Veja que conseguimos usar o mesmo mecanismo de criação da View para lidarmos com as mensagens do sistema. Agora, vamos resolver o problema da mensagem vazia.

Uma solução seria removermos a instrução `this._mensagemView.update(this._mensagem)` do `constructor()` de `NegociacaoController`. No entanto, vamos manter dessa forma para padronizar nossa solução. Convencionamos que a função `update` deve ser sempre chamada toda vez que criarmos nossa view. Sendo assim, vamos à segunda solução.

No template de `MensagemView`, podemos usar um `if` ternário que retornará um parágrafo vazio, sem as classes do Bootstrap, caso a mensagem esteja em branco.

Vamos alterar `client/app/ui/views/MensagemView.js` :

```
// client/app/ui/views/MensagemView.js

class MensagemView {

    // código anterior omitido

    template(model) {

        return model.texto
            ? `
```

Lembre-se deque, para o interpretador do JavaScript, uma string em branco, `0` , `undefined` e `null` são considerados `false` . Se `model.texto` tiver qualquer valor diferente dos que foram listados, será considerado `true` .

Conseguimos resolver a parte das mensagens para o usuário. Mas será que conseguimos melhorar ainda mais o código?

7.3 HERANÇA E REUTILIZAÇÃO DE CÓDIGO

Temos duas Views criadas: `MensagemView` e `NegociacoesView` . Se observarmos, ambas possuem um `constructor()` que recebe um seletor, além de possuírem a propriedade `elemento` . As duas têm os métodos `template()` e `update()` . O método `update()` é idêntico nas duas classes, já o `template()` tem uma implementação diferente, apesar de receber um único parâmetro e devolver sempre uma string.

Para cada nova View criada em nossa aplicação, teremos de repetir boa parte do código que já escrevemos - com exceção da função `template()`, que é diferente para cada classe. Não parece ser um trabalho de cangaceiro ficar repetindo código. Existe uma solução?

Para evitarmos a repetição, criaremos uma classe chamada `View`, que centralizará todo o código que é comum entre todas as Views da nossa aplicação (no caso, o `constructor()` e o método `update()`).

Vamos criar essa classe em `client/app/ui/views/View.js`:

```
// client/app/ui/views/View.js

class View {

  constructor(seletor) {

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

  update(model) {

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

Antes de continuarmos, vamos importar a classe em `client/index.html`. Entretanto, ela não pode ser importada como penúltimo script da nossa página, antes de `app.js`, como fizemos com os demais scripts. Todos nossos arquivos de View que vamos criar dependerão diretamente da classe `View`, e o interpretador JavaScript indicará um erro caso as classes não sejam carregadas antes das demais.

Vamos importar a nova classe antes do script de

NegociacoesView.js :

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

<script src="app/domain/negociacao/Negociacao.js"></script>
<script src="app/controllers/NegociacaoController.js"></script>
<script src="app/ui/converters/DateConverter.js"></script>
<script src="app/domain/negociacao/Negociacoes.js"></script>

<!-- importou aqui -->

<script src="app/ui/views/View.js"></script>

<!-- a declaração de NegociacoesView dependerá da declaração de View -->
<script src="app/ui/views/NegociacoesView.js"></script>
<script src="app/ui/models/Mensagem.js"></script>
<script src="app/app.js"></script>
```

A classe `View` recebeu tudo o que as Views tinham em comum: um `constructor(elemento)` - que guardará internamente um elemento - e `update()`. Vale lembrar que o método `template` não pôde ser definido na classe, porque cada View em nossa aplicação possuirá um código diferente.

Agora que centralizamos o código comum em uma classe, vamos remover o `construtor()` e o método `update()` de `NegociacaoView` e de `MensagemView`. Elas ficarão assim:

```
// client/app/ui/views/NegociacoesView.js

class NegociacoesView {

  template(model) {

    // código do template omitido
  }
}
```



```
// client/app/ui/views/MensagemView.js

class MensagemView {

    template(model) {

        return model.texto
            ? `

</p>`;
    }
}


```

Do jeito que está, nossa aplicação não funcionará, pois tanto `NegociacoesView` como `MensagemView` dependem do código que busca o elemento do DOM (definido no `constructor()`) e a função `update()`, que foram removidos.

Para as classes que alteramos voltem a funcionar, faremos com que **herdem** o código da classe `View`, aquela que agora centraliza em um único lugar o `constructor()`, inclusive a função `update`.

É através da instrução `extends` que podemos herdar outra classe. Vamos alterar `NegociacaoView` e `MensagemView`:

```
// client/app/ui/views/NegociacoesView.js

class NegociacoesView extends View {

    template(model) {

        // código do template omitido
    }
}

// client/app/ui/views/MensagemView.js

class MensagemView extends View {

    template(model) {
```

```

        return model.texto
        ? `<p class="alert alert-info">${model.texto}</p>`
        : `<p></p>`;
    }
}

```

Se cadastrarmos uma nova negociação, veremos que está tudo funcionando.

The screenshot shows a web interface for managing negotiations. At the top, there's a title 'Negociações'. Below it, a light blue banner displays the message 'Negociação adicionada com sucesso!'. The main form contains three input fields: 'Data' with a date picker showing 'dd/mm/aaaa', 'Quantidade' with the value '1', and 'Valor' with the value '0'. To the left of the 'Valor' field is a label 'Valor'. Below the form is a blue 'Incluir' button. At the bottom of the interface, there are two buttons: 'Importar Negociações' and 'Apagar'.

Figura 7.3: Formulário funcionando corretamente

A solução de template dentro do modelo MVC adotada pelo autor foi a mais acessível e didática que ele encontrou para colocar em prática recursos da linguagem JavaScript, sem gastar páginas e mais páginas detalhando a criação de um framework, o que desvirtuaria a proposta do livro. Contudo, não será mera coincidência encontrar diversos conceitos apresentados em frameworks e bibliotecas do mercado.

Conseguimos evitar código duplicado e garantimos o reúso de código através de herança. Contudo, ainda podemos melhorar nosso código um pouquinho mais.

7.4 CLASSES ABSTRATAS?

Na programação orientada a objetos, existe o conceito de classes abstratas. Essas classes podem ter um ou mais métodos abstratos, que nada mais são do que métodos que não possuem implementação, apenas a assinatura que consiste no nome do método e o parâmetro que recebe.

Esses métodos devem ser implementados pelas classes filhas que herdam da classe abstrata. No caso do método `template()`, é impossível que nossa classe `View` tenha uma implementação para ser herdada, pois ela não tem como saber de antemão qual será a template literal retornada pelas suas classes filhas.

No JavaScript, não há classes abstratas, mas podemos tentar emulá-las da seguinte maneira.

Na classe `View`, vamos adicionar o método `template()`, cuja implementação consiste no lançamento de uma exceção:

```
// client/app/ui/views/View.js

class View {

  constructor(seletor) {

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

  update(model) {

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

  // novo método!
  template(model) {

    throw new Error('Você precisa implementar o método templa
```

```
te');  
}  
}
```

Do jeito que está, todas as classes filhas que herdarem de `View` ganharam o método `template()` padrão. Contudo, se esse método não for sobrescrito pela classe filha, esse comportamento padrão resultará em um erro, deixando claro para o programador que ele deve implementá-lo, caso tenha esquecido.

Podemos fazer um simples teste **removendo temporariamente** o método `template()` de `MensagemView` e recarregando nossa aplicação. Receberemos no console a seguinte mensagem:

```
Uncaught Error: Você precisa implementar o método template  
    at MensagemView.template (View.js:15)  
    at MensagemView.update (View.js:10)  
    at new NegociacaoController (NegociacaoController.js:14)  
    at app.js:1
```

Diferente de outras linguagens com tipagem estática (como Java, Rust ou C#), a exceção só será lançada em tempo de execução. Entretanto, ainda assim é uma maneira de ajudar os desenvolvedores fornecendo uma mensagem clara do erro em seu código, caso não implemente os métodos "abstratos" da classe filha.

7.5 PARA SABER MAIS: SUPER

Em nossa aplicação, tanto o constructor de `View` quanto o constructor de suas classes filhas recebem a mesma quantidade de parâmetros. Contudo, quando a classe filha recebe uma quantidade de parâmetros diferente, é necessário escrevermos um pouco mais de código.

Vejamos um exemplo hipotético envolvendo as classes

Funcionario e Gerente .

```
class Funcionario {  
    constructor(nome) {  
        this._nome = nome;  
    }  
    get nome() {  
        return this._nome;  
    }  
    set nome(nome) {  
        this._nome = nome;  
    }  
}
```

```
class Gerente extends Funcionario {}
```

No exemplo anterior, a classe Gerente herdou de Funcionario e, por conseguinte, herdou seu construtor(). Mas o que acontecerá se Gerente receber, além do nome, uma senha de acesso?

```
class Gerente extends Funcionario {  
    constructor(nome, senha) {  
        /* o que fazer ? */  
    }  
}
```

A classe Gerente precisou do seu próprio construtor(), pois recebe dois parâmetros. Nesse caso, precisamos chamar o construtor() da classe pai, passando os parâmetros de que ele precisa:

```
class Gerente extends Funcionario {
```

```

constructor(nome, senha) {

    super(nome); // chama o constructor() do pai

    // propriedade exclusiva de Gerente
    this._senha = senha;
}

// getters e setters exclusivos do Gerente
get senha() {

    return this._senha;
}

set senha(senha) {

    return this._senha = senha;
}
}

```

É importante que a chamada de `super()` seja a primeira instrução dentro do `constructor()` da classe filha; caso contrário, o `this` da classe filha não será inicializado e a instrução `this._senha = senha` resultará em um erro. Agora que já fizemos o teste, podemos **voltar com o método `template()` em `MensagemView`.**

7.6 ADQUIRINDO UM NOVO HÁBITO COM CONST

Aprendemos a favorecer o uso de `let` em vez de `var` nas declarações de variáveis, inclusive entendemos como funciona o *temporal dead zone*. No entanto, podemos lançar mão de `const`. Variáveis declaradas através de `const` obrigatoriamente devem ser inicializadas e não podem receber novas atribuições. Vejamos um exemplo isolado:

```
const nome = 'Flávio';
nome = 'Almeida'; // resulta no erro Uncaught TypeError: Assignme
nt to constant variable
```

No exemplo anterior, não podemos usar novamente o operador `=` para atribuir um novo valor a variável. O fato de usarmos essa declaração específica não quer dizer que a variável seja imutável. Vejamos outro exemplo:

```
const data = new Date();
data.setMonth(10); // funciona perfeitamente
console.log(data.getMonth());
data = new Date() // resulta no erro Uncaught TypeError: Assignme
nt to constant variable
```

Neste último exemplo, por mais que tenha declarado `data` com `const`, através do método `data.setMonth(10)`, conseguimos alterar o mês da data. Isso é a prova real de que `const` não cria objetos imutáveis.

Por fim, valem as seguintes recomendações cangaceiras:

Use `const` sempre que possível.

Utilize `let` apenas se a variável precisar receber novas atribuições, por exemplo, uma variável totalizadora.

Não use `var`, pois a única maneira da variável ter escopo é quando declarada dentro de uma função.

Dessa forma, vamos alterar `NegociacaoController` para fazer uso de `const`:

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

  constructor() {
```

```

    // não vamos atribuir outor valor à variável
    const $ = document.querySelector.bind(document);

    // código posterior omitido
}

// código posterior omitido

```

Podemos até alterar `client/app/app.js` para utilizar `const` na declaração da variável `controller` :

```

// client/app/app.js

// USANDO CONST
const controller = new NegociacaoController();

document
  .querySelector('.form')
  .addEventListener('submit', controller.adiciona.bind(controller));

```

Uma pequena melhoria que utilizaremos ao longo do projeto.

Será que podemos melhorar ainda mais nosso código, por exemplo, enxugando-o e removendo cada vez mais a responsabilidade do programador? É isso que veremos no próximo capítulo.

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/07>

Parte 2 - Força Volante

Procurado pela Força Volante, faminto e abatido, ele chegou a pensar em desistir várias vezes. Porém, a vontade inquebrantável de descobrir o que o esperava no fim era maior do que seu sofrimento. Então, ele recuperou suas forças e se pôs a andar. Durante sua caminhada, ajudou e foi ajudado, lutou sem ser derrubado, correu e foi arrastado. Suas cicatrizes eram a marca da experiência que acumulava. Ele tinha muito o que aprender ainda, mas havia espaço em sua pele para novas experiências.

UM CANGACEIRO SABE DELEGAR TAREFAS

"Tem horas em que penso que a gente carecia, de repente, de acordar de alguma espécie de encanto." - Grande Sertão: Veredas

Nossa aplicação já funciona e totaliza o volume no fim da tabela. Mas ainda falta implementarmos uma regra de negócio no modelo `Negociacoes`. Precisamos criar um método que nos permita esvaziar a lista de negociações encapsulada. Vamos criá-lo.

Primeiro, vamos adicionar o método `esvazia()` em `Negociacoes`:

```
// client/app/domain/negociacao/Negociacoes.js
```

```
class Negociacoes {  
  
  constructor() {  
  
    this._negociacoes = [];  
  }  
  
  adiciona(negociacao) {
```

```

        this._negociacoes.push(negociacao);
    }

    paraArray() {

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

    get volumeTotal() {

        return this._negociacoes
            .reduce((total, negociacao) =>
                total + negociacao.volume, 0);
    }

    // novo método!
    esvazia() {

        this._negociacoes = [];
    }
}

```

O novo método, quando chamado, atribuirá um novo array sem elementos à propriedade `this._negociacoes`. Agora, precisamos que este novo método seja chamado quando o usuário clicar no botão "Apagar". Para isso, criaremos primeiro o método `apaga()` em `NegociacaoController`, aquele que será chamado através do botão "Apagar" de `index.html`:

```

// client/app/controllers/NegociacaoController.js
// código anterior omitido

apaga() {

    this._negociacoes.esvazia();
    this._negociacoesView.update(this._negociacoes);
    this._mensagem.texto = 'Negociações apagadas com sucesso';
    this._mensagemView.update(this._mensagem);
}

```

```
// código posterior omitido
```

Por fim, precisamos associar o clique do botão "Apagar" com a chamada do método que acabamos de criar em nosso controller. Isso não é novidade para nós, já fizemos algo semelhante em `client/app/app.js` :

```
// client/app/app.js

const controller = new NegociacaoController();

document
  .querySelector('.form')
  .addEventListener('submit', controller.adiciona.bind(controller));

// buscando o elemento pelo seu ID
document
  .querySelector('#botao-apaga')
  .addEventListener('click', controller.apaga.bind(controller))
;
```

Quando clicarmos em "Apagar" , a tabela ficará vazia. A View foi atualizada com os dados do modelo da lista de negociações atualizada. Inclusive, a mensagem foi exibida:



Figura 8.1: Lista apagada

Apesar de funcionar como esperado, podemos melhorar um

pouco mais nosso modelo `Negociacoes`. Nele, a propriedade `_negociacoes` não está congelada permitindo que receba novas atribuições. Aprendemos que através de `Object.freeze` podemos impedir novas atribuições às propriedades da instância de um objeto. Que tal lançarmos mão deste recurso também em `Negociacoes`? Alterando a classe:

```
// client/app/domain/negociacao/Negociacoes.js

class Negociacoes {

  constructor() {

    this._negociacoes = [];

    // CONGELOU A INSTÂNCIA
    Object.freeze(this);
  }

  // código anterior omitido
}
```

Excelente, mas como nossa instância não aceita novas atribuições à propriedade `this._negociacoes` o método `esvazia()` da classe não funcionará:

```
// client/app/domain/negociacao/Negociacoes.js

// código anterior omitido

esvazia() {

  // NÃO ACEITARÁ A NOVA ATRIBUIÇÃO
  this._negociacoes = [];
}

// código posterior omitido
```

Para solucionarmos este problema, podemos alterar a implementação do nosso método `esvazia()` para:

```
// client/app/domain/negociacao/Negociacoes.js

// código anterior omitido

    esvazia() {

        this._negociacoes.length = 0;
    }

// código posterior omitido
```

Quando atribuímos 0 à propriedade `length` de um array, eliminamos todos os seus itens.

Nossa aplicação funciona, mas toda vez que tivermos de alterar nossos modelos, precisaremos lembrar de chamar o método `update()` da sua respectiva `View`. A chance de esquecermos de chamar o método é alta, ainda mais se novas `Views` forem introduzidas em nossa aplicação. Será que existe alguma maneira de automatizarmos a atualização da `View`?

8.1 E SE ATUALIZARMOS A VIEW QUANDO O MODELO FOR ALTERADO?

A alteração do modelo é um ponto inevitável em nossa aplicação. Que tal criarmos uma estratégia na qual, toda vez que o modelo for alterado, a sua respectiva `View` seja atualizada?

Para que isso seja possível, o `constructor()` dos nossos modelos precisará receber uma estratégia de atualização que será chamada toda vez que algum de seus métodos que alteram seu estado interno (suas propriedades) for chamado. Vamos começar por `Negociacoes`:

```
// client/app/domain/negociacoes/Negociacoes.js
```

```

class Negociacoes {

    // agora recebe um parâmetro!
    constructor(armadilha) {

        this._negociacoes = [];
        this._armadilha = armadilha;
        Object.freeze(this);
    }

    // código posterior omitido

```

Nossa classe agora recebe o parâmetro `armadilha`, que será guardado em uma propriedade da classe. Esse parâmetro nada mais é do que uma função com a lógica de atualização da View. Agora, nos métodos `adiciona()` e `esvazia()`, chamaremos a função armazenada em `this._armadilha`:

```

// client/app/domain/negociacoes/Negociacoes.js

class Negociacoes {

    constructor(armadilha) {

        this._negociacoes = [];
        this._armadilha = armadilha;
        Object.freeze(this);
    }

    adiciona(negociacao){

        this._negociacoes.push(negociacao);

        // chamando a função
        this._armadilha(this);
    }

    // código anterior omitido

    esvazia() {

```

```

        this._negociacoes.length = 0;

        // chamando a função
        this._armadilha(this);
    }
}

```

Veja que a função armazenada em `this._armadilha` é chamada recebendo como parâmetro `this`. Isso permitirá que a função tenha uma referência para a instância que está executando a operação, no caso, instâncias de `Negociacoes`.

Agora, em `NegociacaoController`, vamos passar uma estratégia de atualização para o `constructor()` do modelo que acabamos de alterar. Por enquanto, usaremos uma *function* em vez de uma *arrow function*:

```

// client/app/domain/negociacoes/Negociacoes.js

class NegociacaoController {

    constructor() {

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

        // passando a estratégia a ser utilizada

        this._negociacoes = new Negociacoes(function(model) {

            this._negociacoesView.update(model);
        });

        this._negociacoesView = new NegociacoesView('#negociacoes');

        // precisa continuar, para atualizar
        // na inicialização do controller
    }
}

```



```

        this._negociacoesView.update(this._negociacoes);

        this._mensagem = new Mensagem();
        this._mensagemView = new MensagemView('#mensagemView');
        this._mensagemView.update(this._mensagem);
    }

    // código posterior omitido

```

Agora, toda vez que os métodos `adiciona()` e `esvazia()` de `Negociacoes` forem chamados, a estratégia da armadilha passada no `constructor()` será chamada. Veja que a armadilha passada como parâmetro receberá `this` ao ser chamada, que é uma referência à instância de `Negociacoes`. É esta referência que acessaremos como `model` na função que passamos para `Negociacoes`, em `NegociacaoController`. Vamos relembrar desse trecho de código:

```

// client/app/domain/negociacoes/Negociacoes.js
// código anterior omitido

// model será a instância de Negociacoes
// estiver chamando adiciona() ou esvazia()
this._negociacoes = new Negociacoes(function(model) {

    this._negociacoesView.update(model);
});

// código posterior omitido

```

Pode parecer um tanto estranho que a função declarada tente utilizar `this._mensagemView` antes de ter sido declarada, mas não há problema nenhum nisso. Isso porque ela não será executada imediatamente, só quando os métodos `adiciona()` e `esvazia()` forem chamados.

Agora, tanto em `adiciona()` quanto em `apaga()` de `NegociacaoController`, removeremos a instrução que faz a

atualização da view de `Negociacoes`. Os dois métodos devem ficar assim:

```
// client/app/controllers/NegociacaoController.js
// código posterior omitido

adiciona(event) {

    event.preventDefault();
    this._negociacoes.adiciona(this._criaNegociacao());

    // não chama mais o update de negociacoesView

    this._mensagem.texto = 'Negociação adicionada com sucesso';
    this._mensagemView.update(this._mensagem);
    this.limpaFormulario()
}

// código posterior omitido

apaga() {

    this._negociacoes.esvazia();

    // não chama mais o update de negociacoesView

    this._mensagem.texto = 'Negociações apagadas com sucesso';
    this._mensagemView.update(this._mensagem);
}
```

Agora que temos todo o código em seu devido lugar, vamos recarregar nossa página e testar o resultado. Nenhuma atualização é feita e ainda recebemos no console a seguinte mensagem de erro:

```
Uncaught TypeError: Cannot read property 'update' of undefined
    at Negociacoes._armadilha (NegociacaoController.js:12)
    at Negociacoes.adiciona (Negociacoes.js:12)
    at NegociacaoController.adiciona (NegociacaoController.js:25)
```

Pelo console, ao procurarmos a instrução que causou o erro, encontramos este trecho de código de `NegociacaoController`:

```
// client/app/controllers/NegociacaoController.js

// código anterior omitido

this._negociacoes = new Negociacoes(function(model) {

    // CAUSA DO ERRO
    this._negociacoesView.update(model);
});

// código posterior omitido
```

Estávamos esperando que a função armadilha passada para o constructor() de Negociacoes fosse capaz de ter acesso à propriedade this._negociacoesView de NegociacaoController. Contudo, devido à **natureza dinâmica** de toda function(), o valor de this em this._negociacoesView.update(model) passou a ser a instância de Negociacoes, que não possui a propriedade que estamos acessando.

Só sabemos o contexto (this) de uma função em tempo de execução (*runtime*) do nosso código. Podemos fazer um teste para vermos esse comportamento com mais clareza adicionando um console.log(this) na função que passamos como armadilha:

```
// client/app/controllers/NegociacaoController.js
// código anterior omitido

this._negociacoes = new Negociacoes(function(model) {

    // imprimirá Negociacoes no console em vez de NegociacaoController
    console.log(this);

    this._negociacoesView.update(model);
});

// código posterior omitido
```

Verificando no console, temos:

```
Negociacoes // exibiu Negociacoes!  
Uncaught TypeError: Cannot read property 'update' of undefined
```

Ele mostra que `this` é o `Negociacoes`. É assim porque a função está sendo executada no contexto de `Negociacoes`. Será que temos como mudar o contexto de `this`?

8.2 DRIBLANDO O THIS DINÂMICO

Uma maneira de solucionarmos o problema é guardando uma referência para o `this` de `NegociacaoController` em uma variável (por exemplo, `self`), e utilizando essa variável na função de armadilha. Vejamos:

```
// client/app/domain/negociacoes/Negociacoes.js  
  
class NegociacaoController {  
  
  constructor() {  
  
    const $ = document.querySelector.bind(document);  
    this._inputData = $('#data');  
    this._inputQuantidade = $('#quantidade');  
    this._inputValor = $('#valor');  
  
    // self é NegociacaoController  
    const self = this;  
  
    this._negociacoes = new Negociacoes(function(model) {  
  
      // continua sendo Negociacoes  
      console.log(this);  
  
      // acessamos self  
      // temos certeza de que é NegociacaoController  
  
      self._negociacoesView.update(model);  
    });  
  }  
};
```

```
// código posterior omitido
```

Essa solução funciona, no entanto, nos obriga a declarar uma variável extra para guardarmos uma referência para o valor de `this` que desejamos usar em nossa função de armadilha. Podemos resolver de outra maneira.

Vamos voltar nosso código para o estágio anterior, que não está funcionando como esperado:

```
// client/app/domain/negociacoes/Negociacoes.js
```

```
class NegociacaoController {  
  
  constructor() {  
  
    const $ = document.querySelector.bind(document);  
    this._inputData = $('#data');  
    this._inputQuantidade = $('#quantidade');  
    this._inputValor = $('#valor');  
  
    this._negociacoes = new Negociacoes(function(model) {  
  
      console.log(this);  
      this._negociacoesView.update(model);  
    });  
  
  }  
  
  // código posterior omitido
```

Para que nossa função passada como armadilha para o `constructor()` de `Negociacoes` passe a considerar a instância de `NegociacaoController` como `this`, e não a instância `Negociacoes`, passaremos mais um parâmetro para o `constructor()` de `Negociacoes` além da armadilha. Seu primeiro parâmetro será o `this`, aquele que se refere a `NegociacaoController`:

```
// client/app/controller/NegociacaoController.js
```

```

class NegociacaoController {

  constructor() {

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

    // o this passado ainda aponta para
    // uma instância de NegociacaoController

    this._negociacoes = new Negociacoes(this, function(model)

{

    console.log(this);
    this._negociacoesView.update(model);
  });

  // código posterior omitido

```

Precisamos alterar `Negociacoes` para que receba e guarde o parâmetro recebido:

```

// client/app/domain/negociacoes/Negociacoes.js

class Negociacoes {

  constructor(contexto, armadilha) {

    this._negociacoes = [];
    this._armadilha = armadilha;
    this._contexto = contexto;
    Object.freeze(this);
  }

  // código posterior omitido

```

Agora, vamos alterar a chamada de `this._armadilha` nos métodos `adiciona()` e `esvazia()`. Faremos isso através da chamada da função `call()` que toda função em JavaScript

possui. Ela recebe dois parâmetros. O primeiro é o contexto que será usado como `this` na chamada da função e o segundo, o parâmetro recebido por esta função:

```
// client/app/domain/negociacoes/Negociacoes.js

class Negociacoes {

  constructor(contexto, armadilha) {

    this._negociacoes = [];
    this._contexto = contexto;
    this._armadilha = armadilha;
    Object.freeze(this);
  }

  adiciona(negociacao) {

    this._negociacoes.push(negociacao);

    // ALTERAÇÃO AQUI
    this._armadilha.call(this._contexto, this);
  }

  // código anterior omitido

  esvazia() {

    this._negociacoes.length = 0;

    // ALTERAÇÃO AQUI
    this._armadilha.call(this._contexto, this);
  }
}
```

Se executarmos o código, o cadastro voltará a funcionar:

Figura 8.2: Formulário funcionando

Esta é uma segunda forma de resolver o problema gerado pelo dinamismo de `this`. Contudo há uma terceira maneira de solucionarmos o problema, porém menos verbosa que as duas que experimentamos. Veremos esta nova forma a seguir.

8.3 ARROW FUNCTION E SEU ESCOPO LÉXICO

A primeira mudança que faremos para escrevermos menos código é removermos o parâmetro `contexto` do `constructor()` de `Negociacoes`. Inclusive, voltaremos para a chamada direta de `this._armadilha` nos métodos `adiciona()` e `esvazia()`:

```
class Negociacoes {

  // não recebe mais contexto
  constructor(armadilha) {

    this._negociacoes = [];
    this._armadilha = armadilha;
    Object.freeze(this);
  }
}
```



```

adiciona(negociacao) {

    this._negociacoes.push(negociacao);

    // chamada direta da função, sem call()
    this._armadilha(this);
}

// código posterior omitido

esvazia() {

    this._negociacoes.length = 0;

    // chamada direta da função, sem call()
    this._armadilha(this);
}
}

```

Precisamos alterar no `constructor()` de `NegociacaoController` os parâmetros passados para `Negociacoes`, que passou a ser apenas a função de armadilha:

```

// client/app/controllers/NegociacaoController.js

class NegociacaoController {

    constructor() {

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

        // não recebe mais this como parâmetro,
        // somente a função!
        this._negociacoes = new Negociacoes(function(model) {

            console.log(this);
            this._negociacoesView.update(model);
        });

        // código posterior omitido
    }
}

```

Do jeito que está, nosso código não funcionará. Mas se trocarmos a `function` passada para o `constructor()` de `Negociacoes` por uma *arrow function*, teremos uma surpresa:

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

  constructor() {

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

    // passou um arrow function!

    this._negociacoes = new Negociacoes(model => {

      console.log(this);
      this._negociacoesView.update(model);
    });

    // código posterior omitido
  }
}
```

Um teste demonstra que nosso código funcionou, sendo muito menos verboso do que o anterior. Como isso é possível? Isto ocorre porque a *arrow function* não é apenas uma maneira sucinta de escrevermos uma função, ela também tem uma característica peculiar: o escopo de seu `this` é **léxico** (estático) em vez de dinâmico.

O `this` de uma *arrow function* obtém seu valor do "código ao redor", mantendo esse valor independente do lugar onde é chamado. É por isso que o `this` da função da armadilha passada aponta para a instância de `NegociacaoController`.

Nossa solução é funcional, mas podemos melhorá-la. É o que

veremos no próximo capítulo.

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/08>

ENGANARAM O CANGACEIRO, SERÁ?

"Jagunço não se escabreia com perda nem derrota - quase tudo para ele é o igual." - Grande Sertão: Veredas

Para liberarmos o desenvolvedor da responsabilidade de atualizar programaticamente a View sempre que o modelo for atualizado, nós colocamos a lógica de atualização em "armadilhas", funções que eram chamadas em métodos que alteravam o estado do modelo.

No entanto, esta solução deixa a desejar porque nos obriga a receber a armadilha em todas as nossas classes de modelo, além de termos de lembrar de chamá-la em cada método que altera o estado do modelo.

A propriedade `this._armadilha` de `Negociacoes` não tem qualquer relação com o domínio que a classe representa. Ou seja, ela está lá apenas por uma questão de infraestrutura. Relembremos do código:

```
// APENAS RELEMBRANDO!

// client/app/domain/negociacoes/Negociacoes.js

class Negociacoes {

  constructor(armadilha) {

    this._negociacoes = []

    // estranho no ninho,
    // sem relação com o domínio de negociações
    this._armadilha = armadilha;

    Object.freeze(this);
  }
}
```

Um modelo do domínio é uma das classes mais reutilizáveis de um sistema, justamente por não conter nenhuma mágica extra que não diga respeito ao problema do domínio que resolve. Contudo, nosso código parece não seguir essa recomendação.

Em nossa aplicação, se mais tarde quisermos usar nossos modelos com frameworks famosos do mercado, como Angular ou React, a propriedade `armadilha` não será mais necessária, mas ainda fará parte da definição da classe do modelo, obrigando os desenvolvedores a lidar com ela. E agora?

De um lado, queremos preservar o modelo, do outro, queremos facilitar a vida do desenvolvedor. Será que podemos conseguir o melhor dos dois mundos?

9.1 O PADRÃO DE PROJETO PROXY

Vimos que não é interessante poluirmos nosso modelo com a função que atualizará sua respectiva View e que ainda precisamos

de uma solução que atualize a View automaticamente toda vez que o modelo for alterado. Existe um padrão de projeto que pode nos ajudar a conciliar esses dois opostos, o **Proxy**.

O padrão de projeto **Proxy**, em poucas palavras, trata-se de "um farsante". Lidamos com um Proxy como se ele fosse a instância do objeto que estamos querendo manipular, por exemplo, uma instância de `Negociacoes`. Ele é uma casquinha que envolve a instância que desejamos manipular e, para cada propriedade e método presente nessa instância, o Proxy terá um correspondente.

Se chamarmos o método `adiciona()` no Proxy, ele delegará a chamada do mesmo método na instância que encapsula. Entre a chamada do método do Proxy e a chamada do método correspondente na instância, **podemos adicionar uma armadilha** - no caso da nossa aplicação, um código que atualizará nossa View. Dessa forma, não precisamos modificar a classe do modelo, apenas criar um Proxy que saiba executar o código que desejamos. Como implementá-lo?

A boa notícia é que não precisamos implementar esse padrão de projeto. O ECMAScript, a partir da versão 2015, já possui na própria linguagem um recurso de Proxy. Vamos aprender a utilizá-lo a seguir.

9.2 APRENDENDO A TRABALHAR COM PROXY

O primeiro passo é editarmos `NegociacaoController` e comentarmos temporariamente o trecho do código que não

funciona mais, aquele que passa a armadilha para o `constructor()` de `Negociacoes`:

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

  constructor() {

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

    // código comentado
    /*
    this._negociacoes = new Negociacoes(model => {
      console.log(this);
      this._negociacoesView.update(model);
    });
    */

    // código posterior omitido
```

Em seguida, vamos remover a armadilha passada para o `constructor()` de `Negociacoes`. Além disso, não podemos nos esquecer de remover as chamadas das armadilhas dos seus métodos. A classe no final ficará assim:

```
// client/app/domain/negociacao/Negociacoes.js

class Negociacoes {

  constructor() {

    this._negociacoes = []
    Object.freeze(this);
  }

  adiciona(negociacao) {
```

```

        this._negociacoes.push(negociacao);
    }

    paraArray() {

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

    get volumeTotal() {

        return this._negociacoes
            .reduce((total, negociacao) =>
                total + negociacao.volume, 0)
    }

    esvazia() {

        this._negociacoes.length = 0;
    }
}

```

Com a alteração que acabamos de fazer, nossa aplicação deixará de funcionar e receberemos no console a seguinte mensagem de erro:

```
Uncaught TypeError: Cannot read property 'paraArray' of undefined
```

Não se preocupe com ela. Deixaremos nossa aplicação quebrada enquanto aprendemos um novo recurso que ajudará a recompô-la.

Agora, faremos uma série de experimentos no console do Chrome para aprendermos a trabalhar com Proxy. Vamos deixar um pouco de lado a classe `Negociacoes` para trabalhar com a classe `Negociacao`, pois essa troca facilitará nossa aprendizagem.

Com o console do Chrome aberto, vamos executar as duas instruções a seguir:

```
// NO CONSOLE DO CHROME
```



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

A primeira instrução cria uma instância de `Negociacao`. A segunda cria uma instância de um `Proxy`. O `constructor()` do `Proxy` recebe como primeiro parâmetro a instância de `Negociacao` que queremos encapsular e, como segundo, um objeto no formato literal `{}`, que contém o código das armadilhas que desejamos executar. Por enquanto, não definimos nenhuma armadilha.

O `Proxy` criado possui todas as propriedades e métodos da classe que encapsula, tanto que se acessarmos no console a propriedade `proxy.valor`, estaremos por debaixo dos panos acessando a mesma propriedade na instância de `Negociacao` que encapsula:

```
// NO CONSOLE DO CHROME  
  
console.log(proxy.valor); // imprime 100  
console.log(proxy.quantidade); // imprime 1  
console.log(proxy.volume); // imprime 100
```

No entanto, não queremos que ninguém tenha acesso direto à instância encapsulada pelo nosso `Proxy`. Logo, podemos limpar nosso console e começar do zero com essa instrução:

```
// NO CONSOLE DO CHROME  
  
negociacao = new Proxy(new Negociacao(new Date(), 1, 100), {});
```

Usamos como nome de variável `negociacao` para mascarar nosso `Proxy`. Em seguida, como primeiro parâmetro do `constructor()` do nosso `Proxy`, instanciamos diretamente uma instância de `Negociacao`.

De nada adianta nosso Proxy se ele não é capaz de executar nossas armadilhas. Precisamos defini-las no objeto passado como segundo parâmetro para o Proxy, um objeto chamado **handler**.

Como vamos executar uma série de operações, será mais cômodo para nós escrever todas essas instruções dentro de uma nova tag `<script></script>`, que adicionaremos temporariamente em `client/index.html`. Ela deve ser a última tag `<script>` da página.

```
<!-- client/index.html -->
<!-- código posterior omitido -->
<!-- última tag script! -->

<script>

    const negociacao = new Proxy(new Negociacao(new Date(), 2, 1000), {

    });
</script>

<!-- código posterior omitido -->
```

Agora que temos tudo no lugar, vamos aprender a criar nossas armadilhas no handler passado para nosso Proxy.

9.3 CONSTRUINDO ARMADILHAS DE LEITURA

Para iniciarmos nossa jornada no mundo do Proxy, começaremos com algo bem modesto. Toda vez que as propriedades do Proxy forem lidas, faremos com que ele exiba uma mensagem indicando o nome da propriedade.

Para isso, em nosso handler, adicionaremos a propriedade

`get` que receberá como parâmetro a função que desejamos executar na leitura das propriedades do nosso Proxy. Vejam que a propriedade `get` não foi por acaso, ela indica claramente que estamos planejando um código para um **acesso de leitura**:

```
<!-- client/index.html -->
<!-- código posterior omitido -->

<script>

    const negociacao = new Proxy(new Negociacao(new Date(), 2, 1000), {

        get: function(target, prop, receiver) {

            console.log(`a propriedade "${prop}" caiu na armadilha!`);

        }

    });

    console.log(negociacao.quantidade);
    console.log(negociacao.valor);
</script>

<!-- código posterior omitido -->
```

A função passada para a propriedade `get` recebe três parâmetros especiais. O primeiro, `target`, é uma referência para o objeto encapsulado pelo Proxy, o objeto verdadeiro. O segundo parâmetro, `prop`, é uma string com o nome da propriedade que está sendo acessada. Por fim, `receiver` é uma referência para o próprio Proxy.

Podemos simplificar um pouco mais nosso código. A partir do ES2015 (ES6), podemos declarar propriedades de objetos que equivalem a funções dessa forma:

```
<!-- client/index.html -->
<!-- código posterior omitido -->
```

```

<script>

    const negociacao = new Proxy(new Negociacao(new Date(), 2, 1000), {

        // AQUI! não é mais get: function(...) {}

        get(target, prop, receiver) {

            console.log(`a propriedade "${prop}" caiu na armadilha!`);

        }

    });

    console.log(negociacao.quantidade);
    console.log(negociacao.valor);
</script>

<!-- código posterior omitido -->

```

Recarregando a página, vemos a saída do Console:

```

// EXIBIDO NO CONSOLE DO CHROME

a propriedade "quantidade" caiu na armadilha!
undefined
a propriedade "valor" caiu na armadilha!
undefined

```

A mensagem da nossa armadilha funcionou perfeitamente. Entretanto, as impressões de `console.log()` resultaram em `undefined` em vez de exibir os valores das propriedades. Por que isso aconteceu?

Toda armadilha, quando executada, deve se responsabilizar por definir o valor que será retornado para quem acessar a propriedade. Como não definimos retorno algum, o resultado foi `undefined`.

Façamos um teste para entendermos ainda melhor a

necessidade desse retorno:

```
<!-- client/index.html -->
<!-- código posterior omitido -->

<script>

    const negociacao = new Proxy(new Negociacao(new Date(), 2, 1000), {

        get(target, prop, receiver) {

            console.log(`a propriedade "${prop}" caiu na armadilha!`);

            // MUDANÇA AQUI! :)
            return 'Flávio';

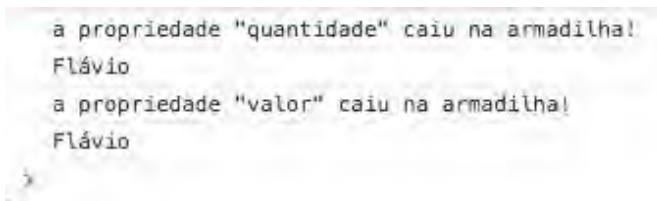
        }

    });

    console.log(negociacao.quantidade);
    console.log(negociacao.valor);
</script>

<!-- código posterior omitido -->
```

No exemplo anterior, por mais estranho que isso possa parecer, estamos retornando a string "Flávio". Olhando no console do Chrome, as instruções `console.log()` exibirão a string "Flávio", tanto para `negociacao.quantidade` quanto em `negociacao.valor`:



```
a propriedade "quantidade" caiu na armadilha!
Flávio
a propriedade "valor" caiu na armadilha!
Flávio
```

Figura 9.1: Return Flávio

Nosso handler retorna algo, mas ainda não é o esperado. Queremos que sejam retornados os respectivos valores das propriedades quantidade e valor do objeto encapsulado pelo Proxy.

A boa notícia é que temos todos os elementos necessários para isso. Quando temos um objeto JavaScript, podemos acessar suas propriedades através de `.` (ponto), ou utilizando um colchete que recebe a string com o nome da propriedade que desejamos acessar. Se temos `target` (a referência para nossa instância) e temos `prop` (a string com o nome da propriedade que está sendo acessada) podemos fazer `target[prop]`

```
<!-- client/index.html -->
<!-- código posterior omitido -->
<script>

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

        get(target, prop, receiver) {

            console.log(`a propriedade "${prop}" caiu na armadilha!`);

            // modificação aqui
            return target[prop];
        }
    });

    console.log(negociacao.quantidade);
    console.log(negociacao.valor);

</script>
<!-- código posterior omitido -->
```

```

    a propriedade "quantidade" caiu na armadilha!
    2
    a propriedade "valor" caiu na armadilha!
    100
  }
}

```

Figura 9.2: Acessando a propriedade do objeto encapsulado pelo Proxy

Nós vimos como executar um código toda vez que as propriedades de um objeto forem lidas. Contudo, isso não resolve o problema de atualização automática da View, porque queremos executar uma armadilha apenas quando as propriedades forem modificadas.

Para realizarmos um teste de escrita, não podemos simplesmente atribuir valores para `negociacao.quantidade` e `negociacao.valor`, porque eles são `getters`. Qualquer valor atribuído para eles é ignorado. Sendo assim, aplicaremos uma "licença poética" para acessar diretamente as propriedades privadas `_quantidade` e `_valor`. Estamos quebrando a convenção, mas é por uma boa causa, para podermos entender como Proxy funciona:

```

<!-- client/index.html -->
<!-- código posterior omitido -->
<script>

    const negociacao = new Proxy(new Negociacao(new Date(), 2, 10), {

        get(target, prop, receiver) {

            console.log(`a propriedade "${prop}" caiu na armadilha!`);

            return target[prop];

        }

    });

```

```

});

// mesmo acessando as propriedades somente leitura
// a armadilha não é disparada
negociacao._quantidade = 10
negociacao._valor = 100;

</script>
<!-- código posterior omitido -->

```

Realmente, nada acontece. Da mesma forma que aprendemos a executar armadilhas para leitura, aprendemos a executar armadilhas para escrita.

9.4 CONSTRUINDO ARMADILHAS DE ESCRITA

Para criarmos armadilhas de escrita, em vez de usarmos `get` em nosso handler, usamos `set`. Ele é bem parecido com o `get`, a diferença está no parâmetro extra `value` que ele recebe:

```

<!-- client/index.html -->
<!-- código posterior omitido -->
<script>

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

        get(target, prop, receiver) {

            console.log(`a propriedade "${prop}" caiu na armadilha!`);

            return target[prop];
        },

        set(target, prop, value, receiver) {

            /* nossa lógica entrará aqui */

        }
    });

```



```

    negociacao._quantidade = 10
    negociacao._valor = 100;

</script>
<!-- código posterior omitido -->

```

O parâmetro `value` é o valor que está sendo atribuído à propriedade que está sendo acessada no Proxy. Sendo assim, podemos implementar nossa armadilha dessa forma:

```

<!-- client/index.html -->
<!-- código posterior omitido -->
<script>

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

        get(target, prop, receiver) {

            console.log(`a propriedade "${prop}" caiu na armadilha!`);

            return target[prop];

        },

        set(target, prop, value, receiver) {

            console.log(`${prop} guarda ${target[prop]}, receberá ${value}`);

            // efetiva a passagem do valor!
            target[prop] = value;

        }

    });

    negociacao._quantidade = 10
    negociacao._valor = 100;

</script>
<!-- código posterior omitido -->

```

Um ponto que faz parte da especificação Proxy do ES2015 (ES6) é a necessidade de retornarmos `true` em nossa armadilha

para indicar que a operação foi bem-sucedida.

```
<!-- client/index.html -->
<!-- código posterior omitido -->
<script>

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

        get(target, prop, receiver) {

            console.log(`a propriedade "${prop}" caiu na armadilha!`);

            return target[prop];

        },

        set(target, prop, value, receiver) {

            console.log(`${prop} guarda ${target[prop]}, receberá ${value}`);

            // se for um objeto congelado, será ignorado
            target[prop] = value;

            // retorna true se conseguiu alterar a propriedade do
            objeto

            return target[prop] == value;

        }

    });

    // armadilha será disparada, mas os valores
    // não serão alterados, pois estão congelados

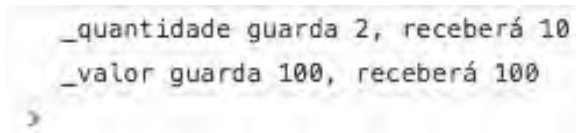
    negociacao._quantidade = 10
    negociacao._valor = 100;

</script>
<!-- código posterior omitido -->
```

A instrução `target[prop] = value` será ignorada para propriedades congeladas do objeto, é por isso que retornamos o resultado de `target[prop] == value` para sabermos se a

atribuição foi realizada com sucesso.

Verificando o console, temos:



```
_quantidade guarda 2, receberá 10
_valor guarda 100, receberá 100
```

Figura 9.3: Novo valor

Podemos escrever um pouco menos com auxílio de um recurso que aprenderemos a seguir.

9.5 REFLECT API

Podemos simplificar um pouco mais nosso código com auxílio da **API Reflect** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect) introduzida no ECMAScript 2015 (ES6). Esta API centraliza uma série de métodos estáticos que permitem a leitura, escrita e chamada de métodos e funções dinamicamente, um deles é o `Reflect.set()`.

O `Reflect.set()` recebe três parâmetros. O primeiro é a instância do objeto alvo da operação e o segundo o nome da propriedade que será alterada. O último parâmetro é o novo valor da propriedade. O método retornará `true` ou `false` caso tenha conseguido alterar ou não a propriedade do objeto:

```
<!-- client/index.html -->
<!-- código posterior omitido -->
<script>

    const negociacao = new Proxy(new Negociacao(new Date(), 2, 100
```

```

    }, {
        get(target, prop, receiver) {
            console.log(`a propriedade "${prop}" caiu na armadilha!`);
            return target[prop];
        },
        set(target, prop, value, receiver) {
            console.log(`${prop} guarda ${target[prop]}, receberá ${value}`);
            return Reflect.set(target, prop, value);
        }
    });

    negociacao._quantidade = 10
    negociacao._valor = 100;

</script>
<!-- código posterior omitido -->

```

A Reflect API possui outros métodos como o `Reflect.get()`. No entanto, sua introdução tornaria nosso código um pouco mais verboso, diferente do uso de `Reflect.set()` que nos poupou uma linha de código. Por exemplo, a instrução `return target[prop]` poderia ter sido substituída por `return Reflect.get(target, prop)`, mas a forma atual nos permite escrever uma quantidade de caracteres menor.

Agora que já compreendemos como um Proxy funciona, podemos voltar para `Negociacoes`, aliás, modelo cuja View precisamos atualizar.

9.6 UM PROBLEMA NÃO ESPERADO

Agora que já consolidamos nosso conhecimento sobre criação de Proxy, deixaremos o modelo `Negociacao` de lado para abordarmos o modelo `Negociacoes`, aquele cuja View precisamos atualizar automaticamente. Vamos realizar um teste criando um Proxy e, através dele, chamaremos o método `esvazia()`:

```
<!-- client/index.html -->
<!-- código posterior omitido -->
<script>

    // começando do zero, apagando o código anterior que existia

    const negociacoes = new Proxy(new Negociacoes(), {

        set(target, prop, value, receiver) {

            console.log(`${prop} guarda ${target[prop]}, receberá
            ${value}`);
            return Reflect.set(target, prop, value);
        }
    });

    negociacoes.esvazia();

</script>
<!-- código posterior omitido -->
```

Recarregando nossa página e observando o resultado no console, vemos apenas a mensagem de erro que já era exibida antes por ainda não termos terminado nossa aplicação. Nenhuma outra mensagem é exibida, ou seja, nossa armadilha não foi executada! Será que o mesmo acontece com o método `adiciona()`? Vamos testá-lo.

Ainda em `client/index.html`, vamos substituir a chamada de `negociacoes.esvazia()` por uma chamada de `negociacoes.adiciona()`, passando uma nova negociação:

```

<!-- client/index.html -->
<!-- código posterior omitido -->
<script>

    const negociacoes = new Proxy(new Negociacoes(), {

        set(target, prop, value, receiver) {

            console.log(`${prop} guarda ${target[prop]}, receberá
            ${value}`);
            return Reflect.set(target, prop, value);
        }
    });

    // agora, chamando o método adiciona
    negociacoes.adiciona(new Negociacao(new Date(), 1, 100));

</script>
<!-- código posterior omitido -->

```

Recarregando nossa página e observando o resultado no console, nada acontece.

Isso acontece porque nossa armadilha só é disparada apenas quando houver atribuição de valor nas propriedades do objeto encapsulado pelo Proxy. Os métodos `esvazia()` e `adiciona()` não realizam qualquer atribuição. O primeiro faz `this._negociacoes.length = 0` e o segundo adiciona um novo elemento em `this._negociacoes` através do método `push()` do próprio array. Vamos relembrar dos métodos!

```

// RELEMBRADO O CÓDIGO, NÃO HÁ MODIFICAÇÃO AQUI

// client/app/domain/negociacoes/Negociacoes.js

class Negociacoes {

    // código omitido

    adiciona(negociacao) {

```

```

        // não há atribuição aqui!
        this._negociacoes.push(negociacao);
    }

    // código omitido

    esvazia() {

        // não há atribuição aqui!
        this._negociacoes.length = 0;
    }

    // código posterior omitido

```

Aliás, não faz sentido `this._negociacoes` receber uma nova atribuição, pois é uma propriedade congelada. E agora? Será que existe alguma solução para esse problema?

Uma solução é termos capazes de indicar quais métodos da classe devem disparar nossas armadilhas. Com essa estratégia, os métodos `adiciona()` e `esvazia()` não precisam mais depender de uma atribuição a alguma propriedade da instância de `Negociacoes` para que suas respectivas armadilhas sejam disparadas.

9.7 CONSTRUINDO ARMADILHAS PARA MÉTODOS

Vamos dar início à nossa solução cangaceira para solucionar o problema da seção anterior. O primeiro ponto é entendermos que, por debaixo dos panos, quando o JavaScript chamada um método, ele realiza um `get` para obter uma referência ao método, e depois um `apply` para passar seus parâmetros. Por enquanto, vamos nos ater ao papel do `get`.

Vamos substituir o `set` que temos no `Proxy` que criamos em `client/index.html` por um `get`. Lembre-se de que `get` não recebe o parâmetro `value` :

```
<!-- client/index.html -->
<!-- código posterior omitido -->
<script>

    const negociacoes = new Proxy(new Negociacoes(), {

        get(target, prop, receiver) {

            /* e agora ? */

        }

    });

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

</script>
<!-- código posterior omitido -->
```

O próximo passo é verificarmos se a propriedade que estamos interceptando é uma função/método, pois a estratégia que estamos construindo é para lidar com esse tipo de dado. Fazemos isso através da função `typeof()` :

```
<!-- client/index.html -->
<!-- código posterior omitido -->

<!-- NÃO TESTE O CÓDIGO AINDA! NÃO FUNCIONARÁ, NÃO TERMINAMOS -->
<script>

    const negociacoes = new Proxy(new Negociacoes(), {

        get(target, prop, receiver) {

            if(typeof(target[prop]) == typeof(Function) {

                /* falta código ainda */

            }

        }

    });

</script>
```



```

    });

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

</script>
<!-- código posterior omitido -->

```

Se deixarmos do jeito que está, aplicaremos nossa estratégia para qualquer método, e não é isso o que queremos. Não faz sentido o método `paraArray()` de `Negociacoes` disparar atualizações de `View`, apenas métodos que alteram o estado.

9.8 UMA PITADA DO ES2016 (ES7)

Além de verificarmos o tipo da propriedade, vamos testar se ela está incluída na lista de métodos que desejamos interceptar:

```

<!-- client/index.html -->
<!-- código posterior omitido -->
<script>

    const negociacoes = new Proxy(new Negociacoes(), {

        get(target, prop, receiver) {

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

                /* falta código ainda */

            }

        }

    });

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

</script>
<!-- código posterior omitido -->

```

Através do método `includes()` do array, verificamos se a

propriedade que está sendo interceptada consta em nossa lista. Aliás, o método `includes()` foi introduzido no **ES2016** (ES7), uma versão posterior ao ES2015 (ES6).

A especificação ES2016 (ES7) define `Array.prototype.includes` e o operador exponencial apenas.

Agora, se a propriedade não for um método/função da nossa lista, deixamos o curso acontecer naturalmente.

```
<!-- client/index.html -->
<!-- código posterior omitido -->
<script>

    const negociacoes = new Proxy(new Negociacoes(), {

        get(target, prop, receiver) {

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

                /* falta código ainda */

            } else {

                // realizando um get padrão
                return target[prop];
            }
        }
    });

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

</script>
<!-- código posterior omitido -->
```

Agora que temos tudo encaixado, precisamos implementar o código que será executado pela armadilha dos métodos que decidimos considerar. É aqui que entra o pulo do gato, em inglês, *cat jump*.

Para os métodos listados em nosso array de nomes de métodos a serem interceptados, vamos fazer com que sua leitura retorne uma nova `function` em vez de uma **arrow function**, com o código que desejamos executar, inclusive aquele que aplicará os parâmetros da função (caso existam):

```
<!-- client/index.html -->
<!-- código posterior omitido -->
<script>

    const negociacoes = new Proxy(new Negociacoes(), {

        get(target, prop, receiver) {

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

                // retorna uma nova função com contexto dinâmico
                // antes que o apply aplicado por padrão pelo
                // interpretador seja chamado.

                return function() {

                    console.log(`${prop}` disparou a armadilha`);

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

                }

            } else {

                // se é propriedade, retorna o valor normalmente
                return target[prop];

            }

        }

    })
```

```
});  
  
negociacoes.adiciona(new Negociacao(new Date(), 1, 100));  
  
</script>  
<!-- código posterior omitido -->
```

Vamos repassar pela mágica que fizemos. Lembrem-se de que o interpretador do JavaScript primeiro realiza um `get` para ler o método, e depois executa um `apply` por debaixo dos panos para executá-lo, passando seus parâmetros. A `function()` que estamos retornando substituirá o método original antes que o `apply` entre em ação. Nela, adicionamos o código da nossa armadilha.

Através de `target[prop].apply` (não confunda com o `apply` que o interpretador fará), estamos chamando o método usando como contexto o próprio `target`. O método `apply()` é parecido com o método `call()` que já vimos, a diferença é que o primeiro recebe seus parâmetros em um array. Perfeito, mas temos um problema.

Nossa `function()` substituta precisa saber se o método original possui ou não parâmetros para passá-los na chamada de `target[prop].apply`. Conseguimos essa informação através de **arguments**.

A variável implícita `arguments` possui estrutura semelhante a um array. Ela está presente em todo método ou função, e dá acesso aos parâmetros passados, mesmo que eles não sejam explicitados. Vejamos um exemplo através do console:

```
// TESTAR NO CONSOLE, EXEMPLO APENAS
```

```
// NÃO RECEBE PARÂMETRO ALGUM

function exibeNomeCompleto() {
    alert(`Nome completo: ${arguments[0]} ${arguments[1]}`);
}

// funciona!
exibeNomeCompleto('Flávio', 'Almeida');
```

O `arguments` foi fundamental para nossa solução, pois não teríamos como prever quantos parâmetros cada método interceptado receberia. Agora que temos nossa solução pronta, vamos alterar `NegociacaoController` para que faça uso dela.

9.9 APLICANDO A SOLUÇÃO EM NEGOCIACAOCONTROLLER

Vamos alterar a classe `NegociacaoController` atribuindo à propriedade `this._negociacoes` um `Proxy`, usando a lógica que já implementamos em `client/index.html`. É praticamente um corta e cola, a diferença é que fazemos uma chamada a `this._negociacoesView.update()` logo abaixo do `console.log()` em que o método disparou a armadilha:

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

    constructor() {

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

        // agora é um proxy!
        this._negociacoes = new Proxy(new Negociacoes(), {
```

```

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

                return function() {

                    console.log(`"${prop}" disparou a arm
adilha`);

                    target[prop].apply(target, arguments)

                    // target é a instância real de Negoc
iacoes

                    // contudo, TEREMOS PROBLEMAS AQUI!
                    this._negociacoesView.update(target);

                }

            } else {

                return target[prop];

            }

        }

    });

    // código posterior omitido
}

```

Não precisamos mais da tag `<script>` com nossos testes, podemos removê-la. Por fim, quando executamos uma nova negociação, recebemos esta mensagem de erro no console:

```
// EXIBIDO NO CONSOLE DO CHROME
```

```
Uncaught TypeError: Cannot read property 'update' of undefined
```

Isso acontece porque o `this` de `this._negociacoesView.update(target)` é o Proxy, e não `NegociacaoController`. Se tivéssemos usado *arrow function*,

resolveríamos esse problema, mas precisamos que o `this` da função seja dinâmico; caso contrário, nossa solução não funcionará.

Uma solução que já vimos antes é guardar uma referência de `this` na variável `self`, para que possamos utilizá-la depois na armadilha:

```
class NegociacaoController {

  constructor() {

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

    // guardando uma referência
    // para a instância de NegociacaoController
    const self = this;

    this._negociacoes = new Proxy(new Negociacoes(), {

      get(target, prop, receiver) {

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

          return function() {

            console.log(`"${prop}" disparou a arm
adilha`);

            target[prop].apply(target, arguments)

            // AGORA USA SELF!
            self._negociacoesView.update(target);

          }

        } else {
```

```

        return target[prop];
    }
    });

    this._negociacoesView = new NegociacoesView('#negociacoes');

    this._negociacoesView.update(this._negociacoes);

    this._mensagem = new Mensagem();
    this._mensagemView = new MensagemView('#mensagemView');
    this._mensagemView.update(this._mensagem);
}
// código posterior omitido

```

Nossa solução funciona, mas imagine aplicá-la para o modelo Mensagem ? Teríamos de repetir muito código, inclusive sua manutenção não será lá uma das mais fáceis. Então, antes de generalizarmos nossa solução, no próximo capítulo vamos pedir a ajuda de outro padrão de projeto para colocar o sertão em ordem.

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/09>

CÚMPLICE NA EMBOSCADA

"... o mais importante e bonito, do mundo, é isto: que as pessoas não estão sempre iguais, ainda não foram terminadas – mas que elas vão sempre mudando. Afinam ou desafinam. Verdade maior. É o que a vida me ensinou." - Grande Sertão: Veredas

No capítulo anterior, aplicamos o padrão de projeto Proxy para deixar clara uma separação entre o modelo e o código de infraestrutura que criamos para mantê-lo sincronizado com a View. No entanto, vimos que podemos aplicar outro padrão de projeto para tornar nossa solução mais genérica. Aplicaremos o padrão de projeto **Factory**.

10.1 O PADRÃO DE PROJETO FACTORY

O padrão de projeto Factory auxilia na criação de objetos complexos, encapsulando os detalhes de criação desses objetos, como no caso dos nossos proxies.

Vamos criar o arquivo `client/app/util/ProxyFactory.js`. A pasta `util` não é um erro de digitação. É uma pasta na qual ficarão os arquivos que não fazem parte do domínio nem da interface (UI) da aplicação.

A classe `ProxyFactory` terá o método estático `create`, seu único método:

```
// client/app/util/ProxyFactory.js

class ProxyFactory {

  static create(objeto, props, armadilha) {

  }

}
```

O método `create` recebe três parâmetros. O primeiro é o objeto alvo do proxy, o segundo é um array com os métodos que desejamos interceptar e, por fim, o último é a função `armadilha` que desejamos executar para os métodos presentes no array `props`.

Antes de mais nada, vamos importar o script que deve vir antes da importação de `NegociacaoController`, em `client/index.html`:

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

<script src="app/domain/negociacao/Negociacao.js"></script>
<script src="app/controllers/NegociacaoController.js"></script>
<script src="app/ui/converters/DateConverter.js"></script>
<script src="app/domain/negociacao/Negociacoes.js"></script>
<script src="app/ui/views/View.js"></script>
<script src="app/ui/views/NegociacoesView.js"></script>
<script src="app/ui/models/Mensagem.js"></script>
<script src="app/ui/views/MensagemView.js"></script>
```

```

<!-- importou o ProxyFactory -->
<script src="app/util/ProxyFactory.js"></script>

<script src="app/app.js"></script>
<!-- código posterior omitido -->

```

Agora, vamos mover o código de `NegociacaoControler`, que cria nosso proxy, para dentro do método `create`. Então, realizaremos os ajustes necessários para que ele leve em consideração os parâmetros do método:

```

// client/app/util/ProxyFactory.js

class ProxyFactory {

  static create(objeto, props, armadilha) {

    // recebe objeto como parâmetro

    return new Proxy(objeto, {

      get(target, prop, receiver) {
        // usa o array props para realizar o includes
        if(typeof(target[prop]) == typeof(Function)
          && props.includes(prop)) {

          return function() {

            console.log(`"${prop}" disparou a armadilha`);

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

            // executa a armadilha que recebe
            // o objeto original

            armadilha(target);

          }

        } else {

          return target[prop];

        }

      }

    });
  }
}

```

```

    }
  });
}
}

```

Não precisamos mais da tag `<script>` , na qual realizávamos nossos testes. **Remova a tag** de `client/index.html` .

Agora que temos nosso `ProxyFactory` , vamos utilizá-lo para criar nosso `Proxy` em `NegociacaoController` . Veja que não precisaremos usar o `self` para termos acesso à instância de `NegociacaoController` em nossa armadilha, porque dessa vez estamos usando *arrow function*:

```

// client/app/controllers/NegociacaoController.js

class NegociacaoController {

  constructor() {

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

    // criando o proxy com auxílio da nossa fábrica!

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

    this._negociacoesView = new NegociacoesView('#negociacoes');

    this._negociacoesView.update(this._negociacoes);

    this._mensagem = new Mensagem();
    this._mensagemView = new MensagemView('#mensagemView');
    this._mensagemView.update(this._mensagem);

  }
}

```

```
// código posterior omitido
```

Veja como conseguimos melhorar em muito a legibilidade do nosso código e, por conseguinte, sua manutenção.

Agora, faremos a mesma coisa com o modelo `Mensagem`. Mas no caso desse modelo, queremos monitorar o acesso à propriedade `texto`:

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

  constructor() {

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

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

    this._negociacoesView = new NegociacoesView('#negociacoes');
    this._negociacoesView.update(this._negociacoes);

    // criando o proxy com auxílio da nossa fábrica!

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

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

```
// código posterior omitido
```

Por fim, vamos alterar os métodos `adiciona()` e `apaga()`, removendo a instrução que realiza o update manualmente da View de Mensagem:

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

  constructor() {

    // código omitido
  }

  adiciona(event) {

    event.preventDefault();
    this._negociacoes.adiciona(this._criaNegociacao());
    this._mensagem.texto = 'Negociação adicionada com sucesso'
;

    // não chama mais o update da view de Mensagem

    this._limpaFormulario();
  }

  // código posterior omitido

  apaga() {

    this._negociacoes.esvazia();
    this._mensagem.texto = 'Negociações apagadas com sucesso'
;

    // não chama mais o update da view de Mensagem
  }
}
```

Se recarregarmos a página no navegador, veremos que o formulário funciona corretamente, **mas não atualizou a mensagem**. O que será que aconteceu?

10.2 NOSSO PROXY AINDA NÃO ESTÁ 100%!

O `ProxyFactory` só está preparado para interceptar métodos; ele não está preparado para interceptar propriedades.

Na classe `Mensagem`, `get texto()` é um *getter* e o `ProxyFactory` não entende que precisa interceptar com o `get`. Já que os *getters* e *setters* são acessados como propriedades, temos também de colocar no `ProxyFactory` um código para lidarmos com propriedades. Para isto, adicionaremos um `set` no handler do nosso proxy:

```
// client/app/util/ProxyFactory.js

class ProxyFactory {

  static create(objeto, props, armadilha) {

    // recebe um objeto como parâmetro
    return new Proxy(objeto, {

      get(target, prop, receiver) {

        // código omitido!

      },

      // NOVIDADE!
      set(target, prop, value, receiver) {

        const updated = Reflect.set(target, prop, value)

        ;

        // SÓ EXECUTAMOS A ARMADILHA
        // SE FIZER PARTE DA LISTA DE PROPS

        if(props.includes(prop)) armadilha(target);

        return updated;

      }

    });

  }

}
```

```

    });
  }
}

```

Vamos recarregar a página no navegador e preencher os campos do formulário. Tudo estará funcionando em nossa aplicação.

Ainda podemos realizar uma melhoria na legibilidade do nosso `ProxyFactory` de último segundo. Nele temos a lógica que determina se a propriedade é uma função, dentro do `if`. Podemos isolar essa comparação em um novo método estático chamado `ehFuncao()`, e utilizá-lo na condição para deixar clara nossa intenção.

Com esta alteração, nosso `ProxyFactory` final estará assim:

```

// client/app/util/ProxyFactory.js

class ProxyFactory {

  static create(objeto, props, armadilha) {

    return new Proxy(objeto, {

      get(target, prop, receiver) {

        // USANDO O MÉTODO ESTÁTICO

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

          return function() {

            console.log(`${prop}` disparou a armadilha`);

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

          }
        }
      }
    });
  }
}

```



```

        } else {

            return target[prop];
        }
    },

    set(target, prop, value, receiver) {

        const updated = Reflect.set(target, prop, value)

;
        if(props.includes(prop)) armadilha(target);
        return updated;
    }

    });
}

// NOVO MÉTODO ESTÁTICO
static _ehFuncao(fn) {

    return typeof(fn) == typeof(Function);
}
}

```

Só alteramos a maneira pela qual organizamos nosso código, sem modificar seu comportamento. Mas aí vem aquela mesma pergunta de sempre: será que podemos melhorar nosso código?

10.3 ASSOCIANDO MODELO E VIEW ATRAVÉS DA CLASSE BIND

No capítulo anterior, criamos nosso `ProxyFactory` que encapsulou grande parte da complexidade para criarmos proxies. Isso possibilitou o reuso de todo o código que escrevemos sem grandes complicações. No entanto, temos dois pontos que deixam a desejar.

O primeiro ponto é precisarmos lembrar de realizar o update

manual da View na inicialização do `constructor()` de `NegociacoesController`. Vejamos:

```
// REVISANDO O CÓDIGO APENAS!

// client/app/controllers/NegociacaoController.js

class NegociacaoController {

  constructor() {

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

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

    this._negociacoesView = new NegociacoesView('#negociacoes');

    // PRECISA CHAMAR O UPDATE

    this._negociacoesView.update(this._negociacoes);

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

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

    // PRECISA CHAMAR O UPDATE
    this._mensagemView.update(this._mensagem);
  }
}
```

Além disso, se observarmos atentamente o código anterior, o `update` da View passado para a armadilha do Proxy é praticamente

idêntico, só mudando a View que chamará o método `update` . Vamos nos livrar dessas responsabilidades.

O que fizemos até agora se assemelha a um mecanismo de *Data binding* (que traduzido para o português, significa "ligação de dados"). Associamos um modelo com a View, de modo que mudanças no modelo atualizam automaticamente a View. Vamos criar a classe `Bind` , que será a responsável em executar as duas ações que o programador ainda precisa realizar.

Criaremos o arquivo `client/app/util/Bind.js` , onde vamos declarar a classe `Bind` . Em seu `constructor()` , receberemos três parâmetros. O primeiro será o modelo da associação, o segundo, a View que deve ser atualizada a cada mudança do modelo e, por fim, o terceiro será o array com os nomes de propriedades e métodos que devem disparar a atualização.

```
// client/app/util/Bind.js

class Bind {

  constructor(model, view, props) {

  }

}
```

Agora, no `constructor()` de `Bind` , vamos criar um proxy através do nosso `ProxyFactory` , repassando os parâmetros recebidos para o método `create()` da factory:

```
// client/app/util/Bind.js

class Bind {

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

```

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

```

Sabemos que toda View em nossa aplicação possui o método `update()` que recebe um modelo, por isso, estamos chamando `view.update(model)` na armadilha do proxy. Retiramos essa responsabilidade do desenvolvedor.

Agora, assim que o proxy é criado, vamos forçar uma chamada ao método `update()` da View, removendo mais uma responsabilidade de desenvolvedor!

```

// client/app/util/Bind.js

class Bind {

    constructor(model, view, props) {

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

        view.update(model);
    }
}

```

Por fim, por mais estranho que isso possa parecer, caso o leitor tenha vindo de outras linguagens como Java ou C#, faremos o `constructor()` de `Bind` retornar o proxy que criamos. Em JavaScript, um `constructor()` pode retornar um objeto de um tipo diferente da classe à qual pertence:

```

// client/app/util/Bind.js

class Bind {

    constructor(model, view, props) {

```

```

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

    view.update(model);

    return proxy;
  }
}

```

Vamos importar o script que declara Bind em client/index.html . Mas muito cuidado, pois NegociacaoController depende de Bind , que depende de ProxyFactory . Chegará um momento neste livro em que você será libertado desse grande problema na linguagem JavaScript, mas ainda não é a hora.

```

<!-- client/index.html -->
<!-- código anterior omitido -->
<script src="app/domain/negociacao/Negociacao.js"></script>
<script src="app/controllers/NegociacaoController.js"></script>
<script src="app/ui/converters/DateConverter.js"></script>
<script src="app/domain/negociacao/Negociacoes.js"></script>
<script src="app/ui/views/View.js"></script>
<script src="app/ui/views/NegociacoesView.js"></script>
<script src="app/ui/models/Mensagem.js"></script>
<script src="app/ui/views/MensagemView.js"></script>
<script src="app/util/ProxyFactory.js"></script>

<!-- importou aqui! -->
<script src="app/util/Bind.js"></script>

<script src="app/app.js"></script>
<!-- código posterior omitido -->

```

Para terminar, alteraremos NegociacaoController e usaremos nossa classe Bind para fazer a associação entre o modelo e a View:

```

// client/app/controllers/NegociacaoController.js

```

```

class NegociacaoController {

    constructor() {

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

        this._negociacoes = new Bind(
            new Negociacoes(),
            new NegociacoesView('#negociacoes'),
            ['adiciona', 'esvazia']
        );

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

    // código posterior omitido

```

Recarregando a página, tudo continua funcionando como antes, só que dessa vez simplificamos bastante a quantidade de código no `constructor()` de `Negociacoes`. Não foi mais necessário declarar as propriedades `_negociacaoView` nem `_mensagemView`, nem mesmo foi necessário chamar o `update()` manualmente das Views na inicialização de `NegociacaoController`.

Através de `Bind`, passamos a instância do modelo, a instância da View e as propriedades ou métodos que desejamos ativar a atualização automática. Nem mesmo foi necessário escrevermos a linha que realiza a atualização da view a partir do modelo.

Para fechar com chave de ouro, que tal enxugarmos uma

gotinha no código que acabamos de escrever?

10.4 PARÂMETROS REST

Em nosso `Bind`, o nome das propriedades e dos métodos que desejamos executar nossas armadilhas foram passados dentro de um array. Apesar de ainda não funcionar, vamos passar cada parâmetro individualmente, abdicando do array:

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

  constructor() {

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

    // não passamos mais "adiciona" e "esvazia" dentro de um
array
    this._negociacoes = new Bind(
      new Negociacoes(),
      new NegociacoesView('#negociacoes'),
      'adiciona', 'esvazia'
    );

    // não passamos mais "texto" dentro de um array
    this._mensagem = new Bind(
      new Mensagem(),
      new MensagemView('#mensagemView'),
      'texto'
    );
  }
}
```

Agora, para que nosso código funcione, vamos alterar a classe `Bind` adicionando três pontos (`...`) antes do parâmetro `props`:

```
// client/app/util/Bind.js

class Bind {

  // ... antes de props
  constructor(model, view, ...props) {

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

    view.update(model);

    return proxy;
  }
}
```

Quando adicionamos `... antes` do nome do último parâmetro, estamos indicando que todos os parâmetros a partir do terceiro, inclusive, serão considerados como fazendo parte de um array . Isso torna reflexível a quantidade de parâmetros que um método ou função pode receber. Mas atenção, só podemos usar o REST operator no último parâmetro. Aliás, este recurso foi introduzido no ES2015 (ES6) em diante.

Até aqui criamos um mecanismo de data binding para simplificar ainda mais nosso código. Podemos dar a tarefa como concluída. Todavia, há um problema antigo que voltará a nos assombrar no próximo capítulo, o problema com datas!

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/10>

DATA DOS INFERNOS!

"O correr da vida embrulha tudo, a vida é assim: esquenta e esfria, aperta e daí afrouxa, sossega e depois desinquieta. O que ela quer da gente é coragem." - Grande Sertão: Veredas

Nossa aplicação continua funcionando como esperado, mas precisamos resolver ainda um problema relacionando ao campo da data do nosso formulário.

11.1 O PROBLEMA COM O INPUT DATE

O `input` do tipo `date` que utilizamos não faz parte da especificação da W3C, sendo um tipo de elemento criado pela equipe do Google Chrome. Como o `input` do tipo `date` não faz parte da especificação, não temos a garantia de que esteja presente nos mais diversos navegadores do mercado. Aliás, esse é o único motivo que nos prende fortemente a este navegador.

Precisamos alterar nossa aplicação para que utilize em seu lugar um `input` do tipo `text`. Primeiro, vamos alterar `client/index.html` e mudar o tipo do `input`:

```

<!-- client/index.html -->
<!-- código anterior omitido -->
<div class="form-group">
  <label for="data">Data</label>

  <!-- era date, passou a ser text -->

  <input type="text" id="data" class="form-control" required au
tofocus/>
</div>
<!-- código posterior omitido -->

```

Contudo, essa alteração não é suficiente. Temos de alterar a lógica do método `paraData()` da classe `DateConverter`. Até o momento, sua implementação está assim:

```

// client/app/ui/converters/DateConverter.js

class DateConverter {

  // código omitido

  static paraData(texto) {

    if(!/^d{4}-d{2}-d{2}$/.test(texto))
      throw new Error('Deve estar no formato aaaa-mm-dd');

    return new Date(...texto.split('-')
      .map((item, indice) => item - indice % 2));
  }
}

```

Como não estamos mais usando o `input` do tipo `date`, não recebemos mais a string no formato `aaaa-mm-dd`. Recebemos exatamente como o usuário digitou no formulário.

11.2 AJUSTANDO NOSSO CONVERTER

Nosso primeiro passo será ajustar nossa expressão regular e a

mensagem de erro, caso o texto a ser convertido não use o padrão:

```
// client/app/ui/converters/DateConverter.js

class DateConverter {

  // código omitido

  static paraData(texto) {

    // NOVA EXPRESSÃO REGULAR
    // A MENSAGEM DE ERRO TAMBÉM MUDOU

    if(!/\d{2}\d{2}\d{4}/.test(texto))
      throw new Error('A data deve estar no formato dd/mm/aa');

    return new Date(...texto.split('-').map((item, indice) =>
item - indice % 2));
  }
}
```

A alteração ainda não é suficiente, precisamos modificar a lógica que desmembra nossa string. O primeiro passo é trocar o separador para `split('/')`. Agora, precisamos inverter a ordem dos elementos para ano, mês e dia. Antes não era necessário, porque a string recebida já estava nessa ordem. Resolvemos isso facilmente através da função `reverse()` que todo array possui:

```
// client/app/ui/converters/DateConverter.js

class DateConverter {

  // código omitido

  static paraData(texto) {

    // NOVA EXPRESSÃO REGULAR
    // A MENSAGEM DE ERRO TAMBÉM MUDOU

    if(!/\d{2}\d{2}\d{4}/.test(texto))
      throw new Error('A data deve estar no formato dd/mm/aa
```

```

aa');

// MUDOU O SEPARADOR PARA / E USOU REVERSE()

return new Date(...texto.split('/')
    .reverse()
    .map((item, indice) =>
        item - indice % 2));
}
}

```

Pronto, isso é suficiente para que nossa aplicação funcione em outros navegadores que não suporte o `input` do tipo `date`. Mas ainda precisamos resolver um problema que já existia mesmo quando usávamos este `input`.

Se digitarmos uma data inválida, nosso validador lançará uma exceção que será apresentada apenas no console do navegador. Nenhuma mensagem informando o usuário sobre a entrada da data errada é exibida para o usuário, e ele fica sem saber o que aconteceu. Chegou a hora de lidarmos com essa questão.

11.3 LIDANDO COM EXCEÇÕES

Podemos capturar exceções em JavaScript lançadas com `throws` através do bloco `try`, e tratá-las no bloco `catch`. No bloco da instrução `try`, temos uma ou mais instruções que potencialmente podem lançar uma exceção. Caso alguma exceção seja lançada, o fluxo do código é direcionado para o bloco do `catch` que recebe como parâmetro um objeto com informações da exceção lançada.

Vamos alterar o método `adiciona()` de `NegociacoesController` para usarmos essa nova sintaxe:

```
// client/app/controllers/NegociacaoController.js
// código anterior omitido

adiciona(event) {

    try {

        event.preventDefault();
        this._negociacoes.adiciona(this._criaNegociacao());
        this._mensagem.texto = 'Negociação adicionada com sucesso';
    ;
        this._limpaFormulario();

    } catch(err) {

        console.log(err);
        this._mensagem.texto = err.message;
    }
}
// código posterior omitido
```

Veja que, no bloco do `catch`, a primeira coisa que fazemos é logar a informação do erro que foi lançado, para logo em seguida atribuímos o texto do erro à `this._mensagem.texto`, através de `err.message`.

Se tentarmos incluir uma nova data inválida, será exibida uma mensagem de erro. Essa validação poderia ser feita de diversas formas, mas a forma atual possibilita introduzir certos conceitos importantes para quem trilha o caminho de um cangaceiro em JavaScript.

Tudo muito bonito, mas há um detalhe que pode passar despercebido por um ninja, mas nunca por um cangaceiro. Vamos alterar a última instrução do método `adiciona()` forçando um erro de digitação:

```
// client/app/controllers/NegociacaoController.js
// código anterior omitido
```

```

adiciona(event) {

    try {

        event.preventDefault();
        this._negociacoes.adiciona(this._criaNegociacao());
        this._mensagem.texto = 'Negociação adicionada com sucesso';

        // ATENÇÃO, FORÇANDO UM ERRO
        // CHAMANDO UM MÉTODO QUE NÃO EXISTE
        this._limpaForm();

    } catch(err) {

        console.log(err);
        this._mensagem.texto = err.message;
    }
}

// código posterior omitido

```

O que acontecerá ao incluirmos negociações? A mensagem de erro será exibida para o usuário:



Figura 11.1: Mensagem de erro que não deveria aparecer

Só faz sentido exibir para o usuário mensagens de alto nível, não mensagem de erro de sintaxe. O problema é que não temos como saber o tipo do erro no bloco do `catch`.

Uma solução é criarmos nossas próprias exceções e, com ajuda

da instrução `instanceof` , perguntaríamos se a exceção lançada é do tipo em que estamos interessados. Vamos alterar o método `adiciona()` :

```
// client/app/controllers/NegociacoesController.js

// código posterior omitido
adiciona(event) {

    try {

        event.preventDefault();
        this._negociacoes.adiciona(this._criaNegociacao());
        this._mensagem.texto = 'Negociação adicionada com sucesso';
    ;

        this._limpaForm();

    } catch(err) {

        console.log(err);
        console.log(err.stack);

        // TESTA SE O TIPO DO ERRO É DE DATA,
        // SE FOR, EXIBE MENSAGEM PARA O USUÁRIO

        if(err instanceof DataInvalidaException) {

            this._mensagem.texto = err.message;

        } else {

            // mensagem genérica para qualquer problema que possa
            acontecer

            this._mensagem.texto = 'Um erro não esperado acontece
            u. Entre em contato com o suporte';
        }
    }
}
```

O código anterior **não funcionará**, pois ainda não criamos a classe `DataInvalidaException` . Porém, isso não nos impede de

analisá-lo. No bloco `catch`, verificamos através da instrução `instanceof` se a exceção é uma instância de `DataInvalidaException`, e exibimos a mensagem de erro para o usuário. Se o erro não for da classe que estamos testando, apenas exibimos a mensagem no console.

Agora que sabemos como nosso código se comportará, chegou a hora de criarmos nossa classe `DataInvalidaException`, porque quem sabe faz hora e não espera acontecer!

11.4 CRIANDO NOSSA PRÓPRIA EXCEÇÃO: PRIMEIRA TENTATIVA

Vamos criar nossa própria classe de erro `DataInvalidaException` herdando de `Error`. Criaremos o arquivo

`client/app/ui/converters/DataInvalidaException.js`, no qual declararemos nossa classe, que começará assim:

```
// client/app/ui/converters/DataInvalidaException.js

class DataInvalidaException extends Error {

  constructor() {

    super('A data deve estar no formato dd/mm/aaaa');
  }
}
```

Como a classe filha `DataInvalidaException` possui um `constructor()` diferente da classe pai `Error`, somos obrigados a chamar `super()`, passando a informação necessária para o `constructor()` do pai, a mensagem *"A data deve estar no formato dd/mm/aaaa"*.

Agora, vamos importar o script da classe que criamos em `client/index.html` como último script:

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

<script src="app/domain/negociacao/Negociacao.js"></script>
<script src="app/controllers/NegociacaoController.js"></script>
<script src="app/ui/converters/DateConverter.js"></script>
<script src="app/domain/negociacao/Negociacoes.js"></script>
<script src="app/ui/views/View.js"></script>
<script src="app/ui/views/NegociacoesView.js"></script>
<script src="app/ui/models/Mensagem.js"></script>
<script src="app/ui/views/MensagemView.js"></script>
<script src="app/util/ProxyFactory.js"></script>
<script src "app/util/Bind js" </script>

<!-- importou aqui -->
<script src="app/ui/converters/DataInvalidaException.js"></script>

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

<!-- código posterior omitido -->
```

Agora, precisamos alterar a classe `DateConverter` para que seu método `paraData()` lance a exceção que acabamos de criar:

```
// client/app/ui/converters/DateConverte.js

class DateConverter {

    // código posterior omitido

    static paraData(texto) {

        // LANÇA A NOVA EXCEÇÃO!

        if (!/\d{2}\\d{2}\\d{4}/.test(texto))
            throw new DataInvalidaException();

        return new Date(...texto.split('/')
            .reverse())
```

```

        .map((item, indice) =>
            item - indice % 2));
    }
}

```

Vamos experimentar preencher os dados do formulário, mas com uma data **inválida**. Isso fará com que a mensagem de erro seja exibida para o usuário.

Agora, se cadastramos uma negociação com data **válida**, teremos o erro de chamada de um método que não existe sendo impresso apenas no console. Será?

11.5 CRIANDO NOSSA PRÓPRIA EXCEÇÃO: SEGUNDA TENTATIVA

Se olharmos as duas informações de erro que são impressas no console, a segunda que exibe `err.stack` indica que o erro foi de `Error`, e não `DataInvalidaException`. Precisamos realizar esse ajuste atribuindo à propriedade `this.name` herdada de `Error` o valor de `this.constructor.name`:

```

// client/app/ui/converters/DataInvalidaException.js

class DataInvalidaException extends Error {

    constructor() {

        super('A data deve estar no formato dd/mm/aaaa');

        // ajusta o nome do erro!
        this.name = this.constructor.name;
    }
}

```

Um novo teste demonstra que há uma paridade do resultado de `console.log(err)` e `console.log(err.stack)`.

Apesar de a nossa solução funcionar, não é muito intuitivo para o programador ter de lembrar sempre de adicionar a instrução `this.name = this.constructor.name` para ajustar o nome da exceção na stack. Hoje temos apenas um tipo de erro criado por nós, mas e se tivéssemos uma dezena deles?

Para evitar que o desenvolvedor precise lembrar de realizar o ajuste que vimos, vamos criar uma nova classe chamada `ApplicationException`. Todas as exceções que criarmos herdarão dessa classe em vez de `Error`.

Criando o arquivo em `client/app/util/ApplicationException.js`:

```
// client/app/util/ApplicationException.js

class ApplicationException extends Error {

  constructor(msg = '') {

    super(msg);
    this.name = this.constructor.name;
  }
}
```

Vamos importar o script em `client/index.html`. Mas atenção: como haverá uma dependência entre `DataInvalidaException` e `ApplicationException`, a última deve ser importada primeiro:

```
<script src="app/domain/negociacao/Negociacao.js"></script>
<script src="app/controllers/NegociacaoController.js"></script>
<script src="app/ui/converters/DateConverter.js"></script>
<script src="app/domain/negociacao/Negociacoes.js"></script>
<script src="app/ui/views/View.js"></script>
<script src="app/ui/views/NegociacoesView.js"></script>
<script src="app/ui/models/Mensagem.js"></script>
<script src="app/ui/views/MensagemView.js"></script>
```

```

<script src="app/util/ProxyFactory.js"></script>
<script src="app/util/Bind.js"></script>

<!-- importou aqui, precisa ser antes de DataInvalidaException -->

<script src="app/util/ApplicationException.js"></script>
<script src="app/ui/converters/DataInvalidaException.js"></script>

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

```

Agora, alterando `DataInvalidaException` para que passe a herdar de `ApplicationException`:

```

// client/app/ui/converters/DataInvalidaException.js

class DataInvalidaException extends ApplicationException {

  constructor() {

    super('A data deve estar no formato dd/mm/aaaa');
  }
}

```

Ao recarregarmos a página e verificarmos as mensagens de erros impressas no console para uma data inválida, vemos que o nome do erro está correto na impressão da `stack`.

Por fim, podemos voltar o nome correto da chamada do método `this._limpaFormulario()`:

```

// client/app/controllers/NegociacaoController.js

// código anterior omitido

adiciona(event) {

  try {

    event.preventDefault();
    this._negociacoes.adiciona(this._criaNegociacao());
    this._mensagem.texto = 'Negociação adicionada com sucesso

```

```
;

// VOLTOU COM O NOME CORRETO DO MÉTODO
this._limpaFormulario();

} catch(err) {
    // código omitido
}

// código posterior omitido
```

Agora que já temos informações mais amigáveis para o usuário, podemos continuar com nossa aplicação. Aliás, uma aplicação não é só feita de conversão de datas, ela também é feita de integrações, assunto do próximo capítulo.

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/11>

PILHANDO O QUE INTERESSA!

"No centro do sertão, o que é doideira às vezes pode ser a razão mais certa e de mais juízo!" - Grande Sertão: Veredas

Nós aplicamos diversos recursos da linguagem JavaScript durante a construção da nossa aplicação. Entretanto, para que ela fique completa, precisará importar negociações de uma API web. O projeto baixado já possui um servidor criado com Node.js, que disponibiliza a API necessária para nossa aplicação.

Vejamos a infraestrutura mínima necessária para que sejamos capazes de levantar o servidor.

12.1 SERVIDOR E INFRAESTRUTURA

Para que seja possível levantar o servidor disponibilizado, é preciso que o Node.js (<https://nodejs.org>) esteja instalado em sua máquina. Mas atenção, utilize sempre as versões pares, evitando as versões ímpares devido à instabilidade da última. A versão 8.1.3 do

Node.js foi utilizada durante a escrita deste livro.

Para levantar o servidor, é necessário um mínimo de traquejo com o terminal (ou prompt de comando, no Windows) do seu sistema operacional. Com o terminal aberto, vamos nos dirigir até a pasta `jscangaceiro/server` . Com a certeza de estarmos dentro da pasta correta, executaremos o comando:

```
npm start
```

Isto fará com que um servidor rode e seja acessível por meio do endereço `http://localhost:3000` . Acessando esse endereço, automaticamente a página `index.html` que estamos trabalhando ao longo do livro será carregada. Se preferir, pode digitar `http://localhost:3000/index.html` .

A partir de agora, só podemos acessar `index.html` através do nosso servidor; caso contrário, não poderemos utilizar vários recursos que aprenderemos ao longo do livro que dependem de requisições assíncronas realizadas do JavaScript para o servidor.

Além de servir nossa aplicação, o servidor disponibiliza três endereços especiais. São eles:

```
localhost:3000/negociacoes/semana  
localhost:3000/negociacoes/anterior  
localhost:3000/negociacoes/retrasada
```

Ao acessarmos o endereço `http://localhost:3000/negociacoes/semana` , sairemos da

página `index.html` e será exibida uma estrutura de dados no formato JSON (*JavaScript Object Notation*):

A screenshot of a web browser window. The address bar shows 'localhost:3000/negociacoes/semana'. The console displays a JSON array of three objects, each representing a trade record with date, quantity, and value.

```
[{"data": "2017-04-13T19:23:44.898Z", "quantidade": 1, "valor": 150}, {"data": "2017-04-13T19:23:44.898Z", "quantidade": 2, "valor": 250}, {"data": "2017-04-13T19:23:44.898Z", "quantidade": 3, "valor": 350}]
```

Figura 12.1: Dados no formato texto

No entanto, não queremos exibir o arquivo JSON na tela saindo da página `index.html`. Nosso objetivo é clicarmos no botão "Importar Negociações", buscar os dados usando JavaScript e inseri-los na tabela de negociações. Como implementamos nosso mecanismo de *data binding*, a tabela será renderizada automaticamente.

Não usaremos jQuery ou nenhum outro atalho que blinde o desenvolvedor dos detalhes de criação de requisições assíncronas. Como estamos trilhando o caminho de um cangaceiro JavaScript, precisaremos lidar diretamente com o objeto `XMLHttpRequest`, o centro nervoso do JavaScript para realização de requisições assíncronas.

12.2 REQUISIÇÕES AJAX COM O OBJETO XMLHTTPREQUEST

Precisamos fazer o botão "Importar Negociações" chamar uma ação em nosso controller. Vejamos o HTML do botão em `client/index.html`:

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

<div class="text-center">
  <button id="botao-importa" class="btn btn-primary text-center
```



```

    type="button">
        Importar Negociações
    </button>
    <button id="botao-apaga" class="btn btn-primary text-center"
type="button">
        Apagar
    </button>
</div>

<!-- código posterior omitido -->

```

Vamos criar o método `importaNegociacoes()` em `NegociacaoController` que, por enquanto, exibirá na tela um alerta:

```

// client/app/controllers/NegociacaoController.js

class NegociacaoController {

    // código posterior omitido

    importaNegociacoes() {

        alert('Importando negociações');

    }
}

```

Agora, alteraremos `client/app/app.js` e associaremos o evento `click` com a chamada do método que acabamos de criar:

```

// client/app/app.js

const controller = new NegociacaoController();

document
    .querySelector('.form')
    .addEventListener('submit', controller.adiciona.bind(controller));

document
    .querySelector('#botao-apaga')
    .addEventListener('click', controller.apaga.bind(controller))
;

```

```
// NOVIDADE AQUI!
// associando o evento à chamada do método

document
  .querySelector('#botao-importa')
  .addEventListener('click', controller.importaNegociacoes.bind
(controller));
```

Excelente, mas estamos repetindo várias vezes a instrução `document.querySelector`. Faremos a mesma coisa que já fizemos em `NegociacaoController`, vamos criar o alias `$` para a instrução, tornando nosso código menos verboso. Nosso `app.js` ficará assim:

```
// client/app/app.js

const controller = new NegociacaoController();

// criou o alias

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

$('.form')
  .addEventListener('submit', controller.adiciona.bind(controller));

$('#botao-apaga')
  .addEventListener('click', controller.apaga.bind(controller));
;

// associando o evento à chamada do método
$('#botao-importa')
  .addEventListener('click', controller.importaNegociacoes.bind
(controller));
```

Se recarregarmos a página no navegador e depois clicarmos em "Importar Negociações", o alert será exibido.

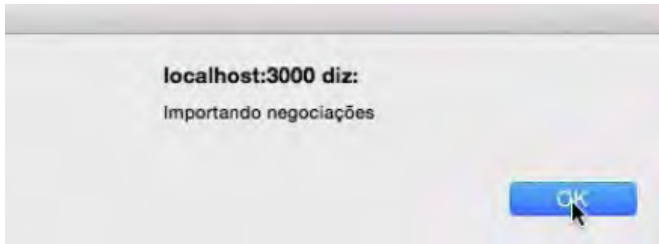


Figura 12.2: Alert no navegador

Com a certeza de que realizamos a associação do evento com o método do controller, substituiremos o `alert` por uma instância de `XMLHttpRequest`, um objeto do mundo JavaScript capaz de realizar requisições:

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

    // código posterior omitido

    importaNegociacoes() {

        const xhr = new XMLHttpRequest();
    }
}
```

Agora, vamos solicitar à instância que acabamos de criar que **abra** uma conexão com o servidor para nós:

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

    // código posterior omitido

    importaNegociacoes() {

        const xhr = new XMLHttpRequest();
        xhr.open('GET', 'negociacoes/semana');
```

```
}  
}
```

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

A requisição ainda não está pronta. Será preciso realizar algumas configurações antes do envio. Isso é feito através de uma função atribuída à propriedade `onreadystatechange` da instância de `xhr` :

```
// client/app/controllers/NegociacaoController.js  
  
class NegociacaoController {  
    // código posterior omitido  
  
    importaNegociacoes() {  
  
        const xhr = new XMLHttpRequest();  
        xhr.open('GET', 'negociacoes/semana');  
  
        xhr.onreadystatechange = () => {  
            // realizaremos nossas configurações aqui  
        };  
    }  
}
```

Toda requisição com `xhr` (ou AJAX, termo comum utilizado) passa por estados - um deles nos dará os dados retornados do servidor. Esses estados são:

- 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 aquele que nos interessa, porque nos dá acesso à resposta enviada pelo servidor. Precisamos descobrir em qual estado estamos quando o `onreadystatechange` for chamado. Para isto, adicionaremos um `if` :

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

    // código posterior omitido

    importaNegociacoes() {

        const xhr = new XMLHttpRequest();
        xhr.open('GET', 'negociacoes/semana');

        xhr.onreadystatechange = () => {

            if(xhr.readyState == 4) {

                }

            };

        }

    }
}
```

O estado 4 indica que a requisição foi concluída com a resposta pronta, contudo não podemos confiar nele. Podemos receber uma resposta de erro do servidor que continua sendo uma resposta

válida.

Além de testarmos o `readyState` da requisição, precisamos testar se `xhr.status` é igual a `200`, um código padrão que indica que a operação foi realizada com sucesso:

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

  // código posterior omitido

  importaNegociacoes() {

    const 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.');
```

Agora, toda vez que o `readyState` for `4` e o `status` for diferente de `200`, exibiremos no console uma mensagem indicando que não foi possível realizar a requisição. Inclusive, como última instrução do método, executaremos nossa requisição através de `xhr.send()`:

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {
```

```

// código posterior omitido

importaNegociacoes() {

    const xhr = new XMLHttpRequest();
    xhr.open('GET', 'negociacoes/semana');

    xhr.onreadystatechange = () => {

        if(xhr.readyState == 4) {

            if(xhr.status == 200) {

                console.log('Obtendo as negociações do servid
or.');
```

```
            } else {

                console.log('Não foi possível obter as negoci
ações da semana.');
```

```
            }

        }

    };

    xhr.send(); // executa a requisição configurada
}

```

A questão agora é: como teremos acesso aos dados que foram retornados pelo servidor?

12.3 REALIZANDO O PARSE DA RESPOSTA

Temos acesso aos dados retornados pela requisição através da propriedade `xhr.responseText`, inclusive é na mesma propriedade que recebemos mensagens de erros:

```

// client/app/controllers/NegociacaoController.js

class NegociacaoController {

    // código posterior omitido

```

```

importaNegociacoes() {

    const 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.');
                console.log(xhr.responseText);
            } else {
                console.log(xhr.responseText);
                this._mensagem.texto = 'Não foi possível obter as negociações da semana';
            }
        }
    };

    xhr.send();
}

```

Em seguida, recarregaremos a página e veremos o que é exibido no console:

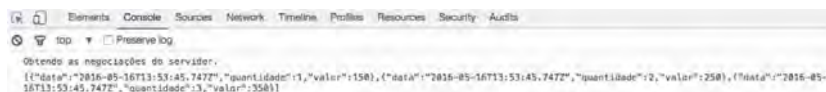


Figura 12.3: Texto exibido no console

Vemos que é um texto sendo exibido, e não um *array*. Como o protocolo HTTP só trafega texto, o JSON é uma representação textual de um objeto JavaScript. Com o auxílio de `JSON.parse()`, convertemos o JSON de string para objeto, permitindo assim manipulá-lo facilmente:


```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

  // código posterior omitido

  importaNegociacoes() {

    const 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');

          // realizando o parse
          console.log(JSON.parse(xhr.responseText));
        } else {
          console.log(xhr.responseText);
          this._mensagem.texto = 'Não foi possível obter as negociações da semana';
        }
      }
    };

    xhr.send();
  }
}
```

No console, vemos a saída de `JSON.parse()` , um array de objetos JavaScript:

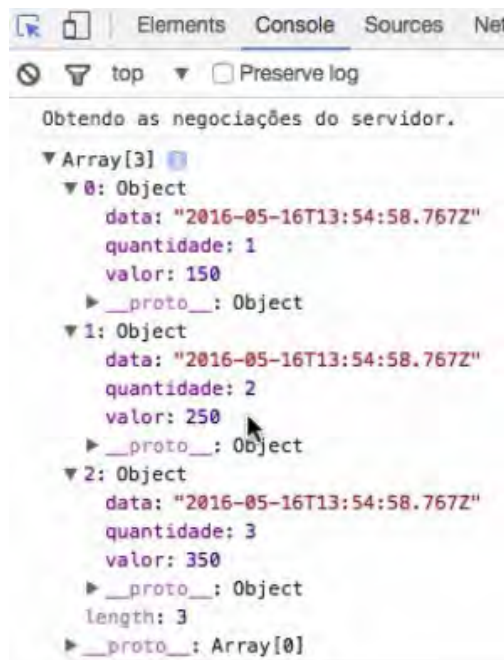


Figura 12.4: Array no console

Por mais que tenhamos convertido o resultado da requisição em um array de objetos, não podemos simplesmente adicionar cada item do array na lista de negociações através de `this._negociacoes.adiciona()`. O método espera receber uma instância de `Negociacao` e os dados que recebemos são apenas objetos com as propriedades `data`, `quantidade` e `valor`. Precisamos converter cada objeto para uma instância de `Negociacao` antes de adicioná-lo.

Realizaremos a conversão através da função `map()` do array de objetos que convertemos. Em seguida, com o array convertido, vamos iterar em cada instância de `Negociacao` pela função `forEach`, adicionando cada uma em `this._negociacoes`:

```

// client/app/controllers/NegociacaoController.js
// código posterior omitido

importaNegociacoes() {

    const xhr = new XMLHttpRequest();
    xhr.open('GET', 'negociacoes/semana');

    xhr.onreadystatechange = () => {

        if(xhr.readyState == 4) {

            if(xhr.status == 200) {

                // converte cada objeto para uma instância de Neg
ociacao
                JSON
                .parse(xhr.responseText)
                .map(objeto => new Negociacao(objeto.data, ob
jeto.quantidade, objeto.valor))
                .forEach(negociacao => this._negociacoes.adic
iona(negociacao));

                this._message.texto = 'Negociações importadas com
sucesso!';

            } else {
                console.log(xhr.responseText);
                this._mensagem.texto = 'Não foi possível obter a
s negociações da semana';
            }
        }
    };

    xhr.send();
}

// código posterior omitido

```

Nosso código está quase completo, porque se tentarmos adicionar uma nova negociação, receberemos uma mensagem de erro.

Agora, se acessarmos o endereço `localhost:3000/negociacoes/semana`, veremos que a data está em um formato de string um pouco diferente.



Figura 12.5: Data no formato estranho

Nós estamos passando `objeto.data` na representação textual diretamente no `constructor()` de `Negociacao`. Em vez disso, passaremos uma nova instância de `Date` que receberá em seu `constructor()` a data no formato string, pois é uma estrutura válida:

```
// client/app/controllers/NegociacaoController.js
// código anterior omitido

// veja que agora temos new Date(object.data)

JSON
  .parse(xhr.responseText)
  .map(objeto => new Negociacao(new Date(objeto.data), objeto.q
    uantidade, objeto.valor))
  .forEach(negociacao => this._negociacoes.adiciona(negociacao)
);

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

// código posterior omitido
```

A imagem mostra uma caixa de diálogo ou uma barra de mensagem com o texto 'Negociações importadas com sucesso.' em uma fonte sans-serif.

Figura 12.6: Negociações importadas com sucesso

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

```
// client/app/controllers/NegociacaoController.js

// código anterior omitido
importaNegociacoes() {

    const xhr = new XMLHttpRequest();

    // FORÇANDO UM ERRO, UM ENDEREÇO QUE NÃO EXISTE
    xhr.open('GET', 'negociacoes/xsemana');

    // código posterior omitido
```

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

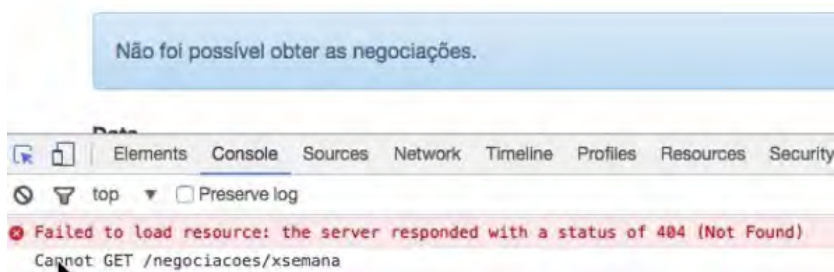


Figura 12.7: Mensagem de erro

Só não esqueçam de voltar com o endereço correto depois do teste!.

Conseguimos realizar requisições assíncronas (Ajax) usando JavaScript padrão, sem apelarmos para bibliotecas como jQuery.

No entanto, nosso código pode ficar ainda melhor.

12.4 SEPARANDO RESPONSABILIDADES

Já somos capazes de importamos negociações, mas o que aconteceria se tivéssemos outros controllers e estes precisassem importar negociações? Teríamos de repetir toda a lógica de interação com o servidor. Além disso, o acesso de APIs não é responsabilidade de `NegociacaoController`, apenas a captura de ações do usuário e realizar a ponte entre o `model` e a `view`.

Vamos isolar a responsabilidade de interação com o servidor em uma classe que a centralizará. Criaremos o arquivo `client/app/domain/negociacao/NegociacaoService.js`:

```
// client/app/domain/negociacao/NegociacaoService.js

class NegociacaoService {

  obterNegociacoesDaSemana() {

  }

}
```

Vamos importar o script que acabamos de criar em `client/index.html`. Entretanto, como ele será uma dependência no `constructor()` de `NegociacaoController`, deve ser importado antes de `app.js`, pois é nesse arquivo que criamos uma instância de `NegociacaoController`.

A classe terá um único método chamado `obterNegociacaoDaSemana()`. Dentro desse método, moveremos o código que escrevermos no método `importaNegociacoes()` de `NegociacaoController`:

```

<!-- client/index.html -->
<!-- código anterior omitido -->

<script src="app/domain/negociacao/Negociacao.js"></script>
<script src="app/controllers/NegociacaoController.js"></script>
<script src="app/ui/converters/DateConverter.js"></script>
<script src="app/domain/negociacao/Negociacoes.js"></script>
<script src="app/ui/views/View.js"></script>
<script src="app/ui/views/NegociacoesView.js"></script>
<script src="app/ui/models/Mensagem.js"></script>
<script src="app/ui/views/MensagemView.js"></script>
<script src="app/util/ProxyFactory.js"></script>
<script src="app/util/Bind.js"></script>
<script src="app/util/ApplicationException.js"></script>
<script src="app/ui/converters/DataInvalidaException.js"></script>

<!-- importou aqui! -->
<script src="app/domain/negociacao/NegociacaoService.js"></script>

<script src="app/app.js"></script>
<!-- código posterior omitido -->

```

Agora, vamos mover todo o código que já escrevemos do método `importarNegociacoes` de `NegociacaoController` para dentro do método da nossa nova classe:

```

// client/app/domain/negociacao/NegociacaoService.js

class NegociacaoService {

  obterNegociacoesDaSemana() {

    // CÓDIGO MOVIDO!

    const 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._negociacoes.adiciona(negociacao));

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

        } else {
            console.log(xhr.responseText);
            this._mensagem.texto = 'Não foi possível obter as negociações da semana!';
        }
    }
};

xhr.send();
}
}

```

Como movemos o código, por enquanto o método `importaNegociacoes()` ficará vazio em `NegociacaoController`.

Agora, no `constructor()` de `NegociacaoController`, vamos guardar na propriedade `this._service` uma instância de `NegociacaoService`, para que possamos acessar dentro de qualquer método da classe:

```

// client/app/controllers/NegociacaoController.js

class NegociacaoController {

    constructor() {

        // código anterior omitido

        this._service = new NegociacaoService();
    }
}

```



```
}
```

Em seguida, ainda em nosso controller, no método `importaNegociacoes()` que se encontra vazio, chamaremos o método `obterNegociacoesDaSemana()` do serviço:

```
// client/app/controllers/NegociacaoController.js
```

```
// código anterior omitido
```

```
importaNegociacoes() {  
    this._service.obterNegociacoesDaSemana();  
}
```

```
// código posterior omitido
```

Do jeito que nosso código se encontra, não conseguiremos importar as negociações, porque o método `obterNegociacoesDaSemana()` faz referência a elementos de `NegociacaoController`, e ambos devem ser independentes.

O primeiro passo para resolvermos esse problema é removermos do método `obterNegociacoesDaSemana()` todas as referências às propriedades de `NegociacaoController`. Ele ficará assim:

```
// client/app/domain/negociacao/NegociacaoService.js
```

```
class NegociacaoService {  
    obterNegociacoesDaSemana() {  
        // CÓDIGO MOVIDO!  
        const xhr = new XMLHttpRequest();  
        xhr.open('GET', 'negociacoes/semana');  
        xhr.onreadystatechange = () => {
```

```

        if(xhr.readyState == 4) {

            if(xhr.status == 200) {

                // NÃO FAZ MAIS O FOREACH
                JSON
                .parse(xhr.responseText)
                .map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor));

                // NÃO EXIBE MENSAGEM

            } else {
                console.log(xhr.responseText);

                // NÃO EXIBE MENSAGEM
            }
        }
    };

    xhr.send();
}

```

Do jeito que deixamos nossa aplicação, não recebemos mais mensagens de erro ao importamos negociações, mas também elas não são inseridas na tabela. Inclusive, não temos qualquer mensagem para o usuário de sucesso ou fracasso. De um lado, temos o método `importaNegociacoes()` de `NegociacaoController` que depende da lista de negociações retornadas do servidor e, do outro, temos a classe `NegociacaoService` que sabe buscar os dados, mas não sabe o que fazer com eles.

Para solucionarmos esse impasse, quem chama o método `this._service.obterNegociacoesDaSemana()` precisará passar uma função com a lógica que será executada assim que o método trazer os dados do servidor. Para entendermos melhor essa

estratégia, primeiro vamos passar a função sem antes alterar o método `obterNegociacoesDaSemana()` :

```
// client/app/controller/NegociacaoController.js
// código anterior omitido

importaNegociacoes() {

    this._service.obterNegociacoesDaSemana((err, negociacoes) =>
    {

        if(err) {
            this._mensagem.texto = 'Não foi possível obter nas ne
gociações da semana';
            return;
        }

        negociacoes.forEach(negociacao =>
            this._negociacoes.adiciona(negociacao));

        this._mensagem.texto = 'Negociações importadas com sucess
o';

    });
}

// código posterior omitido
```

A função que passamos é um **callback** (uma função que será chamada posteriormente), e ela segue o padrão **Error-First-Callback**. Essa abordagem é muito comum quando queremos lidar com código assíncrono. Caso a função que recebeu o callback não consiga executar sua operação, no primeiro parâmetro da callback recebemos um erro e, no segundo, `null` .

Caso a função que recebeu o callback consiga executar sua operação, no primeiro parâmetro do callback receberemos `null` e, no segundo, os dados resultantes da operação (em nosso caso, a lista de negociações). A partir da ausência ou não de valor em

err , lidamos com o sucesso ou fracasso da operação.

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

```
// client/app/domain/negociacao/NegociacaoService.js

class NegociacaoService {

  obterNegociacoesDaSemana(cb) {

    const xhr = new XMLHttpRequest();
    xhr.open('GET', 'negociacoes/semana');

    xhr.onreadystatechange = () => {

      if(xhr.readyState == 4) {

        if(xhr.status == 200) {

          const negociacoes = JSON
            .parse(xhr.responseText)
            .map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor));

          // OPERAÇÃO CONCLUÍDA, SEM ERRO
          cb(null, negociacoes);

        } else {

          console.log(xhr.responseText);

          // ERRO NA OPERAÇÃO!
          cb('Não foi possível obter nas negociações d
a semana', null);
        }
      }
    };

    xhr.send();
  }
}
```

Se ocorrer um erro, executaremos o `cb` exibindo uma

mensagem de alto nível, e informando para o usuário que não foi possível obter as negociações. Ao recarregarmos a página e preenchermos os dados do formulário, veremos a mensagem de sucesso.



Figura 12.8: Negociações foram importadas com sucesso

Nosso código funciona como esperado. Mas será que ele se sustenta ao longo do tempo? Há outra maneira de deixarmos nosso código ainda melhor? Veremos no próximo capítulo.

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/12>

LUTANDO ATÉ O FIM

"Cavalo que ama o dono até respira do mesmo jeito" - Grande Sertão: Veredas

O padrão ***Error-First-Callback*** é bastante comum, principalmente no mundo Node.js para lidar com código assíncrono. Contudo, sua implementação polui a interface de uso de métodos e funções.

Hoje, se quisermos obter uma lista de negociações, chamamos o método `obtemNegociacoesDaSemana()` de `NegociacaoService`. Porém, o método só recebe o callback como parâmetro para termos acesso ao resultado ou ao erro de sua chamada. O parâmetro de `obtemNegociacoesDaSemana()` não faz parte do domínio de negociações, é um parâmetro mais de infraestrutura do nosso código.

A mesma coisa acontece com o método `get()` de `HttpService`. Ele recebe a `url` do servidor em que buscará as informações. Já o segundo parâmetro é um callback para capturarmos o retorno de sucesso ou erro da operação.

Além do que vimos, precisamos sempre testar pela ausência ou não do valor de `err` para sabermos se a operação foi bem-sucedida. Para piorar, podemos cair no **Callback HELL**.

13.1 CALLBACK HELL

Outro ponto dessa abordagem é que podemos cair no **Callback HELL**, uma estrutura peculiar em nosso código resultante de operações assíncronas que lembra uma **pirâmide deitada**. Vejamos um exemplo no qual realizamos quatro requisições `get()` com `HttpService`, para exibirmos no final uma lista consolidada com os dados de todas as requisições:

```
// EXEMPLO, NÃO ENTRA EM NOSSO CÓDIGO

const service = new HttpService();

let resultado = [];

service.get('http://www.endereco1.com', (err, dados1) => {
  resultado = resultado.concat(dados1);
  service.get('http://www.endereco2.com', (err, dados2) => {
    resultado = resultado.concat(dados2);
    service.get('http://www.endereco3.com', (err, dados3) => {
      resultado = resultado.concat(dados3);
      service.get('http://www.endereco4.com', (err, dados4) => {
        resultado = resultado.concat(dados4);
        console.log('Lista completa');
        console.log(resultado);
      });
    });
  });
});
```

Não precisamos meditar muito para percebermos que essa estrutura não é lá uma das melhores para darmos manutenção. Será que existe outro padrão que podemos utilizar para

escrevermos um código mais legível e fácil de manter?

13.2 O PADRÃO DE PROJETO PROMISE

Há o padrão de projeto Promise criado para lidar com a complexidade de operações assíncronas. Aliás, a partir da versão ES2015 (ES6), este padrão tornou-se nativo na linguagem JavaScript.

Uma Promise é o resultado futuro de uma ação; e quando dizemos futuro, é porque pode ser fruto de uma operação assíncrona. Por exemplo, quando acessamos um servidor, a requisição pode demorar milésimos ou segundos, tudo dependerá de vários fatores ao longo da requisição.

Antes de refatorarmos (alterarmos sem mudar comportamento) por completo o nosso código, vamos alterar apenas `NegociacaoController`, mais propriamente seu método `importaNegociacoes()`, para que considere uma Promise como retorno do método `this._service.obtemNegociacoesDaSemana()`.

Alterando nosso código:

```
// client/app/controllers/NegociacaoController.js

// código anterior omitido
importaNegociacoes() {

    // apagou o código anterior, começando do zero
    const promise = this._service.obtemNegociacoesDaSemana();
}
```

Duas coisas notáveis se destacam nessa linha de código. A primeira é a ausência da passagem do callback para o método

`this._service.obtemNegociacoesDaSemana()` . O segundo, o resultado da operação é uma Promise. É através da função `then()` que interagimos com a Promise.

A função `then()` recebe duas funções (callbacks) como parâmetros. A primeira função nos dá acesso ao retorno da operação assíncrona, já o segundo a possíveis erros que possam ocorrer durante a operação. Alterando nosso código:

```
// client/app/controllers/NegociacaoController.js
// código anterior omitido

importaNegociacoes() {

    const promise = this._service.obtemNegociacoesDaSemana();

    promise.then(
        negociacoes => {
            negociacoes.forEach(negociacao => this._negociacoes.a
            diciona(negociacao));
            this._mensagem.texto = 'Negociações importadas com su
            cesso';
        },
        err => this._mensagem.texto = err
    );
}

// código posterior omitido
```

Podemos simplificar o código anterior eliminando a variável `promise` :

```
// client/app/controllers/NegociacaoController.js
// código anterior omitido

importaNegociacoes() {

    this._service.obtemNegociacoesDaSemana()
        .then(
```

```

        negociacoes => {
            negociacoes.forEach(negociacao => this._negociaco
es.adiciona(negociacao));
            this._mensagem.texto = 'Negociações importadas co
m sucesso';
        },
        err => this._mensagem.texto = err
    );
}

// código posterior omitido

```

Vamos recapitular! Métodos que retornam uma Promise não precisam receber um callback, evitando poluir sua interface de uso. A função `then()` evita a condição `if` para tratar o sucesso ou fracasso, pois trata ambos separadamente nas funções de callback que recebe. A primeira função é para obtermos o resultado da operação e, a segunda, erros que possam acontecer e que impedem que a operação seja concluída.

Excelente. Porém, nosso código ainda não funciona, pois o método `obtemNegociacoesDaSemana()` de `NegociacaoService` ainda não retorna uma Promise. Chegou a hora de alterá-lo.

13.3 CRIANDO PROMISES

O primeiro passo para convertermos o método `obtemNegociacoesDaSemana()` de `NegociacaoService` é adicionar todo seu código existente dentro do bloco de uma Promise. Primeiro, vejamos a estrutura de uma Promise quando a estamos criando:

```

// EXEMPLO, NÃO ENTRA NA APLICAÇÃO AINDA

new Promise((resolve, reject) => {

```

```
// lógica da promise!  
});
```

O `constructor()` de uma `Promise` recebe uma função preparada para receber dois parâmetros. O primeiro é uma referência para uma função que deve receber o resultado da operação assíncrona que encapsula. Já o segundo, possíveis erros que possam acontecer.

Vamos alterar o método `obtemNegociacoesDaSemana()` envolvendo todo o código já existente dentro do bloco de uma nova `Promise`:

```
// client/domain/negociacao/NegociacaoService.js  
// código anterior omitido  
  
obtemNegociacoesDaSemana(cb) {  
    return new Promise((resolve, reject) => {  
        /* início do bloco da Promise */  
  
        const xhr = new XMLHttpRequest();  
        xhr.open('GET', 'negociacoes/semana');  
  
        xhr.onreadystatechange = () => {  
            if(xhr.readyState == 4) {  
                if(xhr.status == 200) {  
                    const negociacoes = JSON  
                        .parse(xhr.responseText)  
                        .map(objeto => new Negociacao(new Date(objeto.data), objeto.quantidade, objeto.valor));  
                    cb(null, negociacoes);  
                } else {
```

```

        cb('Não foi possível obter nas negociações',
null);
    }
  }
};

xhr.send();

/* fim do bloco da Promise */

});
}
// código posterior omitido

```

A função passada para o constructor() de Promise possui dois parâmetros, resolve e reject . O primeiro é uma referência para uma função na qual passamos o valor da operação. No segundo, passamos a causa do fracasso.

Agora, nos locais que usamos cb , utilizaremos resolve() e reject() . O método ficará assim:

```

// client/domain/negociacao/NegociacaoService.js
// código anterior omitido

// NÃO RECEBE MAIS O CALLBACK
obtemNegociacoesDaSemana() {

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

    const xhr = new XMLHttpRequest();
    xhr.open('GET', 'negociacoes/semana');

    xhr.onreadystatechange = () => {

      if(xhr.readyState == 4) {

        if(xhr.status == 200) {

          const negociacoes = JSON
            .parse(xhr.responseText)
            .map(objeto => new Negociacao(new Date(ob

```

```

jeto.data), objeto.quantidade, objeto.valor));

        // CHAMOU RESOLVE
        resolve(negociacoes);
    } else {

        // CHAMOU REJECT
        reject('Não foi possível obter nas negociações');
    }
}

};

xhr.send();

});
}
// código posterior omitido

```

Perfeito! A importação das negociações continua funcionando com o novo padrão de organização de código com Promise. No entanto, podemos deixar nosso código ainda melhor.

13.4 CRIANDO UM SERVIÇO PARA ISOLAR A COMPLEXIDADE DO XMLHTTPREQUEST

O método `obtemNegociacoesDaSemana()` de `NegociacaoService`, apesar de funcional, mistura responsabilidades. Nele, temos o código responsável em lidar com o objeto `XMLHttpRequest` e o que trata a resposta específica da lista de negociações.

O problema da abordagem anterior é a duplicação do código de configuração do `XMLHttpRequest` em todos os métodos que precisarem realizar este tipo de requisição. Para solucionarmos esse problema, vamos isolar em uma classe de serviço a complexidade de se lidar com `XMLHttpRequest`, que também

utilizará o padrão Promise:

Vamos criar o arquivo `client/app/util/HttpService.js`:

```
// client/app/util/HttpService.js

class HttpService {

  get(url) {

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

      const xhr = new XMLHttpRequest();

      xhr.open('GET', url);

      xhr.onreadystatechange = () => {

        if(xhr.readyState == 4) {

          if(xhr.status == 200) {

            // PASSOU O RESULTADO PARA RESOLVE
            // JÁ PARSEADO!

            resolve(JSON.parse(xhr.responseText));
          } else {

            console.log(xhr.responseText);

            // PASSOU O ERRO PARA REJECT
            reject(xhr.responseText);
          }
        }
      };

      xhr.send();

    });
  }
}
```

Agora, não podemos nos esquecer de importar o script em

client/index.html . Atenção para a ordem de importação. Como o script será uma dependência no NegociacaoService , ele deve ser importado antes:

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

<script src="app/domain/negociacao/Negociacao.js"></script>
<script src="app/controllers/NegociacaoController.js"></script>
<script src="app/ui/converters/DateConverter.js"></script>
<script src="app/domain/negociacao/Negociacoes.js"></script>
<script src="app/ui/views/View.js"></script>
<script src="app/ui/views/NegociacoesView.js"></script>
<script src="app/ui/models/Mensagem.js"></script>
<script src="app/ui/views/MensagemView.js"></script>
<script src="app/util/ProxyFactory.js"></script>
<script src="app/util/Bind.js"></script>
<script src="app/util/ApplicationException.js"></script>
<script src="app/ui/converters/DataInvalidaException.js"></script>

<script src="app/domain/negociacao/NegociacaoService.js"></script>

<!-- importou aqui! -->
<script src="app/util/HttpService.js"></script>

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

<!-- código posterior omitido -->
```

Vejamos um exemplo de nosso serviço funcionando:

```
// EXEMPLO APENAS, AINDA NÃO ENTRA NA APLICAÇÃO

const http = new HttpService();
http
  .get('negociacoes/semana')
  .then(
    dados => console.log(dados),
    err => console.log(err)
  );
```

Agora, precisamos integrar `HttpService` com `NegociacaoService`. Durante essa integração, novos detalhes sobre Promises saltarão aos olhos.

O primeiro passo é adicionarmos uma instância de `HttpService` como dependência de `NegociacaoService`, através da propriedade `this._http`:

```
// client/app/domain/service/NegociacaoService.js

class NegociacaoService {

  constructor() {

    // NOVA PROPRIEDADE!
    this._http = new HttpService();
  }

  obterNegociacoesDaSemana() {

    // código anterior omitido
  }
}
```

Vamos alterar o método `obterNegociacoesDaSemana()`, para que faça uso do nosso novo serviço:

```
// client/app/domain/service/NegociacaoService.js

class NegociacaoService {

  constructor() {

    this._http = new HttpService();
  }

  obterNegociacoesDaSemana() {

    // ATENÇÃO, RETURN AQUI!

    return this._http
  }
}
```



```

        .get('negociacoes/semana')
        .then(
            dados => {

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

                // ATENÇÃO AQUI!
                return negociacoes;
            },
            err => {
                // ATENÇÃO AQUI!
                throw new Error('Não foi possível obter as negociações');
            }
        );
    }
}

```

Não criamos uma nova Promise, e sim retornamos uma já existente, no caso, a Promise criada pelo método `get()` de `HttpService`. A Promise retornada já sabe lidar com o sucesso ou fracasso da requisição. Além desse detalhe, há mais dois pontos notáveis do nosso código.

O primeiro é o `return negociacoes` do callback de sucesso, passado para `then()`. Por regra, toda Promise que possui um `return` disponibilizará o valor retornado na próxima chamada encadeada à função `then()`. Sendo assim, quem chamar o método `obtemNegociacoesDaSemana()` pode ter acesso ao `return negociacoes`, dessa maneira:

```

// EXEMPLO, NÃO ENTRA NA APLICAÇÃO AINDA

const service = new NegociacaoService();
service
    .obtemNegociacoesDaSemana()
    // AQUI TEMOS ACESSO AO RETURN DA PROMISE
    .then(negociacoes => console.log(negociacoes));

```

Por fim, ainda no método `obtemNegociacoesDaSemana()` , quando a Promise criada pelo método `get()` de `HttpService` falha, lançamos uma exceção através de `throw new Error('Não foi possível obter as negociações')` . Isso é necessário quando queremos devolver uma nova resposta de erro:

```
// EXEMPLO, NÃO ENTRA NA APLICAÇÃO AINDA

const service = new NegociacaoService();
service
  .obtemNegociacoesDaSemana()
  .then(
    negociacoes => console.log(negociacoes),
    // AQUI ACESSAMOS O ERRO DO THROWS
    err => console.log(err);
  );
```

Nosso código continua com o mesmo comportamento, só alteramos sua organização. Continuamos importando as negociações da semana clicando no botão "Importar Negociações" do formulário.

13.5 RESOLVENDO PROMISES SEQUENCIALMENTE

Nossa aplicação, além de importar as negociações da semana, deve importar as negociações da semana anterior. Inclusive, há uma API no servidor disponibilizado que retorna essa informação. No fim, queremos exibir todas as negociações como um todo na mesma tabela.

Em `NegociacaoService` , criaremos o método `obtemNegociacoesDaSemanaAnterior()` que terá um código idêntico ao método que já temos. A única coisa que muda é o

endereço da API e a mensagem de erro:

```
// client/app/domain/negociacao/NegociacaoService.js
// código posterior omitido

obtemNegociacoesDaSemanaAnterior() {

    return this._http
        .get('negociacoes/anterior')
        .then(
            dados => {

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

                return negociacoes;
            },
            err => {
                // ATENÇÃO AQUI!
                throw new Error('Não foi possível obter as negoci
ações da semana anterior');
            }
        );
}

// código anterior omitido
```

Agora, vamos alterar o método `importaNegociacoes()` de `NegociacaoController` para importar as negociações da semana anterior. Vejamos o código primeiro, para em seguida comentarmos a seu respeito:

```
// client/app/controllers/NegociacaoController.js

// código anterior omitido

importaNegociacoes() {

    const negociacoes = [];

    this._service
        .obtemNegociacoesDaSemana()
```

```

.then(semana => {

    // USANDO SPREAD OPERATOR
    negociacoes.push(...semana);

    // QUANDO RETORNAMOS UMA PROMISE, SEU RETORNO
    // É ACESSÍVEL AO ENCADEAR UMA CHAMADA À THEN

    return this._service.obtemNegociacoesDaSemanaAnterior
    ());
})
.then(anterior => {

    negociacoes.push(...anterior);
    negociacoes.forEach(negociacao => this._negociacoes.a
diciona(negociacao));
    this._mensagem.texto = 'Negociações importadas com su
cesso';
})
.catch(err => this._mensagem.texto = err);
}

```

Criamos o array `negociacoes` vazio para que possa receber as negociações da semana e da semana anterior. Começamos chamando o método `this._service.obtemNegociacoesDaSemana()` que retorna uma Promise. Através do método `then()`, temos acesso ao retorno da operação - nenhuma novidade até agora.

Ainda em `then()`, realizamos um `negociacoes.push(...anterior)` usando o *spread operator*, que desmembrará cada elemento do array `anterior`, passando-os como parâmetro para a função `push()`. Ela está preparada para lidar com um ou mais parâmetros.

Depois de incluirmos as negociações da semana no array `negociacoes`, executamos a enigmática instrução `return this._service.obtemNegociacoesDaSemanaAnterior()`. O

`return` não está lá por acaso, toda função `then()` que retorna uma Promise torna o resultado dessa Promise acessível na próxima chamada à `then()`. É por isso que temos:

```
// REVISANDO APENAS!
// código anterior omitido

.then(anterior => {

    negociacoes.push(...anterior);
    negociacoes.forEach(negociacao => this._negociacoes.adiciona(
negociacao));
    this._mensagem.texto = 'Negociações importadas com sucesso';
})
// código posterior omitido
```

Nessa chamada, `anterior` são as negociações das semanas anteriores, resultado da Promise que foi retornada. Veja que essa estratégia evita o *Callback HELL*, pois não temos chamadas de funções aninhadas.

Por fim, temos uma chamada inédita à função `catch()`, presente em toda Promise. Ela permite centralizar o tratamento de erros das Promises em um único lugar. Isso evita que tenhamos de passar o callback para tratar o erro em cada Promise.

Recarregando nossa página e clicando no botão "Importar Negociações", tudo continua funcionando, excelente. Mas ainda falta mais um método em `NegociacaoService`.

Precisamos importar também as negociações da semana **retrasada**. Para isso, vamos criar o método `obtemNegociacoesDaSemanaRetrasada()` em `NegociacaoService`. Assim como o método que criamos anteriormente, ele será idêntico ao `obtemNegociacoesDaSemana()`, apenas o endereço da API e sua

mensagem de erro serão diferentes:

```
// client/app/domain/negociacao/NegociacaoService.js
// código anterior omitido

obtemNegociacoesDaSemanaRetrasada() {

    return this._http
        .get('negociacoes/retrasada')
        .then(
            dados => {

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

                return negociacoes;
            },
            err => {
                throw new Error('Não foi possível obter as negoci
ações da semana retrasada');
            }
        );
}
```

Agora, repetiremos o que já fizemos anteriormente e incorporaremos a chamada do nosso método em `importaNegociacoes()` de `NegociacaoController`:

```
// client/app/domain/negociacao/NegociacaoService.js
// código anterior omitido

importaNegociacoes() {

    const negociacoes = [];

    this._service
        .obtemNegociacoesDaSemana()
        .then(semama => {
            negociacoes.push(...semama);

            return this._service.obtemNegociacoesDaSemanaAnterior
        });
}
```

```

    })
    .then(anterior => {
        negociacoes.push(...anterior);
        return this._service.obtemNegociacoesDaSemanaRetrasada(
a()
    })
    .then(retrasada => {
        negociacoes.push(...retrasada);
        negociacoes.forEach(negociacao => this._negociacoes.adiciona(negociacao));
        this._mensagem.texto = 'Negociações importadas com sucesso';
    })
    .catch(err => this._mensagem.texto = err);
}

// código posterior omitido

```

Nosso método resolve nossas Promises sequencialmente, isto é, ele só tentará resolver a segunda caso a primeira tenha sido resolvida, e só resolverá a terceira caso a segunda tenha sido resolvida. Todavia, em alguns cenários, pode ser interessante resolvermos todas as Promises em paralelo, principalmente quando a ordem de resolução não é importante. É isso que veremos a seguir.

13.6 RESOLVENDO PROMISES PARALELAMENTE

Podemos resolver nossas Promises paralelamente com auxílio de `Promise.all()`, que recebe um array de Promises como parâmetro. Vamos recomençar do zero o método `importaNegociacoes()` de `NegociacaoController`.

```

// client/app/controllers/NegociacaoController.js
// código anterior omitido

```

```

importaNegociacoes() {

    // RECEBE UM ARRAY DE PROMISES

    Promise.all([
        this._service.obtemNegociacoesDaSemana(),
        this._service.obtemNegociacoesDaSemanaAnterior(),
        this._service.obtemNegociacoesDaSemanaRetrasada()
    ])
    .then(periodo => console.log(periodo))
    .catch(err => this._mensagem.texto = err);
}

```

A função `Promise.all()` resolverá todas as Promises em paralelo, o que pode reduzir drasticamente o tempo gasto com essas operações. Quando todas tiverem sido resolvidas, a chamada à função `then()` nos devolverá um array de arrays, isto é, cada posição do array terá a lista de negociações retornada por cada Promise, na mesma ordem em que foram passadas para `Promise.all()`. Podemos verificar essa informação olhando a saída de `console.log(periodo)`.

Não podemos simplesmente iterar nesse array e adicionar cada negociação em `this._negociacoes`, pois cada item do array, como vimos, é um array. Precisamos tornar `periodo` em um array de única dimensão, ou seja, queremos achatar (*flatten*) este array.

Podemos conseguir este resultado facilmente com auxílio da função `reduce()`, que já utilizamos nos capítulos anteriores:

```

// client/app/controllers/NegociacaoController.js
// código anterior omitido

importaNegociacoes() {

    Promise.all([
        this._service.obtemNegociacoesDaSemana(),

```



```

        this._service.obtemNegociacoesDaSemanaAnterior(),
        this._service.obtemNegociacoesDaSemanaRetrasada()
    ])
    .then(periodeo => {

        // periodo ainda é um array com 3 elementos que são array

        periodo = periodo.reduce((novoArray, item) => novoArray.concat(item), []);

        // um array com nome elementos
        console.log(periodo);
    })
    .catch(erro => this._mensagem.texto = erro);
}

```

Com esta alteração, `periodo` passa a ser um array com nove elementos em vez de um com três elementos do tipo array. Agora, podemos adicionar cada item do novo array em nossa lista de negociações:

```

// client/app/controllers/NegociacaoController.js
// código anterior omitido

importaNegociacoes() {

    Promise.all([
        this._service.obtemNegociacoesDaSemana(),
        this._service.obtemNegociacoesDaSemanaAnterior(),
        this._service.obtemNegociacoesDaSemanaRetrasada()
    ])
    .then(periodeo => {

        periodo
            .reduce((novoArray, item) => novoArray.concat(item), [])
            .forEach(negociacao => this._negociacoes.adiciona(negociacao));

        this._mensagem.texto = 'Negociações importadas com sucesso';
    })
}

```

```

    .catch(err => this._mensagem.texto = err);
}

```

// código posterior omitido

Mais um código melhorado, mas ele ainda fere um princípio. A lógica de buscar as negociações do período não deveria estar em `NegociacaoController`, mas em `NegociacaoService`. Vamos criar nesta classe o método `obtemNegociacoesDoPeriodo()`:

// client/app/controllers/NegociacaoController.js
// código anterior omitido

```

obtemNegociacoesDoPeriodo() {

    // ACESSA AOS PRÓPRIOS MÉTODOS ATRAVÉS DE THIS

    return Promise.all([
        this.obtemNegociacoesDaSemana(),
        this.obtemNegociacoesDaSemanaAnterior(),
        this.obtemNegociacoesDaSemanaRetrasada()
    ])
    .then(periodo => {

        // NÃO FAZ MAIS O FOREACH
        return periodo
            .reduce((novoArray, item) => novoArray.concat(item),
                []);
    })
    .catch(err => {

        console.log(err);
        throw new Error('Não foi possível obter as negociações do período');
    });
}

```

Por fim, em `NegociacaoController`, vamos chamar o método que acabamos de criar:

// client/app/controllers/NegociacaoController.js
// código anterior omitido

```

importaNegociacoes() {

    this._service
        .obtemNegociacoesDoPeriodo()
        .then(negociacoes => {

            negociacoes.forEach(negociacao => this._negociacoes.adici
            ona(negociacao));
            this._mensagem.texto = 'Negociações do período importadas
            com sucesso';
        })
        .catch(err => this._mensagem.texto = err);
    }

    // código posterior omitido

```

Excelente. Vimos que `Promise.all()` recebe um array com todas as Promises que desejamos resolver paralelamente. A posição das Promises no array é importante, pois ao serem resolvidas, teremos acesso a um array com o resultado de cada uma delas, através de `then`, seguindo a mesma ordem de resolução do array.

As APIs que acessamos são "boazinhas", cada uma retorna as datas em ordem decrescente. Contudo, não podemos confiar em uma API que não foi criada por nós, razão pela qual precisamos garantir em nosso código a ordenação que desejamos.

13.7 ORDENANDO O PERÍODO

O método `obtemNegociacoesDoPeriodo` de `NegociacaoService` devolve uma Promise que, ao ser resolvida, devolve um array com um período de negociações. Podemos ordenar esse array da maneira que desejarmos através da função `sort()`, presente em todo array.

A função `sort()` recebe uma estratégia de comparação por meio de uma função. Esta função é chamada diversas vezes, comparando cada elemento com o próximo. A lógica de comparação deve retornar 0 se os elementos são iguais; 1 ou superior, se o primeiro elemento é maior que o segundo; e -1 ou inferior se o primeiro elemento é menor do que o segundo.

Alterando o método:

```
// client/app/domain/negociacao/NegociacaoService.js
// código anterior omitido

obtemNegociacoesDoPeriodo() {

    return Promise.all([
        this.obtemNegociacoesDaSemana(),
        this.obtemNegociacoesDaSemanaAnterior(),
        this.obtemNegociacoesDaSemanaRetrasada()
    ])
    .then(periodo => {

        return periodo
            .reduce((novoArray, item) => novoArray.concat(item),
            [])
            .sort((a, b) => a.data.getTime() - b.data.getTime());
    })
    .catch(err => {

        console.log(err);
        throw new Error('Não foi possível obter as negociações do período')
    });
}

// código posterior omitido
```

O resultado de `b.data.getTime()` retorna um número que representa uma data, algo bem oportuno para nossa lógica, pois podemos subtrair o resultado do primeiro elemento pelo segundo. Se forem iguais, o resultado será sempre 0; se o primeiro for maior

que o segundo, teremos um número positivo; e se o primeiro for menor que o segundo, teremos um número negativo. Atendemos plenamente a exigência do método `sort()`.

No entanto, queremos uma ordem descendente, por isso precisamos subtrair `b.data.getTime()` de `a.data.getTime()`:

```
// client/app/domain/negociacao/NegociacaoService.js
// código anterior omitido

obtemNegociacoesDoPeriodo() {

    return Promise.all([
        this.obtemNegociacoesDaSemana(),
        this.obtemNegociacoesDaSemanaAnterior(),
        this.obtemNegociacoesDaSemanaRetrasada()
    ])
    .then(periodo => {

        return periodo
            .reduce((novoArray, item) => novoArray.concat(item),
                [])
            .sort((a, b) => b.data.getTime() - a.data.getTime());
    })
    .catch(err => {

        console.log(err);
        throw new Error('Não foi possível obter as negociações do período')
    });
}

// código posterior omitido
```

Agora temos a garantia de que o período importado sempre nos retornará uma lista ordenada com ordenação descendente pela data das negociações. Daí, vem aquela velha pergunta: será que podemos simplificar ainda mais nosso código?

Vejamos o código que escrevemos por último, aquele que

ordena o período:

```
// ANALISANDO O CÓDIGO
.then(periodo => {

    return periodo
        .reduce((novoArray, item) => novoArray.concat(item), [])
        .sort((a, b) => b.data.getTime() - a.data.getTime());
})
```

Perceba que temos uma única instrução dentro do bloco do `then` que retorna o novo array ordenado. Podemos remover o bloco da arrow function, inclusive a instrução `return`, pois arrow functions possuem uma única instrução sem bloco, mas já assumem um `return` implícito.

```
// ANALISANDO O CÓDIGO
.then(periodo => periodo
    .reduce((novoArray, item) => novoArray.concat(item), [])
    .sort((a, b) => b.data.getTime() - a.data.getTime())
)
```

Aliás, se olharmos os outros métodos da classe `NegociacaoService`, todos possuem apenas uma única instrução no bloco do `then()`.

Com todas as simplificações, nossa classe `NegociacaoService` no final estará assim:

```
// client/app/domain/negociacao/NegociacaoService.js

class NegociacaoService {

    constructor() {

        this._http = new HttpService();
    }

    obterNegociacoesDaSemana() {
```

```

        return this._http
            .get('negociacoes/semana')
            .then(
                dados =>
                    dados.map(objeto =>
                        new Negociacao(new Date(objeto.data), obj
eto.quantidade, objeto.valor))
                ,
                err => {

                    throw new Error('Não foi possível obter as ne
gociações da semana');
                }
            );
    }

    obterNegociacoesDaSemanaAnterior() {

        return this._http
            .get('negociacoes/anterior')
            .then(
                dados => dados.map(objeto =>
                    new Negociacao(new Date(objeto.data), objeto.
quantidade, objeto.valor))
                ,
                err => {

                    throw new Error('Não foi possível obter as ne
gociações da semana anterior');
                }
            );
    }

    obterNegociacoesDaSemanaRetrasada() {

        return this._http
            .get('negociacoes/retrasada')
            .then(
                dados => dados.map(objeto =>
                    new Negociacao(new Date(objeto.data), objeto.
quantidade, objeto.valor))
                ,
                err => {

                    throw new Error('Não foi possível obter as ne
gociações da semana retrasada');
                }
            );
    }

```

```

        }
    );
}

obtemNegociacoesDoPeriodo() {

    return Promise.all([
        this.obtemNegociacoesDaSemana(),
        this.obtemNegociacoesDaSemanaAnterior(),
        this.obtemNegociacoesDaSemanaRetrasada()
    ])
    .then(periodo => periodo
        .reduce((novoArray, item) => novoArray.concat(item),
[]))
        .sort((a, b) => b.data.getTime() - a.data.getTime())
    )
    .catch(err => {

        console.log(err);
        throw new Error('Não foi possível obter as negociações do período')
    });
}
}

```

Excelente! Todavia, se vocês são bons observadores, devem ter reparado que podemos importar várias vezes as mesmas negociações. Que tal evitarmos que isso aconteça? A ideia é não importar novamente negociações de uma mesmo dia, mês e ano.

13.8 IMPEDINDO IMPORTAÇÕES DUPLICADAS

Precisamos adotar algum critério para que possamos determinar se a negociação que estamos importando já havia sido importada ou não. Em outras palavras, precisamos de alguma lógica que indique se uma negociação é igual a outra. Será que o operador `==` pode nos ajudar? Vamos realizar alguns testes no

Console do navegador.

```
n1 = new Negociacao(new Date(), 1, 100);  
n2 = new Negociacao(new Date(), 1, 100);  
n1 == n2 // o retorno é false!
```

Não podemos simplesmente usar o operador `==` para comparar objetos que não sejam dos tipos literais `string`, `number`, `boolean`, `null` e `undefined`, pois ele testa se as variáveis apontam para o mesmo endereço na memória.

Se quisermos que `n1` seja igual a `n2`, podemos fazer isso no console:

```
n1 = n2  
n1 == n2 // o retorno é true
```

No exemplo anterior, a variável `n1` passa a apontar para o mesmo objeto `n2`. Como `n1` e `n2` agora referenciam o mesmo objeto, o operador `==` retornará `true` na comparação entre ambos.

Que tal realizarmos uma comparação entre as datas das negociações? Cairíamos no mesmo problema, pois estaríamos comparando dois objetos do tipo `Date`, e o operador `==` apenas testaria se as propriedades do objeto apontam para o mesmo objeto na memória. Há saída? Claro que há.

Vamos realizar a comparação entre datas dessa maneira:

```
n1.data.getDate() == n2.data.getDate() && n1.data.getMonth() == n  
2.data.getMonth() && n1.data.getFullYear() == n2.data.getFullYear  
( ) // resultado é true!
```

Os métodos `Date` que usamos devolvem números e, por padrão, o operador `==` com este tipo literal compara o valor, e

não a referência na memória. Excelente, conseguimos saber se duas negociações são iguais. Será?

Precisamos melhorar o critério de comparação, pois podemos ter negociações de uma mesma data em nossa lista de negociações. Que tal considerarmos também na comparação a quantidade e o valor?

Testando no console do navegador:

```
n1.data.getDate() == n2.data.getDate() && n1.data.getMonth() == n2.data.getMonth() && n1.data.getFullYear() == n2.data.getFullYear() && n1.quantidade == n2.quantidade && n1.valor == n2.valor // retorna true!
```

Não é nada orientado a objetos termos de repetir essa lógica em todos os lugares em que precisamos testar a igualdade entre negociações, uma vez que ela faz parte do domínio de uma negociação. Vamos alterar a classe `Negociacao` e adicionar o método `equals()` que estará presente em todas as instâncias da classe:

```
// client/app/domain/negociacao/Negociacao.js

class Negociacao {

    // código anterior omitido

    equals(negociacao) {

        return this.data.getDate() == negociacao.data.getDate()
            && this.data.getMonth() == negociacao.data.getMonth()
            && this.data.getFullYear() == negociacao.data.getFullYear()
            && this.quantidade == negociacao.quantidade
            && this.valor == negociacao.valor;

    }
}
```

O método compara a própria instância que está chamando o método `equals()` com outra negociação recebida no método.

Recarregando nossa página e testando nosso método no console do navegador, teremos:

```
n1 = new Negociacao(new Date(), 1, 100);
n2 = new Negociacao(new Date(), 1, 100);
n1.equals(n2); // retorna true
```

Perfeito, mas podemos enxugar bastante o método `equals()` que acabamos de criar.

Quando o critério de comparação entre objetos é baseado no valor de todas as propriedades que possui, podemos usar o seguinte truque com `JSON.stringify()`:

```
// client/app/domain/negociacao/Negociacao.js

class Negociacao {

    // código anterior omitido

    equals(negociacao) {

        return JSON.stringify(this) == JSON.stringify(negociacao)
    }
}
```

A função `JSON.stringify()` converte um objeto em uma representação textual, ou seja, uma string. Convertendo os objetos envolvidos na comparação para string, podemos usar tranquilamente o operador `==` para testar a igualdade entre eles. Bem mais enxuto, não? Porém, essa solução só funciona se o critério de comparação for os valores de todas as propriedades que o objeto possui.

Agora, precisamos alterar a lógica do método `importaNegociacoes` de `NegociacaoController` para que importe apenas novas negociações.

13.9 AS FUNÇÕES `FILTER()` E `SOME()`

Precisamos filtrar a lista de negociações retornadas do servidor, considerando apenas aquelas que ainda não constam no array retornado por `this._negociacoes.toArray()`. A boa notícia é que arrays já possuem uma função com essa finalidade, a `filter()`.

Vejamos isoladamente no console como ela funciona:

```
lista = ['A', 'B', 'C', 'D'];
listaFiltrada = lista.filter(letra => letra == 'B' || letra == 'C');
```

A função `filter()` recebe uma função com a estratégia que será aplicada em cada elemento da lista. Se a função retornar `true` para um elemento, ele será adicionado na nova lista retornada por `filter()`. Se retornar `false`, o elemento será ignorado.

No exemplo anterior, `listaFiltrada` conterá apenas as letras "B" e "C".

```
// client/app/controllers/NegociacaoController.js
// código anterior omitido
```

```
importaNegociacoes() {
  this._service
    .obtemNegociacoesDoPeriodo()
    .then(negociacoes => {
```

```

        negociacoes
        .filter(novaNegociacao => console.log('lógica do filtro aqui'))
        .forEach(negociacao => this._negociacoes.adiciona(negociacao));

        this._mensagem.texto = 'Negociações do período importadas com sucesso';
    })
    .catch(err => this._mensagem.texto = err);
}
// código posterior omitido

```

Analisando o código, vemos que o encadeamento da chamada da função `forEach()` trabalhará com a lista filtrada. Porém, ainda não temos qualquer lógica de filtro aplicada.

Para cada item processado pelo nosso filtro, precisamos verificar se ele já existe em `this._negociacoes.toArray()`. Se não existir, o item fará parte da nova lista. Podemos implementar essa lógica de diversas formas, mas esse é o momento oportuno para aprendemos a utilizar a função `some()`.

A função `some()` itera em cada elemento de uma lista. Assim que encontrar algum (*some*) elemento de acordo com alguma lógica que retorne `true`, parará imediatamente de iterar no array retornando `true`. Se nenhum elemento atender ao critério de comparação, o array terá sido percorrido até o fim e o retorno de `some()` será `false`.

Que tal combinarmos `some()` com `filter()` para não importarmos negociações duplicadas? Primeiro, vejamos como nosso código ficará, para em seguida tecermos comentários sobre ele:

```
// client/app/controllers/NegociacaoController.js
```

```
// código anterior omitido

importaNegociacoes() {

    this._service
        .obtemNegociacoesDoPeriodo()
        .then(negociacoes => {

            negociacoes.filter(novaNegociacao =>

                !this._negociacoes.toArray().some(negociacaoExi
stente =>

                    novaNegociacao.equals(negociacaoExistente)))

                .forEach(negociacao => this._negociacoes.adiciona(neg
ociacao));

            this._mensagem.texto = 'Negociações do período import
adas com sucesso';
        })
        .catch(err => this._mensagem.texto = err);
    }
}

// código posterior omitido
```

A lógica da função `filter()` da lista que desejamos importar é o retorno da função `some()` aplicada na lista já existente. Lembre-se de que `some()` pode retornar `true` ou `false`, valores que satisfazem a função `filter()`. No caso, queremos filtrar nossa lista de negociações a importar considerando apenas as que não fazem parte da lista de negociações já importadas.

Através de `some()` aplicada na lista de negociações já importadas, verificamos se a negociação que desejamos importar não existe na lista. Mas como isso, a função retornaria `false`, então precisamos inverter esse valor para `true`, pois apenas retornando `true` a função `filter()` considerará o elemento na

nova lista. Veja que utilizamos a função `equals()` que criamos em nosso modelo de `Negociacao`.

No final da nossa cadeia de funções, `forEach()` adicionará apenas as negociações que ainda não foram importadas.

Tudo muito bonito, mas o que acontece se fecharmos nosso navegador depois de incluirmos duas ou mais negociações? Com certeza perderemos todo nosso trabalho, porque não há persistência de dados em nossa aplicação. Aliás, este será nosso próximo assunto.

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/13>

Parte 3 - A revelação

Exausto, ele não aguentou e desabou. Durante seu esvaecimento, teve uma nova visão. Nela, Lampião tirava o chapéu, sentava ao lado da fogueira e apontava para a direção oposta. Depois de acordar, foi então que ele se deu conta de que o butim do caminho do cangaceiro é o próprio caminho. Então, ele deu um último gole em sua moringa e começou a trilhar o caminho para lugar nenhum e, ao mesmo tempo, todo lugar, cada vez mais preparado para as surpresas da vida.

A ALGIBEIRA ESTÁ FURADA!

"Rir, antes da hora, engasga." - Grande Sertão: Veredas

Nossa aplicação é funcional. No entanto, se fecharmos e abrirmos novamente o navegador, perdemos todos os dados que havíamos cadastrado. Uma solução é enviarmos os dados da aplicação para uma API de algum servidor que os armazene em um banco de dados, para que mais tarde possamos reavê-los consumindo outra API.

Contudo, não usaremos essa abordagem. Gravaremos os dados em um banco de dados que todo navegador possui, o **IndexedDB**.

14.1 INDEXEDDB, O BANCO DE DADOS DO NAVEGADOR

O IndexedDB é um banco de dados transacional especificado pela W3C (<https://www.w3.org/>), presente nos mais diversos navegadores do mercado. Diferente de bancos relacionais que

utilizam SQL, IndexedDB não se pauta em esquemas (estruturas pré-definidas de tabelas, colunas e seus tipos), mas em objetos JavaScript que podem ser persistidos e buscados com menos burocracia. Este tipo de banco faz muito pouco para validar os dados persistidos, a validação é de responsabilidade da aplicação que o utiliza.

Não podemos sair integrando o IndexedDB em nosso projeto antes de sabermos como ele funciona. É por isso que criaremos o arquivo `db.html`, para que possamos praticar primeiro, e só depois o integraremos em nossa aplicação. Vamos criar o arquivo `client/db.html` com a seguinte estrutura

```
<!-- client/dn.html -->

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Aprendendo IndexedDB</title>
</head>

<body>
  <script>

    /* nossos testes entraram aqui! */

  </script>
</body>
</html>
```

O primeiro passo é requisitarmos uma abertura de conexão ao IndexedDB. É pelo objeto globalmente disponível `indexedDB` e seu método `open()` que realizamos esse passo:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
```

```
</head>
<body>
  <script>

    const openRequest = indexedDB.open("jscangaceiro", 1);

  </script>
</body>
```

O método `open()` recebe dois parâmetros. O primeiro é o nome do banco em que desejamos nos conectar, e o segundo, um número definido por nós que indica a versão do banco - no caso, o valor 1 foi escolhido. Por enquanto, esse segundo parâmetro não terá grande importância para nós, por isso o deixaremos temporariamente de lado nas explicações.

O retorno do método `open()` é uma instância de **IDBOpenDBRequest**, ou seja, uma requisição de abertura do banco. Toda vez que realizamos uma requisição de abertura, precisaremos lidar com uma **tríade** de eventos, são eles:

`onupgradeneeded`

`onsuccess`

`onerror`

ATENÇÃO

Só recarregue a página no navegador quando esse procedimento for aqui indicado. Caso o recarregamento seja feito antes de pontos-chaves do nosso código, você terá problemas de atualização de banco que só serão resolvidos mais tarde.

```
<!-- client/dn.html -->
<! código anterior omitido

<body>
  <script>

    const openRequest = indexedDB.open('jscangaceiro', 1);

    // lidando com a tríade de eventos!

    openRequest.onupgradeneeded = function(e) {

      console.log('Cria ou altera um banco já existente');
    };

    openRequest.onsuccess = function(e) {

      console.log('Conexão obtida com sucesso!');
    };

    openRequest.onerror = function(e) {

      console.log(e.target.error);
    };

  </script>
</body>

<!-- código posterior omitido -->
```

Atribuímos a `openRequest.onupgradeneeded` uma função que será chamada somente se o banco que estivermos nos conectando não existir, ou quando estivermos atualizando um banco já existente, assunto que abordaremos em breve. Em seguida, em `openRequest.onsuccess()`, atribuímos a função que será chamada quando uma conexão for obtida com sucesso. Por fim, em `openRequest.onerror`, atribuímos a função que será chamada caso algum erro ocorra no processo de abertura da conexão. Será através de `e.target.error` que teremos acesso à causa do erro.

Antes de continuarmos que tal usarmos arrow functions para termos um código menos verboso? Não há problema nenhum utilizar este tipo de função para esses três eventos:

```
<!-- client/dn.html -->
<!-- código anterior omitido -->

<body>
  <script>

    const openRequest = indexedDB.open('jscangaceiro', 1);

    openRequest.onupgradeneeded = e => {

      console.log('Cria ou altera um banco já existente');
    };

    openRequest.onsuccess = e => {

      console.log('Conexão obtida com sucesso!');
    };

    openRequest.onerror = e => {

      console.log(e.target.error);
    };

  </script>
```

```
</body>
```

```
<!-- código posterior omitido -->
```

Recarregue sua página no navegador para que possamos verificar no console se as mensagens de `onupgradeneeded` e `onsuccess` são exibidas. Lembre-se de que a primeira será exibida porque estamos criando o banco pela primeira vez e a segunda, porque teremos obtido uma conexão.

Na primeira vez, veremos as seguintes mensagens:

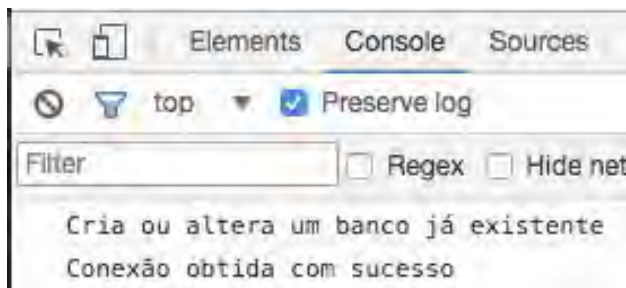


Figura 14.1: Duas mensagens no console

Se recarregarmos mais uma vez a página, veremos apenas uma das mensagens. Faz sentido, pois o banco já foi criado:

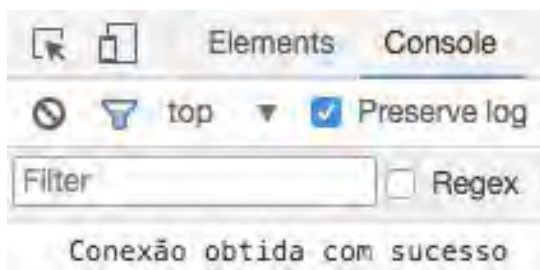


Figura 14.2: Uma mensagem no console

Por mais que as mensagens anteriores indiquem que o banco foi criado, um cangaceiro só acreditará vendo. Podemos visualizar o banco criado no Chrome através de seu console, na aba Application . No canto esquerdo, na seção Storage , temos IndexedDB . Clicando neste item, ele expandirá exibindo o banco jscangaceiro :

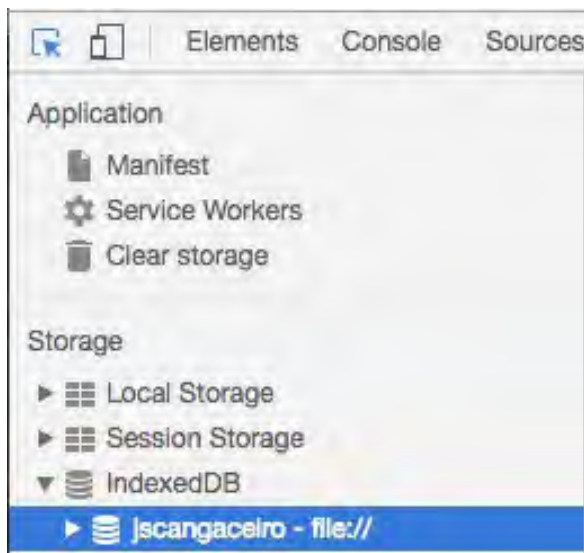


Figura 14.3: Visualizando o banco no Chrome

Agora que já entendemos o papel da tríade de eventos, chegou a hora de interagirmos com o banco que acabamos de criar.

14.2 A CONEXÃO COM O BANCO

Conexões são obtidas através do evento `onsuccess` . O único parâmetro recebido pela função associada ao evento nos dá acesso a uma instância de `IDBDatabase` , aquela que representa nossa

conexão. Acessando a conexão através de `e.target.result` :

```
<!-- client/dn.html -->
<!-- código anterior omitido -->

<body>
  <script>

    const openRequest = indexedDB.open('jscangaceiro', 1);

    openRequest.onupgradeneeded = e => {

      console.log('Criando ou atualizando o banco');
    };

    openRequest.onsuccess = e => {

      console.log('Conexão realizada com sucesso');

      // e.target.result é uma instância de IDBDatabase

      e.target.result;
    };

    openRequest.onerror = e => {

      console.log(e.target.error);
    };

  </script>
</body>
<!-- código posterior omitido -->
```

Já sabemos onde se encontra a conexão, mas precisamos armazená-la em alguma variável para podermos utilizá-la em diferentes ocasiões em nosso programa. Para isso, vamos criar a variável `connection` antes do código que já temos. Estando no escopo global, ela pode guardar o resultado de `e.target.result` :

```
<!-- client/dn.html -->
<!-- código anterior omitido -->
```



```

<body>
  <script>

    // PRECISA SER LET PARA ACEITAR UMA NOVA ATRIBUIÇÃO
    let connection = null;

    const openRequest = indexedDB.open('jscangaceiro', 1);

    openRequest.onupgradeneeded = e => {

      console.log('Criando ou atualizando o banco');
    };

    openRequest.onsuccess = e => {

      console.log('Conexão realizada com sucesso');

      // a variável guarda uma referência para a conexão

      connection = e.target.result;
    };

    openRequest.onerror = e => {

      console.log(e.target.error);
    };

  </script>
</body>
<!-- código posterior omitido -->

```

Excelente, temos uma conexão. Porém, antes de pensarmos em interagir com o banco, precisamos criar uma nova **Object Store**. Entenda uma store como algo análogo às tabelas do mundo SQL.

14.3 NOSSA PRIMEIRA STORE

Criamos novas stores através de uma conexão com o banco durante o evento `onupgradeneeded`. Porém, quando este evento é disparado, o valor da variável `connection` ainda é `null`, pois

ela só receberá seu valor durante o evento `onsuccess` . E agora?

A boa notícia é que podemos obter uma conexão no evento `onupgradeneeded` da mesma maneira que obtemos em `onsuccess` :

```
<!-- client/dn.html -->
<!-- código anterior omitido -->

<body>
  <script>

    let connection = null;

    const openRequest = indexedDB.open('jscangaceiro', 1);

    openRequest.onupgradeneeded = e => {

      console.log('Cria ou altera um banco já existente');

      // obtendo a conexão!
      connection = e.target.result;
    };

    openRequest.onsuccess = e => {

      console.log('Conexão realizada com sucesso');
      connection = e.target.result;
    };

    openRequest.onerror = e => {

      console.log(e.target.error);
    };

  </script>
</body>
<!-- código posterior omitido -->
```

Excelente! Da maneira que estruturamos nosso código, a conexão pode vir tanto de `onupgradeneeded` quanto de `onsuccess` . Já temos a moringa e a cartucheira em mãos para

criarmos nosso store.

Usaremos a seguinte lógica de criação da store. Primeiro, verificaremos se a store que estamos tentando criar já existe; se existir, vamos apagá-la antes de criá-la. A razão dessa abordagem é que o evento `onupgradeneeded` também pode ser disparado quando estivermos atualizando nosso banco.

Para sabermos se uma store existe ou não durante uma atualização, usamos o método `connection.objectStoreNames.contains()` passando o nome da store. Para apagar uma store e criarmos uma nova, usamos `connection.deleteObjectStore()` e `connection.createObjectStore()`, respectivamente:

```
<!-- client/dn.html -->
<!-- código anterior omitido -->
<body>
  <script>

    let connection = null;

    const openRequest = indexedDB.open('jscangaceiro', 1);

    openRequest.onupgradeneeded = e => {

      console.log('Cria ou altera um banco já existente');
      connection = e.target.result;

      // pode ser que o evento esteja sendo disparado durante
      // uma atualização,
      // nesse caso, verificamos se a store existe, se existi
r
      // apagamos a store atual antes de criarmos uma nova

      if(connection.objectStoreNames.contains('negociacoes'))
      {
        connection.deleteObjectStore('negociacoes');
      }
    }
  }
</script>
</body>
```

```

        connection.createObjectStore('negociacoes', { autoIncrement: true });
    };

    // código posterior omitido

</script>
</body>
<!-- código posterior omitido -->

```

Um ponto a destacar é o segundo parâmetro de `connection.createObjectStore()`. Nele, passamos um objeto JavaScript com a propriedade `autoIncrement: true`. Ela é importante, pois criará internamente um identificador único para os objetos que salvamos na store.

Ainda sobre a lógica de atualização, ela poderia ser mais rebuscada, mas apagar a store antiga e criar uma nova é suficiente para nossa aplicação. Contudo, enfrentaremos um problema.

Recarregando nossa página, apenas o evento `onsuccess` é chamado. O evento `onupgradeneeded` só é chamado quando estamos criando um novo banco ou atualizando um já existente. Como já temos um banco criado, precisamos indicar para o IndexedDB que desejamos realizar uma atualização. Como resolver?

14.4 ATUALIZAÇÃO DO BANCO

Para que o evento `onupgradeneeded` seja disparado novamente, a versão do banco que estamos criando deve ser superior à versão do banco existente no navegador. Sendo assim, basta passarmos como segundo parâmetro para a função `indexedDB.open()` qualquer valor que temos certeza de que seja

superior à versão do banco presente no navegador. Em nosso caso, o valor 2 é suficiente:

```
<!-- client/db.html -->
<!-- código anterior omitido -->

<body>
  <script>
    let connection = null;

    // INCREMENTAMOS O NÚMERO DA VERSÃO
    const openRequest = indexedDB.open('jscangaceiro', 2);

    // código posterior omitido
  </script>
</body>
<!-- código posterior omitido -->
```

Desta forma, **recarregando** a página, o evento `onupgradeneeded` será disparado e criará nossa `storage` `negociacoes`.

Atenção, feche e abra o navegador antes de realizar o novo teste. A aba `Application` faz um cache das últimas informações que exibe e a versão do banco continuará 1, mesmo com a nossa mudança. Fechar e abrir o browser realiza um refresh nesta área, exibindo corretamente a nova versão do nosso banco. É algo que pode confundir o leitor.

Por fim, ao acessarmos a aba `Application`, veremos que a versão do banco mudou e que a `Object Store` `negociacoes` foi criada, mas ainda está vazia.



Figura 14.4: Object store negociacoes

Temos nossa store prontinha para persistir dados, assunto da próxima seção.

14.5 TRANSAÇÕES E PERSISTÊNCIA

Chegou ao momento de persistirmos algumas negociações. Para isso, precisamos importar o script `Negociacao.js` para que possamos criar instâncias dessa classe:

```
<!-- client/db.html -->
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Aprendendo IndexedDB</title>
</head>
<body>

  <!-- código anterior omitido -->
  <script src="app/domain/negociacao/Negociacao.js"></script>
</body>
```

```
</html>
```

Agora, vamos criar a função `adiciona()` que mais tarde será chamada por nós para adicionarmos negociações no banco:

```
<!-- client/db.html -->
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Aprendendo IndexedDB</title>
</head>

<body>
  <script>
    let connection = null;

    const openRequest = indexedDB.open('jscangaceiro', 2);

    openRequest.onupgradeneeded = e => {
      // código omitido
    };

    openRequest.onsuccess = e => {
      // código omitido
    };

    openRequest.onerror = e => {
      // código omitido
    };

    // nova função para adicionar negociações
    function adiciona() {

    }
  </script>
  <script src="js/app/domain/negociacao/Negociacao.js"></script>
</body>
</html>
```

Quando a função for chamada, ela terá de ser capaz de gravar uma instância de negociação no IndexedDB .

O primeiro passo no processo de persistência é obtermos uma transação de escrita para, em seguida, através dessa transação, termos acesso à store na qual desejamos persistir objetos. Primeiro, vejamos o código:

```
// client/dn.html
// código anterior omitido

function adiciona() {

    // NOVA INSTÂNCIA DE NEGOCIACAO
    const negociacao = new Negociacao(new Date(), 200, 1);

    const transaction = connection.transaction(['negociacoes'], 'readwrite');
    const store = transaction.objectStore('negociacoes');
}

// código posterior omitido
```

O método `connection.transaction()` recebe como primeiro parâmetro um array com o nome da store cuja transação queremos criar e, como segundo parâmetro, o seu tipo. Passamos `readwrite` para uma transação que permita escrita. Porém, se quiséssemos somente leitura, bastaria trocarmos para `readonly`.

Por intermédio da store transacional, podemos realizar operações de persistência, como inclusão, alteração e remoção.

```
// client/dn.html
// código anterior omitido

function adiciona() {

    const negociacao = new Negociacao(new Date(), 200, 1);

    const transaction = connection.transaction(['negociacoes'], 'readwrite');
    const store = transaction.objectStore('negociacoes');
```



```

    // ATRAVÉS DA STORE, REQUISITAMOS UMA INCLUSÃO
    const request = store.add(negociacao);
}

```

// código posterior omitido

Após criarmos uma instância de `Negociacao`, utilizamos o método `store.add()` para persisti-la. No entanto, essa operação é apenas uma requisição de inclusão. Precisamos saber se ela foi bem-sucedida ou não através das funções de callback que passaremos para os eventos `onsuccess` e `onerror` da requisição:

```

// client/dn.html
// código anterior omitido

function adiciona() {

    const negociacao = new Negociacao(new Date(), 200, 1);

    const transaction = connection.transaction(['negociacoes'], 'r
eadwrite');
    const store = transaction.objectStore('negociacoes');
    const request = store.add(negociacao);

    request.onsuccess = e => {

        console.log('negociação salva com sucesso');
    };

    request.onerror = e => {

        console.log('Não foi possível salvar a negociação')
    };
}

```

// código posterior omitido

Agora só nos resta chamarmos o método `adiciona()` pelo console do Chrome:

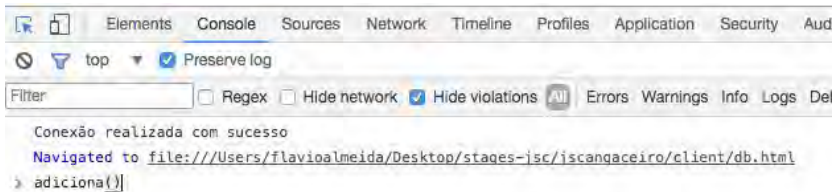


Figura 14.5: Chamando adiciona() pelo console

Com a execução do método, temos a seguinte saída:



Figura 14.6: Resultado de adiciona() no console

Excelente, agora vamos verificar na aba Application os dados da store negociacoes , clicando diretamente na store:



Figura 14.7: Dados da store

Podemos simplificar ainda mais nosso código evitando declarar variáveis intermediárias, encadeando as chamadas de função e removendo o bloco de algumas arrow functions. Nossa função adiciona() ficará assim:

```
// client/db.html
```

```
// código anterior omitido

function adiciona() {

    const negociacao = new Negociacao(new Date(), 200, 1);

    // ENCADEOU AS CHAMADAS
    const request = connection
        .transaction(['negociacoes'], 'readwrite')
        .objectStore('negociacoes')
        .add(negociacao);

    // REMOVEU BLOCO
    request.onsuccess = e =>
        console.log('negociação salva com sucesso');

    // REMOVEU BLOCO
    request.onerror = e =>
        console.log('Não foi possível salvar a negociação');
}

// código posterior omitido
```

Mais enxuto!

Conseguimos gravar a nossa primeira negociação! Mais adiante, seremos capazes de listar os dados.

14.6 CURSORES

Somos capazes de persistir negociações, mas faz parte de qualquer aplicação listá-las. Vamos criar imediatamente abaixo da função `adiciona()` a função `listaTodos()` :

```
<!-- client/db.html -->
<!-- código posterior omitido -->

<body>
  <script>
    // código anterior omitido
```

```

function adiciona() {

    // codigo omitido
}

function listaTodos() {

}

</script>
<script src="app/domain/negociacao/Negociacao.js"></script>
</body>

<!-- código posterior omitido -->

```

O processo é bem parecido com a inclusão de negociações. Precisamos de uma transação, mas em vez de chamarmos o método `add()`, em seu lugar chamaremos o método `openCursor()`, que nos retornará uma referência para um objeto que sabe iterar por cada objeto gravado na store:

```

// client/db.html
// código anterior omitido

function listaTodos() {

    const cursor = connection
        .transaction(['negociacoes'], 'readwrite')
        .objectStore('negociacoes')
        .openCursor();

    cursor.onsuccess = e => {

    };

    cursor.onerror = e => {

        // error.name, para ficar mais sucinta a informação

        console.log('Error: ' + e.target.error.name);

    };
}

```

```
// código posterior omitido
```

O evento `onsuccess` do nosso cursor será chamado na quantidade de vezes correspondente ao número de negociações armazenadas em nossa *object store*. Na primeira chamada, teremos acesso a um objeto ponteiro que sabe acessar a primeira negociação da store. Adicionaremos a negociação no array `negociacoes` que receberá todas as negociações que iterarmos.

Assim que adicionarmos a primeira negociação, solicitamos ao objeto ponteiro que vá para a próxima. Esse processo dispara uma nova chamada à função atribuída a `onsuccess`. Tudo se repetirá até que tenhamos percorrido todos os objetos da store. No final, teremos nosso array com todas as negociações lidas.

Nosso código ficará assim:

```
// client/db.html
// código anterior omitido

function listaTodos() {

  const negociacoes = [];

  const cursor = connection
    .transaction(['negociacoes'], 'readwrite')
    .objectStore('negociacoes')
    .openCursor();

  cursor.onsuccess = e => {

    // objeto ponteiro para uma negociação
    const atual = e.target.result;

    // se for diferente de null, é porque ainda há dado

    if(atual) {

      // atual.value guarda os dados da negociação
```

```

        negociacoes.push(atual.value);

        // vai para a próxima posição chamando onSuccess
s novamente
        atual.continue();

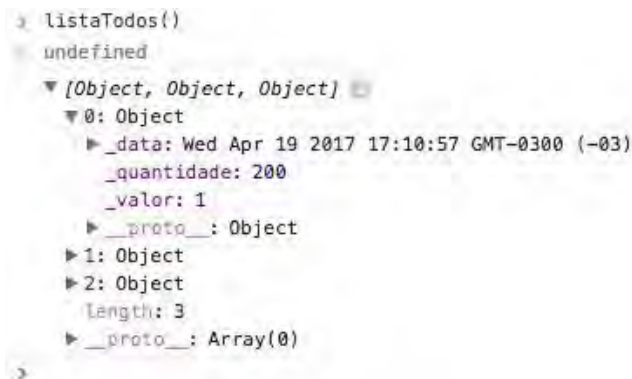
    } else {

        // quando atual for null, é porque não há mais
dados
        // imprimimos no console a lista de negociações
        console.log(negociacoes);
    }
};
}

```

Antes de testarmos nosso método, que tal adicionarmos mais duas negociações para termos pelo menos três elementos em nossa store? Para isso, basta abrirmos o console do Chrome e chamarmos duas vezes mais a função `adiciona()`.

Por fim, podemos ver nossa listagem chamando o método `listaTodos()` também através do terminal. O resultado será:



```

> listaTodos()
undefined
▼ [Object, Object, Object]
  0: Object
    _data: Wed Apr 19 2017 17:10:57 GMT-0300 (-03)
    _quantidade: 200
    _valor: 1
    __proto__: Object
  1: Object
  2: Object
    length: 3
    __proto__: Array(0)
>

```

Figura 14.8: Listando negociações

Será impresso no console um array com todas as nossas negociações. Contudo, se escrutinarmos cada item do array, veremos que é apenas um objeto com as propriedades `_data` , `_quantidade` e `_valor` . Isso acontece porque, quando gravamos um objeto em uma *store*, apenas suas propriedades são salvas.

Sendo assim, antes de adicionarmos no array de negociações, precisamos criar uma nova instância de `Negociacao` com base nos dados:

```
// client/db.html
// código anterior omitido

cursor.onsuccess = e => {

    const atual = e.target.result;

    if(atual) {

        // cria uma nova instância antes de adicionar no array
        const negociacao = new Negociacao(
            atual.value._data,
            atual.value._quantidade,
            atual.value._valor);

        negociacoes.push(negociacao);
        atual.continue();

    } else {

        console.log(negociacoes);

    }
};

cursor.onerror = e => console.log(e.target.error.name);

// código posterior omitido
```

Perfeito. Se olharmos a saída no console, vemos que temos um

array de instâncias da classe `Negociacao` .

Com tudo o que vimos, temos o pré-requisito para integrar nossa aplicação com o IndexedDB. Entretanto, você deve ter visto que nosso código não é lá um dos melhores. No próximo capítulo, veremos como organizá-lo ainda melhor, recorrendo a alguns padrões de projeto do *mercado.patterns*.

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/14>

COLOCANDO A CASA EM ORDEM

"A morte é para os que morrem." - Grande Sertão: Veredas

No capítulo anterior, aprendemos a utilizar o IndexedDB, e persistimos e buscamos negociações. No entanto, gastamos bastante linhas de código lidando com a abertura da conexão. Se mais tarde precisarmos da mesma conexão em outros pontos da nossa aplicação, teremos código repetido e difícil de manter.

Uma solução é isolar a complexidade de criação da conexão em uma classe. Aliás, utilizamos o padrão de projeto Factory para resolver um problema parecido com a complexidade de criação de Proxies. Chegou a hora de criarmos nossa `ConnectionFactory`.

15.1 A CLASSE CONNECTIONFACTORY

Vamos criar o arquivo `client/app/util/ConnectionFactory.js` com o método estático `getConnection()`. Este método nos devolverá uma

Promise pelo fato de a abertura de uma conexão com o banco ser realizada de maneira assíncrona. Por fim, lançaremos uma exceção no `constructor()`, pois só queremos que o acesso seja feito pelo método estático:

```
// client/app/util/ConnectionFactory.js

class ConnectionFactory {

    constructor() {

        throw new Error('Não é possível criar instâncias dessa classe');
    }

    static getConnection() {

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

        });
    }
}
```

Antes de prosseguirmos, vamos importar nossa classe `client/index.html`:

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

<script src="app/domain/negociacao/Negociacao.js"></script>
<script src="app/controllers/NegociacaoController.js"></script>
<script src="app/ui/converters/DateConverter.js"></script>
<script src="app/domain/negociacao/Negociacoes.js"></script>
<script src="app/ui/views/View.js"></script>
<script src="app/ui/views/NegociacoesView.js"></script>
<script src="app/ui/models/Mensagem.js"></script>
<script src="app/ui/views/MensagemView.js"></script>
<script src="app/util/ProxyFactory.js"></script>
<script src="app/util/Bind.js"></script>
<script src="app/util/ApplicationException.js"></script>
<script src="app/ui/converters/DataInvalidaException.js"></script>
```

```

<script src="app/domain/negociacao/NegociacaoService.js"></script>

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

<!-- importou aqui -->
<script src="app/util/ConnectionFactory.js"></script>

<script src="app/app.js"></script>
<!-- código posterior omitido -->

```

Temos o esqueleto da nossa classe, mas ela precisa seguir algumas regras. São elas:

Teremos uma conexão para a aplicação inteira, sendo assim, não importa quantas vezes seja chamado o método `getConnection()` , a conexão retornada deve ser a mesma.

Toda conexão possui o método `close()` , mas o programador não pode chamá-lo, porque a conexão é a mesma para a aplicação inteira. Só `ConnectionFactory` pode fechar a conexão.

Tendo essas restrições em mente, continuaremos a implementação do nosso método, preparando uma resposta para a tríade de eventos de abertura de uma conexão:

```

// client/app/util/ConnectionFactory.js

// variável que guarda a conexão

class ConnectionFactory {

  constructor() {

    throw new Error('Não é possível criar instâncias dessa classe');
  }
}

```

```

static getConnection() {

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

        const openRequest = indexedDB.open('jscangaceiro', 2);

        openRequest.onupgradeneeded = e => {

            };

        openRequest.onsuccess = e => {

            };

        openRequest.onerror = e => {

            };

        });
    }
}

```

Observe que utilizamos o valor `2`, porque foi a última versão do banco usada.

Hoje nossa aplicação possui apenas uma store chamada `negociacoes`. Mas nada nos impede de trabalharmos com mais de uma store. Para tornar nosso código flexível, vamos declarar um array logo acima da classe que por enquanto guarda a store `negociacoes`. Toda vez que precisarmos de novas stores, basta adicionarmos seu nome no array:

```

// array com uma store apenas
const stores = ['negociacoes'];

class ConnectionFactory {

    constructor() {

        throw new Error('Não é possível criar instâncias dessa classe');
    }
}

```

```

    }

    static getConnection() {

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

            const openRequest = indexedDB.open('jscangaceiro', 2);

            openRequest.onupgradeneeded = e => {

                // ITERA NO ARRAY PARA CONSTRUIR AS STORES
                stores.forEach(store => {

                    // lógica que cria as stores
                });

            };

            openRequest.onsuccess = e => {

            };

            openRequest.onerror = e => {

            };

        });
    }
}

```

A criação do array de `stores` foi necessária porque nossa classe não permite que propriedades sejam declaradas segundo nossa convenção. Vale lembrar que `ConnectionFactory` terá apenas um método estático. Agora, só nos resta implementarmos a lógica de criação das stores.

15.2 CRIANDO STORES

Na seção anterior, já preparamos o terreno para que possamos escrever a lógica de criação das nossas stores. Mas em vez de

escrevermos a lógica diretamente na função passada para `onupgradeneeded`, vamos criar outro método estático com esta finalidade, deixando clara nossa intenção:

```
// client/app/util/ConnectionFactory.js

const stores = ['negociacoes'];

class ConnectionFactory {

  constructor() {

    throw new Error('Não é possível criar instâncias dessa classe');
  }

  static getConnection() {

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

      const openRequest = indexedDB.open('jscangaceiro', 2);

      openRequest.onupgradeneeded = e => {

        // PASSA A CONEXÃO PARA O MÉTODO
        ConnectionFactory._createStores(e.target.result);
      };

      openRequest.onsuccess = e => {

      };

      openRequest.onerror = e => {

      };

    });
  }

  // CONVENÇÃO DE MÉTODO PRIVADO
  // SÓ FAZ SENTIDO SER CHAMADO PELA
  // PRÓPRIA CLASSE
}
```

```

static _createStores(connection) {

    stores.forEach(store => {

        // if sem bloco, mais sucinto!

        if(connection.objectStoreNames.contains(store))
            connection.deleteObjectStore(store);

        connection.createObjectStore(store, { autoIncrement:
true });
    });
}
}

```

Terminamos a implementação do evento `onupgradeneed` , agora implementaremos os eventos `onsuccess` e `onerror` . Neles, simplesmente passamos a conexão para `resolve()` e o erro para `reject()` :

```

// client/app/util/ConnectionFactory.js
// código anterior omitido

openRequest.onsuccess = e => {
    // passa o resultado (conexão) para a promise!
    resolve(e.target.result);
};

openRequest.onerror = e => {

    console.log(e.target.error)
    // passa o erro para reject da promise!
    reject(e.target.error.name)
};

// código posterior omitido

```

Já temos uma classe testável. Depois de recarregarmos nossa página `index.html` , através do console, vamos utilizar o método `getConnection()` .

```
// NO CONSOLE!
```

```
ConnectionFactory.getConnection().then(connection => console.log(
connection));
```

A instrução exibirá no console:

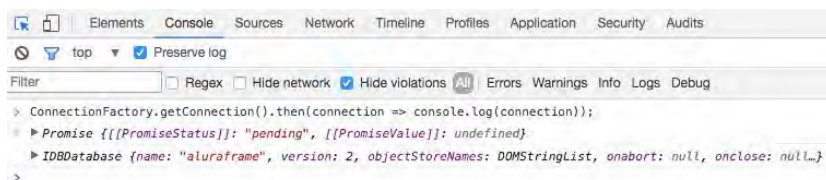


Figura 15.1: Conexão impressa no console

Está tudo funcionando corretamente. Porém, a cada chamada do método `getConnection()`, criaremos uma nova conexão e isso não se coaduna com a regra na qual precisamos ter uma única conexão para a aplicação. Veremos como solucionar isto a seguir.

15.3 GARANTINDO UMA CONEXÃO APENAS POR APLICAÇÃO

Para garantirmos que receberemos a mesma conexão toda vez que o método `getConnection()` for chamado, criaremos a variável `connection` antes da declaração da classe. Ela começará com valor `null`. Em seguida, logo no início do bloco da Promise, verificamos se ela possui valor; se possuir, passamos a conexão para `resolve()`, evitando assim repetir todo o processo de obtenção de uma conexão. Caso `connection` seja `null`, significa que estamos obtendo a conexão pela primeira vez e faremos isso em `onsuccess`, guardando-a na variável e passando-a também para `resolve()`:

```
// client/app/util/ConnectionFactory.js
```



```

const stores = ['negociacoes'];

// COMEÇA SEM CONEXÃO
let connection = null;

class ConnectionFactory {

  constructor() {

    throw new Error('Não é possível criar instâncias dessa classe');
  }

  static getConnection() {

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

      // SE UMA CONEXÃO JÁ FOI CRIADA,
      // JÁ PASSA PARA RESOLVE E RETORNA LOGO!

      if(connection) return resolve(connection);

      const openRequest = indexedDB.open('jscangaceiro', 2)

      openRequest.onupgradeneeded = e => {

        ConnectionFactory._createStores(e.target.result);
      };

      openRequest.onsuccess = e => {

        // SÓ SERÁ EXECUTADO NA PRIMEIRA VEZ QUE A CONEXÃO
        FOI CRIADA

        connection = e.target.result;
        resolve(e.target.result);
      };

      openRequest.onerror = e => {

        console.log(e.target.error)
        reject(e.target.error.name)
      };
    });
  }
}

```

```

    });
}

// código anterior omitido
}

```

Agora, no console, não importa a quantidade de vezes que o método `getConnection()` seja chamado, ele sempre retornará a mesma conexão! Excelente, mas nossa solução nos criou outro problema.

Com a página recarregada, vamos até o console para realizarmos um teste. Vamos digitar os nomes das variáveis `stores` e `connection`, que foram declaradas em `ConnectionFactory`:

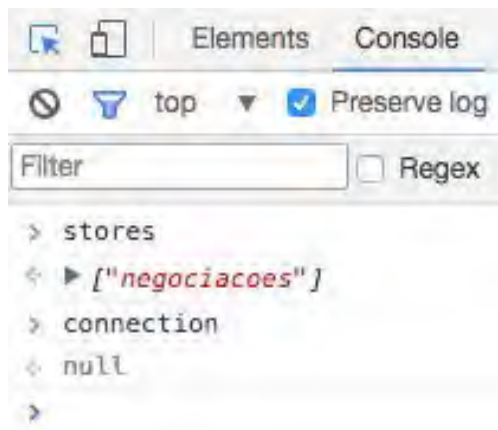


Figura 15.2: Acessando as variáveis

Somos capazes de acessar os valores das variáveis porque elas foram declaradas no escopo global. Além disso, nada impede que o programador acidentalmente, em outra parte da nossa aplicação,

realize atribuições indevidas para elas. E agora?

15.4 O PADRÃO DE PROJETO MODULE PATTERN

Para evitar que `stores` e `connection` vazem para o escopo global, podemos confiná-las em um **módulo**. Variáveis e funções declaradas em um módulo só serão acessíveis dentro do módulo, apenas as variáveis e funções que o programador explicitar serão acessíveis pelo restante da aplicação. Em outras palavras, podemos dizer que um módulo tem seu próprio escopo.

Podemos criar um escopo dentro de um arquivo `script` envolvendo todo seu código dentro de uma `function()`. Variáveis (`var`, `let` ou `const`) e funções declaradas no escopo de uma função são acessíveis apenas pelo bloco da função.

Vamos envolver todo o código do script `ConnectionFactory.js` dentro da função `tmp()`:

```
// client/app/util/ConnectionFactory.js

function tmp() {

    const stores = ['negociacoes'];
    let connection = null;

    class ConnectionFactory {

        constructor() {

            throw new Error('Não é possível criar instâncias dessa classe');
        }

        static getConnection() {
```

```

        // código omitido
    }

    static _createStores(connection) {

        // código omitido
    }
}

tmp(); // chama a função!

```

Reparem que a última instrução é a chamada da função `tmp()` para que seu bloco seja executado.

Depois de recarregarmos o navegador, se tentarmos acessar as variáveis `stores` e `connection`, receberemos *"... is not defined"*, pois elas não vivem mais no escopo global. Resolvemos um problema, mas criamos outro, pois também não seremos capazes de acessar `ConnectionFactory.getConnection()`. Classes também são encapsuladas no escopo de funções:

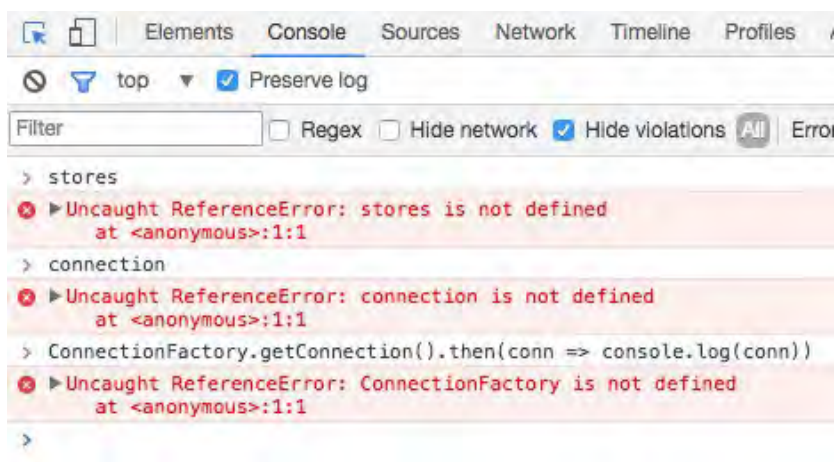


Figura 15.3: Problema ao acessar `ConnectionFactory`

A ideia é minimizarmos o uso do escopo global, mas ele ainda é um mal necessário, pois é através dele que tornamos uma classe acessível para outra.

Se todo o código do script está dentro da função `tmp()` e temos interesse em tornar global a definição da classe, podemos fazer com que a função retorne a classe. Ao armazenarmos o retorno em outra variável declarada fora do escopo da função `tmp()`, ela será mais uma vez acessível globalmente pela nossa aplicação:

```
// client/app/util/ConnectionFactory.s

function tmp() {

  const stores = ['negociacoes'];
  let connection = null;

  // RETORNA A DEFINIÇÃO DA CLASSE
  return class ConnectionFactory {

    constructor() {

      throw new Error('Não é possível criar instâncias dessa classe');
    }

    static getConnection() {

      // código omitido
    }

    static _createStores(connection) {

      // código omitido
    }
  }
}

// A VARIÁVEL VIVE NO ESCOPO GLOBAL
```

```
// PORQUE FOI DECLARADA FORA DA FUNÇÃO
const ConnectionFactory = tmp();
```

Depois de recarregar nossa página, podemos realizar um novo teste. Conseguiremos acessar a classe `ConnectionFactory`, mantendo `stores` e `connection` inacessíveis:



Figura 15.4: Acessando a classe pelo console

Muito bom, mas nosso código pode ficar ainda melhor. Como usamos a função `tmp()` para criar nosso módulo, ela é acessível pelo console, pois está no escopo global e nada nos impede de chamá-la quantas vezes quisermos. Outro ponto é que precisamos lembrar de chamá-la logo depois de sua declaração; caso contrário, a lógica de seu bloco jamais será processada.

Podemos resolver esse problema através de uma IIFE (*Immediately-invoked function expression*), ou *funções imediatas* no português. O primeiro passo é tornar a função `tmp()` anônima, isto é, removendo seu nome. Como não temos mais um nome, precisaremos retirar a instrução da chamada de `tmp()`:

```
// client/app/util/ConnectionFactory.s

function () {

    const stores = ['negociacoes'];
```

```

    let connection = null;

    return class ConnectionFactory {

        // código omitido
    }
}

```

Do jeito que está, o código não funcionará, pois não podemos declarar funções anônimas dessa forma. Contudo, podemos envolver a função entre parênteses:

```

// client/app/util/ConnectionFactory.s

(function () {

    const stores = ['negociacoes'];
    let connection = null;

    return class ConnectionFactory {

        // código omitido
    }

})

```

Esse passo evita erro de sintaxe, mas ainda não é capaz de invocar nossa função. Para isso, basta adicionarmos uma chamada `()` logo após os parênteses:

```

// client/app/util/ConnectionFactory.s

(function () {

    const stores = ['negociacoes'];
    let connection = null;

    return class ConnectionFactory {

        // código omitido
    }

})();

```

```
})();
```

Assim que nosso script for carregado, a função anônima dentro dos parênteses será executada. Como não temos mais um nome de função, ninguém poderá chamá-la através do terminal. Por fim, ela precisa guardar seu retorno em uma variável:

```
const ConnectionFactory = (function () {  
  
    const stores = ['negociacoes'];  
    let connection = null;  
  
    return class ConnectionFactory {  
  
        // código omitido  
    }  
  
})();
```

O conjunto das soluções que acabamos de implementar seguem o padrão de projeto **Module Pattern**. No caso, criamos um módulo que exporta a classe `ConnectionFactory`. Como foi dito nos capítulos anteriores, o ES2015 (ES6) trouxe uma solução nativa para a criação de módulos, mas ainda não é o momento certo de abordá-la.

Por fim, podemos ainda usar uma arrow function no lugar de `function()`:

```
// client/app/util/ConnectionFactory.js  
  
const ConnectionFactory = (() => {  
  
    const stores = ['negociacoes'];  
    let connection = null;  
  
    return class ConnectionFactory {  
  
        // código omitido  
    }  
})();
```



```
}  
})();
```

Agora que garantimos apenas uma conexão para a aplicação inteira, precisamos verificar a segunda regra que consiste no fechamento de uma conexão apenas pela classe `ConnectionFactory` :

15.5 MONKEY PATCH: GRANDES PODERES TRAZEM GRANDES RESPONSABILIDADES

Uma das regras que precisamos seguir é não permitirmos que o desenvolvedor feche a conexão através da conexão criada para `ConnectionFactory` , pois se ele fizer isso, estará fechando a conexão da aplicação inteira. Nada impede o desenvolvedor de chamar o método `close()` da conexão para fechá-la, até porque, é um "caminho natural" para fechá-la:

```
// EXEMPLO DE FECHAMENTO DE CONEXÃO PELO DESENVOLVEDOR
```

```
ConnectionFactory.getConnection().then(conn => conn.close());
```

A única forma aceitável de fechamento é através da própria `ConnectionFactory` , por exemplo, por meio de um método que ainda criaremos, chamado `closeConnection()` :

```
// EXEMPLO DE COMO DEVEMOS FECHAR A CONEXÃO, MÉTODO NÃO EXISTE AI  
NDA
```

```
ConnectionFactory.closeConnection();
```

Se toda conexão possui o método `close()` , como podemos evitar que o programador o chame? Uma solução é aplicarmos o **Monkey Patch**, que consiste na modificação de uma API já existente. No caso, vamos alterar o método `close()` original da

conexão para que lance uma exceção alertando o desenvolvedor de que não pode fechá-lo:

```
// client/app/util/ConnectionFactory.js
// código anterior omitido

openRequest.onsuccess = e => {

    connection = e.target.result;
    connection.close = () => {
        throw new Error('Você não pode fechar diretamente a conexão');
    };
    resolve(connection);
};

// código posterior omitido
```

A natureza dinâmica da linguagem JavaScript nos permite alterar funções em tempo de execução. No caso, estamos atribuindo uma nova lógica para o `close()` da conexão. Depois de recarregarmos nossa página, vamos realizar um novo teste fechando a conexão:

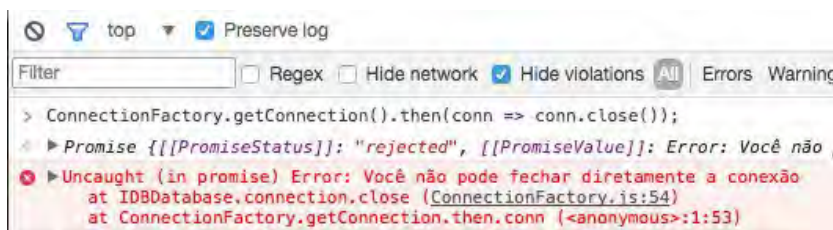


Figura 15.5: Conexão impressa no console

Agora, para concluirmos nossa classe, criaremos o método estático `closeConnection()` em `ConnectionFactory`. Este método será o encarregado de fechar a conexão para nós:

```
// client/app/util/ConnectionFactory.js
```

```

const ConnectionFactory = (() => {

  const stores = ['negociacoes'];
  let connection = null;

  return class ConnectionFactory {

    static getConnection() {

      // código omitido
    }

    static _createStores(connection) {

      // código omitido
    }

    // novo método!

    static closeConnection() {

      if(connection) {
        connection.close();
      }
    }
  }
})();

```

Para testarmos, basta recarregar a página e, no console, executar primeiro a instrução que cria uma conexão, para então fechá-la com nosso novo método:

```

// NO CONSOLE
ConnectionFactory.getConnection().then(conn => console.log(conn))
;
ConnectionFactory.closeConnection();

```

Recebemos uma mensagem de erro, aquela que programamos anteriormente ao tentarmos fechar a conexão:

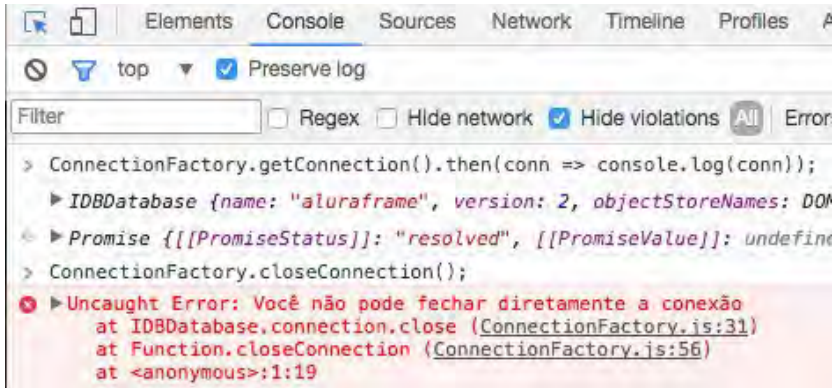


Figura 15.6: Não foi possível fechar a conexão

É fácil descobriremos a causa. Nós substituímos o método `close()` da conexão para evitar que o programador desavisado a feche. No entanto, precisamos do comportamento original para que `ConnectionFactory` seja capaz de fechar a conexão. E agora?

Uma solução é guardarmos uma referência para a função original antes de substituí-la. Essa referência será utilizada pelo método `closeConnection()` da nossa `ConnectionFactory`. Nossa classe no final ficará assim:

```
const ConnectionFactory = (() => {

  const stores = ['negociacoes'];
  let connection = null;

  // VARIÁVEL QUE ARMAZENARÁ A FUNÇÃO ORIGINAL
  let close = null;

  return class ConnectionFactory {

    constructor() {

      throw new Error('Não é possível criar instâncias dessa classe');
    }
  };
});
```

```

    }

    static getConnection() {

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

            if(connection) return resolve(connection);

            const openRequest = indexedDB.open('jscangaceiro'
, 2);

            openRequest.onupgradeneeded = e => {

                ConnectionFactory._createStores(e.target.resu
lt);

            };

            openRequest.onsuccess = e => {

                connection = e.target.result;

                // GUARDANDO A FUNÇÃO ORIGINAL!
                close = connection.close.bind(connection);

                connection.close = () => {
                    throw new Error('Você não pode fechar dir
etamente a conexão');
                };
                resolve(connection);
            };

            openRequest.onerror = e => {

                console.log(e.target.error)
                reject(e.target.error.name)
            };

        });

    }

    static _createStores(connection) {

        // código omitido

    }

```

```

static closeConnection() {

    if(connection) {

        // CHAMANDO O CLOSE ORIGINAL!
        close();
    }
}
}
})();

```

É na variável `close` que guardamos uma referência para a função original antes de realizarmos o *monkey patch*. Aliás, tivemos de apelar mais uma vez para a função `bind()` com `connection.close.bind(connection)`, para que a função não perca seu contexto (no caso, a instância de `connection`).

Com tudo em ordem, recarregaremos mais uma vez a página e realizaremos novamente nosso teste:

```

// NO CONSOLE
ConnectionFactory.getConnection().then(conn => console.log(conn))
;
ConnectionFactory.closeConnection();

```

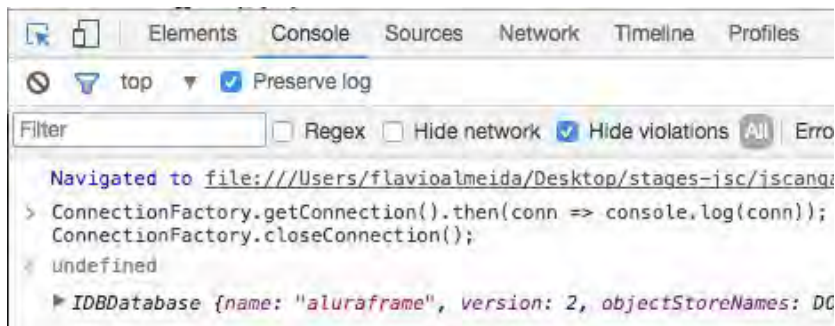


Figura 15.7: Fechando a conexão por ConnectionFactory

Excelente, conseguimos fechar a conexão pela

`ConnectionFactory` e, com isso, atendemos todos os requisitos para lidar com a conexão. Ainda falta organizarmos as operações de persistência, assunto do próximo capítulo.

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/15>

ENTRANDO NA LINHA

"Obedecer é mais fácil do que entender." - Grande Sertão:
Veredas

No capítulo anterior, aplicamos mais uma vez o padrão de projeto *Factory* criando uma classe com a responsabilidade de criar conexões. No entanto, o código anterior deixa a desejar, pois precisamos lidar com os detalhes de persistência de negociações. Se precisarmos incluir negociações em diversas partes da nossa aplicação, teremos de repetir muito código.

Vejamos um exemplo de uso da nossa conexão para incluir uma conexão:

```
// EXEMPLO DE USO APENAS
```

```
const negociacao = new Negociacao(new Date(), 200, 1);
```

```
ConnectionFactory
  .getConnection()
  .then(connection => {

    const request = connection
      .transaction(['negociacoes'], 'readwrite')
      .objectStore('negociacoes')
```



```

        .add(negociacao);

request.onsuccess = e => {

    console.log('negociação savlar com sucesso');
}

request.onerror = e => {

    console.log('Não foi possível salvar a negociação')
}

});

```

A boa notícia é que há outro padrão de projeto que pode nos ajudar a esconder a complexidade de persistência, o padrão de projeto **DAO**.

16.1 O PADRÃO DE PROJETO DAO

O padrão de projeto **DAO** (*Data Access Object*) tem como objetivo esconder os detalhes de acesso aos dados, fornecendo métodos de alto nível para o desenvolvedor. Em nosso caso, criaremos a classe `NegociacaoDao` que terá como dependência uma conexão. Vejamos um exemplo de como será a persistência de uma negociação através do DAO que criaremos a seguir:

```
// EXEMPLO DO QUE QUEREMOS CONSEGUIR FAZER COM NOSSO DAO
```

```
const negociacao = new Negocio(new Date(), 1, 100);
```

```

ConnectionFactory
    .getConnection()
    .then(conn => new NegociacaoDao(conn))
    .then(dao => dao.adiciona(negociacao))
    .then(() => console.log('Negociacao salva com sucesso'))
    .catch(err => console.log('Não foi possível salvar a negociaç
ão'));
});

```

Agora que já sabemos até onde queremos chegar, daremos início à construção do nosso DAO na próxima seção.

16.2 CRIANDO NOSSO DAO DE NEGOCIAÇÕES

O primeiro passo para criarmos nosso DAO é adotarmos a seguinte **convenção**. O nome da classe será o nome do modelo persistido, seguido do sufixo `Dao`. Se em nossa aplicação realizamos operações de persistência com `Negociacao`, então teremos a classe `NegociacaoDao`.

Além da convenção que vimos, a classe DAO precisará receber em seu `constructor()` a conexão com o banco. Isso porque, caso queiramos utilizar o mesmo DAO com o outro banco, basta passarmos a conexão correta, sem termos de alterar qualquer código na classe DAO.

Por fim, todos os métodos de persistência do DAO retornarão Promises, pois operações de persistência com IndexedDB são assíncronas.

Com base nas convenções que acabamos de aprender, vamos criar o arquivo `client/app/domain/negociacao/NegociacaoDao.js` e declarar o esqueleto do nosso DAO:

```
// client/app/domain/negociacao/NegociacaoDao.js
```

```
class NegociacaoDao {  
  
  constructor(connection) {  
  
    this._connection = connection;  
  
  }  
}
```

```

        this._store = 'negociacoes';
    }

    adiciona(negociacao) {

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

            /* lidaremos com a inclusão aqui */
        });
    }

    listaTodos() {

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

            /* lidaremos com os cursos aqui */
        });
    }
}

```

O esqueleto do nosso DAO possui também a propriedade `this._storage`. Ela guarda o nome da storage que será usada pelo DAO. Sem hiato, vamos importar nossa nova classe em `client/index.html`:

```

<!-- client/index.html -->
<!-- código anterior omitido -->

<script src="app/domain/negociacao/Negociacao.js"></script>
<script src="app/controllers/NegociacaoController.js"></script>
<script src="app/ui/converters/DateConverter.js"></script>
<script src="app/domain/negociacao/Negociacoes.js"></script>
<script src="app/ui/views/View.js"></script>
<script src="app/ui/views/NegociacoesView.js"></script>
<script src="app/ui/models/Mensagem.js"></script>
<script src="app/ui/views/MensagemView.js"></script>
<script src="app/util/ProxyFactory.js"></script>
<script src="app/util/Bind.js"></script>
<script src="app/util/ApplicationException.js"></script>
<script src="app/ui/converters/DataInvalidaException.js"></script>

<script src="app/domain/negociacao/NegociacaoService.js"></script>

```

```

<script src="app/util/HttpService.js"></script>
<script src="app/util/ConnectionFactory.js"></script>

<!-- importou aqui -->
<script src="app/domain/negociacao/NegociacaoDao.js"></script>

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

<!-- código posterior omitido -->

```

Agora que já temos o esqueleto do nosso DAO pronto, podemos dar início à implementação da lógica do método `adiciona()`, aquele responsável pela inclusão de novas negociações.

16.3 IMPLEMENTANDO A LÓGICA DE INCLUSÃO

Vamos implementar o método `adiciona()` de `NegociacaoDao`. Não teremos nenhuma novidade, aplicaremos tudo o que aprendemos até agora:

```

// client/app/domain/negociacao/NegociacaoDao.js
// código anterior omitido

adiciona(negociacao) {

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

        const request = this._connection
            .transaction([this._store], 'readwrite')
            .objectStore(this._store)
            .add(negociacao);

        request.onsuccess = e => resolve();
        request.onerror = e => {

            console.log(e.target.error);
            reject('Não foi possível salvar a negociação');
        }
    });
}

```

```

    }
  });
}

```

// código posterior omitido

Não há surpresas, simplesmente quando a operação de inclusão é bem-sucedida, chamamos `resolve()` passando uma mensagem de sucesso. E quando há algum erro, chamamos `reject()` passando uma mensagem de alto nível, indicando que a operação não foi realizada.

Veja que temos `console.log(e.target.error)`, muito importante para que o desenvolvedor esteja escrutinando o console identifique a causa do erro. Recarregando a página, faremos um teste. No console, vamos executar a seguinte instrução que cria uma conexão e o nosso DAO, e adiciona uma nova negociação:

// EXECUTAR NO CONSOLE DO NAVEGADOR

```
const negociacao = new Negociacao(new Date(), 7, 100);
```

```

ConnectionFactory
  .getConnection()
  .then(conn => new NegociacaoDao(conn))
  .then(dao => dao.adiciona(negociacao))
  .then(() => console.log('Negociação salva com sucesso!'))
  .catch(err => console.log(err));

```

Veremos que ele gravou perfeitamente as negociações.

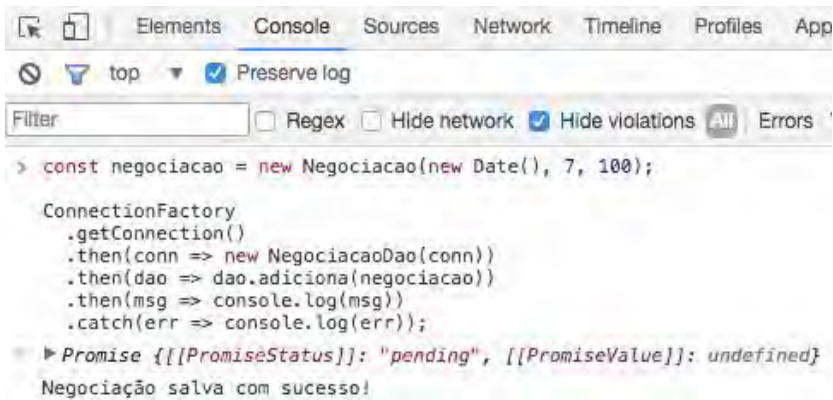


Figura 16.1: Sucesso nas gravação

Agora, vamos entender como se deu o encadeamento das funções. Primeiro, através da Promise retornada por `ConnectionFactory.getConnection()`, temos acesso à conexão encadeando uma chamada à função `then()`. Ainda neste `then()`, de posse da conexão, retornamos implicitamente através de uma arrow function uma instância de `NegociacaoDao` que recebe a conexão retornada pela Promise.

Lembre-se de que as funções `then()` podem retornar qualquer valor, seja este síncrono ou o resultado da chamada de outra Promise. Dessa forma, na próxima chamada a `then()`, temos acesso à instância de `NegociacaoDao`, cujo o método `adiciona()` chamamos ao passar a negociação que desejamos gravar. Como o método `adiciona()` é uma Promise, a próxima chamada encadeada à `then()` só será executada se a operação for realizada com sucesso.

Por fim, temos a chamada da função `catch()` que pode capturar qualquer erro que possa ocorrer na obtenção da conexão

ou da operação de persistência. Faltava ainda implementarmos o método `listaTodos()`, aquele que interage com o cursor do IndexedDB.

16.4 IMPLEMENTANDO A LÓGICA DA LISTAGEM

Vamos implementar o método `listaTodos()` de `NegociacaoDao`. Ainda não teremos nenhuma novidade, então aplicaremos nosso conhecimento já adquirido:

```
// client/app/domain/negociacao/NegociacaoDao.js
// código anterior omitido

listaTodos() {

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

    const negociacoes = [];

    const cursor = this._connection
      .transaction([this._store], 'readwrite')
      .objectStore(this._store)
      .openCursor();

    cursor.onsuccess = e => {

      const atual = e.target.result;

      if(atual) {

        const negociacao = new Negociacao(
          atual.value._data,
          atual.value._quantidade,
          atual.value._valor);

        negociacoes.push(negociacao);
        atual.continue();

      } else {
```

```

        // resolvendo a promise com negociacoes
        resolve(negociacoes);
    }
};

cursor.onerror = e => {
    console.log(e.target.error);
    reject('Não foi possível listar nas negociações');
}

});
}
// código posterior omitido

```

Precisamos testar nosso método, e faremos isso mais uma vez através do console com a seguinte instrução:

```

// NO CONSOLE

ConnectionFactory
  .getConnection()
  .then(conn => new NegociacaoDao(conn))
  .then(dao => dao.listaTodos())
  .then(negociacoes => console.log(negociacoes))
  .catch(err => console.log(err));

```

Será exibido no console um array com todas as negociações que persistimos até o momento. Temos os dois métodos do DAO que criamos. Porém, toda vez que precisarmos de um DAO, o programador terá de obter uma conexão através de `ConnectionFactory`, para em seguida obter uma instância do DAO. Podemos simplificar esse processo isolando a complexidade de criação do DAO em uma nova classe.

16.5 CRIANDO UMA DAOFACTORY

Criaremos a classe `DaoFactory` que isolará a criação de um

DAO para nós. Vamos criar o arquivo `client/util/DaoFactory.js` e declarar nossa classe que possuirá apenas o método estático `getNegociacaoDao()` :

```
// client/app/util/DaoFactory.js

class DaoFactory {

    static getNegociacaoDao() {

        return ConnectionFactory
            .getConnection()
            .then(conn => new NegociacaoDao(conn));

    }

}
```

O método `getNegociacaoDao()` retorna uma Promise que, ao ser resolvida, nos dá acesso a uma instância de `NegociacaoDao` .

Vamos importar a classe em `client/index.html` , para logo em seguida realizarmos um teste:

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

<script src="app/domain/negociacao/Negociacao.js"></script>
<script src="app/controllers/NegociacaoController.js"></script>
<script src="app/utils/DateConverter.js"></script>
<script src="app/domain/negociacao/Negociacoes.js"></script>
<script src="app/ui/views/View.js"></script>
<script src="app/ui/views/NegociacoesView.js"></script>
<script src="app/ui/models/Mensagem.js"></script>
<script src="app/ui/views/MensagemView.js"></script>
<script src="app/util/ProxyFactory.js"></script>
<script src="app/util/Bind.js"></script>
<script src="app/util/ApplicationException.js"></script>
<script src="app/ui/converters/DataInvalidaException.js"></script>

<script src="app/domain/negociacao/NegociacaoService.js"></script>
```

```

<script src="app/util/HttpService.js"></script>
<script src="app/util/ConnectionFactory.js"></script>
<script src="app/domain/negociacao/NegociacaoDao.js"></script>

<!-- importou aqui -->
<script src="app/util/DaoFactory.js"></script>

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

<!-- código posterior omitido -->

```

Após o recarregamento da página, vamos executar a seguinte instrução no console do navegador:

```
// NO CONSOLE
```

```
DaoFactory.getNegociacaoDao().then(dao => console.log(dao))
```

No console, será exibida a instância de `NegociacaoDao` :

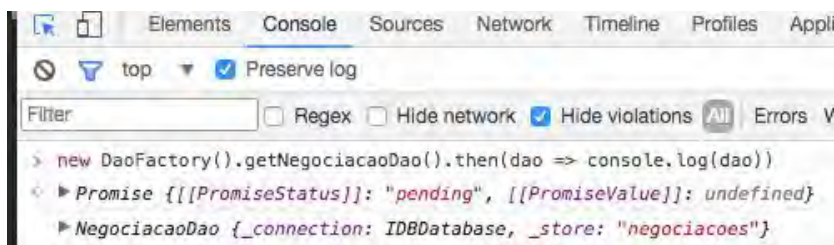


Figura 16.2: DaoFactory em ação

Agora que já temos toda infraestrutura pronta, chegou a hora de realizarmos a persistência das negociações que cadastrarmos.

16.6 COMBINANDO PADRÕES DE PROJETO

Na classe `NegociacaoController` , precisamos alterar o método `adiciona()` para que ele persista a negociação do formulário. Apenas se a operação for bem-sucedida,

adicionaremos a negociação na tabela.

Vamos alterar o bloco `try` que fará uso da classe `DaoFactory`:

```
// client/app/controllers/NegociacaoController.js

// código anterior omitido

adiciona(event) {

    try {

        event.preventDefault();

        // negociação que precisamos incluir no banco e na tabela

        const negociacao = this._criaNegociacao();

        DaoFactory
            .getNegociacaoDao()
            .then(dao => dao.adiciona(negociacao))
            .then(() => {

                // só tentará incluir na tabela se conseguiu antes
                // incluir no banco

                this._negociacoes.adiciona(negociacao);
                this._mensagem.texto = 'Negociação adicionada com
sucesso';

                this._limpaFormulario();
            })
            .catch(err => this._mensagem.texto = err);

    } catch(err) {

        // código omitido
    }
}

// código posterior omitido
```

Vamos recarregar a página, para depois incluirmos uma nova negociação. Será exibida a mensagem de sucesso, inclusive a negociação será inserida na tabela:

Negociações

Negociação adicionada com sucesso

Data

Quantidade

Valor

DATA	QUANTIDADE	VALOR	VOLUME
12/12/2017	1	200	200
			200

Figura 16.3: Negociação adicionada com sucesso

Para não termos sombra de dúvidas da gravação da negociação, vamos abrir o Console e acessar a aba `Application`, para em seguida visualizarmos o conteúdo da `store negociacoes`:

```
▼ Object
  _data: Tue Dec 12 2017 00:00:00 GMT-0200 (-02)
  _quantidade: "1"
  _valor: "200"
```

Figura 16.4: Dados da negociação

Excelente! Agora precisamos exibir todas as negociações salvas no banco.

16.7 EXIBINDO TODAS AS NEGOCIAÇÕES

Em `NegociacaoDao`, já temos o método `listarTodos()` implementado, só precisamos chamá-lo assim que `NegociacaoController` for instanciado. Faremos isso através de seu `constructor()`:

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {

  constructor() {

    // código anterior omitido

    DaoFactory
      .getNegociacaoDao()
      .then(dao => dao.listarTodos())
      .then(negociacoes =>
        negociacoes.forEach(negociacao =>
          this._negociacoes.adiciona(negociacao)))
      .catch(err => this._mensagem.texto = err);

  }

  // código posterior omitido
}
```

Adicionamos cada negociação trazida do banco na lista de negociações que alimenta nossa tabela. Dessa forma, a tabela já será exibida logo de início para o usuário com as negociações que ele já cadastrou.



DATA	QUANTIDADE	VALOR	VOLUME
16/6/2016	1	200	200
23/6/2016	1	200	200
26/6/2016	7	100	700
11/12/2012	13	171	2223
11/12/2012	44	2	88

Figura 16.5: Banco com todas as informações

Apesar de funcionar, é uma boa prática isolarmos em um método todo o código de inicialização de uma classe que não diga respeito à inicialização das propriedades da classe. Vamos criar o método `_init()` para então movermos o código que busca todas as negociações para dentro dele:

```
class NegociacaoController {

    constructor() {

        // código anterior omitido

        // chama o método de inicialização
        this._init();
    }

    _init() {

        DaoFactory
            .getNegociacaoDao()
            .then(dao => dao.listaTodos())
            .then(negociacoes =>
                negociacoes.forEach(negociacao =>
                    this._negociacoes.adiciona(negociacao)))
            .catch(err => this._mensagem.texto = err);
    }

    // código posterior omitido
}
```

```
}
```

Ainda falta uma operação de persistência, aquela que permite excluir negociações.

16.8 REMOVENDO TODAS AS NEGOCIAÇÕES

Nosso `NegociacaoDao` ainda não possui um método que apague todas as negociações cadastradas, e chegou a hora de criá-lo. Ele será praticamente idêntico ao método `adiciona`, a diferença é que não receberá nenhum parâmetro e que chamaremos a função `clear()` em vez de `add()`. Adicionando o novo método `apagaTodos()`:

```
// client/app/domain/negociacao/NegociacaoDao.js
// código anterior omitido

apagaTodos() {

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

        const request = this._connection
            .transaction([this._store], 'readwrite')
            .objectStore(this._store)
            .clear();

        request.onsuccess = e => resolve();

        request.onerror = e => {
            console.log(e.target.error);
            reject('Não foi possível apagar as negociações');
        };

    });
}
// código posterior omitido
```

Por fim, precisamos alterar o método `apaga()` de

NegociacaoController para que apague os dados do banco. Aliás, a tabela só será esvaziada se a operação de persistência der certo:

```
// client/app/controllers/NegociacaoController.js
// código anterior omitido

apaga() {

    DaoFactory
        .getNegociacaoDao()
        .then(dao => dao.apagaTodos())
        .then(() => {
            this._negociacoes.esvazia();
            this._mensagem.texto = 'Negociações apagadas com suce
sso';
        })
        .catch(err => this._mensagem.texto = err);
}
// código posterior omitido
```

Já podemos recarregar nossa página e testarmos a nova funcionalidade clicando no botão "Apagar". O resultando será a exclusão total das negociações do nosso banco e a limpeza da tabela de negociações:

Negociações

Negociações removidas com sucesso

Data

dd/mm/aaaa

Quantidade

1

Valor

0,0

Incluir

Importar Negociações

Apagar

Figura 16.6: Negociações exibidas com sucesso

Agora, quando recarregarmos a página, a tabela de negociações estará vazia.

Negociações

Data

dd/mm/aaaa

Quantidade

1

Valor

0,0

Incluir

Importar Negociações

Apagar

DATA	QUANTIDADE	VALOR	VOLUME
			0

Figura 16.7: Negociações totalmente apagadas

16.9 FACTORY FUNCTION

O código que escrevemos funciona como esperado, criamos a

classe `DaoFactory` com o método estático `getNegociacaoDao()` que esconde a complexidade da criação de instâncias da classe `NegociacaoDao`. Porém, vale a pena uma reflexão.

Linguagens presas exclusivamente ao paradigma da Orientação a Objeto não permitem criar métodos sem que pertençam a uma classe. Não são raras as classes com apenas um método, na maioria das vezes estático para implementar o padrão de projeto Factory. No entanto, como JavaScript é multiparadigma, podemos escrever menos se reimplementarmos o padrão Factory através do paradigma Funcional.

Vamos alterar o arquivo `client/util/DaoFactory.js`. Ele declarará apenas a função `getNegociacaoDao`:

```
// client/app/util/DaoFactory.js

function getNegociacaoDao() {

    return ConnectionFactory
        .getConnection()
        .then(conn => new NegociacaoDao(conn));
}
```

Não temos mais a declaração da classe e poupamos algumas linhas de código.

Agora, precisamos alterar `client/app/controllers/NegociacaoController.js` para usar a função `getNegociacaoDao()` no lugar de `DaoFactory.getNegociacaoDao()`:

```
// client/app/controllers/NegociacaoController.js

class NegociacaoController {
```

```

constructor() {

    // código omitido
}

_init() {

    // MUDANÇA AQUI

    getNegociacaoDao()
    .then(dao => dao.listaTodos())
    .then(negociacoes =>
        negociacoes.forEach(negociacao =>
            this._negociacoes.adiciona(negociacao)))
    .catch(err => this._mensagem.texto = err);
}

adiciona(event) {

    event.preventDefault();

    try {

        const negociacao = this._criaNegociacao();

        // MUDANÇA AQUI

        getNegociacaoDao()
        .then(dao => dao.adiciona(negociacao))
        .then(() => {
            this._negociacoes.adiciona(negociacao);
            this._mensagem.texto = 'Negociação adicionada com
sucesso';
            this._limpaFormulario();
        })
        .catch(err => this._mensagem.texto = err);

    } catch (err) {

        // código omitido
    }
}

// código omitido

```

```

apaga() {

    // MUDANÇA AQUI

    getNegociacaoDao()
    .then(dao => dao.apagaTodos())
    .then(() => {
        this._negociacoes.esvazia();
        this._mensagem.texto = 'Negociações apagadas com suce
sso';
    })
    .catch(err => this._mensagem.texto = err);
}
}

```

Simplificamos ainda mais nosso código. Em suma, quando precisamos apenas de um método para realizar nosso trabalho, podemos evitar a declaração de classes e utilizar em seu lugar a declaração de uma função. A aplicação prática de uma ou outra forma dependerá também da inclinação do desenvolvedor para um ou outro paradigma.

Excelente, terminamos nossa aplicação! Mas será que ela funcionará nos mais diversos navegadores do mercado?

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/16>

DIVIDIR PARA CONQUISTAR

"Sertão: é dentro da gente." - Grande Sertão: Veredas

Vimos que o JavaScript possui duas características que assombram desenvolvedores nessa linguagem: o escopo global e a ordem de carregamento de scripts. O primeiro problema tentamos minimizar através do *module pattern*, mas ele ainda continua existindo. Já com o segundo, nada pôde ser feito, jogando a responsabilidade de importação nas costas do desenvolvedor. Contudo, a partir do ES2015 (ES6), a própria linguagem JavaScript atacou esses problemas através de um sistema nativo de módulos.

17.1 MÓDULOS DO ES2015 (ES6)

O sistema de módulos do ES2015 (ES6) baseia-se em quatro pilares:

1. Cada script é um módulo por padrão, que esconde suas variáveis e funções do mundo externo.

2. A instrução `export` permite que um módulo exporte os artefatos que deseja compartilhar.
3. A instrução `import` permite que um módulo importe artefatos de outros módulos. Apenas artefatos exportados podem ser importados.
4. O uso de um *loader* (carregador), aquele que será o responsável em carregar o primeiro módulo e resolver todas as suas dependências para nós, sem termos de nos preocupar em importar cada script em uma ordem específica.

Um ponto a destacar dos quatro pilares que vimos é o do loader. Mesmo na versão ES2017 do JavaScript não há consenso na especificação do loader, por isso precisamos apelar para bibliotecas de terceiros, capazes de resolver o carregamento de módulos. A biblioteca que usaremos será a **System.js**.

A biblioteca System.js (<https://github.com/systemjs/systemjs>) é um loader universal com suporte aos formatos de módulo AMD, CommonJS, inclusive do ES2015 (ES6), aquele que temos interesse. Como temos o Node.js instalado como requisito de infraestrutura, podemos baixar a biblioteca através do `npm`, seu gerenciador de módulos.

17.2 INSTALANDO O LOADER

Precisamos preparar o terreno antes de instalarmos o System.js. O primeiro passo é abrirmos nosso terminal favorito e, com a certeza de estarmos dentro da pasta `jscangaceiro/client`, executar o comando:

```
npm init
```

Podemos teclar `Enter` para todas as perguntas sem problema algum e, no final, será gerado o arquivo `package.json`. Em poucas palavras, este arquivo guardará o nome e a versão de todas as dependências que baixarmos pelo `npm`.

O arquivo `jscangaceiro/client/package.json` criado possui a seguinte estrutura:

```
{
  "name": "client",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Agora, ainda dentro da pasta `jscangaceiro/client`, vamos instalar o `System.js` através do comando:

```
npm install systemjs@0.20.12 --save
```

O parâmetro `--save` registra o módulo baixado no arquivo `package.json`. Por fim, após o comando terminar de executar, o `System.js` e todas as suas dependências estarão dentro da pasta `jscangaceiro/client/node_modules` criada automaticamente pelo `npm`.

Agora, sem nos assustarmos, removeremos todos os scripts importados em `client/index.html`. O `System.js` será o responsável pelo carregamento dos nossos scripts que serão convertidos em módulos.

Vamos importar a biblioteca e indicar que o primeiro módulo a ser carregado será o `client/app/app.js` :

```
<!-- client/index.html -->
<!-- código anterior omitido -->
  <div id="negociacoes"></div>

  <!-- importou a biblioteca do System.js -->
  <script src="node_modules/systemjs/dist/system.js"></script>

  <!-- indicando qual será o primeiro módulo a ser carregado -->

  <script>
    System
      .import('app/app.js')
      catch(err console.error(error(err))
  </script>
</body>
</html>
```

É muito importante que estejamos acessando nosso projeto através do servidor disponibilizado com o projeto, porque o System.js baixa os módulos através de requisições assíncronas com XMLHttpRequest .

Acessando nossa aplicação por meio de `localhost:3000` e verificando o console do navegador, recebemos a seguinte mensagem de erro:

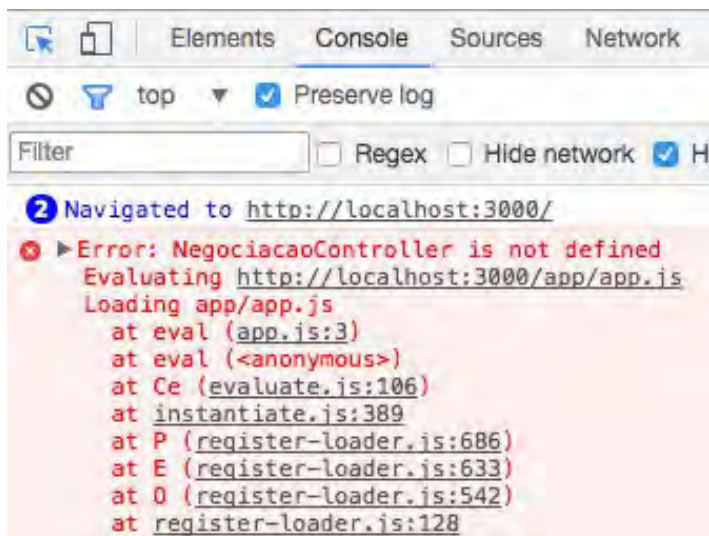


Figura 17.1: Problema no carregamento do módulo

O erro era totalmente esperado, pois ainda precisamos ajustar cada script que criamos tornando-os módulos do ES2015 (ES6).

17.3 TRANSFORMANDO SCRIPTS EM MÓDULOS

Quando adotamos o sistema de módulos do ES2015 (ES6), cada script por padrão será considerado um módulo que confina tudo o que declara. Precisaremos configurar cada módulo indicando o que ele importa e exporta.

Uma técnica que pode ajudar no processo de ajuste é perguntarmos para o módulo quais são suas dependências. Essas dependências precisam ser importadas através da instrução `import`. Depois, precisamos nos perguntar o que o módulo deve

exportar para aqueles que desejam utilizá-lo. Fazemos isso através da instrução `export`.

Analisando o módulo `client/app/app.js`, vemos que ele possui apenas uma dependência, a classe `NegociacaoController` definida no módulo `NegociacaoController.js`. Alteraremos `app.js` e importaremos sua dependência:

```
// client/app/app.js

import { NegociacaoController } from '../controllers/NegociacaoController.js';

const controller = new NegociacaoController();
const $ = document.querySelector.bind(document);

// código posterior omitido
```

Nesse código, estamos importando a classe `NegociacaoController` do módulo `../controller/NegociacaoController.js`. Contudo, isso ainda não funcionará, pois `NegociacaoController.js` não exporta a classe `NegociacaoController`. Precisamos alterá-lo:

```
export class NegociacaoController {

  // código omitido
}
```

A questão agora é que nosso `NegociacaoController` depende de outras classes. Vamos importar todas elas:

```
import { Negociacoes } from '../domain/negociacao/Negociacoes.js';
import { NegociacoesView } from '../ui/views/NegociacoesView.js';
import { Mensagem } from '../ui/models/Mensagem.js';
import { MensagemView } from '../ui/views/MensagemView.js';
import { NegociacaoService } from '../domain/negociacao/NegociacaoService.js';
```

```
import { getNegociacaoDao } from '../util/DaoFactory.js';
import { DataInvalidaException } from '../ui/converters/DataInvalidaException.js';
import { Negociacao } from '../domain/negociacao/Negociacao.js';
import { Bind } from '../util/Bind.js';
import { DateConverter } from '../ui/converters/DateConverter.js';
;

export class NegociacaoController {
  /* código omitido */
}
```

Não podemos deixar de importar nenhuma dependência; caso contrário, ela não será visível para nosso módulo. Agora, precisamos ajustar cada um dos módulos importados:

Negociacoes.js :

```
export class Negociacoes {
  /* código omitido */
}
```

NegociacoesView.js :

```
import { View } from './View.js';
import { DateConverter } from '../converters/DateConverter.js';

export class NegociacoesView extends View {
  /* código omitido */
}
```

A classe `NegociacoesView` precisou importar `View` e `DateConverter`, pois são dependências da classe. Vamos continuar:

Mensagem.js :

```
export class Mensagem {
  /* código omitido */
}
```

MensagemView.js :

```
import { View } from './View.js';

export class MensagemView extends View {
  /* código omitido */
}
```

NegociacaoService.js :

```
import { HttpService } from '../../util/HttpService.js';
import { Negociacao } from './Negociacao.js';

export class NegociacaoService {
  /* código omitido */
}
```

DaoFactory.js :

```
import { ConnectionFactory } from './ConnectionFactory.js';
import { NegociacaoDao } from '../domain/negociacao/NegociacaoDao.js';

export function getNegociacaoDao {
  /* código omitido */
}
```

DataInvalidaException.js :

```
import { ApplicationException } from '../../util/ApplicationException.js';

export class DataInvalidaException extends ApplicationException {
  /* código omitido */
}
```

Negociacao.js :

```
export class Negociacao {
  /* código omitido */
}
```

```
}
```

NegociacaoDao.js :

```
import { Negociacao } from './Negociacao.js';

export class NegociacaoDao {
  /* código omitido */
}
```

DateConverter.js :

```
import { DataInvalidaException } from './DataInvalidaException.js';

export class DateConverter {
  /* código omitido */
}
```

View.js :

```
export class View {
  /* código omitido */
}
```

Bind.js :

```
import { ProxyFactory } from './ProxyFactory.js';

export class Bind {
  /* código omitido */
}
```

Para a classe `ConnectionFactory` , podemos remover a IIFE, pois como estamos usando o sistema de módulo nativo do ES2015 (ES6), variáveis e funções do módulo automaticamente não serão acessíveis fora dele. No caso, basta exportarmos a classe `ConnectionFactory` para que ela seja acessível.

`ConnectionFactory.js` :

```
const stores = ['negociacoes'];
let connection = null;
let close = null;

export class ConnectionFactory {
  /* código omitido */
}
```

ApplicationException.js :

```
export class ApplicationException extends Error {
  /* código omitido */
}
```

HttpService.js :

```
export class HttpService {
  /* código omitido */
}
```

ProxyFactory.js :

```
export class ProxyFactory {
  /* código omitido */
}
```

Todos os módulos foram configurados. Entretanto, se recarregarmos a aplicação e olharmos no console do navegador, veremos a seguinte mensagem de erro:

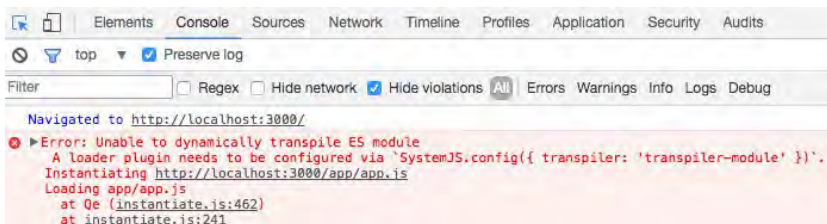


Figura 17.2: Sucesso na gravação

O System.js não conseguiu importar nossos módulos, apesar

deles estarem no formato ES2015 (ES6). É necessária uma configuração extra para que sejam importados pelo nosso loader. Essa configuração extra pode ser realizada através de um **transpiler**, como indicado na mensagem de erro. Mas o que é um transcompilador?

17.4 O PAPEL DE UM TRANSCOMPILADOR

Um transcompilador (*transpiler*) é um compilador que permite realizar transformações em nosso código, adicionando código extra ou até mesmo traduzindo o código-fonte de uma linguagem para outra.

Esse processo pode ser feito diretamente no navegador, mas é altamente desencorajado em produção, devido à quantidade extra de scripts que serão carregados durante o processo. Assim, onera o tempo de carregamento da página e, por conseguinte, impacta no ranking de pesquisa orgânica do Google. Páginas que são carregadas mais rapidamente são favorecidas nesse ranking.

Precisamos de um transcompilador que rode localmente, em tempo de desenvolvimento, e que gere os arquivos modificados que serão carregados pelo navegador. Sem dúvidas, o **Babel** (<https://babeljs.io>) é o transcompilador mais utilizado pela comunidade open source.

Sua estrutura baseada em plugins possibilitou a criação de diversas funcionalidades já prontas para uso. É ele que utilizaremos. Porém, antes de instalarmos e configurarmos o Babel, precisamos realizar um pequeno ajuste em nosso projeto.

Vamos **renomear a pasta** `client/app` para `client/app-`

`src` . O sufixo `src` indica que essa pasta armazena os arquivos originais do projeto.

Com a pasta renomeada, podemos passar para a próxima seção, para instalar o Babel.

17.5 BABEL, INSTALAÇÃO E BUILD-STEP

Quando usamos Babel, estamos adicionando em nosso projeto um `build step` , ou seja, um passo de construção em nossa aplicação. Isso significa que ela não pode ser consumida diretamente antes de passar por esse processo. Sendo assim, o primeiro passo é instalar o Babel em nosso projeto, para então configurá-lo.

Dentro da pasta `jscangaceiro/client` , instalaremos o `babel-cli` através do `npm` :

```
npm install babel-cli@6.24.1 --save-dev
```

Além do `babel-cli` , precisamos instalar o plugin `babel-plugin-transform-es2015-modules-systemjs` que adequará os módulos do ES2015 ao sistema de carregamentos do `System.js`:

```
npm install babel-plugin-transform-es2015-modules-systemjs@6.24.1  
--save-dev
```

Agora, precisamos explicitar que o plugin que baixamos deve ser usado pelo Babel. Para isso, vamos criar o arquivo `.babelrc` dentro de `jscangaceiro/client` , com a seguinte configuração:

```
{  
  "plugins" : ["transform-es2015-modules-systemjs"]  
}
```


Com tudo instalado, vamos adicionar um script no arquivo `package.json` que executará o Babel para nós. O nome do script será `build` :

```
{
  "name": "client",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "babel app-src -d app --source-maps"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "systemjs": "^0.20.12"
  },
  "devDependencies": {
    "babel-cli": "^6.24.1",
    "babel-plugin-transform-es2015-modules-systemjs": "^6.24.1"
  }
}
```

Vamos analisar o script que adicionamos:

```
"build": "babel app-src -d app --source-maps"
```

Já estamos preparados para testar. Tenha certeza de ter renomeado a pasta `client/app` para `client/app-src` , como foi pedido antes.

Vamos testar o comando `npm run build` :

```
npm run build
```

Após o comando ter sido processado, será criada automaticamente a pasta `client/app` com a mesma estrutura de pastas e arquivos de `client/app-src` . A diferença é que o código está transcompilado.

Vamos abrir o arquivo `client/app/domain/negociacao/Negociacao.js` e ver seu conteúdo:

```
System.register([], function (_export, _context) {
  "use strict";

  return {
    setters: [],
    execute: function () {
      class Negociacao {

        /* código anterior omitido */
      }

      _export("Negociacao", Negociacao);
    }
  };
});
```

É uma sintaxe bem diferente do arquivo original, mas suficiente para que o System.js seja capaz de carregar nossas dependências. Porém, se um erro acontecer em nosso arquivo, veremos a indicação da posição do erro pelo debugger do navegador no arquivo transcompilado. Precisamos saber a posição no arquivo original, pois é nele que precisamos dar manutenção.

A boa notícia é que, ao gerarmos nossos arquivos, o Babel gera também um arquivo `.map` (sourcemap) para cada um deles.

17.6 SOURCEMAP

Um arquivo *sourcemap* serve para ajudar no processo de depuração no próprio browser, indicando o erro na estrutura do arquivo original em vez do arquivo transcompilado. Faz todo sentido, pois é no original que daremos manutenção.

O carregamento de arquivos `.map` pode variar entre navegadores. No Chrome, por exemplo, eles serão carregados apenas quando o console do navegador for acessado. Já outros navegadores necessitam que o carregamento seja explicitado através de alguma opção de desenvolvedor. Agora nada nos impede de realizarmos um teste.

Acessando mais uma vez `http://localhost:3000` e recarregando nossa página, ela continua funcionando como antes. Veja que agora só precisamos indicar qual será o primeiro módulo a ser carregado pelo nosso loader que ele se encarregará de resolver todas as suas dependências, removendo assim essa responsabilidade do desenvolvedor.

Além do que vimos, não conseguimos mais acessar as definições das classes que antes viviam no escopo global através do console. Ou seja, não há mais nenhuma informação da nossa aplicação no escopo global.

Caso algum erro tenha acontecido, é importante verificarmos se exportamos e importamos corretamente todos os artefatos da aplicação, para em seguida executarmos mais uma vez o comando `npm run build`. Aliás, ter de executar toda vez este comando a cada modificação é jogar muita responsabilidade no console do desenvolvedor. Na próxima seção, aprenderemos a automatizar esse processo.

17.7 COMPILANDO ARQUIVOS EM TEMPO REAL

Podemos melhorar a experiência do desenvolvedor realizando

o processo de transcompilação automaticamente toda vez que um arquivo for alterado no projeto. A ação deve ser realizada quando fazemos o deploy, para garantir que está tudo compilado.

Mas será que o desenvolvedor vai querer ter esta responsabilidade a todo momento? Por isso, o Babel vem com o *watcher*, um observador de arquivos que automaticamente fará o processo de transcompilação quando um arquivo for alterado. Para habilitá-lo, vamos no arquivo `package.json` e adicionaremos o `watch` :

```
{
  "name": "client",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "babel app-src -d app --source-maps",
    "watch": "babel app-src -d app --source-maps --watch"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "babel-cli": "^6.24.1",
    "babel-plugin-transform-es2015-modules-systemjs": "^6.24.1"
  },
  "dependencies": {
    "systemjs": "^0.20.12"
  }
}
```

No terminal, vamos executar o `watch` :

```
npm run watch
```

Ele vai compilar os arquivos e o terminal ficará monitorando a modificação de todos eles. O processo de compilação correrá bem e, ao recarregarmos a página de cadastro, tudo funcionará

corretamente.

Demos um passo extra introduzindo um transcompilador em nosso projeto, mas será que podemos ir além? Com certeza!

17.8 BARREL, SIMPLIFICANDO A IMPORTAÇÃO DE MÓDULOS

Aprendemos a utilizar o sistema de módulos do ES2015 (ES6), evitando assim o escopo global e a responsabilidade de carregarmos scripts em uma ordem correta. Todavia, podemos organizar ainda melhor nossas importações.

Vejamos primeiro como fizemos as importações em `NegociacaoController`:

```
// client/app-src/controllers/NegociacaoController.js

// APENAS REVISÃO DE CÓDIGO

import { Negociacoes } from '../domain/negociacao/Negociacoes.js'
;
import { NegociacoesView } from '../ui/views/NegociacoesView.js';
import { Mensagem } from '../ui/models/Mensagem.js';
import { MensagemView } from '../ui/views/MensagemView.js';
import { NegociacaoService } from '../domain/negociacao/NegociacaoService.js';
import { getNegociacaoDao } from '../util/DaoFactory.js';
import { DataInvalidaException } from '../ui/converters/DataInvalidaException.js';
import { Negociacao } from '../domain/negociacao/Negociacao.js';
import { Bind } from '../util/Bind.js';
import { DateConverter } from '../ui/converters/DateConverter.js'
;

export class NegociacaoController {

// código posterior omitido
```

Repare que importamos vários artefatos de uma mesma pasta, aliás, repetindo a instrução `import` diversas vezes. Podemos simplificar bastante essa importação através de **barrels**. Um barrel nada mais é do que um módulo que importa e exporta os módulos que importou. Com isso, podemos importar em uma única instrução vários artefatos exportados pelo barrel. Um exemplo:

```
// EXEMPLO, AINDA NÃO ENTRA NA APLICAÇÃO

import { Negociacoes, NegociacaoService, Negociacao } from '../do
main/index';
```

Vamos começar criando o barrel para a pasta `app-src/models`, através do arquivo `app-src/models/index.ts`:

```
// client/app-src/domain/index.ts

export * from './negociacao/Negociacao.js';
export * from './negociacao/NegociacaoDao.js';
export * from './negociacao/NegociacaoService.js';
export * from './negociacao/Negociacoes.js';
```

Agora, faremos a mesma coisa para `client/app-src/ui`:

```
// client/app-src/ui/index.js

export * from './views/MensagemView.js';
export * from './views/NegociacoesView.js';
export * from './views/View.js';
export * from './models/Mensagem.js';
export * from './converters/DataInvalidaException.js';
export * from './converters/DateConverter.js';
```

Por fim, faremos em `client/app-src/utills`:

```
// client/app-src/utill/index.js

export * from './Bind.js';
export * from './ConnectionFactory.js';
export * from './DaoFactory.js';
export * from './ApplicationException.js';
```

```
export * from './HttpService.js';  
export * from './ProxyFactory.js';
```

E então, alteraremos `NegociacaoController` :

```
import { Negociacoes, NegociacaoService, Negociacao } from '../dom  
ain/index.js';  
import { NegociacoesView, MensagemView, Mensagem, DataInvalidaExc  
eption, DateConverter } from '../ui/index.js';  
import { getNegociacaoDao, Bind } from '../util/index.js';  
  
export class NegociacaoController {  
  
    // código posterior omitido
```

Muito menos linhas de código!

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/17>

INDO ALÉM

"Sertão é isto: o senhor empurra para trás, mas de repente ele volta a rodear o senhor dos lados. Sertão é quando menos se espera." - Grande Sertão: Veredas

No capítulo anterior, com a utilização de módulos do ES2015 (ES6), resolvemos tanto o problema do escopo global quanto da ordem de importação de scripts. Ainda há uma melhoria que podemos realizar em nosso código, mas primeiro, vamos relembrar da importância das Promises nele.

Utilizamos o padrão de projeto *Promise* para evitarmos o callback HELL, além de centralizarmos em um único lugar o tratamento de erro das Promises envolvidas na operação. Vejamos o código que implementamos no método `_init()` de `NegociacaoController`:

```
// REVISANDO APENAS!  
  
// client/app-src/controllers/NegociacaoController.js  
// código anterior omitido  
  
_init() {
```



```

    .getNegociacaoDao()
    .then(dao => dao.listaTodos())
    .then(negociacoes =>
        negociacoes.forEach(negociacao =>
            this._negociacoes.adiciona(negociacao)))
    .catch(err => this._mensagem.texto = err); // TRATA O ERRO EM
    UM LUGAR APENAS
}

// código posterior omitido

```

Contudo, mesmo a solução com Promises não torna nosso código tão legível quanto a escrita de um código **síncrono e bloqueante**. Vejamos um código hipotético:

```

// EXEMPLO DE COMO SERIA SE O CÓDIGO
// FOSSE ESCRITO DE MANEIRA SÍNCRONA E BLOQUEANTE

_init() {

    try {

        // bloqueará a execução da aplicação enquanto não terminar
        const dao = getNegociacaoDao();

        // bloqueará a execução da aplicação enquanto não terminar
        const negociacoes = dao.listaTodos();

        negociacoes.forEach(negociacao => this._negociacoes.adiciona(
            negociacao))
        } catch(err) {

            this._mensagem.texto = err;
        }
    }
}

```

Neste exemplo, a função `getNegociacaoDao()` e o método `dao.listaTodos()` são síncronos e bloqueantes. Por serem síncronos, podemos capturar exceções através das cláusulas `try` e `catch`, pois o erro acontece na mesma pilha de execução do nosso programa.

Porém, para que o código anterior não afete a performance da aplicação, cada chamada do método `_init()` deve ser executada em uma `thread` em separado, para não bloquear o restante da aplicação. Linguagens como Java e C# dão suporte a `multithreading`, resolvendo o problema do bloqueio da aplicação.

O JavaScript é *single thread* (apesar de hoje podermos criar `threads` leves com `WebWorkers`, que não são tão flexíveis quanto `threads` em outras linguagens), por isso não podemos executar operações de I/O de maneira bloqueante. Isso porque, se o tempo de execução do método `listaTodos()` demorasse três segundos, a aplicação ficaria congelada durante esse tempo até voltar a responder e processar outras operações executadas pelo usuário.

Contudo, por mais estranho que isso possa soar, a partir do ES2017 (ES8) já podemos escrever um código assíncrono não bloqueante, muito parecido com a maneira que escrevemos um código síncrono bloqueante, impactando diretamente na legibilidade do código da nossa aplicação. Vejamos como na próxima seção.

18.1 ES2017 (ES8) E O AÇÚCAR SINTÁTICO `ASYNC/AWAIT`

A versão ECMAScript 2017 introduziu a sintaxe `async/await` que permite escrevermos códigos assíncronos envolvendo `Promises` de maneira elegante. Vejamos mais uma vez o método `_init()`, mas dessa vez escrito com a nova sintaxe:

```
// APENAS UM EXEMPLO
```

```
async _init() {
```

```

try {
  const dao = await getNegociacaoDao();
  const negociacoes = await dao.listaTodos();
  negociacoes.forEach(negociacao => this._negociacoes.adiciona(
negociacao)))
} catch(err) {
  this._mensagem.texto = err;
}
}

```

Vejam que a estrutura do nosso código é extremamente parecida com a estrutura hipotética síncrona que vimos. A diferença está em dois pontos. O primeiro é a instrução `async` usada antes do nome do método `_init()`. Essa instrução indica que a função está preparada para executar operações assíncronas que podem ser **esperadas**.

O segundo ponto é a presença da instrução `await` antes das chamadas de métodos que devolvem `Promises`. Quando `getNegociacaoDao()` for chamado, o bloco do método `_init()` será suspenso, saindo da pilha de execução principal da aplicação.

Se o nosso método não faz mais parte da pilha de execução da thread principal, o JavaScript continuará a executar outras operações. Mas e agora? Nosso método `_init()` caiu no limbo? Sim, mas até que a Promise retornada por `getNegociacaoDao()` seja resolvida.

Quando a Promise, que já é uma operação assíncrona e não bloqueante, for resolvida, ela internamente fará um *resume* do nosso método `_init()`. Isso o fará voltar para a pilha de execução principal e ficará disponível para ser processado pelo *Event Loop* do JavaScript.

Quando nosso método `_init()` volta a ser executado, recebemos como retorno o valor já resolvido da `Promise` da instrução `getNegociacaoDao()`. Que loucura!

Esse processo se repetirá na próxima instrução com a sintaxe `await` antes da chamada do método `dao.listaTodos()`, que também devolve uma `Promise`. Por fim, podemos tratar qualquer rejeição das `Promises` dentro do bloco `try` e `catch`, algo que não era possível antes. Isso é possível porque, quando houver uma rejeição de uma `Promise`, também haverá o *resume* do método `_init()` e a causa da rejeição será lançada como uma exceção, permitindo identificar com clareza o ponto em que o erro aconteceu em nosso código.

18.2 PARA SABER MAIS: GENERATORS

A sintaxe `async/await` na verdade é um açúcar sintático para o uso concomitante de `Promises` com funções geradoras (*generators*).

Generators são funções que permitem suspender e resumir sua execução em qualquer ponto do seu bloco, mantendo seu contexto original. Outra característica é que podem retornar um valor mesmo sem o uso da instrução `return`. Vamos construir um simples exemplo:

```
// EXEMPLO DE FUNÇÃO GERADORA

function* minhaFuncaoGeradora () {

}

// it é um ITERATOR
let it = minhaFuncaoGeradora();
```

Generators possuem uma declaração especial, e possuem a palavra-chave `function` seguida de um `*` (asterisco). Quando executamos `minhaFuncaoGeradora()`, recebemos como retorno não o seu resultado, mas um **Iterator**, um objeto especial que nos permite resumir o generator e, eventualmente, extrair valores.

A instrução `yield` é aquela responsável em suspender a execução do bloco do gerador e também, quando especificado, retornar um valor que pode ser acessado através do Iterator. É necessário que o valor venha à direita da instrução:

```
// EXEMPLO DE FUNÇÃO GERADORA

function* minhaFuncaoGeradora () {

    for(let i = 0; i < 3; i++) {

        // INSTRUÇÃO YIELD!
        // SUSPENDE A EXECUÇÃO DO BLOCO DA FUNÇÃO
        // E RETORNA O VALOR DE I

        yield i;
    }
}

let it = minhaFuncaoGeradora();
```

Quando executarmos uma função geradora, seu bloco ainda não será executado. É por isso que obrigatoriamente precisamos chamar pelo menos uma vez o método `it.next()` do Iterator retornado. Com a chamada `it.next()`, o bloco da função geradora será executado até que se depare com a instrução `yield` que vai suspendê-lo.

```
// EXEMPLO DE FUNÇÃO GERADORA

function * minhaFuncaoGeradora () {
```

```

    for(let i = 0; i < 3; i++) {

        yield i;
    }
}

let it = minhaFuncaoGeradora();

// EXECUTA O BLOCO DO GENERATOR, QUE SERÁ SUSPENSO
// ASSIM QUE ENCONTRAR A INSTRUÇÃO YIELD

let objeto = it.next();

```

A função `it.next()` nos devolve um objeto que possui as propriedades `value` e `done`. O primeiro nos dá acesso a qualquer valor retornado pela instrução `yield`, já o segundo indica através de `true` ou `false` se o bloco da função do generator chegou ao fim.

```

// EXEMPLO DE FUNÇÃO GERADORA

function * minhaFuncaoGeradora () {

    for(let i = 0; i < 3; i++) {

        // ao suspender, fornece ao iterator o valor de i
        yield i;
    }
}

let it = minhaFuncaoGeradora();
let objeto = it.next();
console.log(objeto.value) // 0 é o valor de i
console.log(objeto.done) // false, ainda não terminou

```

Podemos chamar várias vezes `it.next()` para resumirmos nossa função:

```

// EXEMPLO DE FUNÇÃO GERADORA

function * minhaFuncaoGeradora () {

```

```

    for(let i = 0; i < 3; i++) {

        yield i;
    }
}

let it = minhaFuncaoGeradora();

// EXECUTA O BLOCO DO GENERATOR, QUE SERÁ SUSPENSO
// ASSIM QUE ENCONTRAR A INSTRUÇÃO YIELD

let objeto = it.next();
console.log(objeto.value) // 0
console.log(objeto.done) // false
objeto = it.next();
console.log(objeto.value) // 1
console.log(objeto.done) // false
objeto = it.next();
console.log(objeto.value) // 2
console.log(objeto.done) // false
objeto = it.next();
console.log(objeto.value) // undefined
console.log(objeto.done) // true

```

Por fim, podemos simplificar nosso código desta forma:

```

// EXEMPLO DE FUNÇÃO GERADORA

function * minhaFuncaoGeradora () {

    for(let i = 0; i < 3; i++) {

        yield i;
    }
}

let it = minhaFuncaoGeradora();

let objeto = null;

while(!(objeto = it.next()).done) {
    console.log(objeto.value);
}

```

O mais importante para nós aqui é entendermos o papel simplificador de `async/await` no uso de Generators e Promises.

18.3 REFATORANDO O PROJETO COM ASYNC/WAIT

Chegou a hora de usarmos `async/await` em nosso projeto, pois já vimos como ele funciona e suas vantagens. Vamos começar pelo método que usamos como exemplo hipotético, mas que agora modificaremos efetivamente.

Vamos modificar o método `_init()` de `NegociacaoController` para que faça uso da instrução `async/await`:

```
// client/app-src/controllers/NegociacaoController.js
// código anterior omitido

async _init() {

    try {
        const dao = await getNegociacaoDao();
        const negociacoes = await dao.listaTodos();
        negociacoes.forEach(negociacao => this._negociacoes.adiciona(negociacao));
    } catch(err) {

        // err.message extrai apenas a mensagem de erro da exceção

        this._mensagem.texto = err.message;
    }
}

// código posterior omitido
```

Lembre-se de que só podemos utilizar a sintaxe `await` para uma Promise dentro de uma função `async`; caso contrário,

receberemos a mensagem de erro *"await is a reserved word"*.

Agora, vamos alterar o método `adiciona()` :

```
// client/app-src/controllers/NegociacaoController.js
// código anterior omitido

async adiciona(event) {

  try {

    event.preventDefault();
    const negociacao = this._criaNegociacao();

    const dao = await getNegociacaoDao();
    await dao.adiciona(negociacao);
    this._negociacoes.adiciona(negociacao);
    this._mensagem.texto = 'Negociação adicionada com sucesso';
  }

  this._limpaFormulario();

} catch(err) {

  this._mensagem.texto = err.message;
}

// código posterior omitido
```

O método `apaga()` também:

```
// client/app-src/controllers/NegociacaoController.js
// código anterior omitido

async apaga() {

  try {

    const dao = await getNegociacaoDao();
    await dao.apagaTodos();
    this._negociacoes.esvazia();
    this._mensagem.texto = 'Negociações apagadas com sucesso';
  }

  catch(err) {
```

```

        this._mensagem.texto = err.message;
    }
}

```

// código posterior omitido

Antes de alterarmos o método `importaNegociacoes()`, alteraremos `obtemNegociacoesDoPeriodo` de `NegociacaoService` para que faça uso do `async/await`. O método ficará assim:

// client/app-src/domain/negociacao/NegociacaoService.js

// código anterior omitido

```

async obterNegociacoesDoPeriodo() {
    try {
        let periodo = await Promise.all([
            this.obtemNegociacoesDaSemana(),
            this.obtemNegociacoesDaSemanaAnterior(),
            this.obtemNegociacoesDaSemanaRetrasada()
        ]);
        return periodo
            .reduce((novoArray, item) => novoArray.concat(item),
                [])
            .sort((a, b) => b.data.getTime() - a.data.getTime());
    } catch(err) {
        console.log(err);
        throw new Error('Não foi possível obter as negociações do período');
    };
}

```

Por fim, vamos voltar para a classe `NegociacaoController` e adequar o método `importaNegociacoes()`:

// client/app-src/controllers/NegociacaoController.js
 // código anterior omitido

```

async importaNegociacoes() {

  try {
    const negociacoes = await this._service.obtemNegociacoesD
oPeriodo();
    console.log(negociacoes);
    negociacoes.filter(novaNegociacao =>

      !this._negociacoes.toArray().some(negociacaoExisten
te =>
        novaNegociacao.equals(negociacaoExistente)))

    .forEach(negociacao => this._negociacoes.adiciona(negocia
cao));
    this._mensagem.texto = 'Negociações do período importadas
com sucesso';

  } catch(err) {

    this._mensagem.texto = err.message;
  }
}

```

E para completar, vamos alterar `DaoFactory.js` :

```

// client/app-src/util/DaoFactory.js
import { ConnectionFactory } from './ConnectionFactory.js';
import { NegociacaoDao } from '../domain/negociacao/NegociacaoDao
.js';

export async function getNegociacaoDao() {

  let conn = await ConnectionFactory.getConnection();
  return new NegociacaoDao(conn);
}

```

Caso tenhamos cometido algum erro, o próprio compilador do Babel nos indicará onde erramos. Basta olharmos a mensagem de erro apresentada no console.

A sintaxe `async/await` , como vimos, é um recurso do ES2017 (ES8). É suportado a partir do Chrome 55, tanto para

desktop quanto para Android, do Opera 42, do Firefox 52 e do Edge 15. Todavia, com o auxílio do Babel, podemos transcompilar nosso código para ES2015 (ES6), garantindo assim uma maior folga em termos de compatibilidade entre navegadores.

18.4 GARANTINDO A COMPATIBILIDADE COM ES2015

Para que possamos transcompilar nosso código de ES2017 (ES8) para ES2015 (ES6), precisamos instalar um *preset* do Babel com esta finalidade.

Dentro da pasta `jscangaceiro/client`, vamos executar o comando:

```
npm install babel-preset-es2017@6.24.1 --save-dev
```

Estamos quase lá! Precisamos agora incluir o preset que acabamos de baixar na lista de presets usados pelo Babel durante o processo de transcompilação. Alterando `jscangaceiro/.babelrc`:

```
{
  "presets":["es2017"],
  "plugins" : ["transform-es2015-modules-systemjs"]
}
```

Quando executarmos o comando `npm run watch` ou `npm run build` nosso preset entrará em ação e realizará a conversão de `async/await` para o uso de `Promise`.

Por exemplo, se pegarmos uma parte do código de `DaoFactory.js` antes e depois da compilação, vemos uma mudança clara em sua estrutura:

Antes:

```
// client/app/util/DaoFactory.js
// código anterior omitido

async function getNegociacaoDao() {

    let conn = await ConnectionFactory.getConnection();
    return new NegociacaoDao(conn);
}

// código posterior omitido
```

Depois, com o preset:

```
// client/app/util/DaoFactory.js
// código anterior omitido

function getNegociacaoDao() {
    return _asyncToGenerator(function* () {

        let conn = yield ConnectionFactory.getConnection();
        return new NegociacaoDao(conn);
    })();
}

// código posterior omitido
```

É bom prevenir de todos os lados, como um verdadeiro cangaceiro faz!

18.5 LIDANDO MELHOR COM EXCEÇÕES

Agora que temos uma forma unificada de lidar com exceções através de `try/catch`, podemos convencionar que toda exceção do tipo `ApplicationException` é de negócio e será exibida diretamente para o usuário. As demais exceções serão logadas, para que o desenvolvedor investigue sua causa, e uma mensagem

genérica será exibida em seu lugar.

Vamos criarmos uma função que saiba identificar o tipo da exceção. Faremos isso no próprio módulo `client/app/util/ApplicationException.js`:

```
// client/app-src/util/ApplicationException.js

export class ApplicationException extends Error {

  constructor(msg = '') {

    super(msg);
    this.name = this.constructor.name;
  }
}

// hack do System.js para que a função tenha acesso à definição d
a classe

const exception = ApplicationException;

export function isApplicationException(err) {

  return err instanceof exception ||
    Object.getPrototypeOf(err) instanceof exception
}
```

`Object.getPrototypeOf()` faz parte da especificação ECMAScript que veio substituir a propriedade `__proto__`. Ela retorna o prototype de objeto.

A função que acabamos de criar verifica se o tipo ou o prototype da exceção são instâncias de `ApplicationException`. O teste do prototype é importante, pois, sem ele, `DataInvalidaException` retornaria `false`.

Agora, criaremos mais uma função que retorna a mensagem da exceção ou a mensagem genérica, caso a exceção não seja do tipo `ApplicationException` :

```
// client/app-src/util/ApplicationException.js

// código anterior omitido

// nova função

export function getExceptionMessage(err) {

    if(isApplicationException(err)) {
        return err.message;
    } else {
        console.log(err);
        return 'Não foi possível realizar a operação.';
    }
}
```

Excelente. Agora, em `client/app-src/controllers/NegociacaoController.js` , importaremos a função `getExceptionMessage` :

```
// client/app-src/controllers/NegociacaoController.js

import { Negociacoes, NegociacaoService, Negociacao } from '../domain/index.js';

// removeu DataInvalidaException

import { NegociacoesView, MensagemView, Mensagem, DateConverter }
    from '../ui/index.js';

// importou getExceptionMessage

import { getNegociacaoDao, Bind, getExceptionMessage } from '../util/index.js';
```

Com a função importada, vamos alterar todos os blocos `catch` para:

```
// todos os blocos catch devem ficar assim

} catch(err) {

    this._mensagem.texto = getExceptionMessage(err);
}
```

Queremos que nossa classe `NegociacaoService` também lance exceções de negócio em vez de `Error`, para que a mensagem de alto nível seja exibida para o usuário. Então, vamos alterar `client/app-src/domain/negociacao/NegociacaoService.js`. Primeiro, importaremos `ApplicationException`:

```
// client/app-src/domain/negociacao/NegociacaoService.js

import { HttpService } from '../util/HttpService.js';
import { Negociacao } from './Negociacao.js';
import { ApplicationException } from '../util/ApplicationException.js';
```

Agora, vamos substituir todas as instruções `new Error(/* mensagem */) por new ApplicationException(/* mensagem */)`.

Com as alterações que fizemos, qualquer exceção lançada por `NegociacaoService` será capturada no bloco `catch` em `NegociacaoController`, e sua mensagem de alto nível será exibida para o usuário.

18.6 DEBOUNCE PATTERN: CONTROLANDO A ANSIEDADE

Nossa aplicação acessa uma API externa através de requisição Ajax, assíncrona por natureza. Contudo, é uma boa prática evitar

que a API seja "metralhada" por uma sucessão de requisições indevidas. Nunca sabemos se nossos usuários são ansiosos, não?

No caso da nossa aplicação, nada nos impede de clicarmos 100 vezes no botão "Importar Negociações" que realizará 100 requisições para a API, apesar de não permitirmos mais importações duplicadas. Podemos solucionar o problema que acabamos de ver postergando a ação que desejamos executar.

Se, durante uma janela de tempo, a mesma ação for executada, a anterior será cancelada e passará a valer a última. Dessa forma, se o usuário clicar 100 vezes no botão "Importar Negociações" em uma janela de tempo de 1 segundo, apenas o último clique efetivamente executará a ação.

A solução que acabamos de ver é tão corriqueira no universo JavaScript que se tornou um padrão, inclusive ganhou o nome **Debounce**. Vamos implementá-la!

18.7 IMPLEMENTANDO O DEBOUNCE PATTERN

Criaremos o módulo `client/app-src/util/Debounce.ts`. O módulo exportará a função `debounce` que receberá como parâmetro a **função alvo** e a **janela de tempo** em milissegundos. Seu retorno será uma **nova função**, que encapsula a função original que desejamos aplicar Debounce através de um temporizador. É esta nova função que associaremos ao evento `click` do botão responsável pela importação de negociações:

```
// client/app-src/util/Debounce.ts

export function debounce(fn, milissegundos) {
```

```

    return () => {
        // usa um temporizador para chamar fn()
        // depois de tantos milissegundos
        setTimeout(() => fn(), milissegundos);
    }
}

```

Não podemos nos esquecer de exportarmos `Debounce.js` através do barrel `client/app-src/util/index.js`. Alterando o arquivo:

```

// client/app-src/util/index.js

export * from './Bind.js';
export * from './ConnectionFactory.js';
export * from './DaoFactory.js';
export * from './ApplicationException.js';
export * from './HttpService.js';
export * from './ProxyFactory.js';

// adicionou a exportação!
export * from './Debounce.js';

```

Sabemos que nossa função não está pronta, mas vamos importá-la em `client/app-src/app.ts` e utilizá-la para o evento `click` do botão que importa as negociações:

```

// client/app-src/app.ts

import { NegociacaoController } from './controllers/NegociacaoController.js';

// importando a função

import { debounce } from './util/index.js';

const controller = new NegociacaoController();

// código anterior omitido

```

```
$('#botao-importa')
  .addEventListener('click', debounce(() =>
    controller.importaNegociacoes.bind(controller), 1000));
```

Em vez de usarmos `controller.importaNegociacoes.bind(controller)`, vamos passar uma função que chamará `controller.importaNegociacoes()`. Isso nos permitirá colocar algumas informações no console para nos ajudar a entender o comportamento do nosso código.

```
// client/app-src/app.ts

import { NegociacaoController } from './controllers/NegociacaoCon
troller.js';
import { debounce } from './util/index.js';

const controller = new NegociacaoController();

// código anterior omitido

$('#botao-importa')
  .addEventListener('click', debounce(() => {
    console.log('EXECUTOU A OPERAÇÃO DO DEBOUNCE');
    controller.importaNegociacoes();
  }, 1000));
```

Do jeito que está, se clicarmos 100 vezes em menos de um segundo, cada ação será executada após um segundo. O comportamento desejado é que apenas uma ação seja executada.

Agora, só precisamos lidar com o cancelamento das ações na janela de um segundo. Todas as ações que forem executadas com o clique do botão precisam referenciar o mesmo timer para que possam parar o anterior e criar um novo. Para isso, vamos declarar a variável `timer` no escopo de `debounce`:

```
// client/app-src/util/Debounce.ts
```

```
export function debounce(fn, milissegundos) {

  // guarda o ID de um timer, 0 indica que não há nenhum
  let timer = 0;

  return () => {
    setTimeout(() => fn(), milissegundos);
  }
}
```

A função retornada por `debounce` é aquela que será disparada pelo evento `click` do botão. Sendo assim, a primeira coisa que precisamos fazer é cancelar qualquer timer que já esteja em execução, para em seguida definirmos um novo timer:

```
// client/app-src/util/Debounce.ts

export function debounce(fn, milissegundos) {

  let timer = 0;
  return () => {

    // para o último timer definido
    clearTimeout(timer);

    // a variável timer ganha o ID de um novo temporizador
    // afeta a variável no escopo da função debounce
    timer = setTimeout(() => fn(), milissegundos);
  }
}
```

Vamos recapitular. Quando `debounce` for invocado, retornará uma nova função. É essa nova função que será associada ao evento "click" do botão. Quando o evento for disparado, a função parará qualquer timer que esteja em andamento e criará um novo timer.

Se nenhuma outra ação for empreendida na janela de tempo de um segundo, a função passada como primeiro parâmetro de

`debounce` será executada. Porém, se outro clique for realizado antes da janela de um segundo terminar, o temporizador vigente será cancelado e um novo tomará o seu lugar.

Por fim, podemos utilizar a função `debounce` para qualquer evento em nossa aplicação que precise executar apenas uma ação dentro de uma janela de tempo.

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/18>

CHEGANDO AO LIMITE

"Ser ninja não é mais suficiente, é preciso ser cangaceiro" -
Flávio Almeida

No capítulo anterior, implementamos o padrão Debounce para garantir que a chamada do método `importaNegociacoes` de `NegociacaoController` fosse feita apenas uma vez dentro da janela de tempo de um segundo:

```
// app-src/app.js
// código anterior omitido

// RELEMBRANDO APENAS!

$('#botao-importa')
  .addEventListener('click', debounce(() => {
    console.log('EXECUTOU A OPERAÇÃO DO DEBOUNCE');
    controller.importaNegociacoes();
  }, 1000));
```

Isolamos elegantemente a lógica do pattern Debounce em uma função para que fosse possível reaproveitá-la em outros eventos da nossa aplicação. Contudo, se quisermos utilizar nosso `NegociacaoController` em outros projetos, teremos de lembrar de realizar o `debounce` para qualquer evento que chame o

método.

Uma possível solução seria adicionar a chamada de `debounce` diretamente no método `importaNegociacoes`, para termos certeza de que ela seja aplicada independente do evento no qual o método é chamado. Entretanto, isso poluiria o método com um código que não lhe diz respeito, dificultando sua legibilidade. Apesar de não termos uma solução aparente, podemos consegui-la através do padrão de projeto **Decorator**.

19.1 O PADRÃO DE PROJETO DECORATOR

Decorator é um padrão de projeto de software que permite adicionar um comportamento a um objeto já existente em tempo de execução, ou seja, agrega dinamicamente responsabilidades adicionais a um objeto.

Indo direto ao ponto, queremos o mesmo resultado que já temos com apenas esta alteração em nossa aplicação:

```
// CÓDIGO NÃO COMPILA, APENAS EXEMPLO

// app-src/controllers/NegociacaoController.js

// código anterior omitido

    // DECORANDO O MÉTODO

    @debounce()
    async importaNegociacoes() {

        // código omitido
    }
// código posterior omitido
```

Com a sintaxe que acabamos de ver, a aplicação do `debounce`

através de um Decorator é muito menos invasiva, pois não precisamos misturar a lógica de sua chamada dentro do bloco do método. Além disso, independente do lugar que o método `importaNegociacoes` for chamado, o Decorator será aplicado.

Apesar de muito elegante, não há suporte oficial para este recurso no ECMAScript. Há uma proposta em andamento para que Decorators se tornem padrão na linguagem (<https://github.com/tc39/proposal-decorators>). Vamos desistir de usá-lo? Com certeza, não.

Uma das grandes vantagens de utilizarmos um transcompilador é a capacidade de usarmos recursos modernos da linguagem antes mesmo que se tornem oficiais, ou antes mesmo de sua implementação nos navegadores. Como já estamos usando Babel, basta adicionarmos um plugin para termos suporte a esse fantástico recurso.

Não fique com receio de utilizar este recurso experimental. Por exemplo, a linguagem TypeScript, um superset do ECMAScript 2015 criado pela Microsoft, também faz uso experimental de Decorators, que são suportados através de um processo de transcompilação fornecido pelo próprio compilador da linguagem.

Sem delongas, vamos ativar essa poderosa funcionalidade em nosso projeto.

19.2 SUPORTANDO DECORATOR ATRAVÉS DO BABEL

No terminal, dentro da pasta `client`, vamos instalar o plugin `babel-plugin-transform-decorators-legacy`.

```
npm install babel-plugin-transform-decorators-legacy@1.3.4 --save-dev
```

Em seguida, no arquivo `client/.babelrc`, adicionaremos o plugin na lista de plugins:

```
{
  "presets": ["es2017"],
  "plugins": ["transform-es2015-modules-systemjs", "transform-decorators-legacy"]
}
```

Vamos alterar `app-src/app.js` para removermos a chamada da nossa função `debounce`, que se tornará um Decorator. Nosso código voltará para o estágio anterior, antes do uso da função:

```
// app-src/app.js

import { NegociacaoController } from './controllers/NegociacaoController.js';

// NÃO IMPORTA MAIS DEBOUNCE

// código anterior omitido

// voltou para a forma anterior
$('#botao-importa')
  .addEventListener('click', controller.importaNegociacoes.bind(controller));
```

Antes de continuarmos, há uma pegadinha no Visual Studio Code, caso você o esteja utilizando. Quando ele encontrar a sintaxe `@NomeDoDecorator`, exibirá a seguinte mensagem de erro:

[js] Experimental support for decorators is a feature that is subject to change in a future release. Set the 'experimentalDecorators' option to remove this warning.

Para silenciarmos o Visual Studio Code, basta criarmos o arquivo `client/jsconfig.json` com o seguinte conteúdo:

```
{
  "compilerOptions": {
    "experimentalDecorators": true
  }
}
```

É muito importante que a pasta `client` seja aberta no Visual Studio Code para que ele encontre o arquivo `jsconfig.json`. Se abrirmos diretamente `client/app-src`, o arquivo não será encontrado e a mensagem de erro continuará a ser exibida.

Agora que já temos tudo no lugar, daremos início à criação do nosso Decorator. Aliás, podemos apagar o arquivo `app-src/util/Debounce.js`, inclusive remover sua exportação de `app-src/util/index.js`, que no final ficará assim:

```
// app-src/util/index.js

export * from './Bind.js';
export * from './ConnectionFactory.js';
export * from './DaoFactory.js';
export * from './ApplicationException.js';
export * from './HttpService.js';
export * from './ProxyFactory.js';

// REMOUEU O EXPORT DO ARQUIVO QUE ACABAMOS DE APAGAR
```

Vamos criar o arquivo `app-src/util/decorators/Debounce.js`. Seu esqueleto será assim:

```
// app-src/util/decorators/Debounce.js

export function debounce(milissegundos = 500) {
```

```

    return function(target, key, descriptor) {
        return descriptor;
    }
}

```

Temos a função `debounce` que recebe como parâmetro uma quantidade em milissegundos, o tempo que levará em consideração. Se não receber valor algum, seu valor padrão será de meio segundo graças ao *default parameter* suportado pelo ES2015 (ES6).

Toda função Decorator deve retornar outra função que recebe três parâmetros. O primeiro é o `target`, o alvo do nosso Decorator; e o segundo é o nome da propriedade na qual o Decorator foi utilizado. Em nosso caso, como vamos usá-lo para o método `importaNegociacoes`, o parâmetro `key` guardará o nome do método. Por fim, temos o `descriptor`, que merece um destaque especial.

O `descriptor` é um objeto especial que nos dá acesso à implementação original do método ou função, por meio de `descriptor.value`. É importante que o retorno da função que recebe os três parâmetros que vimos sempre devolva o `descriptor`, modificado ou não.

O primeiro passo será guardar uma referência para o método original, antes de ele ser modificado:

```

export function debounce(milissegundos = 500) {

    return function(target, key, descriptor) {

        const metodoOriginal = descriptor.value;
    }
}

```

```

    return descriptor;
  }
}

```

É importante guardar o método original, porque, ao modificarmos `descriptor.value` para aplicar o `debounce`, precisaremos que o método original seja chamado.

Agora, vamos sobrescrever o `descriptor.value` com uma nova função. Como não sabemos se o método decorado possui ou não parâmetros, usaremos o `REST` operator para que nosso novo método esteja preparado para receber nenhum ou mais parâmetros:

```

// app-src/util/decorators/Debounce.js

export function debounce(milissegundos = 500) {

  return function(target, key, descriptor) {

    const metodoOriginal = descriptor.value;

    let timer = 0;
    descriptor.value = function(...args) {
      clearTimeout(timer);
      // aqui entra a implementação do nosso método
      //que substituirá o original
    }

    return descriptor;
  }
}

```

Através de `metodoOriginal.apply(this, args)`, chamamos o método original passando seu contexto e a quantidade de argumentos recebida pelo método que o substituiu:

```

// app-src/util/decorators/Debounce.js

```

```
export function debounce(milissegundos = 500) {

  return function(target, key, descriptor) {

    const metodoOriginal = descriptor.value;

    let timer = 0;

    descriptor.value = function(...args) {
      clearInterval(timer);

      // chama metodoOriginal depois de X milissegundos!
      timer = setTimeout(() => metodoOriginal.apply(this, a
rgs), milissegundos);
    }

    return descriptor;
  }
}
```

Excelente, isso já é suficiente para que nosso Decorator funcione. Vamos exportá-lo no barrel `app-src/util/index.js`, para em seguida importá-lo e utilizá-lo em `NegociacaoController`.

```
// app-src/util/index.js

export * from './Bind.js';
export * from './ConnectionFactory.js';
export * from './DaoFactory.js';
export * from './ApplicationException.js';
export * from './HttpService.js';
export * from './ProxyFactory.js';
export * from './decorators/Debounce.js';
```

Agora, alterando `NegociacaoController`:

```
import { Negociacoes, NegociacaoService, Negociacao } from '../do
main/index.js';
import { NegociacoesView, MensagemView, Mensagem, DateConverter }
from '../ui/index.js';
```

```

/ importou o decorator
import { getNegociacaoDao, Bind, getMessageException, debounce }
from '../util/index.js';

export class NegociacaoController {

    // código anterior omitido

    @debounce()
    async importaNegociacoes() {

        // código omitido
    }
}

```

Podemos testar a aplicação. Inclusive podemos passar a quantidade em milissegundos que desejamos para o `@debounce`, que ela será respeitada:

```

// EXEMPLO: espera um segundo e meio

@debounce(1500)
async importaNegociacoes() {

    // código omitido
}

```

19.3 UM PROBLEMA NÃO ESPERADO COM NOSSO DECORATOR

Que tal aplicarmos a solução também para o método `adiciona()` de `NegociacaoController`? Tudo bem que não temos uma situação na qual o `debounce` faça muito sentido, mas é uma maneira de podermos testar a reutilização do nosso Decorator:

```

// app-src/controllers/NegociacaoController.js
// código anterior omitido

```

```

@debounce()
async adiciona(event) {
    // código omitido
}

// código posterior omitido

```

Assim que recarregamos nossa página com os arquivos atualizados durante o processo de transcompilação do Babel, quando tentamos adicionar novas negociações, a página é recarregada indevidamente. É como se o `event.preventDefault()` do método `adiciona()` não fosse chamado. O que está acontecendo?

Como o método original é substituído por um novo, este novo método não realiza o `event.preventDefault()`. Como o método decorado só será chamado depois de X milissegundos, o evento `submit` não entende que deve cancelar seu evento padrão, o da submissão do formulário, então a página recarregará. Podemos resolver isso facilmente alterando nosso código da seguinte forma:

```

export function debounce(milissegundos = 500) {

    return function(target, key, descriptor) {

        const metodoOriginal = descriptor.value;

        let timer = 0;

        descriptor.value = function(...args) {

            // MUDANÇA!
            if(event) event.preventDefault();

            clearInterval(timer);
            timer = setTimeout(() => metodoOriginal.apply(this, a
rgs), milissegundos);

```

```

    }

    return descriptor;
  }
}

```

Podemos testar se a variável implícita `event` está disponível durante a chamada do método. Se estiver, chamamos `event.preventDefault()`.

Um novo teste indica que nossa aplicação voltou a funcionar como esperado. Mas será que podemos abusar mais um pouquinho deste recurso? É o que veremos na próxima seção.

19.4 ELABORANDO UM DOM INJECTOR

Em nossa classe `NegociacaoController`, buscamos as tags `input` do formulário através da função `querySelector`, que recebe o selector CSS dos elementos. Veja:

```

// REVISANDO APENAS O CÓDIGO

// app-src/controllers/NegociacaoController.js

// código anterior omitido

export class NegociacaoController {

  constructor() {

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

    // código posterior omitido
  }
}

```

Mas que tal criarmos um Decorator que saiba injetar as

dependências dos elementos do DOM na instância de nossa classe?
Queremos algo como:

```
// EXEMPLO APENAS

@controller('#data', '#quantidade', '#valor')
export class NegociacaoController {

  constructor(inputData, inputQuantidade, inputValor) {

    this._inputData = inputData;
    this._inputQuantidade = inputQuantidade;
    this._inputValor = inputValor;
  }
}
```

Neste exemplo, o Decorator `@controller` recebe como parâmetro os seletores CSS dos elementos que desejamos passar para o novo constructor da classe, na mesma ordem.

Agora que já aprendemos a criar Decorators de métodos, chegou a hora de aprendermos a criar Decorators de classes.

19.5 DECORATOR DE CLASSE

Vamos criar nosso Decorator de classe em `app-src/util/decorators/Controller.js`, com a seguinte estrutura:

```
// app-src/util/decorators/Controller.js

export function controller(...seletores) {

  return function(constructor) {

  }
}
```

Nosso Decorator recebe como parâmetro uma quantidade

indeterminada de parâmetros, por isso usamos o REST Operator. Esses parâmetros serão os seletores CSS dos elementos que desejamos buscar.

Assim como um Decorator de método, ele devolve uma função. Mas desta vez, ao ser chamada, ela nos dá acesso ao constructor da classe decorada. É por meio dessa referência do constructor da classe que podemos criar instâncias dentro do nosso Decorator, inclusive passar para essas instâncias os parâmetros que desejarmos.

Precisamos converter a lista de seletores em uma lista de elementos do DOM. Para isso, usaremos a função `map` que todo array possui:

```
// app-src/util/decorators/Controller.js

export function controller(...seletores) {

  // lista com os elementos do DOM

  const elements = seletores.map(seletor =>
    document.querySelector(seletor));

  return function(constructor) {

  }

}
```

Já temos um array com todos os elementos do DOM que precisamos passar para o constructor da classe decorada. Agora, vamos guardar uma referência para o constructor original da classe, para então começarmos a esboçar um novo constructor :

```
// app-src/util/decorators/Controller.js
```

```
export function controller(...seletores) {

  const elements = seletores.map(seletor =>
    document.querySelector(seletor));

  return function(constructor) {

    const constructorOriginal = constructor;

    const constructorNovo = function() {

    }

  }
}
```

O `constructorNovo` deverá chamar `constructorOriginal` passando todos os parâmetros que ele necessita. Um ponto a destacar é que usamos `function` e não *arrow function* para definir o novo `constructor`. Isso não foi por acaso, pois o `this` do `constructor` deve ser dinâmico e não estático. Se tivéssemos usado *arrow function* lá na frente, receberíamos um erro.

Vamos continuar a implementar o novo `constructor`. A instância criada pela chamada de `constructorOriginal` deve ser retornada:

```
// app-src/util/decorators/Controller.js

export function controller(...seletores) {

  const elements = seletores.map(seletor =>
    document.querySelector(seletor));

  return function(constructor) {

    const constructorOriginal = constructor;

    const constructorNovo = function() {

      return new constructorOriginal(...elements);

    }

  }
}
```

```

    }
  }
}

```

Isso ainda não é suficiente. O `constructorNovo` deve ter o mesmo `prototype` de `constructorOriginal`. Isso é importante, pois a classe que decoramos pode ter herdado de outra classe:

```

// app-src/util/decorators/Controller.js

export function controller(...seletores) {

  const elements = seletores.map(seletor =>
    document.querySelector(seletor));

  return function(constructor) {

    const constructorOriginal = constructor;

    const constructorNovo = function() {

      return new constructorOriginal(...elements);
    }

    // ajustando o prototype
    constructorNovo.prototype = constructorOriginal.prototype
  };

  // retornando o novo constructor
  return constructorNovo;
}

```

Terminamos nosso Decorator. Só precisamos exportá-lo através do barril da pasta `app-src/util`, para então importá-lo e utilizá-lo em `NegociacaoController`.

Vamos alterar o barril:

```

// app-src/util/index.js

export * from './Bind.js';

```

```

export * from './ConnectionFactory.js';
export * from './DaoFactory.js';
export * from './ApplicationException.js';
export * from './HttpService.js';
export * from './ProxyFactory.js';
export * from './decorators/Debounce.js';

// exportou!
export * from './decorators/Controller.js';

```

Agora o controller:

```

// app-src/controller/NegociacaoController.js

// código anterior omitido

// IMPORTOU O DECORATOR

import { getNegociacaoDao, Bind, getMessageException, debounce, c
ontroller } from '../util/index.js';

// UTILIZANDO O DECORATOR

@controller('#data', '#quantidade', '#valor')
export class NegociacaoController {

    // MÉTODO AGORA RECEBE TRÊS PARÂMETROS
    constructor(inputData, inputQuantidade, inputValor) {

        this._inputData = inputData;
        this._inputQuantidade = inputQuantidade;
        this._inputValor = inputValor;

        // código posterior omitido
    }
}
// código posterior omitido

```

Basta recarregarmos nossa página para vermos nosso Decorator em ação. Nosso código é apenas uma amostra do que o Decorator de classes pode fazer. Todavia, um olhar atento perceberá que podemos simplificar ainda mais nosso código.

Como o constructor de `NegociacaoController` agora recebe três parâmetros, podemos usar a mesma estratégia com `Object.assign` que utilizamos na classe `Negociacao` para reduzir a quantidade de instruções:

```
// app-src/controller/NegociacaoController.js

// código anterior omitido

import { getNegociacaoDao, Bind, getMessageException, debounce, controller } from '../util/index.js';

@controller('#data', '#quantidade', '#valor')
export class NegociacaoController {

    // OS PARÂMETROS AGORA COMEÇAM COM UNDERLINE

    constructor(_inputData, _inputQuantidade, _inputValor) {

        // UMA ÚNICA INSTRUÇÃO

        Object.assign(this, { _inputData, _inputQuantidade, _inputValor });

        this._negociacoes = new Bind(
            new Negociacoes(),
            new NegociacoesView('#negociacoes'),
            'adiciona', 'esvazia'
        );

        this._mensagem = new Bind(
            new Mensagem(),
            new MensagemView('#mensagemView'),
            'texto'
        );

        this._service = new NegociacaoService();

        this._init();
    }

    // código posterior omitido
```

Angular (<https://angular.io/>) é um framework que faz uso intenso de decorators, inclusive possui um sistema bem avançado de injeção de dependências. O código que escrevemos é uma caricatura do que este poderoso framework pode nos oferecer.

Excelente. Com mais esta etapa concluída, podemos continuar com nossas simplificações na próxima seção.

19.6 SIMPLIFICANDO REQUISIÇÕES AJAX COM A API FETCH

Estamos a um passo de concluirmos nossa aplicação, mas podemos realizar mais uma melhoria que facilitará em muito a vida do desenvolvedor quando o assunto é requisições assíncronas. Primeiro, vamos rever a classe `HttpService` que criamos para isolarmos o objeto `XMLHttpRequest` :

```
// app-src/util/HttpService.js

export class HttpService {

  get(url) {

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

      const 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();

});
}
}

```

Utilizamos o padrão de projeto Promise para facilitar o uso do método `get()` em todos os lugares que esse tipo de requisição é necessário. Contudo, podemos usar um recurso do próprio JavaScript para realizar requisições Ajax que já suporta Promise por padrão, a **API Fetch**.

A API fetch é experimental, mas suportada a partir do Chrome 42, Firefox 39, Internet Explorer 11, Edge 38, Opera 29 e Safari 10. Ela simplifica drasticamente o código para realizarmos requisições assíncronas, comumente chamadas de requisições Ajax no mundo JavaScript.

A API Fetch é mais um recurso experimental bastante usado no mundo JavaScript. Não se preocupe por enquanto com questões de compatibilidade, mas tenha certeza de estar usando um navegador que a suporte neste momento.

Para poder vermos na prática sua simplicidade, vamos apagar o conteúdo do método `get()` de `HttpService` para reescrevê-lo novamente com a API Fetch.

Acessamos a API através da função global `fetch` que recebe como parâmetro a URL que desejamos realizar uma requisição:

```
// app-src/util/HttpService.js

export class HttpService {

  get(url) {

    return fetch(url)
      .then(res => console.log(res));
  }
}
```

A função `fetch` retorna uma `Promise`, e é por isso que obtemos seu resultado através da chamada encadeada à função `then`. Contudo, a resposta não é parseada automaticamente para nós, mas também não será necessário usarmos `JSON.parse()`, como fizemos com a resposta de `XMLHttpRequest`.

No exemplo anterior, `res` nada mais é do que um objeto que encapsula a resposta. Através dos métodos `text()` e `json()`, lidamos com a resposta no formato texto ou como objetos no formato JSON, respectivamente.

No caso, temos interesse em lidar com a resposta no formato JSON:

```
// app-src/util/HttpService.js

export class HttpService {

  // quem chama o método obtém os dados através de .then()
  get(url) {
```

```

    return fetch(url)
      .then(res => res.json());
  }
}

```

Como estamos usando arrow function sem bloco, o resultado de `res.json()` é retornado automaticamente para a próxima chamada, encadeada à função `then()`.

Apesar de muito mais simples do que a versão com `Promise/XMLHttpRequest`, ainda precisamos lidar com possíveis erros durante a operação. Isso pode ser feito por quem chama o método `get()` bastando encadear uma chamada à função `catch()`. Vejamos um exemplo de uso:

```

// EXEMPLO DE UM APENAS

let service = new HttpService();
service
  .get('http://endereco-de-uma-api')
  .then(dados => console.log(dados))
  .catch(err => console.log(err));

```

Porém, há uma pegadinha com a API Fetch. A função `catch` só será chamada apenas se rejeitarmos a Promise. Para que possamos fazer isso, precisamos saber se a requisição foi realizada com sucesso ou não, com base em `res.ok`.

Alterando o código de `HttpService`:

```

export class HttpService {
  _handleErrors(res) {
    // se não estiver ok, lança a exceção
    if(!res.ok) throw new Error(res.statusText);

    // se nenhum exceção foi lançada, retorna a própria respo

```

```

sta
    return res;
}

get(url) {

    return fetch(url)
        .then(res => this._handleErrors(res))
        .then(res => res.json());
}
}

```

Para manter a organização do código, criamos o método privado `_handleErrors()`. O `.then` no `fetch` devolverá a própria requisição `this._handleErrors` que será acessível no próximo `.then` e será convertido para `json`. Com o `if`, identificamos se tudo funcionou bem com o `res.ok`; caso contrário, cairemos no `else` e lançaremos uma exceção. Quando a exceção for lançada, a `Promise` não vai para o `.then` do `get()`, mas seguirá para o `catch`.

Se atualizarmos a página no navegador, em alguns instantes receberemos a mensagem de que as negociações do período foram importadas corretamente.

19.7 CONFIGURANDO UMA REQUISIÇÃO COM API FETCH

Por padrão, `fetch` executa uma requisição com o método `GET`. E se quisermos enviar um `JSON` através do método `POST` para uma API? Nesse caso, precisaremos configurar a requisição.

A boa notícia é que nosso servidor está preparado para receber uma requisição `POST` que envia um `JSON` para o endereço `/negociacoes`. O servidor apenas exibirá no console os dados

recebidos.

O código de teste que escreveremos ficará no arquivo `client/app-src/app.js` . Assim que nossa aplicação for carregada, uma requisição será feita para o servidor. Poderíamos ter criado outro arquivo, mas isso nos poupará tempo.

O primeiro passo será importar `Negociacao` de `./domain/index.js` , e preparar o cabeçalho da requisição, indicando que o `Content-Type` será `application/json` :

```
// client/app-src/app.js

import { NegociacaoController } from './controllers/NegociacaoController.js';

// importou Negociacao!
import { Negociacao } from './domain/index.js';

// código anterior omitido

const negociacao = new Negociacao(new Date(), 1, 200);
const cabecalhos = new Headers();
cabecalhos.set('Content-Type', 'application/json');
```

É por meio de uma instância de `Headers` que configuramos nosso cabeçalho. O objeto possui o método `set` . É através dele que indicamos o `Content-Type` que utilizaremos.

O próximo passo será criarmos um objeto que chamaremos de `config` . Ele possui as propriedades `method` , `headers` e `body` que recebem respectivamente os valores `POST` , `cabecalhos` e `JSON.stringify(negociacao)` .

```
// client/app-src/app.js

import { NegociacaoController } from './controllers/NegociacaoController.js';
```

```
// importou Negociacao!
import { Negociacao } from './domain/index.js';

// código anterior omitido

const negociacao = new Negociacao(new Date(), 1, 200);
const cabecalhos = new Headers();
cabecalhos.set('Content-Type', 'application/json');

const config = {
  method: 'POST',
  headers: cabecalhos,
  body: JSON.stringify(negociacao)
};
```

Com a configuração necessária, executamos nossa requisição por meio de `fetch` :

```
// client/app-src/app.js

import { NegociacaoController } from './controllers/NegociacaoController.js';

// importou Negociacao!
import { Negociacao } from './domain/index.js';

// código anterior omitido

const negociacao = new Negociacao(new Date(), 1, 200);
const cabecalhos = new Headers();
cabecalhos.set('Content-Type', 'application/json');

const config = {
  method: 'POST',
  headers: cabecalhos,
  body: JSON.stringify(negociacao)
};

fetch('/negociacoes', config)
  .then(() => console.log('Dado enviado com sucesso'));
```

Reparem que `fetch` recebe como segundo parâmetro o

objeto com todas as configurações que fizemos. Recarregando nossa página, ao olharmos para o terminal que roda nosso servidor, vemos os dados da negociação que enviamos.

Servidor escutando na porta: 3000

Dado recebido via POST:

```
{ _data: 2017-06-06T20:13:02.799Z, _quantidade: 1, _valor: 200 }
```

Podemos simplificar um pouco mais o código que escrevemos utilizando o atalho para declaração de objetos literais que já vimos neste livro.

Vamos mudar o nome da constante `cabecalhos` para `headers`, e também vamos guardar o resultado de `JSON.stringify(negociacao)` na constante `body` e o método na constante `method`. Agora, a declaração do objeto `config` ficará mais simples:

```
// client/app-src/app.js

import { NegociacaoController } from './controllers/NegociacaoController.js';
import { Negociacao } from './domain/index.js';

// código anterior omitido

const negociacao = new Negociacao(new Date(), 1, 200);
// mudou para headers!
const headers = new Headers();
headers.set('Content-Type', 'application/json');

// nova constante
const body = JSON.stringify(negociacao);

// nova constante
const method = 'POST';

const config = {
  method,
  headers,
```

```
    body
  };

  fetch('/negociacoes', config)
    .then(() => console.log('Dado enviado com sucesso'));
```

É uma pequena mudança, mas que deixa nosso código ainda mais enxuto e legível.

19.8 VALIDAÇÃO COM PARÂMETRO DEFAULT

Estamos prestes a terminar nossa aplicação, porém há mais uma melhoria que podemos realizar antes de finalizá-la. Vejamos nossa classe `Negociacao` uma última vez:

```
// app-src/domain/negociacao/Negociacao.js

export class Negociacao {

  constructor(_data, _quantidade, _valor) {

    Object.assign(this, { _quantidade, _valor });
    this._data = new Date(_data.getTime());
    Object.freeze(this);
  }

  // código posterior omitido
}
```

Sabemos que os parâmetros recebidos em seu `constructor` são obrigatórios, mas em nenhum momento testamos se foram passados.

Uma solução possível é aplicarmos uma série de condições `if` para cada parâmetro, e então lançarmos uma exceção para essas condições:

```
// app-src/domain/negociacao/Negociacao.js

export class Negociacao {

  constructor(_data, _quantidade, _valor) {

    if(!_data) {
      throw new Error('data é um parâmetro obrigatório');
    }

    if(!_quantidade) {
      throw new Error('quantidade é um parâmetro obrigatório');
    }

    if(!_valor) {
      throw new Error('valor é um parâmetro obrigatório');
    }

    Object.assign(this, { _quantidade, _valor });
    this._data = new Date(_data.getTime());
    Object.freeze(this);
  }

  // código posterior omitido
}
```

A solução que vimos, apesar de funcional, é um tanto verbosa. Uma solução mais elegante é isolarmos a exceção lançada em uma função, para em seguida chamá-la como parâmetro default . Com essa estratégia, a função será chamada caso nenhum parâmetro seja chamado.

Vamos criar o arquivo `app-src/util/Obrigatorio.ts` que exportará a função `obrigatorio` :

```
// app-src/util/Obrigatorio.js

export function obrigatorio(parametro) {

  throw new Error(`${parametro} é um parâmetro obrigatório`);
}
```


Não podemos nos esquecer de adicioná-lo no barril:

```
// app-src/util/index.js

export * from './Bind.js';
export * from './ConnectionFactory.js';
export * from './DaoFactory.js';
export * from './ApplicationException.js';
export * from './HttpService.js';
export * from './ProxyFactory.js';
export * from './decorators/Debounce.js';
export * from './decorators/Controller.js';

// exportando!
export * from './Obrigatorio.js';
```

Agora, na classe `Negociacao`, faremos assim:

```
// app-src/controllers/NegociacaoController.js

import { obrigatorio } from '../../../util/index.js';

export class Negociacao {

  constructor(
    _data = obrigatorio('data'),
    _quantidade = obrigatorio('quantidade'),
    _valor = obrigatorio('valor')) {

    Object.assign(this, { _quantidade, _valor });
    this._data = new Date(_data.getTime());
    Object.freeze(this);
  }
}
```

Experimente alterar o método `_criaNegociacao` omitindo a passagem de um parâmetro para o construtor de `Negociacao`. A execução será lançada e mensagem será exibida para o usuário. Uma solução elegante para tratar a obrigatoriedade de parâmetros.

19.9 REFLECT-METADATA: AVANÇANDO NA METAPROGRAMAÇÃO

Nossa aplicação funciona com esperado, mas o arquivo `client/app-src/app.js`, que inicializa nosso controller, realiza um `POST` através da API Fetch e também uma associação de eventos de elementos do DOM com os métodos de nosso controller. Realizar esse tipo de associação é algo corriqueiro no mundo JavaScript. Será que podemos facilitar esse processo?

Uma possível solução é isolarmos a lógica de associação em um único lugar, para então aproveitá-la toda vez que houver a necessidade de associarmos um método de um controller a um evento. Nesse sentido, podemos criar o Decorator `bindEvent` que receberá como parâmetro o evento (*click*, *submit* etc.), o seletor do elemento do DOM (no qual nosso método será associado) e uma flag para indicarmos se queremos realizar ou não `event.preventDefault()`.

Todavia, **um Decorator de método é aplicado antes da classe ser instanciada**. Como seremos capazes de associar um método a um evento sem termos uma instância?

Podemos solucionar o problema anterior utilizando o Decorator `bindEvent` apenas para adicionar informações extras (metadados) sobre o método no qual foi associado. Em seguida, com auxílio de um Decorator de classe, procuraremos nos métodos da instância da classe os metadados adicionados por `bindEvent`. As informações contidas nesses metadados serão usadas para fazer a associação do evento com o método da instância.

Contudo, o ECMAScript ainda não possui uma maneira especificada para inclusão de metadados na definição de nossas classes. Há até uma proposta (<https://rbuckton.github.io/reflect-metadata/>) em discussão que poderá entrar em uma futura atualização.

A boa notícia é que não precisamos esperar o futuro para utilizarmos o recurso que precisamos. Existe o projeto **reflect-metadata** (<https://github.com/rbuckton/reflect-metadata>) que segue a especificação proposta para a linguagem. Inclusive, frameworks como Angular e linguagens como TypeScript fazem uso deste mesmo projeto.

Já existe uma API de reflexão no ECMAScript 2015 (ES6) acessível através do objeto global `Reflect`. Todavia, o *reflect-metadata* adiciona novas funções que permitem trabalhar com metadados.

Vamos instalar o `reflect-metadata` pelo terminal, dentro da pasta `client`, com o comando:

```
npm install reflect-metadata@0.1.10 --save
```

Precisamos importá-lo como primeiro script em `client/index.htm`:

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

<script src="node_modules/reflect-metadata/Reflect.js"></scri
pt>
<script src="node_modules/systemjs/dist/system.js"></script>
<script>
  System
    .import('app/app.js')
    .catch(err => console.error(err));
</script>
```

</body>

Excelente, agora já podemos dar início à construção do nosso Decorator `bindEvent`.

19.10 ADICIONANDO METADADOS COM DECORATOR DE MÉTODO

Criaremos o Decorator que adicionará o metadado que mais tarde usaremos para completar nossa solução.

Vamos criar o arquivo `client/app-src/util/decorators/BindEvent.js`. Ele terá o seguinte esqueleto:

```
// client/app-src/util/decorators/BindEvent.js

import { obrigatorio } from '../../util/index.js';

export function bindEvent(
  event = obrigatorio('event'),
  selector = obrigatorio('selector'),
  prevent = true) {

  return function(target, propertyKey, descriptor) {

    // é aqui que precisamos adicionar os metadados

    return descriptor;
  }
}
```

Até agora não há nenhuma novidade, inclusive usamos nossa função `obrigatorio` nos dois primeiros parâmetros do Decorator. O valor padrão do último parâmetro será `true`, pois adotaremos como padrão o cancelamento do evento com

```
event.preventDefault() .
```

Agora, através de `Reflect.defineMetadata()` , vamos adicionar nosso metadado que chamaremos de `bindEvent` :

```
// client/app-src/util/decorators/BindEvent.js

import { obrigatorio } from '../../util/index.js';

export function bindEvent(
  event = obrigatorio('event'),
  selector = obrigatorio('selector'),
  prevent = true) {

  return function(target, propertyKey, descriptor) {

    Reflect.defineMetadata(
      'bindEvent',
      { event, selector, prevent, propertyKey },
      Object.getPrototypeOf(target), propertyKey);

    return descriptor;
  }
}
```

A função `Reflect.defineMetadata` recebe quatro parâmetros. O primeiro é o nome que identifica nosso metadado. O segundo é um objeto JavaScript, nosso metadado, com as propriedades `event` , `selector` , `prevent` e `propertyKey` através do atalho para declaração de objetos que já vimos.

Em seguida, precisamos passar o `prototype` da instância que desejamos adicionar o metadado, por isso usamos `Object.getPrototypeOf(target)` e por fim o nome da propriedade do objeto que receberá o metadado. Como vamos utilizar `@bindEvent` apenas em métodos, `propertyKey` será o nome do método.

Vamos adicionar nosso novo Decorator ao *barrel* client/util/index.js :

```
// client/util/index.js

export * from './Bind.js';
export * from './ConnectionFactory.js';
export * from './DaoFactory.js';
export * from './ApplicationException.js';
export * from './HttpService.js';
export * from './ProxyFactory.js';
export * from './decorators/Debounce.js';
export * from './decorators/Controller.js';
export * from './Obrigatorio.js';

// importou!
export * from './decorators/BindEvent.js';
```

Sem hiato, vamos importá-lo e adicioná-lo aos métodos adiciona() , importaNegociacoes() e apaga() de NegociacaoController :

```
// client/app-src/controller/NegociacaoController.js

// importações anteriores omitidas

// importou bindEvent
import { getNegociacaoDao, Bind, getMessageException, debounce, controller, bindEvent } from '../util/index.js';

// associa ao evento submit do elemento com seletor .form
@bindEvent('submit', '.form')
@debounce()
async adiciona(event) { // código omitido }

// código omitido

// associa ao evento click do elemento com seletor #botao-importa
@bindEvent('click', '#botao-importa')
@debounce()
async importaNegociacoes() { // código omitido }

// associa ao evento click do elemento com seletor #botao-importa
```

```
@bindEvent('click', '#botao-apaga')
async apaga() { // código omitido }

// código posterior omitido
```

Durante o processo de transcompilação, ele será processado e, durante essa etapa, adicionará os metadados nos métodos em que foi usado. Contudo, precisamos de um Decorator que seja capaz de acessar a instância da classe, buscar os metadados e fazer as associações de eventos. Já temos o de classe `controller`. Vamos modificá-lo para atender nossa necessidade.

19.11 EXTRAINDO METADADOS COM UM DECORATOR DE CLASSE

Nosso primeiro passo será acessar a instância criada e, com o auxílio de `Object.getOwnPropertyNames()`, percorrer todas as propriedades da classe. Para cada propriedade, verificamos através de `Reflect.hasMetadata()` se o metadado `bindEvent` está presente. Esta função recebe como parâmetro o identificador do metadado, a instância da classe e, por último, o nome da propriedade que está sendo verificada:

```
// client/app-src/util/decorators/Controller.js

export function controller(...seletores) {

  const elements = seletores.map(seletor =>
    document.querySelector(seletor));

  return function (constructor) {

    const constructorOriginal = constructor;

    const constructorNovo = function () {

      // guardando uma referência para a instância
```

```

        const instance = new constructorOriginal(...elements)
;

        // varre cada propriedade da classe
        Object
            .getOwnPropertyNames(constructorOriginal.prototype)
e)
            .forEach(property => {

                if (Reflect.hasMetadata('bindEvent', instance
, property)) {

                    // precisa fazer a associação do evento c
om o método

                }
            });
        }

        constructorNovo.prototype = constructorOriginal.prototype
;

        return constructorNovo;
    }
}

```

Estamos quase lá. Agora, dentro da condição `if`, precisamos extrair o objeto que adicionamos como metadado, para então termos os dados necessários para podermos realizar a associação do evento. Vamos criar uma função auxiliar chamada `associaEvento`, que receberá a instância da classe e o metadado.

É através de `Reflect.getMetadata()` que extraímos a informação, passando como parâmetros o identificador do metadado, a instância da classe e o nome da propriedade que possui o metadado:

```

// client/app-src/util/decorators/Controller.js

export function controller(...seletores) {

    const elements = seletores.map(seletor =>

```



```

        document.querySelector(seletor));

    return function (constructor) {

        const constructorOriginal = constructor;

        const constructorNovo = function () {

            const instance = new constructorOriginal(...elements)
;
            Object
                .getOwnPropertyNames(constructorOriginal.prototype)
e)
                .forEach(property => {
                    if (Reflect.hasMetadata('bindEvent', instance
, property)) {

                        associaEvento(instance,
                            Reflect.getMetadata('bindEvent', inst
ance, property));
                    }
                });
        }

        constructorNovo.prototype = constructorOriginal.prototype
;

        return constructorNovo;
    }
}

function associaEvento(instance, metadado) {

    document
        .querySelector(metadado.selector)
        .addEventListener(metadado.event, event => {
            if(metadado.prevent) event.preventDefault();
            instance[metadado.propertyKey](event);
        });
}

```

Reparem que não associamos diretamente o método da instância ao evento em `associaEvento`. Adicionamos uma

função intermediária que então chama o método da instância. Isso foi necessário para livrarmos do desenvolvedor a responsabilidade de realizar `event.preventDefault` nos métodos do controller.

Agora, só nos resta alterar o arquivo `client/app-src/app.js`. Removeremos todas as instruções que associam eventos, inclusive aquela que cria o `alias $`. Nosso arquivo ficará assim:

```
// client/app-src/app.js

import { NegociacaoController } from './controllers/NegociacaoController.js';
import { Negociacao } from './domain/index.js';

const controller = new NegociacaoController();

// REMOVEU O ALIAS $ E AS ASSOCIAÇÕES DOS EVENTOS

const negociacao = new Negociacao(new Date(), 1, 200);
const headers = new Headers();
headers.set('Content-Type', 'application/json');
const body = JSON.stringify(negociacao);
const method = 'POST';

const config = {
  method,
  headers,
  body
};

fetch('/negociacoes', config)
  .then(() => console.log('Dado enviado com sucesso'));
```

Missão cumprida! Simplificamos bastante a maneira pela qual realizávamos a associação de um evento com os métodos de nosso controller.

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/19>

ENFRENTAMENTO FINAL

*“Como é que você chegou aqui com vida, cabra velho?” -
Lampião*

Temos nossa aplicação totalmente funcional. Todavia, existem pontos que devem ser levados em consideração antes de batermos o martelo e dizermos que a aplicação está pronta para produção:

O System.js carrega o primeiro módulo da aplicação e, a partir dele, identifica e busca suas dependências, realizando diversas requisições para o servidor. Minimizar o número de requisições para o servidor melhora a performance da aplicação. Temos como evitar essas múltiplas requisições?

O CSS do Bootstrap já existia no projeto que baixamos no início do livro. Será que podemos usar o próprio npm para baixar e gerenciar as dependências do front-end?

Não há distinção entre a versão do projeto de produção para a de desenvolvimento. Como

concatenar e minificar os scripts para o ambiente de produção?

Estamos usando um server disponibilizado pelo autor para tornar nosso projeto acessível pelo navegador. Como conseguir o resultado parecido com ferramentas conceituadas do mercado?

Podemos responder a todas as perguntas com auxílio do **Webpack**.

20.1 WEBPACK, AGRUPADOR DE MÓDULOS

Webpack (<https://webpack.github.io/>) é um *module bundler*, um agrupador de módulos usado por ferramentas famosas do mercado, como Angular CLI (<https://cli.angular.io/>), Create React App (<https://github.com/facebookincubator/create-react-app>) e Vue CLI (<https://github.com/vuejs/vue-cli>). Mas seu uso não é exclusivo dessas ferramentas.

O Webpack permite tratar diversos recursos da aplicação como um módulo, inclusive arquivos CSS. Tudo isso é feito durante um processo de build que, no final, gera um único arquivo JavaScript, geralmente chamado de `bundle.js`. Este possui todos os módulos necessários para a aplicação. Neste sentido, o Webpack dispensa o uso de um loader, como o `System.js` que usamos em nosso projeto.

20.2 PREPARANDO O TERRENO PARA O WEBPACK

Vamos preparar o terreno para utilizarmos Webpack em nosso projeto. Primeiro, em `client/index.html`, vamos substituir a importação do script do `System.js` e sua configuração por uma única tag `<script>`, que importa o arquivo `dist/bundle.js` que ainda não existe.

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

<div id="negociacoes"></div>

<script src="node_modules/reflect-metadata/Reflect.js"></scri
pt>
<script src="dist/bundle.js"></script>

</body>

</html>
```

Vamos aproveitar e **apagar a pasta** `client/app`, aquela que contém os arquivos transcompilados pelo `Babel`. Nossos arquivos compilados seguirão outro trâmite.

Não utilizaremos mais `babel-cli`, nem o `systemjs`. Vamos removê-los do nosso projeto, inclusive do arquivo `client/package.json`. Conseguimos isso através dos comandos:

```
npm uninstall babel-cli --save-dev
npm uninstall systemjs --save
```

Como vimos, não usaremos mais o `babel-cli`, mas o `Babel` ainda precisa estar presente, pois o `Webpack` o utilizará para transcompilar nossos arquivos. Então, vamos instalar o **babel-core** (<https://www.npmjs.com/package/babel-core>), uma versão do `Babel` desprovida de linha de comando para ser usada por outras ferramentas. Vamos aproveitar e também instalar o `Webpack` de uma vez:

```
npm install babel-core@6.25.0 webpack@3.1.0 --save-dev
```

É importante utilizar as versões homologadas por este autor ao longo do capítulo. Se mais tarde houver a necessidade de atualizar as bibliotecas, isso pode ser feito no final, depois do leitor ter a certeza de que tudo está funcionando.

Agora que temos as dependências fundamentais baixadas, podemos dar início à configuração do Webpack.

20.3 O TEMÍVEL WEBPACK.CONFIG.JS

Webpack centraliza suas configurações no arquivo `webpack.config.js`. Uma das primeiras configurações que precisamos fazer é indicar qual será o primeiro módulo (*entry*) a ser carregado. O Webpack é inteligente para carregar suas dependências a partir do módulo. Por fim, precisamos dizer o local de saída (*output*) do *bundle* criado, o arquivo com todos os módulos resolvidos da aplicação.

Vamos criar o arquivo `client/webpack.config.js`, que lerá o arquivo `./app-src/app.js`. E o resultado com todos os módulos resolvidos será salvo em `dist/bundle.js`.

```
// client/webpack.config.js

const path = require('path');

module.exports = {
  entry: './app-src/app.js',
  output: {
    filename: 'bundle.js',
```

```

    path: path.resolve(__dirname, 'dist')
  }
}

```

Teremos um único arquivo para ser importado pelo navegador, minimizando assim a quantidade de requisições feitas para o servidor. Nosso próximo passo será adicionarmos em `client/package.json` um `npm` script que chame o binário do Webpack, indicando para ele o arquivo de configuração que deve levar em consideração.

Aliás, vamos remover todos os scripts que configuramos até agora, mantendo apenas o `test`, para então adicionarmos o novo script. Em `client/package.json`, faremos:

```

/* código anterior omitido */

"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build-dev": "webpack --config webpack.config.js"
},

/* código posterior omitido */

```

Agora, através do terminal e dentro da pasta `client`, vamos executar o comando:

```
npm run build-dev
```

Durante o processo de `build`, receberemos a seguinte mensagem de erro no terminal:

```

ERROR in ./app-src/controllers/NegociacaoController.js
Module parse failed: /Users/flavioalmeida/Desktop/20/client/app-s
rc/controllers/NegociacaoController.js Unexpected character '@' (
5:0)
You may need an appropriate loader to handle this file type.
| import { getNegociacaoDao, Bind, debounce, controller, bindEven
t } from '../util/index.js';

```



```
|  
| @controller('#data', '#quantidade', '#valor')  
| export class NegociacaoController {  
|  
| @ ./app-src/app.js 1:0-77
```

Isso acontece porque o Webpack não reconhece a sintaxe do Decorator. Faz sentido, porque este recurso só está disponível através do processo de compilação do Babel.

Para resolvermos este problema, precisamos solicitar ao `babel-core` que transpile nossos arquivos antes do Webpack resolver os módulos. Mas como? É o que veremos na próxima seção.

20.4 BABEL-LOADER, A PONTE ENTRE O WEBPACK E O BABEL

A ponte de ligação entre o Webpack e o `babel-core` é o **babel-loader** (<https://github.com/babel/babel-loader>), um carregador exclusivo voltado para o Babel. Esse loader lerá nossas configurações em `client/.babelrc` quando for executado.

Vamos instalar o loader através do terminal:

```
npm install babel-loader@7.1.0 --save-dev
```

Isso ainda não é suficiente, precisamos fazer com que `client/webpack.config.js` utilize o loader que acabamos de baixar. Fazemos isso adicionando a configuração **module**. Dentro de `module`, podemos ter várias **rules** (regras), e cada regra pode usar um módulo específico quando aplicada.

Alterando nosso arquivo `client/webpack.config.js`:

```
// client/webpack.config.js

const path = require('path');
module.exports = {
  entry: './app-src/app.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader'
        }
      }
    ]
  }
}
```

Por enquanto, temos apenas uma regra. A propriedade `test` indica a condição na qual nosso loader será aplicado. Usamos a expressão regular `/\.js$/` para considerar todos os arquivos que terminam com a extensão `.js`. Durante este processo, excluimos a pasta `node_modules`, pois não faz sentido processar os arquivos dela. Por fim, dentro de `use`, indicamos o **loader** que será utilizado, em nosso caso o `babel-loader`.

Agora já podemos tentar executar mais uma vez nosso processo de build através do terminal:

```
npm run build-dev
```

E mais uma vez, recebemos um aviso nele:

```
Hash: b9e23a2ce60c22bf8d05
```

```
Version: webpack 3.0.0
```

```
Time: 286ms
```

Asset	Size	Chunks	Chunk Names
-------	------	--------	-------------

```
bundle.js 3.87 kB      0 [emitted] main
  [0] ./app-src/app.js 1.38 kB {0} [built] [1 warning]
```

```
WARNING in ./app-src/app.js
System.register is not supported by webpack.
```

O que será que foi dessa vez?

Quando usamos `System.js`, configuramos o `client/.babelrc` para transcompilar nossos módulos com um padrão compatível com o loader. Contudo, esse formato não é suportado pelo Webpack. A boa notícia é que, a partir da sua versão 2.0, Webpack já suporta por padrão o sistema de módulos do ES2015 (ES6). Dessa forma, não precisamos mais do módulo `babel-plugin-transform-es2015-modules-systemjs` e podemos removê-lo. Além disso, também podemos atualizar `client/.babelrc` para não fazer mais uso do plugin.

Removendo o módulo via `npm`:

```
npm uninstall babel-plugin-transform-es2015-modules-systemjs --save-dev
```

Agora, precisamos alterar `client/.babelrc`. Em `plugins`, deixaremos apenas o `transform-decorators-legacy`:

```
// client/.babelrc
{
  "presets": ["es2017"],
  "plugins" : ["transform-decorators-legacy"]
}
```

Agora podemos gerar o build do nosso projeto que ele vai até o final, sem problema algum:

```
npm run build-dev
```

Dentro da pasta `client/dist`, foi gerado o arquivo

`bundle.js` . É um arquivo que contém todos os módulos usados pela aplicação concatenados.

Mesmo ainda sem um servidor rodando, se abrirmos `client/index.html` **diretamente no navegador**, nossa aplicação continuará funcionando.

Aprendemos a realizar um build de desenvolvimento, mas minificar scripts em ambiente de produção é uma boa prática. Aliás, ainda não temos uma separação clara entre esses dois ambientes. Chegou a hora de botar a casa em ordem.

20.5 PREPARANDO O BUILD DE PRODUÇÃO

Vamos criar um novo `npm` script chamado `build-prod` , praticamente idêntico ao `build-dev` , mas que receberá o parâmetro `-p` . Isso indicará para o Webpack que estamos solicitando um build de produção, fazendo-o minificar nosso `bundle.js` no final.

Alterando `client/package.json` :

```
// código anterior omitido

"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build-dev": "webpack --config webpack.config.js",
  "build-prod": "webpack -p --config webpack.config.js"
},

// código posterior omitido
```

Por fim, através do terminal, vamos executar o comando:

```
npm run build-prod
```

Um novo erro é exibido no terminal:

```
ERROR in bundle.js from UglifyJs
Unexpected token: name (Negociacao) [bundle.js:75,4]
```

E agora? Quando usamos o parâmetro `-p`, o Webpack usará o módulo `Uglify` para minificar nossos arquivos. Todavia, este módulo ainda não é compatível com a sintaxe do ES2015 (ES6), razão do erro que recebemos.

Uma solução seria transcompilar nosso projeto para ES5, mas isso tornaria o processo de build mais lento, pois teríamos de transcompilar de ES2017 para ES2016, para só depois transcompilar para ES5. Vamos abdicar do parâmetro `-p` e instalar o plugin **babili-webpack-plugin** do Webpack, que sabe minificar corretamente nosso código.

Através do terminal, instalaremos o plugin com o comando:

```
npm install babili-webpack-plugin@0.1.1 --save-dev
```

Excelente, agora precisamos configurá-lo em `client/webpack.config.js`. Porém, esse plugin só pode ser ativado no build de produção, e não no de desenvolvimento. Para fazermos essa distinção, consultaremos a variável de ambiente `NODE_ENV` através de `process.env.NODE_ENV`, dentro de `webpack.config.js`. O plugin só será ativado quando o valor da variável de ambiente `NODE_ENV` for `production`:

```
// client/webpack.config.js

const path = require('path');

// IMPORTANDO O PLUGIN
const babiliPlugin = require('babili-webpack-plugin');

// COMEÇA COM NENHUM PLUGIN
```

```

let plugins = []

if (process.env.NODE_ENV == 'production') {

    // SE FOR PRODUCTION, ADICIONA O PLUGIN NA LISTA

    plugins.push(new babeliPlugin());
}

module.exports = {
  entry: './app-src/app.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader'
        }
      }
    ]
  },
  // mesma coisa que plugins: plugins
  plugins
}

```

Agora, precisamos alterar nosso arquivo `client/package.json`, que deve atribuir à variável de ambiente `NODE_ENV` o valor `production` toda vez que o script `build-prod` for chamado:

```

// ATENÇÃO, POR ENQUANTO INCOMPATÍVEL COM WINDOWS

// código anterior omitido

"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build-dev": "webpack --config webpack.config.js",
  "build-prod": "NODE_ENV=production webpack --config webpack.c

```

```
onfig.js"
  },
```

```
// código posterior omitido
```

O problema é que esta solução só funciona para mudar variáveis de ambiente no Linux e no MAC, mas não funcionará no Windows. Veremos a solução a seguir.

20.6 MUDANDO O AMBIENTE COM CROSS-ENV

Para termos uma solução de atribuição de variáveis de ambiente que funcione nas plataformas Linux, Mac e Windows, podemos usar o módulo **cross-env** (<https://www.npmjs.com/package/cross-env>). Vamos instalá-lo:

```
npm install --save-dev cross-env@5.0.1
```

Agora, vamos alterar o script `build-prod` em `client/package.json` para fazer uso do `cross-env`:

```
// client/package.json
```

```
// código anterior omitido
```

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build-dev": "webpack --config webpack.config.js",
  "build-prod": "cross-env NODE_ENV=production webpack --config
webpack.config.js"
},
```

```
// código posterior omitido
```

Testando nossa alteração:

```
npm run build-prod
```

Agora, o script `build-prod` criará o `client/dis/bundle.js` minificado, perfeito para ser utilizado em produção.

20.7 WEBPACK DEV SERVER E CONFIGURAÇÃO

Estamos utilizando Webpack para agrupar nossos módulos, mas precisar executar o `npm script build-dev` toda vez que alterarmos um arquivo desanima qualquer cangaceiro. Podemos lançar mão do `webpack-dev-server` (<https://www.npmjs.com/package/webpack-dev-server>), um servidor para ambiente de desenvolvimento que se integra com Webpack, disparando o build do projeto sempre que algum arquivo for modificado, e ainda por cima suporta livereload! Porém, antes de instalá-lo, precisamos realizar uma pequena alteração em `jscangaceiro/server/`.

O servidor disponibilizado no download do projeto no início do livro, além de disponibilizar uma API para ser consumida pela nossa aplicação, também disponibiliza nosso projeto para o navegador. Precisamos desabilitar este último recurso, pois quem vai disponibilizar nossa aplicação para o navegador será o `webpack-dev-server`.

Vamos alterar o arquivo `server/config/express.js` e comentar as seguintes linhas de código:

```
// server/config/express.js

// código anterior omitido

// comentar essas linhas que compartilham os arquivos estáticos
```



```

/*
app.set('clientPath', path.join(__dirname, '../..', 'client'));
console.log(app.get('clientPath'));
app.use(express.static(app.get('clientPath')));
*/

// código posterior omitido

```

Essa alteração é suficiente para que nosso servidor disponibilize apenas a API de negociações.

Precisamos alterar agora `client/app-src/domain/negociacao/NegociacaoService.js`. Nós estávamos usando o endereço relativo da API, porque nossa página ficava no mesmo servidor da API. Porém, como realizamos a divisão, precisamos adicionar `http://localhost:3000` ao endereço:

```

// client/app-src/domain/negociacao/NegociacaoController.js

import { HttpService } from '../..//util/HttpService.js';
import { Negociacao } from './Negociacao.js';

export class NegociacaoService {

  constructor() {

    this._http = new HttpService();
  }

  obterNegociacoesDaSemana() {

    return this._http
      .get('http://localhost:3000/negociacoes/semana')
      // código omitido
  }

  obterNegociacoesDaSemanaAnterior() {

    return this._http

```

```

        .get('http://localhost:3000/negociacoes/anterior')
        // código omitido
    }

    obterNegociacoesDaSemanaRetrasada() {

        return this._http
            .get('http://localhost:3000/negociacoes/retrasada')
            // código omitido
    }

    // código posterior omitido
}

```

Precisamos alterar também `client/app-src/app.js` , pois acessamos a API para realizarmos um `POST` :

```

// client/app-src/app.js

// código anterior omitido
fetch('http://localhost:3000/negociacoes', config)
    .then(() => console.log('Dado enviado com sucesso'));

```

Agora já podemos instalar nosso novo servidor. No terminal, dentro da pasta `client` , executaremos o comando:

```
npm install webpack-dev-server@2.5.1 --save-dev
```

Em seguida, vamos adicionar o `npm script start` em `client/package.json` , que levantará nosso servidor.

```

"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build-dev": "webpack --config webpack.config.js",
  "build-prod": "cross-env NODE_ENV=production webpack --config webpack.config.js",
  "start": "webpack-dev-server"
},

```

Antes de continuarmos, vamos **apagar a pasta** `client/dist` . Ela só existirá quando executarmos os scripts `build-dev` ou

`build-prod` . Quando estivermos usando o `webpack-dev-server` , o build será feito na memória do servidor, evitando a escrita em disco para acelerar o seu carregamento. É coisa de cangaceiro mesmo!

Feche todos os terminais. Em seguida, com um novo aberto e dentro da pasta `client` , execute o comando:

```
npm start
```

Receberemos a seguinte mensagem no console:

```
> client@1.0.0 start /Users/flavioalmeida/Desktop/20/client
> webpack-dev-server
```

```
Project is running at http://localhost:8080/
webpack output is served from /
```

```
// outras mensagens de compilação
```

Acessamos agora nossa aplicação em uma nova porta, a **8080** através do endereço `http://localhost:8080` . Porém, algo parece estar errado.

Nossa página carrega, mas o `dist/bundle.js` indicado na tag `script` de `index.html` não foi encontrado. A razão disso é que nosso servidor disponibilizou o arquivo em memória, mas no endereço `localhost:8080/bundle.js` , e não `localhost:8080/dist/bundle.js` . Podemos ajustar o caminho do `bundle` com uma pequena alteração em `client/webpack.config.js` .

Em `client/webpack.config.js` , dentro de `output` , vamos adicionar a chave `publicPath` . Ela indicará o caminho público acessível através do navegador para nosso `bundle`:

```
// client/webpack.config.js

const path = require('path');
const babiliPlugin = require('babili-webpack-plugin');

let plugins = []

if (process.env.NODE_ENV == 'production') {

  plugins.push(new babiliPlugin());
}

module.exports = {
  entry: './app-src/app.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
    publicPath: "dist"
  },
  // código posterior omitido
}
```

Agora, basta pararmos o servidor e executá-lo novamente com `npm start`, para que ele fique de pé mais uma vez com nossa aplicação acessando corretamente `dist/bundle.js`.

Experimente agora realizar alterações no projeto, por exemplo, adicionando um `alert` no método `adiciona()` de `NegociacaoController`. Um novo bundle em memória será gerado e o navegador automaticamente será recarregado. Fantástico!

Mas ainda não acabou, podemos avançar ainda mais pelo sertão do Webpack!

20.8 TRATANDO ARQUIVOS CSS COMO MÓDULOS

Um outro ponto em nossa aplicação que pode ser melhorado é

a maneira como gerenciamos e utilizamos as dependências de front-end. Vamos começar primeiro atacando a questão dos arquivos CSS.

Dentro de `client/css`, temos os arquivos do Bootstrap que foram disponibilizados com o download do projeto usado no início do livro. Contudo, se quisermos utilizar outras bibliotecas CSS, teremos de baixá-las manualmente, para então copiá-las para dentro da pasta `client/css`.

Com certeza não queremos assumir essa tarefa, ainda mais que temos o `npm`, o gerenciador de pacotes do Node.js que pode nos ajudar a gerenciar as dependências do front-end. O primeiro passo que faremos é **apagar a pasta** `client/css`, com os arquivos do Bootstrap que foram disponibilizados. Não se preocupe, confie!

Agora, como verdadeiros cangaceiros, vamos baixar o Bootstrap através do `npm`:

```
npm install bootstrap@3.3.7 --save-dev
```

Como se não bastasse, removeremos a importação dos arquivos CSS do Bootstrap em `client/index.html`. A tag `<head>` ficará assim:

```
<!-- client/index.html -->
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width">
  <title>Negociações</title>
</head>
```

Agora, em vez de importamos o Bootstrap diretamente em

client/index.html , vamos importá-lo **como se fosse um módulo** em client/app-src/app.js , isso mesmo!

```
// client/app-src/app.js

// ATÉ UM CANGACEIRO VIRARIA OS OLHOS PARA ISSO!
import 'bootstrap/dist/css/bootstrap.css';
import 'bootstrap/dist/css/bootstrap-theme.css';

import { NegociacaoController } from './controllers/NegociacaoController.js';
import { Negociacao } from './domain/index.js';

// código posterior omitido
```

Realizamos o import de bootstrap/dist/css/bootstrap.css . De alguma maneira que ainda não sabemos, o Webpack entende por padrão que desejamos acessar na verdade client/node_modules/bootstrap/dist/css/bootstrap.css . É uma maneira nada convencional de importar um CSS, ainda mais que estamos tratando o CSS do Bootstrap como um módulo! Será que funciona?

Olhando no terminal, vemos a seguinte mensagem de erro:

```
ERROR in ./node_modules/bootstrap/dist/css/bootstrap.css
Module parse failed: /Users/flavioalmeida/Desktop/20/client/node_modules/bootstrap/dist/css/bootstrap.css Unexpected token (7:5)
You may need an appropriate loader to handle this file type.
```

O Webpack não sabe tratar o Bootstrap nem qualquer outro CSS como um módulo. Para que isso seja possível, precisamos ensiná-lo através de um loader que saiba lidar com essa situação.

Para conseguirmos carregar CSS como módulos, precisamos do auxílio do **css-loader** ([https://github.com/webpack-contrib/css-](https://github.com/webpack-contrib/css-loader)

loader) e do **style-loader** (<https://github.com/webpack-contrib/style-loader>). O primeiro plugin transforma os arquivos CSS importados em um JSON, e o segundo utiliza essa informação para adicionar de maneira *inline* os estilos diretamente no HTML, através da tag `<style>`, quando o bundle for carregado.

Vamos parar nosso servidor, para então instalar os loaders através do terminal:

```
npm install css-loader@0.28.4 style-loader@0.18.2 --save-dev
```

Isso ainda não é suficiente, precisamos configurá-los em `client/webpack.config.js`. Realizando a modificação:

```
const path = require('path');
const BabiliPlugin = require('babili-webpack-plugin');

let plugins = []

if (process.env.NODE_ENV === 'production') {

  pluginn.push(new BabiliPlugin());
}

module.exports = {
  entry: './app-src/app.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
    publicPath: "dist"
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader'
        }
      }
    ]
  },
}
```

```

    /* NOVA REGRA AQUI */
    {
      test: /\.css$/,
      loader: 'style-loader!css-loader'
    }
  ]
},
plugins
}

```

Adicionamos uma nova regra que se aplica a todos os arquivos que terminam com a extensão `.css`. Para estes arquivos, os loaders `style-loader` e o `css-loader` entrarão em ação para carregá-los. Será que funciona?

Quando subimos mais uma vez o servidor com o comando `npm start`, recebemos uma sucessão de mensagem de erros, porém, como estamos próximos a nos tornarmos cangaceiros JavaScript, elas não devem nos apavorar.

O problema está na dependência do Bootstrap de arquivos de fontes usados por ele. Durante o build, o Webpack não sabe lidar com esses arquivos, inclusive nem sabe que eles devem estar dentro da pasta `client/dist` durante o processo de distribuição. Precisamos treiná-lo através de novos plugins.

Os plugins **url-loader** (<https://github.com/webpack-contrib/url-loader>) e **file-loader** (<https://github.com/webpack-contrib/file-loader>) são aqueles que conseguirão lidar com o carregamento dos arquivos de fontes utilizados pelo Bootstrap.

Através do terminal e, como sempre, dentro da pasta `client`, vamos instalar os plugins:

```
npm install url-loader@0.5.9 file-loader@0.11.2 --save-dev
```


Por fim, precisamos colocar uma regra para cada tipo de arquivo de fonte acessado pelo Bootstrap. Não se assuste com as expressões regulares, elas são expressões clássicas do mundo Webpack:

```
// client/webpack.config.js

const path = require('path');
const BabiliPlugin = require("babili-webpack-plugin");

let plugins = []

if (process.env.NODE_ENV == 'production') {

  plugins.push(new BabiliPlugin());
}

module.exports = {
  entry: './app-src/app.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
    publicPath: "dist"
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader'
        }
      },
      {
        test: /\.css$/,
        loader: 'style-loader!css-loader'
      },
      {
        test: /\.woff|woff2)(\?v=\d+\.\d+\.\d+)?$/,
        loader: 'url-loader?limit=10000&mimetype=applicat
ion/font-woff'
      },
    ]
  }
}
```

```

      test: /\.ttf(\?v=\d+\.\d+\.\d+)?$/,
      loader: 'url-loader?limit=10000&mimetype=applicat
ion/octet-stream'
    },
    {
      test: /\.eot(\?v=\d+\.\d+\.\d+)?$/,
      loader: 'file-loader'
    },
    {
      test: /\.svg(\?v=\d+\.\d+\.\d+)?$/,
      loader: 'url-loader?limit=10000&mimetype=image/sv
g+xml'
    }
  ],
  plugins
}

```

Os loaders são aplicados da direita para a esquerda. Logo, o `css-loader` será aplicado primeiro para então o `style-loader` entrar em ação.

Agora, parando e subindo mais uma vez nosso servidor, o build do Webpack é realizado sem problema algum e nossa aplicação continua sendo acessível através de `http://localhost:8080`.

A utilização do `css-loader` e do `style-loader` permite importarmos qualquer CSS como um módulo em nossos scripts. Como o `client/app-src/app.js` é o primeiro módulo a ser carregado pela aplicação, nada mais justo do que importarmos o `Bootstrap` e qualquer outro CSS que desejarmos utilizar em nossa aplicação. Podemos até fazer um teste.

Vamos criar o arquivo `client/css/meucss.css` com o seguinte estilo que adiciona uma sombra na borda da tabela de negociações:

```
/* client/css/meucss.css */
```

```
table {  
  box-shadow: 5px 5px 5px;  
}
```

Agora, em `client/app-src/app.js`, vamos importar o arquivo CSS que acabamos de criar. Mas atenção, não podemos fazer simplesmente `import 'css/meucss.css'`, pois o Webpack entenderá que desejamos acessar `client/node_modules/css/meucss.css`, já que este é seu default. Para importarmos corretamente `meucss.css`, precisamos descer uma pasta, por isso usamos `import '../css/meucss.css'`:

```
// client/app-src/app.js
```

```
import 'bootstrap/dist/css/bootstrap.css';  
import 'bootstrap/dist/css/bootstrap-theme.css';
```

```
// importou!  
import '../css/meucss.css';
```

```
import { NegociacaoController } from './controllers/NegociacaoController.js';  
import { Negociacao }
```

```
// código posterior omitido
```

Essa alteração fará com que o nosso navegador seja automaticamente recarregado, levando em consideração um novo arquivo `bundle.js` criado em memória com a alteração que acabamos de fazer.

A prática de importar um arquivo CSS como módulo brilha ainda mais quando trabalhamos com componentes. Uma biblioteca que utiliza essa estratégia é a React (<https://facebook.github.io/react/>).

Tudo funciona como esperado. Mas será que funciona mesmo? É o que veremos a seguir.

20.9 RESOLVENDO O FOUC (FLASH OF UNSTYLED CONTENT)

Um olhar atento perceberá que a aplicação carregará e, durante uma janela de tempo bem pequena, ficará sem estilo algum. Isso acontece porque os estilos que importamos estão sendo aplicados programaticamente pelo `bundle` que foi carregado. Aliás, ele incorpora todo o CSS do Bootstrap, um baita CSS gigante!

Podemos voltar com a abordagem tradicional de carregar os arquivos CSS através da tag `link` e, ao mesmo tempo, aproveitarmos o processo de concatenação e minificação para diferentes ambientes.

Primeiro, vamos alterar `client/index.html` e importar o seguinte CSS que construiremos durante o processo de build:

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

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```

<meta charset="UTF-8">
<meta name="viewport" content="width=device-width">
<title>Negociações</title>
<link rel="stylesheet" href="dist/styles.css">
</head>
<!-- código posterior omitido -->

```

Agora, vamos instalar o plugin **extract-text-webpack-plugin** (<https://github.com/webpack-contrib/extract-text-webpack-plugin>) através do terminal, ainda dentro da pasta `client` :

```
npm install extract-text-webpack-plugin@3.0.0 --save-dev
```

Com o plugin instalado, precisamos importá-lo em `webpack.config.js` e guardar sua instância dentro da nossa lista de plugins. Sua instância recebe como parâmetro o nome do arquivo CSS que será gerado no final. Também será preciso alterarmos a regra (*rule*) dos arquivos terminados em `.css` .

Usaremos a função `extract()` presente no plugin que fará um fallback para `style-loader` , caso não consiga extrair o arquivo CSS. Nosso arquivo `webpack.config.js` estará assim:

```

// client/webpack.config.js

const path = require('path');
const babiliPlugin = require('babili-webpack-plugin');
const extractTextPlugin = require('extract-text-webpack-plugin');

let plugins = []

// ADICIONANDO O PLUGIN,
// MAIS ABAIXO, MODIFICAÇÃO DO LOADER

plugins.push(
  new extractTextPlugin("styles.css")
);

if (process.env.NODE_ENV == 'production') {

```

```

    plugins.push(new babiliPlugin());
  }

  module.exports = {
    entry: './app-src/app.js',
    output: {
      filename: 'bundle.js',
      path: path.resolve(__dirname, 'dist'),
      publicPath: "dist"
    },
    module: {
      rules: [
        {
          test: /\.js$/,
          exclude: /node_modules/,
          use: {
            loader: 'babel-loader'
          }
        },
        // MODIFICANDO O LOADER
        {
          test: /\.css$/,
          use: extractTextPlugin.extract({
            fallback: "style-loader",
            use: "css-loader"
          })
        },
        {
          test: /\.woff(\?v=\d+\.\d+\.\d+)?$/,
          loader: 'url-loader?limit=10000&mimetype=applicat
ion/font-woff'
        },
        {
          test: /\.ttf(\?v=\d+\.\d+\.\d+)?$/,
          loader: 'url-loader?limit=10000&mimetype=applicat
ion/octet-stream'
        },
        {
          test: /\.eot(\?v=\d+\.\d+\.\d+)?$/,
          loader: 'file-loader'
        },
        {
          test: /\.svg(\?v=\d+\.\d+\.\d+)?$/,

```

```

        loader: 'url-loader?limit=10000&mimetype=image/svg+xml'
      },
    ],
  },
  plugins
}

```

Agora, podemos fazer o build da nossa aplicação de desenvolvimento ou de produção, pois nos dois casos teremos todos os arquivos CSS concatenados em um único arquivo, separando-o do `bundle.js`.

Mas será que está tudo na mais perfeita harmonia? Veremos na próxima seção.

20.10 RESOLVEMOS UM PROBLEMA E CRIAMOS OUTRO, MAS TEM SOLUÇÃO!

Um cangaceiro mais atento verificará que o arquivo `dist/style.css` gerado não foi minificado. Podemos resolver isso com o plugin **optimize-css-assets-webpack-plugin** (<https://github.com/NMFR/optimize-css-assets-webpack-plugin>). Todavia, ele depende de um minificador CSS, no caso, usaremos o **cssnano** (<http://cssnano.co>).

Vamos instalar ambos através do terminal:

```

npm install optimize-css-assets-webpack-plugin@2.0.0 --save-dev
npm install cssnano@3.10.0 --save-dev

// client/webpack.config.js

const path = require('path');
const babiliPlugin = require('babili-webpack-plugin');
const extractTextPlugin = require('extract-text-webpack-plugin');

```

```
// IMPORTOU O PLUGIN

const optimizeCSSAssetsPlugin = require('optimize-css-assets-webpack-plugin');

let plugins = []

plugins.push(
  new extractTextPlugin("styles.css")
);

if (process.env.NODE_ENV == 'production') {

  plugins.push(new babiliPlugin());

  // ADICIONOU O PLUGIN PARA O BUILD DE PRODUÇÃO

  plugins.push(new optimizeCSSAssetsPlugin({
    cssProcessor: require('cssnano'),
    cssProcessorOptions: {
      discardComments: {
        removeAll: true
      }
    },
    canPrint: true
  })));
}

// código posterior omitido
```

Agora, ao executarmos no terminal o script `npm run build-prod`, o arquivo `dist/styles.css` estará minificado.

Talvez o leitor esteja se perguntando qual a diferença entre loader e plugin. A grosso modo, o primeiro trabalha com cada arquivo individualmente durante ou antes de o bundle ser gerado. Já o segundo trabalha com o bundle no final do seu processo de geração.

20.11 IMPORTANDO SCRIPTS

Somos capazes de importar bibliotecas diretamente de `client/node_modules`, inclusive aprendemos a tratar arquivos CSS como módulos. Será que o procedimento de importação de bibliotecas JavaScript dentro de `client/node_modules` segue o mesmo processo? Veremos.

Não temos um uso prático em nossa aplicação para os componentes do Bootstrap que utilizam seu script `bootstrap.js`. Contudo, vamos importá-lo para verificar se conseguimos carregá-lo corretamente.

Alterando `client/app-src/app.js`:

```
// client/app-src/app.js

import 'bootstrap/dist/css/bootstrap.css';
import 'bootstrap/dist/css/bootstrap-theme.css';

// carregou o script do bootstrap, apenas o modal
import 'bootstrap/js/modal.js';
import '../css/meucss.css';

// código posterior omitido
```

Bom, pelo menos, no terminal não temos nenhum erro de compilação no terminal. Mas se abrirmos o console do navegador, veremos a seguinte mensagem de erro:

```
Uncaught Error: Bootstrap's JavaScript requires jQuery
```

Faz todo sentido, porque o `bootstrap.js` depende do jQuery (<https://jquery.com>) para funcionar, a biblioteca front-end que ainda vive no coletivo imaginário dos programadores front-end. Como estamos usando o próprio `npm` para gerenciar nossas bibliotecas front-end, nada mais justo do que baixar o jQuery através dele:

```
npm install jquery@3.2.1 --save
```

Agora, em `client/app-src/app.js` , vamos importar o jQuery antes do script do `bootstrap.js` :

```
// client/app-src/app.js

import 'bootstrap/dist/css/bootstrap.css';
import 'bootstrap/dist/css/bootstrap-theme.css';

// importou antes
import 'jquery/dist/jquery.js';

import 'bootstrap/js/modal.js';
import '../css/meucss.css';

// código posterior omitido
```

Sabemos que essa alteração gerará um novo `bundle.js` e fará o recarregamento do navegador. Para nossa surpresa, continuamos com o problema anterior no console do navegador:

```
bundle.js:11106 Uncaught Error: Bootstrap's JavaScript requires j
Query
```

Esse Webpack é cheio de surpresas, não é mesmo? Vejamos como proceder a seguir.

20.12 LIDANDO COM DEPENDÊNCIAS GLOBAIS

O jQuery não é encontrado em tempo de execução porque é uma biblioteca que vive no escopo global. Quando ela é processada pelo Webpack, ela é importada como se fosse um módulo e deixa de ser acessível pelo restante da aplicação. Precisamos de um comportamento diferente.

Para tornarmos o jQuery globalmente disponível, precisamos da ajuda do **webpack.ProvidePlugin** (<https://webpack.js.org/plugins/provide-plugin/>). O **webpack.ProvidePlugin** automaticamente carrega módulos em vez de termos de importá-los em qualquer lugar da nossa aplicação.

A boa notícia é que não precisamos instalá-lo, pois ele é padrão do Webpack. Basta importarmos o módulo **webpack** através de `require('webpack')` em `client/webpack.config.js`:

```
// client/webpack.config.js

const path = require('path');
const babiliPlugin = require('babili-webpack-plugin');
const extractTextPlugin = require('extract-text-webpack-plugin');
const optimizeCSSAssetsPlugin = require('optimize-css-assets-webpack-plugin');

// IMPORTOU O MÓDULO DO WEBPACK!
const webpack = require('webpack');

let plugins = []

plugins.push(
  new extractTextPlugin("styles.css")
);

// O PLUGIN VALE TANTO PARA PRODUÇÃO
// QUANTO PARA DESENVOLVIMENTO

plugins.push(
  new webpack.ProvidePlugin({
    $: 'jquery/dist/jquery.js',
    jQuery: 'jquery/dist/jquery.js'
  })
);

if (process.env.NODE_ENV == 'production') {

  plugins.push(new babiliPlugin());
```

```

    plugins.push(new optimizeCSSAssetsPlugin({
      cssProcessor: require('cssnano'),
      cssProcessorOptions: {
        discardComments: {
          removeAll: true
        }
      },
      canPrint: true
    }));
  }
}

```

// código posterior omitido

Na modificação realizada, estamos tornando `jquery/dist/jquery.js` acessível com `$` e `jQuery` para todos os módulos da aplicação. Esses nomes são importantes, porque são os *alias* usados pela biblioteca.

Podemos remover o `import` do `jQuery` que fizemos em `client/app-src/app.js`, pois, como vimos, ele será carregado pelo `webpack.ProvidePlugin`:

```

// client/app-src/app.js

import 'bootstrap/dist/css/bootstrap.css';
import 'bootstrap/dist/css/bootstrap-theme.css';

// removeu o script do jQuery

import 'bootstrap/js/modal.js';
import '../css/meucss.css';

// código posterior omitido

```

Não tente testar a disponibilidade do `jQuery` através do console do navegador, pois ele foi encapsulado dentro de uma função durante o processo de build.

O console Chrome possui uma função auxiliar chamada `$`, disponível quando o jQuery não é carregado. Isso pode confundir-lo, achando que este é o jQuery que carregamos através do ProvidePlugin.

Para saber se ele está disponível ou não, podemos realizar um teste no próprio `client/app-src/app.js`:

```
// client/app-src/app.js

import 'bootstrap/dist/css/bootstrap.css';
import 'bootstrap/dist/css/bootstrap-theme.css';
import 'bootstrap/js/modal.js';
import '../css/meucss.css';
import { NegociacaoController } from '../controllers/NegociacaoController.js';
import { Negociacao } from '../domain/index.js';

$('h1').on('click', () => alert('Foi clicado!'));

// código posterior omitido
```

Um cangaceiro só acredita vendo. Dessa forma, nada mais justo do que verificar no navegador o funcionamento da aplicação.

Aliás, podemos fazer a mesma coisa com o script do Bootstrap que carregamos em nossa aplicação. Assim como jQuery, ele é ativado globalmente, pois modifica o core do jQuery para suportar suas novas funções.

Podemos testar através de `console.log($('h1').modal)` a existência da função `modal` que só estará disponível se o script `modal.js` do Bootstrap for carregado:

```
// client/app-src/app.js

import 'bootstrap/dist/css/bootstrap.css';
import 'bootstrap/dist/css/bootstrap-theme.css';
import 'bootstrap/js/modal.js';
import '../css/meucss.css';
import { NegociacaoController } from './controllers/NegociacaoController.js';
import { Negociacao } from './domain/index.js';

$('h1').on('click', () => alert('Foi clicado!'));

// provando a existência da função!
console.log('Função adicionada pelo bootstrap:');
console.log($('h1').modal);

// código posterior omitido
```

Excelente! Agora já podemos realizar mais um novo ajuste em nossa configuração do Webpack.

20.13 OTIMIZANDO O BUILD COM SCOPE HOISTING

Cada módulo do nosso bundle é envolvido por um *wrapper*, que nada mais é do que uma função. Contudo, a existência desses wrappers tornam a execução do nosso script no navegador um pouco mais lenta.

Entretanto, a partir do Webpack 3, podemos ativar o **Scope Hoisting**. Ele consiste em concatenar o escopo de todos os módulos em um único wrapper, permitindo assim que nosso código seja executado mais rapidamente no navegador.

Vamos ativar esse recurso apenas no build de produção, adicionando uma instância de `ModuleConcatenationPlugin` em nossa lista de plugins:

```
// client/webpack.config.js

// código anterior omitido

if (process.env.NODE_ENV == 'production') {

    // ATIVANDO O SCOPE HOISTING

    plugins.push(new webpack.optimize.ModuleConcatenationPlugin()
);

    plugins.push(new babiliPlugin());

    plugins.push(new optimizeCSSAssetsPlugin({
        cssProcessor: require('cssnano'),
        cssProcessorOptions: {
            discardComments: {
                removeAll: true
            }
        },
        canPrint: true
    }));
}
// código posterior omitido
```

Isso já é suficiente. Contudo, devido à dimensão reduzida da nossa aplicação, teremos dificuldade de ver na prática as mudanças no tempo de execução da nossa aplicação.

20.14 SEPARANDO NOSSO CÓDIGO DAS BIBLIOTECAS

O `bundle.js`, criado pelo nosso script do Webpack, possui o código dos nossos módulos, inclusive o código das bibliotecas que utilizamos. O problema dessa abordagem é o aproveitamento do cache do navegador.

Toda vez que alterarmos o código do nosso projeto, algo

extremamente comum, geraremos um novo `bundle.js` que deverá ser baixado e cacheado pelo navegador. Contudo, se tivermos dois `bundles`, um para nosso código e outro para as bibliotecas, o último ficará mais tempo no cache do navegador, pois muda com menos frequência.

Conseguimos essa divisão através do `CommonsChunkPlugin`, que já vem com o próprio Webpack. Alterando `client/webpack.config.js`:

```
const path = require('path');
const babiliPlugin = require('babili-webpack-plugin');
const extractTextPlugin = require('extract-text-webpack-plugin');
const optimizeCSSAssetsPlugin = require('optimize-css-assets-webpack-plugin');
const webpack = require('webpack');

let plugins = []

// os dois plugins que já configuramos foram omitidos

plugins.push(
  new webpack.optimize.CommonsChunkPlugin(
    {
      name: 'vendor',
      filename: 'vendor.bundle.js'
    }
  )
);
// código posterior omitido
```

Com esta configuração, estamos dizendo que, para a parte da nossa aplicação chamada de `vendor`, todas as bibliotecas que vamos indicar e que fazem parte de `node_modules` farão parte de `vendor.bundle.js`.

Agora, precisamos dividir nossa aplicação em duas partes:

```
// client/webpack.config.js
```



```
// código anterior omitido

// MUDAMOS O ENTRY!

module.exports = {
  entry: {
    app: './app-src/app.js',
    vendor: ['jquery', 'bootstrap', 'reflect-metadata']
  },
  // código posterior omitido
}
```

Nossa aplicação agora possui dois pontos de entrada, `app` e `vendor`. Este nome não é por acaso, ele deve coincidir com o nome que demos na configuração do `CommonsChunkPlugin`. A propriedade recebe um array com o nome dos módulos em `node_modules` que desejamos adicionar no `bundle` `vendor.bundle.js`.

Há mais um detalhe antes de realizarmos o build do nosso projeto. Precisamos alterar `client/index.html` para que carregue o `vendor.bundle.js`. Não precisaremos mais importar o `reflect-metadata`, pois ele fará parte de `vendor.bundle.js`.

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

<div id="negociacoes"></div>

<!-- importou o novo bundle -->
<script src="dist/vendor.bundle.js"></script>
<script src="dist/bundle.js"></script>

</body>
</html>
```

Agora, ao realizar o build do nosso projeto, nossa pasta `client/dist` terá uma estrutura parecida com:

```
├─ 448c34a56d699c29117adc64c43affeb.woff2
├─ 89889688147bd7575d6327160d64e760.svg
├─ bundle.js
├─ e18bbf611f2a2e43afc071aa2f4e1512.ttf
├─ f4769f9bdb7466be65088239c12046d1.eot
├─ fa2772327f55d8198301fdb8bcfc8158.woff
├─ styles.css
└─ vendor.bundle.js
```

20.15 GERANDO A PÁGINA PRINCIPAL AUTOMATICAMENTE

Tivemos que alterar o arquivo `client/index.html` mais de uma vez ao longo da nossa aventura com o Webpack. Adicionamos a importação de `style.css` gerado pelo `extract-text-webpack-plugin` e também a importação de `vendor.bundle.js` gerado pelo `CommonsChunkPlugin`. Esse processo manual não precisa ser assim. Com o auxílio do **html-webpack-plugin** (<https://github.com/jantimon/html-webpack-plugin>) podemos automatizar esse processo.

O `html-webpack-plugin` é capaz de gerar um novo arquivo `html` com todas as importações de artefatos gerados pelo Webpack automaticamente. Inclusive, podemos indicar um template inicial que sofrerá a transformação no lugar de criarmos um novo. Perfeito, pois já temos um código HTML escrito em `client/index.html`.

Vamos instalar o plugin através do terminal:

```
npm install html-webpack-plugin@2.29.0 --save-dev
```

Antes de continuarmos, vamos **renomear o arquivo** `client/index.html` para `client/main.html`. A ideia é gerarmos o arquivo `index.html` dentro de `client/dist` com

todos os outros arquivos gerados pelo Webpack.

Além de renomear o arquivo, precisamos remover todas as tags `<link>` e `<script>` que importam arquivos CSS e JavaScript respectivamente.

Tendo a certeza das alterações que fizemos antes, só nos resta alterar o arquivo `client/webpack.config.js`. Primeiro, vamos importar e adicionar o plugin como primeiro da nossa lista:

```
// client/webpack.config.js

const path = require('path');
const babiliPlugin = require('babili webpack plugin')
const extractTextPlugin = require('extract-text-webpack-plugin');
const optimizeCSSAssetsPlugin = require('optimize-css-assets-webpack-plugin');
const webpack = require('webpack');

// importou o novo plugin
const HtmlWebpackPlugin = require('html-webpack-plugin');

let plugins = []

// adicionou como primeiro da lista
plugins.push(new HtmlWebpackPlugin({
  hash: true,
  minify: {
    html5: true,
    collapseWhitespace: true,
    removeComments: true,
  },
  filename: 'index.html',
  template: __dirname + '/main.html',
}));

// código posterior omitido
```

O plugin recebe um objeto como parâmetro com suas configurações, são elas:

hash: quando `true` , adiciona um hash no final da URL dos arquivos script e CSS importados no arquivo HTML gerado, importante para versionamento e cache no navegador. Quando um bundle diferente for gerado, o hash será diferente e isso é suficiente para invalidar o cache do navegador, fazendo-o carregar o arquivo mais novo.

minify: recebe um objeto como parâmetro com as configurações utilizadas para minificar o HTML. Podemos consultar todas as configurações possíveis no endereço <https://github.com/kangax/html-minifier#options-quick-reference>

filename: o nome do arquivo HTML que será gerado. Respeitará o valor do `path` de `output` que já configuramos logo no início da criação do arquivo `webpack.config.js` .

template: caminho do arquivo que servirá como template para geração de `index.html` .

Por fim, precisamos de mais uma alteração. Como o arquivo `index.html` gerado ficará em `client/dist/index.html` , precisamos ajustar o caminho dos imports dos scripts e dos estilos. Para isso, vamos remover a propriedade `publicPath` da chave `output` :

```
// client/webpack.config.js

// código anterior omitido

module.exports = {
  entry: {
    app: './app-src/app.js',
    vendor: ['jquery', 'bootstrap', 'reflect-metadata']
  },
```

```

    output: {
      filename: 'bundle.js',
      path: path.resolve(__dirname, 'dist')
    },
    /* removeu publicPath */
  },
  // código posterior omitido

```

Pronto. Agora podemos testar a criação de um build, por exemplo, o de desenvolvimento:

```
npm run build-dev
```

Olhando `client/dist` vemos que o arquivo `index.html` foi gerado:

```

├─ 448c34a56d699c29117adc64c43affeb.woff2
├─ 89889688147bd7575d6327160d64e760.svg
├─ bundle.js
├─ e18bbf611f2a2e43afc071aa2f4e1512.ttf
├─ f4769f9bdb7466be65088239c12046d1.eot
├─ fa2772327f55d8198301fdb8bcfc8158.woff
├─ index.html
├─ styles.css
└─ vendor.bundle.js

```

Nossa solução funciona para o build de produção e também quando estamos usando o Webpack Dev Server.

20.16 SIMPLIFICANDO AINDA MAIS A IMPORTAÇÃO DE MÓDULOS

Quando usamos o System.js, somos obrigados a indicar a extensão de arquivo `.js` em nossos imports. Com Webpack, isso não é mais necessário. Vejamos um exemplo:

```

// client/app-src/util/DaoFactory.js

// REMOUEU .js do nome do arquivo
import { ConnectionFactory } from './ConnectionFactory';

```

```
// REMOVEU .js do nome do arquivo
import { NegociacaoDao } from '../domain/negociacao/NegociacaoDao';

export async function getNegociacaoDao() {

    let conn = await ConnectionFactory.getConnection();
    return new NegociacaoDao(conn);
}
```

Além disso, podemos importar diretamente um barrel sem termos que indicar o arquivo `index.js`, indicando apenas o caminho da pasta que contém `index.js`. Vejamos um exemplo:

```
// client/app-src/controllers/NegociacaoController.js

// removeu o index.js, importando apenas a pasta

import { Negociacoes, NegociacaoService, Negociacao } from '../domain';
import { NegociacoesView, MensagemView, Mensagem, DateConverter } from '../ui';
import { getNegociacaoDao, Bind, getExceptionMessage, debounce, controller, bindEvent } from '../util';

@Controller('#data', '#quantidade', '#valor')
export class NegociacaoController {
    // código posterior omitido
}
```

Escrever menos é sempre bom, ainda mais em algo corriqueiro como a importação de módulos.

20.17 CODE SPLITTING E LAZY LOADING

Nossa aplicação é embrião de uma Single Page Application, aquela que não recarrega durante seu uso e que se pauta fortemente em JavaScript. Aliás, esse tipo de aplicação carrega todos os scripts que precisa no primeiro carregamento da página.

Todavia, quando o tamanho do `bundle` é muito grande, o primeiro carregamento da página deixará a desejar, o que pode impactar na experiência do usuário.

O tempo de carregamento da aplicação pode ser reduzido através do **code splitting** (separação de código) do `bundle` principal da aplicação em `bundle` menores carregando-os sob demanda através da técnica de **lazy loading** (carregamento preguiçoso).

Para que possamos ver esse poderoso recurso em ação, elegeremos o módulo `NegociacaoService` como aquele que será carregado sob demanda.

O primeiro passo é removermos todas as importações estáticas do módulo realizadas em nossa aplicação. Primeiro, vamos alterar `client/app-src/domain/index.js` removendo a importação do módulo. Nosso arquivo ficará assim:

```
// client/app-src/domain/index.js

export * from './negociacao/Negociacao';

export * from './negociacao/NegociacaoDao';

// NÃO IMPORTA MAIS O MÓDULO NegotiacaoService

export * from './negociacao/Negociacoes';
```

Agora, em `client/app-src/controller/NegociacaoController.js` vamos remover a importação de `NegociacaoService` logo no início da declaração do nosso módulo, inclusive não teremos mais a propriedade `this._service`, pois o serviço será carregado sob demanda:

```
// client/app-src/controllers/NegociacaoController.js
```

```
// REMOVEU A IMPORTAÇÃO DE NegociacaoService
import { Negociacoes, Negociacao } from '../domain';

import { NegociacoesView, MensagemView, Mensagem, DateConverter }
  from '../ui';
import { getNegociacaoDao, Bind, getExceptionMessage, debounce, c
ontroller, bindEvent } from '../util';

@controller('#data', '#quantidade', '#valor')
export class NegociacaoController {

  constructor(_inputData, _inputQuantidade, _inputValor) {

    Object.assign(this, { _inputData, _inputQuantidade, _input
Valor })

    this._negociacoes = new Bind(
      new Negociacoes(),
      new NegociacoesView('#negociacoes'),
      'adiciona', 'esvazia'
    );

    this._mensagem = new Bind(
      new Mensagem(),
      new MensagemView('#mensagemView'),
      'texto'
    );

    // REMOVEU this._service = new NegociacaoService();

    this._init();
  }
}

// código posterior omitido
```

Agora, com auxílio da função `System.import` importaremos o módulo `../domain/negociacao/NegociacaoService`. O retorno de `System.import` é uma promise que nos dará acesso a um objeto que representa o módulo importado. Sendo o retorno da função uma promise, podemos usar `await`, pois a chamada `System.import` está dentro de um método `async`, no caso


```

importaNegociacoes :

// client/app-src/controllers/NegociacaoController.js

// código anterior omitido

@bindEvent('click', '#botao-importa')
@debounce()
async importaNegociacoes() {

    try {
        // Lazy loading do módulo.
        // Importaremos { NegociacaoService } do módulo.

        const { NegociacaoService } = await System.import('../dom
ain/negociacao/NegociacaoService');

        // criando uma nova instância da classe
        const service = new NegociacaoService();

        // MUDAMOS DE this._service para service

        const negociacoes = await service.obtemNegociacoesDoPerio
do();
        console.log(negociacoes);
        negociacoes.filter(novaNegociacao =>

            !this._negociacoes.toArray().some(negociacaoExisten
te =>
                novaNegociacao.equals(negociacaoExistente)))
            .forEach(negociacao => this._negociacoes.adiciona(neg
ociacao));

        this._mensagem.texto = 'Negociações do período importadas
com sucesso';
    } catch (err) {
        this._mensagem.texto = getExceptionMessage(err);
    }
}

// código posterior omitido

```

Com as alterações realizadas, faremos um teste. Através do

terminal vamos executar o build de desenvolvimento, o mais rápido:

```
npm run build-dev
```

Dentro da pasta `client/dist` existirá um novo arquivo `bundle`, o arquivo `0.bundle.js`:

```
├─ 0.bundle.js
├─ 448c34a56d699c29117adc64c43affeb.woff2
├─ 89889688147bd7575d6327160d64e760.svg
├─ bundle.js
├─ e18bbf611f2a2e43afc071aa2f4e1512.ttf
├─ f4769f9bdb7466be65088239c12046d1.eot
├─ fa2772327f55d8198301fdb8bcfc8158.woff
├─ index.html
├─ styles.css
└─ vendor.bundle.js
```

Durante o processo de build Webpack é inteligente o suficiente para saber que há um carregamento sob demanda de um módulo que não foi importado estaticamente por nenhum outro módulo da aplicação. Detectado o carregamento sob demanda, um novo `bundle` será criado com o código do módulo que será carregado apenas quando o usuário clicar no botão "Importar negociações", pois é na ação do método do controller chamado pelo botão que o módulo será carregado pela primeira vez. Se mais tarde clicarmos outras vezes no botão, nossa aplicação será automaticamente inteligente para saber que o módulo já foi carregado e não o baixará novamente.

Podemos inclusive testar nossa aplicação através do Webpack Dev Server. Pelo terminal, vamos subir nosso servidor com o comando:

```
npm start
```

Ao acessarmos a aplicação através do endereço `localhost:8080` ela continuará funcionando. Porém, podemos ficar mais pertos da especificação ECMAScript do que imaginamos, assunto da próxima seção.

20.18 SYSTEM.IMPORT VS IMPORT

O ECMAScript 2015 (ES6) especificou uma maneira de definir módulos resolvidos estaticamente em tempo de compilação, porém não definiu uma maneira padronizada de carregar o módulo assincronamente sob demanda. Há uma proposta para introduzir importações dinâmicas na linguagem que está bem avançada e que pode ser acompanhada em <https://github.com/tc39/proposal-dynamic-import>. A boa notícia é que o Chrome e outros navegadores já deram início a sua implementação.

Webpack é compatível com a nova sintaxe `import()`, substituta de `System.import()`, porém o parser do Babel não é capaz de reconhecer a sintaxe `import()` como válida sem o auxílio de um plugin. Podemos resolver isso facilmente instalando o plugin `babel-plugin-syntax-dynamic-import` (<https://www.npmjs.com/package/babel-plugin-syntax-dynamic-import>).

No terminal e dentro da pasta `client` vamos executar o comando:

```
npm install babel-plugin-syntax-dynamic-import@6.18.0 --save-dev
```

Em seguida, vamos alterar `client/.babelrc` e adicionar o plugin na lista de plugins:

```
{
  "presets":["es2017"],
  "plugins" : ["transform-decorators-legacy", "babel-plugin-syntax-dynamic-import"]
}
```

Por fim, podemos alterar a chamada de `System.import` para `import` em `NegociacaoController` :

```
// client/app-src/controllers/NegociacaoController.js

// código anterior omitido

@bindEvent('click', '#botao-importa')
@debounce()
async importaNegociacoes() {

  // MUDOU DE System.import para import

  const { NegociacaoService } = await import('../domain/negociacao/NegociacaoService');

  // código posterior omitido
```

Podemos realizar o build do projeto, seja ele de desenvolvimento ou de produção que tudo continuará funcionando, inclusive o Webpack Dev server.

20.19 QUAIS SÃO OS ARQUIVOS DE DISTRIBUIÇÃO?

E viva o cangaço! Agora que completamos definitivamente nossa aplicação, podemos gerar o build de produção para que seja realizado o deploy no servidor de nossa preferência:

```
npm run build-prod
```

Agora, só precisamos levar para produção o conteúdo da pasta

dist .

Nossa aplicação é um projeto embrionário de uma Single Page Application, aquele tipo de aplicação que não recarrega a página durante seu uso e que depende única e exclusivamente de JavaScript para funcionar. Não temos um sistema de roteamento e outros recursos de uma aplicação mais completa criada por frameworks como Angular, React, Ember entre outros. Mas isso não nos impede de colocá-la no ar.

Vale lembrar que nossa aplicação é constituída de duas partes. A primeira, a aplicação cliente, aquela que roda no navegador, a segunda, a API do backend. Focaremos apenas o processo de deploy (colocar no ar) da aplicação cliente, foco desta obra. Todavia, nada impede que o leitor implemente a API utilizada pelo curso utilizando outras linguagens como PHP, Java, C#, etc. realizando o deploy de sua API utilizando o serviço de hospedagem que mais se familiarizar. Aliás, é muito comum aplicações Single Page Application realizarem essa divisão marcante entre o deploy do cliente e da API.

20.20 DEPLOY DO PROJETO NO GITHUB PAGES

Uma maneira rápida e gratuita sem grande burocracia de realizarmos o deploy da nossa aplicação é através do GitHub Pages (<https://pages.github.com/>). O autor presume que o leitor já tenha uma conta no GitHub, configurado a chave de acesso em sua máquina, saiba clonar um repositório já existente e já tenha instalado o Git em sua máquina. Para o leitor que nunca trabalhou com GitHub, esta é uma boa oportunidade de criar sua conta e

seguir o tutorial no próprio site <https://github.com/>.

Para criarmos uma página no GitHub Pages precisamos criar um novo repositório **obrigatoriamente** com a estrutura **usuario.github.io**. O **usuario** no nome do repositório é o nome do nosso usuário cadastrado no GitHub. Por exemplo, o repositório `flaviohenriquealmeida.github.io` inicia com `flaviohenriquealmeida`, usuário do autor deste livro. Depois de criado, clone o repositório em sua máquina em qualquer diretório que **não seja** o mesmo da nossa aplicação:

```
// clonando o repositório, use o endereço do seu repositório.
```

```
git clone git@github.com:flaviohenriquealmeida/flaviohenriquealmeida.github.io.git
```

Excelente! Agora, vamos voltar para nosso projeto e gerar o build de produção:

```
npm run build-prod
```

Dentro de instantes, será criada a pasta `client/dist` com todos os arquivos do nosso projeto. Vamos copiar o **conteúdo** da pasta `client/dist` para dentro do diretório do repositório que acabamos de clonar.

No terminal, agora dentro da pasta do repositório, vamos executar os comandos:

```
git add .
git commit -am "Primeiro commit"
git push
```

Pronto, nosso projeto já está no ar. Para acessá-lo acessamos a URL <https://nomedousuario.github.io/>, no exemplo que acabamos de ver, a URL é

`https://flaviohenriquealmeida.github.io/` . Contudo, há dois problemas que precisam ser destacados.

Nossa aplicação é carregada no navegador, porém não é capaz de acessar nossa API mesmo utilizando o endereço `http://localhost:3000` . Isso acontece porque o GitHub pages só permitirá requisições Ajax para domínios que utilizem `https` e não `http` . Caso você queira desenvolver sua própria API para sua aplicação não esqueça desse detalhe, inclusive precisará habilitar CORS (https://pt.wikipedia.org/wiki/Cross-origin_resource_sharing) caso contrário seu navegador se recusará a realizar a requisição Consulte a documentação da sua plataforma/linguagem preferida para criação de APIs para saber como proceder na habilitação de CORS.

Outro ponto é que a URL da API utilizada em nossa aplicação aponta para um servidor local, de desenvolvimento. Porém, quando nossa aplicação entra em produção, o endereço deverá ser o da API que está no ar e não o endereço local de desenvolvimento. Dessa forma, precisamos fazer com que nosso build de produção seja capaz de levar em consideração outro endereço diferente de `http://localhost:3000` , isto é, que considere o endereço de uma API em produção. Como faremos isso? É o que veremos na próxima seção.

20.21 ALTERANDO O ENDEREÇO DA API NO BUILD DE PRODUÇÃO

O Webpack já vem por padrão com o plugin `webpack.DefinePlugin` . Este plugin permite definir variáveis que só existirão durante o processo de build e que podem receber

diferentes valores de acordo com o tipo de build que estamos realizando.

Primeiro, vamos declarar a variável `SERVICE_URL` que receberá como valor padrão o endereço `http://localhost:3000`. É uma exigência que a string atribuída à variável seja transformada em um JSON através de `JSON.stringify()`. Em seguida, vamos atribuir um novo valor à variável dentro da condição `if` de produção:

```
// client/webpack.config.js

// código anterior omitido

let SERVICE_URL = JSON.stringify('http://localhost:3000');

if (process.env.NODE_ENV == 'production') {

    SERVICE_URL = JSON.stringify("http://enderecodasuaapi.com");

// código posterior omitido
```

Agora, vamos adicionar o plugin na lista de plugins:

```
// client/webpack.config.js

// código anterior omitido

let SERVICE_URL = JSON.stringify('http://localhost:3000');

if (process.env.NODE_ENV == 'production') {

    SERVICE_URL = JSON.stringify("http://enderecodasuaapi.com");

    plugins.push(new webpack.optimize.ModuleConcatenationPlugin()
);
    plugins.push(new babiliPlugin());
    plugins.push(new optimizeCSSAssetsPlugin({
        cssProcessor: require('cssnano'),
        cssProcessorOptions: {
            discardComments: {
```



```

        removeAll: true
      },
      canPrint: true
    }));
  }
}

```

// ADICIONANDO O PLUGIN NA LISTA

```

plugins.push(new webpack.DefinePlugin({
  SERVICE_URL
}));

```

// código posterior omitido

Excelente! Por fim, precisamos alterar `NegociacaoService.js` e `app.js` para que façam uso da variável `SERVICE_URL`.

Alterando `client/app-src/domain/negociacao/NegociacaoService.js`:

// `client/app-src/domain/negociacao/NegociacaoService.js`

// código anterior omitido

```

obtemNegociacoesDaSemana() {
  // PERCEBAM A MUDANÇA, UTILIZANDO TEMPLATE LITERAL!

  return this._http
    .get(`${SERVICE_URL}/negociacoes/semana`)

  // código omitido
}

obtemNegociacoesDaSemanaAnterior() {
  return this._http
    .get(`${SERVICE_URL}/negociacoes/anterior`)
  // código omitido
}

```

```
obtemNegociacoesDaSemanaRetrasada() {

    return this._http
        .get(`${SERVICE_URL}/negociacoes/retrasada`)

        // código omitido
}

// posterior código omitido
```

Por fim, alterando `client/app-src/app.js` :

```
// client/app-src/app.js

// código anterior omitido

fetch(`${SERVICE_URL}/negociacoes`, config)
    .then(() => console.log('Dado enviado com sucesso'));
```

Com as alterações que fizemos, os projetos criados com os scripts `build-prod` e `build-dev` terão endereços diferentes de API. Aliás, vale lembrar que o Webpack Dev Server levará em consideração o mesmo endereço de API do build de desenvolvimento, faz todo sentido pois este servidor é para ser usado apenas localmente.

Por fim, quando usamos o Webpack Dev Server não faz muito sentido exibirmos no terminal as milhares de mensagens de compilação e criação dos nossos bundles. Queremos apenas mensagens de erro durante o build. Podemos resolver isso facilmente adicionando a propriedade `devServer` no arquivo `client/webpack.config.js` :

```
// client/webpack.config.js

// código anterior omitido

plugins,
devServer: {
```

```
    noInfo: true  
  }  
}
```

A propriedade `devServer: { noInfo: true }` indica que apenas as mensagens de erro serão exibidas no console enquanto servirmos nossa aplicação através do Webpack Dev Server.

20.22 CONSIDERAÇÕES FINAIS

Ao longo deste livro, aplicamos desde o início o modelo MVC e padrões de projetos para resolvermos problemas. O tempo todo transitamos entre os paradigmas orientado a objetos e o funcional.

Fomos introduzidos gradativamente aos recursos da linguagem JavaScript das versões ES5, ES2015 (ES6), ES2016 (ES7), inclusive 2017 (ES8), importantes para nosso projeto.

Antecipamos o futuro usando recursos que ainda estão em processo de consolidação na linguagem, graças ao uso de um transcompilador. Aprendemos Webpack e até deploy realizamos!

Caro leitor, ao trilhar seu caminho até esta seção, você acaba de se tornar um **Cangaceiro JavaScript**. Cada linha de código pode lhe ser útil no seu dia a dia, ou até mesmo servir de inspiração para que você crie suas próprias soluções. Para todos vocês, meus votos de sucesso.

Por fim, é possível acompanhar todas as propostas de mudanças na linguagem JavaScript em <https://github.com/tc39/proposals>, para se manter atualizado. Inclusive, a partir deste endereço, você saberá o estágio atual da implementação de Decorators na linguagem. É importante ficar

atento para os estágios (stages) das propostas. São eles:

Stage 0: qualquer discussão, ideia, mudança ou adição que ainda não foi submetida formalmente como proposta.

Stage 1: uma proposta é formalizada e espera-se abordar preocupações transversais, interações com outras propostas e preocupações de implementação. As propostas nesta etapa identificam um problema discreto e oferecem uma solução concreta para esse problema.

Stage 2: a proposta deve oferecer um rascunho inicial da especificação.

Stage 3: neste estágio avançado, o editor de especificações e os revisores designados devem ter assinado a especificação final. É improvável que uma proposta desta fase mude além das correções para os problemas identificados.

State 4: a proposta chega neste estágio quando existem pelo menos duas implementações independentes que aprovam os testes de aceitação.

Olhar o futuro é ter uma nova perspectiva do presente.

O código completo deste capítulo você encontra em <https://github.com/flaviohenriquealmeida/cangaceiro-javascript/tree/master/20>