

BACKBONE.JS + PHANTOMJS

TESTES WEB NO FRONT-END

Serviços AngularJS
Aprenda a criar seus
próprios serviços na web

IA com JavaScript
Ambientes virtuais
simulados com JavaScript

MVP

R\$ 1.000.000,00
INVESTIDOS EM CONTEÚDO
NOS ÚLTIMOS 12 MESES.

APLIQUE ESSE INVESTIMENTO
NA SUA CARREIRA...

E MOSTRE AO MERCADO
QUANTO VOCÊ VALE!

CONFIRA TODO O MATERIAL
QUE VOCÊ TERÁ ACESSO:

- + de **9.000** video-aulas
- + de **290** cursos online
- + de **13.000** artigos
- DEVMEDIA API's consumido + de **500.000** vezes

POR APENAS
R\$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR
MAIS DO MERCADO!



 **DEVMEDIA**

EXPEDIENTE

Editor

Diogo Souza (diogosouzac@gmail.com)

Consultor Técnico

Daniella Costa (daniella.devmedia@gmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araujo

Atendimento ao leitor

A DevMedia possui uma Central de Atendimento on-line, onde você pode tirar suas dúvidas sobre serviços, enviar críticas e sugestões e falar com um de nossos atendentes. Através da nossa central também é possível alterar dados cadastrais, consultar o status de assinaturas e conferir a data de envio de suas revistas. Acesse www.devmedia.com.br/central, ou se preferir entre em contato conosco através do telefone 21 3382-5038.

Publicidade

publicidade@devmedia.com.br – 21 3382-5038

Anúncios – Anunciando nas publicações e nos sites do Grupo DevMedia, você divulga sua marca ou produto para mais de 100 mil desenvolvedores de todo o Brasil, em mais de 200 cidades. Solicite nossos Media Kits, com detalhes sobre preços e formatos de anúncios.

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique à vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site Java Magazine, entre em contato com o editor, informando o título e mini-resumo do tema que você gostaria de publicar:



DIOGO SOUZA

diogosouzac@gmail.com

Analista de Sistemas Java na Indra Company e já trabalhou em empresas como Instituto Atlântico e Ebix L.A. É instrutor Android, palestrante em eventos sobre Java e o mundo mobile e consultor DevMedia. Conhecimentos e experiências em diversas linguagens e ferramentas de programação e manipulação de dados, bem como metodologias úteis no desenvolvimento de Sistemas diversificados.

Sumário

Conteúdo sobre Boas Práticas

04 – Backbone.js e PhantomJS: Criando testes web em front-end

[Sueila Sousa]

Conteúdo sobre Boas Práticas

15 – Simulações artificiais com JavaScript

[Julio Sampaio]

Conteúdo sobre Boas Práticas

24 – Implementando serviços com AngularJS

[Julio Sampaio]

Artigo no estilo Curso

34 – Bootstrap e jQuery: Criando um PDV Web – Parte2

[Madson Aguiar Rodrigues]

Backbone.js e PhantomJS: Criando testes web em front-end

O desenvolvimento de softwares web modernos assiste, hoje, a um verdadeiro renascimento do JavaScript, com a expansão da popularidade de aplicações dirigidas ao front-end, *single-page*, e que executam em tempo real. Para encabeçar a lista de tecnologias que dão suporte a tal status, temos um número considerável e crescente de frameworks web JavaScript que permitem aos desenvolvedores organizar suas aplicações dentro do que chamamos de “componentes modulares dirigidos a convenções”.

A medida que mais lógica e funcionalidades são transportadas do servidor para o browser, tais frameworks se fazem criticamente necessários para manter o estado da aplicação como um todo, evitando códigos *ad hoc* e desestruturados, provendo abstrações e o bom funcionamento de situações comumente encontradas no ciclo de desenvolvimento, como a comunicação com serviços de integração (banco de dados nos dispositivos locais, Web Services, etc.), geração de arquivos de exportação, relatórios, dentre outras.

Este artigo foca no framework **Backbone.js**, que se destaca por fornecer recursos de balanceamento de configuração, incluindo fortes abstrações em seu núcleo, bem como por ter um excelente suporte da sua comunidade de desenvolvimento. Ele provê um conjunto de interfaces (modelos, coleções, roteadores, views, etc.) para o desenvolvimento de uma aplicação enquanto mantém uma enorme flexibilidade ao permitir se comunicar com engines de geração de templates, bem como eventos extensíveis para comunicação entre diferentes componentes, e uma abordagem genérica para a interação entre código e padrões. Ele é usado em larga escala por aplicações de organizações como USA Today, LinkedIn, Hulu, Foursquare, Disqus, e muitas outras. Essencialmente, o Backbone.js provê ferramentas práticas para a construção de aplicações web *cliente-heavy* (que exigem muito poder de processamento do browser) sem exigir tanto esforço do programador para isso.

Fique por dentro

Este artigo trata de dois dos principais frameworks relacionados à comunicação com o back-end e geração de suítes de testes no front-end, respectivamente: o Backbone.js e o PhantomJS. Através da análise feita sobre uma aplicação implementada tendo como base o Backbone.js, aprenderemos a criar suítes de testes dos mais variados tipos (integração, unitários, etc.) que encapsulem todas as regras de negócio da aplicação, bem como auxiliem no processo de identificação de erros, o que, consequentemente, aumentará a produtividade do desenvolvimento do software e do andamento do projeto como um todo.

Além disso, faremos uma análise comparativa entre três ferramentas auxiliares na criação das suítes: o Mocha, o Chai e o SinonJS, entendendo como comunicar as mesmas, e distinguindo os objetivos de cada uma na prática.

Entretanto, esse mundo de evoluções front-end está repleto de muitos obstáculos em potencial. Mais especificamente, enquanto as possibilidades de criação de aplicações das mais diversas formas com frameworks JavaScript modernos como o Backbone.js são infinitas, um dos principais problemas que circundam esse universo está relacionado à qualidade com que os softwares como um todo são entregues, tanto no que se refere ao código e arquitetura da aplicação, quanto em relação ao bom funcionamento dos mesmos. Isso se deve, principalmente, ao fato de ser uma tecnologia recente, e não ter tido ainda todas as boas práticas e padrões organizacionais absorvidos pelos desenvolvedores que a usam.

As estatísticas e a experiência de profissionais e empresas da área provam que o mundo JavaScript é notoriamente difícil de ser verificado, testado: eventos assíncronos no DOM e dados de requisições são sujeitos a problemas de *timing* (tempo limite de processamento) e até falhas maiores, uma vez que a exibição de comportamentos do sistema é difícil de ser separada da lógica da aplicação, além disso as suítes de teste dependem da interação com um browser em específico. Para acabar de completar, os frameworks front-end adicionam uma camada extra de com-

plexidade com interfaces adicionais que precisam ser isoladas e testadas, com vários componentes interagindo concorrentemente, e a lógica de eventos propagando por sobre as camadas da aplicação.

Com base nisso, trataremos de expor aqui o desafio de identificar as partes de uma aplicação a ser testada, definindo os corretos funcionamentos dos vários componentes da mesma, e verificando se o programa funciona como foi planejado para o fazer em sua total integridade. Outrossim, será pré-requisito para a leitura e compreensão deste artigo conhecimentos básicos em JavaScript (o mínimo para que se sinta confortável com objetos, Orientação a Objetos, tipificação, etc.) e jQuery, uma vez que a biblioteca será usada para minimizar a escrita do código complexo da aplicação. Além disso, é necessário que o usuário já tenha conhecimentos medianos sobre o Backbone.js, visto que não faremos implementação de código, e sim de testes.

Estruturando ambiente de teste

Para configurar uma infraestrutura de testes em sua aplicação, você precisa primeiramente de um plano definido de para onde cada uma das partes e pedaços do teste irão existir fisicamente. Começaremos com uma simples estrutura para o nosso repositório de código, como a demonstrada na **Listagem 1**.

Reproduza essa estrutura em um local de sua máquina, de preferência próximo a raiz de algum dos discos do computador, uma vez que precisaremos fazer algumas execuções que exigirão cópia do caminho absoluto do referido diretório.

O arquivo index.html contém o código da aplicação web, enquanto o teste.html fornece as páginas de teste. Já as bibliotecas da aplicação e de teste estarão contidas nas pastas app/js e teste/js, respectivamente.

Nós faremos uso ainda de três bibliotecas complementares aos nossos testes: o **Mocha**, o **Chai** e o **Sinon.JS**, que contêm um conjunto de recursos bem acoplados para testes em Backbone.js. Em adição a elas, também faremos uso do **PhantomJS** para automatizar nossa infraestrutura de testes e executar os testes a partir da linha de comando.

Configurando o PhantomJS

O PhantomJS é um “pseudo-browser” webkit com uma API JavaScript completa que executa no console do seu sistema operacional. Ele tem suporte nativo a vários padrões dos Web Standards, tais como DOM, seletores CSS, JSON, Ajax, Canvas, SVG, etc. O poder do PhantomJS é tão significativo para uma aplicação desse tipo que ele consegue inclusive acessar e manipular páginas com passe livre para o DOM e para todas as bibliotecas JS, como jQuery, Prototype.js, etc. Além de executar testes com o Backbone.

js, ele também suporta outros frameworks de testes funcionais como o Jasmine, o QUnit e o CasperJS. O PhantomJS também é a escolha padrão do famoso framework JavaScript Yeoman e outros.

Para configurar o PhantomJS basta efetuar o download do arquivo binário e mantê-lo no path do seu terminal de comandos. Para isso, acesse a página de downloads do framework disponível na seção **Links** e clique na opção referente ao seu Sistema Operacional (para este artigo usaremos sempre referências ao Windows 8, mas você conseguirá convertê-las facilmente para o seu SO em específico), tal como demonstrado na **Figura 1**. No momento de escrita deste artigo, a versão mais recente do PhantomJS é a 2.0.0.

Nota

O leitor talvez esteja se perguntando o porquê de usar o PhantomJS que não é tão conhecido quanto o WebRat ou o próprio Selenium. O WebRat é um simulador de navegadores, tal como o PhantomJS, com capacidade de manipulação do DOM, seletores CSS, é de fácil integração com RSpec, Cucumber, dentre outros, porém não tem o suporte a JavaScript que o PhantomJS oferece. Já o Selenium apresenta o problema de subir uma instância do navegador físico para realizar os testes, algo que não é preciso com o PhantomJS. O leitor também pode ficar à vontade para aderir a outros browsers JavaScript de sua preferência.

Listagem 1. Estrutura inicial de diretórios da nossa aplicação.

```
01 app/  
02 index.html  
03 css/  
04 js/  
05 app/  
06 lib/  
07 teste/  
08 teste.html  
09 js/  
10 lib/  
11 spec/
```

The screenshot shows the official PhantomJS download page. At the top, there's a navigation bar with links for SOURCE CODE, DOCUMENTATION, API, EXAMPLES, and FAQ. Below the navigation, a message encourages users to improve the document. The main content area has two main sections: 'Documentation' on the left and 'Download' on the right. The 'Documentation' section includes links for 'Get Started' (with sub-links for Download, Build, Releases, Release Names, and REPL), 'Learn' (with sub-links for Quick Start, Headless Testing, Screen Capture, Network Monitoring, Page Automation, Inter Process Communication, and Command Line Interface), and 'Get Help'. The 'Download' section is titled 'Download' and contains a note about binary packages being available on a volunteer basis. It provides links to download the Windows version (phantomjs-2.0.0-windows.zip) and the source code (phantomjs-2.0.0.tar.gz). A note states that the executable phantomjs.exe is ready to use. Another note specifies that the static build is self-contained and can run on Windows XP or later versions without Qt, WebKit, or other dependencies.

Figura 1. Página oficial de download do PhantomJS

Backbone.js e PhantomJS: Criando testes web em front-end

Após o download, extraia todo o conteúdo do arquivo zip e verifique que dentro da pasta bin temos um executável chamado "phantomjs.exe", copie-o e o transfira para um diretório onde irá salvar a raiz de execução de todos os seus projetos que precisem usar o PhantomJS. Usaremos o caminho "D:\PhantomJS" por atender às requisições iniciais feitas no início do artigo sobre a proximidade com a raiz dos discos. Após isso, você precisa referenciar esse arquivo no path do seu SO, podemos fazer isso de duas maneiras (Windows):

1. Referenciando o arquivo diretamente nas variáveis de ambiente do seu Sistema Operacional. A vantagem é que não precisamos mais nos preocupar em fazer essa referência sempre que precisarmos usar o PhantomJS.
2. Podemos abrir uma instância do terminal de comandos diretamente de dentro do

diretório onde se encontra o arquivo. Essa opção será a que usaremos por motivos de simplificação do processo.

Para isso, clique com o botão direito do mouse, segurando a tecla Shift do teclado, na pasta "PhantomJS" que você criou no disco D: e selecione a opção "Abrir janela de comando aqui". Quando o prompt cmd aparecer, digite o seguinte comando:

```
phantomjs --version
```

Isso irá imprimir a versão do PhantomJS instalada, que no nosso caso será 2.0.0 (**Figura 2**).

Do mesmo modo, nós também temos duas formas padrão de executar código JavaScript via PhantomJS, são elas:

1. Via **REPL** (*Interactive Mode*, ou Modo Interativo): Permite que você digite o cód

digo JavaScript diretamente na linha de comando e receba os resultados em tempo real e não permite execução de arquivos JS (**Figura 3**). Para entrar nesse modo, basta executar o arquivo .exe que você baixou do PhantomJS.

2. Via linha de comando: Esse modo foi o que iniciamos e permite que você execute arquivos JavaScript (extensão .js) completos (**Figura 4**). É a opção mais viável e usada entre ambas.

Para executar o código via linha de comando vamos considerar um exemplo bem simples com funções básicas como a impressão de mensagens de log (muito importantes, principalmente por não ser possível debugar o código JavaScript usando essa ferramenta) e a chamada de funções com passagem de parâmetros (**Listagem 2**). Os parâmetros nessa situação assumem as qualidades de objetos mock.

```
C:\WINDOWS\system32\cmd.exe
D:\PhantomJS>phantomjs --version
2.0.0
D:\PhantomJS>
```

Figura 2. Verificando versão do PhantomJS instalado

```
D:\PhantomJS\phantomjs.exe
phantomjs> phantomjs teste.js
Expected an identifier but found 'teste' instead
phantomjs> //repl-input:1 in global code
phantomjs>
phantomjs>
phantomjs> console.log('Olá Mundo!');
Olá Mundo!
undefined
phantomjs> -
```

Figura 3. Exemplo de uso do modo REPL do PhantomJS

Listagem 2. Código de teste do exemplo ilustrado na **Figura 4**.

```
01 console.log("O componente de log funciona
normalmente comigo!");
02
03 function aloMundo(alô) {
04   return "DevMedia... Olá " + alô;
05 }
06
07 console.log("Vamos dizer olá:", aloMundo("Brasil"));
08
09 phantom.exit();
```

Observe que na linha 9 temos o uso do objeto **phantom**, este que está disponível para todos os testes implicitamente e que deve ter o seu método **exit()** sendo chamado obrigatoriamente ao final de cada suíte de testes, uma vez que o JavaScript nem sempre executa de forma linear, como no caso das funções de callback, por exemplo. O objeto **console**, presente nas linhas 1 e 7, também é acessível implicitamente, assim como os demais objetos que, por padrão, o são em um ambiente de browser real.

```
C:\WINDOWS\system32\cmd.exe
D:\PhantomJS>phantomjs teste.js
O componente de log funciona normalmente comigo!
Vamos dizer olá: DevMedia... Olá Brasil
D:\PhantomJS>
```

Figura 4. Exemplo de uso do modo linha de comando do PhantomJS

Carregando páginas

Através da API do PhantomJS, nós podemos executar qualquer URL e trabalhar com nossas páginas de duas perspectivas:

- Como JavaScript na página;
- Como um usuário olhando para a página.

Para exemplificar, vamos tentar algo mais ousado e palpável. Suponha que você como testador de um sistema que usará o PhantomJS como ferramenta padrão e que, consequentemente, não terá como visualizar as coisas graficamente, deseja gerar imagens para as alterações no DOM (que são perfeitamente possíveis via PhantomJS, mesmo em domínios que não te pertencem) que você fizer pelo JavaScript. Para fazer isso, nós teríamos de criar uma forma de recuperar a página em questão, e, em seguida, recuperar toda a sua árvore de objetos DOM, bem como o JavaScript que a acompanha. Suponha ainda que quiséssemos fazer esse teste com a página inicial do portal da DevMedia, alterando atributos e o código HTML da mesma.

Para tanto, crie um novo arquivo de script no mesmo diretório do anterior (lembre-se de que eles ainda não fazem parte do projeto original), nomeie-o “teste2.html” e adicione o código contido na **Listagem 3**.

Listagem 3. Testando páginas simuladas no PhantomJS.

```
01 var pagina = require('webpage').create();
02
03 pagina.open('http://www.devmedia.com.br', function () {
04   var titulo = pagina.evaluate(function () {
05     var posts = document.getElementsByClassName("dm-busca");
06     posts[0].style.backgroundColor ="#33F55A";
07
08     var tit1 = document.getElementsByClassName("font-dev");
09     tit1[1].innerHTML = 'Phantom';
10    var tit2 = document.getElementsByClassName("font-media");
11    tit2[1].innerHTML = 'JS';
12    return document.title;
13  });
14  pagina.clipRect = { top: 0, left: 0, width: 600, height: 500 };
15  pagina.render(titulo + ".png");
16  phantom.exit();
17});
```

Vamos às observações da referida listagem:

- Na linha 1 nós estamos simplesmente carregando o módulo **webpage** do PhantomJS e criando o objeto que o manipulará. Esse módulo é o responsável por estabelecer a comunicação direta com o browser invisível e interpretar todo o conteúdo de hipertexto tal como um browser real faria.
- Na linha 3 estamos chamando a função **open()** que se encarrega de chamar a referida URL, passada como parâmetro (o site da DevMedia, no nosso caso), mas é dentro da função de segundo parâmetro (callback) que podemos de fato executar o que acontece quando ela é retornada. Toda a informação da página está carregada na variável **pagina**.
- A função **evaluate()** da linha 4 se encarrega de “avaliar” algum elemento da página que deve ser retornado no final da mesma função. No nosso caso, estamos buscando todos os elementos DOM que tem como classe CSS “dm-busca” (linha 5), para assim alterar a cor de fundo, tal como fazemos na linha 6.
- Das linhas 8 a 12 estamos apenas recuperando mais dois elementos DOM, referentes aos dois spans que compõem o nome da logo da DevMedia (que, inclusive, tem estilos definidos diferentes,

por isso dois spans), e após modificando os valores de seus HTML internos (somente texto). Isso fará com que o título do topo da página mude para o texto “PhantomJS”. Finalmente, retornamos o valor do título HTML da página para ser usado na geração da imagem. Perceba que mesmo retornando um valor no final da função, essa não é a única responsabilidade dela, que pode ser usada para quaisquer outras ações de script.

- Na linha 14 estamos definindo o valor da propriedade **clipRect** que serve para determinar quais serão as dimensões da página, como se fosse um browser de verdade. Essa opção é muito útil para testar aplicações que exigem um alto grau de responsividade. Dessa forma, tanto o desenvolvedor quanto o testador poderão visualizar o funcionamento das telas em quaisquer dimensões desejadas.
- Finalmente, na linha 15 chamamos a função **render()** no objeto de página para renderizar a imagem final e na linha 16 finalizamos o PhantomJS como de costume.

Para executar este exemplo basta rodar o comando no prompt:

```
phantomjs teste2.js
```

Se tudo correr bem, você terá uma imagem “DevMedia - Tutoriais, Videos e Cursos de Programação.png” gerada no mesmo diretório onde foi executado o comando. O resultado da imagem pode ser visualizado na **Figura 5**.

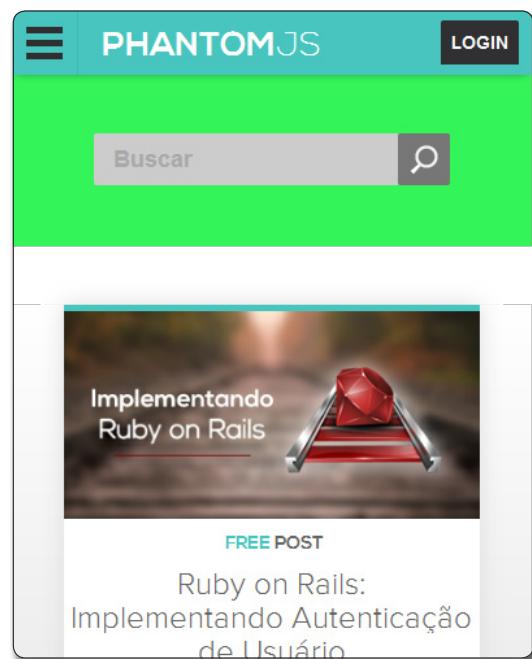


Figura 5. Resultado final de alterações na página home da DevMedia

Perceba que o efeito final foi no título do cabeçalho da página, bem como na sua cor. Se o leitor desejar efetuar o teste nas dimensões de desktop da página, basta mudar os valores da propriedade **clipRect** mencionada.

Caso algo dê errado, como se o PhantomJS não encontrar algum dos elementos do arquivo de script, uma mensagem de erro intuitiva será impressa no console, tal como:

```
TypeError: undefined is not an object (evaluating 'posts[2].style')
phantomjs://webpage.evaluate():5
phantomjs://webpage.evaluate():7
```

Dessa forma, você poderá entender o que está acontecendo, tal como se estivesse em uma ferramenta de análise de logs em console real.

Configurando demais ferramentas

Mocha

O **Mocha** é um framework de testes JavaScript rico que executa no Node.js e no browser, fazendo dos testes assíncronos mais simples e práticos. Os testes no Mocha executam de forma serial, permitindo a análise dos resultados de forma flexível e apurada, enquanto mapeia exceções não capturadas para os casos de testes respectivos.

Além disso, ele suporta inúmeros recursos como suítes de testes, specs, e uma gama enorme de paradigmas de teste. Algumas das features oferecidas pelo Mocha incluem integrações entre front-end e back-end, definição de timeouts versáteis, identificação de testes lentos, e vários tipos de relatórios de testes distintos.

Para executar testes Mocha em um browser, nós precisamos somente de dois arquivos: **mocha.js**, que contém o código JavaScript da biblioteca e **mocha.css**, que salva o estilo CSS para as páginas de relatórios que ele gera. Ambos os arquivos podem ser encontrados na página oficial do projeto no GitHub (vide seção **Links**). No momento da escrita deste artigo, a versão mais recente era a 2.2.4 e será a que usaremos.

Para facilitar o uso, acesse a versão raw dos arquivos e salve-os para o seu workspace na pasta teste/js/lib, assim não ficaremos dependendo de conexão para usá-los.

Chai

Tal como o Mocha, o **Chai** é uma biblioteca de asserções com suporte a BDD (*Behavior-Driven Development*)/TDD (*Test-Driven Development*) para Node.js e browsers, que pode ser associado com qualquer framework de teste JavaScript. Além disso, ele provê uma lista de plug-ins de integração que podem ser acoplados ao framework de modo a incorporar variados tipos de funcionalidades extras. Você também pode desenvolver o seu próprio plugin ou incorporar funcionalidades aos existentes.

Para o nosso projeto, precisamos baixar apenas um único arquivo JavaScript chamado **chai.js**. Na página oficial do framework (disponível na seção **Links**) você encontra o link de download que o redirecionará para a página do Github. Efetue o download do arquivo tal como fizemos para o Mocha e o salve também na pasta teste/js/lib. A versão que usaremos é a mais recente: 2.1.0.

Sinon.JS

O **Sinon.JS** é uma biblioteca que fornece uma suíte poderosa de *test spies* (espiões de teste), stubs e mocks. Os **Spies** são funções que analisam e armazenam informações acerca de uma determinada função e podem ser usados para verificar o histórico comportamental da função no teste. Os **Stubs** são espiões que podem substituir uma função com um comportamento mais ameno para o teste. Já os **Mocks** espionam funções assim como verificam se um determinado comportamento ocorreu durante a execução de um teste.

Essa biblioteca nos permitirá separar comportamentos testáveis da aplicação e focar em uma coisa por vez. Assim como o Chai, nós precisaremos apenas de um simples arquivo JS para usar o Sinon.JS em nossos testes. Na seção **Links** do artigo vocês encontram também o link da página oficial do projeto, que se encontra atualmente na versão 1.14.1. Efetue o download do arquivo e o coloque no mesmo diretório dos anteriores.

Criando os primeiros testes

Uma vez configuradas as bibliotecas de teste que serão usadas como comparativo, nós podemos agora criar uma página de teste que incluirá a aplicação e as referidas bibliotecas, configurar os testes e executá-los, assim como exibir os relatórios no final. Para isso, criemos uma nova página teste.html dentro do diretório / teste e adicionemos nela o código da **Listagem 4**.

Listagem 4. Código HTML da página de testes.

```
01 <html>
02   <head>
03     <title>Testes com Backbone.js</title>
04     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
05     <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
06     <link rel="stylesheet" href="js/lib/mocha.css"/>
07     <script src="js/lib/mocha.js"></script>
08     <script src="js/lib/chai.js"></script >
09     <script src="js/lib/sinon-1.14.1.js"></script>
10   <script>
11     // Configuração.
12     var expect = chai.expect;
13     mocha.setup("bdd");
14     // Executando os testes no evento de load.
15     window.onload = function () {
16       mocha.run();
17     };
18   </script>
19   <script src="js/spec/ola.spec.js"></script>
20   </head>
21   <body>
22     <div id="mocha"></div>
23   </body>
24 </html>
```

Essa é a estrutura mínima para iniciar o processo de testes com os frameworks escolhidos. Vejamos alguns detalhes:

- Até a linha 5, as definições de tags meta e title são padrão da HTML5 (versão que estamos usando) e autoexplicativas.
- Das linhas 6 a 9 estamos importando os arquivos de JavaScript e CSS baixados.

- Das linhas 10 a 18, estamos configurando o Mocha e o Chai. O Chai é configurado para exportar globalmente as funções de assertão. Já o Mocha é configurado para usar a interface de testes bdd e iniciar os mesmos no evento window.onload.
- Na linha 19 estamos adicionando uma importação de um arquivo que ainda não foi criado: ola.spec.js. Ele será importante mais à frente.
- Finalmente, nas linhas 22 a 24 temos o código que o Mocha usará para gerar todo o conteúdo HTML do relatório do teste.

Ao final desse processo, o leitor pode salvar a página e executar no browser para ver como o Mocha irá gera o HTML padrão de relatório, até então vazio.

Crie agora o arquivo ola.spec.js na pasta teste/js/spec e adicione o conteúdo da **Listagem 5** a ele.

Listagem 5. Código JavaScript para a página de testes.

```
01 window.ola = function () {
02   return "Olá Mundo!";
03 };
04
05 describe("Testando biblioteca de testes", function () {
06   describe("Chai", function () {
07     it("deve ser igual usando 'expect'", function () {
08       expect(ola()).to.equal("Olá Mundo!");
09     });
10   });
11   describe("Sinon.JS", function () {
12     it("deve reportar o spy chamado", function () {
13       var olaSpy = sinon.spy(window, 'ola');
14
15       expect(olaSpy.called).to.be.false;
16       ola();
17       expect(olaSpy.called).to.be.true;
18       ola.restore();
19     });
20   });
21});
```

O nosso primeiro exemplo se atém a analisar o funcionamento correto do ambiente, e não funções complexas de código, por isso optamos por analisar um simples Alô Mundo. Nas primeiras linhas da listagem é possível observar que não só criamos a função como também definimos quando ela será chamada (onload). Nesse exemplo, perceba que estamos colocando o código de teste junto com o código da aplicação, o que não pode ocorrer em ambiente de desenvolvimento/produção.

Nas linhas 5, 6 e 11 podemos ver o uso da função **describe()** do Mocha que serve para descrever a hierarquia da nossa suíte de testes. Veja que temos duas suítes: a primeira na linha 6 do Mocha, e a segunda na linha 11 do Sinon.JS. O teste na suíte do Chai usa a função **expect()** para ilustrar uma assertão simples, a mesma que usamos em outros frameworks de testes. Já a suíte do Sinon.JS faz uso de um spy (espião) para rastrear os passos da execução (linhas 13 a 18).

Agora basta salvar todos os arquivos e executar a página de teste. O resultado deverá ser semelhante ao da **Figura 6**.

Testando biblioteca de testes

Chai

- ✓ deve ser igual usando 'expect'

Sinon.JS

- ✓ deve reportar o spy chamado

passes: 2 failures: 0 duration: 0.01s 100%

Figura 6. Resultado da execução dos testes com Chai e Sinon.JS

O relatório do Mocha disponibiliza uma sumarização detalhada do que aconteceu no nosso teste. As colunas do topo na direita mostram que dois testes passaram, nenhum falhou, e os testes juntos levaram 0.01 segundos para executar. Os textos dos cabeçalhos, por sua vez, são referentes aos que nós definimos como primeiros parâmetros da função **describe()**. Cada especificação de teste tem um *tick* verde próximo ao texto, indicando que o mesmo passou.

A página de relatório do teste também disponibiliza algumas opções para análise de subconjuntos da coleção de testes. Veja a **Figura 7**.

Testando biblioteca de testes

Chai

- ✓ deve ser igual usando 'expect'

Sinon.JS

- ✓ deve reportar o spy chamado

Text da Especificação

Botão de Filtro

passes: 2 failures: 0 duration: 0.04s 100%

Figura 7. Especificação de código e filtro na página de relatório

Nela é possível observar que, ao clicar no texto do cabeçalho de qualquer uma das suítes, o resultado será a exibição da div com o código JavaScript que foi testado para a mesma suíte. No canto direito da mesma exibição teremos um botão de filtro, que irá se encarregar de reexecutar aquele teste em específico e exibi-lo individualmente em uma outra página.

Testes de temporização

Todos os nossos testes executaram com sucesso até agora, porém em cenários de desenvolvimento reais temos de lidar com falhas e ineficiências no caminho para criar aplicações web robustas. Para ajudar nisso, a ferramenta de relatórios do Mocha auxilia na identificação de testes lentos e também analisa falhas.

Nota

Os testes de velocidade, apesar de parecerem desnecessários, são muito importantes por identificarem locais do código onde eventuais problemas podem estar acontecendo, e que podem ser corrigidos melhorando a performance da aplicação como um todo. Se tais testes não forem feitos em fase de desenvolvimento ainda, no futuro podem acarretar muitas dores de cabeça quando "stressarmos" a aplicação em produção.

Backbone.js e PhantomJS: Criando testes web em front-end

Para testar isso, criemos um novo arquivo chamado teste-tempo.spec.js dentro da pasta teste/js/spec e adicionemos o código da **Listagem 6** ao mesmo.

Listagem 6. Código JavaScript para teste de temporização.

```
01 describe("Teste de temporização", function() {  
02   it("deve ser um teste rápido", function(done) {  
03     expect("oi").to.equal("oi");  
04     done();  
05   });  
06   it("deve ser um teste médio", function(done) {  
07     setTimeout(function() {  
08       expect("oi").to.equal("oi");  
09       done();  
10     }, 60);  
11   });  
12   it("deve ser um teste lento", function(done) {  
13     setTimeout(function() {  
14       expect("oi").to.equal("oi");  
15       done();  
16     }, 120);  
17   });  
18   it("deve ser um teste falho", function(done) {  
19     setTimeout(function() {  
20       expect("oi").to.equal("oi");  
21       done();  
22     }, 2015);  
23   });  
24});
```

Essa suíte de testes se assemelha à anterior em virtude das funções já conhecidas e reusadas (it, expect, describe...), porém nela podemos ver o uso da função `setTimeout()`, que pertence ao próprio JavaScript, e serve para definir um período de espera na linha de execução do código. Ela é perfeita para que possamos definir o timeout de análise de cada situação. Para fazer com que o código execute de forma assíncrona nós usamos o parâmetro `done` que impõe um *delay* (atraso) à completude do teste até que a função `done()` seja chamada de fato.

O primeiro teste (linha 2) não tem nenhum atraso antes da assertão do teste de chamada callback da função `done()`, já o segundo (linha 6) adiciona 60 milissegundos de latência, o terceiro (linha 12) 120 milissegundos e o teste final (linha 18) adiciona 2015 milissegundos. Esses atrasos intencionais resultarão em diferentes tempos para o relatório do Mocha que tem a seguinte definição de velocidade para seus testes:

- Teste Lento: ≥ 75 milissegundos;
- Teste Médio: ≥ 40 milissegundos;
- Teste Rápido: < 40 milissegundos.

Agora, salve novamente os arquivos e substitua a chamada na página teste.html pela seguinte:

```
<script src="js/spec/teste-tempo.spec.js"></script>
```

O resultado deverá ser igual ao da **Figura 8**.

Veja que o Mocha destaca cada um dos processamentos e te dá uma exata noção do que aconteceu e quanto tempo levou cada

teste exatamente. Os testes em tempo “médio” levam a cor laranja/marrom enquanto os “lentos” são taxados em vermelho. Caso o teste falhe, ou seja, estoure o timeout do framework, você verá uma mensagem semelhante à impressa na última suíte. O Mocha ainda diz em qual arquivo e qual linha do código aconteceu o problema bem como qual o valor de timeout que ele considera ter sido estourado.



Figura 8. Resultado da execução do teste temporizado

Testes de falhas

O teste de temporização que vimos constitui uma das formas de se verificar falhas no código via Mocha. Duas outras falhas merecem uma demonstração no escopo dos nossos testes: assertões e falhas de exceção. Criemos um novo arquivo JS para efetuar estes testes, nomeando-o “falhas.spec.js” na mesma pasta dos anteriores. Adicione o código da **Listagem 7** ao mesmo.

Listagem 7. Código JavaScript para teste de falhas.

```
01 // Configura o Mocha para continuar depois do primeiro erro  
02 // para mostrar ambos os exemplos de falhas.  
03 mocha.bail(false);  
04 describe("Testes de falha", function() {  
05   it("deve falhar na assertão", function() {  
06     expect("oi").to.equal("tchau");  
07   });  
08   it("deve falhar na exceção não esperada", function() {  
09     throw new Error();  
10   });  
11});
```

Neste exemplo, o primeiro teste é uma assertão de falha do Chai, a qual o Mocha encapsula com a mensagem “expected ‘oi’ to equal ‘tchau’”. O segundo teste lança uma exceção não checada (*unchecked*) que o Mocha vai exibir sempre como uma stack trace completa, muito parecida com a que temos em linguagens *server side* como o Java, C#, etc. Para retestar, salve o arquivo e modifique a referência ao arquivo de script na página teste.html novamente:

```
<script src="js/spec/falhas.spec.js"></script>
```

O resultado pode ser visto na **Figura 9**. A grande vantagem de se usar esse tipo de ferramenta é que conseguimos saber exatamente o que aconteceu e onde aconteceu, com uma linguagem natural e altamente inteligível.

```

AssertionError: expected 'oi' to equal 'tchau'
  at Context.<anonymous> (js/spec/falhas.spec.js:6:25)
  at callFn (js/lib/mocha.js:4562:21)
  at Test.Runnable.run (js/lib/mocha.js:4555:7)
  at Runner.runTest (js/lib/mocha.js:4974:10)
  at js/lib/mocha.js:4857:12
  at next (js/lib/mocha.js:4899:14)
  at js/lib/mocha.js:4909:7
  at next (js/lib/mocha.js:4844:23)
  at js/lib/mocha.js:4876:5
  at timeslice (js/lib/mocha.js:6483:27)

Error
  at Context.<anonymous> (js/spec/falhas.spec.js:9:15)
  at callFn (js/lib/mocha.js:4562:21)
  at Test.Runnable.run (js/lib/mocha.js:4555:7)
  at Runner.runTest (js/lib/mocha.js:4974:10)
  at js/lib/mocha.js:4857:12
  at next (js/lib/mocha.js:4899:14)
  at js/lib/mocha.js:4909:7
  at next (js/lib/mocha.js:4844:23)
  at js/lib/mocha.js:4876:5
  at timeslice (js/lib/mocha.js:6483:27)

```

Figura 9. Resultado da execução do teste de falhas

Configurando o Backbone.js

O Backbone.js é um framework que fornece abstrações e funcionalidades muito úteis para arquitetar e desenvolver aplicações web usando JavaScript. Ele traz ordem para as interações caóticas entre o programa e a lógica de exibição, eventos DOM, e comunicação back-end, principalmente por se basear nos conceitos do MVC.

Dentre o conjunto de componentes da biblioteca, podemos citar:

- **Eventos:** O módulo **Backbone.Events** dá aos objetos JavaScript a habilidade de se comunicar com eventos, incluindo as classes de eventos do próprio Backbone.js assim como os eventos de aplicação customizados.
- **Modelos:** O módulo **Backbone.Model** provê um encapsulador de dados que pode sincronizar com o back-end, validar mudanças nos dados, e emitir eventos para outras partes da aplicação. Um model é a unidade de dados fundamental em uma aplicação Backbone.js.
- **Coleções:** A classe **Backbone.Collection** encapsula um conjunto de modelos em uma lista ordenada. As coleções fornecem eventos, sincronização com o back-end, e muitos métodos utilitários para a manipulação e mutação do conjunto de modelos usados.
- **Templates:** O Backbone.js deixa a escolha da biblioteca de templates para o desenvolvedor (Underscore.js, Handlebars, EJS, etc.).
- **Views:** Um objeto **Backbone.View** funciona como um conector entre modelos, coleções, e templates juntos com o ambiente do browser e o DOM. O Backbone.js é cego ao que a visão deve fazer, mas uma view típica referencia uma coleção ou um modelo, bem como mede a interação entre o usuário e os eventos de back-end do servidor.
- **Routers:** Os programas Backbone.js são comumente desenvolvidos como aplicações single-page onde todo o código da página HTML e bibliotecas JavaScript são baixados em um simples load.

A classe **Backbone.Router** mantém um estado interno da aplicação e gerencia o histórico do browser.

Para efetuar os nossos testes com o Backbone.js, uma vez que o desenvolvimento da aplicação não é o foco deste artigo, disponibilizamos para download, junto do pacote de código fonte deste, uma aplicação simples de cadastro de tarefas toda feita com o framework para agilizar o processo. É pré-requisito que o Backbone.js esteja instalado e, conforme falado antes, não iremos nos ater a demonstrar como fazer isso ou quaisquer configurações no framework que não se refiram ao universo de testes.

A aplicação pode ser encontrada na pasta app-tarefas do mesmo pacote de download. Abra-a e no diretório tarefas/app você encontrará a página index.html, execute-a e a aparência da tela deverá ser semelhante à da **Figura 10**.



Figura 10. Tela inicial do aplicativo Backbone.js de cadastro de tarefas

As ações padrão disponibilizadas para cada opção são adicionar uma nova tarefa, editar e remover as existentes. Além disso, ao clicar em um item em específico você será redirecionado para a tela de detalhes da tarefa, na qual podemos efetuar as mesmas mudanças (**Figuras 11** e **12**).

Faça alguns testes e cadastre algumas tarefas você mesmo. Além disso, abra os arquivos de código HTML, JavaScript e CSS do referido projeto, assim poderá ter uma noção melhor de como eles foram estruturados.

Junto com o mesmo arquivo de download, foi junto a pasta referente aos testes do projeto em sua completude. Mas não a use ainda. Em vez disso, reposicione-a em um outro diretório e replique a mesma estrutura de diretórios, porém vazios, no que se refere ao caminho relativo /test, mantendo a pasta js e suas subpastas. Crie um novo arquivo index.html e adicione a estrutura inicial contida na **Listagem 8**.

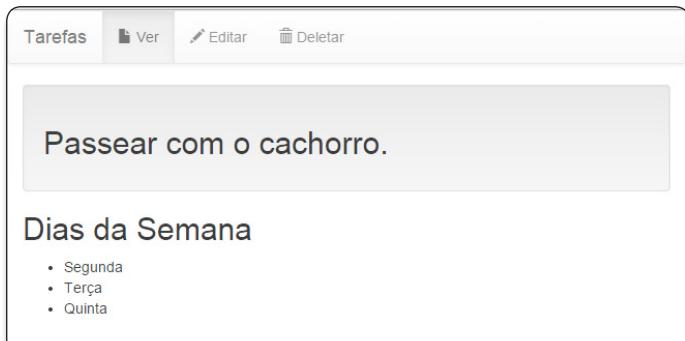
Perceba que essa listagem não traz tantas novidades, uma vez que configura novamente os arquivos para as bibliotecas do Chai, Mocha, Sinon.JS e agora para o Backbone.js, Underscore.js e Bootstrap. A função de script apresentada a partir da linha 35 configura mais uma vez os objetos dos referidos frameworks. Por fim, as importações de script das linhas 46 e 47 definem os arquivos de spec com os testes que criaremos.

Antes de testarmos, precisamos configurar os arquivos JavaScript significativos. O primeiro deles é o arquivo **namespace.spec.js**, visto que tanto o **namespace.js** quanto o **tarefa.js** já vieram preenchidos na pasta **app**. Veja na **Listagem 9** o código que precisamos adicionar a ele.

Backbone.js e PhantomJS: Criando testes web em front-end

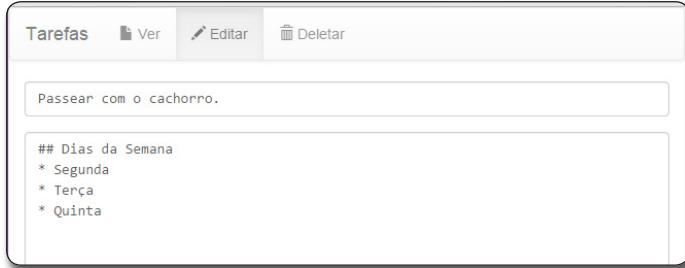
Listagem 8. Código inicial da página index.html de teste do projeto Backbone.js

```
01 <html>
02 <meta charset="utf-8">
03 <head>
04 <title>Testes com Backbone.js</title>
05 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
06 <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
07 <link rel="stylesheet" href="js/lib/mocha.css" />
08 </head>
09 <body>
10 <div id="mocha"></div>
11
12 <div id="fixtures" style="display: none; visibility: hidden;"></div>
13
14 <!-- Libs JavaScript Test -->
15 <script src="js/lib/mocha.js"></script>
16 <script src="js/lib/chai.js"></script>
17 <script src="js/lib/sinon-chai.js"></script>
18 <script src="js/lib/sinon.js"></script>
19
20 <!-- Libs JavaScript Core -->
21 <script src=".app/js/lib/underscore.js"></script>
22 <script src=".app/js/lib/jquery.js"></script>
23 <script src=".app/js/lib/json2.js"></script>
24 <script src=".app/js/lib/backbone.js"></script>
25 <script src=".app/js/lib/backbone.localStorage.js"></script>
26 <script src=".app/js/lib/bootstrap/js/bootstrap.js"></script>
27 <script src=".app/js/lib/showdown/showdown.js"></script>
28
29 <!-- Libs JavaScript Application -->
30 <script src=".app/js/app/namespace.js"></script>
31 <script src=".app/js/app/models/tarefa.js"></script>
32
33 <!-- Configura o Mocha e o Chai -->
34 <script>
35 var expect = chai.expect;
36 mocha.setup({
37   ui: "bdd",
38   bail: false
39 });
40
41 window.onload = function () {
42   (window.mochaPhantomJS || mocha).run();
43 };
44 </script>
45
46 <script src="js/spec/namespace.spec.js"></script>
47 <script src="js/spec/models/tarefa.spec.js"></script>
48 </body>
49 </html>
```



The screenshot shows a Backbone.js application interface. At the top, there's a navigation bar with tabs for 'Tarefas', 'Ver', 'Editar', and 'Deletar'. Below the navigation, a task card displays the text 'Passar com o cachorro.'. To the right, a sidebar titled 'Dias da Semana' lists the days: Segunda, Terça, and Quinta.

Figura 11. Tela de detalhamento do aplicativo Backbone.js de tarefas



The screenshot shows the Backbone.js application in edit mode for the task 'Passar com o cachorro.'. The task details are visible in the main area, and the sidebar with 'Dias da Semana' is still present.

Figura 12. Tela de edição do aplicativo Backbone.js de tarefas

Este arquivo basicamente se encarrega de verificar se os componentes do Backbone foram corretamente carregados. Para isso, criamos duas suítes de teste: a primeira (linha 2) verifica se a classe App que foi configurada no respectivo arquivo namespace.js do projeto app foi carregada corretamente, enquanto a segunda (linha 13) verifica o objeto instanciado via referida classe. Ademais, é importante salientar que mesmo as suítes estando em arquivos diferentes isso não modifica o resultado final, que irá executar na ordem de importação definida no index.html.

Listagem 9. Código do arquivo namespace.spec.js.

```
01 describe("Namespace", function () {
02   it("fornece o objeto 'App'", function () {
03     // A função expect existe e é um objeto.
04     expect(App).to.be.an("object");
05
06     // Espera que todas as propriedades estejam presentes.
07     expect(App).to.include.keys(
08       "Config", "Collections", "Models",
09       "Routers", "Templates", "Views"
10     );
11   });
12
13   it("fornece o objeto 'app'", function () {
14     expect(app).to.be.an("object");
15   });
16});
```

Em seguida, devemos configurar também o arquivo **tarefa.spec.js** em alusão à sua referência no projeto app. Crie esse arquivo e adicione o conteúdo da **Listagem 10** ao mesmo.

Esse arquivo, por sua vez, é mais completo. Dentre os pontos importantes podemos destacar:

- Na linha 4 nós instanciamos um novo objeto (que chamamos de mock) para simular as características e ações do objeto real declarado no projeto app. Perceba que o objeto foi criado vazio, sem nenhum valor atrelado ao mesmo, da mesma forma que o teste que foi feito em seguida.
- O objeto model que representa a tarefa de teste verificará se o mesmo está ok (linha 6), em seguida, testará várias condições como o título, o texto do campo, e a data. Se tudo estiver ok, a suíte é aprovada no teste.
- A partir da linha 12, temos a configuração da suíte que valida valores de teste, tais como os que criamos no exemplo citado.

Esperamos encontrar os mesmos valores que foram inseridos no item de tarefa, caso contrário um erro irá estourar.

Salve os arquivos e execute a página test.html no browser, o resultado pode ser visualizado na **Figura 13**.

Listagem 10. Código do arquivo tarefa.spec.js.

```
01 describe("App.Models.Tarefa", function () {
02   it("tem valores padrão", function () {
03     // Cria um modelo vazio de tarefas
04     var model = new App.Models.Tarefa();
05
06     expect(model).to.be.ok;
07     expect(model.get("title")).to.equal("");
08     expect(model.get("text")).to.equal("**Editar sua tarefa!");
09     expect(model.get("createdAt")).to.be.a("Date");
10   });
11
12   it("configura os atributos passados", function () {
13     var model = new App.Models.Tarefa({
14       title:"Passear com o cachorro",
15       text:"## Dias da Semana* Segunda* Terça* Quinta"
16     });
17
18     expect(model.get("title")).to.equal("Passear com o cachorro");
19     expect(model.get("text")).to.equal("## Dias da Semana* Segunda* Terça* Quinta");
20   });
21});
```



Figura 13. Resultado da execução dos testes no app Backbone.js

Para cada tipo de funcionalidade diferente na camada de visão da nossa aplicação Backbone.js nós devemos criar um conjunto de suítes de teste que as teste. Dentro da pasta test/js/spec/views vamos criar um exemplo para validar isso, especificamente no que se refere à criação das tarefas. Veja na **Listagem 11** o código que deve ser inserido no arquivo tarefa-view.spec.js.

Vejamos algumas observações sobre a mesma:

- Logo na linha 2 começamos a implementação com uma nova função: `before()`. Essa função, assim como suas variantes (`beforeEach()`, `after()`, `afterEach()`, etc.) serve para gerenciar o estado do teste, executando antes ou depois de cada teste dentro da suíte. Na função da referida linha, criamos uma nova div no HTML para conter a divisão entre cada tarefa em específico.
- Na linha 7 usamos a função `beforeEach()` que será chamada antes de cada teste executar, para limpar a div de fixação e instanciar o objeto de TarefaView que será usado.
- Na linha 17 temos a função `afterEach()` que destrói o modelo de visão há cada nova chamada de teste. E, em seguida, na linha 22,

Listagem 11. Código do arquivo tarefa-view.spec.js.

```
01 describe("App.Views.TarefaView", function () {
02   before(function () {
03     // Cria a fixação.
04     this.$fixture = $("<div id='note-view-fixture'></div>");
05   });
06
07   beforeEach(function () {
08     // Esvazia e religa a fixação para cada execução.
09     this.$fixture.empty().appendTo($("#fixtures"));
10
11     this.view = new App.Views.TarefaView({
12       el: this.$fixture,
13       model: new App.Models.Note()
14     });
15   });
16
17   afterEach(function () {
18     // Destruir o modelo também destrói a view.
19     this.view.model.destroy();
20   });
21
22   after(function () {
23     // Remova todas as sub-fixações após a conclusão da suíte de testes.
24     $("#fixtures").empty();
25   });
26
27   it("pode renderizar uma tarefa vazia", function () {
28     var $title = $("#pane-title"),
29         $text = $("#pane-text");
30
31     expect($title.text()).to.equal("");
32     expect($title.prop("tagName")).to.match(/h2/i);
33
34     expect($text.text()).to.equal("Edite sua tarefa!");
35     expect($text.html()).to.equal("<p><em>Edite sua tarefa!</em></p>");
36   });
37
38   it("pode renderizar markdowns mais complicados", function (done) {
39     this.view.model.once("change", function () {
40       var $title = $("#pane-title"),
41           $text = $("#pane-text");
42
43       // Nosso novo title.
44       expect($title.text()).to.equal("Meu Título");
45
46       expect($text.html())
47         .to.contain("Meu Cabeçalho</h2>").and
48         .to.contain("<ul>").and
49         .to.contain("<li>List item 2</li>");
50
51       done();
52     });
53
54     // Faz a nossa tarefa um pouco mais complexa.
55     this.view.model.set({
56       title: "Meu Título",
57       text: "** Meu Cabeçalho\n" +
58             "* List item 1\n" +
59             "* List item 2"
60     });
61   });
62});
```

Backbone.js e PhantomJS: Criando testes web em front-end

a função `after()` que se encarrega de esvaziar a div de fixação para a chamada subsequente.

- Depois, seguindo o mesmo modelo do último teste, temos as duas suítes de teste (começando nas linhas 27 e 38, respectivamente) que farão o teste com o objeto vazio, e depois preenchido com valores mockados, respectivamente.
- Finalmente, na linha 55 temos o uso da função `set()` que irá configurar os dados de teste para o referido objeto de mock.

Após isso, não esqueça de adicionar também a importação dos arquivos tanto do projeto app quanto do projeto test na página HTML:

```
<script src="../app/js/app/views/tarefa-view.js"></script>
```

O resultado da execução irá retornar o mesmo que nos relatórios anteriores para a referida suíte.

O universo de fabricação de testes (de todos os tipos: unitários, de integração, caixa preta/branca, etc.) constitui uma gama de assuntos muito grande para ser completamente abraçada por um artigo só. Em vista disso, é preciso que o leitor esteja munido, inicialmente, de todos os conhecimentos relacionados ao paradigma e às metodologias do teste em si. Somente a partir desse entendimento, é que você estará apto a se aprofundar nas inúmeras ferremantes do mercado. Além das que utilizamos aqui, podemos citar ferramentas como o Jasmine, QUnit do jQuery, JTestDriver, dentre muitas outras.

Cada ferramenta, por sua vez, adiciona um conjunto de funcionalidades, muitas delas com assinatura própria da fabricante, que exigem experiência de quem as usa, quando precisam ser adicionadas a um projeto, principalmente se estiverem combinadas umas com as outras.

Autor



Sueila Sousa

É tester e entusiasta de tecnologias front-end. Atualmente trabalha como analista de testes na empresa Indra, com foco em projetos de desenvolvimento de sistemas web, totalmente baseados em JavaScript e afins. Possui conhecimentos e experiências em áreas como Gerenciamento de processos, banco de dados, além do interesse por tecnologias relacionadas ao desenvolvimento e teste client side.



Links:

Página de download oficial do PhantomJS.

<http://phantomjs.org/download.html>

Página do projeto Mocha no Github.

<https://github.com/mochajs/mocha>

Página oficial do Chai.js.

<http://chaijs.com/>

Planilha oficial do Sinon.JS.

<http://sinonjs.org/>

Conhecimento faz diferença!

Agilidade: Negociação de contratos em projeto
Processo: Medição de Software: Um importante instrumento para a gestão
Gerência de Configuração: Definição + Ferramentas
Engenharia de Software: Edição 28 :: Ano 2
Evolução do Software: Definições, preocupações e custo
Automação de Testes: Cuidados a serem tomados na implantação
Aulas desta edição: Estratégia de Teste Funcional baseada em Casos de Uso – Partes 5 a 9
+ de 290 vídeos para assinantes

Faça já sua assinatura digital! | www.devmedia.com.br/es

Faça um upgrade em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**



DEV MEDIA

Simulações artificiais com JavaScript

O conceito de Inteligência Artificial é comumente definido como a inteligência exibida por máquinas ou softwares que se equipara ao comportamento racional humano. Baseada em conceitos como dedução, raciocínio e resolução automática de problemas, a IA e seus conceitos nunca foram tão buscados, explorados e utilizados pela humanidade como hoje. Alvo de inúmeras críticas e ceticismo por uma parte das pessoas que a estuda, a IA hoje está presente em jogos (RPGs principalmente, em vista da necessidade de ter personagens que evoluam e desenvolvam habilidades em detrimento dos aprendizados e experiências que tiveram), sistemas de animação gráfica, dentre outros.

Quando se trata de implementar IA em sistemas usando uma linguagem de programação comum (como Java, C# ou JavaScript) muitos desenvolvedores desconhecem os passos simples que podem levar a isso, ou até mesmo consideram que a inclusão de bibliotecas avançadas de scripts se faz necessária para lidar com isso.

Este artigo terá como objetivo a construção de um ecossistema virtual, um pequeno mundo que será populado dinamicamente e artificialmente por criaturas que se movimentam e lutam entre si por sobrevivência. Será basicamente uma implementação de um componente inteligente que pode ser usado em cenários reais de jogos, animações gráficas, dentre outros que necessitem que determinados personagens assumam características de inteligência artificial.

Cenário

O cenário de desenvolvimento se restringe a radicalmente simplificar o conceito de mundo simulado. Em teoria, um mundo seria uma grid bidimensional onde cada entidade ocupa uma célula inteira do quadrado, imaginando que esse plano fosse representado por um quadrado de dimensões $\{x, y\}$ que seria dividido em células – menores partes possíveis – e, por sua vez, teriam dimensões $\{x/a, y/b\}$, onde a e b representam os graus de divisão máximos para os eixos horizontais e verticais do plano cartesiano.

Vamos chamar as entidades de criaturas. Há cada rodada, as criaturas recebem uma nova chance de executar alguma ação. Assim, cortamos ambos tempo e espaço em unidades com tamanho fixo: quadrados

Fique por dentro

A inteligência artificial é uma área de muita necessidade, principalmente no universo de jogos quando os mesmos precisam ter personagens inteligentes que evoluam através das experiências no próprio jogo. Esse tipo de inteligência é traduzido nas linguagens de programação por meio de técnicas que permitam aos algoritmos salvar uma grande quantidade de dados e reusá-los para aprimorar o comportamento dos objetos.

Neste artigo trataremos de implementar uma simulação de mundo inteligente, onde vivem criaturas que, por si sós, se alimentam, reproduzem, andam e morrem. Os códigos apresentados também poderão ser usados para implementações com jogos (RPGs, arcade, estilo fazenda, etc.), animações gráficas diversas, dentre outras.

para o espaço e bolas para o tempo. Obviamente, essa não é uma aproximação real e apurada de como isso funciona em algoritmos mais complexos do tipo, mas para os propósitos deste artigo são mais que o suficiente, além de estabelecerem um paralelo mais didático com o desenvolvedor.

Em suma, iremos definir um objeto mundo com um plano, um vetor de strings que cria a grid do mesmo usando um caractere por quadrado. Veja na [Listagem 1](#) o código que representa a criação da referida variável vetor que usaremos para representar este plano.

Perceba que a disposição dos elementos na tela se assemelha à que temos em telas de jogos que geram planos de fundos aleatórios como Pack Man, Super Mário, dentro outros do estilo.

Listagem 1. Representação inicial do modelo de plano bidimensional.

No modelo, os caracteres “@” representam as paredes e pedras geradas no meio do cenário, e o caractere “()” representa as criaturas (quatro, no total). Os espaços são simplesmente espaços comuns como você deve ter percebido. O leitor deve, inclusive, tomar cuidado se for usar essa estrutura em páginas HTML, uma vez que espaços em branco não são reconhecidos pelo browser, portanto devendo usar os caracteres de espaço respectivos.

Esse array pode ser usado para criar o mundo “plano” em 2D que imaginamos. Tal objeto contém informações referentes ao tamanho e conteúdo do nosso mundo. Além disso, o objeto contém um método `toString()` que converte a representação matricial do mesmo em uma string que pode ser impressa, e que nos permite analisar o que está acontecendo em caso de querermos debugar o código. O objeto também tem uma função chamada `turn()`, que permite aos objetos de criatura atualizar (de dentro dele) por uma rodada e atualizar o objeto mundo como um todo, refletindo assim suas ações.

Representando o espaço

A grid que modela o objeto mundo tem largura e altura fixas. Os quadrados são identificados por suas respectivas coordenadas x e y. Para representar o espaço, criaremos uma classe que simulará basicamente o salvamento das duas informações dos eixos base, e a chamaremos de Vector. Certifique-se de criar a estrutura representada na **Listagem 2** dentro do nosso arquivo global.js.

A única novidade nessa listagem compreende o uso dos prototypes no JavaScript, que são estruturas que promovem a herança da Orientação a Objetos por intermédio da criação de classes via funções. Perceba que na linha 6 simplesmente criamos mais um objeto desse tipo, com as informações do objeto Vector recebido como parâmetro, ou seja, essa definição de função basicamente cria cópias do objeto recebido.

Listagem 2. Código de criação do objeto Vector no nosso modelo.

```
01 function Vector(x, y) {
02   this.x = x;
03   this.y = y;
04 }
05 Vector.prototype.maisUm = function(outro) {
06   return new Vector(this.x + outro.x, this.y + outro.y);
07};
```

Nota

Esse tipo de abordagem é muito usado por ferramentas de construção gráfica e/ou jogos como o Unity 3D e o Cocoatouch. Eles usam um objeto Vector com o mesmo conceito para manipular o universo bidimensional. Caso desejássemos migrar para um universo tridimensional, bastaria incluir uma nova propriedade à classe referente ao plano excedente.

Agora precisamos criar um tipo de objeto que molde a grid por si só. Uma grid é parte da representação física do nosso objeto mundo, mas iremos implementá-la em um objeto separado (que será uma propriedade do objeto mundo, por meio de uma

composição) para manter o objeto mundo simples, aplicando assim a divisão de responsabilidades e desacoplamento de código.

Para salvar os valores da grid, temos algumas opções:

1. Podemos usar um vetor de linhas de vetores e usar duas propriedades para acessar um quadrado em específico, algo mais ou menos assim:

```
var grid = [[“superior esquerda”, “superior meio”, “superior direita”],
[“inferior esquerda”, “inferior meio”, “inferior direita”]];
console.log(grid[1][2]);
```

O resultado da execução imprimiria o valor “inferior direita” no Console.

2. Ou podemos usar um vetor simples, com tamanho definido na forma de largura x altura, e decidir qual elemento em {x, y} é encontrado na posição x + (y * largura) do vetor:

```
var grid = [“superior esquerda”, “superior meio”, “superior direita”]
[“inferior esquerda”, “inferior meio”, “inferior direita”];
console.log(grid[2 + (1 * 3)]);
```

O resultado da execução imprimiria o valor “inferior direita” no Console.

Uma vez que o acesso a este vetor estará encapsulado em métodos do objeto do tipo grid, não importa para o código externo que abordagem escolheremos. Vamos usar a segunda opção porque ela facilita muito o processo de criação do vetor. Quando chamarmos o construtor de Array com um simples número como argumento, ele criará um novo array vazio com o tamanho fornecido. Para isso, vamos criar então o código que define o objeto Grid, somente com alguns métodos básicos. Crie um novo arquivo grid.js e adicione o conteúdo da **Listagem 3** ao mesmo.

Listagem 3. Código de criação do objeto Grid no nosso modelo.

```
01 function Grid(largura, altura) {
02   this.espaco = new Array(largura * altura);
03   this.largura = largura;
04   this.altura = altura;
05 }
06 Grid.prototype.isDentro = function(vetor) {
07   return vetor.x >= 0 && vetor.x < this.largura &&
08     vetor.y >= 0 && vetor.y < this.altura;
09 };
10 Grid.prototype.get = function(vetor) {
11   return this.espaco[vetor.x + this.largura * vetor.y];
12 };
13 Grid.prototype.set = function(vetor, value) {
14   this.espaco[vetor.x + this.largura * vetor.y] = value;
15 };
```

Perceba que a semelhança nos construtores de Grid e Vector se definem ao recebimento das dimensões, neste caso a altura e largura. Logo na linha 2 estamos criando um novo objeto Array (classe interna da própria biblioteca do JavaScript) passando como parâmetro o valor calculado da área total da grid (largura x altura). Após isso, definimos três novos métodos para esse objeto:

- **isDentro**: recebe o objeto Vector como parâmetro e verifica se a grid está dentro das dimensões do vetor.
- **get**: recebe o Vector por parâmetro e retorna a posição exata no Array de espaço.
- **set**: recebe o Vector e o novo valor a ser configurado naquela posição x do Array de espaço.

Para testar essas configurações iniciais, podemos criar uma página HTML de testes e dentro dela importar os nossos arquivos JavaScript, além de criar uma tag de script básica para efetuar as chamadas e imprimir no Console do browser. Para isso, crie um novo arquivo chamado “teste.html” dentro da estrutura de diretórios que definimos para os projetos de teste, e adicione o conteúdo da **Listagem 4** ao mesmo.

Listagem 4. Código de criação da página de testes.

```
01 <html>
02   <head>
03     <title>
04       Página de testes: Simulações artificiais
05     </title>
06     <script type="text/javascript" src="global.js"></script>
07     <script type="text/javascript" src="grid.js"></script>
08     <script type="text/javascript">
09       var grid = new Grid(5, 5);
10       console.log(grid.get(new Vector(1, 1)));
11
12       grid.set(new Vector(1, 1), " X");
13       console.log(grid.get(new Vector(1, 1)));
14     </script>
15   </head>
16
17   <body>
18
19   </body>
20 </html>
```

Perceba que no teste (bem básico, sem a inclusão de quaisquer bibliotecas de teste unitários JavaScript) estamos apenas verificando ambas as situações, em detrimento do código já produzido nas listagens anteriores. Perceba também que as chamadas aos arquivos de código JavaScript (linhas 6 e 7) estão apontando para o mesmo diretório onde se encontra o arquivo HTML, mas você

pode ficar à vontade para decidir onde quer colocá-los, desde que se lembre de modificar as referências aos caminhos relativos.

Agora é só executar a página no browser de sua preferência e abrir o Console de logs do mesmo. A maioria deles está disponível através do atalho de teclado F12, mas é possível consultar a documentação oficial do mesmo caso tenha dúvidas de como encontrar. Neste artigo usaremos o Google Chrome, por razões de simplicidade e usabilidade. Ao acessar a opção, verá logo de cara mensagens semelhantes às impressas na **Figura 1**. Os textos “undefined” (que será impresso em virtude da ausência de um objeto em memória na posição estabelecida pela linha 10) e “X” (que será exibido em função da existência desse texto no modelo que criamos de mundo, por causa do valor referenciado no objeto Vector da linha 13) são os resultados finais do teste.

Programando a interface das criaturas

Antes de começarmos no construtor da classe Mundo, devemos entender melhor como os objetos de criatura irão se comportar dentro desse mundo. O objeto mundo irá solicitar aos objetos criaturas que tipo de ações eles irão executar, funcionando da seguinte forma: cada objeto criatura tem o seu próprio método **agir()** que, quando chamado, retorna uma ação. Esta é um objeto com uma propriedade **tipo**, que nomeia o tipo da ação que a criatura quer fazer, por exemplo “mover”. A ação também poderá conter informações extra, tais como a direção que a criatura quer se mover (x, y), etc.

As criaturas, por sua vez, só poderão enxergar os quadrados diretamente ao redor delas na grid. Mas até essa “visão” limitada pode ser útil quando tiver de decidir que ação tomar. Quando o método **agir()** for chamado, ele receberá um objeto de visão (**view**) que permitirá à criatura inspecionar a sua volta. Vamos nomear os oito quadrados que estão em volta dela de “direções da bússola”: “n” para norte, “ne” para nordeste, e assim por diante. Veja na **Listagem 5** o objeto que usaremos para mapear desde os nomes das coordenadas até as configurações da nossa bússola imaginária.

Na listagem, cada coordenada geográfica é representada por um objeto Vector que, por sua vez, está associado a uma String que o identifica (“n”, “ne”, “l”, etc.). O objeto view terá um método **observar()**, que toma uma direção e retorna um caractere, por exemplo “@” quando existe uma parede naquela direção, ou “ ” (espaço em branco) quando não há nada na referida direção. O objeto também fornece os métodos utilitários **find** e **findAll**, herdados da herança implícita. Ambos recebem um mapa de caracteres como argumento. O primeiro retorna à direção em que o caractere pode ser encontrado próximo à criatura ou retorna null se não encontrar nada. O segundo retorna um array contendo todas as direções com aquele caractere. Por exemplo, uma criatura localizada à direita (oeste) da parede receberá “[“ne”, “l”, “sl”] quando

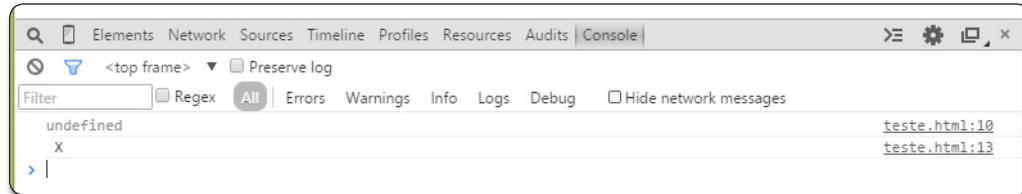


Figura 1. Resultado dos testes sobre o objeto mundo

chamar o método `findAll` no seu objeto `view` com o caractere “\@” como argumento.

Vejamos na **Listagem 6** um exemplo bem simples de uma criatura não inteligente que segue seus próprios passos até que ela atinja um obstáculo e, em seguida, salta para fora em uma direção aleatória.

Listagem 5. Objeto responsável por guardar os valores de direção.

```
01 var directions = {  
02   "n": new Vector(0, -1), // Norte  
03   "ne": new Vector(1, -1), // Nordeste  
04   "l": new Vector(1, 0), // Leste  
05   "s": new Vector(1, 1), // Sudeste  
06   "s": new Vector(0, 1), // Sul  
07   "so": new Vector(-1, 1), // Sudoeste  
08   "o": new Vector(-1, 0), // Oeste  
09   "no": new Vector(-1, -1) // Noroeste  
10};
```

Listagem 6. Objeto responsável por guardar os valores de direção.

```
01 function elementoRandomico(array) {  
02   return array[Math.floor(Math.random() * array.length)];  
03}  
04  
05 var direcoes = "n ne l s l s so o no".split(" ");  
06  
07 function RedondezasCriatura() {  
08   this.direcao = elementoRandomico(direcoes);  
09};  
10 RedondezasCriatura.prototype.agir = function(view) {  
11   if (view.observar(this.direcao) != "")  
12     this.direcao = view.find("") || "s";  
13   return {type: "mover", direcao: this.direcao};  
14};
```

Observe que a função `elementoRandomico()` pega um número aleatório do array e, usando a função da API Math do próprio JavaScript (linha 2) mais algumas operações aritméticas, o transforma em um índice randômico. Usaremos isso novamente mais à frente porque aleatoriedade pode ser muito útil em simulações.

Para escolher uma direção aleatória, o construtor de `RedondezasCriatura` chama a função `elementoRandomico()` em uma matriz de nomes de direção. Também poderíamos ter usado o vetor `Object.keys` para obter essa matriz do objeto “`direcoes`” que definimos anteriormente, mas isso não nos fornece nenhuma garantia sobre a ordem em que as propriedades são listadas. Na maioria das situações, os motores modernos JavaScript retornarão as propriedades na ordem em que foram definidas, mas eles não são necessários para essa finalidade.

O “|| “s”” no método `agir()` está lá para impedir que `this.direcao` receba o valor null se a criatura estiver, de alguma forma, presa sem nenhum espaço vazio em torno dela (por exemplo, quando estiver presa em um canto do objeto mundo e cercada por outras criaturas).

Criando o objeto Mundo

Agora podemos começar a desenvolver o principal objeto do nosso aplicativo, o objeto **Mundo**. O seu construtor usará um plano

e uma legenda como argumentos. Uma legenda é um objeto que nos diz o que cada caractere no mapa significa. Ela contém um construtor para cada caractere, exceto para o caractere de espaço, que sempre será referido como null.

Para isso crie um novo arquivo JavaScript “`mundo.js`”, com o conteúdo da **Listagem 7**.

Listagem 7. Conteúdo inicial do objeto Mundo.

```
01 function elementoPorChar(legenda, char) {  
02   if(char == "")  
03     return null;  
04   var elemento = new legenda[char]();  
05   elemento.charOrigem = char;  
06   return elemento;  
07}  
08  
09 function Mundo(mapa, legenda) {  
10  var grid = new Grid(mapa[0].length, mapa.length);  
11  this.grid = grid;  
12  this.legenda = legenda;  
13  
14  mapa.forEach(function(linha, y) {  
15    for(var x = 0; x < linha.length; x++)  
16      grid.set(new Vector(x, y), elementoPorChar(legenda, linha[x]));  
17  });  
18}  
19  
20 function charPorElemento(elemento) {  
21  if(elemento == null)  
22    return "";  
23  else  
24    return elemento.charOrigem;  
25}  
26  
27 Mundo.prototype.toString = function() {  
28  var output = "";  
29  for(var y = 0; y < this.grid.height; y++) {  
30    for(var x = 0; x < this.grid.width; x++) {  
31      var elemento = this.grid.get(new Vector(x, y));  
32      output += charPorElemento(elemento);  
33    }  
34    output += "\n";  
35  }  
36  return output;  
37};
```

Na função `elementoPorChar()` criamos primeiramente uma instância do tipo “legenda” verificando se o atributo `char` é diferente de null e aplicando o operador `new` nele. Em seguida, adicionamos a propriedade `charOrigem` a ele para tornar fácil o processo de descobrir de qual caractere o elemento foi originalmente criado.

Precisamos desta propriedade `charOrigem` ao implementar o método `toString` do objeto mundo. Este método constrói uma string do tipo mapa através do estado atual do objeto mundo, por intermédio da realização de um loop bidimensional sobre os quadrados da grid.

Observe também que a primeira verificação que fazemos na função `elementoPorChar()` é checar se o valor do `char` passado como parâmetro constitui um valor vazio, atendendo assim às exigências que definimos anteriormente.

Da mesma forma, na linha 20 temos a realização do processo inverso, que é a busca de um caractere em específico a partir de um elemento x. O uso do operador prototype (linha 27) é importante para estabelecermos um critério de herança que deverá ser obedecido por todos os objetos filhos de Mundo. Dessa forma, garantimos que, independentemente da quantidade e qualidade de objetos que forem criados em dependência hierárquica do objeto Mundo, continuaremos tendo os conceitos implementados devidamente obedecidos.

A listagem como um todo será muito importante para manter os elementos organizados dentro do objeto mundo. A base da inteligência artificial é o aprendizado contínuo, logo, quanto mais informações arquivadas pelo sistema, mais chances de sucesso o mesmo terá.

Em se tratando de um objeto **Parede**, temos uma associação simples: ele será usado somente para tomar espaço e não terá nenhum método `agir()`, diferente dos demais objetos. Veja como é simples sua representação:

```
function Parede() {}
```

Finalmente, quando testamos o objeto Mundo criando uma instância baseada no plano que definimos antes e então chamamos o método `toString` nele, teremos uma string muito similar ao plano que criamos como base. Veja na [Listagem 8](#) um possível resultado de exemplo para que você possa entender como a impressão desse objeto funcionará, uma vez que será através dela (`toString`) que poderemos verificar o estado do nosso “mundo” sempre que quisermos.

Listagem 8. Exemplo de impressão do conteúdo do objeto Mundo.

O escopo this

O construtor do objeto Mundo contém uma chamada para um `forEach`. Uma coisa interessante a notar é que dentro da função passada para o `foreach`, não estamos mais diretamente no escopo da função do construtor. Cada chamada de função recebe a sua própria ligação para o operador `this`, logo o `this` na função

interna não se refere ao objeto construído recentemente ao qual o this externo se refere. Na verdade, quando uma função não é chamada como um método, o this vai se referir ao objeto global.

Isso significa que não podemos escrever um `this.grid` para acessar a rede a partir de dentro do loop. Em vez disso, a função externa cria uma variável local normal, `grid`, através da qual a função interna tem acesso ao `grid` genérico.

Isto constitui um erro de design em JavaScript. Felizmente, a próxima versão da linguagem proporciona uma solução para este problema. Enquanto isso, existem soluções alternativas.

Um padrão comum é usar o código `var auto = this` e daí em diante referir-se sempre à variável `auto`, que é uma variável normal e, portanto, visível para funções internas.

Outra solução é a utilização do método `bind`, que nos disponibiliza um objeto “`this`” explicitamente, proporcionando a possibilidade de vincular o mesmo. Façamos um teste então. No mesmo código que criamos antes na página HTML, adicione o conteúdo da [Listagem 9](#) e execute novamente a página.

Listagem 9. Testando bind>this

```
01 var teste = {  
02   prop : 10,  
03   addPropTo : function(vetor) {  
04     return vetor.map(function(elt) {  
05       return this.prop + elt;  
06     }).bind(this);  
07   }  
08 };  
09 console.log(teste.addPropTo([4]));
```

O resultado dessa execução será [14]. Isso se deve por que a função `addPropTo` mapeia o vetor que passamos como argumento e o direciona para a função interna `vetor.map()`, que, por sua vez, adicionará o valor 10 ao resultado final.

A função passada para o mapa é o resultado da chamada ao bind e, portanto, tem o seu this vinculado ao primeiro argumento dado para o bind – o valor da função externa ao this (que detém o objeto de teste).

A maioria dos métodos de ordem superior em matrizes padrão, tais como `forEach` e `map`, tem um segundo argumento opcional que pode também ser usado para fornecer um `this` para as chamadas à função de iteração. Então, pode-se expressar o exemplo anterior de uma forma um pouco mais simples, como mostra a [Listagem 10](#).

Listagem 10. Exemplo de bind/this simplificados

```
01 var teste = {  
02   prop : 10,  
03   addPropTo : function (vetor) {  
04     return vetor.map(function(elt) {  
05       return this.prop + elt;  
06     }, this); // ← sem bind  
07   }  
08};
```

É importante lembrar que essa implementação funciona apenas para as funções de ordem superior que suportam tal parâmetro de **contexto**. Quando não o fazem, é preciso usar uma das outras abordagens.

Em nossas próprias funções de ordem superior, podemos apoiar tal parâmetro de contexto usando o método `call` para chamar a função dada como um argumento. Por exemplo, veja na **Listagem 11** um método `forEach` para o nosso tipo de grid, que chama uma função dada para cada elemento na grid que não é nulo ou indefinido.

Listagem 11. Exemplo de `forEach` para elementos não-nulos e indefinidos.

```
01 Grid.prototype.forEach = function (f, contexto) {  
02   for (var y = 0; y < this.height; y++) {  
03     for (var x = 0; x < this.width; x++) {  
04       var valor = this.espaco[y + y * this.width];  
05       if (valor != null)  
06         f.call(contexto, valor, new Vector(x, y));  
07     }  
08   }  
09};
```

Veja que dessa vez estamos fazendo chamadas explícitas ao método `call`, passando o mesmo contexto recebido (linha 6), de modo que o valor enviado à função obedece a mesma regra de iteração dupla sobre a matriz. Isso irá otimizar, em muito, o funcionamento dos nossos algoritmos.

Animando os objetos

O próximo passo é escrever o método `turn()` para o objeto de Mundo que dá às criaturas a chance de agir. Ele vai passar por cima da grid usando o método `forEach` que acabamos de definir, à procura de objetos com um método `agir()`. Quando encontrar um, a função `turn()` chama esse método para obter um objeto de ação e realizar a mesma quando for válida. Por enquanto, apenas as ações “mover” são compreendidas.

Existe um problema potencial com esta abordagem. Se deixamos as criaturas se moverem à medida que nos deparamos com elas, elas podem se mover para um quadrado que não olhamos ainda, e iremos permitir que elas se movam novamente quando chegarmos a esse quadrado. Assim, temos que manter um array de criaturas que já tiveram sua vez e ignorá-las quando as vermos novamente.

Para isso, adicione o conteúdo da **Listagem 12** ao nosso arquivo `mundo.js`.

Listagem 12. Criando a função `turn()`.

```
01 Mundo.prototype.turn = function() {  
02   var agiu = [];  
03   this.grid.forEach(function(criatura, vector) {  
04     if (criatura.agir && agiu.indexOf(criatura) == -1) {  
05       agiu.push(criatura);  
06       this.letAct(criatura, vector);  
07     }  
08   }, this);  
09};
```

Perceba que nesta listagem temos várias referências a estruturas nativas do JavaScript. A função `push()` usada na linha 5 é célebre dos objetos de vetor/matriz na linguagem e serve para adicionar um novo elemento. Essa estrutura se assemelha à dos métodos `add()` de linguagens como Java, C#, uma vez que define um escopo dinâmico de preenchimento dos vetores, exatamente o que queremos aqui. Veja que na linha 2 criamos um vetor vazio sem definir o seu tamanho limite, o que será útil pois não sabemos que tamanho é esse, já que ele não deverá ter um limite. O JavaScript permite esse tipo de implementação com arrays e faremos amplo uso dela ao longo do artigo.

Além disso, usamos o segundo parâmetro para que o método `forEach` da grid pudesse acessar o `this` correto que está dentro da função interna. O método `letAct` contém a lógica real que permite que as criaturas se movam. Para que tudo funcione, adicionemos também a criação dessa função ao arquivo de `mundo.js`, conforme demonstrado na **Listagem 13**.

Listagem 13. Criando a função `letAct()` para habilitar a ação das criaturas.

```
01 Mundo.prototype.letAct = function(criatura, vector) {  
02   var acao = criatura.agir(new View(this, vector));  
03   if(acao && acao.type == "mover") {  
04     var destino = this.checarDestino(acao, vector);  
05     if(destino && this.grid.get(destino) == null) {  
06       this.grid.set(vector, null);  
07       this.grid.set(destino, criatura);  
08     }  
09   }  
10 };  
11  
12 Mundo.prototype.checarDestino = function(acao, vector) {  
13   if(direcoes.hasOwnProperty(acao.direcao)) {  
14     var destino = vector.plus(direcoes[acao.direcao]);  
15     if(this.grid.isDentro(destino))  
16       return destino;  
17   }  
18};
```

Em primeiro lugar, simplesmente pedimos à criatura para agir, passando para ela um objeto de view que sabe tudo sobre o objeto mundo e sobre a posição atual dela no mesmo. O método `agir()` retorna uma ação de algum tipo.

Se o tipo de ação não for “mover”, ela será ignorada. Caso contrário, se ela tiver a propriedade “direcao” se referindo a uma direção válida, e se o quadrado daquela direção estiver vazio (`null`), então criamos um novo quadrado exatamente onde a criatura mantinha o seu valor como `null`. No final só precisamos armazenar a criatura no quadrado de destino.

Note que a função `letAct` cuida de ignorar entradas sem sentido (ela não assume que a propriedade de direção da ação seja válida ou que a propriedade de tipo faça sentido). Este tipo de programação defensiva faz sentido em algumas situações. A principal razão para fazê-la é quando desejamos validar entradas vindas a partir de fontes que você não controla (como usuário ou arquivos de entrada), mas pode também ser útil para isolar subsistemas uns dos outros. Neste caso, a intenção é que as criaturas possam ser

Listagem 12. Criando a função `turn()`.

```
01 Mundo.prototype.turn = function() {  
02   var agiu = [];  
03   this.grid.forEach(function(criatura, vector) {  
04     if (criatura.agir && agiu.indexOf(criatura) == -1) {  
05       agiu.push(criatura);  
06       this.letAct(criatura, vector);  
07     }  
08   }, this);  
09};
```

programadas por elas mesmas, ou seja, elas não têm de verificar se as suas ações fazem sentido. Elas podem apenas solicitar uma ação, e o mundo vai verificar se a permite.

Estes dois métodos não fazem parte da interface externa de um objeto Mundo. Eles são um detalhe interno. Algumas linguagens fornecem maneiras para declarar explicitamente certos métodos e propriedades particulares e sinalizar um erro quando você tentar usá-los de fora do objeto. O JavaScript não faz isso, então terá que confiar em alguma outra forma de comunicação ao descrever o que faz parte da interface de um objeto. Às vezes, pode ajudar usar um esquema de nomeação para distinguir entre propriedades externas e internas, por exemplo, ao prefixar todas as propriedades internas com um sublinhado (_). Isso fará com que os usos acidentais de propriedades que não sejam parte da interface de um objeto sejam mais fáceis de detectar.

A única parte que falta, o tipo da View, está descrita na [Listagem 14](#). Adicione o referido código em um novo arquivo chamado view.js no mesmo diretório dos demais.

Listagem 14. Criando a função letAct() para habilitar a ação das criaturas.

```
01 function View(mundo, vector) {
02   this.mundo = mundo;
03   this.vector = vector;
04 }
05
06 View.prototype.observer = function(direcao) {
07   var alvo = this.vector.plus(direcoes[direcao]);
08   if(this.mundo.grid.isDentro(alvo))
09     return charPorElemento(this.mundo.grid.get(alvo));
10   else
11     return "@";
12 };
13
14 View.prototype.findAll = function(char) {
15   var encontrados = [];
16   for(var direcao in direcoes)
17     if(this.observer(direcao) == char)
18       encontrados.push(direcao);
19   return encontrados;
20 };
21
22 View.prototype.find = function(char) {
23   var encontrados = this.findAll(char);
24   if(encontrados.length == 0) return null;
25   return elementoRandomico(encontrados);
26 };
```

O método **observar()** descobre as coordenadas que estamos tentando observar e, se elas estiverem no interior da grid, encontra o caractere correspondente ao elemento que fica lá. Por fim, se a função não encontrar nada, ela retornará um caractere referente à simulação da borda da parede.

Perceba que todos os métodos que comentamos antes estão agora devidamente implementados, tais como `find` e `findAll` (linhas 14 e 22). Esses métodos fazem buscas simples dentro do vetor de direções. Veja que, mesmo tal vetor tendo sido criado em um outro arquivo, ambas as funções têm acesso umas às outras em detrimento das importações de arquivos que devem

ser devidamente feitas na página de teste HTML. Esquecer esse tipo de importação implicará automaticamente em erros que serão impressos das mais diversas formas no console de logs do browser. Você poderá corrigi-los manualmente interpretando as mensagens de erro caso elas ocorram.

Agora que adicionamos todos os métodos necessários, será possível testar tudo, fazendo com que o mundo se mova. Para isso, a lógica de teste simplesmente precisa iterar sobre o método turn() definindo a quantidade de movimentos que tais objetos terão. Veja na **Listagem 15** o código para efetuar o referido teste, bem como o seu respectivo resultado impresso por meio da chamada à função `toString()`.

Listagem 15. Testando o exemplo e fazendo o mundo se mover.

Lembre-se de que essa execução leva em consideração a seleção randômica de valores, portanto, os resultados não são previsíveis e as posições de cada criatura assumirão valores variados. Contudo, conseguimos através disso, provar que essa movimentação inteligente é possível.

Adicionando novas formas de vida

O destaque dramático do nosso mundo, se você prestar um pouco de atenção, acontece quando duas criaturas saltam umas contra as outras. Você consegue pensar em outra forma interessante de implementar esse comportamento?

Uma certeza que temos sobre o modelo é que as criaturas se movem ao longo das paredes. Conceitualmente, ela mantém a sua mão esquerda na parede e segue junto à mesma, o que não é inteiramente trivial de implementar.

Precisamos ser capazes de “calcular” as direções da bússola. Até mesmo direções são modeladas por um conjunto de strings, logo precisamos definir nossa própria operação (dirSoma) para calcular as direções relativas. Então, criaremos uma função chamada dirSoma(“n”, 1) que, ao ser chamada, efetuará uma volta de 45 graus no sentido horário do norte, retornando o valor “ne”. Da mesma forma, ao chamar a função com os parâmetros dirSoma(“s”, -2)

significa que giraremos a 90 graus no sentido anti-horário a partir do sul, que fica a leste. Acompanhe a **Listagem 16** que apresenta a criação do objeto SeguidorParede, que será responsável por definir a forma como os objetos deverão efetuar viradas e caminhadas no nosso exemplo.

Listagem 16. Criando objeto SeguidorParede.

```
01 function dirSoma(dir, n) {
02   var index = direcoes.indexOf(dir);
03   return direcoes[(index + n + 8) % 8];
04 }
05
06 function SeguidorParede() {
07   this.dir = "s";
08 }
09
10 SeguidorParede.prototype.agir = function(view) {
11   var inicio = this.dir;
12   if (view.observer(dirSoma(this.dir, -3)) != "") {
13     inicio = this.dir = dirSoma(this.dir, -2);
14   while (view.observer(this.dir) != "") {
15     this.dir = dirSoma(this.dir, 1);
16     if (this.dir == inicio) break;
17   }
18   return { tipo: "mover", direcao: this.dir };
19};
```

Você pode ficar à vontade para definir onde quer inserir o script, no mesmo arquivo de view ou em um novo arquivo separado para facilitar a organização.

O método agir definido na linha 10 tem o único trabalho de “scanear” os arredores da criatura, começando a partir do seu lado esquerdo e indo no sentido horário até que ele encontre um quadrado vazio; quando encontrar, ele se moverá na direção do mesmo.

O que complica nessa situação é que uma criatura pode acabar no meio de um espaço vazio, quer como a sua posição de partida, quer como um resultado de caminhar em torno de outra criatura. Se aplicarmos a abordagem que acabamos de descrever no espaço vazio, a criatura vai apenas continuar a virar à esquerda a cada passo, correndo em círculos infinitamente.

Portanto, há uma verificação extra (a instrução if, na linha 12) para iniciar a scanneamento para a esquerda apenas se parecer que a criatura acabou de passar por algum tipo de obstáculo, isto é, se o espaço atrás e à esquerda da criatura não estiver vazio. Caso contrário, ela começará a scanear diretamente à frente, de modo que andará em linha reta quando estiver em um espaço vazio.

Finalmente, há um teste comparando as variáveis `this.dir` a `inicio` após cada passagem através do laço para se certificar de que o loop não será executado para sempre quando a criatura estiver cercada por paredes ou por outras criaturas e não puder encontrar um quadrado vazio.

Uma simulação mais realista

Para tornar a vida em nosso mundo mais interessante, vamos adicionar os conceitos de alimentação e reprodução.

Cada coisa viva no mundo recebe uma nova propriedade, **energia**, a qual é reduzida ao realizar ações e aumentada ao comer coisas. Quando a criatura tem energia suficiente, ela pode reproduzir, gerando uma nova criatura do mesmo tipo. Para manter as coisas simples, as criaturas em nosso mundo se reproduzem assexuadamente, por si mesmas.

Se as criaturas só se movem ao redor e comem umas às outras, o mundo em breve sucumbirá à lei da entropia crescente, ficará sem energia, e tornar-se-á um deserto sem vida. Para evitar que isso aconteça, adicionaremos plantas ao mundo.

Para fazer este trabalho, vamos precisar de um mundo com um método `letAct` diferente. Poderíamos simplesmente substituir o método do protótipo de Mundo, mas como implementamos a simulação inicialmente com as criaturas seguindo as paredes, vamos trabalhar tentando manter o modelo atual.

Uma solução é usar a herança. Criamos um novo construtor, **VidaComoMundo**, cujo protótipo é baseado no protótipo mundo, mas que substitui o método `letAct`. O novo método `letAct` delega o trabalho de realmente executar uma ação para várias funções armazenadas no objeto **tipoAcoes**.

Para isso, crie um novo arquivo chamado `vidacomomundo.js` e adicione o conteúdo da **Listagem 17** ao mesmo.

Listagem 17. Criando o objeto VidaComoMundo.

```
01 function VidaComoMundo(mapa, legenda) {
02   Mundo.call(this, mapa, legenda);
03 }
04
05 VidaComoMundo.prototype = Object.create(Mundo.prototype);
06 var tipoAcoes = Object.create(null);
07
08 VidaComoMundo.prototype.letAct = function(criatura, vector) {
09   var acao = criatura.agir(new View(this, vector));
10   var manipulados = acao &&
11     acao.tipo in tipoAcoes &&
12     tipoAcoes[acao.tipo].call(this, criatura, vector, acao);
13
14   if (!manipulados) {
15     criatura.energia -= 0.2;
16     if (criatura.energia <= 0)
17       this.grid.set(vector, null);
18   }
19};
```

O novo método `letAct` verifica primeiro se uma ação foi devolvida de uma forma geral, em seguida, se uma função de manipulação para este tipo de ação existe, e finalmente se esse manipulador retornou true, indicando que ele lidou com a ação com sucesso. Observe o uso da função `call` para dar ao manipulador o acesso ao mundo, por meio de sua ligação com o operador `this`.

Se a ação não funcionar por qualquer motivo, a ação padrão é a criatura simplesmente esperar. Ela perde um quinto de energia, e se seu nível de energia cai para zero ou abaixo, a criatura morre e é removida da grid.

Action Handlers

A ação mais simples que uma criatura pode executar é “crescer”, usada pelas plantas. Quando um objeto de ação como {tipo: “crescer”} é retornado, o manipulador de métodos representado na **Listagem 18** será chamado.

Listagem 18. Exemplo de manipulador de método.

```
01 tipoAcoes.crescer = function(criatura) {
02   criatura.energia += 0.5;
03   return true;
04};
```

A ação de crescer sempre funciona e adiciona meio ponto ao nível de energia da criatura. Já para movimentar o objeto temos um código mais complexo, como o da **Listagem 19**.

Listagem 19. Criando o método mover para tipoAcoes.

```
01 tipoAcoes.mover = function (criatura, vector, acao) {
02   var dest = this.checarDestino(acao, vector);
03   if (dest == null || 
04     criatura.energia <= 1 ||
05     this.grid.get(dest) != null)
06     return false;
07   criatura.energia -= 1;
08   this.grid.set(vector, null);
09   this.grid.set(dest, criatura);
10   return true;
11};
```

Esta ação verifica primeiro, usando o método `checarDestino()` definido anteriormente, se a ação fornece um destino válido. Se não, ou se o destino não estiver vazio, ou se a criatura não tiver a energia necessária, a função `mover()` retorna false para indicar que nenhuma ação foi tomada. Caso contrário, desloca-se a criatura e subtrai-se o custo de energia.

Além de tudo isso, as criaturas também podem comer. Veja na **Listagem 20** o código necessário para implementar essa funcionalidade.

Listagem 20. Implementando ação de comer.

```
01 tipoAcoes.comer = function(criatura, vector, acao) {
02   var dest = this.checarDestino(acao, vector);
03   var atDest = dest != null && this.grid.get(dest);
04   if (!atDest || atDest.energia == null)
05     return false;
06   criatura.energia += atDest.energia;
07   this.grid.set(dest, null);
08   return true;
09};
```

Comer outra criatura também envolve o fornecimento de um quadrado de destino válido. Desta vez, o destino não pode estar vazio e deve conter algo com energia, como uma criatura (mas não uma parede, paredes não são comestíveis). Se isso acontecer, a energia da criatura consumida é transferida para quem a comeu, e a vítima é removida da grid.

O mais interessante é que o leitor note que esse tipo de estrutura é flexível e pode ser livremente modificada. O mundo se moverá e tomará rumos próprios baseados na inteligência que você implementou, logo o imprevisível é a melhor resposta para o que pode acontecer. Bons estudos!

Autor



Júlio Sampaio

É analista de sistema e entusiasta da área de Tecnologia da Informação. Atualmente é consultor na empresa Visagio, trabalhando em projetos de desenvolvimento de sistemas estratégicos, é também instrutor JAVA. Possui conhecimentos e experiência em áreas como Engenharia de Software e Gerenciamento de Projetos, tem também interesse por tecnologias relacionadas ao front-end web.



Implementando serviços com AngularJS

AngularJS é um framework open-source de desenvolvimento front-end que possibilita o desenvolvimento de aplicações web, com foco em simplificar tanto a codificação quanto o processo de teste. Além disso, é possível integrá-lo com bibliotecas famosas como o Bootstrap, D3.js e o Apache Cordova (ou PhoneGap), ajudando a acelerar esse tipo de codificação como nunca antes tivemos.

O AngularJS também permite aos desenvolvedores web fazer uso da linguagem de marcação HTML para definir associações de dados, validações, além de *response handlers* para lidar com as ações do usuário em um formato declarativo que também contribui para essa mesma aceleração. Com tudo isso em conjunto, a maior consequência é, de longe, o crescimento e enriquecimento das aplicações cada vez mais ricas em funcionalidades e recursos.

Saber como arquitetar apropriadamente suas aplicações AngularJS há medida que elas crescem é algo tão importante quanto definir um bom modelo em um ciclo de vida de um software para linguagens como Java, C# ou quaisquer outros modelos server side de hoje.

Neste artigo, trataremos de expor algumas formas de bem arquitetar e construir **serviços no AngularJS** que, por sua vez, endereçam as mais usadas e importantes camadas de uma aplicação desse tipo: autenticação, mensageria, logging, acesso a dados, além da camada de acesso às regras da lógica de negócios que é requisito obrigatório para qualquer aplicação que tenha um nível de complexidade moderado. Além disso, veremos também rapidamente como integrar serviços externos do próprio framework com bibliotecas de terceiros dentro de suas aplicações de tal forma que o AngularJS possa tomar vantagem das mesmas com pouco esforço, enquanto você analisa detalhadamente a forma como o framework realiza tais integrações.

A necessidade de serviços

Em tempos onde muito se discute acerca do MVC no lado front-end, bem como das responsabilidades desse padrão no *client side*, inúmeras vertentes da comunidade de desenvolvimento front-end, muitas vezes em contravés àqueles que apoiam e suportam o lado back-end, dizem que a utilização de camadas que fornecem

Fique por dentro

Desenvolver serviços no front-end ainda é uma prática nova na comunidade web, mas que pode trazer inúmeros recursos poderosos para o seu sistema, como a possibilidade de se comunicar com quaisquer tecnologias, atuando no cliente ou não. Esse tipo de recurso já é fornecido por alguns frameworks, mas o AngularJS é o pioneiro, com recursos próprios de criação de diretivas, fábricas, módulos, além de fornecer várias opções de integração entre seus componentes e inclusive com bibliotecas de terceiros. Neste artigo, trataremos de explorar os principais recursos para criação de serviços como notificação de usuários, mensageria, regras de negócio, integrações diversas, dentre outros. Através desse conhecimento, você estará apto a criar facilmente quaisquer tipos de serviços que antes requeriam uma aplicação no servidor, com flexibilidade e enorme produtividade em comparação com o método antigo.

acesso direto a serviços é algo totalmente desnecessário e que foge às responsabilidades da mesma, devendo ser amparadas, portanto, pelas camadas que atuam no servidor da arquitetura. Para entender a contrapartida dessa discussão, é necessário que antes respondamos a algumas questões inerentes a tal dúvida: Por que serviços são realmente necessários no front-end? Como eles apoiam a sua aplicação? Quais são suas responsabilidades na aplicação de uma forma geral?

A primeira sentença a ser analisada diz que “os serviços são a base de tudo”. Isto é, eles fornecem funcionalidades de forma transversal, solicitam e manipulam dados, se integram com outros serviços externos, e incorporam lógica de negócios, tudo isso de uma forma tão simples quanto um objeto JSON possa parecer (que é inclusive o tipo de dado mais usado para representá-los).

Em alusão a uma situação um pouco mais lúdica, os serviços funcionam como a fundação de um prédio. Obviamente um prédio pode ser construído sem a necessidade de uma fundação, porém ele não irá durar tanto quanto duraria se a tivesse. Da mesma forma, sem serviços em uma aplicação, a mesma logo se tornará tão pesada que não se aguentará por muito tempo. A **Figura 1** nos mostra uma representação visual dos componentes do AngularJS e suas interações.

Observe que, conforme discussões se abrangem, maior é a necessidade de entender como ambos os padrões (MVC + Serviços) interagem e se auto ajudam, quando em conjunto.

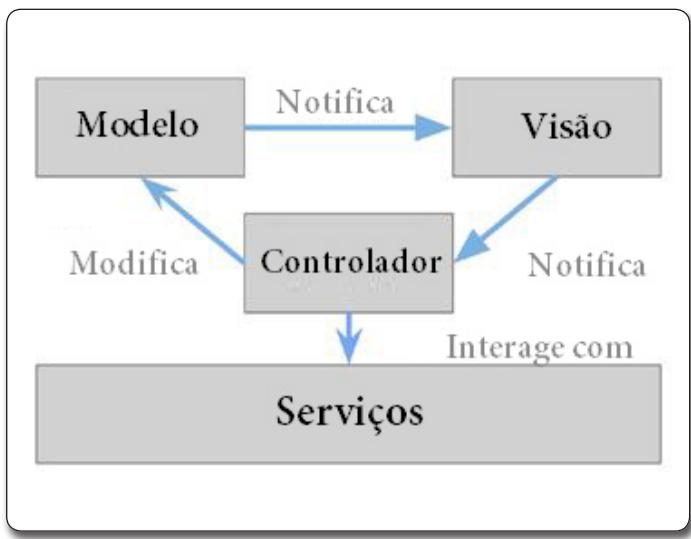


Figura 1. Representação do modelo de componentes do AngularJS e interações

A necessidade de boas práticas

Quando uma equipe de desenvolvimento se depara com um projeto novo, uma das primeiras e mais importantes ações é escolher um framework para facilitar a produtividade do projeto como um todo. Aliado a essa escolha, o arquiteto ou alguém com mais experiência terá o desafio de entender como o mesmo funciona e, mais importante que isso, como aplicar as boas práticas de programação a este framework.

As boas práticas são como guias que nos ajudam a entender o melhor jeito de usar aquele framework, assim como estruturar a sua aplicação de forma a deixá-la manutenível, testável e reusável.

O desenvolvimento utilizando módulos leva em consideração a unidade na aplicação como um todo. Isso quer dizer que cada componente deve ter sempre uma única e específica responsabilidade, caso contrário teremos uma discrepância na implementação e o modelo tenderá a se tornar cada vez mais complexo de manter.

Desenhandos serviços

Antes de iniciarmos a implementação de fato dos nossos serviços, é preciso que o leitor tenha em mente algumas regras essenciais para a criação correta dos mesmos. Vejamos:

- Antes de iniciar qualquer serviço, **tome um tempo pensando sobre ele**. Quais serão as suas funcionalidades (faça uma lista delas, no papel mesmo, se achar melhor)? Como criarei este(s) serviço(s) de modo que eu possa facilmente modificá-lo(s) no futuro, mantendo-o(s) sempre testável(eis) e manutenível(eis)?
- **Meça duas vezes, corte apenas uma**. Isso serve para te ensinar a não se apressar quando for criar serviços e/ou quaisquer implementações. Às vezes é melhor perder um bom tempo planejando direito o que você vai desenvolver, em vez de atropelar o desenvolvimento ágil e precisar refazer no futuro.
- **Foque no desenvolvedor**, e não em você mesmo. Isso quer dizer que, quando construir um serviço, pense em quem irá usá-lo. Muitas vezes esquecemos dessa premissa por não estarmos lidando com algo que será acessado diretamente por um usuário, pois tudo

será abstruído pela “figura do desenvolvedor”. Coloque-se no lugar dele, e pergunte-se se, sendo você o desenvolvedor consumidor, você entenderia o serviço que criou e disponibilizou.

- Faça apenas o que o nome do seu serviço manda fazer. Essa prática já é muito comum entre codificadores para nomes de métodos, por exemplo. Se o seu método se chama cadastrarCliente(), adivinhe qual deve ser a única coisa que ele faz? O mesmo vale para os serviços.

- Mantenha o mais usável possível. Isso quer dizer que, se o seu serviço foi criado por alguma ferramenta de CRUD, por exemplo, que automaticamente gerou métodos de add(), remover(), alterar() e pesquisar(), mas no seu serviço só será usado o método de pesquisa, então remova os demais. Deixe-o simples e direto.

- Documente a sua interface de serviço. Isso será útil não só para outros desenvolvedores que venham a encarar o seu código no futuro, mas principalmente para você mesmo. Afinal, você não é obrigado a decorar tudo o que codifica. Existem inúmeras ferramentas que te ajudam nisso: JSDoc, YUIDoc, Ext Doc, ou algum plugin que a sua IDE disponibilize e que você se sinta à vontade para usar.

Instalando o AngularJS

Para trabalhar com o AngularJS a única coisa que você precisará fazer é importar o seu arquivo JavaScript para o seu projeto, seja através do download direto ou do uso da interface CDN que o mesmo disponibilizará de forma gratuita e online. Para os objetivos deste artigo, faremos sempre uso da segunda opção pela simplicidade da mesma. Na seção **Links** deste artigo você encontrará a URL da página de download oficial do projeto, onde poderá encontrar todas as referências necessárias. Ademais, não será objeto deste instruir o leitor acerca de conceitos básicos do AngularJS, visto que isso fugiria ao foco que é apresentar como criar serviços com a tecnologia.

No momento de escrita deste artigo, a versão mais recente do AngularJS era a 1.3.14, mas o leitor poderá usar versões anteriores sem problemas.

As bases de uma aplicação feita sob os pilares do AngularJS são: **comportamento** e **interação** em conjunto com a **apresentação**. De início, esse tipo de conceito pode ser confuso, especialmente se você estiver acostumado a lidar com outros frameworks onde tais pilares são distintos e funcionam de forma independente. Vejamos o exemplo representado na **Listagem 1**.

Neste exemplo temos basicamente a importação do arquivo js sendo feita dentro da tag de cabeçalho HTML, e um campo de input de formulário seguido de um cabeçalho de texto h1 para exibição do resultado final. No mesmo exemplo, é possível ver a associação de dados bidirecionais sendo feita sem muito trabalho. Essa associação significa que se você mudar os dados exibidos no back-end da aplicação, tais mudanças surtirão efeito no lado front-end automaticamente, e vice-versa.

Perceba também que, no exemplo, temos três operadores incomuns para a HTML, a saber:

- **ng-app**, o mais importante atributo na nossa página, que será responsável por iniciar o processo do AngularJS;
- **ng-model="meuNome"**, que se responsabilizará por interligar o campo de input ao elemento h1 de resultado, automaticamente, exibindo o valor que estiver em um respectivamente no outro;
- **{{meuNome}}**, o mesmo que o item anterior, só que no processo inverso, de exibição.

Listagem 1. Exemplo básico de uso do AngularJS.

```
<!doctype html>
<html ng-app>
<head>
<meta charset="utf-8">
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/
angular.min.js"></script>
</head>
<body>
<div style='padding: 3%'>
<input type="text" ng-model="meuNome" placeholder=
"Digite um nome aqui..." style='margin-bottom: 20px'>
<h1>Olá, {{ meuNome }}!</h1>
</div>
</body>
</html>
```

A execução deste conteúdo no browser resultará em algo semelhante ao que temos na **Figura 2**.

Perceba que essa implementação é demasiadamente simples se comparada ao código necessário para criá-la em outras tecnologias, como JSP, JSF ou ASP.NET, por exemplo. O **item 1** marca a página quando iniciada, nada aconteceu ainda; enquanto o **item 2** demarca o final do teste, quando o usuário preenche o campo que tem sua respectiva label atualizada automaticamente. Para inibir quaisquer efeitos de estilo, os arquivos de CSS do Bootstrap ou de um framework de sua preferência podem ser usados.



Figura 2. Resultado da execução do exemplo AngularJS

A necessidade de testes

O AngularJS foi totalmente projetado com o conceito de testabilidade em mente. Evidentemente, outro conceito, o de *Dependency Injection* (ou Injeção de Dependências), anda de mãos dadas com ele. A injeção de dependências permite que você escreva códigos que sejam extremamente desacoplados dos serviços aos quais os mesmos dependem. Consequentemente, isso nos permite criar códigos que obejam à famosa **Lei de Demeter**, que constitui uma boa prática quando da escrita de códigos de teste. De uma

forma geral, essa lei diz que cada unidade de código deve ter conhecimento limitado sobre as demais.

Em outras palavras, jamais chame objetos encadeados nos seus métodos sob pena de aumentar consideravelmente a complexidade do seu código, tornando-o também mais difícil de ser testado. Em vez disso, passe como parâmetros os objetos com os quais precisa interagir. Veja na **Listagem 2** um exemplo fiel da chamada encadeada de objetos.

Listagem 2. Exemplo de função com objetos encadeados.

```
$scope.getNome = function(autenticacao) {
  if (autenticacao && autenticacao.clienteAtual) {
    return autenticacao.clienteAtual.primeiroNome + '' + autenticacao.cliente
      Atual.ultimoNome;
  }
  return '';
},
```

Observe que o código aceita um objeto chamado “autenticacao”, bem como a forma como ele certifica que ambos os objetos **autenticacao** e **clienteAtual** são válidos antes de acessar as propriedades do objeto **clienteAtual** para retornar o nome do cliente.

Testar o referido código com objetos mock é algo muito trabalhoso, neste exemplo. Você não somente terá de “mockar” todos os métodos chamados no objeto **autenticacao**, como também precisará fazer o mesmo para todos os métodos chamados no objeto **clienteAtual**, tornando o seu código muito maior do que ele necessita ser.

Por outro lado, se você seguir a mesma implementação, porém fazendo uso da Lei de Demeter, poderá simplificar o seu código consideravelmente. Vejamos na **Listagem 3** como atingir esse objetivo.

Listagem 3. Código simplificado pela Lei de Demeter.

```
$scope.getNome = function (cliente) {
  if (cliente) {
    return cliente.primeiroNome + '' + cliente.ultimoNome;
  }
  return '';
},
```

Agora, nosso código está apenas validando o objeto que invoca métodos no mesmo, ao contrário de verificar o objeto que deveria encapsulá-lo. Isso resulta em um código muito mais simplificado, uma vez que tivemos de mockar somente os métodos atrelados ao objeto **cliente**.

Um outro princípio muito importante para a testabilidade do seu código se caracteriza pelo uso das injeções de dependências: sempre passe todas as dependências requeridas para o seu serviço no momento de criação da função, e jamais use o serviço **\$injector** para recuperar dependências, a menos que não haja outra maneira de injetar um serviço obrigatório. Apesar de ser permitido invocar uma instância de um serviço através desse operador, você nunca

poderá testar tal código, isso porque você não poderá garantir que mockou todos os serviços que o seu código invocará.

Ao passar como parâmetros todas as dependências obrigatórias que o seu serviço irá precisar, você terá um *road map* de todos os que precisarão mockar, além de ser possível então determinar facilmente quais métodos são chamados por quais serviços.

Dando continuidade, outro ponto para se ter em mente se refere a limitar seus métodos de construtores para somente receber atribuições de campos e definições de funções. Não efetue chamadas a serviços externos para inicializar dados. Em vez disso, adicione esse tipo de código dentro de métodos de inicialização e/ou configuração que devem ser chamados depois que a instância de seu serviço tiver sido criada. Isso porque ao fazer esse tipo de chamada explícita, você estará dificultando futuros testes e debugs, uma vez que os mesmos serviços externos podem apresentar falhas durante o teste, o que pode te confundir entre qual dos serviços está apresentando erro.

Estruturando o serviço em código

Antes de começar a codificar o seu serviço, você precisa pensar em como não poluir o namespace global e somente expor métodos e propriedades que você pretenda usar para os consumidores quando eles interagirem com o seu serviço.

A razão para não poluir o namespace global é que precisamos proteger o código de eventuais colisões de nomenclaturas que podem ocorrer com outros scripts carregados na página, além de evitar que seu código sobrescreva algum outro código de script.

O jeito mais fácil de prevenir isso é criando um IIFE (*Immediately-Invoked Function Expression*, vide **Box 1**) que você poderá usar para definir o seu módulo e serviço (**Listagem 4**).

Listagem 4. Exemplo básico de uso do IIFE.

```
(function () {  
  'use strict';  
  // definições de módulo aqui  
  // definições de serviço aqui  
});
```

Não somente pode este padrão ser usado para seus serviços, como também para todos os seus controladores e diretrizes. A única vez que é aconselhável não usar esse padrão é se você estiver usando uma ferramenta de construção que concatena seu código em um único arquivo e encapsula-o em uma *closure* (que ocorre quando temos uma função declarada dentro do corpo de outra função, e a função interior referencia variáveis locais da função exterior).

A próxima coisa a se pensar quando escrever o seu serviço é expor somente métodos e propriedades que você pretenda que os usuários de seu serviço tenham acesso. A maneira mais fácil de fazer isso é usando o padrão *Christian Heilmann's Revealing Module Pattern* para definir seu serviço (veja na seção **Links** uma referência para o site do padrão).

A vantagem de utilizar esse padrão é que você não tem que usar o nome do objeto principal para chamar métodos ou variáveis de acesso, além de ter a capacidade para expor apenas os métodos e variáveis para o mundo exterior que você sente que precisam estar acessíveis.

BOX 1. Immediately-Invoked Function Expression (IIFE)

O IIFE (Expressão de Função Imediatamente Invocada, em português literal) é um padrão de design JavaScript comum usado pela maioria das bibliotecas populares (jQuery, Backbone.js, Modernizr, etc.) para colocar todo o código da mesma em um escopo local. É apenas uma função anônima que é encapsulada dentro de um conjunto de parênteses e invocada imediatamente. Um IIFE geralmente é implementado assim: (function() { // Código })();

O exemplo da **Listagem 5** define um serviço de mensageria para publicação/subscrição usando o Revealing Module Pattern. A função **messaging_service** cria uma closure onde definimos os vários métodos de serviço e quaisquer variáveis internas que precisamos para armazenar o estado do mesmo. Na parte inferior da definição da função, retornarmos um objeto que inclui apenas os métodos que queremos tornar visíveis para os consumidores do serviço.

A listagem está auto comentada, porém vejamos algumas observações importantes:

- Na linha 2 vemos o modelo de implementação de serviços padrão que comentamos antes. Este deverá ser seguido sempre que um novo serviço necessitar ser criado.
- Na linha 6 é possível observar o uso da função **module()** do AngularJS, que se encarrega de injetar o módulo requerido para uso posterior. Na mesma linha estamos criando uma factory de mensagerias que será útil para fabricar novos objetos sempre que necessários.
- Os métodos de publicação (linha 15), subscrição (linha 22) e desubscrição (linha 30) fazem uso do array **cache** que será útil para lidar com o salvamento de informações temporárias, otimizando o tempo de acesso às mesmas e aumentando, assim, a performance da aplicação.
- No final da implementação, a partir da linha 41, simplesmente adicionamos os novos métodos de serviço ao objeto que os guardará como referências: a variável **servico**, que será também retornada como resultado final.

Configurando o serviço

O método **provider** dos módulos do AngularJS possibilita a criação de um serviço com configurações básicas de métodos que os consumidores possam invocar sem problemas (em um método **config**, por exemplo).

Através desse método, podemos definir um serviço de logging, por exemplo, de forma extremamente mais simples. Vejamos a **Listagem 6**.

Esse tipo de estrutura é muito semelhante à forma como configuramos logs em outros ambientes, tais como Java, .Net, etc. Até mesmo o uso dos objetos JavaScript em alusão aos que usamos na biblioteca Log4J do Java se equiparam, o mesmo para appenders, níveis de logs, etc.

No entanto, você não precisa sempre usar o método provider para permitir que os seus serviços sejam configurados. É possível usar ambos os métodos service e factory para definir o seu serviço e fornecer métodos de configuração que possam ser chamados diretamente do método de execução do módulo: `run`, que, por sua

Listagem 5. Exemplo de função para serviço de mensageria.

```
01 (function () {
02   'use strict';
03   // Define a fábrica no módulo.
04   // Injeta as dependências.
05   // Aponta para a função de definição da fábrica.
06   angular.module('brew-everywhere').factory('messaging_service',
07     [messaging_service]);
08
09   function messaging_service() {
10
11     //##region Propriedades Internas
12     var cache = {};
13     //##endregion
14
15     //##region Métodos Internos
16     function publicar(topico, args) {
17       cache[topico] && angular.forEach(cache[topico],
18         function (callback) {
19           callback.apply(null, args || []);
20         });
21
22     function subscrever(topico, callback) {
23       if (!cache[topico]) {
24         cache[topico] = [];
25       }
26       cache[topico].push(callback);
27       return [topico, callback];
28     }
29
30     function dessubscrever(handle) {
31       var t = handle[0];
32       cache[t] && angular.forEach(cache[t], function (idx) {
33         if (this == handle[1]) {
34           cache[t].splice(idx, 1);
35         }
36       });
37     }
38     //##endregion
39
40     // Define as funções e propriedades para revelar.
41     var servico = {
42       publicar: publicar,
43       subscrever: subscrever,
44       dessubscrever: dessubscrever
45     };
46
47     return servico;
48   }
49 })();
```

Listagem 6. Exemplo de módulo para serviço de logging.

```
01 angular.module('MinhaApp', ['logging']).config(function(provedorLogging){
02   // Inicializa o serviço de logging
03   provedorLogging.init();
04   // Configura o nível do log
05   provedorLogging.setLogLevel(log4javascript.Level.ALL);
06   // Cria e Add um appender ao logger
07   var appender = new log4javascript.PopupAppender();
08   provedorLogging.setAppender(appender);
09});
```

vez, executa após o bootstrap da sua aplicação. A única coisa que você precisa garantir é que seu serviço siga as práticas de projeto que discutimos anteriormente e não tente chamar quaisquer serviços externos assim que for instanciado.

Não importa como você cria seus serviços, ao fornecer funcionalidade para configurá-los, você faz com que os mesmos sejam mais flexíveis, mais fáceis de usar, além de reduzir a complexidade necessária para incluí-los em uma aplicação.

Gerenciando preocupações cross-cutting

Um bom design em uma aplicação usa camadas para separar áreas de responsabilidade. Se bem feito, cada camada tem uma única responsabilidade e se interliga com as outras camadas usando uma interface bem definida.

As camadas mais populares que você vai ver incluídas em um aplicativo são: dados, lógica de negócios e interface do usuário. No entanto, existem serviços que atravessam todas as outras camadas; aqueles que lidam com preocupações *cross-cutting* (transversais), tais como mensageria, logs, validação de dados, *caching*, internacionalização e segurança.

Comunicar-se com os consumidores do seu serviço se torna muito importante quando você tem métodos que são de longa duração ou que façam chamadas assíncronas AJAX do front-end para um servidor. Obviamente, você não vai querer bloquear a execução do seu aplicativo enquanto o seu serviço está processando ou esperando uma chamada AJAX voltar. Logo, a melhor maneira de lidar com todos esses métodos é executando-os de forma assíncrona.

Existem várias saídas que você pode usar para lidar com tais métodos:

- Solicitar ao consumidor o fornecimento de uma função de retorno que o seu método de serviço irá executar após a conclusão;
- Devolver uma “promise (retorno promessa)” que garante que o seu método de serviço será executado após a conclusão;
- Ou usar um padrão de design de mensagens para notificar o consumidor uma vez que seu método de serviço estiver concluído.

Embora as **funções de callback** e as **promises** funcionem muito bem para notificar um único consumidor sempre que seu método de serviço tenha sido concluído, elas não funcionam direito quando você tem vários consumidores que precisam saber quando os métodos do seu serviço foram finalizados ou quando os dados geridos pelo mesmo foram atualizados.

Isso ocorre especialmente quando você tem várias views visíveis ao mesmo tempo e cada uma precisa ser notificada quando os dados do seu serviço mudam. Um bom exemplo é o carrinho de compras em um aplicativo de e-commerce. À medida que o usuário adiciona itens ao carrinho, o mesmo precisa atualizar a exibição dos conteúdos, juntamente com outros serviços que possam mostrar os custos de transporte, total de compras feitas, e assim por diante.

Para lidar com tais situações, você precisa usar um padrão de projeto *publish/subscribe* tanto para notificar consumidores quando

métodos de longa duração completarem, quanto para notificar todas as views do seu aplicativo quando os dados do seu serviço mudarem.

Uma coisa que você deve sempre ter em mente quando da utilização de funções de callback para mensageria é que é possível diminuir o desempenho do aplicativo se seus manipuladores de callback publicarem outras mensagens como parte do seu processamento. Então, o melhor é usar o serviço `$timeout` para moldar o código que publica mensagens, uma vez que o manipulador de callbacks pode retornar o mais rápido possível e não afetar o desempenho.

O código demonstrado pela **Listagem 7** representa a criação de um serviço de mensageria usado pelo aplicativo de exemplo.

Listagem 7. Exemplo de módulo para serviço de mensageria.

```
01 angular.module('brew-everywhere').factory('messaging', function () {
02   var cache = {};
03
04   var subscrever = function (topico, callback) {
05     if (!cache[topico]) {
06       cache[topico] = [];
07     }
08     cache[topico].push(callback);
09     return [topico, callback];
10   };
11
12   var publicar = function (topico, args) {
13     cache[topico] && angular.forEach(cache[topico],
14       function (callback) {
15         callback.apply(null, args || []);
16       });
17   };
18
19   var dessubscrever = function (handle) {
20     var t = handle[0];
21     if (cache[t]) {
22       for(var x = 0; x < cache[t].length; x++) {
23         if (cache[t][x] === handle[1]) {
24           cache[t].splice(x, 1);
25         }
26       }
27     }
28   };
29
30   var service = {
31     publicar: publicar,
32     subscrever: subscrever,
33     dessubscrever: dessubscrever
34   };
35   return service;
36});
```

Inicialmente, definimos uma variável interna chamada `cache` que será usada para armazenar todos os assinantes para cada uma das mensagens e para os seus métodos de callback. Dentre os três métodos definidos, temos:

- O método `subscrever` recebe uma string de nome `topico` e uma função de callback para invocar quando o mesmo for publicado. Em seguida, ele checa se o cache contém uma propriedade com o nome do tópico; se não existir, uma nova propriedade é adicionada ao objeto e uma matriz é criada e atribuída à mesma.

Em seguida, a função de retorno é inserida na matriz. Finalmente, um identificador é retornado para que o assinante possa utilizar quando desejar cancelar a subscrição.

- O método `publicar` recebe a mesma string `topico` e um array de argumentos que pode ser passado para a callback que assina o tópico em questão. O método verifica se o objeto de cache contém uma propriedade com o nome do tópico. Se assim for, ele itera através de cada item na matriz e chama a função de callback com os argumentos passados.

- O método `dessubscrever` recebe o identificador retornado pelo método `subscrever` e remove a função de callback associada com o tópico. O método primeiro verifica se o objeto de cache contém uma propriedade com o nome do tópico, e em caso afirmativo, ele itera através de cada item da matriz até encontrar um callback que coincida com o callback contido no identificador e, então, remove-o da matriz.

Finalmente, o serviço retorna um objeto com apenas os métodos que queremos tornar públicos para os consumidores, seguindo o Revealing Module Pattern.

Para fazer uso desse mesmo serviço, o consumidor precisará incluir um outro serviço, o de `messaging`, como uma dependência e então subscrever a um ou mais tópicos que o mesmo tenha interesse em ser notificado sempre que novos eventos acontecerem. Para exemplificar essa situação, consideremos um controller simples (**Listagem 8**) que usa esse mesmo serviço de mensageria para autenticar um usuário e, assim, faz uso direto do serviço que acabamos de criar.

O view controller `LoginCtrl` recebe o serviço `messaging` e o serviço de eventos como dependências. O serviço de mensageria é a implementação do serviço `publish/service` que vimos antes, já o serviço `events` é nada mais que um conjunto de constantes que define as várias mensagens que são usadas pela aplicação.

O controller então define um gerenciador de mensagens chamado `autenticacaoUsuarioCompleta` (linha 10) que foi registrado pelo serviço de mensageria com o tópico `_AUTHENTICATE_USER_COMPLETE_`. Em seguida, temos a definição do método `onLogin` (linha 22) que é usado como um gerenciador de ng-click (lembre-se do exemplo que vimos sobre o AngularJS e seus componentes) para logar o usuário através da publicação da mensagem `_AUTHENTICATE_USER_`, tendo os dados de login e senha como argumentos enviados.

Quando o gerenciador de ng-click do AngularJS é executado, ele automaticamente publica a mensagem e então o controller espera até que o serviço responda com a mensagem de completude, que, por sua vez, receberá o objeto passado dentro do callback atribuindo a ele o valor da variável `usuarioAtual`.

O leitor talvez se pergunte onde seria feito o tratamento de erros, por exemplo, que constitui requisito básico e essencial para quaisquer tipos de serviços. Este deve ser feito de fora do controller por outros serviços. No nosso exemplo, se a mensagem de completude nunca chegar, o controller simplesmente não faz nada.

Listagem 8. Exemplo de módulo para serviço de autenticação.

```
01 angular.module('brew-everywhere').controller('LoginCtrl',
  function($scope, messaging, events){
02   //##region Modelos Internos
03   $scope.usuario = '';
04   $scope.senha = '';
05   $scope.usuarioAtual = {};
06   $scope.usuarioAutenticado = null;
07   //##endregion Modelos Internos
08
09   //##region Message Handlers
10   $scope.autenticacaoUsuarioCompleta = function(usuario) {
11     $scope.usuarioAtual = usuario;
12   }
13   $scope.usuarioAutenticado = messaging.subscrever(
14     events.message__AUTHENTICATE_USER_COMPLETE__,
15     $scope.autenticacaoUsuarioCompleta);
16   //##endregion Message Handlers
17
18   //##region View Handlers
19   $scope.$on('$destroy', function(){
20     messaging.dessubscrever($scope.usuarioAutenticado);
21   });
22   $scope.onLogin = function() {
23     messaging.publicar(event.message__AUTHENTICATE_USER__,
24     [$scope.usuario, $scope.senha]);
25   }
26});
```

Ademais, o controller também faz uso do método `$scope.$on` que servirá para assistir à mensagem `$destroy` que será enviada quando o controller for destruído, assim o mesmo poderá cancelar a subscrição sobre quaisquer mensagens que tenham sido inscritas antes. Isso é muito importante para que o serviço de mensageria não tente chamar uma função de callback que, na verdade, não existe.

Gerenciando notificações de usuário

Após o término da construção da camada de mensageria, você pode começar a incluir novas camadas, como a de gerenciamento de notificações de usuário, por exemplo.

Continuemos nossa implementação construindo uma diretiva que agirá diretamente no nosso serviço de mensageria para exibir um GIF animado indicando que o nosso programa está ocupado com alguma tarefa. Veja na **Listagem 9** o código que precisaremos para isso.

A diretiva “carregando” basicamente exibe e esconde o elemento que está atrelado a ela, baseada em duas mensagens: `_SERVER_REQUEST_STARTED_` e `_SERVER_REQUEST_ENDED_`. O template para a diretiva é nada mais que uma div HTML que encapsula a tag de imagem, que, por sua vez, constitui uma animação (um gif).

Agora, sempre que quisermos indicar que a aplicação se encontra ocupada, podemos publicar a mensagem de start para exibir o item de carregamento, e uma vez completada, podemos chamar a mensagem de end para escondê-lo.

Outro bom exemplo de uso do serviço de notificação de usuário é a implementação de uma diretiva de lista de notificações simples que

funciona com dois serviços juntos para exibir erros e mensagens de alerta para o usuário. Os serviços de erros e notificações são muito similares, exceto pelo fato de que cada um lida com mensagens diferentes. Através deles é possível determinar como certas mensagens são exibidas, estabelecendo pilares de controle sobre as mesmas.

Listagem 9. Incluindo diretiva de carregamento temporário.

```
01 angular.module('brew-everywhere').directive('carregando', function
  (messaging, events) {
02   return {
03     restrict: 'E',
04     template:'<div class="row"></div>',
06     link: function(scope, element) {
07       element.hide();
08       var startRequestHandler = function () {
09         // recebe a notificação de início, mostra o elemento
10         element.show();
11       };
12       var endRequestHandler = function() {
13         // recebe a notificação de fim, esconde o elemento
14         element.hide();
15       };
16       scope.startHandle = messaging.subscribe(
17         events.message__SERVER_REQUEST_STARTED__,
18         startRequestHandler);
19       scope.endHandle = messaging.subscribe(
20         events.message__SERVER_REQUEST_ENDED__,
21         endRequestHandler);
22       scope.$on('$destroy', function() {
23         messaging.unsubscribe(scope.startHandle);
24         messaging.unsubscribe(scope.endHandle);
25       });
26     }
27   });
});
```

Suponha que você deseja ter mensagens de erro sendo exibidas em um diálogo do tipo pop-up em vez de em uma lista fixa no corpo da página; uma vez que as mensagens de erro são lançadas por um serviço a parte, você pode facilmente mudar que mensagem o serviço de erros deve publicar sempre que ele receber uma nova mensagem.

Vejamos na **Listagem 10** um exemplo básico de serviço de erros que gerencia tal subscrição.

Esse serviço tem um array interno que é usado para salvar as mensagens de erro à medida que elas são recebidas pelo mesmo. O gerenciador de mensagens `addHandlerMsgErros` faz uso do serviço `$timeout` para permitir que o método retorne imediatamente e, então, publica a mensagem `_ERROR_MESSAGES_UPDATED_` com o conteúdo do array `msgErros`.

O serviço também tem um gerenciador de mensagens para permitir que os consumidores digam ao serviço para limpar todas as mensagens que ele contém (linha 17). O mesmo também fornece um método `init` para que o AngularJS possa carregar o serviço usando o método `run()` do módulo.

Para fazer o exemplo em questão funcionar, precisamos criar uma diretiva de lista de notificações que irá lidar com a atualização das mesmas. Para isso, crie também um novo módulo com o conteúdo presente na **Listagem 11**.

Listagem 10. Exemplo básico de serviço de notificações de erro.

```
01 angular.module('brew-everywhere').factory('erros',function
  ($timeout, messaging, events) {
02   var msgErros = [];
03   var addHandlerMsgErros = function(msg, tipo){
04     if(!msgErros){
05       msgErros = [];
06     }
07     msgErros.push({tipo: tipo, msg: msg});
08   $timeout(function() {
09     messaging.publicar(events.message._ERROR_MESSAGES_UPDATED_,
10       msgErros);
11   }, 0);
12
13   messaging.subscrever(events.message._ADD_ERROR_MESSAGE_,
14     addHandlerMsgErros);
15   var limparMsgErrosHandler = function() {
16     msgErros = [];
17   };
18   messaging.subscrever(events.message._CLEAR_ERROR_MESSAGES_,
19     limparMsgErrosHandler);
20   var init = function(){
21     msgErros = [];
22   };
23   var erros = {
24     init: init
25   };
26   return erros;
27 }).run(function(erros){
28   erros.init();
29});
```

Listagem 11. Incluindo a diretiva de lista de notificações.

```
01 angular.module('brew-everywhere').directive('notificationList', function
  (messaging, events) {
02   return {
03     restrict: 'E',
04     replace: true,
05     templateUrl:'directive/notificationList/notificationList.html',
06     link: function(escopo) {
07       escopo.notificacoes = [];
08       escopo.onmsgErroUpdatedHandler = function (msgErro) {
09         if(!escopo.notificacoes){
10           escopo.notificacoes = [];
11         }
12         escopo.notificacoes.push(msgErro);
13         messaging.publicar(events.message._CLEAR_ERROR_MESSAGES_);
14       };
15
16       messaging.subscrever(events.message._ERROR_MESSAGES_UPDATED_,
17         escopo.onmsgErroUpdatedHandler);
18       escopo.onmsgUsuarioUpdatedHandler = function (msgUsuario) {
19         if(!escopo.notificacoes){
20           escopo.notificacoes = [];
21         }
22         escopo.notificacoes.push(msgUsuario);
23         messaging.publicar(events.message._CLEAR_USER_MESSAGES_);
24       };
25
26       messaging.subscrever(events.message._USER_MESSAGES_UPDATED_,
27         escopo.onmsgUsuarioUpdatedHandler);
28       escopo.$on('$destroy', function() {
29         messaging.dessubscrever(escopo.errorMessageUpdateHandle);
30         messaging.dessubscrever(escopo.msgUsuarioUpdatedHandle);
31       });
32       escopo.acknowledgeAlert = function(indice){
33         escopo.notificacoes.splice(indice, 1);
34       };
35     };
36   });
37});
```

RENOVE JÁ!

Sua assinatura pode estar acabando

Renovando a assinatura de sua revista favorita
você ganha brindes e descontos exclusivos.



A diretiva **notificationList** lança as mensagens “`_ERROR_MESSAGES_UPDATED_`” e “`_USER_MESSAGES_UPDATED_`” e então adiciona as que foram recebidas em um array interno. A mesma diretiva também tem um handler `ng-click` que remove as notificações de dentro do array interno. Ela assiste à mensagem `$destroy` para que possa remover a subscrição do serviço de mensageria. A essa altura, você já deve estar apto a usar o mesmo esquema de criação publicação/subscrição que se equipara aos demais, a única diferença consiste na associação às mensagens corretas, pois o passo a passo é sempre o mesmo.

Além das implementações de notificações que fizemos, o leitor pode testar outros conceitos como a inclusão de logging no processo, OAuth com APIs públicas/privadas, dentre outros.

Encapsulando lógica de negócio

Com o crescimento do JavaScript, tivemos como grande consequência direta o aumento também do número de aplicações e desenvolvedores focados em migrar toda a lógica que antes estava alojada no servidor para o lado cliente. E a velocidade foi um dos maiores benefícios dessa transição, uma vez que não precisamos fazer tantas requisições serializadas a servidores remotos, e isso aumenta em muito a performance.

O conceito de manter lógica de negócio que opera sobre um objeto no próprio objeto é um padrão da programação orientada a objetos que muitas linguagens disponibilizam como parte de suas definições de classe.

Em JavaScript, que é uma linguagem totalmente orientada a objetos, podemos atingir esse mesmo escopo através da redefinição dos nossos objetos prototypes. Analisemos a **Listagem 12**.

Listagem 12. Exemplo de classe JavaScript redefinida.

```
01 var Pessoa = function() {
02   var auto = this;
03
04   auto.nome = 'DevMedia';
05   auto.cor = 'verde';
06   auto.idade = 15;
07   auto.peso = 86.56;
08   auto.altura = 1.78;
09   auto.cpf = 12345678910;
10   auto.endereco = 'Rio de Janeiro';
11 };
12
13 Pessoa.prototype = {
14   andar: function(metros) {
15     console.log("Andou " + metros + " metros");
16   }
17
18   calcularIMC: function() {
19     return (this.peso / (this.altura * this.altura));
20   }
21
22   calcularAnoNasc: function() {
23     return 2015 - this.idade;
24   }
25 }
```

Isso nos permite criar objetos por intermédio do operador `new` usando a definição de prototype, e chamar seus atributos e métodos tal como fazemos em linguagens de programação OO convencionais.

A maior vantagem de usar esse tipo de implementação é que ele nos possibilita criar lógica de negócio como parte do objeto de modelo do sistema. Isso faz com que o código fique mais simples e fácil de implementar, bem como de manter por outrem, removendo a necessidade de criar novas funções para fazer operações adicionais sobre o objeto. Porém, temos também a desvantagem de correr o risco de ter o código de negócio espalhado por toda a aplicação, portanto, o leitor deve estar sempre atento a esse tipo de implementação e, quando possível, usar bibliotecas de terceiros para ajudar a construir tal camada.

Uma outra forma comum de incorporar sua lógica de negócio é criar um serviço que exponha várias funções que a encapsulam, criando arrays de objetos como parâmetros à medida que a lógica os requerer. Por exemplo, podemos pegar o mesmo conteúdo da listagem anterior e adicionar a um serviço encapsulando todas as funcionalidades da mesma (**Listagem 13**).

Listagem 13. Encapsulando classe de negócio em um módulo.

```
01 var Pessoa = function() {
02   var auto = this;
03
04   auto.nome = 'DevMedia';
05   auto.cor = 'verde';
06   auto.idade = 15;
07   auto.peso = 86.56;
08   auto.altura = 1.78;
09   auto.cpf = 12345678910;
10   auto.endereco = 'Rio de Janeiro';
11 };
12
13 angular.module('brew-everywhere').factory('pessoaHelper',function() {
14
15   var andar: function(metros) {
16     console.log("Andou " + metros + " metros");
17   }
18
19   var calcularIMC: function(pessoa) {
20     return (pessoa.peso / (pessoa.altura * pessoa.altura));
21   }
22
23   var calcularAnoNasc: function(pessoa) {
24     return 2015 - pessoa.idade;
25   }
26
27   var helper = {
28     andar: andar,
29     calcularIMC: calcularIMC,
30     calcularAnoNasc : calcularAnoNasc
31   }
32
33   return helper;
34 }
```

Veja como o código é ainda mais facilitado em detrimento da fábrica que criamos para produzir objetos desse tipo. Dessa forma, sempre que quisermos usar as funcionalidades de negócio que a pessoa fornece, basta instanciar um novo helper e usar os métodos

do mesmo. Isso nos permitirá remover a lógica de negócio de um objeto da camada de modelo, que agora poderá ser usada em quaisquer camadas sem termos de nos preocupar em estar ferindo as boas práticas e a qualidade do código como um todo. Veja na **Listagem 14** o código que seria necessário para efetuar uma chamada a qualquer um dos métodos de negócio da classe Pessoa.

Listagem 14. Código de chamada direta ao método calcularIMC de Pessoa.

```
01 var pessoa = new Pessoa();
02 pessoa.peso = 65.2;
03 pessoa.altura = 1.75;
04
05 var imc = helper.calcularIMC(pessoa);
```

Todos os recursos e implementações que vimos até agora fazem parte de um contexto que está longe da experiência direta com o usuário, isto é, precisam passar antes pelos desenvolvedores consumidores para, assim, serem abstraídos pelos respectivos sistemas finais.

Os serviços constituem uma parte fundamental de quase todo sistema web atualmente, não só por fornecer uma ponte fácil de acesso a aplicações de diferentes finalidades e até mesmo tecnologias, mas também por definir a “cara” da sua aplicação, fazer parte da sua assinatura. Quando criar os seus serviços, leve sempre em consideração os conselhos que aprendeu aqui, mas, sobretudo, aprenda a analisar os seus próprios erros.

Obviamente, os mesmos serviços, fábricas, diretivas e módulos que vimos precisam estar atrelados a uma aplicação real para funcionarem em sua totalidade e esse pode e deve ser o seu próximo passo agora. Crie uma aplicação com as tecnologias que

desejar e mixe-a com os serviços aqui criados. Cada aplicação e cada tecnologia traz consigo um background de implementação próprio, e você deve se adaptar a ele com o tempo e com o acúmulo de experiências.

No mais, não deixe de se inteirar também sobre o AngularJS e suas vertentes. Ele constitui um framework que se atualiza muito rapidamente, então os conceitos vistos aqui não podem parar de serem sempre renovados, em vista de um contínuo aprimoramento do seu aprendizado e das suas habilidades como desenvolvedor AngularJS. Então, mãos à obra e bons estudos!

Autor



Júlio Sampaio

É analista de sistema e entusiasta da área de Tecnologia da Informação. Atualmente é consultor na empresa Visagio, trabalhando em projetos de desenvolvimento de sistemas estratégicos, é também instrutor JAVA. Possui conhecimentos e experiência em áreas como Engenharia de Software e Gerenciamento de Projetos, tem também interesse por tecnologias relacionadas ao front-end web.



Links:

Página de download oficial do AngularJS.

<https://docs.angularjs.org/misc/downloading>

Página sobre o padrão “Christian Heilmann’s Revealing Module Pattern”.

<http://christianheilmann.com/2007/08/22/again-with-the-module-pattern-revealing-something-to-the-world/>

Bootstrap e jQuery: Criando um PDV Web – Parte2

ESTE ARTIGO FAZ PARTE DE UM CURSO

Para dar continuidade à construção do PDV será desenvolvido um modelo DER para o banco de dados e a partir deste implementaremos o serviço REST com PHP e o Slim framework. No front-end será criado um layout moderno e responsivo com Bootstrap, onde faremos uso extensivo de JavaScript e jQuery para a implementação das regras de negócio juntamente às requisições assíncronas via Ajax feitas ao servidor remoto para obter acesso aos dados no banco de dados MySQL.

Apesar de estarmos trabalhando essencialmente com os frameworks na camada cliente, será necessária também a utilização de uma infraestrutura de back-end para expor serviços baseados no REST, o que permitirá que os mesmos possam ser consumidos no lado cliente da aplicação, por intermédio, essencialmente, do jQuery e do Ajax. O serviço REST retornará dados no formato JSON, que é um padrão do JavaScript e pode ser manipulado facilmente com jQuery.

O fluxo de execução do nosso PDV Web consiste, basicamente, em um usuário realizar o login no sistema, abrir uma venda, informar o cliente, vender itens que irão compor os dados da venda como um todo e por fim encerrar a venda. Para o desenvolvimento do sistema será criada uma página similar ao um software PAFECF de supermercado, o qual se caracteriza por detalhar os itens vendidos e os totalizar ao final da venda.

DER para o banco de dados MySQL

No desenvolvimento de qualquer sistema é necessário o máximo de documentação possível para que a equipe de desenvolvimento possa construir um software sem ambiguidade. No caso do nosso PDV será desenvolvido um DER para que possamos criar o banco de dados no SGDB MySQL e, a partir do mesmo, nortear a forma como implementaremos todo o back-end e front-end da aplicação.

Fique por dentro

Este artigo finaliza a construção do nosso PDV Web para vendas de produtos utilizando os frameworks Bootstrap e jQuery no front-end e o PHP, MySQL e Slim Framework no back-end. Inicialmente, o leitor poderá entender como funcionam todas essas tecnologias em conjunto e atreladas a regras de negócio próximas da realidade de aplicações corporativas. Verá também a maioria das classes e métodos presentes na API dessas plataformas, e como eles podem te ajudar a simplificar o processo de criação de CRUDs e implementações comuns a esse tipo de sistema. Além disso, configurações de requisições assíncronas usando Ajax também serão alvo deste artigo, o que trará um ar mais profissional ao aplicativo final.

O DER será composto por cinco tabelas:

- **produto**, que irá armazenar as informações de todos os produtos a serem vendidos;
- **cliente**, pois cada venda deverá ser realizada para um cliente em específico;
- **usuario**, para que o funcionário caixa possa logar no sistema e termos, automaticamente, um histórico das vendas efetuadas por ele;
- **venda_cabecalho**, para armazenar informações genéricas sobre as vendas;
- E por fim **itens_venda**, que irá armazenar os dados dos itens vendidos para o cliente com as respectivas quantidades, preço unitário e subtotal, que, por sua vez, serão úteis quando precisarmos formar um cupom não-fiscal.

Para uma melhor compreensão do DER, analise as tabelas e seus campos presentes na **Figura 1**.

Vejamos alguns detalhes importantes sobre as tabelas e campos que formam o DER:

- **Tabela Usuário:** Esta tabela tem como objetivo armazenar todos os usuários que farão uso do sistema. No caso específico do PDV, serão os caixas que efetuarão login para realizar vendas aos clientes. Esta tabela tem vários campos como id do usuário, nome, login, senha (que será armazenada criptografada no banco),

campo ativo para inativar o usuário caso seja necessário e data de cadastro.

• **Tabela Cliente:** A venda de produtos no PDV só será permitida se for informado um cliente no ato da abertura da venda. Neste caso, esta tabela irá servir para o cadastro de clientes. Os campos desta tabela são auto descriptivos e não necessitam de maiores detalhes.

• **Tabela Produto:** Esta tabela tem como objetivo armazenar todos os produtos que poderão ser vendidos pela empresa. O campo id_produto irá armazenar o código identificador do produto (que será autogerado); o campo nome serve para informar a marca do produto e será apresentado na página no ato da venda; o campo descricao irá guardar um detalhamento maior sobre o produto; o campo imagem irá salvar a foto do produto na página; dentre outros.

• **Tabela Venda Cabeçalho:** Esta tabela irá armazenar os dados das vendas realizadas, neste caso o código da venda, código do usuário caixa, código do cliente que realizou a compra, quantidade de itens vendidos, valor total da venda, status da venda (que pode ser finalizada, cancelada, em andamento e outros), e por fim a data e hora que ocorreu a venda.

• **Tabela Itens Venda:** Esta tabela tem como objetivo gravar os dados de cada item vendido, este que representa um produto em específico. Nesta tabela será armazenada a sequência de itens vendidos, valor unitário, subtotal e status do item vendido.

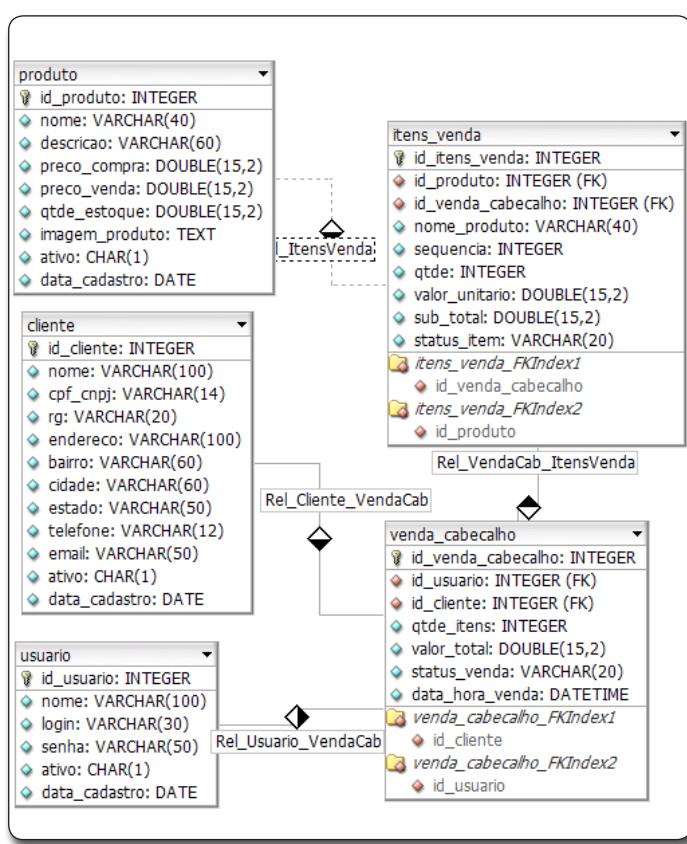


Figura 1. DER do banco de dados PDV Web

Banco de dados MySQL

Para armazenar todos os dados iremos utilizar o SGDB MySQL, sendo primeiro necessário gerar o script SQL através do próprio DBDesigner. Para gerá-lo, siga as seguintes opções: Clique no menu *File > Export > SQL create script*, logo em seguida uma janela irá abrir conforme apresentado na **Figura 2** (essa mesma figura já apresenta o gerenciador do MySQL - o SQLyog - com o banco de dados e as tabelas do PDV Web criados). No campo Target Data Base selecione a opção MySQL e por fim clique no botão *Copy Script to Clipboard*.

```

CREATE TABLE produto (
    id_produto INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
    nome VARCHAR(40) NULL ,
    descricao VARCHAR(60) NULL ,
    preco_compra DOUBLE(15,2) NULL ,
    preco_venda DOUBLE(15,2) NULL ,
    qtde_estoque DOUBLE(15,2) NULL ,
    imagem_produto TEXT NULL ,
    ativo CHAR(1) NULL ,
    data_cadastro DATE NULL ,
    PRIMARY KEY(id_produto));
CREATE TABLE usuario (
    id_usuario INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
    nome VARCHAR(100) NULL ,
    login VARCHAR(30) NULL ,
    senha VARCHAR(50) NULL ,
    ativo CHAR(1) NULL );

```

1 Messages 2 Table Data 3 Info
5 queries executed, 5 success, 0 errors, 0 warnings
Query: CREATE TABLE produto (id_produto INTEGER UNSIGNED
0 row(s) affected
Execution Time : 0.660 sec
Transfer Time : 0.001 sec
Total Time : 0.661 sec

Query: CREATE TABLE usuario (id_usuario INTEGER UNSIGNED
0 row(s) affected
All

Figura 2. Banco de dados PDV Web gerado pelo DBDesigner e SQLyog

Essa opção permite que você cole o conteúdo do comando em qualquer editor de texto. Para isso, crie um banco de dados chamado "pdv_web", abra uma janela de query na ferramenta e cole o script SQL. Veja na **Listagem 1** o script gerado. Após isso, basta executar o mesmo.

Download e instalação do XAMPP

Antes da codificação do PDV Web é necessário baixar o XAMPP, um pacote que já tem o PHP, o servidor Apache e outras ferramentas para controle e administração de websites. Você pode baixar o XAMPP através do link de download do mesmo presente na seção **Links** deste artigo.

Para instalar, basta executar o instalador e ir avançando entre as telas sem alterar as opções já selecionadas. Após a conclusão da instalação, execute o *XAMPP Control Panel* (uma das opções que vem junto do instalador) e aguarde até que o ícone de

administração ao lado do relógio do Windows apareça. Clique duas vezes no ícone para abrir o painel de controle conforme apresentado na **Figura 3**.

Perceba que estamos executando apenas o módulo Apache (destacado na cor verde) que roda na porta 8090.



Figura 3. Painel de controle do XAMPP

O leitor pode optar se deseja executar o MySQL pelo XAMPP ou se deseja instalar um SGBD à parte, como o Workbench (usaremos esta opção pela facilidade que ela apresenta). Caso não tenha instalado o MySQL em sua máquina não se preocupe, pois no ato da instalação do XAMPP o mesmo também foi instalado.

Outro detalhe importante a respeito do XAMPP é a porta que o servidor Apache usa por padrão, a porta 80. Alguns programas, como o Skype, também usam a mesma porta, logo quando você tentar iniciar o servidor poderá ter problemas de barramento. Para resolver isso, basta parar o programa via Gerenciador de Tarefas e tudo funcionará normalmente.

Download e configuração do Slim framework

Iniciaremos a implementação do PDV pelo desenvolvimento do serviço baseado em REST que o front-end irá consumir via Ajax com o auxílio do jQuery. Para isso, efetue o download do zip do Slim Framework através do site que consta na seção [Links](#).

Para usá-lo basta descompactar os arquivos e copiar para dentro da pasta da aplicação web, neste caso a pasta do PDV Web. Essa pasta, assim como toda a estrutura do projeto, deverá ser criada

Listagem 1. Script SQL para gerar o banco de dados PDV Web

```
01 CREATE TABLE produto (
02 id_produto INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
03 nome VARCHAR(40) NULL ,
04 descricao VARCHAR(60) NULL ,
05 preco_compra DOUBLE(15,2) NULL ,
06 preco_venda DOUBLE(15,2) NULL ,
07 qtde_estoque DOUBLE(15,2) NULL ,
08 imagem_produto TEXT NULL ,
09 ativo CHAR(1) NULL ,
10 data_cadastro DATE NULL ,
11 PRIMARY KEY(id_produto));
12
13 CREATE TABLE usuario (
14 id_usuario INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
15 nome VARCHAR(100) NULL ,
16 login VARCHAR(30) NULL ,
17 senha VARCHAR(50) NULL ,
18 ativo CHAR(1) NULL ,
19 data_cadastro DATE NULL ,
20 PRIMARY KEY(id_usuario));
21
22 CREATE TABLE cliente (
23 id_cliente INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
24 nome VARCHAR(100) NULL ,
25 cpf_cnpj VARCHAR(14) NULL ,
26 rg VARCHAR(20) NULL ,
27 endereco VARCHAR(100) NULL ,
28 bairro VARCHAR(60) NULL ,
29 cidade VARCHAR(60) NULL ,
30 estado VARCHAR(50) NULL ,
31 telefone VARCHAR(12) NULL ,
32 email VARCHAR(50) NULL ,
33 ativo CHAR(1) NULL ,
34 data_cadastro DATE NULL ,
35 PRIMARY KEY(id_cliente));
36
37 CREATE TABLE venda_cabecalho (
38 id_venda_cabecalho INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
39 id_usuario INTEGER UNSIGNED NOT NULL ,
40 id_cliente INTEGER UNSIGNED NOT NULL ,
41 qtde_itens INTEGER UNSIGNED NULL ,
42 valor_total DOUBLE(15,2) NULL ,
43 status_venda VARCHAR(20) NULL ,
44 data_hora_venda DATETIME NULL ,
45 PRIMARY KEY(id_venda_cabecalho),
46 INDEX venda_cabecalho_FKIndex1(id_cliente) ,
47 INDEX venda_cabecalho_FKIndex2(id_usuario),
48 FOREIGN KEY(id_cliente)
49 REFERENCES cliente(id_cliente)
50 ON DELETE NO ACTION
51 ON UPDATE NO ACTION,
52 FOREIGN KEY(id_usuario)
53 REFERENCES usuario(id_usuario)
54 ON DELETE NO ACTION
55 ON UPDATE NO ACTION);
56
57 CREATE TABLE itens_venda (
58 id_itens_venda INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
59 id_produto INTEGER UNSIGNED NOT NULL ,
60 id_venda_cabecalho INTEGER UNSIGNED NOT NULL ,
61 sequencia INTEGER UNSIGNED NULL ,
62 qtde INTEGER UNSIGNED NULL ,
63 valor_unitario DOUBLE(15,2) NULL ,
64 sub_total DOUBLE(15,2) NULL ,
65 status_item VARCHAR(20) NULL ,
66 PRIMARY KEY(id_itens_venda) ,
67 INDEX itens_venda_FKIndex1(id_venda_cabecalho) ,
68 INDEX itens_venda_FKIndex2(id_produto),
69 FOREIGN KEY(id_venda_cabecalho)
70 REFERENCES venda_cabecalho(id_venda_cabecalho)
71 ON DELETE NO ACTION
72 ON UPDATE NO ACTION,
73 FOREIGN KEY(id_produto)
74 REFERENCES produto(id_produto)
75 ON DELETE NO ACTION
76 ON UPDATE NO ACTION);
```

dentro do diretório "C:\xampp\htdocs", que é o local padrão para salvar os arquivos de websites quando se utiliza o servidor Apache com XAMPP.

Nota

Para evitar desperdício de espaço, descompacte o arquivo zip do Slim Framework em outro local de sua máquina e copie apenas a pasta Slim e o arquivo .htaccess para dentro da pasta pdv da aplicação PDV Web localizada dentro do diretório htdocs.

Após realizado este procedimento, ainda é necessário configurar o arquivo htaccess do Slim para apontar para o arquivo PHP que terá as rotas de implementação do mesmo referente ao serviço REST da aplicação. Defina o código do arquivo htaccess conforme apresentado na **Listagem 2** e logo em seguida crie um arquivo dentro da pasta pdv denominado `slim_rest.php`. Este arquivo irá conter toda a configuração e implementação do serviço a ser utilizado pelo front-end da aplicação.

Listagem 2. Configuração do arquivo htaccess do Slim framework.

```
01 RewriteEngine On
02
03 RewriteCond %{REQUEST_FILENAME} !-d
04 RewriteCond %{REQUEST_FILENAME} !-f
05 RewriteRule ^ slim_rest.php [QSA,L]
```

A configuração mais importante está na linha 5, que aponta para o arquivo recém-criado `slim_rest.php`. Ela é importante pois esse arquivo representará toda a implementação do serviço baseado no padrão arquitetural REST.

Serviço REST para autenticação de usuários

O primeiro passo para a implementação do serviço que proverá a autenticação de usuário no sistema será a criação do arquivo `Bd.php` que terá toda a configuração necessária para o acesso ao banco de dados. Crie este arquivo dentro para pasta pdv e adicione ao mesmo o código da **Listagem 3**.

A função `getConnection` retorna um objeto da classe PDO que é responsável por encapsular o acesso ao banco de dados. Sempre que for necessário realizar uma operação no banco de dados basta chamar este método e realizar a operação. Vejamos mais alguns detalhes:

- **Linha 3:** Esta linha faz a chamada à função `session_start()` que diz ao PHP que iremos trabalhar com a sessão no lado do servidor e que informações persistentes deverão ser sempre guardadas entre requisições HTTP. Também faremos uso da sessão para armazenar os dados do usuário logado no sistema.
- **Linhas 6 a 9:** Aqui definimos algumas variáveis para armazenar o endereço do servidor MySQL, nome de usuário, senha e o nome do banco de dados.
- **Linha 10:** Aqui é criado um objeto denominado `dbh` que recebe uma instância da classe PDO do PHP. Perceba que foram passados como parâmetros todos os dados necessários para conectar ao banco de dados MySQL.

- **Linha 11:** Nesta linha são configurados alguns atributos para exibir um relatório de erro e exceções caso ocorra algum problema durante o acesso ao banco de dados. Como estaremos trabalhando de forma assíncrona com o serviço REST, será necessário repassar ao cliente o máximo de informação possível caso isso aconteça. Desta forma, será mais fácil identificar erros e tomar as devidas providências para solução.

Listagem 3. Arquivo de configuração para acesso ao banco de dados MySQL.

```
01 <?php
02
03 session_start();
04
05 function getConnection() {
06     $dbhost="localhost";
07     $dbuser="root";
08     $dbpass="root";
09     $dbname="pdv_web";
10     $dbh = new PDO("mysql:host=$dbhost;dbname=$dbname", $dbuser, $dbpass);
11     $dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
12     return $dbh;
13 }
```

Com o arquivo de acesso ao banco de dados devidamente configurado agora podemos implementar o serviço de autenticação de usuário. Para realizar este procedimento, abra o arquivo `slim_rest.php` criado anteriormente e implemente no mesmo o código da **Listagem 4**.

Ela, basicamente, inicia e configura algumas informações do Slim framework, implementando também duas rotas de acesso ao serviço de autenticação:

- A rota `/pdv/autenticarUsuario` para autenticar o usuário;
- E `/pdv/logout` para finalizar a sessão do usuário no sistema.

Essa listagem apresenta várias características importantes, a saber:

- **Linhas 2 e 3:** Aqui é utilizado o comando `require` para que possamos ter acesso às funções definidas no arquivo `Bd.php`.
- **Linhas 5 a 7:** Configuramos o registro de carregamento do Slim, através da criação de um objeto chamado `$app` que, por sua vez, recebe uma instância do próprio Slim. Por fim, é chamada a função `add` do Slim para informar ao `SessionCookie` que podemos trabalhar com a sessão em conjunto com o Slim e o PHP.
- **Linha 10:** Nesta linha é definida a primeira rota chamada `"/autenticarUsuario"` do serviço de autenticação de usuários. Perceba que estamos usando a função `post()` do Slim passando o caminho da rota e o objeto `app` como argumentos. Isso significa que se quisermos acessar este serviço, devemos realizar sempre uma requisição POST ao servidor através do protocolo HTTP.
- **Linhas 11 e 13:** Aqui são criadas duas variáveis para recuperar o login e senha da requisição realizada via POST ao servidor. Perceba que foram utilizadas as funções `request()` e `post()` que recuperam os inputs do formulário HTML submetido ao servidor. Logo em seguida, é criada a variável `senhaCriptada` que recebe a

senha criptografada através da chamada à função `md5()` do PHP. Desta forma, podemos consultar a senha na base de dados sem problemas, uma vez que a mesma se encontra criptografada.

- **Linha 16:** Nesta linha é definida a instrução SQL que verifica se o login e senha do usuário existem e conferem com os dados armazenados no banco. Perceba na instrução que os valores `:login` e `:senha` têm dois pontos na frente, isso significa que mais adiante iremos alterar estes valores pelos valores das variáveis que contém os reais valores de login e senha, e, para isso, faremos uso da função `bindParam()`.

- **Linha 18 a 22:** A função `prepare()` recebe a instrução SQL a ser executada no banco. Logo após chamamos a função `bindParam()` para alterar os valores de `:login` e `:senha` e, por fim, a função `execute()` é chamada para realizar a consulta na base de dados.

- **Linhas 24 e 25:** Neste intervalo, primeiramente é criado um objeto chamado `objUsuario` que recebe o resultado da consulta, e em seguida a função `fetchObject()` retorna o registro pertinente ao login e senha do usuário. Por fim, setamos null à variável `db` para inutilizá-la.

- **Linhas 27 a 38:** Validamos a autenticação do usuário e criamos duas sessions para armazenar o ID e o nome do usuário. Observe nas linhas 30 e 33 a utilização da função `redirect()` para redirecionar o usuário a outra página específica.

- **Linha 40 a 46:** A função `get` do Slim cria uma nova rota chamada `"/logout"` para que o usuário possa realizar o logoff no sistema. Esta rota deve ser acessada via requisição GET do HTTP. Usamos também o `unset` e o `session_destroy` para finalizar a sessão do usuário, e por fim é definida a página de redirecionamento no header, chamando o `exit` em seguida.

- **Linha 49:** A função `run()` do Slim é chamada para, de fato, tornar estas rotas acessíveis.

Bootstrap e jQuery

O Bootstrap é um framework que disponibiliza uma série de componentes CSS, JavaScript, ícones, templates, dentre outros recursos que facilitam o desenvolvimento de aplicações web. Você pode baixá-lo através do endereço também disponível na seção [Links](#) deste artigo.

Uma vez na página de download, deve ser baixado o arquivo da primeira opção: *“Compiled and minified CSS, JavaScript, and fonts.”* que tem uma versão minificada que deixa o carregamento das páginas do website com uma melhor performance. Após o download do zip, você deve extrair o mesmo dentro do diretório `pdv` localizado na pasta `htdocs` do XAMPP. Certifique-se de ter extraído as pastas `css`, `js` e `fonts` para dentro de `pdv`.

Nós também precisaremos baixar o arquivo do jQuery. Acessa a página de download na seção [Links](#) e baixe o arquivo referente à versão 1.11.2 e o posicione dentro do diretório `js` localizado na mesma pasta `pdv`.

Crie também um arquivo chamada `javascript_pdv.js`, pois é neste arquivo que colocaremos todo o código JavaScript da aplicação. É através do jQuery que serão realizadas todas as requisições assíncronas via Ajax ao serviço fornecido pelo Slim Framework.

No arquivo de downloads desse artigo, você encontrará também uma pasta chamada `“img”` com algumas imagens de produtos a serem vendidos no PDV, mas o leitor pode ficar à vontade para usar as que quiser.

Para finalizar esta etapa, veja na [Figura 4](#) como deve ficar a estrutura de pastas e arquivos final do nosso projeto.

Estrutura de páginas Front-end e autenticação de usuário

Com o serviço REST para autenticação de usuário rodando no back-end e os devidos frameworks front-end distribuídos, agora é

Listagem 4. Implementação do serviço para autenticação de usuários no sistema.

```
01 <?php
02 require 'Bd.php';
03 require 'Slim/Slim.php';
04
05 \Slim\Slim::registerAutoloader();
06 $app = new \Slim\Slim();
07 $app->add(new \Slim\Middleware\SessionCookie(array('secret'=>'myappsecret')));
08
09 //Início login-----
10 $app->post("/autenticarUsuario", function () use ($app) {
11     $login = $app->request()->post('login');
12     $senha = $app->request()->post('senha');
13     $senhaCriptada = md5($senha);
14
15
16     $sql = "SELECT * FROM usuario WHERE login = :login AND senha = :senha";
17     try {
18         $db = getConnection();
19         $stmt = $db->prepare($sql);
20         $stmt->bindParam("login", $login);
21         $stmt->bindParam("senha", $senhaCriptada);
22         $stmt->execute();
23
24         $objUsuario = $stmt->fetchObject();
25         $db = null;
26
27         if ($objUsuario != null) {
28             $_SESSION['id_usuario'] = $objUsuario->id_usuario;
29             $_SESSION['nome'] = $objUsuario->nome;
30             $app->redirect("/pdv/pdv_web.php");
31         } else {
32             $erro = "Erro na autenticação";
33             $app->redirect("/pdv/pdv_web.php?erro=".$erro);
34         }
35     } catch(PDOException $e) {
36         echo {"error":{"text": $e->getMessage() }};
37     }
38 });
39
40 $app->get("/logout", function () use ($app) {
41     unset($_SESSION['id_usuario']);
42     unset($_SESSION['nome']);
43     session_destroy();
44     header("location: /pdv/pdv_web.php");
45     exit;
46 });
47 //Fim login-----
48
49 $app->run();
50 ?>
```

o momento de criar a estrutura de páginas do projeto, a começar pela página inicial do PDV Web.

Para isso vamos quebrar a página index em três: header, body e footer, desta forma iremos reaproveitar o header e o footer nas demais páginas que futuramente possam ser incluídas ao projeto. Esta estrutura de divisão é uma boa prática no PHP, pois através do comando **require** podemos incluir estruturas completas de código mesclado entre PHP, HTML e JavaScript em outras páginas do projeto, tornando-o mais manutenível.

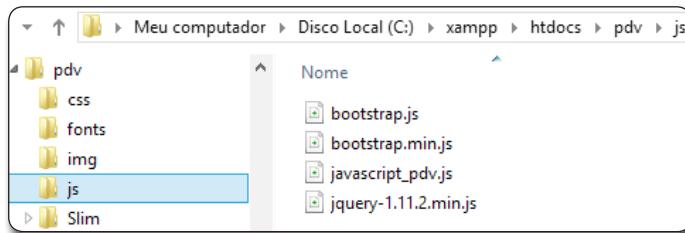


Figura 4. Estrutura de pastas e arquivos do projeto PDV Web

A primeira página a ser criada é a "header.php", que tem com propósito exibir o menu superior para colocar os links da página e também terá embutido o formulário de login para autenticação do usuário. Nesta página, usaremos várias classes do Bootstrap para criar o menu fixo e para que ele se adapte ao tamanho da tela do dispositivo do usuário, com um layout adaptável e responsivo.

Veja na **Listagem 5** como deve ficar a implementação da referida página.

Aqui está toda a implementação necessária para o header do PDV Web juntamente com o formulário de autenticação, porém são necessárias algumas explicações sobre este código. Veja a seguir mais detalhes:

- **Linha 1:** A primeira linha ativa a `session_start()` do PHP e pode, mais à frente, verificar se existe algum usuário logado no sistema.
- **Linha 2:** Aqui temos a tag DOCTYPE que define que o projeto irá utilizar HTML 5.
- **Linha 3:** Define o idioma que será utilizado na página, esta prática é importante pois é verificada pelos motores de busca na internet.
- **Linhas 4 a 7:** Na linha 5 é definido dentro do head o padrão de caracteres a ser utilizado pelo browser para renderizar a página, logo após na linha 6 é definido o *meta viewport* para que o Bootstrap possa capturar a informação sobre tipo e tamanho do dispositivo do cliente que está visualizando a página e assim possa aplicar os recursos de responsividade.
- **Linha 8:** Esta linha realiza uma referência ao arquivo bootstrap.min.css que é um stylesheet que contém todo o código CSS do Bootstrap para ser utilizado na página.
- **Linha 11 e 49:** Aqui é definido o bloco HTML da barra de navegação utilizando as classes CSS navbar, navbar-inverse e navbar-fixed-top do Bootstrap, que irão deixar a navegação fixa no topo da página. É aqui que você deve colocar os links de navegação de sua página conforme exemplo do link fake da linha 27.

Listagem 5. Implementação da página "header.php".

```
01 <?php session_start(); ?>
02 <!DOCTYPE html>
03 <html lang="pt-br">
04 <head>
05   <meta charset="utf-8">
06   <meta name="viewport" content="width=device-width, initial-scale=1">
07   <title>PDV Web</title>
08   <link href="css/bootstrap.min.css" rel="stylesheet">
09 </head>
10 <body>
11   <nav class="navbar navbar-inverse navbar-fixed-top">
12     <div class="container">
13       <div class="navbar-header">
14         <button type="button" class="navbar-toggle collapsed"
15             data-toggle="collapse" data-target="#navbar"
16             aria-expanded="false" aria-controls="navbar">
17           <span class="sr-only">PDV Web</span>
18           <span class="icon-bar"></span>
19           <span class="icon-bar"></span>
20           <span class="icon-bar"></span>
21         </button>
22         <a class="navbar-brand" href="#">PDV Web</a>
23     </div>
24   <div id="navbar" class="navbar-collapse collapse">
25     <?php
26     if(isset($_SESSION['id_usuario'])){
27       echo "<ul class='nav navbar-nav'><li><a href='#'> Seus links aqui.</a>
28       </li></ul>";
29       echo "<label id='id_usuario' hidden='hidden'>".$_SESSION['id_usuario']."</label>";
30       echo "<div class='navbar-form navbar-right'>
31         <div class='form-group'>
32           <label for='usuarioLogado' style='color:#FFFFFF'>
33             Olá, ".$_SESSION['nome']."'!</label>
34           <a class='btn btn-success' href='/pdv/logout'>Sair</a>
35         </div>
36       </div>";
37     }else{
38       echo "<form class='navbar-form navbar-right' method='post'
39             action='/pdv/autenticarUsuario'>
40         <div class='form-group'>
41           <input name='login' type='text' placeholder='login'
42             class='form-control'>
43         </div>
44         <div class='form-group'>
45           <input name='senha' type='password' placeholder='senha'
46             class='form-control'>
47         </div>
48         <button type='submit' class='btn btn-success'>Logar</button>
49       </form>";
50     }
51   ?>
52   </div>
53 </div>
54 </body>
```

- **Linha 26:** Nesta linha tem uma condicional if que, através da função `isset` do PHP, verifica se existe um usuário logado no sistema com o uso do comando `$_SESSION['id_usuario']`, onde "`id_usuario`" é o nome dado à sessão que contém o valor do ID do mesmo. Caso positivo, será executado o bloco de código entre as linhas 27 e 34 que apresenta o nome do usuário logado conforme linha 31. Na sequência, na linha 32, foi inserido um link que

aponta para o serviço "/pdv/logout" para que o usuário possa fazer logoff no sistema.

- **Linhas 35 a 45:** Contém o bloco else, ou seja, caso não exista nenhum usuário logado no sistema será exibido um formulário HTML para autenticar via POST junto ao serviço de rota '/pdv/autenticarUsuario'.

Com a página header criada já temos o sistema de login disponível e em funcionamento, porém é necessário ter o banco de dados populado com dados para testes. Assim sendo, você deve inserir no MySQL o banco de dados contido nos fontes do projeto (caso use o que está disponível no arquivo de fontes, use o login: madson e a senha: 123), para realizar o teste de autenticação de usuário no PDV Web, conforme pode ser visto na **Figura 5**.

Agora é necessário criar a página footer.php que exibirá as informações do rodapé e realizará referências aos arquivos JavaScript que serão necessários utilizar no PDV Web. Veja na **Listagem 6** como deve ficar a implementação dessa página.

No intervalo entre as linhas 7 e 9 temos as referências aos arquivos JavaScript do jQuery, Bootstrap e o js do PDV que será implementado mais adiante. Observe a sequência das referências aos arquivos js, pois existe uma dependência entre eles.

É importante colocar a referências aos arquivos após o último elemento da tag body, desta forma a página terá uma melhor performance no carregamento, uma vez que o browser já terá renderizado todos os elementos DOM da página. Isso é importante, pois evita erros nos códigos JavaScript que buscam elementos na página sem que ela tenha sido renderizada.

Página principal do PDV Web

Vamos criar agora uma nova página PHP chamada 'pdv_web.php'. A mesma terá toda a estrutura de um layout para um PDV

através do uso do Bootstrap e seu sistema de grids que é dividido em linhas e colunas. Primeiramente, vejamos a estrutura estática da página em HTML e CSS para que depois possamos implementar os comportamentos dinâmicos através do JavaScript, jQuery e Ajax.

Em vista do tamanho da página pdv_web.php, dividiremos a sua criação em duas partes, para melhor entendimento. Veja na **Listagem 7** a implementação inicial do arquivo.

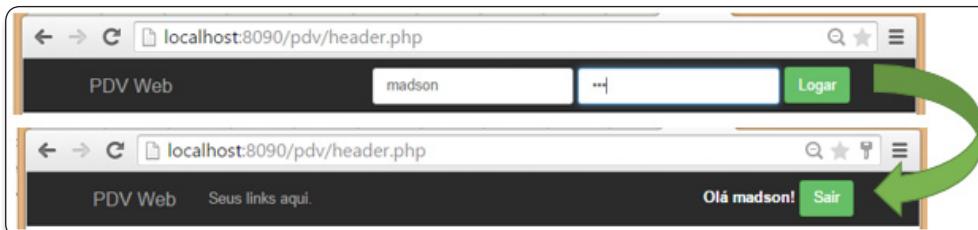
Listagem 7. Implementação da página "pdv_web.php".

```
01 <?php require 'header.php'?>
02 <br> <br>
03 <?php
04 if(isset($_SESSION['id_usuario'])){
05 echo "<div class='container'>
06
07 <div class='panel panel-default'>
08   <div class='panel-heading'><h3>Sistema de vendas
09     <span class='label label-default'>PDV Web</span></h3></div>
10   <div class='panel-body'>
11     <div class='row'>
12       <div class='col-md-12'>
13         <select class='btn btn-default' id='select_cliente'></select>
14         <button class='btn btn-default btn-warning'
15           id='btn_iniciar_venda'>Iniciar venda</button>
16         <button class='btn btn-default btn btn-danger disabled='true'
17           id='btn_encerrar_venda' onclick='encerrarVenda()'>
18           Encerrar venda</button>
19         <span class='label label-default'><label>STATUS:</label></span>
20         &nbsp;
21         <label id='status_venda'>Encerrada</label>&nbsp;
22         <span class='label label-default'><label>CÓD. VENDA:</label>
23         </span> &nbsp;
24         <label id='id_venda_cabecalho'></label> &nbsp;
25         <span class='label label-default'><label>CLIENTE:</label></span>
26         &nbsp;
27         <label id='codigo_cliente'></label>&nbsp;
28         <label id='cliente_nome'></label>
29       </div>
30     </div>
31   </div>
32 </div>
33 </div>
```

Vejamos a seguir os detalhes do código:

- **Linha 1:** Aqui é comando require do PHP é utilizado para embutir na página pdv_web.php todo o código contido na página header. É desta forma que funciona o reaproveitamento de código no PHP.
- **Linha 4:** Aqui é feita a condição que comentamos antes, sobre verificar se o usuário está ou não logado.
- **Linha 5:** Div que utiliza a classe container do Bootstrap para organizar as coisas no layout.
- **Linhas 7 a 9:** Painel do Bootstrap construído através das classes panel, panel-default, panel-heading e panel-body que servem para definir o escopo do design da página.
- **Linha 12:** Definimos o elemento select que irá compor a lista de clientes para que o usuário caixa possa escolher um no ato da abertura de uma venda no PDV Web.

Figura 5. Sistema de autenticação de usuário em funcionamento



- Linha 13:** Aqui foi criado um botão que será responsável pelo disparo do evento click para iniciar uma venda no PDV.
- Linha 14:** Novo botão para encerrar a venda, porém o mesmo só estará ativo para click quando tiver uma venda em andamento.
- Linha 15 a 21:** Definimos vários labels para informar os dados da venda em andamento, como o status da venda, código e nome do cliente.

Em seguida, precisamos definir também o conteúdo HTML da página com a estrutura de navegação. Veja na **Listagem 8** o código que você adicionar ao fim do arquivo.

Vejamos algumas observações acerca do código:

- Linhas 1 a 9:** Div que irá armazenar os itens da venda.
- Linhas 10 a 26:** Os campos preço unitário e valor total do item serão preenchidos automaticamente via JavaScript. Perceba na linha 50 um botão com ID "btnAdicionarItem" que será responsável por disparar o evento para vender o item e adicionar o mesmo no cupom não fiscal que será definido mais adiante na listagem.
- Linhas 26 a 30:** Definimos uma tag img com ID para que possamos exibir as imagens dos produtos que serão vendidos.
- Linhas 34 a 42:** Mais uma coluna para armazenar um contador de itens vendidos que ficará oculto para o usuário conforme linha 62, logo em seguida na linha 64 foi definida uma tabela que irá exibir todos os itens vendidos durante a venda. É importante citar que esta

tabela será preenchida via JavaScript e terá um botão em cada item vendido possibilitando o cancelamento do mesmo pelo usuário.

- Linhas 44 a 53:** Aqui é criado a última row da página do PDV web, esta linha contém apenas uma coluna conforme linha 71 com o papel de exibir a informação referente ao total geral da venda em andamento conforme componentes das linhas 74 e 75. O label de ID 'total_geral' presente na linha 75 será preenchido automaticamente via JavaScript conforme forem sendo inseridos itens na venda em andamento.
- Linha 63:** Comando require para inserir o código presente no arquivo 'footer.php'.

A estrutura HTML do PDV Web já está pronta, portanto você pode executar o projeto e verificar o resultado conforme apresentado na **Figura 6**. Perceba que o PDV ainda não ganhou vida, pois ainda faremos a implementação JavaScript e jQuery no arquivo javascript_pdv.js para o tornar funcional.



Figura 6. Layout do PDV Web em execução

Listagem 8. Implementação da estrutura HTML e de navegação.

```

01 <div class='panel panel-default' id='div_venda_itens' hidden='hidden'>
02   <div class='panel-body'>
03     <div class='row'>
04       <div class='col-md-12'>
05         <div class='breadcrumb'>
06           <h3><label for='nomeProduto' id='nomeProduto'></label></h3>
07         </div>
08       </div>
09     </div>
10   <div class='row'>
11     <div class='col-md-3'>
12       <div class='breadcrumb'>
13         <div class='form-group'>
14           <label for='codigo_produto'>Código produto:</label>
15           <input type='number' class='form-control' id='codigo_produto'>
16           <label for='qtde'>Quantidade:</label>
17           <input type='number' class='form-control' id='qtde_item'>
18           <label for='preco'>Preço unitário:</label>
19           <input type='number' class='form-control' id='preco_unitario' disabled='true'>
20           <label for='codigo_produto'>Valor total:</label>
21           <input type='number' class='form-control' id='valor_total_item' disabled='true'>
22         </div>
23       <button class='btn btn-primary' id='btnAdicionarItem'>
24         Adicionar ITEM</button>
25       </div>
26     </div>
27   <div class='col-md-2'>
28     <div class='breadcrumb'>
29       <img src='class=img-responsive' id='img_produto' alt='Responsive image'>
30     </div>
31   </div>
32 </div>
33
34   <div class='col-md-7'>
35     <div class='breadcrumb'>
36       <input type='number' id='contadorSeqItemVenda' disabled='true' value='1' hidden='hidden'>
37       <!--<label for='cupom'>ITENS CUPOM:</label>-->
38       <table class='table table-striped' id='tabela_cupom'>
39         </table>
40       </div>
41     </div>
42   </div>
43
44   <div class='row'>
45     <div class='col-md-12'>
46       <div class='breadcrumb'>
47         <h3>
48           <label for='rotulo_total_geral'>TOTAL GERAL:</label>
49           <label id='total_geral' disabled='true'>
50             </h3>
51           </div>
52         </div>
53       </div>
54     </div>
55   </div>
56 </div>
57 </div>"};
58 else{
59   echo "<br/> Você deve realizar o login.";
60 }
61 ?>
62
63 <?php require 'footer.php'?>
64
65 <script type="text/javascript">
66   $(document).ready(function() {
67     carregarNoLoadPagina();
68   });
69 </script>

```

Listagem de clientes com jQuery e Ajax

Comecemos a dinamização do PDV pelo preenchimento do select que irá exibir a lista de clientes para que o usuário caixa possa selecionar um cliente e iniciar uma venda.

Primeiramente, implemente o código da **Listagem 9** dentro do arquivo `slim_rest.php`, uma vez que ele será responsável por expor o serviço para retornar a listagem de clientes cadastrados no banco de dados e proporcionar a consulta de clientes por ID.

Listagem 9. Implementação do serviço REST para listagem de clientes e consulta por ID.

```
01 $app->get('/getCliente/:id', function ($id) use ($app) {  
02  
03   $sql = "select * FROM cliente WHERE id_cliente = :id";  
04   try {  
05  
06     $db = getConnection();  
07     $stmt = $db->prepare($sql);  
08     $stmt->bindParam(":id", $id);  
09     $stmt->execute();  
10  
11     $cliente = $stmt->fetchObject();  
12     $db = null;  
13     echo json_encode($cliente, JSON_UNESCAPED_UNICODE);  
14   } catch(PDOException $e) {  
15     echo '{"error":{"text":' . $e->getMessage() .}}';  
16   }  
17 });  
18  
19 $app->get('/getClientes', function () use ($app) {  
20  
21   $sql = "select * FROM cliente ORDER BY nome";  
22   try {  
23     $db = getConnection();  
24     $stmt = $db->query($sql);  
25     $clientes = $stmt->fetchAll(PDO::FETCH_OBJ);  
26     $db = null;  
27     echo json_encode($clientes, JSON_UNESCAPED_UNICODE);  
28   } catch(PDOException $e) {  
29     echo '{"error":{"text":' . $e->getMessage() .}}';  
30   }  
31 });
```

Agora abra o arquivo `javascript_pdv.js` contido dentro da pasta `js` e implemente o código JavaScript e jQuery descrito na **Listagem 10**.

Vejamos maiores detalhes da implementação desta listagem:

- **Linhas 1 a 6:** Foi definida uma função `carregarNoLoadPagina()` que é chamada na página `pdv_web.php` após o carregamento total da mesma no browser. Dentro desta função colocaremos todas as funções necessárias para capturar os eventos que possam ocorrer durante a execução do PDV. Na linha 4 é chamada outra função, `carregarClientesSelect()`, que se encarrega de popular a lista de clientes no select.
- **Linha 12 a 20:** Aqui é utilizada a função `getJSON` do jQuery para acessar o serviço REST no endereço `/pdv/getClientes` e obter a listagem de clientes no formato JSON. Logo em seguida, é realizado um for na listagem de clientes para obter o ID e nome deles e assim popular o select conforme apresentado na linha 17, tudo isso através do uso da função `html()` do jQuery.

Execute novamente o projeto e veja o resultado conforme apresentado na **Figura 7**.



Figura 7. Select de clientes populado com jQuery via Ajax

Listagem 10. Implementação da lista de clientes.

```
01 function carregarNoLoadPagina(){  
02  
03   //Carregar lista de clientes  
04   carregarClientesSelect();  
05  
06 }  
07  
08 //Fim carregarNoLoadPagina()  
09  
10 function carregarClientesSelect(){  
11   $(function(){  
12     $.getJSON("/pdv/getClientes", function(j){  
13       var options = "<option value='0'>Selecione um cliente</option>";  
14       for (var i = 0; i < j.length; i++) {  
15         options += '<option value="' + j[i].id_cliente + "'>' + j[i].nome +  
16           '</option>';  
17       }  
18       $("#select_cliente").html(options);  
19     });  
20 }
```

Vendas no PDV

Agora é hora de iniciar uma venda através do botão “Iniciar Venda”. Para realizar este procedimento, primeiramente é necessário criar o serviço REST que irá auxiliar na inserção e finalização de uma venda no banco de dados. Implemente o código da **Listagem 11** dentro do arquivo `slim_rest.php` para tornar alguns serviços disponíveis que precisamos usar no front-end.

Observe que temos duas rotas que apontam para o serviço: uma que irá abrir uma nova venda no banco de dados conforme linha 1 e outra rota para finalizar uma venda conforme linha 25.

Precisamos também incrementar a funcionalidade propriamente dita do botão de iniciar venda, logo será necessário colocar o código da **Listagem 12** dentro da função `carregarNoLoadPagina`. O código desta listagem tem o papel de escutar o click do botão `btn_iniciar_venda` conforme linha 2. Caso exista um cliente selecionado, ela deve chamar a função `iniciarNovaVenda()` presente na linha 6.

Esse código apenas escuta o click do botão, porém é necessário iniciar uma venda através de uma chamada via Ajax ao serviço REST que irá inserir e definir a abertura de uma nova venda no banco de dados. Veja a implementação da função `iniciarNovaVenda` na **Listagem 13**.

Vejamos a seguir alguns detalhes:

- Linhas 2 a 7:** É desativado o botão para iniciar uma venda e ativado o botão para encerrar a venda em andamento. Perceba ainda que é definida a exibição de vários componentes na página para dar início ao processo de venda de produtos. Por fim, capturamos o código do cliente selecionado para iniciar a venda conforme consta na linha 7.

Listagem 11. Implementação do serviço para iniciar e finalizar uma venda.

```

01 $app->post('/adicionarVendaCabecalho', function () use ($app) {
02
03     $id_usuario = $app->request->post('id_usuario');
04     $id_cliente = $app->request->post('id_cliente');
05     $status_venda = $app->request->post('status_venda');
06     $sql = "INSERT INTO venda_cabecalho (id_usuario, id_cliente, status_venda,
07     data_hora_venda) VALUES (:id_usuario, :id_cliente, :status_venda,
08     :data_hora_venda)";
09     try {
10         $db = getConnection();
11         $stmt = $db->prepare($sql);
12         $stmt->bindParam("id_usuario", $id_usuario);
13         $stmt->bindParam("id_cliente", $id_cliente);
14         $stmt->bindParam("status_venda", $status_venda);
15         $dataAtual = date("Y-m-d H:i");
16         $stmt->bindParam("data_hora_venda", $dataAtual);
17         $stmt->execute();
18         $id_venda_cabecalho = $db->lastInsertId();
19         echo $id_venda_cabecalho;
20     } catch(PDOException $e) {
21         echo '{"error":{"text":' . $e->getMessage() . "}}";
22     }
23 });
24
25 $app->post('/updateVendaCabecalho', function () use ($app) {
26
27     $id_venda_cabecalho = $app->request->post('id_venda_cabecalho');
28     $qtde_itens = $app->request->post('qtde_itens');
29     $valor_total = $app->request->post('valor_total');
30     $status_venda = $app->request->post('status_venda');
31     $sql = "UPDATE venda_cabecalho SET qtde_itens = :qtde_itens,
32     valor_total = :valor_total, status_venda = :status_venda WHERE
33     id_venda_cabecalho = :id_venda_cabecalho";
34     try {
35         $db = getConnection();
36         $stmt = $db->prepare($sql);
37         $stmt->bindParam("id_venda_cabecalho", $id_venda_cabecalho);
38         $stmt->bindParam("qtde_itens", $qtde_itens);
39         $stmt->bindParam("valor_total", $valor_total);
40         $stmt->bindParam("status_venda", $status_venda);
41         $stmt->execute();
42     } catch(PDOException $e) {
43         echo '{"error":{"text":' . $e->getMessage() . "}}";
44     }

```

Listagem 12. Código para escutar evento de click do botão de iniciar venda.

```

01 //escutar click botão para iniciar uma venda
02 $('#btn_iniciar_venda').click(function(){
03     if($("#select_cliente").val() === "0"){
04         alert("Selecione um cliente antes de iniciar uma venda.");
05     }else{
06         iniciarNovaVenda();
07     }
08 });

```

- Linhas 10 a 12:** Aqui é utilizada a função getJSON do jQuery para acessar o serviço /pdv/getCliente via Ajax e recuperar as informações do cliente selecionado para a venda. Logo em seguida, é setado o nome e ID do cliente nos labels da página para exibir estes dados ao usuário caixa.

- Linha 15 a 23:** Neste bloco de código é realizada uma requisição Ajax via POST ao serviço REST no endereço /pdv/adicionarVendaCabecalho para gravar no banco de dados as informações da nova venda aberta com status em andamento.

Para verificar o resultado da implementação de abertura de venda no PDV Web execute o projeto e veja o resultado conforme consta na **Figura 8**.

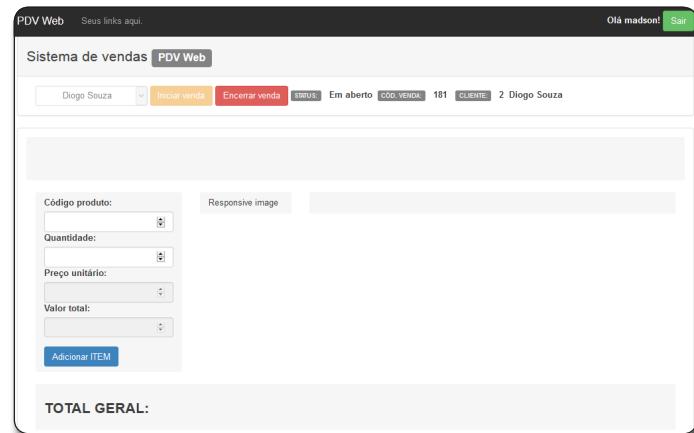


Figura 8. Venda iniciada no PDV Web

Listagem 13. Código para iniciar uma nova venda de fato.

```

01 function iniciarNovaVenda(){
02     $('#btn_iniciar_venda').attr("disabled","disabled");
03     $('#select_cliente').attr("disabled","disabled");
04     $('#btn_encerrar_venda').removeAttr("disabled");
05     $('#status_venda').text("Em aberto");
06     $('#div_venda_itens').show();
07     var cliente_id = $('#select_cliente').val();
08     $(function(){
09         //Busca os dados do cliente selecionado
10         $.getJSON("/pdv/getCliente/" + cliente_id, function(cliente){
11             $('#cliente_nome').text(cliente.nome);
12             $('#codigo_cliente').text(cliente.id_cliente);
13
14             //Adicionar a abertura de uma nova venda no banco
15             $.ajax({
16                 type: "POST",
17                 url: "/pdv/adicionarVendaCabecalho",
18                 dataType: "html",
19                 data: {"id_cliente": cliente_id, "id_usuario": $('#id_usuario').text(),
20                     "status_venda": $('#status_venda').text()},
21                 success: function(retornoldVendaCab){
22                     $('#id_venda_cabecalho').text(retornoldVendaCab);
23                     alert("Aberta uma nova venda de código: " + retornoldVendaCab +
24                         " para o cliente: " + cliente.nome);
25                 });
26             });
27 }

```

Busca de produto por ID e cálculo dos itens vendidos

Para vender itens no PDV Web é necessário que o usuário informe o código do produto e a quantidade a ser vendida. E para implementar esta operação, usaremos função *focusout* do jQuery para escutar o evento de perda de foco nos campos de código do produto e quantidade, ou seja, após a perda do foco no código do produto, validaremos o código e realizaremos uma consulta Ajax ao servidor para buscar todos os dados do mesmo, preenchendo alguns elementos HTML na página. Logo em seguida, faremos o mesmo para o campo quantidade em relação ao valor total.

O primeiro passo para a implementação da busca de produto por ID é criar o serviço para dispor tal funcionalidade. Assim sendo, implemente o trecho de código presente na **Listagem 14** dentro do arquivo `slim_rest.php`. Desta forma, ficará disponível uma nova rota `/getProduto` para consultar o produto através de uma requisição GET via Ajax utilizado o jQuery.

Voltando ao front-end da aplicação, é necessário implementar o trecho de código presente na **Listagem 15** dentro da função `carregarNoLoadPagina()`, que é chamada após a renderização da página no browser. Desta forma, será possível monitorar os eventos *focusout* dos campos código do produto e quantidade.

Listagem 14. Código para consultar um produto pelo ID.

```
01 $app->get('/getProduto/:id', function ($id) use ($app) {  
02     $sql = "select * FROM produto WHERE id_produto = :id_produto";  
03     try {  
04         $db = getConnection();  
05         $stmt = $db->prepare($sql);  
06         $stmt->bindParam("id_produto", $id);  
07         $stmt->execute();  
08         $prduto = $stmt->fetchObject();  
09         $db = null;  
10         echo json_encode($prduto, JSON_UNESCAPED_UNICODE);  
11     } catch(PDOException $e) {  
12         echo '{"error":{"text":' . $e->getMessage() . }}';  
13     }  
14 }));
```

Listagem 15. Código para monitorar os eventos de focusout dos campos.

```
01 //Escutar evento perda de foco para mostrar os dados do produto  
02 $("#codigo_produto").focusout(function(){  
03     if($("#codigo_produto").val() != "" && $("#codigo_produto").val() > 0){  
04         buscaProdutoPorId($("#codigo_produto").val());  
05     }else{  
06         alert("Valor informar o código do produto válido");  
07     }  
08 });  
09  
10 //Escutar evento perda de foco para calcular o total do item  
11 $("#qtdc_item").focusout(function(){  
12     if($("#qtdc_item").val() != "" && $("#qtdc_item").val() > 0){  
13         var valorTotalItem = parseFloat($("#qtdc_item").val())  
14             * parseFloat($("#preco_unitario").val());  
15         $("#valor_total_item").val(valorTotalItem.toString());  
16     }else{  
17         alert("Valor informar a quantidade vendida válida.");  
18     }  
19 });
```

Na linha 2, o jQuery escuta o evento *focusout* através do ID (`#codigo_produto`) do campo código produto. Na linha 4 é chamada outra função `buscaProdutoPorId()` para realizar a consulta via Ajax e retornar os dados do produto. Veja agora na **Listagem 16** como deve ser a implementação da consulta Ajax para a função `buscaProdutoPorId()`.

Listagem 16. Consulta Ajax para buscar os dados do item a ser vendido no PDV.

```
01 function buscaProdutoPorId(id_produto){  
02     $.getJSON("/pdv/getProduto/" + id_produto, function(produto){  
03         $("#preco_unitario").val(produto.preco_venda);  
04         $("#nomeProduto").text(produto.nome);  
05         $("#img_produto").attr("src", produto.imagem_produto);  
06     });  
07 }
```

Vejamos o seu detalhamento:

- Linha 1:** Aqui temos a declaração da função onde é recebido o ID do produto a ser pesquisado.
- Linha 2:** Nesta linha é realizada uma requisição via Ajax para buscar as informações do produto.
- Linha 3 a 5:** De posse dos dados do produto, exibimos na página o preço unitário do produto, nome e a imagem do mesmo.

Para testar a implementação anterior, execute o projeto, inicie uma venda, informe o código do produto e a quantidade a ser vendida e veja o resultado conforme apresentado na **Figura 9**.

The screenshot shows a user interface for a POS system. At the top, there's a header with 'Sistema de vendas' and 'PDV Web'. Below it, a dropdown menu shows 'Diogo Souza' and buttons for 'Iniciar venda' (orange) and 'Encerrar venda' (red). The main area has a title 'Revista Front-end Magazine 4'. On the left, there's a form with fields for 'Código produto' (containing '4'), 'Quantidade' (containing '2'), 'Preço unitário' (containing '14,90'), and 'Valor total' (containing '29,8'). A blue button at the bottom right of the form says 'Adicionar ITEM'. To the right of the form, there's a small thumbnail image of a magazine cover titled 'HTML5 e CANVAS'.

Figura 9. Pesquisa de produto por ID e cálculo subtotal do item

Listagem 17. Serviço REST para inclusão e manutenção de itens na venda.

```

01 $app->post('/adicionarItemVenda', function () use ($app) {
02
03     $id_produto = $app->request->post('id_produto');
04     $id_venda_cabecalho = $app->request->post('id_venda_cabecalho');
05     $sequencia = $app->request->post('sequencia');
06     $nome_produto = $app->request->post('nome_produto');
07     $qtde = $app->request->post('qtde');
08     $valor_unitario = $app->request->post('valor_unitario');
09     $sub_total = $app->request->post('sub_total');
10     $status_item = $app->request->post('status_item');
11
12     $sql = "INSERT INTO itens_venda (id_produto, id_venda_cabecalho, sequencia,
13     nome_produto, qtde, valor_unitario, sub_total, status_item)
14     VALUES (:id_produto, :id_venda_cabecalho, :sequencia, :nome_produto,
15     :qtde, :valor_unitario, :sub_total, :status_item)";
16
17     try {
18         $db = getConnection();
19         $stmt = $db->prepare($sql);
20         $stmt->bindParam("id_produto", $id_produto);
21         $stmt->bindParam("id_venda_cabecalho", $id_venda_cabecalho);
22         $stmt->bindParam("sequencia", $sequencia);
23         $stmt->bindParam("nome_produto", $nome_produto);
24         $stmt->bindParam("qtde", $qtde);
25         $stmt->bindParam("valor_unitario", $valor_unitario);
26         $stmt->bindParam("sub_total", $sub_total);
27         $stmt->bindParam("status_item", $status_item);
28
29         $stmt->execute();
30         $id_itens_venda = $db->lastInsertId();
31     }
32 });
33
34 $app->get('/getItensVenda/:id_venda', function ($id_venda) use ($app) {
35
36     $sql = "select * FROM itens_venda WHERE id_venda_cabecalho =
37     :id_venda_cabecalho ORDER BY sequencia";
38     try {
39         $db = getConnection();
40         $stmt = $db->prepare($sql);
41         $stmt->bindParam("id_venda_cabecalho", $id_venda);
42         $stmt->execute();
43         $itensVenda = $stmt->fetchAll(PDO::FETCH_OBJ);
44         $db = null;
45     } catch(PDOException $e) {
46         echo '{"error":{"text":' . $e->getMessage() . }}';
47     }
48 });
49
50 $app->post('/updateItemVenda', function () use ($app) {
51
52     $id_itens_venda = $app->request->post('id_itens_venda');
53
54     $sql = "UPDATE itens_venda SET status_item = 'cancelado'
55     WHERE id_itens_venda = :id_itens_venda";
56
57     try {
58         $db = getConnection();
59         $stmt = $db->prepare($sql);
60         $stmt->bindParam("id_itens_venda", $id_itens_venda);
61         $stmt->execute();
62         $db = null;
63     } catch(PDOException $e) {
64         echo '{"error":{"text":' . $e->getMessage() . }}';
65     }
66 });

```

Função adicionar item ao cupom

O próximo passo é implementar a funcionalidade de adicionar item ao cupom. Porém, antes disso, precisamos criar o serviço REST com algumas rotas para pesquisa, inclusão e alteração de itens na base de dados conforme apresentado na **Listagem 17**. O código deve ser inserido dentro do arquivo `slim_rest.php` para tornar as rotas das linhas 01, 34 e 50 disponíveis para acesso pela aplicação front-end.

Após todo o serviço REST implementado no back-end, podemos a partir deste ponto trabalhar somente no front-end da aplicação. O próximo passo é adicionar o trecho de código da **Listagem 18** dentro da função `carregarNoLoadPagina()` para que possamos capturar o evento de click do botão adicionar item ao cupom.

O código captura algumas informações do item a ser vendido e logo na linha 6 chama a função `adicionarItemCupom()` para gravar o item no banco de dados. Veja os detalhes da função `adicionarItemCupom()` conforme apresentado na **Listagem 19**.

A última listagem apenas grava os dados do item vendido na base de dados através de uma requisição Ajax conforme linha 4. Logo na sequência, recupera todos os itens vendidos relacionados à venda em andamento e popula a tabela `tabela_cupom` com a utilização da função `html()` do jQuery presente na linha 32.

Listagem 18. Código para escutar o evento click do botão adicionar item ao cupom.

```

01 //Escuta o evento click para adicionar o item ao cupom
02 $("#btnAdicionarItem").click(function(){
03     if($("#qtde_item").val() !== "" && $("#qtde_item").val() >
04     0 && $("#codigo_produto").val() > 0 ){
05
06         var statusItem = "vendido";
07         adicionarItemCupom($("#codigo_produto").val(), $("#id_venda_cabecalho").text(),
08         ($("#contadorSeqItemVenda").val(), $("#nomeProduto").text(), $("#qtde_item").val(),
09         $("#preco_unitario").val(), $("#valor_total_item").val(), statusItem));
10     }else{
11         alert("Valor informar o código do produto e uma quantidade válida.");
12     }
13 });

```

Execute o projeto novamente, faça a venda de itens e obtenha o resultado apresentado na **Figura 10**.

Função cancelamento de item do cupom

Como em qualquer PDV, às vezes pode ser necessário cancelar um item no cupom, assim sendo perceba que ao lado de cada item vendido foi adicionado um botão para cancelamento. O click no botão cancelamento irá chamar a função `cancelarItemVenda()` conforme pode ser visto na **Listagem 20**.

Bootstrap e jQuery: Criando um PDV Web – Parte2

Listagem 19. Código para gravar item vendido no banco de dados por requisição Ajax com jQuery.

```
01 function adicionarItemCupom(codigo_produto, codigo_venda, seqItemVenda,
02   nome_produto, qtde_item, preco_unitario, valor_total_item, statusItem){
03   var codigo_vend = parseFloat(codigo_venda.replace(","));
04   //Adicionar item vendido no banco de dados
05   $.ajax({
06     type:"POST",
07     url:"/pdv/adicionarItemVenda",
08     dataType:"html",
09     data: {"id_produto": codigo_produto, "id_venda_cabecalho":
10       codigo_vend, "sequencia": seqItemVenda, "nome_produto":
11       nome_produto, "qtde": qtde_item, "valor_unitario": preco_unitario,
12       "sub_total": valor_total_item, "status_item": statusItem},
13     success: function(retornoldItemVenda){
14
15     //Incrementa e seta + 1 na sequencia de itens
16     var seq = parseInt(seqItemVenda) + 1;
17     $("#contadorSeqItemVenda").val(seq);
18
19     //Preencher a tabela com os itens vendidos
20     $.getJSON("/pdv/getItensVenda/" + codigo_vend, function(itens){
21
22     //Definir cabeha tabela
23     var linhasTabela = '<tr><th>ITEM </th> <th>CÓDIGO </th>
<th>DESCRIÇÃO </th> <th>QDTE </th> <th>PREÇO </th>
<th>TOTAL </th> <th>STATUS </th><th></th> </tr>';
24
25     var totalGeralVenda = 0.0;//Armazenar o total geral
26
27     for (var i = 0; i < itens.length; i++) {
28
29       linhasTabela += "<tr><th>" + itens[i].sequencia + "</th><th>" +
30       itens[i].id_produto + "</th><th>" + itens[i].nome_produto + "</th><th>" +
31       itens[i].qtde + "</th><th>" + itens[i].valor_unitario + "</th><th>" +
32       itens[i].sub_total + "</th><th>" + itens[i].status_item +
33       "</th><th><button class='glyphicon glyphicon-remove' aria-hidden='true' onClick='cancelarItemVenda(" +
34       "+ itens[i].id_itens_venda +"," + itens[i].sequencia +",
35       + itens[i].id_venda_cabecalho +")'></th></tr>";
36
37     }
38
39     //Calcula o total geral da venda
40     if(itens[i].status_item === "vendido"){
41       totalGeralVenda += parseFloat(itens[i].sub_total);
42     }
43
44     //Exibir tabela e setar total geral
45     $("#table#tabela_cupom").html(linhasTabela);
46     $("#total_geral").text("R$ " + totalGeralVenda.toString());
47   });
48
49   //Limpa os campos
50   $("#codigo_produto").val("");
51   $("#qtde_item").val("");
52   $("#preco_unitario").val("");
53   $("#valor_total_item").val("");
54   $("#nomeProduto").text("");
55   $("#img_produto").attr("src","");
56 });
57
58 };
```

Listagem 20. Código para cancelamento de item no cupom

```
01 function cancelarItemVenda(id_itens_venda, sequencia, id_venda_cabecalho){
02   $.ajax({
03     type:"POST",
04     url:"/pdv/updateItemVenda",
05     dataType:"html",
06     data: {"id_itens_venda": id_itens_venda},
07     success: function(retornoldItemCancelado){
08
09     //Preencher a tabela com os itens vendidos
10     $.getJSON("/pdv/getItensVenda/" + id_venda_cabecalho,
11       function(itens){
12
13     //Definir cabeha tabela
14     linhasTabela = '<tr><th>ITEM </th> <th>CÓDIGO </th>
<th>DESCRIÇÃO </th> <th>QDTE </th> <th>PREÇO </th> <th>TOTAL
</th> <th>STATUS </th><th></th> </tr>';
15     var totalGeralVenda = 0.0;
16
17     for (var i = 0; i < itens.length; i++) {
18       linhasTabela += "<tr><th>" + itens[i].sequencia + "</th><th>" +
19       itens[i].id_produto + "</th><th>" + itens[i].nome_produto +
20       "</th><th>" + itens[i].qtde + "</th><th>" + itens[i].valor_unitario +
21       "</th><th>" + itens[i].sub_total + "</th><th>" + itens[i].status_item +
22       "</th><th><button class='glyphicon glyphicon-remove' aria-hidden='true' onClick='cancelarItemVenda(" +
23       "+ itens[i].id_itens_venda +"," + itens[i].sequencia +")'></th></tr>";
24
25     }
26
27     //Calcula o total geral da venda
28     if(itens[i].status_item === "vendido"){
29       totalGeralVenda += parseFloat(itens[i].sub_total);
30     }
31
32     //Exibir tabela e setar total geral
33     $("#table#tabela_cupom").html(linhasTabela);
34     $("#total_geral").text("R$ " + totalGeralVenda.toString());
35   });
36
37   alert("O item: " + sequencia + " foi cancelado.");
38 }
39 });
40
41 };
```

O código desta listagem realiza o update do item a ser cancelado na base de dados, e logo em seguida recarrega os itens vendidos na página descontando do valor total da venda o valor do item cancelado.

Execute o projeto novamente, venda alguns itens e realize o cancelamento de um item. Após isso, você irá obter um resultado conforme apresentado na **Figura 11**, onde o item de número 3 foi cancelado. Perceba também que o valor total geral da venda foi recalculado após o cancelamento do item.

Função de encerramento da venda

A última funcionalidade a ser implementada no PDV é o encerramento da venda em andamento. Para implementar isso, copie o código da **Listagem 21** para o arquivo javascript_pdv.js.

O código define uma função chamada **encerrarVenda()**, que, por sua vez, está amarrada ao evento de click do botão “Encerrar Venda” localizado na parte superior da página. Execute o projeto, venda alguns itens e encerre a venda para obter o resultado apresentado na **Figura 12**.

Figura 10. Inclusão de itens no cupom

Figura 11. Exemplo de cancelamento de item

Figura 12. Mensagem de encerramento da venda no PDV Web

Ao longo deste artigo, foi possível acompanhar todo o desenvolvimento de uma aplicação real conforme exemplo apresentado do PDV Web. O desenvolvimento de aplicações desse tipo, muitas vezes, cai em meio comum, motivado principalmente pela aproximação com o desinteresse causado por aplicações do tipo CRUD nos profissionais da área.

Listagem 21. Código para encerramento da venda.

```

01 function encerrarVenda(){
02   //Recuperar dados para finalizar venda
03   var id_venda = $("#id_venda_cabecalho").text();
04   var id_venda = parseInt(id_venda);
05   var qtde_itens_vendidos = $("#contadorSeqItemVenda").val();
06   var total_geral_venda = $("#total_geral").text().replace('R$ ');
07
08
09 //Alterar a tabela venda cabeçalho para finalizada
10 $.ajax({
11   type:"POST",
12   url:"/pdv/updateVendaCabecalho",
13   dataType:"html",
14   data: {"id_venda_cabecalho": id_venda, "qtde_itens": qtde_itens_vendidos,
15   "valor_total": total_geral_venda, "status_venda": "Finalizada"},
16   success: function(){
17     alert("Venda finalizada - Total: " + $("#total_geral").text());
18     $("#table#tabela_cupom").html("");
19     $("#btn_iniciar_venda").removeAttr("disabled");
20     $("#select_cliente").removeAttr("disabled");
21     $("#btn_encerrar_venda").attr("disabled","disabled");
22     $("#status_venda").text("Finalizada");
23     $("#div_venda_itens").hide();
24     $("#cliente_nome").text("");
25     $("#codigo_cliente").text("");
26     $("#contadorSeqItemVenda").val(1);
27     $("#id_venda_cabecalho").text("");
28   }
29 });
30
31 }
```

Independentemente do tipo de aplicação que você venha a desenvolver, seja qual for a sua finalidade, é importante que exista planejamento e organização. O Bootstrap consegue trazer tudo isso de forma integrada, principalmente pela abstração de muitas tarefas já não mais necessárias. Cabe a você, como desenvolvedor, se aprimorar cada vez mais e praticar bastante os conceitos vistos aqui e disponíveis ao longo da documentação oficial do framework.

Autor



Madson Aguiar Rodrigues

Formação acadêmica em Análise e Desenvolvimento de Sistemas pela UNOPAR, pós-graduação em Engenharia de Sistemas pela ESAB e especialista em Tecnologias para aplicações Web pela UNOPAR. Trabalha com desenvolvimento de software há sete anos com uso a da plataforma .NET, JAVA e Mobile com Android.



Links:

Download XAMP

<http://www.XAMP.com/en/>

Slim framework

<http://www.slimframework.com/>

Download Bootstrap

<http://getBootstrap.com/getting-started/#download>

Download biblioteca jQuery

<http://jquery.com/download/>

Somos tão apaixonados por tecnologia que o nome da empresa diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia. Somos uma equipe composta de gente que entende e gosta do que faz, assim como você.



Estrutura

100% NACIONAL.
Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!



Porta 80

WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce |
Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486