Front-end magazine

Edição 04

DEVMEDIA

Designs Responsivos em HTML5
Conheça todos os passos para criar sites multiplataforma

HTML5 E CANVAS

CONSTRUINDO UM JOGO 2D COM JAVASCRIPT

Arquitetura AngularJS
Principais técnicas para decidir qual a melhor arquitetura em JavaScript

Grunt + CoffeeScript
A união perfeita para criar aplicações escaláveis e automatizadas

Front-end magazine

Edição 04 • 2015

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 a 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

Jogos em HTML5: Criando um jogo 2D com Canvas – Parte 1

Domine os recursos do Canvas HTML5 implementando um jogo estilo shoot em up

Este artigo faz parte de um curso

Fique por dentro

Este artigo tem como objetivo mostrar de forma prática e direta os conhecimentos e técnicas mais utilizados no desenvolvimento de jogos usando HTML5 em conjunto com Canvas e JavaScript. O mesmo se propõe a explorar os recursos da tecnologia, bem como as boas práticas da comunidade, através do desenvolvimento de um jogo estilo shoot 'em up, onde os personagens se atacam no plano horizontal com combos, contagem de pontos e reinicialização do mesmo. Desta forma, o desenvolvedor poderá usar conceitos como otimização de performance, reaproveitamento de objetos, orientação a objetos, herança e programação multicamada dentro de um cenário real, otimizando seu nível de conhecimento e, consequentemente, sua produtividade.

Atualmente, uma imensa maioria das aplicações que encontramos nas lojas da Play e App Store correspondem a aplicativos desenvolvidos usando somente HTML5 e JavaScript. Essa ampla adoção de duas tecnologias deveras comuns se deve às cada vez mais frequentes adoções por parte dos desenvolvedores de frameworks multiplataforma como o Cordova, PhoneGap, etc. que usam a tecnologia HTML5 como base e simplificam, consideravelmente, a criação e manutenção de aplicativos móveis e distribuídos.

Aliada a tais fatores, encontra-se também uma preferência exponencial por aplicações que visam o entretenimento, como jogos, players e derivados. E quando se trata de jogos, as estatísticas mostram que os desenvolvidos sob a ótica 2D (bidimensionais), que em sua grande maioria são bem mais simplistas e diretos, ganham a preferência dos usuários em disparado. Ao mesmo tempo, do ponto de vista técnico, eles também são mais simples de desenvolver em comparação aos jogos 3D, o que acaba criando uma gama maior de possibilidades para as empresas.

Para possibilitar maior flexibilidade e rapidez no desenvolvimento de jogos para esse público, as empresas, por sua vez, optam cada vez mais pela HTML5 como opção barata, fácil de utilizar e que consome menos recursos em se tratando de ferramentas, softwares proprietários e/ou de terceiros, bem como por sua desnecessidade de ambientes complexos e uso de mais de uma linguagem de programação. Tudo isso é possível através do recurso que a tecnologia apresenta chamado Canvas, que são elementos da HTML5 que possibilitam a "pintura" de todos os tipos de gráficos, desde simples linhas até gráficos mais complexos, tendo como base a linguagem JavaScript.

Neste artigo, nós aprenderemos os recursos do Canvas e como ele se integra ao trio HTML5, CSS e JavaScript através da criação de um jogo no estilo shoot 'em up, um gênero de jogos para computador em que o personagem principal se concentra em atirar nos alvos (inimigos) em cenários horizontais, de modo que outros aspectos do jogo são simplificados para facilitar tudo isso. No mesmo jogo, também criaremos pontuações, a possibilidade de reiniciar a partida caso o player seja atingido, um botão para silenciar a música de fundo, assim como a geração de cenários e players de forma randômica, para deixar o jogo mais interessante.

Os personagens serão baseados em um jogo de guerra, com soldados futuristas contra os monstros espaciais que invadiram a terra. Para controlar o player serão usadas as teclas de navegação do teclado (←↑↓→), que permitirão mover o mesmo por toda a tela, dentro do perímetro máximo que determinarmos para ele (vamos criar uma barreira invisível para que o player não chegue muito perto dos inimigos, e vice-versa); e para atirar será usada a tecla de barra de espaço do teclado. O jogo irá rodar enquanto o jogador não morrer, e o placar será incrementado sem fim, consequentemente; até que o mesmo seja atingido por um dos inimigos (que também virão infinitamente). Neste jogo, não implementaremos recursos como vidas, pausa ou aumento da dificuldade com o tempo, mas são recursos que o leitor estará apto a desenvolver após este artigo.

O ambiente para desenvolvimento é bem simples, basta ter um editor de texto e um navegador web de sua preferência. É recomendável usar algum editor que reconheça arquivos de extensões .html e.js, como o Notepad++ (vide seção Links). Mas antes de começarmos a implementação de fato, vamos entender melhor o que são Canvas e como eles funcionam.

Canvas

A HTML5 disponibiliza em sua lista de tags básicas, a tag **<canvas>**. Através dela, os desenvolvedores front-end tem em mãos um poderoso recurso para desenhar grafos, gráficos de jogos ou outras imagens visuais em tempo real, usando apenas JavaScript. É comum vermos vários plug-ins jQuery ou JavaScript fazendo uso de canvas para exibir gráficos de linha, pizza, dentre outras variantes. Em outras palavras, um canvas (ou telas, em tradução livre) é um retângulo na sua página onde você pode usar Java-Script para desenhar qualquer coisa que deseje.

O Canvas foi criado inicialmente pela Apple para ser usado no interior do seu Sistema Operacional, num componente chamado *OS X WebKit*, em 2004, e trouxe consigo uma gama de novos recursos ao SO. Pouco tempo depois foi patenteado pelo *Web Hypertext Application Technology Working Group* (WHATWG), e hoje é amplamente usado e padronizado por todos os navegadores mais recentes.

Para garantir que você terá pleno acesso aos recursos do Canvas, é aconselhado usar a versão mais recente do seu browser, uma vez que versões mais antigas não o proveem em suas implementações padrão. Veja na **Figura 1** a lista de browsers que suportam o Canvas, com suas respectivas versões e S.O.

Conforme observado, o Internet Explorer só provê suporte nativo ao recuso a partir de sua versão 9. Se você desejar habilitar o recurso nas versões 7.0 ou 8.0, existe um projeto chamado **explorercanvas** (ver seção **Links**), que foi criado para servir como extensão ao IE nessas versões e simular a funcionalidade do Canvas no navegador, bastando para isso incluir algumas linhas de script nas páginas que o usuário deseja executar. Além disso, em se tratando de versões de browser para dispositivos móveis, o Canvas já vem como recurso básico nos mesmos. A partir do iPhone 1.0 e Android 1.0 já temos o recurso disponível.

SUPORTE AO CANVAS			
		MAC OS	WINDOWS
SAFARI	5.1	✔	--
FIREFOX	8	✔	✔
	9	✔	--
OPERA	11.1	✔	--
CHROME	15	✔	✔
	17	✔	✔
IE	7	--	✖
	8	--	✖
	9	--	✔
	10	--	✔

Figura 1. Lista de navegadores web que suportam o Canvas

A estrutura HTML de um canvas é relativamente simples e se aproxima muito da que já estamos acostumados em páginas web:

```
<canvas id="meuCanvas" width="250" height="250">
  Esse texto será exibido caso seu browser não suporte o Canvas
</canvas>
```

Neste exemplo, estamos fazendo uso apenas de três atributos básicos das tags HTML, como largura, altura e identificação. Mas os demais atributos padrão de tags também podem ser usados e iremos demonstrar outros exemplos ao longo do artigo.

Adicionalmente, é importante termos em mente que desenhar qualquer coisa com canvas em HTML é uma operação pesada, que consome muitos recursos do browser. Por causa disso, nós precisaremos reduzir a quantidade de "desenhos" que faremos com o canvas para garantir que a performance seja boa o suficiente para não interromper a execução do jogo.

Criando o projeto

Antes de começar a programar efetivamente, vamos criar uma estrutura básica de diretórios para o nosso projeto. Veja na **Listagem 1** a estrutura padrão que usaremos para o nosso aplicativo de jogo.

Listagem 1. Estrutura básica de pastas para o projeto

```
jogo-guerra-espacial
--------| imgs
--------| js
--------| css
```

Essa estrutura nos ajudará a não nos perdermos dentre os diferentes tipos de arquivos que manipularemos no projeto.

Agora crie um novo arquivo de extensão .html e o nomeie "index .html". Esse será o arquivo de conteúdo HTML principal do projeto. Adicione o conteúdo da **Listagem 2** ao mesmo.

Listagem 2. Conteúdo da página inicial do jogo

```
<!DOCTYPE html>
<html>
  <head>
    <title>Jogo: Guerra do Espaço</title>
    <style>
      #planoDeFundo {
        background: transparent;
        left: 0px;
        top: 0px;
        position: absolute;
      }
    </style>
  </head>
  <body onload="iniciar()">
    <!-- Esse canvas será usado inicialmente para o plano de fundo do jogo -->
    <canvas id="planoDeFundo" width="948" height="592">
      Seu navegador não suporta o recurso de Canvas.
      Por favor, tente novamente com um navegador diferente.
    </canvas>
    <script src="js/global.js"></script>
  </body>
</html>
```

Criamos um elemento canvas e incluímos o texto dentro dele, no caso do navegador não suportar o recurso de canvas. O método **iniciar()** representará o construtor dos canvas que usaremos e será responsável por carregar todos os recursos iniciais. A largura e altura do elemento receberam estes valores para que se adéquam ao tamanho das imagens que usaremos posteriormente para representar os cenários do jogo. Por fim, carregamos o arquivo JavaScript que conterá todas as funções de script do jogo como um todo.

Crie, em seguida, um novo arquivo na pasta js do nosso projeto chamado "global.js", que conterá as funções JavaScript.

Criando os cenários

A partir de agora, vamos trabalhar essencialmente no arquivo de JavaScript. Como o mesmo irá se estender um pouco, vamos nos ater às funções. Criaremos função por função, que deverão ser adicionadas em sequência no mesmo arquivo, para facilitar o entendimento do código geral. A primeira coisa a se criar é o objeto que será responsável por desenhar coisas no jogo, como mostra a **Listagem 3**.

O tipo de estrutura que vemos nessa listagem é bem comum em alguns frameworks JavaScript, onde temos a estrutura de uma classe sendo representada, tal como é feita com componentes orientados a objetos. Neste caso, definimos um escopo de objeto desenhável, com os recursos, atributos e funções mínimos necessários para se pintar na tela algum objeto JavaScript representativo. Quando quisermos criar um objeto de tipo Jogador ou Inimigo, por exemplo, basta que herdemos desta classe e suas características serão, obrigatoriamente, associadas ao novo objeto. Em termos de programação, o chamamos de **objeto abstrato**. As vantagens dessa abordagem vão além dos conceitos básicos de herança, atingindo também o polimorfismo e a facilidade na manutenção do código, quando desejarmos alterar estruturas que devam refletir em todos os objetos "desenháveis", por exemplo.

Listagem 3. Código da classe Desenhavel

```
/**
 * Cria o objeto "Desenhavel" que será a classe base para
 * todos os objetos desenháveis do jogo. Define também as variáveis padrão
 * que todos os objetos filhos herdarão, assim como as funções padrão.
 */
function Desenhavel() {
  this.velocidade = 0;
  this.larguraCanvas = 0;
  this.alturaCanvas = 0;

  // Define uma função abastrata para ser sobrescrita nos objetos filho
  this.desenhar = function() {
  };

  this.iniciar = function(x, y) {
    // Variáveis padrão do eixo cartesiano
    this.x = x;
    this.y = y;
  }
}
```

Podemos observar também que na mesma declaração temos uma função **iniciar()** sendo declarada também de forma genérica. Ela será responsável por receber os valores do objeto, estes que serão definidos no modelo de plano cartesiano, com as respectivas coordenadas **x** e **y**, referentes à posição onde aquele objeto se encontra dentro do limite de pintura da tela. Nós também precisaremos salvar a informação referente à velocidade que aquele objeto deve se mover na tela, visto que teremos diferentes velocidades para cada "desenhável" no jogo. Finalmente, criamos a função **desenhar()** (também genérica), com o intuito de fornecer aos objetos filhos a possibilidade de sobrescrever a forma como os mesmos serão impressos na tela.

Em sequência, criaremos agora um novo objeto **repositorio** no nosso modelo. Veja na **Listagem 4** o código para tal.

Listagem 4. Objeto de repositório das imagens

```
/**
 * Define um objeto para manter todas as nossas imagens do jogo para
 * evitar que elas sejam criadas mais de uma vez.
 */
var repositorio = new function() {
  this.planofundo = new Image();

  // Configura os caminhos (src) das imagens
  this.planofundo.src = "imgs/pf.png";
}
```

Esse código basicamente cria um objeto de domínio, que será responsável por guardar os objetos a serem desenhados na tela (até o momento somente a imagem de plano de fundo). Esse tipo de estratégia serve tanto para não deixar os objetos soltos dentro do código, tendo que criar várias referências para eles, quanto para impedir que o navegador use recursos extra para criar um objeto por vez em várias chamadas distintas. À medida que formos evoluindo no artigo, esse objeto passará a receber novas entradas, dadas as adições dos demais atores no jogo.

Nota

Para que o código funcione é necessário, obviamente, que as imagens estejam nos lugares definidos. Para isso, baixe as mesmas diretamente do link de download de fonte deste artigo e posicione-as corretamente no seu projeto.

Agora que temos os objetos de template desenhável e o repositório de imagens, vamos lidar com a implementação do plano de fundo do cenário. Adicione o código contido na **Listagem 5** ao seu arquivo js.

Listagem 5. Função que cria e anima o plano de fundo

```
/**
 * Cria o objeto PlanoFundo que se tornará um filho do
 * objeto Desenhavel. O plano de fundo será desenhado nesse objeto
 * e criará a ilusão de movimento ao deslocar a imagem.
 */
function PlanoFundo() {
  this.velocidade = 1; // Redefine a velocidade do plano de fundo para pintura

  // Implementa a função abstrata
  this.desenhar = function() {
    // Pinta o plano de fundo
    this.x -= this.velocidade;
    this.context.drawImage(repositorio.planofundo, this.x, this.y);

    // Desenha outra imagem na borda superior da primeira imagem
    this.context.drawImage(repositorio.planofundo,
    this.x + this.larguraCanvas, this.y);

    // Se a imagem for deslocada para fora da tela, redefine-a
    if (Math.abs(this.x) >= this.larguraCanvas)
      this.x = 0;
  };
}
// Define o PlanoFundo como herdeiro das propriedades de Desenhavel
PlanoFundo.prototype = new Desenhavel();
```

É importante entender como funciona o objeto PlanoFundo, pois ele introduz os conceitos de velocidade, animação de elementos e cálculos com os valores do plano cartesiano. Entendamos, portanto, o que são alguns dos envolvidos no código desse objeto:

• **this.x, this.y**: Sempre que virmos essas referências no código, aqui se faz uma associação aos valores dos eixos x e y do plano bidimensional do jogo.

• **context**: Provê métodos (drawImage(), por exemplo) e propriedades para desenhar nos canvas. Também pode ser recuperado através do método getContext().

• **larguraCanvas**: No nosso projeto, vamos referenciar as dimensões de dois elementos básicos: a tela total de pintura do jogo e os objetos desenháveis, com os atributos largura, altura e larguraCanvas, alturaCanvas, respectivamente.

• **Math.abs()**: Método JavaScript usado para obter o valor absoluto (ou seja, positivo) de um número.

Veja que logo no início da função desenhar() nós estamos decrementando o valor da variável x (já que estamos navegando na horizontal) de acordo com a velocidade, que, por sua vez, foi definida com valor 1, isto é, 1 pixel por frame. Isso porque essa função será chamada várias vezes por segundo no processo de pintura da tela e de seus componentes. Esse decremento servirá para definir como a imagem será repintada, uma vez que suas proporções devem ser calculadas para que ao terminar de exibi-la, voltemos para o seu início para exibi-la de novo, e assim sucessivamente.

Após, verificamos se o valor absoluto (absoluto, neste caso, porque a variável começará a ficar negativa e falharia nesse teste se não o usássemos) do eixo x é maior ou igual à largura do canvas. Em caso positivo, nós resetamos o valor de x para que tudo comece mais uma vez. Finalmente, usamos o **prototype** (operador responsável por incutir herança em JavaScript) para carimbar o objeto em questão como um herdeiro de Desenhavel.

Uma vez com o plano de fundo implementado, ainda precisamos criar a estrutura que irá se encarregar de subir os objetos em memória, explicitamente, como faríamos, por exemplo, com o método **main()** do Java. Para isso, adicione o conteúdo da **Listagem 6** no seu arquivo de scripts.

O objeto Jogo consta de apenas duas funções. A mais básica delas, a função **jogar()**, será responsável por fazer as chamadas à função animar(), que criaremos em seguida, para iniciar o loop de animação. A outra função, **iniciar()**, é uma versão sobrescrita da função de Desenhavel, que irá recuperar o objeto de canvas, verificar se ele é válido, definir as configurações iniciais do plano de fundo e setar as configurações do objeto PlanoFundo, como a largura e altura do canvas; respectivamente.

A função sobrecarregada "getContext" serve tanto para verificar se um canvas é válido, quanto para retornar o objeto de contexto do mesmo, informando o universo de pintura, neste caso com valor "2d". A função ainda retorna um valor booleano de acordo com a validade do objeto de canvas, já que não terá valia alguma um objeto canvas inválido.

Isso é basicamente tudo o que precisamos para o nosso cenário no que se refere à estrutura. Agora, só precisamos criar um loop que itere infinitamente sobre os mesmos recursos, exibindo-os e apagando-os. Para isso, vamos dar uma olhada na **Listagem 7**, que contém o código final dessa nossa primeira implementação.

Essa função será a responsável por, de fato, efetuar a pintura do objeto de plano de fundo no nosso jogo.

Quando se trata de definir um algoritmo que calcule o intervalo necessário para executar cada chamada à pintura da tela, é perigoso utilizar o seu próprio. Isso porque muitos dos algoritmos

de terceiros usam a função setTimeout() do JavaScript para estabelecer os intervalos de execução, quando na verdade ela não é otimizada para execuções em 60FPS, por exemplo. Cada browser, hoje em dia, tem seu próprio algoritmo de iteração, otimizado para a realidade do canvas e a função **getFrameAnimacao()** faz exatamente isso, buscando o algoritmo válido para cada browser de forma estilizada. Ela foi criada pelo desenvolvedor Paul Irish, e é amplamente aceita na comunidade (ver seção **Links**). Ainda assim, adicionamos a chamada direta à função setTimeout() em última instância, caso esteja usando um browser mais antigo.

Para finalizar o plano de fundo, basta que criemos um novo objeto Jogo e chamemos sua função de inicialização (**Listagem 8**). Para facilitar a visualização e entendimento, você pode posicionar esse código no início do arquivo. Lembre-se que o JavaScript consegue encontrar os objetos dentro da página independentemente de onde eles estejam alocados, tal como em outras linguagens OO.

Agora é só executar o documento HTML no browser e o resultado será semelhante ao da **Figura 2**.

Listagem 6. Criando o objeto principal do jogo

```
/**
 * Cria um objeto mais genérico que se encarregará de lidar com os dados do jogo.
 */
function Jogo() {
   this.iniciar = function() {
      // Recupera o elemento canvas
      this.pfCanvas = document.getElementByld('planoDeFundo');
      var retorno = false;

      // Testa para verificar se o canvas é suportado
      if (this.pfCanvas.getContext) {
         // inicializa o objeto de plano de fundo
         this.planofundo = new PlanoFundo();
         this.planofundo.iniciar(0,0); // Inicia no ponto 0,0

         this.pfContext = this.pfCanvas.getContext('2d');

         // inicializa os objetos configurando as propriedades em questão
         PlanoFundo.prototype.context = this.pfContext;
         PlanoFundo.prototype.larguraCanvas = this.pfCanvas.width;
         PlanoFundo.prototype.alturaCanvas = this.pfCanvas.height;

         retorno = true;
      }
      return retorno;
   };

   // Inicia o loop de animação
   this.jogar = function() {
      animar();
   };
}
```

Listagem 7. Código do loop de animação para tela de pintura.

```
/**
 * O loop de animação. Chama a função requestAnimationFrame
 * para otimizar o loop do jogo e desenha todos os objetos do jogo. Esta
 * função deve ser uma função gobal e não pode estar dentro de um objeto.
 */
function animar() {
   getFrameAnimacao( animar );
   jogo.planofundo.desenhar();
}


/**
 * Essa é uma função criada por Paul Irish que
 * tenta encontrar a primeira API que trabalhe com otimização
 * de loops, caso contrário executa um setTimeout().
 */
window.getFrameAnimacao = (function(){
   return  window.requestAnimationFrame       ||
         window.webkitRequestAnimationFrame ||
         window.mozRequestAnimationFrame    ||
         window.oRequestAnimationFrame      ||
         window.msRequestAnimationFrame     ||
         function(/* function */ callback, /* DOMElement */ element){
            window.setTimeout(callback, 1000 / 60);
         };
})();
```

Listagem 8. Código de chamada efetiva para executar o jogo.

```
/**
 * Inicializa o jogo e dá o play.
 */
var jogo = new Jogo();

function iniciar() {
   if(jogo.iniciar())
      jogo.jogar();
}
```

Figura 2. Tela do plano de fundo do jogo

Note que a tela navega da direita para a esquerda no plano horizontal. Esse efeito se explica através da criação do objeto de PlanoFundo, feita anteriormente. Se desejar inverter o sentido da navegação, basta inverter o sinal de decremento para incremento no mesmo objeto.

No arquivo de download do projeto, estão disponíveis duas imagens de plano de fundo. Modifique o valor do atributo src do objeto planodefundo criado no repositório de imagens para "pf2 .png" e veja o resultado dessa pequena mudança na **Figura 3**.

Múltiplas camadas em Canvas

Desenhar imagens grandes ou em grande quantidade usando Canvas é dispendioso e deve ser evitado sempre que possível. Além de usar um outro canvas para renderizações fora da tela, nós podemos usar canvas em camadas, umas sobre as outras.

Figura 3. Ilustração do segundo plano de fundo do jogo

Ao usar a transparência na tela em primeiro plano, podemos contar com a GPU para compor o contexto na hora de renderizar. Você pode configurá-lo da forma mostrada na **Listagem 9**, com dois canvas absolutamente posicionados um em cima do outro.

Listagem 9. Exemplo de canvas multicamadas

```
<canvas id="test" width="500" height="150" style="position: absolute; z-index: 0">
</canvas>
<canvas id="test2" width="500" height="150" style="position: absolute;
  z-index: 1">
</canvas>
```

A vantagem sobre ter apenas um canvas aqui, é que quando tiramos ou limpamos o do primeiro plano, nós nem sempre precisamos modificar o fundo. Se o seu aplicativo de jogos ou multimídia pode ser dividido em primeiro e segundo plano, considere renderizar estes em canvas separados para obter um ganho significativo de performance. O gráfico da **Figura 4** compara o uso de canvas simples em relação a outros em que se meramente redesenha e limpa o primeiro plano, para diferentes tipos de browsers.

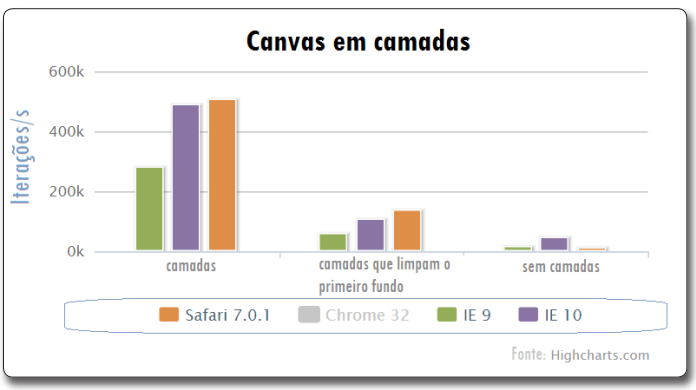

Figura 4. Gráfico comparativo entre o uso ou não de multicamadas no Canvas

Como a percepção humana para esse tipo de situação é imperfeita, você pode tirar proveito renderizando o fundo apenas uma vez ou em uma velocidade mais lenta em comparação com o primeiro plano. Por exemplo, você pode desenhar o plano de frente cada vez que você renderizar, porém desenhando o plano de fundo somente em cada frame.

Note também que esta abordagem generaliza bem para qualquer número de canvas compostos se o seu aplicativo funciona melhor com este tipo de estrutura.

Criando o Player e Ataques

Agora precisamos adicionar o player (o personagem principal) ao jogo e os seus ataques de combos. Antes disso, nós precisamos efetuar a criação de mais dois canvas na nossa tela HTML, um para ser usado para renderização dos inimigos e outro para controlar os movimentos do nosso player. Para isso, vamos modificar o conteúdo original do arquivo para o representado na **Listagem 10**.

Listagem 10. Novo conteúdo da página index.html.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Jogo: Guerra do Espaço</title>
    <style>
      #planoDeFundo, #principal, #player_mov {
        background: transparent;
        left: 0px;
        top: 0px;
        position: absolute;
      }
      #background {
        z-index: -2;
      }
      #main {
        z-index: -1;
      }
      #ship {
        z-index: 0;
      }
    </style>
  </head>
  <body>
    <!-- Esse canvas será usado inicialmente para o plano de fundo do jogo -->
    <canvas id="planoDeFundo" width="948" height="360">
      Seu navegador não suporta o recurso de Canvas. Por favor, tente novamente
      com um navegador diferente.
    </canvas>
    <!-- Canvas para os inimigos e para os poderes que eles lançarem -->
    <canvas id="principal" width="948" height="360">
      Seu navegador não suporta o recurso de Canvas. Por favor, tente novamente
      com um navegador diferente.
    </canvas>
    <!-- Canvas para limitar o quadrante de movimentação do player -->
    <canvas id="player_mov" width="948" height="360">
      Seu navegador não suporta o recurso de Canvas.
      Por favor, tente novamente com um navegador diferente.
    </canvas>
    <script src="js/global.js"></script>
  </body>
</html>
```

Para entender o uso dos três canvas criados, façamos uma comparação entre eles e uma edição de imagem no Photoshop, por exemplo. Quando queremos mesclar recursos no Photoshop, nós usamos as camadas, que nada mais são que pedaços da imagem organizados **em forma de pilha**.

Quando se trata de canvas, a situação é a mesma: temos várias camadas, cada uma representada por uma tag <canvas> diferente, que também são organizadas em pilhas e servem para definir onde cada pedaço do todo (o jogo) irá atuar, existir graficamente.

Na nossa listagem temos três canvas:

• O primeiro, já criado antes, será responsável por "rolar" o plano de fundo, criando o efeito de cenário infinito.

• O segundo será dedicado especialmente ao nosso player principal, pois ele pode estar se movendo ou não. E gerenciar esse controle com um canvas só iria nos dar muito mais trabalho.

• O terceiro e último, será usado para exibir os inimigos e as bolas de poder (combos) dos personagens, já que estas exibições serão em maior quantidade, em diferentes direções e velocidades, e randômicas.

Nós também removemos a função iniciar() do atributo onload do corpo HTML e acrescentamos alguns atributos z-index ao CSS, que serão importantes para definir qual camada ficará por sobre as demais. Veremos o porquê mais adiante.

A primeira alteração no lado JavaScript irá consistir em incrementar o script de criação do objeto Desenhavel, uma vez que agora precisaremos manipular não somente a forma como os objetos são desenhados, mas também como eles se movem. Dentro do mesmo objeto, adicione o conteúdo da **Listagem 11**. Note que também estamos recuperando na função iniciar() os valores de largura e altura para os atributos do mesmo objeto desenhável, valores que precisaremos manipular mais intimamente daqui por diante.

Listagem 11. Inclusão de atributos e métodos na classe Desenhavel

```
this.mover = function() {
};
//...
this.iniciar = function(x, y, largura, altura) {
  // Variáveis padrão do eixo cartesiano
  this.x = x;
  this.y = y;
  this.largura = largura;
  this.altura = altura;
}
```

Uma vez modificado o template desenhável, precisamos também incluir os novos objetos referentes ao nosso player e aos combos de poder que o mesmo lançará, dentro do nosso repositório de imagens. Vejamos na **Listagem 12** como fica o nosso código após estas alterações.

Observe que logo no início da função criamos mais dois objetos de imagem: o nosso player e o combo de ataque do mesmo. O restante do código serve para que iniciemos o jogo somente após termos todos os três objetos carregados, através da manipulação da variável contadora **numImagens**.

O motivo por termos criado esse mecanismo foi porque existe um bug no IE 9 ou inferior, chamado de *race condition* (condição de corrida), onde os recursos das imagens não carregavam em tempo hábil o suficiente para serem usadas pela função posterior. Dessa forma, garantimos que todos os recursos visuais estarão previamente disponíveis antes que o objeto de repositório seja chamado. Isso também justifica o porquê de termos removido a chamada à função iniciar() no onload do corpo da página inicial.

Listagem 12. Código de repositório de imagens atualizado.

```
var repositorio = new function() {
  // Define os objetos de imagens
  this.planofundo = new Image();
  this.player = new Image();
  this.combo = new Image();

  var numImagens = 3;
  var numCarregados = 0;
  function imgCarregada() {
    numCarregados++;
    if (numCarregados === numImagens) {
      window.iniciar();
    }
  }
  this.planofundo.onload = function() {
    imgCarregada();
  }
  this.player.onload = function() {
    imgCarregada();
  }
  this.combo.onload = function() {
    imgCarregada();
  }

  // Configura os caminhos (src) das imagens
  this.planofundo.src = "imgs/pf.png";
  this.player.src = "imgs/player1.png";
  this.combo.src = "imgs/combo1.png";
}
```

Agora que temos os objetos referentes às imagens no repositório, vamos começar criando os objetos de ação mais simples, os combos de ataque do player. Veja a **Listagem 13**.

O objeto de Combo consta de apenas três métodos: o método de configuração, que irá receber os valores das variáveis de localização dos objetos de combo, mais o valor da velocidade que adicionamos anteriormente ao template desenhável; o método de desenho que irá efetuar um trabalho de pintura da imagem semelhante ao que fizemos no plano de fundo; e o método de limpeza, que reinicia os valores das propriedades que estamos usando no mesmo objeto. Repare que também criamos uma variável de classe "vivo" para verificar se o objeto encontra-se impresso na tela ou não.

Quando pensamos em um nível mais crítico em relação à forma como os objetos serão criados e usados na memória do navegador, por muitas vezes achamos que o espaço é infinito e os recursos ilimitados. No caso do nosso jogo, se analisarmos a quantidade de combos que o player fará, criar cada um deles e destruí-los em seguida seria um desperdício de recursos que poderiam ser reaproveitados, já que não tem características tão particulares assim.

Listagem 13. Criando o objeto de combos.

```
function Combo() {
   this.vivo = false; // Será marcado como true se o combo estiver em uso

   /*
    * Valores dos combos
                  */
   this.configurar = function(x, y, velocidade) {
      this.x = x;
      this.y = y;
      this.velocidade = velocidade;
      this.vivo = true;
   };

   /*
    * Função que desenha os combos
    */
   this.desenhar = function() {
      this.context.clearRect(this.x, this.y, this.largura, this.altura);
      this.x += this.velocidade;
      if (this.x <= 0 - this.largura) {
         return true;
      }
      else {
         this.context.drawImage(repositorio.combo, this.x, this.y);
      }
   };

   /*
    * Reinicia as propriedades do combo
    */
   this.limpar = function() {
      this.x = 0;
      this.y = 0;
      this.velocidade = 0;
      this.vivo = false;
   };
}
Combo.prototype = new Desenhavel();
```

Para fazer isso, poderíamos criar um **pool de objetos**, isto é, uma estrutura de dados que reusa objetos velhos e, em consequência, não precisa criar ou remover novos, diminuindo esse *lack* de memória nessa parte da implementação. Adicione, portanto, o código da **Listagem 14** ao nosso arquivo de script.

Quando o pool é inicializado, nós populamos um vetor com vários objetos Combo vazios, que serão reaproveitados, conforme definimos. Quando o pool precisa criar um novo combo para usar, ele confere o último item no vetor e verifica se ele está atualmente em uso. Caso positivo, o pool está cheio e nenhum combo pode ser reusado. Em caso negativo, o pool libera o último item do vetor e então o adiciona de volta ao início do mesmo. Isso permite que o pool consiga usar todos os combos que lhe cabem.

Quando o mesmo pool "anima" os combos, ele verifica se o combo está em uso e se tiver, o desenha. Se a função desenhar() retorna true, então o combo está pronto para ser reusado e o pool limpa o combo e o remove do vetor, o adicionando no final do mesmo. Esse reaproveitamento melhora em muito a performance do aplicativo.

Listagem 14. Código de criação do objeto de pool do jogo.

```
function Pool(tamanhoMax) {
   var tamanho = tamanhoMax; // Máximo de combos permitidos no pool
   var pool = [];

   /*
    * Popula o vetor de pool com objetos de combo
    */
   this.iniciar = function() {
      for (var i = 0; i < tamanho; i++) {
         // Inicializa o objeto de combo
         var combo = new Combo();
         combo.iniciar(0,0, repositorio.combo.width,
                  repositorio.combo.height);
         pool[i] = combo;
      }
   };

   /*
    * Pega o último item da lista e inicializa-o e
    * empurra-o para a frente do vetor.
    */
   this.get = function(x, y, velocidade) {
      if(!pool[tamanho - 1].vivo) {
         pool[tamanho - 1].configurar(x, y, velocidade);
         pool.unshift(pool.pop());
      }
   };

   /*
    * Usado para que o player seja capaz de obter dois combos de uma só vez. Se
    * apenas a função get() for usada duas vezes, o player é capaz de
    * de atacar com um combinação de 1 em vez de 2.
    */
   this.getDois = function(x1, y1, velocidade1, x2, y2, velocidade2) {
      if(!pool[tamanho - 1].vivo &&
         !pool[tamanho - 2].vivo) {
            this.get(x1, y1, velocidade1);
            this.get(x2, y2, velocidade2);
          }
   };

   /*
    * Desenha quaisquer combos. Se um combo vai para fora da tela,
    * ele o limpa e o empurra para a frente do vetor.
    */
   this.animar = function() {
      for (var i = 0; i < tamanho; i++) {
         // Apenas desenha quando encontrarmos um combo que não está vivo
         if (pool[i].vivo) {
            if (pool[i].desenhar()) {
               pool[i].limpar();
               pool.push((pool.splice(i,1))[0]);
            }
         }
         else
            break;
      }
   };
}
```

Com o terreno pronto para os combos e o pool que os gerencia, agora podemos finalmente criar o player. Vamos a analisar o script de criação da **Listagem 15**.

Da mesma forma que definimos a velocidade para os objetos de plano de fundo e combos, agora iniciamos a configuração do objeto Player definindo uma velocidade de três pixels/frame, além de criar o pool de combos e o inicializar com 50 objetos pré-criados.

Listagem 15. Script de criação do objeto de player principal do jogo.

```
function Player() {
  this.velocidade = 3;
  this.poolCombos = new Pool(50);
  this.poolCombos.iniciar();

  var intervaloTiros = 15;
  var cont = 0;

  this.desenhar = function() {
    this.context.drawImage(repositorio.player, this.x, this.y);
  };
  this.mover = function() {
    cont++;
    // Determina se vamos mover o player
    if (STATUS_CHAVES.left || STATUS_CHAVES.right ||
      STATUS_CHAVES.down || STATUS_CHAVES.up) {
      // Assim que o player for redesenhado, apagamos a imagem antiga
      this.context.clearRect(this.x, this.y, this.largura, this.altura);

      // Atualiza as coordenados dos eixos e redesenha.
      if (STATUS_CHAVES.left) {
        this.x -= this.velocidade
        if (this.x <= 0) // Mantém o player dentro da tela
          this.x = 0;
        } else if (STATUS_CHAVES.right) {
        this.x += this.velocidade
        if (this.x >= this.larguraCanvas/8*2)
          this.x = this.larguraCanvas/8*2;
      } else if (STATUS_CHAVES.up) {
        this.y -= this.velocidade
        if (this.y <= this.alturaCanvas/100)
          this.y = this.alturaCanvas/100;
      } else if (STATUS_CHAVES.down) {
        this.y += this.velocidade
        if (this.y >= this.alturaCanvas - this.altura)
          this.y = this.alturaCanvas - this.altura;
      }

      // Finaliza redesenhando o player
      this.desenhar();
    }

    if (STATUS_CHAVES.space && cont >= intervaloTiros) {
      this.atirar();
      cont = 0;
    }
  };

  /*
  * Atira dois Combos
  */
  this.atirar = function() {
  this.poolCombos.getDois(this.x+46, this.y +16, 3,
  this.x+73, this.y + 26, 3);
    };
}
Player.prototype = new Desenhavel();
```

Além disso, também criamos a variável **intervaloTiros**, que irá definir quanto tempo após um combo o outro será lançado (considerando que estamos implementando tiros duplos por vez).

Na sequência, temos três funções principais:

• A função **desenhar()** faz o mesmo trabalho das anteriores, capturando o objeto respectivo do repositório.

• A função **mover()** é um pouco mais arrojada, pois é ela a responsável por processar os movimentos dos objetos através da captura dos eventos das teclas de navegação, verificando quando o objeto estiver nos limites do topo, esquerda ou base do jogo, para que não os ultrapasse, assim como estipulando um limite invisível à direita, impedindo que o player não se aproxime demais dos inimigos. É nessa função também que verificamos o clique na barra de espaço do teclado, que será responsável por disparar os combos, chamando a função atirar(), assim como estipulando um limite de tempo entre cada tiro.

• E por último, a função **atirar()** será responsável por chamar a função getDois() do Combo para disparar dois combos por vez, calculando a distância x, y pré-estipulada dos mesmos.

E já que estamos falando sobre como manipular as teclas de input do teclado, em vez de usarmos um controle manual das mesmas, vamos usar uma abordagem desenvolvida originalmente pelo programador **Doug McInnes** (vide seção **Links**) que sugeriu uma forma mais performática e produtiva de lidar com a movimentação de objetos do tipo "asteroides", com base na inércia e outras fórmulas físicas. Vejamos na **Listagem 16** qual código devemos acrescentar ao nosso arquivo de scripts para isso.

O código inicialmente cria um objeto JSON que contém as keycodes (códigos-chave) para cada tecla do teclado, por exemplo a seta esquerda é representada pela constante "left", a direita pela "right" e assim por diante. Ele também é responsável por criar um vetor de status de true/false para que saibamos quando uma tecla foi pressionada. Repare no uso que fazemos das funções de eventos do JavaScript no DOM, onkeydown() e onkeyup(), neste caso usadas para verificar o exato momento em que as chaves mudam de valor. Estamos usando ambas as funções por causa do caráter de teste do jogo, no que se refere a quando o jogador pressionar muito rapidamente a tecla.

Para finalizar a implementação basta que modifiquemos o objeto Jogo e a função animar() para lidar com os novos objetos criados e suas respectivas propriedades. Para isso, modifique os recursos em questão para os demonstrados na **Listagem 17**.

Agora nosso objeto Jogo está preparado para recuperar e configurar cada um dos canvas em camadas que criamos na nossa página HTML, bem como suas propriedades de largura e altura. Note que a forma como preenchemos a apresentação do objeto player difere do plano de fundo, por precisarmos calcular uma média das dimensões da tela e posicionar nosso objeto exatamente no meio à esquerda da mesma. Ao final, o método jogar() chama o método desenhar() do player para o pintar na tela, e a função **animar()** movimenta o objeto animando-o em seguida.

Se o leitor desejar alterar o posicionamento deste objeto e dos demais, deverá seguir essa linha de raciocínio, alterando as propriedades numéricas do método iniciar(), onde passamos os parâmetros de exibição. Outrossim, é importante sempre manter as proporções entre tamanho do canvas e tamanho

dos objetos com operações como divisão e multiplicação, assim nossos objetos se redimensionarão ante os diferentes tamanhos de telas. Há cada novo objeto que criarmos no jogo, essa ordem deverá ser atendida no momento de acrescentá-los: recuperar o canvas, definir suas propriedades, configurar sua localização na tela e chamar o método de desenho quando iniciar.

Listagem 16. Código de controle das teclas e entradas do usuário.

```
// Os códigos de teclas que serão mapeados quando o usuário pressiona um botão.
// Código original por Doug McInnes
CODIGOS_TECLAS = {
  32: 'space',
  37: 'left',
  38: 'up',
  39: 'right',
  40: 'down',
}

// Cria uma matriz para guardar os CODIGOS_TECLAS e define todos os seus valores
// como false. Verificando true/false é a maneira mais rápida para verificar o estado
// de uma tecla pressionada e qual foi pressionada ao determinar
// quando mover e em qual direção.
STATUS_CHAVES = {};
for (codigo in CODIGOS_TECLAS) {
  STATUS_CHAVES[CODIGOS_TECLAS[codigo]] = false;
}
/**
 * Configura o documento para ouvir eventos onkeydown (disparado quando
 * qualquer tecla do teclado é pressionada para baixo). Quando uma tecla é
* pressionada, ele define a direção adequada para true para que
* possamos saber qualchave era.
 */
document.onkeydown = function(e) {
  // Firefox and opera use charCode instead of keyCode to
  // return which key was pressed.
  var keyCode = (e.keyCode) ? e.keyCode : e.charCode;
  if (CODIGOS_TECLAS[keyCode]) {
e.preventDefault();
STATUS_CHAVES[CODIGOS_TECLAS[keyCode]] = true;
  }
}
/**
 * Configura o documento para ouvir eventos ownkeyup (disparado quando
 * qualquer tecla do teclado é liberada). Quando uma tecla é liberada,
 * ele define a direção adequada para false para que possamos saber qual
 * chave era.
 */
document.onkeyup = function(e) {
  var keyCode = (e.keyCode) ? e.keyCode : e.charCode;
  if (CODIGOS_TECLAS[keyCode]) {
    e.preventDefault();
    STATUS_CHAVES[CODIGOS_TECLAS[keyCode]] = false;
  }
}
```

Listagem 17. Código de atualização do objeto Jogo e função animar().

```
function Jogo() {
    this.iniciar = function() {
        // Recupera o elemento canvas
        this.pfCanvas = document.getElementById('planoDeFundo');
        this.playerCanvas = document.getElementById('player_mov');
        this.principalCanvas = document.getElementById('principal');
        var retorno = false;

        // Testa para verificar se o canvas é suportado
        if (this.pfCanvas.getContext) {
            // inicializa o objeto de plano de fundo
            this.planofundo = new PlanoFundo();
            this.planofundo.iniciar(0,0); // Inicia no ponto 0,0

            this.pfContext = this.pfCanvas.getContext('2d');
            this.playerContext = this.playerCanvas.getContext('2d');
            this.principalContext = this.principalCanvas.getContext('2d');

            // inicializa os objetos configurando as propriedades em questão
            PlanoFundo.prototype.context = this.pfContext;
            PlanoFundo.prototype.larguraCanvas = this.pfCanvas.width;
            PlanoFundo.prototype.alturaCanvas = this.pfCanvas.height;

            Player.prototype.context = this.playerContext;
            Player.prototype.larguraCanvas = this.playerCanvas.width;
            Player.prototype.alturaCanvas = this.playerCanvas.height;

            Combo.prototype.context = this.principalContext;
            Combo.prototype.larguraCanvas = this.principalCanvas.width;
            Combo.prototype.alturaCanvas = this.principalCanvas.height;

            // Inicializa o objeto player
            this.player = new Player();
            // Configura o player para aparecer no meio da tela à esquerda
            var playerIniX = repositorio.player.width / 4;
            var playerIniY = this.playerCanvas.height / 3 + repositorio.player.height;
            this.player.iniciar(playerIniX, playerIniY, repositorio.player.width,
            repositorio.player.height);

            retorno = true;
        }
        return retorno;
    };

    // Inicia o loop de animação
    this.jogar = function() {
        this.player.desenhar();
        animar();
    };
}

function animar() {
    getFrameAnimacao( animar );
    jogo.planofundo.desenhar();
    jogo.player.mover();
    jogo.player.poolCombos.animar();
}
```

Com tudo isso configurado, o resultado da execução do jogo será semelhante ao da **Figura 5**.

Figura 5. Resultado da execução do jogo com player e combos

Se o leitor tiver configurado tudo corretamente, esse será o resultado. Caso o jogo não apareça exatamente como na figura mostrada, certifique-se de refazer os passos para garantir que o código confere. Adicionalmente, você também pode clicar no atalho F12 dentro do browser e a tela de depuração para desenvolvedores irá abrir. Se algo de errado estiver acontecendo no JavaScript ela irá mostrar mensagens esclarecendo o que é.

Autor

Júlio Sampaio
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 do Notepad++
http://notepad-plus-plus.org/download/

Página do projeto ExplorerCanvas
http://code.google.com/p/explorercanvas/

Página do algoritmo de otimização de animação
http://www.paulirish.com/2011/requestanimationframe-for-smart-animating/

Página da técnica canvas do Doug McInnes
http://www.dougmcinnes.com/2010/05/12/html-5-asteroids/

Analisando arquiteturas client-side com AngularJS

Aprenda a selecionar a melhor arquitetura criando um projeto de exemplo em JavaScript

O desenvolvimento de uma arquitetura client-side traz consigo desafios arquiteturais equivalentes ao modelo tradicional implementado no lado servidor. Então como podemos modelar e implementar uma arquitetura no lado cliente capaz de atender aos requisitos arquiteturais de software de um determinado domínio?

Para o propósito deste artigo foi selecionado o domínio de uma rede social para realizar a implementação baseada nos requisitos da arquitetura que serão descritos empregando o modelo FURPS, um acrônimo que representa um modelo de classificação de atributos de qualidade de software (requisitos funcionais e não funcionais), e que classifica os requisitos arquiteturais por funcionalidade, usabilidade, confiabilidade, desempenho e suportabilidade.

Neste contexto, a escolha do framework AngularJS foi realizada devido a sua popularidade e ao fato de suas características serem compatíveis com uma arquitetura robusta que possibilita a separação em camadas, assim como outras características importantes como a injeção de dependências.

Fique por dentro

Este artigo tem como objetivo identificar os padrões de projeto e as características de uma arquitetura para aplicações web client-side, para então propor um modelo de arquitetura utilizando o framework AngularJS, de forma que atenda aos requisitos arquiteturais de uma aplicação exemplo. Na prática, demonstraremos um conjunto de técnicas para implementar uma arquitetura executável e reutilizável, exemplificando a estrutura do projeto, padrões utilizados, ferramentas e código-fonte, bem como os demais atores envolvidos no ciclo de vida do software de uma forma geral.

Arquitetura de Software

Segundo Martin Fowler, a arquitetura de *software* é um termo com muitas definições, porém sem um consenso entre os autores. No entanto, ele afirma que sempre existem dois elementos em comum, o primeiro é o desmembramento do sistema em partes e o segundo são as decisões difíceis de mudar.

A arquitetura de *software* de um programa ou sistema computacional é composta pelas estruturas do sistema, que, por sua vez, são compostas por elementos de *software*, suas propriedades visíveis externamente e as relações entre elas. As formas arquiteturais recorrentes, depois de observadas e analisadas, podem gerar modelos padronizados, que podem ser utilizados mesmo em sistemas diferentes na forma dos chamados "estilos arquiteturais", modelos de representação de arquitetura.

O estilo arquitetural mais utilizado em aplicações complexas é o estilo em camadas, também conhecido como *layers*, no qual ele define como um conjunto de subsistemas do *software* organizados em forma de bolo, onde cada camada é acomodada sobre a camada inferior. Nesta estrutura a maior camada utiliza os serviços das camadas inferiores, mas as camadas inferiores não conhecem as camadas superiores e por isso não acessam os seus serviços. Entre os benefícios disso, podemos citar: coesão; separação de responsabilidades; redução de dependências; possibilidade de substituição da implementação.

Padrões de Projeto

O *software* é frequentemente lembrado pelas suas partes como as funções, arquivos fonte, módulos, objetos, métodos, classes, pacotes, bibliotecas, componentes, serviços, subsistemas e assim por diante. Tudo isto representa visões corretas nas quais os desenvolvedores trabalham. Porém, esta visão é focada nas partes, que por sua vez não contemplam a grande quantidade de relacionamentos e decisões que estão por trás do *software*.

Os padrões de projeto, também conhecidos como *design patterns*, surgem para descrever, capturar e nomear técnicas para solução de problemas de *software*. Diferentemente da utilização

de experiência de forma empírica, os padrões são documentados, reutilizados em outras aplicações e compartilhados para disseminação do conhecimento.

Os principais padrões de projetos que serão empregados neste artigo estão descritos a seguir.

Dependency Injection

O padrão DI (*Dependency Injection*) é uma prática onde os objetos são desenhados de uma maneira na qual recebem instâncias de objetos de outras partes do código, ao invés de construí-los internamente. Isto significa que qualquer implementação da interface que é utilizada pelo objeto pode ser substituída sem alterar o código, resultando na simplificação dos testes e na diminuição do acoplamento.

Service Oriented Architecture

O padrão SOA (*Service Oriented Architecture*) consiste em uma coleção de componentes distribuídos que fornecem e/ou consomem serviços. Neste padrão, os componentes dos fornecedores e dos consumidores de serviços podem utilizar diferentes plataformas e linguagens de programação. Serviços são em sua grande maioria independentes e frequentemente pertencem a diferentes sistemas ou até mesmo diferentes organizações.

RESTful Web Services

O REST (*Representational State Transfer*) é um estilo arquitetural para aplicações em rede proposto por **Roy Fielding** em 2000. Ele consiste em um conjunto de restrições para tratar a separação de responsabilidades, visibilidade, confiabilidade, escalabilidade, desempenho, etc. O grande diferencial do REST é a utilização da infraestrutura da web, como por exemplo, o protocolo HTTP, que permite construir aplicações distribuídas e publicar serviços nesta infraestrutura. Pode ser denominado *RESTful Web Services*, os serviços web construídos utilizando as tecnologias HTTP, URI, XML e JSON que são padronizadas para utilização na web.

Active Record

O *Active Record* é um padrão arquitetural para mapeamento de objetos de domínio a estruturas de dados, representando uma tabela ou um documento e encapsulando o acesso ao repositório de dados. Ele é recomendado para domínios com pouca lógica de negócio e que se aproximam muito da modelagem dos dados.

Requisitos Arquiteturais

Para analisar os requisitos arquiteturais, primeiramente precisamos compreender como identificá-los e classificá-los.

Um requisito de software pode ser chamado de requisito arquitetural sempre que for significativo para a arquitetura. Eles ainda podem ser classificados como implícitos ou explícitos, os primeiros são requisitos com particularidades que os caracterizam como significantes para a arquitetura, como por exemplo: alto risco, alta prioridade ou baixa estabilidade. Já os requisitos arquiteturais explícitos são aqueles de ordem técnica, como por exemplo: desempenho, escalabilidade ou tecnologia específica.

Modelo FURPS+

O modelo **FURPS+** é uma classificação de requisitos arquiteturais proposta por **Robert Grady**, em 1992, que pode auxiliar na identificação dos requisitos de um sistema e na sua organização e classificação na especificação de requisitos.

As letras do acrônimo significam respectivamente: funcionalidade (*functionality*), usabilidade (*usability*), confiabilidade (*reliability*), desempenho (*performance*) e suportabilidade (*supportability*). Já o símbolo "+" é utilizado para representar restrições impostas à solução, que também são consideradas como requisitos, como por exemplo, as restrições de design, de implementação, de interface e físicas.

Cada classificação pode ser descrita por diversas características e essas também podem agrupar outras características em sua essência conforme lista a seguir.

- Funcionalidade: auditoria, gerenciamento de erros, gerenciamento do sistema, localização, segurança, licenciamento, transações, entre outros;
- Usabilidade: acessibilidade, aparência e interação do usuário;
- Confiabilidade: precisão, disponibilidade e recuperabilidade;
- Desempenho: eficiência, escalabilidade e velocidade;
- Suportabilidade: adaptabilidade, compatibilidade, configurabilidade, instalabilidade, manutenibilidade e testabilidade.

Aplicações Web Client-side

Com a evolução da web, os *frameworks server-side* irão migrar cada vez mais para o cliente, fazendo com que mais códigos sejam criados e executados no navegador web. Com isso, os códigos criados em JavaScript irão aumentar e a necessidade de organizar e estruturar as aplicações *client-side* demandará a necessidade da modelagem da arquitetura *client-side* e da utilização de *design patterns*. Além disso, novos frameworks irão surgir para atender essas novas demandas.

AngularJS

O AngularJS é um *framework* MVC *client-side* escrito em JavaScript, que é executado no navegador web, e que proporciona aos desenvolvedores a possibilidade de criarem aplicações modernas e dinâmicas utilizando o conceito de SPA (*Single-page applications*). Ele é um *framework* de uso geral, isto é, pode ser utilizado para desenvolver qualquer tipo de aplicação, mas ele realmente se destaca na criação de aplicações web do tipo CRUD (*Create Read Update Delete*), que é normalmente utilizado em aplicações corporativas.

O AngularJS é um *framework* relativamente novo no cenário atual de *frameworks* JavaScript. Com a versão 1.0 lançada em junho de 2012 e a última versão 1.3 lançada em outubro de 2014, ele atraiu muita atenção devido a suas características únicas como o inovador sistema de *client-side templates* e da facilidade de desenvolvimento, testes e manutenção. Além disto, eles destacam a criação e a manutenção do *framework* que é de responsabilidade do Google Inc., por uma equipe de engenheiros dedicada, e a licença *open source*, que é respeitada e há uma grande contribuição da comunidade.

Ele foi criado inicialmente para lidar com aplicações web *client-side* grandes, sem toda a complexidade que era inerente ao desenvolvimento neste tipo de arquitetura, pois a HTML fora criada para páginas estáticas e nos *frameworks* atuais não se tinha a separação de responsabilidades, entre outros problemas estruturais. Eles queriam resolver estes problemas fazendo com que a criação de aplicações fosse simplificada, mas ao mesmo tempo o desenvolvedor tivesse o poder de tomar decisões de projeto, principalmente para aplicações de grande porte.

Criando Aplicação de Exemplo

Para o propósito deste artigo, será implementada uma aplicação de exemplo de uma rede social para a publicação de textos e comentários. Esta escolha foi realizada devido à grande quantidade de requisitos arquiteturalmente significativos que este tipo de aplicação exige, diferentemente de outros domínios de aplicação baseados apenas em formulários de cadastro.

Os requisitos arquiteturais para o domínio de software de uma rede social foram descritos e classificados utilizando o modelo FURPS+ e estão descritos na **Tabela 1**.

Análise

Em aplicações escritas em AngularJS, a separação de camadas é clara e o principal modelo empregado é o MVC, no qual a **visão** é o DOM (*Document Object Model*), que é a estrutura do HTML em memória, os **controladores** são classes JavaScript e o **modelo** são os dados armazenados nas propriedades dos objetos JavaScript. Este padrão possibilita uma melhor organização do código, gerando grandes benefícios como a facilidade de extensão, manutenção e testes.

Para atender aos requisitos R03, R07 e R08, será aplicado o padrão MVC *client-side*, utilizando o *framework* AngularJS. Ele delega toda a responsabilidade pelo fluxo de apresentação da aplicação para o lado cliente. Este modelo faz com que a lógica do cliente seja separada do servidor, não importando a tecnologia utilizada no servidor, porém atendendo ao contrato dos serviços utilizados pelo cliente.

Normalmente as aplicações web criam as páginas HTML utilizando *templates* e juntando os dados para montar todo o conteúdo da página no servidor, para então enviar ao *browser*. No AngularJS, os *templates* e os dados são trazidos do servidor para serem montados no *browser*, então o servidor passa a servir apenas páginas estáticas e dados via algum tipo de serviço. Este recurso é inovador e tem por trás grandes decisões de projeto, além de possuir alguns benefícios como a redução do tráfego de dados entre o cliente e o servidor, diminuindo o uso da banda e a melhoria da usabilidade.

Além disso, o padrão *MVC client-side* e as suas características fazem com que toda a comunicação com o servidor seja realizada por meio de *web services* e os componentes do servidor centralizem apenas a lógica de negócio da aplicação. Desta forma, os padrões SOA e REST são inerentemente aplicáveis nesta arquitetura e em decorrência disso, atendem aos requisitos R02 e R05 que dizem respeito a disponibilidade e performance. Assim, uma aplicação servidor que implementa os padrões SOA e REST de forma estrita é escalável, pois os serviços resultantes são *stateless* (não mantém o estado entre as requisições), e aplicam técnicas de processamento distribuído e cache, garantindo uma resposta rápida para o cliente e uma escalabilidade linear.

A diretiva (*directive*) é um recurso destaque do AngularJS, que permite a implementação de *templates* HTML de forma estática ou dinâmica, estendendo a sintaxe do HTML para a criação ou alteração de componentes e a integração com componentes de terceiros.

Para uma melhor experiência do usuário e atendimento dos requisitos R01 e R04, serão implementadas interfaces de usuário simplificadas para o público alvo, utilizando bibliotecas de componentes web para criar interfaces responsivas para todos os tamanhos de telas e acessíveis em qualquer dispositivo. Estes objetivos serão alcançados utilizando o Bootstrap (vide seção **Links**) e suas extensões para o AngularJS.

Código	Requisito Arquitetural	FURPS+
R01	Interface visual simples e intuitiva. O usuário de qualquer nível de experiência de informática deverá utilizar a aplicação sem nenhuma ajuda ou manual.	Usabilidade
R02	Todos os componentes do sistema deverão estar disponíveis 99,99% do tempo medido semanalmente, isto é, terá tolerância a falhas de 1 minuto por semana. Sem tolerância de interrupção do serviço por manutenção programada.	Confiabilidade
R03	A aplicação cliente deverá ser web e compatível com o padrão HTML5, isto é, funcionará em todos os dispositivos que aplicam este padrão, inclusive dispositivos móveis.	Funcionalidade
R04	A interface da aplicação deverá se adaptar a qualquer tamanho de tela ou resolução, sem prejudicar a aparência e acessibilidade.	Usabilidade
R05	A aplicação deverá ser escalável para qualquer número de usuários, apenas adicionando mais instâncias do servidor conforme o número de usuários simultâneos.	Performance
R06	Os componentes da aplicação deverão ser reutilizáveis, extensíveis e testáveis para que o desenvolvimento de novas funcionalidades não afete o comportamento da aplicação.	Suportabilidade
R07	Deve haver uma separação lógica entre a aplicação cliente e a aplicação servidor, a primeira consumirá os serviços da segunda para prover os dados e realizar as ações.	Restrição de design
R08	A aplicação cliente deve rodar inteiramente no navegador web e utilizar a linguagem JavaScript em conjunto com o framework AngularJS.	Restrição de implementação

Tabela 1. Requisitos Arquiteturais da aplicação

A injeção de dependências do AngularJS é um recurso inovador nos *frameworks client-side*, que permite utilizar um estilo de desenvolvimento no qual em vez de instanciar as dependências manualmente, as classes apenas solicitam o que elas precisam.

Desta forma, o atendimento do requisito R06, que diz respeito ao reuso, extensão e testes dos componentes será atendido utilizando os padrões DI e *Active Record*, que são recomendados para reduzir o acoplamento entre os mesmos, possibilitando a substituição da implementação sem a necessidade de alteração de outros componentes, bem como facilitando os testes. No AngularJS, todos os componentes que são gerenciados por ele suportam a injeção de dependências, desta forma, praticamente toda a aplicação pode se comunicar com um baixo nível de acoplamento. Um exemplo claro é a variável *$scope*, presente em todas as classes de controladores.

A implementação do servidor, no qual a aplicação cliente será baseada, implementa serviços no modelo *RESTful Web Services*, utilizando o formato de comunicação JSON (*JavaScript Object Notation*) sobre o protocolo HTTP. Os recursos que serão expostos como serviços para a comunicação com a aplicação cliente serão dispostos em uma estrutura de nomes utilizando o formato URI (*Uniform Resource Identifier*) descritos na **Listagem 1**.

Listagem 1. Modelagem dos serviços utilizando o padrão REST.

```
/user -> Informações do usuário logado
/user/post -> Publica uma nova postagem
/post/{id} -> Informações da postagem
/post/{id}/like -> Realizar o "like" da postagem
/people -> Listagem de pessoas da rede social
/person/{id} -> Informações da pessoa
/person/{id}/addFriend -> Adicionar a pessoa como amigo
/person/{id}/friends -> Listagem de amigos da pessoa
/person/{id}/timeline -> Listagem das postagens da pessoa
```

Antes de iniciar a implementação de uma aplicação web *client-side*, é preciso descrever o seu modelo estrutural, que é fundamental para a organização do código-fonte e facilita a manutenção e evolução da mesma.

No nosso modelo, a melhor divisão dos arquivos da aplicação é proporcionando que a mesma seja realizada por componentes e funcionalidades. Desta forma, podemos dispor os arquivos, conforme a **Listagem 2**. Perceba que a listagem é auto comentada, então cada arquivo vem com sua descrição em sequência.

Implementação

Para a implementação da aplicação de exemplo, deverá ser realizada a preparação das ferramentas utilizadas no desenvolvimento e as configurações iniciais do projeto. Primeiramente deverá ser configurado o versionamento do código-fonte utilizando a ferramenta de controle de versão de sua preferência. É recomendável utilizar o **Git** no serviço gratuito **GitHub** (seção **Links**), que também disponibiliza um servidor para a publicação de páginas estáticas.

Listagem 2. Estrutura de arquivos da aplicação.

```
/ -> Diretório raiz com as configurações do projeto
/app -> Armazena todos os arquivos necessários para a aplicação
index.html -> Arquivo principal da aplicação no formato SPA
/assets -> Armazena os recursos diversos utilizados pela aplicação (exemplo:
imagens, fontes)
/components -> Armazena os componentes visuais criados para a aplicação
(exemplo: directives, widgets);
/libs -> Armazena as bibliotecas de terceiros (exemplo: AngularJS, Bootstrap, etc.);
/states -> Armazena as telas da aplicação com a camada de visão e de controle
(exemplo: tela de login);
/scripts -> Armazena os arquivos de scripts da aplicação, assim como a camada de
modelo (app.js);
/styles -> Armazena as folhas de estilos globais da aplicação (exemplo: main.less);
/test -> Armazena os arquivos de testes automatizados com os mesmos diretórios
relativos aos arquivos da aplicação, porém com a lógica para a realização dos
testes.
```

No ambiente de desenvolvimento do projeto será utilizado um editor de arquivos para os formatos HTML, JS e CSS. Pode-se utilizar um editor de sua preferência, mas o ideal é que ele suporte HTML5, seja *free* e tenha boa integração com o AngularJS, como por exemplo o **Brackets**, cujo link de download pode ser encontrado na seção **Links** deste artigo.

Para a execução da aplicação, gerenciamento de dependências e construção do projeto foi utilizada a plataforma Node .js, juntamente com o Grunt, Bower e Karma para executar um servidor web na máquina local, gerenciar as dependências da aplicação e realizar os testes necessários.

As técnicas e práticas analisadas e a utilização das funcionalidades do *framework* serão demonstradas neste artigo por meio do código-fonte implementado para a funcionalidade de listagem de amigos da rede social, através do estilo arquitetural do MVC.

A camada de modelo é implementada utilizando serviços que são classes JavaScript injetáveis via DI e que possibilitam a utilização de boas práticas de programação como a orientação a objetos e a separação de responsabilidades. Um dos principais exemplos são os serviços *$http* e *$resource* que são responsáveis respectivamente pela interação com o HTTP e *RESTful Web Services*. No entanto, nesta aplicação foi utilizada uma biblioteca de terceiros chamada *angularjs-rails-resource* (certifique-se de efetuar o seu download via seção **Links**) utiliza o serviço *$http* para realizar chamadas no estilo REST.

A **Listagem 3** representa o código-fonte da camada de modelo do recurso da pessoa, um dos modelos utilizados pela funcionalidade em questão, que utiliza os padrões *RESTful Web Services*, *Active Record* e *Dependency Injection*, além das funcionalidades de injeção de dependências e serviços do *framework*. Esta listagem demonstra a criação de um serviço REST para a pessoa, configurando o endereço da API no servidor e as referências para outros serviços que estão vinculados às propriedades do objeto da pessoa, como a listagem dos amigos e a *timeline*. Além disso, será usado o método **addFriend** no serviço para adicionar a pessoa como amigo do usuário logado que executou a chamada.

Listagem 3. Camada de modelo da pessoa, arquivo: app/scripts/models/person.js.

```
'use strict';

angular.module('app')
.factory('PersonResource', function(railsResourceFactory, railsSerializer, Config) {
    var Person = railsResourceFactory({
      url: Config.API.url + 'person',
      name: 'person',
      serializer: railsSerializer(function() {
        this.resource('friends', 'PersonResource');
        this.resource('timeline', 'PostResource');
      })
    });

    Person.prototype.addFriend = function() {
      return Person.$post(this.$url() + '/addFriend');
    };

    return Person;
});
```

A **Listagem 4** representa o código-fonte da camada de visão da tela de listagem de amigos, que utiliza as funcionalidades de *client-side templating*, *data binding*, *directives* e *filters* do *framework*.

O recurso de *data binding* possibilita realizar o mapeamento de componentes visuais para propriedades de objetos JavaScript e sincronizá-los automaticamente, fazendo com que sejam eliminados códigos desnecessários para a sincronização entre as camadas de visão e do modelo.

Já o filtro é um recurso que permite alterar a forma de demonstração dos dados, realizando uma filtragem ou convertendo dados que serão exibidos para o usuário, e transformando dados brutos em formatos legíveis como, por exemplo, a conversão de uma data e hora em UTC / *Unix Epoch* (Ex.: 1388534400), para um formato legível (Ex.: 01/01/2014 00:00:00).

Nesta listagem é utilizada a diretiva **ng-repeat** para iterar sobre a listagem de amigos e ordená-los por nome.

Listagem 4. Camada de visão da listagem de amigos, arquivo: app/states/friends/friends.html.

```
<div class="row">
  <div class="col-xs-12 col-sm-12 col-md-12" cg-busy="{promise: promise}">
    <h3 class="cv-title">Friends</h3>
    <div class="well well-sm" ng-repeat="friend in friends | orderBy:'name'">
      <div class="row">
        <div class="col-xs-12">
          <i class="glyphicon glyphicon-user pull-left"></i>
           Name: {{friend.name}}<br />
           Gender: {{friend.gender}}<br />
           Birthdate: {{friend.birthDate | date}}
        </div>
      </div>
    </div>
  </div>
</div>
```

Já a **Listagem 5** representa o código-fonte da camada de controle da tela de listagem de amigos, que utiliza a camada de modelo da pessoa e as funcionalidades de injeção de dependências, *data binding* e serviços do *framework*.

Listagem 5. Camada de controle da listagem de amigos, arquivo: app/states/friends/friends.js.

```
'use strict';

angular.module('app')
.config(function($stateProvider) {
    $stateProvider.state('friends', {
      url: '/friends',
      templateUrl: 'states/friends/friends.html',
      controller: 'FriendsCtrl'
    });
  })
  .controller('FriendsCtrl', function($scope, UserResource) {
    $scope.promise = UserResource.query().then(function(user) {
      $scope.user = user;

      user.person.get().then(function(person) {
          $scope.friends = person.friends;

        angular.forEach(person.friends, function(item) {
          item.get();
        });
      });
    });
  });
```

O *data binding* é realizado utilizando a variável *$scope*, que é a "cola" entre o controller e a view e é injetado em todos os controladores. Esta variável é hierárquica, isto é, a cada nível de controlador ela pode herdar e reutilizar variáveis definidas em escopos anteriores. O escopo principal da aplicação é o *$rootScope* e também pode ser injetado no controlador.

Neste exemplo, primeiro definidos a rota do controlador utilizando a função config e injetando o gerenciador de rotas melhorado, o UI-Router. Depois é declarado o controlador utilizando o serviço de usuário para realizar uma query no servidor. Depois do usuário retornado, ele é atribuído ao escopo e também é realizada uma requisição para obter os amigos da pessoa relacionada ao usuário. Como os detalhes dos amigos são *lazy*, isto é, são carregados sob demanda, é necessário realizar uma iteração na lista para obter os detalhes.

Todas as operações HTTP são abstraídas do controlador, pois o serviço que foi implementado na camada de modelo se encarrega dos detalhes da implementação. Também vemos que foi utilizado o conceito de *Promises* ao invés dos *callbacks* normalmente utilizados em JavaScript para o controle de fluxo de operações assíncronas.

O **Promises** é um padrão de programação assíncrona no qual funções são executadas de forma assíncrona e podem ser satisfeitas, rejeitadas ou retornar uma exceção e o seu estado é armazenado e propagado para promises encadeadas. No JavaScript, elas são implementadas em várias bibliotecas e a partir do *ECMAScript 6* serão padronizadas na linguagem.

No AngularJS, elas são geradas utilizando o serviço *$q*.

Conforme o requisito R08, a implementação da aplicação cliente foi realizada utilizando a linguagem JavaScript com o *framework* AngularJS e seguindo o padrão MVC. O resultado foi uma aplicação executável em qualquer navegador web com suporte a HTML5, porém sem a persistência dos dados ou autenticação do usuário, pois os dados são pré-carregados para um usuário específico sob fins de demonstração.

O *layout* resultante é responsivo a qualquer tamanho de tela e com um nível de acessibilidade adequado para usuários com qualquer nível de experiência. Além disso, a aplicação funciona em dispositivos móveis, conforme exemplo exibido na **Figura 1**, que demonstra a aplicação executando num dispositivo móvel com o sistema operacional Android. Desta forma, a implementação atende aos requisitos R01, R03 e R04.

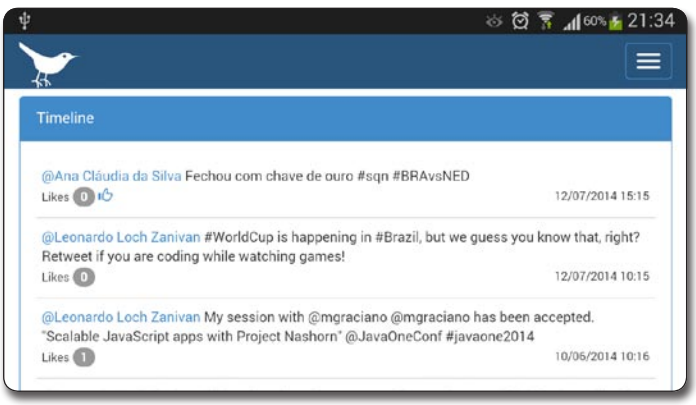

Figura 1. Tela principal da aplicação demonstrando a linha do tempo do usuário.

Apesar do objetivo da aplicação de exemplo ser a realização da implementação apenas do lado cliente, foi necessário simular os serviços do servidor para aplicar o padrão REST. Para isto, foram utilizadas técnicas de *mock* através da funcionalidade de injeção de dependências do AngularJS, que permite substituir uma implementação real por outra de teste. Esta característica atendeu ao requisito R06 e possibilitou que a aplicação cliente ficasse totalmente desacoplada da aplicação servidor, atendendo em conformidade ao requisito R07.

Em decorrência dos padrões aplicados, todas as características de escalabilidade e disponibilidade são atingíveis por provedores Cloud de IaaS (infraestrutura como serviço), como a *Amazon Web Services* e o *Microsoft Azure*, que permitem o aumento da infraestrutura de servidores de forma elástica e ilimitada, atendendo aos requisitos R02 e R05.

Além das funcionalidades do AngularJS utilizadas na aplicação de exemplo, existem muitas outras que merecem destaque e que são utilizadas internamente pelas bibliotecas de terceiros utilizadas para melhorar a produtividade no desenvolvimento e aplicação com AngularJS. Isto só é possível devido ao sistema modular do AngularJS, desta forma podemos adicionar recursos de autenticação, *caching*, internacionalização, localização, animações e rotas da aplicação integrados a aplicação existente.

Testes

Assim como qualquer software, aplicações escritas em JavaScript para o browser também precisam de automatização de testes. Por muito tempo aplicações web escritas em JavaScript sofreram por não terem suítes de testes adequadas para a escrita de testes unitários e *end-to-end*. Além disso, da forma com que eram escritas as aplicações JavaScript era praticamente impossível realizar testes unitários. Os testes E2E também acabavam ficando caros pela dificuldade de realizar mocks e a necessidade de um servidor real.

Para resolver estes problemas, o AngularJS foi concebido com testes automatizados em mente, inclusive o próprio framework é validado com uma grande suíte de testes escritos utilizando as ferramentas Karma e Protractor que o Google criou.

Podemos dividir os testes em unitários e *end-to-end*, os testes unitários já são bastante conhecidos e utilizados em várias plataformas e linguagens de programação. Eles tratam do teste das unidades do código JavaScript, como por exemplo testar os métodos de um controlador. Já os testes *end-to-end* ou E2E, são testes de ponta a ponta, executados em um browser real e validando o comportamento da aplicação em diversos browsers, além da integração entre os componentes e regressões (bugs).

Os testes são escritos utilizando o framework **Jasmine,** que é uma DSL específica para a escrita de testes em JavaScript. Para a execução dos testes unitários é utilizado o Karma, um executor de testes headless (sem browser), já para a execução dos testes E2E é utilizado o Protractor, porém utilizando o *Selenium Web Driver* para realizar a integração com os browsers. Todos os itens podem ser facilmente encontrados em seus respectivos sites oficiais.

Na **Listagem 6** é demonstrado um teste unitário básico que realiza a cobertura da lógica implementada no controlador de erros. Este controlador tem por objetivo receber um parâmetro com código de erro HTTP e atribuir a uma variável que será demonstrada na tela.

Para realizar o teste, o objeto $stateParams é criado para ser injetado no controlador com um código pré determinado, desta forma podemos garantir a assertividade do teste. Depois, o controlador é instanciado e armazenado em uma variável para que seja utilizado posteriormente nos testes declarados com a função "it".

Na **Listagem 7** é demonstrado um teste E2E básico, que tem por função testar o comportamento da aplicação quando é acessada a página raiz da aplicação, no arquivo index.html. Em testes E2E primeiramente é necessário instanciar o Protractor que funciona como uma espécie de browser, nele podemos navegar entre páginas e executar ações, porém tudo é realizado via programação. Neste teste navegamos até a página raiz "/" e verificamos se o título da página está de acordo com o esperado.

Para que seja possível testar a aplicação enquanto desenvolvemos e ainda não possuímos os RESTful Web Services implementados no servidor ou ainda não seja possível utilizar um servidor real, podemos utilizar o recurso de mock HTTP do AngularJS. Este recurso é chamado de $httpBackend e faz com que seja possível interceptar as chamadas para o serviço $http e simular respostas reais.

Listagem 6. Teste unitário do controlador da tela de erro, arquivo: test/unit/states/error/error.js

```
'use strict';

describe('Controller: ErrorCtrl', function () {
  var ErrorCtrl, scope;

  beforeEach(function () {
    module('app', function ($provide) {
      $provide.value('$stateParams', {
        code: 1337
      });
    });

    inject(function ($controller, $rootScope) {
      scope = $rootScope.$new();
      ErrorCtrl = $controller('ErrorCtrl', {
        $scope: scope
      });
    });
  });

  it('should attach error code', function () {
    expect(scope.errorCode).toEqual(1337);
  });
});
```

Listagem 7. Teste end-to-end da página inicial, arquivo: test/e2e/states/index.js

```
'use strict';

describe('Index route', function () {
  var ptor = protractor.getInstance();

  describe('index', function () {
    it('should display the correct title', function () {
      ptor.get('/#');
      expect(ptor.getTitle()).toContain('Social Network');
    });
  });
});
```

Além de facilitar na escrita de testes, este importante recurso pode possibilitar o desenvolvimento do front-end da aplicação sem depender de um servidor real ou implementar aplicações como protótipo ou prova de conceito para fins de validação.

Na **Listagem 8** é demonstrada a implementação de um mock do serviço de postagem que é realizado na inicialização da aplicação, por meio do método "run". Primeiramente é verificado se a aplicação está configurada para usar mocks, então os dados iniciais de postagens são carregados utilizando um arquivo no formato *json* que já possui os dados previamente registrados, depois são interceptados dois métodos do serviço. O primeiro método é o GET para a postagem utilizando o *id* e o segundo é o comando "like" utilizando o método POST para o *id* da postagem. O retorno das chamadas é como se fosse uma chamada HTTP real, com o conteúdo no formado JSON, o código de resposta e os cabeçalhos HTTP.

Também devemos testar a performance da aplicação enquanto desenvolvemos, por isso ela também deverá ser medida no lado cliente, isto é, no navegador web. Para realizar essa tarefa existem extensões para os principais navegadores que realizam essas medições, já no caso do AngularJS existe uma extensão específica para o navegador Chrome chamada **Batarang,** que demonstra claramente as variáveis no escopo e a performance dos *bindings*.

Como os recursos da máquina do usuário também são limitados, é necessário utilizar mecanismos que o próprio AngularJS disponibiliza para contornar esses problemas, entre eles podemos utilizar o serviço de cache, para armazenar dados e templates, desabilitar o *two-way data binding* do modelo utilizando o recurso de *bind once*, utilizar a opção "track by" da diretiva ng-repeat para reduzir o custo de manipulação do DOM, habilitar o tempo de espera para o binding do modelo utilizando a opção debounce na diretiva ng-model e implementar o carregamento sob demanda nas telas, também conhecido como **lazy loading**.

Listagem 8. Mock do RESTful Web Service de postagens, arquivo: app/mocks/post.js

```
'use strict';

angular.module('app')
  .run(function(Config, $httpBackend, $log, $http, PostRepository) {
    if (!Config.API.useMocks) {
      return;
    }

    var collectionUrl = Config.API.url + 'post';
    var IdRegExp = /[\d\w-_]+$/.toString().slice(1, -1);
    var postById = new RegExp(collectionUrl + '/' + IdRegExp);

    //Load mock data
    $http.get('mocks/json/post.json').then(function(response) {
      angular.forEach(response.data, function(item, key) {
        PostRepository.insert(item.id, item);
      });
    });

    //GET post/:id
    $httpBackend.whenGET(postById).respond(function(method, url, data,
    headers) {
      var id = url.match(new RegExp(IdRegExp))[0];
      return [200, PostRepository.getById(id), { /*headers*/ }];
    });

    //POST post/:id/like
    var IdRegExp2 = /[\d\w-_]+/.toString().slice(1, -1);
    $httpBackend.whenPOST(new RegExp(collectionUrl + '/' + IdRegExp2 + '/like'))
    .respond(function(method, url, data, headers) {
      var postId = url.match(new RegExp('(' + IdRegExp2 + ')(/like)'))[1];
      var post = PostRepository.getById(postId);
      post.likes ++;

      return [200, post, { /*headers*/ }];
    });
  });
```

AngularJS no Mundo Enterprise

É fato a dominância do AngularJS entre os frameworks JavaScript para desenvolvimento web. Não se sabe ao certo até quanto isso irá permanecer, o que se sabe é que se formou uma comunidade gigante e participativa ao seu redor e cada vez mais empresas e produtos dependem dela. Além disso, a integração entre ferra-

mentas de terceiros e IDEs com o AngularJS é algo que não vimos acontecer com outros frameworks JavaScript, reforçando o lado *enterprise* que está interessado nela.

A versão 2.0 já foi anunciada e está em fase de desenvolvimento, tem sua previsão inicial para o final de 2015, porém após declarações da equipe do AngularJS realizadas em Outubro de 2014 muito tem se especulado a respeito do futuro do framework e da quebra de compatibilidade com a versão 1.x.

A nova versão não será compatível com os códigos já escritos e também utilizará outra sintaxe, até mesmo recomendará outra linguagem, o **AtScript**, uma linguagem de script compatível com o TypeScript e Dart. Apesar de ser uma nova linguagem, ela será compilada em JavaScript e irá rodar nos navegadores atuais, com exceção de navegadores que não recebem mais atualização ou não são totalmente compatíveis com a HTML5, como o IE 9, por exemplo.

Mas por que a preocupação? A versão atual (1.3) é considerada estável e receberá correções, pequenas melhorias e atualizações de segurança por um longo tempo por parte da equipe do Google, até porque eles possuem centenas de aplicações escritas em AngularJS. Além disso, ele é um dos projetos mais ativos do GitHub, com mais de 1000 contribuidores, por isso irá continuar recebendo pull requests para agregar funcionalidades novas.

Os exemplos demonstrados nas listagens, assim como a aplicação completa estão disponíveis no repositório da aplicação de exemplo. Além disso, algumas informações extras como a lista de bibliotecas de terceiros utilizadas com seus respectivos links estão descritas no arquivo README. Também é possível ver a aplicação em execução no browser acessando o endereço da demonstração publicado no GitHub (seção **Links**).

Considerando a arquitetura implementada é possível concluir que a arquitetura *client-side* utilizando o *framework* AngularJS é viável no ponto de vista da criação de aplicações robustas com requisitos arquiteturais fortes, podendo substituir a implementação tradicional do padrão MVC no servidor, reduzindo a complexidade da aplicação devido a separação das responsabilidades da camada de apresentação e da lógica de negócio.

A arquitetura proposta também demonstra uma série de benefícios decorrentes dos requisitos arquiteturais apresentados, pois atende aos requisitos de escalabilidade, interoperabilidade, usabilidade, manutenibilidade e experiência do usuário.

Além disso, o *framework* AngularJS provou-se produtivo, pois com poucas linhas de código foi possível escrever uma aplicação inteiramente em JavaScript. Outro fator importante é a independência do desenvolvimento, possibilitando o paralelismo e otimizando a divisão de tarefas e a alocação dos recursos de um projeto de software.

Autor

Leonardo Zanivan
leonardo.zanivan@gmail.com

Especialista em arquitetura de software com experiência na plataforma Java EE e membro JCP. Atualmente trabalha como arquiteto de software na Trier Sistemas. É apaixonado por novas tecnologias, programação e pesquisa. No seu tempo livre também palestra em eventos para desenvolvedores e ministra cursos de programação. Já palestrou em eventos como o JavaOne e The Developers Conference.

Links:

Página oficial do projeto Bootstrap
http://getbootstrap.com/

Página de download do Brackets
http://brackets.io/

Página oficial do Grunt JS
http://gruntjs.com/

Página no Github do projeto Karma
http://karma-runner.github.io/

Página oficial do AngularJS
http://angularjs.org

Página no Github do projeto Protactor
http://angular.github.io/protractor/

Página no Github do projeto do artigo
http://panga.github.io/arquitetura-angularjs/

Trabalhando com Design Responsivo em HTML5

Conheça todas as etapas que compõem a criação de um site responsivo

Em um passado recente, os sites eram desenvolvidos visando atender a um público oriundo, em sua grande maioria, de computadores. Os computadores eram a grande maioria. A variação existente entre os modelos de monitores utilizados era muito pouca, o que tornava possível a definição de uma estratégia que visava definir uma largura máxima que não comprometesse a experiência dos usuários.

Essa estratégia foi bastante satisfatória até pouco tempo atrás, mas atualmente ela se tornou obsoleta devido ao fato de não conseguirmos mais definir uma largura máxima, em função da existência de uma grande variedade de dispositivos utilizados para navegar na internet. São celulares, tablets, netbooks, notebooks, computadores, videogames, televisores, etc., o que torna a tarefa de proporcionar uma boa experiência para todos esses tipos de dispositivos um grande desafio.

Como podemos desenvolver um site que atenda a cada um desses milhares de dispositivos? Uma resposta pode ser a criação de versões especificas e independentes voltadas a atender cada tipo utilizado separadamente. Porém, a evolução constante e a criação de novos dispositivos a todo momento tornam essa tarefa inviável tanto pelas constantes mudanças quanto pelo altíssimo custo de criação/manutenção.

Buscando uma solução para isso, **Ethan Marcotte** publicou no site "A List Apart" um artigo com o nome de "Responsive Web Design". Neste artigo, entre outras coisas, Ethan diz que devemos aceitar que diferentemente do design impresso (restrito a limitação física de uma página ou superfície), no design para internet essas restrições físicas não existem. Ademais, também não devemos nos sustentar em soluções que restrinjam a internet, pelo contrário, devemos projetar soluções flexíveis e adaptativas. Esse artigo deu origem a uma nova forma de se enxergar o design e o desenvolvimento de sites, resultando no conceito hoje conhecido como: **design responsivo**.

O design responsivo visa solucionar esse problema apresentando um layout que seja adaptável aos mais diversos tipos de dispositivos existentes hoje no mercado. Essa adaptação acontece através da detecção de algumas características presentes nos mesmos que possibilitam a criação de regras especificas para cada um, possibilitando, assim, a manipulação do layout e oferecendo uma navegação mais simples, intuitiva e contável aos seus usuários de acordo com o dispositivo utilizado.

Dentre as características essenciais a um site responsivo, destacam-se:

- Adaptar o layout da página de acordo com a resolução em que está sendo visualizada;
- O layout deve ser fluido e não deve fazer uso de medidas fixas, possibilitando a adaptação natural ao dispositivo em questão;

Fique por dentro

Esse artigo tem o objetivo de dar ao leitor uma compreensão geral e completa sobre as técnicas e conceitos mais importantes para o planejamento e desenvolvimento de sites responsivos. Através da criação de um exemplo de interface e menu de navegação responsivos, nós abordaremos desde os conceitos de desenvolvimento móvel, mobile first, as meta tags que o HTML5 usa para incutir responsividade, até um comparativo com a forma como esse tipo de recurso era implementado antigamente. Veremos ainda também o que são media types, breakpoints e como eles se encaixam no ciclo de vida da criação de projetos totalmente responsivos.

• Simplificar elementos da tela para dispositivos móveis, onde o usuário normalmente tem menos tempo e menos atenção durante a navegação;
• Redimensionar as imagens e vídeos para que não sobrecarreguem a transferência de dados e também para que se adaptem ao dispositivo garantindo que os mesmos se apresentem de forma nítida, sem cortes e que não façam uso da barra de rolagem para serem visualizados;
• Ocultar ou remover elementos desnecessários nos dispositivos menores;
• Adaptar o tamanho de botões, links e menus para interfaces *touch* onde o ponteiro do mouse é substituído pelo dedo do usuário.

Nota

Tornar um site que não é responsivo em um site responsivo não significa somente diminuir ou aumentar o tamanho dos elementos para que se encaixem nos mais variados tamanhos de tela, pois isso comprometeria toda a experiência do usuário.

Mobile First

O *Mobile First* é uma estratégia de desenvolvimento que diz que todo o planejamento de um projeto web deve iniciar pelos dispositivos móveis e somente depois gradualmente para os outros dispositivos, até chegar nos notebooks e desktops.

Em um passado recente, a prática de planejar e desenvolver projetos web voltados para dispositivos como computadores era bastante comum, até por questões históricas, para atender a demanda dos computadores, e onde se concentrava a web até então.

Começar pelos dispositivos móveis permite que a atividade de priorização do conteúdo seja feita de forma mais intuitiva, uma vez que em um dispositivo móvel somente as principais informações e funcionalidades de um projeto web serão disponibilizadas por conta do pouco espaço disponível e também pelas diferentes condições de uso e necessidades.

O processo de criação dos layouts também é beneficiado pelo uso da estratégia. Primeiramente serão produzidas versões mais simples voltadas para os dispositivos móveis, estas versões irão conter somente as informações principais, pois não há espaço disponível o suficiente para exibir tudo que teríamos na versão desktop. Outrossim, se isso for um pré-requisito por parte do cliente, então uma estratégia de design, estruturação e usabilidade deverá ser iniciada visando reduzir as dificuldades por trás da mesma. Posteriormente serão produzidas versões mais complexas voltadas para outros dispositivos, estas versões poderão conter mais informações de acordo com o espaço disponível proveniente de cada um.

Podemos dizer que para os dispositivos móveis é essencial que o layout seja objetivo e focado totalmente no conteúdo.

A Necessidade de Adaptatividade

Como a web em landscape (modo paisagem de visualização) torna-se cada vez mais complexa, está se tornando extremamente importante oferecer experiências web sólidas para um crescente número de contextos. Felizmente, o web design responsivo dá aos criadores web algumas ferramentas para fazer layouts que respondam a qualquer tamanho de tela. Desde grades fluídicas, imagens flexíveis até consultas de mídia para obter bons layouts, independentemente do tamanho de dimensões da tela do dispositivo.

Nota

Segundo a 9ª pesquisa TIC Domicílios divulgada pelo CETIC.br (Centro Regional de Estudos para o Desenvolvimento da Sociedade da Informação), somente em 2013 o número de brasileiros que acessaram a internet por meio de celular ou tablet atingiu 52,5 milhões, o que corresponde a 31% da população do país. O percentual mais que dobrou nos últimos dois anos, quando em 2011 esse número era apenas de 15%.

No entanto, o contexto móvel é muito mais do que apenas o tamanho da tela. Os nossos dispositivos móveis estão conosco onde quer que vamos, desbloqueando novos casos de uso todos os dias. Porque temos constantemente nossos dispositivos móveis com a gente, a conectividade pode ser todo o tabuleiro, variando de sinais wi-fi fortes no sofá para 3G ou EDGE quando fora de casa. Além disso, telas de toque abrem novas oportunidades de interagir diretamente com o conteúdo e a ergonomia móvel leva a diferentes considerações ao projetar layout e funcionalidade.

A fim de criar um site que é realmente concebido para o contexto móvel e não apenas para pequenas telas, queremos garantir que vamos enfrentar os muitos desafios do desenvolvimento móvel upfront. As restrições do contexto móvel nos forçam a focar em qual conteúdo é essencial e como apresentar esse conteúdo o mais rápido possível.

Progressive Enhancement x Graceful Degradation

A internet é um ambiente extremante dinâmico, devido às constantes evoluções das tecnologias que a envolvem. Um navegador que hoje é o mais moderno do mercado pode se tornar obsoleto amanhã com a chegada de uma nova tecnologia, caso essa não seja devidamente suportada. E o mesmo pode ocorrer a um site que seja considerado moderno da mesma forma. Por essa razão, abordagens como SEO (*Search Engine Optimization*), cache e *traffic tracking* (onde se foca em analisar de onde os tráficos de acesso vêm e quais conteúdos num site são preferidos pelos usuários), ganham cada vez mais popularidade em empresas que conseguem enxergar essa realidade.

Como fazer para utilizar o que há de mais moderno em termos de desenvolvimento web sem consequentemente perder os usuários de navegadores mais antigos que não suportam certas tecnologias? Para resolver esse dilema, podemos utilizar os conceitos do **Progressive Enhancement** (Melhoria Progressiva) e do **Graceful Degradation** (Degradação Graciosa). Eles constituem dois conceitos diferentes, mas que possuem em comum um mesmo objetivo: suportar o maior número possível

de usuários oferecendo a mesma experiência ou funcionalidade independentemente do dispositivo, conexão ou navegador utilizado.

O Graceful Degradation segue a linha de raciocínio na qual devemos utilizar no desenvolvimento o que há de melhor e mais moderno disponível em termos de tecnologia, visando primeiramente os navegadores mais recentes que suportam as mesmas e para os navegadores mais antigos que não as suportam, realizando adaptações e viabilizando a navegação de modo que a experiência do usuário não seja comprometida.

Já o Progressive Enhancement segue uma linha de raciocínio oposta, na qual devemos utilizar para o desenvolvimento recursos que possam ser suportados por todos os tipos de navegadores, inclusive os mais antigos, realizando melhorias progressivamente para os navegadores mais novos que suportem tecnologias mais modernas.

Podemos dizer que entre os dois conceitos não existe um melhor a ser escolhido quanto a forma de oferecer suporte aos navegadores, pois ambos têm o mesmo objetivo. Entretanto, uma vez que estamos seguindo uma estratégia Mobile First é aconselhável ir de encontro ao segundo, pois desta forma conseguimos apresentar uma versão simplificada, que utiliza um menor número de recursos tecnológicos para um smartphone, por exemplo, que geralmente possui limitações quanto a processamento e conexão; e apresentar uma versão mais completa, que utiliza um maior número de recursos tecnológicos para um computador ou notebook, que possui maior poder de processamento e melhor conexão.

Meta Tag Viewport

O termo **viewport** corresponde à área disponível para exibição de conteúdo que cada dispositivo possui.

O valor do viewport varia de acordo com o dispositivo. Por exemplo, em um computador, o viewport corresponde a área (altura e largura) em pixels ocupada pelo navegador. Portanto, quando o navegador é redimensionado para uma tela menor, automaticamente o seu viewport também se torna menor. A mesma lógica acontece quando o navegador é redimensionado para uma dimensão maior. Já em dispositivos móveis, não existe a possibilidade de redimensionar o navegador, pois as aplicações ocupam 100% da área disponível, ou seja, 100% do viewport.

Os sites não projetados e desenvolvidos com foco em dispositivos móveis, quando são acessados a partir dos mesmos, tem o zoom ajustado automaticamente fazendo com eles sejam encaixados de acordo com a largura do dispositivo. Isso ocorre porque nos dispositivos móveis a largura do viewport foi padronizada para 980 pixels na maioria das fabricantes.

Esse valor relacionado ao viewport foi adicionado pelo fato de que quando a navegação ocorre a partir de um dispositivo móvel (de viewport reduzido) seria exibida somente uma parte do conteúdo, o que obrigaria o usuário a navegar fazendo o uso das barras de rolagem, e prejudicando, consequentemente, a experiência de navegação, ao passo que implementando desta forma o usuário poderá usar manualmente de acordo com a necessidade.

Consequentemente, essa característica causa diversos problemas em sites já projetados para o mundo mobile. Isso porque o dispositivo irá encolher o site para exibi-lo de acordo com a largura de 980 pixels, o que impossibilita o funcionamento das regras responsivas, cujo problema a Meta Tag Viewport visa solucionar a partir de agora.

A meta tag viewport possui um formato semelhante ao de outras tags meta, conforme podemos observar a seguir:

```
<meta name="viewport" content="">
```

Na declaração do content é possível especificar uma diversidade de parâmetros, tais como:

- **width:** define a largura do viewport.
- **height:** define a altura do viewport.
- **initial-scale:** define a escala inicial (zoom) do viewport.

Para que possamos assegurar o funcionamento das regras responsivas em todos os dispositivos, devemos declarar que a largura do viewport será igual à mesma largura do dispositivo e que a escala inicial (zoom) é 1, conforme demonstrado a seguir:

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

Media Type

O atributo Media Type é o responsável por designar quais regras CSS serão interpretadas por um determinado tipo de dispositivo, oferecendo a melhor apresentação possível do conteúdo sem a necessidade de alteração do mesmo.

Através desse atributo é possível apresentar o site de maneira diferente aplicando o estilo mais apropriado de acordo com o tipo do dispositivo utilizado. Por exemplo, um determinado conteúdo apresentado de uma forma para um computador e de uma outra forma completamente diferente para uma impressora, e assim sucessivamente.

Dentre os diversos tipos de Media Type, alguns se destacam e são importantes ao conhecimento do leitor:

- **All:** Se refere a todos os tipos de dispositivos.
- **Braille:** Se refere aos dispositivos táteis.
- **Embossed:** Se refere aos dispositivos que imprimem em braille.
- **Handheld:** Se refere aos dispositivos de mão, normalmente com telas pequenas de baixa resolução e largura de banda limitada.
- **Print:** Se refere aos dispositivos de impressão.
- **Projection:** Se refere a apresentações do tipo slides.
- **Screen:** Se refere a monitores ou dispositivos com telas coloridas e resolução adequada.
- **Speech:** Se refere a sintetizadores de voz.
- **TTY:** Se refere a terminais, teletypes e dispositivos portáteis com display limitado.

• **TV:** Se refere a televisores ou dispositivos com baixa resolução, quantidade de cores e scroll limitado.

Nós podemos incluir o atributo Media Type em um site de diversas formas. Para sintetizar, observe o conteúdo da **Tabela 1** e as situações com os respectivos exemplos para cada uma.

Fazendo um rápido comparativo de usabilidade em relação a estes termos, é possível observar que com o passar dos anos o uso de Media Types se tornou obsoleto devido à modernização e à grande quantidade de dispositivos com características totalmente diferentes pertencentes a um mesmo tipo. Com isso, somente os tipos **screen** e **print** ainda são utilizados em grande escala.

Por exemplo, os smartphones não pertencem ao tipo **handheld**. Embora sejam considerados dispositivos de mão, eles possuem monitores com telas coloridas e de alta resolução e por isso serão interpretados como sendo do tipo **screen**.

Media Queries

São simples expressões que foram adicionadas na versão 3 do CSS e, juntamente com o Media Type, formam a base de todo layout responsivo. Na prática, funcionam como uma extensão do atributo Media Type, adicionando e combinando características com o uso de operadores lógicos, permitindo que sejam criadas regras mais especificas e eficientes e garantindo um controle maior para a apresentação do conteúdo em diferentes tipos de screens.

Essas expressões possuem, como resultado, um valor booleano, ou seja, são necessariamente verdadeiras ou falsas. As regras associadas a essas expressões somente serão aplicadas quando o resultado for verdadeiro.

As Media Queries adicionam características detectáveis, que são denominadas como Media Features. Cada uma delas possui funções e valores específicos, tais como os definidos na **Tabela 2**.

Para realizar a combinação entre os Media Types e as Media Features que formam as regras condicionais e resultam nas

Descrição	Exemplo
Utilizando o atributo media na chamada de um arquivo CSS em um arquivo HTML.	<link rel="stylesheet" href="arquivo.css" type="text/css" media="print">
Utilizando o atributo media na chamada de um arquivo CSS em um arquivo XML.	<? xml-stylesheet rel="stylesheet" href="arquivo.css" media="all" ?>
Utilizando a tag @import dentro da tag style em um arquivo HTML.	<style type="text/css" media="screen"> @import "arquivo.css"; </style>
Utilizando a tag @import em um arquivo CSS.	@import url("estilo.css") print;

Utilizando a tag @media em um arquivo CSS.	@media screen {
	body {
	background-color: #F00;
	}
	}

Tabela 1. Lista de situações e exemplos de inclusão dos atributos media type.

Atributo	Descrição
aspect-ratio	Se refere à proporção entre os valores de largura e altura. Os valores são compostos pela divisão da largura/valor da altura. Não aceita os prefixos min/max.
device-aspect-ratio	Se refere à proporção entre os valores de largura e altura do dispositivo. Os valores são compostos por valor da largura/valor da altura. Não aceita os prefixos min/max.
Color	Se refere ao número de bits por cor. Os valores são numéricos. Aceita os prefixos min/max.
color-index	Se refere ao número de entradas na tabela de pesquisa de cores do dispositivo de saída. Os valores são numéricos. Aceita os prefixos min/max.
Height	Se refere à altura da janela do navegador. Os valores são medidas de comprimento. Aceita os prefixos min/max.
device-height	Se refere à altura da mídia. Os valores são medidas de comprimento. Aceita os prefixos min/max.
Width	Se refere à largura da janela do navegador. Os valores são medidas de comprimento. Aceita os prefixos min/max.
device-width	Se refere à altura da mídia. Os valores são medidas de comprimento. Aceita os prefixos min/max.
Grid	Se refere ao tipo de dispositivo, se é orientado a bitmaps ou grids. Os valores são 1 para dispositivos orientados a grid e 0 para dispositivos orientados a bitmap. Não aceita os prefixos min/max.
Monochrome	Se refere ao número de bits por pixel em um buffer de quadros monocromáticos. Os valores são numéricos. Aceita os prefixos min/max.
Orientation	Se refere à orientação da mídia. Os valores são portrait para vertical e landscape para horizontal. Não aceita os prefixos min/max.
Resolution	Se refere à resolução, densidade por pixel, utilizada pelo dispositivo. Os valores são em DPI ou DCM. Aceita os prefixos min/max.
Scan	Se refere ao tipo de formação de imagens para televisores. Os valores são progressive ou interlace. Não aceita os prefixos min/max.

Tabela 2. Lista de atributos aceitos pelas media queries.

Media Queries, são utilizados os **operadores**. Dentre eles, podemos destacar:

• **not:** É utilizado quando se deseja que o resultado de uma determinada expressão seja o oposto ao real. Veja na **Listagem 1** um exemplo do seu uso.

• **only:** É utilizado quando se deseja prevenir que navegadores antigos que não suportam Media Features tentem processar a expressão. Na **Listagem 2** podemos ver um exemplo fiel disso.

• **and:** É utilizado em todas as Media Queries, tendo como função primária ser o elo entre o Media Type e a Media Feature. É também responsável pelas expressões múltiplas, isto é, quando usamos mais de uma Media Feature para compor a expressão.

• **",":** É utilizado para juntar duas ou mais expressões diferentes que deverão executar um mesmo conjunto de regras. Funciona como um "ou" condicional na lógica de programação.

Listagem 1. Exemplo de utilização do operador lógico "not".

```
01 @media not print and (min-width: 768px) {
02    body {
03       background-color: #FF0000;
04    }
05 }
```

Neste exemplo, todos os dispositivos que não forem do tipo print com largura mínima de 768 pixels terão a cor de fundo do corpo do documento vermelha. Sem o uso do operador not, o resultado seria o oposto, e a cor de fundo do corpo do documento seria vermelha para todos os dispositivos do tipo print com a largura mínima de 768 pixels.

Listagem 2. Exemplo de utilização do operador lógico "only".

```
01 @media only screen and (max-width: 320px) {
02    h1 {
03       text-decoration: underline;
04    }
05 }
```

Já neste exemplo, em dispositivos do tipo screen que tenham largura até 320 pixels, os elementos **h1** terão seus textos exibidos com *underline*. Sem o operador only, a expressão iria ser executada por todos os dispositivos, e no caso de dispositivos que não suportam Media Features, somente o Media Type seria reconhecido e as mesmas seriam ignoradas. O que resultaria na execução da regra referente ao h1 em dispositivos nos quais esta não deveria ser executada.

Na **Listagem 3** temos uma expressão múltipla relacionada aos dispositivos do tipo screen, onde duas Media Features são utilizadas dentro de uma mesma expressão. Uma Media Feature corresponde à largura mínima de 320 pixels e outra à largura máxima de 480 pixels. Portanto, caso positiva, o elemento header terá a largura de 100% em dispositivos do tipo screen com a largura entre 320 e 480 pixels.

Listagem 3. Exemplo de utilização do operador lógico "and".

```
01 @media only screen and (min-width: 320px) and (max-width: 480px) {
02    header {
03       width: 100%;
04    }
05 }
```

Na **Listagem 4** há duas expressões diferentes, uma para dispositivos do tipo handheld com largura máxima de 500 pixels e outra para dispositivos do tipo screen com largura máxima de 620 pixels. Ambas as expressões são totalmente independentes uma da outra, mas será executada, caso positivas, a mesma regra CSS, que consiste justamente em deixar negrito todo o texto existente dentro de um elemento de parágrafo p.

Listagem 4. Exemplo de utilização do operador lógico ",".

```
01 @media only handheld and (max-width: 500px), only screen (max-width: 620px) {
02    p {
03       font-weight: bold;
04    }
05 }
```

Breakpoints

Os **breakpoints** são delimitadores das regras CSS para atenderem diferentes especificações, que são criados pelo uso de Media Queries. Tratam-se também de uma forma de organizar o uso das Media Queries, visando a criação de blocos previamente definidos de acordo com os dispositivos que serão suportados, e onde serão inseridas as respectivas regras CSS. Isso evitará, desta forma, a utilização de diversas regras pontuais para a resolução de casos específicos.

Não existe um número ou um grupo exato de breakpoints a serem utilizados. A definição dos breakpoints está diretamente ligada a características especificas do site em questão, como o conteúdo e os dispositivos que serão suportados.

Portanto, um novo breakpoint deve ser criado sempre que a apresentação do conteúdo em determinado dispositivo não oferecer a melhor experiência possível ou simplesmente não for satisfatória.

Observe na **Listagem 5** um exemplo bastante simples do uso de breakpoints.

Veja que a implementação desse tipo de estrutura é muito semelhante a estruturas já existentes em outras ferramentas e linguagens de programação, como o Visual Studio, por exemplo. Além de trazer um ar de organização ao conteúdo dos arquivos CSS, ele também ajuda na hora de buscar por determinadas regras ou selecionar um conjunto delas para alteração.

Colocando em prática

Agora que já conhecemos em detalhes o que são os termos Media Types, Media Features e Media Queries, é hora de colocar em prática o conhecimento adquirido. Neste exemplo, vamos criar um site responsivo que deverá apresentar quatro versões de layout diferentes. A saber:

• **Primeira versão:** Se refere aos dispositivos considerados pequenos, com largura até 480 pixels;
• **Segunda versão:** Se refere aos dispositivos considerados médios, com largura maior que 480 pixels e até 1024 pixels;
• **Terceira versão:** Se refere aos dispositivos considerados grandes, com largura maior que 1024 pixels;
• **Quarta versão:** Se refere aos dispositivos de impressão.

Listagem 5. Exemplo de uso dos breakpoints.

```
/* INICIO REGRAS PARA TODOS OS DISPOSITIVOS */
 INSIRA REGRAS CSS AQUI!
/* FIM REGRAS PARA TODOS OS DISPOSITIVOS*/

/* INICIO REGRAS PARA DISPOSITIVOS DE IMPRESSÃO */
 @media print {
   INSIRA REGRAS CSS AQUI!
 }
/* FIM REGRAS PARA DISPOSITIVOS DE IMPRESSÃO */

/* INICIO REGRAS PARA DISPOSITIVOS COM LARGURA MÍNIMA DE 320 PIXELS.
   POR EXEMPLO: SMARTHPHONES */
 @media screen and (min-width: 320px) {
   INSIRA REGRAS CSS AQUI!
 }
/* FIM REGRAS PARA DISPOSITIVOS COM LARGURA MÍNIMA DE 320 PIXELS.
   POR EXEMPLO: SMARTHPHONES */

/* INICIO REGRAS PARA DISPOSITIVOS COM LARGURA MÍNIMA DE 768 PIXELS.
   POR EXEMPLO: TABLETS */
 @media screen and (min-width: 768px) {
   INSIRA REGRAS CSS AQUI!
 }
/* FIM REGRAS PARA DISPOSITIVOS COM LARGURA MÍNIMA DE 768 PIXELS.
   POR EXEMPLO: TABLETS */

/* INICIO REGRAS PARA DISPOSITIVOS COM LARGURA MÍNIMA DE 1024 PIXELS.
   POR EXEMPLO: COMPUTADORES DESKTOP E NOTEBOOKS */
 @media screen and (min-width: 1024px) {
   INSIRA REGRAS CSS AQUI!
 }
/* FIM REGRAS PARA DISPOSITIVOS COM LARGURA MÍNIMA DE 1024 PIXELS.
   POR EXEMPLO: COMPUTADORES DESKTOP E NOTEBOOKS */
```

Todas as quatro versões possuirão características especificas. Primeiramente, crie um documento HTML e nomeie-o como "index.html", e adicione o conteúdo presente na **Listagem 6**, que se refere a estrutura inicial da página, contendo apenas algumas declarações das tags que vimos anteriormente, bem como um título e uma tag h1 no corpo da página.

Repare que estamos usando as configurações referentes à nova especificação da HTML5 para melhor se adequar a realidade do artigo.

Após a criação do arquivo, já com a estrutura inicial, vamos incluir o conteúdo especifico referente à cada versão do layout.

Na **Listagem 7** está presente o conteúdo especifico referente à primeira versão, que se refere aos dispositivos considerados pequenos, com largura até 480 pixels. Faça a inclusão do conteúdo dentro da tag <body> logo após o título da página representando pela tag h1.

Listagem 6. Marcação HTML inicial.

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>

    <!-- INICIO META TAGS -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <!-- FIM META TAGS -->

    <!-- INICIO TITULO DO DOCUMENTO -->
    <title>Teste Layout Responsivo</title>
    <!-- FIM TITULO DO DOCUMENTO -->

  </head>
  <body>

    <!-- INICIO TITULO DA PÁGINA -->
    <h1 class="titulo">Teste Layout Responsivo</h1>
    <!-- FIM TITULO DA PÁGINA -->

  </body>
</html>
```

Listagem 7. Conteúdo especifico da primeira versão.

```
<!-- INICIO CONTEUDO PRIMEIRA VERSAO -->
    <section class="conteudo-dispositivo pequeno">
      <h2 class="tipo">Dispositivo Pequeno</h2>
      <p class="descricao">
        Esse texto será exibido somente em dispositivos com largura até
        480 pixels.
      </p>
    </section>
<!-- FIM CONTEUDO PRIMEIRA VERSAO -->
```

No conteúdo presente na **Listagem 8** temos o código referente à segunda versão, que se refere aos dispositivos considerados médios, com largura maior que 480 pixels e até 1024 pixels. Insira este conteúdo no arquivo index.hml logo após o conteúdo da primeira versão da listagem anterior.

Listagem 8. Conteúdo especifico da segunda versão.

```
<!-- INICIO CONTEUDO SEGUNDA VERSAO -->
 <section class="conteudo-dispositivo medio">
 <h2 class="tipo">Dispositivo Médio</h2>
  <p class="descricao">
   Esse texto será exibido somente em dispositivos com largura maior
   que 480 pixels e até 1024 pixels.
  </p>
 </section>
<!-- FIM CONTEUDO SEGUNDA VERSAO -->
```

Em seguida, temos a **Listagem 9** que apresenta o conteúdo específico referente à terceira versão, que se dirige aos dispositivos considerados grandes, com largura maior que 1024 pixels. Insira este conteúdo no arquivo index.hml logo após o conteúdo referente à segunda versão. Note que os contextos se assemelham em relação ao texto e tags criados nas mesmas seções (section). Logo, finalizaremos com a explanação em tempo real de execução, posteriormente.

Finalmente, no conteúdo presente na **Listagem 10** temos o código referente à quarta versão citada, que se refere aos dispositivos de

impressão. Insira este conteúdo no arquivo index.hml logo após o conteúdo referente à terceira versão.

A estrutura dos conteúdos específicos apresentados respectivamente nas listagens anteriores é a mesma, o que muda é apenas o conteúdo de cada uma delas. Dentro de cada tag <section> existem dois elementos, o primeiro é referente à tag <h2> que tem a função de descrever qual versão do layout está sendo apresentada e o outro é a tag <p> que possui um texto descritivo específico para cada versão.

Listagem 9. Conteúdo especifico da terceira versão.

```
01 <!-- INICIO CONTEUDO TERCEIRA VERSAO -->
02 <section class="conteudo-dispositivo grande">
03      <h2 class="tipo">Dispositivo Grande</h2>
04      <p class="descricao">
05          Esse texto será exibido somente em dispositivos com largura maior que
            1024 pixels.
06      </p>
07 </section>
08 <!-- FIM CONTEUDO TERCEIRA VERSAO -->
```

Listagem 10. Conteúdo especifico da versão para dispositivos de impressão.

```
01 <!-- INICIO CONTEUDO QUARTA VERSAO -->
02 <section class="conteudo-dispositivo impressao">
03      <h2 class="tipo">Dispositivo de Impressão</h2>
04      <p class="descricao">
05          Esse texto será exibido somente em dispositivos de impressão.
06      </p>
07 </section>
08 <!-- FIM CONTEUDO QUARTA VERSAO -->
```

Agora com o arquivo index.html devidamente preenchido, vamos criar o arquivo CSS a ser utilizado. Denomine o arquivo como "stylesheet.css" e, em seguida, adicione o trecho de código CSS presente na **Listagem 11**.

Listagem 11. Regras CSS comuns a todos os dispositivos.

```
01 /* INICIO REGRAS CSS COMUNS À TODOS OS DISPOSITIVOS */
02 .titulo {
03    margin: 20px 0 80px;
04    text-align: center;
05 }
06
07 .conteudo-dispositivo {
08    display: none;
09    text-align: center;
10 }
11 /* FIM REGRAS CSS COMUNS À TODOS OS DISPOSITIVOS */
```

Este trecho possui regras comuns que serão aplicadas a todos os dispositivos independentemente da versão apresentada, tais como definições gerais de espaçamento, margens, alinhamento de texto e exibição dos componentes na página.

Observe também que por se tratar de regras CSS, as mesmas serão aplicadas em cascata, obedecendo ao modelo de herança do CSS3.

Na **Listagem 12** temos as Media Queries responsáveis pela detecção e aplicação das regras CSS referentes às versões de um a quatro. Insira esse conteúdo no arquivo stylesheet

Listagem 12. Regras CSS referentes à primeira, segunda, terceira e quarta versões.

```
01 /* INICIO REGRAS CSS REFERENTES À PRIMEIRA VERSÃO */
02 @media only screen and (max-width: 480px) {
03
04   body {
05      background-color: #D46A6A;
06      color: #FFF;
07   }
08
09   .conteudo-dispositivo.pequeno {
10      display: block;
11   }
12
13 }
14 /* INICIO REGRAS CSS REFERENTES À SEGUNDA VERSÃO */
15 @media only screen and (min-width: 481px) and (max-width: 1024px) {
16
17   body {
18      background-color: #758AA8;
19      color: #FFF;
20   }
21
22   .conteudo-dispositivo.medio {
23      display: block;
24   }
25
26 }
27 /* INICIO REGRAS CSS REFERENTES À TERCEIRA VERSÃO */
28 @media only screen and (min-width: 1025px) {
29
30   body {
31      background-color: #D4CE6A;
32      color: #000;
33   }
34
35   .conteudo-dispositivo.grande {
36      display: block;
37   }
38
39 }
40 /* INICIO REGRAS CSS REFERENTES À QUARTA VERSÃO */
41 @media print {
42
43   .titulo {
44      margin: 0 0 50px;
45      text-align: left;
46   }
47
48   .conteudo-dispositivo {
49      text-align: left;
50   }
51
52   body {
53      background-color: #FFF;
54      color: #000;
55   }
56
57   .conteudo-dispositivo.impressao {
58      display: block;
59   }
60
61 }
```

.css logo após as regras CSS comuns a todos os dispositivos. As mesmas regras, por sua vez, se responsabilizarão somente por atribuir características mais simples como cor, cor de fundo e visualização dos componentes, para facilitar o entendimento de uma forma geral.

Podemos notar que as regras referentes à cada versão são bastante semelhantes. Todas são compostas por uma Media Query e suas respectivas regras que correspondem à forma como o conteúdo será apresentado em uma das versões definidas. Ela se deve, em grande parte, à semelhança existente também na estrutura HTML implementada.

Na versão especifica para dispositivos de impressão, temos uma situação um pouco diferente das demais. Pois além de possuir uma Media Query responsável pela detecção e aplicação das regras CSS aos dispositivos de impressão, essa versão possui também algumas regras com a função de anular as regras CSS comuns a todos os dispositivos, apresentadas anteriormente. Isso se deve ao fato de que por se tratar de um dispositivo de impressão, as regras referentes à distribuição e cor dos elementos na página deixaram de fazer sentido.

Agora o arquivo stylesheet.css está completo e precisamos incluir no arquivo index.html a chamada para o mesmo, possibilitando assim que possamos visualizar na prática o funcionamento das Media Queries.

Verifique na **Listagem 13** o conteúdo para efetuar a chamada ao arquivo stylesheet.css que deverá ser inserido dentro da tag <head> logo após a tag <title>.

Listagem 13. Chamada para o arquivo stylesheet.css.

```
01 <!-- INICIO ARQUIVOS CSS -->
02 <link type="text/css" rel="stylesheet" href="stylesheet.css">
03 <!-- FIM ARQUIVOS CSS -->
```

Agora podemos visualizar o resultado da execução do arquivo HTML. Na **Figura 1** apresentamos o resultado que será exibido em dispositivos com largura até 480 pixels, referenciados pela primeira versão.

Na **Figura 2** apresentamos o resultado que será exibido em dispositivos com largura maior que 480 pixels e até 1024 pixels, referenciados pela segunda versão.

Na **Figura 3** apresentamos o resultado que será exibido em dispositivos com largura maior que 1024 pixels, referenciados pela terceira versão.

E finalmente na **Figura 4** temos o resultado que será exibido em dispositivos de impressão, referenciados pela quarta versão.

Criando uma navegação responsiva

Neste segundo exemplo estaremos tratando de explorar como os conceitos que aprendemos até agora podem ser aplicados para construir um menu de navegação responsivo, este que representa um dos principais componentes dos websites hoje em dia, presente na maioria deles quando se trata de sites responsivos.

Figura 1. Resultado apresentado pela primeira versão

Figura 2. Resultado apresentado pela segunda versão

Figura 3. Resultado apresentado pela terceira versão

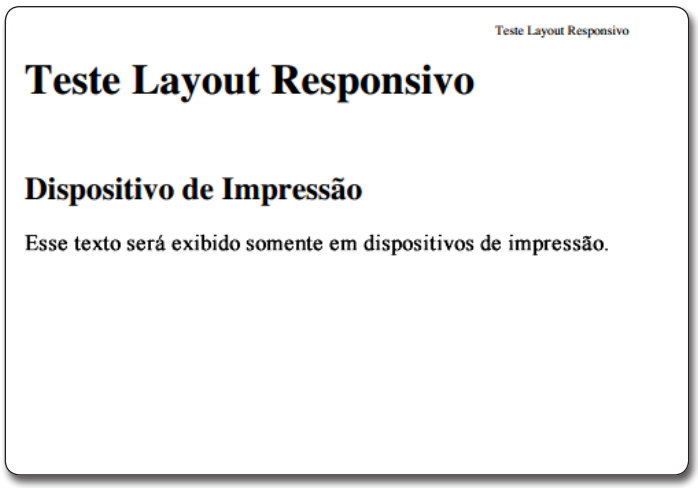

Figura 4. Resultado apresentado pela quarta versão.

Para isso, vamos começar criando uma nova página html no mesmo diretório da anterior, e nomeá-la "menu.html". Após isso, se certifique de criá-la com todo o conteúdo inicial de tags e viewport que falamos até então, como mostra a **Listagem 14**.

Listagem 14. Página inicial do menu.html

```
01<!DOCTYPE html>
02<html>
03<head>
04   <title>Exemplo menu responsivo</title>
05   <meta name="viewport" content="width=device-width, initial-scale=1,
       maximum-scale=1">
06</head>
07<body>
08   <nav class="menu-nav">
09      <ul class="menu-nav">
10         <li><a href="#">Home</a></li>
11         <li><a href="#">Que Somos</a></li>
12         <li><a href="#">Clientes</a></li>
13         <li><a href="#">Missão</a></li>
14         <li><a href="#">Produtos</a></li>
15         <li><a href="#">Contato</a></li>
16      </ul>
17      <a href="#" id="menu">Menu</a>
18   </nav>
19</body>
20</html>
```

Note que esse tipo de estrutura é bem semelhante à forma como outros frameworks lidam para a criação de menus responsivos, como o Bootstrap e o jQuery. Como você pode ver, temos seis links do menu principal e acrescentamos um link depois deles. Este link extra será usado para puxar o menu de navegação quando ele estiver escondido em uma tela pequena.

Crie agora um arquivo style.css na mesma altura do arquivo HTML, ele será usado para conter as regras de CSS do nosso exemplo. Vamos começar adicionando algumas regras na forma como visualizaremos o conteúdo, bem como o tipo de zoom definido, como mostra a **Listagem 15**.

Listagem 15. Limpando os valores antes e depois do menu.

```
.menu-nav:before,
.menu-nav:after {
   content: " ";
   display: table;
}
.menu-nav:after {
   clear: both;
}
.menu-nav {
   *zoom: 1;
}
```

Em seguida, adicione também o conteúdo da **Listagem 16**. Nele, a tag **nav** que define a navegação terá o valor de 100% da largura total da janela do navegador, enquanto o **ul**, que contém nossos links do menu principal, terá 600px de largura. Então, vamos flutuar (float) os links do menu à esquerda, de modo que eles serão exibidos horizontalmente lado a lado. Lembre-se que um elemento flutuante também fará com que o seu elemento pai assim seja.

Listagem 16. Configurações iniciais de estilo para o menu.

```
body {
   background-color: #ccc123;
}
nav {
   height: 40px;
   width: 100%;
   background: #255461;
   font-size: 11pt;
   font-family: 'PT Sans', Arial, sans-serif;
   font-weight: bold;
   position: relative;
   border-bottom: 2px solid #283744;
}
nav ul {
   padding: 0;
   margin: 0 auto;
   width: 600px;
   height: 40px;
}
nav li {
   display: inline;
   float: left;
}
```

Note que, com a marcação HTML usada até agora, já adicionamos o clearfix no atributo de classe, tanto para o nav quanto para o ul, especificamente para esclarecer as coisas em torno de quando nós flutuamos os elementos dentro deles mesmos usando esse truque CSS. Então, vamos adicionar agora as regras da **Listagem 17** na folha de estilo.

Uma vez que temos seis links de menu e também configuramos o container para 600px, cada link do menu terá 100px de largura. Os links do menu serão separados por uma borda à direita de 1px de largura, exceto para o último. Lembre-se que a largura do menu será expandida para 1px fazendo com que o tamanho total seja de 101px, com a adição da borda. Então, aqui nós mudamos o

modelo box-sizing para boder-box, a fim de manter o menu com 100px. Finalmente, adicionamos as propriedades de alteração do modelo quando nos estados de :active e :hover.

Com tudo isso implementado, a visualização da página no browser será parecida com a demonstrada na **Figura 5**.

Figura 5. Visualização da tela de menu

Listagem 17. Regras de estilo para os elementos de link de menu.

```
nav a {
   color: #fff;
   display: inline-block;
   width: 100px;
   text-align: center;
   text-decoration: none;
   line-height: 40px;
   text-shadow: 1px 1px 0px #233144;
}
nav li a {
   border-right: 1px solid #576979;
   box-sizing:border-box;
   -moz-box-sizing:border-box;
   -webkit-box-sizing:border-box;
}
nav li:last-child a {
   border-right: 0;
}
nav a:hover, nav a:active {
   background-color: #8a99e3;
}
nav a#pull {
   display: none;
}
```

Uma vez que a nossa navegação é inicialmente fixa em 600px de largura, vamos primeiro definir os estilos de quando ela é vista em 600px ou em tamanho menor de tela, de modo que, em termos práticos, esse seja o nosso primeiro breakpoint.

Neste tamanho de tela, cada um dos dois links do menu será exibido lado a lado, de modo que a largura da ul aqui será de 100% da janela do navegador, enquanto os links do menu terão 50% da largura. Para isso adicione o conteúdo da **Listagem 18** ao arquivo de CSS.

Então, em seguida, nós também definimos como a navegação é exibida quando a tela ficar menor que 480px (de modo que este é o nosso segundo breakpoint).

Neste tamanho de tela, o link extra que nós adicionamos antes começará visível. Nós também daremos ao link um ícone de "Menu" no seu lado direito usando o pseudoelemento **:after**, enquanto os links do menu principal serão escondidos para economizar mais espaços verticais na tela.

Listagem 18. Código de redimensionamento do menu.

```
@media screen and (max-width: 600px) {
   nav {
      height: auto;
    }
    nav ul {
      width: 100%;
      display: block;
      height: auto;
    }
    nav li {
      width: 50%;
      float: left;
      position: relative;
    }
    nav li a {
      border-bottom: 1px solid #576979;
      border-right: 1px solid #576979;
   }
    nav a {
      text-align: left;
      width: 100%;
      text-indent: 25px;
    }
}

@media only screen and (max-width : 480px) {
   nav {
      border-bottom: 0;
   }
   nav ul {
      display: none;
      height: auto;
   }
   nav a#pull {
      display: block;
      background-color: #255461;
      width: 100%;
      position: relative;
   }
   nav a#pull:after {
      content:"";
      background: url('nav-icon.png') no-repeat;
      width: 30px;
      height: 30px;
      display: inline-block;
      position: absolute;
      right: 15px;
      top: 10px;
   }
}

/*Smartphone*/
@media only screen and (max-width : 320px) {
   nav li {
      display: block;
      float: none;
      width: 100%;
   }
   nav li a {
      border-bottom: 1px solid #576979;
   }
}
```

Por último, quando a tela ficar menor que 320px o menu será exibido verticalmente de cima para baixo. Agora você pode tentar redimensionar o browser e verá os resultados para cada exibição.

Neste ponto, o menu ainda estará escondido e só será visível quando for necessário, tocando ou clicando no link "Menu". Nós podemos conseguir esse efeito usando a função **slideToggle()** do jQuery. Para isso, crie uma nova tag script dentro da tag head do seu arquivo HTML e inclua a função representada pela **Listagem 19**.

Listagem 19. Função JavaScript para exibir o menu toggle.

```
<script>
  $(function() {
    var pull = $('#pull');
      menu = $('nav ul');
      menuHeight = menu.height();
      $(pull).on('click', function(e) {
      e.preventDefault();
      menu.slideToggle();
  });
      $(window).resize(function(){
        var w = $(window).width();
        if(w > 320 && menu.is(':hidden')) {
          menu.removeAttr('style');
        }
      });
  });
</script>
```

Não esqueça de adicionar as referências aos arquivos do jQuery, bem como do normalize.js que vieram junto do pacote de download dos fontes neste artigo.

Isso é basicamente o que precisamos para fazer esse exemplo funcionar. O resultado final de execução num smartphone pode ser visualizado na **Figura 6**.

Através desses exemplos podemos verificar que ao fazer uso das Media Queries é possível apresentar um único site com conteúdo totalmente diferente e de diversas formas, de acordo com o dispositivo utilizado, explorando devidamente suas características, a fim de melhorar a experiência do usuário sem a necessidade de alterar o conteúdo, e de forma extremamente simples.

Figura 6. Exibição do menu num smartphone.

Após a leitura deste artigo, o leitor terá adquirido conhecimento sobre técnicas e conceitos importantes e extremamente necessários para o planejamento correto do desenvolvimento de sites responsivos. Além dos exemplos práticos que conseguem maximizar o impacto no aprendizado como um todo.

Autor

Thiago Pereira Perez
Desenvolvedor front-end com grande experiência em desenvolvimento, manutenção e otimização de projetos web de todo porte. Atualmente atuando como coordenador de desenvolvimento front-end no bomnegócio.com.

Links:

Site com estatísticas sobre o uso de navegadores
http://gs.statcounter.com/

Site para testar layouts responsivos.
http://www.codeorama.com/responsive/

Site com diversos exemplos de sites responsivos.
http://mediaqueri.es/

Dicas para projetos de sites responsivos
http://css-tricks.com/notes-agency-starting-their-first-responsive-web-project/

Grunt.JS: Criando um sistema escolar com Grunt e CoffeeScript

Desenvolva uma aplicação para controle de alunos com automação de tarefas e templates JavaScript

Quando falamos de desenvolvimento de software em geral, requisitos como web design ou deixar a execução de certas atividades ser feita no navegador do cliente para tornar a aplicação mais performática, fazem com que as ferramentas e linguagens que trabalham no front-end sejam mais cobradas em tal atendimento. Para completar, não é difícil ver equipes que tenham termos como produtividade, código organizado e limpo, desenvolvimento ágil e múltiplas integrações e versionamentos, presentes na lista de obrigações de um projeto de software.

Várias tecnologias contribuíram e contribuem para isso, desde os pré-processadores de CSS, como o Sass e o Less, que ajudaram na dinamização do CSS antes feita de forma totalmente estática; até a criação de bibliotecas de compressão/minimização ou de testes unitários. Mas um assunto que ainda traz demasiadas repercussões nesse universo novo, principalmente por se tratar de algo quase que completamente atrelado ao universo *server side*, é a **automação de tarefas**. No mundo JavaScript, uma das bibliotecas de automação de tarefas mais usadas é o **Grunt**. Ele nasceu com o objetivo de aplicar na prática os conceitos de produtividade no desenvolvimento e integração de atividades junto ao JavaScript e às bibliotecas e frameworks que o seu projeto fizer uso.

Muitas empresas como Twitter, jQuery e Adobe fazem uso do Grunt em seus projetos padrão, ao passo que o mesmo consegue executar em conjunto com diversas tecnologias, tais como o Sass e Less (citados anteriormente), handlebars e o CoffeeScript.

O **CoffeeScript**, por sua vez, introduz um conceito parecido com o dos processadores de CSS, porém com "pré-compilação" de JavaScript. Através dele podemos, em uma linguagem abstraída e mais simplificada, compilar e gerar JavaScript. O conceito de sintaxe é muito próximo ao de linguagens como o Ruby, o que acabou atraindo a atenção de muita gente desde que foi lançada sua primeira versão.

Neste artigo iremos trabalhar essencialmente com estas duas tecnologias e tentaremos abordar o núcleo e os principais conceitos tanto do Grunt como do CoffeeScript, através da criação de uma aplicação simples e prática para gerenciamento de alu-

Fique por dentro

Este artigo tem como objetivo introduzir duas das tecnologias mais debatidas pela comunidade de desenvolvimento front-end do momento: a ferramenta de gerência de tarefas automatizadas para JavaScript, Grunt; e o framework de geração de código JavaScript, CoffeeScript. Através da criação de uma aplicação que controla uma lista simples de alunos, exploraremos os principais recursos de ambas as tecnologias, não só individualmente, mas também em conjunto com si mesmas, e com outras ferramentas tão importantes quanto, tais como jQuery, Bootstrap, Sass e Compass. Veremos também como gerar builds dinâmicas, criar threads de execução em hot deploy, e entender conceitos importantes de orientação a objetos e estrutura de dados atrelados a esse universo.

nos, tal como demonstrado na **Figura 1**. A mesma aplicação vai dispor de algumas listagens, com foco em entender como o CoffeeScript funciona para esse tipo de modelo programático, visualizando suas integrações junto ao Grunt. Mas antes, vamos entender melhor como funcionam ambas as tecnologias.

Grunt

Em uma definição curta e simples, o Grunt é uma ferramenta de linha de comando baseada em tarefas para projetos em JavaScript.

De uma forma mais detalhada, podemos dizer que quando trabalhamos com um projeto em JavaScript, existe uma série de coisas que devem ser feitas regularmente, tais como concatenar arquivos, rodar alguma ferramenta de detecção de erros e problemas no código (como o JSHint, por exemplo), executar testes (unitários, de integração) ou modificar seus scripts. Se você estiver copiando e colando o seu código JavaScript no site do JSHint (vide seção **Links**), provavelmente perceberá que há uma maneira melhor de fazer isso com o Grunt. Mesmo se estiver usando o objeto "cat" para concatenar arquivos ou uma linha de comando para comprimir os mesmos, seria bom ter um único conjunto unificado de comandos para todas aquelas tarefas extras, que você teve de fazer para cada projeto JavaScript que trabalhou, certo?

E é exatamente isso a que o Grunt se propõe. Com ele nós podemos simplesmente nos ater às regras de negócio e ao código em si das aplicações e esquecer as tarefas repetidas de automatização, geração ou gerenciamento. Todavia, mesmo analisando todos os conceitos e exemplos que demos até aqui, muita gente ainda considera que o Grunt seja desnecessário e esteja nos projetos apenas como mais uma ferramenta que irá tomar o seu tempo. Vejamos algumas considerações, portanto:

• Precisamos realmente das coisas que o Grunt faz?

• Provavelmente sim. Mesmo se você já for um desenvolvedor experiente e conseguir lidar com tarefas simples como as que listamos de forma rápida, você provavelmente usa muitas ferramentas distintas para fazer cada coisa, e o Grunt pode ajudar a trazê-las para o mesmo contexto. Se você ainda não faz todas elas, provavelmente deveria e o Grunt pode ajudar. No final das contas, produtividade é a palavra-chave.

• Mas o Grunt roda no Node.js e você não sabe Node...

• Você não precisa saber Node.js para trabalhar com o Grunt, da mesma forma que não precisa saber C++ para usar o Word, ou PHP para usar o WordPress.

• Temos outras formas de fazer o que o Grunt faz....

• Provavelmente, mas a pergunta é: estão elas todas organizadas em um só lugar, configuradas para executar automaticamente quando for preciso, e compartilhadas para cada uma das pessoas que trabalhem no seu projeto?

• O Grunt é uma ferramenta de linha de comandos, e você não sabe usar esse tipo de coisa.

• Infelizmente esse é de fato um ponto negativo da ferramenta, pois não temos uma versão gráfica, com GUI. Mas como desenvolvedor é extremamente importante que você saiba lidar com interfaces de linha de comando.

Figura 1. Tela do aplicativo de gerenciamento de alunos.

Gruntfile.js

O Gruntfile.js constitui o arquivo de configuração do Grunt e será basicamente composto pelo formato descrito na **Listagem 1**.

Listagem 1. Formato padrão do arquivo de configuração Gruntfile.js.

```
// constantes e funções
module.exports = function (grunt) {
 grunt.initConfig({
   // configuração
 });
 // tarefas do usuário
}
```

Nota

É importante notar que o Grunt segue a especificação do CommonJS, um projeto desenvolvido para normatizar e padronizar convenções e estilos de JavaScript. Para isso, o Grunt exporta a si mesmo como um módulo que contém suas configurações e tarefas.

A fim de facilitar o processo de configuração, os usuários Grunt devem, idealmente, armazenar quaisquer portas, funções e outras constantes, que são utilizadas na parte superior do arquivo, como variáveis globais. Isso garante que se uma função ou constante muda de repente, a edição da variável global no topo seria muito mais simples do que alterar seu valor em cada localização no arquivo. Além disso, as constantes ajudam a fornecer informações através da atribuição de um nome de variável para cada valor desconhecido.

Por último, o Gruntfile vai encerrar com uma lista de tarefas definidas pelo utilizador. Por padrão, todos os plug-ins do Grunt têm uma tarefa respectiva que pode ser chamada para realizar a saída desejada. Tarefas definidas pelo usuário podem ser referenciadas no final do Gruntfile para

serem executadas assincronamente. Por exemplo, você pode querer definir uma tarefa para administradores de sistemas que vão enviar uma versão pronta para produção da sua aplicação web, combinando compressão, concatenação, e compilação de tarefas. Você também pode definir uma tarefa "watch" para desenvolvedores web para auto compilar arquivos CoffeeScript ou Sass que exigem compilação para ser usados. Para uma equipe de estagiários, uma tarefa que combina plug-ins de validação para várias linguagens de codificação pode ser utilizada para evitar erros e auxiliar no seu processo de aprendizagem.

CoffeeScript

Basicamente, o CoffeeScript é uma pequena linguagem de programação que compila em JavaScript. Inspirado em linguagens como Ruby e Python, sua sintaxe consegue aumentar a legibilidade do código fonte, promovendo a brevidade no mesmo, e oferecendo novas funcionalidades, tal como a compreensão de listas. O código fonte do CoffeeScript é frequentemente muito mais curto do que o seu equivalente em JavaScript, e isso é possível ainda sem sacrificar o desempenho em tempo de execução.

Em dezembro de 2009, o desenvolvedor Jeremy Ashkenas anunciou o lançamento do CoffeeScript e seu compilador, que foi inicialmente escrito em Ruby e, posteriormente, reescrito em CoffeeScript. Dentre as vantagens do framework JavaScript, temos:

• O CoffeeScript oferece uma sintaxe mais sucinta e coerente. A maioria dos desenvolvedores estima que o uso do CoffeeScript consegue diminuir a quantidade final de código em até um terço.

• O CoffeeScript engloba as partes boas do JavaScript, como o ótimo modelo de objetos que subjaz o JavaScript, enquanto resolve algumas de suas partes ruins, como a eliminação da declaração with.

• O código compilado do CoffeeScript muitas vezes executa tão rápido quanto código JavaScript, senão mais. Como Ashkenas relatou: “[Você] pode evitar declarações forEach lentas, e obter a velocidade nativa para loops com muitas operações.”

• O CoffeeScript oferece muitos recursos úteis, como “a correção de classes baseadas em protótipos, compreensões sobre matrizes e objetos, literais de função vinculados, variáveis lexicais seguras, desestruturação de atribuições”, e muito mais.

Obviamente, o CoffeeScript tem algumas desvantagens, como o fato de termos ainda um outro compilador entre o desenvolvedor e o JavaScript antes de chegarmos ao resultado final. Desvantagem essa que tenta ser suprida com a geração de códigos JavaScript limpos e legíveis. Além disso, as melhorias constantes na ferramenta fizeram com a mesma se destacasse e saísse da lista de críticas fortes que recebia no início quando não tinha um sistema definido de depuração de código (corrigido a partir da sua versão 1.6.1).

O compilador, por sua vez, pode ser dividido em duas categorias: *core compiler* (compilador núcleo), que executa em um ambiente JavaScript (como um browser, por exemplo), e o *command-line compiler* (compilador de linha de comando), que executa o JavaScript em tempo real no prompt de comandos.

Configurando o ambiente

Instalando o npm

O **npm** é, em poucas palavras, um gerenciador de pacotes para a plataforma Node.js, que, no nosso projeto, irá ajudar a simplificar a instalação, atualização e configuração dos pacotes de software que o mesmo precisar. Dessa forma, pode-se facilmente gerenciar as dependências de um projeto, sem quaisquer preocupações, de forma programável.

Como o Grunt é um projeto JavaScript construído usando o Node.js, naturalmente usaremos o npm para o controle de versão. O npm também será usado para instalar e atualizar os plug-ins do Grunt e o Bower (gerenciador de pacotes padrão que usaremos no projeto). O registro npm também abriga uma variedade de outras ferramentas, incluindo o motor de testes Mocha, o motor de geração de templates Jade, e o framework web ExpressJS.

Nota: O leitor deve ter notado que estamos lidando com muitas ferramentas e integrações até agora, mas não se preocupe. A maioria delas não precisará ser entendida, uma vez que atuarão em segundo plano apenas para auxiliar como dependências.

Adicionalmente, a melhor razão para se usar o npm no nosso caso é que se precisarmos instalar um pacote que foi construído usando o Node.js, seria necessário baixar os arquivos e criar um pacote manualmente. Usando o npm, pode-se instalar e atualizar pacotes de software com um simples comando. Ele também permite a instalação de projetos de uma versão específica, uma técnica que será utilizada ao longo do artigo para assegurar que as instruções do projeto alinham-se de forma consistente com o software.

Para instalar o npm, basta que instalemos o Node.js na máquina, uma vez que o mesmo já vem atrelado ao framework por padrão. Para tanto, acesso a página de downloads do Node.js (vide seção **Links**) e baixe o arquivo .msi de instalação para Windows, de acordo com a sua versão - 32 ou 64 bits (na página você também encontrará opções de instalação para os demais Sistemas Operacionais). Execute o arquivo e siga todos os passos até o fim sem alterar as opções que já vierem marcadas por padrão. Após isso, é importante verificar se a instalação ocorreu com sucesso abrindo o seu prompt de comandos e executando os comandos a seguir:

```
node –v
npm -v
```

Isso irá imprimir as versões do Node e npm instalados, respectivamente. Caso apareça algo como “comando não encontrado”, refaça a instalação.

Para usar o npm, basta ter em mente alguns comandos importantes do mesmo:

• npm install <pacote>: comando usado para instalar um pacote no diretório atual.

• npm install –g <pacote>: faz a mesma coisa do comando anterior, porém definindo um escopo global para o mesmo pacote na máquina.
• npm ls: serve para listar todos os pacotes do Node instalados no diretório atual.
• npm install grunt: comando que usaremos para instalar a instância do Grunt.
• npm update: serve para atualizar todos os pacotes no seu diretório atual. Muito útil, pois funciona exatamente como no *Maven*, sem nos preocuparmos com gerenciamento de versões.

Instalando o Bower

O **Bower** é um gerenciador de pacotes para o desenvolvimento front-end web. Ele será usado para instalar bibliotecas do lado do cliente para o nosso projeto. O Bower é semelhante ao npm na medida em que ajuda a facilitar a instalação, atualização e configuração de pacotes. A principal diferença está na forma como ambas as ferramentas foram implementadas. Enquanto o npm usa uma árvore de dependências aninhadas e pode lidar com a instalação de várias versões de um pacote, o Bower mantém a sua árvore de dependências plana, sem aninhamentos. Isso indica que cada cliente que usa o Bower só precisa de uma versão por dependência instalada, diminuindo desse modo a necessidade de espaço extra para cada projeto.

Sem o Bower, seria o mesmo que tradicionalmente baixar as bibliotecas front-end dos sites oficiais e colocá-las nos projetos. Infelizmente, essas bibliotecas não incluem os metadados que são necessários para automatizar a atualização ou configuração das mesmas. Usando o Bower e seu cache interno, nós seremos capazes de facilmente atualizar, instalar ou remover pacotes por demanda.

Uma vez instalado o npm, nós podemos instalar o Bower simplesmente executando o seguinte comando:

```
npm install –g bower@1.3.8
```

A flag **-g** permite que você execute o Bower como um binário global. Contanto que você esteja conectado à Internet e tenha o espaço necessário para a instalação, não deverá lidar com quaisquer erros durante todo este processo. O valor **1.3.8** após o @ refere-se à versão que estamos baixando e é a mais recente no momento de escrita deste artigo. Se o autor desejar usar alguma versão anterior, tome bastante cuidado pois isso pode ocasionar alguns problemas referentes ao download de pacotes e plug-ins importantes para o correto funcionamento dos projetos.

Após isso, o Bower provavelmente estará configurado corretamente na sua máquina. Para verificar se tudo ocorreu bem, basta rodar o comando a seguir no prompt cmd, e a versão do Bower será impressa como 1.3.8:

```
bower –v
```

Dentre os comandos básicos que precisaremos para usar o Bower, destacam-se:

• bower install <pacote>: usado para instalar o pacote (dependência) referenciado. Adicionalmente, o usuário pode definir qual versão do pacote deseja instalar, bastando para isso adicionar o caractere # precedido da versão após o nome do pacote.
• bower isntall jquery: instala e configura a instância do jQuery pro projeto.
• bower uninstall <pacote>: remove o pacote em questão.

Instalando o Grunt

Não há casos de uso para os quais o Grunt deva ser instalado globalmente. Ao iniciar um novo projeto, você deve configurar o mesmo como um projeto de faceta Grunt, isto é, dizer ao seu S.O. que aquele projeto está configurado para executar com o Grunt. Para isso, basta executar o seguinte comando:

```
npm install grunt
```

Se tudo ocorrer bem, você verá o processo completo acontecer. Para verificar, digite o comando:

```
grunt –v
```

Se a mensagem "Grunt: command not found" aparecer, então o seu SO não reconheceu a instalação e os pacotes subsequentes. Se isso acontecer, digite o comando a seguir:

```
npm install –g grunt-cli
```

Aguarde o processamento e no final você verá algo parecido com o conteúdo da **Listagem 2** no seu console. O log mostra apenas as versões e a forma como a ferramenta será sincronizada em suas atualizações futuras.

Listagem 2. Resultado da instalação do Grunt.

```
D:\>npm install -g grunt-cli
C:\Users\Julio\AppData\Roaming\npm\grunt ->
C:\Users\Julio\AppData\Roaming\npm\node_modules\grunt-cli\bin\grunt
grunt-cli@0.1.13 C:\Users\Julio\AppData\Roaming\npm\node_modules\grunt-cli

├── resolve@0.3.1
├── nopt@1.0.10 (abbrev@1.0.5)
└── findup-sync@0.1.3 (lodash@2.4.1, glob@3.2.11)
```

Criando página de teste

Para verificar se todas as configurações foram feitas corretamente até aqui e analisar como todas estas extensões trabalham em conjunto, bem como ter uma noção melhor do que o Grunt gera no final, façamos uma criação rápida de um "Alô Mundo". Para isso, efetue o download do pacote de fontes deste artigo no topo da página e verifique que temos quatro projetos:
• O projeto "alo-mundo-grunt" serve como modelo inicial para ser usado neste exemplo que vamos desenvolver.

• O projeto "alo-mundo-grunt-final" é o projeto funcional final.
• O projeto "grunt-coffee-app-escolar" servirá como template para a aplicação escolar a ser desenvolvida.
• E o projeto "grunt-coffee-app-escolar-final" é referente ao projeto funcional final do artigo.

Posicione o arquivo do projeto de Alô Mundo em um local de sua preferência e de fácil localização via cmd. Agora, vamos configurar a faceta desse projeto executando os dois comandos:

```
npm install
bower install
```

Você verá várias linhas de log serem impressas até que o processo termine. Se algo de errado acontecer, o motivo será impresso também no console. Note que além de termos que configurar as ferramentas a nível de SO, também precisamos configurá-las individualmente para cada projeto. Os comandos são os mesmos, mas o npm lidará de formas distintas para ambas as situações. Note também que após esse procedimento, novas pastas serão criadas automaticamente dentro do diretório raiz, tais como "node_modules" (pasta que conterá os plug-ins e extensões) e "bower_components" (guardará os arquivos de componentes do bower, tal como o jQuery, etc).

Em seguida, dentro da pasta src, crie um novo arquivo chamado "index.html" e adicione o conteúdo da **Listagem 3** ao mesmo.

Listagem 3. Página inicial do Alô Mundo.

```
<!doctype html>
<!--[if lt IE 7]>  <html class="no-js lt-ie9 lt-ie8 lt-ie7"> <![endif]-->
<!--[if IE 7]>     <html class="no-js lt-ie9 lt-ie8"> <![endif]-->
<!--[if IE 8]>     <html class="no-js lt-ie9"> <![endif]-->
<!--[if gt IE 8]><!-->
<html class="no-js">
<!--<![endif]-->
 <head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
  <title>Alô Mundo - Grunt | CoffeeScript</title>
  <meta name="description" content="">
  <meta name="viewport" content="width=device-width">
 </head>
 <body>
  <div id="container">
  </div>
  <!-- build:js scripts/compiled.js -->
  <script type="text/javascript" src="bower_components/jquery/jquery.js">
  </script>
  <script type="text/javascript" src="scripts/main.js"></script>
  <!-- endbuild -->
 </body>
</html>
```

Esta será a estrutura HTML básica que usaremos para o exemplo. Não há muito a acrescentar exceto pelo uso das tags de importação dos scritps que faremos uso neste exemplo. Note que o jQuery se encontra entre eles, mas sequer baixaremos qualquer arquivo do mesmo, pois os gerenciadores serão responsáveis por lidar com tais dependências. É importante que o leitor se atenha a analisar bem a estrutura de tags <meta> do cabeçalho, pois quando efetuarmos a compressão dos arquivos, o resultado será bastante diferente.

Após isso, crie também um arquivo dentro da pasta src/scripts chamado "main.js" e adicione o seguinte conteúdo:

```
$(function() {
  $('#container').html('Alo Mundo');
});
```

Isso será o bastante para imprimir a mensagem que desejamos na tela. Para tal, execute o arquivo HTML em qualquer navegador e você verá a mensagem impressa. Perceba que sequer fizemos qualquer download de biblioteca para o projeto, o que atesta o funcionamento das extensões de automação.

Finalmente, para efetuar o "deploy" da aplicação, isto é, gerar os arquivos comprimidos que desejamos, vamos editar o arquivo Gruntfile.js que já veio na pasta de template. Se o mesmo não existir você deverá criar um com a configuração modelo mínima vista anteriormente. Como o arquivo é deveras grande, vamos fazer as configurações aos poucos entendendo cada uma. Para criar a versão de deploy do projeto, nós precisaremos criar uma pasta (dist) e limpar sempre o conteúdo dela antes de atualizá-la. Para tanto, é necessário criar uma tarefa que faça isso. Veja na **Listagem 4** o código necessário.

Listagem 4. Tarefa para limpar a pasta de deploy.

```
config['clean'] = {
  build: {
    files: [{
    dot: true,
    src: [
      'dist/*',
      '!dist/.git*'
    ]
    }]
  }
};
```

Adicione todos os conteúdos de tarefas logo após a criação do objeto config. Esse código realiza a configuração necessária para o plug-in "grunt-contrib-clean". Repare que a configuração segue o modelo de JSON básico, com objetos e listas sendo configuradas com chaves e colchetes. O atributo **files** distribui todos os arquivos ou padrões de diretórios de acordo com o vetor de strings configurado na opção **src**. A opção **dot**, por sua vez, é responsável por definir que todos os arquivos devem ser excluídos, inclusive os arquivos escondidos. Por fim, o valor **"!dist/.git"** serve para configurar que nenhum arquivo com essa extensão deve ser adicionado ao pacote final e pode ser usada para quaisquer extensões, como SVN, CVS, etc.

Logo após, nós devemos limpar o arquivo HTML. Para fazer isso, adicione o objeto de configuração definido na **Listagem 5** depois da declaração do objeto anterior.

Listagem 5. Código de configuração da limpeza do HTML

```
config['htmlmin'] = {
  build: {
    options: {
      collapseBooleanAttributes: true,
      removeAttributeQuotes: true,
      removeRedundantAttributes: true,
      removeEmptyAttributes: true
    },
    files: [{
      expand: true,
      cwd: 'src',
      src: '{,*/}*.html',
    dest: 'dist'
    }]
  }
};
```

Note que a opção **collapseBooleanAttributes** vai ajudar na remoção das atribuições desnecessárias dos atributos para os tipos booleanos, como os tipos "readonly" e "disabled". Em outras palavras, ela transforma a tag **<input readonly="readonly">** em **<input readonly>**.

A opção **removeAttributeQuotes** irá remover as aspas duplas das atribuições sempre que possível. Isso vai transformar a tag ** ** em ** </ div>**, por exemplo.

Ao longo dos anos, os navegadores tornaram-se melhor em heuristicamente determinar o conteúdo entre tags. Como tal, certos atributos podem agora ser considerados redundantes. Ao habilitar a opção **removeRedundantAttributes**, você poderá automaticamente remover atributos como **type="text/javascript"** de vez das suas páginas HTML.

Por último, habilite a opção **removeEmptyAttributes** para remover valores atribuídos com strings vazias.

Paralelo a isso, nós precisaremos configurar também a forma como os arquivos JavaScript serão comprimidos. Adicione o conteúdo da **Listagem 6** na sequência do arquivo de configuração.

O plug-in **grunt-usemin** será usado para comprimir nossos arquivos JavaScript. O plug-in **usemin**, por sua vez, irá unificar os demais plug-ins **grunt-contrib-concat**, **grunt-contrib-uglify**, e **grunt-ver**. Todos estes plug-ins lidam essencialmente com a compressão tanto no lado dos scripts quanto do HTML em si, ambos andam em conjunto. Observe que os destinos sempre apontam para a pasta dist que será usada para essa finalidade. A opção mangle serve para desligar a ofuscação e pode causar alguns erros de percalço, então tome cuidado se tiver lidando com muitas exceções no console de log do seu navegador.

Para finalizar, basta que criemos mais uma tarefa a fim de evitar o famoso problema dos navegadores modernos que armazenam localmente em cache arquivos usando URLs para diminuir o tempo de carregamento. Isso indica que o JavaScript, o CSS, e até mesmo arquivos de imagem podem ser carregados localmente em vez de a partir do servidor web oficial. Para resolver isso, é aconselhável armazenar em cache seus arquivos, adicionando uma **HashKey** do arquivo antes do nome do mesmo. Desta forma, toda vez que um arquivo é atualizado, seu hash o será também, atualizando consequentemente o nome do arquivo. Para conseguir isso com os arquivos JavaScript em nosso projeto, vamos adicionar o bloco de código da **Listagem 7** ao Gruntfile.js.

Repare também que adicionamos ao final do script, após a inicialização da configuração do Grunt, uma variável de vetor para armazenar os nomes de todos os objetos de tarefas criados. Estas mesmas tarefas irão executar de forma síncrona quando dermos o build no projeto, pela ordem que foram definidas. E para finalmente gerar a build, basta executarmos o comando a seguir no terminal, de dentro da pasta raiz do projeto:

Listagem 6. Código de configuração para compressão e plug-in uglify

```
config['useminPrepare'] = {
  options: {
    dest: 'dist'
  },
  html: 'src/index.html'
};
config['usemin'] = {
  options: {
  dirs: ['dist']
  },
            html: ['dist/{,*/}*.html']
};

config['uglify'] = {
  options: {
    mangle: false
  }
};
```

Listagem 7. Objeto de configuração da HashKey e variável vetor de tarefas.

```
config['rev'] = {
  files: {
    src: [
      'dist/scripts/{,*/}*.js',
    ]
  }
};

grunt.initConfig(config);

var tasks = [
  'clean',
  'useminPrepare',
  'htmlmin',
  'concat',
  'uglify',
  'rev',
  'usemin'
];
```

```
grunt build
```

Após isso, você verá vários logs no console e quando finalizar, poderá encontrar uma nova pasta dist criada no projeto, com vários arquivos dentro, dentre os quais estarão as versões

comprimidas do arquivo js, bem como do HTML. Abra os mesmos arquivos com o editor de texto e analise a sua constituição. Note que o conteúdo gerado é completamente gerenciado via JavaScript, até mesmo o conteúdo do corpo HTML. Essa é a ideia base destes frameworks. Você poderá executar o arquivo dist/index.html normalmente no seu browser e o resultado será o mesmo.

Criando o projeto escolar

Para dar início à construção da aplicação, verifique o arquivo "grunt-coffee-app-escola" que você baixou nos fontes do artigo e o posicione em um local de fácil navegação. Após isso, façamos as mesmas configurações do projeto anterior, adicionando as dependências do npm e Bower ao mesmo. Adicionalmente, a biblioteca Bootstrap também será adicionada para que possamos usar seus recursos de componentes visuais.

Comecemos então, pela configuração das tarefas no arquivo Gruntfile. Após a variável config vamos adicionar o código da **Listagem 8**.

Listagem 8. Código que cria tarefa para "assistir" as alterações nos arquivos.

```
var config = {};
  config['watch'] = {
    options: {
    nospawn: true
  },
  coffee: {
    files: ['src/coffee/{,*/}*.coffee'],
    tasks: ['coffee:server']
  },
  compass: {
    files: ['src/styles/{,*/}*.{scss,sass}'],
    tasks: ['compass:server']
  }
};
```

Essa configuração serve explicitamente para informar ao Grunt quais arquivos devem ser recriados e atualizados ante as mudanças diretas nos arquivos do CoffeeScript que estarão localizados na pasta coffee, e do Sass, presentes no diretório css. Essa configuração somente funcionará se você tiver as últimas versões do Ruby e Compass (seção **Links**) instaladas no seu computador e devidamente reconhecíveis pelo S.O., através das variáveis de ambiente. Após instalar o Ruby o comando **gem** automaticamente estará disponível. O Compass é uma gem do Ruby e pode ser instalada através do comando:

```
gem install bundler -r --source http://rubygems.org/
```

Você pode informar a versão que deseja configurar, mas é mais aconselhado usar sempre esta versão do comando, que irá lidar com as mais recentes. Demonstraremos o uso do watch mais a frente.

Continuando com a lista de tarefas, vejamos o código da **Listagem 9**.

Listagem 9. Código das tarefas do compass e coffee.

```
config['compass'] = {
  options: {
    sassDir: 'src/styles/sass',
    cssDir: 'src/styles',
    importPath: 'src/bower_components',
    relativeAssets: false
  },
  dist: {},
  server: {}
};

config['coffee'] = {
  dist: {
    files: [{
      expand: true,
      cwd: 'src/coffee',
      src: '{,*/}*.coffee',
      dest: 'dist/scripts',
      ext: '.js'
    }]
  },
  server: {
    files: [{
      expand: true,
      cwd: 'src/coffee',
      src: '{,*/}*.coffee',
      dest: 'src/scripts',
      ext: '.js'
    }]
  }
};
```

Neste código podemos ver a criação de duas tarefas: **compass** e **coffee**. Ambas lidarão com a forma como o deploy dos arquivos de CSS e CoffeeScript será feito, respectivamente. Observe que na declaração da primeira, nós simplesmente criamos referências aos seus diretórios, onde serão encontrados os arquivos de CSS das dependências, do Sass, bem como a não aceitação de caminhos relativos para lidar com os imports. Na segunda, temos a tarefa **grunt-contrib-coffee** sendo criada, com a distribuição dos arquivos de deploy e do servidor. As mesmas características de diretórios podem ser visualizadas através das opções src, dest e cwd. O leitor também deverá se atentar ao fato de que estas são as pastas que precisará manipular quando for trabalhar com os arquivos. Se desejar alterar seus nomes, fique à vontade.

Finalmente, execute o comando para iniciar o "watch" no console:

```
grunt watch
```

Isso irá iniciar a thread grunt-contrib-watch que criamos antes e verificar sempre que você fizer alterações nos arquivos e salvá-los. Por enquanto, vamos deixar a mesma executando em segundo plano, depois voltamos pra ela.

Como primeira alteração de fato na implementação, vamos navegar até o arquivo src/styles/sass e incluir o seguinte trecho de código no mesmo:

```
@import 'bootstrap-sass/lib/bootstrap.scss';
```

Esse trecho irá se responsabilizar por importar as dependências de CSS do Bootstrap para o arquivo de CSS do projeto. Note que após salvar o arquivo, um novo arquivo main.css será criado na pasta src/styles com todo o conteúdo de CSS que precisaremos para o exemplo. Esse é o resultado de criar uma thread para "assistir" às alterações como fizemos.

Em seguida, vamos começar as alterações nos arquivos do CoffeeScript, de fato. Abra o arquivo main.coffee na pasta src/coffee/app e adicione o conteúdo da **Listagem 10** ao mesmo.

Listagem 10. Código do arquivo principal do CoffeeScript no projeto.

```
Aluno = window.GLOBALS.Aluno
Painel = window.GLOBALS.Painel

$ ->
  lista_alunos = [
    {
      nome: "John"
      serie: 4
      notas: 40
      inadimplente: true
  }
    {
      nome: "Jane"
      serie: 4
      notas: 40
      inadimplente: true
    }
    {
      nome: "Max"
      serie: 4
      notas: 40
      inadimplente: false
    }
  ]

  alunos = {}

  for modelo in lista_alunos
    alunos[modelo.nome] = new Aluno modelo.nome, modelo.serie, modelo.
    notas, modelo.inadimplente

  painel = new Painel $("#tabela-alunos"), alunos
```

Comecemos a análise deste código pela quantidade de recursos que o CoffeeScript aceita, que se assemelham à forma como trabalhamos com linguagens de programação comuns, principalmente se você tiver familiaridade com o Ruby. Logo no início do arquivo, vemos a declaração de dois objetos que serão usados no exemplo. O CoffeeScript trabalha de forma semelhante às outras linguagens no que se refere à importação de classes, o conceito é o mesmo: devemos importar as classes antes de usá-las em outra classe. O início da declaração do conteúdo do arquivo vem após o sinal **->**, este que será usado também para determinar o início de outros blocos de código como métodos e construtores. Em seguida, repare que criamos a variável **lista_alunos** que servirá como uma espécie de Collection para a criação dos objetos de Aluno e seus respectivos atributos e valores. O vetor será usado para completar a lista de alunos criada em sequência através do uso do forEach (famoso no Java, e com o mesmo conceito), criando, finalmente, um objeto Painel e passando a lista de alunos como parâmetro.

Repare que a sintaxe do CoffeeScript se assemelha ao JSON quando da criação de objetos. Além disso, a criação de novos objetos implica na passagem de parâmetros e valores sem o uso de parênteses como em outras linguagens como o Java.

Vamos criar agora as classes de entidades que serão usadas para compor os atores do nosso aplicativo. Abra o arquivo "pessoa.coffee" no diretório src/coffee/entidades e o preencha com o código da **Listagem 11**.

Listagem 11. Classe Pessoa no projeto CoffeeScript.

```
class Pessoa
  constructor: (@nome) ->
    throw "Nome inválido" if @nome.length is 0

  getNome: ->
    @nome

window.GLOBALS.Pessoa = Pessoa
```

Aqui nós podemos ver a representação fiel do início de uma herança no CoffeeScript. Esta classe simples será usada como classe pai para os demais tipos de pessoa que vierem a surgir no projeto. A maior vantagem nessa abordagem, além da simplificação do código para o desenvolvedor, é a possibilidade de lidar com os prototypes do JavaScript de forma abstraída. Além disso, também estamos criando o nosso primeiro construtor com uma sintaxe deveras familiar. Os atributos de classe devem ser referenciados com o uso do **@** no início de cada nome e o lançamento de exceções pode ser feito através da palavra reservada **throw**. Repare que a condição para lançar a exceção é sempre feita após o lançamento da mesma com a respectiva mensagem. Isso é um padrão na linguagem e você deve obedecer. Para delimitar o início e fim dos blocos, nós usamos a indentação em conjunto com as linhas em branco, ou seja, sempre que quiser criar um novo método ou bloco, dê o espaço de uma linha abaixo.

Dando sequência, criaremos agora então a classe filha de Pessoa, **Aluno**. Para isso, abra o arquivo aluno.coffee no mesmo diretório e acrescente o código da **Listagem 12** ao mesmo.

A única novidade neste arquivo é o uso do **super()** logo no início do construtor. Assim como em outras linguagens OO, essa expressão é usada para efetuar uma chamada direta ao construtor da classe pai passando o valor do atributo nome como parâmetro. Dentro do mesmo construtor ainda, temos duas validações que serão úteis para verificar se os valores da série e nota do aluno estão sendo passados corretamente. O restante do código se atém à criação dos métodos de get, bem como os métodos para aprovar e reprovar o aluno.

Por último, vamos criar agora a criação do conteúdo do arquivo painel.coffee (**Listagem 13**).

Por ser muito grande, vamos dividir o entendimento deste arquivo em duas partes. Na mesma listagem é possível observar

que, além do construtor do painel que recebe os parâmetros de identificador do corpo HTML e a lista de alunos, respectivamente; e dos métodos de get/set, temos o método **init**, que será responsável por inicializar os atributos e objetos para uso posterior na tela e que está sendo chamado diretamente pelo construtor. Esse método basicamente configura o comportamento do clique no botão de inclusão de alunos que irá automaticamente verificar o valor do campo de texto, adicionando o item à lista e, consequentemente, à tabela HTML.

Listagem 12. Código de criação da classe Aluno.

```
Pessoa = window.GLOBALS.Pessoa

class Aluno extends Pessoa
  constructor: (@nome, @serie, @notas, @inadimplente) ->
    super(@nome)
    throw "Série inválida" if @serie < 1
    throw "Nota inválida" if @notas < 0

  getSerie: ->
    @serie

  getNotas: ->
    @notas

  getInadimplente: ->
    @inadimplente

  isReprovado: ->
    @serie is 0

  aprovar: ->
    @serie++
    @notas += 60

  reprovar: ->
    @serie = 0
    @notas = 0

window.GLOBALS.Aluno = Aluno
```

Listagem 13. Código do arquivo painel.coffee para exibição – parte 1.

```
Aluno = window.GLOBALS.Aluno

class Painel
  constructor: (@painel, @alunos) ->
    @init()

  init: ->
    _this = this

    @render()

    $("#btn-incluir").click ->
    chave = $("#input-aluno").val()
      _this.alunos[chave] = new Aluno chave, 1, 40, false
      _this.render()

  getAlunos: ->
    @alunos

  setAlunos: (alunos) ->
    @alunos = alunos
```

A segunda parte dessa classe é referente ao método render (**Listagem 14**) que irá recuperar o objeto DOM **tbody** da tabela na página HTML e percorrer a lista de alunos criadas no main.coffee inicialmente. Na iteração, verificamos se o aluno é inadimplente para assim criar a checkbox correspondente, bem como montar as colunas da tabela com o conteúdo de cada um deles. Por fim, o método irá também escutar os cliques nos botões de aprovação e reprovação da tabela, tratando de remover ou incrementar o conteúdo do aluno naquela linha.

Listagem 14. Código do arquivo painel.coffee para exibição - parte 2.

```
render: ->
    _this = this

    @painel.find("tbody").html ""

    for chave, valor of @alunos
      if valor.isReprovado()
        continue

      if valor.inadimplente
        check = "<input type='checkbox' name='inadimplencia' checked='true'
        disabled='true'/>"
      else
        check = "<input type='checkbox' name='inadimplencia' disabled='true'/>"

      linha =
      "<tr data-id='#{chave}'>
        <td>#{chave}</td>
        <td>#{valor.serie}</td>
        <td>#{valor.notas}</td>
        <td>#{check}</td>
        <td>
          <button class='btn btn-primary btn-aprovar'
          data-id='#{chave}'>Passar</button>
          <button class='btn btn-danger btn-reprovar' data-id='#{
          chave}'>Reprovar</button>
        </td>
      </tr>"
      @painel.find("tbody:last").append(linha)

    @painel.find(".btn-aprovar").each ->
      $(this).click ->
      chave = $(this).attr("data-id")
      _this.alunos[chave].aprovar()
        _this.render()

    @painel.find(".btn-reprovar").each ->
      $(this).click ->
        chave = $(this).attr("data-id")
        _this.alunos[chave].reprovar()
      _this.render()

window.GLOBALS.Painel = Painel
```

Observe-se sempre em relação ao uso da identação. É muito importante fazê-la corretamente para que o framework entenda onde começa e termina cada instrução.

Ao final desse processo, verifique que a thread watch criou vários arquivos .js correspondentes ao conteúdo dos arquivos CoffeeScript no diretório src/scripts. Antes de testar, não podemos esquecer de criar o arquivo HTML que conterá o conteúdo da

tabela a ser exibida. Para isso, crie um novo arquivo "index.html" na raiz da pasta src e adicione o conteúdo da **Listagem 15**.

Listagem 15. Conteúdo do arquivo index.html do nosso projeto.

```
<!doctype html>
<!--[if lt IE 7]>   <html class="no-js lt-ie9 lt-ie8 lt-ie7"> <![endif]-->
<!--[if IE 7]>      <html class="no-js lt-ie9 lt-ie8"> <![endif]-->
<!--[if IE 8]>      <html class="no-js lt-ie9"> <![endif]-->
<!--[if gt IE 8]><!-->
<html class="no-js">
<!--<![endif]-->
 <head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
  <title>Sistema de Gerenciamento de Alunos</title>
  <meta name="description" content="">
  <meta name="viewport" content="width=device-width">
  <link rel="stylesheet" href="styles/main.css">
 </head>
 <body>
  <header class="navbar navbar-static-top navbar-inverse" role="banner">
   <div class="container">
    <div class="navbar-header">
     <button class="navbar-toggle" type="button"
      data-toggle="collapse" data-target=".bs-navbar-collapse">
      <span class="sr-only">Toggle navigation</span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
     </button>
     <a class="navbar-brand">Sistema de Gerenciamento de Alunos</a>
    </div>
    <nav class="collapse navbar-collapse" role="navigation">
    </nav>
   </div>
  </header>
  <div class="container">
   <div class="row">
    <div class="col-lg-12">
     <table id="tabela-alunos" class="table table-hover table-striped
      table-border">
      <thead>
       <tr>
        <th>Nome</th>
        <th>Série</th>
        <th>Media Notas</th>
        <th>Inadimplente</th>
        <th>Ações</th>
       </tr>
      </thead>
      <tbody>
      </tbody>
     </table>
     <input id="input-aluno" class="form-control input-sm"
      style="width: 200px; display: inline-block" /> <a id="btn-incluir"
      class="btn btn-primary">Adicionar</a>
    </div>
   </div>
  </div>
  <!-- build:js scripts/compiled.js -->
  <script type="text/javascript" src="bower_components/jquery/jquery.js">
  </script>
  <script type="text/javascript" src="scripts/app/init.js"></script>
  <script type="text/javascript" src="scripts/entidades/pessoa.js"></script>
  <script type="text/javascript" src="scripts/entidades/aluno.js"></script>
  <script type="text/javascript" src="scripts/entidades/painel.js"></script>
  <script type="text/javascript" src="scripts/app/main.js"></script>
  <!-- endbuild -->
 </body>
</html>
```

Esse arquivo é muito semelhante ao primeiro que criamos no projeto de Alô Mundo, e difere apenas pela criação da tabela e a inclusão de todas as classes CSS nos elementos do corpo HTML. É importante não faltar com essas classes, pois elas tratarão de aplicar o estilo na página tal como associado às regras do Bootstrap. Além disso, também estamos efetuando a criação do botão de adição de novos alunos à listagem seguindo os mesmos requisitos de código JavaScript que foram criados nos arquivos coffee. Perceba também que os imports dos arquivos JavaScript no final precisam estar na ordem definida para evitar esquecimento de algum objeto ou até mesmo erro de carregamento no browser. É importante não remover os comentários antes e depois deles, pois serão usados para efetuar a build.

Agora você pode executar o arquivo HTML no navegador e visualizar a impressão da **Figura 1** no mesmo. Você poderá testar e verificar o correto funcionamento de todos os botões e ações da página. Caso alguma coisa não esteja funcionando corretamente, selecione a tecla F12 (no Chrome) para entrar no modo desenvolvedor. As mensagens de erro estarão disponíveis em vermelho e irão te ajudar a ver se você esqueceu alguma coisa.

Para finalizar a build, nós ainda precisamos efetuar algumas alterações no arquivo de Gruntfile. A primeira delas é incluir no mesmo arquivo as tarefas que criamos para o projeto Alô Mundo: clean, useminPrepare, usemin, htmlmin, uglify e rev, sem alterações. Após isso, ainda precisamos de mais duas tarefas (**Listagem 16**): **copy**, usada para efetuar a transferência dos arquivos de uma pasta ao seu destino final; e **cssmin**, que se responsabilizará por comprimir os arquivos de CSS.

Listagem 16. Tarefas copy e cssmin no arquivo Gruntfile.

```
config['copy'] = {
  build: {
    files: [{
      expand: true,
      dot: true,
      cwd: 'src',
      dest: 'dist',
      src: []
    }]
  }
};

config['cssmin'] = {
  dist: {
    files: {
      'dist/styles/main.css': [
        'src/styles/{,*/}*.css'
      ]
    }
  }
};
```

Para finalizar, vamos adicionar a mesma lista de tarefas num vetor, porém com as novas agora adicionadas (**Listagem 17**).

Mais uma vez, para gerar o build basta executar o comando **grunt build** e tudo estará nos conformes para ir pra produção.

Listagem 17. Vetor com a lista de tarefas atualizada.

```
var tasks = [
  'clean:build',
  'useminPrepare',
  'htmlmin',
  'cssmin',
  'concat',
  'uglify',
  'copy',
  'rev',
  'usemin'
];
```

Se o leitor encontrar quaisquer problemas pelo caminho, ou alguma das ferramentas não funcionar, verifique os passos e refaça-os até que esteja tudo ok. A vantagem de usar esse tipo de recurso é que só perdemos tempo configurando tudo uma vez, e as atualizações de versões podem ser feitas automaticamente com os respectivos comandos de update.

Não é preciso ser um expert para dominar os recursos do Grunt e do CoffeeScript, e apenas com o desenvolvimento de um sistema simples como esse já conseguimos ver muito do potencial de ambas as tecnologias. Mas ainda há muito mais a ser explorado, como polimorfismo, encapsulamento e outros conceitos de orientação a objetos, listas, estruturas dinâmicas, etc.

Autor

Júlio Sampaio
Analista de sistema e entusiasta da área de Tecnologia da Informação. Atualmente é desenvolvedor na empresa iFactory, 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 do JSHint
http://www.jshint.com/

Página de download oficial do Node.js
http://nodejs.org/download/

Página oficial de instalação do Bower
http://bower.io/#install-bower

Página de download do Ruby para Windows
http://rubyinstaller.org/downloads/

Página de instalação do Compass
http://compass-style.org/install/

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