

Conheças as melhores práticas de programação assíncrona para JavaScript

# Front-end magazine

Edição 02



**Explorando o jQuery Mobile**  
Desenvolva aplicações mobile  
de forma rápida e eficiente

# TESTES UNITÁRIOS

DOMINE A CONSTRUÇÃO DE TESTES  
DE UNIDADE COM JAVASCRIPT



# MVP

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

APLIQUE ESSE INVESTIMENTO  
NA SUA CARREIRA...

E MOSTRE AO MERCADO  
QUANTO VOCÊ VALE!

CONFIRA TODO O MATERIAL  
QUE VOCÊ TERÁ ACESSO:

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

POR APENAS  
**R\$ 69,90\*** mensais

\*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR  
MAIS DO MERCADO!



 **DEVMEDIA**

## EXPEDIENTE

### Editor

Diogo Souza ([diogosouzac@gmail.com](mailto:diogosouzac@gmail.com))

### Consultor Técnico

Daniella Costa ([daniella.devmedia@gmail.com](mailto:daniella.devmedia@gmail.com))

### Produção

**Jornalista Responsável** Kaline Dolabella - JP24185

**Capa e Diagramação** Romulo Araujo

### Atendimento ao leitor

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

### Publicidade

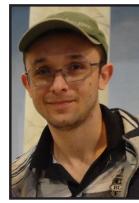
[publicidade@devmedia.com.br](mailto:publicidade@devmedia.com.br) – 21 3382-5038

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

### Fale com o Editor!

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

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



**DIOGO SOUZA**

[diogosouzac@gmail.com](mailto:diogosouzac@gmail.com)

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

# Sumário

### Conteúdo sobre Boas Práticas

## 04 – Principais Features do jQuery Mobile

[ Júlio Sampaio ]

### Conteúdo sobre Boas Práticas

## 16 – Programação assíncrona com Node.js

[ Caio Ribeiro ]

### Artigo no estilo Curso

## 26 – Testes unitários em JavaScript: Introdução - Parte 1

[ Sueila Sousa ]

# Principais Features do jQuery Mobile

Conheça todas as principais features do framework JavaScript mais famoso para desenvolvimento mobile

A experiência com o mundo mobile se torna cada vez mais aparente e necessária no que se refere à quantidade de pessoas que usam dispositivos ou lidam com o universo móvel. No ponto de vista de um desenvolvedor, a experiência com o mundo móvel pode já ter sido vivenciada antes com HTML5, JavaScript ou tecnologias relativas, mas talvez nenhuma delas seja tão confortável quanto o jQuery é para desenvolver aplicações para ambientes móveis. Além disso, é uma forma um tanto quanto poderosa de criar aplicações web, sejam elas para desktop ou móvel, e o jQuery Mobile se encaixa perfeitamente nesse cenário.

O jQuery Mobile foi construído tendo como base a própria biblioteca JavaScript do jQuery e focando, de fato, em dar ao desenvolvedor maneiras fáceis de criar aplicações móveis sem tanto esforço e complexidade quanto em outras tecnologias. Basicamente, ficará a cargo do desenvolvedor customizar seu HTML com arquivos de CSS, e isso rapidamente se transforma num tormento, especialmente se for necessário tratar de temas e estilização como na maioria dos casos.

O framework jQuery Mobile é um framework UI de código fonte aberto e destinado a atender múltiplas plataformas. Ele foi feito usando HTML5, CSS3, além do framework jQuery e de seguir os padrões da Open Web. Ele provê ainda uma porção de widgets amigáveis que já são especificamente trabalhados e desenhados para o mundo móvel. Ele tem um poderoso framework para criar temas para estilizar as aplicações. Além disso, suporta Ajax para vários tipos de tarefas, como navegação de páginas e transições.

Como o jQuery Mobile segue a linha dos padrões web abertos, você pode ter certeza de que as aplicações terão o máximo de compatibilidade e suporte dentre a grande variedade de browsers e plataformas que existem. É possível, portanto, criar uma aplicação uma vez e ela

## Fique por dentro

O desenvolvimento de aplicações web tem se mostrado essencial no universo de programação de uma forma geral. Mais que isso, é cada vez mais importante saber unir a web ao mundo dos aparelhos móveis, obtendo uma integração rápida, produtiva e eficiente. O jQuery Mobile dá um framework do jQuery para desenvolvimento mobile baseado no jQuery original e em JavaScript, que fornece vários recursos importantes para trabalhar de forma flexível, como responsividade, tematização, aplicação de efeitos, etc.

Neste artigo veremos de forma prática a maioria dos conceitos básicos necessários para entender o que esse framework é capaz de fazer, assim como conhecer todos os recursos mais usados do mesmo e como eles podem ajudar no desenvolvimento de aplicações reais.

irá funcionar de forma igual para dispositivos Android, iOS, tablets, Blackberry, Windows, etc., até mesmo em plataformas em crescimento como a Boot2Gecko e o Tizen. O mesmo código também irá executar normalmente no Chrome, Firefox, Opera, dentre os diversos tipos de browsers que existem. O interessante é saber que esse tipo de tecnologia também terá um funcionamento similar dentre outros tipos de dispositivos, como TV inteligentes, bem como quaisquer outros que se encaixem nos padrões da web aberta. Veja na seção **Links** a lista completa de browsers certificados suportados, plataformas.

Em algumas plataformas mais antigas, algumas das funcionalidades, como animações CSS em 3D e Ajax provavelmente não serão suportadas, mas com o jQuery Mobile o conceito de “Progressive Enhancement” explica que funcionalidades básicas serão suportadas inicialmente. No futuro, quando browsers e plataformas se tornarem mais poderosas, as aplicações automaticamente farão uso dessas capacidades e oferecerão suporte de atualização. Na maioria dos cenários, o desenvolvedor não terá que escrever código ou interferir de alguma forma. Esse é um

grande passo quando comparamos aplicações web com aplicações móveis nativas.

Enquanto escrevendo código para aplicações nativas, será necessário escrever código em diferentes linguagens, baseado na plataforma selecionada. Será necessário também compilar o mesmo código para cada plataforma, e construir pacotes binários para cada uma delas que possam executar nos dispositivos específicos. Atualizar uma versão da mesma aplicação significar ter de voltar e refazer todo o exercício de verificar/corrigir o código, reconstruir e reempacotar. E a complexidade pode chegar a níveis bem mais elevados considerando a quantidade de novas versões geradas. Depois de um certo ponto, é provável que o gerenciamento de tal versionamento se torne inviável ou relativamente complicado.

Claro que existem vantagens em se usar aplicações nativas. A performance deverá ser um fator crucial a se considerar. Alguns tipos de aplicações necessitam de um tempo de resposta alto, o que leva a considerar código nativo como a melhor saída para tal situação. Paralelo a isso, através de aplicações nativas é possível acessar o núcleo dos Sistemas Operacionais e os recursos dos dispositivos, como câmeras, calendários, leitores, etc. Coisas que não são facilmente implementadas nos dias de hoje simplesmente com HTML5.

A HTML5 é um novo mundo que foi apresentado aos desenvolvedores web front-end. A distância entre essa linguagem e as nativas diminui a cada dia, uma vez que uma grande gama de recursos semelhantes pode ser implementada e usada a partir de simples chamadas JavaScript. O framework jQuery Mobile tem uma lista variada de plugins e ferramentas que podem ajudar a construir esse tipo de aplicação. Além de ter também uma comunidade muito ativa e atualizada a todas as constantes mudanças sofridas pela plataforma.

Outrossim, as aplicações web tem evoluído. No princípio, elas usavam código nativo puro para construir UIs, depois veio a era do Flash e outros plugins UI baseados no mesmo princípio (Como o Silverlight, por exemplo). Porém, essas tecnologias têm seus dias contados frente ao gigantesco crescimento das tecnologias cross-platform, como o jQuery Mobile.

Mas o que significa dizer que o jQuery Mobile foi feito para mobile? Primeiramente, ele foi otimizado para toques. É comum ver usuários reclamarem de sites que são adaptados do desktop para o móvel e não conseguirem fornecer sequer um clique de um botão sem falhas, ou certas funcionalidades não ocorrendo da forma que acontecem no uso da aplicação através de um browser no computador. O jQuery Mobile se encaixa perfeitamente a este cenário, uma vez que o mesmo foi criado para se adaptar a todos os diferentes tamanhos de tela, seja ela a de um smartphone ou de um notebook. Além disso, jQuery Mobile é responsivo. Isso já diz muito. Frameworks responsivos que consigam fornecer adaptação de telas para ambientes totalmente diferentes são difíceis de encontrar, ainda mais fazendo o que se propõem a fazer. Nesse tipo de cenário, a única preocupação do desenvolvedor é escrever o código uma vez, e deixar com que o jQuery se encarrega dos redimensionamentos e demais tarefas.

## Criando a primeira aplicação em jQuery Mobile

Programaticamente falando, o framework requer o uso de sintaxe (Marcações HTML) para a maioria das tarefas básicas e para construir toda a UI. Você voltar um pouco para os scripts JavaScript somente, onde a sintaxe declarativa não ajuda muito e logicamente para adicionar a camada lógica. Essa estratégia difere um pouco da abordagem usada por outros frameworks, uma vez que a maioria requer muito mais código JavaScript além de ter uma linha de aprendizado mais íngreme.

Se você já é familiarizado com HTML, CSS e JavaScript/jQuery, então será muito fácil aprender jQuery Mobile, caso contrário é aconselhável dar uma boa estudada sobre esses assuntos antes de prosseguir no artigo.

Sobre a IDE a ser usada, existem diversas que podem facilitar a forma de visualização do Drag and Drop assim como o desenvolvimento com o jQuery Mobile por completo. Porém, para iniciar seu aprendizado, basta utilizar a sua ferramenta editora de texto favorita. Aqui iremos usar o Notepad++, que já vem preparado para fornecer uma lista de recursos poderosos para trabalhar com esse tipo de desenvolvimento (veja na seção **Links** a página oficial do Notepad++). Também será necessário o uso de um browser.

### Nota

Seria interessante se o leitor dispusesse de vários browsers para verificar o funcionamento das aplicações construídas aqui, tanto desktop quanto mobile.

Existem duas formas básicas de usar o jQuery Mobile no seu site. Ambas envolvem a forma como você irá importar e usar os arquivos js e css da biblioteca. A primeira forma lida com a importação dos arquivos usando CDN (Content Delivery Network), isto é, usando URLs do próprio site do jQuery que nos permitirão fazer uso do jQuery Mobile desde que tenhamos uma conexão com a internet. O código da **Listagem 1** mostra como fazer isso.

### Listagem 1. Import dos arquivos do jQuery Mobile via CDN

```
<head>
<title>jQuery Mobile - Front-end Magazine</title>
<link rel="stylesheet" href="http://code.jquery.com/mobile/1.4.4/
jquery.mobile-1.4.4.css" />
<script type="text/javascript" src="http://code.jquery.com/
jquery-1.11.0.min.js"></script>
<script type="text/javascript" src="http://code.jquery.com/mobile/1.4.4/
jquery.mobile-1.4.4.js"></script>
</head>
```

É possível ver as importações dos links diretamente dos servidores do jQuery, e todas dentro da tag HTML `<head>`. Além disso, vale ressaltar que estamos usando as últimas versões disponíveis no momento de escrita deste artigo. Note também a importação do arquivo JavaScript do jQuery, uma vez que a biblioteca é uma das dependências para uso do jQuery Mobile.

A segunda forma de se importar é efetuando o download dos arquivos na origem fisicamente. Basta efetuar o download a partir

# Principais Features do jQuery Mobile

dos mesmos links usados na **Listagem 1** e adicionar as chamadas relativas aos diretórios dos mesmos. Para esse tipo de abordagem é interessante renomear os arquivos, removendo as versões dos mesmos. Você pode usar essa abordagem quando estiver trabalhando em modo offline, ou quando desenvolver uma aplicação que não tiver acesso à internet.

Criemos agora a estrutura básica inicial de diretórios para o projeto, sempre dividindo os arquivos em pastas específicas para o mesmo, mantendo, assim, um nível de organização mínima para os projetos (**Listagem 2**).

Em seguida, crie a página index.html, tal como demonstrado na estrutura de diretórios e a preencha com o conteúdo visto na **Listagem 1**, adicionando também o conteúdo da **Listagem 3**.

## Listagem 2. Estrutura de diretórios para o projeto

```
root
  js
    // Seus arquivos jquery, caso efetue o download dos mesmos
  img
  css
  pages
  index.html
```

## Listagem 3. Conteúdo do corpo da página jQuery Inicial

```
<!DOCTYPE html>
<html>
<head>
<title>jQuery Mobile - Front-end Magazine</title>
<meta charset="UTF-8">
<meta name='viewport' content='width=device-width, initial-scale=1'>
<link rel="stylesheet" href="http://code.jquery.com/mobile/1.4.4/
  jquery.mobile-1.4.4.css"/>
<script type="text/javascript" src="http://code.jquery.com/
  jquery-1.11.0.min.js"></script>
<script type="text/javascript" src="http://code.jquery.com/mobile/1.4.4/
  jquery.mobile-1.4.4.js"></script>
</head>
<body>
<!-- Página Principal -->
<div id='main' data-role='page'>
<div data-role='header'>
<h1>Olá, Seja Bem vindo(a)!</h1>
</div>
<div id='content' data-role='content'>
<p>Introdução ao jQuery</p>
</div>
<div data-role='footer'>
<h4>Revista Front-end Magazine</h4>
</div>
</div>
</body>
</html>
```

Note que nesta listagem adicionamos algumas coisas novas. Podemos observar algumas mudanças:

- Adicionamos o elemento simples DOCTYPE no início do documento, isso quer dizer que estamos usando HTML5 por padrão na página, não somente refletido nesta declaração em específico, mas também nos demais elementos dentro das páginas;

- Adicionamos duas novas tags <meta> dentro da tag <head> da página que se referem, respectivamente, à definição do encoding da página para exibição de caracteres especiais (UTF-8), e à definição de tamanho que a tela deverá ocupar dentro do dispositivo que a estiver exibindo;

- A divisão do corpo da página é básica e semelhante à forma como trabalhamos com tabelas (dividida em cabeçalho, corpo e rodapé). Dessa forma, basta adicionar os elementos data-role às div's com seus respectivos valores e a mágica será feita. O jQuery Mobile irá configurar a página estruturalmente com os elementos específicos, bem como aplicará os efeitos e design associados de forma automática.

Vejamos como a tela ficou no seu browser no final através da **Figura 1**.



**Figura 1.** Tela de Boas Vindas do jQuery Mobile, redimensionada

## Nota

Atente que todos os elementos têm seu data-role específicos, inclusive o elemento página, que será referenciado por data-role="page".

Note que, conforme discutimos até aqui, o design responsivo é característica padrão do jQuery Mobile, tanto que você pode efetuar os testes de responsividade no seu próprio browser, redimensionando o mesmo. Além disso, é possível ver a aplicação do estilo padrão do jQuery Mobile, distribuído nas cores, fontes e tamanhos que o mesmo utiliza.

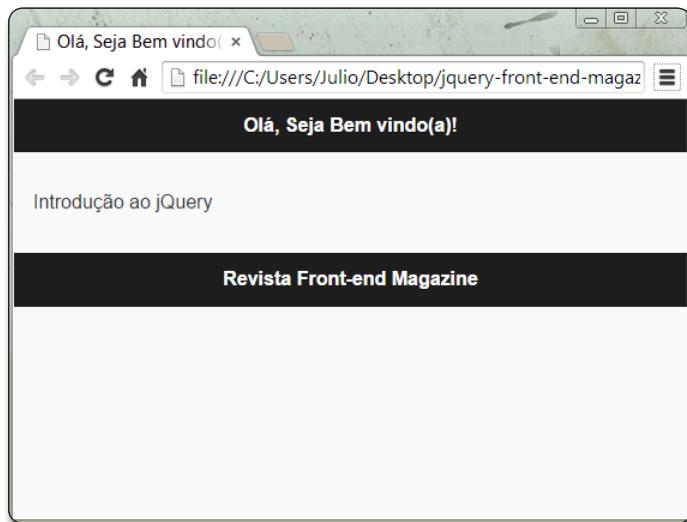
## Temas

Com o jQuery Mobile, você também tem em mãos a possibilidade de trabalhar com os skins (temas) criados pela comunidade que estão disponíveis para estilizar suas aplicações de outra forma.

Por padrão, o jQuery Mobile disponibiliza cinco temas básicos chamados de *color swatches* (amostra de cores), que são nomeados das letras **a** até **e**. Por padrão, o esquema **d** é usado quando criado uma página. É possível mudar esse tipo de configuração apenas

adicionando um novo atributo aos elementos divisores (div's), tal como na **Listagem 4**.

O resultado pode ser visualizado na **Figura 2**.



**Figura 2.** Tela de Boas Vindas com tema b aplicado

#### **Listagem 4.** Configurando temas com o jQuery Mobile

```
<div id='main' data-role='page' data-theme='a'>
  <div data-role='header' data-theme='b'>
    ...
  </div data-role='footer' data-theme='b'>
```

## Templates

Os templates são formações pré-formadas com o intuito de facilitar o aproveitamento de esforços, como estruturação e design, de modo a aumentar a produtividade dentro de um dado framework.

Em uma aplicação que usa um template simples, cada página da aplicação terá seu próprio arquivo HTML. Uma página será sempre envolvida dentro de um container de página definido como a div que vimos nos exemplos anteriores. Quando você executa a aplicação, o jQuery Mobile irá carregar a primeira página da aplicação (ou a página principal, main) dentro do DOM, no qual a referência será mantida durante todo o ciclo de vida

da aplicação. O único momento onde a página principal deixará de existir será quando o usuário navegar para outras páginas, que agora serão marcadas como páginas ativas. A exceção da página principal, todas as outras páginas serão removidas do DOM quando o usuário sai delas. Isso garante uma navegação mais otimizada. Por sua vez, a navegação entre as páginas é feita usando links âncoras. Os âncoras são decorados como botões usando o atributo *data-role="button"*. Clicar num link desses fará com que a navegação ocorra com algumas transições CSS3, e a nova página será aberta via Ajax.

Mas antes de prosseguirmos, vamos dar uma olhada na **Tabela 1** que traduz exatamente a diferença entre os diferentes e principais níveis de telas que podemos ter nos templates, a página e o diálogo.

Vejamos então como colocar ambos conceitos em prática, isto é, como criar uma página HTML simples com um template que nos permita navegar entre duas páginas da aplicação.

Dentro da pasta pages crie um novo arquivo HTML e nomeie-o como "principal.html". Dentro desse arquivo adicione o conteúdo da **Listagem 5**.

#### **Listagem 5.** Página principal para controle de navegação

```
<body>
  <!-- Página Principal -->
  <div id='main' data-role='page' data-theme='a'>
    <div data-role='header' data-theme='b'>
      <h1>Principal</h1>
    </div>
    <div id='content' data-role='content'>
      <a href="pagina2.html" data-role="button">Ir para Página 2</a>
    </div>
    <div data-role='footer' data-theme='b'>
      <h4>Rodapé</h4>
    </div>
  </div>
</body>
```

Observe que preservamos o conteúdo das tags de cabeçalho e da página jQuery. Como estamos usando template, não tem problema então usar páginas HTML separadas. Isso quer dizer que a chamada à página 2, demonstrada no exemplo irá funcionar. Logo, criemos então a página que será redirecionada, nomeando-a de "pagina2.html" e adicione o conteúdo da **Listagem 6**.

Página	Diálogo
Um página (ou page) no jQuery Mobile é o objeto básico escrito em conjunto com o container <code>&lt;div data-role="page"&gt;</code> que será exibido na tela. Você pode embutir vários controles HTML5 e widgets dentro de uma página. O framework jQuery Mobile automaticamente calcula e exibe todos estes controles, fazendo deles amigáveis ao toque para dispositivos como tal. Sua aplicação pode ter um série de arquivos HTML individuais cada um representando uma simples página, ou ele pode ter um simples arquivo HTML contendo múltiplas div's dentro dele. Você pode fornecer também outros links internos para abrir outras páginas usando Ajax em conjunto com o CSS3. Dessa forma, conforme vimos até então, as demais páginas serão escondidas.	Um diálogo é uma página que tem um atributo <code>data-role="dialog"</code> . Você também pode carregar uma página como um diálogo adicionando o atributo <code>data-rel="dialog"</code> ao link da página. O diálogo é estilizado de forma diferente à forma como a página ganha seu design, e isso ele também aparece no meio da tela abaixo da página. Ademais, o diálogo também provê um botão para fechá-lo no seu cabeçalho.

**Tabela 1.** Diferenças entre os tipos de telas no jQuery Mobile

# Principais Features do jQuery Mobile

## Listagem 6. Página secundária a ser chamada pela principal

```
<div id='main' data-role='page' data-theme='a'>
<div data-role='header' data-theme='b'>
<h1>Principal</h1>
</div>
<div id='content' data-role='content'>
<a href="#" data-role="button" data-rel="back">< Voltar</a>
</div>
<div data-role='footer' data-theme='b'>
<h4>Rodapé</h4>
</div>
</div>
```

Se não funcionar, pode ser que tenha algum problema de versão relacionado. Dessa forma, basta adicionar o atributo data-ajax="false" ao mesmo link e o efeito será aplicado normalmente.

Note também que usamos na segunda página o atributo "data-rel" com o valor back para efetuar o redirecionamento para a página anterior no histórico do browser. Isso também significa que se você tiver ambos href e data-rel no mesmo link, o jQuery Mobile irá ignorar o href e executar apenas o efeito de voltar para a página anterior.

Quando se trata de templates para múltiplas páginas, o jQuery Mobile necessitará que o arquivo HTML tenha múltiplas páginas dentro dele mesmo. Cada página é encapsulada dentro de uma página container como uma <div data-role="page">. O ID da página é usado para identificar as páginas por linking ou invocando ações nelas. O id da página deve ser único, assim como os demais id's para os demais componentes. Quando você executar a aplicação, o jQuery Mobile irá carregar todas as páginas disponíveis no DOM e exibe a primeira página que ele encontra no HTML. A navegação entre as páginas é especificada através do mesmo uso de âncoras visto anteriormente, e você pode personalizar esses links como botões usando o atributo data-role="button". Ao clicar em qualquer link, a navegação ocorre com as mesmas transições CSS3, e a nova página é então exibida via Ajax. Vejamos então como fazer isso.

Crie uma nova página HTML e nomeie-a "multi.html" e adicione o conteúdo da **Listagem 7** nela.

Note que agora adicionamos as duas páginas juntas dentro de um mesmo arquivo HTML. Depois, mudamos as referências para que o link apontasse agora para o id da div de página 2, ao invés da página física como fazíamos antes. Dessa forma, temos uma implementação mais robusta e livre de erros de Ajax e navegação. O funcionamento será praticamente o mesmo, porém com essa diferença imperceptível aos olhos do usuário final.

Em contrapartida, esse tipo de abordagem deve levar em consideração alguns fatores (dentre os quais alguns negativos):

- Um template multipáginas será mais pesado em função do crescimento do DOM (Muitas vezes insignificante diante da diminuição de requisições necessárias);
- A aplicação necessitará suporte JavaScript. Isso irá limitar sua escolha de plataformas alvo e você terá que ignorar algumas plataformas legadas. Mas isso também é algo pouco preocupante, dada a cada vez maior gama de browsers que aderem aos padrões atuais.

vel, dada a cada vez maior gama de browsers que aderem aos padrões atuais.

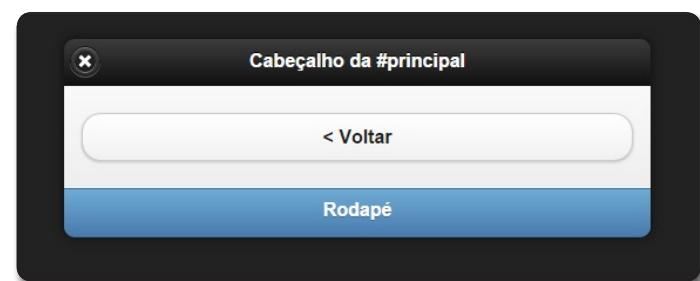
- Por outro lado, apenas o primeiro carregamento será mais lento, depois disso todos os demais serão rápidos, uma vez que as páginas já estão carregadas.

## Listagem 7. Conteúdo da página inicial de multi template

```
<div id="principal" data-role="page">
<div data-role="header">
<h1>Cabeçalho da #principal</h1>
</div>
<div data-role="content">
<a href="#pagina2" data-role="button" data-ajax="false">Ir página 2</a>
</div>
<div data-role="footer">
<h4>Rodapé da #principal</h4>
</div>
</div>
<div id="pagina2" data-role="page" data-title="Template Multi-Página">
<div data-role="header">
<h1>Cabeçalho da #principal</h1>
</div>
<div data-role="content">
<a href="#" data-role="button" data-rel="back">< Voltar</a>
</div>
<div data-role="footer" data-theme='b'>
<h4>Rodapé</h4>
</div>
</div>
```

## Diálogos

Para converter uma página em um diálogo é bem simples, basta adicionar o atributo data-rel="dialog" ao link que chamará a mesma página. Dessa forma, teremos o modo diálogo selecionado. Faça isso para o mesmo exemplo usando o template multipáginas criado e fazendo a modificação devida. O resultado poderá ser observado na **Figura 3**.



**Figura 3.** Tela da listagem transformada em Diálogo

Também é possível personalizar a forma como o diálogo abre, através dos efeitos de transição, comuns à maioria dos componentes desse tipo no jQuery. Tente por exemplo adicionar os atributos a seguir e verificar o comportamento do diálogo após:

```
data-transition="slidedown"
data-transition="flip"
```

Além disso, você também pode querer alterar certas características gráficas dos componentes do jQuery que vêm de forma padrão, como a posição que o botão de fechar vem posicionado, por exemplo (à esquerda). Para tal, você precisará conhecer um pouco mais do CSS do framework, como podemos ver na **Listagem 8**. Crie uma nova página “dialog-customizado.html” e adicione o conteúdo da listagem na mesma.

#### **Listagem 8.** Estilizando diálogo no jQuery Mobile

```
<div id="principal" data-role="page">
<div data-role="header">
<h1>Cabeçalho da #principal</h1>
</div>
<div data-role="content">
<a href="#dialogo" data-role="button" data-rel="dialog">Ir página 2</a>
</div>
<div data-role="footer">
<h4>Rodapé da #principal</h4>
</div>
</div>
<div id="dialogo" data-role="page" data-title="Template Multi-Página">
<div class="ui-corner-top ui-overlay-shadow ui-header ui-bar-a" role="banner">
<a href="#main" data-icon="delete" data-iconpos="notext"
class="ui-btn-right ui-btn ui-btn-icon-notext ui-btn-corner-all ui-
shadow ui-btn-up-a"
title="Fechar" data-theme="a" data-transition="pop" data-direction="reverse">
<span class="ui-btn-inner ui-btn-corner-all">
<span class="ui-btn-text">Fechar</span>
<span class="ui-icon ui-icon-delete ui-icon-shadow"></span>
</span>
</a>
<h1 class="ui-title" tabindex="0" role="heading" aria-level="1">
Diálogo Customizado</h1>
<div data-role="content">
<a href="#" data-role="button" data-rel="back">< Voltar</a>
</div>
<div data-role="footer" data-theme='b'>
<h4>Rodapé</h4>
</div>
</div>
</div>
```

Repare que mantivemos a estrutura final das páginas que ví-nhamos criando até então, porém a estruturação muda um pouco, uma vez que desejamos assemelhar a implementação à criada pelo jQuery Mobile por padrão. As mudanças consistirão basicamente em analisar o HTML gerado pelo framework e substituir determinadas classes CSS, como a classe “ui-btn-right” em vez da “ui-btn-left”, o que fará com que o componente botão se alinhe à direita. O resultado pode ser verificado na **Figura 4**.



**Figura 4.** Tela de diálogo customizado com botão à direita

Esse tipo de implementação é muito importante pois é através dela que muitas mudanças importantes no projeto serão feitas. É preciso conhecer bem o framework para inferir alterações significativas no mesmo, de tal forma que surtam o mesmo efeito anterior. Você pode praticar esse tipo de implementação analisando o código gerado pelo jQuery Mobile em alguma ferramenta de análise e inspeção.

#### **Transições**

Conforme falamos anteriormente, o jQuery Mobile faz uso extensivo de CSS3 para aplicar determinados efeitos, dentre estes os mais comuns, de transição. A lista de efeitos é limitada, o que significa que pode não atender todas as necessidades do desenvolvedor quando da implementação de certas funcionalidades. Por padrão, os efeitos “fade” são usados para as páginas e o “pop” é usado para os diálogos (você pode encontrar a lista de todos os efeitos no site oficial do projeto), todavia, você pode querer navegar entre páginas usando um efeito customizado que funcionará de forma diferente aos padrões do framework.

Para criar nosso próprio efeito, precisaremos atuar com CSS, logo crie uma nova página HTML (“efeito-custom.html”) e adicione a tag `<style>` à mesma. Em seguida, adicione o conteúdo da **Listagem 9**, de forma a prover os recursos básicos para aplicar o efeito desejado.

#### **Listagem 9.** CSS3 para aplicar efeito de salto na página

```
.salto.in, .salto.in.reverse {
-webkit-transform: translateY(0);
-webkit-animation-name: saltoin;
-webkit-animation-duration: 1s;
-webkit-animation-timing: cubic-bezier(0.1, 0.2, 0.8, 0.9);
}
@-webkit-keyframes saltoin {
0% { -webkit-transform: translateY(100%); }
90% { -webkit-transform: translateY(-10%); }
100% { -webkit-transform: translateY(0); }
}
.salto.out, .salto.out.reverse {
-webkit-transform: translateY(100%);
-webkit-animation-name: saltoout;
-webkit-animation-duration: 1s;
-webkit-animation-timing: cubic-bezier(0.1, 0.2, 0.8, 0.9);
}
@-webkit-keyframes saltoout {
0% { -webkit-transform: translateY(0); }
90% { -webkit-transform: translateY(110%); }
100% { -webkit-transform: translateY(100%); }
}
```

Além disso, para o efeito funcionar, adicione o atributo `data-transition="salto"` ao link. Note que o efeito aplicado será de uma espécie de salto na página tanto indo quanto voltando na mesma.

Também podemos usar JavaScript para aplicar efeitos nas páginas do jQuery Mobile. Por exemplo, criemos na mesma página do exemplo anterior uma nova tag `<script>` dentro da própria tag `<head>` e adicionemos o código contido na **Listagem 10**.

## Listagem 10. Exemplo de função JavaScript para criar efeito de transição

```
<script type="text/javascript">
function transicaoCustomizada(nome, reverso, $para, $de) {
var diferido = new $.Deferred();
// Define a animação customizada
$para.width("0");
$para.height("0");
$para.show();
$de.animate({ width: "0", height: "0", opacity: "0" },
{ duration: 750 },
{ easing:'easein' }
);
$para.animate({ width: "100%", height: "100%", opacity: "1" },
{ duration: 750 },
{ easing:'easein' }
);
// Padroniza o template do jQuery Mobile
classeReversa = reverso ? "reverso": "";
classeView = "ui-mobile-viewport-transitioning viewport-" + nome;
$para.add($de).removeClass("out in reverso " + nome);
if ( $de && $de[0] != $para[0] ) {
$de.removeClass($.mobile.classePaginaAtiva);
}
$para.parent().removeClass(classeView);
diferido.resolve(nome, reverso, $para, $de);
$para.parent().addClass(classeView );
if ($de) {
$de.addClass(nome + " out" + classeReversa);
}
$para.addClass($.mobile.classePaginaAtiva + " " + nome + " in" + classeReversa);
return diferido.promise();
}
// Registra a transição customizada
$.mobile.transitionHandlers["slide-sumir"] = transicaoCustomizada;
</script>
```

Além disso, altere também o atributo data-transition para o novo valor: slide-sumir. Após o teste, você verificará que o efeito mudou para uma mistura de slide para a esquerda/direita com um efeito de sumiço. Todo esse processo foi possível graças à sobrescrita do objeto Deferred, com a configuração da altura e largura do mesmo, possibilitando que as mesmas pudessem ser aplicadas ao efeito de animação em conjunto com a função show() do próprio jQuery. Note que as configurações de duração das animações são essenciais para definir o escopo do efeito, este que pode ser alterado à vontade pelo desenvolvedor de forma a alcançar maiores níveis de personalização.

Uma vez terminada a transição, o código deve garantir que a página correta esteja ativa. Logo após, é necessário chamar a função diferido.resolve() e retornar a função promise(). Finalmente, é importante registrar a transição customizada de forma a poder associá-la ao atributo jQuery correspondente. A função transitionHandlers() se encarregará disso.

Não existe nenhuma regra básica que defina qual das duas implementações é melhor, isso vai depender da forma como você está trabalhando baseada nos padrões do projeto e/ou restrições de clientes. Usar CSS3 é mais intuitivo e fácil para a maioria dos profissionais que trabalham com web, além de mais fácil de manter e não requerer conhecimentos de lógica de programação. Além disso, é provável que no futuro novas versões do jQuery

Mobile surjam com novas regras de transição e acabem quebrando as suas customizadas.

## Componentes e Widgets

### Toolbar

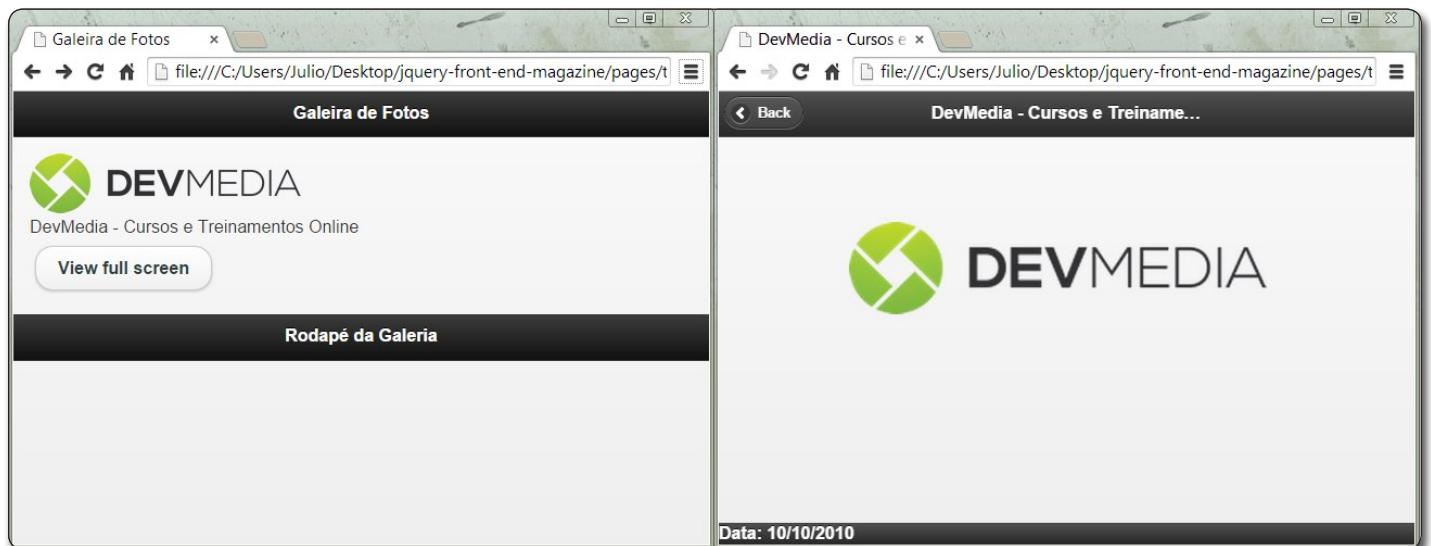
No jQuery Mobile existem duas toolbars (barra de ferramentas) básicas, o Header (Cabeçalho) e o Footer (Rodapé). Até agora já abordamos os dois nos exemplos utilizados, mas onde e para o que eles podem ser usados? O que o jQuery consegue abordar a mais em relação a essas áreas? O header basicamente sempre exibe o nome da página, o título do cabeçalho, mas pode ser usado para controlar também a navegação através do **navbar**. O footer pode ser usado para várias coisas, tais como incluir botões e formulários de controle e pode ser customizado para as necessidades mais específicas. Ele pode ser usado como controle de navegação também, assim como para exibir informações relacionadas à empresa, copyright, etc.

Vejamos o código presente na **Listagem 11**. Ele exemplifica um bom uso da toolbar para controlar a navegação de páginas.

## Listagem 11. Exemplo de função JavaScript para criar efeito de transição

```
<div id="principal" data-role="page">
  <div data-role="header">
    <h1>Galeria de Fotos</h1>
  </div>
  <div data-role="content">
    
    <br>DevMedia - Cursos e Treinamentos Online
    <br><a href="#photo" data-role="button" data-inline="true">
      View full screen</a>
  </div>
  <div data-role="footer">
    <h4>Rodapé da Galeria</h4>
  </div>
</div>
<div id="photo" data-role="page" data-fullscreen="true" data-add-back-btn="true">
  <div data-role="header" data-position="fixed">
    <h1>DevMedia - Cursos e Treinamentos Online</h1>
  </div>
  <div data-role="content">
    
  </div>
  <div data-role="footer" data-position="fixed">
    Data: 10/10/2010
  </div>
</div>
```

O que fizemos foi basicamente simular uma galeria de fotos, com uma imagem apenas (verifique os arquivos de download ou use qualquer imagem). Criamos uma página simples, com navegação multi-página, nenhuma novidade. Detalhes são os novos atributos usados na página referente à imagem em tamanho aumentado, que conta com atributos como data-add-back-btn, que concede à div a função de controladora de navegação. Além disso, na segunda tela é possível analisar a visualização do rodapé somente no



**Figura 5.** Telas de representação do uso da toolbar como galeria de fotos

final da página, conferindo um ar de aplicação mais trabalhada, ao contrário do padrão que é sempre exibir o footer ao final de cada página e seu respectivo conteúdo. Veja na **Figura 5** o resultado do exemplo da **Listagem 11**.

Vários são os tipos de implementação possíveis com a toolbar, desde a implementação de tabulação dando efeitos mais divisórios às telas, até a criação de cabeçalhos customizados com a inclusão de botões de controle, personalização de cores, CSS, imagens, etc.

E em relação ao rodapé, pode ser customizado de igual forma. Basta substituir a div de footer da **Listagem 11** pelo conteúdo ilustrado na **Listagem 12** e teremos um rodapé totalmente personalizado.

Veja o resultado ilustrado na **Figura 6**.



**Figura 6.** Resultado de footer personalizado

Essa implementação já trouxe mais novidades sobre o uso de toolbar. A classe ui-bar, adicionada à div footer, já nos informa que faremos configurações além das já padrões existentes, possibilitando a adição de controles à mesma. Em seguida, adicionamos a classe ui-grid-a que nos habilita a usar um fieldset com um layout de duas colunas. A div container com a role fieldcontain-

#### **Listagem 12.** Footer personalizado para exemplo da Listagem 11

```
<div data-role="footer" data-position="fixed" class="ui-bar">
<fieldset class="ui-grid-a">
<div class="ui-block-a" data-role="fieldcontain">
<label for="notaSlider">Que nota você dá (1-10)?</label>
<input type="range" name="notaSlider" id="notaSlider" value="5"
min="1" max="10"/>
</div>
<div class="ui-block-b">
<div data-role="fieldcontain">
<fieldset data-role="controlgroup" data-type="horizontal">
<legend>Compartilhe:</legend>
<input type="radio" name="arquivoCompartilhado"
id="arquivoCompartilhadoNinguem"
value="arquivoCompartilhado-1" checked="checked" data-theme="c"/>
<label for="arquivoCompartilhadoNinguem">Ninguém</label>
<input type="radio" name="arquivoCompartilhado"
id="arquivoCompartilhadoAmigos"
value="arquivoCompartilhado-2" data-theme="c"/>
<label for="arquivoCompartilhadoAmigos">Amigos</label>
<input type="radio" name="arquivoCompartilhado"
id="arquivoCompartilhadoPublico"
value="arquivoCompartilhado-3" data-theme="c"/>
<label for="arquivoCompartilhadoPublico">Público</label>
</fieldset>
</div>
</div>
</fieldset>
</div>
```

na primeira coluna tem como classe a ui-block-a que define que esse conteúdo deve estar posicionado na primeira coluna.

Para criar o slider basta criar um input de formulário comum e tipá-lo como “range”, informando seus limites nos atributos min e max. Finalmente, para criar os botões de seleção toggle, basta criar um fieldset com o data-role configurado para controlgroup e inputs para cada uma das opções, não esquecendo de referenciá-las com o mesmo atributo name.

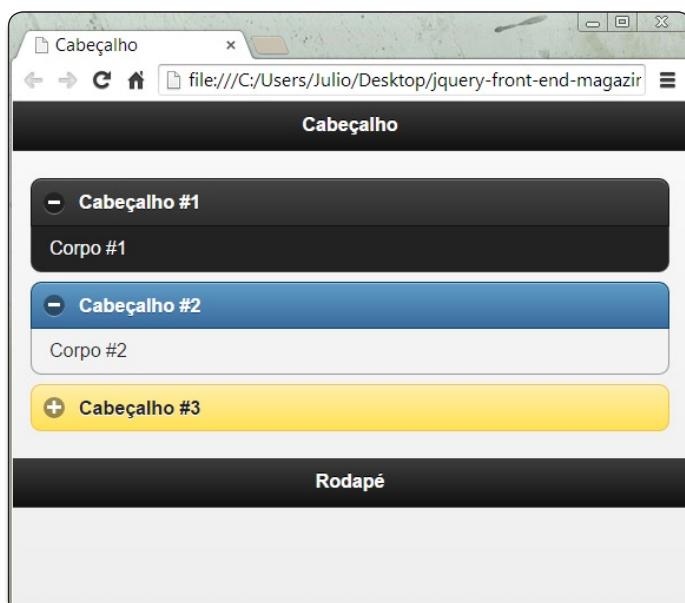
# Principais Features do jQuery Mobile

## Botões

Existe uma infinidade de ações que podem ser efetuadas dentro de uma aplicação, assim como a necessidade de se criar os mais diversos tipos de botões que atendam a tais comandos. O jQuery Mobile possibilita o uso dos elementos botões como links, controladores de navegação, listas, controladores de formulários, dentre outros. Além disso, várias são as formas que podem ser exibidos, dentro dos limites de estilo estipulados pelo framework.

Vejamos o exemplo demonstrado na **Listagem 13**, bem como seu resultado na **Figura 7**.

Substitua o conteúdo da div content pelo apresentado na Listagem. O data-role=collapsible possibilita facilmente a criação de listas selecionáveis e em efeito sanfona. Ao fazer uso desse recurso, os ícones de mais/menos são automaticamente importados pelo jQuery.



**Figura 7.** Exemplo de lista em efeito sanfona

## Listagem 13. Exemplo de uso de botões como listas de seleção no jQuery Mobile

```
<div data-role="content">
<div data-role="collapsible" data-collapsed="false" data-theme="a"
    data-content-theme="a">
    <h3>Cabeçalho #1</h3>
    Corpo #1
</div>
<div id="collapsr" data-role="collapsible" data-collapsed="false"
    data-theme="b"
    data-content-theme="b">
    <h3>Cabeçalho #2</h3>
    Corpo #2
</div>
<div id="collapsr" data-role="collapsible" data-collapsed="true" data-theme="e"
    data-content-theme="e">
    <h3>Cabeçalho #3</h3>
    Corpo #3
</div>
```

## Formulários

A tag comum da HTML, <form>, usada para lidar com a comunicação HTTP da sua aplicação dinâmica é totalmente encapsulada pelo jQuery Mobile de forma a torná-la amigável ao toque e executar em diversos dispositivos de forma responsiva. Um formulário pode ter múltiplos controles listados por padrão verticalmente e você pode agrupá-los usando um fieldset com o data-role setado para controlgroup, conforme demonstrado anteriormente. Além disso, ainda pode contar com todo o poder do Ajax para fazer todo o processo de comunicação com o servidor.

Mas talvez uma das funcionalidades mais interessantes desse tipo de recurso seja a facilidade de validar formulários usando o próprio jQuery Mobile. Em conjunto com a HTML5 ele consegue se valer de validações padrão de página, assim como criar suas próprias. Dê uma olhada na **Listagem 14**.

## Listagem 14. Formulário exemplo para validação de campos

```
...
<script type="text/javascript">
$(document).ready(function() {
    $('#form').submit(function() {
        var len = $('#comentario').val().length;
        if (len < 10 || len > 100) {
            $('#erro').text('Comentário com tamanho inválido!').show().fadeOut(5000);
            return false;
        } else
        return true;
    });
});
</script>
</head>
<body>
<!-- Página Principal -->
<div id="principal" data-role="page">
    <div data-role="header">
        <h1>Cabeçalho</h1>
    </div>
    <div data-role="content">
        <form id='form' action="#" method='post'>
            <div data-role='fieldcontain'>
                <label for='nome'>Nome</label>
                <input id='nome' name='nome' type='text' required placeholder='Digite seu Nome' />
            </div>
            <div data-role='fieldcontain'>
                <label for='email'>Email</label>
                <input id='email' name='email' type='email' required placeholder='Digite seu Email' />
            </div>
            <div data-role='fieldcontain'>
                <label for='comentario'>Comentários</label>
                <textarea id='comentario' name='comentario' required placeholder='Digite seus comentários <Minímo 10/Máximo 50>'></textarea>
            </div>
            <div id='erro' style='color: #f00'></div>
            <input type='submit' data-transition='pop' value='Enviar' />
        </form>
    </div>
    <div data-role="footer">
        <h4>Rodapé</h4>
    </div>
</div>
</body>
```

Observe que a criação do formulário já atende a algumas especificações padrão de HTML5 e outros tipos de formulário, fazendo uso do recurso de placeholder, campos obrigatórios, etc. A única novidade está nos data-role do fieldset e nos campos de input do formulário (**Figura 8**). Atente também para o uso das funções com o jQuery para lidar com os valores dos campos. Nesse tipo de cenário, o uso do jQuery pode ser feito de igual para o mundo web.

Um outro tipo de implementação bem comum para esse tipo de recurso de formulário é iniciar um dado formulário com uma lista de opções já selecionadas, como os valores de um campo de select no formulário. Vejamos a **Listagem 15**.

O resultado pode ser visualizado na **Figura 9**. Perceba as mudanças feitas no atributo data-placeholder, bem como as opções visíveis pelos elementos opt1 e opt2, além dos grupos criados para facilitar a associação de elementos.

The screenshot shows a web page titled "Cabeçalho". It contains three input fields: "Nome" with value "DevMedia", "Email" with value "devmedia@devmedia.com", and "Comentários" with value "abc". Below the form, a red message says "Comentário com tamanho inválido!". At the bottom is a blue "Enviar" button and a black "Rodapé" footer.

**Figura 8.** Resultado da validação do campo de Comentários

The screenshot shows a dropdown menu titled "Menu de Seleção Básico" containing several options. One option, "Opção 1", is checked. Another option, "Opção Disabilitada", is grayed out. The menu also includes sections for "Opções no Grupo1" and "Opções no GrupoA", each with two checked options: "Group Opção1" and "Group Opção2" under Group1, and "Group OpçãoA" and "Group OpçãoB" under GroupA.

**Figura 9.** Página de formulário exemplo com campos selecionados

**Listagem 15.** Exemplo de formulário com valores pré-selecionados

```
<script type="text/javascript">
$(document).ready(function() {
  $('#selectid').selectmenu({
    theme:'d',
    inline: false,
    corners: true,
    icon: 'star',
    iconpos: 'left',
    shadow: true,
    iconshadow: true});
});

</script>
</head>
<body>
<!-- Página Principal -->
<div id="principal" data-role="page">
  <div data-role="header">
    <h1>Cabeçalho</h1>
  </div>
  <div data-role="content">
    <form action="#" method="post">
      <div data-role="fieldcontain">
        <label for="selectid" class="select">Menu de Seleção Básico</label>
        <select name="selectid" id="selectid" multiple data-native-menu="false" data-overlay-theme='e'>
          <option value='Menu de Seleção Básico' data-placeholder="true">Menu de Seleção Básico</option>
          <option value="opt1">Opção 1</option>
          <option value="disabledopt1" disabled>Opção Desabilitada</option>
          <option value="opt2">Option 2</option>
        <optgroup label="Opções no Grupo1">
          <option value="grp11">&ampnbsp&ampnbsp&ampnbsp&ampnbspGroup Opção1</option>
          <option value="grp22">&ampnbsp&ampnbsp&ampnbsp&ampnbspGroup Opção2</option>
        </optgroup>
        <optgroup label="Opções no GrupoA">
          <option value="grpA1">&ampnbsp&ampnbsp&ampnbsp&ampnbspGroup OpçãoA</option>
          <option value="grpB1">&ampnbsp&ampnbsp&ampnbsp&ampnbspGroup OpçãoB</option>
        </optgroup>
      </select>
    </div>
    </form>
  </div>
  <div data-role="footer">
    <h4>Rodapé</h4>
  </div>
  </div>
</body>
```

### Página de Loading

Por padrão, o jQuery Mobile exibe uma animação com o tema a sem nenhum texto quando tenta carregar uma página. Se acontecer quaisquer erros no meio do caminho, então a página para de processar e uma mensagem “Error Loading Page” é exibida com o tema e. Com o jQuery Mobile, é possível, com poucas linhas de código, sobreescriver as chamadas padrão, por um conjunto de status preestabelecidos que podem controlar as mensagens a serem exibidas no lado cliente. Veja na **Listagem 16**.

### Temas

O framework jQuery Mobile provê um sistema leve para tematizar componentes, que suporta muitas propriedades CSS3, como

# Principais Features do jQuery Mobile

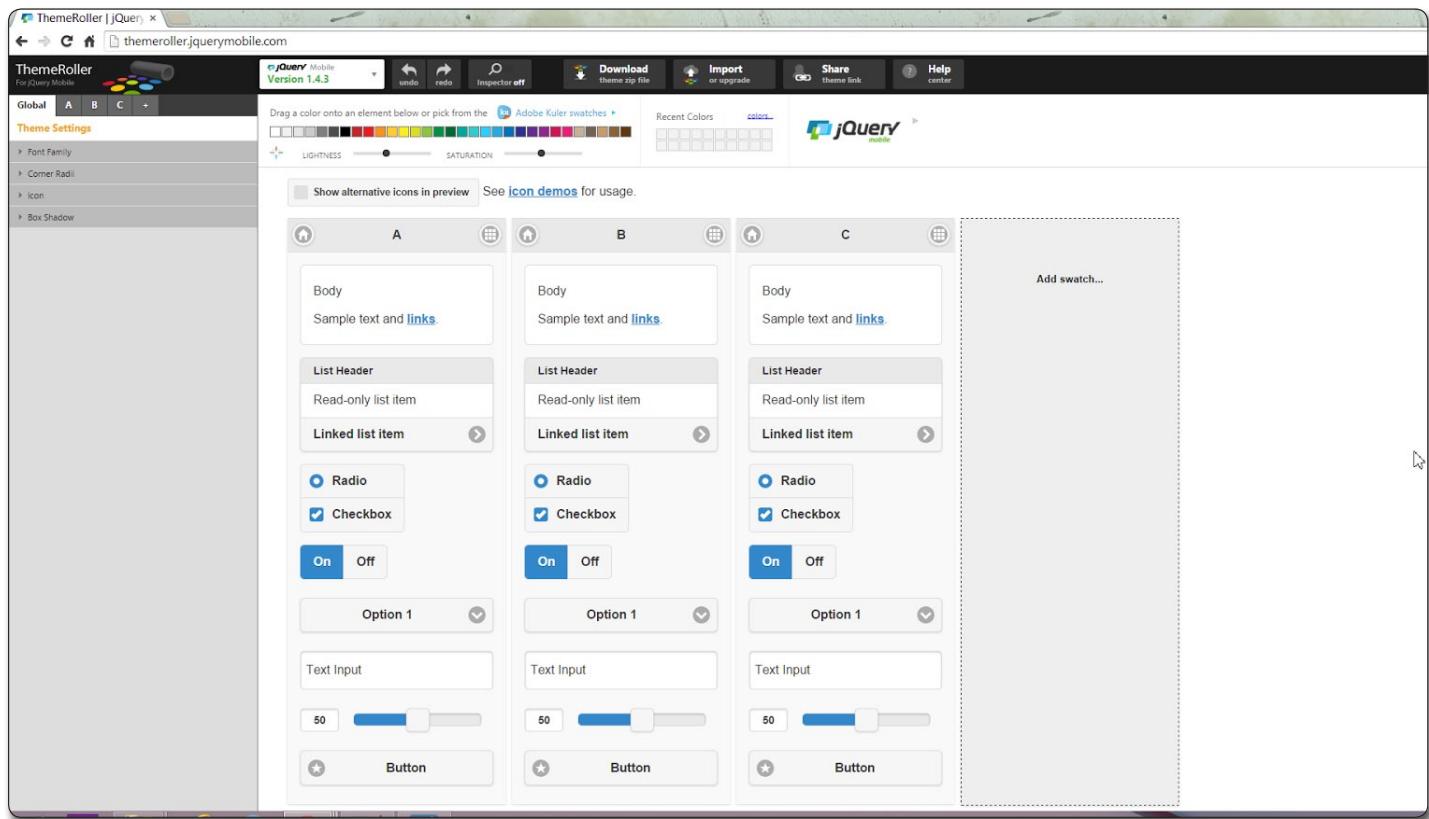


Figura 10. Página do ThemeRoller para criação de temas

bordas redondas, sombreamento e gradientes. Ele também fornece uma lista de ícones que você pode usar em suas aplicações móveis. Além disso, você pode se valer das amostras e temas já existentes no jQuery Mobile. Você também pode usar o site ThemeRoller (seção Links) para criar seus próprios temas (Figura 10), de modo a ter ainda mais flexibilidade na hora de montar o escopo da sua aplicação.

#### Listagem 16. Exemplo de sobreescrita das mensagens de erro e loading

```
$(document).ready(function() {
$(document).bind("mobileinit", function() {
$.mobile.loadingMessage = "Carregando...";
$.mobile.loadingMessageTextVisible = true;
$.mobile.loadingMessageTheme = "b";
$.mobile.pageLoadErrorMessage = "Desculpa, aconteceu um problema!";
$.mobile.pageLoadErrorMessageTheme = "b";
});});
```

Existem muitas outras features acerca do jQuery Mobile que não serão cobertas aqui, até mesmo para não fugir do foco, que é de fato apresentar a tecnologia aos que ainda não conhecem o poder que ela fornece ao desenvolvimento de aplicações web mobile. O jQuery Mobile nasceu com o intuito de unir o poder do já famoso framework JavaScript, o jQuery, ao recente e cada vez mais requisitado mundo web mobile, de forma a fornecer suporte, componentes de fácil uso e modificação, além de uma manutenção

adequada dos recursos usados, dadas as frequentes atualizações sofridas pelas bibliotecas ante à comunidade desenvolvedora.

Ainda teremos muito a melhorar nesse framework, principalmente relacionado aos estilos muito característicos, mas hoje o jQuery Mobile se prova de extrema valia para quem desenvolve para a web.

#### Autor



##### Júlio Sampaio

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



#### Links:

**Lista de dispositivos e plataformas suportados pelo jQuery Mobile**  
[www.jquerymobile.com/gbs/](http://www.jquerymobile.com/gbs/)

**Página oficial do Notepad++**  
[www.notepad-plus-plus.org/](http://www.notepad-plus-plus.org/)

**Site do ThemeRoller jQuery Mobile.**  
[www.themeroller.jquerymobile.com/](http://www.themeroller.jquerymobile.com/)



**DEVMEDIA**

# DÊ UM SALTO EM CONHECIMENTO!

Acesse o maior  
portal para  
desenvolvedores  
da América  
Latina!



**20**  
mil  
posts

**430**  
mil  
cadastrados

**10**  
milhões de  
page-views  
por mês

# Programação assíncrona com Node.js

## Boas práticas de programação assíncrona com JavaScript server-side

Tudo na web se trata de consumismo e produção de conteúdo. Ler ou escrever blogs, assistir ou enviar vídeos, ver ou publicar fotos, ouvir músicas e assim por diante. Isso fazemos naturalmente todos os dias na internet. E cada vez mais aumenta a necessidade dessa interação entre os usuários com os diversos serviços da web. De fato, o mundo inteiro quer interagir mais e mais na internet, seja através de conversas com amigos em chats, jogando games online, atualizando constantemente suas redes sociais ou participando de sistemas colaborativos. Esses tipos de aplicações requerem um poder de processamento extremamente veloz para que seja eficaz a interação em tempo real entre cliente e servidor. E mais, isto precisa acontecer em uma escala massiva, suportando de centenas a milhões de usuários.

Então o que nós, desenvolvedores, precisamos fazer? Precisamos criar uma comunicação em tempo real entre cliente e servidor — que seja rápida, atenda muitos usuários ao mesmo tempo e utilize recursos de I/O (dispositivos de entrada e saída) de forma eficiente. Qualquer pessoa com experiência em desenvolvimento web sabe que a versão atual do HTTP 1.1 não foi projetada para suportar tais requisitos. E pior, infelizmente existem aplicações que adotam de forma incorreta o uso deste protocolo, implementando soluções *workaround* (“Gambiarras”) que requisitam constantemente o servidor de forma assíncrona, geralmente aplicando a técnica de *long-polling*. Para sistemas trabalharem em tempo real, servidores precisam enviar e receber dados utilizando comunicação bidirecional, ao invés de utilizar intensamente request/response do HTTP aplicando Ajax. E também temos que manter esse tipo comunicação de forma leve e rápida para permitir escalabilidade em uma aplicação.

### O problema das arquiteturas bloqueantes

Os sistemas para web desenvolvidos sobre plataformas como .NET, Java, PHP, Ruby ou Python possuem uma

### Fique por dentro

Quando se trata de desenvolvimento front-end, o Node.js é disparado uma das tecnologias que mais intrigam e impressionam a comunidade de desenvolvimento, por suas várias características fundamentais, como poder usar JavaScript no servidor, bem como o papel que ele exerce nessa comunicação. Para o leitor que já está familiarizado com JavaScript, Node.js é só mais um curto passo ao aprendizado. Neste artigo veremos desde conceitos de configuração, ambiente, características, posicionamento atual, até as vantagens da programação assíncrona, foco central que discute como implementar um bom código que evite a criação do famoso anti-pattern *callback-hell* do JavaScript.

característica em comum: eles paralisam um processamento enquanto utilizam um I/O no servidor. Essa paralisação é conhecida como modelo *blocking-thread*. Para entender melhor esse conceito, vamos a um exemplo: temos uma aplicação web e nela cada processo é uma requisição feita pelo usuário. Com o decorrer da aplicação, novos usuários irão acessá-la, gerando múltiplos processos no servidor. Um sistema de arquitetura bloqueante vai enfileirar cada requisição que são realizadas ao mesmo tempo e depois ele vai processando, uma a uma. Este modelo não permite múltiplos processamentos ao mesmo tempo. Enquanto uma requisição é processada, as demais ficam em espera, ou seja, a aplicação bloqueia os demais processos de fazer I/O no servidor, mantendo-os em um pequeno período de tempo numa fila de requisições ociosas.

Esta é uma arquitetura clássica, existente em diversos sistemas web e que possui um design ineficiente. É gasta grande parte do tempo mantendo requisições em uma fila ociosa enquanto é executado I/O para apenas uma requisição. Para exemplificar melhor, tarefas de I/O são tarefas de enviar e-mail, consultar o banco de dados, leitura de arquivos em disco. E essas tarefas bloqueiam o sistema inteiro enquanto não são finalizadas. Com o aumento de usuários acessando o sistema, a frequência de gargalos será maior, surgindo a necessidade de se fazer uma escalabilidade

vertical (*upgrade* dos servidores) ou escalabilidade horizontal (inclusão de novos servidores trabalhando para um *load balancer*). Ambos os tipos se tornam custosos, quando se fala em gastos com infraestrutura. O ideal seria buscar novas tecnologias que façam bom uso dos servidores existentes, que permitam o uso intenso e máximo do processamento atual.

Foi baseado neste problema que, no final de 2009, Ryan Dahl, e mais 14 colaboradores, criaram o Node.js. Esta tecnologia possui um modelo inovador: sua arquitetura é totalmente *non-blocking thread* (não-bloqueante) que apresenta uma boa performance com consumo de memória e utiliza ao máximo e de forma eficiente o poder de processamento dos servidores, principalmente em sistemas que produzem uma alta carga de processamento. Sistemas criados com Node.js estão livres de aguardarem por muito tempo o resultado de seus processos, e mais importante, não sofrem *dead-locks*, pelo simples fato de trabalharem em *single-thread*. Além dessas vantagens, desenvolver sistemas nessa plataforma é super simples e prático.

O Node.js é uma plataforma altamente escalável e de baixo nível, logo, você terá a possibilidade de programar diretamente com diversos protocolos de rede e internet ou utilizar bibliotecas que acessam recursos do sistema operacional. Para programar em Node.js basta dominar a linguagem JavaScript. Ele utiliza a *engine* Javascript V8, a mesma utilizada no navegador Google Chrome.

## Características do Node.js

### **Single-thread**

Suas aplicações serão *single-thread*, ou seja, cada aplicação terá instância de uma única *thread* por processo iniciado. Se você está acostumado a trabalhar com programação *multithread*, por exemplo, Java ou .NET, infelizmente não será possível com Node.js, mas saiba que existem outras maneiras de se criar um sistema que com processamento paralelo. Por exemplo, você pode utilizar uma biblioteca nativa chamada de clusters, que é um módulo que permite implementar uma rede de processos de sua aplicação, você cria N processos de sua aplicação e uma delas se encarrega de balancear a carga, permitindo processamentos paralelos um único servidor. Outra maneira é adotar a programação assíncrona nas tarefas se seu servidor.

Esse será o assunto mais abordado durante o decorrer deste artigo, pelo qual explicaremos diversos cenários e exemplos práticos de como são executadas em paralelo, funções em assíncronas não-bloqueante.

### **Event-Loop**

Node.js é orientado a eventos. Ele segue a mesma filosofia de orientação de eventos do JavaScript de browser; a única diferença são os eventos, ou seja, não existem eventos de **click** do mouse, **keyup** do teclado ou qualquer evento de componentes do HTML. Na verdade, trabalhamos com eventos de I/O como, por exemplo, o evento **connect** de um banco de dados, um **open** de um arquivo, um **data** de um *streaming* de dados e muitos outros.

O Event-Loop é o agente responsável por escutar e emitir eventos. Na prática, ele é basicamente um loop infinito que a cada iteração verifica em sua fila de *listening* de eventos se um determinado evento foi disparado. Quando ocorre, é emitido um evento. Ele o executa e envia para fila de executados. Quando um evento está em execução, nós podemos programar qualquer lógica dentro dele e isso tudo acontece graças ao mecanismo de *callback* de função do JavaScript.

Esta técnica que permite trabalhar em cima do design *event-driven* com Node.js. Ele foi inspirado pelos frameworks Event Machine do Ruby e Twisted do Python. Porém, o Event-loop do Node.js é mais performático por que seu mecanismo é nativamente executado de forma não-bloqueante. Isso faz dele um grande diferencial em relação aos seus concorrentes que realizam chamadas bloqueantes para iniciar seus respectivos loops de eventos.

### **Por que deve-se aprender Node.js?**

- **JavaScript everywhere:** Praticamente o Node.js usa JavaScript como linguagem de programação *server-side*. Essa característica permite que você reduza e muito sua curva de aprendizado, afinal a linguagem é a mesma do JavaScript *client-side*, seu desafio nesta plataforma será de aprender a fundo como funciona a programação assíncrona para se tirar maior proveito dessa técnica em sua aplicação. Outra vantagem de se trabalhar com JavaScript é que você poderá manter um projeto de fácil manutenção. Você terá facilidade em procurar profissionais para seus projetos, e vai gastar menos tempo estudando uma nova linguagem *server-side*. Uma vantagem técnica do JavaScript comparador com outras linguagens de *back-end* é que você não irá utilizar mais aqueles frameworks de serialização de objetos JSON (*JavaScript Object Notation*), afinal o JSON *client-side* é o mesmo no *server-side*, há também casos de aplicações usando banco de dados orientado a documentos (por exemplo: MongoDB ou CouchDB) e neste caso toda manipulação dos dados são realizadas através de objetos JSON também.

- **Comunidade ativa:** Esse é um dos pontos mais fortes do Node.js. Atualmente existem várias comunidades no mundo inteiro trabalhando muito para esta plataforma, seja divulgando posts e tutoriais, palestrando em eventos e principalmente publicando e mantendo +70000 módulos no site NPM (*Node Package Manager*). Aqui no Brasil temos dois grupos bem ativos no Google Groups temos o NodeBR e no Facebook tem o grupo Node.js Brasil (ver seção **Links**).

- **Ótimos salários:** Desenvolvedores Node.js geralmente recebem bons salários. Isso ocorre pelo fato de que infelizmente no Brasil ainda existem poucas empresas adotando essa tecnologia. Isso faz com que empresas que necessitem dessa tecnologia paguem salários na média ou acima da média para manterem esses desenvolvedores em seus projetos. Outro caso interessante são as empresas que contratam estagiários ou programadores juniores que tenham ao menos conhecimentos básicos de JavaScript, com o objetivo de treiná-los para trabalhar com Node.js. Neste caso, não espere um alto salário e sim um amplo conhecimento preenchendo o seu currículo.

- **Ready for real-time:** O Node.js ficou popular graças aos seus frameworks de interação real-time entre cliente e servidor. O SockJS, Socket.IO, Engine.IO são alguns exemplos disso. Eles são compatíveis com o recente protocolo WebSockets e permitem tratar dados através de uma única conexão bidirecional, tratando todas as mensagens através de eventos JavaScript.
- **Big players:** LinkedIn, Walmart, Groupon, Microsoft e Paypal são algumas das empresas usando Node.js atualmente, no Brasil conheço algumas empresas por exemplo: Agendor, Sappos, Neostack e tem mais um monte de outras empresas e projetos (veja na seção [Links](#)).

## Onde posso usar o Node.js?

### Chats

Um chat é o mais típico exemplo de aplicação multi-usuário em tempo real. Desde IRC até muitos protocolos proprietários e abertos sobre portas não-padrão, até mesmo a habilidade de implementar tudo hoje no Node.js com websockets rodando sobre a mesma porta padrão 80.

A aplicação de chat é realmente é um ótimo exemplo de ser usada com Node.js: é leve, tem alto tráfego de dados, porém exige pouco processamento/computação e que executa dentre vários dispositivos. Além de tudo, é bem simples, ótimo para os desenvolvedores que estão iniciando o aprendizado na tecnologia, cobrindo a maior parte dos paradigmas usados pela linguagem. Basicamente, uma aplicação desse tipo funciona dentro de um domínio de website já pronto onde as pessoas podem ir e efetuar a troca de mensagens entre si conectados por uma estrutura de programação e redes. No lado do servidor, temos uma aplicação simples que implementa duas coisas:

- Uma requisição GET que serve a página web contendo ambas mensagem e botão de enviar para inicializar uma nova entrada de mensagem;
- Websockets que escutam por novas mensagens emitidas pelos seus clientes.

No lado cliente nós temos uma página HTML com uma série de handlers configurados, um para o botão de Envio, que seleciona a mensagem e a envia para o websocket, e outro que escuta por mensagens que estão chegando no cliente. Obviamente, este é um modelo simples e básico, mas que baseia os demais em variância às suas complexidades.

### API sobre um objeto DB

Embora o Node.js realmente brilhe com aplicações em tempo real, é natural ter que expor os dados a partir de um objeto de banco de dados (MongoDB, por exemplo). Dados JSON armazenados permitem que o Node.js funcione sem a diferença de impedâncias e conversão de dados. Por exemplo, se você estiver usando Rails, você deve converter JSON para modelos binários e em seguida, expô-los de volta como JSON sobre o HTTP quando os dados são consumidos por frameworks como o Backbone.js, Angular.js, etc.,

ou mesmo por simples chamadas jQuery Ajax. Com o Node.js, você pode simplesmente expor seus objetos JSON com uma API REST para o cliente a consumir. Além disso, você não precisa se preocupar com a conversão entre JSON e tudo aquilo que for ler ou escrever em seu banco de dados (se estiver usando MongoDB). Em suma, você pode evitar a necessidade de múltiplas conversões usando um formato de serialização de dados uniforme em todo o cliente e servidor de banco de dados.

### Entradas de Filas

Se você está recebendo uma grande quantidade de dados simultâneos, o banco de dados pode se tornar um gargalo. O Node.js pode facilmente lidar com as próprias conexões simultâneas. Mas porque o acesso a bancos de dados é uma operação bloqueante, nos depararemos certamente com problemas. A solução é reconhecer o comportamento do cliente antes de os dados serem realmente gravados na base de dados.

Com essa abordagem, o sistema mantém a sua capacidade de resposta sob uma carga pesada, o que é particularmente útil quando o cliente não precisa de confirmação firme de uma gravação de dados bem-sucedida. Exemplos típicos incluem: o registro ou gravação de dados de rastreamento de usuário, com processamento em lotes e não utilizados até mais tarde; bem como operações que não precisam ser refletidas instantaneamente (como a atualização de um count 'Likes' no Facebook), onde a consistência eventual (tantas vezes usada no mundo NoSQL) é aceitável.

Os dados são colocados em fila por algum tipo de cache ou enfileiramento de mensagens de infraestrutura (por exemplo, RabbitMQ, ZeroMQ) e digerido por um processo de banco de dados separado em lotes e escrita, ou computação intensiva de serviços de processamento de back-end, escrito em uma plataforma de melhor desempenho para tais tarefas. Comportamento semelhante pode ser implementado com outras linguagens/frameworks, mas não no mesmo hardware, com o mesmo alto rendimento mantido.

### Fluxo de Dados

Em plataformas mais tradicionais da web, solicitações e respostas HTTP são tratadas como eventos isolados; na verdade, eles são realmente streams. Esta observação pode ser utilizada no Node.js para construir alguns recursos interessantes. Por exemplo, é possível processar arquivos enquanto eles ainda estão sendo carregados, uma vez que os dados vêm no meio de um stream e podemos processá-los de forma online. Isso poderia ser feito para áudio em tempo real ou codificação de vídeo e proxy entre as diferentes fontes de dados.

### Proxy

O Node.js é facilmente utilizado como um proxy do lado do servidor, onde ele pode lidar com uma grande quantidade de conexões simultâneas de uma maneira non-blocking. É especialmente útil para proxys de diferentes serviços com diferentes tempos de resposta ou a coleta de dados a partir de vários pontos de origem.

Considere como exemplo um aplicativo que do lado do servidor se comunica com recursos de terceiros, puxando dados de diferentes fontes ou armazenando ativos como imagens e vídeos para serviços em nuvem de terceiros.

Embora existam servidores proxy dedicados, utilizando Node.js, ao invés, pode ser útil se a sua infraestrutura de proxy é inexistente ou se você precisa de uma solução para o desenvolvimento local. Por isso, você pode construir um aplicativo do lado do cliente com um servidor de desenvolvimento Node.js para ativos e solicitações de proxy/Stubbing API, enquanto em produção você lidaria com tais interações com um serviço dedicado de proxy (nginx, HAProxy, etc.).

## Painel de Monitoramento de Aplicações

Outro caso de uso comum, em que o Node com websockets se encaixa perfeitamente é o rastreamento de visitantes do site e visualização de suas interações em tempo real.

Você poderia reunir estatísticas em tempo real a partir de seu usuário, ou mesmo movê-lo para o próximo nível através da introdução de interações específicas com seus visitantes, abrindo um canal de comunicação quando chegam a um ponto específico no seu filtro.

Imagine como é possível melhorar um dado negócio, se for possível saber o que os visitantes estavam fazendo em tempo real, se for possível visualizar suas interações. Com o tempo real, nos dois sentidos bases de Node.js, agora é possível.

## Painel de Monitoramento de Sistemas

Agora, vamos visitar o lado da infraestrutura das coisas. Imagine, por exemplo, um provedor de SaaS que quer oferecer a seus usuários uma página de serviço de monitoramento (por exemplo, página de status do GitHub). Com o evento de circuito Node.js, podemos criar um poderoso painel baseado na web que verifica os estados dos serviços de forma assíncrona e envia dados para os clientes que usam websockets.

Tanto interna quanto os status dos serviços públicos podem ser relatados ao vivo e em tempo real, utilizando esta tecnologia. Indo um pouco além com essa ideia e tentando imaginar um Centro de Operações de Rede (NOC) aplicações de monitoramento em um operador de telecomunicações, nuvem/rede/provedor de hospedagem, ou alguma instituição financeira, todos executados na pilha web aberta apoiada pelo Node.js e websockets em vez de Java e/ou Java Applets, como foi visto por muito tempo.

## Hands on Node.js

Para configurar um ambiente Node.js, independente de qual sistema operacional, as dicas serão as mesmas. Somente alguns

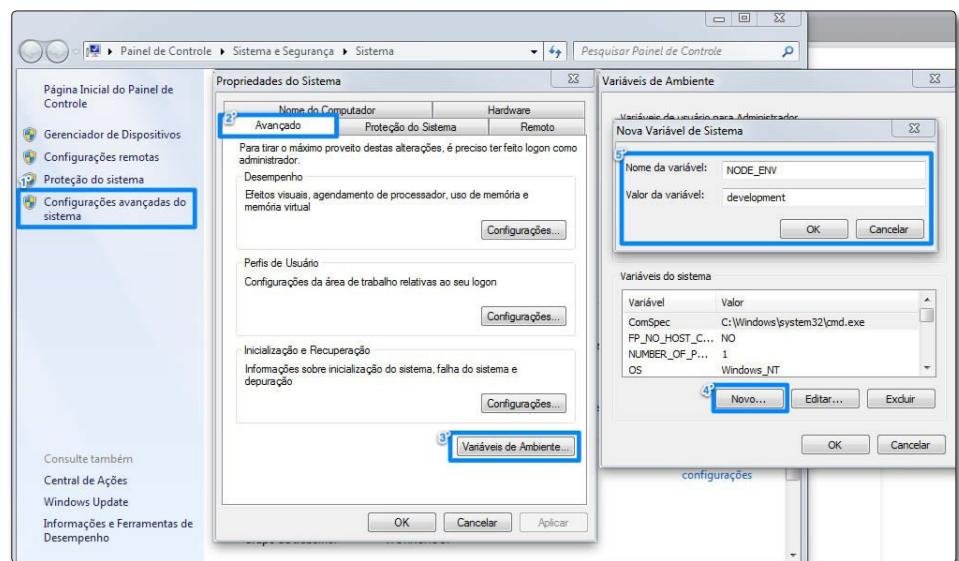


Figura 1. Configurando variável de ambiente no Windows 7

procedimentos serão diferentes para cada sistema, principalmente para o Windows, mas não será nada grave.

## Instalação

O primeiro passo é acessar seu site oficial (ver seção [Links](#)). Em seguida, clique em Install para baixar automaticamente a última versão compatível com seu sistema operacional (isto é se seu sistema é Windows ou MacOS).

Caso você use Linux recomendo que leia em detalhes a Wiki do repositório Node.js (ver na seção [Links](#)), pois lá é explicado as principais instruções sobre como instalá-lo através de um *package manager* de uma distribuição Linux.

### Nota

Todos os códigos aplicados neste artigo funcionarão apenas em versões do Node.js 0.8.X ou superior, exceto o último exemplo do artigo que utiliza a implementação de Generators do ECMAScript 6 que somente vai funcionar em uma versão unstable do Node.js 0.11.X.

Agora instale-o e caso não ocorra problemas, basta abrir seu terminal ou prompt de comandos e digite o seguinte comando para ver as respectivas versões do Node.js e NPM que foram instaladas:

```
node -v && npm -v
```

A última versão estável que está sendo utilizada neste artigo é **Node v0.10.31** e **NPM 1.4.23**.

## Configurando o ambiente de desenvolvimento

Para configurar o ambiente de desenvolvimento basta adicionar a variável de ambiente NODE\_ENV no sistema operacional.

# Programação assíncrona com Node.js

Em sistemas Linux ou MacOS, basta acessar com um editor de texto qualquer e em modo *super user (sudo)* o arquivo .bash\_profile ou .bashrc e no final do arquivo adicione a seguinte linha de comando:

```
export NODE_ENV='development'
```

Clique com botão direito no ícone **Meu Computador** e selecione a opção **Propriedades** e no lado esquerdo da janela clique no link **Configurações avançadas do sistema**. Na janela seguinte, acesse a aba **Avançado** e clique no botão **Variáveis de Ambiente....**. Agora no campo **Variáveis do sistema** clique no botão **Novo...** e no campo **nome da variável** digite *NODE\_ENV* e no campo **valor da variável** digite *development*, tal como demonstrado na **Figura 1**. Após finalizar essa tarefa, reinicie seu computador para carregar essa variável automaticamente no sistema operacional.

## Rodando o Node.js

Para testarmos o ambiente, executaremos o nosso primeiro programa de *Hello World*. Abra seu terminal ou prompt de comando e execute o comando *node*. Este comando vai acessar o modo REPL (*Read-Eval-Print-Loop*) que permite executar códigos JavaScript diretamente pela tela preta. Agora digite *console.log("Hello World")* e tecle **ENTER** para executá-lo na hora (**Figura 2**).



Figura 2. Hello World em Node.js via terminal

## Gerenciando dependências e módulos usando o NPM

Assim como o RubyGems do Ruby ou o Maven do Java, o Node.js também possui seu próprio gerenciador de pacotes, ele se chama NPM. Ele se tornou tão popular pela comunidade, que foi a partir da versão 0.6.X que foi integrado no instalador do Node.js, tornando-se o gerenciador padrão desta plataforma. Isto simplificou a vida dos desenvolvedores na época, pois fez com que diversos projetos se convergissem para esta plataforma até os dias de hoje. Utilizar o NPM é muito fácil, então vejamos os comandos principais para que você tenha noções de como usá-los:

- npm install nome-do-modulo: instala um módulo no projeto;
- npm install -g nome-do-modulo: instala um módulo global;
- npm install nome-do-modulo --save: instala o módulo no projeto, atualizando o package.json na lista de dependências;
- npm list: lista todos os módulos do projeto;
- npm list -g: lista todos os módulos globais;
- npm remove nome-do-modulo: desinstala um módulo do projeto;

- npm remove -g nome-do-modulo: desinstala um módulo global;
- npm update nome-do-modulo: atualiza a versão do módulo;
- npm update -g nome-do-modulo: atualiza a versão do módulo global;
- npm -v: exibe a versão atual do npm;
- npm adduser nome-do-usuario: cria uma conta no npm através do site do NPM (ver seção Links);
- npm whoami: exibe detalhes do seu perfil público npm (é necessário criar uma conta antes);
- npm publish: publica um módulo no site do npm (é necessário ter uma conta NPM antes).

## Entendendo o package.json

Todo projeto Node.js é chamado de módulo, mas o que é um módulo? No decorrer da leitura, perceba que vamos discutir muito sobre o termo módulo, biblioteca e framework, e, na prática, eles possuem o mesmo significado. O termo módulo surgiu do conceito de que o JavaScript trabalha com uma arquitetura modular. E todo módulo é acompanhado de um arquivo descritor, conhecido pelo nome de package.json.

Este arquivo é essencial para um projeto Node.js. Um package.json mal escrito pode causar bugs ou impedir o funcionamento correto do seu módulo, pois ele possui alguns atributos chaves que são compreendidos pelo Node.js e NPM.

Na **Listagem 1** temos um package.json que contém os principais atributos para descrever um módulo.

Listagem 1. Formato de um package.json.

```
{  
  "name": "meu-primeiro-node-app",  
  "description": "Meu primeiro app Node.js",  
  "author": "Caio <caio@email.com>",  
  "version": "1.2.3",  
  "private": true,  
  "dependencies": {  
    "modulo-1": "1.0.0",  
    "modulo-2": "~1.0.0",  
    "modulo-3": ">=1.0.0"  
  },  
  "devDependencies": {  
    "modulo-4": "*"  
  }  
}
```

Com esses atributos, você já descreve o mínimo possível o que será sua aplicação. O atributo **name** é o principal. Com ele, você descreve o nome do projeto, nome pelo qual seu módulo será chamado via função `require('meu-primeiro-node-app')`. Em **description**, descrevemos o que será este módulo. Ele deve ser escrito de forma curta e clara, fornecendo um resumo do módulo. O **author** é um atributo para informar o nome e e-mail do autor. Utilize o formato `Nome <email>` para que sites como npm reconheça corretamente esses dados. Outro atributo principal é o **version**, com o qual definimos a versão atual do módulo. É extremamente recomendado que tenha este atributo, se não

será impossível instalar o módulo via comando npm. O atributo private é um booleano, e determina se o projeto terá código aberto ou privado para download no npm.

Os módulos no Node.js trabalham com **três níveis de versionamento**. Por exemplo, a versão 1.2.3 está dividida nos níveis: *Major* (1), *Minor* (2) e *Patch* (3). Repare que no campo dependencies foram incluídos 4 módulos, sendo que cada um utilizou uma forma diferente de definir a versão do projeto. O primeiro, o modulo-1, somente será incluído sua versão fixa, a 1.0.0. Utilize este tipo versão para instalar dependências cuja atualizações possam quebrar o projeto pelo simples fato de que certas funcionalidades foram removidas e ainda as utilizamos na aplicação. O segundo módulo já possui uma certa flexibilidade de update. Ele utiliza o caractere “~” que faz atualizações a nível de *patch* (1.0.x). Geralmente essas atualizações são seguras, trazendo apenas melhorias ou correções de bugs. O modulo-3 atualiza versões que sejam maior ou igual a 1.0.0 em todos os níveis de versão. Em muitos casos, utilizar “>=” pode ser perigoso, porque a dependência pode ser atualizada a nível *major* ou *minor*, contendo grandes modificações que podem quebrar um sistema em produção, comprometendo seu funcionamento e exigindo que você atualize todo código até voltar ao normal. O último, o modulo-4, utiliza o caractere “\*”; este sempre pegará a última versão do módulo em qualquer nível. Ele também pode causar problemas nas atualizações e tem o mesmo comportamento do versionamento do modulo-3. Geralmente ele é utilizado em devDependencies, que são dependências focadas para testes ou de uso exclusivo para ambiente de desenvolvimento, e as atualizações dos módulos não prejudicam o comportamento do sistema que já está no ar.

## Escopos de variáveis locais e globais

Assim como no browser, utilizamos o mesmo JavaScript no Node.js. Ele também utiliza **escopos locais e globais** de variáveis. A única diferença é na forma como são implementados os escopos de variáveis globais. No browser, as variáveis globais são criadas da forma como pode ser vista na **Listagem 2**.

Em qualquer browser, a palavra-chave *window* permite criar variáveis globais que são acessadas em qualquer lugar. Já no Node.js, utilizamos a palavra-chave *global* para aplicar essa mesma técnica (**Listagem 3**).

### Listagem 2. Variáveis globais no client-side.

```
window.hoje = new Date();
alert(window.hoje);
```

### Listagem 3. Variáveis globais no server-side do Node.js.

```
global.hoje = new Date();
console.log(global.hoje);
```

Ao utilizar *global* mantemos uma variável global, acessível em qualquer parte do projeto sem a necessidade de chamá-la via *require* ou passá-la por parâmetro em uma função.

Esse conceito de variável global é existente na maioria das linguagens de programação, assim como sua utilização, portanto é recomendado trabalhar com o mínimo possível de variáveis globais para evitar futuros gargalos de memória na aplicação.

## CommonJS

O Node.js utiliza nativamente o padrão *CommonJS* para organização e carregamento de módulos. Na prática, diversas funções deste padrão serão utilizadas com frequência em um projeto Node.js. A função `require('nome-do-modulo')` é um exemplo disso, ela carrega um módulo. E para criar um código modular, carregável pela função `require`, utilizam-se as variáveis globais: `exports` ou `module.exports`. Veja nas listagens a seguir dois exemplos de módulos para Node.js. Primeiro, crie o código `hello.js` (**Listagem 4**) e depois crie o código `human.js` (**Listagem 5**).

### Listagem 4. Criando um módulo com `module.exports`.

```
module.exports = function(msg) {
  console.log(msg);
};
```

### Listagem 5. Criando um módulo com `exports`.

```
exports.hello = function(msg) {
  console.log(msg);
};
```

A diferença entre o `hello.js` e o `human.js` está na maneira como eles serão carregados. Em `hello.js` carregamos uma única função modular, e em `human.js` é carregado um objeto com funções modulares. Essa é a grande diferença entre eles. Para entender melhor na prática, crie o código `app.js` para carregar esses módulos:

```
var hello = require('./hello');
var human = require('./human');
hello('Olá pessoal!');
human.hello('Olá galera!');
```

Tenha certeza de que esses módulos `hello.js`, `human.js` e `app.js` estão na mesma pasta, e em seguida, rode o comando:

```
node app.js
```

E então, o que aconteceu? O resultado foi praticamente o mesmo: o `app.js` carregou os módulos `hello.js` e `human.js` via função `require()`, em seguida foi executada a função `hello("Olá pessoal!")` que imprimiu a mensagem "Olá pessoal!" e, por último, o objeto `human` executou sua função `human.hello("Olá galera!")`.

Um detalhe final é que também é possível criar um objeto com função modular usando `module.exports`, apenas faça com que o módulo retorne um objeto com funções públicas, semelhante ao código da **Listagem 6**.

E assim você terá um módulo com mesmo comportamento do `human.hello("olá")`.

# Programação assíncrona com Node.js

**Listagem 6.** Emulando o comportamento das funções exports usando module.exports.

```
module.exports = function() {
  return {
    hello: function(msg) {
      console.log(msg);
    }
  };
};
```

## Programando de forma assíncrona

Agora vamos para a última parte do nosso artigo. Agora que já temos uma base sobre o que é e como usar o Node.js, vamos focar em aprender boas práticas sobre funções assíncronas. Afinal este é o paradigma principal desta plataforma, e é muito importante dominar esses conceitos para se tirar melhor proveito desta tecnologia, assim como entender como funciona funções assíncronas. O código a seguir exemplifica as diferenças entre uma função síncrona e assíncrona em relação à linha do tempo na qual elas são executadas. Basicamente, criaremos um loop de cinco iterações, sendo que a cada iteração será criado um arquivo texto com o mesmo conteúdo "Hello Node.js!". Primeiro vamos começar com o código síncrono. Crie o arquivo text\_sync.js com o código da **Listagem 7**.

Agora vamos criar o arquivo text\_async.js, com seu respectivo código, diferente apenas na forma de chamar a função fs.writeFileSync, que será a versão assíncrona fs.writeFile, pelo qual seu retorno de sucesso acontece através da execução de uma função de callback existente no terceiro argumento da função (**Listagem 8**).

**Listagem 7.** Escrevendo arquivo de texto de forma síncrona.

```
var fs = require('fs');
for(var i = 1; i <= 5; i++) {
  var file = "sync-txt" + i + ".txt";
  fs.writeFileSync(file, "Hello Node.js!");
  console.log("Criando arquivo:" + file);
}
```

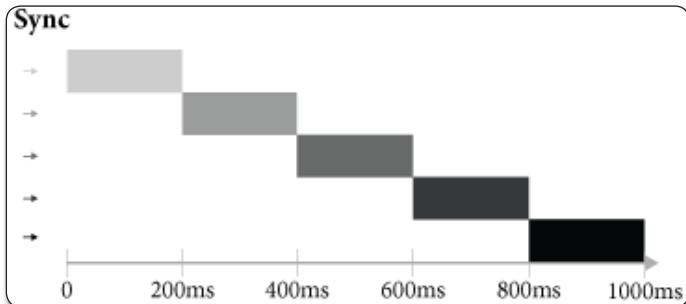
**Listagem 8.** Escrevendo arquivo de texto de forma assíncrona.

```
var fs = require('fs');
for(var i = 1; i <= 5; i++) {
  var file = "async-txt" + i + ".txt";
  fs.writeFile(file, "Hello Node.js!", function(err, out) {
    console.log("Criando arquivo:" + file);
  });
}
```

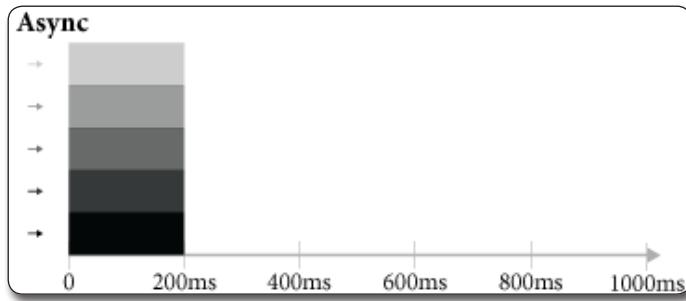
Execute os comandos node text\_sync.js e depois node text\_async.js. Se forem gerados 10 arquivos no mesmo diretório do código-fonte, então deu tudo certo. Mas a execução de ambos foi tão rápida que não foi possível visualizar as diferenças entre o text\_async.js e o text\_sync.js. Para entender melhor as diferenças, veja os gráficos hipotéticos a seguir que ocorrem quando executamos esses

módulos. O text\_sync.js, por ser um código síncrono, invocou chamadas de **I/O bloqueantes** e gerou o gráfico da **Figura 3**.

Repare no tempo de execução, o text\_sync.js hipoteticamente demorou 1000 milissegundos, isto é, 200 milissegundos para cada escrita de arquivo. Já em text\_async.js foram criados os arquivos de forma totalmente assíncrona, ou seja, as chamadas de I/O eram não-bloqueantes, e isso permitiu escrever arquivos em paralelo, como na **Figura 4**.



**Figura 3.** Timeline de execução síncrona



**Figura 4.** Timeline de execução assíncrona

Isto fez com que o tempo de execução levasse hipoteticamente 200 milissegundos, afinal foi invocada cinco vezes, em paralelo, a função fs.writeFile, e isso maximizou o uso de processamento e I/O e minimizou o tempo de execução. Só para finalizar esse gráfico é apenas hipotético, e foi usado para exemplificar como funciona o assincronismo do Node.js, nem sempre um conjunto de funções assíncronas vão executar em paralelo de forma tão perfeita como foi apresentado nesses gráficos.

## Threads vs Assincronismo

Por mais que as funções assíncronas possam executar em paralelo várias tarefas, elas jamais serão consideradas uma Thread (por exemplo as Threads do Java). A diferença é que Threads são manipuláveis pelo desenvolvedor, ou seja, você pode pausar a execução de uma Thread ou fazê-la esperar o término de uma outra Thread para ser executada. Chamadas assíncronas apenas invocam suas funções em e você não controla elas, apenas trabalha com seus retornos através de uma função callback.

Pode parecer vantajoso ter o controle sobre a execução de Threads a favor de um sistema que executa tarefas em paralelo, mas pouco conhecimento sobre eles pode transformar seu sis-

tema em um caos de travamentos de dead-locks, afinal elas são executadas de forma bloqueante. Este é o grande diferencial das chamadas assíncronas, elas executam em paralelo suas funções sem travar processamento das outras e, principalmente, sem bloquear a aplicação.

É fundamental que o seu código Node.js invoque o mínimo possível de funções bloqueantes. Toda função síncrona impedirá, naquele instante, que o Node.js continue executando os demais códigos até que aquela função seja finalizada. Por exemplo, se essa função fizer um I/O em disco, ele vai bloquear o sistema inteiro, deixando o processador ocioso enquanto ele utiliza outros recursos do servidor, como por exemplo leitura em disco, utilização da rede etc. Sempre que puder, utilize funções assíncronas para aproveitar essa característica principal do Node.js.

## Assincronismo versus Síncronismo

Caso você ainda não esteja convencido sobre as vantagens do processamento assíncrono vou lhe mostrar como e quando utilizar bibliotecas assíncronas não-bloqueantes através de um teste prático.

Para exemplificar melhor, os códigos adiante representam um benchmark comparando o tempo de bloqueio de execução **assíncrona vs síncrona**. Para isso, crie três arquivos: processamento.js, leitura\_async.js e leitura\_sync.js. Criaremos isoladamente o módulo leitura\_async.js que será responsável por fazer leitura assíncrona de arquivo grande, tal como na [Listagem 9](#).

Em seguida, crie o módulo leitura\_sync.js, que realizará leitura síncrona no mesmo arquivo ([Listagem 10](#)).

### Listagem 9. Código de benchmark de tempo de bloqueio em leitura assíncrona.

```
var fs = require('fs');
var leituraAsync = function(arquivo){
  console.log("Fazendo leitura assíncrona");
  var inicio = new Date().getTime();
  fs.readFile(arquivo);
  var fim = new Date().getTime();
  console.log("Bloqueio assíncrono: "+(fim - inicio)+"ms");
};
module.exports = leituraAsync;
```

### Listagem 10. Código de benchmark de tempo de bloqueio em leitura síncrona.

```
var fs = require('fs');
var leituraSync = function(arquivo){
  console.log("Fazendo leitura síncrona");
  var inicio = new Date().getTime();
  fs.readFileSync(arquivo);
  var fim = new Date().getTime();
  console.log("Bloqueio síncrono: "+(fim - inicio)+"ms");
};
module.exports = leituraSync;
```

Para finalizar, vamos carregar esses módulos dentro do código processamento.js. Basicamente este módulo principal vai fazer download da última versão do instalador Node.js que tem em média 7 MB de tamanho. Quando o download terminar ele vai enviar o arquivo para os módulos realizarem uma leitura de

conteúdo, no término de cada leitura será apresentado o tempo de bloqueio que cada módulo obteve, como observado através da [Listagem 11](#).

### Listagem 11. Código de execução do benchmark síncrono vs assíncrono.

```
var http = require('http');
var fs = require('fs');
var leituraAsync = require('./leitura_async');
var leituraSync = require('./leitura_sync');
var arquivo = "./node.exe";
var stream = fs.createWriteStream(arquivo);
var download = "http://nodejs.org/dist/latest/x64/node.exe";
http.get(download, function(res) {
  console.log("Fazendo download do Node.js");
  res.on('end', function(){
    console.log("Download finalizado!");
    console.log("Executando benchmark sync vs async...");
    leituraAsync(arquivo);
    leituraSync(arquivo);
  });
});
```

Rode o comando node processamento.js para executar o benchmark. E agora, ficou clara a diferença entre o modelo bloqueante e o não-bloqueante? Parece que o módulo leituraAsync executou muito rápido, mas não quer dizer que o arquivo foi lido. Ele recebe um último parâmetro, que é um callback indicando quando o arquivo foi lido, que não passamos na invocação que fizemos.

Ao usar o fs.readFileSync(), bastaria fazer var conteúdo = fs.readFileSync(). Mas qual é o problema dessa abordagem? **Ela bloqueia todo o processamento da aplicação!** Somente quando o bloqueio terminar as demais linhas de código serão executadas, em contra partida o fs.readFile() continua executando qualquer código que não estiver dentro de seu callback, então com isso você pode, por exemplo, programar sua aplicação para fazer outras tarefas em paralelo.

A seguir, temos o resultado do benchmark realizado em na máquina que possui as seguintes configurações:

- MacBook Air 2011;
- Processador: Core i5 1.6GHz;
- Memória: 4GB DDR 3;
- Disco: 128GB SSD.

Veja a pequena, porém significante, diferença de tempo entre as duas funções de leitura realizada na máquina ([Figura 5](#)).

```
[vagrant:~] $ node processamento.js
Fazendo download do Node.js
Download finalizado!
Fazendo leitura assíncrona
Bloqueio assíncrono: 1ms
Fazendo leitura síncrona
Bloqueio síncrono: 8ms
[vagrant:~] $ ]
```

**Figura 5.** Resultado do benchmark I/O Async vs I/O Sync

## Evitando Callback Hell

De fato, vimos o quanto é vantajoso e performático trabalhar de forma assíncrona, porém, em certos momentos, inevitavelmente implementaremos diversas funções assíncronas, que serão encadeadas uma na outra através de suas funções callback. No código da **Listagem 12** vemos um exemplo desse caso.

**Listagem 12.** Exemplo de Callback Hell.

```
var fs = require('fs');
fs.readdir(__dirname, function(error, contents) {
  if (error) { throw error; }
  contents.forEach(function(content) {
    var path = __dirname + content;
    fs.stat(path, function(error, stat) {
      if (error) { throw error; }
      if (stat.isFile()) {
        console.log('%s %d bytes', content, stat.size);
      }
    });
  });
});
```

Reparam na quantidade de callbacks encadeados que existem em nosso código. **Detalhe:** ele apenas faz uma simples leitura dos arquivos de seu diretório e imprime na tela seu nome e tamanho em bytes. Uma pequena tarefa como essa deveria ter menos encadeamentos, concorda? Agora, imagine como seria a organização disso para realizar tarefas mais complexas? Praticamente o seu código seria um caos e totalmente difícil de fazer manutenções.

Por ser assíncrono, você perde o controle do que está executando em troca de ganhos com performance, porém, um detalhe importante sobre assincronismo é que, na maioria dos casos, os callbacks bem elaborados possuem como parâmetro uma variável de erro. Verifique nas documentações sobre sua existência e sempre faça o tratamento deles na execução do seu callback: `if (erro) { throw erro; }`. Isso vai impedir a continuação da execução aleatória quando for identificado um erro.

Uma boa prática de código JavaScript é criar funções que expressem seu objetivo e de forma isoladas, salvando em variável e passando-as como callback. Ao invés de criar funções anônimas, por exemplo, crie um arquivo chamado `callback_heaven.js` com o código da **Listagem 13**.

Veja o quanto melhorou a legibilidade do seu código. Dessa forma deixamos mais semântico e legível o nome das funções e diminuímos o número de encadeamentos das funções de callback. A boa prática é ter o bom senso de manter no máximo até dois encadeamentos de callbacks. Ao passar disso, significa que está na hora de criar uma função externa para ser passada como parâmetro nos callbacks, em vez de continuar criando um *callback hell* em seu código.

## Evitando Callbacks Hell usando Generators

Muitos recursos interessantes estão surgindo para a nova especificação JavaScript conhecida por ECMAScript6 ou ES6, o **Generators** é um deles e seu objetivo principal é minimizar callback hell em

seu código. Em resumo **Generators** é um recurso que permite escrever funções assíncronas sem callbacks, utilizando uma sintaxe de código síncrono, retornando valores da função em um array que representa os possíveis parâmetros de uma função callback.

**Listagem 13.** Minimizando callback hell declarando funções em variáveis.

```
var fs = require('fs');
var lerDiretorio = function() {
  fs.readdir(__dirname, function(error, diretorio) {
    if (error) return error;
    diretorio.forEach(function(arquivo) {
      ler(arquivo);
    });
  });
}
var ler = function(arquivo) {
  var path = __dirname + arquivo;
  fs.stat(path, function(error, stat) {
    if (error) return error;
    if (stat.isFile()) {
      console.log('%s %d bytes', arquivo, stat.size);
    }
  });
}
lerDiretorio();
```

Como esse recurso ainda não é oficial, somente alguns browsers (últimas versões do Chrome e Firefox) o utilizam no client-side. Já no server-side temos que habilitar no Node.js, porém somente existe para as versões instáveis: **0.11.X** e possivelmente será oficializada na próxima versão estável: **0.12.x**.

### Observação

Para utilizar uma versão 0.11.X recomendo que faça download e instalação das versões Nightlies do Node.js (ver na seção [Links](#)). Lembrando que não recomendado utilizar uma versão instável em uma aplicação em produção.

Com a versão **0.11.X** instalada em sua máquina, basta executar suas aplicações utilizando a flag `--harmony`, por exemplo:

```
node --harmony app.js
```

Com `harmony` habilitado, podemos usar alguns recursos do ES6, incluindo o **Generators**. Porém será necessário uma instalar uma biblioteca adicional que permite trabalhar com **Generators** e também faz algumas magias extras. Para isso instale o módulo `suspend`. Antes de criarmos os códigos de Callback e Generators, instale o módulo `suspend` utilizando o seguinte código:

```
npm install suspend --save
```

Agora vamos a implementação. A seguir temos dois códigos que fazem a mesma tarefa: ambos criam um arquivo de texto, escreve nele um timestamp e, no fim, excluir o próprio arquivo gerado. O código da **Listagem 14** utiliza vários call-backs.

Na **Listagem 15** temos a segunda parte do código, que é uma versão otimizada que implementa Generators para lidar com os call-backs.

#### **Listagem 14.** Outro exemplo de callback hell.

```
var fs = require("fs");
var time = new Date().getTime();
fs.writeFile("log.txt", time, function(err) {
  console.log("Iniciando log");
  fs.readFile("log.txt", function(err, text) {
    console.log("Timestamp:" + text);
    fs.unlink("log.txt", function() {
      console.log("Log finalizado");
    });
  });
});
```

#### **Listagem 15.** Implementando Generators para minimizar callback hell.

```
var fs = require('fs');
var suspend = require('suspend');
var resume = suspend.resume;
var time = new Date().getTime();
suspend(function* () {
  yield fs.writeFile("log.txt", time, resume());
  console.log("Iniciando log");
  var text = yield fs.readFile("log.txt", resume());
  console.log("Timestamp" + text);
  yield fs.unlink("log.txt", resume());
  console.log("Log finalizado");
})();
```

Simplesmente o encadeamento de callbacks diminuiu com **Generators**, e isso deixou seu código mais limpo e menos complexo.

De fato o Node.js é uma excelente opção para desenvolvedores front-end conhecerem um pouco sobre back-end sem precisar aprender uma nova linguagem, afinal esta plataforma o mesmo JavaScript client-side, apenas com alguns detalhes diferentes. Outro detalhe importante é sobre as vantagens das funções assíncronas e seu I/O não-bloqueante. Afinal como vimos em um benchmark, testamos uma ação de I/O simples que fazia uma leitura de um único arquivo de mais ou menos 7 MB e o tempo de bloqueio foi muito menor do que uma leitura bloqueante.

Agora, imagine esta mesma ação em larga escala, lendo múltiplos arquivos de 1 GB ao mesmo tempo ou realizando múltiplos I/Os em seu servidor para milhares de usuários, tudo isso sem bloquear a aplicação. Esse é um dos pontos fortes do Node.js!

## Autor



### Caio Ribeiro Pereira

[caio.ribeiro.pereira@gmail.com](mailto:caio.ribeiro.pereira@gmail.com)

É um Web Developer, com experiência nessa sopa de letreiras: Node.js, Meteor, JavaScript, CSS, Ruby, Java, MongoDB, Redis, LevelDB, Agile, XP e TDD. Bacharel em Sistemas de Informação, mantém ativamente o blog [Underground WebDev](#) abordando assuntos sobre Node.js, JavaScript, apaixonado por programação, tecnologias e seriados. Moderador do Meteor Brasil e participante das comunidades NodeBR e DevInSantos. Desde 2011 palestra nos eventos DevInSantos e Exatec, abordando temas atuais sobre Node.js e JavaScript. Autor dos livros da Casa do Código: Aplicações web real-time com Node.js e Meteor - Criando aplicações web real-time com JavaScript.



## Links:

### **Blog Underground WebDev**

[www.udgwebdev.com/](http://www.udgwebdev.com/)

### **Ruby EventMachine**

[www.rubyeventmachine.com/](http://www.rubyeventmachine.com/)

### **Python Twisted**

[www.twistedmatrix.com/](http://www.twistedmatrix.com/)

### **Google Groups NodeBR**

[www.groups.google.com/forum/#!forum/nodebr](http://www.groups.google.com/forum/#!forum/nodebr)

### **Site oficial do Node.js**

[www.nodejs.org/](http://www.nodejs.org/)

### **Wiki de instalação do Node.js**

[www.github.com/joyent/node/wiki/Installing-Node.js-via-package-manager/](http://www.github.com/joyent/node/wiki/Installing-Node.js-via-package-manager/)

### **Site oficial do NPM**

[www.npmjs.org/](http://www.npmjs.org/)

# Testes unitários em JavaScript: Introdução - Parte 1

Conheça o que há de mais recente e em evidência no mercado de testes unitários para o cliente side

ESTE ARTIGO FAZ PARTE DE UM CURSO

**N**o advento das tecnologias front-end e da utilização de novos padrões de desenvolvimento, como a HTML5, novas fórmulas e conceitos foram criados de forma a suprir a cada vez maior necessidade de atender a todos os diversos cenários de software.

O uso de testes unitários para verificar se um módulo ou unidade do código está funcionando corretamente como esperado já não constitui nenhuma novidade. Afinal, tal conceito foi outrora introduzido na comunidade de desenvolvedores por Kent Beck, através de linguagens mais antigas como Smalltalk, C, C++ e até mesmo no Java. Porém, se analisarmos bem o termo é sempre extremamente associado às linguagens que se constituem “linguagens server side”, isto é, executam apenas do lado do servidor, muitas com processos bem definidos de compilação, interpretação, plataforma integrada ou multiplataforma, etc. Até mesmo uma rápida busca na web pelo termo irá remeter em inúmeras bibliografias, blogs e sites sobre o assunto com foco voltado para esse tipo de linguagem.

É interessante notar que até mesmo o conceito foi desenvolvido e evoluído sobre esse tipo de linguagem. Diante disso, este artigo visa construir uma ideia adaptada dos testes unitários para o universo front-end, especificamente focado no uso e construção dos exemplos em cima da linguagem de script mais famosa do mercado: JavaScript. Logo, aqui abordaremos os tópicos

## Fique por dentro

Nenhuma categoria do ciclo de vida de software está em tanta evidência quanto os testes unitários. Ao mesmo passo, o advento das tecnologias front-end, aliado aos processos e boas práticas que cada vez mais se fazem necessários de serem aplicados a esse meio, trazem à tona a real necessidade de unir ambos universos: testes de unidade ao desenvolvimento client side.

Este artigo explicita isso trazendo a primeira de duas partes sobre o desenvolvimento de testes unitários usando JavaScript. Veremos todos os conceitos, práticas e metodologias que norteiam o que há de mais recente no assunto, assim como modelos básicos de como eles se aplicam ao uso do JavaScript e seus frameworks.

mais famosos dos testes, desde os conceitos, teorias e boas práticas, até a escolha das ferramentas adequadas para tal.

### Testes unitários

Antes de entender quais divisões ou aplicações de testes diversas existem no universo de programação, é importante antes entender o que é, de fato, um teste unitário. O próprio nome já consegue nos dizer muito acerca desse procedimento: um teste ou mais testes que verificam unidades de composição de, no nosso caso, sistemas diversificados.

Se procurarmos mais formalmente, veremos algumas definições mais elaboradas: “Em programação de computadores, teste unitário ou teste de unidade constitui um método de teste de software pelo qual unidades individuais de código fonte, configuradas com um ou mais módulos de programas de computador juntos com dados de controle associados são testados para determinar se eles estão aptos para o uso final.”

Se pensarmos programaticamente uma unidade de código, e consequentemente de software, deve constituir o menor pedaço testável de uma aplicação. Em linguagens orientadas a objetos isso pode representar um método, ou até mesmo um bloco menor de código fonte. Porém, quando partimos para o paralelo das linguagens procedurais essa mesma unidade pode significar o módulo inteiro programado, mesmo sendo comuns as divisões por módulos menores ou funções e procedures individuais.

Em resumo, os testes unitários nos permitem adentrar métodos e classes para verificar a consistência da implementação dos mesmos. E isso significa que uma bateria de testes bem elaborada traz consigo uma carga de segurança alta em relação ao código implementado, uma vez que temos agora uma camada que seguramente avalia todas as unidades de código quando da refatoração ou alteração dos mesmos. Em outras palavras, é possível introduzir novas partes ao código existente (um método, por exemplo) e poder verificar sua consistência de igual forma.

Alguns desenvolvedores, no entanto, consideram que a prática de escrever “código para testar código” constitui um desperdício de tempo, tempo este que poderia ser usada para focar nas próximas iterações de desenvolvimento, bugs, ou quaisquer atividades associadas. Porém, quando se trata do desenvolvimento de aplicações, com várias pessoas trabalhando na mesma equipe (muitas delas alterando os mesmos pontos de código), com várias iterações, versões do código e dependendo da complexidade do sistema, os testes unitários vêm para, na verdade, salvar tempo. Basta considerar o mapeamento de erros que te deixaria rápida e seguramente atualizar o código fonte, por exemplo.

A ideia de testar partes do código como um todo não é recente, data de meados de 1970, quando Kent Beck, um dos responsáveis também pela metodologia de desenvolvimento ágil XP (Extreme Programming), idealizou o conceito baseado na necessidade de mais produtividade quando da aplicação de mudanças ao software durante o seu ciclo de vida. A consolidação, porém, só veio quando da criação dos frameworks xUnit, que permitiam a verificação de partes separadas do código-fonte com os mesmos fins descritos até então.

Como o termo até então remetia apenas à linguagem Smalltalk, e com a popularização crescente do Java, tornou-se necessário então a criação de um framework para a linguagem, daí nasceu o famoso JUnit. Apesar de ser uma linguagem server side (no conceito de aplicações cliente-servidor), o Java e o JUnit tiveram papel de extrema importância para a popularização e aceitação dos testes unitários por parte dos desenvolvedores de uma forma geral. A partir dessa linguagem, foi que tivemos uma comunidade ativa e aberta à discussão do assunto, além de contribuir para a evolução do conceito e respectivas implementações.

## Design do teste

Existem uma série de propriedades a serem consideradas quando da criação e utilização de testes unitários:

- Ele deve ser automatizado e permitir repetição;
- Ele deve ser fácil de implementar;

- Uma vez escrito, ele deve ser mantido para usos futuros;
- Qualquer um deve estar apto a executá-lo;
- Ele deve executar através do clique de um botão;
- Ele deve executar rapidamente.

Com um teste unitário, o sistema sobre o teste (SUT) deve ser pequeno e apenas relevante para os desenvolvedores que trabalham proximamente ao código. Um teste unitário deve considerar operações do tipo: código lógico, código que contém muitos branches, cálculos or algo que de alguma forma requeira algum tipo de decisão. Simples métodos getters e setters são exemplos de código não lógico.

Quando um software é desenvolvido sob a perspectiva do teste unitário, a combinação de escrever um teste unitário para especificar a interface mais o refactoring das atividades depois que o teste passa constituem o design formal do teste. Cada teste unitário pode ser visto como um elemento de design especificando classes, métodos e comportamentos observáveis. O teste exemplificado na **Listagem 1** ilustra bem essa ideia.

Na mesma listagem é possível observar um conjunto de casos de teste que especificam um número de elementos da implementação. A interface Soma é responsável por manter o contrato de assinaturas de acordo com a sua classe de implementação, SomaImpl. A implementação verifica (assert) o comportamento de diferentes chamadas (situações, casos) ao método add().

Neste caso, os testes unitários, tendo sido escritos antes, atuam como um documento de design especificando a forma e comportamento de uma determinada solução, mas não detalhes de implementação, que serão deixados para o desenvolvedor. Seguindo o paradigma “Faça a mais simples coisa que possa possivelmente funcionar”, a solução mais fácil que irá fazer com que o teste passe está na classe SomaImpl e interface Soma.

Ao contrário de outros métodos de design, usar testes unitários como uma especificação de design tem uma vantagem significativa: o documento de design pode ser usado para verificar se a implementação “adere” ao design. Com esse método de design os testes nunca passarão se os desenvolvedores não implementarem a solução de acordo com o design.

## Tipos de teste

É extremamente importante saber a diferença entre os tipos de teste existentes, bem como suas nomenclaturas e conceitos. Existem três divisões principais de testes: o teste de unidade (que tratamos aqui), de integração e de sistema. Vejamos as diferenças do teste de unidade para com as outras:

- O **teste de integração**, como o próprio nome já diz, é o teste responsável por averiguar a integração entre duas partes do seu sistema. Um dos exemplos mais clássicos desse tipo de teste é representado pelo padrão de projetos DAO, que faz acesso à base de dados para persistir as informações manipuladas pela linguagem de programação. Ao desenvolver testes para uma classe desse tipo você automaticamente estará criando um meio de verificar se a “integração” entre ambos, código e banco de dados, existe

e está conforme. Existem vários outros tipos de integrações que podem ser verificadas via testes de integração, tais como código que se comunica com web services, que efetuam operações de escrita e leitura sobre arquivos, que verificam a comunicação assíncrona da implementação de filas e JMS, ou até mesmo que verificam o funcionamento de mensagens enviadas via sockets, por protocolos HTTP, etc.

• O teste de sistema é uma expressão usada para designar se o funcionamento do sistema como um todo está procedendo. O foco, portanto, deixar de ser a unidade (teste unitário) assim como a integração (teste de integração) e passa agora a focar na junção de todas as pequenas partes do sistema constituindo o conjunto completo de tarefas a serem executadas pelo sistema. Também chamado de “teste de caixa preta”, esse tipo de teste lida com a averiguação de tudo, desde o banco de dados, métodos e classes até as integrações. Para algumas vertentes, os testes de aceitação, que ficaram famosos com o advento do desenvolvimento ágil e são caracterizados pela aceitação ou não do time ágil, são no final testes de sistema.

Diane dessa situação, muitos desenvolvedores se perguntam qual tipo de teste escolher e/ou como associar cada nível de teste a uma situação em específico no ciclo de desenvolvimento do mundo real. Na verdade, não existe bala de prata, é preciso ter o discernimento de associar cada situação a um tipo de teste específico. Por exemplo, as classes responsáveis por lidar com o negócio da aplicação geralmente podem ser testadas em unidade, de uma maneira isolada, enquanto classes que se comunicam com web services necessitam de um teste de integração aplicado,

dada a necessidade do outro ambiente para tal teste. Além disso, vale frisar a importância do teste bem feito, o que quase sempre significará ter testes para ambas as situações onde o teste funciona e quebra.

## Tipos de processos

Além de analisar os tipos de testes que existem, é importante também saber a diferença entre estes e os diferentes tipos de processos de teste que existem e que envolvem todo esse ciclo. Essas famosas “práticas de desenvolvimento” se dividem em três nomes tão famosos quanto: TDD - Test-Driven Development, BDD - Behavior-driven Design e DDD - Domain-driven Design (Figura 1).

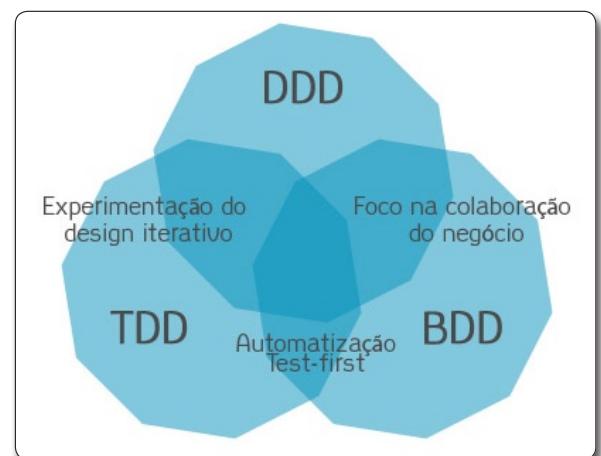


Figura 1. Três tipos de processos de desenvolvimento usando testes

Listagem 1. Exemplo de teste unitário em Java

```
public class AdicaoTeste {  
  
    // pode somar números positivos 1 e 1?  
    public void testeSomaNumeroPositivoUmEUm() {  
        Soma soma = new Somimpl();  
        assert(soma.add(1, 1) == 2);  
    }  
  
    // pode somar os números positivos 1 e 2?  
    public void testeSomaNumeroPositivoUmEDois() {  
        Soma soma = new Somimpl();  
        assert(soma.add(1, 2) == 3);  
    }  
  
    // pode somar os números positivos 2 e 2?  
    public void testeSomaNumeroPositivoDoisEDois() {  
        Soma soma = new Somimpl();  
        assert(soma.add(2, 2) == 4);  
    }  
  
    // O zero é neutro?  
    public void testeSomaZeroNeutro() {  
        Soma soma = new Somimpl();  
        assert(soma.add(0, 0) == 0);  
    }  
  
    // pode somar os números negativos -1 e -2?  
}
```

```
public void testeSomaNumerosNegativos() {  
    Soma soma = new Somimpl();  
    assert(soma.add(-1, -2) == -3);  
}  
  
// pode somar um número positivo e um negativo?  
public void testeSomaPositivoENegativo() {  
    Soma soma = new Somimpl();  
    assert(soma.add(-1, 1) == 0);  
}  
  
// E quanto aos números grandes?  
public void testeSomaNumerosGrandes() {  
    Soma soma = new Somimpl();  
    assert(soma.add(1234, 988) == 2222);  
}  
  
interface Soma {  
    int add(int a, int b);  
}  
class Somimpl implements Soma {  
    int add(int a, int b) {  
        return a + b;  
    }  
}
```

## TDD

O TDD, Test-Driven Development (ou Desenvolvimento Orientado a Testes) é uma metodologia famosa que cresceu junto com o desenvolvimento ágil e que foca no lema “testar primeiro, desenvolver depois”. Em outras palavras, é uma técnica que visa iterações curtas e que propõe que um dado desenvolvedor implemente primeiramente o caso de teste específico para uma situação de construção de funcionalidade, correção de bug, etc., e em seguida, e somente em seguida, crie o código fonte que atenderá e será validado pelo caso de teste feito anteriormente.

Ele ganhou mais atenção nos anos posteriores como diferentes metodologias junto ao processo de software que vinha surgindo. Dissecando o nome, desenvolvimento sugere um processo de pleno direito com a análise, projeto lógico e físico, implementação, teste, análise, integração e implantação, e orientado a testes implica o quanto concreto os testes automatizados devem conduzir o processo de desenvolvimento. O TDD também é comumente chamado de programação test-first.

Quando se houve falar sobre essa terminologia, principalmente associada a uma explicação do seu conceito, a ideia é aparentemente simples. Apenas dando uma breve olhada na própria palavra é possível ver que TDD remete a ter testes que dirigem o desenvolvimento do software.

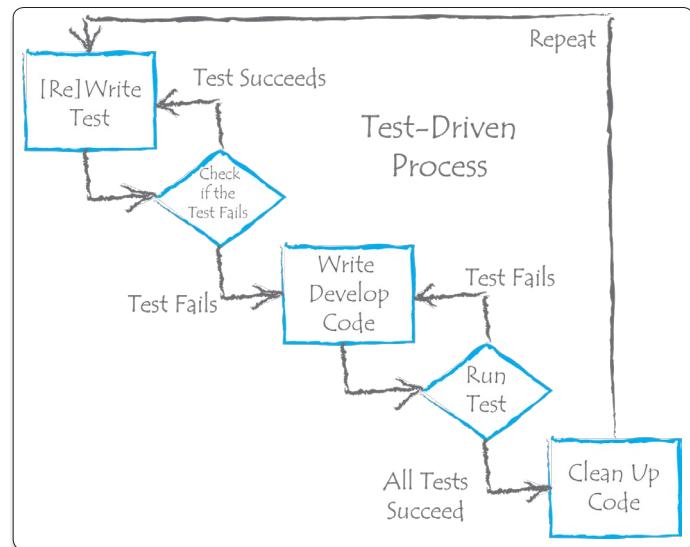
Se adentramos um pouco além na definição desse conceito, nos deparamos com cinco diferentes estágios:

1. Antes de qualquer coisa, o desenvolvedor escreve alguns testes;
2. O desenvolvedor então executa alguns destes testes e (obviamente) eles irão falhar porque nenhuma destas features estão de fato implementadas;
3. Em seguida, o desenvolvedor na verdade implementa todos os testes feitos antes, agora em código fonte da linguagem escondida;
4. Se o desenvolvedor escreve o seu código bem, então no próximo estágio ele irá ver seus testes serem executados com sucesso;
5. O desenvolvedor pode agora refatorar seu código, adicionar comentários, limpar o projeto de uma forma geral, como ele deseja porque o desenvolvedor sabe que se o código novo quebrar em algum local, então os testes irão alertá-lo sobre a falha.

O ciclo pode somente continuar se o desenvolvedor tem mais features a adicionar. Veja na **Figura 2** a representação desse fluxo.

Vejamos um exemplo então de como um desenvolvedor deveria fazer isso. Digamos que um desenvolvedor quer escrever uma função que faz algo bem simples, como calcular um fatorial (obviamente um exemplo bem simples, mas isso será o suficiente para descrever como o comportamento TDD deva ser). A abordagem normal para TDD indica usar a função e então o assert para que o resultado satisfaça um determinado valor.

Os testes deverão ser parecidos com o código demonstrado na **Listagem 2**, totalmente desenvolvido sobre o framework JavaScript Mocha.



**Figura 2.** Fluxo de execução do TDD

Os testes com certeza falharão porque a função ainda não foi escrita, mas é justamente essa a intenção: verificar como o comportamento do código deveria funcionar, mesmo não tendo um código ainda para testes. Agora que já entendemos a abordagem nua e crua, criemos o código da função que irá satisfazer os testes. Ele deverá se parecer com o código representado pela **Listagem 3**.

**Listagem 2.** Exemplo de TDD com JavaScript Mocha

```
var assert = require('assert'), factorial = require('../index');
suite('Teste', function () {
  setup(function () {
    // Cria quaisquer objetos que possam ser necessários
  });
  suite('#fatorial()', function () {
    test('igual a 1 para configurações de tamanho zero', function () {
      assert.equal(1, factorial(0));
    });
    test('igual a 1 para configurações de tamanho um', function () {
      assert.equal(1, factorial(1));
    });
    test('igual a 2 para configurações de tamanho dois', function () {
      assert.equal(2, factorial(2));
    });
    test('igual a 6 para configurações de tamanho três', function () {
      assert.equal(6, factorial(3));
    });
  });
});
```

**Listagem 3.** Resultado de comando de execução do Java version

```
module.exports = function (val) {
  if (val < 0) {
    return NaN;
  }
  if (val === 0) {
    return 1;
  }

  return val * factorial(val - 1);
};
```

Agora se executarmos os testes, poderemos ver que todos eles passarão com sucesso. Isso é como o TDD funciona.

Ao utilizar esse tipo de processo, alguns benefícios se tornam claros logo já, como o baixo acoplamento com uma série de testes que comprovam o seu comportamento. Um dos resultados mais visíveis do TDD é que os testes desenvolvidos sobre o mesmo fornecem uma espécie de documentação informal do sistema, conforme falamos anteriormente, ditando não só o que o sistema pode fazer mas também o que não deve fazer (e é aí onde entram os testes falhos). Além disso, pelo simples fato de tudo isso estar integrado ao software, podemos dizer que o produto teste nunca ficará ultrapassado, o que difere das atividades de escrita de documentação e/ou comentar código.

Alguns outros prós se fazem mais sutis, porém importantes, no ciclo de vida do software bem como na adesão de produtividade ao projeto, tais como a diminuição do tempo usado para depurar código, o que implicará no lucro de tempo e, consequentemente, dinheiro. Tudo isso é possível pelo simples fato de o TDD informar a quem o usa sempre que um erro ocorrer no sistema, diminuindo o tempo de depuração e deixando-a menos custosa.

## BDD

O BDD foi primeiramente introduzido por Dan North no seu artigo “Instrução ao BDD” e é uma metodologia que evoluiu das práticas de TDD. BDD é uma técnica de design sobre user stories, que deve ser escrita em uma linguagem inteligível por não-programadores. Isso possibilita que todos os envolvidos no projeto, executivos, testers, usuários e outros funcionários a se tornarem mais ativos no desenvolvimento. As user stories são centradas na seguinte sintaxe:

Título (uma linha descrevendo a história)

Narrativa:

Como um(a) [regra]

Eu quero [funcionalidade]

Que então [beneficia]

Critério de aceitação: (apresentado como cenários)

Cenário 1: Título

Dado um [contexto] e [mais alguns contextos]...

Quando [evento]

Então [saída] e [outras saídas]...

Como ocorre com o TDD, BDD é uma técnica de design outside-in. No BDD as histórias são escritas primeiro, então verificadas e priorizadas pelos usuários e envolvidos não-técnicos. O programador então cria o código para atender às histórias descritas. Em BDD, existem três princípios-base:

- Negócio e tecnologia devem referenciar o mesmo sistema da mesma forma;
- Qualquer sistema deve ter um valor identificado e variável para o negócio;

- Análises do tipo up-front, design e planejamento têm um retorno cada vez menor.

O primeiro princípio base está solidificado sobre a user story, escrita conforme mostramos acima, de uma maneira não-técnica. Essa noção de linguagem natural segue muitos frameworks BDD, mesmo em um nível de código. O segundo princípio base de adicionar valor de negócio pode ser visto na história como a parte “Que então [beneficia]”.

O terceiro princípio tem o mesmo significado do processo TDD. Um design upfront é feito da forma mais pequena o possível, mesmo que o design esteja indo por além do teste e refactoring.

Juntos os princípios ajudam os desenvolvedores a mitigar os riscos de criar recursos extra ou de criar funcionalidades de forma errada. Até hoje, o BDD não tem recebido tanta atenção na comunidade de pesquisas, mas isso provavelmente tenha relação com a sua idade e sua origem de método de pesquisa, o TDD, que ainda é alvo de muito contradição.

Ok, podemos dizer que entendemos o que é BDD. Mas, é ainda brota algumas clássicas e famosas confusões. Alguns dizem que o BDD é similar ao TDD (fato que justifica ainda mais o porquê de uma ter surgido a partir da outra), alguns outros irão dizer que é o mesmo que TDD porém com alguns guidelines melhores, ou ainda que são abordagens totalmente diferentes para o desenvolvimento.

Independente da definição, isso não importa muito. A principal coisa a se saber é que o “BDD é feito para eliminar problemas que o TDD venha a causar”.

Em contraste ao TDD, o BDD consiste em escrever comportamento e especificação que então dirija o desenvolvimento do nosso software. Comportamento e especificação devem ser vistos como similares para testes mas a diferença é deveras importante.

Vejamos, então, novamente o problema descrito na seção anterior, quando falávamos sobre TDD, sobre escrever uma função para calcular o valor fatorial de um número ([Listagem 4](#)).

A principal diferença é somente a escrita. O BDD usa um estilo mais verboso então isso pode ser lido quase como uma sentença.

Isso é o mesmo que dizer que o BDD elimina problemas que o TDD venha a causar. A habilidade de ler seus testes como uma sentença é uma maneira mais cognitiva de como podemos pensar acerca dos mesmos testes. O argumento atua no “se você pode ler seus testes fluidamente, você naturalmente irá escrevê-los de uma forma melhor e mais compreensiva”.

Apesar de esse exemplo ser muito simples e não ilustrar totalmente isso, os testes BDD devem estar focados nas funcionalidades, e não nos resultados (essa também é uma das principais diferenças entre os dois). Frequentemente, você ouvirá que o BDD existe para ajudar no design do software, e não testá-lo como o TDD é suposto de fazer.

## BDD

DDD, ou Domain-Driven Development, é um conjunto de padrões e métodos que focam na elaboração de aplicações que se caracte-

rizam por refletir a cobrança compreendida e satisfatória de um negócio qualquer. Analisando sobre outra ótica, ele constitui uma inovação na forma de formar qualquer pensamento sobre a prática de desenvolvimento e seus métodos. O DDD lida com a construção de uma espécie de molde da realidade por inicialmente entendê-la por completo e só depois pôr terminologia, regras e meio lógico juntos de forma abstrata no código fonte, o que comumente se chama de modelo de domínio. Não constitui nenhum framework ou ferramenta, mas sim uma espécie de plugin conceitual que pode ser acoplado ao projeto em qualquer momento.

#### Listagem 4. Resultado de comando de execução do Java version

```
var assert = require('assert'), factorial = require('../index');

describe('Teste', function () {
  before(function(){
    // Coisas a se fazer antes dos testes, como importações
  });

  describe('#factorial', function () {
    test('igual a 1 para configurações de tamanho zero', function () {
      assert.equal(1, factorial(0));
    });

    test('igual a 1 para configurações de tamanho um', function () {
      assert.equal(1, factorial(1));
    });

    test('igual a 2 para configurações de tamanho dois', function () {
      assert.equal(2, factorial(2));
    });

    test('igual a 6 para configurações de tamanho três', function () {
      assert.equal(6, factorial(3));
    });
  });

  after(function () {
    // Qualquer coisa a se fazer depois dos testes serem finalizados
  });
});
```

O DDD pode ser entendido também como uma forma de automatizar um negócio e seus processos, de forma que estejamos sempre focados no software em si para que só assim o negócio seja atendido por completo.

É comum, inclusive, associar o desenvolvimento de softwares corporativos a essa ideia em específico, mas o DDD vai além e mostra formas e padrões de conceitualizar tudo isso em um modelo de domínio.

#### TDD x BDD

A situação de ter de escolher entre TDD e BDD é complicada. Isso depende de haver ou não um framework de testes apropriado para a sua linguagem escolhida, o que os seus colegas de trabalho sentem em relação ao mesmo, e dependendo da situação muitos outros fatores entram na contagem.

Alguns defendem que o BDD é sempre melhor que o TDD porque ele tem a possibilidade de eliminar problemas que possam aparecer quando do uso do TDD.

A chave para o BDD é que ele pode prevenir esses problemas; isso não é garantido. Problemas como má organização do código, más práticas de design, etc. sempre irão persistir. Você deixará de ter de escrever testes ruins e em contrapartida terá funcionalidades mais robustas.

Particularmente, nenhum dos estilos é melhor que o outro, isso realmente depende da situação, e de quem usa. Alguém que sabe como escrever ótimos testes TDD pode ter alguns problemas como alguém que sabe escrever ótimos testes BDD. Se você se pegar escrevendo testes incompletos usando TDD e quiser um design de software melhor, então dê uma chance ao BDD.

#### Ainda mais processos

Existem ainda algumas outras terminologias usadas para classificar outras vertentes de teste usadas como processos nos projetos ágeis. Vejamos algumas delas abaixo:

- FDD – Feature Driven Development (Desenvolvimento Guiado por Funcionalidades): O FDD é um tipo de teste feito para manter e criar projetos por intermédio de uma lista de atividades simples, de forma a incentivar o desenvolvimento de novos códigos, bem como compartilhar o conhecimento acerca dos mesmos. Isso permite, então, a principal meta do FDD seja alcançada no ciclo de vida do software: resultados contínuos, funcionais e tangíveis.
- ATDD – Acceptance test-driven development (Desenvolvimento Guiado por Testes de aceitação): Conforme falamos anteriormente, esse tipo de teste comumente é confundido com os testes do tipo TDD, por causa da similaridade com que o desenvolvimento ágil associa os testes de aceitação aos mesmos. Nesse tipo de teste, o trabalho está totalmente direcionado aos testes de aceitação, o que significa que pode haver uma analogia entre este e o TDD.

#### Quando escrever testes unitários?

Nós já discutimos inúmeros conceitos relacionados à utilização dos testes unitários até aqui. Talvez a pergunta mais pertinente que possa surgir nesse momento, antes mesmo de entender e aceitar todos os conceitos até então apresentados, seja: Por que eu devo escrever testes unitários? Quando devo fazer isso?

Alguns desenvolvedores lidam com esse processo como mais uma prática chata a ser seguida dentro dos processos de uma empresa. Mais ainda, talvez antes de se fazer tais perguntas, sejam necessários antes um contato inicial com o ato de escrever testes. Somente assim, você terá insumo suficiente para discernir se deve ou não escrever testes de unidade, e aí, então, teremos a primeira pergunta respondida. Mas e quanto ao “quando”?

É fácil para um gerente semi técnico dizer “escreva todos os testes do início” (em alguns casos, dizer isso pra você mesmo será tão fácil quanto), mas a verdade é que você frequentemente não saberá quais são os pequenos componentes que são necessários quando apresentados juntos a um caso de uso de nível considerável.

É comum, por exemplo, que desenvolvedores façam os seus testes à medida que se vai desenvolvendo o código, à medida que todos as “unidades” requeridas para implementar o caso de uso são feitas. Ao mesmo tempo, você estará constantemente mudando funções sobre esse código, refatorando e abstraindo também algumas partes, e tentando se antecipar ao que possivelmente irá causar perda de tempo durante esse processo.

Consideremos o seguinte cenário: “como um usuário, ao logar na aplicação você será levado para uma tela de menu principal. Como podemos quebrar essa operação genérica em unidades?

Imediatamente podemos selecionar o “login de formulário” como uma implementação desse caso de uso que pode ser testada, e iremos adicionar algumas linhas de código de teste requeridas para implementar essa funcionalidade corretamente (**Listagem 5**).

**Listagem 5.** Exemplo de divisão dos passos para realizar o teste unitário

```
describe('login usuário', function() {
  // crítico
  it('garanta que os endereços de email inválidos serão pegas', function() {});
  it('garanta que os endereços de email válidos irão passar da validação', function() {});
  it('garanta que o formulário de submissão modifique o caminho padrão', function() {});

  // é bom ter...
  it('garanta que o helper client-side helper seja exibido para campos vazios', function() {});
  it('garanta que ao pressionar enter no campo de senha envie o formulário', function() {});
});
```

Dependendo das restrições implementadas, você deve decidir se deve ou não pular alguns passos, mas felizmente é possível ver com facilidade o quanto crítico são cada um deles, isto é, que se eles falharem podem bloquear o uso da aplicação.

Um outro exemplo clássico alusivo é o da construção de um carro. Para construir um carro do início ao fim, cada parte que faz a motor, o chassi, os pneus, devem ser verificados individualmente para estar em ordem antes que elas possam de fato ser atreladas ao carro.

Se o tempo for dito insuficiente para averiguar todas estas peculiaridades, então possivelmente peças importantes serão despriorizadas e poderão não estar funcionando corretamente. Por exemplo, os pneus podem acabar não sendo verificados em relação à pressão, a costura interior pode não ser verificada contra danos. Isso pode resultar em um carro falho, mas isso não seria seu problema, aparentemente. Apenas uma dica: verifique com cautela o que pode ou não ser deixado de lado no momento de testar.

## Teste Unitário em JavaScript

Essa é uma história de controvérsias e recente. Apesar de JavaScript já ocupar espaço considerável na comunidade de desenvolvimento front-end, inclusive até quando comparada a outras linguagens de programação mais famosas como o Java,

C# ou PHP, sendo considerada referência como linguagem, o JavaScript ainda tem muito a evoluir junto a suas companheiras de lado (client side).

É muito fácil, e comum, ignorar testes unitários quando se embarca em um projeto de grande escala que faz uso de JavaScript. Muitas vezes motivados pelo grande costume de desenvolver testes apenas no lado do servidor, uma vez que muitas das regras de validação são sempre duplicadas em ambos os lados cliente e servidor. Porém, a necessidade de construir testes de unidade para JavaScript é tão real quanto com quaisquer outras linguagens. As ferramentas e procedimentos de testes unitários, por sua vez, não são tão claros para JavaScript quanto para as demais linguagens, o que acaba por construir uma imagem mais defensiva dos programadores para com o teste usando a linguagem de script.

O JavaScript vem de uma longa caminhada contra muitos conceitos, e anticonceitos. Houve um tempo onde ele era facilmente descartado, talvez adequado para validações não-críticas, mas não mais que isso. Ao longo dos anos, o pior obstáculo que programas feitos em JavaScript enfrentavam era sempre escrever aplicações em larga escala, ou ao menos mitigáveis. Com o tempo, os dialetos JavaScript foram ficando mais consistentes entre os browsers do que eram antes, e ferramentas como o jQuery podem ajudar a deixar essas diferenças (que ainda existem) ainda mais leves aos olhos dos desenvolvedores. Os debuggers finalmente alcançaram melhores patamares de qualidade, jamais vistos antes (sim, debugar JavaScript era terrível), atraindo, assim, o que antes afastava.

### Nota

O debug se tornou melhor ao longo do tempo, mas ainda depende do browser usado, pois cada fabricante define suas próprias ferramentas de depuração.

Para todos os casos, com o advento da HTML5 e das plataformas mobile, o JavaScript está se tornando cada vez menos ignorável. Aplicações front-end inteiras estão sendo desenvolvidas em JavaScript, e enquanto a ferramenta suporta defasagens antigas escondidas em linguagens como Java e C#, isso significa que temos evoluído.

E dentro desse escopo, uma área que permanece particularmente mudando, e que irá começar a ficar incrivelmente importante nos projetos JavaScript de escopo crescente são os testes unitários.

Para ilustrar uma situação mais próxima da programação, vejamos um exemplo de função básica em JavaScript, conforme o código a seguir:

```
function addQuatroAoNumero(num){
  return num + 4;
}
```

A mesma função basicamente adiciona o valor 4, como um número, à variável passada por parâmetro. Na **Listagem 6** podemos

verificar então um possível caso de teste para uma função de execução.

Após passar o valor 5 para a função sendo testada, o teste checa que o valor retornado é 9. Se o teste suceder, a mensagem "Passou!" é impressa no console de um browser moderno, caso contrário, a mensagem "Falhou!" aparecerá. Para executar este exemplo, você precisará efetuar dois passos básicos:

1. Importar o arquivo js que contém o código apresentado nas duas listagens anteriores.

2. Abrir a página da **Listagem 7** no browser.

Ao invés de usar o console do browser para exibir os valores, você pode imprimir os mesmos dentro da página ou dentro de uma janela pop-up através da função "alert()" do próprio JavaScript.

Dentro do universo de teste unitário, os elementos usados para verificar se certa condição é satisfeita são chamados de "assertions" (afirmações). Na **Listagem 6** o assertion está concentrado na operação:

```
addQuatroAoNumero (num) === valorEsperado
```

Para os casos em que se tem muitos assertions sendo feitos, é interessante fazer uso de um framework de teste unitário JavaScript para auxiliar.

## Escolhendo um Framework

Se você desenvolve com Java, você provavelmente não perde muito tempo decidindo que framework de teste unitário irá usar. O mais famoso deles, o jUnit, é sempre a opção preferida da maioria dos desenvolvedores por razões óbvias de qualidade. O panorama do JavaScript não é tão resolvido assim, e

que framework escolher pode afetar profundamente as suas capacidades de testes.

Vejamos pois alguns dos frameworks mais famosos para JavaScript, bem como suas particularidades que ajudarão a analisar quando adotar um ou outro.

### Listagem 6. Caso de teste para a função addQuatroAoNúmero()

```
(function testAddQuatroAoNúmero () {  
    var num = 5, valorEsperado = 9;  
  
    if (addQuatroAoNúmero (num) === valorEsperado) {  
        console.log("Passou!");  
    } else {  
        console.log("Falhou!");  
    }  
});
```

### Listagem 7. Página HTML para execução do exemplo

```
<!DOCTYPE html>  
<html>  
    <head>  
        <meta http-equiv="Content-type" content="text/html; charset=utf-8">  
        <title>Exemplo Teste Unitário JavaScript</title>  
        <script type="text/javascript" src="js/script.js"></script>  
    </head>  
    <body></body>  
</html>
```

## qUnit

O qUnit é um ótimo candidato para começar. Antes de tudo, ele é consideravelmente fácil de aprender. De início pode ser um pouco complicado para entender alguns comportamentos que

## Conhecimento faz diferença!

The advertisement features several magazine covers from the 'Engenharia de Software' series. One cover highlights 'Gerência de Configuração' (Configuration Management) and 'Aulas desta edição: Estratégia de Teste Funcional baseada em Casos de Uso – Partes 5 a 9'. Another cover shows a robot and discusses 'Automação de Testes' (Automation Testing). Other covers mention 'Processo: Medição de Software: Um importante instrumento de gerenciamento' and 'Agilidade: Acompanhamento de projetos ágeis distribuído através do Daily Meeting'. The overall theme is professional software engineering education.

## Faça um upgrade em sua carreira

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



DEV MEDIA

são menos intuitivos, mas assim que se entende os problemas por trás destes comportamentos tudo começa a fluir mais rapidamente. Ele possibilita interação com o DOM; você pode, por exemplo, identificar uma DIV na sua página de teste para ser resetada para seu estado original entre os testes, promovendo assim atomicidade aos testes. Além disso, ele também provê suporte a “testes assíncronos”, o que constitui uma funcionalidade vital para suas aplicações.

## js-test-driver

O js-test-driver foca no teste paralelo em múltiplos browsers. Ele tem sua própria linguagem de casos de teste e assertions, distinta do qUnit. Existem projetos que possibilitam ambos de serem usados juntos, apesar de não existir um que suporte todas as capacidades de ambos frameworks. Diferente do qUnit, ele não usa partes do DOM para relatar resultados e reinicia o DOM inteiro entre um teste e outro. Isso impõe uma restrição a ser testada.

## Ambiente de Execução dos Testes

Essa é outra questão que parece muito mais complicada com JavaScript do que com Java: em qual ambiente devo executar meus testes?

Você deve sempre querer que o seu ambiente de teste seja compatível com o ambiente de produção o mais próximo possível. Mas para muitas aplicações JavaScript, um pedaço crítico desse ambiente, o browser com seu respectivo motor JavaScript, é escondido pelo usuário. Mesmo se os problemas cross-browser (vários browsers) não forem tão ruins quanto eles eram antes, ainda é muito imaturo dizer que um teste feito em um browser terá um resultado consistente por além dos demais.

Ao mesmo tempo, testar em todos os diferentes browsers traz mais desafios quando você começar a pensar em testes automatizados e CI. Esse é o grande desafio que o js-test-driver é designado a resolver. Se você quer uma solução puramente qUnit, você provavelmente irá encontrar por si só escrevendo seus próprios scripts ou fazendo sem automação.

Existem vários ambientes de execução JavaScript com suporte fácil a scripts, como o NodeJS, por exemplo, que será muito útil para alguns testes automatizados de alto nível. Porém, eles terão suas próprias limitações (NodeJS não tem suporte a DOM nativo, apesar dos add-ons existirem) e em todo caso não conseguirá simular o comportamento do browser.

Uma abordagem multicamada deveria se aplicar melhor às suas necessidades. Você poderia criar alguns scripts de teste para executar no NodeJS por CI, e então rodar periodicamente uma suíte de testes em cada browser alvo. Alguns bugs deverão não ser pegos como seriam se usasse uma CI completa.

## Estrutura do Código

Nos primórdios do jUnit, não era incomum encontrar código Java que não se dirigisse diretamente aos testes unitários. Ao longo dos anos, isso passou a ser visto como um teste de qualidade de um

design. O JavaScript tem, como melhor, começado a desenvolver essa disciplina, e algumas das funcionalidades da linguagem que empurram os desenvolvedores Java na direção certa (como encapsulamento) não gozam do mesmo nível de suporte nativo no JavaScript.

Existem alguns frameworks que podem ajudar na escrita de testes unitários. O ExtJS, por exemplo, põe um sistema de classes no topo do modelo de objetos do JavaScript. Ferramentas como essa podem ajudar se você as permitir, porém como elas não fazem parte da linguagem, você pode sempre escolher ir contra as mesmas. Escrever JavaScript testável é, e para o futuro continuará sendo, uma arte a ser modelada de acordo com a prática.

Não é algo simples manter sempre o foco na testabilidade. Mesmo que você programa todas as linhas do seu código com os olhos voltados para a testabilidade do mesmo, sempre aparecerão determinados comportamentos no código que fugirão à regra, como ubiquidade ou comportamentos assíncronos. Todo motor JavaScript é single-threaded, mesmo que algumas tarefas simples como requisitar um dado do servidor (usando Ajax, por exemplo), respondendo à chamada UI, ou explicitamente retardando uma atividade (através de um setTimeout ou setInterval), são todas feitas assincronamente.

Então, o que um executor de testes deve fazer? Você é chamado a agrupar o código que está testando.

1. O framework é configurado para executar os testes;
2. O código de teste configura pré-condições e faz uma chamada para o código a ser testado;
3. O código de teste checa os assertions;
4. O framework reporta os resultados e vai para o próximo teste.

Mas e se o código a ser testado faz alguma coisa de forma assíncrona? Então, podemos verificar alguns outros passos específicos:

1. O framework é configurado para executar os testes;
2. O código de teste configura pré-condições e faz uma chamada para o código a ser testado;
3. O código a ser testado inicia algo, e então ele chama alguma função para finalizar a tarefa;
4. O framework reporta (imprecisamente) os resultados e vai para o próximo teste;
5. Outras funções de fila devem executar;
6. A parte em fila do código a ser testada recebe sua vez de executar, e finaliza a coisa toda que foi iniciada no passo 3;
7. O código de teste verifica os assertions.

Dessa forma você consegue uma porção de testes falhando por uma razão não muito boa.

O qUnit endereça isso com a função asynchronousTest(). Isso funciona apenas como a função regular test(), exceto pelo fato do framework não assumir que o teste está finalizado quando o código de teste retorna. Você deve informar a ela quando o teste está finalizado chamando a função start().

Esse tipo de comportamento soa um pouco retrógrado. Você chama a função start() quando você precisa finalizar o teste, porque isso diz ao framework para reiniciar o processamento. Esse mecanismo inteiro faz com que os testes executem em lockstep, o que parece que é síncrono, mas você pode chamar asynchronousTest() porque o nome referencia a natureza assíncrona do código sendo testado.

## Objetos Mock

A natureza dinâmica do JavaScript traz, por sua vez, vários benefícios e flexibilidades para escrever objetos mock, mesmo usando um framework de objetos mock.

Em Java, o seu objeto mock tem que compartilhar uma interface com, e/ou ser uma subclasse de, a classe que o substitui. Dependendo da sofisticação do seu framework mock, isso pode impor várias restrições e/ou fazer com que o desenvolvedor pule alguns passos para ir direto ao trabalho pronto. Em JavaScript, você pode criar um objeto que implemente exatamente o mesmo comportamento do mock. De certa forma isso também ajuda se o seu código tiver sido escrito com algum tipo de inversão de controle embutida, caso contrário ao menos você irá se beneficiar com a forma fácil de interceptar uma função chamada no JavaScript.

Desenvolver código unitário não é uma tarefa simples, tampouco personalizável e imediatista como muitos desenvolvedores a consideram. Exige conhecimentos acerca do processo, técnicas e intimidade com as ferramentas e processos envolvidos. Existe, sobretudo, experiência que, por sua vez, só pode ser adquirida

com o tempo e o acúmulo de situações durante a vida como desenvolvedor.

Teste unitário para JavaScript, então, é mais complicado de assimilar principalmente em vista do histórico que a linguagem traz consigo, não sendo considerada por muito tempo sequer para programação básica, no até então advento das tecnologias server side.

## Autor



**Sueila Sousa**

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



## Links:

**Site do W2Schools, mantenedor do JavaScript**  
[www.w3schools.com/js/](http://www.w3schools.com/js/)

**Página oficial do qUnit**  
[www.qunitjs.com/](http://www.qunitjs.com/)

**Página do projeto do JS Test Driver**  
[www.code.google.com/p/js-test-driver/](http://www.code.google.com/p/js-test-driver/)

# 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](http://porta80.com.br) | [comercial@porta80.com.br](mailto:comercial@porta80.com.br) | [twitter.com/porta80](http://twitter.com/porta80)

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