



Front-end
m a g a z i n e

Edição 03



DEV MEDIA

SASS NA PRÁTICA

PRÉ-PROCESSAMENTO DE CSS NA WEB

Testes Unitários com JavaScript
Conheça na prática os principais
frameworks de teste para JavaScript

Construindo servidores com Node.js
Todo o poder do Node.js na criação e
configuração de servidores web

MVP

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

APLIQUE ESSE INVESTIMENTO
NA SUA CARREIRA...

**E MOSTRE AO MERCADO
QUANTO VOCÊ VALE!**

CONFIRA TODO O MATERIAL
QUE VOCÊ TERÁ ACESSO:

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

POR APENAS
R\$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR
MAIS DO MERCADO!

 **DEVMEDIA**



EXPEDIENTE

Editor

Diogo Souza (diogosouzac@gmail.com)

Consultor Técnico

Daniella Costa (daniella.devmedia@gmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araujo

Atendimento ao leitor

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

Publicidade

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

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

Fale com o Editor!

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

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



DIOGO SOUZA

diogosouzac@gmail.com

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

Sumário

Conteúdo sobre Boas Práticas

04 – SASS CSS: Pré-processadores na prática

[Rafael Milléo]

Conteúdo sobre Boas Práticas

16 – Como construir um servidor web com Node.js

[William Carvalho]

Artigo no estilo Curso

26 – Testes unitários em JavaScript: Conheça os principais frameworks - Parte 2

[Sueila Sousa]

Sass: CSS Pré-processado na prática

Nos últimos anos, o CSS vem se tornando um recurso cada vez mais obrigatório na lista de tecnologias base de uma empresa, com requisitos e necessidades também cada vez mais complexas. Isso se deve em grande parte devido ao aumento de dispositivos com os mais variados tamanhos de tela (celulares, flobets, tablets, dentre muitas outras criações), o que exige um design mais arrojado dos websites. Além disso, o design também assume características imprescindíveis para os mesmos websites, principalmente os que alcançam demasiado sucesso, sem mencionar nos fatores de arquitetura da informação, recursos, SEO, revezamento de conhecimento e compartilhamento da grande quantidade de informação manipulada.

Quando se falava em CSS há algum tempo atrás, o termo simplicidade era de certa forma bem usado, dadas as restritas condições que os browsers e plataformas disponibilizavam, além de recursos simples e comuns. Agora o profissional front-end tem de lidar com folhas de estilo cada vez mais complexas e extensas, além do fato de algumas vezes ter de se aventurar no jQuery para conseguir aquele efeito ou aquela medida perfeita que o CSS não consegue fazer e, por sua vez, usando um pouco do processamento do browser do cliente. Em contrapartida, muitos dos célebres problemas de padronização de estilos e regras, muitos destes que incluíam o Internet Explorer em quaisquer das suas versões anteriores à versão 8, foram e estão sendo corrigidos com o passar do tempo.

É comum vermos, por exemplo, aquele problema típico do front-end de copiar e colar trechos de código e só alterar os seletores: o problema vem depois quando se tenta fazer manutenção em um e esquece de fazer naquela cópia que foi feita para aquela página que ninguém se lembra mais que existe.

Visando atingir todos esses cenários problemáticos e muitos outros que aqui ainda serão citados, é que foram criadas ferramentas de pré-processamento de CSS. Estas envolvem uma sintaxe bastante próxima à do CSS, porém com algumas alterações que muitos front-end sonharam, como o uso de variáveis, funções, importação de código, dentre várias outras.

Neste artigo vamos explorar ao máximo o uso do Sass (*Syntactically Awesome StyleSheets*), que resumidamente,

Fique por dentro

Neste artigo serão descritos os principais conceitos, recursos e funcionalidades da plataforma Sass, sua aplicabilidade e um breve tutorial para elaboração de um aplicativo de teste, demonstrando todo o potencial da biblioteca de pré-processamento de CSS. Este tema é útil para a elaboração de aplicações ou módulos de sistemas que estejam sujeitos à sobrecarga de recursos de formatação e padronização de folhas de estilo, assim como ao mau uso dos mesmos, visando estabelecer um contato inicial, porém completo, por sobre o framework e seus usos diversos no cenário de desenvolvimento web atual.

se trata de um módulo desenvolvido em Ruby que faz a leitura de dois tipos de arquivos com extensão **Sass** ou **SCSS**. Estas extensões tratam de sintaxes do próprio SASS que vamos falar mais adiante, e fica à escolha do desenvolvedor qual das duas vai utilizar. Basicamente, o sistema faz uma tradução para um arquivo CSS com todas as regras interpretadas e atualizadas, porém aplicando todas as funcionalidades do SASS.

O SASS pode ser utilizado de três formas: utilização como módulo independente, como uma ferramenta de linha de comando ou como um plugin de framework **Rack-enabled** ou **Ruby on Rails**. Para este artigo vamos utilizá-lo diretamente no terminal de comandos.

Vale ressaltar que também existem outros sistemas similares, tal como o *LESS* e o *Foundation*. Para aprender o Sass (ou qualquer outra destas ferramentas) não é necessário nenhum conhecimento aprofundado de programação nem a instalação de várias ferramentas e ambientes. Para este artigo utilizaremos a versão mais atual até a redação deste artigo, a 3.3.14.

Instalação e Configuração

O Sass é uma ferramenta que necessita do ambiente Ruby para poder executar suas tarefas de leitura e interpretação do código. Caso prefira, existem aplicativos que constroem todo o ambiente para poder iniciar o uso do Sass, sendo alguns deles gratuitos (basta conferir a lista completa de opções disponíveis na seção **Links**), mas a instalação manual não é algo tão complicado e nem demanda muito tempo.

A instalação do Sass varia muito pouco conforme o sistema operacional: caso você esteja usando o Windows basta baixar o instalador do Ruby e realizar a instalação normalmente. No caso do Ubuntu, basta acessar o terminal e digitar o comando:

sudo apt-get install ruby.

Ou caso a sua distribuição Linux seja derivada do RedHat, é necessário digitar o comando:

sudo yum install ruby.

Já nos sistemas Mac, o Ruby já vem pré-instalado, então a partir daí basta acessar o terminal e executar o seguinte:

gem install Sass

Com isso, o ambiente já está preparado para as implementações que faremos neste artigo. Para conferir se tudo está correto, digite o comando de verificação da versão do Sass:

Sass -v

Sintaxes

O Sass tem duas sintaxes diferentes de escrita: a própria sintaxe Sass e a sintaxe SCSS. A diferença entre elas é bem sutil e está presente em apenas alguns detalhes. Vejamos um exemplo da sintaxe SCSS representado pela **Listagem 1**.

Listagem 1. Exemplo de uso da sintaxe SCSS.

```
body{
  margin:0;
  padding:0;
  text-align:center;
}
```

O SCSS lembra muito o CSS tradicional que utilizamos, difere apenas em alguns aspectos e, em alguns casos, para quem está habituado com JavaScript e afins, lembra a estruturação padrão do JSON. Veja na **Listagem 2** um exemplo de uso dessa sintaxe.

Listagem 2. Exemplo de uso da sintaxe Sass.

```
body
  margin: 0
  padding: 0
  text-align: center
```

Mas a sintaxe do Sass é mais sucinta e direta, inclusive na chamada de funções. Entretanto, tenha cuidado, pois esta sintaxe leva em consideração a indentação do código e inclusive o espaço que separa o atributo do valor. Na documentação oficial do Sass (ver seção **Links**) todos os exemplos são disponibilizados nas duas sintaxes, mas para fins didáticos, neste artigo utilizaremos apenas o padrão SCSS para deixarmos bem claro o que estamos fazendo e não nos perdemos em algum erro de indentação ou algum espaço que fique faltando.

Mãos à obra

Primeiramente vamos criar um diretório em um local de sua escolha e criar também um arquivo HTML com um conteúdo que atenda a todos os testes que serão feitos. Veja na **Listagem 3** o código que deve ser adicionado ao mesmo arquivo.

Iremos criar uma estrutura básica inicial de HTML e seletores de classes e identificadores necessários para manipular os elementos. Veja que ela se divide basicamente em elementos HTML diversos, tais como parágrafos, títulos, campos de formulário e algumas imagens. Tais componentes serão muito úteis mais à frente quando precisarmos explorar ao máximo os recursos do Sass, uma vez que o mesmo age diretamente nos componentes do DOM da página HTML. Após isso, execute a página em um browser qualquer e veja o resultado (**Figura 1**).

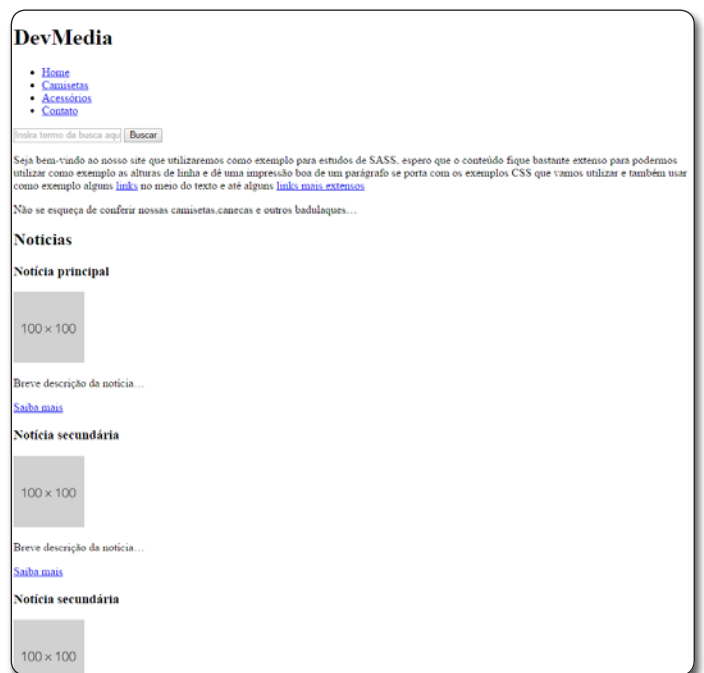


Figura 1. Tela Inicial do Sistema

Em seguida, crie um arquivo de extensão SCSS no mesmo diretório e insira o conteúdo apresentado na **Listagem 4**. Esse código será responsável por definir apenas as características básicas iniciais do estilo da página, tais como margem e alinhamento, tal como no CSS, os efeitos serão aplicados.

Até aqui nenhuma novidade, inclusive esta é uma sintaxe CSS comum e já bem conhecida pela maioria dos desenvolvedores web, mas o importante por enquanto é nos atermos ao processo de criação de uma folha de estilo pré-processada. Após salvarmos o arquivo SCSS é necessário compilá-lo, e para isso abra o terminal e navegue até o diretório onde foi criado o arquivo SCSS. Em seguida execute o seguinte código:

```
Sass [nome_do_arquivo].scss [nome_do_arquivo].css
```


Listagem 3. Código HTML inicial – Modelo Sass.

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="style.css">
    <meta charset="utf-8">
    <title>DevMedia - Apreendendo Sass</title>
  </head>
  <body>
    <header>
      <h1>DevMedia</h1>
      <nav>
        <ul>
          <li><a href="#">Home</a></li>
          <li><a href="#">Camisetas</a></li>
          <li><a href="#">Acessórios</a></li>
          <li><a href="#">Contato</a></li>
        </ul>
      </nav>
      <form id="search">
        <input type="text" placeholder="Insira termo da busca aqui" />
        <input type="submit" value="Buscar" />
      </form>
    </header>
    <section id="content">
      <p>Seja bem-vindo ao nosso site que utilizaremos como exemplo para estudos de SASS, espero que o conteúdo fique bastante extenso para podermos utilizar como exemplo as alturas de linha e dê uma impressão boa de um parágrafo se porta com os exemplos CSS que vamos utilizar e também usar como exemplo alguns <a href="#">links</a> no meio do texto e até alguns <a href="#">links mais extensos</a></p>

      <p id="content-advise">Não se esqueça de conferir nossas camisetas, canecas e outros badulaques...</p>

      <h2>Notícias</h2>
      <div class="products">
        <div class="box size-1">
          <h3>Notícia principal</h3>
          
          <p>Breve descrição da notícia...</p>
          <a href="#">Saiba mais</a>
        </div>
        <div class="box size-2">
          <h3>Notícia secundária</h3>
          
          <p>Breve descrição da notícia...</p>
          <a href="#">Saiba mais</a>
        </div>
        <div class="box size-2">
          <h3>Notícia secundária</h3>
          
          <p>Breve descrição da notícia...</p>
          <a href="#">Saiba mais</a>
        </div>
      </div>
      <h2>Produtos</h2>
      <div class="products">
        <div class="box size-3">
          <h3>Camiseta DevMedia</h3>
          
          <p>R$40,00</p>
          <a href="btn comprar">Comprar!</a>
        </div>
        <div class="box size-3">
          <h3>Caneca DevMedia</h3>
          
          <p>R$10,00</p>
          <a href="btn comprar">Comprar!</a>
        </div>
        <div class="box size-3">
          <h3>Chaveiro DevMedia</h3>
          
          <p>R$5,00</p>
          <a href="btn comprar">Comprar!</a>
        </div>
      </div>
    </section>
    <aside id="sidebar"></aside>
    <footer>Site desenvolvido apenas para estudos relacionados a SASS</footer>
  </body>
</html>
```

Listagem 4. Código SCSS inicial de exemplo.

```
body{
  margin:0;
  padding:0;
  text-align:center;
}
```

E pronto, temos uma folha de estilo CSS gerada pelo Sass. Isso é apenas para exemplificar como funciona o processamento junto à ferramenta, mas vamos ousar mais no nosso código usando o básico do conceito de “aninhamento de seletores”, dentre outros conceitos que o farão entender melhor tais implementações. Adicione o código da **Listagem 5** ao final do arquivo SCSS.

Note que o mesmo consta de códigos com tags aninhadas representando uma hierarquia pré-estabelecida. Agora precisamos novamente executar aquele comando Sass no terminal:

```
Sass [nome_do_arquivo].scss [nome_do_arquivo].css
```

O resultado no arquivo CSS será semelhante ao representado na **Listagem 6**.

Listagem 5. Código SCSS para alinhamento do conteúdo da página.

```
#content{
  margin:0 auto;
  text-align:left;
  p{
    font-size:1.2em;
    font-family:"Arial" sans-serif;
    a{
      color:#00FF00;
    }
  }
}
```

Listagem 6. Resultado CSS de execução do SCSS da Listagem 5.

```
#content {
  margin: 0 auto;
  text-align: left; }
#content p {
  font-size: 1.2em;
  font-family: "Arial" sans-serif; }
#content p a {
  color: #00FF00; }
```

Repare que o código gerado também tem uma indentação semelhante à do CSS convencional que os desenvolvedores fazem, pois ele segue um padrão de aninhamento. Mais adiante vamos demonstrar como alterar esta formatação caso seja de sua preferência.

Além disso, se repararmos nos atributos do parágrafo no conteúdo, podemos ver que há uma repetição dos atributos de fonte, normalmente no CSS colocaríamos todos os valores em um único atributo *font*, mas com o SASS temos outra facilidade que pode nos ajudar muito, que é o reuso da aplicação de regras em hierarquia já disponibilizada pelo CSS. Então vamos alterar estas propriedades do parágrafo para as mostradas na **Listagem 7**.

Listagem 7. Alterando propriedades do parágrafo para otimizar CSS.

```
...
font: {
  size: 1.2em;
  family: "Arial" sans-serif;
}
...
```

E antes de vermos o resultado temos de executar mais uma vez o comando para compilar o SCSS, mas vamos facilitar nossa vida a partir de agora. No SASS existe um comando para o arquivo SCSS ser compilado automaticamente a cada vez que salvarmos o mesmo. Para isto, execute o seguinte comando no terminal e então a partir daí não teremos mais que mexer nele (mas não feche o terminal).

```
Sass --watch [nome-do-arquivo].scss:[nome-do-arquivo].css
```

Agora que não precisamos mais nos preocupar com estas configurações, vamos prosseguir implantando uma referência ao elemento pai de um seletor. Para isto utilizamos a notação **&** que faz referência ao seletor pai imediato do bloco onde foi inserido. Vamos ver seu funcionamento na prática adicionando dentro do bloco de âncora um seletor pai e uma série de regras para o hover (**Listagem 8**).

Listagem 8. Bloco de âncora para o hover

```
a{
  color: #00FF00;
  &:hover{
    text-decoration: none;
  }
}
```

Ok, você deve estar se perguntando por que não fazer isso normalmente usando o seletor da tag? Este foi um exemplo muito simplista, que não conseguirá explicitar todo o poder desse recurso. Vamos então adicionar um CSS para o nosso aviso da página inicial. Insira o código da **Listagem 9** dentro do bloco referente ao seletor `#content`. E você vai perceber que quando o arquivo for compilado e gerado, o resultado será igual ao da **Listagem 10**.

Listagem 8. Bloco de âncora para o hover

```
a{
  color: #00FF00;
  &:hover{
    text-decoration: none;
  }
}
```

Listagem 9. Código para aviso na página inicial

```
&-advise{
  background: #49c5bf;
  padding: 10px;
}
```

Listagem 10. Resultado da compilação do código de aviso inicial.

```
#content-advise {
  background: #FF0;
  padding: 10px; }
```

Esta referência traz também o modificador do seletor, se é uma classe, um id ou um elemento, então é possível perceber que ele faz uma cópia dos seletores pai de uma forma bastante crua e direta.

Mixins

Outra funcionalidade bastante útil do Sass são os **mixins**, que são conjuntos de parâmetros e valores que podem ser reutilizados no seu código CSS bastando apenas chamar o nome do mixin que é criado. É como uma espécie de recurso pré-fabricado que guarda o estado e as características assimiladas ao mesmo inicialmente e que poderão ser aplicados em outros lugares com as mesmas propriedades resguardadas. Para nossos campos do formulário vamos criar um mixin através do código da **Listagem 11**.

Listagem 11. Exemplo de mixin para o formulário

```
@mixin form-input{
  color: #000;
  font:{
    family: Arial;
    size: 1.2em;
    weight: normal;
  }
  padding: 5px;
  width: 150px;
}
```

Para chamá-lo em um elemento do nosso formulário, basta fazer o seguinte:

```
form.cadastro input[type='text']{
  @include form-input;
}
```

A saída gerada no CSS é o trecho exato do nosso mixin declarado anteriormente. Vamos supor que ao invés de ficar declarando toda

vez que estas regras se aplicam apenas aos campos de texto, podemos também incluir o próprio seletor no nosso mixin. Para tal, altere o conteúdo do arquivo para o apresentado na **Listagem 12**.

Listagem 12. Incluindo o próprio seletor no mixin

```
@mixin form-input{
  input[type='text']{
    color:#000;
    font:{
      family:Arial;
      size:1.2em;
      weight:normal;
    }
    padding:5px;
    width:150px;
  }
}
```

Além disso, precisamos alterar também a chamada do nosso mixin, logo que ele não precisa de um seletor, tal como o código a seguir apresenta:

```
form.cadastro { @include form-input; }
```

Mas vamos supor que esta largura que definimos no nosso mixin possa variar conforme o formulário e inclusive o tipo de mídia que a página vai exibir, como podemos solucionar isso com o Sass? Para isto, poderíamos criar mixins com os mais diferentes tamanhos ou até mesmo o definir diretamente nas regras do elemento em questão. Mas há uma alternativa melhor: passar parâmetros pelo mixin. Vamos alterar a declaração dele para a que é exibida na **Listagem 13**.

Listagem 13. Definindo as regras do mixin com parâmetros

```
@mixin form-input($largura_do_campo){
  input[type='text']{
    color:#000;
    font:{
      family:Arial;
      size:1.2em;
      weight:normal;
    }
    padding:5px;
    width:$largura_do_campo;
  }
}
```

Podemos ver que agora na declaração consta uma variável chamada `$largura_do_campo` e que veremos mais adiante. Como passo final, vamos chamar o mixin com o parâmetro `$largura_do_campo` preenchido:

```
form.cadastro { @include form-input(150px); }
```

Se por acaso não quiser ficar sempre preenchendo este valor do parâmetro, você pode definir um valor padrão, da seguinte forma (tal como é feito em algumas linguagens de programação):

```
@mixin form-input($largura_do_campo: 150px){
  ...
}
```

Note que se você tem proximidade com a sintaxe das linguagens de programação orientadas a objetos, então será mais fácil entender como tais seletores e recursos funcionam no Sass.

A partir daí você pode chamar o método `form-input` sem atribuir um valor no parâmetro da função. Um outro exemplo é para aquelas regras de CSS3 que dependem de parâmetros para cada navegador, como por exemplo o *multi-columns*, que em CSS3 pode ser definido como na **Listagem 14**.

Então por que não criar um mixin para gerar estes valores automaticamente? Veja na **Listagem 15** como ficaria o mesmo código *multi-columns* se feito a partir de um mixin.

Listagem 14. Exemplo de *multi-columns* no CSS3.

```
-moz-column-count: 3;
-moz-column-gap: 5px;
-webkit-column-count: 3;
-webkit-column-gap: 5px;
column-count: 3;
column-gap: 5px;
```

Listagem 15. Mixin de exemplo para gerar valores *multi-columns* automaticamente

```
@mixin multi-columns($count, $gap){
  -moz-column-count: $count;
  -moz-column-gap: $gap;
  -webkit-column-count: $count;
  -webkit-column-gap: $gap;
  column-count: $count;
  column-gap: $gap;
}
```

A chamada, por sua vez, ficaria bastante simples:

```
@include multi-columns(3, 5px);
```

Vendo toda esta gama de possibilidades que os mixins podem trazer, o nosso código consequentemente fica bem mais fácil de manter e evoluir.

Sass Script

O uso do Sass Script é um dos principais atrativos do Sass, aliado ao uso de funções e variáveis para realizarmos o processamento de dados para o CSS. Todos os recursos do Sass Script podem ser usados dentro e fora dos mixins, e o Sass Script tem um potencial muito grande de tornar o seu CSS em algo muito mais automatizado e de fácil manutenção. Com recursos como variáveis, estruturas de decisão (`if/else`) e de repetição (`for/while/each`), a escrita de script Sass pode oferecer suportes e ferramentas ao CSS que irão reduzir muito o trabalho na criação do front-end de um site.

Nome	Descrição	Exemplo
Numbers	Valores numéricos com sufixo px, em ou sem nenhum sufixo	10px, 1.2em, 50
String	Sequências de caracteres com ou sem aspas (tanto duplas quanto simples)	"Teste", 'teste', TESTE
Colors	Cores em qualquer formato aceito pelo CSS, nominal, hexa, RGB ou RGBA	red, #FF6600, rgba(0,0,0,0.75)
Booleans	Valores binários de true ou false	true, false
List	Representa uma série de valores de um atributo CSS, similares as declarações de padding ou font. Os valores podem ser separados por vírgula ou somente por espaço	10px 20px 10px 0 10px Arial Bold
Map	Bastante similar aos vetores, mas os índices têm de ser definidos previamente	(chave1: valor1, chave2: valor2, chave3: valor3)
Null	Valor nulo	null

Tabela 1. Tipos de variáveis do Sass

Variáveis

As variáveis servem para armazenar valores, mas uma diferença das variáveis do Sass são os tipos de variáveis, conforme demonstrado na **Tabela 1**.

O tipo Colors foi criado especialmente para lidar com o tratamento de cores de forma estruturada, com recursos que permitam defini-las de forma mais simples, escolhendo entre hexadecimal, RGB, etc. As demais variáveis representadas pela tabela se encaixam nos mesmos conceitos das que as linguagens de programação fazem uso.

Este é um exemplo simples de declaração de uma variável (você pode inserir esta variável em qualquer parte do código, desde que seja antes de chamar – por isso, é recomendado inserir no começo do script):

```
$largura_maxima: 1024px;
```

Para utilizarmos esta variável no nosso código basta chamar pelo nome dela, conforme a **Listagem 16**.

Listagem 16. Exemplo de uso de uma variável no Sass

```
...
#content{
  margin:0 auto;
  text-align:left;
  width:$largura_maxima;
  ...
}
```

No Sass também é possível criar variáveis globais, ou seja, variáveis que podem ser declaradas em um único arquivo e quando ele for importado em outros arquivos Sass, a variável não precisa ser redeclarada. Para declarar uma variável global basta adicionar a notação `!` antes da mesma (`!global`):

```
$variavel_global: "Conteudo da variavel global"!global;
```

Para adaptar o conceito ao nosso projeto exemplo, vá até o script SCSS e altere a variável `$largura_maxima` para uma variável global:

```
$largura_maxima: 1024px !global;
```

Outra notação bastante curiosa é a `!default`, uma forma de dizer que se a variável já foi definida antes ela não precisa ter seu valor alterado na segunda declaração. Para entender melhor vejamos o código da **Listagem 17**.

Listagem 17. Exemplo de uso da notação default

```
$variavel: "Conteudo original";
$variavel: "Conteudo novo"!default;
.teste{
  content: $variavel;
}
```

O conteúdo da variável "variável" é modificado duas vezes, mas na segunda temos a inclusão do elemento `default`. O resultado, portanto, não será alterado novamente:

```
.teste {
  content: "Conteudo original"; }
```

Lists e Maps

O **List** trata de um tipo de variável de valores múltiplos, da mesma forma que quando é declarado, por exemplo, um atributo de `margin` onde temos valores das quatro margens do elemento:

```
margin: 10px 15px 20px 0;
```

Ele é semelhante às estruturas de coleções, mas na maioria das vezes assumirá somente o valor máximo de quatro posições. Ele representa a constituição de outros valores que usamos geralmente associados aos sufixos **left**, **right**, **top** e **bottom**. Este valor atribuído é exatamente uma variável do tipo `list`, onde temos diversos valores separados por espaço (neste caso). Para entendermos melhor vamos criar uma variável do tipo `list` e colocá-la no nosso CSS. Portanto adicione as linhas da **Listagem 18** no arquivo SCSS.

Vale a pena ressaltar também que para uma variável ser `list` ela não necessariamente precisa ter quatro valores separados por barra de espaço ou vírgula. A variável pode muito bem conter três ou dois valores apenas, e se houver apenas um valor, ela será classificada como uma variável numérica.

Note também que não precisamos definir o tipo da variável explicitamente, pois o Sass irá entender e assimilar os tipos por si só.

Listagem 18. Exemplo de uso da variável List

```
...
$largura_maxima: 1024px;
$margem_boxes_home: 10px 5px 0 5px;
...
}

.home_box{
  margin:$margem_boxes_home;
  width:300px;
}
&-advise{
...

```

Uma das vantagens de se utilizar variáveis do tipo list também está relacionada a algumas funções próprias para list, como:

- join (para juntar duas listas);
- append (que adiciona um valor ao fim de uma lista);
- zip (que combina diversas listas em uma única lista multidimensional);
- Dentre outras funções que facilitam o acesso a valores de listas.

Já os **Maps** lembram bastante variáveis de vetor das linguagens de programação, onde temos valores associados a um índice, como por exemplo:

```
$paleta_cores: (fundo: #333333, conteudo: #FFFFFF, fonte: #000000, link: #49c5bf);
```

Este mapa pode ajudar a sempre seguir a paleta de cores do site e para utilizar este tipo de variável basta usar uma função chamada `map-get`, e como parâmetro terá de ser passado o nome da variável de tipo map e o índice que deseja receber o valor.

Vamos adicionar a variável `$paleta_cores` ao nosso código SCSS logo após a declaração das outras variáveis que usamos antes e aplicar esta paleta de cores, tal como apresentado na **Listagem 19**.

Veja que nosso código terá um modelo de dados pré-salvo que poderá ser reusado nas demais partes do código. Além disso, não esqueça que os escopos de variável local e global também se aplicam a essas estruturas, portanto tenha cuidado com os tipos de uso que faz delas.

Os maps também são manipulados por funções, assim como as lists, tais como:

- `map-get` (que utilizamos em nosso exemplo): para recuperar um valor;
- `map-merge`: para mesclar dois mapas em apenas um;
- `map-remove`: para remoção de algum valor;
- `map-has-key`: para verificar se algum índice já existe no mapa.

Realizando operações

O Sass também disponibiliza a realização de operações, mas não só de valores numéricos, como também de cores. Primeiramente vamos nos atentar às operações aritméticas com números, poden-

do realizar as quatro operações básicas da matemática, além de verificar o valor de resto para divisões (%).

Listagem 19. Exemplo de uso das variáveis Map no nosso modelo

```
...
$largura_maxima: 1024px;
$margem_boxes_home: 10px 5px 0 5px;
$paleta_cores: (fundo: #333333, conteudo: #FFFFFF, fonte: #000000, link: #49c5bf);
body{
  background-color: map-get($paleta_cores, fundo);
  margin:0;
  padding:0;
  text-align:center;
}
#content{
  background-color: map-get($paleta_cores, conteudo);
  color: map-get($paleta_cores, fonte);
...
a{
  color:map-get($paleta_cores, link);
  &:hover{
...

```

Ou seja, se desejarmos realizar uma operação de soma de duas margens para um atributo de margin de um elemento HTML, podemos seguir os exemplos:

```
margin: 10px + 25px;
margin: 10px * 5;
```

Veja que não é preciso se preocupar com a remoção dos sufixos de px ou de quaisquer outros que venham acompanhados ao número em questão. Tais operações antes só podiam ser efetuadas através de linguagens como JavaScript.

Além destas operações, também é possível realizar comparações de valores com os seguintes operadores relacionais exibidos na **Tabela 2**.

Tipo	Operador relacional
<	Menor
>	Maior
<=	Menor ou igual
>=	Maior ou igual
==	Igual
!=	Diferente

Tabela 2. Lista de Operadores Relacionais

Podemos realizar uma mudança no nosso código que facilitará a forma como manipulamos as características dimensionais da nossa página. Inserimos uma nova classe chamada `box`, que serão caixas de conteúdo, três por linha na página principal do nosso site e colocamos um valor fixo de largura, no caso 300px, mas

para fazermos de uma forma que o código dependa menos de valores fixos, vamos alterar o conteúdo dessa mesma linha para o apresentado na **Listagem 20**.

Listagem 20. Código de largura com valor calculado.

```
...
.home_box{
  margin:$margem_boxes_home;
  width:$largura_maxima / 3;
}
...
```

Então dividimos a largura do conteúdo pelo número de boxes que teremos por linha. E assim o nosso CSS foi compilado para o resultado gerado:

```
#content .home_box {
  margin: 10px 5px 0 5px;
  width: 341.33333px;}
```

Porém, existe um problema em relação ao Sass com divisão de valores, principalmente no atributo font. Devido ao fato da propriedade font ter suporte à definição de não apenas o tamanho da fonte, mas também a altura da linha (line-height), ou seja, ao invés de escrevermos:

```
font-size:12px;
line-height:16px;
```

Podemos encurtar para o seguinte código:

```
font: 12px/16px;
```

É exatamente aí onde devemos prestar atenção ao utilizarmos divisão de valores, pois o Sass pode compilar, por exemplo, o código SCSS “font:12px/6px;” para exatamente o mesmo valor em CSS. Para contornarmos este problema basta delimitar a operação com parênteses, desta forma o próprio Sass vai interpretar isto como uma operação matemática e fará a compilação da forma esperada, de uma maneira muito semelhante a como fazemos com esse tipo de operação nas linguagens de programação e afins.

Outra espécie de operação são as operações com string, que basicamente fazem o uso de concatenação, como por exemplo:

```
$minha_string:“Olá” + “Mundo!”;
```

O conteúdo da variável será “Olá Mundo!”.

Uma outra diferença que o Sass tem para as outras linguagens de programação é que as partes a serem concatenadas não precisam ter o mesmo tipo de aspas, ou não ter aspas. Sendo assim, os exemplos ilustrados na **Listagem 21** também são compilados pelo Sass de forma natural e esperada.

Listagem 21. Exemplos de uso de concatenação de strings

```
$minha_string:“Olá” + Mundo!;
$minha_string:‘Olá’ + “Mundo!”;
$minha_string:Olá + Mundo!;
$minha_string:‘Olá’ + ‘Mundo!’;
```

Ele também provê recursos de interpolação para a representação de valores de variáveis ou até mesmo de operações declaradas na própria string:

```
$minha_string:“A soma de 7 e 3 é: #{7 + 3}”;
```

Podemos usar a interpolação para inserirmos o valor de uma variável sem termos de usar a notação de concatenação:

```
$minha_string:“O conteudo da variavel é: #{$variavel} e ela se chama variavel.”;
```

Neste caso, as strings serão compiladas como o esperado, com os valores já somados e o valor da variável.

Estruturas de controle

Agora que já vimos como o Sass realiza operações, vamos ver as estruturas de controle e escrevermos funções e mixins cada vez mais úteis e reutilizáveis.

Condicionais (if / else)

Estruturas condicionais tem uma organização bastante simples e no Sass são iguais às estruturas condicionais da maioria das linguagens de programação, mas sem parênteses obrigatórios em volta da condição. Veja um exemplo bem simples e seu resultado do Sass na **Listagem 22**.

Listagem 22. Exemplo de estrutura condicional no Sass

```
body{
  @if 2 + 3 == 5 {
    color: #FF0000;
  }
}
// E assim, o arquivo CSS irá apresentar este resultado:
body {
  color: #FF0000;}
```

Ainda podemos colocar um else para que um bloco de código seja compilado no caso da condição ser falsa, como mostra a **Listagem 23**.

Além disso, o acréscimo de mais tipos de verificações encadeadas com @else if aumenta a gama de condições. Veja na **Listagem 24** um bom exemplo disso.

Para fixarmos este conteúdo, vamos alterar o mixin form-input no nosso código SCSS que utiliza algumas estruturas condicionais: abra o arquivo SCSS que estamos usando de exemplo e altere as linhas de comando para as apresentadas na **Listagem 25**.

Listagem 23. Exemplo de uso do else na estrutura if.

```
body{
  @if 2 + 3 > 5 {
    color: #FF0000;
  } @else{
    color: #000000;
  }
}
```

Listagem 24. Exemplo de else if encadeados.

```
$variavel: 42;
body{
  @if $variavel < 10{
    color: #FF0000;
  } @else if $variavel < 30{
    color: #00FF00;
  } @else {
    color: #FFFF00
  }
}
```

Listagem 25. Adicionando estruturas de condição ao código modelo

```
...
@mixin form-input($largura_do_campo: 150px){
  input[type="text"]{
    color: #000;
    font:{
      family: Arial;
      size: 1.2em;
      weight: normal;
    }
    padding: 5px;

    @if $largura_do_campo > $largura_maxima {
      width: $largura_maxima;
    } @else {
      width: $largura_do_campo;
    }
  }
}
...
```

Agora nossa função não estará sujeita a erros dos desenvolvedores que chamarem a função com um valor de tamanho maior que o tamanho total do conteúdo.

Laços de repetição (@while/@for/@each)

Sempre devemos atentar ao fato de incluir o incremento (ou decremento) da variável de contagem que é verificada a cada iteração do while, evitando assim um loop infinito.

Vamos fazer um laço while para criarmos os estilos das boxes da home, cada uma com uma largura diferente. Para isso, adicione as linhas de código da **Listagem 27** ao seu arquivo SCSS (insira o trecho dentro do seletor #content, ao final dele).

Já o @for é uma estrutura mais enxuta que realiza a mesma função do @while e tem a seguinte sintaxe (caso fôssemos reproduzir o mesmo efeito que o while que criamos anteriormente):

```
@for $cont from 1 through 6{
  .item-size-#{ $cont } { width: 100px * $cont; }
}
```

Listagem 26. Exemplo de uso do while no Sass.

```
$cont: 1;
@while $cont <= 6 {
  li.item-size-#{ $cont } { width: 100px * $cont; }
  $cont: $cont + 1;
}
```

Listagem 27. Criando laço while para estilos de box

```
...
#content{
  ...
  $cont: 1;
  @while $cont <= 3 {
    .box.size-#{ $cont } {
      $margem_box: 20px;
      $largura_box: ($largura_maxima / $cont);

      @if $cont > 1{
        $largura_box: $largura_box - $margem_box;
        float: left;
        margin-right: $margem_box;
      }
      width: $largura_box;
    }

    $cont: $cont + 1;
  }
  ...
}
```

Este laço irá criar uma lista de classes do .item-size-1 até .item-size-6, mas existe também uma variação de palavra-chave no @for, onde podemos trocar o through por to e, desta forma o laço não irá passar pelo valor 6 (irá apenas de 1 a 5). A sintaxe seria:

```
@for $cont from 1 to 6
...
```

O @for também é um tipo de laço que não pode ter alteração no incremento do valor da contagem e nem definir que o valor de contagem, na verdade, pois sofrerá um decremento.

Existe também um outro tipo de laço que é mais específico para fazermos a leitura de tipos de variável map e list, ou simplesmente passar por uma lista já criada na própria condição da estrutura, bem semelhante ao foreach de outras linguagens bem conhecidas, como podemos ver no código da **Listagem 28**.

Desta forma criamos uma série de marcações CSS para cada espécie de rede social de uma forma mais inteligente. Vejamos o resultado na **Listagem 29**.

Mas, e se ao invés de uma lista declarada diretamente na estrutura de repetição, quiséssemos ser mais organizados e colocássemos em uma variável à parte? Então nosso código SCSS ficaria semelhante ao exibido na **Listagem 30**.

E no caso de uma variável de map, seria algo igualmente simples. Veja no exemplo da **Listagem 31** como isto pode tornar nosso código mais interessante;

O @each pode ser muito útil para este tipo de dado e pode automatizar muito o nosso código, e inclusive, o @each pode tomar

formas de repetição mais complexas, se tratando de associação múltipla, onde não precisamos percorrer apenas uma lista ou mapa, mas quantos forem necessários. Vejamos o exemplo da **Listagem 32**, onde é possível ver que a estrutura de repetição irá passar por cada lista declarada depois da notação `in` associando cada valor para as variáveis declaradas no começo da estrutura `@each`.

Listagem 28. Exemplo de foreach no Sass.

```
@each $rede-social in facebook, google_plus, twitter, pinterest{
  .social-share li.#{ $rede-social}{
    background-image: url('/images/#{ $rede-social}.png');
  }
}
```

Listagem 29. Resultado das marcações para exemplo de rede social

```
.social-share li.facebook {
  background-image: url('/images/facebook.png'); }
.social-share li.google_plus {
  background-image: url('/images/google_plus.png'); }
.social-share li.twitter {
  background-image: url('/images/twitter.png'); }
.social-share li.pinterest {
  background-image: url('/images/pinterest.png'); }
```

Listagem 30. Declarando código de rede social em variável a parte

```
$redes-sociais: facebook, google_plus, twitter, pinterest;

@each $rede-social in $redes-sociais{
  .social-share li.#{ $rede-social}{
    background-image: url('/images/#{ $rede-social}.png');
  }
}
```

Listagem 31. Exemplo de foreach em conjunto com map.

```
$elementos: (p: 1em, h1: 1.6em, h2: 1.4em, label: 1.2em);
@each $elemento, $tamanho in $elementos {
  #{ $elemento } {
    font-size: $tamanho;
  }
}
```

Listagem 32. Exemplo do @each no modelo

```
@each $estado, $cidade, $cor in (sao-paulo, santos, #0000FF),
  (minas-gerais, belo-horizonte, #FF0000),
  (parana, curitiba, #00FF00) {
  ##{$estado} ##{$cidade} {
    color: $cor;
  }
}
```

Funções nativas do Sass

O Sass oferece uma gama de funções próprias para auxiliar o desenvolvimento dos scripts. Vamos conhecer a lista das principais, mas caso queira ver o restante da lista (algo em torno de 80 funções) basta acessar a documentação oficial na seção **Links**.

Gerais

- **type-of(\$variavel):** Retorna qual o tipo de variável;

- **unit(\$valor):** Retorna qual o tipo de unidade está associado ao valor da variável, podendo variar entre px, em e cm;
- **inspect(\$variavel):** Traz o valor da variável passada por parâmetro, é muito útil para debug de valores.

Numéricas

- **abs(\$valor):** Traz o valor absoluto de um número, ou seja, se o valor negativo ele será positivo da mesma forma;
- **Random(\$limite):** Gera um valor inteiro aleatório. O limite é um parâmetro opcional caso não esteja determinado o valor poderá ser 0 ou 1;
- **percentage(\$valor):** Através de qualquer valor retorna uma porcentagem, pode ser um ponto flutuante, tal como 0.4 ou uma expressão: (40 / 100).

String

- **str-length(\$string):** Retorna qual o total de caracteres de uma string;
- **str-index(\$string, \$substring):** Retorna qual a posição na string de um determinado trecho de string;
- **str-slice(\$string, \$inicio, [\$fim]):** Traz uma string com base no índice que for fornecido no início e opcionalmente o do fim, caso o fim da string a ser extraída não esteja definido a função irá extrair até o fim da string.

Cores

- **rgba(\$vermelho, \$verde, \$azul, \$alpha):** Esta função gera uma cor rgba com base nos valores dos parâmetros. Mas também pode ser utilizada com hexa ou valores nominais, por exemplo:
 - **rgba(#FF0000, 0.5):** Então o resultado será: rgba(255, 0, 0, 0.5). Ou até mesmo, esta função irá apresentar o mesmo resultado: rgba(red, 0.5);
- **mix(\$cor1, \$cor2, [\$variacao]):** Retorna uma cor misturada com base nas duas cores informadas nos parâmetros, pode ser em qualquer formato (hexa, rgb, rgba, nominal, etc). O terceiro parâmetro de variação quer dizer o quanto a primeira cor vai prevalecer sobre a segunda, é um valor percentual, então tendo em vista esta chamada da função:
 - **mix(red, green, 10%):** Quer dizer que a cor vermelha será 10% misturada com 90% de verde.

Existe uma série de funções de cores disponíveis no Sass, inclusive recursos avançados de saturação e matiz de cores, sem mencionar nas funções de mapas e listas.

Notações

No Sass também é possível se utilizar de algumas funções que tem como prefixo o `@`. Estas funções são mais abrangentes e extremamente úteis para tornar o nosso código melhor, sendo desde funções de importação de arquivos, definição de tipo de tela que o site será visualizado, herança, dentre várias outras.

@import

Sim, a importação de arquivos já é um recurso CSS já bastante conhecido pelos desenvolvedores front-end, mas no Sass, este tipo de import é um pouco diferente. Nele, o import vai trazer todo o conteúdo do arquivo no CSS compilado, tornando o arquivo final um arquivo único. Para realizar a importação basta criar um arquivo de extensão SCSS e no arquivo que deseja importar basta adicionar a notação import. Para o nosso exemplo que estamos desenvolvendo, vamos criar um arquivo de reset (coloque qualquer conteúdo que julgar necessário para um reset) e importá-lo ao nosso style.scss, da seguinte forma:

```
@import "reset.scss";
```

É importante dizer que se você quiser, não precisa colocar a extensão do arquivo, seja ela SCSS ou Sass. Então você pode perfeitamente escrever também desta forma:

```
@import "reset";
```

E até mesmo adicionar diversos arquivos de uma só vez:

```
@import "reset", "grid-1080", "bordas-arredondadas";
```

No Sass, o @import não se restringe a importação de arquivos Sass e SCSS, podendo muito bem importar um arquivo CSS também, mas neste caso sua extensão tem de ser especificada, pois o Sass, ao receber uma importação de arquivo sem extensão, automaticamente procura apenas as extensões Sass e SCSS. O @import também pode acessar condições de mídia, URL externas e a notação url() sendo uma importação original do CSS. Vejamos alguns exemplos de importações que são aceitas no Sass na **Listagem 33**.

Listagem 33. Exemplos diversos de importação no Sass.

```
@import "arquivo.css";
@import "reset" screen;
@import "http://site.com/api/css/style.css";
@import url(reset);
```

Retornando a questão do import do Sass, existe também um termo designado para arquivos Sass e SCSS que não são compilados para CSS, estes arquivos são chamados de **partials**, onde a diferença deles é um underscore antes do nome do arquivo. Por exemplo, se fosse um arquivo com uma paleta de cores com uma série de regras CSS e mixins úteis para definir as cores do nosso site, chamaríamos o arquivo de _paleta_cores.scss. E a chamada seria assim:

```
@import "paleta_cores"
```

Pois não é necessário incluir o underscore na chamada do arquivo, pois este underscore apenas serve para o próprio Sass identificar o

arquivo como um partial, e que o mesmo não precisa ser compilado para CSS, além de facilitar a identificação do tipo de arquivo para o desenvolvedor de forma mais intuitiva.

Outro aspecto importante no import é que podemos realizar a importação em qualquer parte do nosso arquivo SCSS, de forma que se quisermos importar um arquivo que tenha regras restritas a um elemento do site podemos fazer uma importação aninhada, desta forma:

```
form#cadastro{
  @import "formularios";
}
```

E dentro do arquivo formularios.scss ter uma série de regras SCSS que serão respeitadas somente dentro da tag form com id cadastro do seu site.

@media

Com o crescimento das mais diversas mídias de acesso à internet, tal como os smartphones, tablets, Smart TVs e dentre outros, as **media queries** estão ficando cada vez mais comuns em folhas de estilo. E no caso do Sass, a diretiva @media não é diferente das media queries do CSS, mas a diferença é a sua inclusão em qualquer parte do código, em meio aos aninhamentos, de forma que não precisaríamos repetir nossos seletores. Por exemplo, voltando ao nosso arquivo SCSS, atualize-o tal como na **Listagem 34**.

Listagem 34. Uso dos aninhamentos com media no modelo

```
...
#content{
  background-color: map-get($paleta_cores, conteudo);
  color: map-get($paleta_cores, fonte);
  margin: 0 auto;
  text-align: left;
  width: $largura_maxima;
  @media (max-width: 600px){
    margin: 0;
    width: 100%;
  }
  p{
    ...
```

E desta forma, o nosso código será compilado para o mesmo apresentado na **Listagem 35**.

Também é possível ainda aninhar uma media query dentro de outra. Veja na **Listagem 36** um exemplo disso. O resultado seria o exibido na **Listagem 37**.

E até mesmo podemos utilizar variáveis para realizar media queries de forma dinâmica, um exemplo disto pode ser visto no caso da **Listagem 38**.

@extend

A notação de herança é muito útil no Sass principalmente quando estamos lidando com elementos que não tem nenhuma relação hierárquica, mas em suma, suas propriedades são bastante parecidas.

Listagem 35. Resultado da inclusão da media no modelo.

```
width: 1024px; }
@media (max-width: 600px) {
  #content {
    margin: 0;
    width: 100%; }
  #content p {
```

Listagem 36. Media aninhadas umas nas outras.

```
@media (max-width: 600px){
  margin:0;
  width: 100%;
  @media (orientation: landscape){
    width:500px;
  }
}
```

Listagem 37. Resultado das medias aninhadas.

```
@media (max-width: 600px) {
  #content {
    margin: 0;
    width: 100%; }
  @media (max-width: 600px) and (orientation: landscape) {
    #content {
      width: 500px; }
  }
```

Listagem 38. Variáveis para media queries.

```
$midia: screen;
$atributo: max-width;
$valor: 600px;
@media #{$midia} and ($atributo: $valor){
  margin: 0;
  width: 100%;
}
```

Como por exemplo, se no nosso site o botão de busca e o botão de envio de um formulário tivessem o mesmo estilo visual, mas em hierarquia, eles não tem a mesma relação, supondo que nossa barra de busca ficasse no header do site e o formulário no content.

Para isto podemos usar a notação `@extend`, então vamos alterar novamente nosso arquivo SCSS para criarmos este exemplo, inserindo o código da **Listagem 39** antes do seletor `content`.

E assim que compilarmos o SCSS, o resultado é um trabalho repetitivo que nos livramos, como mostra a **Listagem 40**.

Podemos perceber que o código trouxe também o estilo do `input` texto, o que não é interessante pela proposta que tínhamos. A primeira saída que você pode imaginar para trazer especificamente o `input` submit seria alterar o seletor do `extend`:

```
@extend form.cadastro input[type='submit'];
```

Mas isto na verdade não vai funcionar, porque o `extend` não aceita seletores aninhados, apenas referências diretas. Vale lembrar também que o `extend` só vai achar seletores que estão declarados na própria folha de estilo, ele não vai encontrar seletores de parciais ou outros arquivos CSS importados.

Além disso, podemos também realizar a herança de mais de uma classe no nosso código SCSS, tal como representado na **Listagem 41**.

Assim, o seletor do `form#search` será atrelado também aos atributos designados para o `#content-advice`. Além de herança múltipla, podemos também realizar um encadeamento de heranças. Isto não só é uma facilidade mas também pode ser uma armadilha, logo que podemos querer herdar apenas alguns aspectos do próprio elemento, não o que ele herda.

Listagem 39. Exemplo de uso da notação `@extend`

```
...
header{
  form#search{
    @extend form.cadastro;
  }
}
#content{
  ...
  form.cadastro {
    @include form-input(150px);

    input[type="submit"]{
      background-color: map-get($paleta_cores, link);
      color:#FFFFFF;
      cursor:pointer;
    }
  }
  ...
}
```

Listagem 40. Resultado do uso da anotação `@extend`.

```
#content form.cadastro input[type='text'], #content header form#search
input[type='text'], header #content form#search input[type='text'] {
  color: #000;
  font-family: Arial;
  font-size: 1.2em;
  font-weight: normal;
  padding: 5px;
  width: 150px; }
#content form.cadastro input[type='submit'], #content header form#search
input[type='submit'], header #content form#search input[type='submit'] {
  background-color: #49c5bf;
  color: #FFFFFF;
  cursor: pointer; }
```

Listagem 41. Herança em mais de uma classe.

```
form#search{
  @extend form.cadastro;
  @extend #content-advice;
}
```

@at-root

Esta notação, como o próprio nome diz, insere todos os seletores e atributos declarados dentro dele na raiz do documento SCSS. Veja um exemplo simples no nosso modelo na **Listagem 42**.

Este código SCSS reproduz o seletor (`home_box`, neste caso) na raiz do próprio documento, removendo todo o aninhamento que havia antes pelos seus nós pais.

Notações auxiliares (@debug, @warn, @error)

Algumas vezes precisamos, durante o processo de desenvolvimento, utilizar de alguns artifícios para realizar o debug que resolveria algum problema ou acompanharia se o sistema está agindo de forma esperada. A notação `@debug` é um meio de fazer a

análise de conteúdo de variáveis no momento em que os arquivos Sass ou SCSS são compilados. Sendo assim, se em meio ao nosso arquivo SCSS nós colocarmos o seguinte conteúdo (em qualquer parte do script, mas recomendo inserir ao final):

```
@debug "Este é o meu debug, e está tudo bem..."
```

Ao salvarmos o arquivo e ele ser compilado (seja de forma manual ou pela facilidade do watch do Sass) no terminal, a nossa mensagem será impressa neste formato:

```
style.scss:103 DEBUG: Este é o meu debug, e está tudo bem...
```

Listagem 42. Exemplo de uso do @at-root.

```
...
#content{
...
  @at-root{
    .home_box{
      margin:$margem_boxes_home;
      width:$largura_maxima/3;
    }
  }
...
}
```

Sendo que a mensagem tem como formato o nome do arquivo, a linha em que foi emitida a mensagem, o tipo da mensagem e o seu conteúdo. Já o @warn é uma notação útil principalmente para os mixins, onde se o desenvolvedor inserir um tamanho absurdo ou um valor inválido na chamada do mixin, você pode emitir este tipo de mensagem na compilação. A sintaxe do @warn é a seguinte:

```
@warn "Este é o meu warn, e está mais ou menos..."
```

Sendo que a mensagem de warning que o terminal irá exibir será parecida com esta:

```
WARNING: Este é o meu warn, e está mais ou menos...
on line 104 of style.scss
```

Com um padrão diferente do @debug, mas trazendo as mesmas informações.

Por último mas não menos importante, o @error, que tem a mesma utilidade da notação @warn, mas este irá impedir a compilação do arquivo SCSS. Sua sintaxe é igual às duas últimas citadas:

```
@error "Este é o meu error, e deu tudo errado..."
```

Assim que salvarmos o arquivo o terminal exibirá uma mensagem em um vermelho forte, e também se você abrir o arquivo CSS que

foi compilado, verá que todo o CSS se foi, trazendo a mensagem de erro logo nas primeiras linhas e um backtrace completo:

```
error style.scss (Line 105: Este é o meu error, e deu tudo errado...)
```

Este pode ser um recurso útil num caso de última instância, onde um valor de parâmetro incorreto possa causar um loop infinito ou uma saída indesejada.

Este artigo cobriu apenas uma pequena parcela de todo o potencial do Sass afim de introduzir a ferramenta que pode acelerar em muito o desenvolvimento front-end, além de trazer possibilidades para uma folha de estilo que seja mais completa, compatível com todos os navegadores e que utilize o máximo de recursos que o CSS3 pode trazer, além de poder criar layouts responsivos para todos os dispositivos. E como vale lembrar que esta não é a única ferramenta de pré-processamento CSS, existem outras opções no mercado, tal como citado ao decorrer do mesmo.

Existem também boas bibliotecas criadas para o Sass, que é o caso da Bourbon (ver seção **Links**), que traz uma série de facilidades e mixins e pode auxiliar mais ainda no desenvolvimento do front-end. Além disso, também estão sendo criados no mercado frameworks CSS que utilizam por base o Sass, como por exemplo o Compass.

Autor



Rafael Milléo Carrenho

rafael.milleo@gmail.com (Twitter: @milleo)

Desenvolvedor web com experiência há 5 anos no mercado, com experiência em agências e grandes empresas. Hoje prestando serviços no Portal R7 com PHP e Wordpress. Além de paralelamente exercer trabalhos de pesquisas científicas voltadas a sistemas hiperídia e Interação humano computador pela Universidade Presbiteriana Mackenzie.



Links:

Site oficial do Sass.

<http://sass-lang.com/>

Projeto do Sass no Github.

<https://github.com/sass/sass/>

Projeto do Bootstrap Sass no Github.

<https://github.com/twbs/bootstrap-sass/>

Lista de aplicativos de ambiente Sass.

<http://Sass-lang.com/install>

Site do Bourbon.

<http://bourbon.io/>

FÓRUM

DEVMEDIA

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



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

ACESSE AGORA
www.devmedia.com.br/forum



Construindo um servidor web com Node.js

Ao iniciar os trabalhos com Node.js, rapidamente nota-se que mesmo o mais simples dos códigos acaba viabilizando muitas funcionalidades. Mesmo que você seja iniciante, com poucas linhas de código conseguirá criar aplicações de troca de mensagens entre clientes e servidor, como o exemplo mais clássico de aplicações em tempo real, um chat.

Caso possua experiência em outras tecnologias como Java ou PHP, pode até pensar que isso não é “algo do outro mundo”, contudo, o grande diferencial é que o Node.js foi planejado e criado para lidar especificamente com aplicações desse tipo, em rede. E mais, ele foi projetado para lidar com alta concorrência em tempo real. Com isso, ele é capaz de lidar com milhares de clientes conectados ao mesmo tempo, consumindo pouca memória e exigindo pouco processamento do servidor.

Em se tratando de uma tecnologia voltada para o desenvolvimento de aplicações em rede, o Node.js foi construído para ser orientado a eventos e I/O não bloqueante, ou seja, a entrada e saída de dados não bloqueiam os processos de execução das aplicações, fazendo com que a concorrência entre vários clientes simultâneos seja mais performática e eficiente.

Para quem está dando os primeiros passos com o Node, pode parecer um pouco confuso já começar com um exemplo de um servidor web, pois este envolve conceitos de rede, cliente, servidor, o protocolo HTTP, etc. Por outro lado, como o Node foi elaborado tendo como base alguns destes conceitos, a maioria das tecnologias mencionadas já são fornecidas e até gerenciadas pelos próprios módulos dele.

Caso você ainda não esteja familiarizado com esta solução e seu paradigma orientado a eventos, vamos fazer uma rápida revisão.

O Node.js é, basicamente, um combinado do engine JavaScript do Google Chrome, denominado V8, com a utilização de um recurso do sistema operacional que monitora requisições de operações de entrada e saída e que dispara a resposta ao solicitante assim que uma resposta estiver de fato disponível. Em outras palavras, esse monitoramento e, consequentemente, essa resposta, é o que conhecemos popularmente como callback. Cada sistema operacional possui a sua própria implementação

Fique por dentro

Com o intuito de apresentar os conceitos e recursos do Node.js, plataforma JavaScript que vem sendo amplamente adotada em projetos que trazem a escalabilidade como importante requisito, neste artigo vamos criar um servidor web simples com suporte a páginas estáticas. Para isso, serão demonstrados alguns módulos padrão da biblioteca, tais como file system e HTTP, e como “pensar orientado a eventos”, fundamento básico da tecnologia em estudo.

desse mecanismo. Em ambientes Unix, por exemplo, esse recurso é chamado de epoll.

Tendo o V8 como engine para interpretar JavaScript e o próprio sistema operacional fornecendo o I/O não bloqueante, restou aos desenvolvedores do Node.js fazerem a junção dessas funcionalidades, agregando a camada de acesso a arquivos, criptografia, rede, e outras features comuns em qualquer linguagem de programação.

Como funcionará o servidor web

O conceito de um servidor web é relativamente simples. Como qualquer tipo de aplicação que age como um servidor, iniciamos uma socket que fica “escutando” uma determinada porta e que recebe conexões de sockets cliente que desejam trocar informações com este servidor.

Sempre que um servidor é implementado, é preciso definir um protocolo, ou seja, definir a forma como o cliente e o servidor irão se comunicar. Uma das maneiras mais convenientes de se estabelecer essa comunicação entre é definindo comandos para que o cliente solicite informações ao servidor, que por sua vez pode interpretar esses comandos, associando-os a uma função específica, executá-la e devolver uma resposta à solicitação feita pelo cliente.

No caso específico de um servidor web, esse protocolo já existe e já é encapsulado pelo Node.js em sua biblioteca padrão: o protocolo HTTP.

Embora ele facilite a nossa vida, fornecendo uma API completa e cheia de módulos associados, não só para o protocolo HTTP, mas também para uma gama imensa de recursos relacionados a redes, cabe ao desenvolvedor decidir o que fazer com esses recursos, ou seja, fica a seu critério estabelecer como o servidor web deve se comportar, receber e responder as requisições, controlar o acesso

através de login, ou a quantidade de conexões simultâneas que ele estará apto a receber. Em outras palavras, o Node fornece as ferramentas “pré-prontas” para que você possa se concentrar nas regras de negócio da aplicação, sem precisar concentrar esforços na implementação de protocolos HTTP e outras particularidades da conexão entre cliente e servidor.

Dito isso, eis o que vamos construir para que nosso servidor web funcione como queremos:

- Definir um diretório */static*, onde ficarão os arquivos providos pelo servidor;
- Dentro do diretório */static* criaremos subdiretórios para separar os arquivos por tipo (js, css, etc.);
- Criar um arquivo *index.js*, que representará o arquivo principal de nossa aplicação;
- Criar um módulo *server.js*, que representará nosso servidor web;
- Criar um módulo *filehandler*, que será responsável por carregar os arquivos solicitados pelo browser (nosso cliente);
- Criar um arquivo JSON chamado *config.json*, contendo as configurações do servidor web, tais como diretórios padrão, tipos de arquivo, etc.;
- Definir um parâmetro opcional que poderá ser fornecido no momento de iniciar a aplicação, viabilizando alterar o valor da porta na qual o servidor receberá requisições.

Preparação do ambiente

Como esse artigo tem por objetivo apresentar uma aplicação básica, não utilizaremos módulos de terceiros como Socket.IO. Porém, para propósitos de agilizar o desenvolvimento, vamos baixar, através do npm (BOX 1), o nodemon.

BOX 1. npm

É um programa que faz parte dos arquivos binários do Node.js. Sua função é baixar e instalar pacotes de módulos para o Node.js criados pela comunidade.

O nodemon é uma solução que executa qualquer aplicação escrita em Node.js e que fica “vigiando” os arquivos desta. Com isso, caso algum arquivo presente no diretório onde o nodemon foi iniciado for alterado, a aplicação é reiniciada automaticamente, o que faz com que não tenhamos que parar a execução de nossa aplicação e reiniciá-la a cada alteração no código.

Além disso, caso a aplicação “quebre”, ela continuará sendo vigiada pelo nodemon, e quando o problema que causou a interrupção da aplicação for corrigido, ou seja, o trecho de código que possuir erros for reparado, ele reiniciará a aplicação automaticamente.

Agora que você tem uma ideia de como utilizar o nodemon para favorecer a agilidade no desenvolvimento de aplicações, vamos executá-lo em nossa aplicação. Para isso, no entanto, precisamos ter algum arquivo para passar como parâmetro na linha de comando.

Esse é um bom momento para criarmos não só o arquivo principal de nossa aplicação, mas também toda a sua estrutura de

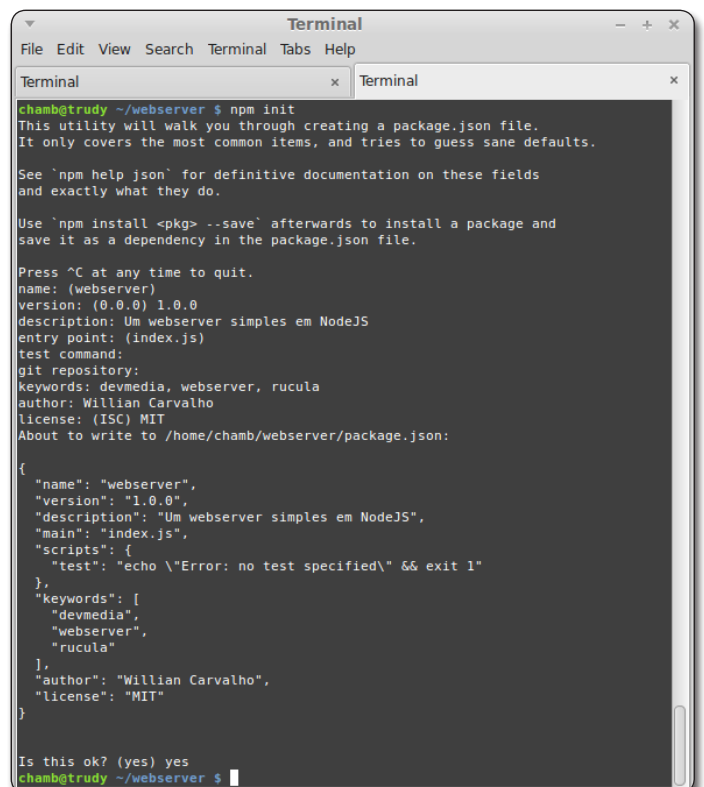
diretórios. Sendo assim, vamos criar um diretório com o nome que acharmos mais apropriado para a aplicação – por exemplo: *webserver*. A partir desse diretório, crie os demais conforme a **Listagem 1**.

Listagem 1. Estrutura de diretórios da aplicação.

```
webserver
├── static
│   ├── css
│   ├── image
│   └── script
```

Nossa aplicação residirá na raiz do diretório *webserver*. Dentro deste temos o diretório *static*, que será responsável por manter os arquivos solicitados pelo browser através de requisições HTTP. A partir do diretório *webserver* que acabamos de criar, execute o comando *npm init*. Este comando irá gerar um arquivo de nome *package.json* com informações sobre a aplicação, como: dependências, licença, autor, etc.

Quando o comando *npm init* é executado, ele solicitará algumas informações ao desenvolvedor para preencher adequadamente o *package.json*. Você deverá ver uma tela semelhante à exibida na **Figura 1**. Preencha-as como achar melhor, porém, mantenha a opção *entry point* com o valor padrão *index.js*, pois este será o arquivo principal do servidor web.



```
Terminal
File Edit View Search Terminal Tabs Help

Terminal
chamb@trudy ~/webserver $ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (webserver)
version: (0.0.0) 1.0.0
description: Um webserver simples em NodeJS
entry point: (index.js)
test command:
git repository:
keywords: devmedia, webserver, rucula
author: William Carvalho
license: (ISC) MIT
About to write to /home/chamb/webserver/package.json:

{
  "name": "webserver",
  "version": "1.0.0",
  "description": "Um webserver simples em NodeJS",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "devmedia",
    "webserver",
    "rucula"
  ],
  "author": "William Carvalho",
  "license": "MIT"
}

Is this ok? (yes) yes
chamb@trudy ~/webserver $
```

Figura 1. Executando o comando *npm init* e respondendo às perguntas de inicialização

Caso não possua o *nodemon* instalado, execute o comando *npm install -g nodemon* no seu prompt do Windows ou terminal, caso esteja trabalhando em um ambiente Unix, como Ubuntu ou MacOS. Isso fará com que o *npm* instale o *nodemon* de maneira global, ou seja, você estará apto a utilizá-lo de qualquer lugar do seu sistema de arquivos.

Criando os arquivos do servidor web

Agora que já temos o *nodemon* instalado, vamos criar os arquivos de código para que ele possa executá-los e monitorá-los. Para tanto, a partir do diretório *webserver*, crie os seguintes arquivos:

- *index.js*: Arquivo principal da aplicação, um container que agrupará todos os módulos do sistema;
- *server.js*: Módulo que será responsável pelo servidor web;
- *filehandler.js*: Módulo que será utilizado pelo *server.js* e é responsável por manipular os arquivos solicitados pelo usuário;
- *config.json*: Arquivo de configuração cujos dados serão lidos e utilizados pela aplicação.

Neste momento seu diretório *webserver* deve possuir a mesma estrutura apresentada na **Listagem 2**.

Nota

Caso esteja em um ambiente Unix, você pode executar o comando `touch index.js server.js filehandler.js config.json` e todos estes arquivos serão criados de uma só vez.

Listagem 2. Estrutura final de diretórios e arquivos da aplicação.

```
webserver
├── config.json
├── filehandler.js
├── index.js
├── package.json
├── server.js
├── static
│   ├── css
│   ├── image
│   └── script
```

O arquivo *index.js*

O conteúdo do arquivo principal da aplicação, *index.js*, ainda que pequeno, diz bastante coisa sobre o servidor web, pois é a partir dele que faremos a inclusão e utilização de todos os módulos da nossa solução. Sendo assim, é importante entendermos o que está acontecendo em cada linha, pois do ponto de vista da compreensão do código, o *index.js* representa uma visão macro de tudo que nosso servidor web poderá fazer. O seu conteúdo é exposto na **Listagem 3**.

A principal e real função do *index.js* é funcionar como uma espécie de *container*, centralizando a utilização dos módulos, que serão separados por responsabilidades, ou seja, cada módulo será responsável por uma parte do trabalho realizado. Note, contudo, que o arquivo principal não precisa necessariamente “saber” como os módulos fazem seu trabalho. Basta que ele saiba que esses módulos realizam as tarefas que ele precisa.

Listagem 3. Conteúdo do arquivo *index.js*.

```
process.title = 'MyWebServer';

1. var args = process.argv,
2.   port = args[2] || 7070,
3.   webServer = require('./server');
4.
5. webServer.listen(port, function() {
6.   console.log('Server started at port ' + port);
7. });
```

Na primeira linha, ainda que não seja uma instrução fundamental para a aplicação, modificando o atributo **process.title**, definimos um nome para o processo do servidor no sistema operacional. Assim, quando você estiver executando sua aplicação em um sistema operacional derivado do Linux, por exemplo, será possível ver o nome do processo, bem como seu id. Um exemplo disso pode ser verificado na **Figura 2**.

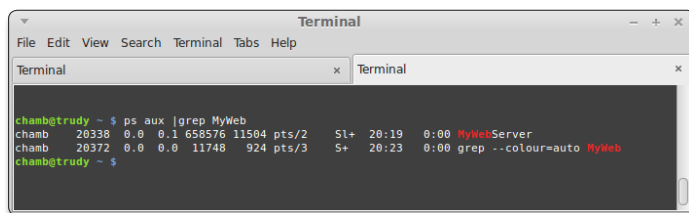


Figura 2. Comando *ps* demonstrando o nome associado ao processo da aplicação

Na declaração das variáveis vemos o módulo **process** sendo usado novamente, desta vez para recuperar os argumentos da linha de comando, através do atributo **argv**. Este objeto (**args**) é um atributo especial do módulo **process** que preserva todos os parâmetros passados quando o comando *node* (ou *nodemon*, no nosso caso, já que estamos em fase de desenvolvimento) é executado.

Por motivos de organização do código, criamos uma variável chamada **args** para armazenarmos uma referência do objeto **argv** do módulo **process**. **argv** funciona como um array, contendo uma lista dos argumentos passados para o programa na linha de comando do prompt do sistema operacional. Seu primeiro elemento representará o executável *node* (ou o *nodemon*, como já explicado). O segundo argumento representará o nome do arquivo sendo executado (no nosso caso o *index.js*), e a partir do terceiro elemento (**args[2]**) temos os argumentos opcionais da aplicação, que no nosso caso será a porta na qual o servidor deve ser inicializado, caso seja fornecida.

Logo após a declaração do objeto **args**, armazenamos o primeiro argumento opcional, ou seja, o terceiro elemento de **args**, assumindo que este é o valor da porta pela qual o servidor web “escutará” por requisições do browser. Caso nenhum argumento seja encontrado, utilizamos o valor padrão 7070.

A terceira e última variável declarada, **webServer**, é o objeto que representa o servidor web propriamente dito e que discutiremos em breve. Nesse momento basta entendermos com que tipo de

conteúdo e como a variável **webServer** está sendo carregada, o que nos leva à instrução **require**, do Node.js, que será explicado a seguir.

Para carregar um módulo em uma aplicação, seja ele criado pelo próprio desenvolvedor, seja ele um módulo importado pelo *npm*, como o Socket.IO ou um driver de banco de dados como o MySQL ou MongoDB, ou ainda, um módulo da biblioteca padrão do Node.js, como o módulo *fs*, é preciso utilizar a instrução **require**, passando como argumento uma **String** com o nome do arquivo correspondente ao módulo que queremos usar. Se quisermos carregar um módulo criado por nós, o nome do arquivo precisa começar com o caminho relativo `“./”` e não é necessário adicionar o prefixo `.js` ao final. Assim, o **require** irá carregar o módulo e associá-lo à variável **webServer**.

Por fim, utilizamos o método **listen()** do objeto **webServer**. Este método recebe como argumento a porta na qual o servidor deve receber conexões dos clientes, e como segundo argumento opcional, uma função de callback que será disparada assim que o servidor for iniciado. No nosso caso, a função apenas exibe uma mensagem no console, dizendo que o servidor foi iniciado na porta especificada.

A única dependência do *index.js*, como vocês podem notar, é o módulo importado pelo **require**, chamado **server**, ou seja, o arquivo *server.js*, apresentado na **Listagem 4** e encontrado no mesmo diretório do arquivo principal. Este arquivo é o responsável por implementar o servidor web propriamente dito.

Nas oito primeiras linhas do *server.js* podemos identificar a inicialização de variáveis e a importação de alguns módulos, como já vimos anteriormente. Contudo, é possível notar algumas peculiaridades em relação à importação do módulo *server.js*, visto no *index.js*.

A primeira coisa a se notar é que, na importação do módulo *http*, não foi utilizado o caminho relativo `“./”`. Isso porque, diferente do módulo *server.js* que nós mesmos escrevemos, o módulo *http* faz parte da biblioteca do Node.js. Para qualquer módulo da API padrão, este deve ser chamado sem a necessidade de se especificar qualquer caminho.

A segunda peculiaridade, ainda que não seja muito visível nesse momento, é a importação do objeto **config**. Este se trata de um arquivo com a extensão e formato JSON. Isso mesmo, o **require** é capaz de importar módulos escritos para o Node, assim como pode converter automaticamente arquivos JSON em um objeto literal JavaScript. No contexto de nossa aplicação, isso quer dizer que, assim como alguns programas utilizam arquivos `.ini` ou `.xml` para ler configurações do sistema, estamos utilizando um arquivo JSON para essa finalidade. Vamos discutir esse arquivo com mais detalhes mais à frente.

Perceba também, falando ainda sobre a importação de módulos, a importação do método **parse()**, que pertence ao objeto **url**, cujo papel é separar cada trecho de qualquer URL e retornar um objeto com cada um deles. Note que, ao invés de importar todo o objeto **url**, com todos os seus métodos e atributos, importamos apenas o método que precisamos utilizar, semelhante à impor-

Listagem 4. Conteúdo do arquivo *server.js*, implementação do servidor web

```
01. var http = require('http'),
02.   config = require('./config'),
03.   fileHandler = require('./filehandler'),
04.   parse = require('url').parse,
05.   types = config.types,
06.   rootFolder = config.rootFolder,
07.   defaultIndex = config.defaultIndex,
08.   server;
09.
10. module.exports = server = http.createServer();
11.
12. server.on('request', onRequest);
13.
14. function onRequest(req, res) {
15.   var filename = parse(req.url).pathname,
16.       fullPath,
17.       extension;
18.
19.   if(filename === '/') {
20.     filename = defaultIndex;
21.   }
22.
23.   fullPath = rootFolder + filename;
24.   extension = filename.substr(filename.lastIndexOf('.') + 1);
25.
26.   fileHandler(fullPath, function(data) {
27.     res.writeHead(200, {
28.       'Content-Type': types[extension] || 'text/plain',
29.       'Content-Length': data.length
30.     });
31.     res.end(data);
32.   }, function(err) {
33.     res.writeHead(404);
34.     res.end();
35.   });
36. }
37. }
```

tação de métodos no Python ou a utilização de métodos estáticos do Java, ainda que esse último tenha uma filosofia e finalidade bem diferentes.

Além dessas novidades, podemos ver a importação do módulo **filehandler** e o uso de variáveis para guardar a referência de alguns atributos do objeto **config**, como o content type do arquivo solicitado, o diretório padrão dos arquivos providos pelo servidor web (**rootFolder**) e o nome do arquivo padrão quando nenhum arquivo for especificado ao chamar a URL do servidor (**defaultIndex**).

Na linha 10 usamos **module.exports**, cujo valor atribuído a ele é exatamente o que o **require** irá importar, seja lá onde for chamado. **module.exports**, como o próprio nome sugere, é a maneira como o sistema de módulos do Node.js exporta módulos e outros objetos a partir de algum arquivo, de forma que esses módulos possam ser importados e utilizados por outros módulos da aplicação.

Como na maioria das linguagens de programação, nós acabamos separando nosso código por responsabilidades lógicas, cada qual em um arquivo, classe ou módulo diferentes, para manter a modularidade da aplicação. Isso é geralmente uma boa prática, pois além de mantermos a aplicação organizada em responsabilidades lógicas, esses módulos podem ser reaproveitados por outros

trechos do código. Por exemplo, um módulo responsável por se conectar a um banco de dados pode ser construído de forma a se conectar com um ou mais bancos diferentes. Paralelamente a isso, é possível ter vários módulos com propósitos específicos que precisam se conectar a bancos de dados. Logo, não faz sentido replicar esse código em cada um desses módulos, sendo mais interessante importar o módulo que já fornece tal recurso.

No nosso caso, estamos “exportando” a variável **server**, atribuindo a ela uma instância de nosso servidor web, através do método **createServer()** do módulo **http**.

Embora tenhamos escrito essa linha com múltiplas atribuições ao mesmo tempo, com o formato **module.exports = server = http.createServer()**, visando manter o código mais enxuto, ela pode não ser tão legível, chegando a ser confusa para algumas pessoas. Para contornar esse problema, poderíamos, por exemplo, instanciar o servidor web, atribuí-lo à variável **server** e então, na linha seguinte, atribuir **server** a **module.exports**. Você pode atribuir qualquer coisa que quiser ao **module.exports**, como funções, variáveis e objetos literais.

Na linha 12 podemos observar uma das mais notáveis e significativas características do Node: os tão famosos eventos!

Na maioria dos casos, eventos são passados como o primeiro argumento de uma função, em forma de string contendo seu nome, como por exemplo, “onload”, “onclick”, “request”, ou simplesmente “on”. Uma vez que o nome do evento é fornecido como argumento de uma função, informa-se a função de callback como segundo argumento, representando a ação a ser executada quando o evento passado no primeiro argumento for disparado.

Para facilitar a compreensão, vamos pegar como exemplo o evento click, comumente utilizado em páginas web. Podemos atribuir o evento click a um botão, a um link ou a qualquer outro elemento HTML que quisermos, mas vamos utilizar um botão por ser o exemplo mais natural.

Uma vez que temos a referência a um botão, através da atribuição da instância deste a uma variável como **myButton** (por exemplo: **var myButton = document.querySelector("#myButton")**), teremos acesso à função **addEventListener()**, que recebe como argumentos uma **String** com o nome do evento e uma função, a ser executada sempre que o evento descrito no primeiro argumento for disparado, como: **myButton.addEventListener('click', triggerEvent)**. A partir desse registro, toda vez que o usuário clicar no botão, a função **triggerEvent** será executada.

O mesmo conceito se aplica a programas escritos com Node.js. Contudo, ao invés de usarmos o método **addEventListener()** do JavaScript para páginas HTML, temos o método **on()** do objeto **server**, que também recebe dois argumentos. O primeiro argumento é uma **String** com o nome do evento que queremos “escutar” e o segundo argumento é uma função de callback com os argumentos **res** (*response*) e **req** (*request*) com a ação a ser realizada a cada vez que o evento for disparado.

No caso da nossa aplicação estamos chamando a função **on** do objeto **server**, passando como primeiro argumento o evento **request**, que é um evento pré-definido pelo servidor HTTP do

Node.js, sendo disparado toda vez que uma nova requisição do cliente for enviada ao servidor (o que normalmente ocorre através de um browser).

Para o segundo argumento da função **on()**, passamos a função **onRequest()**, declarada na linha 14. Esta corresponde a todas as ações a serem realizadas para cada nova requisição de um cliente. Outra opção seria passar uma função anônima diretamente no segundo argumento. Porém, para manter o código mais didático, este argumento foi declarado com uma *named function*, ou seja, uma função declarada com um nome, podendo ser reusada em outros trechos do código.

A função **onRequest()** recebe dois argumentos (**res** e **req**) e é executada quando o evento **request** é disparado. O argumento **res** representa a resposta (*response*) a ser enviada ao cliente, tais como códigos HTTP de erro ou sucesso, e o conteúdo da resposta, como um código HTML a ser interpretado pelo browser, um JSON, XML, uma imagem e assim por diante.

O argumento **req** (*request*) representa a requisição recebida pelo servidor, enviada pelo cliente. Este objeto possui todas as informações relacionadas ao cliente, tais como IP, informações do browser, que arquivo está sendo solicitado (páginas HTML, imagens, etc.), entre outras informações que precisam ser de conhecimento do servidor.

Na linha 15 utilizamos o método **parse()** para interpretar o endereço HTTP solicitado pelo browser e transformá-lo em um objeto JSON, para facilitar o manuseio. Quando o endereço é submetido ao método **parse()**, este retorna um objeto contendo atributos da URL separados convenientemente para que possamos trabalhar com os valores da URL, como **pathname**, **hostname**, **port**, **search** e outros. No nosso caso utilizaremos o atributo **pathname**, pois ele contém o caminho e o nome do arquivo solicitado pelo browser. Em seguida, atribuímos o caminho do arquivo à variável **filename** que, mais tarde, será fornecida ao nosso módulo de leitura de arquivos.

Na linha 19, caso o valor da variável **filename** seja a string “/”, quer dizer que o usuário não especificou nenhum arquivo ao enviar a requisição ao servidor, o que nos leva a carregar o arquivo padrão, representado pela variável **defaultIndex**. Lembre-se que **defaultIndex** possui o valor que foi carregado do arquivo de configuração *config.json*.

Na linha 23 atribuímos à variável **fullPath** o caminho completo do arquivo solicitado pelo usuário, juntando o conteúdo da variável de configurações **rootFolder**, que assim como a variável **defaultIndex**, foi carregada a partir do arquivo de configuração *config.json*, somado ao conteúdo da variável **filename**, que acabamos de analisar.

Já na linha 24 utilizamos a função **substr()** do módulo **String** do JavaScript para capturar apenas a extensão do arquivo solicitado pelo usuário. A partir dessa informação podemos saber qual será o tipo de conteúdo (*Content Type*) a ser entregue ao usuário.

Caso você não esteja familiarizado com o protocolo HTTP, cada arquivo oferecido pelo servidor web é entregue com uma série de outras informações, sendo uma delas o tipo de conteúdo e o seu

tamanho em bytes. Isso informa ao browser como ele deve lidar com o arquivo. Por exemplo, se o servidor web informar que o tipo de conteúdo é *text/plain*, o browser saberá que se trata de um texto simples. Por outro lado, se o servidor informar que o tipo de conteúdo é *image/png*, o browser saberá que se trata de uma imagem, e que ele deve tratar o conteúdo como uma imagem e “pintá-la” na tela.

Alguns desses tipos de conteúdo também estão armazenados no arquivo *config.json*, para que possam ser lidos e interpretados pela aplicação, fornecendo o tipo de conteúdo correto de acordo com a solicitação de arquivo feita pelo usuário. Esses tipos de conteúdo foram carregados na variável **types**, um subobjeto do objeto **config**. Mais adiante iremos entender a estrutura de dados do arquivo *config.json* e como ele está organizado.

Agora que sabemos qual arquivo devemos carregar para enviar ao usuário e qual é o *content type*, ou seja, com qual tipo de arquivo o browser deve interpretar os dados recebidos, podemos utilizar a função **fileHandler()** para realizar a tarefa de ler este arquivo do diretório **static** para nós. Portanto, da linha 26 ao fim do arquivo, podemos ver a função **fileHandler()**, outro módulo de nossa aplicação cujo conteúdo analisaremos mais à frente.

A função **fileHandler()** é importada na linha 3 através da instrução **require** do Node.js. Como verificado, o sistema de exportação e importação de módulos do Node.js pode exportar qualquer coisa, seja essa coisa um objeto, uma variável ou mesmo uma função, como é caso de **fileHandler()**. Isso serve de exemplo para mostrar que podemos importar diferentes tipos de dados, como quando importamos o módulo **server**, por exemplo, que é um objeto mais complexo, composto de variáveis e funções.

Assim como *index.js* apenas precisava fazer uso do módulo **server**, o módulo **server** apenas precisa saber usar a função **fileHandler()**, e é por isso que a isolamos em um novo arquivo. Recapitulando o que foi abordado no começo do artigo, lembre-se que dividimos os módulos do sistema em responsabilidades lógicas. Nosso *index.js* é responsável por iniciar a aplicação, servindo como um container da aplicação, e por isso depende do arquivo *server.js* para fazer todo o trabalho associado ao servidor web.

O arquivo *server.js*, por sua vez, é responsável por toda a lógica associada ao servidor web e por interpretar requisições (requests) e respostas (responses) HTTP, porém ele depende de um módulo responsável pela leitura de arquivos, e é aqui que entra a dependência do **fileHandler()**.

A função **fileHandler()** recebe três argumentos: o caminho completo de onde o arquivo deve ser lido, uma função de callback de sucesso e uma função de callback de erro.

A função de sucesso possui um argumento que representa os dados carregados do arquivo solicitado, e o corpo da função (note que dessa vez utilizamos uma função anônima, para mostrar que você tem liberdade de trabalhar da maneira que achar melhor) utiliza o método **writeHead()** do objeto **res**. O primeiro argumento passado a esse método é o código HTTP (200 quer dizer que a resposta está Ok) e o segundo argumento é um objeto com informações

do cabeçalho HTTP; no caso, o tipo de conteúdo (*Content-Type*) e o tamanho dos dados em bytes (*Content-length*).

Note que foi usado o formato **types[extension]** para capturar o valor da extensão do objeto **types**. Caso não esteja familiarizado com essa notação, em JavaScript é possível recuperar o valor do atributo de um objeto usando-se o ponto seguido do nome do atributo, ou utilizar colchetes envolvendo o nome do atributo entre aspas. Por exemplo: **objeto.atributo** ou **objeto['atributo']**. Como capturamos a extensão em forma de string, faz mais sentido utilizar o formato com colchetes, ou seja, **types[extension]**.

Após configurarmos todos os parâmetros a serem enviados no cabeçalho HTTP de volta ao cliente, temos na linha 31 o método **end()** que encerra a comunicação com o cliente, enviando os dados solicitados. Na linha 33 utilizamos a mesma técnica empregada para enviar o conteúdo solicitado ao cliente; porém, por se tratar de uma função de callback de erro, informamos o código HTTP 404, que quer dizer que o arquivo solicitado não foi encontrado, e o método **end()** é passado sem qualquer argumento.

Obviamente existem muitos outros códigos HTTP, como 500, 304, 401, entre outros, cada qual com seu significado e interpretado de forma diferente pelo browser.

Agora que entendemos como nosso servidor está funcionando, vamos detalhar o funcionamento da função **fileHandler()**, utilizada nas últimas linhas deste. A implementação desta função é realizada no arquivo *filehandler.js*, conforme o código da **Listagem 5**.

Listagem 5. *filehandler.js* – responsável por ler os arquivos solicitados pelo cliente.

```
01. var fs = require('fs');
02.
03. module.exports = function(filename, successFn, errorFn) {
04.   fs.readFile(filename, function(err, data) {
05.     if(err) {
06.       errorFn(err);
07.     } else {
08.       successFn(data);
09.     }
10.   });
11. };
```

Com o Node.js é possível exportar qualquer coisa, até mesmo funções, e é justamente uma função que é exportada a partir do arquivo *filehandler.js*.

Na linha 1 importamos o módulo **fs** (filesystem) da API padrão do Node, e como o próprio nome diz, serve para manipulação de arquivos e diretórios, sendo responsável por quase todo o trabalho de nosso módulo.

Já sabemos o que está acontecendo na linha 3 e já sabemos para que servem os argumentos da função, mas recapitulando rapidamente, temos o nome do arquivo a ser lido, uma função de callback caso a leitura do arquivo ocorra com sucesso e outra função de callback caso a leitura do arquivo falhe.

O único método que usaremos do módulo **fs** é o **readFile()**. Este recebe uma string como primeiro argumento, contendo o caminho e nome do arquivo a ser lido, e o segundo argumento

é uma função de callback que recebe outros dois argumentos: um objeto de erro e os bytes do arquivo lido, caso a leitura seja bem sucedida.

O simples fato do argumento **err** possuir alguma informação que represente um valor válido, isto é, o valor da variável for diferente de **null**, **undefined**, uma string vazia, 0 ou **false**, significa que um erro ocorreu, como arquivo não encontrado, acesso negado ou qualquer outro tipo de problema ao recuperar os dados do arquivo solicitado.

Em nosso código, se qualquer erro ocorrer, disparamos a função de callback **errorFn()**, que deve ser fornecida pelo “usuário” da função **filehandler()**, no nosso caso, o nosso módulo **server**. Caso não haja erro, o conteúdo do arquivo é recuperado e passado como argumento à função de callback **successFn()**, também declarada no módulo **server**.

Listagem 6. Conteúdo de `config.json` – configurações utilizadas pela aplicação.

```
{
  "rootFolder": "./static",
  "defaultIndex": "/index.html",
  "types": {
    "htm": "text/html",
    "html": "text/html",
    "jpg": "image/jpeg",
    "jpeg": "image/jpeg",
    "png": "image/png",
    "css": "text/css",
    "js": "text/javascript",
    "json": "application/json"
  }
}
```

Na **Listagem 6** é exibido o conteúdo do arquivo `config.json`.

O arquivo `config.json` conta com três atributos: **rootFolder**, **defaultIndex** e **types**, referentes às variáveis que carregamos no `server.js`.

Note que, para que possa ser carregado corretamente pelo Node, `config.json` precisa ser um JSON válido, ou seja, deve respeitar as regras de um JSON, expostas a seguir:

- As chaves devem estar entre aspas duplas;
- Os valores que são strings precisam estar entre aspas duplas;
- Os valores numéricos não precisam estar compreendidos entre aspas;
- Os subobjetos devem estar entre chaves;
- As listas devem estar entre colchetes.

O atributo **rootFolder** contém o caminho a partir de onde o servidor web deve ler arquivos – no nosso caso, o diretório `/static`. Já o atributo **defaultIndex** contém o nome do arquivo padrão, caso o cliente não especifique o arquivo desejado, e o atributo **types** contém um objeto cujas chaves são extensões de arquivos, como por exemplo, `“html”`, e o valor para cada chave é o *content type*, como por exemplo, `“text/html”` como valor da chave `“html”`. Lembre-se que é através do objeto **types** que conseguimos saber o *content type* dos arquivos.

Você pode verificar exemplos da utilização do atributo **types** na linha 28 do arquivo `server.js`, onde isolamos a extensão do arquivo (ex.: `html`) e, a partir dessa extensão, obtemos o tipo do conteúdo, através da expressão **types[extension]**, onde, na ocasião, **extension** representa o valor `html`.

Uma vez que você tenha implementado todos os arquivos, podemos executar a aplicação com o comando `nodemon index.js`; ou, se você já havia executado o `nodemon` desde o começo, pode notar que ele recarrega a aplicação a cada vez que um arquivo é modificado.

Caso tudo ocorra como esperado, você verá uma tela parecida com a mostrada na **Figura 3**.

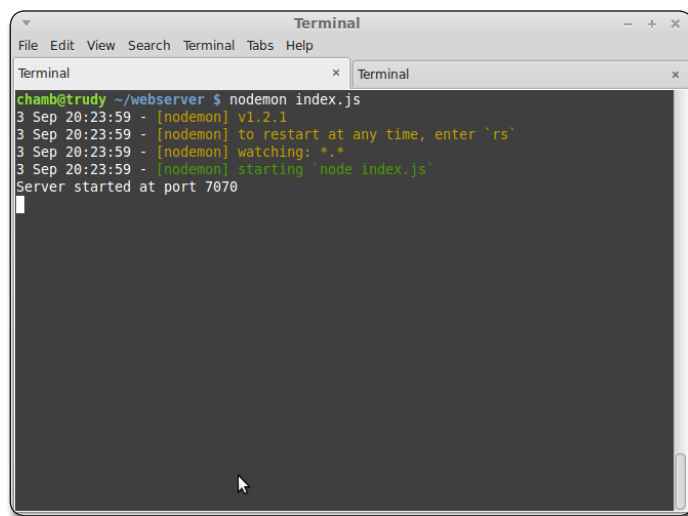


Figura 3. Nodemon executando a aplicação

Para ver o resultado de nosso desenvolvimento, basta abrir seu navegador e acessar `http://localhost:7070`, preferencialmente, com as ferramentas de desenvolvedor habilitadas na seção de rede. Para isso, no Google Chrome, por exemplo, navegue até o menu **Tools > Developer Tools**, onde teremos acesso a uma porção de informações sobre a página sendo visualizada, bem como os arquivos que foram baixados, tempo de download de cada um, verificar se algum arquivo falhou por qualquer motivo que seja e assim por diante.

Muitas vezes, enquanto estamos desenvolvendo uma aplicação web, pode ocorrer de não termos nenhuma informação visual no navegador em si. Porém, mantendo as ferramentas do desenvolvedor habilitadas, temos acesso a muitas informações por trás dos bastidores da comunicação entre navegador e servidor web.

Dando continuidade, após ter habilitado as ferramentas do desenvolvedor e acessado a URL do seu servidor, você deverá ver uma tela semelhante à exibida na **Figura 4**.

Caso você também tenha visto uma tela em branco com uma informação em vermelho no log de rede, por mais estranho que seja, o servidor web funcionou! Contudo, como não temos nenhum arquivo na pasta `static`, o servidor retornou um erro 404, conforme codificamos no `server.js`, o que é bastante recompensador, porém,

frustrante ao mesmo tempo, já que não exibimos nenhum conteúdo no browser.

Para resolver esse problema, vamos criar algum conteúdo a partir da pasta *static*, lembrando que já havíamos criado a estrutura de diretórios dentro dessa pasta. Deste modo, crie o arquivo *index.html* em *static* com o código demonstrado na **Listagem 7**.

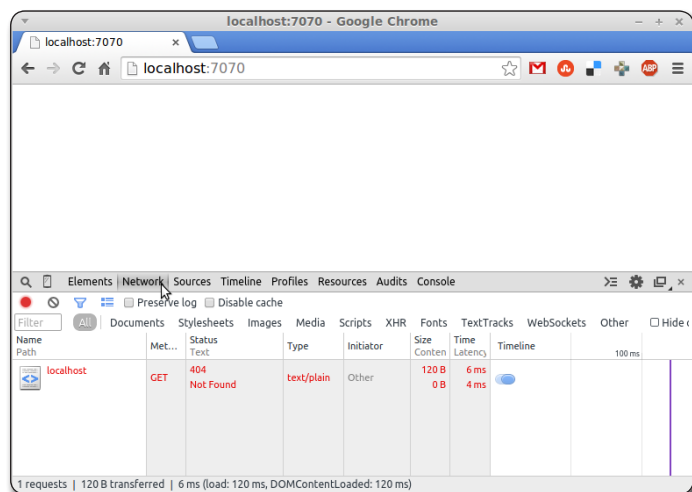


Figura 4. Primeiro teste do servidor, com resposta HTTP 404 – arquivo não encontrado

Listagem 7. Conteúdo da página *index.html*.

```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" type="text/css" href="css/style.css">
  <title>My web server</title>
</head>
<body>
  <h1>Index</h1>
  Go to <a href="teste.html">test</a>
</body>
</html>
```

Como você pode notar nesse pequeno arquivo HTML, importamos o estilo *css/style.css* e definimos um link com a famosa tag `<a>` para outra página HTML, chamada *teste.html*. Nesse momento, se recarregarmos o browser, seremos capazes de visualizar algum conteúdo, que seria todo o conteúdo da página *index.html* que acabamos de criar. Porém, como ainda não criamos o estilo *style.css*, nossa página aparecerá sem nenhuma formatação, como pode ser observado na **Figura 5**. Também, se tentarmos clicar no link, nenhum conteúdo será exibido, pois também ainda não criamos a página *teste.html*.

Agora já é possível ver o conteúdo de *index.html* que, embora não tenhamos especificado ao chamar a URL *http://localhost:7070*,

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

declaramos em nossa aplicação que, na ausência deste, o *index.html* seria automaticamente escolhido pelo servidor web.

Nas ferramentas do desenvolvedor do browser, localizadas na parte inferior do browser, se você acessar a aba de rede, poderá notar que temos o código de sucesso 200, referente ao *index.html*, e um erro 404, referente ao arquivo *style.css*, que ainda não definimos. Para solucionar esse problema, crie o arquivo *style.css* na pasta *css* com o conteúdo da **Listagem 8**.

Listagem 8. Conteúdo do arquivo *style.css*.

```
body {
  font-family: arial, helvetica;
  font-size: 16px;
}
```

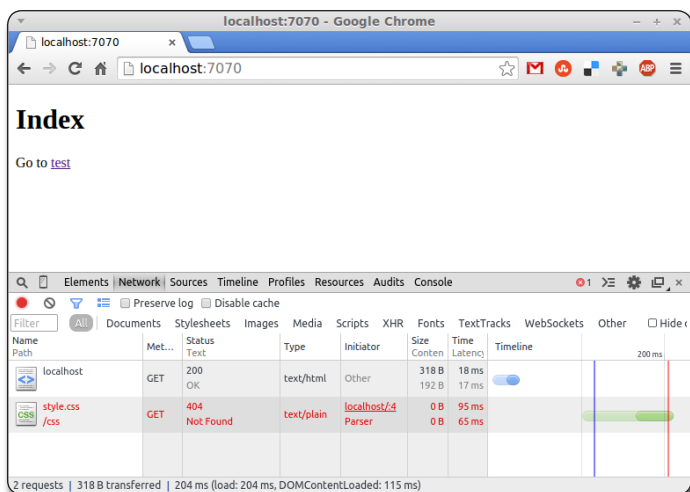


Figura 5. Página *index.html* sem o estilo CSS

Como pode ser notado, este CSS é um arquivo bem simples, apenas para modificar o estilo da fonte, visto que nosso objetivo aqui é testar o servidor web. Ao carregar a página novamente, você verá algo semelhante ao demonstrado na **Figura 6**.

Agora que definimos o estilo CSS para nossa página *index.html* e vimos que a página ficou mais bonita, tendo seu texto formatado, resta criar o conteúdo de *teste.html*. Com esta pendência, já sabemos que o link que aponta para *teste.html* irá falhar, então, antes de clicar no link, crie o arquivo *teste.html* na pasta *static*. Seu conteúdo pode ser visualizado na **Listagem 9**.

Note que assim como em *index.html*, a página *teste.html* importa o estilo *style.css*. Porém, além de importar tal arquivo de estilo, para demonstrar que nossa aplicação está pronta para carregar mais tipos de arquivo além de arquivos HTML e CSS, *teste.html* também importa um arquivo JavaScript, do caminho *script/app.js*, e ainda carrega uma imagem (disponível em *image/logo.png*), com a tag ``.

Para que seja possível carregar a imagem mencionada, copie-a para a pasta *image*. O código do script importado por *teste.html* pode ser visualizado na **Listagem 10**.

Listagem 9. Arquivo *teste.html*, chamado por um link da página *index.html*.

```
<!DOCTYPE html>
<html>
<head>
  <title>My test page</title>
  <link rel="stylesheet" type="text/css" href="css/style.css">
</head>
<body>
  <h1>Hello world</h1>
  

  <p>Go back to <a href="index.html">the index page</a>.</p>

  <script type="text/javascript" src="script/app.js"></script>
</body>
</html>
```

Listagem 10. Conteúdo do arquivo *app.js*

```
(function(window) {

  var d = window.document,
      b = d.body,
      bStyle = b.style;
      bStyle.backgroundColor = 'silver';

})(this);
```

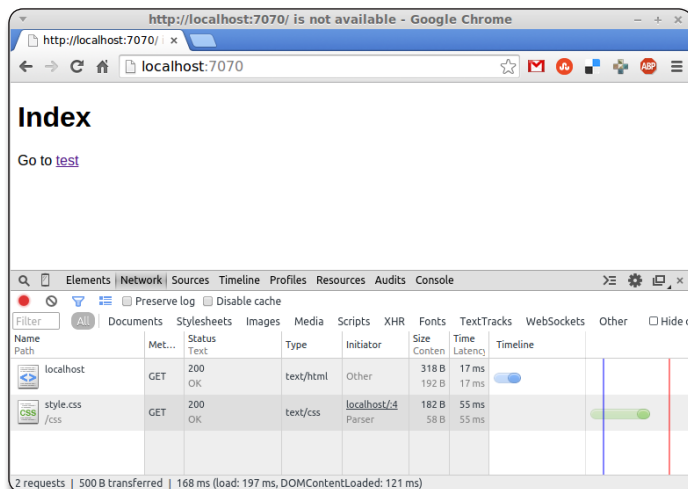


Figura 6. Resultado completo do acesso ao servidor web, com o estilo CSS carregado

Mais uma vez o arquivo JavaScript é bastante simples, pois serve apenas para testarmos se nosso servidor web está funcionando corretamente. Nesse caso, nosso script pinta o fundo da tela de cinza claro.

Agora que criamos um conteúdo simples para testar as funcionalidades do servidor web, volte ao browser, recarregue a tela e então clique no link para ver se a página de teste funcionou, conforme a **Figura 7**.

Ao constatar que tudo correu bem, nosso teste estará completo e nosso servidor web estará pronto.

Esta é uma aplicação simples em Node.js, porém, que pode não parecer tão simples para iniciantes devido ao número de tecnologias e conhecimentos prévios que o desenvolvedor precisa ter.

Além disso, mesmo sendo uma aplicação básica, ela demonstra diversos aspectos e recursos da tecnologia que você utilizará no dia a dia, caso siga em frente com o desenvolvimento com Node.js.

Caso esteja começando ou já teve algum contato inicial básico, uma boa sugestão é prestar bastante atenção nos seguintes pontos: primeiramente, a programação orientada a eventos, que como reforçamos diversas vezes ao decorrer do artigo, é o coração do Node.js; o sistema de carregamento de módulos, observando como exportar e importar corretamente módulos, objetos e funções, conteúdo demonstrado na importação e exportação dos módulos `server` e `fileHandler()`; e procure conhecer mais a fundo a biblioteca padrão do Node.js, todos os seus recursos, como as facilidades que economizam a implementação de coisas que já estão prontas, como os módulos `http` e `fs`, conforme abordado no decorrer do nosso estudo.

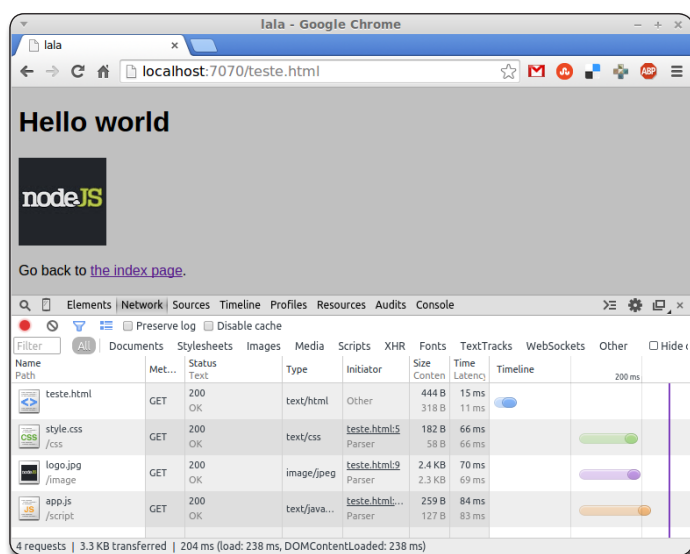


Figura 7. Resultado da exibição da página teste.html

Explorando outros recursos do Node.js

Agora que você já tem uma noção da capacidade do Node.js e da quantidade de possibilidades que a tecnologia oferece, tente aprimorar o seu servidor web.

Antes disso, no entanto, uma boa sugestão é começar realizando uma cópia de segurança do código desenvolvido. Feito isso, tente deixar o servidor mais sofisticado. Aqui vão algumas sugestões:

- Cadastre novos tipos de conteúdo no `config.json` e verifique se funcionou corretamente;
- Implemente outros códigos HTTP;
- Tente criar um sistema de log utilizando o módulo `fs`. Pesquise os métodos do módulo na documentação oficial.

Outra sugestão é tentar criar uma aplicação do zero, como um servidor de echo (você envia uma mensagem a um servidor e ele te responde com a mesma mensagem), ou faça experiências no CLI (*Command Line Interface*) do Node.js. Para isso, basta abrir um terminal e digitar o comando `node`. Você verá que o prompt de seu sistema operacional mudará para o CLI do Node.js, onde você pode começar a escrever qualquer código em JavaScript, que é a linguagem natural do Node.js. Para sair do CLI, basta digitar `CTRL+C` duas vezes.

Assim, você notará que o Node.js é uma tecnologia muito divertida de aprender e fácil de testar, e quanto mais você aprende, mais se interessa e acaba se aprofundando nos estudos.

Autor



Willian Carvalho

o.chambs@gmail.com

Programador desde 2000, trabalha com desenvolvimento para web e já passou por ASP, PHP e principalmente Java. Formado em Tecnologia da Informação pela FASP, atualmente é front-end engineer na Nagra, onde trabalha com JavaScript para aplicações para TV digital.

Links:

Documentação do Node.js

<http://nodejs.org/documentation/api/>

Informações sobre códigos HTTP

<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

Testes unitários em JavaScript: Conheça os principais frameworks - Parte 2

ESTE ARTIGO FAZ PARTE DE UM CURSO

Desenvolvimento front-end e testes unitários são dois termos extremamente recentes no universo de desenvolvimento de software se comparados a tantos outros que já estão bem fixados e definidos no mesmo meio. E quando falamos em unir ambas as tecnologias, chegamos em um ponto, muitas vezes, definido como tabu: será que realmente funciona implementar testes unitários em ambientes, linguagens e plataformas front-end? A resposta é mais que sim.

Nesta segunda parte do curso vamos explorar os frameworks JavaScript de testes de unidade que fornecem os recursos e atalhos necessários para simplificar mais ainda o tempo de desenvolvimento.

Testes e Frameworks

Você provavelmente sabe que o teste é bom, mas o primeiro obstáculo a superar quando se tenta escrever testes de unidade para o código do lado do cliente é a falta de quaisquer unidades reais. O código JavaScript é escrito para cada página de um site ou de cada módulo de uma aplicação e está intimamente misturado com a lógica de back-end e HTML. No pior dos casos, o código é completamente misturado com a HTML, como nos manipuladores de eventos in-line. Este é provavelmente o caso de quando nenhuma biblioteca JavaScript está sendo usada para qualquer abstração DOM. Escrever manipuladores de eventos em linha é muito mais fácil do que usar as APIs DOM para vincular esses eventos. Mais e mais desenvolvedores estão pegando uma biblioteca como jQuery para lidar com a abstração do DOM, o que lhes permite mover os eventos in-line para

Fique por dentro

Esse artigo é extremamente útil aos desenvolvedores que ainda têm dúvidas tanto à opção por escrever testes unitários para código JavaScript, como na hora de escolher um framework que atenda às necessidades sempre tão específicas de cada projeto.

Através de exemplos práticos examinaremos quais são as medidas mais adequadas para determinar qual framework melhor se encaixa na realidade de negócio de cada projeto, bem como analisaremos fatores muito importantes que diferem um framework de outro e o encaixam dentro de categorias específicas. Veremos também como migrar o seu código entre os mesmos frameworks, dadas as condições de configuração impostas por eles ante a conceitos como acoplamento, coesão e otimização.

os scripts específicos, ambos na mesma página ou até mesmo em um arquivo JavaScript separado. No entanto, colocar o código em arquivos separados não significa que ele está pronto para ser testado como uma unidade.

Mas afinal, o que é uma unidade? Na melhor das hipóteses, é uma função pura que você pode lidar de alguma forma: uma função que sempre lhe dá o mesmo resultado para uma dada entrada. Isso faz com que o teste de unidade fique muito mais fácil, mas na maior parte do tempo você precisará lidar com os efeitos colaterais, que aqui significam manipulações DOM.

De um jeito ou de outro, ainda é muito útil descobrir quais unidades podemos usar para estruturar o nosso código e para construir testes de unidade em conformidade.

Escolhendo um framework de testes

Existe uma série de propriedades a serem consideradas quando da escolha de um framework de testes, e este é um ponto extremamente necessário, pois tal escolha irá influenciar na forma como se programa os testes, os limites que os mesmos terão (dentro os

quais podemos encontrar limites de sintaxe, recursos, integrações, etc.), bem como outros fatores como documentação, apoio da comunidade e principalmente a força do framework no que diz respeito à certeza de que ele não será descontinuado.

As IDEs JavaScript não oferecem a mesma ajuda como as IDEs para linguagens tipadas e estáticas como o Java e o C#. Isso também impede a comunidade de padronizar uma estrutura de testes simples que possa oferecer recursos facilmente adaptáveis aos diferentes ambientes de execução.

Além disso, a depuração no passado sempre foi feita manualmente através de mensagens de alerta e extensões do navegador como o Firebug, por exemplo. Para o TDD é importante que a linguagem e o ambiente suportem padrões de teste, debug e refatoração, afim de perceber os padrões de projeto e teste como um todo.

Sintaxe

À medida que se investiga diferentes estruturas de teste, podemos notar que a sintaxe é um fator de diferenciação em todos eles. Este fator é bem pessoal e realmente vai depender do que faz o desenvolvedor se sentir confortável quando está utilizando uma ferramenta dessas. Isso vale não somente para frameworks de testes unitários, mas também para quaisquer tecnologias que façam uso de um código de comunicação, como linguagens de programação, ferramentas de gerenciamento de redes, dentre outros exemplos possíveis.

Por exemplo, o **JUnit** é mais que um framework de teste declarativo, pois sua API consiste de funções chamadas *test*, *equals*, *strictEqual*, *deepEqual*, etc.

Outra estrutura de teste, o **Mocha**, é mais parecido com o rspec (**BOX 1**) em que se lê de forma mais sentencial do que estrutural. Sua API consiste de funções chamadas *describe*, *it*, *assert.equal*, etc.

BOX 1. Rspec

É ferramenta de teste para a linguagem de programação Ruby. Nascido sob a bandeira do Desenvolvimento Behaviour-Driven, ele é projetado para fazer desenvolvimento orientado a testes uma experiência produtiva e agradável.

A sintaxe de um framework é criada para suportar uma metodologia escolhida, que em teoria será o TDD ou BDD (vistos no primeiro artigo deste curso).

A sintaxe TDD é puro JavaScript com métodos de biblioteca similares aos frameworks xUnit. Veja, por exemplo, o código apresentado na **Listagem 1** que exibe um exemplo no JSTestDriver. Repare que as sentenças **assert** são típicas dessa sintaxe.

As sintaxes BDD são mais diversificadas, mas elas têm em comum o fato de estarem mais próximas de uma linguagem natural, já que este é um dos princípios do BDD. Veja nas **Listagens 2 e 3** dois exemplos de diferentes sintaxes BDD.

Na **Listagem 2** nós temos uma sintaxe JavaScript, mas os métodos de biblioteca são nomeados muito proximamente aos princípios do

BDD. A sentença **assertEqual()** antes usada na **Listagem 1**, agora se apresenta como **expect().toEqual()**, o que é mais legível.

Na **Listagem 3** temos uma DSL (*Custom Domain-Specific Language*) que pertence ao JSpec. Aqui, outro passo importante é tomado para traduzir para uma linguagem mais natural.

Listagem 1. Exemplo da sintaxe TDD com JSTestDriver.

```
TestCase (" FizzBuzzDevMedia", {
  "test on 0 return 0": function () {
    var result = fizzbuzz (0);
    assertEquals (" Zero ", 0, result );
  }
};
```

Listagem 2. Exemplos da sintaxe BDD com Jasmine e JSpec BDD DSL.

```
describe (" FizzBuzz", function () {
  it(" return 0 on 0", function () {
    expect ( fizzbuzz (0) ).toEqual (0);
  });
});
```

Listagem 3. Exemplos da sintaxe BDD com JSpec BDD DSL.

```
describe 'FizzBuzz'
it 'should return null on null '
  fizzbuzz (0) . should . equal 0
end
end
```

Características

Quando você começa a testar em JavaScript, vai rapidamente encontrar-se em situações onde precisa-se que o framework de teste ajude o desenvolvedor a testar certos casos de uso. Nesse mesmo universo, alguns termos característicos são usados para ajudar a identificar tais cenários. Por exemplo, vejamos a lista de terminologias a seguir:

- **Spies** – São estruturas usadas para ajudar a detectar de onde o código chamou uma determinada função. Isso é útil para identificar a origem de erros e bugs que são encontrados no momento do teste;
- **Stubs** – Os stubs são muito semelhantes aos spies, exceto pelo fato de que eles contêm o comportamento já pré-definido. Por exemplo, suponha que se quer um método que espie algum código em específico e quando o mesmo for chamado, retorne 'abc'. Com os stubs esse tipo de comportamento é possível;
- **Servidor Fake** – Os servidores fake servem para facilitar o processo de simulação do envio de requisições que seria feito em um servidor real. Obviamente, como se trata de JavaScript, a única forma de fazer isso seria usando AJAX. Se o código emite chamadas AJAX e se quer falsificar a resposta da chamada AJAX, um servidor falso irá ajudá-lo mais facilmente a testar isso;
- **Relógio Fake** – Da mesma forma, podemos criar relógios fake para simular o comportamento do código ante mudanças no relógio usado pelo ambiente para extrair valores como data, tempo, intervalos, contadores, etc. Se o código reage de forma diferente baseado na data ou hora, então um relógio falso irá ajudá-lo a controlar exatamente qual é a hora, enquanto se faz os testes.

Suporte

O framework de teste é apoiado pela comunidade? Essa é uma das primeiras perguntas que se deve fazer quando se escolhe um. Por exemplo, o grunt (a ferramenta de ambiente de construção) suporta a execução do QUnit em um servidor PhantomJS. Isto torna mais fácil de escrever testes e tê-los executados automaticamente toda vez que se salvar o arquivo de teste.

Outra pergunta interessante é verificar se seria prático excluir os arquivos do projeto quando não quiser mais usá-los? Isso facilitaria a vida no momento em que optasse por trocar de framework, migrá-lo ou até mesmo combiná-lo com outros. Esse grau de granularidade determina se o framework é altamente acoplado ou não.

Ambiente

O ambiente de execução mais comum para as aplicações JavaScript é o browser, e em muitas situações o próprio JavaScript dependerá do objeto DOM dos browsers para executar de forma correta. Porém, pode-se executar os seus testes em qualquer ambiente que simule todas as características de uma engine de browser para JavaScript.

A ideia da execução fora do browser tornar o JavaScript mais útil tem ganhado popularidade à medida que as linguagens e aplicações server-side tem ficado maiores e mais complexas. Até esse ponto, a lógica interna das aplicações pode ser verificada em um ambiente que simule o JavaScript como se ele estivesse no lado cliente da execução ou um CMS (Content Management Systems), por exemplo, que irá lidar com toda a comunicação com o DOM por si só.

Atualmente, o Mozilla suporta o Rhino, por exemplo, como uma engine JavaScript implementada em Java. O Rhino provê uma execução generalizada do modelo JavaScript sem a API do browser DOM, o que facilita os testes e torna o mesmo apto a ser usado como ambiente padrão de testes.

Execução

A principal diferença quando estamos falando sobre execução dos testes, seria se o programador precisar consultar o browser ou se não receber nenhum resultado do mesmo. Efetuar um simples refresh na página do navegador para cada teste executado é algo fácil de implementar e configurar, mas não é rápido e simples o suficiente como o processo do TDD dita.

O problema irá crescer de acordo com o número de navegadores que precisam executar os testes, uma vez que todos eles precisarão ser abertos e recarregados automaticamente. A vantagem desse método é a transparência oferecida, pois a biblioteca por si só será um arquivo .js, facilitando o processo de exploração e extensão usando técnicas já conhecidas. Muitos dos executores de testes utilizarão esse tipo de teste *in-browser*, criando uma página HTML onde os arquivos de teste de código serão carregados através de tags **script**. Para bibliotecas de teste é possível que programadores criem seus próprios testes *in-browser* através do uso de métodos das mesmas bibliotecas.

A alternativa para esse tipo de estratégia é chamada de testes *headless*, que implica que os testes são executados a partir da linha de comando ou de dentro da IDE e os resultados são retornados para a interface. Eles podem ser executados em diferentes ambientes, alguns em Rhino, enquanto outros empurram os testes para um browser e exibem apenas os resultados retornados.

Por outro lado, esse tipo de teste pode ter mais demanda de configuração com atenção especial para as configurações locais de servidores e potenciais conexões a browsers ou máquinas remotas. Os arquivos de teste e código precisam de certas configurações para garantir as referências corretas e, em alguns casos, monitorar operações de pesquisa para as alterações de arquivos.

Biblioteca de Testes

A biblioteca de testes decide como o teste é feito, o que pode ser feito e como as demandas serão atreladas ao programador.

A biblioteca consiste de métodos para determinar o ciclo de vida de configuração, testes individuais, assim como agrupar os testes em suítes. Ela também tem métodos de comparação, chamados de mecanismos de asserção no TDD e qualificadores no BDD, o que determina a saída dos testes. O alcance e cobertura dos métodos de asserção e o ciclo de vida de configuração entre as bibliotecas pode variar, mas em geral como eles serão construídos depende dos objetivos de design suportados pela metodologia usada.

Quando estes fatores são cobertos, uma biblioteca de testes pode adicionar funcionalidades para garantir um teste facilitado, alta legibilidade e menos repetições para o desenvolvedor.

Com base em todos os fatores descritos, neste artigo iremos trabalhar essencialmente com três frameworks de testes: o **QUnit**, o **YUI Test** e o **JSTestDriver**. Eles representam os três frameworks mais usados e mais maduros em todos os fatores apresentados no momento, além de uma vasta documentação e atualizações constantes da comunidade.

Construindo os testes

Este artigo é para ajudá-lo com o problema mais difícil: extrair o código existente e testar as partes importantes, descobrindo e corrigindo bugs no código pelo meio do caminho.

O processo de extrair o código e colocá-lo em uma forma diferente, sem modificar o seu comportamento atual, é chamado de refatoração (ou *refactoring*). O refactoring é um excelente método de melhorar o design do código de um programa, porque qualquer mudança realmente pode modificar o comportamento do mesmo, sendo mais seguro então fazer somente quando os testes unitários estiverem prontos.

Este problema quer dizer que para adicionar testes ao código existente você tem que correr o risco de quebrar as coisas. Então, até que você tenha cobertura sólida com testes de unidade, você precisa continuar testando manualmente para minimizar esse risco.

Vejamos um exemplo prático na **Listagem 4** que exemplifica uma situação onde temos um código JavaScript básico com uma função **nossaData()** que irá receber um parâmetro com a mesma

data a ser processada. A função, por sua vez, irá capturar a data fornecida e verificar, através da criação de um novo objeto de data atual, a diferença de tempo entre uma e outra. O resultado será impresso na variável “diferença_dias” que irá guardar a mensagem correspondente ao total da diferença calculada entre ambas as datas.

Listagem 4. Exemplos de código de teste - busca por títulos.

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Exemplos de data mutilados</title>
<script>
function nossaData(time){
  var data = new Date(time || "");
  diff = ((new Date().getTime() - data.getTime()) / 1000),
  diferenca_dias = Math.floor(diff / 86400);

  if (isNaN(diferenca_dias) || diferenca_dias < 0 || diferenca_dias >= 31) {
    return;
  }

  return diferenca_dias == 0 && (
    diff < 60 && "Agora" ||
    diff < 120 && "1 minuto atrás" ||
    diff < 3600 && Math.floor( diff / 60 ) + " minutos atrás" ||
    diff < 7200 && "1 hora atrás" ||
    diff < 86400 && Math.floor( diff / 3600 ) + " horas atrás" ||
    diferenca_dias == 1 && "Ontem" ||
    diferenca_dias < 7 && diferenca_dias + " dias atrás" ||
    diferenca_dias < 31 && Math.ceil( diferenca_dias / 7 ) + " semanas atrás";
  )
}
window.onload = function(){
  var links = document.getElementsByTagName("a");
  for (var i = 0; i < links.length; i++) {
    if (links[i].title) {
      var data = nossaData(links[i].title);
      if (data) {
        links[i].innerHTML = data;
      }
    }
  }
};
</script>
</head>
<body>
<ul>
<li class="entry" id="post57">
<p>abc abc acb acb ...</p>
<small class="extra">
  Postado em <a href="/2014/12/abc/57/" title="2014-12-02T20:24:17Z">
    02 de Dezembro de 2014</a>
  por <a href="/john/">Sueila Valente</a>
</small>
</li>
<!-- Mais itens -->
</ul>
</body>
</html>
```

Se você executar esse exemplo, verá um problema: nenhuma das datas são substituídas. O código funciona, no entanto ele percorre todas as âncoras na página e checa por uma propriedade de título em cada uma. Se não houver uma, ele passa para a

função nossaData(), e se ela retornar um resultado, ele atualiza o innerHTML do link com o resultado.

Além disso, temos um outro problema: para qualquer data anterior a 31 dias, a função nossaData() apenas retornará “undefined” (implicitamente, com um simples return), deixando o texto da âncora como está. Então, para ver o que deveria acontecer, nós podemos programar uma data *hardcoded* (fixa no código). Veja na **Listagem 5** o resultado da nossa página após as alterações.

Perceba que nós mantemos o comportamento criado no **Listagem 4**, porém com a adição de uma data fixa na chamada da função dentro do onLoad da página. Essa adição poderá ser feita quantas vezes forem necessárias pelo testador-programador, de modo a atingir os diversos cenários de teste.

Listagem 5. Código de teste após alterações de data.

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Exemplos de data mutilados</title>
<script>
function nossaData(time, time2){
  var data = new Date(time || "");
  diff = ((new Date().getTime() - data.getTime()) / 1000),
  diferenca_dias = Math.floor(diff / 86400);
  if (isNaN(diferenca_dias) || diferenca_dias < 0 || diferenca_dias >= 31) {
    return;
  }

  return diferenca_dias == 0 && (
    diff < 60 && "Agora" ||
    diff < 120 && "1 minuto atrás" ||
    diff < 3600 && Math.floor( diff / 60 ) + " minutos atrás" ||
    diff < 7200 && "1 hora atrás" ||
    diff < 86400 && Math.floor( diff / 3600 ) + " horas atrás" ||
    diferenca_dias == 1 && "Ontem" ||
    diferenca_dias < 7 && diferenca_dias + " dias atrás" ||
    diferenca_dias < 31 && Math.ceil( diferenca_dias / 7 ) + " semanas atrás";
  )
}
window.onload = function(){
  var links = document.getElementsByTagName("a");
  for (var i = 0; i < links.length; i++) {
    if (links[i].title) {
      var data = nossaData("2014-12-02T20:24:17Z", links[i].title);
      if (data) {
        links[i].innerHTML = data;
      }
    }
  }
};
</script>
</head>
<body>
<ul>
<li class="entry" id="post57">
<p>abc abc acb acb ...</p>
<small class="extra">
  Postado em <a href="/2014/12/abc/57/" title="2014-12-02T20:24:17Z">
    02 de Dezembro de 2014</a>
  por <a href="/john/">Sueila Valente</a>
</small>
</li>
<!-- Mais itens -->
</ul>
</body>
</html>
```

Agora, as ligações devem dizer “2 horas atrás”, “Ontem” e assim por diante. Isso já é alguma coisa, mas ainda não é uma unidade testável real. Mesmo que tudo isso funcione, qualquer pequena alteração para a marcação provavelmente quebrará o teste, resultando em uma relação custo-benefício muito ruim para um teste como esse.

Em vez disso, vamos refatorar o código apenas o suficiente para ter algo que possamos testar a unidade.

Precisamos fazer duas mudanças para que isso aconteça:

1. passar a data atual para a função `nossaData()` como argumento, em vez de ter que usar apenas `new Date` e;
2. extrair a função para um arquivo separado, para que possamos incluir o código em uma página separada para os testes de unidade.

Remova o conteúdo da tag `script` interna à tag `head` e adicione o mesmo em um novo arquivo `.js`. Logo após, altere o nosso código fonte para o que está na **Listagem 6**.

Listagem 6. Código de teste refatorado.

```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <title>Exemplos de data mutilados</title>
  <script src="nossadata.js"></script>
</script>
  window.onload = function() {
    var links = document.getElementsByTagName("a");
    for ( var i = 0; i < links.length; i++ ) {
      if (links[i].title) {
        var date = nossaData("2014-12-02T20:24:17Z", links[i].title);
        if (date) {
          links[i].innerHTML = date;
        }
      }
    }
  };
</script>
</head>
<body>
<ul>
  <li class="entry" id="post57">
    <p>abc abc acb acb ...</p>
    <small class="extra">
      Postado em <a href="/2014/12/abc/57/" title="2014-12-02T20:24:17Z">
        02 de Dezembro de 2014</a>
      por <a href="/john/">Sueila Valente</a>
    </small>
  </li>
  <!-- Mais itens -->
</ul>
</body>
</html>
```

Na listagem tem o código com a melhoria explícita da adição do tag de `script` que agora importa o arquivo de JavaScript dinamicamente quando a página for executada. Com esta abordagem também não é preciso se preocupar com nenhuma mudança no HTML, pois o mesmo conseguirá enxergar as funções como antes.

Agora que temos algo para testar manualmente, vamos escrever alguns testes unitários e discutir logo em seguida, como mostra a **Listagem 7**.

Listagem 7. Testes de Unidade para o exemplo criado.

```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <title>Exemplos de data mutilados</title>
  <script src="prettydate.js"></script>
</script>
  function test(valor, esperado) {
    resultados.total++;
    var result = prettyDate("2014-12-02T22:25:00Z", valor);
    if (result !== esperado) {
      resultados.bad++;
      console.log("Esperado " + esperado + ", mas foi " + result);
    }
  }
  var resultados = {
    total: 0,
    bad: 0
  };
  test("2014/12/02 22:24:30", "Agora");
  test("2014/12/02 22:23:30", "1 minuto atrás");
  test("2014/12/02 21:23:30", "1 hora atrás");
  test("2014/12/01 22:23:30", "Ontem");
  test("2014/11/30 22:23:30", "2 dias atrás");
  test("2014/11/26 22:23:30", undefined);
  console.log("De " + resultados.total + " testes, " + resultados.bad + " falharam, "
    + (resultados.total - resultados.bad) + " passaram.");
</script>
</head>
<body>
  <!-- ... -->
</body>
</html>
```

Isto irá criar um framework de testes ad-hoc, usando apenas o console para a saída. Não tem nenhuma dependência ao DOM em tudo, então poderemos muito bem executá-lo em um ambiente JavaScript não-navegador, como o Node.js ou Rhino, extraindo o código na tag `script` para o seu próprio arquivo.

Se um teste falhar, irá produzir o esperado e real resultado para esse teste. No final, ele vai mostrar um resumo do mesmo com o número total de testes, os que falharam e os que foram bem-sucedidos.

Se todos os testes passarem você verá o seguinte no console:

De 6 testes, 0 falhou, 6 passaram.

Para ver como uma asserção falha se parece, nós podemos mudar alguma coisa no teste para quebrá-lo e ter algo como:

Esperado 2 dias atrás, mas foi 2 dias atrás.

De 6 testes, 1 falhou, 5 passaram.

Embora essa abordagem ad-hoc seja interessante como uma prova de conceito (você realmente pode escrever um executor de

teste em apenas algumas linhas de código), é muito mais prático usar um framework de testes unitários existente que proporciona uma melhor saída e mais infraestrutura para a escrita e organização de testes.

QUnit JavaScript

O QUnit é um framework JavaScript de testes unitários easy-to-use (fácil para uso) poderoso. É usado pelo jQuery, jQuery UI e o projeto jQuery Mobile e é capaz de testar qualquer código JavaScript genérico, incluindo o próprio.

Para usar o QUnit no seu projeto você precisa:

1. Baixar o arquivo qunit.css e o arquivo qunit.js (ver seção [Links](#)), diretamente do site oficial do projeto.
2. Criar uma página HTML contendo tags específicas que importam os arquivos de CSS e o JavaScript que você baixou.

Veja na [Listagem 8](#) como ficaria o nosso código de exemplo após as devidas alterações incluindo o QUnit.

Listagem 8. Código de exemplo após inclusão do QUnit.

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
<title>Exemplos de data mutilados refactorados</title>

<script src="nossadata.js"></script>
<script src="qunit.js"></script>
<script>
test("nossadata basics", function() {
  var now = "2014-12-02 22:25:00";
  equal(nossadata(now, "2014/12/02 22:24:30", "Agora"));
  equal(nossadata(now, "2014/12/02 22:23:30", "1 minuto atrás"));
  equal(nossadata(now, "2014/12/02 21:23:30", "1 hora atrás"));
  equal(nossadata(now, "2014/12/01 22:23:30", "Ontem"));
  equal(nossadata(now, "2014/11/30 22:23:30", "2 dias atrás"));
  equal(nossadata(now, "2014/11/26 22:23:30", undefined));
});
</script>
</head>
<body>
<div id="qunit"></div>
</body>
</html>
```

Junto com o clichê HTML de costume, temos três arquivos incluídos: dois arquivos para o QUnit (**qunit.css** e **qunit.js**) e os **nossadata.js** anterior.

Então, há um outro bloco de script com os testes reais. O método “test” é chamado uma vez, passando uma string como primeiro argumento (nomeação do teste) e passando uma função com o segundo argumento (que irá executar o código real para este teste). Este código define o variável “now”, que é reutilizada abaixo, e em seguida, chama o método “equal” algumas vezes com argumentos variados. O método de igualdade é uma das várias asserções que o QUnit oferece. O primeiro argumento é o resultado de uma chamada para `nossadata()`, com a variável “now” como o primeiro parâmetro e uma sequência de datas como segundo.

O segundo argumento para “equal” é o resultado esperado. Se os dois argumentos para “equal” são o mesmo valor, então a asserção vai passar; caso contrário, ela irá falhar.

Finalmente, no elemento do corpo temos um código específico do QUnit. Estes elementos são opcionais. Se estiver presente, o QUnit irá utilizá-los para a saída dos resultados do teste, como podemos ver na [Figura 1](#).

Com uma execução falha, teríamos algo parecido com a [Figura 2](#).



Figura 1. Resultado da execução dos testes - Interface do QUnit.

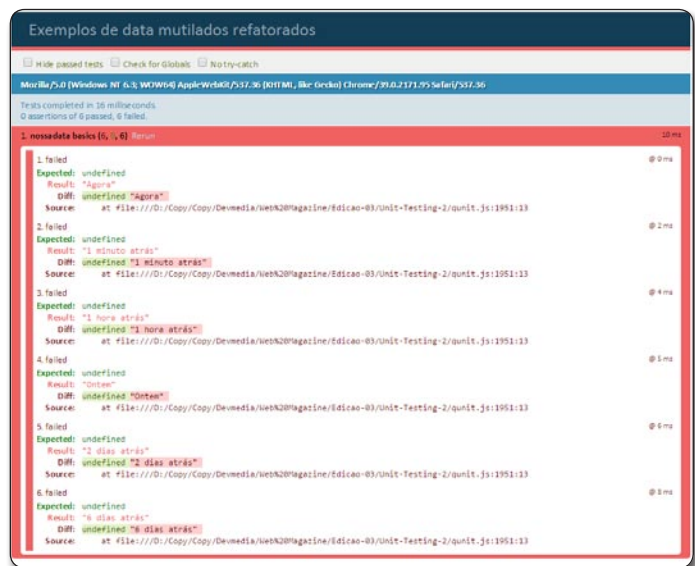


Figura 2. Resultado da execução falha dos testes - Interface do QUnit.

Como o teste contém uma asserção falhando, o QUnit não esconde os resultados para esse teste e podemos ver imediatamente o que deu errado. Junto com a saída dos valores previstos e reais temos uma comparação entre os dois, o que pode ser útil para a comparação de sequências maiores. Aqui, é bastante óbvio que deu errado.

Continuando a fase de refactorings, as asserções estão atualmente bastante incompletas, porque ainda não estamos testando as **n** semanas anteriores à variante. Antes de adicioná-lo, devemos considerar a refatoração do código de teste. Atualmente, estamos chamando `nossadata()` para cada afirmação e passando o argumento `now`. Poderíamos facilmente refatorar isso em um método de asserção customizado, como mostrado na [Listagem 9](#).

Nessa listagem extraímos a chamada para `nossadata()` na função `date`, colocando a variável `now` para dentro da função. Terminamos com apenas os dados relevantes para cada afirmação, tornando-se mais fácil de ler, enquanto a abstração subjacente permanece bastante óbvia.

Listagem 9. Método de assertion nossaData refatorado.

```
test("nossadata basics", function() {
  function date(then, expected) {
    equal(nossaData("2014/12/02 22:25:00", then), expected);
  }
  date("2014/12/02 22:24:30", "Agora");
  date("2014/12/02 22:23:30", "1 minuto atrás");
  date("2014/12/02 21:23:30", "1 hora atrás");
  date("2014/12/01 22:23:30", "Ontem");
  date("2014/11/30 22:23:30", "2 dias atrás");
  date("2014/11/26 22:23:30", undefined);
});
```

Testando a Manipulação do DOM

Agora que a função `nossaData()` foi testada o suficiente, vamos focar um pouco no exemplo inicial. Ao longo das mudanças na função `nossaData()`, o QUnit também selecionou alguns elementos do DOM e atualizou-os dentro do evento de load "window". Aplicando os mesmos princípios de antes, nós devemos estar aptos a refatorar o código e testá-lo. Além disso, vamos introduzir um módulo para essas duas funções, para evitar desordenar o namespace global e para sermos capazes de dar a essas funções individuais nomes mais significativos.

Vejamos então os códigos representados nas Listagens 10 e 11, que retratam esse cenário de mudanças.

Assim, as mudanças no arquivo `nossadata.js` resultam na Listagem 11.

Listagem 10. Mudanças de nomenclaturas no código.

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Exemplos de data mutilados refatorados</title>

<script src="nossadata.js" type="text/javascript"></script>
<script src="qunit.js" type="text/javascript"></script>
<link href="qunit.css" rel="stylesheet"/>
<script>
  test("nossadata.format", function() {
    function date(then, expected) {
      equal(nossaData("2014/12/02 22:25:00", then), expected);
    }
    date("2014/12/02 22:24:30", "atrásra");
    date("2014/12/02 22:23:30", "1 minuto atrás");
    date("2014/12/02 21:23:30", "1 hora atrás");
    date("2014/12/01 22:23:30", "Ontem");
    date("2014/11/30 22:23:30", "2 dias atrás");
    date("2014/11/26 22:23:30", undefined);
  });

  test("nossaData.update", function() {
    var links = document.getElementById("qunit-fixture").
    getElementsByTagName("a");
    equal(links[0].innerHTML, "28 de Janeiro de 2008");
    equal(links[2].innerHTML, "27 Janeiro de 2008");
    nossaData.update("2014/12/02 22:25:00Z");
    equal(links[0].innerHTML, "2 horas atrás");
    equal(links[2].innerHTML, "Ontem");
  });

  test("nossaData.update, one day later", function() {
    var links = document.getElementById("qunit-fixture").
```

O resultado da execução você pode conferir na Figura 3.

A nova função `nossaData.update` é um extrato do exemplo inicial, mas com o argumento `now` para passar a `nossaData.format`. O teste baseado no QUnit para essa função começa selecionando todos os elementos dentro de um elemento `#qunit-fixture`. Na marcação atualizada no elemento do corpo, o `<div id = "qunit-fixture"> ... </div>` é novo. Ele contém um extrato da marcação de nosso exemplo inicial, o suficiente para escrever testes úteis. Ao colocá-lo no elemento `#qunit-fixture`, você não precisa se preocupar com as mudanças do DOM de um teste afetando outros testes, porque o QUnit irá reiniciar automaticamente a marcação após cada teste.

Vamos dar uma olhada no primeiro teste de `nossaData.update`: depois de selecionar essas âncoras, duas asserções verificam se estes têm os seus valores de texto iniciais. Posteriormente, `nossaData.update` é chamada, passando uma data fixa por parâmetro (a mesma que nos testes anteriores).

Exemplos de data mutilados refatorados ■ noglobals ■ notrycatch

Hide passed tests

Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.95 Safari/537.36

Tests completed in 62 milliseconds.
14 tests of 14 passed, 0 failed.

1. `nossadata.format(0, 6, 6)` [Rerun](#)
2. `nossadata.update(0, 4, 4)` [Rerun](#)
3. `nossadata.update, one day later(0, 4, 4)` [Rerun](#)

Figura 3. Resultado da execução dos testes refatorados

```
getElementsTagName("a");
equal(links[0].innerHTML, "28 de Janeiro de 2008");
equal(links[2].innerHTML, "27 de Janeiro de 2008");
nossaData.update("2014/12/02 22:25:00Z");
equal(links[0].innerHTML, "Ontem");
equal(links[2].innerHTML, "2 dias atrás");
});
</script>
</script>
</head>
<body>
<div id="qunit"></div>
<div id="qunit-fixture">
  <ul>
    <li class="entry" id="post57">
      <p>abc abc acb acb ...</p>
      <small class="extra">
        Postado em <a href="/2014/12/abc/57/" title="2014-12-02T20:24:17Z">
          02 de Dezembro de 2014</a>
        por <a href="/john/">Sueila Valente</a>
      </small>
    </li>
    <li class="entry" id="post57">
      <p>abc abc acb acb ...</p>
      <small class="extra">
        Postado em <a href="/2014/12/abc/57/" title="2014-12-01T20:24:17Z">
          01 de Dezembro de 2014</a>
        por <a href="/john/">Sueila Valente</a>
      </small>
    </li>
  </ul>
</div>
</body>
</html>
```

Listagem 11. Mudanças de organização no código JS do projeto.

```
var nossaData = {
  format: function(now, time){
    var date = new Date(time || "");
    diff = (((new Date(now)).getTime() - date.getTime()) / 1000),
    day_diff = Math.floor(diff / 86400);

    if (isNaN(day_diff) || day_diff < 0 || day_diff >= 31) {
      return;
    }

    return day_diff === 0 && (
      diff < 60 && "Agora" ||
      diff < 120 && "1 minuto atrás" ||
      diff < 3600 && Math.floor( diff / 60 ) + " minutos atrás" ||
      diff < 7200 && "1 hora atrás" ||
      diff < 86400 && Math.floor( diff / 3600 ) + " horas atrás" ||
      day_diff === 1 && "Ontem" ||
      day_diff < 7 && day_diff + " dias atrás" ||
      day_diff < 31 && Math.ceil( day_diff / 7 ) + " semanas atrás";
    ),

  update: function(now) {
    var links = document.getElementsByTagName("a");
    for ( var i = 0; i < links.length; i++ ) {
      if (links[i].title) {
        var date = nossaData.format(now, links[i].title);
        if (date) {
          links[i].innerHTML = date;
        }
      }
    }
  }
};
```

Depois, mais duas asserções são executadas, agora verificando se a propriedade `innerHTML` destes elementos têm a data formatada corretamente, "2 horas atrás" e "Ontem".

Outros Frameworks

YUI Library

O **Yahoo! User Interface Library (YUI)** é uma biblioteca open-source feita em JavaScript para a construção de aplicações web interativas ricas usando técnicas como Ajax, DHTML, e DOM scripting. O YUI inclui vários recursos do núcleo do CSS e está disponível sob a licença BSD. O desenvolvimento do projeto começou em 2005 e algumas propriedades do Yahoo!, como o My Yahoo! e a página Yahoo!, começaram a usar o YUI no verão daquele ano. Este foi liberado para uso público em fevereiro de 2006. Foi desenvolvido ativamente por uma equipe central de engenheiros do Yahoo! e é hoje um projeto amplamente aceito e reconhecido pela comunidade.

O YUI Test, um componente dentro da biblioteca YUI, é um framework de testes unitários extenso e completo. Para começar com o YUI Test, você precisa:

1. Importar o YUI no cabeçalho da página HTML, como se segue:
2. `<script src="http://yui.yahooapis.com/3.18.1/build/yui/yui-min.js"></script>`
3. No arquivo de script de teste, instanciar a função YUI. Carregar os módulos necessários, test e console, como mostrado na **Listagem 12**.

Listagem 12. Carregar teste e console dos módulos YUI.

```
YUI().use("test","console",function(Y){
  // Casos de teste aqui
});
```

Nota

Veja na seção **Links** o link necessário para efetuar o download do YUI Test. Estamos usando a versão 3.18.1, a mais recente na data de escrita deste artigo.

O módulo de teste é claramente necessário para fins de teste. O módulo de console não é obrigatório, mas o exemplo que faremos vai usá-lo para imprimir os resultados. Os casos de teste vão dentro da chamada de retorno, com a instância `Y` global como argumento.

O YUI Test usa o construtor **Y.Test.Case()** para instanciar um novo caso de teste e o construtor **Y.Test.Suite()** para instanciar um conjunto de testes. Um conjunto de testes (ou suíte de testes), de forma semelhante ao JUnit, contém vários casos de teste. Você pode adicionar casos de teste para uma suíte de testes usando o método **add()**.

Crie uma nova página HTML e adicione o código a seguir nela:

```
YUI().use("test","console",function(Y){
```

Esse código representa o repositório web que responderá aos comportamentos de alguns testes.

Em seguida, crie um novo arquivo JavaScript e adicione o conteúdo da **Listagem 13**. Nele podemos ver a adição de algumas funções de medição de temperatura e conversão. Apenas funções básicas para simularmos os comportamentos no YUI Test.

Listagem 13. Código JavaScript das conversões de temperatura.

```
function converterDeCelsiusParaFahrenheit(c){
  var f = c * (9/5) + 32;
  return f;
}

function converterDeFahrenheitParaCelsius(f){
  var c = (f - 32) * (5/9);
  return c;
}
```

E por fim, crie um novo arquivo `.js` "test_temp.js" para guardar o código de teste para as mesmas funções, como mostra a **Listagem 14**.

Se estiver atento, verá que o código apresentado nessa listagem está em QUnit. Isso quer dizer que a representação dele aqui se dará apenas com o objetivo de comparar os resultados e efeitos de implementação em ambos os frameworks. Não exibiremos o resultado no QUnit para não estender muito o artigo.

Agora vejamos como fica o nosso código fonte usando o YUI Test. A **Listagem 15** mostra como criar uma suíte e um caso de teste para o teste em questão usando o referido framework.

Listagem 14. Suíte de testes para as funções de conversão.

```
module ("Conversão de Temperatura")

test("conversão para F", function(){
  var actual1 = converterDeCelsiusParaFahrenheit(20);
  equal(actual1, 68, ?Valor não correto?);

  var actual2 = converterDeCelsiusParaFahrenheit(30);
  equal(actual2, 86, ?Valor não correto?);
})

test("conversão para C", function(){
  var actual1 = converterDeFahrenheitParaCelsius(68);
  equal(actual1, 20, ?Valor não correto?);

  var actual2 = converterDeFahrenheitParaCelsius(86);
  equal(actual2, 30, ?Valor não correto?);
})
```

Listagem 15. Suíte de testes para funções de conversão com o YUI Test.

```
YUI().use("test", "console", function (Y) {
  var suite = new Y.Test.Suite("Suíte de Conversão de Temperatura");

  // add o caso de teste
  suite.add(new Y.Test.Case({
    name: "Conversão de Temperatura",

    setUp : function () {
      this.celsius1 = 20;
      this.celsius2 = 30;

      this.fahrenheit1 = 68;
      this.fahrenheit2 = 86;
    },

    testConversaoCtoF: function () {
      Y.Assert.areEqual(this.fahrenheit1, converterDeCelsiusParaFahrenheit(
        this.celsius1));

      Y.Assert.areEqual(this.fahrenheit2, converterDeCelsiusParaFahrenheit(
        this.celsius2));
    },

    testConversaoFtoC: function () {
      Y.Assert.areEqual(this.celsius1, converterDeFahrenheitParaCelsius(
        this.fahrenheit1));

      Y.Assert.areEqual(this.celsius2, converterDeFahrenheitParaCelsius(
        this.fahrenheit2));
    }
  }));
});
```

Vejamos algumas observações sobre a listagem:

- O método `setUp()` está disponível. O YUI Test fornece os métodos `setUp()` e `tearDown()` a nível de caso de teste e suíte de testes.
- Os nomes de métodos começam com a palavra `test` e contêm assertivas.
- O exemplo utiliza o tipo de assertiva `Y.Assert.areEqual()`, que é semelhante à função de `equal()` no QUnit.

• O YUI Test oferece uma vasta gama de métodos para asserções, tais como:

- `Y.Assert.areSame()`, que é o equivalente ao `strictEqual()` no QUnit.
- Tipo de dados assertivos: `Y.Assert.isArray()`, `Y.Assert.isBoolean()`, `Y.Assert.isNumber()`, e assim por diante.
- Asserções de valor especial: `Y.Assert.isFalse()`, `Y.Assert.isNaN()`, `Y.Assert.isNull()`, e assim por diante.

Para executar os testes no YUI Test use o objeto **Y.Test.Runner**. Você precisa adicionar casos de teste ou suítes de teste a esse objeto e então chamar o método `run()` para rodar os testes. O código a seguir mostra como fazer isso:

```
Y.Test.Runner.add(suite);
Y.Test.Runner.run();
```

JSTestDriver

O JSTestDriver visa ajudar desenvolvedores JavaScript a usar boas práticas de TDD e fazer testes de unidade de escrita tão facilmente como o que já existe hoje para Java com JUnit, por exemplo. Com a ferramenta poderosa JSTestDriver (JSTD) você pode executar JavaScript na linha de comando em vários navegadores. Após baixar o JSTestDriver (ver seção **Links**) ele vem com um arquivo JAR que permite iniciar um servidor, capturar um ou vários navegadores, e executar testes nos navegadores. Você não precisa de um executor de HTML, como acontece com outros frameworks, mas é necessário um arquivo de configuração. A **Listagem 16** mostra um exemplo de arquivo de configuração.

Listagem 16. Exemplo de arquivo de configuração - `jsTestDriver.conf`.

```
server: http://localhost:4224
load:
  - js/src/*.js
test:
  - js/test/*.js
```

O arquivo de configuração é escrito em YAML, que é um bom formato para arquivos de configuração. Ele contém informações sobre o servidor para executar, bem como a localização dos arquivos de código fonte e teste.

Para executar os testes com JSTD basta:

1. Iniciar o servidor de teste. A partir da linha de comando, vá para a pasta onde o arquivo `jsTestDriver.jar` foi salvo e execute o seguinte comando:

```
java -jar JsTestDriver-1.3.3d.jar -port 4224
```

A porta especificada na **Listagem 16** deve ser a mesma que a especificada no arquivo de configuração. Por padrão, o JSTD procura pelo arquivo `jsTestDriver.conf` no mesmo diretório onde o arquivo JAR reside.

2. Registre um ou vários navegadores para o servidor, copiando e colando a URL `http://localhost:4224/capture` nos navegadores em teste.

Teste o mesmo código fonte que você usou para os exemplos anteriores (conversão de temperaturas), mas desta vez usando a sintaxe JSTD. A **Listagem 17** mostra como converter os casos de teste a partir da **Listagem 14** para o QUnit e da **Listagem 15** para o YUI Test.

Listagem 17. Exemplo de arquivo de configuração `jsTestDriver.conf`.

```
TestCase("Conversão de Temperatura", {
  setUp: function () {
    this.celsius1 = 20;
    this.celsius2 = 30;

    this.fahrenheit1 = 68;
    this.fahrenheit2 = 86;
  },

  testConversaoCtoF: function () {
    assertSame(this.fahrenheit1, converterDeCelsiusParaFahrenheit(this.celsius1));
    assertSame(this.fahrenheit2, converterDeCelsiusParaFahrenheit(this.celsius2));
  },

  testConversaoFtoC: function () {
    assertSame(this.celsius1, converterDeFahrenheitParaCelsius(this.fahrenheit1));
    assertSame(this.celsius2, converterDeFahrenheitParaCelsius(this.fahrenheit2));
  }
});
```

O código não é muito diferente da versão YUI. O JSTD usa a função `TestCase()` para definir um caso de teste. É possível definir pode definir os métodos de teste usando uma declaração in-line ou você pode estender o protótipo da instância `TestCase`. Os métodos `setup()` e `tearDown()` estão disponíveis para cada caso de teste.

Para executar os testes, basta executar o seguinte comando:

```
java -jar JsTestDriver-1.3.3d.jar --tests all
```

A **Figura 4** mostra o resultado da execução destes testes.

```
*****
Total 6 tests (Passed: 6; Fails: 0; Errors: 0) (2.00 ms)
Chrome 16.0.912.63 Mac OS: Run 2 tests (Passed: 2; Fails: 0; Errors: 0) (1.00 ms)
Safari 534.52.7 Mac OS: Run 2 tests (Passed: 2; Fails: 0; Errors: 0) (2.00 ms)
Firefox 7.0.1 Mac OS: Run 2 tests (Passed: 2; Fails: 0; Errors: 0) (0.00 ms)
```

Figura 4. Resultados dos testes JSTD

O JSTD também se integra bem com o seu sistema de integração contínua para fazer parte de sua construção contínua. Ele oferece integração com IDEs como Eclipse (plug-in) ou TextMate (bundle).

Com o foco atual no lado do cliente das aplicações web, os testes unitários com código JavaScript tornaram-se essenciais. Há alguns frameworks que podem ajudar você a realizar seus objetivos, e este artigo analisou três dos principais frameworks mais populares disponíveis no mercado hoje: o QUnit, o YUI Test e o JSTestDriver.

Os frameworks que vimos aqui representam apenas uma pequena parte do universo de opções disponíveis para testes de unidade. O mais importante é que você sempre entenda os conceitos para que assim possa aplicá-los na prática junto aos mesmos.

Existem inúmeras outras possibilidades de melhoria para os códigos aqui apresentados e você pode melhorá-los de diversas formas. Além disso, outros frameworks podem ser explorados dadas as suas condições. Leia mais sobre os frameworks, faça testes e mais testes até se sentir familiarizado com tais tecnologias. Com os fontes disponibilizados para download você terá um material rico para avançar seus estudos na criação de testes de unidade com JavaScript.

Autora



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 oficial do QUnit

www.qunitjs.com/

Página oficial de download do JS QUnit

www.code.jquery.com/qunit/qunit-1.16.0.js/

Página oficial de download do CSS QUnit

www.code.jquery.com/qunit/qunit-1.16.0.css

How-to: Executar grunt com QUnit

www.github.com/gruntjs/grunt-contrib-qunit

Site oficial do projeto YUI

www.yuilib.com/

Página do projeto js-test-driver no Google Code

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

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