Desenvolvimento Web com AngularJS

2017

SUMÁRIO

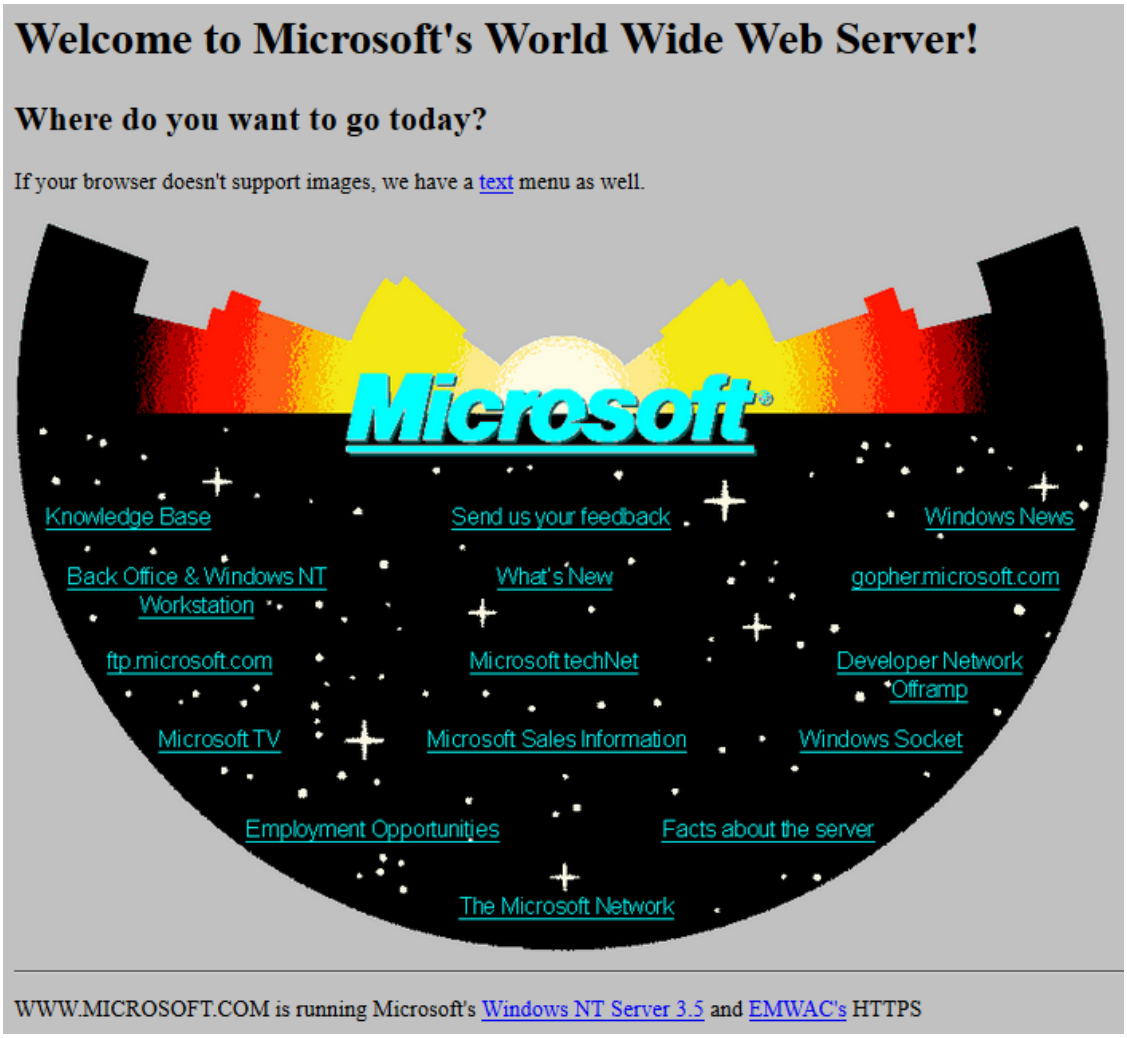
WORLD WIDE WEB

Surgimento

A *World Wide Web*, também conhecida como *WWW* ou apenas *web*, foi criada para exibição de documentos em hipermídia interligados na internet, a fim de tornar mais fácil o compartilhamento de documentos de pesquisas.

O primeiro site criado pode ser visto no seguinte endereço:
http://info.cern.ch/hypertext/WWW/TheProject.html

Site da Microsoft em 1994:



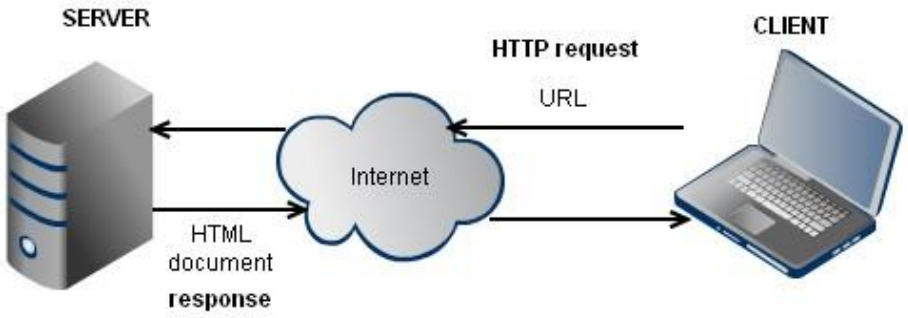
Como funciona

Para exibir uma página de internet é necessário um navegador compatível, o qual realizará uma requisição web e exibirá o retorno para o usuário.

Um site é hospedado em um servidor HTTP, comumente utilizado para fornecimento de conteúdo na internet.

A grosso modo, o usuário digita um endereço no navegador e este envia uma requisição ao servidor onde o site está hospedado. O servidor processa essa informação e retorna o conteúdo requisitado. O navegador recebendo essa informação, processo o documento e exibe para o usuário.

Ilustração de uma requisição web:



WEB 2.0 e SPAs

PÁGINAS DINÂMICAS

Até então, os documentos exibidos na internet eram apenas páginas estáticas e não era possível ter seu conteúdo alterado dinamicamente.

Embora essas páginas suprissem a necessidade do momento, com o passar do tempo fomos tendo a necessidade de páginas mais ricas, dinâmicas e interativas. Criando um conceito hoje chamado de WEB 2.0.

WEB 2.0

Web 2.0 foi um termo popularizado pela O'Reilly Media, uma companhia de mídia americana, para conceitualizar a *web como plataforma*.

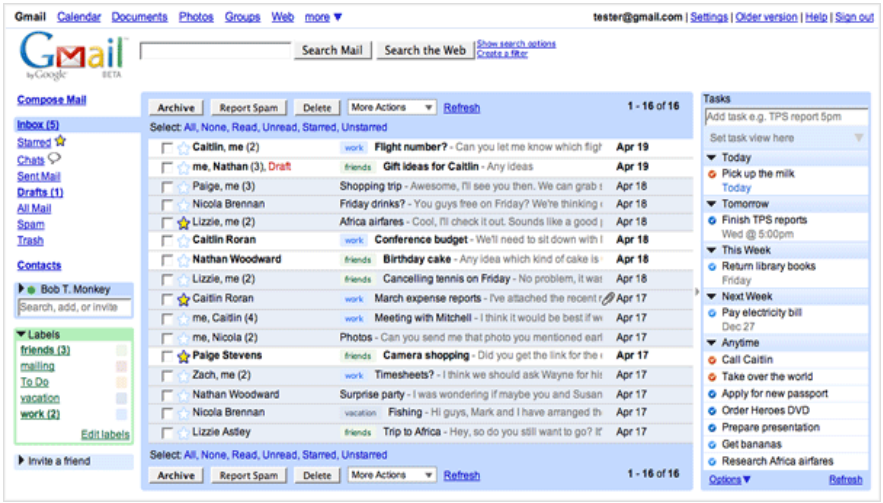
SINGLE PAGE APPLICATIONS

Com a rápida ascenção da Web 2.0, as páginas foram ficando cada vez mais complexas, exigindo maior interação com o usuário.

Para facilitar o desenvolvimento de aplicações na web com alta complexidade, foi criado o conceito de *Single Page App*, ou apenas *SPA*.

Como o próprio nome diz, SPA é basicamente uma aplicação que tem apenas uma página de entrada. Dessa forma o usuário não precisa esperar a aplicação recarregar a página inteira após cada interação, melhorando a experiência do usuário, performance e, por conta de aspectos tecnológicos, facilidade de manutenção.

O Google foi um dos pioneiros nesse conceito, tendo o Gmail como seu carro chefe.



Camadas

Uma página web é basicamente dividida em 3 camadas: exibição, estilo e comportamento.

HTML é a linguagem usada para definir a estrutura da página, contendo a informação propriamente dita.

CSS é usada para definir o estilo da página, como tamanho das fontes, cores, etc.

JavaScript é responsável pela parte comportamental.

HYPERTEXT MARKUP LANGUAGE

Introdução

Assim como a XML (usada para estrutura de dados, comunicação, entre outros), a *HTML* é derivada da *SGML* (Standard Generalized Markup Language).

O HTML é composto por *tags* (marcadores), delimitadas pelos sinais de menor < e maior >.

Exemplo de um parágrafo em HTML:

```
<p>Um parágrafo em HTML é representado pela tag *p*</p>
```

Estrutura

Uma página HTML é composta de cabeçalho e corpo, sendo representados pelas palavras head e body, respectivamente.

No cabeçalho inserimos informações como o título e outras informações usadas pelos navegadores.

No corpo definimos o documento a ser visto pelo usuário, onde é possível definir toda a estrutura a ser exibida.

Exemplo de uma página HTML:

```
<html>
  <head>
    <title>Título da página</title>
  </head>
  <body>
    <h1>Título exibido no corpo da página</h1>
    <p>Um parágrafo</p>
  </body>
</html>
```

O exemplo acima será renderizado como a seguir:

Elementos e tags existentes

Seguem os elementos e suas tags HTML suportados pelos navegadores atuais:

Nome	Significado
<a>	Âncora: usado para ligar a outro recurso web
<abbr>	Abreviação
<address>	Endereço
<area>	Área
<article>	Elemento artigo
<aside>	Elemento à parte
<audio>	Conteúdo de som
<b>	Texto em negrito
<base>	Elemento base
<bdo>	Representa explicitamente a direção do texto
<blockquote>	Bloco de citação
<body>	Corpo da página
 	Insere uma quebra de linha
<button>	Botão
<canvas>	Utilizado para a renderização de gráficos
<caption>	Legenda da tabela
<cite>	Citação
<code>	Texto de código computacional
<col>	Coluna
<colgroup>	Grupo de colunas
<command>	Botão de comando
<datalist>	Lista suspensa
<dd>	Definição da descrição
<del>	Texto suprimido
<details>	Detalhes
	Bloco de documento
<dl>	Lista de definição
<dt>	Termo de definição

	Texto enfatizado
<embed>	Elemento embutido
<fieldset>	Grupo de campos
<figcaption>	Legenda de uma figura
<figure>	Figura
<footer>	Rodapé da página
<form>	Formulário
<h1> à <h6>	Títulos, onde o valor 1 representa um título maior do que o valor 6
<head>	Cabeçalho principal do documento
<header>	Cabeçalho principal da página
<hgroup>	Grupo de títulos
<hr>	Linha horizontal
<html>	Raiz de um documento HTML
<i>	Texto em itálico
<iframe>	Janela de navegação aninhada
<img>	Inclui uma imagem
<input>	Campo de entrada
<ins>	Texto inserido
<kbd>	Texto do teclado
<label>	Etiqueta
<legend>	Título de um grupo de controles formulário
	Item de uma lista
<link>	Link de recursos
<map>	Mapa de imagens
<mark>	Marcação
<menu>	Menu de comandos
<meta>	Define um meta-informação
<meter>	Elemento de medida
<nav>	Elemento de navegação
<noscript>	Exibido se scripts estiverem desativados
<object>	Objeto incorporado
	Lista ordenada

<optgroup>	Grupo de opções
<option>	Opção
<output>	Resultado/saída de um cálculo
<p>	Parágrafo
<param>	Define parâmetro de plugins invocados pelos elementos object, não representando nada por si só
<pre>	Texto pré-formatado
<progress>	Progresso da conclusão de uma ação, como por exemplo um download
<q>	Breve citação
<ruby>	Anotação ruby
<rp>	Parênteses de texto ruby
<rt>	Componentes de texto ruby
<samp>	Amostra de programa ou sistema de computação
<script>	Representa um script
<section>	Seção do documento
<select>	Lista selecionável
<small>	Texto pequeno
<source>	Permite indicar diversas fontes para elementos de mídia
	Utilizado para um elemento dentro do fluxo de texto
	Texto grande
<style>	Define um estilo
<sub>	Texto com subscrição
<sup>	Texto sobrescrito
<tbody>	Corpo da tabela
<td>	Célula da tabela
<textarea>	Área de texto
<tfoot>	Rodapé da tabela
<th>	Célula de cabeçalho da tabela
<thead>	Representa o cabeçalho da tabela
<time>	Indica horas
<title>	Título da pagina
<tr>	Linha da tabela

	Lista não ordenada
<var>	Variável
<video>	Elemento de vídeo ou filme

Comentários

É comum colocarmos algum tipo de explicação no código, a fim deste ser facilmente entendido por outra pessoa.

Em HTML, um comentário é iniciado com <!-- e finalizado com -->. Tudo entre essas tags será tratado como comentário, ou seja, não será renderizado pelo navegador.

Exemplo de comentário:

```
<!-- comentário -->
```

HTML5

SURGIMENTO

Em 2004, o WHATWG (Web Hypertext Application Technology Working Group) começou a trabalhar em um novo padrão HTML, enquanto a W3C (World Wide Web Consortium) concentrava seus esforços no XHTML. Em 2009 os dois grupos se uniram e trabalharam juntos no desenvolvimento do HTML5.

Com o crescimento de dispositivos móveis, como o iPhone, e a morte do Flash, o HTML5 foi ganhando força até tomar conta do mercado e obter suporte em todos os navegadores atuais. Substituindo a necessidade de qualquer plugin de terceiros para a criação de páginas web ricas em conteúdo e até mesmo animações.

DOCTYPE

DOCTYPE é uma tag usada para informar ao navegador a versão de HTML na página em questão.

Essa tag deve estar obrigatoriamente acima da tag de início da página, `html`. Conforme exemplo abaixo:

```
<!DOCTYPE html>
<html>
  <head></head>
  <body></body>
</html>
```

A declaração da DOCTYPE no HTML5 é extremamente simples, diferentemente de como era antes.

Exemplo de DOCTYPE antes do HTML5:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
```

DOCTYPE em HTML5:

```
<!DOCTYPE html>
```

HTML SEMÂNTICO

Antes do HTML5, tentávamos identificar as partes de um documento usando classes de CSS ou IDs.

Exemplo:

```
<body>
  <div id="cabecalho">
    <!-- Conteúdo do cabeçalho -->
  </div>
  <div class="secao" id="principal">
    <!-- Conteúdo principal -->
  </div>
  <div class="secao" id="destaques">
    <!-- Painéis com destaques -->
  </div>
  <div id="rodape">
    <!-- Conteúdo do rodapé -->
  </div>
</body>
```

Agora com essa *nova* versão de HTML, existem novas tags que podem identificar cada parte da página.

Exemplo:

```
<body>
  <header>
    <!-- Conteúdo do cabeçalho -->
  </header>
  <section id="principal">
    <!-- Conteúdo principal -->
  </section>
  <section id="destaques">
    <!-- Painéis com destaques -->
  </section>
  <footer>
    <!-- Conteúdo do rodapé -->
  </footer>
</body>
```

Essas novas tags não trazem nenhuma diferença no visual, apenas carregam um significado semântico atrelado a elas.

Com isso algum leitor de tela, por exemplo, é capaz de ler o código e identificar as partes julgadas importantes por ele.

Os motores de busca também podem utilizar esse código semântico para buscar e mostrar ao usuário o conteúdo de seu real interesse mais facilmente.

As *divs* não deixarão de existir pois ainda cumprem bem seu papel, mas não são mais necessárias para identificar a estrutura semântica da página.

Para explicar todo esse conceito de semântica, o WHATWG (Web Hypertext Application Technology Working Group) fez um documento sobre o assunto. O qual pode ser conferido no seguinte endereço: **https://html.spec.whatwg.org/multipage/semantics.html**

CASCADING STYLE SHEETS

Introdução

Cascading Style Sheets (CSS) foi proposta por ***Håkon Wium Lie*** em 1994 e publicamente lançada em 1996.

Foi desenvolvida com a intenção de prover *folhas de estilo* para a web, mas demorou a emplacar e apenas em 2000 o primeiro navegador com **total suporte** à CSS1 foi lançado, *Internet Explorer 5.0*.

Hoje é essencial para qualquer página na web e já está na sua terceira versão.

Atualmente a linguagem não é baseada em versões, apenas seus módulos. O termo CSS3 engloba todas as novidades pós *CSS2.1*, logo pode ser simplesmente chamada de CSS.

Logotipo do CSS3:

Estrutura

CSS não possui tags como HTML, mas seletores com propriedades e valores. Como a seguir:

```
seletor {
   propriedade: valor;
}
```

De uma forma genérica, o *seletor* representa o elemento no qual a regra será aplicada. A *propriedade* define o atributo a ser utilizado. *Valor* nada mais é que o valor a ser aplicado na propriedade do elemento.

Exemplo de uma regra CSS aplicando branco como cor de fundo do corpo da página:

```
body {
  background-color: white;
}
```

Aplicação

Há três formas de inserir estilos em uma página HTML:

- Inline, ou em linha
- Folha de estilos interna
- Folha de estilos externa

INLINE

Usado para aplicar estilos diretamente no elemento desejado.

Para inserir esse tipo de estilo, é necessário um atributo `style` no elemento a aplicar a regra.

Exemplo de CSS inline:

```
<p style="color: blue;">Sou um parágrafo com o texto em azul</p>
```

Esse tipo de estilização não é recomendável por ter de ser repetida em cada elemento, impossibilitando o aproveitamento do código.

FOLHA DE ESTILOS INTERNA

Essa opção facilita o aproveitamento de estilos dentro de uma mesma página, já que as regras serão aplicadas a todos os elementos referenciados nos seletores.

Para isso, basta indicar uma tag `style` no `head` do documento.

Exemplo:

```
<!DOCTYPE html>
<html>

  <head>
    <style>
      p {
        color: blue;
      }
    </style>
  </head>

  <body>
```

```
    <p>Sou um parágrafo com o texto em azul</p>
    <p>Também tenho o texto em azul!</p>
    <p>Eu também!!</p>
  </body>

</html>
```

Aqui temos um melhor aproveitamento do código, porque todos os elementos referenciados nos seletores serão afetados pelas regras ali criadas.

Mas, e quando temos mais de uma página? Ainda temos de copiar todo o código feito em uma para todas as outras.

FOLHA DE ESTILOS EXTERNA

Com uma folha de estilos externa, ou external style sheet, os estilos podem ser aplicados em mais de um documento. Assim diminuímos a repetição de código e aumentamos a produtividade.

Conseguimos vincular uma folha de estilos a uma página simplesmente referenciando no `head` da página com uma tag `link`.

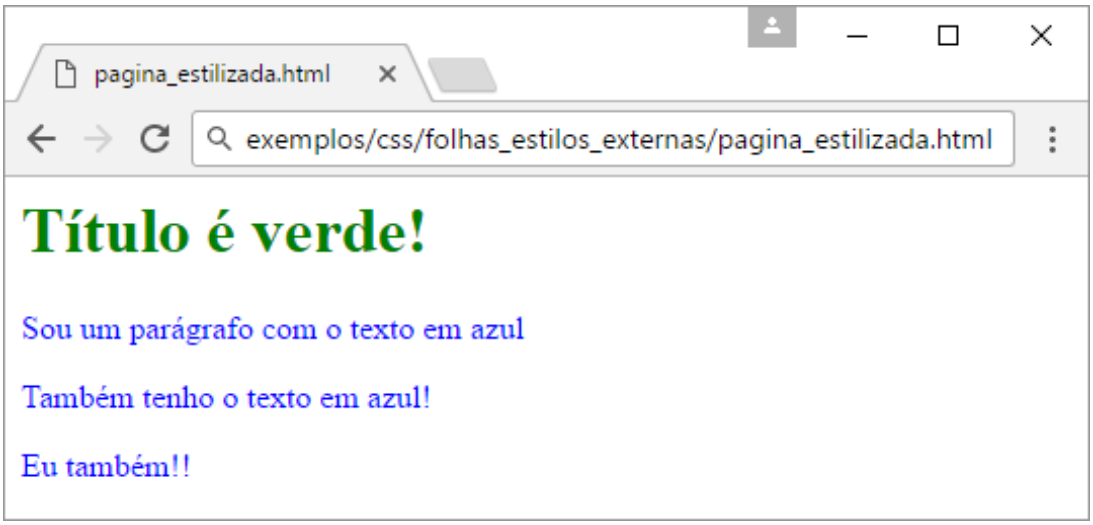
```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="meus_paragrafos_azuis.css">
  </head>
  <body>
    <p>Sou um parágrafo com o texto em azul</p>
    <p>Também tenho o texto em azul!</p>
    <p>Eu também!!</p>
  </body>
</html>
```

Também podemos referenciar mais de uma folha de estilos na mesma página:

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="meus_paragrafos_azuis.css">
    <link rel="stylesheet" type="text/css" href="titulos_sao_verdes.css">
  </head>
  <body>
    <h1>Título é verde!</h1>
    <p>Sou um parágrafo com o texto em azul</p>
    <p>Também tenho o texto em azul!</p>
    <p>Eu também!!</p>
```

```
  </body>
</html>
```

Assim será reproduzido esse documento no navegador:



CORES

Em CSS, podemos aplicar cores em diversos lugares e de diversas formas. Podemos definir cor do texto, cor de fundo de algum elemento ou da página em si, cor do botão, cor da borda e outros. Por exemplo, para definir a cor de fundo de uma `div` para verde, basta fazer o seguinte:

```
div {
  background-color: green;
}
```

Para definir que os parágrafos terão o texto em azul:

```
p {
  color: blue;
}
```

Como pode ver, basta inserir o nome da cor e o navegador irá reproduzi-la ao usuário. Mas esta não é a única maneira de dizer a cor desejada, temos as seguintes:

- Por nome, também chamado de *keyword*
- Valores em RGB
- Valores hexadecimais
- Valores HSL

Keyword

Como vimos anteriormente, um keyword é nada além do nome da cor em inglês. Exemplo de algumas cores e suas representações:

	black
	silver
	gray
	white
	maroon
	red
	purple
	fuchsia
	green
	lime
	olive
	yellow
	navy
	blue
	teal

A representação de keywords pode variar entre sistemas e navegadores diferentes, portanto seu uso não é recomendável.

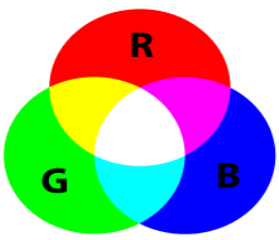
RGB

RGB significa Red, Green, Blue (vermelho, verde e azul em inglês). Dessa forma podemos especificar a quantidade de cada uma dessas cores para produzir a cor final desejada.

Podendo ser escrita da seguinte maneira:

```
rgb(vermelho, verde, azul);
```

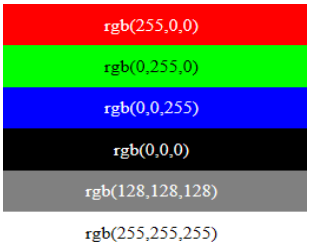
Coloca-se um número de *0 a 255* onde deseja aplicar a intensidade de cada cor. Conseguimos criar a cor desejada *misturando* essas cores, como podemos ver na ilustração:



Dessa forma, para deixar um parágrafo com seu texto em vermelho podemos fazer assim:

```
p {
  color: rgb(255, 0, 0);
}
```

Alguns exemplos de cores:



Hexadecimal

Os valores em hexadecimal funcionam da mesma forma de um em *rgb()*. Na verdade, os dois representam valores na escala RGB, apenas são representados de maneiras diferentes.

Ao invés de os valores irem de 0 a 255, vão de *00 a FF*.

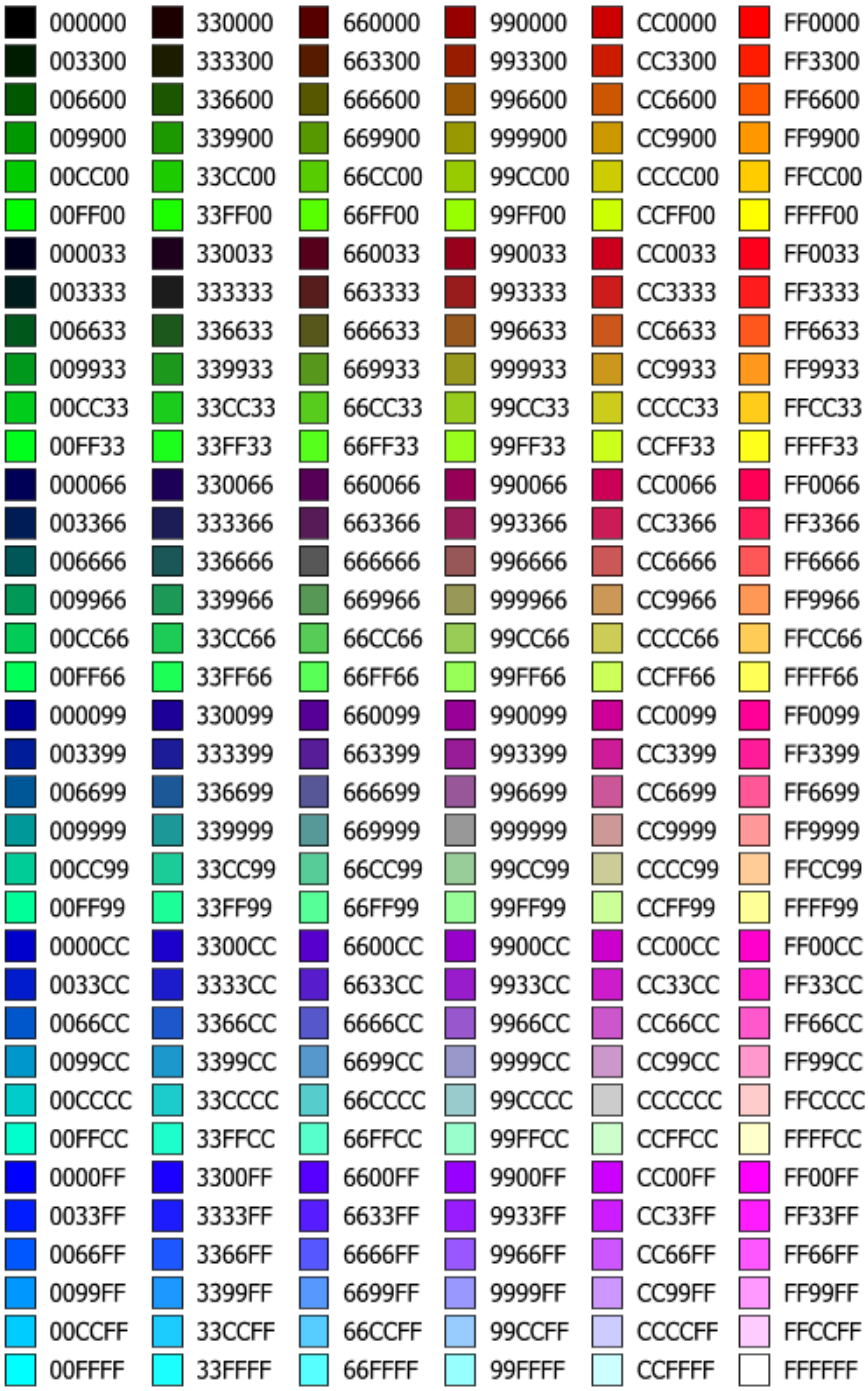
FF em hexadecimal é igual a 255 em decimal.

Para atribuirmos um valor hexadecimal é necessário colocar um hashtag `#` antes do valor. Por exemplo: `#FFFFFF`.

Sendo assim, para aplicar vermelho ao texto de um parágrafo deve ser feito assim:

```
p {
  color: #FF0000;
}
```

Alguns exemplos usando código hexadecimal:



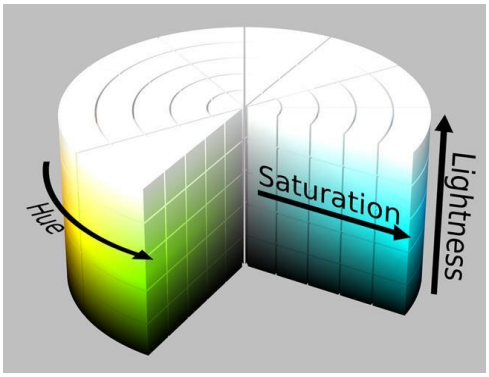
Pode-se usar tanto letra maiúscula como minúscula, o navegador vai interpretar da mesma forma.

HSL

É um modelo relativamente novo e, portanto, não é suportado em navegadores mais antigos. Sua sigla significa **H**ue, **S**aturation e **L**ightness (matiz, saturação e luminosidade).

```
hsl(matiz, saturação, luminosidade);
```

Para entender como funciona, segue uma representação visual desse modelo e o significado de cada parâmetro:



Matiz

Matiz se refere à cor em si. Exemplo: vermelho, verde, azul, amarelo. Nesse parâmetro é especificado o ângulo da volta no cilindro, começando de 0 (vermelho).

Saturação

Aqui é definido o quanto dessa cor deseja aplicar, variando de 0 a 100%. Quanto maior esse valor, mais *pura* é a cor. Quanto menor, mais cinza ela se torna.

Luminosidade

Nada mais é que a quantidade de luz a ser aplicada na mistura. Também varia de 0 a 100%, correspondente a ausência e total presença de luz, respectivamente.

Aplicação

Com isso, para aplicarmos vermelho em um parágrafo seria:

```
p {
   color: hsl(720, 100%, 50%);
}
```

TEXTO

Além de definir cores, o CSS também pode ser usado para mudar as fontes do documento. Sendo possível deixar os textos em negrito, itálico, etc.

Fonte

Para mudar a fonte a ser usada em algum elemento, basta usar a propriedade `font-family`:

```
p {
  font-family: arial;
}
```

Também podemos definir uma lista de fontes, assim o navegador pode escolher de acordo com sua disponibilidade:

```
p {
  font-family: "Trebuchet MS", Verdana, sans-serif;
}
```

O navegador procura a fonte indo da esquerda para a direita, aplicando a que encontrar primeiro. Caso nenhuma tenha sido encontrada, ele usa a padrão.

Tamanho

Podemos definir o tamanho do texto exibido com a propriedade `font-size`.

Por exemplo, para definir um parágrafo para `20px` basta:

```
p {
  font-size: 20px;
}
```

Estilo

Para definirmos o texto como itálico, usamos a propriedade `font-style`:

```
p {
  font-style: italic;
}
```

Para deixar o texto negrito, usamos `font-weight`.

```
p {
  font-weight: bold;
}
```

Para o texto ficar sublinhado, sobrelinhado ou tachado, temos a `text-decoration`, usando os valores `underline`, `overline` e `line-through`, respectivamente:

```
p {
  text-decoration: underline;
}
```

Muitas dessas propriedades citadas anteriormente podem ser utilizadas com a propriedade `font`.

Por exemplo, um parágrafo em itálico, com tamanho 20px e fonte Arial seria assim:

```
p {
  font: italic 20px Arial;
}
```

Além de alterar o estilo da fonte, também podemos alterar o alinhamento do texto em si. Isso é possível com a propriedade `text-align`, tendo `left`, `right`, `center` e `justify` como seus valores:

```
p {
  text-align: center;
}
```

BORDAS

Por padrão, os elementos em HTML já possuem uma borda de tamanho 0 para não ser exibida até ser requerida.

Shorthand

Assim como a propriedade `font`, também temos um shorthand para aplicar uma borda rapidamente a um elemento.

Para isso usamos a propriedade `border`. Para aplicar uma borda com a espessura de 3px, sólida (sem tracejado) com a cor vermelha, fazemos assim:

```
p {
  border: 3px solid rgb(255,0,0);
}
```

Também temos shorthands para os lados da borda:

```
p {
  border-top: 3px solid rgb(255,0,0);
  border-right: 2px solid rgb(0,255,0);
  border-bottom: 3px solid rgb(255,0,0);
  border-left: 2px solid rgb(0,255,0);
}
```

Mas também podemos alterar esses valores em propriedades independentes, para maior liberdade.

Espessura

Para definir a espessura, usamos a propriedade `border-width`.

```
p {
  border-width: 3px;
}
```

Também conseguimos definir espessuras diferentes para um dos lados da borda, na seguinte ordem:

```
p {
  border-width: cima direita baixo esquerda;
}
```

Ou seja, para definir uma borda com a espessura de 5px em cima, 3px à direita, 5px em baixo e 10px à esquerda faremos assim:

```
p {
  border-width: 5px 3px 5px 10px;
}
```

Também temos propriedades separadas para cada um desses *lados*:

```
p {
  border-top-width: 5px;
  border-right-width: 3px;
  border-bottom-width: 5px;
  border-left-width: 10px;
}
```

Cor

Para definir a cor, podemos usar a propriedade `border-color`. Funcionando da mesma maneira da espessura, temos uma propriedade que agrupa todos os lados e também cada lado separadamente.

```
p {
  border-color: rgb(255, 0, 0);
}
p {
  border-color: rgb(255, 0, 0) rgb(0, 255, 0) rgb(255, 0, 0) rgb(0, 255, 0);
}
p {
  border-top-color: rgb(255, 0, 0);
  border-right-color: rgb(0, 255, 0);
  border-bottom-color: rgb(255, 0, 0);
  border-left-color: rgb(0, 255, 0);
}
```

Estilo

O estilo é definido usando a propriedade `border-style`.

Essa propriedade pode receber os seguintes valores:

- none
- hidden
- dotted
- dashed
- solid
- double
- groove
- ridge
- inset
- outset

Bordas arredondadas

Há um tempo atrás, imagens eram usadas para simular bordas arredondadas. Mas o CSS3 trouxe uma nova propriedade `border-radius`.

Com essa nova propriedade, podemos definir o raio do arredondamento que quisermos. Para definir uma borda de 5px:

```
p {
  border-radius: 5px;
}
```

Ela também funciona como as outras propriedades, também sendo uma shorthand e tendo suporte para definição de cada *lado*.

Com tudo que vimos, podemos produzir algo assim:

```
p {
  border: 10px solid #000000;
  border-radius: 10px 40px 40px 10px;
  width: 200px;
  height: 100px;
  background-color: #cccccc;
}
```

Comentários

Assim como em HTML, às vezes precisamos de comentários.

Podemos inserir comentários dentro das nossas definições de regras, desde que não esteja *dentro* de uma propriedade. Basta colocarmos entre /* e */, como a seguir:

```
p {
  /* Aqui vão as regras para o parágrafo */
  background-color: #cccccc; /* Aqui definimos a cor de fundo */
}
```

SELETORES

Como visto anteriormente, para definirmos o estilo de algum elemento na página devemos escrever uma regra com a seguinte estrutura:

```
seletor {
  propriedade: valor;
}
```

Esse seletor nada mais é que uma *expressão* a ser comparada pelo navegador, a fim de decidir se determinada regra será aplicada aos elementos da página.

O navegador, ao renderizar a página, inicia um processo de comparação dos elementos presentes nela às regras de CSS definidas.

Os elementos são testados para saber se passam pela expressão utilizada no seletor. Se o elemento atender à regra, os estilos ali presentes são aplicados.

Até o momento, colocamos tags como sendo nosso seletor, por exemplo:

```
p {
  background-color: #cccccc;
}
```

Mas existem outros tipos de seletores:

- Por tipo, ou tag
- Classes
- ID
- Por atributo

Por tipo

Aqui usamos a tag do elemento, como a, div, button, etc.

É o tipo de expressão usada neste documento em todos os exemplos anteriores.

Classes

Todo elemento HTML possui um atributo denominado `class`, usado aqui para definirmos um mesmo estilo a determinados elementos.

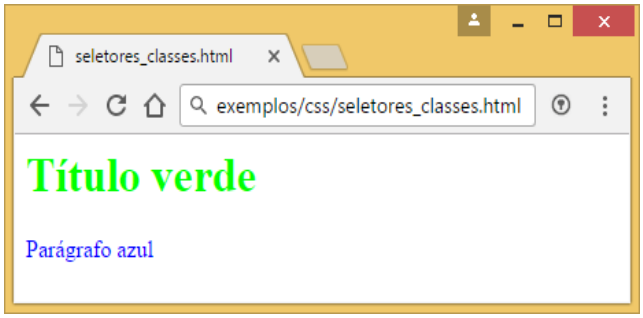
Por exemplo:

```
<!DOCTYPE html>
<html>
  <body>
    <h1 class="verde">Título verde</h1>
    <p class="azul">Parágrafo azul</p>
  </body>
</html>
```

Podemos observar o atributo `class` nos elementos `h1` e `p`, com os valores `verde` e `azul`, respectivamente.

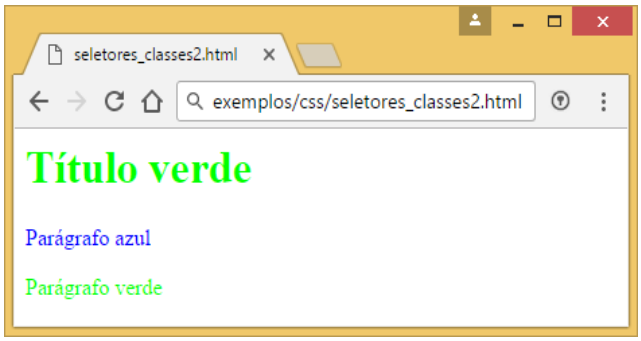
Com isso, podemos definir o CSS:

```
.verde {
  color: rgb(0,255,0);
}
.azul {
  color: rgb(0,0,255);
}
```



Também com as mesmas regras, podemos definir a mesma cor para tipos de elementos diferentes:

```
<!DOCTYPE html>
<html>
  <body>
    <h1 class="verde">Título verde</h1>
    <p class="azul">Parágrafo azul</p>
    <p class="verde">Parágrafo verde</p>
  </body>
</html>
```



ID

Além do atributo `class`, os elementos em HTML também possuem o `id` que funciona de forma semelhante.

A principal diferença entre eles é que o ID deve ser único, não podendo ter outro elemento com o mesmo ID na mesma página.

Ele é usado para podermos criar estilos específicos para um elemento, semelhantemente à uma regra inline.

Por exemplo:

```
.verde {
   color: rgb(0,255,0);
}
.azul {
   color: rgb(0,0,255);
}
#principal {
   /* Aplicamos borda no parágrafo principal */
   border-style: dotted;
}
```

Aplicado na seguinte página:

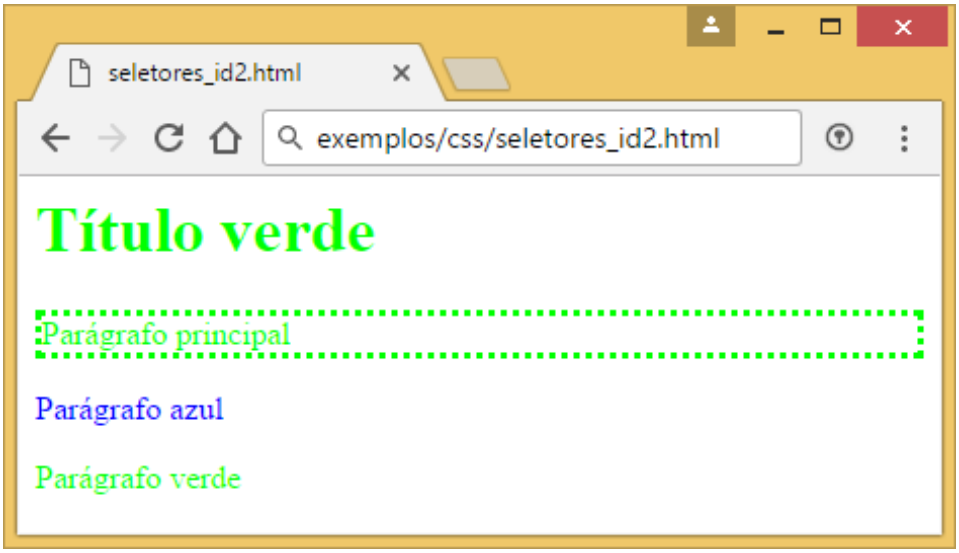
```
<!DOCTYPE html>
<html>
   <body>
      <h1 class="verde">Título verde</h1>
      <p id="principal">Parágrafo principal</p>
      <p class="azul">Parágrafo azul</p>
      <p class="verde">Parágrafo verde</p>
   </body>
</html>
```

Temos o seguinte efeito:

Também podemos usar duas regras no mesmo elemento, por exemplo:

```
<!DOCTYPE html>
<html>
  <body>
    <h1 class="verde">Título verde</h1>
    <p id="principal" class="verde">Parágrafo principal</p>
    <p class="azul">Parágrafo azul</p>
    <p class="verde">Parágrafo verde</p>
  </body>
</html>
```

Ficando assim:



Exemplo seletores ID 2

Por atributo

Também é possível criar regras baseando-se nos atributos dos elementos.

Por exemplo, o elemento `input` possui um atributo de nome `type` onde especificamos o tipo desse input, podendo ser `text`, `password`, `button`, etc.

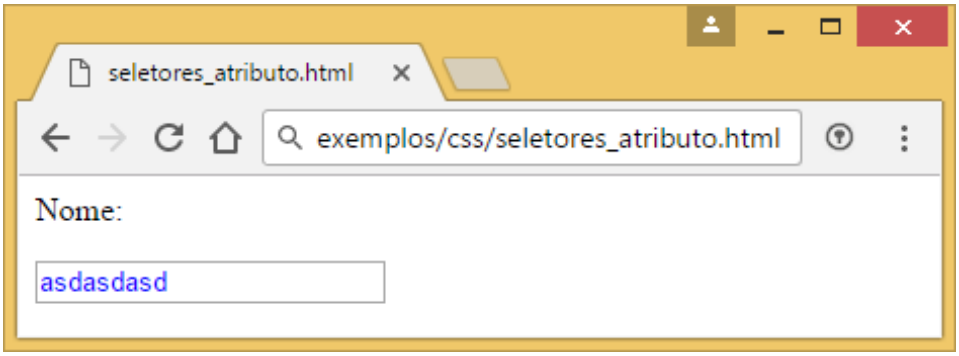
Sendo assim, podemos criar uma regra para todos os elementos input do tipo `text`:

```
input[type=text] {
  color: rgb(0,0,255);
}
```

E aplicando no seguinte HTML:

```
<!DOCTYPE html>
<html>
  <body>
    <p>Nome:</p>
    <input type="text" />
  </body>
</html>
```

Tem como resultado:



Exemplo seletores por atributo

COMBINADORES

Além de podermos criar regras com expressões para elementos diretamente, também é possível combiná-las para termos ferramentas ainda mais poderosas.

Para isso, usamos combinadores e alguns de seus tipos são:

- Elemento descendente
- Elemento filho

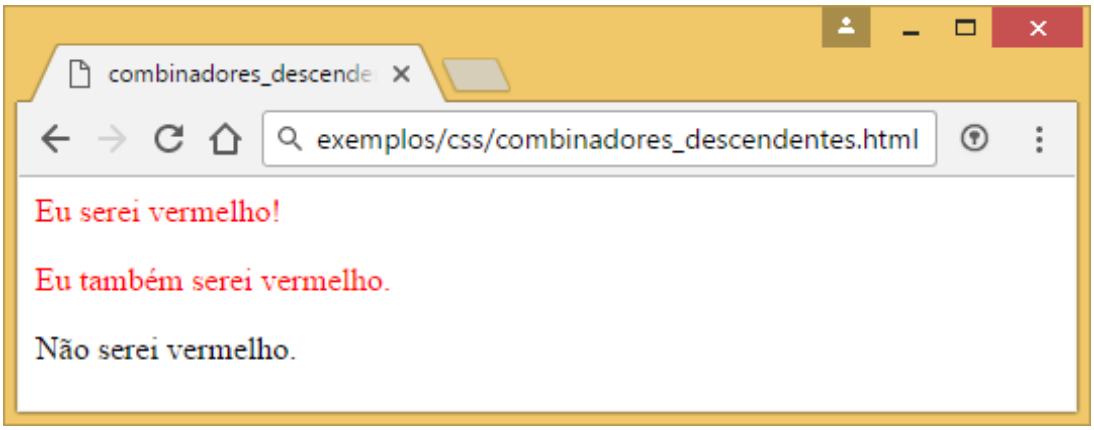
Elemento descendente

O combinador " " (espaço) é utilizado para encontrar um elemento na parte da árvore de descendentes do elemento anterior a ele. Exemplo:

```
div span {
  color: red;
}
```

```
<!DOCTYPE html>
<html>
  <body>
    <div>
      <span>Eu serei vermelho!</span>
      <p>
        <span>Eu também serei vermelho.</span>
      </p>
    </div>
    <p>Não serei vermelho.</p>
  </body>
</html>
```

Será exibido assim:



Elemento filho

O combinador > (maior) é usado para encontrar um filho imediato ao especificado à esquerda do combinador. Exemplo:

```
div > span {
  color: red;
}
```

```
<!DOCTYPE html>
<html>
  <body>
    <div>
      <span>Eu serei vermelho!</span>
      <p>
        <span>Não sou mais vermelho.</span>
      </p>
    </div>
    <p>Não serei vermelho.</p>
  </body>
</html>
```

Tendo como resultado:

Exemplo combinador filho

JAVASCRIPT

Surgimento

JS, como também é chamada, foi criada por **Brendan Eich** em 1996 enquanto trabalhava na Netscape.

Apesar de seu nome sugerir ser uma versão *simplificada* de Java, é completamente diferente.

JS foi inspirada em linguagens como Lisp e Scheme, mas ainda assim teve sua sintaxe inspirada pelo Java.

ECMAScript

Desde o início do projeto de criação, a linguagem já teve nomes como Mocha e LiveScript, hoje também é chamada de ECMAScript.

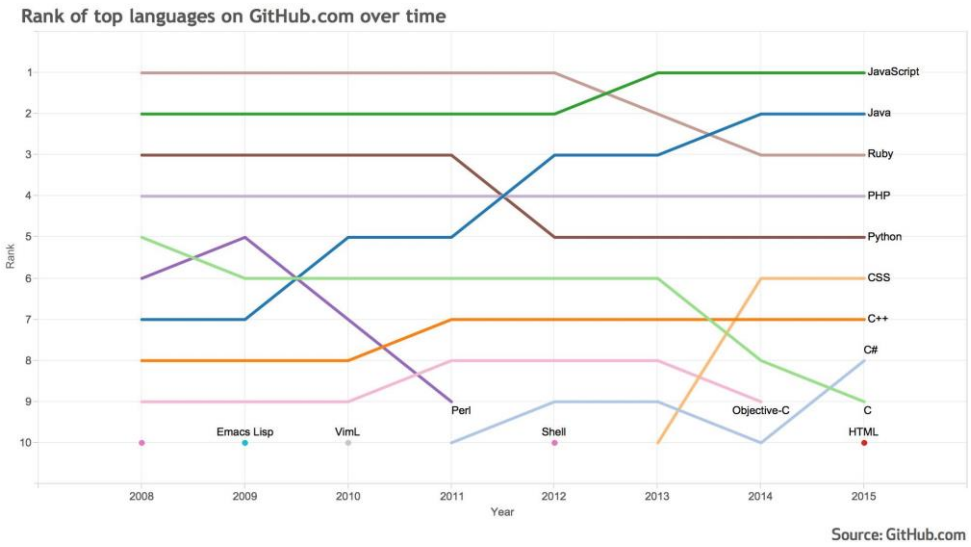
Apesar de não ser totalmente errado, ECMAScript se refere à base para a criação do JavaScript. É um padrão usado para definir o funcionamento dessa linguagem.

Qualquer linguagem pode ser feita com base nesse padrão, como foi o caso da ActionScript (para desenvolvimento com Flash) e JScript (criado pela Microsoft para ser usada no Internet Explorer).

Tem esse nome, ECMAScript, por ser mantida pela ECMA International desde 1996. Um órgão que tem o objetivo de manter esse padrão, fazer correções e lançar atualizações.

Uso

Hoje em dia, JS/ES é uma das linguagens mais populares.



Ranking de popularidade no GitHub

Além de ser usada em sites, também está presente em servidores web (NodeJS), banco de dados (MongoDB) e aplicações desktop (Electron).

A popular IDE Visual Studio Code foi escrita com JavaScript.

Estrutura

JS é uma linguagem interpretada, com suporte à orientação a objetos e fracamente tipada.

VARIÁVEIS

Variável é o que usamos para armazenar os valores posteriormente úteis para o nosso software.

Para criar uma variável em JS:

```
var minhaVariavel;
```

Para atribuir um valor a ela, basta informá-lo depois de um = (sinal de igual):

```
minhaVariavel = 5;
```

Acima, atribuímos o valor 5 à nossa variável minhaVariavel.

Podemos mudar o seu valor da mesma forma que atribuímos:

```
minhaVariavel = 'tenho um novo valor';
```

Podemos atribuir qualquer tipo de valor a uma variável. Portanto, JS é uma linguagem fracamente tipada, diferente de Java, por exemplo.

STRINGS

Strings são representações de texto e para definir uma string em JS, basta colocar seu valor entre ' (aspas simples) ou " (aspas duplas).

O navegador interpretará da mesma forma usando aspas simples ou duplas, sendo uma questão de gosto ou praticidade.

As aspas simples são, de longe, as mais utilizadas. Logo, é extremamente recomendável usá-la dessa forma para se manter um padrão.

Exemplo:

```
'aqui está um texto'
"aqui também é um texto"
```

PONTO E VÍRGULA ;

O uso de ponto e vírgula em JS é opcional. A linguagem não nos obriga a colocar ao final de cada linha, assim como Java ou C#.

Na verdade, essa pontuação é inserida automaticamente em alguns pontos, por isso não é obrigatória. Esse mecanismo é denominado *ASI (Automatic Semicolon Insertion)*.

COMENTÁRIOS

Para definirmos um comentário em JS, basta colocar // antes do conteúdo desejado:

```
// aqui é um comentário
```

Também podemos definir blocos de comentários, usado quando possuem mais de uma linha:

```
/*
   Bloco de comentário
   são permitidas múltiplas linhas!
*/
```

COMPARAÇÃO

Muitas vezes precisamos comparar os valores contidos nas variáveis, a fim de fazermos alguma validação, verificar se o usuário preencheu um determinado campo, etc.

Para isso temos dois operadores:

- == (igual)
- === (igual estrito)

Os dois trabalham de forma similar, mas o primeiro apenas compara os valores das variáveis. Já o segundo, também verifica se os valores são do mesmo tipo.

Exemplo:

```
2 == '2' // true
```

Ao usar o igual para compararmos os valores 2 (numérico) e '2' (texto), o resultado é verdadeiro.

Já se usarmos o igual estrito, é falso:

```
2 === '2' // false
```

É verdadeiro caso se os dois valores forem do mesmo tipo:

```
2 === 2 // true
```

NULL E UNDEFINED

Em JS, temos dois valores representando o vazio.

`undefined` significa que uma variável foi declarada, mas ainda não tem valor. Exemplo:

```
var minhaVariavel;
```

`null` é um valor que pode ser atribuído a uma variável. Exemplo:

```
minhaVariavel = null;
```

Apesar de representarem o *vazio*, possuem algumas diferenças quando comparados:

```
null == undefined   // true
null === undefined  // false
null === null       // true
```

Podemos observar que os dois representam o mesmo valor, mas possuem tipos diferentes.

TYPEOF

`typeof` é um operador usado para retornar o tipo desejado de uma variável ou valor. Exemplo:

```
typeof 'texto'     // string
typeof 0           // number
typeof true        // boolean
typeof new Date()  // object
typeof undefined   // undefined
typeof null        // object
```

Aplicação

Para usarmos o JS em uma página HTML, podemos fazer de forma semelhante ao CSS colocando o código no meio do documento ou vincularmos um arquivo externo.

CÓDIGO EMBUTIDO

Para embutirmos um código JavaScript na página, basta colocarmos em uma tag `script`.

Exemplo:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Título da página</title>
    <script>
      var minhaVariavel = 'Olá!';
    </script>
  </head>
</html>
```

Podemos colocar essa tag em qualquer parte do código:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Título da página</title>
  </head>
  <body>
    <script>
      var minhaVariavel = 'Olá!';
    </script>
  </body>
</html>
```

ARQUIVO EXTERNO

Assim como CSS, um código JS também pode ser feito em um arquivo externo e referenciado na página HTML.

Basta criarmos um arquivo com a extensão .js e referenciá-lo na tag script com o atributo src:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Título da página</title>
  </head>
  <body>
    <script src="meuArquivoJavaScript.js"></script>
  </body>
</html>
```

É recomendado que o código seja colocado logo antes de acabar o corpo da página, exceto em casos específicos que é recomendado colocar na head ou em outra parte.

Dessa forma o navegador busca os arquivos JS após renderizar toda a página, melhorando o tempo de carregamento.

CAIXAS DE DIÁLOGO

Existem três formas de exibirmos caixas de diálogo em um navegador:

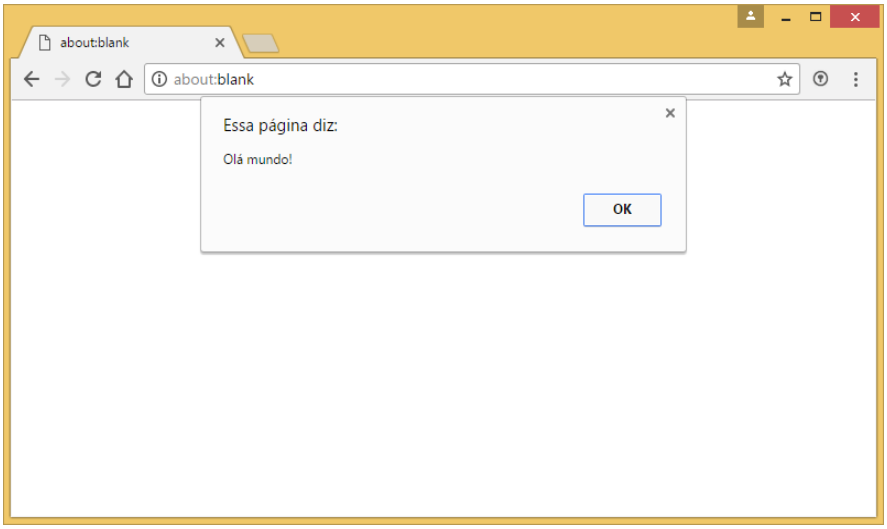
- alert
- prompt
- confirm

Eles são usados para exibir um alerta, caixa de entrada e caixa de confirmação, respectivamente.

alert

Esse método mostra uma caixa de diálogo com a mensagem definida na chamada. Como segue:

```
alert('Olá mundo!');
```

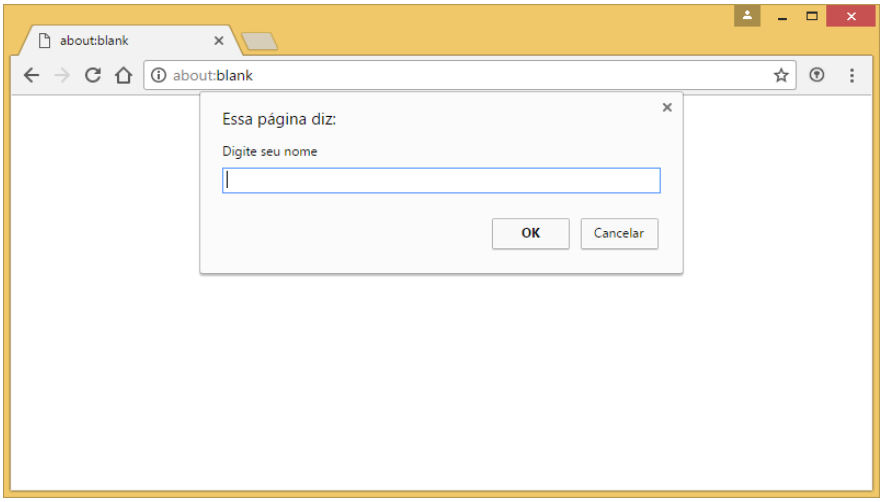


prompt

Esse método trabalha de forma parecida, mas além de mostrar uma mensagem ao usuário, também exibe um campo para o usuário inserir um texto. Essa função se encarrega de retornar o texto em uma string.

```
prompt('Digite seu nome');
```

Exibindo o seguinte:

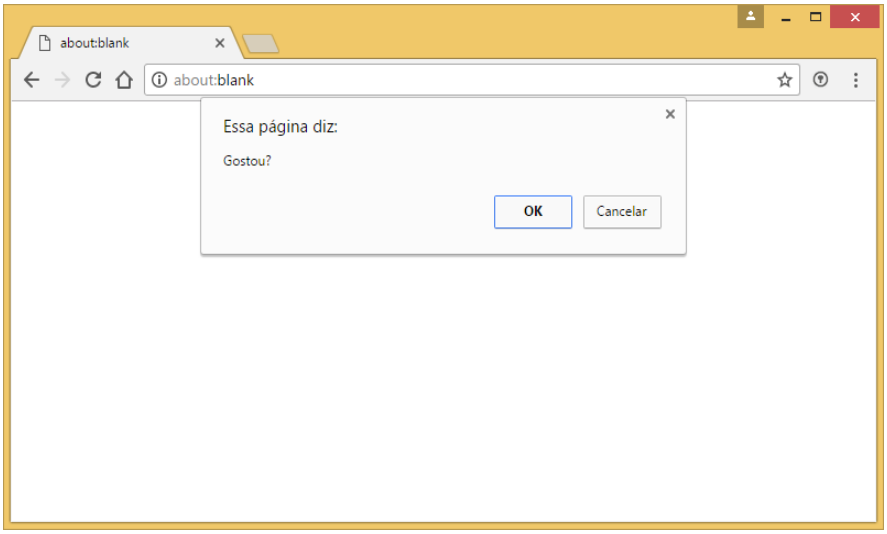


confirm

Mostra uma caixa de diálogo com dois botões: um para confirmar e outro para cancelar. Retornando `true` ou `false` de acordo com a escolha do usuário:

```
confirm('Gostou?');
```

Exibindo o seguinte:



DOCUMENT OBJECT MODEL

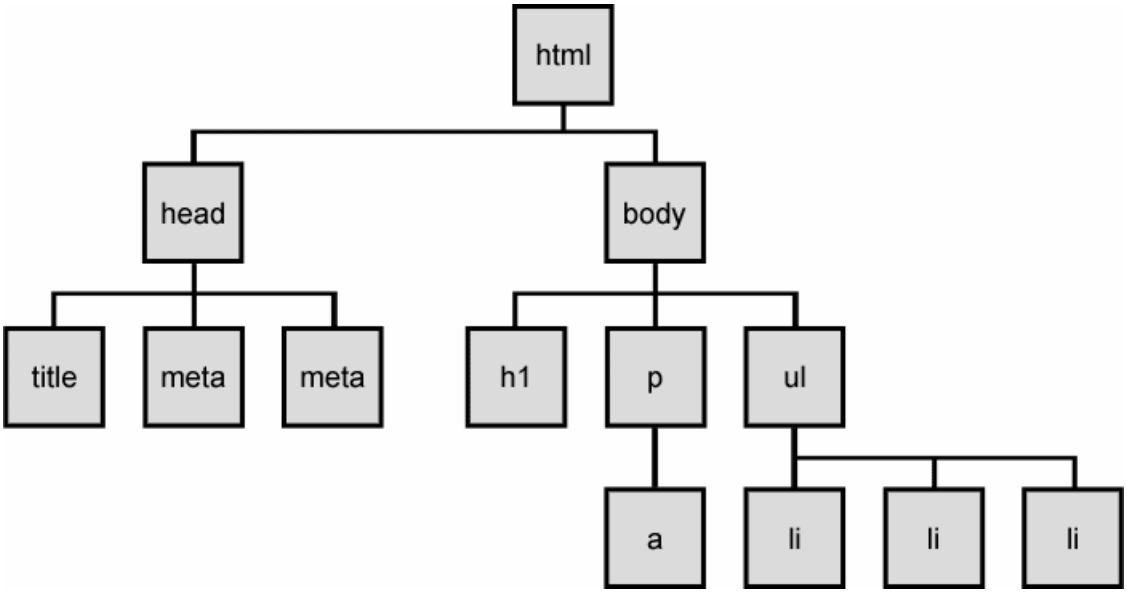
Apesar de ser possível mostrar caixas de diálogos para o usuário, essa funcionalidade está longe de ser a ideal para aplicações complexas.

Comumente, construímos formulários no próprio documento HTML para o usuário preencher os campos necessários e assim obtermos os dados ali inseridos.

Esses campos, ou elementos, definidos em uma página HTML são organizados pelo navegador em uma espécie de árvore. Essa organização é chamada de *DOM (Document Object Model)*.

Além de definir essa estrutura, os navegadores dispõem de uma API para acessarmos o DOM programaticamente.

Segue uma representação gráfica do DOM:



DOM API

Com essa API podemos acessar diversas áreas do nosso documento.

Para exibirmos o título dessa página abaixo em um alerta:

```
<!DOCTYPE html>
<html>
<head>
  <title>Olá, mundo!</title>
</head>
<body></body>
</html>
```

Utilizamos esse código:

```
alert(window.document.title); // Olá, mundo!
```

USO ESTRITO

Por padrão, o JS não nos avisa de alguns erros possíveis de acontecer. Como uso de variáveis não declaradas, uso de palavras reservadas, ou recursos considerados obsoletos.

Para mudar isso, a versão 5 da ECMAScript trouxe a `use strict;`.

Essa diretiva faz o navegador interpretar o código de uma maneira mais estrita, como diz o próprio nome. Nos forçando a escrever um código de melhor qualidade.

Para saber mais, veja em: **https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Strict_mode**

Para ativar o movo estrito, basta colocar 'use strict'; no começo do arquivo ou função que deseja aplicar:

```
<script>
  'use strict';
  texto = 'Olá'; // erro, variável não declarada
</script>
```

FUNÇÕES

Muitas vezes precisamos criar blocos de código para serem executados conforme a nossa necessidade.

Para isso, criamos funções:

```
function mostrarNumeroCinco() {
  alert(5);
}
```

Existem duas maneiras de criarmos uma função:

- Declaração
- Expressão

Declaração de função

Uma função é declarada com a seguinte estrutura:

```
function nomeDaFuncao(parametros) {
  // instruções
}
```

Para, por exemplo, criarmos uma função que mostra um número em uma caixa de diálogo:

```
function mostrarNumero(numero) {
  alert(numero);
}
```

Expressão de função

Além de uma função poder ser declarada, também é possível criá-la com uma expressão:

```
var nomeDaFuncao = function(parametros) {
  // instruções
}
```

A diferença aqui é que estamos *guardando* uma função anônima em uma variável, não declarando como fizemos anteriormente.

Execução de uma função

Tanto uma função criada a partir de uma declaração quanto de uma expressão é executada da mesma maneira:

```
nomeDaFuncao(parametro);
```

Colocamos o nome da função e informamos os parâmetros necessários entre parênteses.

Caso a função não necessite de parâmetros, basta os parênteses:

```
funcaoSemParametros();
```

Escopo

As variáveis criadas dentro de uma função pertencem a ela e não podem ser referenciadas fora:

```
function minhaFuncao() {
  var minhaVariavel = 5;
  console.log(minhaVariavel); // 5
}
console.log(minhaVariavel); // variável não existe
```

Ao contrário de linguagens como C, C# e Java, JS trabalha com escopos por função, não por blocos.

Isso significa que todas as variáveis definidas dentro de uma função podem ser referenciadas em qualquer parte dela, por fazer parte do mesmo escopo:

```
function minhaFuncao() {
  var minhaVariavel = 5;

  if (minhaVariavel === 5) {
    var minhaOutraVariavel = 6;
  }

  console.log(minhaOutraVariavel); // 6
}
```

Hoisting

Durante a execução de uma função o navegador inicia de um processo chamado de hoisting.

Esse processo varre todo o corpo da função procurando por declarações e já as executa primeiro.

Por exemplo:

```
function minhaFuncao() {
   console.log('primeira linha');
   var variavel; // essa será a primeira linha executa
   variavel = 'minha variável';
   console.log(variavel);
}
```

Com isso, podemos escrever esse código e executará normalmente:

```
function minhaFuncao() {
   variavel = 'minha variável';
   var variavel;
   console.log(variavel); // minha variável
}
```

Lembre-se que apenas a criação da variável passa por esse processo, não sua inicialização. Portanto:

```
function minhaFuncao() {
   console.log(variavel); // undefined
   variavel = 'minha variável';
   var variavel;
   console.log(variavel); // minha variável
}
```

Esse processo também existe para declarações de funções. Mas diferentemente das variáveis, o corpo da função também fica disponível.

Exemplo:

```
minhaFuncao() // minha variável

function minhaFuncao() {
   var variavel = 'minha variável';
   return variavel;
}
```

ECMAScript 2015

A nova versão da ECMAScript, também conhecida como ES6, trouxe novas maneiras de inicializar variáveis:

- let
- const

let

O `let` trabalha de forma semelhante ao `var`, **mas cria um escopo por bloco**:

```
function minhaFuncao() {
  let minhaVariavel = 5;

  if (minhaVariavel === 5) {
    let minhaOutraVariavel = 6;
  }

  console.log(minhaOutraVariavel); // erro de referëncia
}
```

Além disso o processo de hoisting fica diferente, fazendo com que a variável não esteja disponível antes da sua inicialização:

```
function minhaFuncao() {
  variavel = 'minha variável'; // erro de referência
  let variavel;
}
```

const

`const` possui um escopo igual ao `let`, porém não permite que uma variável seja atribuída com outro valor após a sua inicialização:

```
let minhaVariavel = 5;
minhaVariavel = 6; // podemos atribuir outro valor

const minhaConstante = 6;
minhaConstante = 7; // erro de execução
```

Closures

Definição de closure na **MDN**.

Closures (fechamentos) são funções que se referem a variáveis livres (independentes). Em outras palavras, a função definida no closure "lembra" do ambiente em que ela foi criada.

```
function minhaFuncao() {
  const x = 10;
  function somarDez(numero) {
    const resultado = numero + x;
    return resultado;
  }
  return somarDez;
}
const funcaoSomarDez = minhaFuncao();
const numeroSomadoDez = funcaoSomarDez(5);
console.log(numeroSomadoDez); // 15
```

Declaramos uma função de nome `minhaFuncao` que ao ser executada retorna outra função de nome `somarDez`.

Armazenamos essa função retornada em uma variável de nome `funcaoSomarDez`.

Essa função estava dentro do escopo da `minhaFuncao`, portanto tem acesso a todas as variáveis ali criadas.

Dessa forma, a `funcaoSomarDez` *carrega* todo o escopo da `minhaFuncao`. Podendo ser executada mesmo depois de termos *saído* do escopo da *funçao mãe*.

Uma closure é um tipo de objeto que combina a função e o ambiente onde foi criada, contendo as variáveis daquele escopo. Nesse caso, a `funcaoSomarDez` é uma closure que contém a função `somarDez` e variável `x`.

VETORES

Um vetor, ou array, é uma estrutura de dados usada para armazenar uma coleção de valores, agrupados continuamente na memória.

Criação

JS tem um objeto `Array` para definir vetores, podendo ser inicializado da seguinte maneira:

```
const meusNumeros = [10, 20, 30];
```

Acessando itens

Para acessarmos algum item de um vetor, basta colocar o índice do mesmo entre colchetes após o nome da variável:

```
const meusNumeros = [10, 20, 30];
const primeiroNumero = [0];
console.log(primeiroNumero); // 10
```

Índice é a posição do elemento em um vetor, começando de `0` (zero).

Para descobrirmos o índice de um elemento, usamos o método `indexOf`:

```
const meusNumeros = [10, 20, 30];
const posicaoNumeroDez = meusNumeros.indexOf(10);
console.log(posicaoNumeroDez); // 0
```

Adicionando itens

Para adicionar itens usamos o método `push`:

```
const meusNumeros = [10, 20, 30];
meusNumeros.push(40);
console.log(meusNumeros); // [10, 20, 30, 40]
```

Removendo itens

Para remoção de itens temos disponível o método `pop`, o qual remove o último item adicionado ao array.

```
const meusNumeros = [10, 20, 30];
meusNumeros.pop();
console.log(meusNumeros); // [10, 20]
```

Apesar de ter sua utilidade, esse método pode ser bem limitado por apenas remover o último item. Para remover outros, usamos o `splice`.

Esse método retorna um novo array baseado em posições de início e quantidade de itens passadas por parâmetro:

```
const meusNumeros = [10, 20, 30];
const numeroVinte = meusNumeros.splice(1, 1);
console.log(meusNumeros); // [10, 30]
console.log(numeroVinte); // [20]
```

Com esse método, podemos remover mais de um elemento:

```
const meusNumeros = [10, 20, 30];
const numeroVinte = meusNumeros.splice(0, 2);
console.log(meusNumeros); // [30]
console.log(numeroVinte); // [10, 20]
```

Laços de repetição

Muitas vezes precisamos *varrer* o array. Para isso temos alguns métodos, dentre eles:

- forEach
- map
- filter
- some

forEach

Esse método é usado para iterar de forma simples no array. Recebe uma função como parâmetro e a executa em cada item.

Exemplo:

```
const meusNumeros = [10, 20, 30];
meusNumeros.forEach(function (item) {
  console.log(item);
});
```

map

O `map` funciona de forma semelhante ao `forEach`, porém retorna um novo array com base no retorno da função executada em cada item:

```
const meusNumeros = [10, 20, 30];
meusNumeros.map(function (item) {
  const numeroSomadoCinco = item + 5;
  return numeroSomadoCinco;
});
console.log(meusNumeros); // [15, 25, 35]
```

filter

Utilizado para filtrarmos os itens de um array.

O método `filter` retorna um novo array contendo apenas os elementos onde a função recebida tenha retornado `true`:

```
const meusNumeros = [10, 20, 30];
const numerosMaioresQuinze = meusNumeros.filter(function (item) {
  if (item > 15) {
    return true;
  }
  return false;
});
console.log(numerosMaioresQuinze); // [20, 30]
```

OBJETOS

JavaScript é uma linguagem com suporte a OO, podendo representar objetos das seguintes formas:

- Notação literal
- Object.create()
- Função construtora
- Classes

Notação literal

Essa é a maneira mais rápida de criar objetos, apesar de não ser a mais usual.

Para criarmos um objeto basta usarmos um `inicializador de objeto`, ou `object literal`:

```
const pessoa = {
  nome: 'Nome',
  idade: 15
};
```

Aqui, `pessoa` é o nome do novo objeto. `nome` e `idade` são suas propriedades, com os valores `'Nome'` e `15`, respectivamente.

Também é possível criar um objeto vazio:

```
const objeto = {};
```

Object.create()

Esse método cria um objeto a partir de um `protótipo de objeto`, o qual é bastante semelhante a um `object literal`.

Exemplo:

```
const Pessoa = {
  nome: 'Nome',
  idade: 15
};
const novaPessoa = Object.create(Pessoa);
```

Função construtora

Essa é a forma que nos dá a maior liberdade na criação de objetos.

Criamos uma função construtora e usamos o `new` para construirmos uma instância do objeto:

```
function Pessoa() {
  this.nome = 'Nome';
```

```
  this.idade = 15;
}
const novaPessoa = new Pessoa();
```

Classes

O ES2015 trouxe mais uma bem-vinda novidade ao JS: classes.

Funciona de forma parecida a função construtora, porém mais simplificada:

```
class Pessoa {
  constructor() {
    this.nome = 'Nome';
    this.idade = 15;
  }
}
const novaPessoa = new Pessoa();
```

Propriedades

Propriedade de um objeto nada mais é que uma variável ligada à ele. Nos exemplos anteriores tínhamos as variáveis `nome` e `idade` ligadas aos nossos objetos.

Além disso, nos exemplos anteriores criamos as propriedades do objeto na hora de sua definição. Mas também podemos definir após o objeto já ter sido criado, de duas maneiras:

- Notação de ponto
- Notação de colchetes

Notação de ponto

Essa é a notação mais usada, por sua simplicidade e familiaridade com outras linguagens com suporte a OO.

Exemplo de uma classe:

```
class Pessoa {
  constructor() {
    this.nome = 'Nome';
    this.idade = 15;
  }
}
```

Aqui usamos a notação de ponto para introduzir duas propriedades no objeto a ser criado.

Para acessar essas propriedades, basta colocarmos o nome do nosso objeto, um ponto e o nome da propriedade em questão:

```
const novaPessoa = new Pessoa();
console.log(novaPessoa.idade); // 15
```

Da mesma forma, conseguimos adicionar propriedade a esse objeto:

```
const novaPessoa = new Pessoa();
novaPessoa.sobrenome = 'Sobrenome';
consoloe.log(novaPessoa.sobrenome); // Sobrenome
```

Também podemos alterar os valores das propriedades já definidas:

```
const novaPessoa = new Pessoa();
novaPessoa.idade = 23;
consoloe.log(novaPessoa.idade); // 23
```

Notação de colchetes

Aqui, como o nome diz, usamos uma string entre colchetes para acessar as propriedades de um objeto:

```
novaPessoa['sobrenome'];
```

Todas as operações possíveis na notação de ponto são possíveis com a de colchetes.

```
const novaPessoa = new Pessoa();
console.log(novaPessoa['idade']); // 15

novaPessoa['sobrenome'] = 'Sobrenome';
console.log(novaPessoa['sobrenome']); // Sobrenome
```

JAVASCRIPT OBJECT NOTATION

JSON, como é chamado, é um formato de troca de dados baseado em objetos JavaScript.

Foi criado por conta da dificuldade em trabalhar com estruturas em XML. É mais leve e mais fácil de ler.

Exemplo:

```
{
  "nome": "Nome",
  "idade": 15
}
```

Como podemos ver, é praticamente um literal. Apesar de as propriedades terem de ser declaradas com " (aspas duplas) em volta delas.

JSON e JS

JSON é um formato pensado especialmente para trabalhar em conjunto com o JS. Portanto, é extremamente simples transformar um objeto JS para um JSON:

```
const pessoa = {
  nome: 'Nome',
  idade: 15
}
const pessoaEmJson = JSON.stringify(pessoa);
console.log(pessoaEmJson); // "{ "nome": "Nome", "idade": 15 }"
```

O contrário também é verdade. Para convertermos um objeto JSON para JS basta usarmos o método `parse`:

```
const pessoaEmJson = '{ "nome": "Nome", "idade": 15 }';
const pessoaEmJs = JSON.parse(pessoaEmJson);
console.log(pessoaEmJs.idade); // 15
```

ANGULARJS

Definição

Superheroic JavaScript MVW Framework

AngularJS é um framework open-source mantido pelo Google com o objetivo de ajudar na criação de páginas dinâmicas.

Faz uso do HTML para criação de templates e o estende com diversas funcionalidades.

Inicialização

Para iniciarmos uma aplicação AngularJS, utilizamos a diretiva `ng-app` no elemento da página que irá conter a aplicação.

Exemplo:

```
<!DOCTYPE html>
<html ng-app="app">
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.4/angular.min.js"></script>
  </head>
  <body>
```

```
    <label>Insira seu nome:</label>
    <input type="text" ng-model="nome" />
    <h1>Olá {{ nome }}!</h1>
  </body>
</html>
```

Também é possível inicializar a aplicação manualmente criando um módulo e utilizando o método `bootstrap`:

```
angular.module('app', []);
angular.bootstrap(document, ['app']);
```

Exemplo:

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.4/angular.min.js"></script>
    <script>
      window.onload = inicializar;
      function inicializar() {
        angular.module('app', []);
        angular.bootstrap(document, ['app']);
      }
    </script>
  </head>
  <body>
    <label>Insira seu nome:</label>
    <input type="text" ng-model="nome" />
    <h1>Olá {{ nome }}!</h1>
  </body>
</html>
```

Controllers

Controllers são objetos utilizados para controlar a página, também chamada de `view`. São usados para definir todo o comportamento da view.

Exemplo anterior usando um controller:

```
<!DOCTYPE html>
<html ng-app="app">

  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.4/angular.min.js"></script>
    <script>
```

```
        angular.module('app', []);

        angular.module('app')
          .controller('PrimeiroController', PrimeiroController);

        function PrimeiroController($scope) {
          $scope.nome = 'João';
        }
      </script>
    </head>

    <body ng-controller="PrimeiroController">
      <label>Insira seu nome:</label>
      <input type="text" ng-model="nome" />
      <h1>Olá {{ nome }}!</h1>
    </body>

</html>
```

DATA BINDING

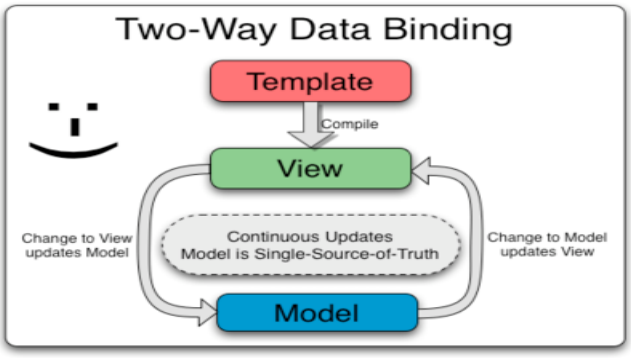
Data binding é o processo de sincronização automática feito pelo AngularJS. Foi desenvolvido para facilitar o controle dos dados exibidos na página.

Com esse processo, toda a alteração de dados realizada na página é automaticamente refletida no objeto a ela vinculado.

Nos exemplos anteriores podemos ver o AngularJS se encarregar de manter o `input[text]` e o texto contido no `h1` em sincronia, não sendo necessário que nos preocupemos com isso:

```
<input type="text" ng-model="nome" />
<h1>Olá {{ nome }}!</h1>
```

Ilustração do processo de Data Binding:



$SCOPE

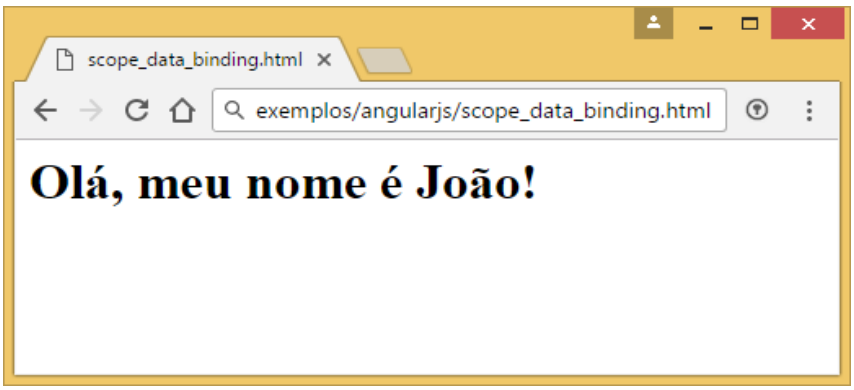
Scope é um objeto do AngularJS que se refere ao modelo de dados das views.

É usado como uma *cola* entre a view e o controller, fazendo a intermediação de dados entre essas duas camadas.

No exemplo abaixo podemos ver que criando a propriedade nome no $scope do controller reflete no elemento vinculado a ela na view.

```
<!DOCTYPE html>
<html ng-app="app">
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.4/angular.min.js"></script>
    <script>
      angular.module('app', []);
      angular.module('app')
        .controller('PrimeiroController', PrimeiroController);
      function PrimeiroController($scope) {
        $scope.nome = 'João';
      }
    </script>
  </head>
  <body ng-controller="PrimeiroController">
    <h1>Olá, meu nome é {{ nome }}!</h1>
  </body>
</html>
```

Tendo o seguinte resultado ao executarmos no navegador:



Para se aprofundar no assunto: **https://github.com/angular/angular.js/wiki/Understanding-Scopes**

$WATCH

Além de ser a liga entre o controller e a view, o objeto $scope possui algumas funções úteis.

O método `$watch` é uma dessas. Usado para avisar quando alguma propriedade é alterada.

No exemplo abaixo, será escrito no console o nome digitado a cada alteração:

```
<!DOCTYPE html>
<html ng-app="app">

  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.4/angular.min.js"></script>
    <script>
      angular.module('app', []);

      angular.module('app')
        .controller('PrimeiroController', PrimeiroController);

      function PrimeiroController($scope) {
        $scope.nome = 'João';

        $scope.$watch('nome', aoAlterarNome);
        function aoAlterarNome(novoValor, valorAntigo) {
          console.log($scope.nome);
        }
      }
    </script>
  </head>

  <body ng-controller="PrimeiroController">
    <label>Insira seu nome:</label>
    <input type="text" ng-model="nome" />
    <h1>Olá {{ nome }}!</h1>
  </body>

</html>
```

Diretivas

Diretivas são marcas interpretadas pelo AngularJS com a finalidade de adicionar ou alterar o comportamento do DOM ou seus elementos.

Pode-se, por exemplo, existir uma diretiva que altera a cor da fonte para azul ou todas as letras ficarem maiúsculas.

Para mais detalhes, consulte a documentação oficial em: **https://docs.angularjs.org/guide/directive** e **https://docs.angularjs.org/api/ng/directive**

DIRETIVAS NATIVAS

O próprio AngularJS traz algumas diretivas por padrão.

Algumas delas são:

- ng-app
- ng-controller
- ng-bind
- ng-model
- ng-checked
- ng-show
- ng-hide
- ng-if
- ng-repeat
- ng-click
- ng-change
- ng-class

`ng-app` e `ng-controller` já foram citadas em exemplos anteriores. A primeira define o escopo da aplicação e a segunda *aplica* um determinado controller àquele elemento.

ng-bind

Essa diretiva serve para vincular um dado a um elemento. O conteúdo da expressão irá substituir o conteúdo do elemento vinculado.

```
<h1 ng-bind="titulo"></h1>
```

ng-model

Também é usada para vincular um elemento a uma propriedade no escopo, porém restrita aos elementos `input`, `select` e `textarea`.

Assim como a ng-bind, atualiza o elemento com a propriedade vinculada. Mas também faz o inverso, atualiza a propriedade com o valor colocado no elemento.

```
<input type="text" ng-model="nome" />
```

ng-checked

Vincula a propriedade ao atributo checked do elemento em questão.

```
<input type="checkbox" ng-checked="maior" />
```

ng-show

Faz o elemento aparecer ou sumir se a expressão atribuída for true ou false, respectivamente.

```
<h3 ng-show="nome === 'Wiley'">Que nome legal!</h3>
```

ng-hide

Faz o oposto da ng-show. O elemento fica invisível quando a expressão tiver true como resultado.

```
<h3 ng-hide="time === 'Santo'">Time encontrado.</h3>
```

ng-if

Essa diretiva faz um efeito parecido à ng-show. Porém, ao invés de simplesmente fazer o elemento ficar invisível, o remove da DOM.

```
<h3 ng-if="maior === true">Conteúdo permitido</h3>
```

ng-repeat

Utilizada para listar vários elementos de uma lista.

```
<ul>
  <li ng-repeat="nome in nomes">{{ nome }}</li>
</ul>
```

ng-click

Define um método a ser executado quando o elemento é clicado.

```
<button ng-click="mostrarAlerta()">Mostrar alerta</button>
```

ng-change

Define uma expressão a ser executada quando o valor do elemento é alterado.

```
<input type="text" ng-model="nome" ng-change="aoAlterarNome()" />
```

ng-class

Atribui classes CSS dinamicamente ao elemento.

Podendo receber o nome das classes:

```
<p ng-class="'verde borda'">Verde com borda</p>
```

Um vetor as contendo:

```
<p ng-class="['verde', 'borda']">Verde com borda</p>
```

Ou um objeto com a seguinte estrutura:

```
{ 'nomeClasse': 'expressão' }
```

Por exemplo:

```
<p ng-class="{ 'verde': true, 'borda': true }">Verde com borda</p>
```

DIRETIVAS CUSTOMIZADAS

Muitas vezes precisamos ter um controle maior sobre o DOM e para isso criamos nossas próprias diretivas.

Para criar uma nova diretiva, podemos chamar a função `directive` informando seu nome e um objeto com algumas definições, chamado de `Directive Definition Object`.

A estrutura da chamada é:

```
directive('nomeDaDiretiva', funcaoRetornandoDDO);
```

Por exemplo, uma diretiva que exibe `Hello, World!` seria:

```
angular.module('app', []);

angular.module('app')
  .directive('nome', helloWorldDirective);

function helloWorldDirective() {
  const definitionObject = {
    template: '<span>Hello, World!</span>'
  };
  return definitionObject;
}
```

O nome da diretiva atribuído no método `directive` deve estar em `camelCase` e na view deve ser usada com `kebab-case`.

Tomemos a `ngModel` como exemplo, é usada na view como `ng-view`.

Esse object definition possui diversas propriedades. Vejamos algumas delas:

- template
- templateUrl
- restrict
- controller
- require
- link
- controllerAs
- scope
- bindToController

Para mais informações veja em: **https://docs.angularjs.org/api/ng/service/$compile#directive-definition-object**

template

Aqui definimos o corpo da diretiva, em HTML.

Tendo duas opções, uma string ou uma função retornando o markup.

Exemplo com string:

```
const definitionObject = {
  template: '<span>Hello, World!</span>'
};
```

Exemplo com função:

```
const definitionObject = {
  template: function (elemento, atributos) {
    return '<span>Hello, World!</span>';
  }
};
```

templateUrl

Tem o mesmo objetivo da `template`, porém recebe uma URL com o endereço do template a ser utilizado.

Também tendo as opções de string e função retornando a URL.

Exemplo com string:

```
const definitionObject = {
  templateUrl: 'endereco/do/template.html'
};
```

Exemplo com função:

```
const definitionObject = {
  templateUrl: function (elemento, atributos) {
    return 'endereco/do/template.html';
  }
};
```

restrict

Serve para definir o tipo da diretiva, como ela poderá ser referenciada na view.

Tendo como opções:

Opção	Significado	Exemplo
E	Nome do elemento	<diretiva></diretiva>
A	Atributo	
C	Classe	
M	Comentário	<!-- directive: diretiva '' -->

controller

Define o controller da diretiva. Podendo ser o próprio nome ou uma função construtora.

require

Especifica quais outras diretivas são requeridas pela atual.

O valor dessa propriedade pode ser:

- string com o nome de uma diretiva
- array contendo um ou mais nomes de diretivas
- um objeto cujas propriedades refletem os nomes das diretivas

Podemos usar alguns prefixos para dizer onde procurar as diretivas listadas nessa propriedade:

Prefixo	Onde procurar
nenhum	No próprio elemento da diretiva
^	No elemento e em seus ancestrais
^^	Apenas nos ancestrais do elemento

Um erro será disparado caso as diretivas listadas não sejam encontradas. Para torná-las opcionais, basta colocar um `?`. Exemplo:

```
const definitionObject = {
  require: ['?^ngModel']
};
```

link

A propriedade `link` é uma função responsável em registrar os eventos e controlar o DOM.

A função pode receber os seguintes parâmetros, nessa ordem:

- scope
- elemento
- atributos
- controller
- função do transclude

O parâmetro `controller` recebe o próprio controller da diretiva caso esta não tenha a propriedade `require` definida.

Caso `require` esteja definida, traz os controllers das diretivas ali mencionadas.

scope

Nessa propriedade definimos se e como será criado o escopo dessa diretiva, quando for usada em uma view.

Temos três opções:

Opção	Significado
`false`	Não criará escopo próprio, usará o do elemento superior
`true`	Criará um novo escopo, herdando todo o escopo pai
`{}` (objeto)	Cria um escopo isolado, com as propriedades definidas no objeto

Escopo isolado

Caso a opção tenha sido criar um escopo isolado, a diretiva não terá acesso aos *escopos pais*.

A comunicação dela se dará por meio de propriedades definidas no objeto atribuído à propriedade `scope`.

Exemplo, definindo o objeto:

```
const definitionObject = {
  scope: {
    texto: '@'
  }
};
```

Essas propriedades são refletidas como atributos no elemento, ao ser usada na view. Exemplo:

```
<diretiva texto="Hello, World!"></diretiva>
```

Tipo de vínculo

Também deve-se definir o tipo de vínculo a ser usado nessas propriedades. Para fazer essa definição, usamos símbolos no valor das propriedades.

Essas são as opções disponíveis:

Símbolo	Significado
@	O valor será interpretado como texto
=	Ligação bidirecional (two way binding)
<	Ligação unidirecional
&	Expressão que será traduzida para uma função no escopo da diretiva

Vínculo opcional

Esses vínculos estabelecidos como necessários para o funcionamento da diretiva são obrigatórios, fazendo com que um erro será disparado caso algum não seja informado.

Porém, podemos deixá-los opcionais colocando um ? (ponto de interrogação) à frente do seu tipo:

```
const definitionObject = {
  scope: {
    texto: '?@'
  }
};
```

Nomeando vínculos

Podemos definir nomes diferentes para serem usados externamente. Para isso, basta inserir o nome após o tipo de vínculo:

```
const definitionObject = {
  scope: {
    texto: '@meuTexto'
  }
};
```

Dessa forma, a diretiva deverá ser utilizada conforme abaixo:

```
<diretiva meu-texto="Hello, World!"></diretiva>
```

Assim como o nome da diretiva, o nome da propriedade deve ser definido com camelCase e na view com kebab-case.

controllerAs

Para facilitar a identificação das propriedades usadas na view, podemos dar nomes aos nossos controllers:

```
<body ng-controller="NomeController as nomeCtrl">
  <label>Insira seu nome:</label>
  <input type="text" ng-model="nomeCtrl.nome" />
  <h1>Olá {{ nomeCtrl.nome }}!</h1>
</body>
```

Com essa mudança, as propriedades são adicionadas diretamente na instância do controller. Não sendo necessário o uso de $scope:

```
angular
  .module('app')
  .controller('NomeController', NomeController);

function NomeController() {
  this.nome = 'João';
}
```

DIRETIVAS

Também é possível nomear os controllers nas diretivas, usando a propriedade controllerAs do objeto de definição:

```
const objectDefinition = {
  controllerAs: 'diretivaCtrl'
}
```

Mas com uma particularidade, as propriedades definidas em scope não são vinculadas à controller automaticamente. Ainda são acessadas via $scope:

```
const objectDefinition = {
  scope: {
    texto: '@'
  },
  controller: function($scope) {
    console.log(this.texto); // undefined
```

```
    console.log($scope.texto); // Hello, World!
  },
  controllerAs: 'diretivaCtrl'
}
```

bindToController

Esse comportamento pode ser alterado com a propriedade `bindToController`, a definindo como `true`:

```
const objectDefinition = {
  scope: {
    texto: '@'
  },
  controller: function() {
    console.log(this.texto); // Hello, World!
  },
  controllerAs: 'diretivaCtrl',
  bindToController: true
}
```

Também podemos simplificar removendo o objeto `scope`, movendo seu conteúdo para `bindToController`:

```
const objectDefinition = {
  bindToController: {
    texto: '@'
  },
  controller: function() {
    this.$onInit = function() {
      console.log(this.texto); // Hello, World!
    };
  },
  controllerAs: 'diretivaCtrl'
}
```

Componentes

Apesar de o *Angular 2+* ter sido lançado, o *AngularJS 1.x* vem sendo atualizado com algumas melhorias.

Essas atualizações vieram para facilitar a migração para a versão 2+ e deixar a aplicação melhor.

Uma delas trouxe os chamados componentes. Agora além do método `directive`, temos `component` como opção para criarmos *pedaços* de tela.

Apesar de ter outro nome, `component` é apenas uma diretiva com comportamento diferente do padrão. Portanto, uma diretiva pode fazer tudo que um componente faz.

A adição desse método pode reduzir a quantidade de código escrito e a chance de introduzirmos *bugs* na aplicação.

UTILIZAÇÃO

Como dito anteriormente, para criar um componente é usado o método `component`.

Vamos usar como exemplo a nossa diretiva `helloWorld`:

```
angular
  .module('app')
  .directive('helloWorld', helloWorldDirective);

function helloWorldDirective() {
  const definitionObject = {
    template: '<span>Hello, World!</span>'
  };
  return definitionObject;
}
```

Agora *transformando-a* em componente:

```
angular
  .module('app')
  .component('helloWorld', {
    template: '<span>Hello, World!</span>'
  });
```

Em `component` o segundo parâmetro deve ser um object definition, não uma função que o retorna como em `directive`.

BINDINGS

Assim como diretivas, componentes podem receber parâmetros na sua chamada. A definição desses parâmetros agora se dá pela propriedade `bindings`:

```
const objectDefinition = {
  bindings: {
    texto: '@'
  },
  controller: function() {
    this.$onInit = function() {
      console.log(this.texto); // Hello, World!
    };
  }
}
```

COMPONENTE VS DIRETIVA

Além do já mencionado, existem algumas diferenças entre uma diretiva e um componente.

Como podemos ver no exemplo abaixo, tudo declarado no objeto `bindings` é automaticamente vinculado ao controller. Com isso, foram retiradas as propriedades `scope` e `bindToController`.

bindings

O objeto `bindings` é a união dessas duas propriedades:

```
angular
  .module('app')
  .component('helloWorld', {
    bindings: {
      texto: '@'
    },
    controller: function() { }
  });
```

Funciona exatamente como uma diretiva usando `bindToController`:

```
angular
  .module('app')
  .directive('helloWorld', function () {
    return {
      bindToController: {
        texto: '@'
      },
      controller: function() { }
    }
  });
```

controllerAs

A propriedade `controllerAs` ainda existe, porém, o valor padrão passou a ser `$ctrl`.

Escopo

Todo componente tem seu escopo isolado por padrão, não podendo ser alterado.

link

A função não existe em componentes. Caso seja necessário controlar o DOM deve-se usar uma diretiva.

Obtendo dados

Praticamente toda aplicação necessita de dados para funcionar. Seja o nome de quem está usando o sistema, uma lista de tarefas, os dados para o cadastro de um cliente, etc.

Esses dados normalmente estão em algum servidor, podendo ser acessados através de uma URL.

Para obtermos esses dados, fazemos uma requisição para o servidor, tratamos o retorno e mostramos os dados ao usuário, caso necessário.

Essa requisição normalmente é assíncrona, chamada de requisição AJAX.

AJAX

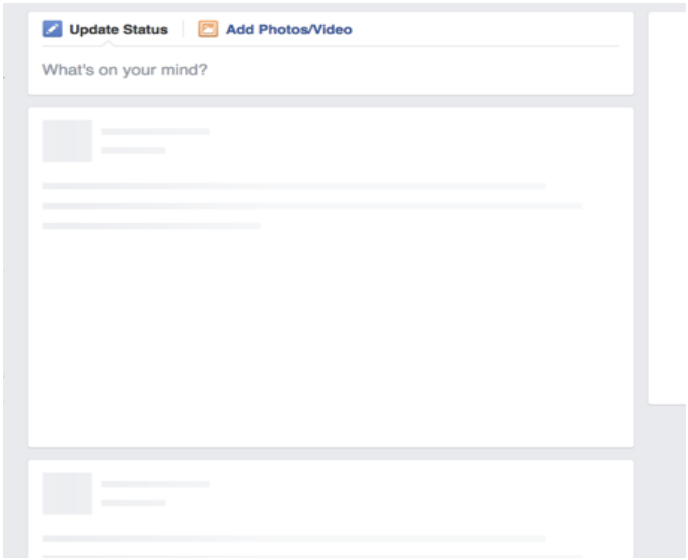
AJAX é um acrônimo para *Asynchronous JavaScript And XML*.

Como o próprio nome sugere, é uma técnica para fazer requisições assíncronas pelo JavaScript usando XML.

Apesar de referenciar XML no nome, hoje em dia JSON é muito mais usado para esse tipo de requisição por conta da sua praticidade.

Por ser assíncrona, podemos realizar requisições desse tipo sem deixar o usuário esperando o carregamento da página em si.

Podemos, por exemplo, carregar todo o HTML e CSS para exibirmos uma página básica e depois carregar os dados para assim o usuário poder interagir com a página.



Facebook carregando dados de forma assíncrona

XMLHTTPREQUEST

Para fazermos uma requisição AJAX, basta usarmos o objeto `XMLHttpRequest` já presente nos navegadores:

```
let httpRequest = new XMLHttpRequest();
httpRequest.onreadystatechange = aoMudarStatus;
httpRequest.open('GET', 'url/para/requisicao', true);
httpRequest.send(null);

function aoMudarStatus() {
  if (httpRequest.readyState === XMLHttpRequest.DONE) {
    alert(httpRequest.responseText); // mostra resposta do servidor
  }
}
```

Não precisamos de nenhum framewok ou biblioteca para fazer requisições AJAX, porém esse código pode ficar bem grande dependendo do seu objetivo.

Caso precise rodar em outros navegadores, pode ficar assim:

```
const httpRequest;
if (window.XMLHttpRequest) {
  httpRequest = new XMLHttpRequest();
} else if (window.ActiveXObject) { // IE
  try {
    httpRequest = new ActiveXObject("Msxml2.XMLHTTP");
  }
  catch (e) {
    try {
      httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
    }
    catch (e) {}
  }
}
httpRequest.onreadystatechange = aoMudarStatus;
httpRequest.open('GET', 'url/para/requisicao', true);
httpRequest.send(null);

function aoMudarStatus() {
  if (httpRequest.readyState === XMLHttpRequest.DONE) {
    // mostra resposta do servidor no console
    console.log(httpRequest.responseText);
  }
}
```

JQUERY.AJAX

Com o intuito de facilitar a vida dos desenvolvedores, várias bibliotecas surgiram. Uma delas foi a jQuery.

jQuery trouxe uma infinidade de ferramentas úteis à época para o desenvolvimento web, ficando popular rapidamente. Muitos ainda a usam hoje em dia.

Dentre todas as facilidades, o método `ajax` diminuía consideravelmente o tamanho do código para fazer uma requisição ajax. O próprio jQuery tratava de verificar qual era o navegador e usava o objeto certo para fazer a requisição.

Exemplo com a mesma chamada AJAX feita anteriormente com JS nativo, com jQuery:

```
jQuery.ajax({
   type: 'GET',
   url: 'url/para/requisicao',
   async: true,
   success: function(data) {
      console.log(data);
   }
});
```

Ou ainda:

```
jQuery.get('url/para/requisicao', function(data) {
   console.log(data);
});
```

$HTTP

O AngularJS é um framework completo e também traz uma solução para realizarmos requisições AJAX: `$http`.

Esse objeto é um serviço baseado nos métodos providos pelo jQuery, portanto possui muita semelhança.

Exemplo de uma simples requisição:

```
$http({
   method: 'GET',
   url: 'url/para/requisicao'
}).then(function(response) {
   console.log(response.data);
});
```

Também tendo formas simplificadas:

```
$http
  .get('url/para/requisicao')
  .then(function(response) {
    console.log(response.data);
  });
```

Alguns dos métodos disponíveis nesse serviço e sua descrição:

Método	Descrição
GET	Obter informações do servidor
POST	Criar informações no servidor
PUT	Atualizar informações no servidor
DELETE	Excluir informações do servidor

Os métodos de criação, atualização ou exclusão devem enviar quais informações no servidor devem ser alteradas.

Para isso, basta informar na propriedade `data` do objeto de configuração:

```
$http({
  method: 'POST',
  url: 'url/para/criar/usuario',
  data: {
    nome: 'João',
    idade: 15
  }
});
```

PROMISES

Promise é um objeto usado para fazermos chamadas assíncronas e esperar sua resolução (ou falha) para continuar o trabalho.

Uma requisição AJAX é o típico cenário onde a aplicação pode ter de esperar muito tempo até os dados serem retornados pelo servidor, principalmente com uma conexão lenta.

O serviço `$http` do AngularJS já implementa uma interface de promise e a expõe para usarmos, basta usar o método `then` do objeto retornado pelo serviço:

```
// O serviço $http retorna um objeto de promise, o qual podemos guardar em uma variável qualquer
const requisicaoPromise = $http({
  method: 'GET',
  url: 'url/para/requisicao'
});
```

```
// Assim podemos usar o método exposto nesse objeto para esperarmos a conclusão da requisição
requisicaoPromise.then(function (response) {
  // essa função será executada com os dados retornados pelo servidor
});
```

O método `then` recebe dois parâmetros, um para quando a chamada terminou com sucesso e outra para a ocorrência de alguma falha:

```
const requisicaoPromise = $http({
  method: 'GET',
  url: 'url/para/requisicao'
});

requisicaoPromise.then(aoObterSucesso, aoOcorrerFalha);

function aoObterSucesso(response) {}
function aoOcorrerFalha(response) {}
```

Ambos os métodos recebem um objeto `response`, o qual possui as seguintes propriedades:

Propriedade	Significado
`data`	O corpo da resposta, contém os dados requisitados
`status`	Um código HTTP informando o status da requisição
`headers`	O Cabeçalho da requisição
`config`	O objeto de configuração usado
`statusText`	O texto referente ao `status`

Serviços

Serviços são objetos providos pelo AngularJS, ou criados por nós, para prover alguma funcionalidade.

`$http` é um exemplo de serviço nativo do AngularJS.

Para saber sobre todos os serviços nativos veja em: **https://docs.angularjs.org/api/ng/service**

Existem três tipos de serviços no AngularJS:

- Providers
- Factories
- Services

Todo serviço é um objeto `singleton`, ou seja, a mesma instância é usada em toda a aplicação.

Vamos focar nos dois últimos tipos.

FACTORIES

É a principal forma de se fazer serviços no AngularJS.

Como o próprio nome diz, funciona como uma fábrica, podendo retornar qualquer tipo de objeto.

Exemplo:

```
angular
  .module('app')
  .factory('minhaFactory', minhaFactory);
function minhaFactory() {
  const objeto = {
    nome: 'João'
  };
  return objeto;
}
```

Com isso, podemos usar essa factory em nossos controllers:

```
angular
  .module('app')
  .controller('meuController', meuController);
function meuController(minhaFactory) {
  console.log(minhaFactory.nome); // João
}
```

Podemos, por exemplo, fazer a factory retornar uma simples função:

```
angular
  .module('app')
  .factory('minhaFactory', minhaFactory);
function minhaFactory() {
  return function() {
    console.log('sou uma função');
  };
}
```

```
angular
  .module('app')
  .controller('meuController', meuController);
function meuController(minhaFactory) {
  // Podemos executar a factory
  minhaFactory(); // sou uma função
}
```

SERVICES

Diferentemente de uma factory, um service não pode ser usado para retornar qualquer tipo de dado.

O AngularJS trata esse tipo de serviço como uma classe, construindo um objeto a partir dela quando for requisitada por algum outro serviço ou controller.

Sendo assim, podemos definir um service como uma classe:

```
class MeuController() {
  constructor() {
    this.nome = 'Joào';
  }
}
angular
  .module('app')
  .service('meuService', MeuController);
```

Injeção de dependência

Esse é um tópico que merece uma atenção especial, inclusive tendo uma página própria no guia oficial do AngularJS: **https://github.com/angular/angular.js/wiki/Understanding-Dependency-Injection**

O termo *injeção de dependência* é um design pattern onde os objetos recebem as suas dependências, ao invés de criá-las.

Por exemplo, um `Carro` depende de `Motor` para andar. Sem esse pattern a representação em classes seria algo como:

```
class Carro() {
  constructor() {
    this.motor = new Motor();
  }
}
class Motor() {}
```

O problema disso é o alto acoplamento entre as duas classes, fazendo com que uma alteração na construção de `Motor` reflita em uma alteração em `Carro`.

Para amenizarmos esse problema, usamos a injeção de dependência. E o código ficaria assim:

```
class Carro() {
  constructor(motor) {
    this.motor = motor;
  }
```

```
}
class Motor() {}
```

`Carro` não é mais responsável pela construção do objeto `motor`, apenas o recebe.

Mas ainda assim alguém precisa construir esse objeto `motor` e para isso o AngularJS conta um *sistema de injeção de dependência* próprio.

Dessa forma, o próprio AngularJS se encarrega de construir essa instância de motor e assim *injetar* na classe `Carro`.

Com esse sistema, os objetos (controllers, diretivas, etc.) podem simplesmente definir quais são as suas dependências no seu construtor e o framework se encarregará de injetá-las.

Há três formas de usarmos essa injeção de dependência:

- Implícita
- Inline Array
- Propriedade `$inject`

ANOTAÇÃO IMPLÍCITA

Quando colocamos no construtor de nossos controllers ou services que dependemos de algum outro objeto, o AngularJS tenta buscar essa referência usando o nome da variável.

Por exemplo, aqui o AngularJS procurará um service chamado `$http` e um `meuService`:

```
class MeuController
  constructor ($http, meuService) {  }
}
angular
  .module('app')
  .controller('meuController', MeuController);
```

ANOTAÇÃO INLINE ARRAY

Também podemos usar os nomes dos serviços a serem injetados com uma string.

Para isso, ao invés de passarmos uma função no segundo parâmetro do método `controller`, usamos um array contendo todas as dependências e a função em seguida:

```
class MeuController
  constructor ($http, meuService) {  }
}
angular
  .module('app')
  .controller('meuController', ['$http', 'meuService', MeuController]);
```

Dessa forma o AngularJS procurará as referências com o nome informado no array e assim podemos colocar qualquer nome nas variáveis recebidas no construtor:

```
class MeuController
  constructor (request, servicoEhMeu) {  }
}
angular
  .module('app')
  .controller('meuController', ['$http', 'meuService', MeuController]);
```

Podemos até colocar todos os parâmetros em uma variável:

```
class MeuController
  constructor (request, servicoEhMeu) {  }
}
const params = ['$http', 'meuService', MeuController];
angular
  .module('app')
  .controller('meuController', params);
```

ANOTAÇÃO COM PROPRIEDADE $INJECT

Aqui usamos uma propriedade no serviço ou controller que estamos criando, funcionando de forma semelhante a usar uma variável.

Essa propriedade recebe um array contendo todas as dependências do objeto em questão. Dessa forma só precisamos passar o construtor para o método de criação, assim como era com a anotação implícita:

```
class MeuController
  constructor ($http, servicmeuServiceoEhMeu) {  }
}
MeuController.$inject = ['$http', 'meuService'];
angular
  .module('app')
  .controller('meuController', MeuController);
```

INJEÇÃO DE DEPENDÊNCIA ESTRITA

Apesar de termos maneiras de fazer a anotação de forma implícita e explícita, não é recomendado o uso da primeira.

Quando o código passa por um processo de minificação, para deixá-lo mais leve, todas as variáveis podem ser renomeadas para nomes completamente diferentes.

Tomemos esse controller como exemplo:

```
class MeuController
  constructor ($http, meuService) {  }
}
angular
  .module('app')
  .controller('meuController', MeuController);
```

Depois de um processo de minificação pode ficar assim:

```
class A { constructor (x, y) { } }; angular.module('app').controller('meuController', A);
```

Dessa forma o AngularJS não conseguirá identificar os serviços ali necessários, já que os nomes não correspondem a nenhum serviço chamado `x` ou `y`.

Por não ser uma boa prática, o AngularJS dispõe de uma diretiva usada para identificar esse tipo de problema e proibir o seu uso: `ng-strict-di`.

Basta colocarmos no mesmo elemento onde iniciamos a aplicação e o modo estrito é ativado:

```
<!DOCTYPE html>
<html ng-app="app" ng-strict-di>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.4/angular.min.js"></script>
  </head>
  <body>
    <label>Insira seu nome:</label>
    <input type="text" ng-model="nome" />
    <h1>Olá {{ nome }}!</h1>
  </body>
</html>
```