CSS-Eficiente

Especificidade CSS

Existem 4 categorias que definem o nível de especificidade:

1.Estilos inline

2.IDs

3.Classes, pseudoclasses e atributos

4.Elementos e pseudoelementos

A regra de aplicação do css visa os números acima, ou seja se tiver um número nas primeiras casas o estilo que se sobressairá serão os primeiros níveis.

Exemplo:

#content h1 = 0,1,0,1

Podemos usar **!important** para sobressair um estilo em detrimento de todos os outros.

No caso de **empate** o princípio de cascata entrara em ação.

SMACSS:

• Base;

• Layout;

• Module (Módulo);

• State (Estado);

• Theme (Tema);

Base: são as regras padrão. São quase que exclusivamente seletores de elementos, mas também pode haver seletores de atributos, pseudoclasses ou seletores mais avançados, como seletores-irmãos.

Layout: dividem a página em seções, podendo conter 1 ou mais módulos.

Module: contém as regras principais de um projeto, as que dão o “volume” de CSS. Elas são, como indica o próprio nome, as partes modulares e, consequentemente, as mais usadas.

State: descrevem como determinado layout ou módulo se comporta em determinada condição ou “estado” (ativo ou inativo; oculto ou visível). Também descrevem como um módulo pode ser diferente dependendo da página em que está; por exemplo, ser apresentado de um jeito na página inicial e de outro em uma interna.

Theme: são similares às Estado, haja vista que descrevem como layouts ou módulos devem se parecer em determinadas situações. A maioria dos sites não precisa de temas, mas é bom estar preparado para quando for o caso.

Pre-Processadores:

Um pré-processador é um programa que recebe texto e efetua conversões léxicas nele. As conversões podem incluir substituição de macros, inclusão condicional e inclusão de outros arquivos.

Algumas possibilidades com pré-processadores são:

• Usar variáveis;

• Usar funções;

• Aninhamento de regras;

• Operações aritméticas;

• Estender regras a partir de regras pré-existentes

Pre-processadores mais conhecidos:

• Sass (http://sass-lang.com/) ;

• Less (http://lesscss.org/) ;

• Stylus (http://learnboost.github.io/stylus/) .

SASS

Aninhamento de Regras

O CSS não possui suporte a estrutura hierárquica, porém o sass oferece esse suporte.

Exemplo:

CSS:

```
nav ul {
list-style: none;
margin: 0;
padding: 0;
 }
nav li {
display: inline-block;
 }
nav a {
 display: block;
 padding: 6px 12px;
 text-decoration: none;
}
```

SASS:

```
nav {
ul {
 margin: 0;
 padding: 0;
 list-style: none;
 }
li {
 display: inline-block;
 }
 a {
```

```
 display: block;

 padding: 6px 12px;

 text-decoration: none;

}

 }
```

Como compilar:

O html não lê os arquivos .sass ou .scss então devemos compilar eles e transformarem em css para que possam ser aplicados a pagina.

Para compilarmos, iremos navegar ate a pasta do arquivo pelo cmd, uma forma de encurtar isso é abrindo a pasta e no explorador de arquivos escrever cmd.

Com isso vamos usar o comando:

sass nomedoarquivo.scss:nomedoarquivo.css

Referência ao ascendente:

No sass algumas vezes é necessário referenciar a classe ascendente, como por exemplo quando formos acrescentar um **hover** ficaria como: **&:hover{}** o **&** significa o nome da classe ascendente.

Sempre que, em Sass, você deparar-se com &, já sabe que a intenção é repetir o seletor ascendente imediato na regra, independente de que parte dela esteja.

Variaveis:

Variáveis em Sass são como variáveis em qualquer linguagem de programação: referências nominais capazes de armazenar valores que podem ser chamadas em trechos de código subsequentes para resgatar e usá-los conforme necessário.

Para declarar uma variável, basta dar um nome qualquer (sem espaços e/ou caracteres especiais) precedido de $, usar : e dar o valor que se queira, como em:

```
$mainColor: #c0ffee;
```

Variáveis podem conter quaisquer valores usados em CSS

Interpolação de variáveis:

Para se fazer isso, referencia-se a variável usando **#{VARIAVEL}** e, automaticamente, a interpolação acontece.

Mixins:

Mixins permitem que se façam agrupamentos de declarações CSS para serem reusados onde se queira.

Para se trabalhar com eles, o par **@mixin**/ **@include** sempre estará presente. O primeiro para definir o mixin, em si; o segundo, para indicar em qual ponto do código se quer usá-lo.

```
@mixin border-radius($radius) {
 -webkit-border-radius: $radius;
 -moz-border-radius: $radius;
 -ms-border-radius: $radius;
 border-radius: $radius;
 }
 .box {
 @include border-radius(10px);
 }
```

Extensão/Herança:

No caso específico de Sass, é possível usar **@extend** para compartilhar uma série de propriedades/valores de várias regras diferentes em uma mesma regra.

Como exemplo, suponha que você tenha esta regra:

```
 .default-box {
 background-color: #efefef;
 border: 1px solid #000;
 color: #333;
 }
```

Caso seja preciso criar variações disso com Sass, uma das maneiras possíveis seria estendê-la em outra regra! Assim:

```
 .alert-box {
```

```
 @extend .default-box;
 font-size: 2em;
 }
```

o que compilaria para:

```
 .default-box, .alert-box {
background-color: #efefef;
 border: 1px solid #000;
 color: #333;
}
 .alert-box {
 font-size: 2em;
 }
```

Percebe o poder que isso oferece? Você pode ter um arquivo com definições genéricas de regras que podem ser estendidas em quaisquer outras do projeto.

Seletores placeholder:

pode haver situações em que determinadas regras que serão estendidas só precisem existir para isso e não precisem estar presentes no CSS compilado. Para isso, saiba que existem os seletores placeholder (placeholder selectors).

Criar um placeholder selector é como uma regra comum, com a diferença de que não se coloca um elemento, classe ou ID, mas sim um %. Por exemplo:

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

Para indicar a extensão:

```
.my-module {
@extend %bold;
border: 1px solid #ccc;
}
```

A grande diferença está no CSS compilado, que seria somente:

```
.my-module {
 display: block;
 }
.my-module {
 border: 1px solid #ccc;
 }
```

Outra ocasião bastante útil para eles é quando se tem que usar fontes personalizadas em projetos. Dá pra fazer algo como:

```
 @font-face {
 font-family: 'custom_font';
 src: url('font/my-custom-font.eot');
 src: url('font/my-custom-font.eot?#iefix') format('embedded-opentype'),
url('font/my-custom-font.woff2') format('woff2'),
 url('font/my-custom-font.woff') format('woff'),
 url('font/my-custom-font.ttf') format('truetype');
 font-weight: normal; font-style: normal;
 }

%custom-font {
 font-family: 'custom_font';
 }
```

Importação:

Para trabalhar com diversos arquivos o Sass oferece um recurso de importação, usando **@import**.

Podemos colocar um ou mais arquivos no import.

Quando acrescentamos _ antes do nome do arquivo, esse arquivo se torna um **partial**, isso significa que na compilação, partials não geram sua contraparte .css, ou seja apenas o css do arquivo que esta importando o partial será carregado, no **@import** não é necessário colocar o _ no nome do arquivo partial.

Podemos também importar arquivos de subdiretórios exemplo: @import 'base/base'; isso significa que o arquivo “base” está na subpasta base que está na mesma pasta que o arquivo que está importando o arquivo.

Mais Características do Sass:

Sass é uma linguagem poderosíssima, que facilmente encheria um livro inteiro, além do que já foi apresentado:

• Operadores aritméticos (+, -, *, /);

• Estruturas de controle (if, while, for, each);

• Diferentes resultados de compilação (nested, expanded, compact, compressed);

• Funções personalizadas. Tudo isso você encontra no site oficial do Sass (http://sass-lang.com/) .

Css Namespaces:

Tentando explicar de maneira simples e sucinta, namespaces são como regiões no código, nas quais nomes de variáveis, de funções etc. são válidos dentro destas linguagens de programação.

Na prática, trata-se apenas de colocar um prefixo nos nomes dos seletores para conseguir atribuir um namespace e começar a se valer dos benefícios de imediata identificação e melhor compreensão do código.

Os namespaces de css mais conhecidos são:

• o-: objeto;

• c-: componente;

• u-: utilitário;

• t-: tema;

• s-: escopo (scope);

• is-/has-: estado/ condição;

• _ : hack (!);

• js-: JavaScript;

• qa-: quality assurance (QA).

Task Runners:

Task runners são ferramentas que se destinam a automatizar tarefas variadas que devem acontecer em determinados momentos do desenvolvimento.

De maneira semelhante aos pré-processadores CSS (), no mundo da automatização de tarefas, também há dois task runners que são usados pela maioria das pessoas:

• Grunt (http://gruntjs.com/) ;

• Gulp (http://gulpjs.com/) .

Como instalar o Gulp:

Gulp é JavaScript com Node. A primeira coisa que você precisará é instalar o Node.js instruções em http://nodejs.org/. Tomando por base que você está

Depois disso, instale o Gulp globalmente no sistema com:

npm install -g gulp

Já que estamos lidando com Node, um arquivo package.json na raiz do projeto faz-se necessário para indicar quais módulos serão usados. É possível que o Node o crie automaticamente com npm init, o que iniciará um prompt interativo que perguntará um monte de coisas (tais como: nome do projeto, versão, descrição, licença, autor etc.).

Caso queira ser mais prático basta executar:

echo "{}" > package.json

Já com o arquivo package.json na raiz do projeto com um JSON válido, execute:

npm install --save-dev gulp

Com isso, o Gulp já está instalado e pronto para ser usado no projeto.

- npm install -g gulp
- echo "{}” > package.json
- npm install --save-dev gulp

gulpfile.js:

Agora que o básico para se mexer com Gulp está preparado, é hora do gulpfile.js. Nesse arquivo, que também deve ficar na raiz, é onde os scripts das tarefas instaladas constam. É aqui, efetivamente, que os comandos sobre o que fazer e como fazer são passados para o Gulp.