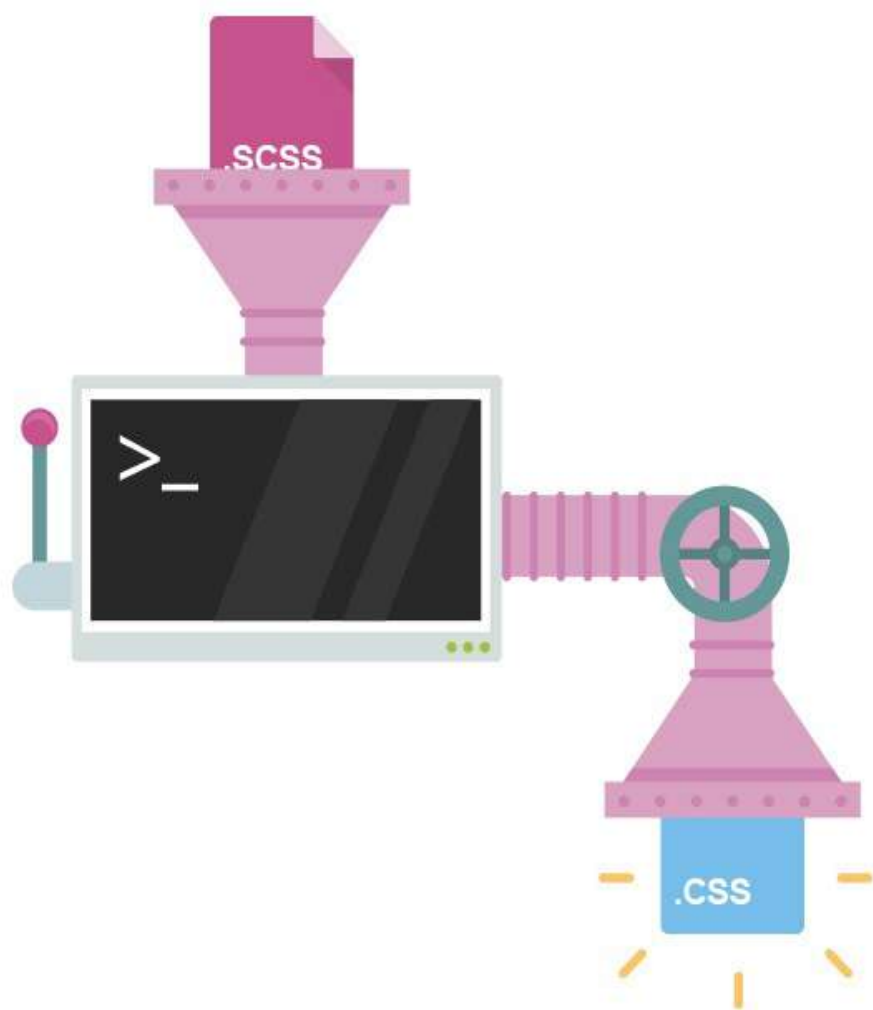


Sass

Aprendendo pré-processadores CSS



© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios:
eletrônicos, mecânicos, gravação ou quaisquer outros.

Adriano Almeida

Livros para o programador

Rua Vergueiro, 3185

www.casadocodigo.com.br

ISBN

- Impresso e PDF: 978-85-5519-204-3
- Epub: 978-85-5519-205-0
- Mobi: 978-85-5519-206-7

AGRADECIMENTOS

"I am Alpha and Omega, the beginning and the end, I will give unto him that is athirst of the fountain of the water of life, freely." — Revelation 21:6

Quando um dos meus chefes deu a ideia *"E se você fizer um livro de Sass?"*, achei inocentemente que seria o mestre do foco e escreveria em poucas semanas. Gosto de terminar as coisas que começo. Mas, rapaz, escrever é complicado. Exige uma dedicação fora do comum. Um post que você faz em meia hora não se compara em nada com isso.

Mas, no fim, deu tudo certo e a prova é que você está aí lendo este livro. Contudo, não o escrevi sozinho.

Quero agradecer a Caelum e todos os "caelumers" por me ajudarem direta, ou indiretamente, na concepção deste livro, desde um papo motivador até as revisões técnicas e gramaticais. A todos meus alunos que sempre me ensinaram algo a mais em todas as aulas que ministrei.

E muito obrigado a você, leitor, que investiu seu dinheiro neste livro e investirá sua dedicação a ele.

Dedico este livro a Paula Midori, Claudia e Ivo, pelas enormes doses de paciência, amor e conselhos que me dão diariamente. São meus três fortes pilares que tenho na vida e tenho certeza de que, sem eles, não seria nada.

Também deixo minha lembrança póstuma a Issao Nakayama, com quem convivi relativamente pouco, mas que com certeza sempre será lembrado.

SOBRE O AUTOR

Natan Souza é front-end designer no grupo Caelum desde 2015, e instrutor dos cursos presenciais de front-end e UX. Além disso, também produz cursos online dessas áreas para a Alura, incluindo os cursos de Sass e LESS.

Começou a dar seus primeiros cliques no Photoshop ainda em 2005, o que o levaria a se interessar pela área de Design, e graduando-se bacharel em Design Digital anos mais tarde.

Está focado na área de web e UX desde 2009, passando por empresas como FIAP e PMESP. Atuou como front-end e designer em toda a sua trajetória profissional.

- Twitter: @designernatan
- LinkedIn: <http://bit.ly/linkedinDoNatan>
- Site: <http://www.designernatan.com.br>

PREFÁCIO

Não importa se você já trabalha com front-end há dois, cinco, ou mesmo dez anos. Nossa área é um ser totalmente orgânico que muda constantemente, o tempo todo. Toda dia surge alguma técnica, framework ou linguagem nova. E caso um desses vire "moda" e caia no gosto dos desenvolvedores por algum motivo, lá vamos nós mais uma vez estudar e adaptar nossa rotina para abraçar a novidade.

Sempre há outra opção. Você pode muito bem se fechar na sua zona de conforto e negar tudo de novidade que vem de fora — o que acredito ser uma atitude bem compreensível, visto que o medo do desconhecido está impregnado em nosso DNA. Apesar de compreensível, pessoalmente acredito que essa escolha seja bem perigosa, uma vez que existe muito risco envolvido, e você pode acabar parado no tempo.

Depois de um certo amadurecimento de nossa área, alguns desenvolvedores começaram a ficar chateados por algumas deficiências que o CSS comum tinha na época, como a impossibilidade de criar variáveis ou mesmo aninhar regras CSS. Isso foi um dos motivos de começarem a surgir tecnologias que suprissem essas necessidades, os chamados **pré-processadores CSS**.

Isso aconteceu aproximadamente de 2006 para cá, e os que mais se destacaram foram o **Sass** e o **LESS**. Ambos começaram a ser assunto de posts e palestras poucos anos depois em toda a área de front-end. E até outros grandes nomes surgiram, como o Myth e o Stylus, tendo este último ganhado muitos holofotes de uns tempos para cá.

O problema de qualquer pré-processador, pegando o Sass e o

LESS como exemplo, é que os browsers não os entendem nativamente, mesmo sendo linguagens de estilos. A única linguagem desse tipo que os browsers compreendem atualmente é o bom e velho CSS. E é justamente por esse motivo que é necessário pegar códigos feitos em Sass/LESS/*SeuPreProcessadorFavoritoAqui* e compilá-los em CSS comum, para que assim o browser consiga entendê-lo de fato. Algo que ilustro com a figura:

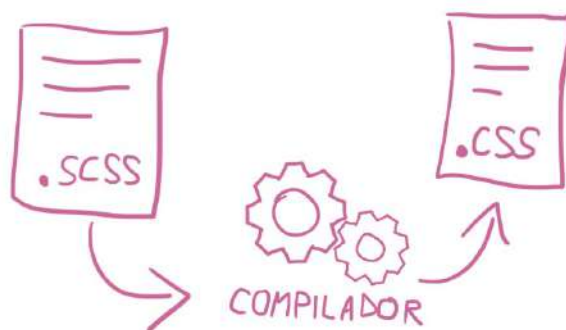


Figura 1: Entra Sass/LESS, sai CSS

Com isso em mente, um pré-processador é basicamente um programa que pega alguns dados como entrada, e devolve-os de uma forma diferente, tal que outro programa consiga entendê-los. No nosso caso, aqui neste livro, os dados de entrada serão arquivos `.scss` ou `.sass`, que são compilados em um arquivo `.css`, podendo assim ser lido pelos browsers.

Algumas empresas que atualmente utilizam o Sass como pré-processador são o Dropbox, o Walmart e o Airbnb!

O objetivo deste livro é mostrar de forma totalmente prática o **Sass**, mostrando algumas de suas funcionalidades por um caminho **mais "mão na massa", e menos documentação**. E por qual razão escolher o Sass? Simplesmente é o pré-processador mais utilizado atualmente.

Como pré-requisitos, é fortemente aconselhável que você já conheça bem HTML5 e CSS3. Este livro é focado em apresentar o vasto mundo de pré-processadores para quem **nunca** teve contato com nenhum deles, seja Sass, LESS ou Stylus.

Pronto? Então vamos!

Você pode discutir sobre este livro no Fórum da Casa do Código: <http://forum.casadocodigo.com.br/>.

Caso você deseje submeter alguma errata ou sugestão, acesse <http://erratas.casadocodigo.com.br>.

Sumário

1 Preparando o ambiente	1
1.1 Instale o Ruby	2
1.2 Instalando o Sass	4
1.3 Usa Mac ou Linux?	6
1.4 Resumo	6
2 O projeto Apeperia	7
2.1 Apresentando o projeto	8
2.2 Dando uma olhada no código	13
2.3 E lá vêm as alterações	13
2.4 A primeira variável	16
2.5 Compilando automaticamente	21
2.6 Resumo	22
3 Reutilizando seu código com mixins	25
3.1 Mais um mixin	29
3.2 Evitando criar mixins loucamente	30
3.3 Isolando o image replacement	36
3.4 Resumo	37
4 Um perigoso atalho no código	38
4.1 E no mobile?	47

4.2 Mais um exemplo	48
4.3 Resumo	51
5 Organizando a bagunça	53
5.1 Cada um no seu quadrado	60
5.2 Unidos somos mais rápidos	63
5.3 Resumo	70
6 Cores de forma mais fácil	71
6.1 Claro! E escuro?	75
6.2 Comentando seu código	77
6.3 Resumo	78
7 Melhorando a performance com extends e placeholders	79
7.1 Praticando mais	85
7.2 Mixins e extends: quando usar um ou outro?	87
7.3 Resumo	89
8 Aproximando regras CSS e media queries	91
8.1 Variável na media query	100
8.2 Isolando regras inteiras	101
8.3 Resumo	104
9 Códigos prontos com Compass	105
9.1 Instalando o Compass e deixando de vigia	106
9.2 Limpando um pouco a sujeira	109
9.3 Abrindo a caixa de ferramentas	110
9.4 Terceirizando geração de sprites	114
9.5 Configurando tudo para tirarmos férias mais cedo	120
9.6 Resumo	129
10 Calculando e retornando valores	131
10.1 Fazendo o trabalho da calculadora	134

10.2 Não consigo usar o que não tenho	137
10.3 Retornando coisas	138
10.4 Arredondando a galera	143
10.5 Resumo	144
11 Conselhos finais	146
11.1 Qual o melhor pré-processador?	149
11.2 Outras features	149
11.3 Links da saideira	150

PREPARANDO O AMBIENTE

Teve uma época na minha vida em que comecei a me incomodar com algumas limitações do CSS, como muita repetição de código e manutenções um tanto quanto burocráticas. Estava na hora de mudar isso.

Antes de começar a trabalhar com **Sass** e o vasto mundo de pré-processadores CSS, eu acreditava ser imprescindível o uso de um Mac ou um Linux. E que, se fosse possível, daria muito trabalho instalar tudo no meu computador velho de guerra, assim acabei postergando meus estudos.

Para meu espanto, descobri que estava enganado com relação a usar Sass em meu Windows e que, sem trocar de sistema operacional, seria possível sim trabalhar com o *"Syntactically Awesome Stylesheets"*. Se você não sabe o que é isso, é apenas o significado da sigla Sass, algo como *"folhas de estilo sintaticamente demais"*. É o que veremos na prática neste livro.

Fazendo uma analogia, o Sass é um CSS que foi picado por uma aranha radioativa e ganhou superpoderes. Mas não tenha medo, aprenderemos como controlá-lo.

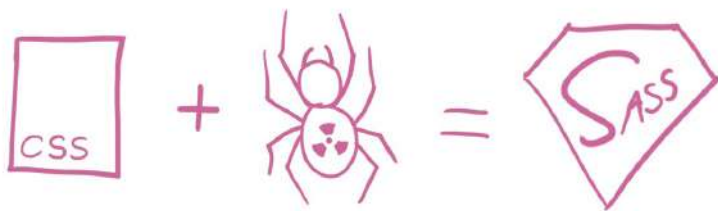


Figura 1.1: Sass é o CSS com superpoderes

Se você está lendo este livro, já rompeu uma das barreiras iniciais que muitas vezes nós mesmos colocamos em nosso próprio caminho: o medo. Medo do novo, medo de perder tempo, medo de ser difícil, medo. Logo, meus parabéns.

Não faremos um projeto do zero, mas sim uma refatoração de um código de uma empresa, feita em CSS comum. Ao término do livro, você poderá iniciar seus projetos com Sass desde o início. Para começarmos logo nossos trabalhos, precisamos inicialmente preparar todo o terreno para, aí sim, seguirmos viagem.

Vamos ver neste capítulo uma das maneiras de se instalar o Sass e configurar nosso ambiente de trabalho, para que você consiga seguir a leitura tranquilamente e para praticarmos todo esse conteúdo juntos. Se você já o instalou, pode pular este capítulo. E se usa Mac ou Linux, pule para a última parte.

1.1 INSTALE O RUBY

O Sass depende do Ruby para funcionar, então, a primeira coisa é: instalar o Ruby. Baixe-o no seguinte link, de acordo a versão do seu Windows, atentando à observação da figura a seguir: <http://rubyinstaller.org/downloads>.

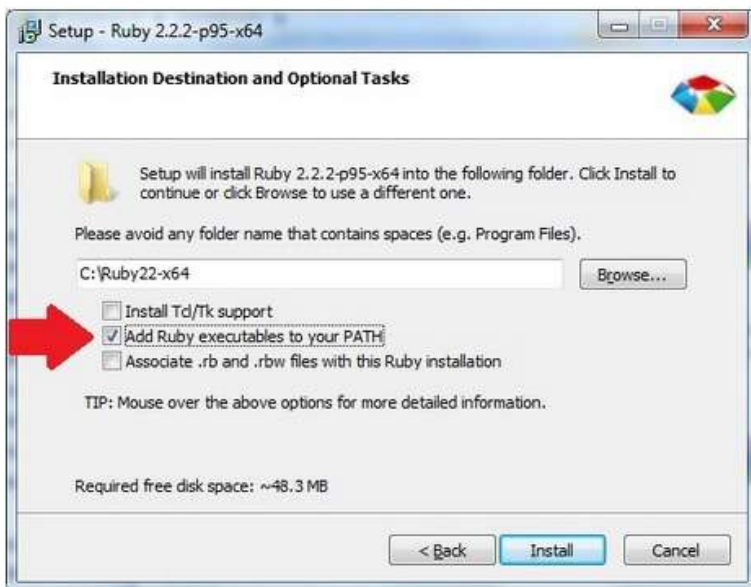


Figura 1.2: Setup do Ruby

CUIDADO: não esqueça de marcar a opção “*Add Ruby executable to your PATH*”!

32 ou 64BITS?!

Se você não sabe se o seu Windows é 32 ou 64 bits, basta dar o atalho `Win+PauseBreak`. Será mostrada uma tela com informações do seu sistema, e em *Tipo de sistema*, a versão que você tem instalada.

Conferindo se está tudo bem

Vamos conferir se está tudo bem até agora, indo ao prompt / terminal (Win+R > cmd) e dando o comando:

```
ruby -v
```

Deve aparecer a versão do Ruby como na figura a seguir. Não se preocupe se a versão aparecer diferente:

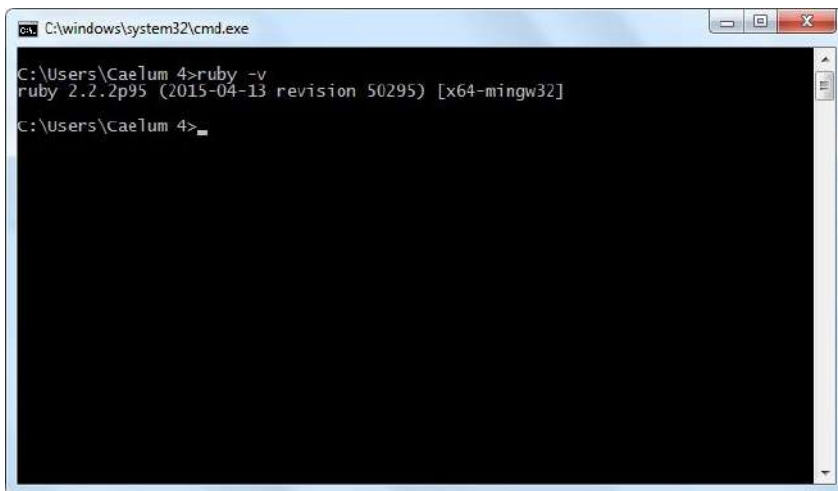
A screenshot of a Windows command prompt window. The title bar shows the path 'C:\windows\system32\cmd.exe'. The command prompt shows the user 'Caelum' at the 'C:\Users\Caelum' directory. The command 'ruby -v' has been entered, and the output is 'ruby 2.2.2p95 (2015-04-13 revision 50295) [x64-mingw32]'. The prompt is now 'C:\Users\Caelum 4>'.

Figura 1.3: Versão do Ruby

1.2 INSTALANDO O SASS

No prompt mesmo, vamos dar o comando:

```
gem install sass
```

DEU ERRO?

Confira se o Ruby está realmente instalado e repita o processo. Se ainda não funcionar, instale manualmente a *gem*, tendo esse guia como referência: <https://stackoverflow.com/questions/5778804/installing-ruby-gems-manually>.

Checando o Sass

Agora para conferir se o Sass foi instalado mesmo, no terminal vamos dar o comando:

```
sass -v
```

Deve aparecer a versão do Sass instalada:

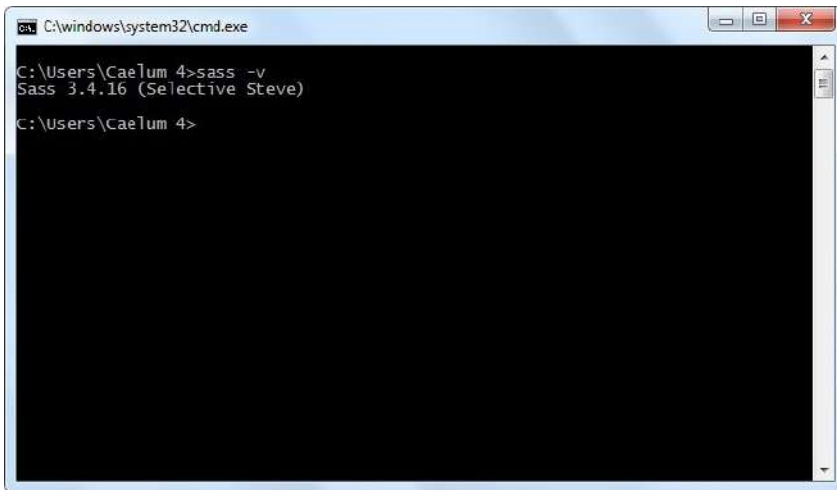
A screenshot of a Windows command prompt window. The title bar at the top reads 'C:\windows\system32\cmd.exe'. The command prompt shows the user 'Cae'lum' at the 'C:\Users\Cae'lum' directory. The command 'sass -v' has been entered, and the output is 'Sass 3.4.16 (Selective Steve)'. The prompt is now 'C:\Users\Cae'lum 4>'.

Figura 1.4: Versão do Sass

Ambiente preparado e pronto! Vamos lá!

1.3 USA MAC OU LINUX?

Se você usa Mac, o Ruby já vem instalado, só instalar o Sass do mesmo jeito do Windows: indo ao terminal e instalando a `gem` do Sass.

Se você usa Linux, o seguinte comando instala o Ruby para você:

```
\curl -sSL https://get.rvm.io | bash -s stable
```

1.4 RESUMO

Agora que temos o ambiente instalado e preparado, podemos dar continuidade aos nossos estudos. Ao longo do livro, veremos as principais características de um pré-processador, tendo o Sass como nosso companheiro nessa jornada. No próximo capítulo, analisaremos o projeto que melhoraremos e já sairemos utilizando uma *feature* muito interessante do Sass.

O PROJETO APEPERIA

Neste livro, trabalharemos na prática o Sass com o site da Apeperia, uma empresa que faz aplicativos sob demanda para pequenas e médias empresas. Ela possui três planos pagos de acordo com a necessidade do cliente.

Esta empresa já possui um site funcional com HTML e CSS, mas está com dificuldades de manutenção. Foi usado o CSS tradicional e muitas coisas foram complicadas de serem feitas e implementadas, como:

- mudanças frequentes nas cores utilizadas no site;
- refatoração de seletores;
- muito código repetido;
- dependência do Photoshop para gerar *CSS Sprites*;
- organização do CSS;
- *media queries*;
- *cross-browser*.

Utilizaremos o Sass como pré-processador para solucionar esses e outros problemas, que podem deixar o nosso cotidiano profissional bem maçante e nada interessante. Vamos dar uma olhada em funções, variáveis, *placeholders* e boas práticas.

Veremos também um framework feito em Sass, chamado Compass, que facilita nosso trabalho com *CSS Sprites*, por exemplo. Não se preocupe com esses termos agora, eles serão explicados e

mostrados na prática conforme forem surgindo as necessidades do projeto que desenvolveremos.

Refatoraremos o código CSS do Apeperia para que até o fim do livro você tenha conhecimento de cada termo e *features* importantes e presentes tanto no Sass quanto no LESS. Você já conseguirá inclusive começar a construção de um site do zero utilizando tudo isso que aprenderá aqui. Veremos como a manutenção ficará muito mais fácil daqui para a frente.

2.1 APRESENTANDO O PROJETO

Vamos dar uma olhada na estrutura do site:

- **Cabeçalho**



Figura 2.1: Header Apeperia

- **Destaque**



Figura 2.2: Destaque Apeperia

- **Sobre/institucional**



Figura 2.3: Sobre Apeperia

- Planos

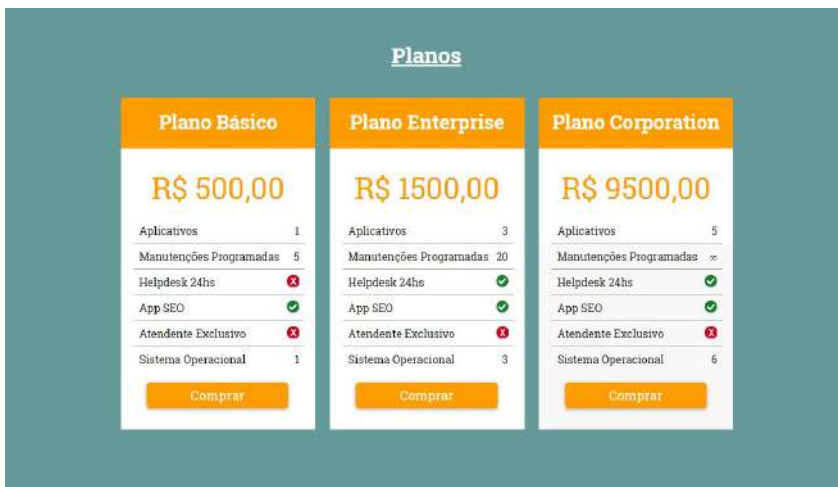


Figura 2.4: Planos Apeperia

- Blog



Figura 2.5: Blog Apeperia

- **Formulário de contato**

Contato

Digite seu nome...

Digite seu e-mail...

Digite o assunto...

Mensagem...

Enviar

Figura 2.6: Formulário de contato Apeperia

- **Rodapé**



Figura 2.7: Rodapé Apeperia

- **Site completo**



2.2 DANDO UMA OLHADA NO CÓDIGO

Vamos abrir o arquivo HTML principal dessa página, o `index.html`. Esse arquivo e o restante do projeto se encontram neste link: <https://github.com/designernatan/livro-sass>. Se possível, dê uma estudada com carinho no código, tanto HTML quanto CSS, para se situar melhor no projeto, como se você tivesse acabado de entrar na equipe de *front-end* da Apeperia.

Para resetar os estilos aplicados pelo browser, chamamos o `normalize.css`, que é um dos muitos CSS *Resets* que existem por aí. A diferença é que ele não é tão agressivo como a maioria. Chamamos também nossa folha de estilos comum, a `estilos.css`, que foi criada pela equipe de *front-end* da empresa. Há também o `media-queries.css`, que contém algumas regras sobrescritas, e finalmente uma chamada para o **Google Fonts** para usarmos a fonte Roboto.

O código basicamente é esse:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <title>Apeperia - apps sob medida</title>
  <link rel="stylesheet" href="css/normalize.css">
  <link rel="stylesheet" href="css/estilos.css">
  <link rel="stylesheet" href="css/media-queries.css">
  <link href='http://fonts.googleapis.com/css?family=Roboto+Slab
:400,300,700' rel='stylesheet' type='text/css'>
</head>
...
</html>
```

2.3 E LÁ VÊM AS ALTERAÇÕES

Vejamos primeiramente o `estilos.css`. Se abrirmos o arquivo, veremos que ele está muito bem organizado e o site funciona muito bem. Porém, o designer pede para fazermos uma alteração. Ele pede para mudarmos a cor laranja que vemos por todo o site.

Se temos de alterar a cor, ou seja, um estilo, faremos pelo CSS. Procurando no arquivo de estilos, encontramos este código:

```
/** Header */
header {
  border-top: 5px solid #fe9b00;
  background: rgba(0, 102, 153, 0.8);
  height: 90px;
  width: 100%;
  position: absolute;
}
```

Ele é o código da borda superior do cabeçalho (*header*). Será que a cor que estamos procurando é a `#fe9b00`? Para testarmos, basta escrever qualquer outra e vermos se muda ao atualizarmos a página. A nova cor pedida pelo designer é a `#c69`, então fazemos a mudança:

```
border-top: 5px solid #c69;
```

E, de fato, vemos a mudança:



Figura 2.9: Borda com cor nova

A borda superior mudou de cor. Façamos a mesma coisa para o botão na área de destaque:

```
.destaque button {
  margin-top: 1em;
  background: #c69;
  ...
}
```

E mais uma vez funcionou, sem novidades:



Figura 2.10: Nova cor no botão

E continuamos a fazer essa troca. Onde estiver a cor laranja, trocamos para a nova cor, rosa. Perceba o trabalho que tivemos para trocar uma por uma a cor dos elementos do site. Isso provavelmente nos fará cair em algum erro, como esquecer de mudar em alguma regra. E confiar em um *replace* pode ser um tiro no pé.

O que podemos fazer para solucionar este problema é setar uma cor padrão para que, se no futuro tivermos de mudá-la, basta fazermos isso em apenas um lugar:

```
/** Header **/  
header {  
  border-top: 5px solid cor-padrao;  
  ...  
}
```

O que estamos fazendo nada mais é do que setarmos uma variável `cor-padrao`. Vamos implementá-la no topo do arquivo de estilos:

```
cor-padrao: #c69;
```

Ao atualizarmos a página... Nada acontece! Isso porque a grande maioria dos browsers atualmente não dá suporte para variáveis no CSS. Daqui há uns tempos, poderemos usar uma sintaxe como o exemplo a seguir, primeiro declarando a variável:

```
.elemento {  
  --cor-padrao: #c69;  
}
```

E depois a usando de fato:

```
.elemento {  
  background: var(--cor-padrao);  
}
```

AS VARIÁVEIS ATUALMENTE

Se quiser usar variáveis hoje com CSS, recomendo dar uma lida sobre *PostCSS*.

2.4 A PRIMEIRA VARIÁVEL

Porém, como dito, o suporte atual ainda é escasso. Precisamos usar variáveis hoje, mas dando pleno suporte para os browsers atuais. Uma das soluções para esse problema é usar um pré-processador. Para criar a variável proposta utilizando o Sass, coloque o código a seguir na primeira linha do nosso `estilos.css`:

```
$cor-padrao: #c69;
```

E para usar essa variável no meio do nosso CSS, basta a chamarmos:

```
header {  
  border-top: 5px solid $cor-padrao;  
  ...  
}
```

}

Altere também todos os hexadecimais `#fe9b00` (laranja) para a `$cor-padrao` nas regras:

- `.destaque button`
- `.plano h3`
- `.plano div`
- `.plano button`
- `.contato button`

Ou seja, sempre que precisamos criar uma variável, a sintaxe é sempre `$` + o nome da variável.

Para que essas mudanças aconteçam, precisamos primeiramente transformar o arquivo CSS em um arquivo SCSS, que é uma das sintaxes que o Sass utiliza — a mais parecida com o CSS padrão e a que utilizaremos neste livro. Então, vamos alterar a extensão do arquivo para a extensão `.scss`.

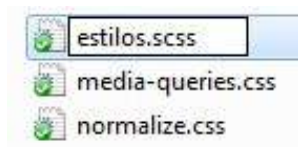


Figura 2.11: CSS para SCSS

Podemos agora mudar a chamada do `estilos.css` para nossa nova extensão no `head` do nosso `index.html`:

```
<link rel="stylesheet" href="css/estilos.scss">
```

Com isso, nossa página que deveria ficar ok, com o rosa substituindo todo o laranja utilizado no site, fica um pouco estranha:



- [sobre](#)
- [planos](#)
- [blog](#)
- [contato](#)

Aplicativos sob medida, para pequenas e médias empresas

A Apeperia é o jeito inovador de se comprar apps, você assina um plano, nós criamos os apps e fazemos toda a manutenção!

[Conheça nossos planos](#)

Sobre

O que fazemos

No Apeperia, nós criamos aplicativos personalizados de uma maneira diferente, preocupados com o modelo atual em que um aplicativo é criado entregue e a manutenção é vista como um custo extra para o cliente, decidimos criar um novo modelo de negócios, em que você assina nosso serviço, nós criamos o aplicativo e a manutenção já é inclusa no serviço, dessa maneira você não se preocupa em ter o app sempre na versão mais atualizada.

Com equipes especializadas, equipes exclusivas para projetos especiais agora é a hora da sua empresa ter o próprio aplicativo com qualidade.

Múltiplas plataformas

Na Apeperia não temos preconceito com nenhuma plataforma, fazemos apps fantásticos para qualquer device!

Não importa se você quer um app para iOS, Android, Windows Phone ou BlackBerry. Nós sabemos fazer!

- Apple
- Android
- Windows Phone
- BlackBerry

Figura 2.12: Ficou sem estilo

Infelizmente, os browsers atualmente também não entendem nenhuma linguagem de estilo diferente do CSS tradicional. Por isso, mesmo colocando para puxar o arquivo `estilos.scss`, é como se a página tivesse perdido todo o estilo! Precisamos compilar esse arquivo em um arquivo CSS normal para que o browser consiga reconhecê-lo.

Para isso, navegue via prompt de comando/terminal até a pasta CSS do projeto, e dê o comando:

```
sass estilos.scss:estilos.css
```

TEM MEDO DO TERMINAL?

Existem programas que compilam arquivos Sass através de uma interface gráfica, como o Koala (<http://koala-app.com>). Mas recomendo que você se acostume com o terminal, pois o usaremos durante todo o livro.

Após esse comando, espera-se que seja criado um arquivo

chamado `estilos.css` . Abra esse arquivo e confira a regra do `header` , percebendo que na propriedade `border-top` agora já consta a cor padrão:

```
28  /** Header **/  
29  ▾ header {  
30      border-top: 5px solid #c69;  
31      background: rgba(102, 153, 153, 0.8);  
32      height: 90px;  
33      width: 100%;  
34      position: absolute;  
35  }
```

Figura 2.13: Header depois de compilar

Não só no `header` , mas em todo lugar em que colocamos a variável `$cor-padrao` , agora está o hexadecimal do rosa (`#c69`).

Outro lugar em que podemos usar o recurso de variável, a fim de facilitar a manutenção do CSS, é na cor auxiliar (o verde utilizado no site). Vamos mudar para o azul `#069` , azul este que também é utilizado no logo do Apeperia:



Figura 2.14: Cores do logo

Já sabemos criar variáveis:

```
$cor-auxiliar: #069;
```

E a maneira de utilizá-las também. Como na classe `planos` e no pseudoelemento `footer::before`:

```
.planos {  
  background: $cor-auxiliar;  
}  
  
...  
  
footer::before {  
  content: "";  
  background: $cor-auxiliar;  
  ...  
}
```

Confira se as primeiras linhas do seu `.scss` estão com as variáveis como na figura seguinte:

A screenshot of a code editor with a dark background. It shows the first few lines of an SCSS file. Line 1: \$cor-padrao: #c69; Line 2: \$cor-auxiliar: #069; Line 3: (empty) Line 4: /** Geral **/ Line 5: body { Line 6: font-family: "Roboto Slab", serif; Line 7: } Line 8: (empty)

```
1  $cor-padrao: #c69;  
2  $cor-auxiliar: #069;  
3  
4  /** Geral **/  
5  body {  
6    font-family: "Roboto Slab", serif;  
7  }  
8
```

Figura 2.15: Variáveis no topo

Agora sempre que precisarmos mudar as cores mais utilizadas do site inteiro, basta alterarmos os valores dessas variáveis! Não se esqueça de compilar o arquivo novamente para conseguir ver as alterações em seu browser:

```
sass estilos.scss:estilos.css
```

Agora repare que, no `background` do `header`, o verde antigo também é usado, porém com uma opacidade:

```
background: rgba(102, 153, 153, 0.8);
```

Poderíamos criar outra variável chamada `$cor-auxiliar-com-opacidade-de-oitenta-por-cento`. Mas, além de não me parecer uma boa ideia e esse nome não me parecer um bom nome, precisamos criar variáveis para qualquer variação dessa cor? E se a cor auxiliar mudar para roxo? E se tivermos de alterar todas essas variações depois?

Com o Sass, não precisamos ser tão burocráticos. Basta chamarmos a variável da cor auxiliar, colocando-a no lugar do código RGB, dessa forma:

```
header {  
  ...  
  background: rgba($cor-auxiliar, 0.8);  
  ...  
}
```

Quando compilado, o CSS dessa parte fica:

```
header {  
  ...  
  background: rgba(0, 102, 153, 0.8);  
  ...  
}
```

2.5 COMPILANDO AUTOMATICAMENTE

Tenho certeza de que, se você está fazendo o projeto agora, está cansado de ficar tendo que ir ao terminal e repetir o comando de compilar sempre que faz alguma modificação no seu `.scss`. Uma coisa bacana seria se o Sass ficasse vigiando, para que, quando fizessemos alguma alteração, ele compilasse automaticamente. Pois isso é possível, e é mais simples que você imagina.

No terminal, basta colocar a opção de `watch` no comando que compila o código:

```
sass --watch estilos.scss:estilos.css
```


Após esse comando, o terminal informa que o Sass está assistindo qualquer alteração. E caso você queira parar esse `watch`, basta dar o comando `Ctrl + C`.

Faça um teste:

1. Altere o valor da cor padrão para outra qualquer, como o roxo `#52e`.
2. Salve o arquivo.
3. Vá ao terminal.
4. Veja a mensagem informando que foi detectada uma alteração, e ainda qual o arquivo que foi modificado.
5. Abra o CSS e o `index` no browser, e verifique se está tudo ok.

Com isso, nós nos livramos de ficar tendo que falar para o Sass "compila aí" sempre que alterarmos uma vírgula no nosso `.scss`.

2.6 RESUMO

Como vimos nesse contato inicial com o Sass, uma das interessantes *features* que ele possui é permitir o uso de variáveis para utilizarmos em nosso CSS. Para isso, tivemos de mudar a extensão do arquivo de `.css` para `.scss`.

Criamos e utilizamos variáveis e usamos o prompt/terminal para dar o comando para compilar de Sass para CSS. E para facilitar nossa vida, deixamos essa ação de compilar de modo automático.

Por enquanto, o site do Apeperia deve estar parecido com o da figura seguinte, mas fique livre para usar outras cores nas variáveis `$cor-padrao` e `$cor-auxiliar`.



Sobre

O que fazemos

No Apeperia, não criamos aplicativos personalizados de uma maneira diferente, preocupados com o modelo atual em que um aplicativo é criado, entregue e a manutenção é vista como um custo extra para o usuário, decidimos criar um novo modelo de negócios, em que você assina nosso serviço, nós criamos o aplicativo e a manutenção já é incluída no serviço, dessa maneira você não se preocupa em ter o app sempre na versão mais atualizada.

Com nossos especialistas, equipes exclusivas para projetos especiais, agora é a hora da sua empresa ter o próprio aplicativo com qualidade.

Múltiplas plataformas

No Apeperia não fazemos preconceito com nenhuma plataforma, fazemos apps fantásticos para qualquer device.

Não importa se você quer um app para iOS, Android, Windows Phone ou BlackBerry. Nós sabemos fazer!



Planos

Plano Básico	Plano Enterprise	Plano Corporation
R\$ 500,00	R\$ 1500,00	R\$ 9500,00
Aplicativos 1	Aplicativos 3	Aplicativos 5
Manutenções Programadas 5	Manutenções Programadas 20	Manutenções Programadas 50
Helpdesk 24hs	Helpdesk 24hs	Helpdesk 24hs
App SEO	App SEO	App SEO
Atendente Exclusivo	Atendente Exclusivo	Atendente Exclusivo
Sistema Operacional 1	Sistema Operacional 3	Sistema Operacional 5
Comprar	Comprar	Comprar

Blog



iOS ou Android?

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Porstibus solaria huius officia consetetur quare rem durimus recusandae repellat quareat dignissimos nam nunc voluptatem ex atque pariatur molestiae maiores, ea nullam.

Postagens antigas

Resoluções de iPhones

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Deserunt architecto libero officia ut molestias culpa velut consequuntur quare.

Blackberry morreu?

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Deserunt architecto libero officia ut molestias culpa velut consequuntur quare.

Nokia + Microsoft = S

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Deserunt architecto libero officia ut molestias culpa velut consequuntur quare.

Contato

[Enviar](#)



Figura 2.16: Apeperia com variáveis

No próximo capítulo, vamos isolar trechos de nosso código a fim de facilitar mais ainda sua manutenção.

REUTILIZANDO SEU CÓDIGO COM MIXINS

Uma coisa que todo desenvolvedor front-end ouve durante a vida, seja de amigos ou de colegas do trabalho, é o famoso conceito de DRY (*Don't Repeat Yourself*), algo como "*não se repita*". Ignorando que isso parece muito uma música dos Menudos, ficar repetindo código é visto com maus olhos, seja pelo pessoal de front-end ou de back-end.

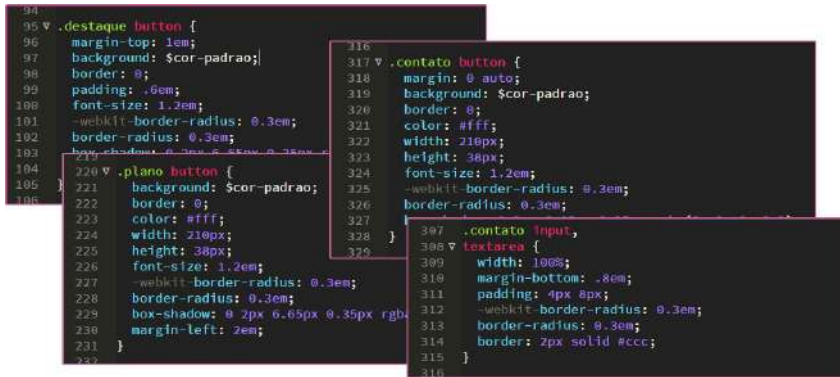
No CSS especificamente, quando possível, sempre agrupamos seletores usando a vírgula, como neste exemplo no próprio CSS do Apeperia:

```
.menu-rodape,  
.social {  
  position: absolute;  
}
```

A ideia é que, caso o valor desse `position` mude em ambas as classes, precisaríamos mudar apenas em uma única regra CSS, e não em duas separadas. Mas fazer isso com **toda** declaração deixaria nosso código difícil de manter, pois caso o valor mudasse no futuro, teríamos de achar o seletor, retirar a classe ou a tag que seria isolada, e criar outro seletor.

Como tudo na vida, acredito que precisamos ter bom senso em nosso código também. Peguemos outro exemplo agora. Repare que a propriedade `border-radius` é utilizada em quatro seletores

diferentes, com o mesmo prefixo (`webkit`) e o mesmo valor (`0.3em`):



```
94
95 ▾ .destaque button {
96   margin-top: 1em;
97   background: $cor-padrao;
98   border: 0;
99   padding: .6em;
100   font-size: 1.2em;
101   -webkit-border-radius: 0.3em;
102   border-radius: 0.3em;
103   box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.35);
104 }
105
220 ▾ .plano button {
221   background: $cor-padrao;
222   border: 0;
223   color: #fff;
224   width: 210px;
225   height: 38px;
226   font-size: 1.2em;
227   -webkit-border-radius: 0.3em;
228   border-radius: 0.3em;
229   box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.35);
230   margin-left: 2em;
231 }
232
316
317 ▾ .contato button {
318   margin: 0 auto;
319   background: $cor-padrao;
320   border: 0;
321   color: #fff;
322   width: 210px;
323   height: 38px;
324   font-size: 1.2em;
325   -webkit-border-radius: 0.3em;
326   border-radius: 0.3em;
327 }
328
397 .contato input,
398 ▾ textarea {
399   width: 100%;
400   margin-bottom: .8em;
401   padding: 4px 8px;
402   -webkit-border-radius: 0.3em;
403   border-radius: 0.3em;
404   border: 2px solid #ccc;
405 }
406
```

Figura 3.1: Border-radius pelo código

Caso o **valor** desse `border-radius` mude, precisaríamos fazer essa alteração em diversos lugares — o mesmo problema da cor padrão do nosso site. Como nos prevenimos nesse problema que pode vir a acontecer? Se você pensou agora em criar uma variável para solucionar isso, você acertou!

Vamos criar uma variável chamada `$raio` para esse valor do `border-radius`, junto com as nossas outras variáveis, no topo do `.scss`:

```
$cor-padrao: #c69;
$cor-auxiliar: #069;
$raio: 0.3em;
```

Agora sempre onde tiver esse valor de `0.3em`, chamaremos a variável `$raio`, como nas regras da figura anterior. Pegarei por ora a regra do `.destaque button` como exemplo:

```
.destaque button {
  ...
  -webkit-border-radius: $raio;
  border-radius: $raio;
  ...
}
```

Dê uma conferida se o Sass já detectou a mudança e já compilou o CSS como deveria. Se não detectou, confira se você está utilizando o `watch` de forma correta:

```
sass --watch estilos.scss:estilos.css
```

E se precisássemos utilizar outro prefixo sempre que usarmos o `border-radius` ? O prefixo `-moz-` , por exemplo. Teríamos de fazer essa alteração em quatro regras diferentes, onde está o *DRY*? Sempre que repetimos muito um valor, acabamos solucionando isso com uma variável, e quando repetimos muito um trecho de código, podemos utilizar uma funcionalidade presente em todos os pré-processadores mais conhecidos, chamada **mixin**.

Para criar esse mixin, para que o código da borda arredondada seja isolado, basta colocarmos o seguinte código, no topo do nosso arquivo `.scss` :

```
@mixin borda-arredondada {  
  -webkit-border-radius: $raio;  
  border-radius: $raio;  
}
```

Ou seja, sempre que precisarmos criar um mixin, utilizamos o `@mixin` *mais* um nome que faça sentido com o que ele representa. No nosso caso, ficou `borda-arredondada` .

Felizmente (ou infelizmente), o mixin não é mágico. Precisamos também chamá-lo onde ele é necessário. Sempre que precisarmos chamar/incluir um mixin em uma regra CSS, basta usar a sintaxe `@include` *mais* o nome do mixin. Veja, por exemplo, a regra `.destaque button` .

```
.destaque button {  
  ...  
  -webkit-border-radius: $raio;  
  border-radius: $raio;  
  @include borda-arredondada;  
  ...  
}
```

O código compilado ficará dessa forma:

```
.destaque button {  
  ...  
  -webkit-border-radius: 0.3em;  
  border-radius: 0.3em;  
  -webkit-border-radius: 0.3em;  
  border-radius: 0.3em;  
  ...  
}
```

Ops! Como já estamos colocando o `border-radius` através do `mixin` `borda-arredondada`, não precisamos mais do seu código original. Então o retiramos:

```
.destaque button {  
  ...  
  /* excluir essa linha */ -webkit-border-radius: $raio;  
  /* excluir essa linha */ border-radius: $raio;  
  @include borda-arredondada;  
  ...  
}
```

E deixamos somente a inclusão do `mixin`:

```
.destaque button {  
  ...  
  @include borda-arredondada;  
  ...  
}
```

Compilando ficará desta maneira:

```
.destaque button {  
  ...  
  -webkit-border-radius: 0.3em;  
  border-radius: 0.3em;  
  ...  
}
```

Um `mixin` nada mais é do que um jeito mais inteligente e prático de reutilizar nosso código. Com ele, podemos fazer alterações em um único lugar.

Agora só resta incluir nosso recém-criado *mixin* onde ele é

necessário no restante do código:

- `.plano button`
- `.contato input, textarea`
- `.contato button`

Não se esqueça de ir verificando o CSS compilado e dando uma olhada no `index.html` para ver como está ficando o resultado final.

3.1 MAIS UM MIXIN

Repare que todos os botões da página possuem uma sombra. Vamos criar outro mixin para isolar essa declaração, a fim de melhorar nosso código.



Figura 3.2: Botões com sombra

Primeiramente, criamos o mixin para essa sombra padrão logo abaixo do mixin `borda-arredondada` :

```
@mixin sombra-padrao {  
    box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);  
}
```

Agora, onde tiver o `box-shadow` no código, precisamos substituí-lo pela inclusão do mixin `sombra-padrao` , como no botão de destaque no exemplo:


```
.destaque button {
  margin-top: 1em;
  background: $cor-padrao;
  border: 0;
  padding: .6em;
  font-size: 1.2em;
  @include borda-arredondada;
  @include sombra-padrao;
  font-weight: bold;
}
```

Só resta incluir também nos outros botões:

- .plano button
- .contato button

Tudo ok! Agora façamos um teste colocando o prefixo do webkit , adicionando mais uma regra nesse mixin:

```
@mixin sombra-padrao {
  -webkit-box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);
  box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);
}
```

Com essas alterações feitas, se for necessário mudar a cor, a opacidade ou mesmo adicionar outro prefixo na regra CSS dessa sombra no futuro, podemos simplesmente alterá-la através desse mixin. Faça mais testes colocando valores absurdamente altos no sombra-padrao , como por exemplo:

```
@mixin sombra-padrao {
  -webkit-box-shadow: 50px 20px 60.65px 10.35px rgba(255, 0, 0, 0.3);
  box-shadow: 50px 20px 60.65px 10.35px rgba(255, 0, 0, 0.3);
}
```

Só não esqueça de desfazer os testes, se não o designer briga conosco!

3.2 EVITANDO CRIAR MIXINS LOUCAMENTE

Falando no designer, ele pede para que alteremos o quão redonda é a borda do botão de destaque, **somente** o do destaque. O novo valor é **20px**.

Como é que estamos colocando essa borda arredondada no botão? Com o mixin `borda-arredondada` ! E estamos usando a variável `$raio` para montá-lo também. Então só precisamos mudar o valor dessa variável para 20px:

```
$raio: 20px;
```

Conferindo no browser, deveria ficar dessa forma:



Figura 3.3: Botão destaque com 20px de border-radius

Maravilha, isso foi fácil, não? Mas descendo um pouco a tela, repare nos outros botões e nos campos:

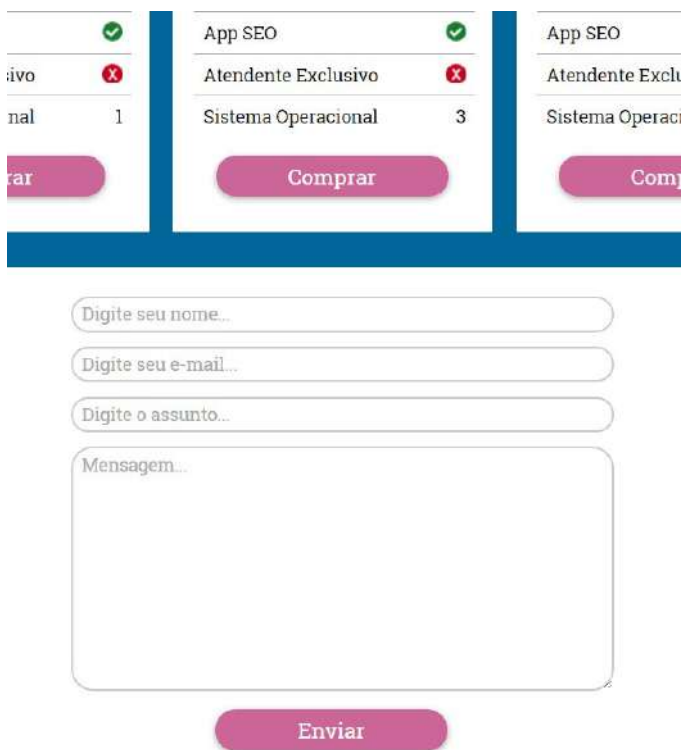


Figura 3.4: Arredondou tudo

Agora todos ficaram com 20px de `border-radius`. O que foi pedido é que apenas o botão de destaque tivesse esse valor, os outros elementos devem continuar com 0.3em.

Há algumas maneiras para resolver isso. Primeiro, volte o valor da variável `$raio` para seu valor original, 0.3em.

Uma das maneiras de solucionar isso é criar outro mixin para a borda mais arredondada, e outra variável, com o valor de 20px:

```
$raio-grande: 20px;

@mixin borda-mais-arredondada {
  -webkit-border-radius: $raio-grande;
  border-radius: $raio-grande;
```

```
}
```

Agora incluímos esse mixin no botão de destaque, substituindo o outro mixin:

```
.destaque button {  
  ...  
  @include borda-mais-arredondada;  
}
```

Agora sim, cada um com seu próprio valor, sem bagunça. Mas lembra do *DRY*? Não estamos repedindo código aqui?

E se pedirem uma borda menos arredondada? Criar outro mixin e outra variável, algo como `borda-mais-ou-menos-arredondada` ? E se precisarmos modificar as declarações? Teremos de alterar em vários lugares?

Outra forma de resolvermos nosso problema sem criar vários mixins é que, quando incluirmos o mixin `borda-arredondada` , passemos também um valor para ele, um **parâmetro**. Para isso, apenas colocamos o valor entre parênteses, junto com a inclusão do mixin, dessa forma:

```
.destaque button {  
  ...  
  @include borda-mais-arredondada(20px);  
}
```

Com isso, o seguinte erro aparece no terminal:

```
error estilos.scss (Line 118: Mixin borda-arredondada takes 0 arguments but 1 was passed.)
```

Nós esquecemos de preparar o mixin `borda-arredondada` para aceitar o parâmetro/argumento que estamos passando na sua inclusão no botão de destaque.

Para consertar isso, basta colocarmos a variável `$raio` entre parênteses, na criação do mixin. E podemos excluir a variável `$raio-grande` e o mixin `borda-mais-arredondada` .

```
@mixin borda-arredondada($raio) {
  -webkit-border-radius: $raio;
  border-radius: $raio;
}
```

E mais uma vez... Outra mensagem de erro:

```
error estilos.scss (Line 237: Mixin borda-arredondada is missing a
rgument $raio.)
```

Aí diz que o mixin `borda-arredondada`, na linha 237 especificamente, está com o argumento `$raio` faltante. Aconteceu que, quando colocamos a variável `$raio` como argumento no mixin, somos obrigados a passar um valor sempre que o chamarmos. O que fazer então? Podemos passar esses argumentos (o valor `0.3em`) em **todas** as chamadas desse mixin:

```
.plano button {
  ...
  @include borda-arredondada(0.3em);
  ...
}

...

.contato input,
textarea {
  ...
  @include borda-arredondada(0.3em);
  ...
}

.contato button {
  ...
  @include borda-arredondada(0.3em);
  ...
}
```

Ainda bem que são poucas declarações que tivemos de alterar, mas isso não parece muito sustentável. Se precisarmos mudar esse valor no futuro, teremos de alterar em vários lugares. Não quero ter de ficar passando um valor sempre que chamar o mixin. Seria interessante se pudéssemos passar um valor padrão para ele. E

podemos, desta forma:

```
@mixin borda-arredondada($raio: 0.3em) {  
  -webkit-border-radius: $raio;  
  border-radius: $raio;  
}
```

Colocando esse valor padrão, podemos remover a variável `$raio` do começo do nosso arquivo e o argumento de `0.3em` nas inclusões dos mixins pelo nosso código, voltando-os como eram antes:

```
.plano button {  
  ...  
  @include borda-arredondada;  
  ...  
}  
  
...  
  
.contato input,  
textarea {  
  ...  
  @include borda-arredondada;  
  ...  
}  
  
.contato button {  
  ...  
  @include borda-arredondada;  
  ...  
}
```

Se precisarmos de um valor padrão, conseguimos colocá-lo direto na criação do mixin. E se esse valor padrão do argumento já bastar, não precisamos especificar nenhum valor na chamada do mixin.

ENXERGANDO MELHOR

Se as cores da sua IDE estiverem estranhas usando Sass, procure algum plugin/extensão disponível que deixe a sintaxe do Sass destacada.

No Sublime Text, especificamente, existe um *package* chamado *Syntax Highlighting for Sass*.

3.3 ISOLANDO O IMAGE REPLACEMENT

Há diversas técnicas para fazer o chamado *image replacement*. Isto é, quando você precisa esconder o texto de um elemento no HTML, mas ainda precisa da imagem de fundo. Uma dessas técnicas existentes para conseguirmos isso está sendo usada em nosso projeto no `estilos.scss` :

```
text-indent: -9999px;  
overflow: hidden;  
background-repeat: no-repeat;
```

Sendo mais específico, estamos repetindo esse código nestas duas regras:

- `.plataformas li`
- `.social li a`

Caso resolvamos alterar a técnica utilizada, ou adicionar outras declarações para complementá-la, teríamos de fazer isso em dois lugares diferentes. Quando repetimos muito um valor, resolvemos isso criando uma variável. Agora já sabemos o que fazer para isolar um trecho de código, criando um *mixin*.

Vamos isolar esse código em um *mixin* chamado `image-`

replacement :

```
@mixin image-replacement {  
  text-indent: -9999px;  
  overflow: hidden;  
  background-repeat: no-repeat;  
}
```

Agora basta incluirmos o nosso novo mixin nas regras mencionadas anteriormente:

```
.plataformas li {  
  ...  
  @include image-replacement;  
  ...  
}  
  
...  
  
.social li a {  
  ...  
  @include image-replacement;  
  ...  
}
```

3.4 RESUMO

Vimos neste capítulo que os mixins são uma "mão na roda" para não cometermos *DRY*, podendo ser reutilizados livremente. Podemos inclusive passar argumentos com valores padrões na criação dos mixins, permitindo-nos uma maior flexibilidade quando se trata da manutenção do nosso código.

Fica o convite ao leitor de começar a acumular e organizar seus próprios mixins, a fim de reutilizá-los não só nesse projeto, como também em projetos pessoais e profissionais.

A seguir, veremos como facilitar mais ainda nossa vida no CSS, usando um recurso que usamos no HTML sem dar muita importância: o aninhamento.

UM PERIGOSO ATALHO NO CÓDIGO

Selecioneamos muitos elementos HTML pelo CSS quando estamos fazendo um site. Por exemplo, quando precisamos estilizar diversos elementos com um único seletor, podemos fazer de diversas formas, como usando classes e pseudoclasses. Ou se desejarmos estilizar elementos únicos, é possível usar os IDs para isso, algo como `#menu` ou `#form-cursos`.

Uma coisa bem comum de acontecer é, dado aquele momento de correria e pressa ao escolher o nome de uma classe, para que consigamos selecionar aquele determinado elemento, digitamos a primeira coisa que nos vem à cabeça. Aí saem umas classes assim:

```
.underlineNegrito {  
  text-decoration: none;  
  font-weight: 900;  
}
```

Ou assim:

```
.vaiProlado {  
  float: left;  
  clear: both;  
}
```

Legal, isso funcionaria, restando-nos apenas chamar essas classes no HTML.

Mas vão se passando alguns dias (ou horas), e não lembramos

mais com tanta clareza do que se tratava aquela classe `vaiProLado` só batendo o olho em seu nome. Ou pior ainda, quando algum outro desenvolvedor vai mexer no nosso código, e fica com uma linda cara de interrogação ao tentar entender do que se tratam aquelas classes.

Uma coisa que recomendo aos meus alunos dos cursos presenciais de front-end na Caelum é pensar com um certo carinho em bons nomes de classe. Dando uma olhada no nosso arquivo `estilos.scss`, perceba que o único menu que o site possui é chamado pela classe `menu-opcoes`.

Como não existem outros menus, poderíamos deixar o nome dessa classe mais entendível mudando-a para `menu-principal`, por exemplo, para deixar claro que é o principal menu de navegação da página:

Agora:

```
.menu-opcoes {  
  position: absolute;  
  right: 0;  
  top: -.5em;  
  font-size: 1.2em;  
  font-weight: lighter;  
}  
  
.menu-opcoes ul {  
  padding-left: 0;  
}  
  
.menu-opcoes li {  
  display: inline-block;  
  width: 5em;  
}  
  
.menu-opcoes a {  
  color: white;  
  text-decoration: none;  
}  
  
.menu-opcoes a:hover {
```

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

Depois:

```
.menu-principal {
  position: absolute;
  right: 0;
  top: -.5em;
  font-size: 1.2em;
  font-weight: lighter;
}

.menu-principal ul {
  padding-left: 0;
}

.menu-principal li {
  display: inline-block;
  width: 5em;
}

.menu-principal a {
  color: white;
  text-decoration: none;
}

.menu-principal a:hover {
  text-decoration: underline;
}
```

Tranquilo, é apenas uma pequena melhoria de nomeação de classes, coisa que fazemos normalmente. Ao menos, deveríamos, correto?

Com isso, chegamos a esse resultado:



Figura 4.1: Menu sem classe

O HTML não é mágico (ainda) de replicar automaticamente as alterações que fazemos em nosso CSS. Logo, precisamos mudar a antiga classe `menu-opcoes` no arquivo `index.html` para a nova classe `menu-principal`:

```
<nav class="menu-principal">
  <ul>
    <li><a href="#">sobre</a></li>
    <li><a href="#">planos</a></li>
    <li><a href="#">blog</a></li>
    <li><a href="#">contato</a></li>
  </ul>
</nav>
```

Agora sim está funcionando como deveria:



Figura 4.2: Menu com classe

Tivemos de mudar o nome da classe em quatro lugares diferentes. Sorte a nossa que o `estilos.css` não é muito extenso e possui cerca de 450 linhas. Para uma comparação, um dos CSSs do Facebook possui quase 9.000 linhas! Claro que isso no código a que o desenvolvedor tem acesso (sem compressão), ou seja, o que vem

para nós já está otimizado e minificado.

O ponto é: seria bacana se pudéssemos mudar o nome de um seletor, e todos os seletores que têm alguma relação direta com ele mudassem junto também.

No HTML, como sabemos que um elemento **A** é filho de um elemento **B**? Simplesmente vemos se **A** está dentro de **B**! Um elemento está aninhado em outro. Com Sass, podemos fazer isso também, **aninhar seletores**, usando seu recurso de aninhamento (*nesting*). Vamos refatorar nosso código!

As regras do menu principal, por enquanto, estão assim:

```
.menu-principal {
  position: absolute;
  right: 0;
  top: -.5em;
  font-size: 1.2em;
  font-weight: lighter;
}

.menu-principal ul {
  padding-left: 0;
}

.menu-principal li {
  display: inline-block;
  width: 5em;
}

.menu-principal a {
  color: white;
  text-decoration: none;
}

.menu-principal a:hover {
  text-decoration: underline;
}
```

Essa `ul` não é filha da classe `menu-principal` ? Pois vamos aninhá-la:

```
.menu-principal {
```

```

position: absolute;
right: 0;
top: -.5em;
font-size: 1.2em;
font-weight: lighter;

ul {
  padding-left: 0;
}
}

```

E é só isso! Com esse recurso de aninhamento, podemos ganhar mais flexibilidade ainda no nosso código. Continuemos com a mesma lógica no restante do código CSS do `menu-principal`, aninhando outros seletores dentro dele:

```

.menu-principal {
  position: absolute;
  right: 0;
  top: -.5em;
  font-size: 1.2em;
  font-weight: lighter;

  ul {
    padding-left: 0;
  }

  li {
    display: inline-block;
    width: 5em;
  }

  a {
    color: white;
    text-decoration: none;
  }

  a:hover {
    text-decoration: underline;
  }
}

```

Compilando o código, ficará igual o mostrado anteriormente:

```

.menu-principal {

```

```

    position: absolute;
    right: 0;
    top: -.5em;
    font-size: 1.2em;
    font-weight: lighter;
}

.menu-principal ul {
    padding-left: 0;
}

.menu-principal li {
    display: inline-block;
    width: 5em;
}

.menu-principal a {
    color: white;
    text-decoration: none;
}

.menu-principal a:hover {
    text-decoration: underline;
}

```

Agora se essa classe `menu-principal` mudar para `menu-juquinha` ou `menu-header`, só precisamos alterar em apenas um lugar, e o Sass vai se encarregar de interpretar as regras que estão aninhadas e compilar o CSS normalmente.

Outra alteração bacana que podemos fazer é nesse `a`. Se amanhã eu precisar alterá-lo para um outro elemento, como uma `div` por exemplo, terei de alterar isso em dois lugares:

```

.menu-principal {
    ...

    ...

    a {
        color: white;
        text-decoration: none;
    }

    a:hover {
        text-decoration: underline;
    }
}

```

```
}  
}
```

Podemos notar que esse `a:hover` está atrelado à regra anterior. Se o `a` mudar para `div` ou `article`, é natural quisermos manter o `:hover` também.

Pensando nisso, podemos aninhá-los desta forma:

```
.menu-principal {  
  ...  
  ...  
  
  a {  
    color: white;  
    text-decoration: none;  
  
    a:hover {  
      text-decoration: underline;  
    }  
  }  
}
```

Compilando, encontramos um pequeno *bug*:

```
.menu-principal a {  
  color: white;  
  text-decoration: none;  
}  
  
.menu-principal a a:hover {  
  text-decoration: underline;  
}
```

Terá o Sass endoidado?! Não, não! Ele está certo, nós que endoidamos e erramos. Ele fez o que pedimos e ponto.

Pedimos que o `a:hover` fosse compilado sendo filho do `a`, e foi exatamente isso que ele fez. O `a:hover` não é para ser filho do `a`, queremos *linkar* o seletor de tag em si. O `a:hover` precisa ser exatamente igual ao pai dele. Para isso, basta usarmos o `&`, da

seguinte forma:

```
.menu-principal {  
  ...  
  ...  
  
  a {  
    color: white;  
    text-decoration: none;  
  
    &:hover {  
      text-decoration: underline;  
    }  
  }  
}
```

Compilando, temos este resultado, desta vez do jeito que queríamos:

```
.menu-principal a {  
  color: white;  
  text-decoration: none;  
}  
  
.menu-principal a:hover {  
  text-decoration: underline;  
}
```

Flores para todos os lados, *nesting* é uma maravilha e vamos usar loucamente sem dó nem piedade! **Não!** Aninhamento tem de ser usado com uma boa dose de cautela, pois o CSS gerado pode vir a se tornar algo assim:

```
.menu-principal div p span .icone-relogio {  
  background: #210;  
}
```

Por boa prática, seletores de descendentes assim deixam nosso CSS nada performático por conta do processo de leitura do browser (*render*), sempre indo da direita para a esquerda. Logo, usar um seletor desse tipo é o mesmo que dizer: *"não me importo com*

performance, deixa assim mesmo que está funcionando".

Se você liga para performance ou já leu um pouco a respeito disso, sabe que devemos evitar de ir muito a fundo na hierarquia desse tipo de seletor. Outro ponto é que, usando aninhamento, seu Sass fica totalmente amarrado com a sua marcação. Mudou o HTML, vai ter de mudar o Sass também, e vice-versa.

Justamente por isso o *nesting* pode vir a ser um perigoso problema, principalmente quando perdemos o controle e não checamos o CSS gerado. Em suma: vá com calma.

4.1 E NO MOBILE?

Se você testou no browser viu que está tudo ok. Mas e no mobile? Redimensione a janela de seu browser e perceba que o menu está com um pequeno defeito:



Figura 4.3: Bug no menu em telas pequena

Tratamos design responsivo com *media queries*, então vamos verificar o arquivo `media-queries.css` para encontrar e resolver nosso pequeno bug no menu:

```
@media (max-width: 980px) {  
  ...  
  
  .menu-opcoes {  
    position: static;  
    display: block;  
    text-align: center;  
  }  
  
  .menu-opcoes li {
```

```

        margin: .4em;
    }

    .destaque {
        font-size: 14px;
        height: 250px;
    }

    .menu-opcoes,
    .menu-opcoes li,
    .post-destaque,
    .posts-antigos,
    form {
        width: auto;
    }
}

```

Pegamos! Precisamos alterar a classe `.menu-opcoes` para `.menu-principal`, pois se não esses estilos nunca serão aplicados. Fazendo essa alteração, nosso header fica assim:



Figura 4.4: Menu consertado no mobile

Erro consertado! Agora a experiência do usuário no celular não será mais prejudicada por um erro tão banal e simples de ser resolvido.

4.2 MAIS UM EXEMPLO

A classe `itens-plano` é outro lugar em que podemos treinar o aninhamento. No código atual:

```

.itens-plano {
    padding-left: 1.2em;
}

```

```

}

.itens-plano li {
  font-size: 1em;
  list-style: none;
  border-bottom: 1px solid #acacac;
  padding: .5em;
  width: 14.8em;
}

.itens-plano li:last-child {
  border: 0;
}

.itens-plano span {
  float: right;
}

```

O `li` está dentro do `itens-plano`, podemos aninhá-los dessa forma:

```

.itens-plano {
  padding-left: 1.2em;

  li {
    font-size: 1em;
    list-style: none;
    border-bottom: 1px solid #acacac;
    padding: .5em;
    width: 14.8em;
  }
}

```

Continuando a refatoração, a próxima regra é para selecionar o último `li` com a pseudoclasse `:last-child`. Podemos atrelá-los usando o `&`:

```

.itens-plano {
  padding-left: 1.2em;

  li {
    font-size: 1em;
    list-style: none;
    border-bottom: 1px solid #acacac;
    padding: .5em;
    width: 14.8em;

```

```

    &:last-child {
        border: 0;
    }
}

```

Por último, o `span` que faltava:

```

.itens-plano {
    padding-left: 1.2em;

    li {
        font-size: 1em;
        list-style: none;
        border-bottom: 1px solid #acacac;
        padding: .5em;
        width: 14.8em;

        &:last-child {
            border: 0;
        }
    }

    span {
        float: right;
    }
}

```

Pronto! Novamente o código CSS ficará igual o mostrado antes de usarmos o aninhamento:

```

.itens-plano {
    padding-left: 1.2em;
}

.itens-plano li {
    font-size: 1em;
    list-style: none;
    border-bottom: 1px solid #acacac;
    padding: .5em;
    width: 14.8em;
}

.itens-plano li:last-child {
    border: 0;
}

.itens-plano span {
    float: right;
}

```

}

E o site do Apeperia continua lindo. Tanto no menu principal:



Figura 4.5: "Menu principal"

Quanto na lista de serviços dos planos:

Plano Básico	Plano Enterprise	Plano Corporation
R\$ 500,00	R\$ 1500,00	R\$ 9500,00
Aplicativos 1	Aplicativos 3	Aplicativos 5
Manutenções Programadas 5	Manutenções Programadas 20	Manutenções Programadas ∞
Helpdesk 24hs	Helpdesk 24hs	Helpdesk 24hs
App SEO	App SEO	App SEO
Atendente Exclusivo	Atendente Exclusivo	Atendente Exclusivo
Sistema Operacional 1	Sistema Operacional 3	Sistema Operacional 6
Comprar	Comprar	Comprar

Figura 4.6: "Lista de itens plano"

4.3 RESUMO

Vimos como o recurso de aninhamento pode ajudar na manutenção do código. Mas como um tio um dia disse: *"Com grandes poderes, vêm grandes responsabilidades"*. Cuidado para não cair em um *nesting hell* e fazer seletores descendentes com muitos níveis.

Consertamos um bug no menu quando a janela do browser era redimensionada, simplesmente arrumando a classe que era usada nas regras dentro do *media query*. Outra coisa que vimos também é

como o uso do `&` pode nos ajudar para atrelar a regra de um `:hover` ao elemento original.

Boas práticas no CSS, boas práticas no Sass. Um Sass mal feito gera um CSS mal feito. Agora vamos dar uma organizada nesse CSS "compridão" e dividir cada bloco do site.

ORGANIZANDO A BAGUNÇA

No HD do meu computador pessoal (Jotalhão II) e em meu Dropbox, preocupo-me bastante com a organização de todos meus arquivos. Quem nunca passou por aquela situação de ter certeza que possui um arquivo, mas não conseguir encontrar de jeito nenhum?

Por exemplo, você fez um *freelance* para uma empresa há sete meses, e agora eles querem outro contrato para você fazer atualizações no site. Será que você sabe onde os arquivos fontes estão? E como esses arquivos estão separados?

Em nossos projetos, para não perdermos tempo no futuro, tentamos organizar nosso código da melhor maneira possível. Mas será que conseguimos cumprir esse objetivo?

Voltando para o Apeperia. Se precisássemos alterar a largura da imagem na seção de **blog**, teríamos de procurar em nosso arquivo `estilos.scss` a regra correspondente.



iOS ou Android?

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Possimus soluta fugiat officia consectetur quas rem ducimus recusandae repellat quaerat dignissimos nam iure, voluptatem ex atque pariatur molestiae maiores, ea nulla!

Figura 5.1: Imagem do post destaque

Para isso, procuramos a regra responsável por isso. Estamos dizendo que a largura da imagem será 100% a largura de seu container , no caso, algum elemento com a classe `.post-destaque` :

```
.post-destaque img {  
  width: 100%;  
}
```

A sorte é que o desenvolvedor que fez esse CSS teve o bom senso de deixar o código comentado e com uma divisão lógica de cada seção da página. Se fosse aquele CSS do Facebook (com 9 mil linhas), procurar uma regra de uma seção específica seria um belo tormento para nós.

No contexto do nosso projeto Apeperia, se precisássemos fazer uma alteração dessas também, como faríamos? Precisaríamos procurar no olho, ou dando o comando de localizar (normalmente Ctrl + F) .

Outra situação: e se precisássemos copiar toda a seção de **contato** para uma *landing page* do Apeperia? Faríamos algo assim:

1. Abrir o `estilos.scss` ;
2. Procurar todas as regras da parte contato;
3. Copiá-las;
4. Colá-las no CSS do outro projeto.

Novamente dependeríamos da prévia organização do CSS para que não perdêssemos horas e horas nessa tarefa, além de termos de fazer tudo manualmente. Algo que podemos fazer para ajudar nesse problema é separar o CSS de cada parte do site em um arquivo à parte.

Assim, quando precisarmos alterar algo do *header*, bastaria irmos a um arquivo chamado **header**. Vamos começar!

Primeiramente, isolaremos as variáveis que colocamos no `estilos.scss` em um novo arquivo chamado **variaveis.scss** . E o mesmo com os *mixins*, criaremos um arquivo chamado **mixins.scss** .

Atente-se em **recortar** os códigos e colar em seus respectivos arquivos. Por enquanto, a pasta do projeto estará como a da figura:

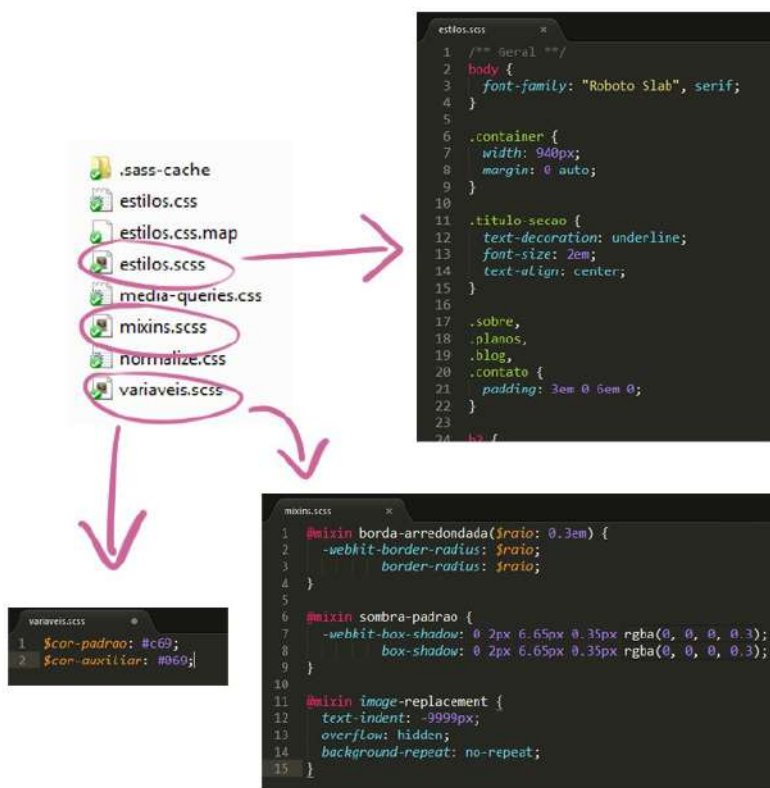


Figura 5.2: Novos arquivos variaveis.scss e mixins.scss

Na mesma ideia, façamos o mesmo com o restante do estilos.scss, criando os seguintes arquivos:

- geral.scss
- header.scss
- destaque.scss
- sobre.scss
- planos.scss
- blog.scss
- contato.scss
- footer.scss

Feito isso, nosso `estilos.scss` ficará vazio, sem nenhuma linha de código. Se testarmos no browser, ele deverá ficar assim:



Figura 5.3: Layout sumiu

Por qual razão isso aconteceu?! Oras, se estamos chamando em nosso HTML o arquivo `estilos.scss` e esse mesmo arquivo agora não possui nenhuma linha de código, o que está faltando?

Precisamos que o arquivo `estilos.scss` seja o responsável por "reunir a galera", por chamar todos os arquivos que criamos, por **importar** esses arquivos! No `estilos.scss`, importamos cada um dos arquivos que criamos utilizando o `@import` mais o nome do arquivo:

```
@import "geral.scss";
@import "header.scss";
@import "destaque.scss";
@import "sobre.scss";
@import "planos.scss";
@import "blog.scss";
@import "contato.scss";
@import "footer.scss";

@import "variaveis.scss";
@import "mixins.scss";
```

Seguindo exatamente essa ordem dos *imports*, dará o seguinte erro em nosso terminal, dizendo que a variável `$cor-padrao` não foi definida:

```
error header.scss (Line 3: Undefined variable: "$cor-padrao".)
```

Mas nem mexemos na variável `$cor-padrao` ! Mas será que ela já foi criada antes de precisarmos dela? Isso aconteceu justamente por isso; estamos tentando usar uma variável que **ainda** não foi criada.

Ela só é criada na penúltima linha do nosso arquivo, quando importamos o `variaveis.scss` . Para consertar isso, basta o chamarmos antes dos demais:

```
@import "variaveis.scss";

@import "geral.scss";
@import "header.scss";
@import "destaque.scsss";
@import "sobre.scss";
@import "planos.scss";
@import "blog.scss";
@import "contato.scss";
@import "footer.scss";
@import "mixins.scss";
```

O erro foi consertado, vitória! Agora, apareceu outro...

```
error destaque.scss (Line 32: Undefined mixin 'borda-arredondada'.
)
```

Mesmo problema com o *mixin*, estamos tentando usá-lo e ele ainda não foi criado! Para consertar isso, nós o importamos **antes** dos demais também, desta forma:

```
@import "mixins.scss";
@import "variaveis.scss";

@import "geral.scss";
@import "header.scss";
@import "destaque.scsss";
@import "sobre.scss";
```

```
@import "planos.scss";
@import "blog.scss";
@import "contato.scss";
@import "footer.scss";
```

Por sorte nenhum *mixin* que criamos depende de nenhuma variável externa, do arquivo `variaveis.scss` especificamente. Se no futuro fizermos isso, dará o mesmo problema de tentar usar algo que ainda não foi criado. Para evitar isso, basta invertermos a ordem dos imports :

```
@import "variaveis.scss";
@import "mixins.scss";

@import "geral.scss";
@import "header.scss";
@import "destaque.scsss";
@import "sobre.scss";
@import "planos.scss";
@import "blog.scss";
@import "contato.scss";
@import "footer.scss";
```

Por conta de todos esses problemas, as variáveis e os mixins devem ser preferencialmente os primeiros arquivos que importamos em nosso código. Entretanto, com o código anterior, ainda dará um erro:

```
error estilos.scss (Line 6: File to import not found or unreadable
: destaque.scsss.)
```

Ahá! Colocamos o nome da extensão do arquivo `destaque.scss` de forma errada: tem um "s" a mais ali! Uma forma para resolver isso é não passar a extensão dos arquivos `.scss` , simplesmente não a colocando. Façamos isso então:

```
@import "variaveis";
@import "mixins";

@import "geral";
@import "header";
@import "destaque";
@import "sobre";
```

```
@import "planos";  
@import "blog";  
@import "contato";  
@import "footer";
```

Pronto. Então, sempre que precisarmos importar um arquivo `.scss`, basta colocar o caminho dele, e nem precisa especificar a extensão. O Sass considera que todo `@import` que fazemos é um arquivo `.scss`.

5.1 CADA UM NO SEU QUADRADO

Modularizar nosso CSS é o *start* para organizarmos essa bagunça, mas agora temos 500 arquivos jogados na pasta CSS do projeto:

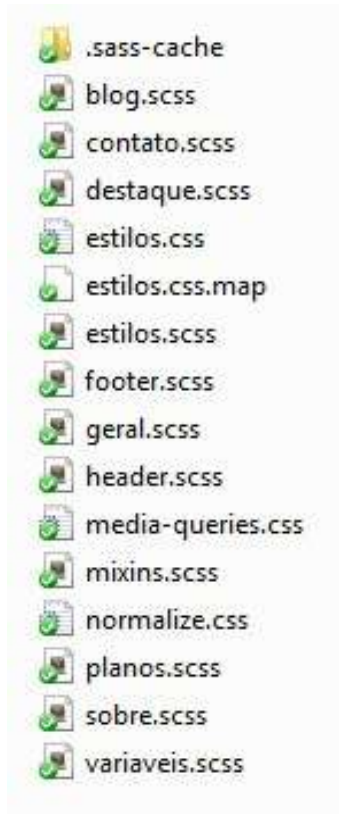


Figura 5.4: Ainda bagunçado

Uma maneira de organizar esses arquivos é separá-los em pastas. Criaremos as seguintes pastas dentro da pasta CSS do projeto:

- layout
- helpers
- base

Na pasta `layout`, ficaram os arquivos das seções do site, como o `header.scss` e o `planos.scss`. Já na pasta `helpers`, fica tudo o que nosso código depende para conseguir funcionar, como o `variaveis.scss` e o `mixins.scss`. Finalmente em `base`, os

arquivos que são a base do nosso site, como o `normalize.css` . Não se preocupe com o `media-queries.css` , teremos um capítulo focado nele.

Por fim, a nova estrutura de pastas dentro da pasta CSS agora fica desta maneira:

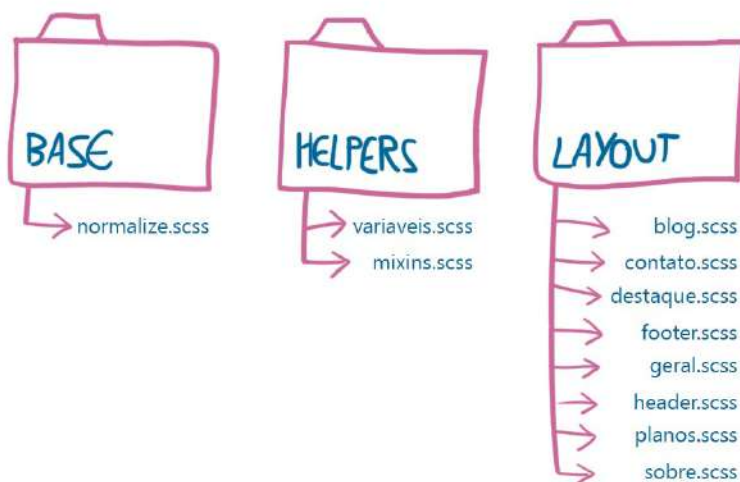


Figura 5.5: Estrutura das pastas

Só precisamos ajustar o caminho dos *imports* no arquivo `estilos.scss` :

```
@import "helpers/variaveis";
@import "helpers/mixins";

@import "layout/geral";
@import "layout/header";
@import "layout/destaque";
@import "layout/sobre";
@import "layout/planos";
@import "layout/blog";
@import "layout/contato";
@import "layout/footer";
```

Agora demos uma boa arrumada, nosso código está modularizado e organizado, e tudo está em seu devido lugar. Poderíamos ir mais a fundo ainda, separando em arquivos isolados cada componente do site, como o menu principal ou os planos, e colocá-los em uma pasta chamada `components`, por exemplo.

5.2 UNIDOS SOMOS MAIS RÁPIDOS

Testando no browser, podemos notar logo de cara dois pequenos problemas de layout:



Figura 5.6: Problemas no layout

O que terá acontecido? Em nosso `index.html`, tirando a fonte que estamos pegando do Google Fonts, estamos chamando três CSS diferentes, são eles:

- `normalize.css`
- `estilos.css`
- `media-queries.css`

Essas três chamadas no `<head>`, no código em si, estão da seguinte forma:

```
<link rel="stylesheet" href="css/normalize.css">
<link rel="stylesheet" href="css/estilos.css">
```

```
<link rel="stylesheet" href="css/media-queries.css">
```

Opa! Esquecemos de alterar o caminho do nosso *reset* CSS, o Normalize! Como ele agora está dentro da pasta `base`, basta alterarmos seu caminho e corrigimos os *bugs*!

```
<link rel="stylesheet" href="css/base/normalize.css">
```

Uma prática muito comum para melhorar a performance de sites é diminuir o número de *requests* que a página faz para o servidor — ou seja, menos pedidos de arquivos que nosso HTML solicita ao servidor. Quando tratamos de HTTP2, isso não é um problema, mas para fins de HTTP normal, diminuir o número de *requests* é uma boa. Podemos deixar o tempo de carregamento do site mais rápido, como ilustro na figura seguinte:

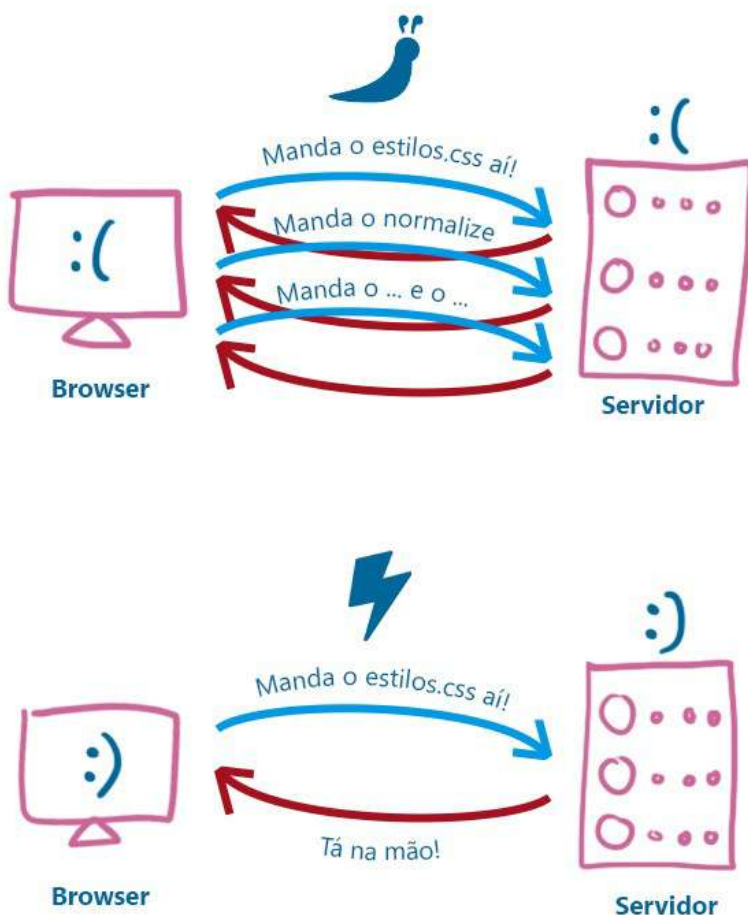


Figura 5.7: Requests

Focando no Apeperia, nas chamadas dos CSS especificamente, algo interessante para fazermos é juntá-las em um único arquivo. Ou seja, concatenar todos os arquivos CSS em um único arquivo CSS.

Onde estamos importando todos nossos arquivos `.scss` ? No `estilos.scss` ! Vamos tentar importar o `normalize` e o `media-`

queries nele também!

Começando pelo `normalize`, como ele é um *reset*, devemos chamá-lo antes de todas as regras CSS de layout:

```
@import "helpers/variaveis";
@import "helpers/mixins";

@import "base/normalize";

@import "layout/geral";
...
```

Erro! Quando ocultamos a extensão do arquivo que estamos importando, o Sass acha que sua extensão é qual? Scss! Mas o `normalize` é CSS! Então só precisamos especificar a extensão correta:

```
@import "helpers/variaveis";
@import "helpers/mixins";

@import "base/normalize.css";

@import "layout/geral";
...
```

Agora sim, né?

```

1  @import url(base/normalize.css);
2
3  /** Geral **/
4  body {
5      font-family: "Roboto Slab", serif; }
6
7  .container {
8      width: 940px;
9      margin: 0 auto; }
10
11  .titulo-secao {
12      text-decoration: underline;
13      font-size: 2em;
14      text-align: center; }
15
16  .sobre,
17  .planos,
18  .blog,
19  .contato {
20      padding: 3em 0 6em 0; }

```

Figura 5.8: Import do normalize ok

"Péra" lá, cara pálida! Esse `import` aí na primeira linha não está concatenando, é só a sintaxe de `import` do CSS comum. O *request* ainda será feito! O que precisamos é **concatenar** o `normalize` e o `estilos`. Precisamos **recortar** o primeiro, e **colar** no segundo! Mas não manualmente, pois aí deixaríamos margem para erro.

Como que concatenamos todos os arquivos de layout no `estilos.scss` ? Importando arquivos `.scss` ! Se todo arquivo `.css` pode ser um `.scss` , basta renomearmos o `normalize` para `.scss` .



Figura 5.9: Normalize agora com a extensão .scss

No arquivo `estilos.scss` , removemos a extensão do arquivo que estamos importando, o `normalize` , para indicar que agora queremos importar um arquivo `.scss` :

```
@import "base/normalize";
```

Done! Verificando o código CSS gerado, podemos notar que o `normalize` faz parte do arquivo também! Ele foi concatenado! Primeiro vem ele, depois vem o CSS do Apeperia:

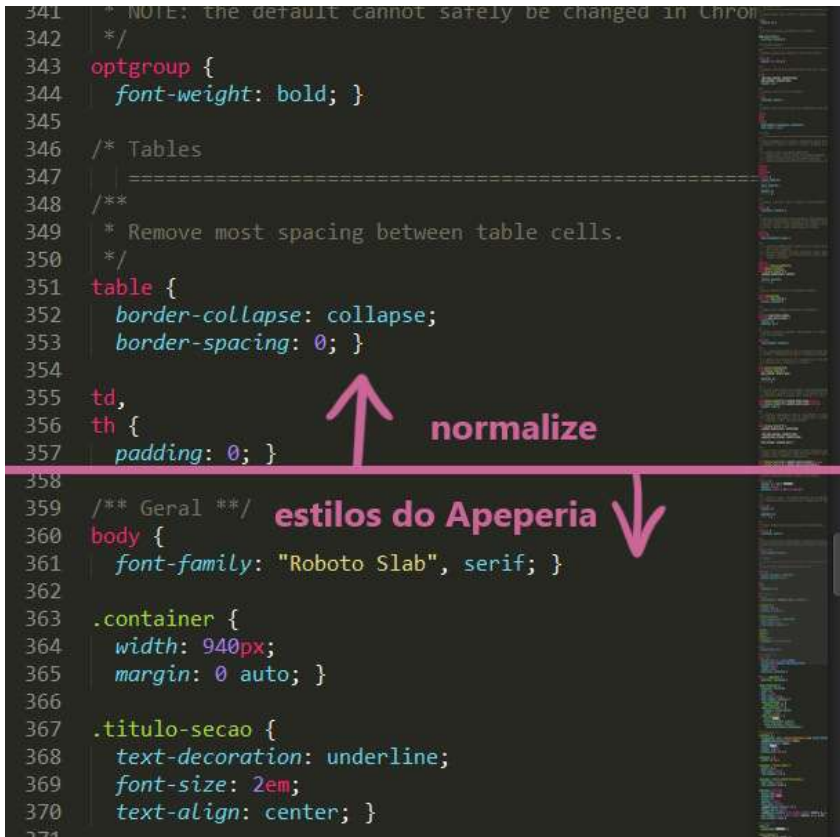


Figura 5.10: Normalize agora com a extensão scss

Agora podemos remover a chamada do `normalize` em nosso `index.html` e correr para o abraço. Nosso `<head>` fica desta maneira:

```
<head>
  <meta charset="UTF-8">
  <title>Apeperia - apps sob medida</title>
  <link rel="stylesheet" href="css/estilos.css">
  <link rel="stylesheet" href="css/media-queries.css">
  <link href='http://fonts.googleapis.com/css?family=Roboto+Slab
:400,300,700' rel='stylesheet' type='text/css'>
</head>
```


5.3 RESUMO

Usando os *imports*, podemos deixar nosso código muito mais modular e com fácil manutenção, de uma maneira inteligente. Vimos que, quando queremos importar arquivos SCSS, nem precisamos passar a extensão, pois ele já considera que o arquivo é um `.scss`.

Criamos também uma estrutura básica de pastas para que os arquivos ficassem um pouco mais organizados. Lembrando de que ela foi uma sugestão, e que você pode criar a estrutura que mais faça sentido com você e/ou seu time.

Outro ponto interessante que trabalhamos foi a concatenação de arquivos, quando importamos todos eles em um único, o `estilos.scss`. A fim de melhorar a performance, diminuimos o número de requisições que o HTML faz, concatenando o `normalize` com o `estilos.css`, apenas renomeando-o para `.scss` e o importando com o `@import` normal.

No próximo capítulo, vamos cortar a dependência do Photoshop/Sketch para quando o chefe pedir uma cor um pouco mais clara ou escura.

CORES DE FORMA MAIS FÁCIL

Alterações sempre existirão, felizmente ou infelizmente. Se você já trabalha na área, sabe como é "bacana" fazer quinze versões de um layout, ou ter de alterar o menu de lugar 4 vezes. Se tratando de layout especificamente, já fiz muito `layout2.psd` , `layout-final.psd` e `layout-final2-agora-vai.psd` . Acontece.

A cor de um elemento visual, um botão ou um logo é uma das primeiras coisas que vem à mente quando pensamos em *rebranding*, mudar a identidade visual da empresa. Um problema que existe na área de design são os chamados "flanelinhas de layout". Se você, querido leitor, for da área de design, já deve ter ouvido frases como: "*coloca um pouco mais pra direita*", "*deixa uma cor mais sóbria*" e "*tenta dar um ar mais honesto*". Acontece, também.

Uma coisa que eu perdia tempo era quando o designer ou meu chefe pedia para deixar a cor mais clara ou mais escura. Ou quando eu mesmo sentia a necessidade de alterar uma determinada cor, pois o contraste não ficou muito bom ou algo do gênero.

Por exemplo, pensando na cor padrão do Apeperia, o `#c69` , digamos que nosso chefe pediu para que essa cor seja um pouco mais clara.

Meu processo era:

1. Copiar o hexadecimal dessa cor;
2. Ir ao Photoshop;
3. Clicar na ferramenta de cor;
4. Colar o hexadecimal da cor;
5. Puxar a cor escolhida um pouco para cima;
6. Copiar o novo hexadecimal;
7. Colá-lo no CSS.

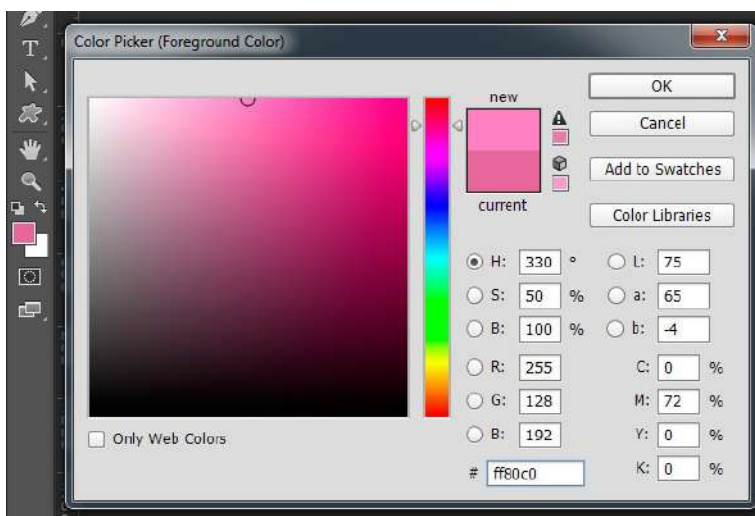


Figura 6.1: O color picker do Photoshop

COLOR PICKER

Caso queira testar e não tiver o Photoshop, há ferramentas online para seleccionar cores facilmente, como o <http://www.colorpicker.com>.

"Só" isso! Como já adiantei sua vida, é só pegar o rosa mais claro #ff80c0 e alterar a cor padrão do Apeperia, que está no arquivo

```
variaveis.scss :
```

```
$cor-padrao: #ff80c0;
```

Não foi tão sofrido! O site fica assim:



Figura 6.2: O color picker do Photoshop

E se eu disser que a esposa do chefe disse que não gostou do rosa novo, e é para voltar para o original? Será que, no nosso cotidiano, nós salvamos toda pequena modificação? E se tivéssemos fechado nosso editor de texto? Ctrl + Z não funcionaria! Qual era mesmo a cor antiga?

Temos uma outra forma de deixarmos a cor mais clara, conservando a original. Podemos "terceirizar" esse trabalho.

Primeiro, voltemos a cor padrão para o #c69 da vitória:

```
$cor-padrao: #c69;
```

O Sass possui algumas funções nativas especializadas em modificar cores. Nós passamos uma cor para ela, dizemos o que queremos mudar, e o Sass calcula isso para nós! São as **funções de cor**, ou *color functions*.

Uma dessas funções é a `lighten`, que, como o próprio nome diz, permite deixarmos uma cor um pouco mais clara. Vamos testá-la:

```
$cor-padrao: lighten(#c69);
```

Só salvar para ver o... Erro?

```
error helpers/variaveis.scss (Line 2: wrong number of arguments (1 for 2) for `lighten`)
```

Aí diz que o número de argumentos para a função `lighten` está errado, ela precisa de dois argumentos.

Bom, nós estamos usando uma função para deixar uma cor mais clara, mas não informamos quão mais clara! Como a esposa do chefe só disse que queria um pouco mais clara, digamos que 5 ou 10% já basta. Agora precisamos especificar essa informação quando usamos a função `lighten`:

```
$cor-padrao: lighten(#c69, 10%);
```

Agora sem erros, focando na parte de planos do site, ficamos com o seguinte resultado:

Plano Básico	Plano Enterprise	Plano Corporation
R\$ 500,00	R\$ 1500,00	R\$ 9500,00
Aplicativos 1	Aplicativos 3	Aplicativos 5
Manutenções Programadas 5	Manutenções Programadas 20	Manutenções Programadas ∞
Helpdesk 24hs 	Helpdesk 24hs 	Helpdesk 24hs 
App SEO 	App SEO 	App SEO 
Atendente Exclusivo 	Atendente Exclusivo 	Atendente Exclusivo 
Sistema Operacional 1	Sistema Operacional 3	Sistema Operacional 6
Comprar	Comprar	Comprar

Figura 6.3: Planos com o rosa mais claro

O interessante é que se precisarmos desfazer essa alteração, bastaria retirarmos a função. A cor original será preservada! E não dependemos mais de uma ferramenta com *color picker* para isso. Se nosso querido chefinho pedir para deixar a cor mais clara ainda, basta alterarmos o segundo argumento da função `lighten`.

6.1 CLARO! E ESCURO?

Se existe uma função para deixar uma cor mais clara, deve ter também uma função para deixar a cor mais escura! E existe! É a função `darken`.

E da mesma forma que a `lighten`, ela só precisa de duas informações, dois argumentos para funcionar:

- A cor que queremos alterar;
- A porcentagem de quão escuro queremos.

Podemos fazer um teste na cor auxiliar, ainda no arquivo `variaveis.scss`:

```
$cor-auxiliar: darken(#069, 20);
```

E temos um azul bem mais escuro na cor auxiliar como resultado:



Figura 6.4: Planos com o azul mais escuro

Reparou que não passei o símbolo de porcentagem (%) na função? O Sass assume que o segundo valor é uma porcentagem automaticamente, nem com isso precisamos nos preocupar.

Para não deixar nosso layout muito diferente das cores do logo, vamos voltar as cores para as originais. Marquei essa página e voltei aos capítulos iniciais em que tratamos de variáveis para pegar os códigos das cores e... **Não faça isso!** As cores estão intactas, só precisamos retirar as funções de cor:

```
$cor-padrao: #c69;  
$cor-auxiliar: #069;
```

INDO ALÉM

Fica o convite ao leitor de dar uma olhada na documentação do Sass para ver outras funções de cor interessantes: <http://sass-lang.com/documentation/Sass/Script/Functions.html>.

6.2 COMENTANDO SEU CÓDIGO

Uma boa prática de qualquer linguagem é sempre utilizar os comentários a fim de deixar seu código bem documentado, seja para explicar ou separar seções diferentes. Nosso código já veio bem comentado. Podemos encontrar comentários separando as seções como `/** Header */` ou `/** Destaque */`. Como separamos essas seções em vários arquivos, poderíamos até mesmo subtrair esses comentários, mas manter o padrão.

Os arquivos `variaveis.scss` e `mixins.scss` não estão comentados, então corrigiremos isso. Primeiro, no `variaveis.scss`, vamos inserir um comentário informando do que se trata esse arquivo:

```
/** Variaveis */  
$cor-padrao: #c69;  
$cor-auxiliar: #069;
```

Depois, faremos o mesmo no `mixins.scss`:

```
/** Mixins */  
@mixin borda-arredondada($raio: 0.3em) {  
    ...  
}  
  
@mixin sombra-padrao {  
    ...  
}  
  
...
```

Vendo o CSS gerado, no topo dele, podemos notar que nossos comentários estão aparecendo no CSS gerado, o que não faz muito sentido, uma vez que esses comentários deveriam ficar somente no nosso código Sass. Uma outra forma de fazer comentários nos arquivos do Sass é usar o comentário de linha, o `//`.

Alteremos os comentários que fizemos para essa nova forma, primeiro no `variaveis.scss`:


```
// Variaveis
$cor-padrao: #c69;
$cor-auxiliar: #069;
```

Depois no `mixins.scss` :

```
// Mixins
@mixin borda-arredondada($raio: 0.3em) {
  ...
}

@mixin sombra-padrao {
  ...
}

...
```

Conferindo o CSS gerado, note que os comentários sumiram, e que constam somente em nossos arquivos de desenvolvimento.

6.3 RESUMO

Vimos que não precisamos depender da memória e de ferramentas quando se trata de alterações das cores dos nossos sites. O Sass possui nativamente várias funções de cor, em que só precisamos passar a cor que queremos alterar e a porcentagem dessa mudança.

Comentar código é uma prática bacana. Vimos como deixar um determinado comentário somente no arquivo do Sass, para ele não ir para o CSS gerado. Fazemos isso apenas alterando a sintaxe de comentário padrão do CSS (`/* */`) para comentários de linha, usando o `//` .

Agora, vamos evitar repetição de nosso código, estendendo nossos *mixins*, e deixar nosso CSS mais performático.

MELHORANDO A PERFORMANCE COM EXTENDS E PLACEHOLDERS

Todo desenvolvedor front-end ouve durante a vida, seja de amigos ou de colegas do trabalho, o famoso conceito/estratégia DRY (*Don't Repeat Yourself*). Relaxa que não vou repetir o mesmo texto do capítulo 3. *Reutilizando seu código com mixins*, afinal, DRY no livro também. Uma vantagem de evitar código repetido é que, muitas vezes, acabamos melhorando a **performance** de nosso site/projeto.

E como deixar o código mais rápido? Aí que está: o usuário vai ver o código ou a página sendo carregada? O que é mais importante para ele? Por isso o site/app não precisa, aos olhos do usuário, ser de fato rápido ou performático. Entretanto, **parecer** rápido é importante.

Entrar em um site e se deparar com uma tela em branco por mais de 1,5 segundos dá tempo até para refletir sobre a vida. Por isso que essa questão de performance sempre surge em *meetups* da área, e é assunto de posts e palestras.

No código do Apeperia, no `destaque.scss`, vamos dar uma

olhada em algumas partes do CSS que foi gerado pelo Sass, aproximadamente na linha 430:

```
.destaque button {  
  ...  
  -webkit-box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);  
  box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);  
  ...  
}
```

Como que colocamos esse `box-shadow` aí? Com o `mixin` `sombra-padrao`. Vamos relembrar do código de criação desse *mixin*, que está no arquivo `mixins.scss`:

```
@mixin sombra-padrao {  
  -webkit-box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);  
  box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);  
}
```

E vamos agora dar uma olhada em como incluímos esse *mixin* no nosso código, que agora está no arquivo `destaque.scss`:

```
.destaque button {  
  ...  
  @include sombra-padrao;  
  ...  
}
```

Voltando ao CSS gerado, repare que o código desse *mixin* (`box-shadow: 0 2px 6.65px...`) é repetido diversas vezes em nosso código, mais especificamente, em todos os botões. Facilitamos a manutenção do nosso código usando esse *mixin*. Porém, pensando em performance, o CSS gerado ainda pode dar uma melhoria.

Precisamos bolar algo para que esse código da sombra (`box-shadow: 0 2px 6.65px...`) não fique se repetindo. Se fôssemos fazer manualmente, uma solução seria juntar as declarações que fazem a sombra e colocá-las com suas respectivas regras, mas todas juntas:

```
.destaque button,  
.plano button,
```

```
.contato button {
  -webkit-box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);
  box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);
}
```

Mas fazer essa ou qualquer alteração no CSS gerado não teria muito futuro, pois quando fizéssemos alguma alteração em qualquer arquivo Sass, ela se perderia quando nosso código fosse compilado.

O *mixin* tem essa característica. Sempre que o incluímos em uma regra CSS, essencialmente ele vai copiando e colando blocos de código, várias vezes. Se incluirmos em 500 lugares diferentes, ele vai copiar o código 500 vezes. Para evitar isso, vamos usar um recurso do Sass chamado ***extend***.

O *extend* nos permite que compartilhemos código entre várias regras CSS, como fizemos no código anterior, mas de uma maneira dinâmica. Vamos criar um *extend* chamado `sombra-padrao` abaixo dos mixins que já criamos, no arquivo `mixins.scss`.

A ideia do ***extend*** é que ele ***estende*** um outro seletor. Logo, para criarmos um, apenas criamos uma regra CSS comum, seja um ID ou uma classe:

```
.sombra-padrao {
  -webkit-box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);
  box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);
}
```

Classe criada, agora precisamos estendê-la no botão de destaque. Quando usamos o `@include`, estamos incluindo um *mixin*. Para estendermos uma classe, usaremos o `@extend`, podendo agora retirar o *include* do *mixin* `sombra-padrao`:

```
.destaque button {
  ...
  @extend .sombra-padrao;
  ...
}
```

Conferindo no CSS gerado, o `.plano button` foi lá para a primeira linha, isolado, antes mesmo do código do `normalize`.

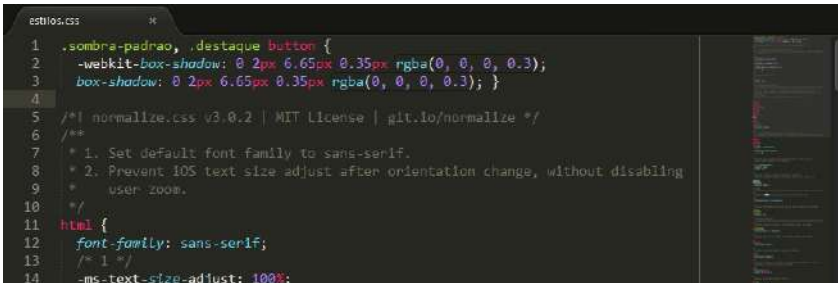


Figura 7.1: Regra da sombra foi para o topo do CSS

E conferindo o restante do CSS do `.destaque button`, podemos ver que está tudo ok com as outras declarações feitas:

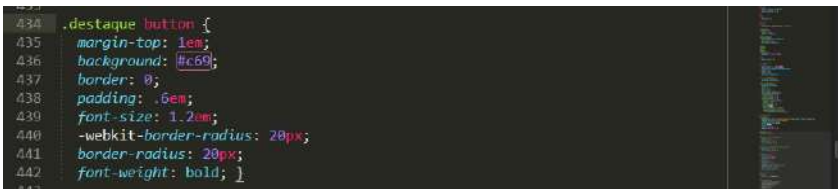


Figura 7.2: Restante das declarações

Mas agora vamos reparar bem no código que foi gerado:

```
.sombra-padrao, .destaque button {  
  -webkit-box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);  
  box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);  
}
```

O que é essa classe `.sombra-padrao` ali junto com o nosso botão? Dando um `Ctrl + F` no código-fonte da nossa página, no Chrome especificamente, podemos observar que ela não está sendo utilizada em lugar algum:



Figura 7.3: Restante das declarações

A classe está ali pois, quando a estendemos no `.destaque button`, o Sass deixou esse resquício, uma classe que não é usada. Foi pensando nisso que, na versão 3.2 do Sass, uma funcionalidade especial chamada `placeholder` foi criada. Ela também é conhecida como "classe silenciosa", justamente porque ela não aparece no CSS gerado.

Ele é bem parecido com uma classe, mas no lugar do `.` (ponto), usamos um `%` (porcentagem). Vamos criar um *placeholder* para a sombra. Primeiro, no arquivo `mixins.scss`, modificaremos a classe `.sombra-padrao` que criamos anteriormente. Basta substituir o ponto pela porcentagem:

```

%sombra-padrao {
  -webkit-box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);
  box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);
}

```

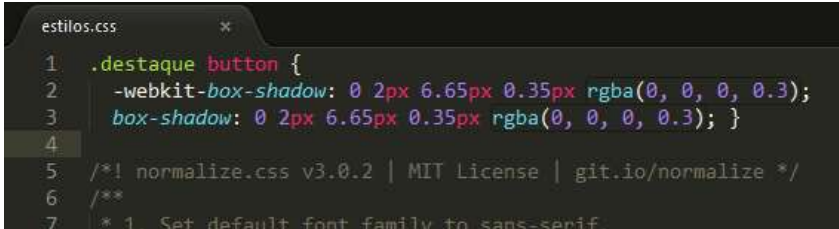
Agora no arquivo `destaque.scss`, na regra do botão (a última que está lá), mudamos de classe para *placeholder*:

```

.destaque button {
  ...
  @extend %sombra-padrao;
  ...
}

```

Conferindo o CSS gerado (`estilos.css`), logo na primeira linha conseguimos ver que está tudo perfeito. Agora o seletor da sombra não aparece mais inutilmente em nosso código:



```
1 .destaque button {
2   -webkit-box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);
3   box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3); }
4
5 /*! normalize.css v3.0.2 | MIT License | git.io/normalize */
6 /**
7  * 1. Set default font family to sans-serif.
```

Figura 7.4: Restante das declarações

Vamos continuar incluindo o `placeholder` da sombra nas regras dos outros botões, substituindo o `mixin` de sombra existente. Primeiro no `planos.scss` :

```
.plano button {
  ...
  @include borda-arredondada;
  @extend %sombra-padrao;
  ...
}
```

Depois no `contato.scss` :

```
.contato button {
  ...
  @include borda-arredondada;
  @extend %sombra-padrao;
}
```

Acabamos de adicionar o *placeholder* da sombra dentro da regra do botão de plano e de contato abaixo das outras declarações e dos *includes* dos *mixins* da borda. Checando o CSS, logo na primeira linha, chegamos a este resultado:

```
.destaque button, .plano button, .contato button {
  -webkit-box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);
  box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);
}
```

Conseguimos fazer esse código da sombra, que era repetido três vezes, ficar isolado como uma única regra, ganhando mais pontos no quesito **performance** de nosso site.

Podemos agora deixar nosso arquivo `mixins.scss` um pouco mais limpo, removendo o *mixin* da sombra, já que agora estamos usando um `placeholder` para isso:

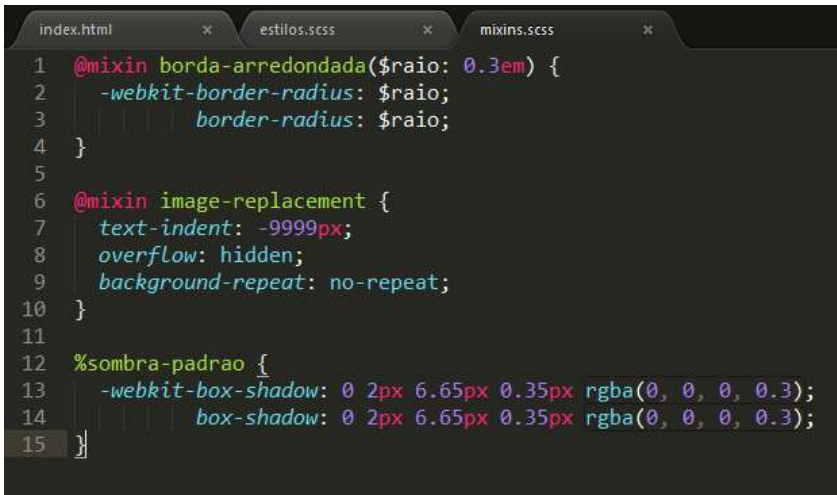


Figura 7.5: Tirando o mixin da sombra

7.1 PRATICANDO MAIS

Entendido o *extend*, podemos treinar agora fazendo o mesmo processo com o *mixin* do `image-replacement` no arquivo `mixins.scss`:

```
@mixin image-replacement {
  text-indent: -9999px;
  overflow: hidden;
  background-repeat: no-repeat;
}
```

O `image-replacement` (substituição por imagem) serve para exibir uma imagem em algum elemento que originalmente foi feito

com texto. Estamos usando essa técnica em dois lugares em nosso CSS:

- `.plataformas li`
- `.social li a`

O problema é que as declarações estão sendo repetidas, pois estamos usando um *mixin*. Vamos resolver isso substituindo esse *mixin* do `image-replacement` por um *placeholder*:

```
%image-replacement {  
  text-indent: -9999px;  
  overflow: hidden;  
  background-repeat: no-repeat;  
}
```

Placeholder criado. Agora vamos pegar as regras CSS `.plataformas li` e `.social li a`, e substituir a inclusão do *mixin* `image-replacement` pelo *placeholder* `%image-replacement`. Primeiro no arquivo `sobre.scss`:

```
.plataformas li {  
  ...  
  @extend %image-replacement;  
  ...  
}
```

Depois no arquivo `footer.scss`:

```
.social li a {  
  ...  
  @extend %image-replacement;  
  ...  
}
```

Conferindo no topo do CSS gerado, chegamos ao seguinte resultado:

```
.plataformas li, .social li a {  
  text-indent: -9999px;  
  overflow: hidden;  
  background-repeat: no-repeat;  
}
```

Pronto, nosso CSS acaba de ficar um pouco mais performático, pois isolamos as declarações para fazer o *image replacement* em um *placeholder*.

7.2 MIXINS E EXTENDS: QUANDO USAR UM OU OUTRO?

Já criamos dois *placeholders*, e estamos utilizando-os em vários locais, sempre com a sintaxe `@extend` do Sass. Outro código de *mixin* que podemos tentar isolar é o responsável pela borda arredondada:

```
@mixin borda-arredondada($raio: 0.3em) {  
  -webkit-border-radius: $raio;  
  border-radius: $raio;  
}
```

Repare que esse `border-radius` é repetido em todos os botões que usamos no site, como o botão de destaque, por exemplo. Mas vamos olhar na única diferença visual entre o botão de destaque e o botão dos planos, nem precisamos olhar o código, o layout já denuncia:



Figura 7.6: Diferença nas bordas

Colocamos na propriedade `border-radius` valores diferentes entre esses botões. Só vamos dar uma conferida em seus respectivos códigos. Primeiro no arquivo `destaque.scss` :

```
.destaque button {
  ...
  @include borda-arredondada(20px);
  ...
}
```

Depois no arquivo `plano.scss` :

```
.plano button {
  ...
  @include borda-arredondada;
  ...
}
```

Neste último é usado o valor de `0.3em` , que está vindo do padrão que deixamos na criação do *mixin*. No arquivo `mixins.scss` , podemos conferir como está o código até o momento:

```
@mixin borda-arredondada($raio: 0.3em) {
  ...
}
```

Vamos então criar um *placeholder* para a borda, no arquivo `mixins.scss` :

```
%borda-arredondada($raio: 0.3em) {
  -webkit-border-radius: $raio;
  border-radius: $raio;
}
```

Olhando no terminal, podemos visualizar a seguinte mensagem de erro:

```
error helpers/mixins.scss (Line 18: Invalid CSS after "%borda-arredondada": expected selector, was "($raio: 0.3em)")
```

Aparentemente nosso *placeholder* está com problemas. Aí diz que cometemos algum erro de sintaxe. Isso porque os *extends* não funcionam da mesma forma que os *mixins*.

Nos *mixins* conseguimos especificar valores diferentes logo na inclusão. Já nos *extends* isso não é possível, pois ele é feito

justamente para juntar várias regras em uma, com os mesmos valores.

Se precisamos passar um valor padrão ou qualquer valor diferente para uma determinada regra CSS, o *mixin* é a saída mais adequada. Isso porque o *extend* não dá suporte para isso, já que inviabilizaria seu propósito.

Podemos apagar o *placeholder* da borda que criamos, pois, no nosso projeto, o *mixin* é o mais indicado. Deixando o arquivo `mixins.scss` desta forma:

```
// Mixins
@mixin borda-arredondada($raio: 0.3em) {
  -webkit-border-radius: $raio;
    border-radius: $raio;
}

%image-replacement {
  text-indent: -9999px;
  overflow: hidden;
  background-repeat: no-repeat;
}

%sombra-padrao {
  -webkit-box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);
    box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);
}
```

7.3 RESUMO

Vimos outra maneira de não repetir nosso código utilizando *extends*, mais especificamente criando *placeholders*. Evitamos códigos inchados apenas ao agrupar seletores que compartilham os mesmos estilos em uma única regra.

Vimos também que, quando usamos o *extend*, o código no CSS compilado vai para o topo do documento, o que poderia vir a ser um risco por conta de alguma outra regra mais abaixo sobrescrevendo a primeira. Conseguimos fazer o código da sombra

padrão e do nosso *image replacement*, que era repetido três vezes, ficar isolado e aparecer no código apenas uma única vez.

Aprendemos também que, às vezes, o *mixin* é nossa única saída. Resumidamente, se você chegou a uma situação em que está repetindo muito o mesmo código em vários lugares, você pode usar o *mixin* ou o *extend*. Se é **exatamente** o mesmo código, com os mesmos valores, o *extend* é o mais indicado.

Sugiro que você faça testes em seus projetos para ver a diferença entre eles. Recomendo esse post interessantíssimo à respeito do tema que dá vitória para o *mixin*: <http://csswizardry.com/2016/02/mixins-better-for-performance>.

Aprendemos tudo isso a fim de melhorar sempre mais a performance do nosso CSS, entregando um CSS mais performático. Agora vamos dar uma atenção para aquele *media-queries.css* e ver no que o Sass consegue facilitar nossa vida em relação a *media queries*.

APROXIMANDO REGRAS CSS E MEDIA QUERIES

Com o advento do mobile no mundo, foi uma tendência natural termos de começar a pensar em nossos sites para diferentes tamanhos de tela, para deixar o site adaptável. O espaço começou a ficar menor.

Então, chegava determinado momento em um projeto de um site que precisávamos deixar o site usável tanto no desktop quanto no celular. Mas como fazer isso?

Identificar pelo dispositivo é uma boa. Mas até que ponto um aparelho é considerado mobile ou desktop? Ou tablet, ou *phablet*? Por isso, foi adotada uma convenção de se pegar o **tamanho** da tela como parâmetro. Se a tela tiver **X** pixels de largura, fique de uma maneira; se tiver **Y**, fique de outra. E uma das formas que temos para fazer isso é utilizando o recurso de *media query*, disponível a partir do CSS3.

Em nosso projeto, usamos esse recurso a fim de adaptar o layout para resoluções abaixo de 980px , como podemos verificar logo no início do arquivo `media-queries.css` :

```
@media (max-width: 980px) {  
  ...  
}
```

E estamos importando esse CSS em nosso `<head>` , no arquivo

index.html :

```
<head>
  ...
  <link rel="stylesheet" href="css/estilos.css">
  <link rel="stylesheet" href="css/media-queries.css">
  <link href='http://fonts.googleapis.com/css?family=Roboto+Slab
:400,300,700' rel='stylesheet' type='text/css'>
</head>
```

Visando diminuir o número de *requests*, uma outra opção seria criar toda esse *media query* internamente, direto no CSS principal:

```
...

.social li a {
  display: block;
  width: 55px;
}

@media (max-width: 980px) {
  .container {
    width: 90%;
  }

  header {
    height: auto;
  }

  ...
}
```

Precisamos dar um jeito de juntar nosso arquivo `media-queries.css` com os outros. No capítulo 5. *Organizando a bagunça*, já vimos como solucionar isso: basta concatenarmos com os demais arquivos, importando-o no `estilos.scss` como os demais arquivos.

```
...
@import "layout/footer";

@import "media-queries.css";
```

Feito isso, vamos abrir o arquivo CSS gerado:

```
estilos.css
1 @import url(media-queries.css);
2 .plataformas li, .social li a {
3   text-indent: -9999px;
4   overflow: hidden;
5   background-repeat: no-repeat; }
6
7 .destaque button, .plano button, .contato button {
8   -webkit-box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);
9   box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3); }
10
11 /*! normalize.css v3.0.2 | MIT License | git.io/normalize */
12 /*!*/
```

Figura 8.1: Import com sintaxe do CSS normal

Opa, ele usou a sintaxe de *import* do CSS normal e não juntou os arquivos em um só. Mas também já vimos que, para que o Sass concatene os arquivos de fato, precisamos transformar o arquivo CSS em um arquivo SCSS.

Vamos renomeá-lo!

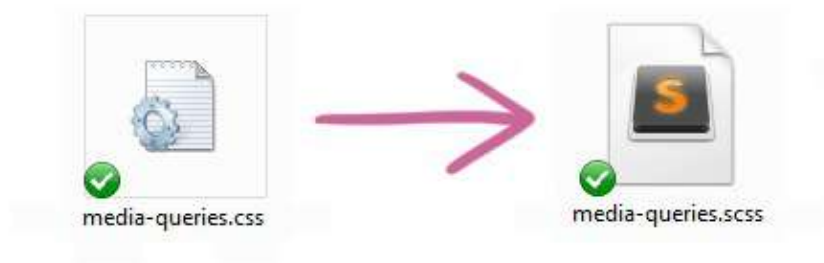


Figura 8.2: Renomeando o arquivo de media queries


Agora basta retirarmos a extensão de seu `import`, ainda no arquivo `estilos.scss`:

```
...
@import "layout/footer";
```



```
@import "media-queries";
```

Conferindo o CSS gerado, podemos observar que agora sim o Sass realizou a concatenação do nosso `media-queries.scss` com os outros arquivos:

A screenshot of a code editor window titled 'estilos.css'. The code is as follows:

```
701
702 .social a[href*="instagram"] {
703     background-image: url(../imagens/instagram.png); }
704
705 @media (max-width: 980px) {
706     .container {
707         width: 90%; }
708
709     header {
710         height: auto; }
711
712     header, container {
```

Figura 8.3: Media queries concatenados

Podemos então remover a chamada do CSS de *media queries* do nosso `index.html`, afinal, ela já pertence ao `estilos.css`:

```
<head>
  <meta charset="UTF-8">
  <title>Apeperia - apps sob medida</title>
  <link rel="stylesheet" href="css/estilos.css">
  <link href='http://fonts.googleapis.com/css?family=Roboto+Slab
:400,300,700' rel='stylesheet' type='text/css'>
</head>
```

Bacana, menos outro *request*! Mas repare que as regras das *media queries* estão um tanto quanto afastadas do código que elas sobrescrevem.

Precisamos ficar lembrando das regras que colocamos e de quais precisamos sobrescrever. Outro ponto é que, caso o nome da classe `.container` mude, teríamos de alterar seu nome tanto no arquivo `geral.scss` quanto no `media-queries.scss`.

O Sass resolveu esses problemas ao permitir aninharmos *media*

queries dentro de regras CSS. No arquivo `geral.scss`, podemos fazer uma *media query* direto na regra da classe `.container`, por exemplo. Uma *media query* aninhada:

```
.container {  
  width: 940px;  
  margin: 0 auto;  
  
  @media (max-width: 980px) {  
    width: 90%;  
  }  
}
```

Em nosso CSS gerado, aproximadamente na linha 370, podemos conferir o resultado:

```
.container {  
  width: 940px;  
  margin: 0 auto;  
}  
  
@media (max-width: 980px) {  
  .container {  
    width: 90%;  
  }  
}
```

Com essa *feature* de *media queries* aninhados, não precisamos mais ficar lembrando de quais eram as regras que colocamos e quais precisamos sobrescrever, pois agora está tudo próximo, a regra original e a regra sobrescrita.

Como já aninhamos a *media query* da classe `.container`, não precisamos mais dela no arquivo `media-queries.scss`. Portanto, podemos removê-la:



Figura 8.4: Media queries concatenados

Façamos o mesmo processo com algumas outras *media queries* que seriam interessantes estarem próximas das regras originais. Por exemplo, no arquivo `header.scss`, podemos criar a *media query* aninhada na regra do próprio `header`:

```
header {
  border-top: 5px solid $cor-padrao;
  ...

  @media (max-width: 980px) {
    height: auto;
  }
}
```

Conferindo o CSS gerado, fica assim:

```
header {
  border-top: 5px solid $cor-padrao;
  ...
}

@media (max-width: 980px) {
  header {
    height: auto;
  }
}
```

E novamente, podemos excluir o `header { height: auto; }` do arquivo `media-queries.scss`.

Ainda no arquivo `header.scss` , repare que a segunda regra (`header .container`) pode ser aninhada na regra anterior (`header`). Fazendo isso, nosso código fica assim:

```
header {  
  border-top: 5px solid $cor-padrao;  
  ...  
  
  .container {  
    position: relative;  
  }  
  
  @media (max-width: 980px) {  
    height: auto;  
  }  
}
```

Agora dando uma olhada no `media-querie.scss` novamente, veja que seria interessante aninhar essa *media query* do `header .container` também. Ela está apenas mudando o posicionamento da classe `.container` que está dentro do `header` para estático. Façamos exatamente isso, da seguinte forma:

```
header {  
  border-top: 5px solid $cor-padrao;  
  ...  
  
  .container {  
    position: relative;  
  }  
  
  @media (max-width: 980px) {  
    height: auto;  
  
    .container {  
      position: static;  
    }  
  }  
}
```

Com isso feito, mais uma vez podemos excluir a regra do `media-queries.scss` , desta vez a regra do `header .container` .

E agora com o `header h1` , podemos fazer o mesmo processo, excluindo-a do arquivo `media-queries.scss` e colocando-a internamente no `header.scss` :

```
header {
  border-top: 5px solid $cor-padrao;
  ...

  .container {
    position: relative;
  }

  @media (max-width: 980px) {
    height: auto;

    .container {
      position: static;
    }

    h1 {
      max-width: 50%;
      margin: 0 auto;
    }
  }
}
```

No `media-queries.scss` , façamos o mesmo com a regra `header h1 img` . Nessa regra não estamos pegando as imagens, dentro do `h1` , que estão dentro do `header` ? Então vamos aninhá-la também, direto no `h1` do arquivo `header.scss` , não esquecendo de excluí-la do `media-queries.scss` . Aproveitaremos também para comentar um pouco o código.

```
header {
  border-top: 5px solid $cor-padrao;
  ...

  .container {
    position: relative;
  }

  @media (max-width: 980px) {
    height: auto;
```

```

.container {
  position: static;
}

h1 {
  max-width: 50%;
  margin: 0 auto;

  img {
    max-width: 100%;
    margin: .5em auto;
    display: block;
  }
} // fim h1
} // fim media query

} // fim header

```

Pronto, agora todas as regras de *media queries* relacionadas ao `header` estão próximas umas das outras, deixando a manutenção do nosso código menos trabalhosa. Note que todas as regras que antes estavam em um arquivo à parte estão agora internamente direto nas regras do CSS — nesse caso, no arquivo `header.scss`.

O código do nosso arquivo `media-queries.scss`, por enquanto, está desta forma:

```

@media (max-width: 980px) {
  .menu-principal {
    position: static;
    display: block;
    text-align: center;
  }

  .menu-principal li {
    margin: .4em;
  }

  .destaque {
    font-size: 14px;
    height: 250px;
  }

  .menu-principal,
  .menu-principal li,
  .post-destaque,

```

```

.posts-antigos,
form {
  width: auto;
}
}

```

8.1 VARIÁVEL NA MEDIA QUERY

Passado algum tempo, os chefes do Apeperia notam que o tamanho de tela mais utilizado entre os usuários *mobile* mudou. Antes era 980px , agora é 950px . É nosso trabalho fornecer uma página usável para esse novo padrão, mudando o valor de nossas *media queries*.

Primeiro temos de localizar **todas** as *media queries* para alterarmos seu valor, valor este que está sendo repetido muitas vezes. Também precisamos lembrar de cada lugar que possui uma *media query*. O que podemos fazer para evitar esse transtorno toda vez que esse valor mudar? Isolá-lo em uma variável!

Vamos criar uma variável chamada \$ponto-de-quebra com o valor de 950px , no fim do nosso `variaveis.scss` :

```

// Variaveis
$cor-padrao: #c69;
$cor-auxiliar: #069;

$ponto-de-quebra: 950px;

```

Agora troquemos os valores de 980px , que estão nas *media queries*, pela variável recém-criada. Primeiramente no arquivo `geral.scss` :

```

.container {
  ...

  @media (max-width: $ponto-de-quebra) {
    width: 90%;
  }
}

```

E depois no arquivo `header.scss` :

```
header {  
  ...  
  
  .container {  
    position: relative;  
  }  
  
  @media (max-width: $ponto-de-quebra) {  
    height: auto;  
  
    ...  
  }  
}
```

Feito isso, o CSS deve ser compilado normalmente. Agora, caso seja necessário no futuro alterar esse valor de `950px` , basta alterarmos o valor na variável `$ponto-de-quebra` .

8.2 ISOLANDO REGRAS INTEIRAS

Uma situação possível que pode nos pegar desprevenidos no futuro é se precisássemos alterar o `max-width` de nossas *media queries* para `min-width` . Teríamos de alterar em **diversos** lugares.

Pegemos como exemplo o código do `header` no arquivo `header.scss` :

```
/** Header */  
header {  
  ...  
  
  @media (mix-width: $ponto-de-quebra) {  
    height: auto;  
  
    .container {  
      position: static;  
    }  
  
    h1 {  
      ...  
  
      ...  
    } // fim h1  
  }  
}
```



```

    } // fim media query
} // fim header

```

Não estamos isolando o ponto de quebra da *media query* em uma variável? Então, outra abordagem que podemos fazer é isolar não somente esse ponto de quebra, mas a regra da *media query* inteira!

No arquivo `variaveis.scss`, estamos usando a variável `$ponto-de-quebra`. Ela não é para mobile? Vamos criar uma variável que remeta a isso então, uma *media query* para mobile:

```
$mq-mobile: "(max-width: 950px)";
```

Repare que usamos as aspas, justamente para dizer ao Sass que é uma variável do tipo texto, uma *string*. Agora no arquivo `header.scss`, basta chamarmos nossa variável recém-criada, desta forma:

```

/** Header */
header {
    ...

    @media $mq-mobile {
        height: auto;

        .container {
            position: static;
            ...
        }
    } // fim media query
} // fim header

```

Bacana! Com isso, nos deparamos com uma bela mensagem de erro em nosso terminal:

```
error layout/header.scss (Line 13: Invalid CSS after " @media ":
expected media query (e.g. print, screen, print and screen), was "
$mq-mobile {"
```

Como assim está esperando uma *media query*? Mas a colocamos

ali, o Sass que não achou! Será?

Precisamos dizer para ele para pegar esse valor (`$mq-mobile`) e encaixar dentro de outro valor (`@media`). Essa funcionalidade é o que chamamos de interpolação, ou *interpolation*.

Para isso, basta colocarmos a variável `$mq-mobile` em volta de chaves e com um cerquilha, desta maneira:

```
/** Header */
header {
  ...

  @media #{$mq-mobile} {
    height: auto;

    ...
  } // fim media query
} // fim header
```

Maravilha, tudo funcionando! Vamos chamar a mesma variável no arquivo `geral.scss` também:

```
.container {
  width: 940px;
  margin: 0 auto;

  @media #{$mq-mobile} {
    width: 90%;
  }
}
```

Podemos chamá-la no arquivo `media-queries.scss` também:

```
@media #{$mq-mobile} {

  .menu-principal {
    position: static;
    display: block;
    text-align: center;
  }

  ...
}
```

}

Assim, ganhamos mais velocidade quando se trata de manutenção, o que acaba gerando mais produtividade, pois podemos focar em outros aspectos do projeto.

8.3 RESUMO

Aprendemos uma outra forma de fazer *media queries*. Em vez de deixar o código em outro arquivo ou no final do arquivo principal, o Sass possui a funcionalidade de deixar a *media query* aninhada diretamente no seletor. Assim, as regras originais e as regras de sobrescrita ficam mais próximas uma das outras.

Não se preocupe com a repetição do código `@media (max-width...` , pois normalmente o servidor compacta essas informações antes de enviar para browser. Então, sem problemas! Aliás, algo bacana de isolar é o *breakpoint* que utilizamos em nossas *media queries*, pois isso deixa nosso código mais flexível.

Vimos também que, além de conseguirmos isolar apenas o *breakpoint* usado em uma *media query*, podemos isolar a *media query* por inteiro, toda a sua regra, seja `max-width` , `min-width` ou qualquer outra. Também não podemos esquecer que, sempre que chamarmos uma variável de texto em nosso código, devemos interpolá-la, usando a sintaxe `#{ $nome-da-variavel }` .

Agora vamos ver um "canivete suíço" para quando estamos trabalhando com Sass, o **Compass**.

CÓDIGOS PRONTOS COM COMPASS

Do que seria o desenvolvimento web se as pessoas guardassem tudo para si, não é mesmo? Grandes projetos *open source*, como o Bootstrap, surgiu dentro de uma empresa (Twitter), mas acabou se desenvolvendo muito mais a partir do momento em que foi colocado no GitHub e dezenas de pessoas passaram a contribuir em seu código.

Outro exemplo é o jQuery, que caiu no gosto da área de front-end. Agora é desenvolvido e mantido por várias pessoas através do GitHub. Um dos motivos para isso é ele conter um jeito mais fácil e rápido de se desenvolver projetos.

Visando seguir os passos do Twitter, poderíamos pegar todo o nosso código do Apeperia, todos os *mixins* e *extends*, e disponibilizarmos na internet, a fim de facilitar a vida de outros desenvolvedores que passem pelos mesmos problemas que nós — como a criação de um *mixins* de borda arredondada, por exemplo. Mas será que fomos nós os primeiros desbravadores de código que precisamos fazer um *image replacement* na vida? Ou uma sombra padrão? Provavelmente não.

Outra pessoa já teve essa ideia. Ela pegou vários códigos Sass e disponibilizou na internet, um framework CSS totalmente baseado em Sass, batizado de **Compass**. Fazendo uma comparação, é como

se fosse um Bootstrap, recheado de códigos prontos.



Figura 9.1: Site do Compass — Junho 2016)

9.1 INSTALANDO O COMPASS E DEIXANDO DE VIGIA

Para podermos usar esses códigos prontos do Compass, é preciso instalá-lo via terminal. Vamos dar os seguintes comandos:

```
gem update --system
```

Mac ou Linux? Não se preocupe, esses comandos funcionam da mesma maneira neles. Já que já instalamos o Ruby no primeiro capítulo

Com esse comando, espera-se a exibição da seguinte mensagem:

```
RubyGems system software updated
```

Agora vamos instalar o Compass:

```
gem install compass
```

Depois, deve aparecer uma mensagem informando que a instalação foi feita. Maravilha!

O Compass é um framework que precisa ser criado na raiz do nosso projeto. Então, navegaremos até a **raiz** do nosso projeto Apeperia via terminal e daremos o seguinte comando:

```
compass create
```

Será que foi? Vamos dar uma olhada na pasta do Apeperia:

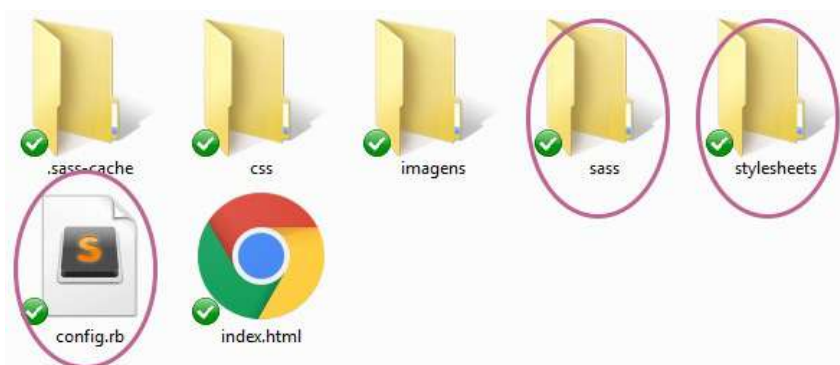


Figura 9.2: Pastas e arquivo criados

Excelente, esse arquivo `config.rb` provavelmente é algo relacionado à configuração do Compass. Vamos abri-lo em nosso editor de texto:

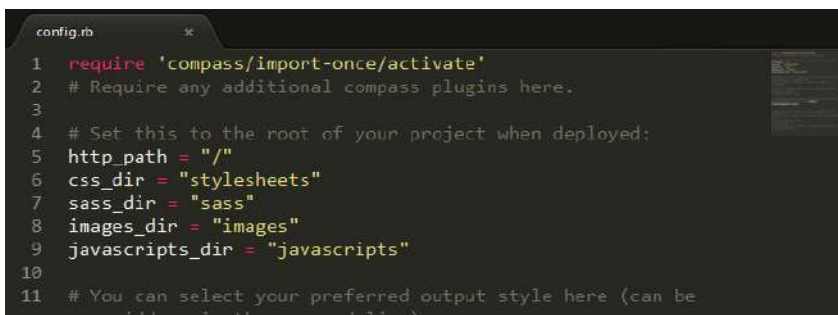


Figura 9.3: Pastas e arquivo criados

O Compass criou a pasta `stylesheets` e `sass` em nosso projeto, pois são esses nomes que vem por padrão. Vamos alterá-los para deixar no mesmo padrão que estamos utilizando:

```
http_path = "/"
css_dir = "css"
sass_dir = "css"
images_dir = "imagens"
javascripts_dir = "js"
```

Bacana. Mas será que compila? Da mesma forma que tínhamos aquele `sass --watch...`, nós temos um comando para falar para o Compass ficar de olho em qualquer alteração em nosso arquivo `estilos.scss`. Daremos o seguinte comando para isso:

```
compass watch css/estilos.scss
```

Com isso, o terminal informará que o Compass está de prontidão para mudanças, bingo! Vamos fazer uma alteração qualquer para garantir que está tudo funcionando como antes. Alteremos no arquivo `variaveis.scss` o valor da variável `$ponto-de-quebra` para **945px**:

```
$ponto-de-quebra: 945px;
```

Salvando, podemos verificar no terminal que o Compass notou a mudança e compilou o CSS sem problemas!

9.2 LIMPANDO UM POUCO A SUJEIRA

Não sei se você reparou, mas quando o Compass virou nosso vigia e compilou o CSS, ele colocou diversos comentários, um em cada regra! É interessante comentar o código, mas assim já é demais!

```
1  /* line 7, helpers/mixins.scss */
2  .plataformas li, .social li a {
3      text-indent: -9999px;
4      overflow: hidden;
5      background-repeat: no-repeat;
6  }
7
8  /* line 13, helpers/mixins.scss */
9  .destaque button, .plano button, .contato button {
10     -webkit-box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);
11     box-shadow: 0 2px 6.65px 0.35px rgba(0, 0, 0, 0.3);
12 }
13
```

Figura 9.4: Pastas e arquivo criados

O Compass faz isso por padrão. Ele já vem configurado dessa maneira. Para mudarmos isso, basta alterarmos seu arquivo de configuração, o `config.rb`. Repare no texto nas linhas 17 e 18:

```
# To disable debugging comments that display the original location
# of your selectors. Uncomment:
# line_comments = false
```

Achamos! Para que a linha 18 deixe de ser um comentário, basta removermos o caractere `#`:

```
line_comments = false
```

Vamos fazer mais uma pequena alteração para o Compass identificar uma mudança e compilar o CSS novamente. No arquivo `variaveis.scss`, vamos voltar o valor da variável `$ponto-de-quebra` para **950px**:

```
$ponto-de-quebra: 950px;
```


Salvando e verificando no terminal... Nada! Isso acontece pois sempre que o `config.rb` for alterado, precisamos rodar o `compass watch` novamente. Vamos encerrar o atual `watch` dando o atalho `Ctrl+C` no terminal, e rodando o comando de `watch` novamente:

```
compass watch css/estilos.scss
```

No `variaveis.scss`, vamos de novo alterar o valor da variável que foi escolhida por nós como teste, a `$ponto-de-quebra`:

```
$ponto-de-quebra: 945px;
```

Feito isso, o CSS deve ser compilado, mas agora sem os comentários.

9.3 ABRINDO A CAIXA DE FERRAMENTAS

Em nosso arquivo `mixins.scss`, criamos um *placeholder* chamado sombra padrão, com a propriedade `box-shadow` e tudo mais. Mas será que colocamos todos os prefixos necessários? Será que não é necessário colocar o prefixo `-moz-` ou o `-o-`?

Em vez de usarmos um `box-shadow` nosso, podemos utilizar um pronto, do Compass. Como são várias pessoas que contribuem com o Compass, a chance de o código ser mais assertivo é um pouco maior.

Para conseguirmos usar os códigos CSS do Compass, precisamos primeiramente importar suas bibliotecas. A biblioteca que precisamos para usar o `box-shadow` é a **css3**.

OUTRA CAIXA DE FERRAMENTAS

Além do Compass, há o **Bourbon**, uma outra biblioteca de mixins que se diz mais leve que o Compass. Veja mais em: <http://bourbon.io>.

Para incluí-la, vamos ao arquivo `mixins.scss`. Depois de todo o código já inserido lá, vamos colocar o seguinte:

```
@import "compass/css3";
```

Agora faremos um teste no arquivo `destaque.scss`, e no lugar do *placeholder* da sombra padrão que colocamos no `.destaque button`, vamos colocar o mixin pronto do Compass:

```
.destaque button {  
  ...  
  @include single-box-shadow;  
  font-weight: bold;  
}
```

Conferindo o CSS gerado, lá pela linha 520, veja que de fato não tínhamos colocado um prefixo necessário, prefixo este que serve para versões antigas do Firefox:

```

517
518 .destaque button {
519     margin-top: 1em;
520     background: #c69;
521     border: 0;
522     padding: .6em;
523     font-size: 1.2em;
524     -webkit-border-radius: 20px;
525     border-radius: 20px;
526     -moz-box-shadow: 0px 5px #333333;
527     -webkit-box-shadow: 0px 5px #333333;
528     box-shadow: 0px 5px #333333;
529     font-weight: bold;
530 }
531

```

Figura 9.5: Prefixo -moz- na sombra padrão do Compass

Podemos conferir também no browser. Repare que a sombra foi criada com sucesso, mas está com um aspecto muito duro:



Figura 9.6: Sombra padrão do Compass

Vamos melhorar o seu design deixando-a mais esfumada, com um desfoque maior, algo mais parecido com nosso layout original. Para isso, no `destaque.scss`, basta chamarmos o mixin `box-shadow` do Compass, na regra do botão de destaque. Mixin este que aceita passarmos valores a fim de deixarmos a sombra do nosso jeito.

Vamos passar os mesmos valores que nosso antigo placeholder tinha:

```
.destaque button {
```

```
...
@include box-shadow(black 0 2px 6.65px);
font-weight: bold;
}
```

Conferindo no browser, a sombra já está bem melhor!



Figura 9.7: Sombra do Compass

Agora vamos substituir nosso placeholder `%sombra-padrao` pelo mixin de sombra do Compass em todos os lugares que aplicamos sombra.

No `planos.scss`, no regra do botão:

```
.plano button {
...
@include box-shadow(black 0 2px 6.65px);
margin-left: 2em;
}
```

No `contato.scss`, também na regra do botão:

```
.contato button {
...
@include box-shadow(black 0 2px 6.65px);
}
```

Podemos conferir no browser que está tudo funcionando como deveria. Utilizamos o código pronto do Compass para gerar a sombra.



Figura 9.8: Sombras ok em toda a página

O Compass possui outros códigos prontos para fazer animações, borda arredondada, filtros, flexbox, e por aí vai. Podemos sempre checar quais existem e como usá-los acessando a referência de código do Compass, em <http://compass-style.org>.

9.4 TERCEIRIZANDO GERAÇÃO DE SPRITES

A boa prática diz que solicitar menos arquivos do servidor é interessante, pois aí temos menos *requests*, o que acaba melhorando a performance de nosso site/aplicação. Isso nem sempre é verdade, uma vez que o HTTP/2 já é realidade hoje em dia. Mas, em geral, ainda é recomendado diminuir o número de *requests*.

Uma das soluções de contorno que a área de front-end pensou para diminuir os *requests* foi colocar várias imagens em uma imagem só, a fim de fazer apenas uma requisição no lugar de dezenas. Algo que é semelhante ao que fizemos no capítulo 5. *Organizando a bagunça*. Assim, no CSS fazemos o posicionamento adequado via `background-position`.

Isso é feito no arquivo `planos.scss`. Repare no código que está no final do arquivo:

```
.icone-check, .icone-x {
```

```
background: url(../imagens/sprite.png) no-repeat;
width: 18px;
height: 18px;
}

.icone-check {
background-position: -5px -5px;
}

.icone-x {
background-position: -33px -5px;
}
```

Essas regras são responsáveis por mostrar os ícones da parte de planos:



Aplicativos	3
Manutenções Programadas	20
Helpdesk 24hs	✓
App SEO	✓
Atendente Exclusivo	✗
Sistema Operacional	3

Figura 9.9: Ícones de um dos planos do Apeperia

Juntar várias imagens em uma e mudar seu posicionamento no CSS, visando diminuir as requisições, é a técnica conhecida como **CSS Sprite**. Só que um de seus maiores problemas é o tempo.

É muito trabalhoso fazer e manter um *sprite*. Tanto que, por isso, surgiram ferramentas para ajudar nessa tarefa, como o *Sprite Cow*, no qual você importa a imagem de um *sprite* e ele gera os códigos CSS de forma rápida. Entretanto, ainda temos de gerar a

imagem.

Outra solução seria mandarmos tudo isso para o designer da equipe e ele que se vire com tudo! Mas isso é maldade, não façamos isso, ok? Sério!

O processo para fazer um *sprite* de um *set* de ícones seria algo assim:

1. Separar vários ícones;
2. Abri-los no Photoshop;
3. Colocá-los no mesmo arquivo;
4. Tentar manter uma certa organização nesse arquivo;
5. Salvar a grande imagem com fundo transparente;
6. No CSS, fazer algo como o código anterior;
7. Acertar exatamente a relação entre elemento X posição da imagem;
8. Fazer **tudo** de novo se tiver de atualizar e colocar mais um ícone.

Ufa! Fácil, mas muito trabalhoso. Vamos agilizar todo esse processo colocando a máquina para fazer isso para nós. O Compass se encarregará dessa tarefa!

Vamos dar uma olhada na pasta `imagens` da Apeperia. Lá podemos ver a imagem a seguir, a `sprite.png`, nosso único sprite:

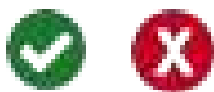


Figura 9.10: Sprite com os ícones de check e de X

Estamos com o CSS usando essa imagem da seguinte maneira, no arquivo `planos.scss` :

```
.icone-check, .icone-x {
background: url(../imagens/sprite.png) no-repeat;
width: 18px;
height: 18px;
}

.icone-check {
background-position: -5px -5px;
}

.icone-x {
background-position: -33px -5px;
}
```

O primeiro passo é isolar os ícones que queremos em nosso novo *sprite* em uma pasta, ainda dentro da pasta de imagens.

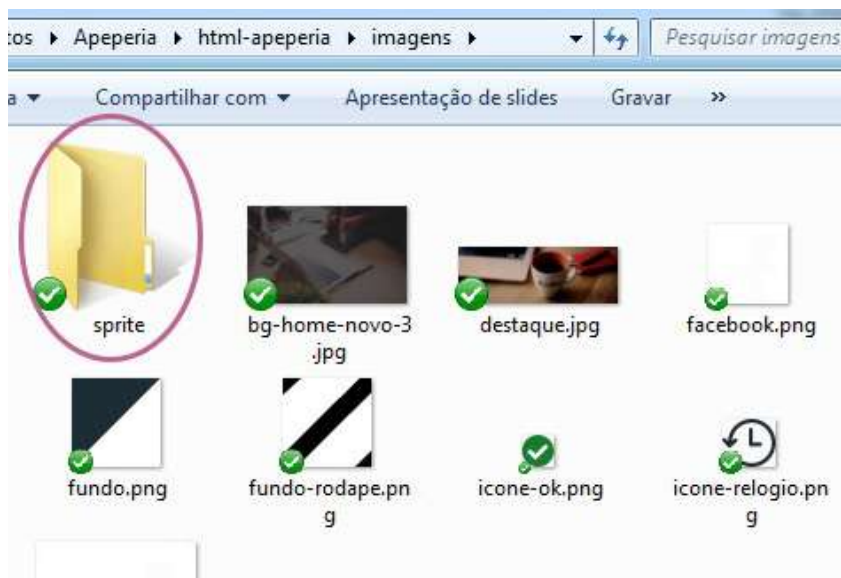


Figura 9.11: Pasta sprite na pasta imagens

Dentro dessa pasta, vão as imagens:

- icone-check.png
- icone-x.png

Agora precisamos dizer para o Compass que, sempre que tiver alguma imagem com a extensão `.png` na pasta `sprite`, é para ser gerada uma nova imagem. Para isso, passamos mais um `import` no arquivo `estilos.scss`, no começo dele:

```
@import "sprite/*.png";  
  
@import "helpers/mixins";  
...
```

Salvando, podemos ver no terminal a seguinte mensagem do Compass:

```
create imagens/sprite-s1e1f3a7a9e.png
```

Olhando na pasta `imagens`, podemos notar que esse arquivo foi criado de fato. Para fazer o nome da imagem, o Compass pegou o nome da nossa pasta (`sprite`) e colocou um *hash*, para evitar problemas com o *cache* do browser.

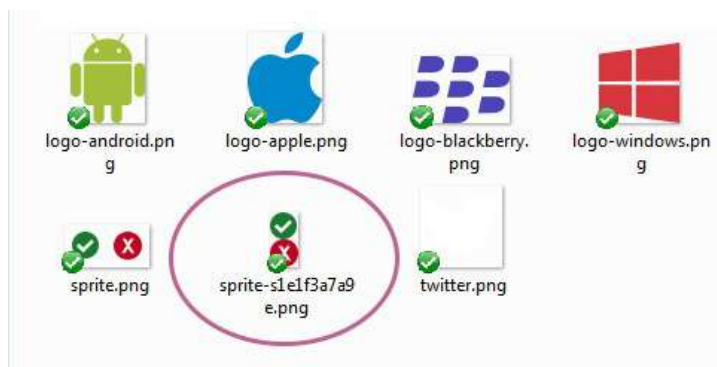


Figura 9.12: Sprite criado

Fácil, não? Hora dos testes! Não pode ser assim tão fácil... Ou pode? Vamos pegar o arquivo `icone-relogio.png` e fazer uma cópia na pasta `sprite`.

E nada acontece! Isso porque o `watch` do Compass só leva em consideração mudanças em código para mudar alguma coisa. Se

fizermos uma alteração qualquer no `estilos.scss` :

```
@import "sprite/*.png";  
// só testando  
  
@import "helpers/mixins";  
...
```

O Compass é esperto suficiente para remover o sprite criado anteriormente, e gerar outro. Veja no terminal:

```
remove imagens/sprite-s1e1f3a7a9e.png  
create imagens/sprite-s9a09e4f7a9.png
```



Figura 9.13: Sprite teste

Só com essa *feature* do Compass já conseguimos agilizar o processo de gerar os sprites em si, eliminando qualquer necessidade de fazer a imagem no Photoshop ou em outro programa de edição de imagens.

Vamos apenas remover o `icone-relogio.png` de dentro da pasta `sprite` para voltar nosso projeto para o estado antes do teste. Podemos remover também o comentário de teste que acabamos de colocar no `estilos.scss`. E já que o sprite está sendo gerado pelo Compass, vamos deletar o arquivo `sprite.png` original que já veio com o projeto na pasta `imagens`.

Outro ponto que podemos reparar é que o sprite gerado está com os ícones muito próximos um do outro. Isso pode dificultar nossa vida no futuro quando formos usar o `background-position` :



Figura 9.14: Ícones muito colados

Podemos configurar o Compass para deixar um espaço entre esses ícones. Para isso, no arquivo `estilos.scss` , basta adicionarmos uma variável indicando esse espaço, antes do `@import 'sprite/*.png';` . Digamos que uns 5px:

```
$sprite-spacing: 5px;
```

Repare que o prefixo dessa variável é justamente o nome da pasta que criamos anteriormente, `sprite` . Conferindo a imagem, podemos ver que o Compass agora deixou o espaço entre os elementos como pedimos:



Figura 9.15: Ícones com espaçamento de 5px

9.5 CONFIGURANDO TUDO PARA TIRARMOS FÉRIAS MAIS CEDO

Gerar sprite automático é uma beleza com o Compass. Agora vamos dar uma olhada no browser para ver como ficou a seção de planos, seção esta que usava o `sprite` :



Figura 9.16: Cadê os ícones?!

Opa! Para onde foram os ícones? Vamos usar o *DevTools* para achar o problema, clicando com o botão direito no local onde um deles apareceria, e então em *Inspecionar elemento*.

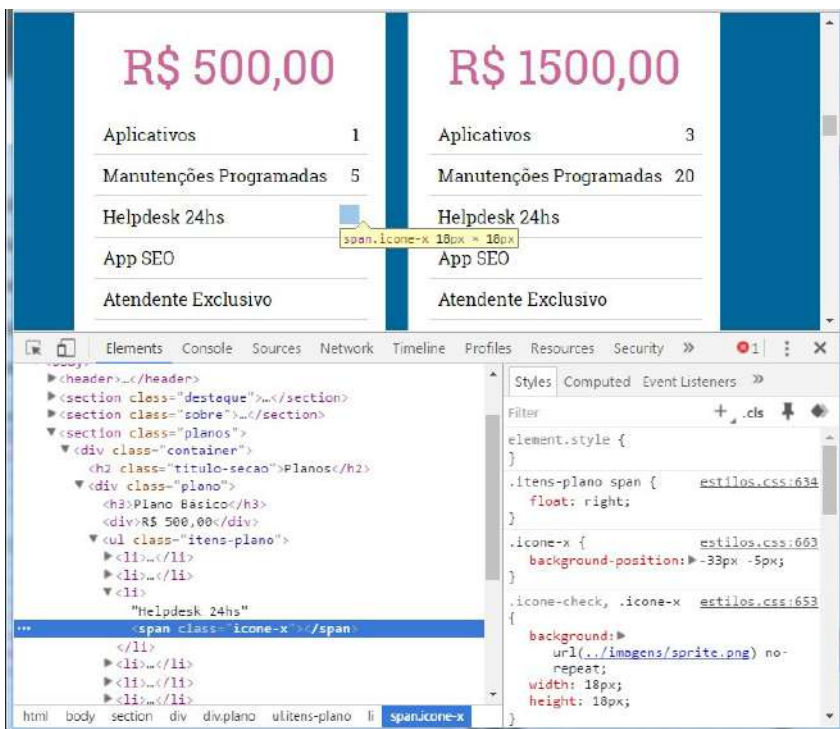


Figura 9.17: Inspeccionando elemento

Perceba no canto inferior direito que ainda estamos chamando como background a imagem antiga do sprite , que deletamos!

```
.icone-check, .icone-x estilos.css:653
{
  background: url(.../imagens/sprite.png) no-repeat;
  width: 18px;
  height: 18px;
}
```

Figura 9.18: Sprite errado

Só um erro honesto de nome de arquivo. Vamos ao arquivo planos.scss para corrigir isso, colocando o caminho para a

imagem correta. Atente-se para pegar exatamente o nome da imagem que foi gerada pelo Compass. No meu caso, fica assim:

```
.icone-check, .icone-x {  
background: url(../imagens/sprite-s1e035a9ecf.png) no-repeat;  
width: 18px;  
height: 18px;  
}
```

E conferindo no browser:

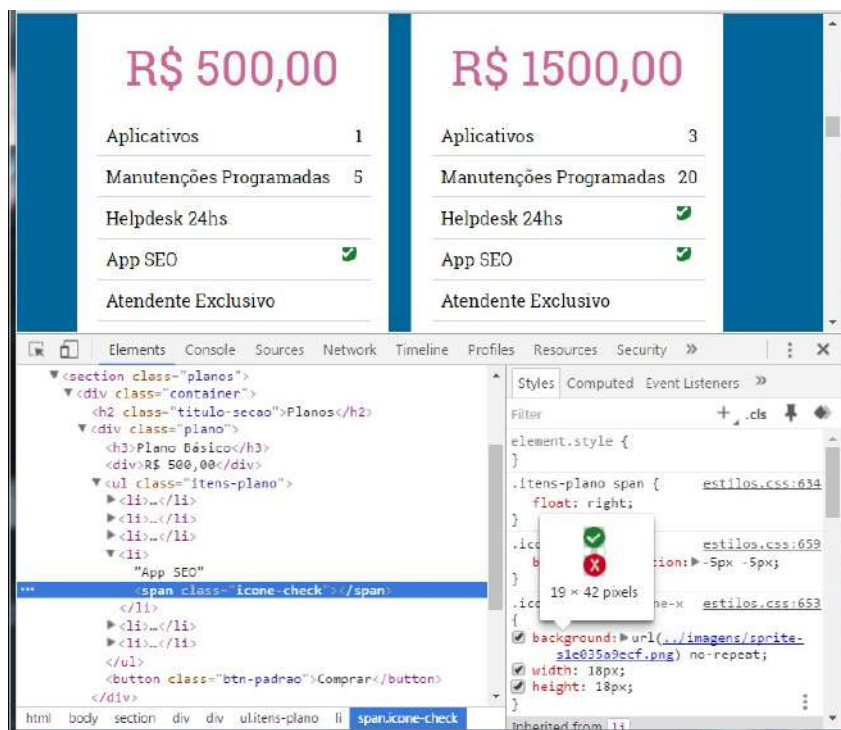


Figura 9.19: Sprite certo, posições erradas

Bom, pelo menos está puxando a imagem. Aparentemente, é uma questão de ajuste no posicionamento! Temos no mesmo arquivo regras que mexem com o `background-position`! Mas imagina ter de refazer isso sempre que alterarmos o sprite? Sempre que for gerado um sprite novo ter de:

- Copiar o novo nome da imagem;
- Colar no CSS;
- Mudar o posicionamento de cada `background` .

Ia ser legal se o Compass se encarregasse de todo esse processo. Sabemos que ele consegue gerar a imagem, mas agora é preciso fazer os ajustes de nomes de arquivo e posicionamento de `background` . Para que o Compass faça isso para nós, vamos incluir o mixin a seguir no lugar da propriedade `background` :

```
.icone-check, .icone-x {
@include all-NomeDaPasta-sprites;
width: 18px;
height: 18px;
}
```

`NomeDaPasta` sempre será o nome da pasta que isolamos os ícones. No nosso caso, é a pasta `sprite` , ficando desta maneira:

```
.icone-check, .icone-x {
@include all-sprite-sprites;
width: 18px;
height: 18px;
}
```

Como o Compass que vai calcular o posicionamento dos *backgrounds* dos ícones para nós, vamos **remover** as linhas a seguir:

```
.icone-check {
background-position: -5px -5px;
}

.icone-x {
background-position: -33px -5px;
}
```

E removeremos também as medidas de largura e altura também, deixando o código desta forma, somente com o mixin que gera o código do CSS *sprite*:

```
.icone-check, .icone-x {
@include all-sprite-sprites;
```

```
}
```

Abrindo o CSS compilado, o `estilos.css`, lá pela linha 650, repare que o Compass criou regras novas, que não estavam em nosso arquivo original:

```
.icone-check .sprite-icone-check, .icone-x .sprite-icone-check {  
  background-position: 0 0;  
}  
.icone-check .sprite-icone-x, .icone-x .sprite-icone-x {  
  background-position: 0 -23px;  
}
```

Se checarmos o tamanho de cada ícone que está na pasta `sprite`, veremos que a largura e a altura bateram certinho com o que o Compass gerou! Mas esses seletores estão meio estranhos. Como assim `.icone-check .sprite-icone-check`? Elemento com a classe `.sprite-icone-check` que é filho de um elemento com a classe `.icone-check`? Não queríamos que o seletor fosse criado dessa forma.

Podemos consertar isso colocando o *mixin* que criamos **fora** da regra dos ícones, mudando isso no arquivo `planos.scss`:

```
@include all-sprite-sprites(true);  
  
.icone-check, .icone-x {  
  //  
}
```

Como nessa regra (`.icone-check, .icone-x {}`) não há nenhuma declaração CSS, podemos removê-la. Agora o *mixin* está sendo incluído no **final** de **um** dos muitos arquivos que temos. Vamos melhorar a organização por aqui e movê-lo para o arquivo `estilos.scss`:

```
$sprite-spacing: 5px;  
@import "sprite/*.png";  
@include all-sprite-sprites;  
  
@import "helpers/mixins";  
...
```


Agora conferindo no `estilos.css` , desta vez no topo do documento:

```
.sprite-sprite, .sprite-icone-check, .sprite-icone-x {
background-image: url('/imagens/sprite-s1e035a9ecf.png');
background-repeat: no-repeat;
}

.sprite-icone-check {
background-position: 0 0;
}

.sprite-icone-x {
background-position: 0 -23px;
}
```

Problemas... Onde está a largura e a altura nessas regras? Precisamos delas para que o Sprite CSS funcione, como estava anteriormente.

Para resolver isso, basta passarmos um parâmetro no *include* que fizemos há pouco no `estilos.scss` , desta maneira:

```
$sprite-spacing: 5px;
@import "sprite/*.png";
@include all-sprite-sprites(true);

@import "helpers/mixins";
```

Checando o `estilos.css` , repare que agora o Compass está pegando a altura e largura individual dos ícones:

```
.sprite-sprite, .sprite-icone-check, .sprite-icone-x {
background-image: url('/imagens/sprite-s8ca5edb2ee.png');
background-repeat: no-repeat;
}

.sprite-icone-check {
background-position: 0 0;
height: 18px;
width: 18px;
}

.sprite-icone-x {
background-position: 0 -23px;
```

```
height: 19px;  
width: 19px;  
}
```

Agora, será que o Compass já mudou o nome das classes no HTML? Infelizmente não. Esse trabalho fica por nossa conta.

Os nomes das classes desse código foram baseados no nome da nossa pasta (`sprite`) e nos arquivos que ali estão (`icone-check.png` e `icone-x.png`). Logo, precisamos atualizar em nosso `index.html` os nomes das classes dos elementos `span` , que estão dentro da seção de Planos. Onde há a classe `icone-x` , trocaremos pela classe `sprite-icone-x` , e o mesmo com a classe `icone-check` , mudando-a para `sprite-icone-check` . Vejo o exemplo a seguir:

```
<li>  
Helpdesk 24hs  
<span class="sprite-icone-x"></span>  
</li>
```

Feito isso, há dois possíveis cenários em sua máquina agora:

- os ícones apareceram;
- os ícones não apareceram;

Isso basicamente depende de como você está testando o projeto. Se estiver apenas abrindo o HTML, os ícones **não** apareceram.

Se tiver testando via *localhost*, ou com o *live preview* do editor Brackets por exemplo, estará tudo ok. Se esse for este seu caso, pule para o resumo deste capítulo.

Agora vamos resolver o problema para nós que abrimos direto o `index.html` no browser, sem *localhost*.

Dando *botão direito* > *Inspecionar elemento* onde ficaria um dos ícones, repare no caminho do *background*:

```
file:///imagens/sprite-s8ca5edb2ee.png
```

Agora repare na URL do `background` da primeira regra do CSS gerado, o nosso `estilos.css` :

```
.sprite-sprite, .sprite-icone-check, .sprite-icone-x {  
background-image: url('/imagens/sprite-s8ca5edb2ee.png');  
background-repeat: no-repeat;  
}
```

Podemos perceber que o Compass gerou um endereço relativo ao *localhost*, e não à pasta CSS. Para alterarmos esse caminho, vamos abrir novamente o arquivo de configuração do Compass, o `config.rb`, localizado na raiz de nosso projeto.

Agora na quinta linha, na configuração do `http_path`, basta fazermos a alteração para que o Compass gere os endereços voltando uma pasta, desta forma:

```
http_path = "../"
```

Não esqueça que para a configuração começar a valer precisamos resetar o *watch* do Compass, no terminal. Para isso, vá ao terminal e dê o comando `Ctrl + C` e finalize o processo.

Agora basta rodar o *watch* do Compass novamente:

```
compass watch css/estilos.scss
```

Feito isso, nosso site já volta ao normal:



Figura 9.20: Sprite finalmente ok

9.6 RESUMO

Neste capítulo, vimos o Compass, uma ferramenta que traz vários mixins prontos a fim de agilizarmos nosso trabalho com Sass. Também usamos um desses mixins prontos para fazer a sombra padrão que está sendo usada em todos os botões do Apeperia.

Quando o Compass gera o CSS, ele deixa o CSS com vários comentários. Logo, aprendemos que podemos removê-los facilmente apenas mexendo no `config.rb`, arquivo de configuração do Compass, que fica sempre na raiz do projeto.

E acredito que o *climax* do capítulo foi quando geramos um CSS *Sprite* de forma automática, tanto a geração da imagem em si (tchau, Photoshop!) quanto a posição dos `background-position`, que também foi feita automaticamente. Também demos uma pequena arrumada na `.png` que ele gera, aumentando o espaço entre as

imagens, apenas usando a variável `$sprite-spacing` .

A seguir, finalizaremos o Apeperia com uma calculadora.

CALCULANDO E RETORNANDO VALORES

Quando sai um modelo de celular novo da marca que gostamos, com algum recurso que não conhecíamos até então (mas pelos comerciais parece que mudará nossa vida), o que pensamos? *"Preciso desse iNexus Phone novo!"*. É bem natural quisermos experimentar coisas novas, sejam elas coisas ligeiramente supérfluas ou sensações diferentes em uma viagem para o continente asiático.

Uma coisa de que todo desenvolvedor front-end gosta é de novidades, seja esta uma ferramenta de edição de texto nova ou alguma propriedade do CSS. Por exemplo, em nosso arquivo `footer.scss`, aproximadamente na linha 30, nosso CSS veio com uma unidade de medida introduzida com o CSS3, a *rem*.

A ideia dessa unidade é parecida com a unidade *em*, que pega o tamanho do `font-size` do elemento pai. A *rem* é relativa ao pai de todos em nosso documento HTML, a própria tag HTML, que normalmente tem o valor de 16px.

Um truque conhecido para ajudar na aplicação tanto da *em* quanto da *rem* em nossos CSSs é colocar o `font-size` da tag HTML, dessa maneira:

```
html {  
  font-size: 62.5%;  
}
```

Então, nos outros elementos, a razão "EM *versus* PX" ficava mais fácil de ser aplicada:

```
h1 {
  font-size: 1.5rem; /* = 15px */
}

.menu-opcoes li {
  width: 2.6rem; /* = 26px */
}
```

Simples! Vamos brincar um pouco e atualizar nosso código! No arquivo `footer.scss`, na regra `footer .container`, mudemos para *rem* as unidades utilizadas:

```
footer .container {
  padding-top: 3rem;
  height: 6.5rem;
  position: relative;
}
```

Sensação boa de usar coisa nova, não? E ainda com essa alteração, não dependemos mais do `font-size` do pai direto desse *container* — diferente da antiga *em*.

Feita a alteração, jogamos para produção, e quinze minutos depois o chefe liga falando que o site está com o rodapé desconfigurado! Mas só alteramos a unidade de medida utilizada! O que pode ter sido? Dando uma pesquisa no *Can I Use* para ver o suporte da *rem*:

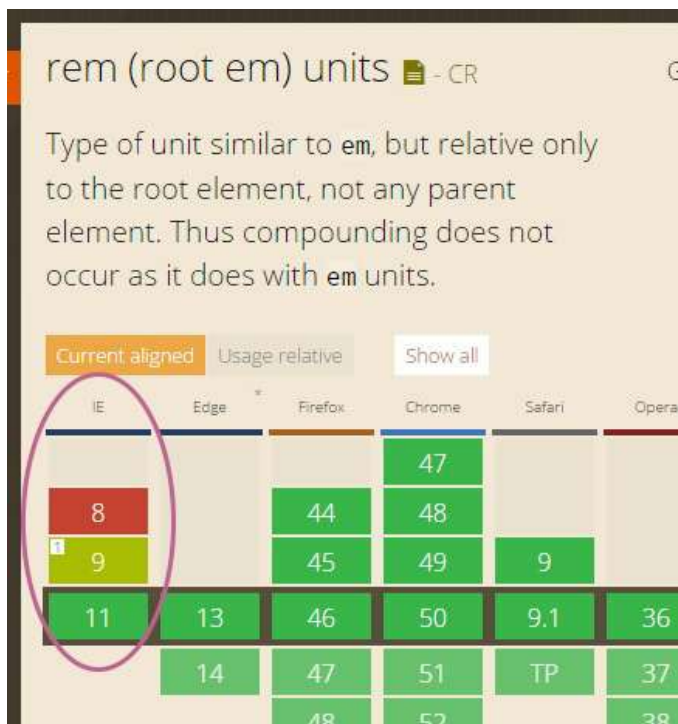


Figura 10.1: Can I Use — rem

Elementar, meu caro leitor. Podemos supor que nosso querido chefe está usando o IE 8. Como ele é gente boa, podemos explicar que ninguém mais acessa a internet nessa carroça, e que seria interessante ele atualizar o browser (cuidado ao dizer isso ao seu chefe).

Nisso, o chefe atualiza e fica tudo normal, missão cumprida! Mas será que de fato os usuários do Apeperia não acessam o site através do IE 8? Pedindo o relatório sobre isso para o Analista SEO do Apeperia, observamos que 10% dos usuários ainda utilizam o IE 8... Ah, não!

Precisamos dar suporte para eles. 10% é algo bem relevante, mas usar a *rem* não é possível. A solução seria convertê-la de alguma

forma para pixels, unidade esta que qualquer browser entende. Ainda bem que todo sistema operacional tem uma calculadora, não é mesmo?

10.1 FAZENDO O TRABALHO DA CALCULADORA

Como dito há pouco, essa versão do IE, a 8, não entende a *rem*. Vamos convertê-la! Considerando que a grande maioria dos browsers coloca **16px** de tamanho de fonte padrão, basta fazermos a conta na calculadora.

Só para lembrar, temos esse código no `footer.scss` :

```
footer .container {  
  padding-top: 3rem;  
  height: 6.5rem;  
  position: relative;  
}
```

Se queremos três vezes o tamanho da fonte do HTML, que tem 16px, no `padding-top` :



Figura 10.2: Calculadora com o valor 48

Agora é só ficar fazendo a conta na calculadora (ou de cabeça para os mais matemáticos) e fica tudo bem. Um minuto... Ficar fazendo conta, sério? Ia ser legal se o Sass já fizesse isso para nós, afinal, é só uma continha que qualquer calculadora de R\$ 1,00 consegue fazer. Pois bem, ele consegue fazer isso para nós.

Por conta do SassScript, linguagem de script do Sass, conseguimos realizar operações aritméticas, inclusive criar funções e condicionais. No mesmo exemplo anterior, se queremos que o Sass faça uma conta, basta falarmos qual conta que queremos:

```
footer .container {  
  padding-top: 3 * 16  
  height: 6.5 * 16;  
  ...  
}
```

Vendo o CSS compilado:

```
footer .container {
```

```
padding-top: 48;
height: 104;
...
}
```

Opa! Mas altura de 104 o quê? 104mm? 104%? Temos de falar a unidade que queremos, se não o Sass não coloca nenhuma. Veja:

```
footer .container {
padding-top: 3 * 16px;
height: 6.5px * 16;
...
}
```

Não importa em qual número colocamos a unidade de medida, desde que ela exista. Olhando o CSS compilado, chegamos ao resultado esperado:

```
footer .container {
padding-top: 48px;
height: 104px;
...
}
```

Tudo funcionando! Até chegar o dia que precisemos mudar o tamanho da fonte do HTML. Aí teremos de lembrar de alterar essa conta, e em qual (ou quais) arquivo ela existe. Sempre que repetimos muito um valor, o que fazemos? Criamos uma variável. Basta criá-la na mesma regra que vimos há pouco e fazer seu uso:

```
footer .container {
$tamanho-da-fonte-padrao: 16px;
padding-top: 3 * $tamanho-da-fonte-padrao;
height: 6.5 * $tamanho-da-fonte-padrao;
...
}
```

Agora basta fazermos as alterações no `footer.scss` :

```
footer .container {
$tamanho-da-fonte-padrao: 16px;
padding-top: 3 * $tamanho-da-fonte-padrao;
height: 6.5 * $tamanho-da-fonte-padrao;
...
}
```

```
}
```

Pronto! Podemos conferir o resultado no `estilos.scss` :

```
footer .container {  
  padding-top: 48px;  
  height: 104px;  
  position: relative;  
}
```

10.2 NÃO CONSIGO USAR O QUE NÃO TENHO

Vimos na parte anterior que o Sass nos permite utilizar contas. Outro local interessante para convertermos nosso código e deixar o Sass fazer o cálculo para nós é no arquivo `planos.scss` . Logo no seu começo, temos esta regra:

```
.plano {  
  background: white;  
  width: 18em;  
  display: inline-block;  
  margin: 1em 0 0 1.1em;  
  padding-bottom: 2em;  
}
```

Essa `width` , como nas outras propriedades, poderia ter sua unidade de medida alterada para que seja sempre relacionada ao tamanho da fonte do HTML. Vamos focar na largura. Para facilitar tudo, podemos usar a variável que criamos anteriormente:

```
.plano {  
  background: white;  
  width: 18 * $tamanho-da-fonte-padrao;  
  ...  
}
```

Olhando no terminal, temos um erro:

```
error css/estilos.scss (Line 12 of css/layout/planos.scss: Undefined  
variable: "$tamanho-da-fonte-padrao".)
```

Como assim variável indefinida, sendo que a definimos? Nós a criamos no... `footer.scss` . Será que é isso? Vamos mover a

criação da variável `$tamanho-da-fonte-padrao` para o `variaveis.scss` para testar:

```
$tamanho-da-fonte-padrao: 16px;
```

E a regra do `footer .container`, no `footer.scss`, deixemos desta forma:

```
footer .container {  
  padding-top: 3 * $tamanho-da-fonte-padrao;  
  height: 6.5 * $tamanho-da-fonte-padrao;  
  position: relative;  
}
```

Conferindo no terminal, podemos ver que tudo corre bem. O que acontece é que, semelhante a linguagens de programação (como o JavaScript, por exemplo), o Sass possui o conceito de escopo. Ou seja, uma variável criada em um local específico só pode ser utilizada naquele mesmo local.

Para resolver nosso problema e permitir que qualquer arquivo use a variável, deixamos ela direto no `variaveis.scss`, tornando-a global/pública. Essa ideia de escopo vale tanto para arquivos diferentes quanto para regras diferentes. Se mantivéssemos a criação da variável na regra `footer .container`, ela só existiria ali dentro daquele escopo específico, podendo ser usada em declarações dessa mesma regra CSS.

10.3 RETORNANDO COISAS

Se um dia o nome da variável `$tamanho-da-fonte-padrao` mudar, teríamos de mudar em vários locais diferentes. Só no `footer.scss` seriam dois lugares! Estamos repetindo muito a variável.

Outro ponto é se precisarmos mudar a operação matemática do cálculo de largura, que é realizada no arquivo `planos.scss`, na

```
regra .plano :
```

```
.plano {  
  background: white;  
  width: 18 * $tamanho-da-fonte-padrao;  
  ...  
}
```

Precisaríamos fazer a mudança nessa regra e, caso estivéssemos usando em outros lugares, teríamos se fazer a mudança neles também. Se temos muito código repetido, podemos usar o quê? Um mixin! É uma solução para deixar isso mais fácil. Vamos criar então um mixin que retorne a largura para nós.

No arquivo `mixins.scss`, vamos adicionar o mixin seguinte, no final do arquivo:

```
@mixin retorna-largura {  
  width: 18 * $tamanho-da-fonte-padrao;  
}
```

Agora basta incluirmos o mixin no `planos.scss`, na regra do `.plano`:

```
.plano {  
  background: white;  
  @include retorna-largura;  
  display: inline-block;  
  ...  
}
```

Funcional! Agora podemos reutilizar esse mixin em outros lugares que precisem do cálculo da largura. Mas sempre com os **mesmos 18** usados no cálculo.

Uma abordagem mais interessante para deixar esse código mais independente seria passar o valor desse multiplicador na hora da inclusão do mixin, aí quem o chama que fica encarregado de passar o valor. Em `mixins.scss`:

```
@mixin retorna-largura($multiplicador) {  
  width: $multiplicador * $tamanho-da-fonte-padrao;  
}
```

```
}
```

Agora de volta no `planos.scss`, precisamos passar o valor que precisamos, **18** em nosso caso:

```
.plano {  
  background: white;  
  @include retorna-largura(18);  
  display: inline-block;  
  ...  
}
```

Tudo funcionando! Agora podemos usar esse mesmo mixin em outros lugares, como no `footer.scss`, apenas ajustando o valor passado:

```
footer .container {  
  @include retorna-largura(3);  
  height: 6.5 * $tamanho-da-fonte-padrao;  
  position: relative;  
}
```

Testando no browser:



Figura 10.3: Footer bugado

Opa, o que houve? Bom, na regra `footer .container` original, não precisávamos retornar a largura, e sim o `padding-top`. Nosso mixin está muito atrelado ao fato de ele só retornar uma largura, pois essa é a função dele.

Uma solução seria criar **n** mixins para retornar **todas** as propriedades que usamos, algo como `retorna-padding-top` ou `retorna-largura`. Porém, isso levaria tempo e o código não

ficaria muito bonito, não é mesmo?

A única coisa que precisamos fazer é pegar o multiplicador e multiplicar pelo tamanho da fonte padrão, certo? Então, em vez de um mixin, podemos criar um pedaço de código que faça essa conta para nós e retorne um resultado. Esse pedaço de código é o que chamamos de **função**. Para isso, primeiramente voltemos o `footer .container` para seu original:

```
footer .container {
  $tamanho-da-fonte-padrao: 16px;
  padding-top: 3 * $tamanho-da-fonte-padrao;
  height: 6.5 * $tamanho-da-fonte-padrao;
  ...
}
```

Agora no `mixins.scss`, basta alterarmos o mixin `retorna-largura` feito anteriormente pelo código a seguir:

```
@function retorna-largura($multiplicador) {
  width: $multiplicador * $tamanho-da-fonte-padrao;
}
```

Nesse mesmo código, não queremos mais que ele calcule a largura, queremos que ele simplesmente **retorne** algum resultado:

```
@function retorna-largura($multiplicador) {
  @return $multiplicador * $tamanho-da-fonte-padrao;
}
```

A ideia é que essa função faça um cálculo baseando-se no tamanho da fonte. O entendimento de seu nome pode ser melhorado da seguinte forma:

```
@function multiplica-pela-fonte($multiplicador) {
  @return $multiplicador * $tamanho-da-fonte-padrao;
}
```

Basta invocarmos essa função nos lugares que vimos anteriormente, passando também o respectivo valor como parâmetro, como no arquivo `.planos.scss`:


```
.plano {
  background: white;
  width: multiplica-pela-fonte(18);
  display: inline-block;
  margin: 1em 0 0 1.1em;
  padding-bottom: 2em;
}
```

E no arquivo `footer.scss`, na regra `footer .container`:

```
footer .container {
  padding-top: multiplica-pela-fonte(3);
  height: multiplica-pela-fonte(6.5);
  position: relative;
}
```

Agora sim! Podemos conferir no browser que está tudo ok com os planos e o rodapé também:



Figura 10.4: Planos ok

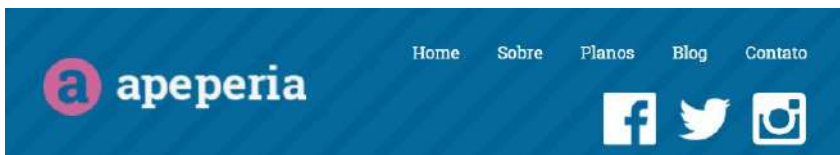


Figura 10.5: Footer ok

10.4 ARREDONDANDO A GALERA

Nossa função `multiplica-pela-fonte` está funcional, mas está um pouco frágil. Caso o desenvolvedor passe um valor quebrado na chamada dessa função, o resultado poderá ficar quebrado também. Testemos isso alterando o `planos.scss` :

```
.plano {  
  background: white;  
  width: multiplica-pela-fonte(18.73);  
  ...  
}
```

No CSS compilado, resultará no seguinte resultado:

```
.plano {  
  background: white;  
  width: 299.68px;  
  ...  
}
```

Uma solução interessante seria se essa função já retornasse o valor já **arredondado**, para evitar esses valores quebrados no CSS que vai para produção. Isso é possível usando uma das funções que o Sass possui, a que arredonda valores.

Para isso, em `planos.scss` , basta fazermos uso da função `round` :

```
.plano {  
  background: white;  
  width: round(multiplica-pela-fonte(18.73));  
  ...  
}
```

Conferindo no CSS gerado, podemos ver que o valor da largura obtém como resultado o valor de **300px**. Ou seja, a função `round` arredonda o resultado para o número inteiro próximo. Funciona!

OUTRAS FUNÇÕES

Se você quer dar uma olhada em mais algumas funções que o Sass possui, acesse: <http://sass-lang.com/documentation/Sass/Script/Functions.html>.

Agora é só adicionarmos o `round` em todos os lugares que usam a função. Mas isso poderia dar uma trabalhadeira! Onde que é feito todo o cálculo de multiplicar pela fonte? Não é justamente na criação da função? Podemos passar o `round` lá mesmo. Para isso, primeiramente voltemos o `planos.scss` para seu estado anterior:

```
.plano {  
  background: white;  
  width: multiplica-pela-fonte(18.73);  
  ...  
}
```

E agora em `mixins.scss` :

```
@function multiplica-pela-fonte($multiplicador) {  
  @return round($multiplicador * $tamanho-da-fonte-padrao);  
}
```

Feito isso, nossa função está um pouco mais protegida. Outros desenvolvedores da nossa equipe agora podem passar valores quebrados, mas o CSS final ficará com o valor arredondado.

10.5 RESUMO

Vimos neste capítulo que podemos calcular valores, graças à linguagem de script do Sass. Resolvemos uma incompatibilidade com navegadores antigos, transformando uma unidade de medida em outra.

Pegamos o cálculo de multiplicação pela fonte e, primeiramente, fizemos um mixin, para logo em seguida transformá-lo em uma função. Ou seja, um trecho de código isolado que faz a conta, e só precisa retornar um valor.

Vimos também que, da mesma forma que outras linguagens, temos de tomar cuidado onde criamos as variáveis que usamos no nosso código, pois o conceito de escopo é presente no Sass. O local no qual a variável é criada influencia diretamente em seu uso. Vimos também uma função pronta do Sass, que arredonda valores, a `round`.

Agora vamos para a despedida, com algumas dicas e conselhos sobre Sass!

CONSELHOS FINAIS

Durante todo este livro, espero que você tenha aprendido Sass e consiga inserir seu uso no seu *workflow*. Espero também que tenha tido curiosidade de conhecer outros pré-processadores e tenha curtido o livro.

Meu conselho é que conheça o LESS e o Stylus também, e decida qual você acha mais interessante. Mesmo porque, se você domina bem o Sass, você automaticamente sabe 80% de qualquer outro.

Talvez algum colega desenvolvedor seu diga que Sass é um lixo, e que você deveria aprender LESS ou Stylus. Infelizmente, ferramentas despertam um sentimento de "defendo o que gosto" mais do que deveriam. Mas no fim, são apenas ferramentas.

Seu usuário e seu chefe não se importam nem um pouco com qual você vai escolher. Eles querem é o resultado, o trabalho concluído, a geração de valor.

Cito aqui um ponto pró-Sass:

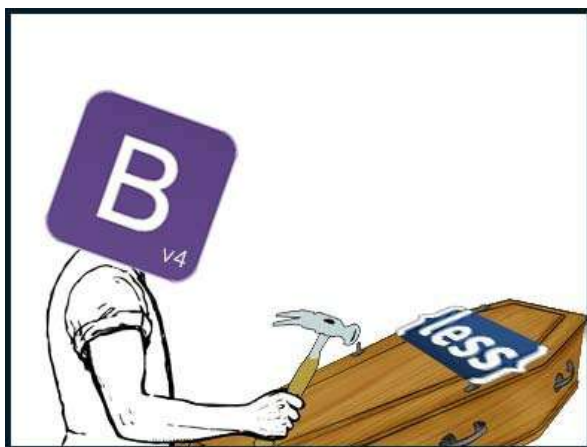


Figura 11.1: Bootstrap v4

O Bootstrap na versão 4, atualmente em Alpha, trocou o LESS pelo Sass. Quando anunciaram essa troca, muita gente comentou que esse seria o último prego no caixão do LESS. Acredito que concorrência é sempre bom para todos nós, então espero que o LESS continue por aí por muitos anos, junto com o Stylus, e que apareçam várias outras ferramentas que nos ajudem.

Sass é uma poderosíssima ferramenta e, com tamanho poder, vem uma grande responsabilidade. Cuidado! Apesar de facilitar a manutenção, não é porque você está utilizando Sass em seus projetos que seu CSS ficará mais rápido ou mais performático. Esses itens são de responsabilidades nossa, e não das ferramentas que utilizamos.

O Sass, sozinho, não faz seu CSS melhor. Entretanto, deixa sua vida muito mais prática! Podemos nos preocupar menos com questões burocráticas, como manutenção de código e trabalhos manuais. E isso acaba nos dando mais tempo para pensar em resolver problemas, fazer testes, enfim, sermos mais eficientes em nossos trabalhos.

As boas práticas que aprendemos e aplicamos no CSS comum (*vanilla*) também são válidas no Sass. E isso vale para qualquer outro pré-processador, seja o LESS, Stylus ou qualquer outro que vir pela frente.

Muitas pessoas possuem aversão a pré-processadores, e um dos argumentos é que eles deixam o CSS com muito lixo, pesado etc. Mas quando o CSS comum fica mal organizado, você joga a culpa no editor de texto? O Sass é apenas uma ferramenta, não terceirize a responsabilidade de um código ruim para ele, ou para qualquer outra ferramenta. O que vai fazer a diferença entre um bom e um mau código é **você**.

Algo que aplicamos no livro inteiro é verificar o CSS compilado, o que considero uma boa prática, principalmente nos primeiros contatos com o Sass. Às vezes, podemos ter "comido bola" com algo, e o CSS acabou sendo gerado com algum defeito ou problema de performance. Defeito este que pode ter uma resolução prática e simples, caso entendamos de fato como o Sass funciona.

Os mixins são uma excelente ajuda na hora da manutenção. Só tome bastante cuidado para não poluir seu código com mixins desnecessários.

Por exemplo, se você possui algumas declarações que são chamadas em várias regras CSS, avalie a criação de um **extend**. Agora, caso os valores nessas declarações mudem muito — ou seja, se é necessário passar valores individuais —, o uso do mixin torna-se passível de ser necessário.

Com relação ao aninhamento de seletores, vá com calma. Ele é bem bacana, mas deve ser usado com cuidado. Você pode acabar quebrando a especificidade do seu CSS e comprometendo a reusabilidade desses seletores, e vai ter de apelar para o `!important`.

É comum termos de compilar o Sass, concatenar arquivos, otimizar imagens, minificar arquivos etc. Existem ferramentas para isso, pois fazer manualmente levaria muito tempo. Por isso que existem **automatizadores de tarefa front-end**, ou *task runners*, como o Gulp e o Grunt.

Você basicamente escolhe um deles e o configura para realizar tarefas à distância de um comando. Há até plugins para colocar seu projeto em repositórios remotos no GitHub. Se quiser já fazer com que o próprio Sass minifique seu arquivo, basta dar o comando:

```
sass --watch estilos.scss:/estilos.css --style compressed
```

TASK RUNNERS

As ferramentas mais usadas desse tipo atualmente são o Gulp (<http://gulpjs.com>) e o Grunt (<http://gruntjs.com/>).

11.1 QUAL O MELHOR PRÉ-PROCESSADOR?

Essa pergunta é bem polêmica e os alunos do curso presencial de front-end na Caelum sempre a levantam. Tenho experiência prática com o Sass e o LESS, mas prefiro o primeiro por gostar mais da sintaxe.

Por exemplo, enquanto no Sass para criar um mixin você usa o `@mixin` e o inclui dando um `@include`, no LESS você cria uma classe CSS comum e a chama lá no meio da sua regra CSS. Você pode comparar as features individuais dos pré-processadores mais usados atualmente aqui: <https://csspre.com/compare>.

11.2 OUTRAS FEATURES

Apesar de não terem sido necessárias em nosso projeto que desenvolvemos ao longo do livro, o Sass possui outras features interessantes que valem a pena serem mencionadas aqui. Recomendo uma leitura complementar, da própria documentação (<http://sass-lang.com>), dos seguintes itens:

- Controladores de fluxo (`@if` , `@else`)
- Laços de repetição (`@for` , `@while`)
- Diretiva para debugar o código (`@debug`)
- Diretivas erros e alertas
- Source map (ajuda a debugar no Devtools)
- Susy (não é uma feature, mas sim um framework para montar grids customizadas - <http://susy.oddbird.net>)

11.3 LINKS DA SAIDEIRA

Deixo aqui alguns links que enxergo como boas referências para discutir e aprender mais sobre o Sass e nossa área de front-end em geral:

- **Blogs**
 - <http://blog.caelum.com.br> — Blog principal do grupo Caelum, que não se limita só a assuntos técnicos, possuindo conteúdos de UX e Agilidade também.
 - <http://blog.alura.com.br> — Blog da plataforma de cursos online Alura, conteúdos das áreas de Mobile, Dev, Front-end, Infra, Design/UX e Business.
 - <http://blog.sass-lang.com> — Blog da galera que desenvolve o Sass. Sempre quando há novas features, eles postam a respeito.

- <http://thesassway.com> — Gosto bastante desse por dividir seus posts em níveis, do iniciante ao avançado.

- **Fóruns:**

- <http://www.guj.com.br>
- <https://github.com/frontendbr>
- <http://forum.tableless.com.br>
- <http://forum.casadocodigo.com.br>

- **Comunidades:**

- Meetup CSS
- Meetup FrontUX
- Femug

- **Eventos:**

- Conferência W3C
- Front in Sampa (in Rio, in Fortaleza etc.)
- BrasilJS

Caso você tenha críticas, sugestões e comentários, pode falar comigo! Estou sempre indo a eventos da área e postando coisas no meu Twitter (@designernatan), então, todo feedback é muito bem-vindo. Você também pode entrar em contato pelo fórum da Casa do Código, em <http://forum.casadocodigo.com.br>.

Meu conselho final abrange mais que o Sass: nunca pare de estudar. Sempre se informe, pratique e discuta, seja para a ferramenta X, o framework Y ou o processo Z. Vamos perder menos tempo discutindo ferramentas e mais tempo discutindo soluções? Espero de verdade que tenha gostado e até a próxima!