Custom Css framework

What is a CSS Framework?

It is a set of organized css classes helps in implementing project design system, from typography, layouts and grids, colors, icons, components and coding conventions, to voice and tone, style-guide and documentation, to make building things on the web faster, better, and easier.

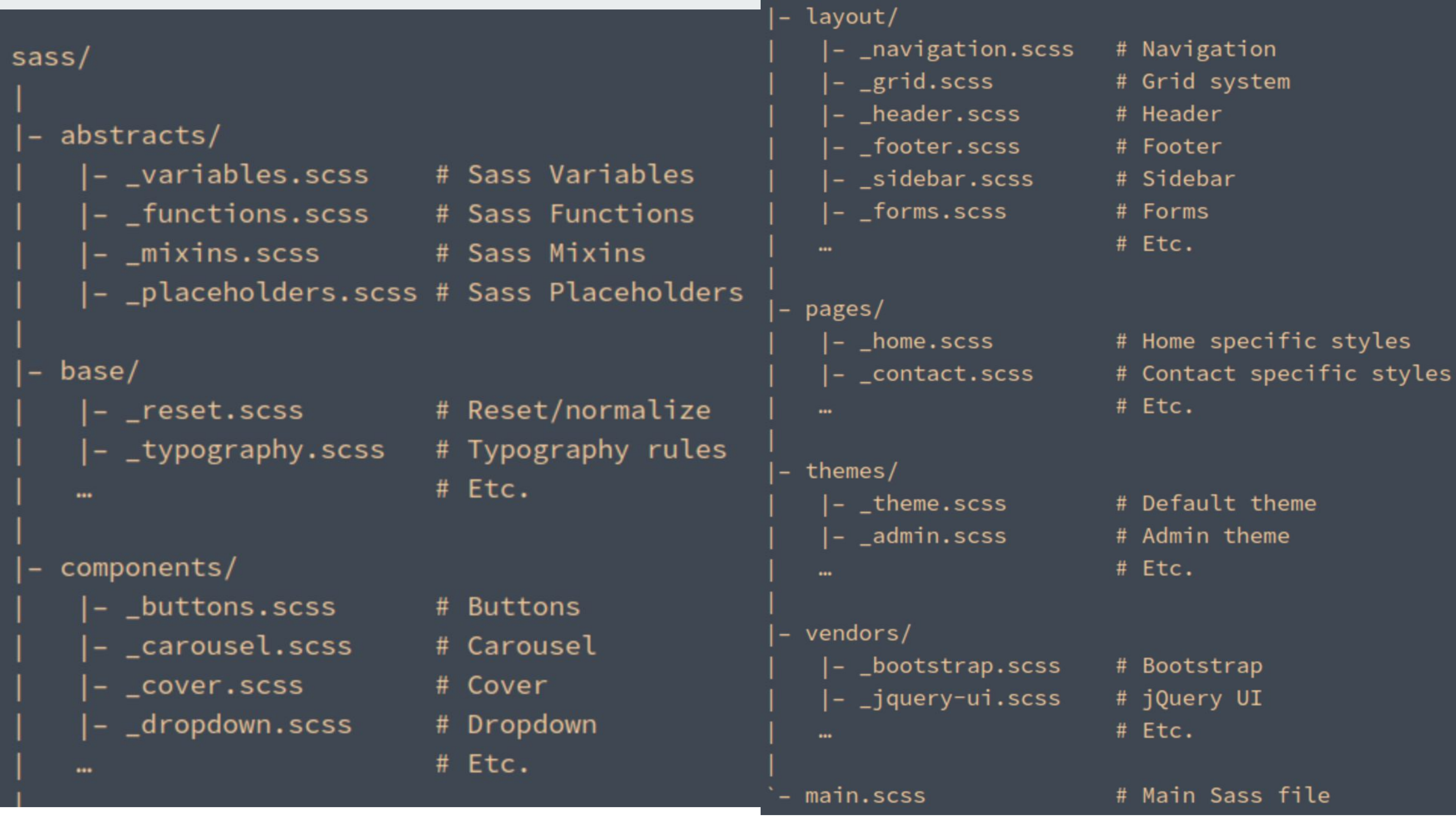
Ex: https://getbootstrap.com/docs/4.5/components/badge/

While Bootstrap is great, sometimes it might not be the best solution for your project. Other times, you simply want to create your own classes that do very specialized things you need in your project.

Why css framework ?

- **Organization**: A messy codebase can become a nightmare to work with. Making sure we had a well thought-out structure and approach was very important.
- **Maintainability**: Over time, there are going to be new developers jumping in to fix bugs and add features. We needed to have proper guidelines and conventions to make it easy for people to do things correctly.
- **Responsiveness**: Having seen a steady rise in mobile/tablet traffic, it was very important to make the experience platform-agnostic.
- **Scalability**: The company will grow in the future, and so should its website. Creating promotional pages and/or new product pages should no longer be an unpleasant (and near impossible) task.

Folder Structure



```
sass/
|
|- abstracts/
|   |- _variables.scss    # Sass Variables
|   |- _functions.scss    # Sass Functions
|   |- _mixins.scss       # Sass Mixins
|   |- _placeholders.scss # Sass Placeholders
|
|- base/
|   |- _reset.scss        # Reset/normalize
|   |- _typography.scss   # Typography rules
|   …                     # Etc.
|
|- components/
|   |- _buttons.scss      # Buttons
|   |- _carousel.scss     # Carousel
|   |- _cover.scss        # Cover
|   |- _dropdown.scss     # Dropdown
|   …                     # Etc.
|
|- layout/
|   |- _navigation.scss   # Navigation
|   |- _grid.scss         # Grid system
|   |- _header.scss       # Header
|   |- _footer.scss       # Footer
|   |- _sidebar.scss      # Sidebar
|   |- _forms.scss        # Forms
|   …                     # Etc.
|
|- pages/
|   |- _home.scss         # Home specific styles
|   |- _contact.scss      # Contact specific styles
|   …                     # Etc.
|
|- themes/
|   |- _theme.scss        # Default theme
|   |- _admin.scss        # Admin theme
|   …                     # Etc.
|
|- vendors/
|   |- _bootstrap.scss    # Bootstrap
|   |- _jquery-ui.scss    # jQuery UI
|   …                     # Etc.
|
`- main.scss              # Main Sass file
```

Hyphen Delimited Strings Naming Convention

Writing class names in css way (like css properties).

It is arguably more readable.

Correct => `.some-class {   font-weight: 10em}`

Wrong => `.someClass {   font-weight: 10em}`

BEM Naming Convention

There are 3 problems that CSS naming conventions try to solve:

- To know what a selector does, just by looking at its name
- To have an idea of where a selector can be used, just by looking at it
- To know the relationships between class names, just by looking at them

BEM attempts to divide the overall user interface into small reusable components.

The BEM approach ensures that everyone who participates in the development of a website works with a single codebase and speaks the same language. Using proper naming will prepare you for the changes in design of the website.

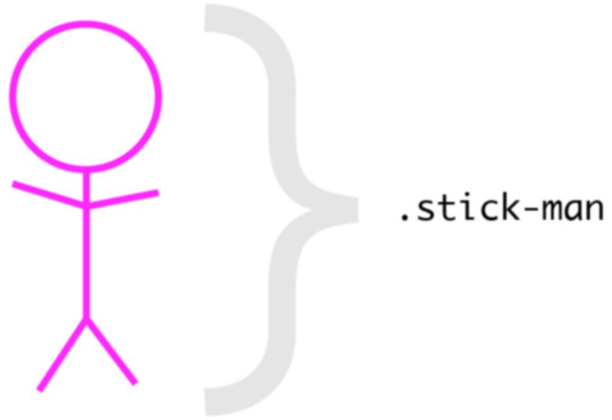
Block

The stick-man represents a component, such as a block of design.

You may have already guessed that the **B in BEM stands for 'Block'.**

In the real world, this 'block' could represent a site navigation, header, footer, or any other block of design.

Following the practice explained above, an ideal class name for this component would be `.stick-man{...}`.



Element

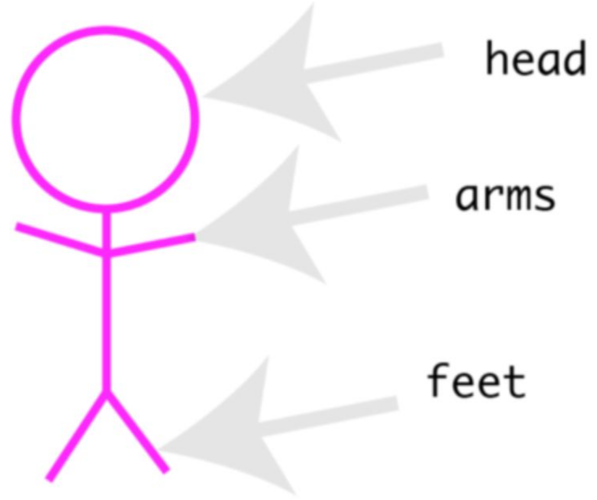
The E in ‘BEM’ stands for Elements.

For instance, the stick-man has a head, two gorgeous arms, and feet

The head , feet, and arms are all elements within the component. They may be seen as child components, i.e. children of the overall parent component.

element class names are derived by adding two underscores, followed by the element name.

```
.stick-man__head { ... };    .stick-man__arms {...}
```



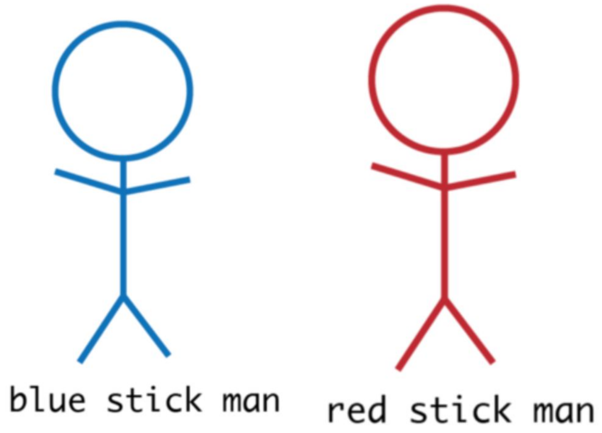
Modifiers

What if the stick-man was modified and we could have a blue or a red stick- man?

In the real world, this could be a red button or blue button. These are modifications of the component in question.

modifier class names are derived by adding two hyphens followed by the element name.

`.stick-man--blue {...};   .stick-man--red {...};`



What if we had stick-men of different head sizes?

This time the element has been modified. Remember, the element is a child component within the overall containing block.

The `.stick-man` represents the Block , `.stick-man__head` the element.

As seen in the example above, double hyphens may also be used like so:

`.stick-man__head--small { ... }; .stick-man__head--big {...};`

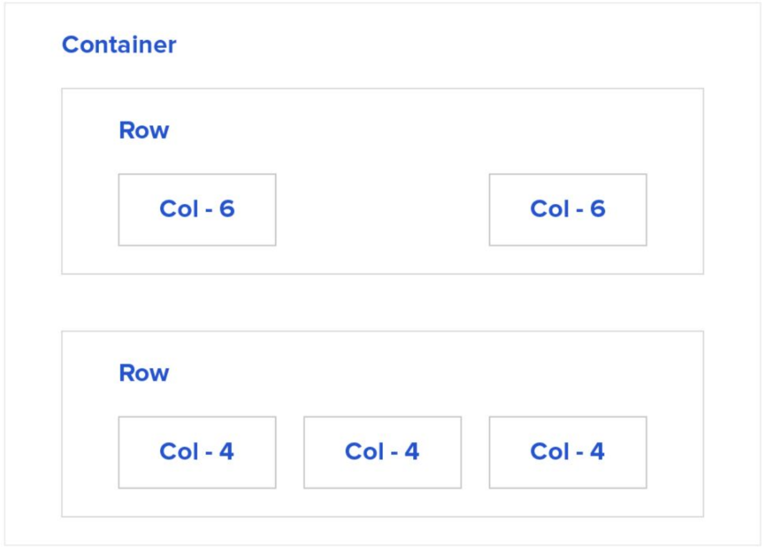
small head stick man
big head stick man

Flexbox Grid system

Basic Grid Elements Containers, Rows, and Columns.

1- Containers:
There are two types of containers: .container and .container--fluid.
.container =>is defined by being 100% of the width and have equal margins left and right

```
$grid__bp-md: 768; => grid__breakpoint-medium
.container {
 max-width: $grid__bp-md * 1px;
 margin: 0 auto;
}
```



2- Fluid container:

.container--fluid: which always has 100% width, we just override those properties with a modifier

```
&--fluid {
   margin: 0;
   max-width: 100%;
 }
```

3- Row:

We will use Flexbox to position a row's child elements, making them wrap so they don't overflow and giving them 100% width inside the row (so we can nest them later).

```
&__row {
  display: flex;
  flex-wrap: wrap;
  width: 100%;
}
```

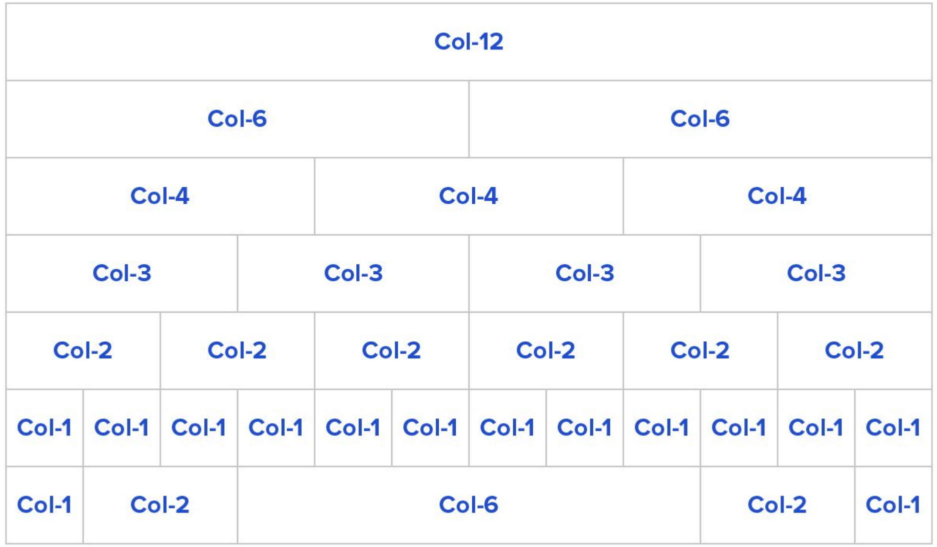
This will position the child elements side by side and wrap them into new lines if the sum of their width is greater than itself

3- Columns:

Columns define into how many parts the row is divided and how much they occupy.

We’re going to do a 12 column layout. This means we can divide the row in 1 or up to 12 parts.

When we want to have 1 column, its width will be 100%, If we want 12 columns. Then each should occupy 8.333...% or (1/12 * 100%) of the width



With Flexbox, to distribute content in this manner, we can use flex-basis, for example we can get the elements each to occupy 25% of the width by applying the following:
`flex-basis: (3/ 12 ) * 100%; ⇒ (1/4)*100% => 25%`

let’s call .col-1, a class for a column div that will have 8.333%of the width since twelve of them should fit before they have to wrap to a new line. The percentage will increment all throughout until .col-12, which will occupy 100%.

```
$grid__cols: 12;  => number of cols variables
@for $i from 1 through $grid__cols { => loop from 1 to 12
  .col-#{$i} { => create new classes at my css file .col-1, .col-2, .... , .col-12
    flex-basis: (1 / ($grid__cols / $i) ) * 100%; => (ex: at i=3 => (1/(12/3)) * 100%)) => (1/4)*100% => 25%
  }
}
```
let’s say we want to divide the width into four equal parts. We would need `.col-3` since it fits 4 times in 12, that means that `.col-3` should have 25% flex-basis.

Screen Width-dependent Columns

We create new classes with screen names: .col-* to

- .col-md-* for medium screen
- .col-sm-* for small screen
- .col-lg-* for large screen

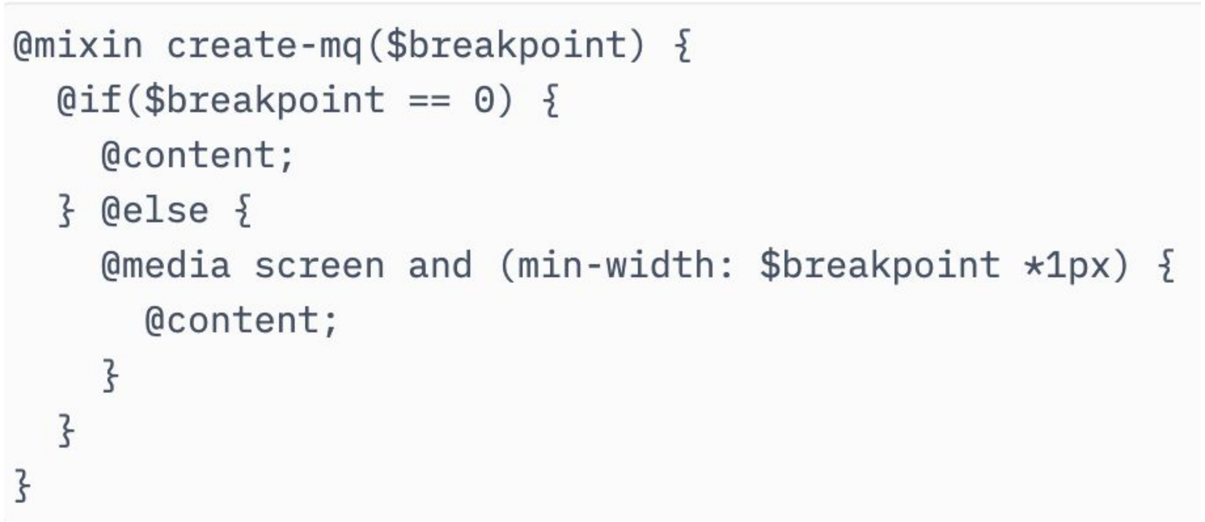
```
@media screen and (min-width: $grid__bp-md * 1px) {
 @for $i from 1 through $grid__cols {
  &__col-md-#{$i} {
   flex-basis: (1 / ($grid__cols / $i) ) * 100%;
  }
 }
}
```

Imagine an element with .col-sm-12 and .col-md-4. The expected behavior will be that below the breakpoint “md” (medium) it will have 100% width and above it it will have 33.333%;

We will have code like this in html:

```html
<div class="container">
	<div class="container__row">
		<div class="container__col-sm-12 container__col-md-4">sm12, md4</div>
		<div class="container__col-sm-12 container__col-md-4">sm12, md4</div>
		<div class="container__col-sm-12 container__col-md-4">sm12, md4</div>
	</div>
</div>
```

Media Query Mixin



```
@mixin create-mq($breakpoint) {
  @if($breakpoint == 0) {
    @content;
  } @else {
    @media screen and (min-width: $breakpoint *1px) {
      @content;
    }
  }
}
```

@include create-mq(768) { ....};

Now I have three main elements:

- modifiers: “”, -sm, -md, -lg, -xl;
- grid__cols: from 1 to 12;
- breakpoint:
    - $grid__bp-sm: 576;
    - $grid__bp-md: 768;
    - $grid__bp-lg: 992;
    - $grid__bp-xl: 1200;

I should make a loop for each grid--cols at each modifier and breakpoint so we can create mixin for it called **create-col-classes**

```
@mixin create-col-classes($modifier, $grid__cols, $breakpoint) {
 @include create-mq($breakpoint) {
  @for $i from 1 through $grid__cols {
   &__col#{$modifier}-#{$i} {
    flex-basis: (1 / ($grid__cols / $i) ) * 100%;
   }
  }
 }
}
```

Usage in scss grid file:

```
$map-grid-props: ('-sm': 0, '-md': $grid__bp-md, '-lg': $grid__bp-lg);
@each $modifier , $breakpoint in $map-grid-props {
  @include create-col-classes($modifier, $grid__cols, $breakpoint);
}
```

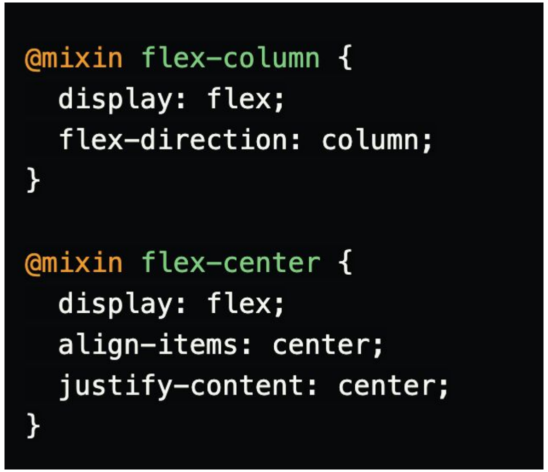
Follow this for more abstraction: https://www.toptal.com/sass/css3-flexbox-sass-grid-tutorial

Colors/theme.scss

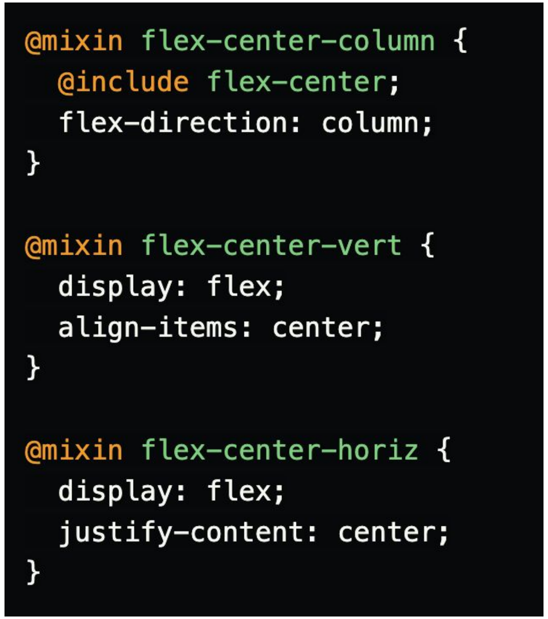
Colors variables can be defined at base folder at colors.scss => `$dark-gray: blue;`

Color theme should be created at theme folder at related theme => .bg-dark-gray: $dark-gray;

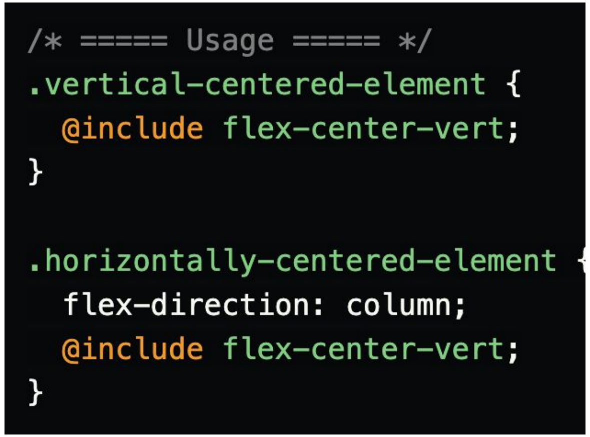
Some flex mixin



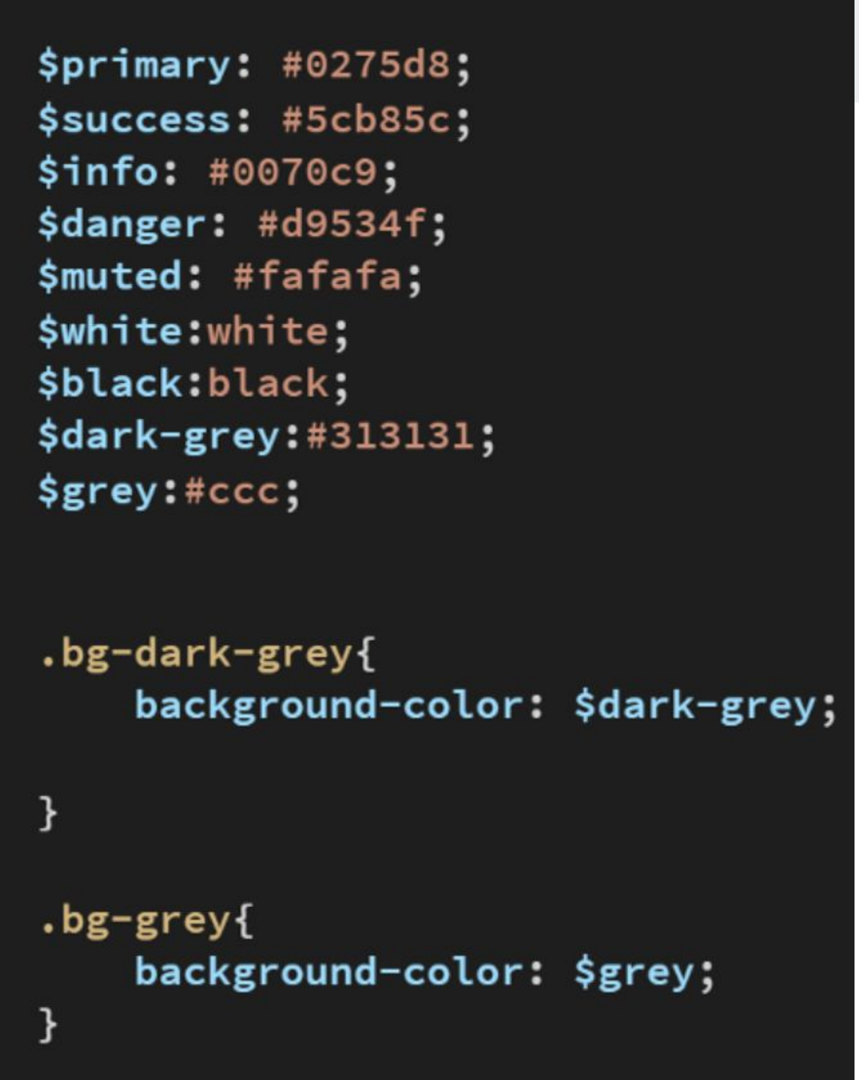
```scss
@mixin flex-column {
  display: flex;
  flex-direction: column;
}

@mixin flex-center {
  display: flex;
  align-items: center;
  justify-content: center;
}
```



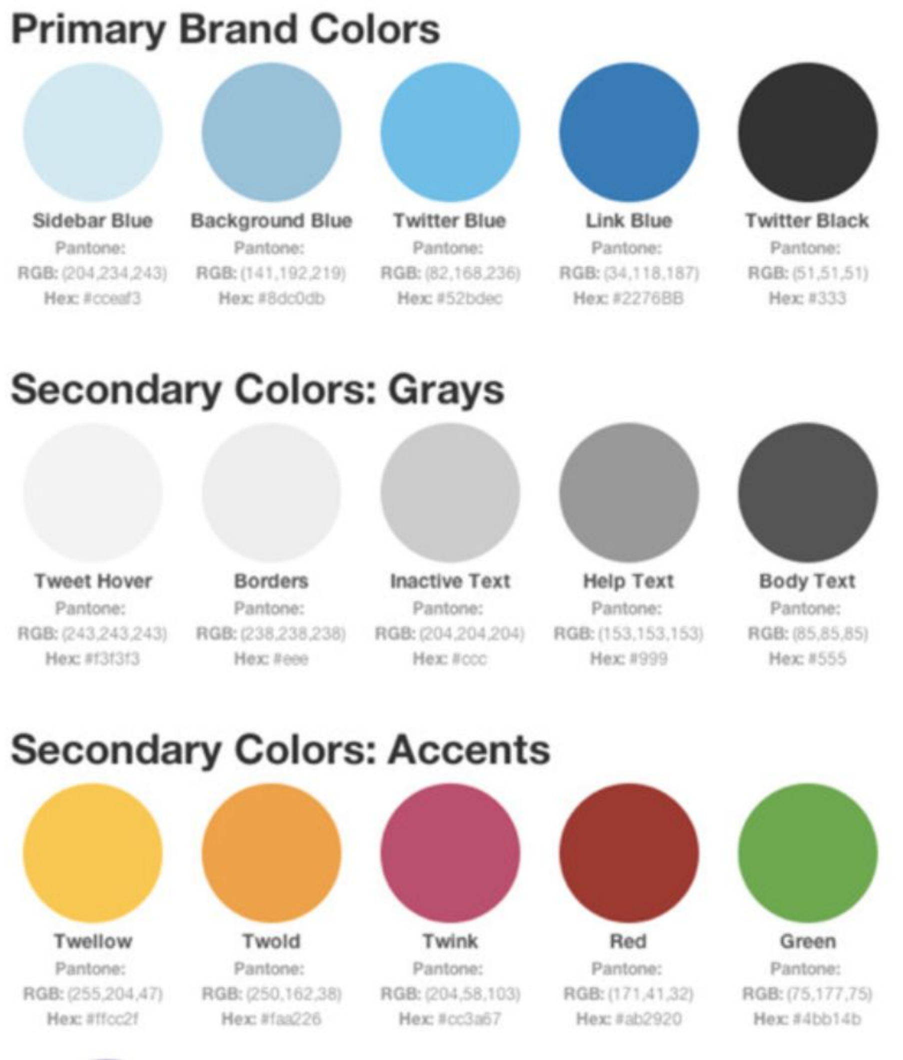
```scss
@mixin flex-center-column {
  @include flex-center;
  flex-direction: column;
}

@mixin flex-center-vert {
  display: flex;
  align-items: center;
}

@mixin flex-center-horiz {
  display: flex;
  justify-content: center;
}
```



```scss
/* ===== Usage ===== */
.vertical-centered-element {
  @include flex-center-vert;
}

.horizontally-centered-element {
  flex-direction: column;
  @include flex-center-vert;
}
```



```scss
$primary: #0275d8;
$success: #5cb85c;
$info: #0070c9;
$danger: #d9534f;
$muted: #fafafa;
$white:white;
$black:black;
$dark-grey:#313131;
$grey:#ccc;


.bg-dark-grey{
    background-color: $dark-grey;

}

.bg-grey{
    background-color: $grey;
}
```



Primary Brand Colors

Sidebar Blue	Background Blue	Twitter Blue	Link Blue	Twitter Black
Pantone:	Pantone:	Pantone:	Pantone:	Pantone:
RGB: (204,234,243)	RGB: (141,192,219)	RGB: (82,168,236)	RGB: (34,118,187)	RGB: (51,51,51)
Hex: #cceaf3	Hex: #8dc0db	Hex: #52bdec	Hex: #2276BB	Hex: #333

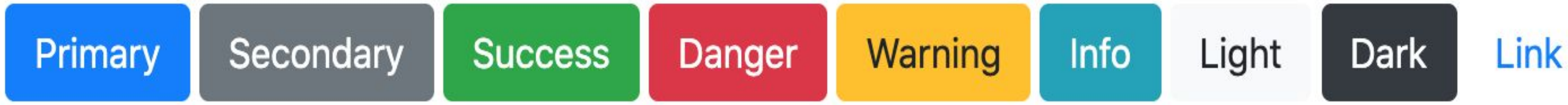
Secondary Colors: Grays

Tweet Hover	Borders	Inactive Text	Help Text	Body Text
Pantone:	Pantone:	Pantone:	Pantone:	Pantone:
RGB: (243,243,243)	RGB: (238,238,238)	RGB: (204,204,204)	RGB: (153,153,153)	RGB: (85,85,85)
Hex: #f3f3f3	Hex: #eee	Hex: #ccc	Hex: #999	Hex: #555

Secondary Colors: Accents

Twellow	Twold	Twink	Red	Green
Pantone:	Pantone:	Pantone:	Pantone:	Pantone:
RGB: (255,204,47)	RGB: (250,162,38)	RGB: (204,58,103)	RGB: (171,41,32)	RGB: (75,177,75)
Hex: #ffcc2f	Hex: #faa226	Hex: #cc3a67	Hex: #ab2920	Hex: #4bb14b

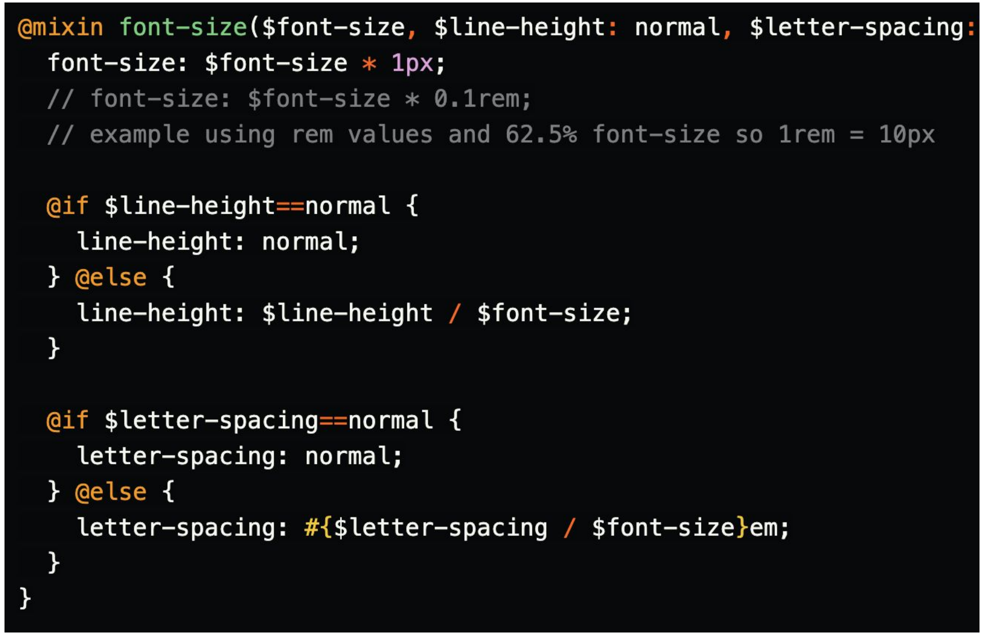
Buttons



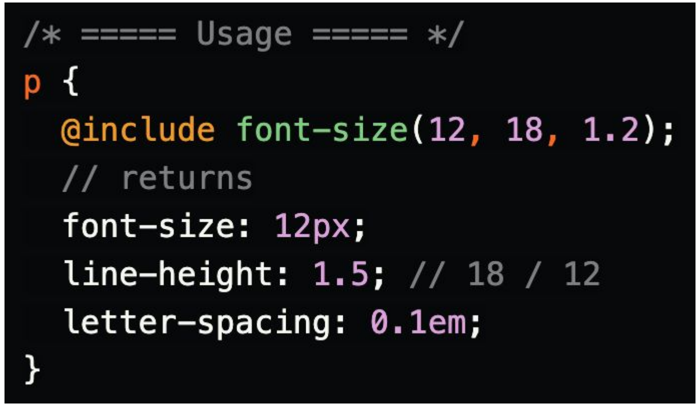
```
.primary-btn {....};
.secondary-btn{....};
.success-btn{....};
.danger-btn{....};
.warning-btn{....};
.info-btn{....};
.light-btn{....};
.dark-btn{....};
```

```
.primary-btn{
    color: white;
     background-color: $primaryColor;
      &:hover {
           background-color: $primaryHoverColor;
       }
}
```

Fonts

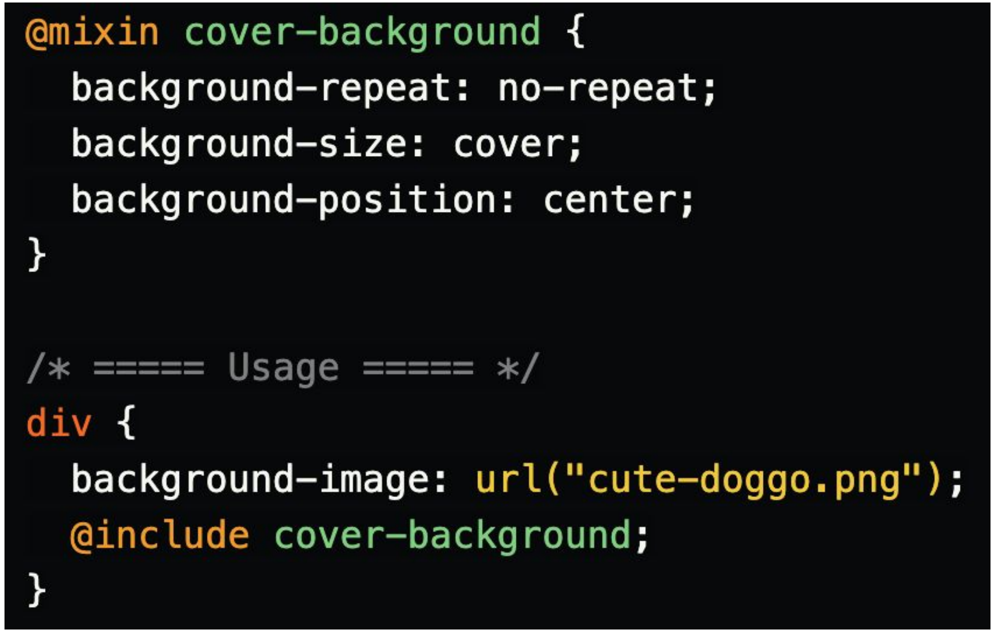


```scss
@mixin font-size($font-size, $line-height: normal, $letter-spacing:
  font-size: $font-size * 1px;
  // font-size: $font-size * 0.1rem;
  // example using rem values and 62.5% font-size so 1rem = 10px

  @if $line-height==normal {
    line-height: normal;
  } @else {
    line-height: $line-height / $font-size;
  }

  @if $letter-spacing==normal {
    letter-spacing: normal;
  } @else {
    letter-spacing: #{$letter-spacing / $font-size}em;
  }
}
```



```scss
/* ===== Usage ===== */
p {
  @include font-size(12, 18, 1.2);
  // returns
  font-size: 12px;
  line-height: 1.5; // 18 / 12
  letter-spacing: 0.1em;
}
```

Cover Background



```
@mixin cover-background {
  background-repeat: no-repeat;
  background-size: cover;
  background-position: center;
}

/* ===== Usage ===== */
div {
  background-image: url("cute-doggo.png");
  @include cover-background;
}
```

Form task

Name

First Name

Email

Email address

Select

Please select ▼

Additional Comments

☐ **Remember me**

Submit

Run SCSS using npm script

1- install node js as we discussed at SASS presentation

2- Initialize NPM:

- Open terminal
- Make sure you are on project directory
- Write this command `npm init`
- Package.json file will be created

3- then install Node-Sass: write npm `install node-sass` in terminal

4- then add the following script to package.json file at scripts object after test command

- `"scss": "node-sass --watch scss -o css"`

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "scss": "node-sass --watch scss -o css"
}
```

node-sass: Refers to the node-sass package.

--watch: An optional flag which means “watch all .scss files in the scss/ folder and recompile them every time there’s a change.”

scss: The folder name where we put all our .scss files.

-o css: The output folder for our compiled CSS.

5- Run the Script: write `npm run scss` in the terminal

References

https://www.freecodecamp.org/news/css-naming-conventions-that-will-save-you-hours-of-debugging-35cea737d849/
http://getbem.com/naming/
https://www.toptal.com/sass/css3-flexbox-sass-grid-tutorial
https://css-tricks.com/design-systems-building-future/
https://dev.to/alemesa/10-awesome-sass-scss-mixins-5ci2
https://webdesign.tutsplus.com/tutorials/watch-and-compile-sass-in-five-quick-steps--cms-28275
https://stackoverflow.com/questions/58474760/sass-use-not-loading-partial