SCSS (Sass - Syntactically Awesome Style Sheets)

Amr A Khllaf

July 16, 2025

Contents

1 Intro To SCSS & Variables

1.1 What is SCSS?

- SCSS combines the features of CSS with programming concepts, allowing you to use variables, functions, and control structures to create more dynamic and maintainable stylesheets.
- SCSS (Sassy CSS) is a `preprocessor` scripting language that is interpreted or compiled into Cascading Style Sheets (CSS). It is a superset of CSS, meaning that every valid CSS stylesheet is also a valid SCSS file. SCSS allows for more advanced features such as variables, nesting, and mixins, which can help streamline the styling process and make stylesheets more maintainable.
- SCSS is part of the Sass (Syntactically Awesome Style Sheets) family, which also includes the original Sass syntax. SCSS files use the `.scss` file extension.
- SCSS is widely used in web development to enhance the capabilities of CSS, making it easier to write and manage styles for complex web applications.
- What is the difference between SCSS and Sass?
  - SCSS is a **`newer syntax`** that is more similar to CSS, while Sass is an **`older syntax (version)`** that **uses a more indentation-based syntax**. SCSS files use the `.scss` extension, while Sass files use the `.sass` extension. Both are part of the Sass family and can be compiled to CSS.
- SCSS is a **`superset of CSS`**, meaning that any valid CSS code is also valid SCSS code. This allows developers to gradually adopt SCSS features without having to rewrite existing CSS.

1.2 How to use SCSS?

You can set up SCSS in your project using one of the following methods:

- **Using npm:** Install a Sass compiler such as `node-sass` or `sass` via npm. This allows you to compile SCSS files to CSS from the command line or as part of your build process.
- **Using the Live Sass Compiler extension in Visual Studio Code:** This extension automatically compiles SCSS files to CSS whenever you save changes, providing instant feedback and streamlining your workflow. Simply install the extension, enable it, and start writing SCSS—your CSS output will be generated in real time.

To use SCSS in your project, you need to follow these steps:

1. **Install a Sass compiler**: You can use tools like `node-sass`, `dart-sass`, or `sass` to compile SCSS files into CSS. If you're using Node.js, you can install `node-sass` via npm:

   ```
   npm install node-sass --save-dev
   ```

2. **Create an SCSS file**: Create a file with the `.scss` extension, for example, `styles.scss`.
3. **Write SCSS code**: In your SCSS file, you can use variables, nesting, mixins, and other SCSS features to write your styles.

   ```scss
   // Example of SCSS code
   $primary-color: #3498db;
   ```

```scss
body {
  background-color: $primary-color;
  color: white;

  h1 {
    font-size: 2em;
    margin-bottom: 0.5em;
  }
}
```

Alternatively, you can use the **Live Sass Compiler** extension in Visual Studio Code. This tool automatically compiles your SCSS files to CSS whenever you save changes, providing instant feedback and streamlining your workflow. Simply install the extension, enable it, and start writing SCSS—your CSS output will be generated in real time.

1.3 Variables in SCSS

- Variables in SCSS are defined using the `$` symbol followed by the variable name. They can store values such as colors, fonts, or any CSS value.
- Example of defining and using a variable in SCSS:

```scss
// Variables for SCSS styles
$primary-color: #3498db;
$secondary-color: #2ecc71;
$red-color: #e74c3c;
$padding: 20px 15px;
$flex-direction: column;

h2 {
  color: $primary-color;
  font-size: 24px;
  text-align: center;
  background-color: $red-color;
}
```

- In this example, we define several variables for colors and padding. The `$primary-color` variable is used to set the text color of the `h2` element, while the `$red-color` variable is used for the background color.
- Variables can be used throughout your SCSS file, making it easy to maintain and update styles. If you need to change the primary color, you only need to update the variable definition, and all instances of that variable will automatically reflect the change.

1.4 Difference between Variables in CSS and SCSS

- In CSS, variables are defined using the `--` prefix and accessed with the `var()` function. They are known as CSS custom properties.
- In SCSS, variables are defined using the `$` prefix and can be used directly without the need for a function.
- Example of CSS variables:

```css
:root {
  --primary-color: #3498db;
  --secondary-color: #2ecc71;
  --red-color: #e74c3c;
}

h2 {
  color: var(--primary-color);
  font-size: 24px;
  text-align: center;
  background-color: var(--red-color);
}
```

- In CSS, variables (custom properties) are defined using the `--` prefix and accessed with the `var()` function. These variables are typically declared in the `:root` selector to make them globally available throughout the stylesheet. CSS variables are dynamic and can be updated at runtime, allowing for flexible theming and changes via JavaScript.
- In SCSS, variables are defined using the `$` prefix and are processed at compile time. SCSS variables are scoped to the file or block in which they are defined, and their values are fixed once compiled to CSS. Unlike CSS variables, SCSS variables cannot be changed at runtime.
- Example of SCSS variables:

```scss
// Variables for SCSS styles
$primary-color: #3498db;
$secondary-color: #2ecc71;
$red-color: #e74c3c;
$padding: 20px 15px;
$flex-direction: column;

h2 {
  color: $primary-color;
  font-size: 24px;
  text-align: center;
  background-color: $red-color;
}
```

1.5 Organizing SCSS Variables for Reusability

To keep your styles organized and maintainable, it's best practice to define all your variables in a separate SCSS file, such as `_variables.scss`. You can then import this file into any other SCSS file where you need access to those variables.

Example:

1. **Create a variables file**
   In `_variables.scss`, define your variables:
   - The **`Underscore(_)`** in the filename indicates that this file is a partial, which means it won't be compiled into a separate CSS file but can be imported into other SCSS files.

```scss
// _variables.scss
$primary-color: #3498db;
$secondary-color: #2ecc71;
$red-color: #e74c3c;
$padding: 20px 15px;
$flex-direction: column;
```

2. **Import variables into other SCSS files**
   At the top of your SCSS file, use the `@import` directive:

```scss
// styles.scss
@import "variables";

h2 {
  color: $primary-color;
  background-color: $red-color;
}
```

- This approach makes your stylesheets more modular and easier to maintain, as you can update variables in one place and have those changes reflected throughout your project.

---

2 Nesting In Selectors & Properties and For Loops In SCSS

2.1 Nesting in SCSS

- Nesting in SCSS allows you to write styles in a hierarchical manner, making it easier to read and maintain. You can nest selectors inside other selectors to reflect the structure of your HTML.
- Example of nesting in SCSS:

```scss
// styles.scss
@import "variables";
h2 {
  color: $primary-color;
  font-size: 24px;
  text-align: center;
  background-color: $red-color;

  // Nested styles for h2 elements inside a parent class
  .parent {
    padding: $padding;
    background-color: $primary-color;

    .child-1 {
      width: 100px;
      height: 20px;
      margin: 10px auto;
      background-color: $secondary-color;
    }
  }
}
```

- In this example, the styles for `.child-1` are nested inside the `.parent` class, which is itself nested inside the `h2` selector. This reflects the HTML structure and makes it clear that `.child-1` is a child of `.parent`, which is in turn related to the `h2` element.
- Nesting helps to avoid repetition and keeps your styles organized, especially when dealing with complex components or layouts.

2.2 For Loops in SCSS

- SCSS supports control structures like loops, which allow you to generate styles dynamically. The `@for` directive is used to create loops in SCSS.
- You can access the loop index using the `#{}` syntax, which allows you to interpolate the index value into your styles.
- Example of using a for loop in SCSS:

```
.parent {
  padding: $padding;
  background-color: $primary-color;
}
@for $i from 1 to 7 {
  .child-#{$i} {
    width: 100px;
    height: 20px;
    margin: 10px auto;
    background-color: darken($secondary-color, $i * 5%);
  }
}
// This will create classes .child-1, .child-2, ..., .child-6 with
↪  different background colors
// The last number isn't included in the loop, so it will create 6
↪  classes.
```

- But if i want to include the last number in the loop, I can use `@for $i from 1 through 7` instead of `@for $i from 1 to 7`.

```
@for $i from 1 through 7 {
  .child-#{$i} {
    width: 100px;
    height: 20px;
    margin: 10px auto;
    background-color: darken($secondary-color, $i * 5%);
  }
}

// This will create classes .child-1, .child-2, ..., .child-7 with
↪  different background colors
//  $i * 5% : make a calculation to darken the secondary color: 2 * 5% =
↪  10%, 3 * 5% = 15%, and so on.
```

2.3 Nesting for Properties in SCSS

- SCSS also allows you to nest properties within a selector, which can help keep your styles organized and reduce repetition.
- This approach treats font properties as a grouped object, promoting the DRY (Don't Repeat Yourself) principle by reducing repetition and making your styles easier to maintain.

```
h2 {
  color: $primary-color;
  font-size: 24px;
  text-align: center;
  background-color: $red-color;

  font: {
    size: 26px;
    weight: 700;
    family: "Segoe UI", Tahoma, Geneva, Verdana, sans-serif;
    style: italic;
  }
}
// This will compile to CSS with the properties nested under the h2
↪  selector
```

2.4 &(and) Operator in SCSS

- The `&` operator in SCSS is used to reference the parent selector within nested styles. It allows you to create more complex selectors by combining the parent selector with additional classes or pseudo-classes.
- Example of using the `&` operator:

```
.parent {
  padding: $padding;
  background-color: $primary-color;
  transition: all 0.5s;
  &:hover {
    background-color: lighten(black, 10%);
  }
  -webkit-transition: all 0.5s;
  -moz-transition: all 0.5s;
  -ms-transition: all 0.5s;
  -o-transition: all 0.5s;
}
// This will apply a hover effect to the .parent class, changing its
↪  background color when hovered over.
// The transition properties ensure a smooth color change effect.
// lighten(black, 10%) will lighten the black color by 10%.
```

2.4.1 lighten() and darken() Functions in SCSS

- The `lighten()` and `darken()` functions in SCSS are used to adjust the brightness of colors. They take a color and a percentage value as arguments, allowing you to create variations of a base color.

```
$primary-color: #3498db;
$secondary-color: #2ecc71;

.button {
  background-color: $primary-color;
  &:hover {
    background-color: lighten($primary-color, 10%);
  }
}

.alert {
  background-color: darken($secondary-color, 10%);
}
```

- In this example, the `.button` class uses the `lighten()` function to create a lighter background color when hovered over, while the `.alert` class uses the `darken()` function to create a darker background color.

---

3 Handling Changes on Each Selector and If Conditions

3.1 Handling Changes on Each Selector

- In SCSS, you can handle changes on each selector by using the `@each` directive. This allows you to iterate over a list or map of values and apply styles dynamically.
- Example of using `@each` to handle changes on each selector:

```
$colors: (
  primary: #3498db,
  secondary: #2ecc71,
  danger: #e74c3c,
);

@each $name, $color in $colors {
  .btn-#{$name} {
    background-color: $color;
    &:hover {
      background-color: lighten($color, 10%);
    }
  }
}
```

- We can use operators like `+`, `-`, `*`, and `/` to perform calculations on values in SCSS. These operators can be used with numbers, colors, and other values to create dynamic styles.

```
$base-font-size: 16px;

h1 {
  font-size: $base-font-size * 2;
}

.button {
  padding: $base-font-size / 2;
}

--- @for $i from 1 to 7 {
  .child-#{$i} {
    width: 100px + ($i * 50px); // Increasing width by 50px for each child
    height: 20px;
    margin: 10px auto;
    background-color: darken($secondary-color, $i * 5%);
    font-family: "Segoe UI", Tahoma, Geneva, Verdana, sans-serif;
  }
}
```

3.1.1 rgba Function in SCSS

- The `rgba()` function in SCSS is used to define colors with red, green, blue, and alpha (opacity) values. It allows you to create colors with varying levels of transparency.

```
.button {
  background-color: rgba($color: #09c, $alpha: 0.8); // 80% opacity
}
```

```
@for $i from 1 to 7 {
  .child-#{$i} {
    width: 100px + ($i * 50px);
    height: 20px;
    margin: 10px auto;
    // background-color: darken($secondary-color, $i * 5%);
    background-color: rgba($color: #fff, $alpha: $i / 10);
    font-family: "Segoe UI", Tahoma, Geneva, Verdana, sans-serif;
  }
}
```

- In this example, the `rgba()` function is used to create a color with 80% opacity. The `$color` parameter specifies the base color, and the `$alpha` parameter sets the opacity level.

3.2 If Conditions in SCSS

- SCSS supports conditional statements using the `@if` directive. This allows you to apply styles based on certain conditions, making your stylesheets more dynamic.
- Example of using `@if` conditions in SCSS:

```
$theme: dark;
@each $name, $color in $colors {
  .btn-#{$name} {
    background-color: $color;
    @if $theme == dark {
      color: white;
    } @else {
      color: black;
    }
    &:hover {
      background-color: lighten($color, 10%);
    }
  }
}
```

- In this example, the `@if` directive checks the value of the `$theme` variable. If it is set to `dark`, the text color is set to white; otherwise, it is set to black. This allows you to easily switch between themes or apply different styles based on conditions.
- Another Example of using `@if` conditions in SCSS:

```
@for $i from 1 to 7 {
  @if $i %2 == 0 {
    .child-#{$i} {
      width: 100px + ($i * 50px);
      height: 20px;
      margin: 10px auto;
      // Using darken to create a darker shade of the secondary color
      background-color: darken($secondary-color, $i * 5%);
      font-family: "Segoe UI", Tahoma, Geneva, Verdana, sans-serif;
    }
  } @else {
    .child-#{$i} {
      width: 100px + ($i * 50px);
      height: 20px;
      margin: 10px auto;
      // Using rgba to create a semi-transparent background
      background-color: rgba($color: #fff, $alpha: $i / 10);
      font-family: "Segoe UI", Tahoma, Geneva, Verdana, sans-serif;
    }
  }
}
```

- In this example, the `@if` directive checks if the loop index `$i` is even or odd. If it is even, it applies a darker shade of the secondary color; if it is odd, it uses a semi-transparent white background. This allows you to create alternating styles based on the index value.

---

4 Extends Selectors and Mixins and Differences in SCSS

4.1 Extends Selectors in SCSS

- The `@extend` directive in SCSS allows you to share styles between selectors, promoting code reuse and reducing duplication. When you extend a selector, the styles of the extended selector are applied to the extending selector.
- It's Like **inheritance** in object-oriented programming, where one class can inherit properties and methods from another class.

- Example of using `@extend` in SCSS:

```scss
.button {
  padding: 10px 20px;
  border: none;
  border-radius: 5px;
  background-color: $primary-color;
  color: white;
  font-family: "Segoe UI", Tahoma, Geneva, Verdana, sans-serif;
}

.btn-primary {
  @extend .button;
  background-color: $primary-color;
}

.btn-secondary {
  @extend .button;
  background-color: $secondary-color;
}
```

- In this example, the `.btn-primary` and `.btn-secondary` classes extend the styles of the `.button` class. This means that they inherit all the styles defined in `.button`, while also applying their own specific background colors.
- The `@extend` directive helps to keep your styles DRY (Don't Repeat Yourself) and makes it easier to maintain your stylesheets by avoiding duplication.
- Another Example of using `@extend` in SCSS:

```scss
h2 {
  color: $primary-color;
  font-size: 24px;
  text-align: center;
  background-color: $red-color;
}

.test {
  @extend h2;
  padding: $padding;
}
```

- In this example, the `.test` class extends the styles of the `h2` element, inheriting its color, font size, text alignment, and background color. Additionally, it adds its own padding.

4.2 Mixins in SCSS

- **Mixins:** in SCSS are **reusable blocks of styles** that can be included in multiple selectors. They allow you to define a set of styles once and apply them wherever needed, promoting code reuse and maintainability.
- Mixins themselves do not appear in the compiled CSS file; only the styles included via `@include` are output to CSS. This means mixins serve as reusable templates for styles,

but their definitions are not directly present in the final CSS.

- Mixins can also accept parameters, allowing you to create dynamic styles based on the values passed to the mixin.
- Example of using mixins in SCSS:

```scss
@mixin button-styles($bg-color, $text-color) {
  padding: 10px 20px;
  border: none;
  border-radius: 5px;
  background-color: $bg-color;
  color: $text-color;
  font-family: "Segoe UI", Tahoma, Geneva, Verdana, sans-serif;
}
.btn-primary {
  @include button-styles($primary-color, white);
}
.btn-secondary {
  @include button-styles($secondary-color, black);
}
```

- In this example, the `@mixin` directive defines a mixin called `button-styles`, which accepts two parameters: `$bg-color` and `$text-color`. The mixin contains styles for buttons, including padding, border, border-radius, background color, and text color.

- The `@include` directive is used to apply the mixin to the `.btn-primary` and `.btn-secondary` classes, passing the appropriate colors as arguments. This allows you to create buttons with consistent styles while varying their background and text colors.

- Another Example of using mixins in SCSS:

```scss
@mixin transitionMixin() {
  -webkit-transition: all 0.5s;
  -moz-transition: all 0.5s;
  -ms-transition: all 0.5s;
  -o-transition: all 0.5s;
}

.test {
  @extend h2;
  padding: $padding;
  transition: all 0.5s;
  &:hover {
    background-color: darken($primary-color, 10%);
  }
  @include transitionMixin();
}
```

- In this example, the `@mixin` directive defines a mixin called `transitionMixin`, which includes transition properties for smooth animations. The `.test` class extends the styles of the `h2` element and includes the mixin to apply the transition styles. This allows you to easily reuse the transition styles in other selectors as well.
- Mixins are particularly useful for creating complex styles that need to be reused across different parts of

4.3 Mixins With Parameters in SCSS

- Mixins can also accept parameters, allowing you to create dynamic styles based on the values passed to the mixin. This makes mixins even more powerful and flexible.
- Example of using mixins with parameters in SCSS:

```scss
@mixin button-styles($bg-color, $text-color, $padding: 10px 20px) {
  padding: $padding;
  border: none;
  border-radius: 5px;
  background-color: $bg-color;
  color: $text-color;
  font-family: "Segoe UI", Tahoma, Geneva, Verdana, sans-serif;
}

.btn-primary {
  @include button-styles($primary-color, white);
}
```

- Another Example of using mixins with parameters in SCSS:

```scss
@mixin transitionMixin($time) {
  -webkit-transition: all $time;
  -moz-transition: all $time;
  -ms-transition: all $time;
  -o-transition: all $time;
}

.test {
  @extend h2;
  padding: $padding;
  transition: all 0.5s;
  &:hover {
    background-color: darken($primary-color, 10%);
  }
  @include transitionMixin(1s);
}
```

- We can give a default value to the `$padding` parameter in the `button-styles` mixin, so if we don't pass a value for it when including the mixin, it will use the default value of `10px 20px`.

```scss
@mixin button-styles($bg-color, $text-color, $padding: 10px 20px) {
  padding: $padding;
  border: none;
  border-radius: 5px;
  background-color: $bg-color;
  color: $text-color;
  font-family: "Segoe UI", Tahoma, Geneva, Verdana, sans-serif;
}
```

```scss
.btn-secondary {
  @include button-styles($secondary-color, black);
}
```

- In this example, the `@mixin` directive defines a mixin called `button-styles`, which accepts three parameters: `$bg-color`, `$text-color`, and an optional `$padding` with a default value of `10px 20px`. The mixin contains styles for buttons, including padding, border, border-radius, background color, and text color.
- The `@include` directive is used to apply the mixin to the `.btn-primary` and `.btn-secondary` classes, passing the appropriate colors as arguments. If no value is provided for `$padding`, it will default to `10px 20px`. This allows you to create buttons with consistent styles while varying their background and text colors, and optionally adjusting the padding.
- Another Example of using mixins with parameters in SCSS:

```scss
@mixin transitionMixin($time, $property: all) {
  -webkit-transition: $property $time;
  -moz-transition: $property $time;
  -ms-transition: $property $time;
  -o-transition: $property $time;
}

.test {
  @extend h2;
  padding: $padding;
  transition: all 0.5s;
  &:hover {
    color: darken($primary-color, 10%);
  }
  @include transitionMixin(1s, color);
}
```

- In this example, the `@mixin` directive defines a mixin called `transitionMixin`, which accepts two parameters: `$time` for the duration of the transition and an optional `$property` with a default value of `all`. The mixin includes transition properties for smooth animations.
- The `.test` class extends the styles of the `h2` element and includes the mixin to apply the transition styles. The `@include` directive is used to apply the mixin with a duration of `1s` and the property set to `color`. This allows you to easily reuse the transition styles in other selectors while customizing the duration and property as needed.

4.4 Differences Between `@extend` and `@include` in SCSS

4.4.1 Comparison Between `@extend` and `@include` in SCSS

Feature	`@extend`	`@include` (Mixins)
Purpose	Inherit styles from another selector	Include reusable blocks of styles (mixins)
Usage	`@extend .selector;`	`@include mixin-name(parameters);`
Parameters	Cannot accept parameters	Can accept parameters for dynamic styling

Feature	`@extend`	`@include` (Mixins)
Output	Merges selectors in the compiled CSS	Generates new CSS rules for each inclusion
Flexibility	Less flexible, mainly for shared styles	More flexible, supports logic and customization
Best For	Code reuse via inheritance	Code reuse via reusable, customizable style blocks

- Use `@extend` for inheritance and merging selectors.
- Use `@include` for reusable, parameterized styles.

---

5 Functions and Differences between Functions and Mixins in SCSS

5.1 Functions in SCSS

- Functions in SCSS are used to perform calculations or transformations on values and return a result. They can be built-in functions provided by SCSS or custom functions defined by the user.
- Functions can take parameters and return a value, which can then be used in your styles.

```scss
@function convertRemToPx($rem) {
  @return $rem * 16px; // Assuming 1rem = 16px
}

.test {
  @extend h2;
  padding: $padding;
  transition: all 0.5s;
  &:hover {
    color: darken($primary-color, 10%);
  }
  @include transitionMixin(1s, color);

  font-size: convertRemToPx(1.5); // Convert 1.5rem to px
}
```

5.2 Differences Between Functions and Mixins in SCSS

Feature	Functions	Mixins
Purpose	Perform calculations and return values	Include reusable blocks of styles
Usage	`@function name($args) { ... }`	`@mixin name($args) { ... }`
Parameters	Can accept parameters	Can accept parameters
Return	`Always returns a value`	Does not return a value

Feature	Functions	Mixins
Output	Used within property values	Generates new CSS rules for each inclusion
Flexibility	Less flexible, mainly for calculations	More flexible, supports logic and customization
Best For	Code reuse via calculations	Code reuse via reusable, customizable style blocks
Call Syntax	`function-name(arguments)`	`@include mixin-name(arguments)`
Placement	Used within property values or calculations	Generates new CSS rules for each inclusion

- You Should call the function in a property value or as part of a calculation.
- Mixins are called using the `@include` directive, which generates new CSS rules based on the mixin's definition.

5.3 Example of Using Functions and Mixins Together

```
@mixin button-styles($bg-color, $text-color) {
  padding: 10px 20px;
  border: none;
  border-radius: 5px;
  background-color: $bg-color;
  color: $text-color;
  font-family: "Segoe UI", Tahoma, Geneva, Verdana, sans-serif;
}

@function convertRemToPx($rem) {
  @return $rem * 16px; // Assuming 1rem = 16px
}
.test {
  @extend h2;
  padding: $padding;
  transition: all 0.5s;
  &:hover {
    color: darken($primary-color, 10%);
  }
  @include transitionMixin(1s, color);

  font-size: convertRemToPx(1.5); // Convert 1.5rem to px
}
.btn-primary {
  @include button-styles($primary-color, white);
}
.btn-secondary {
  @include button-styles($secondary-color, black);
}
```

5.4 Comparison between `darken()` & `lighten()` in SCSS

The `darken()` and `lighten()` functions in SCSS are used to adjust the brightness of colors by increasing or decreasing their lightness.

Name	`darken()`	`lighten()`
Purpose	Makes a color darker by reducing lightness	Makes a color lighter by increasing lightness
Example	`darken($primary-color, 10%)`	`lighten($primary-color, 10%)`
Description	Returns a color 10% darker than input	Returns a color 10% lighter than input

Usage Example:

```scss
$primary-color: #3498db;

.button-dark {
  background-color: darken($primary-color, 15%);
}

.button-light {
  background-color: lighten($primary-color, 15%);
}
```

- Use `darken()` to create deeper, richer shades for hover states or emphasis.
- Use `lighten()` to create softer, lighter shades for backgrounds or highlights.
- Both functions help you generate color variations dynamically, improving maintainability and consistency in your styles.