Week 6: SASS/SCSS (Comprehensive Guide)

1. Introduction to SASS/SCSS

What is SASS/SCSS?

- **SASS (Syntactically Awesome Style Sheets)** is a CSS preprocessor that introduces programming-like features into CSS, making stylesheets more efficient and maintainable. SASS helps by providing tools like variables, nesting, and mixins that make it easier to manage large stylesheets.
- **SCSS (Sassy CSS)** is the newer version of SASS, fully compatible with CSS syntax, making it more familiar and easier to adopt for beginners.

Why SASS/SCSS?

- **Modularity**: Break styles into small, manageable files.
- **Maintainability**: Reuse code through variables, mixins, and partials.
- **Advanced Features**: Use functions and operators for dynamic styling.
- **Efficiency**: Reduces repetition and allows for cleaner, DRY (Don't Repeat Yourself) code.

Setting Up SASS

- Install globally via npm:

  ```
  npm install -g sass
  ```

- Compile SASS files into CSS using:

  ```
  sass input.scss output.css
  ```

- Optionally, enable automatic watching of files:

  ```
  sass --watch input.scss:output.css
  ```

2. Core Features of SASS/SCSS

Variables

Variables in SASS/SCSS are similar to variables in programming, where you can define reusable values like colors, font sizes, and spacing.

Example:

```
$primary-color: #3498db;
$font-size-base: 16px;
$spacing-unit: 8px;

body {
  color: $primary-color;
  font-size: $font-size-base;
  margin: $spacing-unit * 2;
}
```

Explanation for Students:

- **Best Practice**: Use variables for any value that is reused across your stylesheets. This makes it easy to update values site-wide (e.g., changing a color scheme).
- **Teaching Tip**: Have students build a small color palette and assign it to various page elements (e.g., buttons, headers) to practice variable usage.

Nesting

Nesting allows selectors to be placed inside one another, mimicking the structure of HTML, which reduces code repetition and enhances readability.

Example:

```
nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;

    li {
      display: inline-block;

      a {
        text-decoration: none;
        color: $primary-color;
      }
    }
  }
}
```

Explanation for Students:

- **Best Practice**: Limit nesting to a few levels deep (2–3 levels). Over-nesting can lead to complex and difficult-to-maintain CSS.
- **Teaching Tip**: Have students practice nesting by styling a navigation menu or a card component. Ask them to think about how they would style these elements without nesting, and compare.

Operators

Operators in SASS allow for performing basic arithmetic operations, such as adding padding, adjusting width, or calculating values dynamically.

Example:

```scss
$base-padding: 10px;

.container {
  padding: $base-padding * 2; // 20px
  margin: $base-padding + 5px; // 15px
}
```

Explanation for Students:

- **Best Practice**: Use operators for consistent spacing, margin, and padding calculations across your layout.
- **Teaching Tip**: Give students exercises that involve calculating font sizes or spacing based on a base value to reinforce operator usage.

Mixins

Mixins allow you to create reusable blocks of code that can be included wherever needed, reducing redundancy and enabling consistency.

Example:

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

.header {
  @include flex-center;
  height: 200px;
}
```

Explanation for Students:

- **Best Practice**: Use mixins for code that will be repeated across multiple components or elements.
- **Teaching Tip**: Encourage students to create mixins for repetitive design patterns such as centering elements, creating grid layouts, or defining button styles.

Functions

Functions are similar to mixins but return a value instead of reusable blocks of styles. They are useful for calculations that yield a single result.

Example:

```scss
@function rem-calc($px-value) {
  @return $px-value / 16px * 1rem;
}

body {
  font-size: rem-calc(18px); // 1.125rem
}
```

Explanation for Students:

- **Best Practice**: Use functions for complex calculations that involve dynamic values, like converting pixels to rems for scalable typography.
- **Teaching Tip**: Have students write functions to calculate responsive padding or margins. It reinforces how functions can streamline their code.

Extends

Extends allow you to share CSS properties between selectors, promoting DRY code. It is ideal when several elements share common styles but still need some unique tweaks.

Example:

```scss
.button {
  padding: 10px 20px;
  border-radius: 5px;
}

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

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

Explanation for Students:

- **Best Practice**: Use extends when multiple components share a common base style but need slight modifications.

- **Teaching Tip**: Have students create a base style for buttons or cards and then extend these styles to add variations (primary, secondary, etc.).

3. Using Partials and Imports for Organizing Styles

Partials

Partials in SASS are small, segmented files that contain pieces of your CSS. They allow you to keep your styles modular and organized. Partials start with an underscore (`_`), which tells SASS not to compile them on their own.

Example:

```scss
// _variables.scss
$primary-color: #3498db;
$secondary-color: #2ecc71;
$font-size-base: 16px;

// _buttons.scss
@mixin button($color) {
  padding: 10px;
  background-color: $color;
  border: none;
  border-radius: 5px;
}

// main.scss
@import "variables";
@import "buttons";
```

Imports

You can import these partials into a main SASS file using the `@import` directive.

Example:

```scss
@import "variables";
@import "buttons";

.button-primary {
  @include button($primary-color);
}
```

Explanation for Students:

- **Best Practice**: Break down large files into logical partials (e.g., `_buttons.scss`, `_typography.scss`, `_colors.scss`).

- **Teaching Tip**: Have students build a small project and separate their SASS into different files. This shows them the importance of modularity in larger projects.

4. Building a Simple Design System with SASS

Design Systems

A design system is a standardized set of styles and components that ensure consistency across a project. SASS allows you to build such systems efficiently by leveraging variables, mixins, and functions.

Steps for Building a Simple Design System

1. **Define a Color Palette**:

```scss
// _colors.scss
$primary-color: #3498db;
$secondary-color: #2ecc71;
$background-color: #f8f9fa;
$text-color: #333;
```

2. **Establish Typography Rules**:

```scss
// _typography.scss
$font-base: "Roboto", sans-serif;
$font-size-base: 16px;
$font-size-large: 24px;

body {
  font-family: $font-base;
  font-size: $font-size-base;
  color: $text-color;
}

h1,
h2,
h3 {
  color: $primary-color;
}
```

3. **Standardize Spacing**:

```scss
// _spacing.scss
$spacing-small: 8px;
$spacing-medium: 16px;
$spacing-large: 32px;

.container {
  padding: $spacing-large;
}

.card {
  margin: $spacing-medium 0;
  padding: $spacing-medium;
}
```

4. **Create Reusable UI Components**:

```
// _buttons.scss
@mixin button($bg-color) {
  background-color: $bg-color;
  padding: 10px 20px;
  color: white;
  border-radius: 5px;
  cursor: pointer;
  transition: background-color 0.3s;

  &:hover {
    background-color: darken($bg-color, 10%);
  }
}

.button-primary {
  @include button($primary-color);
}

.button-secondary {
  @include button($secondary-color);
}
```

Outcome:
By building a simple design system, students will:

- Define consistent colors, typography, and spacing.
- Create reusable components (e.g., buttons, cards) that promote uniformity.
- Organize styles

into maintainable partials, simulating a professional workflow.

Project Ideas for Students:

1. **Personal Portfolio**: Build a personal portfolio site using a modular SASS structure, creating components for buttons, cards, and navigation.
2. **E-commerce Website**: Design an e-commerce product page with reusable styles and components like product cards, buttons, and typography.

3. **Landing Page**: Create a landing page for a startup using SASS variables and mixins for consistent design and responsive features.