SASS Tutorial: A Comprehensive Guide with Examples

Created by	A Arun kumaar Krishna
Created time	@September 5, 2024 5:29 PM
Tags	CSS SASS

Chapter 1: Introduction to SASS

SASS (Syntactically Awesome Style Sheets) is a powerful CSS preprocessor that extends the capabilities of standard CSS. It introduces features like variables, nesting, mixins, and more, which make writing and maintaining stylesheets easier and more efficient.

Key benefits of SASS include:

- Improved code organization
- Reusability of styles
- Easier maintenance
- Time-saving features

Chapter 2: Variables

Variables in SASS allow you to store and reuse values throughout your stylesheets. This is particularly useful for colors, font stacks, and other values that you might want to reuse or update globally.

```
// Defining variables
$primary-color: #3498db;
$secondary-color: #e74c3c;
$font-stack: Helvetica, sans-serif;
$base-font-size: 16px;

// Using variables
body {
  font-family: $font-stack;
  font-size: $base-font-size;
  color: $primary-color;
}

.button {
  background-color: $secondary-color;
  color: white;
  padding: $base-font-size / 2 $base-font-size;
}

// You can also use variables in calculations
.container {
  width: 100%;
  max-width: $base-font-size * 50; // 800px
}
```

Chapter 3: Nesting

SASS allows you to nest your CSS selectors in a way that follows the same visual hierarchy of your HTML. This can help make your stylesheets more readable and maintainable.

```scss
nav {
  background: #333;
  padding: 10px;

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

    li {
      display: inline-block;

      a {
        color: white;
        text-decoration: none;
        padding: 5px 10px;

        &:hover {
          background-color: #555;
        }
      }
    }
  }
}
```

In this example, we've nested selectors to match the structure of a navigation menu. The '&' symbol is used to refer to the parent selector, allowing us to easily add pseudo-classes or modifiers.

Chapter 4: @importing

SASS improves upon CSS's @import by allowing you to import SASS and CSS stylesheets. This feature helps in organizing your code into smaller, more manageable files.

```scss
// _variables.scss
$primary-color: #3498db;
$font-stack: Helvetica, sans-serif;

// _reset.scss
html, body, ul, ol {
  margin: 0;
  padding: 0;
}

// _typography.scss
body {
  font: 100% $font-stack;
  color: $primary-color;
}

// main.scss
@import 'variables';
@import 'reset';
@import 'typography';

.container {
  width: 80%;
  margin: 0 auto;
}
```

By using @import, you can split your styles into logical sections and combine them in your main stylesheet. This approach promotes modularity and makes your stylesheets easier to manage.

Chapter 5: Mixins

Mixins allow you to define styles that can be re-used throughout your stylesheet. They are particularly useful for vendor prefixes and commonly repeated style patterns.

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

// Mixin with parameters
@mixin box-shadow($x, $y, $blur, $color) {
  -webkit-box-shadow: $x $y $blur $color;
     -moz-box-shadow: $x $y $blur $color;
          box-shadow: $x $y $blur $color;
}

// Using mixins
.container {
  @include center-element;
}

.card {
  @include box-shadow(2px, 2px, 5px, rgba(0,0,0,0.3));
}

// Mixin with content
@mixin media-query($width) {
  @media screen and (max-width: $width) {
    @content;
  }
}

.sidebar {
  width: 30%;
  @include media-query(768px) {
    width: 100%;
  }
}
```

Chapter 6: Extend/Inheritance

The @extend directive lets you share a set of CSS properties from one selector to another. This is useful for creating relationships between selectors without duplicating code.

```
%message-shared {
  border: 1px solid #ccc;
  padding: 10px;
  color: #333;
}

.success {
  @extend %message-shared;
  border-color: green;
}

.error {
  @extend %message-shared;
  border-color: red;
}

.warning {
  @extend %message-shared;
  border-color: yellow;
}
```

In this example, .success, .error, and .warning all inherit the styles from %message-shared, but each has its own border color. This approach reduces repetition and makes it easier to maintain consistent styles across similar elements.

Chapter 7: CSS in JS

While not a SASS feature, CSS-in-JS is a styling technique where JavaScript is used to style components. It's commonly used in modern front-end frameworks like React. Here's an example using styled-components:

```
import styled from 'styled-components';

// Creating a styled component
const Button = styled.button`
  background: ${props => props.primary ? 'palevioletred' : 'white'};
  color: ${props => props.primary ? 'white' : 'palevioletred'};
  font-size: 1em;
  margin: 1em;
  padding: 0.25em 1em;
  border: 2px solid palevioletred;
  border-radius: 3px;
`;

// Using the styled component
render(
  <div>
    <Button>Normal</Button>
    <Button primary>Primary</Button>
  </div>
);

// Extending styles
const TomatoButton = styled(Button)`
  color: tomato;
  border-color: tomato;
`;

render(
  <div>
    <Button>Normal Button</Button>
    <TomatoButton>Tomato Button</TomatoButton>
  </div>
);
```

CSS-in-JS offers benefits like dynamic styling, scoped styles, and the ability to use JavaScript logic in your styles. However, it's important to note that this approach is different from SASS and comes with its own set of pros and cons.

Chapter 8: Functions

Here's a simple tutorial on how to use **SASS** to create a small CSS example with calculations inside a function that includes conditionals. We'll cover:

1. Creating a SASS function.
2. Using calculations within the function.
3. Adding conditional logic (`@if`).
4. Compiling the SASS into CSS.

Steps to Create the SASS Example:

1. Basic Setup:

Ensure you have SASS installed. If you haven't already installed it, you can do so via npm:

```
bashCopy code
npm install -g sass
```

Now you can compile your `.scss` file into CSS by running:

```
bashCopy code
sass input.scss output.css
```

2. Create a SASS Function with Conditionals and Calculations

Let’s create a SASS function that calculates the `font-size` and `padding` of a button based on a given size parameter. The function will handle two sizes: "small" and "large". It will use conditional logic (`@if` / `@else`) to adjust the CSS properties dynamically.

SASS Code (input.scss)

```
scssCopy code
// SASS function with conditional and calculations
@function calculate-size($size) {
  @if $size == "small" {
    @return (
      font-size: 12px,
      padding: 5px 10px
    );
  } @else if $size == "large" {
    @return (
      font-size: 20px,
      padding: 15px 30px
    );
  } @else {
    // Default if size is not recognized
    @return (
      font-size: 16px,
      padding: 10px 20px
    );
  }
}

// Use the function in the button class
.button {
  @each $size in ("small", "large", "default") {
    &--#{$size} {
      $size-styles: calculate-size($size);

      font-size: map-get($size-styles, font-size);
      padding: map-get($size-styles, padding);
      background-color: blue;
      color: white;
      border: none;
      border-radius: 5px;
      cursor: pointer;

      &:hover {
        background-color: darken(blue, 10%);
      }
    }
  }
}
```

Explanation:

- **`@function calculate-size($size)`**: This is a custom SASS function that accepts a `$size` parameter. It uses the `@if` condition to determine whether the size is "small", "large", or a default value. It calculates the corresponding `font-size` and `padding` for each case.
- **`@each $size in ("small", "large", "default")`**: This loops over the sizes to create multiple button classes, applying the styles returned from the function.
- **`map-get()`**: We use `map-get()` to extract values like `font-size` and `padding` from the return value of the `calculate-size` function (which is a map).

3. Compile the SASS into CSS

After writing your SASS file, compile it using the `sass` command:

```
bashCopy code
sass input.scss output.css
```

4. Output CSS

Here's what the compiled CSS will look like:

```
cssCopy code
.button--small {
  font-size: 12px;
  padding: 5px 10px;
  background-color: blue;
  color: white;
  border: none;
  border-radius: 5px;
  cursor: pointer;
}

.button--small:hover {
  background-color: #0000b3;
}

.button--large {
  font-size: 20px;
  padding: 15px 30px;
  background-color: blue;
  color: white;
  border: none;
  border-radius: 5px;
  cursor: pointer;
}

.button--large:hover {
  background-color: #0000b3;
}

.button--default {
  font-size: 16px;
  padding: 10px 20px;
  background-color: blue;
  color: white;
  border: none;
  border-radius: 5px;
  cursor: pointer;
}

.button--default:hover {
  background-color: #0000b3;
}
```

5. Using the Generated CSS in HTML

Here's an HTML example that uses the generated CSS:

```
htmlCopy code
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Button Example</title>
    <link rel="stylesheet" href="output.css">
</head>
<body>

    <button class="button--small">Small Button</button>
    <button class="button--large">Large Button</button>
    <button class="button--default">Default Button</button>

</body>
</html>
```

Key Features Demonstrated:

- **SASS Functions**: We used the `@function` directive to create a custom function.
- **Conditionals (`@if`)**: We used `@if` to apply different styles based on the value of the `$size` parameter.
- **Calculations**: The function calculates different styles like `font-size` and `padding`.
- **Looping with `@each`**: The `@each` directive generates multiple button styles for each size ("small", "large", and "default").

This example demonstrates a practical use of SASS functions and conditionals to create reusable styles with dynamic behavior.