Fonctionalités de base.

Les variables

En scss, il est possible de déclarer des variables de la façon suivante :

```
$var regle_css
```

Voyons quelques exemples

```
$my_color: hsla(202, 100%, 16%, 1);
$my_font: 'Poppins', sans-serif;
$font_xs: 0.75rem
$dark_mode: false
```

Ainsi, il devient possible de stocker des informations que l'on veut utiliser pour toute l'application à un endroit particulier. Maintenant, si des designer me demandent de changer la palette graphique du site internet sur lequel je suis en train de travailler, je pourrais y arriver simplement en changeant la valeur de cette variable.

> ***Bon à savoir***
>
> Les variables SCSS sont par défaut des **constantes**, il n'est pas censé être possible de les réassigner. Elles ont également une portée globale, tout fichier scss du projet faisant l'import d'un module scss récupérera ses variables.
> Il est cependant possible de changer des variables sous certaines conditions que nous verrons dans les fonctionalités avancées.

Modules

Déclarer des variables peut être pratique, mais si je souhaite travailler dans plusieurs fichiers séparés et que mes variables vont être utilisées dans toute l'application, je ne souhaite pas nécessairement avoir à copier coller ces variables dans chaque fichier SCSS de mon projet.

Pour cela, il est possible d'importer un fichier SCSS dans un autre fichier SCSS.La syntaxe à utiliser est la suivante :

```
@use 'mon_fichier_scss';
```

Ainsi, pour reprendre le project que nous avons commencé, nous allons créer un fichier scss contenant les variables générales du projet. Nous allons appeler ce fichier `_token.scss`

Voici son contenu :

```
$clr_primary: hsla(202, 100%, 16%, 1);
$clr_accent: hsla(45, 100%, 49%, 1);
$white: #ffffff;
$black: #000000;
```

Ensuite, nous allons créer un fichier _button.scss contenant les différents style de bouton que nous voulons pour notre application. Évidemment, nous allons devoir importer le fichier token.scss pour pouvoir utiliser les variables que nous avons définies à l'intérieur.

Voici son contenu :

```
@use 'token';

.btn-primary {
    display: flex;
    justify-content: center;
    align-items: center ;
    padding: 0.5rem;
    border-radius: 0.2rem;
    background: token.$clr_primary;
    color: token.$white;
}

.btn-accent {
    display: flex;
    justify-content: center;
    align-items: center ;
    padding: 0.5rem;
    border-radius: 0.2rem;
    background: token.$clr_accent;
    color: token.$black;
}

.btn-accent:hover{
    cursor: pointer;
    background: lighten($color: token.$clr_accent, $amount: 5);
}

.btn-primary:hover{
    cursor: pointer;
    background: lighten($color: token.$clr_primary, $amount: 5);
}
```

Déjà, nous avons quelque chose d'intéressant. Maintenant, nous pouvons toujours faire quelques remarques sur ce code :

- Le style du bouton souffre toujours du style par défaut des boutons donnée par le navigateur
- Il y a beaucoup de répétition dans notre code.

- On aimerai aussi pouvoir faire en sorte de pouvoir mettre le sytle du hover dans le style de la classe du bouton plutôt que d'avoir à redéfinir une règle CSS pour le hover des boutons.

Pour gérer le style par défaut, il peut être intéressant d'importer un fichier de **reset css** dans notre projet. Nous allons pour cela utiliser celui proposé par Tailwind. Pour cela, effectuez simplement un copier collé des quelques lignes de code dans le fichier _reset.css présent dans l'annexe de ce cours.

Ensuite, pour ce qui est de la répétition de code, en utilisant simplement les règles de css, nous pourrions faire plusieurs classes, comme ceci : .

```
@use 'token';

.btn {
    display: flex;
    justify-content: center;
    align-items: center ;
    padding: 0.5rem;
    border-radius: 0.2rem;
}

.btn-primary {
    background: token.$clr_primary;
    color: token.$white;
}

.btn-accent {
    background: token.$clr_accent;
    color: token.$black;
}

.btn-accent:hover{
    cursor: pointer;
    background: lighten($color: token.$clr_accent, $amount: 5);
}

.btn-primary:hover{
    cursor: pointer;
    background: lighten($color: token.$clr_primary, $amount: 5);
}
```

Maintenant, pour avoir le type de bouton qui nous intéresserait, il faudrait à la fois mettre la classe btn et la classe btn-primary pour avoir un bouton normale.

Cependant, SCSS nous propose une solution qui permet de ne pas démultiplier les classes : les mixin.

Un mixin est une forme de composant CSS qu'il est possible d'appeler un peu à la façon d'une fonction dans un langage de programmation plus classique.

Pour déclarer et faire appel à un mixin, on utilise la syntaxe suivante :

```
/*Déclaration du mixin*/
@mixin mon_mixin($arg_1, ..., $arg_n){
    ...
}
.ma-class {
    /* appel du mixin*/
    @include mon_mixin(css_value_1, ..., css_value_n)
}
```

Pour nos boutons, nous pourrions donc procéder comme cela :

```
@mixin btn($clr_bg, $clr_txt){
    display: flex;
    justify-content: center;
    align-items: center ;
    padding: 0.5rem;
    border-radius: 0.2rem;
    background: $clr_bg;
    color: $clr_txt;
}

@mixin btn_hover($clr){
    cursor: pointer;
    background: lighten($color: $clr, $amount: 5);
}

.btn-primary {
    @include btn(token.$clr_primary, token.$white)
}

.btn-accent {
    @include btn(token.$clr_accent, token.$black)
}

.btn-accent:hover{
    @include btn_hover(token.$clr_accent)
}

.btn-primary:hover{
    @include btn_hover(token.$clr_primary)
}
```

Il est également possible de préciser des arguments par défaut dans un mixin, avec la syntaxe suivante :

```
@mixin mon_mixin($arg_1 : css_value_1, ..., $arg_n : css_value_n){
    ...
}
```

Par exemple, si l'on voulait que les boutons aient un style par défaut, on pourrait faire comme ceci:

```
@use 'token';

@mixin btn($clr_bg : token.$clr_primary, $clr_txt : token.$white){
    display: flex;
    justify-content: center;
    align-items: center ;
    padding: 0.5rem;
    border-radius: 0.2rem;
    background: $clr_bg;
    color: $clr_txt;
}

@mixin btn_hover($clr : token.$clr_primary){
    cursor: pointer;
    background: lighten($color: $clr, $amount: 5);
}

.btn-primary {
    @include btn;
}

.btn-accent {
    @include btn(token.$clr_accent, token.$black);
}

.btn-accent:hover{
    @include btn_hover(token.$clr_accent);
}

.btn-primary:hover{
    @include btn_hover
}
```

Nous commençons à avoir quelque chose d'un peu plus compact et facile à maintenir. Cependant, nous pouvons remarquer que les couleurs sur le hover sont en fait dépendantes des couleurs du bouton lui-même, et l'on ne voudrait pas se tromper en faisant par exemple quelque chose comme ça :

```
@use 'token';

@mixin btn($clr_bg : token.$clr_primary, $clr_txt : token.$white){
    display: flex;
    justify-content: center;
    align-items: center ;
    padding: 0.5rem;
    border-radius: 0.2rem;
    background: $clr_bg;
    color: $clr_txt;
}
```

```
@mixin btn_hover($clr : token.$clr_primary){
    cursor: pointer;
    background: lighten($color: $clr, $amount: 5);
}

.btn-primary {
    @include btn;
}

.btn-accent {
    @include btn(token.$clr_accent, token.$black);
}

.btn-accent:hover{
    @include btn_hover
}

.btn-primary:hover{
    @include btn_hover(token.$clr_accent);
}
```

Comment faire pour rendre cet aspect moins prone à l'erreur ? Une solution pour cela, le "nesting de règles"

Le nesting des règles CSS en SCSS

Il est possible, en SCSS, de "nester les règles" dans une règle CSS. Par exemple, si nous avons les règles suivantes:

```
nav {
    display : flex;
    flex-direction : column;
    gap : 1rem;
}

nav ul li {
    width : 5rem;
    height: 5rem;
    border-radius: 9999px;
}

nav ul li:hover {
    cursor : pointer;
}
```

Nous pouvons les réprésenter de la façon suivante en scss:

```
nav {
    display : flex;
    flex-direction : column;
    gap : 1rem;
    ul {
        li {
            width : 5rem;
            height: 5rem;
            border-radius: 9999px;
            &:hover {
                cursor : pointer;
            }
        }

    }
}
```

Donc, pour nos boutons, nous pouvons finalement arriver à ce résultat :

```
@use 'token';

@mixin btn($clr_bg : token.$clr_primary, $clr_txt : token.$white){
    display: flex;
    justify-content: center;
    align-items: center ;
    padding: 0.5rem;
    border-radius: 0.2rem;
    background: $clr_bg;
    color: token.$white;
    &:hover {
        cursor: pointer;
        background-color: lighten($color: $clr_bg, $amount: 5);
    }
}

.btn-primary {
    @include btn;
}

.btn-accent {
    @include btn(token.$clr_accent, token.$black);
}
```

Maintenant, dernière chose : on remarque facilement l'effet du hover lorsque l'on passe notre sourie au dessus du bouton bleu, mais moins du jaune. Pour cette couleur particulière, on voudrait que l'effet soit plus marqué. Comment faire ?

Les conditions en SCSS

SCSS permet de créer du CSS de façon conditionnelle, grâce à des opérateurs de comparaison. Voici la façon dont on écrit une condition:

```
@mixin mon_mixin($arg_1 : true, ..., $arg_n) {
    //On accepte les valeurs booléennes...
    @if $arg_1 {
        ...
    } @else {
        ...
    }
    //... ainsi que les opérateurs de comparaison
    @if $arg_n == css_value {
        ...
    } @else {
        ...
    }
}
```

Si nous reprenons le cas de nos boutons, nous pouvons donc rajouter cette règles pour rendre l'effet du hover plus visible sur le bouton jaune.

```
@use 'token';

@mixin btn($clr_bg : token.$clr_primary, $clr_txt : token.$white){
    display: flex;
    justify-content: center;
    align-items: center ;
    padding: 0.5rem;
    border-radius: 0.2rem;
    background: $clr_bg;
    color: token.$white;
    &:hover {
        cursor: pointer;
        @if $clr_bg == token.$clr_accent {
            background-color: lighten($color: $clr_bg, $amount: 10);
        } @else {
            background-color: lighten($color: $clr_bg, $amount: 5);
        }
    }
}

.btn-primary {
    @include btn;
}

.btn-accent {
    @include btn(token.$clr_accent, token.$black);
}
```

Et voilà pour les concepts de base ! Avec ceci, vous devriez déjà être capable de créer des fiches de style SCSS cohérentes. N'oubliez pas que le plus important pour votre code est d'être lisible et maintenable, donc n'essayez pas de mettre des fonctionalité SCSS partout juste parce que vous le pouvez. Le SCSS permet énormément de chose, et peut vous créer des complications si vous ne faites pas attention à bien l'utiliser.