# processeurs & frameworks css

- Les **apports d'un processeur** type sass :
  - pouvoir d'expression amélioré
  - code mieux maintenable
  - code plus réutilisable

# Quelques exemples de problèmes

- Évolutivité et maintenance :
  - « foncer tous les bleus et changer les rouges en orange »
  - apport des variables
- Réutilisation :
  - « je veux le même bouton que sur la page d'accueil, mais avec un bleu dégradé à la place du vert »
  - apport de @extend ou @mixin
- Couplage html-css

# couplage html-css

- Css générique réutilisable (framework)
- Html **non "sémantique"** totalement lié au css

```
<input class="btn btn-lg btn-green flat">
```

```
.btn {
    ...
}
 .btn-lg {
    ...
 }
 .btn-green {
    ...
```

# couplage html-css

- Html « **sémantique** » : classes liées au document et au rôle de l'élément dans le document
- Css : **peu réutilisable** car lié à 1 application

```
<input class="valid_inscription" type="submit" value="...">
```

```
.valid_inscription {
    border: 1px solid #030303;
    font-size: 1rem;
    line-height: 1.42rem;
}
```

# apport d'un processeur : html sémantique+css générique réutilisable

```
<input class="valid_inscription">
```

```
$btn-green-bg: #318054;

@import "mySassyButtons";

.valid_inscription {
  @extend .btn;
  @extend .btn-lg;
  @extend .btn-green;
  @extend .flat;
  font-weight: 600;
}
```

```
.typography {
  i, em { @extend .italic; }
  b, strong { @extend .bold; }

  h1, .h1 { @extend .h1; margin: 1em 0 0.5em; }
  h2, .h2 { @extend .h2; margin: 1em 0 0.5em; }
  h3, .h3 { @extend .h3; margin: 1em 0 0.5em; }
  h4, .h4 { @extend .h4; margin: 1em 0 0.5em; }
  h5, .h5 { @extend .h5; margin: 1em 0 0.5em; }
  h6, .h6 { @extend .h6; margin: 1em 0 0.5em; }

  p, ul, ol, pre { @extend .block-margins; }

  ul { @extend .unordered-list; }
  ol { @extend .ordered-list; }

  pre, code { @extend .fixed; }
}
```

```
<div class="blog-post typography">
  <h1>Blog post</h1>
  ...
</div>
```

http://thesassway.com/advanced/modular-css-typography

# bénéfices (1)

- on obtient du code :
- **mieux lisible** grâce à l'imbrication de sélecteurs
- plus facilement maintenable et plus évolutif grâce aux variables, expressions et  mixins
- **plus facilement réutilisable** grâce aux variables, extends  et mixins
- tout en bénéficiant de fonctions prédéfinies

- **découplage HTML – CSS facilité et effectif**

# bénéfices (2)

- On peut se faire sa bibliothèque/framework d'éléments de mise en page prédéfinis mais adaptables et paramétrables, sous forme de classes ou mieux, de mixins

- Les gros frameworks sont faits avec un préprocesseur css :
  - Twitter bootstrap,  Foundation, Materialize : sass

# framework css

- les apports d'un framework css : des classes prédéfinies facilement utilisable et permettant une mise en forme rapide
  - grille générique prédéfinie, typographie,
  - composants : navbar, boutons, alertes, forms ...
  - des media-query
  - des plugin javascript

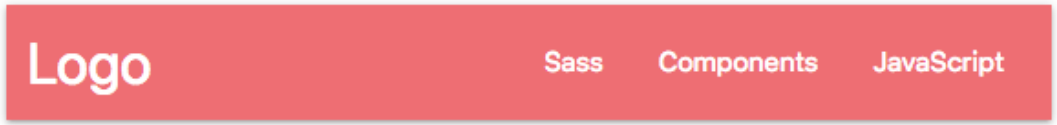
# les inconvénients

- uniformise les designs
- code html envahi par le framework css
  - introduit un couplage très fort entre document html et mise en forme css
  - rend quasiment impossible la refonte css en utilisant un autre framework
- conduit à un code html non sémantique
  - beaucoup de class liés à la mise en forme et non pas à la structure du document

# utiliser un framework avec sass

- **1ere solution :** utiliser un framework fournit sous forme de classes css avec la directive sass @extend
  - permet de réaliser effectivement le découplage html / css
  - html : classes sémantiques
  - css : définition des classes sémantiques en réutilisant les classes du framework
  - rappel: @extend dans une media-query

# exemple materialize




```
<nav>
 <div class="nav-wrapper">
   <a href="#" class="brand-logo">
           Logo</a>
   <ul id="nav-mobile"
    class="right hide-on-med-and-down">
     <li><a href="sass.html">
           Sass</a></li>
     <li><a href="badges.html">
        Components</a></li>
     <li><a href="collapsible.html">
        JavaScript</a></li>
    </ul>
  </div>
</nav>
```

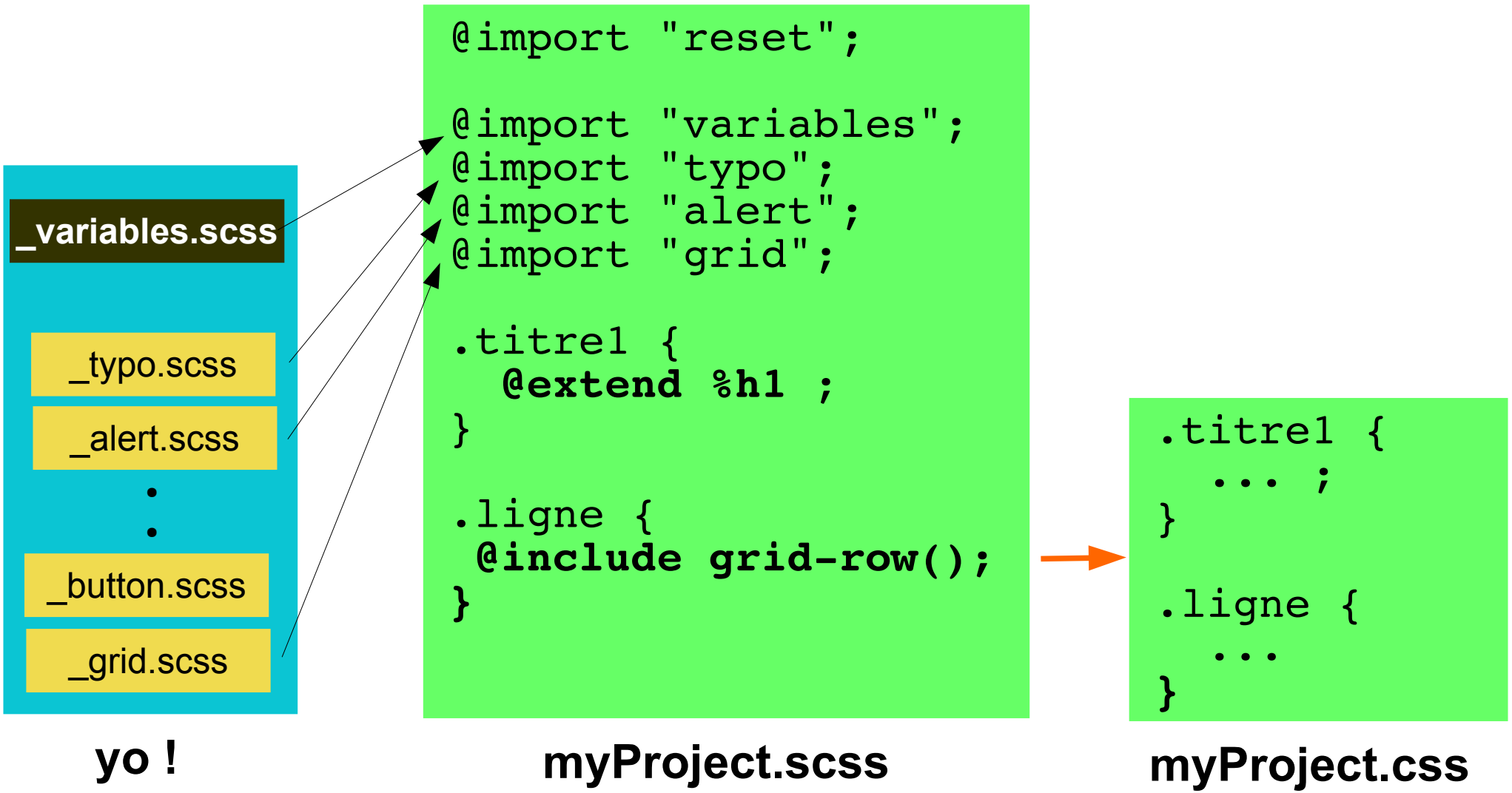
```
<nav class="navigation">
  <a href="#">Maison</a>
  <ul>
    <li><a href="#">pomme</a></li>
    <li><a href="#">poire</a></li>
    <li><a href="#">pêche</a></li>
    <li><a href="#">melon</a></li>
  </ul>
</nav>
```

```
nav.navigation {
  @extend .nav-wrapper;

  &>a:first-child {
    @extend .brand-logo, .right;
  }

  &>ul {
    @extend .left,
            .hide-on-med-and-down;
    &>li:nth-child(3) {
      @extend .active;
    }
}}
```

# 2ème solution

- on utilise la version sass/scss du framework/grille sous la forme de classes réutilisables et de mixins
  - bootstrap, foundation :
    - écrits en sass,
    - mixins fournis
  - susy, neat, griddle : des grilles écrites en sass

# utiliser/fabriquer un framework **scss**



# un framework **scss**

- le développeur importe les modules dont il a besoin sous forme de mixins/placeholder
- c'est lui qui définit les règles et classes qui seront présentes dans le css de son projet : le lien framework-html est créé *par le développeur*
- il peut ainsi créer un code html plus *sémantique*
- *le fichier* `_variables.scss` *lui permet d'* ***adapter*** le framework à son projet en redéfinissant des valeurs pour les différentes variables ayant une valeur par défaut définie dans les différents modules

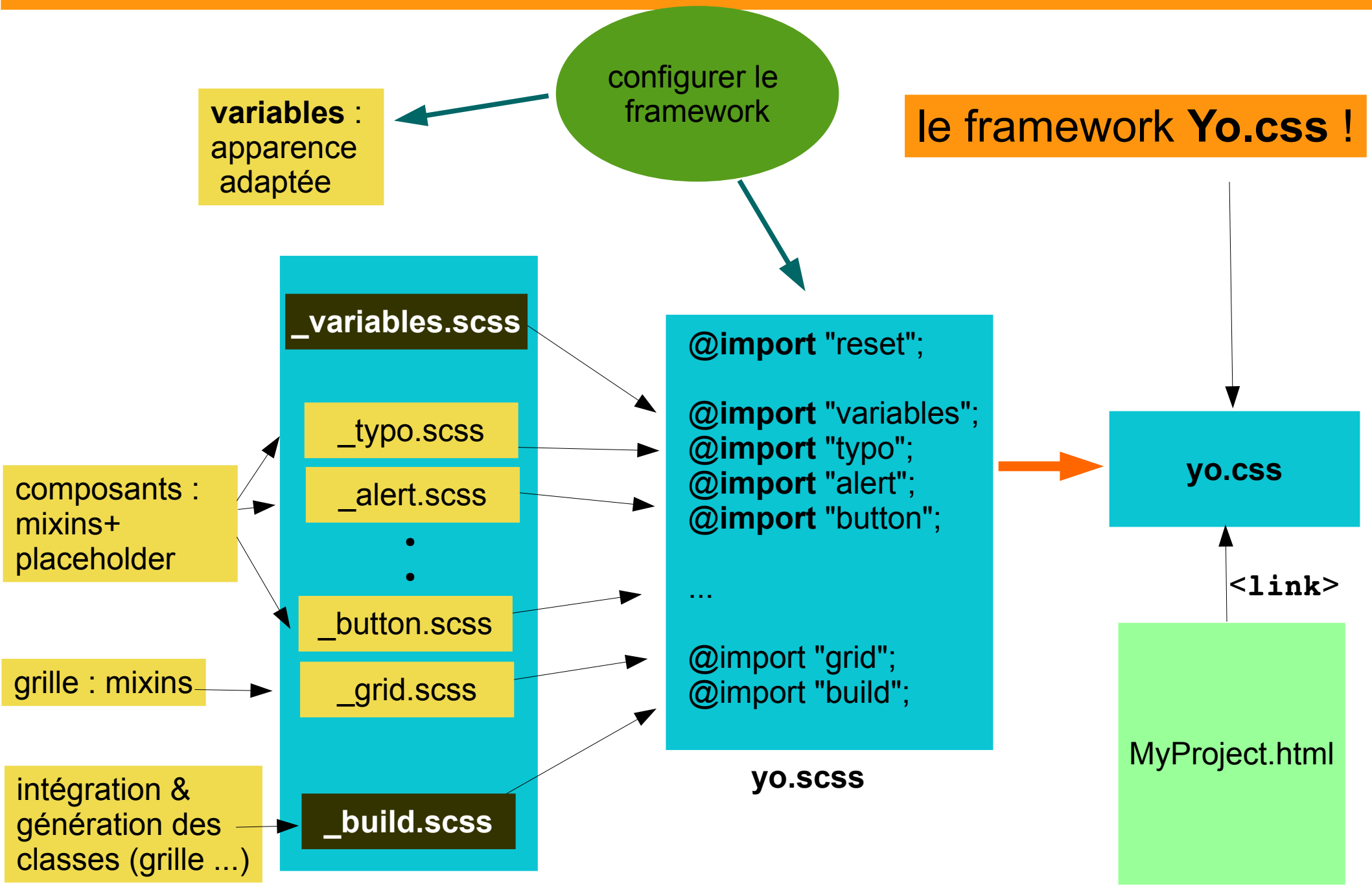
# un framework css en scss

- générer en scss les différentes classes du framework css
- le framework scss est défini sous forme de classes/placeholder et mixins
- le framework css est construit par un programme scss qui :
  - définit/génère les noms des classes css
  - produit le code associé à partir de variables, @extend et @include

```
@mixin grid-col($cols, $numcols) {
      box-sizing: border-box;
      float: left;
      width: $cols * …  $numcols… %;
      margin: 0 1%;
}

@mixin grid-offset($cols, $numcols) {
      margin-left: $cols* … %;
}

@for $cols from 1 to $numcols {
      .span-#{$cols} {
            @include grid-col($cols, $numcols);
      }
}

@for $cols from 1 to $numcols {
      .offset-#{$cols} {
            @include grid-offset($cols, $numcols);
      }
}
```

```
.span-1 {
      …
}
.span-2 {
      …
}
.span-3 {
      …
}
.span-n {
      …
}


.offset-1 {
      …
}
.offset-2 {
      …
}
.offset-3 {
      …
}
.offset-n {
      …
}
```

# un framework **css** en scss



# framework css en scss

- le fichier `_variables.scss` permet de ***customiser*** le framework en redéfinissant des valeurs pour les différentes variables ayant une valeur par défaut définie dans les différents modules
- le fichier `_build.scss` génère les **règles** et les **classes** qui seront présentes dans le framework css à partir des mixins/placeholders définis dans les modules
- le développeur utilise ces classes définies dans le framework dans son code html : le lien framework-html est créé *par le framework*

# fabriquer / utiliser un framework

- lorsque l'on **fabrique** un framework : penser à le livrer sous forme **css et scss** pour permettre de l'utiliser sous forme de classes/règles prédéfinies et sous forme de modules scss
- lorsque l'on **utilise** un framework, préférer la version **scss** qui permet de produire un code html de meilleure qualité
- lorsque l'on **développe/intègre** une appli web, essayer de systématiquement produire des placeholders/mixins réutilisables