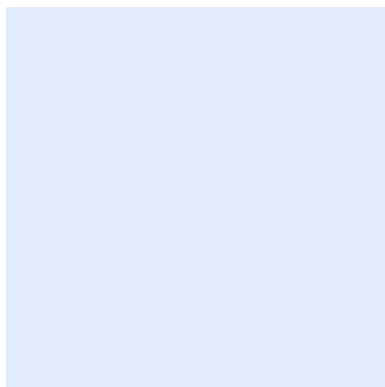


Référence SPIP/Z/2017.001 – Ed1.7



GUIDE Z

CONCEPTION DU SQUELETTE Z-GALA



FICHE D'IDENTIFICATION

Rédacteur	Eric Lupinacci
Projet	SPIP
Étude	Conception du squelette Z-Gala
Nature du document	Guide
Date	06/01/2022
Nom du fichier	Guide Z - Creation du squelette Z-Gala.docx
Référence	SPIP/Z/2017.001 – Ed1.7
Dernière mise à jour	16/01/2022 15:32:30
Langue du document	Français
Nombre de pages	47

TABLE DES MATIERES

1.	INTRODUCTION	6
2.	LE FRAMEWORK Z V2	6
2.1	INTRODUCTION	6
2.2	OBJECTIFS DU FRAMEWORK Z	6
2.3	ORGANISATION ET GENERATION D'UNE PAGE HTML	6
2.4	LAYOUT UNIQUE, BLOCS Z ET FICHIERS DIST	7
2.5	RETOUR SUR LE HEAD	9
2.5.1	LE FICHIER <code>HEAD/\$ {TYPE-PAGE} .HTML</code>	9
2.5.2	LE FICHIER <code>INCLUDE/HEAD .HTML</code>	9
2.5.3	LE FICHIER <code>HEAD_JS/\$ {TYPE-PAGE} .HTML</code>	11
2.6	PAGES AUTOMATIQUES ET ECHAFAUDAGE	11
2.7	THEMES	11
3.	ETAPE 1 : UNE PAGE SOMMAIRE AVEC LE LAYOUT GALA 23	12
3.1	INTRODUCTION	12
3.2	CREATION DU <code>PAQUET .XML</code>	12
3.3	DECLARATION DES BLOCS Z	12
3.4	CREATION DES REPERTOIRES	13
3.5	MISE EN PLACE DU LAYOUT UNIQUE	13
3.6	MISE EN PLACE DU « HEAD »	14
3.7	MISE EN PLACE DE LA PAGE D'ACCUEIL	15
4.	ETAPE 2 : CHANGER DE LAYOUT GALA	16
4.1	QUELLES SOLUTIONS POUR PARAMETRER LE LAYOUT ?	16
4.2	DES THEMES POUR CHAQUE LAYOUT	16
4.3	UNE CONFIGURATION DU LAYOUT	17
5.	ETAPE 3 : ENRICHIR LES STYLES ET FINALISER LE BODY	18
5.1	RENOMMER DES BLOCS DU BODY	18
5.2	TINYTYPO POUR LA TYPOGRAPHIE	18
5.3	NETTOYER LES LAYOUTS GALA	19
5.4	OOCSS POUR LES BOITES	19
5.5	AMELIORER LES BOITES	20

5.6	AJOUTER UN FIL D'ARIANE	21
6.	ETAPE 4 : MODERNISER LES CSS AVEC LESS	22
6.1	POURQUOI DU LESS	22
6.2	INTEGRER LES FICHIERS LESS DE TINYTYPO	22
6.3	PASSER LES STYLES DE BOITES SOUS LESS	22
6.3.1	FICHER <code>CSS/BOX.CSS</code>	23
6.3.2	FICHER <code>CSS/BOX_SKINS.CSS</code>	24
6.3.3	CONCLUSION	24
6.4	LES LAYOUTS GALA EN LESS	24
6.5	FIL D'ARIANE ET HABILLAGE EN LESS	25
6.5.1	FIL D'ARIANE	25
6.5.2	HABILLAGE	25
6.6	CONCLUSION	26
7.	ETAPE 5 : INTEGRER LES PREMIERS ELEMENTS SPIP	26
7.1	AJOUT DES STYLES DE SPIP	26
7.2	CHOISIR UNE POLICE DE BASE	27
7.3	AJOUTER UNE BARRE PRINCIPALE DE NAVIGATION HORIZONTALE	27
7.4	UTILISER LES BALISES HTML5	27
7.5	AMELIORER LE FIL D'ARIANE	29
7.6	STRUCTURER LA PAGE SOMMAIRE	29
7.7	AJOUTER LA PAGE ARTICLE	30
7.8	AJOUTER LA PAGE RUBRIQUE	30
7.9	CONCLUSION	30
8.	ETAPE 6 : INTEGRER LE NOISETIER	31
8.1	POURQUOI LE NOISETIER ?	31
8.2	FONCTIONNEMENT DU NOISETIER	31
8.2.1	IDENTIFICATION DES PAGES	32
8.2.2	IDENTIFICATION DES BLOCS	32
8.2.3	DESCRIPTION DES TYPES DE NOISETTE	33
8.2.4	COMPILATION DES NOISETTES	33
8.3	MISE EN PLACE DANS Z-GALA	34
8.3.1	IDENTIFICATION DES BLOCS	34
8.3.2	DESCRIPTION DES PAGES	34
8.3.3	REORGANISATION DES INCLUSIONS EN NOISETTES	35
8.3.4	FIL D'ARIANE	36
8.3.5	CONTENU DE LA PAGE SOMMAIRE	36

8.3.6	LA BARRE DE NAVIGATION PRINCIPALE	36
8.3.7	LA PAGE ARTICLE	37
8.3.8	LA PAGE RUBRIQUE	39
8.3.9	LES PAGES DE DEMO	39
8.3.10	LA PAGE METEO DE RAINETTE	40
8.3.11	LES PAGES 404 ET 401	40
8.3.12	LA NAVIGATION SECONDAIRE	40
8.4	CONCLUSION	41
9.	MODERNISER LESS AVEC SCSS	41
9.1	CONVERTIR LES FICHIERS LESS EN SCSS	41
9.2	UTILISER LE PLUGIN SCSS PHP	41
9.3	SIMPLIFIER L'IMPLEMENTATION DES LAYOUTS GALA	42
10.	UN LAYOUT RESPONSIVE	43
10.1	PRINCIPES DE DESIGN RESPONSIVE	43
10.2	MISE EN ŒUVRE	44
10.2.1	LES BREAKPOINTS	44
10.2.2	LES LAYOUTS	44
10.2.3	LES MEDIAS QUERIES	46
10.2.4	LA CONFIGURATION	47

1. INTRODUCTION

Les chapitres qui composent ce guide ont pour but de décrire une méthode de création d'un squelette Z dont le layout utilise la norme Gala. Ce guide propose une liste d'étapes simples permettant de réussir sans trop de problèmes son squelette Z.

2. LE FRAMEWORK Z V2

Ce chapitre a pour but de rappeler les concepts principaux du framework Z. Les informations fournies ne concernent que la version 2 du framework Z basée sur le plugin Z-Core.

2.1 Introduction

En premier lieu, il est nécessaire de lire ou de relire les articles sur SPIP-Contrib :

- [Le Squelette Zpip](#), qui reprend les concepts de base du framework dans sa version v1 ;
- [Différence entre Zcore et Zpip v1](#), qui explique la différence entre Zpip, le squelette Z de base qui utilise le framework Z v1 et Z-Core le plugin fournissant les mécanismes du framework Z v2.

et l'article [Framework Z](#) sur Nursit qui présente le framework Z dans sa version v2 utilisé par le squelette Spipr-dist.

Les rappels qui sont faits dans ce chapitre s'inspirent fortement de ces articles fondateurs mais sont reformulés ou précisés sous une forme différente qui me permet personnellement de mieux m'y retrouver.

2.2 Objectifs du framework Z

Le framework Z permet :

- de séparer le layout du reste des squelettes et ainsi de limiter la duplication du code et de simplifier sa maintenance,
- de proposer un mécanisme de thèmes incluant des CSS (et éventuellement un layout spécifique).
- de faire varier la composition du contenu d'un type d'objet,

Ces mécanismes sont supportés par le plugin Z-Core.

2.3 Organisation et génération d'une page HTML

Une page HTML se compose toujours de trois éléments de base, à savoir :

- une balise `<html>` englobante précédée en général d'un doctype,
- une balise `<head>` incluse contenant les meta, CSS et autres inclusions JS,
- une balise `<body>` incluse contenant les éléments visibles de la page.

Le framework Z permet la construction de toute page HTML suivant ce modèle en imposant une organisation standard des éléments structurant de la page.

En premier lieu, Z-Core embarque un fichier `page.html` à la racine qui est appelé sur chaque affichage d'une page. Ce fichier ne contient qu'une inclusion, celle du fichier `structure.html` auquel on passe les paramètres d'environnement, le type de page `{type-page}` (ou sommaire par défaut) et éventuellement une composition.

C'est donc le fichier `structure.html` intégré à la racine de Z-Core qui construit la page HTML de façon standard en incluant successivement dans la balise `<html>` - précédée d'un doctype - les éléments suivants :

- un head propre à chaque page pour des CSS particulières par l'inclusion d'un fichier `head/{type-page}.html` ;
- un head commun à toutes les pages par l'inclusion d'un fichier `inclure/head.html` ;
- un head propre à chaque page pour des JS spécifiques par l'inclusion d'un fichier `head_js/{type-page}.html` ;
- un layout unique par l'inclusion du fichier `body.html`.

Le mécanisme Z de construction d'une page HTML impose donc une organisation physique du squelette Z en dossiers et fichiers. A ce stade, un squelette Z doit avoir l'organisation minimale suivante :

- un `body.html` à la racine du squelette¹ ;
- un dossier `head/` ;
- un dossier `head_js/` ;
- un dossier `inclure/` avec un fichier `head.html` contenant les CSS et autres données communes incluses systématiquement dans le head de chaque page².

2.4 Layout unique, blocs Z et fichiers dist

Par défaut, le framework Z applique un layout unique à toutes les pages : le fichier `body.html`, colonne vertébrale d'un squelette Z. C'est lui qui va définir l'agencement et l'identité des blocs principaux de chaque page dans la balise `<body>`. Ci-dessous un exemple de `body.html` issu de Zpip-dist.

```
<body>
  <div class="page">
    <div class="header" id="header">
      <INCLUDE{fond=header/#ENV{type-page},env} />
    </div>
    <div class="nav hornav clearfix" id="nav">
      <INCLUDE{fond=inclure/nav,env} />
    </div>

    <div class="container">
      <div class="content" id="content">
        <INCLUDE{fond=content/#ENV{type-page},env} />
      </div>

      <div class="extral col" id="extral">
```

¹ Z-Core embarque aussi un fichier `body.html` très rudimentaire. Il convient toujours de le redéfinir pour les besoins du squelette.

² Z-Core embarque aussi un fichier `inclure/head.html` plutôt destiné à Zpip-dist. Il est donc presque toujours nécessaire de le redéfinir pour les besoins du squelette.

```

        <INCLUDE{fond=extra1/#ENV{type-page},env} />
    </div>
    <div class="extra2 col" id="extra2">
        <INCLUDE{fond=extra2/#ENV{type-page},env} />
    </div>
</div>

<div class="footer" id="footer">
    <INCLUDE{fond=footer/#ENV{type-page},env} />
</div>
</div>
</body>

```

On distingue dans cet exemple que le body peut contenir des blocs Z comme le header, le contenu ou le footer et des blocs « non-Z » comme la barre de navigation d'id `nav`. Un bloc Z se reconnaît par l'inclusion d'un squelette `${nom-bloc}/${type-page}.html` à l'intérieur du bloc nommé défini en général par une balise `<div>`. Même si le nommage des blocs tend à se normaliser il est possible d'utiliser les noms que l'on souhaite. La liste des blocs Z est en outre définie dans une variable globale incluse dans le fichier d'options du squelette. Par exemple, pour Zpip-dist, le fichier d'options contient l'instruction suivante :

```

if (!isset($GLOBALS['z_blocs']) OR !$GLOBALS['z_blocs'])
    $GLOBALS['z_blocs'] =
    array('content', 'extra1', 'extra2', 'head', 'head_js', 'header', 'footer');

```

Lors de l'affichage d'une page `${type-page}`, Z-Core essaye de trouver chacune des inclusions afin de calculer le contenu de la balise `<body>`. Si on considère l'exemple de Zpip-dist dont le body contient 5 blocs Z et qu'on rajoute les deux head comme définis dans le chapitre précédent, une page est construite à partir de 7 inclusions spécifiques. C'est parfois un reproche que l'on fait à Z, avis que je ne partage pas, car au contraire, l'organisation a priori modulaire du framework Z permet de limiter l'écriture du code.

En outre, Z-Core propose un mécanisme qui, si le fichier `${nom-bloc}/${type-page}.html` n'existe pas, recherche un fichier `${nom-bloc}/dist.html`. Ces fichiers `dist.html` sont donc des squelettes communs réutilisables pour tout ou partie des pages du site; c'est en particulier souvent le cas pour l'en-tête et le pied de page du site ainsi que dans une moindre mesure pour les blocs secondaires mais leur utilisation n'a pas de sens pour le corps de la page.

Par contre, Z-Core renverra une erreur si aucun fichier ni `${nom-bloc}/${type-page}.html` ni `dist.html` n'est trouvé. Il est donc nécessaire de créer des fichiers `dist.html` pour chaque bloc sauf le corps y compris pour les deux head même si ceux-ci sont vides. A ce stade, on peut compléter l'organisation minimale d'un squelette Z :

- un `body.html` à la racine du squelette qui définit la liste des blocs Z ;
- un dossier `head/` avec à minima un fichier `dist.html` vide (sinon on utilise celui fourni par Z-Core) ;
- un dossier `head_js/` avec à minima un fichier `dist.html` vide (sinon on utilise celui fourni par Z-Core) ;
- un dossier `inclure/` avec un fichier `head.html` ;
- un dossier par bloc Z sauf le corps avec à minima un fichier `dist.html` vide ;
- un dossier pour le bloc Z du corps avec à minima un fichier `sommaire.html` pour la page d'accueil.

2.5 Retour sur le head

La balise `<head>` est un élément structurel (au même titre que les tags `<html>` ou `<body>`) d'un document HTML. Elle contient les informations essentielles à la description du document qui sera affiché dans le corps de la page, à savoir :

- le titre du document (inclus dans la balise `<title>`) ;
- les références aux feuilles de style utilisées par le document ;
- les références aux javascripts utilisés par le document ;
- les balises `<meta>` pouvant spécifier l'auteur, la description du contenu, l'encodage et de nombreuses autres meta-données.

Certaines de ces informations sont communes à toutes les pages et d'autres spécifiques à une page donnée. Le framework Z propose une structuration des informations communes et spécifiques en imposant trois niveaux de head appelés successivement par le fichier `structure.html`, à savoir, `head/${type-page}.html`, `includure/head.html` et `head_js/${type-page}.html`.

2.5.1 Le fichier `head/${type-page}.html`

Ce fichier est appelé spécifiquement pour une page donnée `${type-page}`. Par défaut, si le fichier spécifique à la page n'existe pas Z-Core se rabat sur le fichier `head/dist.html`. Classiquement ce head permet d'insérer avant les données communes aux pages des informations spécifiques comme le titre de la page, la description de l'article pour une page article et parfois des CSS spécifiques. Cette inclusion est une sorte de pre-head.

L'environnement est passé à ce fichier.

2.5.2 Le fichier `includure/head.html`

Ce fichier est appelé systématiquement par Z-Core à chaque construction d'une page. Il est censé contenir les inclusions communes à toutes les pages. On y retrouve donc classiquement :

- les meta permettant de déclarer le « generator » et le « charset » du site ;
- le lien vers le flux RSS si disponible ;
- les CSS communes classées dans un ordre précis que l'on discutera plus avant dans cet article ;
- les balises `#INSERT_HEAD_CSS` et `#INSERT_HEAD` pour autoriser l'insertion de CSS ou de JS par les plugins tiers ;
- les éventuelles feuilles de styles nécessaires aux thèmes voire à une personnalisation propre ;
- le favicon.

Cette liste n'est pas exhaustive mais est représentative du contenu de ce fichier. L'ordre dans lequel sont positionnées les instructions est bien entendu primordial. Pour Zpip-dist, le contenu du fichier est le suivant:

```

[(<#REM)
    Head standard de toutes les pages ; les elements specifiques
    (title, description) figurent dans le squelette article.html
]
[(<#REM) Preciser le charset ]
<meta http-equiv="Content-Type" content="text/html; charset=<#CHARSET" />

[(<#REM) Fierement fabrique avec SPIP ]
<meta name="generator" content="SPIP[ (<#SPIP_VERSION)]" />

[(<#REM) Lien vers le flux RSS du site ]
<link rel="alternate" type="application/rss+xml" title="<:syndiquer_site:>" href="<#URL_PAGE{backend}>" />

[<link rel="stylesheet" href="<#CSS{css/reset.css}>" type="text/css" />]
[<link rel="stylesheet" href="<#CSS{css/font.css}>" type="text/css" />]
[<link rel="stylesheet" href="<#CSS{css/typo.css}>" type="text/css" />]
[<link rel="stylesheet" href="<#CSS{css/box.css}>" type="text/css" />]
[<link rel="stylesheet" href="<#CSS{css/box_skins.css}>" type="text/css" />]
[<link rel="stylesheet" href="<#CSS{css/form.css}>" type="text/css" />]

[(<#REM) Balise permettant aux plugins d'insérer des fichiers CSS ]
#INSERT_HEAD_CSS

[<link rel="stylesheet" href="<#CSS{css/spip.css}>" type="text/css" />]
[<link rel="stylesheet" href="<#CSS{css/spip.form.css}>" type="text/css" />]
[<link rel="stylesheet" href="<#CSS{css/spip.comment.css}>" type="text/css" />]
[<link rel="stylesheet" href="<#CSS{css/spip.list.css}>" type="text/css" />]
[<link rel="stylesheet" href="<#CSS{css/spip.petition.css}>" type="text/css" />]
[<link rel="stylesheet" href="<#CSS{css/spip.pagination.css}>" type="text/css" />]
[<link rel="stylesheet" href="<#CSS{css/spip.portfolio.css}>" type="text/css" />]
[<link rel="stylesheet" href="<#CSS{css/clear.css}>" type="text/css" />]
[<link rel="stylesheet" href="<#CSS{css/grid.css}>" type="text/css" />]
[<link rel="stylesheet" href="<#CSS{css/button.css}>" type="text/css" />]
[<link rel="stylesheet" href="<#CSS{css/layout.css}>" type="text/css" />]

[(<#REM) Feuille de style du theme ]
[<link rel="stylesheet" href="<#CSS{css/theme.css}>" type="text/css" />]

[(<#REM) Feuille de style CSS pour l'impression, n'existe pas par default ]
[<link rel="stylesheet" href="<#CSS{css/print.css}>" type="text/css" media="print" />]

[(<#CHEMIN{inc-theme-head.html}>|oui)
#INCLUDE{fond=inc-theme-head,env}
]

[(<#REM) Feuille de style personnalisee pour surcharger les precedentes
    A noter par default cette css n'existe pas
]
[<link rel="stylesheet" href="<#CSS{css/perso.css}>" type="text/css" />]

[(<#REM) Balise permettant aux plugins d'insérer des appels javascript ou css ;
    C'est ici que SPIP va insérer l'appel de la librairie jQuery
    Et appeler a la fin compacte_head pour agréger et compacter tout le head dans des fichiers statiques
    si l'option est cochée dans Configuration
]
#INSERT_HEAD

[<link rel="icon" type="image/x-icon" href="<#CHEMIN{favicon.ico}>" />]
[<link rel="shortcut icon" type="image/x-icon" href="<#CHEMIN{favicon.ico}>" />]]

```

Il faut noter que l'environnement n'est pas fourni à ce fichier de head pour des raisons de performance, il n'est donc pas possible d'y introduire une personnalisation liée à une variable d'environnement; il faut passer par une configuration ou le faire dans les autres fichiers de head.

2.5.3 Le fichier `head_js/${type-page}.html`

Ce fichier est lui aussi appelé spécifiquement pour une page donnée `${type-page}`. Par défaut, si le fichier spécifique à la page n'existe pas Z-Core se rabat sur le fichier `head/dist.html`. Classiquement ce head permet d'insérer après les données communes aux pages des informations spécifiques, en particulier les scripts JS. Cette inclusion est une sorte de post-head.

L'environnement est passé à ce fichier.

2.6 Pages automatiques et échafaudage

A compléter.

2.7 Thèmes

A compléter.

3. ETAPE 1 : UNE PAGE SOMMAIRE AVEC LE LAYOUT GALA 23

La première étape du guide Z consiste à créer un squelette minimal basé sur un layout Gala sans aucune modification ni des blocs HTML ni des styles originels.

3.1 Introduction

A chaque fois que j'ai besoin de créer un squelette «*from scratch*», même très simple, je me tourne vers le framework Z qui, une fois en place, permet facilement d'ajouter des pages, de modifier les styles, voire de modifier le layout d'une page donnée. Alors pourquoi ne pas prendre un squelette existant ? C'est souvent le cas avec Zpip-dist v2, sauf quand je souhaite un layout fluide comme le layout Gala 23. Et là je bloque pour modifier mon Zpip-dist ou créer un thème qui me conviendrait.

Alors j'ai décidé de reprendre la mécanique depuis le début et de décortiquer chaque concept qui sous-tend le framework Z afin d'être capable de construire mon propre jeu de squelettes Z basé sur les layouts Gala. Et pour éviter de me reposer éternellement les mêmes questions, j'ai noté toutes les étapes de cette aventure afin d'en faire un guide.

Au préalable, il est conseillé de lire l'article [Comprendre le framework Z](#) qui fait un tour d'horizon des concepts Z et de leur implémentation dans Z-Core. En effet, le point de départ de mon nouveau squelette est bien le framework Z v2 bâti à partir du plugin Z-Core.

3.2 Création du paquet.xml

Comme je distribue toujours mes squelettes sous forme de plugin pour plus simplicité, je commence par créer le `paquet.xml` puisque la compatibilité projetée du squelette est SPIP ≥ 3 . La seule particularité du fichier réside dans le nécessaire du plugin Z-Core qui est comme indiqué plus haut, la brique de base du modèle Z.

Je choisis le préfixe **zgala** et le nom **Z-Gala** pour mon plugin. Je termine cette opération en créant le fichier de langue `lang/paquet-zgala_fr.php` pour traduire le slogan et la description du squelette.

3.3 Déclaration des blocs Z

A la lecture du contenu de la balise `<body>` d'un layout Gala quelconque, on distingue clairement 5 blocs de base qui sont appelés à devenir des blocs Z. Ce sont les blocs « header », « content », « navigation », « extra » et « footer ».

A ces blocs, il faut rajouter les blocs Z qui composent la balise `<head>` c'est-à-dire « head » et « head_js ». On crée le fichier d'options du squelette, `zgala_options.php`, dans lequel on insère le code PHP suivant de déclarations des 7 blocs Z :

```
<?php
// Définition des blocs Z utilisés par le squelette
$GLOBALS['z_blocs'] =
array('content', 'navigation', 'extra', 'head', 'head_js', 'header', 'footer');
```

3.4 Création des répertoires

Chaque bloc Z nécessite un dossier à la racine du squelette afin de recevoir un squelette propre à une page ou le squelette générique `dist.html`. On crée donc les répertoires correspondant, à savoir :

- header/
- content/
- navigation/
- extra/
- footer/
- head/
- head_js/

Comme il va falloir très vite créer le fichier commun du « head », `inclure/head.html` et quelques styles on ajoute aussi les dossiers suivants à la racine du squelette :

- inclure/
- css/

3.5 Mise en place du layout unique

Pierre angulaire du framework Z, le layout unique est matérialisé par le fichier `body.html` à la racine du squelette. Comme je souhaite un layout Gala 23, je copie le contenu du `body.html` de ce layout Gala dans le `body.html` de mon squelette. Il suffit alors remplacer le contenu des balises `<div>` qui portent les blocs Z par les inclure adéquats et on obtient le contenu suivant :

```
<body>
  <div class="container" id="container">
    <div class="header" id="header">
      <INCLUDE{fond=header/#ENV{type-page},env} />
    </div> <!-- .HEADER -->

    <div class="wrapper" id="wrapper">
      <div class="content" id="content">
        <INCLUDE{fond=content/#ENV{type-page},env} />
      </div> <!-- .CONTENT -->
    </div> <!-- .WRAPPER -->

    <div class="navigation col" id="navigation">
      <INCLUDE{fond=navigation/#ENV{type-page},env} />
    </div> <!-- .ASIDE -->
    <div class="extra col" id="extra">
      <INCLUDE{fond=extra/#ENV{type-page},env} />
    </div> <!-- .EXTRA -->

    <div class="footer" id="footer">
      <INCLUDE{fond=footer/#ENV{type-page},env} />
    </div> <!-- .FOOTER -->
  </div> <!-- .CONTAINER -->
</body>
```

On notera que j'ai dupliqué le nom des « id » en classe afin de déclarer plus tard des styles moins spécifiques et plus facilement surchargeables.

3.6 Mise en place du « head »

Le contenu standard de la balise `<body>` étant défini, on peut s'attaquer au contenu de la balise `<head>`. Dans un premier temps on se limite à un contenu basique:

- une meta pour le *charset* et une autre pour le *generator*; ces meta sont communes à toutes les pages,
- une feuille de styles unique, `css/zgala.css`, pour y intégrer les déclarations du layout Gala 23 et ce pour toutes les pages,
- le titre de la page et une meta pour sa description.

Pour les deux premiers points on crée le fichier `inclure/head.html` et on y intègre les instructions suivantes:

```
[ (#REM)
  Head standard de toutes les pages ; les éléments spécifiques
  (title, description) figurent dans le squelette head/${type-page}.html
  ou par défaut dans head/dist.html.
  Ce head surcharge celui de Z-Core.
]
[ (#REM) Préciser le charset ]
<meta http-equiv="Content-Type" content="text/html; charset=#CHARSET" />

[ (#REM) Fièrement fabriqué avec SPIP ]
<meta name="generator" content="SPIP[ (#SPIP_VERSION)]" />

[ (#REM) 1. Base CSS ]
[<link rel="stylesheet" href="( #CSS{css/zgala.css}|direction_css)" type="text/css"
/>]

[ (#REM) 2. Style des plugins nécessités ou utilisés par le plugins ]

[ (#REM) Balise permettant aux plugins d'insérer des fichiers CSS ]
#INSERT_HEAD_CSS

[ (#REM) Balise permettant aux plugins d'insérer des appels javascript ou css ;
  C'est ici que SPIP va insérer l'appel de la librairie jQuery
  Et appeler à la fin compacte_head pour agréger et compacter tout le head dans
  des fichiers statiques
  si l'option est cochée dans la configuration
]
#INSERT_HEAD
```

Ensuite on crée la feuille de styles `layout.css` dans le dossier `css/`. On y insère tel quel le bloc de style du layout Gala 23 ce qui donne :

```
html,body{margin:0;padding:0}
body{font: 76% arial,sans-serif}
p{margin:0 10px 10px}
a{display:block;color: #981793;padding:10px}
div#header h1{height:80px;line-height:80px;margin:0;
padding-left:10px;background: #EEE;color: #79B30B}
div#content p{line-height:1.4}
div#navigation{background:#B9CAFF}
div#extra{background:#FF8539}
div#footer{background: #333;color: #FFF}
div#footer p{margin:0;padding:5px 10px}

div#wrapper{float:left;width:100%;margin-left:-200px}
```

```
div#content{margin-left:200px}
div#navigation{float:right;width:200px}
div#extra{float:right;clear:right;width:200px}
```

Enfin, on termine la mise en place du « head » en créant le fichier par défaut du « pre-head », à savoir, `head/dist.html` dans lequel on insère un titre et une description générique et le fichier par défaut du « post-head », `head_js/dist.html` que l'on laisse vide.

Pour information, le contenu du fichier `head/dist.html` est le suivant:

```
<title>[ (#NOM_SITE_SPIP|textebrut) ]</title>
[<meta name="description" content="( #DESCRIPTIF_SITE_SPIP|couper{150}|textebrut) "
/>]
```

3.7 Mise en place de la page d'accueil

Le but de cette phase étant d'afficher une page d'accueil reflétant exactement la page du layout Gala 23, il nous reste à créer les blocs Z constitutifs de la balise `<body>`.

Pour le corps de la page, soit le bloc « content », on crée un fichier `content/sommaire.html` puisque c'est le nom de la page d'accueil sous SPIP. Pour les autres blocs, « header », « footer », « navigation » et « extra », on ne crée que les fichiers par défaut soit les `dist.html`. Pour chacun de ces blocs on insère le contenu d'exemple des layouts Gala. En recalculant la page d'accueil du site on reconnaît la page classique Gala pour le layout 23 !

4. ETAPE 2 : CHANGER DE LAYOUT GALA

Lors de cette deuxième étape, le squelette Z minimal issu de l'étape 1 va être enrichi de façon à permettre le choix de tout layout Gala d'une façon simple.

4.1 Quelles solutions pour paramétrer le layout ?

Le layout du squelette est par définition commun à l'ensemble des pages. Les styles associés doivent donc être inclus dans le fichier « head » commun à savoir, `inclure/head.html`. Or, nous avons vu dans [Comprendre le framework Z](#) que ce fichier, pour des raisons de performance ne reçoit pas l'environnement. Il est donc exclu d'utiliser un paramètre url pour choisir le layout.

Le framework Z permet la création de thèmes. Un thème peut recevoir des styles et un body spécifique si besoin. C'est donc une première solution que nous allons étudier.

Si le fichier « head » commun ne reçoit pas l'environnement il est quand même possible de le paramétrer en utilisant de la configuration propre au squelette stockée dans une meta spécifique. C'est une deuxième solution que nous envisagerons par la suite.

4.2 Des thèmes pour chaque layout

En premier lieu on crée un dossier `themes/` à la racine du site et un sous-dossier `gala24/` qui va accueillir notre thème Z-Gala pour le layout Gala 24. J'ai choisi un thème équivalent au layout 23 mais avec les colonnes navigation et extra à gauche.

On commence par créer le `paquet.xml` qui va permettre de décrire le thème et permettre si besoin d'installer le thème comme un plugin. Le `paquet.xml` doit contenir les éléments habituels de la balise `<paquet>` comme le préfixe, la version, la compatibilité SPIP, le logo et la catégorie. Certains de ces éléments sont contraints, à savoir:

- le préfixe doit commencer par `theme_` ;
- le logo doit s'appeler `vignette` et être un gif, un jpeg ou un png positionné à la racine du thème ;
- et la catégorie doit contenir la valeur prédéfinie `theme`.

Pour compléter le `paquet.xml` j'ajoute son nom, les auteurs, une licence et une balise `<utilise>` qui va permettre de préciser à Zen-Garden que le thème est compatible avec le squelette Z-Gala. Il est possible aussi d'ajouter un slogan et une description en utilisant des items de langue comme pour tout plugin.

On ajoute, à la racine du thème, son logo que l'on récupère sur la [page des layouts Gala](#). Pour changer le layout Gala 23 en 24 il suffit, en l'état du squelette Z-Gala, de remplacer les styles inclus dans le fichier `css/zgala.css`. On crée donc dans le dossier `css/` du thème un fichier `zgala.css` dans lequel on insère les styles du layout Gala 24.

Pour terminer, on ajoute un `define` dans le fichier d'options du squelette Z-Gala pour indiquer à Zen-Garden de toujours filtrer les thèmes compatibles à Z-Gala quand celui-ci est actif.

En se rendant sur la page de Zen-Gargen dans l'espace privé, une galerie de thèmes - réduite à un thème – s’affiche avec la vignette du layout Gala 24. En cliquant sur « Apercevoir », on affiche la page sommaire avec le layout Gala 24 qui remplace le layout 23 utilisé par défaut. On peut aussi activer le switcher de thèmes dans le public et obtenir ainsi sur la page sommaire une liste des thèmes en haut à gauche de la page.

L'utilisation des thèmes pour changer le layout est donc une solution qui fonctionne parfaitement. On pourrait créer les 40 thèmes Gala de cette manière. Maintenant, il est un peu fastidieux de créer un thème, à savoir, une vignette, un `paquet.xml` et un fichier de langue uniquement pour changer quelques styles essentiellement dédiés à la structuration des pages. De plus, en utilisant un thème pour changer le layout, il n'est plus possible d'utiliser un thème pour modifier les couleurs ou autres décorations du squelette ce qui rend notre squelette Z-Gala beaucoup moins attrayant.

4.3 Une configuration du layout

L'idée est de considérer que le layout est un élément stable du squelette et qu'il faut laisser les couleurs et autres décorations aux thèmes. Il nous reste donc la configuration de plugin pour introduire un choix de layout.

Dans un premier temps on récupère l'archive de tous les layouts Gala. Comme lors de l'étape 1, on extrait de chaque fichier de layout uniquement la partie des styles que l'on conserve dans un fichier nommé `layout_galann.css` où `nn` représente le numéro du layout de 01 à 40. Tous les fichiers ainsi créés sont stockés dans le dossier `css/layouts/` du squelette.

Pour initialiser une configuration on va créer un formulaire de configuration simpliste qui nous autorisera à saisir un numéro de 1 à 40 identifiant le layout à utiliser. On pourra plus tard améliorer la présentation pour simuler une galerie de layouts plus visuelle. Je ne vais pas m'attarder sur la construction dudit formulaire mais pour mémoire il faut créer les fichiers suivants :

- `prive/squelettes/contenu/configurer_zgala.html`, pour matérialiser la page de configuration ;
- les fichiers `formulaires/configurer_zgala.html` et `formulaires/configurer_zgala.php` pour construire le formulaire sachant que le fichier PHP ne sert qu'à contrôler la valeur de la saisie entre 1 et 40;
- le fichier de langue `lang/zgala_fr.php` pour y insérer les quelques items de langue nécessaires.

Il est maintenant possible d'utiliser la configuration du layout dans l'appel du fichier de CSS dans le « head » commun. L'inclusion du fichier `zgala.css` est remplacé par :

```
[ (#REM) 1. Base CSS ]
[ <link rel="stylesheet" href="( #CSS{ [css/layouts/layout_gala( #CONFIG{zgala/layout, 23} |str_pad(2, 0, 0)).css} ) |direction_css)" type="text/css" /> ]
```

On notera le padding à deux caractères avec le caractère «0» pour constituer le nom exact du fichier de layout. A ce stade il ne reste plus qu'à supprimer le fichier `css/zgala.css` qui est devenu obsolète et à conclure l'étape en passant le plugin en version 0.2.0.

Pour finir, rendez-vous sur la page de configuration du squelette, choisissez un layout différent de 23, par exemple, le 7 qui est en 3 colonnes fixes et affichez la page sommaire en la recalculant (attention, le thème 24 défini au début de l'article ne doit pas être encore actif).

5. ETAPE 3 : ENRICHIR LES STYLES ET FINALISER LE BODY

Lors de cette troisième étape, le squelette Z issu de l'étape 2 va être structurellement amélioré avec une barre de navigation principale et un fil d'Ariane. Les blocs seront renommés pour plus de sémantique et une base typographique sera incluse.

5.1 Renommer des blocs du body

En premier lieu, on effectue quelques renommages de blocs afin d'être plus conforme à la pensée unique actuelle et aussi à Spipr-dist. Ainsi :

- le bloc « container » du layout gala est renommé en `page`,
- et le bloc Z « navigation » en `aside` qui est plus HTML5.

En outre, afin d'être moins spécifique dans les styles on va utiliser les classes plutôt que les id associés aux balises `<div>` de blocs. Pour mettre en place ces changements il faut :

- modifier le `body.html` en renommant les attributs « id » et « class » comme défini ci-dessus ;
- réitérer le renommage dans l'ensemble des fichiers de styles des layouts Gala en modifiant les `div#xxx` en `.xxx`;
- renommer le bloc Z « navigation » en `aside` dans l'instruction `define` du fichier d'options;
- et renommer le dossier `navigation/` en `aside/`.

En recalculant la page, on retrouve le layout Gala classique sans aucune modification. Maintenant, pour faire évoluer notre squelette vers un squelette SPIP stylé il est nécessaire de développer les styles de base de Gala.

5.2 TinyTypo pour la typographie

Inutile de repartir de zéro quand il existe déjà d'excellentes bases typographiques. Celle que nous allons utiliser est **TinyTypo**, créée par Tetue sous licence MIT et disponible sur la page [TinyTypo](#). TinyTypo est une base CSS qui fait le minimum typographique nécessaire à la rédaction de documents web. Comme le dit son auteure :

Facile à utiliser, elle vient compléter votre reset habituel pour servir de base à vos propres styles, sans fioriture, de façon non intrusive. Faite de HTML et CSS, elle est simple et robuste, très légère (sans image, ni fonte, ni JavaScript). Elle est personnalisable en LESS.

TinyTypo est respectueuse des standards W3C, naturellement responsive, prévue pour l'écran et l'impression, pour le multilinguisme et soucieuse d'accessibilité.

Comme l'indique le manuel d'installation, il suffit d'ajouter les fichiers listés ci-après dans le dossier `css/` du squelette Z-Gala et d'inclure dans le fichier `inclure/head.html` l'appel aux feuilles CSS dans l'ordre suivant :

- `reset.css`
- `clear.css`
- `font.css`

- typo.css
- links.css
- media.css

Ces inclusions sont insérées en premier dans la liste des styles avant celle du layout qui est déplacé après la balise `#INSERT_HEAD_CSS` qui permet d'intégrer des styles propres à un plugin. Pour conclure avec TinyTypo et aussi tester la typographie on crée une page de test nommée `tinytypo` en recopiant le contenu des balises `<section>` de celle fournie avec le package TinyTypo et en les insérant dans un fichier `content/tinytypo.html`. Le framework Z fera la reste !

5.3 Nettoyer les layouts Gala

Il faut se rappeler que les styles Gala comportent, outre ceux des blocs proprement dits, quelques styles de typographie pour les balises `<h1>`, `<a>` et `<p>` et une instruction de reset sur les balises `<html>` et `<body>`. Il faut donc supprimer ces styles de tous les fichiers de layout afin de ne laisser que les éléments structurels.

Si l'on veut conserver pour l'instant la coloration du fond de certains blocs, il faut conserver les quelques styles correspondant. Cependant, comme ils sont identiques dans tous les fichiers de layout, on les factorise dans un fichier `css/theme.css` qui préfigure une notion que nous discuterons dans une prochaine étape. En rafraichissant la page sommaire on obtient notre layout Gala avec la nouvelle typographie.

La partie des styles inclus dans le fichier `inclure/head.html` s'est bien enrichie comme le montre le code suivant :

```
[ (#REM) Base CSS - TinyTypo de Tetue http://tinytypo.tetue.net/ ]
[<link rel="stylesheet" href="#CSS{css/reset.css}" type="text/css" />]
[<link rel="stylesheet" href="#CSS{css/clear.css}" type="text/css" />]
[<link rel="stylesheet" href="#CSS{css/font.css}" type="text/css" />]
[<link rel="stylesheet" href="#CSS{css/typo.css}" type="text/css" />]
[<link rel="stylesheet" href="#CSS{css/links.css}" type="text/css" />]
[<link rel="stylesheet" href="#CSS{css/media.css}" type="text/css" />]

[ (#REM) Balise permettant aux plugins d'insérer des fichiers CSS ]
#INSERT_HEAD_CSS

[ (#REM) Layout ]
[<link rel="stylesheet" href="#CSS{[css/layouts/layout_gala(#CONFIG{zgala/layout,
23}|str_pad{2, 0, 0}).css]}|direction_css)" type="text/css" />]

[ (#REM) Feuille de style du theme ]
[<link rel="stylesheet" href="#CSS{css/theme.css}" type="text/css" />]
```

5.4 ooCSS pour les boîtes

Le plugin Z-Core fournit, outre le framework Z, un système standard de boîtes dont le modèle est issu des travaux [ooCSS](#) de Nicole Sullivan. Ce système propose, d'une part, des balises pour créer les boîtes et d'autre part, des CSS pour les styler.

Pour mémoire les balises sont :

- `#BOITE_OUVRIER`, pour initialiser la boîte ;

- `#BOITE_PIED`, pour insérer un pied à la boite (facultatif) ;
- `#BOITE_FERMER`, pour clore la boite.

Deux fichiers CSS sont associés à ce système de boites, à savoir :

- `box.css`, qui contient les styles des conteneurs et autres éléments structurels ;
- `box_skins.css`, qui contient les styles de personnalisation des différentes boites pour en faire une note, un message d'erreur ou d'information...

Ces fichiers sont inclus dans le dossier `css/` du plugin Z-Core, de fait, comme celui-ci est nécessité par le squelette Z-Gala, il n'est nul besoin de les recopier dans le squelette ; il suffit de les inclure via le fichier `inclure/head.html`. Les inclusions sont positionnées à la suite de celles de TinyTypo.

Pour compléter ce système de boites, un troisième fichier CSS, `css/form.css`, est inclus à la suite des deux précédents, afin de donner un style minimum aux éléments d'un formulaire. Ce fichier est fourni par la dist de SPIP ce qui permet de l'inclure dans Z-Gala sans avoir à le recopier.

Le fichier `inclure/head.html` s'enrichit donc des lignes suivantes:

```
[ (#REM) CSS des boites de messages et des popups ]
[<link rel="stylesheet" href="#CSS{css/box.css}" type="text/css" />]
[<link rel="stylesheet" href="#CSS{css/box_skins.css}" type="text/css" />]
[ (#REM) CSS très basiques pour les éléments de formulaires ]
[<link rel="stylesheet" href="#CSS{css/form.css}" type="text/css" />]
```

Pour tester le rendu des boites, on crée une page `boite` en ajoutant un unique fichier `content/boite.html` avec des appels divers aux balises de boite. Tout comme la page `tinytypo`, le framework Z fera le reste.

5.5 Améliorer les boites

Le rendu de la page `boite` n'est pas optimal. Nous allons nous y pencher.

Avec l'inclusion du fichier `css/form.css` on obtient des boites avec une double bordure ce qui est assez laid. Si l'on supprime cette inclusion on se retrouve toujours avec une bordure externe, cette fois de couleur noire mais plus de fond en dehors du bloc de classe `.inner`. C'est mieux mais pas encore satisfaisant.

Si l'on y regarde de plus près, cette bordure est due à TinyTypo qui inclus aussi une classe `.box` qui vient en collision avec notre système de boite ooCSS. Le plus simple est donc de renommer la classe de TinyTypo qui est juste un encadré typographique en `.tinybox`, mais on introduit un fork dans TinyTypo qu'il faudra maintenir en parallèle.

Coté couleur, je souhaiterais me rapprocher de celles assez standard et de bon goût des alertes de Bootstrap. Je recopie donc les fichiers CSS des boites de Z-Core ainsi que les images dans le dossier `css/` du squelette Z-Gala afin de pouvoir les modifier à loisir :

- dans `css/box_skins.css`, pour les couleurs et pour les liens,
- et dans `css/box.css`, une touche d'arrondi en positionnant le radius à 4px pour la classe `.inner`.

Enfin, que faire de `css/form.css` ? Comme je ne souhaite plus que ce fichier modifie mon système de boîtes maintenant à mon goût, je supprime les styles en cause - je n'ai d'ailleurs pas compris pourquoi ces styles dans ce fichier - et je conserve les autres styles qui eux concernent bien les formulaires. J'avais préalablement recopié le fichier dans le dossier `css/` du squelette Z-Gala.

En recalculant la page `boite` on peut constater les modifications.

5.6 Ajouter un fil d'Ariane

Le fil d'Ariane est souvent inclus comme un élément du contenu d'une page, soit dans le bloc content. C'est pratique car on peut le personnaliser en fonction de la page mais d'un autre côté cela demande à le supprimer quand on ne l'utilise pas. Le squelette Spipr-dist a résolu ce problème en faisant du fil d'Ariane un bloc Z indépendant inclus dans le conteneur englobant le bloc Z « content ». On reproduit ce schéma pour notre squelette Z et pour cela :

- on ajoute le bloc « breadcrumb » dans la liste des bloc Z du fichier d'options en prenant bien soin de ne pas le positionner en premier car il faut toujours y laisser le **contenu de la page** pour traiter correctement les 401 et 404 ;
- on modifie le fichier `body.html` en ajoutant un appel à une inclusion propre au fil d'Ariane ;
- on crée un répertoire `breadcrumb/` et un fichier `breadcrumb/dist.html` avec un texte d'exemple comme pour les autres blocs. Le HTML est classiquement une liste non ordonnée dont la classe est positionnée à `.breadcrumb`. On notera que contrairement aux autres blocs Z ce bloc n'est pas inclus d'office au niveau du body dans un `<div>` au nom du bloc ;
- et bien entendu, on ajoute les styles adéquats pour le fil d'Ariane. Comme ces styles ne dépendent pas du layout on crée un fichier `css/breadcrumb.css` minimal avec un fond qui permet d'identifier le bloc et on l'appelle dans `includure/head.html` après l'inclusion du layout.

6. ETAPE 4 : MODERNISER LES CSS AVEC LESS

Lors de cette étape les CSS de TinyTypo, des boîtes de messages et du layout vont être passés en LESS pour profiter des capacités de paramétrage du langage.

6.1 Pourquoi du LESS

Le CSS est un langage déclaratif simple : la plupart du temps, on applique une valeur à une propriété. Mais de nos jours, avec le développement des attributs propriétaires Mozilla ou Webkit le code se trouve dupliqué en de nombreux endroits. En outre, il est impossible d’imbriquer des sélecteurs en CSS pour éviter les répétitions de code. Enfin, il est impossible de paramétrer les styles comme les couleurs par exemple et ainsi d'en changer facilement de façon cohérente dans l'ensemble du site.

Avec SPIP la paramétrage des CSS est déjà possible en utilisant les variables de configuration et des fichiers CSS dynamiques au format HTML. De même, Arno* avait aussi fait avancer l'écriture des CSS avec le plugin «CSS imbriqués». Néanmoins, il n'existait rien de standard.

C'est ainsi que sont nés les « pré-processeurs » LESS ou SASS, pour combler les manques du CSS et apporter aussi d'autres fonctionnalités de type programmation. Même si il est toujours possible de se passer de ces « pré-processeurs », j'ai décidé d'intégrer cet outil dans le squelette en imaginant qu'il apporterait une capacité de configuration utile dans le futur.

6.2 Intégrer les fichiers LESS de TinyTypo

TinyTypo est, par défaut, développée en LESS. Il suffit donc de récupérer les sources LESS et de les recopier dans un nouveau dossier `less/` à la racine du squelette Z-Gala. La compilation de ces fichiers fournira les fichiers CSS dans le dossier `css/`.

Les variables LESS de TinyTypo sont définies dans le fichier `var.less`. On le renomme en `var_tinytypo.less` en n'oubliant pas de le faire dans chacun des fichiers LESS de TinyTypo. En effet, au fur et à mesure du passage en LESS des différents éléments de style, on gardera chaque ensemble de variables dans un fichier séparé. Je ne sais pas si cela est la meilleure approche mais ça permettra dans un premier temps de distinguer l'utilisation des variables.

On continue ainsi de modifier légèrement TinyTypo mais cela reste très maîtrisable surtout quand on pense que seuls les fichiers LESS seront maintenant nécessaires.

6.3 Passer les styles de boîtes sous LESS

Pour migrer vers des sources en LESS nous allons procéder par étapes. La première sera d'utiliser un outil web [CSS 2 LESS](#) qui permet de créer un fichier LESS à partir d'un fichier CSS. Bien entendu, ce fichier n'est pas optimisé mais l'utilisation de ce service web va éviter de tâtonner à partir de notre CSS. Le service se contente de créer les règles imbriquées et certaines variables automatiquement.

6.3.1 Fichier `css/box.css`

On copie le contenu du fichier dans le bloc de gauche de la page de conversion du service « CSS 2 LESS » et on récupère le contenu LESS dans le bloc de droite. On crée un fichier `less/box.less` avec le contenu ainsi obtenu³. Ensuite, on liste les règles identiques et on les écrit sur une même ligne comme suit :

```
.tl, .tr {
  /* position/zoom required for IE7, 6, 5.5 */
  .top-corner(32000px, @box-corner-size);
}
```

Pour les règles vides, il faut insérer un commentaire afin qu'elles soient conservées suite à la compilation (mais est-ce vraiment utile ?) :

```
.bl, .br { /* pas de style particulier */ }
```

Ensuite, on crée une mixin avec deux paramètres que l'on nomme `.top-corner()` pour le bloc de 4 règles utilisés deux fois dans les classes `.tr` et `.tl` des boîtes de classe `.complex`. Ce n'est pas vital dans ce cas mais cela permet de se familiariser avec les concepts de LESS. Cette mixin ne produit pas de règle CSS `.top-corner`. On obtient le code suivant :

```
/* box de type complex */
.top-corner(@h, @w) {
  height:@h;
  margin-bottom:-@h;
  overflow:hidden;
  width:@w;
}
.complex {
  /* this zoom required for IE5.5 only*/
  *position:relative;
  *zoom:1;
  overflow:hidden;
  .tl, .tr {
    /* position/zoom required for IE7, 6, 5.5 */
    .top-corner(32000px, @box-corner-size);
  }
  .bl, .br { /* pas de style particulier */ }
  .top {
    height:@box-complex-top-height;
  }
  .bottom {
    height:@box-complex-bottom-height;
  }
}
```

Maintenant, le plus compliqué est de définir le paramétrage que l'on souhaite fournir à notre système de boîtes. Il faut éviter la tentation du « tout paramètre » et surtout détecter la cohérence entre deux valeurs pour savoir si elles doivent ou pas partager la même variable LESS. L'outil « CCS 2 LESS » fait déjà une partie du travail en détectant les valeurs identiques utilisées plusieurs fois avec la même propriété mais cela reste notoirement insuffisant.

³ On peut aussi utiliser un nom différent temporairement afin de comparer le fichier compilé avec le fichier existant et le renommer quand on est satisfait du résultat

Les variables ainsi définies sont isolées dans un fichier `less/var_box.less`, ce qui nous amène à rajouter une directive `@import` dans le fichier `less/box.less`.

6.3.2 Fichier `css/box_skins.css`

On renouvelle les opérations précédentes sur ce fichier. C'est bien entendu le plus intéressant car comme son nom l'indique il est destiné à personnaliser l'habillage des boîtes. On obtient un code propre et correctement paramétré. Je me suis principalement attaché à paramétrer les boîtes d'alerte, les autres étant moins utilisées dans l'espace public.

Pour les boîtes d'alerte, on reproduit la méthode de Bootstrap, à savoir, définir les couleurs du texte à partir de la couleur du fond. Pour cela les fonctions LESS `darken()` et `spin()` sont parfaitement adaptées. On fait de même avec la couleur du fond d'un lien lors de son survol en utilisant la fonction `lighten()`.

6.3.3 Conclusion

La liste des variables est assez importante et on aurait sans doute pu limiter un peu leur nombre surtout pour la structure des boîtes. Mais finalement ça ne coûte pas grand-chose du fait de la compilation.

6.4 Les layouts Gala en LESS

Il y a 40 layouts Gala que l'on peut scinder en 2 catégories : les layouts à 3 colonnes et les layouts à 2 colonnes. Pour chacune de ces catégories on distingue des sous-catégories selon que la structure est fixe, fluide ou hybride. On obtient ainsi :

○ Layouts à 3 colonnes :

- de 1 à 6, la page est fluide, toutes les colonnes sont fluides et exprimées en pourcentage ;
- de 7 à 12, la page est fixe, toutes les colonnes sont fixes et exprimées en pixels ;
- de 13 à 18, la page est fluide, les colonnes « aside » et « extra » sont fixes et exprimées en pixels ;
- de 19 à 22, la page est fluide, les colonnes « aside » et « extra » sont soit fixes et exprimées en pixels, soit fluides et exprimées en pourcentage.

○ Layouts à 2 colonnes :

- de 23 à 24, la page est fluide, les colonnes « aside » et « extra » sont fixes et exprimées en pixels ;
- de 25 à 26, la page est fluide, les colonnes « aside » et « extra » sont fluides aussi et exprimées en pourcentage ;
- de 27 à 28, la page est fluide, la colonne « content » prend toute la largeur et les colonnes « aside » et « extra » se répartissent la largeur sous « content » ;
- de 29 à 30, la page est fluide, la colonne « aside » est fluide aussi et exprimée en pourcentage et la colonne « extra » prend toute la largeur sous « content » ;
- de 31 à 32, la page est fluide, la colonne « aside » est fixe et exprimée en pixels et la colonne « extra » prend toute la largeur sous « content » ;

- de 33 à 34, la page est fixe, toutes les colonnes sont fixes et exprimées en pixels ;
- de 35 à 38, la page est fixe, toutes les colonnes sont fixes et exprimées en pixels et la colonne « extra » est sous la colonne « content » avec la même largeur ;
- de 39 à 40, la page est fixe, la colonne « content » prend toute la largeur et les colonnes « aside » et « extra » se répartissent la largeur sous « content ».

Quand on étudie la structure des CSS on imagine qu'il existe forcément une fonction qui permettrait de calculer l'ensemble des layouts à partir de paramètres comme la largeur des colonnes, la fluidité, la position des colonnes, etc. Mais en cherchant un peu elle apparaît bien vite très complexe.

Même avec les fonctionnalités « programmatiques » de LESS cette fonction semble très difficile à mettre en œuvre. Parfois on a l'impression qu'il faut factoriser le code à l'intérieur d'une même sous-catégorie et parfois il semble que ce soit la position des blocs qui permette la factorisation.

Cette factorisation n'étant pas essentielle pour l'instant, je me suis rabattu sur une structure avec un LESS par layout et une mixin de même nom, soit `.gala`, incluse dans un namespace spécifique au layout ce qui permet de la coder différemment pour chaque layout. Il y a donc beaucoup de duplication de code mais cela permet au moins de paramétrer la largeur des colonnes de chaque layout. De toute façon, notre stratégie de configuration du layout nécessite que nous ayons toujours un fichier CSS par layout.

Le fichier `var_layout.less` contient plus de paramètres que nécessaire afin d'assurer que chaque changement de configuration du layout puisse permettre d'afficher un layout parfait. En fait, le fichier ne devrait contenir que trois paramètres qui devraient s'adapter au layout choisi et seraient exprimés soit en pourcentage soit en pixels. Nous reviendrons sur ce sujet dans une étape ultérieure.

Enfin, je profite de cette étape pour renommer les layouts en `gala_nn.css` et reporter cette modification dans le fichier `inclure/head.html`. En outre, j'ai rajouté le numéro du layout gala dans le titre du « header ».

6.5 Fil d'Ariane et habillage en LESS

6.5.1 Fil d'Ariane

Le fil d'Ariane ne fait pas partie des blocs Gala. Ce bloc a été rajouté dans le « body » du squelette afin d'être inclus dans l'échafaudage Z. Nous avons défini des styles pour ce bloc dans le fichier `css/breadcrumb.css`.

Nous allons donc utiliser LESS pour générer les styles du fil d'Ariane en créant le fichier de styles `less/breadcrumb.less` et les variables associées dans `less/var_breadcrumb.less`. On en profite aussi pour extraire la couleur de fond du fil d'Ariane de ces styles pour les transférer dans les styles du thème. En conséquence, le fil d'Ariane ne possède que 4 variables, les marges et les paddings.

6.5.2 Habillage

Le fichier d'habillage actuel du squelette, `css/theme.css`, est composé d'une série de styles définissant la couleur des blocs Gala uniquement. On y rajoute donc la couleur du fil d'Ariane et on crée ainsi les deux fichiers LESS qui généreront ces styles, à savoir, `less/theme.less` et les variables associées dans `less/var_theme.less`.

On garde toujours pour l'instant les quelques styles typographiques pour le « header » et le « footer » afin de conserver le *look & feel* Gala. Ces styles disparaîtront à l'étape suivante quand on positionnera les styles SPIP et les bases éditoriales du squelette.

6.6 Conclusion

Suite à cette étape, nous avons extrait un nombre conséquent de paramètres sur lesquels il sera possible de jouer pour agrémenter le squelette assez simplement, en particulier pour la structure des layouts. Pour les layouts Gala, l'optimisation du code sera discutée dans une étape ultérieure. Le squelette Integraal propose une implémentation en SASS très astucieuse dont il faudrait s'inspirer mais cela pourrait nous pousser à passer du LESS au SASS pour bénéficier des fonctions avancées de SASS.

Comme LESS est un « pré-processeur », le squelette utilise toujours des fichiers CSS en inclusion. Il sera donc possible d'inclure d'autres fichiers CSS sans pour autant les passer en LESS. En l'occurrence, le fichier `css/form.css` n'est pas migré en LESS car d'une part, aucun paramètre n'est identifiable et d'autre part, ces styles seront pour la plupart surchargés par les formulaires SPIP que nous intégreront dans l'étape suivante.

7. ETAPE 5 : INTEGRER LES PREMIERS ELEMENTS SPIP

Lors de cette étape, les premiers éléments SPIP vont être intégrés au squelette : des pages de base (sommaire, article, rubrique) et leurs CSS, un menu et un fil d'Ariane finalisé. L'utilisation des balises HTML5 sera privilégié.

7.1 Ajout des styles de SPIP

Maintenant que la structure du squelette est définie (HTML et CSS) on peut s'atteler à construire le HTML des premières pages du squelette ainsi que leurs styles. Comme il est inutile de réinventer la roue, on va s'inspirer des styles déjà définis par le squelette Zpip-dist v2 qui n'est malheureusement plus supporté au profit de Spipr-dist.

On insère donc dans le fichier `inclure/head.html` les lignes suivantes :

```
[ (#REM) 4. CSS pour SPIP ]
[<link rel="stylesheet" href="#CSS{css/spip.css}" type="text/css" />]
[<link rel="stylesheet" href="#CSS{css/spip.form.css}" type="text/css" />]
[<link rel="stylesheet" href="#CSS{css/spip.comment.css}" type="text/css" />]
[<link rel="stylesheet" href="#CSS{css/spip.list.css}" type="text/css" />]
[<link rel="stylesheet" href="#CSS{css/spip.petition.css}" type="text/css" />]
[<link rel="stylesheet" href="#CSS{css/spip.pagination.css}" type="text/css" />]
[<link rel="stylesheet" href="#CSS{css/spip.portfolio.css}" type="text/css" />]
```

On place les fichiers correspondants dans le dossier `css/` du squelette Z-Gala. En fait, si `spip.comment.css` et `spip.petition.css` proviennent bien tous deux de Zpip-dist v2, les autres ont subis quelques modifications mineures et proviennent d'un package dérivé concocté par b_b. `spip.css` contient les styles associés au code généré par SPIP. Pour les autres styles, le nom des fichiers est suffisamment explicite pour comprendre de quoi il s'agit.

7.2 Choisir une police de base

Pour un site web le choix de la police est essentielle comme l'explique Tetue dans son article [Stop Arial 11px !](#). En suivant les conseils de l'article j'ai choisi la police **Lucida Grande** en ajoutant la ligne suivante dans le fichier `css/fonts.less` qui était resté vide jusqu'alors et qui produira le fichier CSS de même nom.

```
body {
    font-family: 'Lucida Grande', 'Lucida Sans Unicode', Helvetica, sans-serif;
}
```

En rafraichissant la page on peut voir la nouvelle police à l'œuvre. Pour l'impression, j'ai choisi la police **Helvetica** de base.

7.3 Ajouter une barre principale de navigation horizontale

Structure classique des sites web, la barre principale de navigation horizontale du squelette Z-Gala est incluse sous le « header ». Comme d'autres menus que nous définirons par la suite, j'ai décidé d'utiliser le plugin **Menus** pour ma barre de navigation principale afin de pouvoir accueillir un contenu simplement configurable.

Pour ce faire, étant donné que cette barre est affichée sur l'ensemble des pages et n'a pas vocation à s'adapter à la page affichée, on ajoute l'inclusion d'un bloc « non Z » dans le « body » : `inclure/nav.html`. L'inclusion est englobée dans une balise `<nav>` de classes `.navbar` et `.hornav` et d'id `navbar` comme suit :

```
<nav class="navbar hornav clearfix" id="navbar">
    <INCLUDE{fond=inclure/navbar, env} />
</nav> <!-- .NAVBAR -->
```

On crée l'inclusion `inclure/navbar.html` qui appelle une autre inclusion, celle du plugin Menus sachant que nous avons choisi l'identifiant `navbar` pour la barre principale de navigation. On obtient donc le code suivant :

```
<div class="menu">
    #INCLUDE{fond=inclure/menu, env, identifiant=navbar}
</div>
```

Pour obtenir la liste des secteurs du site et le lien d'accueil comme je le souhaite pour l'instant, il faut créer le menu d'identifiant `navbar` avec deux entrées, l'une pour le lien accueil et l'autre pour la liste des rubriques limitée aux secteurs.

Les styles utilisés (`.hornav`, `.menu`, `.menu-items`, `.item`) sont définis dans le fichier `css/spip.list.css` car la barre est fondée sur une structure de liste non ordonnée. Quelques styles supplémentaires dans `css/theme.css` sont néanmoins nécessaires afin de décorer la barre plus précisément. On peut alors rafraichir la page pour voir le résultat.

7.4 Utiliser les balises HTML5

Comme le HTML5 commence à être correctement supporté par les différents navigateurs, nous allons utiliser les nouvelles balises pour créer nos pages. Quoi de mieux pour commencer que le fichier `body.html` et sa structure de blocs.

HTML5 propose des nouvelles balises de type *sectioning content* et *grouping content* qui sont à même de préciser la sémantique des blocs du « body ».

Ces balises sont :

- `<header>` et `<footer>` que l'on va utiliser pour les blocs de même nom de notre « body » ;
- `<nav>` pour encapsuler la barre principale de navigation du site ;
- `<main>` pour définir le bloc de contenu principal de la page, soit tous les blocs hors « nav », « header » et « footer » (cette balise est toujours unique dans la page) ;
- `<aside>` pour encapsuler nos blocs « aside » et « extra ».

J'ai donc choisi de structurer le nouveau `body.html` comme suit, ce qui me semble assez logique mais reste uniquement une interprétation personnelle :

```
<body>
  <div class="page" id="page">
    <header class="header" id="header">
      <INCLUDE{fond=header/#ENV{type-page}, env} />
    </header> <!-- .HEADER -->
    <nav class="navbar hornav clearfix" id="navbar">
      <INCLUDE{fond=inclure/navbar, env} />
    </nav> <!-- .NAVBAR -->

    <main>
      <div class="wrapper" id="wrapper">
        <div class="content" id="content">
          <INCLUDE{fond=breadcrumb/#ENV{type-page}, env} />
          <INCLUDE{fond=content/#ENV{type-page}, env} />
        </div> <!-- .CONTENT -->
      </div> <!-- .WRAPPER -->

      <aside class="aside" id="aside">
        <INCLUDE{fond=aside/#ENV{type-page}, env} />
      </aside> <!-- .ASIDE -->
      <aside class="extra" id="extra">
        <INCLUDE{fond=extra/#ENV{type-page}, env} />
      </aside> <!-- .EXTRA -->
    </main>

    <footer class="footer" id="footer">
      <INCLUDE{fond=footer/#ENV{type-page}, env} />
    </footer> <!-- .FOOTER -->
  </div> <!-- .PAGE -->
</body>
```

A l'occasion, j'ai remarqué aussi que le fichier `reset.css` de TinyTypo ne contient pas la balise `<main>`. Il faut donc la rajouter, ce qui fait un fork de plus pour TinyTypo.

L'utilisation des balises `<nav>`, `<aside>`, `<footer>` et `<header>` ne se limite pas au seul « body » et peut être étendue au sein des contenus des blocs eux-mêmes, ce que nous utiliserons par la suite. En outre, nous serons aussi amenés à utiliser d'autres balises HTML5 structurantes comme `<article>`, `<section>`, `<figure>` et `<figcaption>`.

Plusieurs articles permettent de se faire une idée de l'utilisation de ces nouvelles balises. Néanmoins, chacun devra se faire sa propre opinion car les articles ne se rejoignent pas tous sur l'utilisation précise de ces balises : [HTML5 se dévoile](#), [Éléments HTML5 de structure](#), [W3C HTML 5.2](#).

7.5 Améliorer le Fil d'Ariane

A ce stade, le fil d'Ariane du squelette Z-Gala est matérialisé uniquement par le fichier `breadcrumb/dist.html` au contenu statique qui s'applique à l'ensemble des pages qui ne possèdent pas un fil d'Ariane spécifique. Nous allons pour l'instant se focaliser sur ce fichier uniquement en lui donnant un contenu générique et dynamique.

Le squelette Spipr-dist possède un tel fichier dont le contenu peut être réutilisé tel quel. Ce fichier fait appel à une inclusion `breadcrumb/inc-objet.html` qui calcule la hiérarchie à afficher quand la page en cours d'affichage est celle d'un objet au sens de SPIP. Il est donc nécessaire de copier aussi ce fichier dans le squelette Z-Gala.

On notera que contrairement à Bootstrap sur lequel s'appuie le squelette Spipr-dist, le séparateur des éléments du fil d'Ariane est inscrit dans le HTML et non dans le CSS en utilisant le sélecteur `:before`. Néanmoins, comme le caractère séparateur est inclus dans une balise `<span>` de classe `.divider` il sera possible de le styler mais pas de le changer via une variable LESS.

Il ne reste plus qu'à rajouter des styles dans le fichier LESS `less/breadcrumb.less` ainsi que les variables associées dans `less/var_breadcrumb.less` et on peut afficher le nouveau Fil d'Ariane – sans intérêt – pour la page d'accueil. Il faudra se poser la question plus tard si on conserve un fichier LESS séparé pour la structure du fil d'Ariane ou si l'on regroupe tout dans `less/theme.less` comme pour les autres blocs.

7.6 Structurer la page sommaire

Pour l'instant, le but est de créer un sommaire simple composé entre autres choses de la liste des articles récents. La question est quelle structure HTML5 sied le plus à ce sommaire ? J'ai beau naviguer sur le web fébrilement pendant des heures, il est difficile de se faire une vraie conviction comme pour le « body ». J'ai donc opté pour un sommaire dont la structure s'inspire de celle du site de romy.tetue.net, à savoir, un bloc de présentation du site, une sélection d'articles du site et un bloc avec les derniers éléments récents du site comme les articles et les commentaires. La structure HTML5 est faite d'une section par bloc.

Le fichier `content/sommaire.html` contient l'appel à 4 inclusions, une pour chaque élément précédemment cité, le tout inclus dans une balise `<section>` :

- `inclure/inc-bienvenue.html`
- `inclure/inc-vedette.html`
- `liste/articles_resume.html`
- `liste/forums_resume.html`

Pour les inclusions « bienvenue » et « vedette » nous allons utiliser l'objet sélection du plugin **Sélection Editoriale** afin d'éviter l'utilisation des mots-clés. Il faudra donc ajouter le plugin dans les `<necessite>` du `paquet.xml`. Pour les résumés d'articles on utilise l'inclusion existante dans Z-Core. Seule la liste des commentaires restera vide ou du moins juste remplie avec un titre.

Enfin, comme le fil d'Ariane pour le sommaire n'apporte rien, je le supprime en créant un fichier `breadcrumb/sommaire.html` vide. On voit ainsi l'intérêt d'avoir fait du fil d'Ariane un bloc Z.

7.7 Ajouter la page article

Comme pour la page sommaire, je vais créer une page article rapidement en recopiant celle de Spip-dist, sans modifier les ni les styles ni le contenu sauf ce qui concerne les pétitions. Le but est juste de créer cette page afin d'avoir le loisir de naviguer dans le site.

Nous reviendrons dans une étape ultérieure sur cette page afin d'affiner et de simplifier le HTML et les classes CSS.

Le plus important pour cette page c'est qu'il est nécessaire de créer un fichier `article.html` à la racine car on veut passer des paramètres spécifiques qui ne sont pas fournis par Z-Core (voir le fichier `page.html`). Comme on peut le voir ci-après, la page article voit son environnement enrichi avec l'identifiant de la rubrique et celui du secteur.

```
<BOUCLE_principale_article(ARTICLES){id_article}>
<INCLURE{fond=structure,env,id_rubrique=#ENV{id_rubrique,#ID_RUBRIQUE},id_secteur=#
ID_SECTEUR,type-page=article,composition=#COMPOSITION} />
</BOUCLE_principale_article>
```

Il suffit dès lors de recopier et modifier légèrement comme précisé plus haut le fichier `content/article.html` de Spip-dist dans le dossier homonyme de Z-Gala.

7.8 Ajouter la page rubrique

Comme pour la page article, je vais créer une page rubrique rapidement en recopiant celle de Spip-dist, sans modifier les ni les styles ni le contenu sauf ce qui concerne les brèves et les sites. Le but est juste de créer cette page afin d'avoir le loisir de naviguer dans le site.

Nous reviendrons dans une étape ultérieure sur cette page afin d'affiner et de simplifier le HTML et les classes CSS.

Le plus important pour cette page c'est qu'il est nécessaire de créer un fichier `rubrique.html` à la racine car on veut passer des paramètres spécifiques qui ne sont pas fournis par Z-Core (voir le fichier `page.html`). Comme on peut le voir ci-après, la page rubrique voit son environnement enrichi avec l'identifiant du secteur.

```
<BOUCLE_principale_article(ARTICLES){id_article}>
<INCLURE{fond=structure,env,id_rubrique=#ENV{id_rubrique,#ID_RUBRIQUE},id_secteur=#
ID_SECTEUR,type-page=article,composition=#COMPOSITION} />
</BOUCLE_principale_article>
```

Il suffit dès lors de recopier et modifier légèrement comme précisé plus haut le fichier `content/rubrique.html` de Spip-dist dans le dossier homonyme de Z-Gala.

7.9 Conclusion

A l'issue de cette étape, nous avons un squelette qui permet d'afficher des objets standard de SPIP avec une typographie adéquate. Les styles ne sont pas finalisés pour toutes les inclusions des pages sommaire, article et rubrique et l'on s'appuie encore sur des inclusions de Z-Core ou de la Dist mais nous y reviendrons dans une étape ultérieure de finition.

8. ETAPE 6 : INTEGRER LE NOIZETIER

Lors de cette étape, nous allons étudier comment intégrer le plugin **noiZetier** dans le squelette Z-Gala afin d'être en capacité de concevoir le contenu de chaque bloc Z de façon modulaire et configurable. Les modules ainsi utilisés sont appelées **noisettes** en langage SPIP ce qui explique le nom du plugin.

8.1 Pourquoi le noiZetier ?

A l'étape précédente nous avons construit une page sommaire composée de 4 inclusions ou noises, à savoir, un texte de bienvenue, une liste d'articles mis en exergue, la liste des articles récents et celle des derniers commentaires postés. Cette composition est immuable tout comme l'ordre dans lequel les noises sont agencées dans la page. Si l'on veut changer l'ordre ou éviter d'afficher l'une ou l'autre de ces noises il est nécessaire de réécrire ou de surcharger le fichier `content/sommaire.html`.

Le squelette Sarka-SPIP propose deux méthodes pour éviter partiellement la réécriture, à savoir :

- la configuration par formulaire de l'affichage d'une noise permettant ainsi au webmestre de choisir son agencement ;
- des pipelines au début ou à la fin de certains blocs afin d'y insérer des noises.

C'est déjà une avancée mais cela ne permet pas de choisir la localisation exacte des noises et la mise en œuvre reste un peu artisanale.

Le noiZetier, lui, industrialise ces méthodes et permet de choisir et d'agencer les noises qui composent certains blocs Z des pages du site mais aussi de les décrire d'une manière plus explicite voire de gérer une configuration, tout cela au sein d'une même interface très pratique. Bien entendu, cela demande de construire son squelette d'une façon précise pour que les noises soient reconnues et gérées par le noiZetier, ce que nous discuterons plus avant.

Le noiZetier est également compatible avec les compositions de squelette (voir le plugin [Compositions](#) sur SPIP-Contrib) et permet de créer à la volée des pages ou des compositions « virtuelles » qui ne diffèrent que par les noises ajoutées aux différents blocs du squelette de base.

C'est donc pour toutes ces raisons que j'ai choisi d'intégrer ce plugin au squelette Z-Gala.

8.2 Fonctionnement du noiZetier

Le noiZetier s'adapte à tout type de squelette Z ou non. Néanmoins, depuis la branche 3, le plugin est dédié à Z. Son fonctionnement repose sur 3 principes :

- déterminer les pages et les blocs pouvant recevoir des noises ;
- lister les types de noise disponibles, leurs caractéristiques et leur capacité à s'insérer dans une page précise, plusieurs pages ou toutes les pages ;
- compiler et insérer les noises à la fin de chaque bloc ou à un endroit précisé par un marqueur.

Les paragraphes qui suivent abordent chaque principe en se limitant au fonctionnement avec un squelette Z comme Z-Gala et à l'utilisation de la version 3 ou ultérieure du noiZetier.

8.2.1 Identification des pages

En premier lieu, le noiZetier doit déterminer la liste des pages Z disponibles dans son environnement, c'est-à-dire accessibles via le « Path SPIP » (au sens de la fonction `find_in_path()`). Par exemple, si l'on considère le squelette Z-Gala issu de l'étape 5, les pages disponibles sont : sommaire, article, rubrique, tinytypo et boite et si le plugin Rainette est aussi activé, sa page meteo est aussi détectée.

Pour identifier ces pages le noiZetier considère le répertoire qui accueille le contenu principal de chaque page Z, à savoir, celui issu du bloc identifié comme le premier élément de la variable globale des blocs Z (`$GLOBALS['z_blocs'][0]`).

Les compositions classiques situées dans ce répertoire – par exemple, `article-evenement.html` - sont aussi automatiquement détectées par le noiZetier.

Sans rien spécifier d'autre, le noiZetier détecte les pages et ne les reconnaît, donc ne les affiche dans l'interface de configuration, que par leur identifiant (article, sommaire...). Néanmoins, il est possible de préciser un titre, une description, un icône ainsi que de d'autres caractéristiques pour chaque page par l'intermédiaire d'un fichier YAML (`article.yaml`) ou XML de même nom (`sommaire.xml`) dont le format est identique à celui des compositions. Le format YAML est à privilégier car le XML sera déprécié dans les prochaines versions.

```
nom: '<zgala:page_article_titre:>'
description: '<zgala:page_article_description:>'
icon: 'article-xx.svg'
```

Il est possible de forcer l'utilisation d'un fichier descriptif XML ou YAML en positionnant la constante `_NOIZETIER_LISTER_PAGES_SANS_XML` à `true`., ce qui est aussi conseillé pour éviter de charger des pages non clairement identifiées.

8.2.2 Identification des blocs

Une fois les pages identifiées, le noiZetier doit déterminer quels blocs sont susceptibles de recevoir des noisettes et ce, pour chaque page.

Par défaut pour un squelette Z, sans rien préciser, le noiZetier utilise la variable globale des blocs Z, à savoir, `$GLOBALS['z_blocs']` et considère que tous les blocs Z de cette liste sont capables d'accueillir des noisettes. Si l'on souhaite modifier cette liste pour l'ensemble des pages il est possible d'utiliser le pipeline `noizetier_blocs_defaut` pour principalement supprimer des blocs comme `head` ou `head_js`, par exemple.

Par contre, si l'on souhaite préciser pour chaque page une liste de blocs Z différente, on peut utiliser le fichier XML de description de la page et lui adjoindre des balises `<bloc>` de la façon suivante :

```
<bloc id="content" nom="plugin:chaîne" description="plugin:chaîne"
icon="img/fichier.png" />
<bloc id="aside" nom="plugin:chaîne" description="plugin:chaîne"
icon="img/fichier.png" />
```

Cette option n'est valable que dans le fichier XML ; elle est dépréciée dans le noiZetier v3 qui propose à la place la configuration des blocs exclus page par page. En outre, dans cette version il est possible de décrire chaque bloc avec un fichier YAML `bloc.yaml` positionné dans le répertoire du bloc.

8.2.3 Description des types de noisette

Un type de noisette est un fichier HTML placée dans un sous-répertoire défini par configuration situé dans le répertoire d'un plugin ou dans le répertoire `squelettes/`. Par défaut, le noiZetier utilise le répertoire relatif `noisettes/`.

Pour préciser la ou les pages dans lesquelles un type de noisette peut être inséré il faut formater son nom :

- un type de noisette pouvant s'insérer sur n'importe quelle page sera nommé tout simplement `nom_type_noisette.html`. Par convention, seuls les lettres minuscules et l'underscore (`_`) sont autorisés.
- un type de noisette ne pouvant être inclus que sur une page spécifique sera nommée `type_page-nom_type_noisette.html`⁴. Ainsi, un type de noisette spécifique à la page article sera nommé `article-nom_type_noisette.html` et le noiZetier ne la proposera que sur la page article et les compositions de la page article.
- Un type de noisette ne pouvant être proposée que pour une composition spécifique sera nommé `type_page-composition-nom_type_noisette.html`.

Pour être utilisée par le noiZetier, un type de noisette doit impérativement être accompagnée d'un fichier YAML de configuration de même nom. Ce fichier permet :

- de préciser le titre (seule information obligatoire), la description , l'icône et la nature conteneur ou pas du type de noisette ;
- de limiter les paramètres de contexte de la noisette ;
- de préciser si le type de noisette nécessite l'activation d'un plugin particulier pour être affichée ;
- de préciser les modes d'inclusion Ajax, dynamique, statique ;
- et de définir les paramètres de configuration du type de noisette lui-même qui seront présentés dans un formulaire généré automatiquement par le noiZetier (saisie YAML).

Pour plus d'information sur la déclaration des noisettes il est conseillé de lire l'article [Déclarer les noisettes au NoiZetier](#) de SPIP-Contrib.

8.2.4 Compilation des noisettes

Le noiZetier utilise le pipeline `recuperer_fond` pour intercepter les appels à un squelette `bloc/page.html`. Le noiZetier ajoute à la fin du squelette ou à un endroit précis identifié par un marqueur (`<!--noisettes-->`) une inclusion du squelette `conteneur_compiler.html` qui se charge d'inclure les noisettes proprement dites.

Sauf si la configuration de la noisette précise le contraire, le noiZetier insère les noisettes avec le mode Ajax activé et en inclusion statique.

⁴ On notera que l'usage du tiret (-) est réservé pour séparer les composants du nom de la noisette comme la page, la composition et la noisette elle-même. Il faut donc utiliser uniquement l'underscore pour le nom de chaque composant.

8.3 Mise en place dans Z-Gala

Il faut tout d'abord ajouter le noiZetier comme un plugin nécessité dans le `paquet.xml` de Z-Gala. On rajoute donc la ligne :

```
<necessite nom="noizetier" compatibilite="[3.0.49;]" />
```

8.3.1 Identification des blocs

Pour la détermination des pages de Z-Gala susceptibles d'accueillir des noisettes, la variable globale `$GLOBALS['z_blocs']` définie dans le squelette est suffisante.

Pour sélectionner les blocs Z par défaut nous allons utiliser le pipeline `noizetier_blocs_defaut` afin d'éliminer les blocs `head` et `head_js` de la liste. Pour ce faire il faut d'une part ajouter la déclaration du pipeline dans le `paquet.xml` de Z-Gala et d'autre part, créer le fichier `zgala_pipelines.php` à la racine du squelette afin d'y insérer le code adéquat dans le pipeline.

Si, en outre, pour certaines pages il apparaissait nécessaire de réduire encore la liste des blocs par défaut nous utiliserons la configuration de la page qui depuis la version 3 du noiZetier permet de configurer les blocs exclus page par page.

On ajoute aussi pour chaque bloc son fichier YAML descriptif.

8.3.2 Description des pages

Etant donné que le squelette Z-Gala est développé pour être générique et utilisable le plus facilement possible il me paraît nécessaire d'expliciter au mieux chaque élément qui le compose. C'est pourquoi chaque page et chaque bloc du squelette seront décrits avec les fichiers YAML précités.

Il faut donc créer pour chaque page du squelette Z-Gala un fichier YAML de même nom pour y insérer les caractéristiques et paramètres idoines. Pour les titres et les descriptions je vais utiliser les chaînes de langue afin d'offrir un squelette entièrement traduit y compris dans l'espace privé.

On crée donc les fichiers YAML des pages sommaire, article et rubrique dans le dossier `content/` du squelette. On ajoute aussi la description des pages 401 et 404 fournies par Z-Core mais non décrites pour le noiZetier pour l'instant. Et comme je souhaite aussi utiliser le plugin Rainette par la suite, je reproduis ce schéma avec une page météo qui accueillera les noisettes fournies par Rainette. Les pages boîte et tinytypo seront elles transférées en démo (voir paragraphe 8.3.9).

On vérifie ainsi que la détection des pages se fait bien via le path SPIP.

On notera également les icônes des pages qui sont adressés via le path image de SPIP : répertoires `prive/themes/spip/images/`. Il est donc inutile de rajouter des icônes pour les pages article et rubrique qui possèdent déjà un icône dans SPIP.

En se rendant sur la page `noizetier_pages` dans l'interface privée – accessible par le menu Squelettes / Gérer les noisettes - la liste des pages présente l'apparence ci-après. En cliquant sur un lien de page, par exemple, la page article, on vérifie bien que la liste des blocs proposés se limite aux blocs Z à l'exception de `head` et `head_js`.

Toutes les pages

Pages liées à un type de contenu

Pages non liées à un type de contenu

Contenus possédant une configuration de noisettes

Configuration du noiZetier

Recharger les pages

Recharger les types de noisette

Exporter la configuration

Importer une config.

noiZetier

Sélectionnez la page dont vous souhaitez configurer les noisettes.

	Page 401 (401) Page d'erreur 401 – L'accès nécessite un login	 
	Page 404 (404) Page d'erreur 404 – Page non trouvée	 
	Page Article (article) Page par défaut des objets article	 
	Page Météo (meteo) Page des conditions et des prévisions météorologiques fournies par Rainette	 
 La page est inactive car le ou les plugins suivants sont désactivés : rainette.		
	Page Rubrique (rubrique) Page par défaut des objets rubrique	 
	Page Sommaire (sommaire) Page d'accueil du site	 

On remarque que la page météo est listée avec un message d'avertissement car le plugin Rainette n'est pas encore activée. Il est cependant possible dans la version 3 du noiZetier de configurer cette page mais tant que le plugin Rainette ne sera pas activé aucun affichage ne sera disponible.

8.3.3 Réorganisation des inclusions en noisettes

Il est temps maintenant de s'occuper des noisettes. Si on clique sur une page dans l'affichage de la liste des pages on distingue sur la gauche une liste de types de noisette applicables à toutes les pages. En effet, le noiZetier propose par défaut deux noisettes basiques, à savoir, un bloc de texte libre et un bloc de code libre. Et comme nous avons aussi activé le plugin Menus la noisette menu est proposée ainsi qu'une noisette sélection éditoriale fournie par le plugin de même nom.

On voit donc que certains plugins proposent déjà des noisettes. Il est possible de s'exercer à utiliser l'interface en ajoutant certaines de ces noisettes dans un bloc Z, par exemple content de la page sommaire, et de vérifier que ces noisettes s'affichent bien après le contenu existant.

Pour faire de notre Z-Gala un squelette générique, il faut que nous puissions composer à loisir chaque bloc de chaque page. Il est donc nécessaire que le contenu de chaque bloc soit vide par défaut (ou du moins minimal).

A ce stade du développement de Z-Gala, le bloc content de chaque page est rempli avec du contenu non débrayable, le fil d'Ariane avec un code générique qui peut s'appliquer à presque toutes les pages

utilisées. La barre principale de menu est, elle, incluse dans un bloc non-Z donc ne pourra pas être concernée par des noisettes. En outre, certains blocs possèdent un contenu bidon pour repérer le bloc concerné.

Pour obtenir notre contenu entièrement débrayable et configurable, il faut migrer notre contenu en autant de noisettes réutilisables.

8.3.4 Fil d'Ariane

Pour le fil d'Ariane, il suffit de convertir l'inclusion en noisette en réalisant les opérations suivantes :

- copier le fichier `breadcrumb/dist.html` en `noisettes/breadcrumb.html` ;
- supprimer le code HTML inséré dans le fichier `breadcrumb/dist.html` ;
- déplacer l'inclusion `breadcrumb/inc-objet.html` dans `inclure/` et le renommer en `inc-breadcrumb_objet.html` ;
- supprimer le fichier `breadcrumb/sommaire.html` qui ne sert plus à rien car nous pouvons configurer chaque page différemment ;
- créer le fichier YAML de description du type de noisette dans `noisettes/`.

Ensuite, il faut ajouter la noisette fil d'Ariane ainsi créée dans le bloc `breadcrumb` des pages concernées. L'interface pour réaliser cette opération est très facile d'utilisation et permet en un drag'n drop de rajouter la noisette dans les pages article et rubrique.

On notera donc qu'il est nécessaire de répéter l'ajout du fil d'Ariane dans chacune des pages où il doit être affiché, soit presque toute. Cette opération a été facilitée dans la version 3 du noizetier qui propose une option pour propager une même noisette dans une liste de pages choisies.

8.3.5 Contenu de la page sommaire

Le contenu de la page sommaire est constitué de 4 inclusions bien définies. Pour migrer ce bloc en noisettes il suffit donc de transformer les inclusions actuelles en noisettes.

En premier lieu on déplace les inclusions `inclure/inc-bienvenue.html`, `inclure/inc-vedette.html` dans le dossier `noisettes/` en les renommant en `sommaire-bienvenue.html` – pour la limiter à la page sommaire – et `vedette.html`. Le code de chacune de ces types de noisette peut rester identique. On adjoint les fichiers descriptifs en YAML en précisant bien que le contexte passé peut être réduit au minimum.

Pour les deux autres inclusions du sommaire, on crée un fichier type de noisette qui encapsule l'inclusion générique `liste/articles-resume.html` ou `liste/forums-resume.html` dans une balise `<div>` comme cela est fait dans le contenu actuel du sommaire. On y adjoint aussi les fichiers YAML. Les types de noisettes se nomment `articles_recents` et `forums_recents`.

Enfin, on vide le fichier `content/sommaire.html` de son contenu. Tout est prêt pour configurer le sommaire avec les noisettes ainsi définies.

8.3.6 La barre de navigation principale

La barre de navigation principale, comme on a pu le voir dans un chapitre précédent n'est pas incluse dans un bloc Z car elle est guère modifiable en fonction de la page. Cet a priori se tient mais il nous

conduit à ne pas pouvoir utiliser le noiZetier pour configurer ce bloc. Par exemple, on pourrait imaginer un menu déroulant plutôt que le menu actuel voire d'intégrer un formulaire de recherche à droite du bloc.

Il est donc nécessaire de muer le bloc de navigation en bloc Z. La capacité du noiZetier v3 à copier en une fois une noisette dans une liste de pages choisies permet en outre de limiter le désagrément de devoir répliquer une noisette de navigation sur tout le site.

Pour transformer le bloc en bloc Z de nom `nav` il faut donc :

- ajouter la chaîne 'nav' dans la liste stockée dans la variable globale `$GLOBALS['z_blocs']` ;
- créer un répertoire `nav/` à la racine du plugin et y stocker un fichier `dist.html` vide par défaut. Etant donné que cette barre n'est pas destinée à s'adapter à la page, il est inutile de créer un fichier propre à chaque page, la configuration du bloc s'intégrera toujours dans le fichier `dist.html` ;
- modifier le code du fichier `body.html` afin que le bloc `nav` soit inclus comme un bloc Z ;
- créer une noisette reflétant le menu horizontal du bloc actuel nommée `navbar.html`.

Afin de pouvoir configurer un menu quelconque j'ai supprimé la classe `hornav` du bloc et j'ai adapté les styles en conséquence. Finalement l'id `navbar` a été déplacé du bloc `nav` à la noisette `navbar.html`. Le bloc de navigation principale du body devient :

```
<nav class="nav clearfix" id="nav">
  <INCLUDE{fond=nav/#ENV{type-page},env} />
</nav> <!-- .NAVBAR -->
```

et la noisette `navbar.html` :

```
[<div class="menu" id="navbar">
  (#INCLUDE{fond=inclure/menu, env, identifiant=navbar})
</div>]
```

Il suffit maintenant de configurer le nouveau bloc `nav` de chacune des pages du squelette pour obtenir une barre de navigation identique mais cette fois entièrement configurable.

8.3.7 La page article

Le cas des pages article et rubrique est classique. Le squelette Aveline qui est basé sur le framework Z v1 et le noiZetier propose un découpage du bloc content en n types de noisette, à savoir :

- le contenu principal (y compris le cartouche) ;
- le lien hypertexte ;
- la liste des documents joints (hors images) ;
- le portfolio des images ;
- les mots-clés liées à l'article ;
- les commentaires postés et le formulaire ;
- les signatures de pétition et le formulaire ;

ainsi que d'autres types de noisette liées à des plugins, comme le formulaire de notation qui nécessite le plugin « Notation » et les boutons précédent/suivant qui nécessite le plugin « Précédent Suivant ».

Le découpage semble pertinent à ceci près que je préférerais découper le contenu principal en cartouche et contenu textuel afin de pouvoir plus facilement jouer sur le contenu et la position du cartouche. De même, la noisette lien hypertexte me paraît bien luxueuse eu égard à son intérêt et je ne souhaite pas utiliser les pétitions.

Contrairement à Aveline, Z-Gala utilise la structure HTML5 et ses nouvelles balises. Mais combiner sémantique des balises HTML5 et construction par noisette s'avère ni simple ni totalement satisfaisant comme on va pouvoir s'en apercevoir ci-après.

En comparant plusieurs sources sur le net, le site [HTML5 Doctor](#), celui de [Tetue](#), [Alsacréations](#) et enfin le fichier `content/article.html` de Z-Gala copié de Spipr-dist, les recommandations ne sont pas toutes cohérentes. Parfois la balise `<article>` englobe tout le contenu quel qu'il soit, parfois non, parfois des balises `<section>` sont incluses parfois elles sont remplacées par des balise `<aside>`...

En mélangeant tout ça et en secouant bien le tout je me suis décidé pour une structure où la balise `<article>` englobe l'ensemble des blocs liés à l'article comme le montre le code ci-après. Cette structure me permet de définir chaque balise incluse comme une noisette, les éléments liés à l'article étant chacun dans une balise `<aside>`.

Il sera aussi facile au vu de la découpe d'insérer une table des matières où l'on souhaite, par exemple, après le cartouche comme cela se fait souvent.

```
<article>
  <header class="cartouche">titre, date, commentaires...</header>

  <section class="main">chapo, texte, hyperlien</section>

  <footer>post-scriptum, notes de fin de page</footer>

  <aside>[(&#x21;INCLUDE{fond=inclure/documents, id_article})]</aside>
  <aside>[(&#x21;INCLUDE{fond=inclure/portfolio, id_article})]</aside>
  <aside>
    <INCLUDE{fond=inclure/forum, id_article,
repondre_url=(#PARAMETRES_FORUM|url_reponse_forum),
repondre_titre=<:repondre_article:>} />
  </aside>
</article>
```

Mais pour obtenir ce résultat, il faut que les noisettes soient insérées à l'intérieur de la balise `<article>` : le fichier `content/article.html` ne peut donc plus être vide. Pour ce faire, il faut utiliser une nouvelle fonctionnalité du noizetier qui sait détecter le pattern `<!--noisettes-->` pour déterminer où insérer les noisettes configurées et non plus en fin de fichier. Le fichier `content/article.html` devient donc :

```
<article>
  <!--noisettes-->
</article>
```

Il suffit alors de construire les types de noisettes qui composent la structure de l'article pour obtenir la nouvelle page article tout en noisettes et leur donner un nom préfixé ou pas par **article-** pour celles uniquement dédiées à la page article. En configurant la page et en l'affichant après rafraichissement on retrouve la page article initiale.

Le problème de cette approche est qu'il a été nécessaire de multiplier les boucles ARTICLES dans chaque noisette ou presque afin d'afficher les champs liés. Cela doit forcément jouer sur les performances en comparaison de la boucle ARTICLES unique de la page article initiale. Il a aussi fallu désactiver l'ajax par défaut de certaines noisettes car il ne sert à rien. L'intérêt d'activer cette fonctionnalité par défaut apparaît superflue mais le noisetier v3 permet de gérer cette option finement aujourd'hui.

Pour parfaire aussi la configuration YAML des types de noisette il faut préciser si chaque noisette nécessite ou pas l'activation d'un plugin pour fonctionner. Si cela est assez naturel pour les plugins hors distribution SPIP ça l'est moins pour les plugins-dist. J'ai donc rajouté le nécessite Medias pour les noisettes documents et portfolio ainsi que le nécessite Forum pour la noisette des commentaires.

8.3.8 La page rubrique

La problématique de la page rubrique est en tout point identique à celle de la page article. Pour obtenir une page toute en noisettes identique à la page actuelle il convient de créer les types de noisette listés ci-après et de vider le contenu du bloc `content/rubrique.html` :

- le cartouche de la rubrique ;
- le contenu principal de la rubrique composé de son chapo et de son texte descriptif ;
- le pied composé des notes ;
- les documents joints ;
- le portfolio des images ;
- les sous-rubriques ;
- les articles publiés dans la rubrique.

Les sites et les brèves de la rubrique ne seront pas inclus dans cette version du squelette Z-Gala mais pourront être développées plus tard comme des types de noisette additionnels. En procédant comme pour la page article on retrouve rapidement une page rubrique idoine entièrement composée de noisettes.

En faisant une légère modification dans les inclusions `inclure/inc-documents.html` et `inclure/inc-portfolio.html` il a été possible de rendre le type de noisette portfolio et documents joints génériques et donc applicables aussi bien aux articles qu'aux rubriques.

8.3.9 Les pages de démo

Il n'y a aucun intérêt à découper ces pages en noisettes. Ces pages sont destinées à démontrer certaines fonctionnalités, à savoir, la typographie et le framework ooCSS intégré à notre squelette. Aussi, on migre ces pages en pages de démo SPIP en les transférant dans un nouveau dossier `demo/`. Elles ne seront donc plus détectées comme des pages du squelette.

Pour éviter de modifier les fichiers déplacés de `content/` vers `demo/`, je crée une page `demo/zgala.html` qui ressemble beaucoup à la page `structure.html` de Z-Core et appelle la démo voulue à partir d'un paramètre d'URL nommé `demo` qui peut prendre la valeur `tinytypo` (par défaut) ou `boite`.

Une barre horizontale permet de passer d'une démo à une autre et aussi de revenir à la page d'accueil du site.

8.3.10 La page météo de Rainette

Le cas de la page « meteo » est intéressant car il diffère des autres pages. Cette page est fournie par le plugin Rainette mais sans description XML ou YAML. Il est donc nécessaire de rajouter cette description dans le squelette Z-Gala afin qu'elle soit proposée dans l'interface du noiZetier, ce qui a été fait un peu plus tôt.

Maintenant cette page propose un bloc unique sans noisette alors que le plugin Rainette ne fournit aucun type de noisette. Si l'on veut réorganiser cette page il faut donc définir une nouvelle page météo dans Z-Gala mais aussi des types de noisette (l'une pour les conditions temps réel et l'autre pour les prévisions, à minima).

Je dirais que logiquement un plugin devrait proposer uniquement des noisettes et laisser les pages aux squelettes qui décideront dès lors de son agencement. Le problème est de définir un type de noisette suffisamment générique pour rester compatible avec tous les squelettes Z sachant que chacun peut définir son organisation en blocs et en styles. En cela, les concepts de noisette conteneur et de capsule du noiZetier v3 permettent aujourd'hui de mieux assurer la compatibilité d'un type de noisette avec la multitude des squelettes Z.

Ceci étant, j'ai décidé de supprimer la page météo du plugin Rainette v3, de l'ajouter explicitement au plugin Z-Gala et de créer des noisettes génériques dans le plugin Rainette v3 afin de les utiliser dans le squelette Z-Gala.

8.3.11 Les pages 404 et 401

Les pages 404 et 401 sont nativement fournies par Z-Core. Le squelette Z-Gala a jusque-là uniquement ajouté les fichier YAML de description pour assurer leur détection. Néanmoins, comme souvent, ces pages Z n'ont pas le format souhaité pour être compatible avec notre squelette, c'est pourquoi je reste toujours dubitatif quant à leur inclusion dans un plugin non squelette.

On surcharge donc le HTML fournit par Z-Core en copiant les fichiers du bloc content uniquement. Pour l'instant la modification consiste juste à supprimer le fil d'Ariane inclus dans le fichier.

8.3.12 La navigation secondaire

Pour conclure cette étape d'intégration du noiZetier, nous allons ajouter deux éléments classiques de navigation secondaire dans le bloc aside des pages concernées : le menu des rubriques a priori inclus dans toute page du site et le menu des articles d'une rubrique affichés sur la page d'un article donné.

En créant le type de noisette du menu des rubriques d'identifiant navrub, on duplique en fait le type de noisette navbar en utilisant un menu d'identifiant navrub. Pour l'instant le code des deux types de noisette est un peu redondant mais cela suffit pour cette étape. On propage cette noisette dans le bloc nav de toutes les pages du site.

Pour le type de noisette du menu des articles d'une rubrique on recommence la même opération en créant un type de noisette d'identifiant navart basé sur un menu d'identifiant homonyme. On installe cette noisette uniquement dans le bloc de la page article.

8.4 Conclusion

A l'issue de cette étape, nous avons un squelette dont les blocs principaux, content, breadcrumb, nav et aside sont entièrement personnalisable avec des noisettes. Une liste de types de noisette est déjà disponibles mais demande à être améliorée et leurs styles ne sont pas finalisés mais nous y reviendrons dans une étape ultérieure de finition.

9. MODERNISER LESS AVEC SCSS

Les capacités « programmatiques » de LESS sont quelques peu limitées. Quand on a un préprocesseur fournissant une syntaxe simplifiée et performante des CSS, des variables pour personnaliser les styles on a envie d'aller plus loin et d'écrire du code. Et c'est là que LESS laisse à désirer.

L'alternative qui est aujourd'hui largement utilisée est SCSS, une autre syntaxe de préprocesseur CSS mais qui apporte une véritable capacité à écrire du code en CSS. Cette syntaxe est utilisée par nombre de frameworks et devient quasiment un standard de fait. Il est alors presque naturel de faire évoluer les styles de notre squelette vers du SCSS.

9.1 Convertir les fichiers LESS en SCSS

En premier lieu, il faut convertir l'ensemble des fichiers LESS vers du SCSS. Étant donné la simplicité des styles déjà définis pour le squelette, il est aisé de passer d'une syntaxe à une autre ; je ne m'étendrais donc pas sur la méthode de conversion des instructions. Ce qui est important, c'est que le fichier CSS résultant soit le même, ce qui est bien le cas.

On peut donc s'affranchir du répertoire `less/` et créer un répertoire `scss/` contenant les fichiers SCSS. Le répertoire `css/` lui contient toujours les fichiers résultant, soit maintenant ceux générés à partir des sources SCSS.

Cette opération de conversion doit être faite également pour tous les layouts Gala même si cela ne va être que temporaire. A ce stade, rien n'a changé dans notre squelette mais nos CSS sont maintenant issus de source en SCSS.

9.2 Utiliser le plugin SCSS PHP

Qui dit préprocesseur CSS, dit génération des sources SCSS en fichiers CSS. Pour générer les fichiers CSS il est possible d'utiliser des outils externes en ligne de commande comme `dart-scss`. C'est l'outil que j'utilise via une intégration dans mon IDE PHPStorm : à chaque modification d'un fichier SCSS, mon IDE régénère le fichier CSS correspondant. Pratique non ?

Néanmoins, il existe un plugin SPIP nommé SCSS PHP qui permet de s'abstenir de générer manuellement les CSS et qui réalise cette opération on-line à partir des seuls sources SCSS. Il n'est donc plus nécessaire de conserver les fichiers CSS dans le squelette lui-même, seuls les SCSS sont utiles.

Pour se faire, on commence par rajouter le nécessite du plugin SCSS PHP dans le `paquet.xml` du squelette et bien entendu on installe le plugin.

Étant donné qu'il devient inutile de conserver les fichiers CSS issus de sources SCSS dans le squelette et que `css/` contient aussi des fichiers CSS non issus de sources SCSS, je supprime les fichiers CSS résultant de `css/`, je transfère les fichiers SCSS dans `css/` et je supprime le dossier `scss/`.

Mon dossier `css/` contient maintenant principalement des fichiers SCSS et quelques fichiers CSS.

Ce qui est intéressant, c'est que nous utilisons déjà la balise `#CSS` pour inclure les fichiers CSS dans les pages du squelette. Comme SCSS PHP étend la balise `#CSS` pour prendre en compte la compilation des SCSS si nécessaire, il n'est nul besoin de modifier notre fichier `inclure/head.html`.

9.3 Simplifier l'implémentation des layouts Gala

L'utilisation de SCSS et du plugin SCSS PHP nous permet de simplifier la gestion des layouts Gala. La syntaxe SCSS qui fournit des directives telles que `@if` et `@else`, permet d'envisager de générer les CSS du layout souhaité en intégrant tous les layouts dans un même fichier.

Sans insister sur le code exact – car nous allons le modifier à l'étape suivante – nous intégrons l'ensemble des layouts Gala dans un seul fichier `css/layout.scss` qui s'appuie sur une liste de variables contenues dans `css/_var_layout.scss`. Le dossier `css/layouts/` peut ainsi être supprimé.

Pour fournir le choix du layout provenant de la configuration du squelette (variable `layout` de la meta `zgala`) on utilise le pipeline `scss_variables` de SCSS PHP. Il faut donc au préalable déclarer ce pipeline dans le `paquet.xml` du squelette. La fonction de mise en œuvre du pipeline est la suivante :

```
function zgala_scss_variables($data) {  
  
    include_spip('inc/config');  
    $config = lire_config('zgala', array());  
  
    // Ajout de la configuration du layout  
    $layout = array('layout-gala-id' => $config['layout'] ?? 1);  
    $data = array_merge($data, $layout);  
  
    return $data;  
}
```

A ce stade nous n'avons plus qu'un fichier pour décrire le layout souhaité et tous les CSS paramétrables spécifiques au squelette sont décrits en SCSS.

10. UN LAYOUT RESPONSIVE

10.1 Principes de design responsive

En 2022, il est impossible de proposer un nouveau squelette qui ne soit pas adaptable aux différents écrans que nous utilisons, à savoir, ceux des smartphones, des tablettes ou des desktops. Ces différents écrans possèdent des tailles en pixels qui varient du simple au double voire plus.

L'un des aspects du design responsive est de permettre au site de s'adapter le plus harmonieusement possible à la taille de l'écran. Pour notre squelette, si l'on choisit un layout de base valable par défaut pour des écrans suffisamment grands (on verra plus loin ce que cela veut dire), le design responsive devra adapter ce layout initial : par exemple, passer d'un layout 3 colonnes à un layout 2 colonnes voire à un layout 1 colonne pour les plus petits écrans.

Les points de rupture déclenchant le changement de layout sont appelés des *breakpoints*. Il correspondent, si on simplifie au maximum, à une taille d'écran minimale ou maximale exprimée en pixels (on peut aussi préciser l'orientation du média, portrait ou paysage mais je n'utiliserai pas cette caractéristique). La question est comment définir ces breakpoints ? Existe-t-il des standards ?

La réponse à ces questions est complexe car les avis ne sont pas unanimes. D'aucuns vous dirons qu'il faut définir les breakpoints en fonction du squelette. Soit, mais pour un squelette générique aux layouts multiples comme Z-Gala il me paraît plus raisonnable d'adopter une position médiane comme les frameworks. A ce titre, le site <https://ricostacruz.com/tailwindcss/media-query-breakpoints> rappelle la stratégie choisie par plusieurs frameworks connus.

Framework	Small	Medium	Large	Extra Large
Bulma	-	min 769px	min 1024px	min 1216px
Bootstrap 3	-	min 768px	min 992px	min 1200px
Bootstrap 4	min 576px	min 768px	min 992px	min 1200px
Tailwind	min 576px	min 768px	min 992px	min 1200px
Zurb foundation	-	min 640px	min 1024px	min 1200px

On voit que certaines valeurs sont les mêmes bien (si l'on étend la recherche à d'autres articles on voit apparaître de nouvelles valeurs et de nombreux autres points de rupture). Finalement, je décide de me rapprocher de Bootstrap 4. Néanmoins, étant donné les caractéristiques des layouts Gala, je choisis de ne pas faire de différence entre Large et Extra Large : les **3 breakpoints retenus sont donc 576px, 768px et 992px**.

En retenant 3 breakpoints, on définit donc 4 intervalles pour lesquels il faut définir la stratégie de layout :

- **Taille $\geq 992px$** : on affiche le layout choisi par configuration sans restriction, qu'il soit fixe ou fluide ;
- **Taille $\geq 768px$ et $< 992px$** : on affiche un layout 2 colonnes le plus proche du layout configuré en conservant le caractère fluide ou fixe (les layouts Gala fixes sont initialement définis à 700px) ;

- **Taille $\geq 576\text{px}$ et $< 768\text{px}$** : on affiche un layout 2 colonnes fluides le plus proche du layout configuré ;
- **Taille $< 576\text{px}$** : on affiche un layout une colonne fluide où les blocs breadcrumb, content, aside et extra s'empilent dans cet ordre. C'est le seul layout qui n'est pas Gala.

10.2 Mise en œuvre

10.2.1 Les breakpoints

La mise en place des breakpoints consiste à définir un tableau de valeurs (appelé une map en SCSS) dans le fichier `css/_var_layout.scss`. Les valeurs représentent les bornes minimales des intervalles identifiés dans le paragraphe précédent. Pour anticiper un éventuel besoin de différencier les écrans larges et extra larges, j'ai choisi de définir également l'intervalle extra large (taille $> 1200\text{px}$).

Pour identifier chaque intervalle, ceux-ci sont nommés de façon classique en utilisant l'abréviation courante comme `sm` pour *small*, `md` pour *medium*, `lg` pour *large*, etc. Ainsi, la déclaration est exprimée de la façon suivante :

```
$layout-breakpoints: (
  xs: 0,
  sm: 576px,
  md: 768px,
  lg: 992px,
  xl: 1200px,
) !default;
```

On notera que pour éviter toute collision avec les autres variables SCSS, toutes les variables du fichier du fichier `css/_var_layout.scss` sont préfixées par `layout-`.

10.2.2 Les layouts

Pour mettre en place les CSS des layouts Gala, on utilise, d'une part, une configuration minimale décrivant chaque layout – variable `$layout-config` - et, d'autre part, une mixin `layout-gala($gala-id)` établissant les styles constitutifs d'un layout fourni en paramètre, styles basés sur une grille CSS.

Trouver une heuristique relativement simple pour élaborer une « fonction layout Gala » n'est pas une chose aisée. L'équilibre entre la configuration descriptive de chaque layout et l'automatisation des traitements induits a été difficile à accoucher mais l'utilisation d'une grille CSS a permis d'en simplifier la mise en œuvre. Les paragraphes suivants décrivent quelques principes qui ont conduit à la réalisation de la mixin.

En premier lieu, la structure des layouts Gala appliquée à notre squelette montre que les blocs « header », « nav » et « footer » sont toujours transverses et à la même position dans la page. Il n'ont donc pas d'incidence sur la structure différenciant un layout d'un autre. Cette **structure n'est conditionnée que par les blocs « wrapper », « aside » et « extra »**.

En second lieu, l'utilisation d'une grille CSS pour les 3 blocs précités montre qu'il n'y a que 2 grilles possibles pour structurer l'ensemble des layouts. Ces grilles sont les suivantes :

- **3 colonnes et 1 ligne** pour tous les layouts à 3 colonnes ;
- **2 colonnes et 2 lignes** pour tous les layouts à 2 colonnes.

Avec ces deux grilles il est donc possible de générer les 40 compositions de layout.

En corollaire, on en déduit que pour un layout 3 colonnes le numéro de ligne est toujours égal à 1 pour tous les blocs et qu'il suffit de préciser le numéro de colonne de chaque bloc. Pour un layout 2 colonnes, la grille à 4 cellules implique qu'il y a toujours un bloc qui utilise deux cellules : il est parfois possible de le déduire de la configuration mais dans certains cas il est nécessaire de le définir en configuration.

Enfin, pour terminer la configuration des layouts il est nécessaire de préciser si le **layout est globalement fixe ou fluide** et, éventuellement si les **blocs aside ou extra sont fluides ou fixes**. Le bloc wrapper est toujours considéré comme fluide même si la page est fixe (dans ce cas on considère qu'il prend la place restante).

La configuration des layouts est illustrée ci-après. On notera que les valeurs par défaut ne sont pas incluses dans la configuration pour l'alléger (layout fluide, bloc fluide, bloc sur une ligne ou une colonne).

```
1: {
  'blocks': {
    'wrapper': ('column': 2),
    'aside': ('column': 1),
    'extra': ('column': 3),
  },
}
7: {
  'fixed': true,
  'blocks': {
    'wrapper': ('column': 2),
    'aside': ('column': 1, 'fixed': true),
    'extra': ('column': 3, 'fixed': true),
  },
}
23: {
  'blocks': {
    'wrapper': ('column': 1),
    'aside': ('column': 2, 'fixed': true),
    'extra': ('column': 2, 'fixed': true, 'row': 2),
  },
  'medias': ('md': 23, 'sm': 25),
},
30: {
  'blocks': {
    'wrapper': ('column': 1),
    'aside': ('column': 2),
    'extra': ('column': 1, 'row': 2, 'column-span': 2),
  },
}
35: {
  'fixed': true,
  'blocks': {
    'wrapper': ('column': 1),
    'aside': ('column': 2, 'fixed': true, 'row-span': 2),
    'extra': ('column': 1, 'row': 2),
  },
}
```

On notera que le nombre de colonnes d'un layout est déduit des numéros de colonne.

10.2.3 Les medias queries

Une fois la mixin `layout-gala()` établie, il suffit de l'appeler dans les media queries qui vont matérialiser les intervalles définis préalablement. Pour simplifier la mise en place des CSS liées à un intervalle donné, on définit une mixin `respond-to($breakpoint-id)`, unanimement décrite dans de nombreux articles et utilisée dans les frameworks (par exemple, KNACSS).

L'intérêt de cette mixin est de pouvoir être utilisée partout dans nos fichier SCSS pour définir les styles liées à un intervalle donné : pour anticiper cette possibilité les breakpoints ont été transférés dans le fichier `css/_var_breakpoint.scss` et la mixin dans le fichier `css/_breakpoint.scss` qui est maintenant importé par `css/layout.scss`.

Le code de cette mixin est le suivant :

```
@mixin respond-to($breakpoint-id) {
  // Si le breakpoint existe bien dans la map
  @if map-has-key($layout-breakpoints, $breakpoint-id) {
    // On inscrit une media query avec le breakpoint
    @media (min-width: map-get($layout-breakpoints, $breakpoint-id)) {
      @content;
    }
  }
  // Si le breakpoint n'existe pas dans la map on prévient
  @else {
    @warn "Le breakpoint d'id '#{ $breakpoint-id }' est invalide. "
    + "Vérifier la liste des breakpoints disponibles dans la map '$layout-breakpoints'.";
  }
}
```

L'instruction `@content` sera remplacée à chaque appel par les styles souhaités.

Une fois cette mixin en place, il est nécessaire de mettre en œuvre la stratégie de design responsive que l'on souhaite. Pour notre squelette, en appliquant la logique recommandé de « *mobile first* », la stratégie choisie est la suivante :

1. Pour l'intervalle « xs », la layout empile les blocs wrapper, aside et extra dans cet ordre ;
2. Pour l'intervalle « sm », le layout affiché dérive du layout de base choisi, mais sur 2 colonnes fluides ;
3. Pour l'intervalle « md », le layout affiché dérive du layout de base choisi, mais sur deux colonnes en respectant au maximum la fluidité ou pas des blocs aside et extra ;
4. Pour les intervalles « lg » et « xl », le layout choisi est affiché tel quel.

Pour ce faire, la configuration des layouts a été enrichie par une configuration des layouts alternatifs affichés pour les intervalles « sm » et « md » (« xs » est un layout unique non Gala et « xl » correspond à « lg »). La configuration et le code SCSS des *media queries* est le suivant :

Pour la configuration du layout 20 :

```
'medias': ('md': 27, 'sm': 25),
```

Et le code des *media queries* :

```
/* Layout en mode medium (md) */
@include respond-to(sm) {
  $alternate-id: map-get($layout-config, $layout-id, 'medias', 'sm');
  @include layout-gala($alternate-id);
}
/* Layout en mode medium (md) */
@include respond-to(md) {
  $alternate-id: map-get($layout-config, $layout-id, 'medias', 'md');
  @include layout-gala($alternate-id);
}
/* Layout en mode large (lg) */
@include respond-to(lg) {
  @include layout-gala($layout-id);
}
```

L'intérêt de la configuration est qu'il sera aussi possible d'ajuster les changements de layout si l'on constate que le choix fait ne s'avère pas judicieux. Pour anticiper cette

10.2.4 La configuration

L'une des avancées de SCSS est sa capacité à paramétrer les CSS au travers de variables globales. En outre, l'utilisation du plugin SCSS PHP permet de configurer ces variables dans un formulaire SPIP et de fournir les valeurs aux fichiers SCSS au travers du pipeline `scss_variables`.

En l'état, seul le numéro de layout est utilisé en paramètre des CSS de layout. On étend ce paramétrage rudimentaire aux variables suivantes :

- la largeur de la page nécessaire quand le layout est fixe ;
- la largeur des blocs wrapper, aside et extra en mode fluide et en mode fixe selon le layout utilisé.

Pour simplifier l'accès à ce paramétrage, on enrichit le formulaire unique de configuration du squelette de la saisie de ces nouvelles variables. En outre, on met en place une configuration spécifique du layout en transformant le formulaire `configurer_zgala.html` en `configurer_zgala_layout.html` en précisant un méta casier `zgala/layout`. Ainsi, les variables du layout seront maintenant inscrites dans le tableau associatif d'index 'layout' de la meta `zgala`.

Dans cette étape on ne cherche pas à fournir une interface léchée et cohérente en fonction du layout mais juste un moyen efficace de paramétrer le layout choisi. Nous aborderons la problématique de configuration dans une prochaine étape.

10.2.5 Conclusion

A la fin de cette étape, le squelette a été modernisé par l'utilisation adéquate des SCSS pour construire les layouts Gala, le paramétrer et le rendre responsive. L'introduction d'une grille CSS nativement disponible dans CSS3 a aussi permis de simplifier le codage des styles de bloc.

Z-Gala est prêt à recevoir un premier ensemble de noisettes pour remplir les pages existantes.