Programmer avec SPIP

DOCUMENTATION À L'USAGE DES DÉVELOPPEURS ET WEBMESTRES

Version SPIP 2.0.10+ / 44% documenté / novembre 2009

SPIP est un système de publication et une plateforme de développement. Après un rapide tour d'horizon de SPIP, nous décrirons son fonctionnement technique et expliquerons comment développer avec, en s'attachant à donner des exemples utiles aux programmeurs.

Cette documentation s'adresse à un public de webmestres ayant des connaissances en PHP, SQL, HTML, CSS et JavaScript.

Sommaire

Notes sur cette documentation

Licence et libertés

Fruit de longues heures d'écriture, cette documentation est une somme de connaissances issue de la communauté SPIP. Tout ce travail est distribué sous licence libre Creative Commons - Paternité - Partage des Conditions Initiales à l'Identique (cc-by-sa). Vous pouvez utiliser ces textes quelque soit l'usage (y compris commercial), les modifier et les redistribuer à condition de laisser à vos lecteurs la même liberté de partage.

Une amélioration constante

Cette œuvre - en cours d'écriture - fait l'objet de nombreuses relectures mais n'est certainement pas indemne de toute erreur. N'hésitez pas à proposer des améliorations ou signaler des coquilles en utilisant le formulaire de suggestion mis à disposition sur le site internet de la documentation (http://programmer.spip.org). Vous pouvez aussi discuter de l'organisation (des contenus, de la technique) et des traductions sur la liste de discussion « spip-programmer » (sur abonnement).

Écrire un chapitre

Si vous êtes motivé par ce projet, vous pouvez proposer d'écrire un chapitre sur un sujet que vous maitrisez ou refondre un chapitre existant pour le clarifier ou le compléter. Nous essaierons alors de vous accompagner et vous soutenir dans cette tâche.

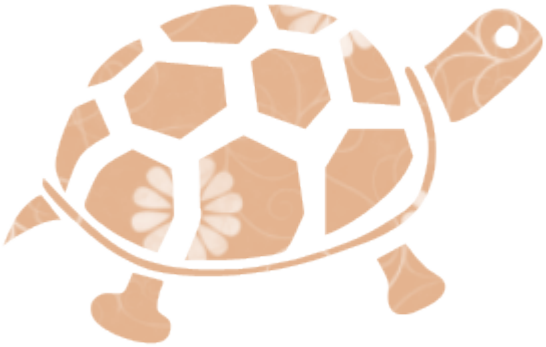
Traductions

Vous pouvez également participer à la traduction de cette documentation en anglais. L'espace privé du site (http://programmer.spip.org) permet de discuter des traductions en cours d'élaboration. Ceci dit, il n'est pas prévu de traduire la documentation dans d'autres langues tant que l'organisation des différents chapitres n'est pas stabilisée, ce qui peut durer encore plusieurs mois.

Codes informatiques et caractéristiques des langues

Par souci de compatibilité, les codes informatiques qui servent d'exemple ne contiennent que des caractères du code ASCII. Cela signifie entre autre que vous ne trouverez aucun accent dans les commentaires accompagnant les exemples dans l'ensemble de la documentation. Ne soyez donc pas étonnés par cette absence.

Bonne lecture.



Introduction

Présentation de SPIP et de son fonctionnement général.

Qu'est-ce que SPIP ?

SPIP 2.0 est un logiciel libre développé sous licence GNU/GPL3. Historiquement utilisé comme un système de publication de contenu, il devient progressivement une plateforme de développement permettant de créer des interfaces maintenables et extensibles quelle que soit la structure des données gérées.

Que peut-on faire avec SPIP ?

SPIP est particulièrement adapté pour des portails éditoriaux mais peut tout aussi bien être utilisé comme système d'auto-publication (blog), de wiki, de réseau social ou pour gérer toute donnée issue de MySQL, PostGres ou SQLite. Des extensions proposent aussi des interactions avec XML.

Comment fonctionne-t-il ?

SPIP 2.0 nécessite a minima PHP 4.1 (et 10 Mo de mémoire pour PHP) ainsi qu'une base de données (MySQL, PostGres ou SQLite).

Il possède une interface publique (front-office), visible de tous les visiteurs du site (ou en fonction d'autorisations particulières) et une interface privée (back-office) seulement accessible aux personnes autorisées et permettant d'administrer le logiciel et les contenus du site.

Des gabarits appelés « squelettes »

Toute l'interface publique (dans le répertoire squelettes-dist) et une partie de l'interface privée (dans le répertoire prive) utilisent, pour s'afficher, des gabarits appelés « squelettes », mélange de code à produire (le plus souvent HTML) et de syntaxe SPIP.

Lorsqu'un visiteur demande à afficher la page d'accueil du site, SPIP va créer une page HTML grâce à un squelette nommé `sommaire.html`. Chaque type de page possède un squelette particulier, tel que `article.html`, `rubrique.html`...

Ces squelettes sont analysés puis compilés en langage PHP. Ce résultat est mis en cache. Ce sont ces fichiers PHP qui servent à produire ensuite les pages HTML renvoyées aux visiteurs d'un site. Pages qui sont elles-aussi mises en cache.

Simple et rapide

Les boucles `<BOUCLE>` sélectionnent des contenus, les balises `#BALISE` les affichent.

Liste des 5 derniers articles :

```
<B_art>
  <ul>
    <BOUCLE_art(ARTICLES){!par date}{0,5}>
      <li><a href="#URL_ARTICLE">#TITRE</a></li>
    </BOUCLE_art>
  </ul>
</B_art>
```

Dans cet exemple, la boucle identifiée `_art` fait une sélection dans la table SQL nommée `ARTICLES`. Elle trie les données par date anti-chronologique `{!par date}` et sélectionne les 5 premiers résultats `{0,5}`. La balise `#URL_ARTICLE` affiche un lien vers la page présentant l'article complet, alors que la balise `#TITRE` affiche le titre de l'article.

Résultat :

```
<ul>
    <li><a href="Recursivite,246">Récursivité</a></li>
    <li><a href="Parametre">Paramètre</a></li>
    <li><a href="Argument">Argument</a></li>
    <li><a href="Adapter-tous-ses-squelettes-en-une">Adapter
tous ses squelettes en une seule opération</a></li>
    <li><a href="Afficher-un-formulaire-d-edition">Afficher
un formulaire d'édition, si autorisé</a></li>
</ul>
```

Écriture des squelettes

SPIP génère des pages HTML à partir de fichiers appelés squelettes contenant un mélange de code HTML, de boucles et de critères, de balises et de filtres. Sa force est de pouvoir extraire du contenu de base de données de manière simple et compréhensible.

Boucles

Une `boucle` permet de sélectionner du contenu issu d'une base de données. Concrètement, elle sera traduite par une requête SQL optimisée permettant d'extraire le contenu demandé.

Syntaxe des boucles

Une boucle déclare donc une table SQL, sur laquelle extraire les informations, ainsi que des `critères` de sélection.

```
<BOUCLE_nom(TABLE){critere}{critere}>
 ... pour chaque réponse...
</BOUCLE_nom>
```

Une boucle possède obligatoirement un nom (identifiant unique à l'intérieur d'un même squelette), ce nom est accolé au mot `BOUCLE`. Ici donc, le nom de la boucle est « _nom ».

La table est définie soit par un alias (alors écrit en majuscules), soit par le nom réel de la table, alors écrit en minuscules, par exemple « spip_articles ».

Les critères sont écrits ensuite, entre accolades, par exemple `{par nom}` pour trier les résultats dans l'ordre alphabétique selon le champ « nom » de la table SQL en question.

Exemple

Cette boucle liste toutes les images du site. Le critère `{extension IN jpg,png,gif}` permet de sélectionner les fichiers possédant une extension parmi les trois listées.

```
<BOUCLE_documents(DOCUMENTS){extension IN jpg,png,gif}>
    [(#FICHIER|image_reduire{300})]
</BOUCLE_documents>
```

La balise `#FICHIER` contient l'adresse du document, auquel on applique un `filtre` nommé « image_reduire » qui redimensionne l'image automatiquement à 300 pixels si sa taille est plus grande et retourne une balise HTML permettant d'afficher l'image (balise `<img>`)

Syntaxe complète des boucles

Les boucles, comme d'ailleurs les balises, possèdent une syntaxe permettant de multiples compositions. Des parties optionnelles s'affichent une seule fois (et non pour chaque élément). Une partie alternative s'affiche uniquement si la boucle ne renvoie aucun contenu. Voici la syntaxe (`x` étant l'identifiant de la boucle) :

```
<Bx>
    une seule fois avant
<BOUCLEx(TABLE){critère}>
    pour chaque élément
</BOUCLEx>
    une seule fois après
</Bx>
    afficher ceci s'il n'y a pas de résultat
<//Bx>
```

Exemple

Cette boucle sélectionne les 5 derniers articles publiés sur le site. Ici, les balises HTML `<ul>` et `</ul>` ne seront affichées qu'une seule fois, et uniquement si des résultats sont trouvés pour les critères de sélection. Si aucun article n'était publié, les parties optionnelles de la boucle ne s'afficheraient pas.

```
<B_derniers_articles>
  <ul>
<BOUCLE_derniers_articles(ARTICLES){!par date}{0,5}>
  <li>#TITRE, <em>[(#DATE|affdate)]</em></li>
</BOUCLE_derniers_articles>
  </ul>
</B_derniers_articles>
```

La balise `#DATE` affiche la date de publication de l'article. On lui affecte un `filtre` « affdate » supplémentaire permettant d'écrire la date dans la langue du contenu.

Résultat :

```
<ul>
  <li>Contenu d'un fichier exec (squelette), <em>13
octobre 2009</em></li>
  <li>Liens AJAX, <em>1er octobre 2009</em></li>
  <li>Forcer la langue selon le visiteur, <em>27
septembre 2009</em></li>
  <li>Definition, <em>27 September 2009</em></li>
  <li>List of current pipelines, <em>27 September
2009</em></li>
</ul>
```

Les boucles imbriquées

Il est souvent utile d'imbriquer des boucles les unes dans les autres pour afficher ce que l'on souhaite. Ces imbrications permettent d'utiliser des valeurs issues d'une première boucle comme critère de sélection de la seconde.

```
<BOUCLEx(TABLE){criteres}>
    #ID_TABLE
    <BOUCLEy(AUTRE_TABLE){id_table}>
        ...
    </BOUCLEy>
</BOUCLEx>
```

Exemple

Ici, nous listons les articles contenus dans les premières rubriques du site grâce au critère `{racine}` qui sélectionne les rubriques de premier niveau (à la racine du site), que l'on appelle généralement « secteur » :

```
<B_rubs>
  <ul class='rubriques'>
```

```
  <BOUCLE_rubs(RUBRIQUES){racine}{par titre}>
    <li>#TITRE
      <B_arts>
        <ul class='articles'>
        <BOUCLE_arts(ARTICLES){id_rubrique}{par titre}>
          <li>#TITRE</li>
        </BOUCLE_arts>
        </ul>
      </B_arts>
    </li>
  </BOUCLE_rubs>
  </ul>
</B_rubs>
```

La boucle `ARTICLES` contient simplement un critère de tri `{par titre}` et un critère `{id_rubrique}`. Ce dernier indique de sélectionner les articles appartenant à la même rubrique.

Résultat :

```
<ul class='rubriques'>
    <li>en
    </li>
    <li>fr
        <ul class='articles'>
            <li>Notes sur cette documentation</li>
            <li>Autre article</li>
        </ul>
    </li>
</ul>
```

Les boucles récursives

Une boucle dite récursive (n), contenue dans une boucle parente (x), permet d'exécuter la boucle (x) une nouvelle fois, en transmettant automatiquement les paramètres nécessaires. Donc, à l'intérieur de la boucle (x), on appelle cette même boucle (c'est ce qu'on nomme la « récursion ») avec d'autres arguments. Ce processus se répètera tant que la boucle appelée retourne des résultats.

```
<BOUCLEx(TABLE){id_parent}>
  ...
  <BOUCLEn(BOUCLEx) />
  ...
</BOUCLEx>
```

Lorsqu'un site possède de nombreuses sous-rubriques, ou de nombreux messages de forums, ces boucles récursives sont souvent utilisées. On peut ainsi afficher facilement des éléments identiques hiérarchisés.

Exemple

Nous allons de cette manière afficher la liste complète des rubriques du site. Pour cela, nous bouclons une première fois sur les rubriques, avec un critère pour sélectionner les rubriques filles de la rubrique en cours : `{id_parent}`. Nous trions aussi par numéro (un rang donné aux rubriques pour les afficher volontairement dans un certain ordre), puis par titre de rubrique.

```
<B_rubs>
  <ul>
  <BOUCLE_rubs(RUBRIQUES){id_parent}{par num titre,
titre}>
    <li>#TITRE
    <BOUCLE_sous_rubs(BOUCLE_rubs) />
    </li>
  </BOUCLE_rubs>
  </ul>
</B_rubs>
```

Au premier passage dans la boucle, `id_parent` va lister les rubriques à la racine du site. Elles ont le champ SQL `id_parent` valant zéro. Une fois la première rubrique affichée, la boucle récursive est appelée. SPIP appelle de nouveau la boucle « _rubs ». Cette fois la sélection `{id_parent}` n'est plus la même car ce critère liste les rubriques filles de la rubrique en cours. S'il y a effectivement des sous-rubriques, la première est affichée. Puis aussitôt et une nouvelle fois, mais dans cette sous-rubrique, la boucle « _rubs » est exécutée. Tant qu'il y a des sous rubriques à afficher, ce processus récursif recommence.

Ce qui donne :

```
<ul>
<li>en
    <ul>
        <li>Introduction</li>
        <li>The templates
            <ul>
                <li>Loops</li>
            </ul>
        </li>
        <li>Extending SPIP
            <ul>
                <li>Introduction</li>
                <li>Pipelines</li>
                ...
            </ul>
        </li>
        ...
    </ul>
</li>
<li>fr
    <ul>
        <li>Introduction</li>
        <li>Écriture des squelettes
            <ul>
                <li>Boucles</li>
                <li>Balises</li>
                <li>Critères de boucles</li>
                ...
            </ul>
        </li>
        ...
    </ul>
</li>
</ul>
```

En savoir plus !

Comprendre les principes de la récursivité en programmation n'est pas forcément facile. Si ce qui est expliqué ici vous laisse perplexe, lisez l'article consacré de SPIP.net qui explique cela avec d'autres mots : http://www.spip.net/fr_article914.html

Boucle sur une table absente

Lorsqu'on demande à SPIP d'interroger une table qui n'existe pas, celui-ci affiche une erreur sur la page pour tous les administrateurs du site.

Cependant cette absence peut être parfois justifiée, par exemple si l'on interroge une table d'un plugin qui peut être actif ou non. Pour cela un point d'interrogation placé juste avant la fin de la parenthèse permet d'indiquer que l'absence de la table est tolérée :

```
<BOUCLE_table(TABLE ?){criteres}>
   ...
</BOUCLE>
```

Exemple

Si un squelette utilise le plugin « Agenda » (qui propose la table EVENEMENTS), mais que ce squelette doit fonctionner même en absence du plugin, il est possible d'écrire ses boucles :

```
<BOUCLE_events(EVENEMENTS ?){id_article}{!par date}>
   ...
</BOUCLE_events>
```

Balises

Les balises servent la plupart du temps à afficher ou calculer des contenus. Ces contenus peuvent provenir de différentes sources :

- de l'environnement du squelette, c'est à dire de certains paramètres transmis au squelette ; on parle alors de contexte de compilation.
- du contenu d'une table SQL à l'intérieur d'une boucle
- d'une autre source spécifique. Dans ce cas là, les balises et leurs actions doivent obligatoirement être indiquées à SPIP alors que les 2 sources précédentes peuvent être calculées automatiquement.

Syntaxe complète des balises

Comme les boucles, les balises ont des parties optionnelles, et peuvent avoir des arguments. Les étoiles annulent des traitements automatiques.

```
#BALISE
#BALISE{argument}
#BALISE{argument, argument, argument}
#BALISE*
#BALISE**
[(#BALISE)]
[(#BALISE{argument})]
[(#BALISE*{argument})]
[ avant (#BALISE) apres ]
[ avant (#BALISE{argument}|filtre) apres ]
[ avant (#BALISE{argument}|filtre{argument}|filtre) apres ]
...
```

Règle de crochets

L'écriture complète, avec parenthèses et crochets est obligatoire dès lors qu'un des arguments de la balise utilise aussi parenthèses et crochets ou lorsque la balise contient un filtre.

```
// risque de mauvaises surprises :
#BALISE{[(#BALISE|filtre)]}
// interpretation toujours correcte :
[(#BALISE{[(#BALISE|filtre)]})]
// bien que cette ecriture fonctionne en SPIP 2.0, elle n'est
pas garantie :
#BALISE{#BALISE|filtre}
// l'utilisation d'un filtre exige crochets et parentheses :
```

```
[(#BALISE|filtre)]
```

Exemple

Afficher un lien vers la page d'accueil du site :

```
<a href="#URL_SITE_SPIP">#NOM_SITE_SPIP</a>
```

Afficher une balise HTML `<div>` et le contenu d'un `#SOUSTITRE` s'il existe :

```
[<div class="soustitre">(#SOUSTITRE)</div>]
```

L'environnement #ENV

On appelle environnement l'ensemble des paramètres qui sont transmis à un squelette donné. On parlera aussi de contexte de compilation.

Par exemple, lorsqu'un visiteur demande à afficher l'article 92, l'identifiant de l'article (92) est transmis au squelette `article.html`. A l'intérieur de ce squelette là, il est possible de récupérer cette valeur grâce à une balise spéciale : `#ENV`. Ainsi `#ENV{id_article}` afficherait "92".

Certains paramètres sont automatiquement transmis au squelette, comme la date actuelle (au moment du calcul de la page) affichable avec `#ENV{date}`. De la même manière, si l'on appelle un squelette avec des arguments dans l'URL de la page, ceux-ci sont transmis à l'environnement.

Exemple

L'URL `spip.php?page=albums&type=classique` va charger un squelette `albums.html`. Dedans, `#ENV{type}` permettra de récupérer la valeur transmise, ici « classique ».

Contenu des boucles

Le contenu extrait des sélections réalisées avec des boucles SPIP est affiché grâce à des balises. Automatiquement, lorsqu'une table possède un champ SQL « x », SPIP pourra afficher son contenu en écrivant #X.

```
<BOUCLEx(TABLES)>
#X - #NOM_DU_CHAMP_SQL - #CHAMP_INEXISTANT<br />
</BOUCLEx>
```

SPIP ne créera pas de requête SQL de sélection totale (`SELECT * ...`) pour récupérer les informations demandées, mais bien, à chaque fois, des sélections spécifiques : ici, ce serait `SELECT x, nom_du_champ_sql FROM spip_tables`.

Lorsqu'un champ n'existe pas dans la table SQL, comme ici « champ_inexistant », SPIP ne le demande pas dans la requête, mais essaie alors de le récupérer dans une boucle parente – si il y en a – lorsque le champ existe dans la table SQL concernée. Si aucune boucle parente ne possède un tel champ, SPIP le cherche alors dans l'environnement, comme si l'on écrivait `#ENV{champ_inexistant}`.

Exemple

Imaginons une table SQL "chats" contenant 5 colonnes « id_chat », « race », « nom », « age », « couleur ». On pourra lister son contenu de la sorte :

```
<B_chats>
  <table>
    <tr>
      <th>Nom</th><th>Age</th><th>Race</th>
    </tr>
    <BOUCLE_chats(CHATS){par nom}>
      <tr>
        <td>#NOM</td><td>#AGE</td><td>#RACE</td>
      </tr>
    </BOUCLE_chats>
  </table>
</B_chats>
```

Automatiquement, SPIP, en analysant le squelette, comprendra qu'il doit récupérer les champs `nom`, `age` et `race` dans la table SQL `chats`. Cependant, il n'ira pas récupérer les champs dont il n'a pas besoin (ici `couleur`), ce qui évite donc de surcharger le serveur de base de données en demandant des champs inutiles.

Contenu de boucles parentes

Il est parfois utile de vouloir récupérer le contenu d'une boucle parente de celle en cours, à travers une balise. SPIP dispose d'une écriture pour cela (n étant l'identifiant de la boucle voulue) :

```
#n:BALISE
```

Exemple

Afficher systématiquement le titre de la rubrique en même temps que le titre de l'article :

```
<BOUCLE_rubs(RUBRIQUES)>
  <ul>
    <BOUCLE_arts(ARTICLES){id_rubrique}>
      <li>#_rubs:TITRE - #TITRE</li>
    </BOUCLE_arts>
  </ul>
</BOUCLE_rubs>
```

Balises prédéfinies

Nous l'avons vu, nous pouvons extraire avec les balises des contenus issus de l'environnement ou d'une table SQL. Il existe d'autres balises qui ont des actions spéciales explicitement définies.

Dans ces cas là, elles sont déclarées (dans SPIP) soit dans le fichier ecrire/public/balises.php, soit dans le répertoire ecrire/balise/

Voici quelques exemples :

- `#NOM_SITE_SPIP` : retourne le nom du site
- `#URL_SITE_SPIP` : retourne l'url du site (sans le / final)
- `#CHEMIN` : retourne le chemin d'un fichier `#CHEMIN{javascript/jquery.js}`
- `#CONFIG` : permet de récupérer des informations sur la configuration du site (stockée en partie dans la table SQL « spip_meta »). `#CONFIG{version_installee}`
- `#SPIP_VERSION` : affiche la version de SPIP
- ...

Nous en verrons bien d'autres par la suite.

Balises génériques

SPIP dispose de moyens puissants pour créer des balises particulières pouvant s'adapter au contexte de la page, de la boucle ou simplement au nom de la balise.

Ainsi, il est possible de déclarer des balises qui auront toutes le même préfixe et effectueront ainsi un traitement commun propre à chaque type de balise.

Ces types de balises sont déclarées dans le répertoire ecrire/balise/. Ce sont les fichiers `*_.php`.

On trouve ainsi :

- `#LOGO_` pour afficher des logos d'article, de rubrique ou autre : `#LOGO_ARTICLE`
- `#URL_` pour déterminer une URL d'un objet SPIP, comme `#URL_MOT` à l'intérieur d'une boucle `MOTS`
- `#FORMULAIRE_` pour afficher un formulaire défini dans le répertoire `/formulaires` tel que `#FORMULAIRE_INSCRIPTION`

Traitements automatiques des balises

La plupart des balises SPIP, dont toutes celles issues de la lecture de la base de données effectuent des traitements automatiques pour bloquer des codes malveillants qui pourraient être ajoutés par des rédacteurs au moment de l'écriture de l'article (du code PHP ou des scripts JavaScript).

En plus de ces traitements, d'autres peuvent être définis pour chaque champ SQL afin de faire appliquer automatiquement les traitements sur le champ en question. Ces opérations sont définies dans le fichier ecrire/public/interfaces.php par un tableau global `$table_des_traitements`. La clé du tableau est le nom de la balise, la valeur un tableau associatif : sa clé « 0 » définit un traitement quelle que soit la table concernée, une clé « nom_de_la_table » (sans le préfixe) définit un traitement pour une balise d'une table spécifique.

Les traitements sont donnés par une chaîne de caractères `fonction(%s)` explicitant les fonctions à appliquer. Dedans, « %s » sera remplacé par le contenu du champ.

```
$table_des_traitements['BALISE'][]= 'traitement(%s)';
$table_des_traitements['BALISE']['objets']= 'traitement(%s)';
```

Deux usages fréquents des filtres automatiques sont définis par des constantes pouvant être utilisées :

- `_TRAITEMENT_TYPO` applique les traitements typographiques,
- `_TRAITEMENT_RACCOURCIS` applique les traitements typographiques et les traductions des raccourcis SPIP.

Exemple

Les balises `#TITRE` et `#TEXTE` reçoivent des traitements, qui s'appliquent quelle que soit la boucle, définis de cette façon :

```
$table_des_traitements['TEXTE'][]=
_TRAITEMENT_RACCOURCIS;
$table_des_traitements['TITRE'][]= _TRAITEMENT_TYPO;
```

La balise `#FICHIER` effectue un traitement uniquement dans les boucles documents :

```
$table_des_traitements['FICHIER']['documents']=
'get_spip_doc(%s)';
```

Empêcher les traitements automatiques

Les traitements de sécurité et les traitements définis s'appliquent automatiquement sur les balises, mais il est possible d'éviter cela pour certaines particularité d'un squelette. L'extension « étoile » d'une balise est conçue pour :

```
// tous les traitements
#BALISE
// pas les traitements definis
#BALISE*
// meme pas les traitements de securite
#BALISE**
```

Exemple

Retarder l'application des traitements typographiques et des raccourcis SPIP sur le texte d'une page (le filtre `propre` est appliqué normalement automatiquement), pour ajouter, avant, un filtre effectuant une action quelconque :

```
[<div
class="texte">(#TEXTE*|filtre_quelconque|propre)</div>]
```

Des balises à connaître

Dans le jeu de balises spécifiques dont dispose SPIP par défaut, un certain nombre sont assez souvent utilisées et doivent donc être mentionnées ici.

Nom	Description
#AUTORISER (p.29)	Tester des autorisations
#CACHE (p.29)	Définir la durée du cache
#CHEMIN (p.30)	Retrouver l'adresse d'un fichier
#DESCRIPTIF_SITE_SPIP (p.30)	Retourner le descriptif du site
#EDIT (p.31)	Éditer du contenu (avec le plugin « crayons »)
#ENV (p.31)	Récupérer une variable dans l'environnement
#EVAL (p.32)	Évaluer une expression via PHP
#EXPOSE (p.32)	Mettre en évidence l'élément en cours de lecture (dans une liste, un menu)
#GET (p.33)	Récupérer une valeur stockée par `#SET`
#INCLURE (p.35)	Inclure un squelette
#INSERT_HEAD (p.36)	Balise d'insertion de scripts dans le `<head>` pour SPIP ou des plugins
#INTRODUCTION (p.36)	Afficher une introduction
#LANG (p.37)	Obtenir le code de langue
#LANG_DIR (p.38)	Retourner le sens d'écriture
#LESAUTEURS (p.38)	Afficher les auteurs d'un article
#MODELE (p.39)	Insère un modèle de mise en page
#NOTES (p.40)	Afficher les notes crées avec le raccourci SPIP `[[]]`
#REM (p.41)	Mettre un commentaire dans le code
#SELF (p.41)	Retourne l'URL de la page courante
#SESSION (p.42)	Récupère une information de session
#SESSION_SET (p.42)	Définir des variables de session
#SET (p.43)	Stocker une valeur, récupérable avec `#GET`

Nom	Description
#VAL (p.43)	Retourne une valeur

#AUTORISER

#AUTORISER permet de tester des autorisations d'accès à du contenu, de gérer des affichages spécifiques pour certains visiteurs. Un chapitre spécifique (Gestion d'autorisations (p.147)) est consacré à cette problématique.

```
[(#AUTORISER{action,objet,identifiant}) Je suis autorisé ]
```

La présence de cette balise, comme la balise #SESSION génère un cache différent pour chaque visiteur authentifié sur le site, et un cache pour les visiteurs non authentifiés.

Exemple

Tester si un visiteur à le droit

- de voir un article donné,
- de modifier un article donné

```
[(#AUTORISER{voir,article,#ID_ARTICLE}) Je suis autorisé
à voir l'article]
[(#AUTORISER{modifier,article,#ID_ARTICLE}) Je suis
autorisé à modifier l'article]
```

#CACHE

#CACHE{duree} permet de définir la durée de validité du cache d'un résultat de calcul d'un squelette, exprimée en secondes. Lorsque cette durée est dépassée, le squelette est calculé de nouveau.

Cette balise est généralement placée au tout début des squelettes. En son absence, par défaut, la durée est de 24h (défini par la constante _DUREE_CACHE_DEFAUT).

Exemple

Définir un cache d'une semaine :

```
#CACHE{3600*24*7}
```

#CHEMIN

`#CHEMIN{repertoire/fichier.ext}` retourne l'adresse relative d'un fichier dans l'arborescence de SPIP. Lire à ce sujet La notion de chemin (p.84).

Exemple

Retourner l'adresse du fichier « habillage.css ». S'il existe dans le dossier `squelettes/`, c'est cette adresse qui sera donnée, sinon ce sera l'adresse du fichier présent dans le répertoire `squelettes-dist/`.

```
#CHEMIN{habillage.css}
```

Le fichier squelettes-dist/inc-head.html l'utilise pour charger la feuille de style correspondante dans la partie `<head>` du code HTML. Si le fichier est trouvé, la balise HTML`<link>` est affichée.

```
[<link rel="stylesheet"
href="(#CHEMIN{habillage.css}|direction_css)" type="text/
css" media="projection, screen, tv" />]
```

Notons que le filtre `direction_css` permet d'inverser toute la feuille de style CSS (`left` par `right` et inversement) si le contenu du site est dans une langue s'écrivant de droite à gauche.

#DESCRIPTIF_SITE_SPIP

`#DESCRIPTIF_SITE_SPIP` retourne le descriptif du site défini dans la page de configuration de l'interface privée.

Exemple

Dans la partie `<head>` du code HTML, il est ainsi possible de définir la méta « description » avec cette balise, particulièrement utile sur la page d'accueil du site (fichier `sommaire.html`).

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

Notons que le filtre `couper{150}` coupe le contenu à 150 caractères (en évitant de couper un mot en deux) ; le filtre `textebrut` supprime toute balise HTML.

#EDIT

`#EDIT{nom_du_champ}` : cette balise seule, ne fait rien et ne renvoie rien… Mais couplée avec le plugin « crayons », elle permet d'éditer des contenus depuis l'interface publique si on y est autorisé. Elle retourne dans ce cas des noms de classes CSS qui seront utilisées par un script jQuery fourni par ce plugin.

```
<div class="#EDIT{champ}">#CHAMP</div>
```

Exemple

Pouvoir éditer le champ « titre » :

```
<h2[ class="(#EDIT{titre})"]>#TITRE</h2>
<h2 class="#EDIT{titre} autre_classe">#TITRE</h2>
```

#ENV

`#ENV{parametre}` – nous l'avons vu (L'environnement #ENV (p.22)) – récupère des variables d'environnement transmises au squelette. Un second argument permet de donner une valeur par défaut si le paramètre demandé n'est pas présent dans l'environnement ou si son contenu est vide.

```
#ENV{parametre, valeur par défaut}
```

La valeur du paramètre récupéré est automatiquement filtrée avec `entites_html`, qui transforme le texte en entité HTML (`<` devient ainsi `<`). Pour éviter cet échappement, on peut utiliser une étoile :

```
#ENV*{parametre, valeur par défaut}
```

Enfin, la balise `#ENV` toute seule retourne un tableau sérialisé de tous les paramètres d'environnement.

Exemple

Récupérer un identifiant d'article, sinon la chaîne « new » :

```
#ENV{id_article,new}
```

Afficher tout l'environnement (utile pour débugguer) :

```
[<pre>(#ENV|unserialize|print_r{1})</pre>]
```

#EVAL

`#EVAL{expression}`, très peu usité, permet d'afficher un résultat d'une évaluation par PHP de l'expression transmise.

Exemple

```
#EVAL{3*8*12}
#EVAL{_DIR_PLUGINS}
#EVAL{$GLOBALS['meta']}
```

#EXPOSE

#EXPOSE permet de mettre en valeur un résultat dans une liste. Lorsqu'on boucle sur une table et que #ENV{id_table} est présent dans l'environnement, ou #ID_TABLE dans une boucle de niveau supérieur, alors #EXPOSE renverra un code particulier si la boucle passe sur la même valeur d'identifiant.

Sa syntaxe est :

```
#EXPOSE{texte si oui}
#EXPOSE{texte si oui, texte si non}
// expose tout seul renvoie 'on' ou ''
#EXPOSE
```

Exemple

Lister les articles de la rubrique en cours, en affectant une classe CSS « on » sur l'article actuel.

```
<ul>
<BOUCLE_arts(ARTICLES){id_rubrique}{par num titre,
titre}>
  <li[ class="(#EXPOSE{on})"]>#TITRE</li>
</BOUCLE_arts>
</ul>
```

Résultat :

```
<ul>
  <li>#AUTORISER</li>
  ...
  <li>#ENV</li>
  <li>#EVAL</li>
  <li class="on">#EXPOSE</li>
  ...
</ul>
```

#GET

`#GET{variable}` permet de récupérer la valeur d'une variable locale stockée avec `#SET{variable, valeur}`. Voir aussi #SET (p.43).

Un second argument permet de récupérer une valeur par défaut si le paramètre demandé n'existe pas ou si son contenu est vide.

```
#GET{variable, valeur par défaut}
```

Exemple

Si « utiliser_documentation » vaut « oui », le dire :

```
#SET{utiliser_documentation,oui}
[(#GET{utiliser_documentation}|=={oui}|oui)
    On utilise la documentation !
]
```

Afficher un lien vers la page d'accueil du site, sur une image « mon_logo.png » si elle existe, sinon sur « logo.png », sinon sur le logo du site :

```
[(#SET{image,[(#CHEMIN{mon_logo.png}
    |sinon{#CHEMIN{logo.png}}
    |sinon{#LOGO_SITE_SPIP})]})]
[<a href="#URL_SITE_SPIP/">(#GET{image}
    |image_reduire{100})</a>]
```

Différencier l'absence d'un élément dans l'environnement : définir comme valeur par défaut `#ENV{defaut}` lorsque `#ENV{activer}` n'existe pas. Pour cela, le filtre `is_null` nous permet de tester que `#ENV{activer}` n'est pas défini. Si `#ENV{activer}` existe mais est vide, il sera utilisé. On peut ainsi différencier le cas de l'envoi d'une valeur vide dans un formulaire, comme ci-dessous lorsque la valeur envoyée est celle de l'input « champ_activer_non »

```
[(#SET{valeur,[(#ENV{activer}
    |is_null|?{#ENV{defaut},#ENV{activer}})]})]
```

```
<input type="radio" name="activer"
id="champ_activer_oui"[
(#GET{valeur}|oui)checked='checked'] value='on'  />
<label for="champ_activer_oui"><:item_oui:></label>
<input type="radio" name="activer"
id="champ_activer_non"[
(#GET{valeur}|non)checked='checked'] value='' />
<label for="champ_activer_non"><:item_non:></label>
```

#INCLURE

#INCLURE permet d'ajouter le résultat d'une inclusion dans le squelette en cours. On parle d'inclusion « statique » car le résultat de compilation est ajouté au squelette en cours, dans le même fichier de cache. Cette balise est donc différente d'une inclusion « dynamique » avec <INCLURE.../> qui, elle, crée un fichier de cache séparé (avec une durée de cache qui lui est propre).

```
// ecriture a preferer
[(#INCLURE{fond=nom_du_squelette, argument, argument=xx})]
// autre ecriture comprise, mais a eviter
[(#INCLURE{fond=nom_du_squelette}{argument}{argument=xx})]
```

Si du point de vue du résultat visible, utiliser <INCLURE> ou #INCLURE provoque un affichage identique, du point de vue interne la gestion est différente. L'inclusion dynamique <INCLURE> va générer plus de fichiers de cache autonomes. L'inclusion statique #INCLURE crée moins de fichiers, mais tous de plus grosse taille car le contenu inclus est alors dupliqué sur chaque page en cache.

Exemple

Ajouter au squelette en cours le contenu résultant de la compilation du squelette « inc-navigation.html », auquel on passe le contexte « id_rubrique »

```
[(#INCLURE{fond=inc-navigation, id_rubrique})]
```

Nota : les inclusions `inc-head`, `inc-navigation` des squelettes par défaut de SPIP sont appelées par des inclusions dynamiques, et non statiques comme cet exemple.

#INSERT_HEAD

`#INSERT_HEAD` placé entre les balises HTML `<head>` et `</head>` permet d'ajouter automatiquement certains scripts JavaScript ou CSS. Certains scripts sont ajoutés par défaut par SPIP (jQuery par exemple), d'autres par des plugins. Se référer aux pipelines insert_head (p.0) et jquery_plugins (p.0) qui s'occupent d'ajouter ces scripts.

Dans les squelettes par défaut de SPIP, cette balise est insérée à la fin du squelette squelettes-dist/inc-head.html.

#INTRODUCTION

`#INTRODUCTION` affiche un extrait du contenu d'un champ SQL « texte » (si la table possède ce champ). Dans le cas des articles, cet extrait est puisé dans le champ « descriptif », sinon dans le « chapo », sinon dans le champ « texte ». L'extrait peut aussi être défini, au moment de la rédaction du contenu, en encadrant l'introduction souhaitée par des balises `<intro>` et `</intro>`.

Un argument permet de définir la longueur maximum de l'introduction :

```
#INTRODUCTION{longueur}
```

Exemple

Donner à la balise HTML meta « description » un texte introductif sur les pages articles (exemple dans squelettes-dist/article.html) :

```
<BOUCLE_principale(ARTICLES) {id_article}>
...
[<meta name="description"
content="(#INTRODUCTION{150}|attribut_html)" />]
```

```
...
</BOUCLE_principale>
```

Afficher les 10 derniers articles avec une introduction de leur contenu :

```
<B_articles_recents>
    <h2><:derniers_articles:></h2>
    <ul>
        <BOUCLE_articles_recents(ARTICLES) {!par date}
{0,10}>
        <li>
            <h3><a href="#URL_ARTICLE">#TITRE</a></h3>
            [<div class="#EDIT{intro}
introduction">(#INTRODUCTION)</div>]
        </li>
        </BOUCLE_articles_recents>
    </ul>
</B_articles_recents>
```

#LANG

#LANG affiche le code de langue, pris dans l'élément le plus proche de la balise. Si la balise est placée dans une boucle, #LANG renverra le champ SQL « lang » de la boucle s'il existe, sinon, celui de la rubrique parente, sinon celui de l'environnement (#ENV{lang}), sinon la langue principale du site (#CONFIG{langue_site}).

#LANG* permet de ne retourner que la langue d'une boucle ou de l'environnement. Si aucune n'est définie, la balise ne renvoie alors rien (elle ne retourne donc pas la langue principale du site).

Exemple

Définir la langue dans la balise HTML de la page :

```
<html xmlns="http://www.w3.org/1999/xhtml"
xml:lang="#LANG" lang="#LANG" dir="#LANG_DIR">
```

Définir la langue dans un flux RSS (exemple issu de squelettes-dist/backend.html) :

```
<rss version="2.0"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:content="http://purl.org/rss/1.0/modules/
content/"
>
<channel[ xml:lang="(#LANG)"]>
    <title>[(#NOM_SITE_SPIP|texte_backend)]</title>
    ...
    <language>#LANG</language>
    <generator>SPIP - www.spip.net</generator>
    ...
</channel>
</rss>
```

#LANG_DIR

#LANG_DIR retourne le sens d'écriture d'un texte en fonction de la langue, soit « ltr » (pour « left to right »), soit « rtl » (pour « right to left »). Comme #LANG, la langue est prise dans la boucle la plus proche ayant un champ « lang », sinon dans l'environnement, sinon dans la langue principale du site. Cette balise est très pratique pour des sites multilingues qui mélangent des langues n'ayant pas le même sens d'écriture.

Exemple

Afficher le texte d'une rubrique dans le sens qui lui convient :

```
<BOUCLE_afficher_contenu(RUBRIQUES){id_rubrique}>
<div dir='#LANG_DIR'>#TEXTE</div>
</BOUCLE_afficher_contenu>
```

#LESAUTEURS

#LESAUTEURS affiche la liste du ou des auteurs d'un article (ou d'un article syndiqué), séparés par une virgule. Lorsque le champ SQL « lesauteurs » n'existe pas sur la table demandée, comme sur la table des articles, cette balise charge un modèle de mise en page squelettes-dist/modeles/lesauteurs.html.

Exemple

Dans une boucle ARTICLES, indiquer les auteurs :

```
<small>[<:par_auteur:> (#LESAUTEURS)]</small>
```

#MODELE

#MODELE{nom} insère le résultat d'un squelette contenu dans le répertoire modeles/. L'identifiant de la boucle parente est transmis par défaut avec le paramètre « id » à cette inclusion.

Des arguments supplémentaires peuvent être transmis :

```
// ecriture a preferer
[(#MODELE{nom, argument=xx, argument})]
// autre ecriture comprise, mais a eviter
[(#MODELE{nom}{argument=xx}{argument})]
```

Ces inclusions peuvent aussi être appelées depuis la rédaction d'un article (avec une écriture spécifique) :

```
// XX est l'identifiant de l'objet à transmettre.
<nomXX>
// arguments avec des |
<nomXX|argument=xx|argument2=yy>
```

Exemple

Lister les différentes langues de traductions d'un article, avec un lien pour y accéder :

```
<BOUCLE_art(ARTICLES){id_article}>
#MODELE{article_traductions}
</BOUCLE_art>
```

#NOTES

`#NOTES` affiche les notes (renvois en bas de page) qui ont été calculées par l'affichage des balises précédentes. Ainsi si une balise, dont on calcule les raccourcis SPIP avec le filtre `propre`, ou avec un traitement automatique, contient des notes, elles pourront être affichées avec `#NOTES`, après leur calcul.

```
[(#BALISE|propre)]
#TEXTE
#NOTES
```

Précisions sur les notes

C'est la fonction `traiter_raccourcis()` appelée par le filtre `propre` qui exécute une fonction (`inc_notes_dist()` du fichier ecrire/inc/notes.php qui stocke temporairement les notes en mémoire. Dès que la balise `#NOTES` est appelée, ces notes sont retournées et vidées de la mémoire.

Imaginons un texte dans le « chapo » et le « texte » d'un article comme cela :

```
// chapo :
Dans le chapo, une note [[Note A]] et une autre [[Note B]]
// texte :
Dans le texte, une note [[Note C]] et une autre [[Note D]]
```

Lors de l'affichage dans un squelette, les deux syntaxes ci-dessous produiront des contenus différents. La première affichera toutes les notes numérotées de 1 à 4 après le contenu du texte :

```
<BOUCLE_art(ARTICLES){id_article}>
```

```
#CHAPO
#TEXTE
#NOTES
</BOUCLE_art>
```

Dans cette seconde écriture, les notes du « chapo » sont d'abord affichées (numérotées de 1 à 2) après le contenu de #CHAPO, puis les notes du texte (numérotées aussi de 1 à 2), après le contenu de #TEXTE :

```
<BOUCLE_art(ARTICLES){id_article}>
#CHAPO
#NOTES
#TEXTE
#NOTES
</BOUCLE_art>
```

Exemple

L'appel des notes se fait souvent après l'affichage de tous les autres champs d'un article, cela pour prendre en compte toutes les notes calculées. L'affichage est simple :

```
[<div
class="notes"><h2><:info_notes:></h2>(#NOTES)</div>]
```

#REM

#REM permet de commenter du code dans les squelettes.

```
[(#REM) Ceci n'est pas une pipe ! Mais un commentaire ]
```

Note : Le code contenu dans la balise est tout de même interprété par SPIP, mais rien n'est affiché. Un filtre qui se trouve dedans sera effectivement appelé (ce qui n'est pas forcément ce que l'on souhaite) :

```
[(#REM|filtre)]
[(#REM) [(#BALISE|filtre)] ]
```

#SELF

`#SELF` retourne l'URL de la page en cours.

Cette URL ne peut être calculée correctement dans une inclusion que si le paramètre `self` ou `env` lui est transmis afin de créer un cache différent pour chaque URL.

```
<INCLURE{fond=xx}{env} />
```

#SESSION

`#SESSION{parametre}` affiche des informations sur le visiteur connecté. Une session peut être considérée comme des informations individuelles, conservées sur le serveur le temps de la visite du visiteur. Ainsi, ces informations peuvent être retrouvées et réutilisées lorsque celui-ci change de page.

La présence de cette balise, comme pour la balise `#AUTORISER`, génère un cache différent par visiteur authentifié sur le site, et un cache pour les visiteurs non authentifiés.

Exemple

Afficher le nom du visiteur s'il est connu :

```
#SESSION{nom}
```

Afficher une information si le visiteur est authentifié sur le site, c'est à dire qu'il possède un `id_auteur` :

```
[(#SESSION{id_auteur}|oui) Vous êtes authentifié ]
```

#SESSION_SET

La balise `#SESSION_SET{parametre, valeur}` permet de définir des variables de session pour un visiteur, qui pourront être récupérées par `#SESSION{parametre}`.

Exemple

Définir un parfum de vanille !

```
#SESSION_SET{parfum,vanille}
#SESSION{parfum}
```

#SET

`#SET{variable,valeur}` permet de stocker des valeurs localement, au sein d'un squelette. Elles sont récupérables, dans le même squelette, avec `#GET{variable}`. Voir aussi #GET (p.33).

Exemple

Stocker une couleur présente dans l'environnement, sinon une couleur par défaut :

```
#SET{claire,##ENV{couleur_claire,edf3fe}}
#SET{foncee,##ENV{couleur_foncee,3874b0}}
<style class="text/css">
#contenu h3 {
    color:[(#GET{claire})];
}
</style>
```

#VAL

`#VAL{valeur}` permet de renvoyer la valeur qu'on lui donne, tout simplement. Cette balise sert principalement pour envoyer un premier argument à des filtres existants.

```
#VAL{Ce texte sera retourne}
```

Exemple

Retourner un caractère avec la fonction PHP `chr` :

```
[(#VAL{91}|chr)]   // [
[(#VAL{93}|chr)]   // ]
```

Parfois le compilateur de SPIP se mélange les pinceaux entre les crochets que l'on souhaite écrire, et les crochets d'ouverture ou de fermeture des balises. Un exemple fréquent est l'envoi d'un paramètre tableau dans un formulaire (`name="champ[]"`), lorsque le champ est inclus dans une balise :

```
// probleme : le ] de champ[] est confondu
// avec la fermeture de la balise #ENV
[(#ENV{afficher}|oui)
<input type="hidden" name="champ[]" value="valeur" />
]
// pas de probleme ici
[(#ENV{afficher}|oui)
<input type="hidden"
name="champ[(#VAL{91}|chr)][(#VAL{93}|chr)]"
value="valeur" />
]
```

Critères de boucles

Les critères de boucles permettent de réaliser des sélections de données parfois complexes.

Syntaxe des critères

Les critères de boucles s'écrivent entre accolades après le nom des tables d'une boucle.

```
<BOUCLE_nom(TABLE){critere1}{critere2}...{critere n}>
```

Tout champ SQL dans la table peut devenir un critère de sélection, séparé par un opérateur. Mais d'autres critères peuvent être créés au besoin. Ils sont définis dans le fichier ecrire/public/criteres.php

Des balises peuvent aussi être utilisées comme paramètres des critères, mais il n'est pas possible d'utiliser leurs parties optionnelles. Principalement, l'usage des crochets est impossible :

```
<BOUCLE_nom(TABLE){id_table=#BALISE}> OK
<BOUCLE_nom(TABLE){id_table=(#BALISE|filtre)}> OK
<BOUCLE_nom(TABLE){id_table=[(#BALISE)]}> Echec
```

Exemple

Cette boucle ARTICLES dispose de 2 critères. Le premier extrait les articles dont le champ SQL « id_rubrique » de la table SQL « spip_articles » vaut 8. Le second indique de trier les résultats par titre.

```
<BOUCLE_arts(ARTICLES){id_rubrique=8}{par titre}>
  - #TITRE<br />
</BOUCLE_arts>
```

Critères raccourcis

Un critère peut avoir une écriture simplifiée `{critere}`. Dans ce cas là, SPIP traduit la plupart du temps par `{critere=#CRITERE}` (sauf si une fonction spéciale a été définie pour le critère en question dans ecrire/public/criteres.php).

```
<BOUCLEx(TABLES){critere}>...
```

Exemple

```
<BOUCLE_art(ARTICLES){id_article}>...
```

Ainsi `{id_article}` effectue une sélection `{id_article=#ID_ARTICLE}`. Comme toute balise SPIP, `#ID_ARTICLE` est récupéré dans les boucles les plus proches s'il existe, sinon dans l'environnement `#ENV{id_article}`.

Critères optionnels

Parfois il est utile de faire une sélection uniquement si l'environnement contient la balise demandée. Par exemple, on peut souhaiter filtrer des boucles en fonction d'une recherche particulière uniquement si une recherche est effectuée, sinon tout afficher. Dans ce cas, un point d'interrogation permet cela :

```
<BOUCLEx(TABLES){critere?}>...
```

Exemple

Afficher soit tous les articles du site (si aucune variable `id_article`, `id_rubrique` ou `recherche` n'existe), soit une sélection en fonction des critères présents. Ainsi, si l'on appelle le squelette avec les paramètres `id_rubrique=8` et `recherche=extra`, la boucle sélectionnera simplement les articles répondant à l'ensemble de ces critères.

```
<BOUCLE_art(ARTICLES)
{id_article?}{id_rubrique?}{recherche?}>
- #TITRE<br />
</BOUCLE_art>
```

Opérateurs simples

Tous les critères effectuant des sélections sur des champs SQL disposent d'un certain nombre d'opérateurs.

```
{champ opérateur valeur}
```

Voici une liste d'opérateurs simples :

- `=` : opérateur d'égalité `{id_rubrique=8}` sélectionne les entrées « id_rubrique » égales à 8.
- `>` : opérateur de supériorité stricte. `{id_rubrique>8}` sélectionne les entrées « id_rubrique » strictement supérieures à 8.
- `>=` : opérateur de supériorité. `{id_rubrique>=8}` sélectionne les entrées « id_rubrique » supérieures ou égales à 8.
- `<` : opérateur d'infériorité stricte. `{id_rubrique<8}` sélectionne les entrées « id_rubrique » strictement inférieures à 8.
- `<=` : opérateur d'infériorité. `{id_rubrique<=8}` sélectionne les entrées « id_rubrique » inférieures ou égales à 8.
- `!=` : opérateur de non égalité `{id_rubrique!=8}` sélectionne les entrées « id_rubrique » différentes de 8.

L'opérateur IN

D'autres opérateurs permettent des sélections plus précises. L'opérateur `IN` sélectionne une liste d'éléments. La liste peut être donnée soit par une chaîne séparée par des virgules, soit par un tableau (au sens PHP) retourné par une balise ou un filtre de balise.

```
<BOUCLEx(TABLES){champ IN a,b,c}>
<BOUCLEx(TABLES){champ IN #ARRAY{0,a,1,b,2,c}}>
<BOUCLEx(TABLES){champ IN (#VAL{a:b:c}|explode{:})}>
```

L'opérateur inverse, `!IN` sélectionne les entrées non listées après l'opérateur.

```
<BOUCLEx(TABLES){champ !IN a,b,c}>
```

Exemple

Sélectionner les images liées à un article :

```
<BOUCLE_documents(DOCUMENTS){id_article}{extension IN
png,jpg,gif}>
  - #FICHIER<br />
</BOUCLE_documents>
```

Sélectionner les rubriques, sauf certaines :

```
<BOUCLE_rubriques(RUBRIQUES){id_rubrique !IN 3,4,5}>
  - #TITRE<br />
</BOUCLE_rubriques>
```

L'opérateur ==

L'opérateur ==(ou sa négation !== permettent de sélectionner des contenus à partir d'expressions régulières. Ils permettent donc des sélections pouvant être extrêmement précises, mais pouvant aussi être gourmandes en énergie et temps pour le gestionnaire de base de données.

```
<BOUCLEx(TABLES){champ == expression}>
<BOUCLEx(TABLES){champ !== expression}>
```

Exemple

Sélection des titres commençant par « Les » ou « les » :

```
<BOUCLE_arts(ARTICLES){titre == ^[Ll]es}>
 - #TITRE<br />
</BOUCLE_arts>
```

Sélection des textes ne contenant pas le mot « carnaval » :

```
<BOUCLE_arts(ARTICLES){texte !== 'carnaval'}>
 - #TITRE<br />
</BOUCLE_arts>
```

Sélection des textes contenant « carnaval » suivi, à quelques caractères près (entre 0 et 20), de « Venise ».

```
<BOUCLE_arts(ARTICLES){texte == 'carnaval.{0,20}Venise'}>
 - #TITRE<br />
</BOUCLE_arts>
```

L'Opérateur « ! »

Les critères conditionnels de négation simple, effectués sur des champs extérieurs à la table (des champs créant une jointure sur une autre table) ne font pas toujours ce que l'on suppose au premier abord.

Ainsi le critère `{titre_mot!=rose}` sélectionne, sur une boucle ARTICLES tous les articles qui ne sont pas liés au mot clé « rose ». Mais le type de jointure créé fait qu'il sélectionne tous les articles ayant au moins un mot clé, donc au moins un mot clé, qui n'est pas « rose ».

Or, bien souvent, on cherche simplement à afficher tous les articles n'ayant pas le mot « rose », même ceux qui n'ont aucun mot clé. C'est cela qu'effectue l'opérateur `{!critere}`, qui permet de créer une seconde requête de sélection qui sera utilisée comme critère de sélection de la première :

```
<BOUCLE_articles(ARTICLES){!titre_mot = 'X'}> ...
```

Dans ce cas précis, les articles ayant un mot clé X sont sélectionnés, puis enlevés de la sélection SQL principale par un `NOT IN (requête de sélection)`.

Cette écriture est aussi valable lorsqu'on force un champ de jointure, ainsi on pourrait tout aussi bien écrire :

```
<BOUCLE_articles(ARTICLES){!mots.titre = 'X'}> ...
```

Exemple

Sélectionner les rubriques qui n'ont aucun article dont le titre commence par un « L » ou un « l ». Attention tout de même, cette requête utilisant une expression régulière (`^[Ll]`) nécessite plus de calculs pour le gestionnaire de bases de données.

```
<BOUCLE_rub(RUBRIQUES){!articles.titre == '^[Ll]'}> ...
```

Filtres de balises

Les filtres permettent de modifier le résultat des balises.

Syntaxe des filtres

Les filtres s'appliquent sur les balises en utilisant le caractère « | » (pipe). En pratique, ils correspondent à l'appel d'une fonction PHP existante ou déclarée dans SPIP.

```
[(#BALISE|filtre)]
[(#BALISE|filtre{argument2, argument3, ...})]
```

Lorsqu'un filtre « x » est demandé, SPIP cherche une fonction nommée « filtre_x ». Si elle n'existe pas, il cherche « filtre_x_dist », puis « x ». Il exécute alors la fonction qu'il a trouvée avec les arguments transmis. Il est important de comprendre que le premier argument transmis au filtre (à la fonction PHP donc) est le résultat de l'élément à gauche du filtre.

Exemple

Insérer un élément `title` sur un lien. Pour cela, on utilise les filtres `|couper`, qui permet de couper un texte à la taille voulue, et `|attribut_html`, qui permet d'échapper les apostrophes qui pourraient gêner le code HTML généré (exemple : `title='à tire d'ailes'` poserait problème à cause de cette apostrophe.).

Le filtre `|couper` s'applique sur le résultat de la balise `#TITRE`, le filtre `|attribut_html` sur le résultat du filtre `|couper`. On peut donc chaîner les filtres.

```
<a href="#URL_ARTICLE"
title="[(#TITRE|couper{80}|attribut_html)]">Article
suivant</a>
```

Filtres issus de classes PHP

Une écriture peu connue permet aussi d'exécuter des méthodes d'une classe PHP. Si l'on demande un filtre « x ::y », SPIP cherchera une classe PHP « filtre_x » possédant une fonction « y » exécutable. S'il ne trouve pas, il cherchera une classe « filtre_x_dist » puis enfin une classe « x ».

```
[(#BALISE|class::methode)]
```

Exemple

Imaginons une classe PHP définie comme ci-dessous. Elle contient une fonction (récursive par ailleurs) qui permet de calculer une factorielle (`x! = x*(x-1)*(x-2)*...*3*2*1`).

```
class Math{
  function factorielle($n){
    if ($n==0)
      return 1;
    else
      return $n * Math::factorielle($n-1);
    }
}
```

Elle peut être appelée comme ceci :

```
[(#VAL{9}|Math::factorielle)]
// renvoie 362880
```

Filtres de comparaison

Comme sur les critères de balise, des filtres de comparaison sont présents. Ils s'utilisent de la sorte :

```
[(#BALISE|operateur{valeur})]
```

Voici une liste d'opérateurs :

- `==` (vérifie une égalité)
- `!=`
- `>`

- `>=`
- `<`
- `<=`

Exemple

```
[(#TITRE|=={Dégustation}|oui)
    Ceci parle de dégustation !
]
[(#TEXTE|strlen|>{200}|oui)
    Ce texte a plus de 200 caractères !
]
```

Le filtre `|oui` permet de cacher le résultat du test. En son absence, `[(#TITRE|==Dégustation}) ici ]` afficherait, si le test était vrai « 1 ici » (1 signifiant dans ce cas "vrai", ou *true* en PHP)

Filtres de recherche et de remplacement

D'autres filtres permettent d'effectuer des comparaisons ou des recherches d'éléments. C'est le cas des filtres « |match » et « |replace »

- `match` permet de tester si l'argument reçu vérifie une expression régulière transmise en second argument du filtre.
- `replace` permet de remplacer du texte, en suivant aussi une expression régulière.

```
[(#BALISE|match{texte})]
[(#BALISE|replace{texte,autre texte})]
```

Exemple

```
// affiche "texte oui"
[(#VAL{Ce texte est joli}|match{texte}) oui ]
// affiche "oui"
[(#VAL{Ce texte est joli}|match{texte}|oui) oui ]
// n'affiche rien
```

```
[(#VAL{Cet écureuil est joli}|match{texte}) non ]
// affiche "oui"
[(#VAL{Cet écureuil est joli}|match{texte}|non) oui ]

// affiche "Ce chat est joli"
[(#VAL{Ce texte est joli}|replace{texte,chat})]
```

Les filtres de test

D'autres filtres de test et de logique existent. On trouvera les filtres « ? », « sinon », « oui », « non », « et », « ou », « xou » qui permettent permettent de répondre à la plupart des besoins.

- `|?{vrai,faux}` retourne "faux" si ce qui entre dans le filtre est vide ou nul, sinon "vrai".
- `|sinon{ce texte}` retourne "ce texte" seulement si ce qui entre dans le filtre est vide, sinon, retourne simplement l'entrée.
- `|oui` retourne un espace ou rien. C'est équivalent à `|?{' ',''}` ou `|?{' '}` et permet de retourner un contenu non vide (un espace) pour signaler que les parties optionnelles des balises doivent s'afficher.
- `|non` est l'inverse de `|oui` et est équivalent à `|?{'',' '}`
- `|et` permet de vérifier la présence de 2 éléments
- `|ou` vérifie la présence d'un des deux éléments
- `|xou` vérifie la présence d'un seul de deux éléments.

Par ailleurs, SPIP comprendra les équivalent anglais « yes », « not », « or », « and » et « xor »

Exemple

```
// affiche le chapeau s'il existe, sinon le début du
texte
[(#CHAPO|sinon{#TEXTE|couper{200}})]
 // affiche "Ce titre est long" seulement si le titre
fait plus de 30 caracteres
[(#TITRE|strlen|>{30}|oui) Ce titre est long ]

[(#CHAPO|non) Il n'y a pas de chapo ]
```

```
[(#CHAPO|et{#TEXTE}) Il y a un chapo, et un texte ]
[(#CHAPO|et{#TEXTE}|non) Il n'y a pas les deux ensemble ]
[(#CHAPO|ou{#TEXTE}) Il y a soit un chapo, soit un texte,
soit les deux ]
[(#CHAPO|ou{#TEXTE}|non) Il y a ni chapo, ni texte ]
[(#CHAPO|xou{#TEXTE}) Il y a soit un chapo, soit un texte
(mais pas les deux, ni aucun) ]
[(#CHAPO|xou{#TEXTE}|non) Il y a soit rien, soit tout,
mais pas l'un des deux ]
```

Inclusions

Pour faciliter la maintenance des squelettes générés, il est important de mutualiser les codes identiques. Cela se réalise grâce aux inclusions.

Inclure des squelettes

Créer des inclusions, c'est à dire des morceaux de codes précis, permet de mieux gérer la maintenance de ses squelettes. Dans la pratique, certaines parties d'une page HTML de votre site vont être identiques quel que soit le type de page. C'est souvent le cas de l'affichage d'un portfolio, d'un menu de navigation, de la liste des mots clés attachés à une rubrique ou un article, etc.

Tout squelette SPIP existant peut être inclus dans un autre par la syntaxe suivante :

```
<INCLURE{fond=nom_du_fichier}{parametres transmis} />
```

Transmettre des paramètres

Vous pouvez transmettre des paramètres aux inclusions. Par défaut, rien n'est transmis à une inclusion hormis la date du calcul. Pour passer des paramètres au contexte de compilation du squelette, il faut explicitement les déclarer lors de l'appel à l'inclusion :

```
<INCLURE{fond=squelette}{param} />
<INCLURE{fond=squelette}{param=valeur} />
```

Le premier exemple avec `{param}` seul récupère la valeur de `#PARAM` et la transmet au contexte de compilation dans la variable `param`. Le second exemple attribue une valeur spécifique à la variable `param`. Dans les deux cas, dans le squelette appelé, nous pourrons récupérer `#ENV{param}`.

Transmettre tout le contexte en cours

Le paramètre `{env}` permet de transmettre le contexte de compilation du squelette en cours à celui inclus.

Exemple

```
// fichier A.html
<INCLURE{fond=B}{type}{mot=triton} />
// fichier B.html
<INCLURE{fond=C}{env}{couleur=rouge} />
// fichier C.html
Type : #ENV{type} <br />
Mot : #ENV{mot} <br />
Couleur : #ENV{couleur}
```

Si l'on appelle la page `spip.php?page=A&type=animal`, celle-ci transmet les paramètres `type` et `mot` au squelette `B.html`. Celui-ci transmet tout ce qu'il reçoit et ajoute un paramètre `couleur` en appelant le squelette `C.html`.

Dans le squelette `C.html`, on peut alors récupérer tous les paramètres transmis.

Ajax

SPIP permet de recharger simplement des éléments de page en AJAX.

Paginations AJAX

Les inclusions qui possèdent le critère `{ajax}` permettent de recharger dans la page seulement la partie incluse. La plupart du temps, il faudra aussi inclure le critère `{env}` dès lors qu'il y a une pagination dans l'inclusion.

```
<INCLURE{fond=inclure/fichier}{env}{ajax} />
```

Lorsque l'on couple ce critère d'inclusion avec la balise `#PAGINATION`, les liens de pagination deviennent alors automatiquement AJAX. Plus précisément, tous les liens du squelette inclus contenus dans une classe CSS `pagination`.

```
<p class="pagination">#PAGINATION</p>
```

Exemple

Lister les derniers articles. Cette inclusion liste les derniers articles par groupe de 5 et affiche un bloc de pagination.

```
<INCLURE{fond=modeles/liste_derniers_articles}{env}{ajax}
/>
```

Fichier `modeles/liste_derniers_articles.html` :

```
<B_art>
  #ANCRE_PAGINATION
  <ul>
    <BOUCLE_art(ARTICLES){!par date}{pagination 5}>
      <li><a href="#URL_ARTICLE">#TITRE</a></li>
    </BOUCLE_art>
  </ul>
  <p class="pagination">#PAGINATION</p>
</B_art>
```

Résultat : Une pagination ajax, de 5 en 5...

```
<a id="pagination_art" name="pagination_art"/>
<ul>
    <li><a href="Recursivite,246"
title="art246">Récursivité</a></li>
    <li><a href="Parametre"
title="art245">Paramètre</a></li>
    ...
</ul>
<p class="pagination">
    <strong class="on">0</strong>
    <span class="separateur">|</span>
    <a rel="nofollow" class="lien_pagination noajax"
href="Pagination-AJAX?debut_art=5#pagination_art">5</a>
    <span class="separateur">|</span>
    <a rel="nofollow" class="lien_pagination noajax"
href="Pagination-
AJAX?debut_art=10#pagination_art">10</a>
    <span class="separateur">|</span>
    ...
    <a rel="nofollow" class="lien_pagination noajax"
href="Pagination-
AJAX?debut_art=205#pagination_art">...</a>
</p>
```

Liens AJAX

Outre les inclusions contenant une pagination, il est possible de spécifier des liens à recharger en AJAX en ajoutant dessus la classe CSS `ajax`.

```
<a class="ajax"
href="[(#URL_ARTICLE|parametre_url{tous,oui})]">Tout
afficher</a>
```

Exemple

```
<INCLURE{fond=modeles/liste_articles}{env}{ajax} />
```

Fichier `modeles/liste_articles.html` : Afficher ou cacher l'introduction des articles :

```
<ul>
<BOUCLE_art(ARTICLES){!par date}{0,5}>
    <li>#TITRE
        [(#ENV{afficher_introduction}|=={oui}|oui)
            <div>#INTRODUCTION</div>
        ]
    </li>
</BOUCLE_art>
</ul>
[(#ENV{afficher_introduction}|=={oui}|oui)
    <a class="ajax"
href="[(#SELF|parametre_url{afficher_introduction,''})]">
    Cacher les introductions</a>
]
[(#ENV{afficher_introduction}|=={oui}|non)
    <a class="ajax"
href="[(#SELF|parametre_url{afficher_introduction,oui})]">
    Afficher les introductions</a>
]
```

Éléments linguistiques

La gestion et la création d'espaces multilingues est toujours une chose délicate à gérer. Nous allons voir dans cette partie comment gérer des éléments d'interface multilingue.

SPIP dispose pour gérer les textes des interfaces (à distinguer des contenus éditoriaux donc) de deux éléments : les chaînes de langues appelées idiomes et la balise multilangue appelée polyglotte.

Syntaxe des chaînes de langue

Les chaînes de langue, nommées « idiomes » dans SPIP, sont des codes dont les traductions existent dans des fichiers stockés dans les répertoires `lang/` de SPIP, des plugins ou des dossiers squelettes.

Pour appeler une chaîne de langue, il faut simplement connaître son code :

```
<:bouton_ajouter:>
<:navigation:>
```

La syntaxe générale est celle-ci :

```
<:cle:>
<:prefix:cle:>
```

Fichiers de langues

Les fichiers de langue sont stockés dans les répertoires `lang/`. Ce sont des fichiers PHP nommés par un préfix et un code de langue : `prefixe_xx.php`.

Contenu des fichiers

Ces fichiers PHP déclarent un tableau associatif. À chaque clé correspond une valeur. Tous les codes problématiques sont échappés (accents), et certaines langues ont des valeurs écrites en signes hexadécimaux (cas du japonais, de l'hébreu...).

```
<?php
$GLOBALS[$GLOBALS['idx_lang']] = array(
```

```
    'cle' => 'valeur',
    'cle2' => 'valeur2',
    // ...
);
```

Exemple

Voici un extrait du fichier de langue du squelette du site Programmer (`documentation_fr.php`) :

```
<?php
$GLOBALS[$GLOBALS['idx_lang']] = array(
    //C
    'choisir'=>'Choisir...',
    'conception_graphique_par'=>'Th&egrave;me graphique
adapt&eacute; de ',
    //E
    'en_savoir_plus' => 'En savoir plus !',
    //...
);
```

Utiliser les codes de langue

Tout item de langue peut être appelé de la sorte dans un squelette SPIP :

```
<:prefix:code:>
```

Chercher un code dans plusieurs fichiers

Il est possible de chercher un code dans plusieurs fichiers. Par défaut, si le préfixe n'est pas renseigné, SPIP cherche dans les fichiers `local_xx.php`, puis `spip_xx.php`, puis `ecrire_xx.php`. S'il ne trouve pas le code dans la langue demandé, il cherche dans la langue française. S'il ne trouve toujours pas, il affiche le code langue (en remplaçant les soulignés par des espaces).

On peut indiquer de chercher dans plusieurs fichiers avec cette syntaxe :

```
<:prefixe1/prefixe2/.../prefixeN:choisir:>
```

Surcharger un fichier de langue

Pour surcharger des items de langue présents dans un fichier de langue de SPIP, par exemple, `ecrire/lang/spip_xx.php` ou dans un fichier de langue de plugin, `lang/prefixe_xx.php`, il suffit de créer un fichier `squelettes/local_xx.php` et d'y insérer les items modifiés ou nouveaux.

Exemple

Choisir la bonne documentation !

```
<:documentation:choisir:>
```

Si `bouton_ajouter` n'est pas trouvé dans le fichier de langue « documentation », le chercher dans celui de « spip », sinon de « ecrire » :

```
<:documentation/spip/ecrire:bouton_ajouter:>
```

Syntaxe complète des codes de langue

La syntaxe complète est la suivante :

```
<:prefixe:code{param=valeur}|filtre{params}:>
```

Paramètres

Les codes de langue peuvent recevoir des paramètres qui seront insérés dans les valeurs au moment de la traduction. Les paramètres sont alors écrits dans les fichiers de langue entre signe arobase (`@`).

Un code de langue pourrait donc être :

```
'creer_fichier'=>'Cr&eacute;er le fichier @fichier@ ?',
```

Appel des paramètres

On appelle ce paramètre comme indiqué :

```
<:documentation:creer_fichier{fichier=tete_de_linote.txt}:>
```

Filtrer les codes de langue

L'intérêt est assez faible, mais il est possible de passer les codes de langue dans les filtres exactement comme les balises de SPIP, par exemple :

```
<:documentation:longue_description|couper{80}:>
```

Codes de langue en PHP

Une fonction existe en PHP pour récupérer les traductions des codes de langue : `_T`.

Elle s'utilise très simplement comme ceci :

```
_T('code');
_T('prefixe:code');
_T('prefixe1/.../prefixeN:code');
_T('prefixe:code', array('param'=>'valeur'));
```

Chaînes en développement

Vous trouverez enfin parfois la fonction `_L`, qui signifie : « Chaîne à mettre en code de langue quand le développement sera fini ! ». En gros, pendant les phases de développement de SPIP ou de plugins, les chaînes de langues évoluent souvent. Pour éviter de mélanger les chaînes correctement traduites et les nouvelles qui vont évoluer, la fonction `_L` est utilisée.

```
_L('Ce texte devra &ecirc;tre traduit !');
```

Lorsque le développement est stabilisé, un parcours du code à la recherche des « _L » permet de remplacer alors les chaînes par des codes de langue appropriés (en utilisant alors la fonction `_T`).

Exemple

Le plugin « Tickets » possède un fichier de langue `lang/tickets_fr.php` contenant (entre autre) :

```
$GLOBALS[$GLOBALS['idx_lang']] = array(
```

```
    // ...
    'ticket_enregistre' => 'Ticket enregirtr&eacute;',
);
```

Lorsque l'on crée un nouveau ticket, le retour du formulaire indique que celui-ci a bien été enregistré en transmettant la chaîne de langue au paramètre `message_ok` du formulaire d'édition de tickets :

```
$message['message_ok'] = _T('tickets:ticket_enregistre');
// soit = "Ticket enregistr&eacute;" si on est en
francais.
```

Les Polyglottes (multi)

Une balise (au sens HTML cette fois) `<multi>`, comprise à la fois des squelettes et des contenus édités par les rédacteurs, permet de sélectionner un texte particulier en fonction de la langue demandée.

Elle s'utilise comme ceci :

```
<multi>[fr]en français[en]in english</multi>
```

Elle permette donc d'écrire à l'intérieur des squelettes des éléments multilingues facilement, sans passer par les codes et chaînes de langue.

Utilisation par les rédacteurs

Cette écriture est surtout utilisée par les rédacteurs (ou via un plugin de saisie plus adapté !) pour traduire un site lorsqu'il y a peu de langues (2 ou 3) à traduire. `<multi>` est donc plus utilisé du coté éditorial que pour l'écriture de squelettes.

Multilinguisme

SPIP est capable de gérer un site multilingue. On peut entendre deux choses par multilingue :

- avoir la langue de l'interface qui s'adapte au visiteur, par exemple pour afficher les dates ou pour le sens de lecture,
- avoir des contenus en plusieurs langues, et non uniquement l'interface, comme par exemple avoir une version du site en français et une autre en anglais, ou pouvoir traduire des articles déjà rédigés dans une langue vers une autre langue,
- ou pourquoi pas un mélange des deux (interface en arabe avec des textes en français...)

SPIP possède une syntaxe et différents outils pour gérer le multilinguisme.

Différents multilinguismes

Il y a de nombreuses possibilités pour développer un site multilingue sous SPIP, par exemple :

- créer un secteur (rubrique à la racine du site) par langue, avec des contenus autonomes,
- créer le site dans une langue principale et déclarer des traductions des articles dans les différentes langues souhaitées,
- ou encore définir la langue de chaque rubrique du site ou de chaque article...

Chaque solution a ses avantages et ses inconvénients et ce choix éditorial influencera quelque peu l'écriture des squelettes. Nous allons voir de quels outils disposent les squelettes pour les sites multilingues.

En savoir plus !

Un excellent dossier sur le multilinguisme a été réalisé par Alexandra Guiderdoni pour la SPIP Party de Clermont-Ferrand en 2007. Sa lecture sera bénéfique pour comprendre les subtilités et se poser les bonnes questions lors de la réalisation d'un site multilingue :
http://www.guiderdoni.net/SPIP-et-l...

La langue de l'environnement

SPIP transmet au premier squelette la langue demandée par le visiteur du site que l'on peut récupérer via `#ENV{lang}` dans un squelette. Par défaut, ce sera la langue principale du site, qu'il est possible de modifier avec le formulaire `#MENU_LANG` qui liste les langues prévues pour le multilinguisme de votre site.

Lorsqu'on qu'on utilise le formulaire `#MENU_LANG`, la langue sélectionnée est conservée dans un cookie et une redirection est effectuée sur la page en cours avec le paramètre d'URL `lang` défini sur la langue choisie. Le paramètre `lang` ainsi transmis va pouvoir être utilisé par SPIP. Il sera aussi possible d'utiliser ultérieurement le cookie pour forcer la langue d'affichage.

La langue peut par ailleurs être définie de façon précise lors de l'inclusion d'un squelette en utilisant le paramètre `lang` :

```
<INCLURE{fond=A}{lang=en} />
```

La langue de l'objet

Certains objets éditoriaux de SPIP, c'est le cas des rubriques et des articles, possèdent un champ de langue dans leur table SQL permettant d'indiquer en quelle langue ils sont rédigés (ou à quelle langue ils appartiennent).

On récupère la langue de la rubrique ou de l'article en cours par `#LANG` dans une boucle `RUBRIQUES` ou `ARTICLES`.

Lorsque la rubrique en cours n'a pas de langue précise affectée, c'est celle de sa rubrique parente qui est utilisée, sinon la langue principale du site.

Exemple

Affiche les articles et les langues des 2 premières rubriques du site :

```
Votre langue : #ENV{lang}
<B_rubs>
  <ul>
```

```
  <BOUCLE_rubs(RUBRIQUES){racine}{0,2}>
    <li>#TITRE : #LANG
      <B_arts>
        <ul>
        <BOUCLE_arts(ARTICLES){id_rubrique}>
          <li>#TITRE : #LANG</li>
        </BOUCLE_arts>
        </ul>
      </B_arts>
    </li>
  </BOUCLE_rubs>
  </ul>
</B_rubs>
```

Résultat :

```
Votre langue : fr
  <ul>
    <li>en : en
      <ul>
        <li>Notes about this documentation : en</li>
      </ul>
    </li>
    <li>fr : fr
      <ul>
        <li>Notes sur cette documentation : fr</li>
      </ul>
    </li>
  </ul>
```

Critères spécifiques

Des critères de boucles spécifiques permettent de récupérer les articles dans les langues souhaitées.

lang

Déjà, simplement le critère `{lang}` permet de sélectionner la langue du visiteur, ou la langue choisie :

```
// langue du visiteur
<BOUCLE_art(ARTICLES){lang}> ... </BOUCLE_art>
```

```
// langue anglaise (en)
<BOUCLE_art(ARTICLES){lang=en}> ... </BOUCLE_art>
```

traduction

Le critère {traduction} permet de lister les différentes traductions d'un article :

```
<BOUCLE_article(ARTICLES){id_article}>
  <ul>
    <BOUCLE_traductions(ARTICLES) {traduction}{par lang}>
      <li>[(#LANG|traduire_nom_langue)]</li>
    </BOUCLE_traductions>
  </ul>
</BOUCLE_article>
```

Ici, toutes les traductions d'un article seront affichées (y compris l'article en cours, que l'on peut enlever avec le critère {exclus}).

origine_traduction

Ce critère permet de retrouver l'article source d'un article traduit :

```
<BOUCLE_article(ARTICLES){id_article}>
  <BOUCLE_origine(ARTICLES) {origine_traduction}>
    #TITRE (#LANG)
  </BOUCLE_origine>
</BOUCLE_article>
```

Exemple

Afficher un article dans la langue du visiteur si possible, sinon dans la langue principale. On commence par lister dans une rubrique, les articles qui servent de source à la création des traductions. Ensuite, on cherche s'il existe une traduction dans la langue demandée par le visiteur. Selon la réponse on affiche le titre de l'article traduit ou de l'article source.

```
<BOUCLE_art1(ARTICLES){id_rubrique}{origine_traduction}>
    <BOUCLE_art2(ARTICLES){traduction}{lang=#ENV{lang}}>
        // si une traduction existe
        <li>#TITRE</li>
    </BOUCLE_art2>
```

```
        // sinon
        <li>#TITRE</li>
    </B_art2>
</BOUCLE_art1>
```

Forcer la langue selon le visiteur

Le paramètre `forcer_lang`

Le formulaire `#MENU_LANG` stocke la langue choisie dans un cookie. Ce cookie peut donc être employé pour réafficher le site dans la langue qu'il avait choisit. Une des manières d'y arriver est de définir la variable globale `forcer_lang` dans un fichier d'options.

```
$GLOBALS['forcer_lang'] = true;
```

Sa présence indique à SPIP de systématiquement rediriger la page demandée en ajoutant le paramètre d'URL `lang` avec la valeur du cookie de langue s'il existe, sinon la langue principale du site.

Cette globale `forcer_lang` a cependant aussi une autre action : elle indique en même temps que les chaines de langue de l'interface s'affichent dans la langue du visiteur, et non dans la langue des articles ou rubriques.

Autre utilisation du cookie

Une autre possibilité peut être d'utiliser la préférence de l'utilisateur, mais de ne pas forcément rediriger vers le paramètre d'URL `lang`, cela en utilisant la fonction `set_request` de SPIP pour ajouter un paramètre `lang` calculé que SPIP réutilisera ensuite lorsqu'il appellera la fonction `_request`.

Exemple

L'exemple ci-dessous, issu d'un fichier d'option, calcule la langue à utiliser. Ce calcul, ici se passe en deux temps :

- on analyse si l'URL est de la forme `http://nom.domaine/langue/reste_de_l_url`, où « langue » peut être un des codes

de langues définis du site (« fr », « en » ou « es » par exemple) et dans ce cas précis, on utilise la langue trouvée,
- sinon, la fonction `utiliser_langue_visiteur()` prend la langue du cookie, sinon la langue du navigateur.

Enfin, si la langue calculée est différente du cookie, le cookie est recréé.

```
// on ajoute la langue d'origine dans le contexte
systematiquement.
if (!$langue = _request('lang')) {
    include_spip('inc/lang');
    $langues = explode(',',
$GLOBALS['meta']['langues_multilingue']);
    // si la langue est definie dans l'url (en/ ou fr/)
on l'utilise
    if (preg_match(',^' .
$GLOBALS['meta']['adresse_site'] . '/(' .
join('|',$langues) . ')/,', 'http://' .
$_SERVER['HTTP_HOST'].$_SERVER['REQUEST_URI'], $r)) {
        $langue = $r[1];
        changer_langue($langue);
    } else {
        $langue = utiliser_langue_visiteur();
        if (!in_array($langue, $langues)) {
            //$langue = "en"; // pour ne pas s'embeter !
            $langue = $GLOBALS['meta']['langue_site'];
        }
    }
    // stocker dans $_GET
    set_request('lang', $langue);
}
// stocker la langue en cookie...
if ($langue != $_COOKIE['spip_lang']) {
    include_spip('inc/cookie');
    spip_setcookie('spip_lang', $langue);
}
```

Choix de la langue de navigation

Par défaut, lorsqu'on navigue sur un article anglais, les éléments de l'interface sont traduits en anglais.

En utilisant le formulaire de sélection `#MENU_LANG`, celui-ci change par défaut les éléments de l'interface et ceux des articles par la langue sélectionnée.

Sauf que si nous sommes déjà dans un article d'une certaine langue, par exemple anglais, donc avec l'interface en anglais et le menu de langue qui indique « English », et que l'on demande à afficher le français via le menu de langue, l'URL de la page ajoute un paramètre `lang=fr`, mais rien ne se passe d'autre, l'article reste en anglais et son interface aussi : en fait, c'est le contexte de l'article qui est alors prioritaire sur ce que demande le visiteur.

On peut vouloir à l'inverse, afficher l'interface en français, mais lire l'article anglais tout de même. Pour que l'interface soit indépendante de la langue de l'article/rubrique en cours, il faut définir la variable globale `forcer_lang` :

```
// forcer la langue du visiteur
$GLOBALS['forcer_lang']=true;
```

Forcer un changement de langue d'interface

Dernier point particulier de multilinguisme, on souhaite parfois avoir un mélange de langues entre l'interface et les contenus, mais en gardant une certaine cohérence. Précisément lorsqu'on souhaite afficher les articles dans la langue source si ils n'ont pas encore été traduits, sinon dans la langue de traduction. Dans ce cas là, on doit activer `forcer_lang`

Cependant, lorsque sur l'affichage d'un article, on liste les différentes traductions existantes, par exemple avec le modèle `modeles/articles_traductions.html` de SPIP, le lien généré ne changera pas la langue de l'interface, vu que `forcer_lang` conserve la langue du visiteur.

Si l'on désire que le fait de cliquer un lien de traduction implique un changement de langue d'interface (dans la même langue que la traduction appelée), il faut éditer le modèle `articles_traductions.html` ou en créer un nouveau. On utilise alors l'action « converser » permettant de générer un lien particulier qui redirigera sur l'article voulu dans la langue d'interface voulue de la sorte :

```
[(#VAL{converser}
    |generer_url_action{[redirect=(#URL_ARTICLE
```

```
        |parametre_url{var_lang,#LANG})]})]
```

Exemple de modèle complet (et complexe !) :
Ceci est un modèle qui liste les différentes traductions d'un article. Si ce n'est pas la traduction en cours de lecture, un lien est proposé qui indique la langue de traduction.

```
<BOUCLE_article(ARTICLES){id_article}>
<BOUCLE_traductions(ARTICLES) {traduction} {par lang} {','}>[
    (#TOTAL_BOUCLE|>{1}|?{' '})
    <span lang="#LANG" xml:lang="#LANG" dir="#LANG_DIR"[
class="(#EXPOSE)"]>
        [(#EXPOSE{'',<a href="[(#VAL{converser}
            |generer_url_action{[redirect=(#URL_ARTICLE
                |parametre_url{var_lang,#LANG})]})]"
rel="alternate" hreflang="#LANG"[
title="(#TITRE|attribut_html|couper{80})"]>})]
        [(#LANG|traduire_nom_langue)]
        #EXPOSE{'',</a>}
    </span>
]</BOUCLE_traductions>
</BOUCLE_article>
```

Liaisons entre tables (jointures)

Une jointure, en langage SQL est ce qui permet d'obtenir des informations de plusieurs tables réunies en une seule requête. Il est possible de réaliser quelques jointures avec le langage de boucle de SPIP.

Jointures automatiques

Lorsque dans une boucle il est demandé un critère qui n'appartient pas à la table de la boucle, SPIP essaie automatiquement de trouver une table liée qui contient le champ demandé.

SPIP a deux manières de trouver les tables liées : soit les liaisons sont explicitement déclarées, soit elles sont calculées.

Exemple

Récupérer les documents qui sont insérés dans les textes des articles ou autre objet éditorial (par un modèle `<docXX>` par exemple), et non simplement liés à cet objet. Le champ `vu` appartient à la table `spip_documents_liens`. Une jointure se créé donc pour obtenir le résultat souhaité.

```
<BOUCLE_doc(DOCUMENTS){0,10}{vu=oui}>
- #FICHIER<br />
</BOUCLE_doc>
```

Déclarations de jointures

Les liaisons entre tables sont déclarées dans SPIP dans le fichier ecrire/public/interfaces.php. D'autres déclarations peuvent être ajoutées avec le pipeline « declarer_tables_interfaces ».

Cette déclaration peut-être :

```
// proposer une jointure entre les rubriques et les documents
$tables_jointures['spip_rubriques'][]= 'documents_liens';
```

```
// proposer une jointure entre articles et auteurs, en
specifiant le champ de la jointure
$tables_jointures['spip_articles']['id_auteur']=
'auteurs_articles';
```

Cela indique des liaisons possibles entre tables. Lorsque 2 tables peuvent avoir plusieurs champs qui peuvent se lier, on peut indiquer précisément le champ de la liaison.

Exceptions

Il est même possible de créer des jointures lors d'appel à des champs inexistants, par exemple l'appel au critère `{titre_mot=yy}` peut conduire à une jointure sur la table « spip_mots » alors même que le champ SQL « titre_mot » n'existe pas dans la table de cette manière :

```
$exceptions_des_jointures['titre_mot'] = array('spip_mots',
'titre');
```

Automatisme des jointures

Lorsqu'elles ne sont pas explicitement déclarées à SPIP, les jointures sont calculées. Pour cela, SPIP compare entre eux les noms des champs des différentes tables.

Lorsqu'une boucle, par exemple `AUTEURS` cherche un critère absent de sa table, par exemple `{prenom=Daniel}`, SPIP va regarder dans les autres tables qu'il connaît et qui ont des champs homonymes à la table auteur (par exemple la clé `id_auteur`) si elles possèdent le champ « prenom » demandé. Si l'une d'elles le possède, une jointure sera réalisée entre ces deux tables.

Par exemple, si une table `AUTEURS_ELARGIS` existe (plugin « Inscription 2 ») avec les champs « id_auteur » et « prenom », une jointure serait réalisée.

objet, id_objet

SPIP 2.0 introduit une nouvelle recherche de jointure. Les clés primaires d'une table, dans ce cas « id_auteur » de la table `spip_auteurs`, en plus d'être cherchées dans des champs homonymes sur d'autres tables, sont aussi cherchées dans les tables possédant le couple de champ « objet » et « id_objet », tel que, ici, « objet=auteur ». C'est par exemple le cas de la table `spip_documents_liens`.

Forcer des jointures

La détection automatique par SPIP a parfois des limites et deux syntaxes permettent de forcer des tables à joindre, ou des critères de tables à utiliser.

```
// forcer une table
<BOUCLE_table(TABLE1 table2 tablen){...}>
// forcer un champ d'une table
<BOUCLE_table(TABLE){table.champ}>
```

Exemple

Ces deux boucles sélectionnent les articles dont un auteur possède un nom contenant « lie » (comme « Emilie »).

```
<BOUCLE_art(ARTICLES auteurs_articles
auteurs){nom==lie}{0,5}>
- #TITRE / #NOM<br />
</BOUCLE_art>
<hr />
<BOUCLE_art2(ARTICLES){auteurs.nom==lie}{0,5}>
- #TITRE / #NOM<br />
</BOUCLE_art2>
```

Cependant, une différence de taille existe : actuellement, seule l'écriture déclarant l'ensemble des tables permet de faire afficher une balise `#CHAMP` d'une autre table. Ainsi, `#NOM` ne sera renseigné que dans la première boucle.

Accéder à plusieurs bases de données

SPIP permet de lire très facilement des bases de données existantes, au format MySQL, PostGres ou SQLite, et de présenter leur contenu dans des squelettes.

Déclarer une autre base

Pour accéder à une autre base de données, il faut que SPIP dispose des codes d'accès à la base en question. Actuellement, les bases secondaires déclarées sont correctement gérées en lecture. L'écriture par contre dans ces bases externes n'est pas encore correctement prise en compte en SPIP 2.0.

Pour déclarer une autre base de données, deux solutions :

- utiliser l'interface graphique prévue pour (Configuration > maintenance du site > Déclarer une base)
- écrire selon la syntaxe prévue un fichier de connexion dans le répertoire `config/` (ou le répertoire défini par la constante `_DIR_CONNECT`).

Fichier de connexion `config/xx.php`

Pour un fichier de connexion `tarabiscote.php`, son contenu sera :

```
<?php
if (!defined("_ECRIRE_INC_VERSION")) return;
define('_MYSQL_SET_SQL_MODE',true);
$GLOBALS['spip_connect_version'] = 0.7;
spip_connect_db('localhost','','lutilisateur','le passe
world','tarabiscote','mysql', 'spip','');
?>
```

On appelle donc une fonction `spip_connect_db()` avec pour arguments, dans l'ordre :

1. adresse du serveur sql
2. numéro de port pour la connexion si nécessaire
3. nom d'utilisateur
4. mot de passe
5. nom de la base de données
6. type de serveur (mysql, pg, sqlite2, sqlite3...)
7. préfixe des tables
8. connexion des utilisateurs par ldap ?

Accéder à une base déclarée

Chaque base supplémentaire ainsi déclarée peut-être appelée via les boucles SPIP de cette manière :

```
<BOUCLE_externe(nom:TABLE)>
```

Le paramètre **nom** correspond au nom du fichier de connexion.

Exemple

J'ai testé WordPress il y a quelques temps et j'ai donc une base fonctionnelle. En créant un fichier de connexion `wordpress.php` il m'est possible de récupérer grâce à cela, les 5 dernières publications comme ceci :

```
<BOUCLE_articles(wordpress:WP_POSTS){0,5}{!par
post_date}{post_status=publish}>
    <h2>#POST_TITLE</h2>
    <div class="texte">#POST_CONTENT</div>
</BOUCLE_articles>
```

Le paramètre « connect »

Lorsqu'il n'est pas spécifié de fichier de connexion à utiliser dans les boucles, SPIP utilise le fichier de connexion par défaut (souvent nommé `connect.php`).

Pour toutes ces boucles là, on peut transmettre via l'url une connexion particulière qui sera alors appliquée en indiquant le paramètre `connect=nom`.

Exemple

Si vous avez 2 sites SPIP avec deux squelettes différents (un site A et un site B). En copiant le fichier de connexion du site A dans le site B (en le renommant en `A.php`) et inversement, vous pourrez alors naviguer selon les différentes combinaisons :

- `http://A/` (le contenu du site A s'affiche avec le squelette A)
- `http://B/` (le contenu du site B s'affiche avec le squelette B)
- `http://A/?connect=B` (le contenu du site B s'affiche avec le squelette A)
- `http://B/?connect=A` (le contenu du site A s'affiche avec le squelette B)

Pour résumer, passer un `connect=nom` dans l'url permet d'utiliser le fichier de connexion « nom » dans toutes les boucles des squelettes qui n'ont pas de connexion définie, comme `<BOUCLE_a(ARTICLES)>`.

Inclure suivant une connexion

Il est possible de passer une connexion particulière via une inclusion :

```
<INCLURE{fond=derniers_articles}{connect=demo.example.org}>
[(#INCLURE{fond=derniers_articles,
connect=demo.example.org})]
```

Une inclusion ne transmet pas automatiquement la connexion parente ; pour propager une connexion il faut la spécifier dans l'inclusion :

```
<INCLURE{fond=derniers_articles}{connect}>
[(#INCLURE{fond=derniers_articles, connect})]
```

Étendre SPIP

SPIP a été conçu depuis longtemps pour être adaptable. Il existe de nombreuses solutions pour l'affiner selon ses propres besoins, ou pour créer de nouvelles fonctions.

Cette partie explique différents moyens à la disposition des programmeurs pour étendre SPIP.

Généralités

Squelettes, plugins, chemins d'accès, fonctions `_dist()`... Voici quelques explications pour éclaircir tout cela !

Squelettes ou plugins ?

utiliser le dossier squelettes

Le dossier `squelettes/` permet de stocker tous les fichiers nécessaires à la personnalisation de votre site (squelettes, images, fichiers Javascript ou CSS, librairies PHP...).

ou créer un plugin

Un plugin, stocké dans un répertoire `plugins/nom_du_plugin/` permet également de stocker tout ce dont vous avez besoin pour votre site, exactement comme un dossier « squelettes ». Il permet par contre quelques actions supplémentaires. Cela concerne principalement l'exécution possible de traitements à l'installation ou à la désinstallation du plugin.

Alors, plugin ou simple dossier squelettes ?

D'une manière générale, on utilisera plutôt le dossier squelettes pour installer tout ce qui est spécifique à un site. Dès qu'un code est générique et réutilisable, le proposer sous forme de plugin sera plus adapté.

Déclarer des options

Lorsque un visiteur du site demande à voir une page, qu'elle soit déjà en cache ou non, SPIP exécute un certain nombre d'actions, dont celles de charger des fichiers d'options. Ces options peuvent par exemple définir des constantes ou modifier des variables globales.

Ces options peuvent être créées dans le fichier `config/mes_options.php` ou depuis un plugin en déclarant le nom du fichier dans `plugin.xml` tel que `<options>prefixePlugin_options.php</options>`.

Tous les fichiers d'options (celui du site, puis de tous les plugins) sont chargés à chaque appel de l'espace public et de l'espace privé ; ils doivent donc être les plus légers et économes possible.

Cet exemple, issu d'une contribution nommée « switcher », propose de modifier le jeu de squelettes utilisé par le site (l'adresse du dossier plus précisément) en fonction d'un paramètre `var_skel` dans l'url.

```
<?php
// 'nom' => 'chemin du squelette'
$squelettes = array(
  '2008'=>'squelettes/2008',
  '2007'=>'squelettes/2007',
);
// Si l'on demande un squelette particulier qui existe,  on
pose un cookie, sinon suppression du cookie
if (isset($_GET['var_skel'])) {
  if (isset($squelettes[$_GET['var_skel']]))
    setcookie('spip_skel', $_COOKIE['spip_skel'] =
$_GET['var_skel'], NULL, '/');
  else
    setcookie('spip_skel', $_COOKIE['spip_skel'] = '',
-24*3600, '/');
}
// Si un squelette particulier est sauve, on le definit comme
dossier squelettes
if (isset($_COOKIE['spip_skel']) AND
isset($squelettes[$_COOKIE['spip_skel']]))
  $GLOBALS['dossier_squelettes'] =
$squelettes[$_COOKIE['spip_skel']];
?>
```

Déclarer des fonctions

Contrairement aux fichiers d'options, les fichiers de fonctions ne sont pas chargés systématiquement, mais seulement au calcul des squelettes.

Ils permettent par exemple de définir de nouveaux filtres utilisables dans les squelettes. Ainsi, créer un fichier `squelettes/mes_fonctions.php` contenant le code ci-dessous, permet d'utiliser dans les squelettes le filtre "hello_world" (assez inutile !).

```
<?php
function filtre_hello_world($v, $add){
    return "Titre:" . $v . ' // Suivi de: ' . $add;
}
```

```
?>
```

```
[(#TITRE|hello_world{ce texte s'ajoute après})]
```

(affiche « Titre:titre de l'article // Suivi de : ce texte s'ajoute après »)

Pour utiliser de tels fichiers avec les plugins, il suffit de déclarer le nom du fichier dans `plugin.xml` par exemple `<fonctions>prefixePlugin_fonctions.php</fonctions>`. Il peut y avoir plusieurs déclarations dans un même plugin.

fonctions spécifiques à des squelettes

Parfois, des filtres sont spécifiques à un seul fichier de squelette. Il n'est pas toujours souhaitable dans ce cas de charger systématiquement toutes les fonctions connues à chaque calcul de page. SPIP permet donc de créer des fonctions qui ne seront appelées qu'au calcul d'un squelette particulier.

Il suffit de déclarer un fichier homonyme au squelette, dans le même répertoire, en le suffixant de `_fonctions.php`.

En reprenant l'exemple ci dessus, on pourrait tout à fait imaginer `[(#TITRE|hello_world{ce texte s'ajoute après})]` contenu dans un fichier `squelettes/world.html` et la fonction `hello_world` déclarée dans le fichier `squelettes/world_fonctions.php`

La notion de chemin

SPIP s'appuie sur un certain nombre de fonctions et de squelettes, contenus dans différents dossiers. Lorsqu'un script demande à ouvrir un fichier de SPIP pour charger une fonction ou lire un squelette, SPIP va regarder si le fichier existe dans un certain nombre de dossiers qu'il connaît. Dès qu'il le trouve, le fichier est alors utilisé.

Les dossiers sont parcourus selon une certaine priorité, définie par une constante `SPIP_PATH`, et éventuellement complété par une globale `$GLOBALS['dossier_squelettes']`

La lecture par défaut est la suivante :

- squelettes
- plugin B dépendant du plugin A
- plugin A
- squelettes-dist
- prive
- ecrire
- .

Surcharger un fichier

Une des premières possibilités pour modifier un comportement de SPIP est de copier un des fichiers de SPIP dans un répertoire de plus haute priorité, par exemple dans un plugin, ou dans son dossier de squelettes, en conservant la même arborescence (sans prendre en compte le dossier `ecrire/` néanmoins).

Ainsi, on pourrait imaginer modifier la façon dont SPIP gère le cache en copiant `ecrire/public/cacher.php` dans `squelettes/public/cacher.php`, puis en modifiant cette copie. C'est elle qui serait lue par défaut et en priorité.

Cette façon de procéder est à utiliser en connaissance de cause. Effectivement, ce genre de modifications est très sensible aux évolutions de SPIP ; vous risqueriez d'avoir de grandes difficultés de mises à jour à chaque changement de version de SPIP.

Surcharger une fonction _dist

De nombreuses déclarations de fonctions dans SPIP sont prévues pour n'être étendues qu'une seule fois. Ces fonctions possèdent l'extension « _dist » dans leur nom. Toutes les `balises`, `boucles` ou les `critères` sont déclarés de la sorte et peuvent donc être étendus de façon simple : en déclarant (par exemple dans le fichier `mes_fonctions.php`) la même fonction, sans le suffixe « _dist » dans le nom.

Il existe dans le fichier ecrire/public/boucles.php la fonction `boucle_ARTICLES_dist`. Elle peut être surchargée en déclarant une fonction :

```
function boucle_ARTICLES($id_boucle, &$boucles) {
...
}
```

Les pipelines

À certains endroits du code sont définis des « pipelines ». Les utiliser est une des meilleures façons de modifier ou d'adapter des comportements de SPIP.

Qu'est-ce qu'un pipeline ?

Un pipeline permet de faire passer un bout de code de main en main pour le compléter ou le modifier. Chaque plugin peut utiliser un pipeline existant. Pour cela, il déclare son utilisation dans le fichier `plugin.xml` de la sorte :

```
<pipeline>
   <nom>header_prive</nom>
   <inclure>cfg_pipeline.php</inclure>
</pipeline>
```

- Nom : indique le nom du pipeline à utiliser,
- Inclure : indique le nom du fichier qui contient la fonction à exécuter au moment de l'appel du pipeline (`prefixPlugin_nomPipeline()`).

Le fichier `config/mes_options.php` permet aussi d'ajouter des éléments à exécuter à un pipeline en déclarant :

```
$GLOBALS['spip_pipeline']['insert_head'] .=
"|nom_de_la_fonction";
```

Quels sont les pipelines existants ?

La liste des pipelines intégrés à SPIP (mais des plugins peuvent en créer de nouveaux) est visible dans le fichier ecrire/inc_version.php. Plusieurs types de pipelines existent, certains concernent les traitements typographiques, d'autres les enregistrements en base de données, d'autres l'affichage des pages privées ou publiques...

Déclarer un nouveau pipeline

Cela se passe en deux temps. Il faut tout d'abord déclarer l'existence du pipeline dans un fichier d'option `$GLOBALS['spip_pipeline']['nouveau_pipe'] = ''`

Ensuite, il faut l'appeler quelque part, soit dans un squelette soit dans un fichier PHP. La balise `#PIPELINE` ou la fonction PHP `pipeline()` utilisent les mêmes arguments.

- Squelettes : `#PIPELINE{nouveau_pipe,contenu au demarrage}`
- Php : `$data = pipeline('nouveau_pipe',"contenu au demarrage");`.

Dans les deux écritures, un premier texte « contenu au demarrage » est envoyé dans le pipeline. Tous les plugins ayant déclaré l'utilisation de ce pipeline vont recevoir la chaîne et pouvoir la compléter ou modifier. Après le dernier, le résultat est renvoyé.

Des pipelines argumentés

Il est souvent indispensable de passer des arguments issus du contexte, en plus des données renvoyées par le pipeline. Cela est permis en transmettant un tableau d'au moins deux clés avec une clé nommée "data". À la fin du chaînage des pipelines, seule la valeur de 'data' est renvoyée.

```
$data = pipeline('nouveau_pipe',array(
    'args'=>array(
        'id_article'=>$id_article
    ),
    'data'=>"contenu au demarrage"
);
```

```
[(#PIPELINE{nouveau_pipe,
    [(#ARRAY{
        args,[(#ARRAY{id_article,#ID_ARTICLE})],
        data,contenu au demarrage
    })]})]
```

Liste des pipelines

Cette partie décrit l'utilisation de certains pipelines de SPIP.

Nom	Description
accueil_encours (p.90)	Ajouter du contenu au centre de la page d'accueil.
accueil_gadget (p.91)	Ajouter des raccourcis en haut du contenu de la page d'accueil.
accueil_informations (p.92)	Informer sur les statistiques des objets éditoriaux sur la page d'accueil.
affichage_final (p.92)	Effectue des traitements juste avant l'envoi des pages publiques.
affiche_droite (p.93)	Ajouter du contenu dans la colonne « droite » de l'espace privé.
affiche_enfants (p.94)	Modifier ou compléter le contenu des listes présentant les enfants d'un objet dans l'espace privé
affiche_gauche (p.95)	Ajouter du contenu dans la colonne « gauche » de l'espace privé.
affiche_hierarchie (p.96)	Modifier le code HTML du fil d'ariane de l'espace privé.
affiche_milieu (p.97)	Ajouter du contenu au centre de la page sur les pages privées.
ajouter_boutons (p.98)	Ajouter des boutons dans le menu de l'espace privé.
ajouter_onglets (p.100)	Ajouter des onglets dans les pages de l'espace privé.
autoriser (p.102)	Charger des fonctions d'autorisations.
body_prive (p.103)	Insérer du contenu après `<body>` dans l'espace privé.
boite_infos (p.104)	Afficher des informations sur les objets dans les boîtes infos de l'espace privé.
declarer_tables_auxiliaires (p.105)	déclarer des tables SQL « auxiliaires »
declarer_tables_objets_surnoms (p.106)	Indiquer la relation entre le type d'objet et sa correspondance SQL.

Nom	Description
declarer_tables_principales (p.107)	Déclarer des tables SQL « principales »
editer_contenu_objet (p.109)	Modifier le contenu HTML des formulaires.
formulaire_charger (p.110)	Modifier le tableau de valeurs envoyé par la fonction `charger` d'un formulaire CVT.
insert_head (p.111)	Ajouter des contenus dans le `<head>` public.
jquery_plugins (p.112)	Ajouter des scripts JavaScript (espace public et privé).
pre_boucle (p.113)	Modifier les requêtes SQL servant à générer les boucles.
pre_liens (p.114)	Traiter les raccourcis typographiques relatifs aux liens
rechercher_liste_des_champs (p.115)	Définir des champs et des pondérations pour les recherches sur une table.
rechercher_liste_des_jointures (p.116)	Définir des jointures pour les recherches sur une table.
recuperer_fond (p.117)	Modifie le résultat de compilation d'un squelette
styliser (p.118)	Modifier la recherche des squelettes utilisés pour générer une page.

accueil_encours

Ce pipeline permet d'ajouter du contenu au centre de la page d'accueil de l'espace privé, par exemple pour afficher les nouveaux articles proposés à la publication.

```
$res = pipeline('accueil_encours', $res);
```

Ce pipeline reçoit un texte et retourne le texte complété.

Exemple

Le plugin « breves », s'il existait, l'utiliserait pour ajouter la liste des dernières brèves proposées :

```
function breves_accueil_encours($texte){
    $texte .= afficher_objets('breve',
afficher_plus(generer_url_ecrire('breves')) .
_T('info_breves_valider'), array("FROM" => 'spip_breves',
'WHERE' => "statut='prepa' OR statut='prop'", 'ORDER BY'
=> "date_heure DESC"), true);
    return $texte;
}
```

accueil_gadget

Ce pipeline permet d'ajouter des liens en haut du contenu de la page d'accueil de l'espace privé, dans le cadre qui liste différentes actions possibles (créer une rubrique, un article, une brève…).

```
$gadget = pipeline('accueil_gadgets', $gadget);
```

Ce pipeline reçoit un texte et retourne le texte complété.

Exemple

Le plugin « breves », s'il existait, l'utiliserait pour ajouter un lien pour créer ou voir la liste des brèves en fonction du statut de l'auteur connecté :

```
function breves_accueil_gadgets($texte){
    if ($GLOBALS['meta']['activer_breves'] != 'non') {
        // creer sinon voir
        if ($GLOBALS['visiteur_session']['statut'] ==
"0minirezo") {
            $ajout =
icone_horizontale(_T('icone_nouvelle_breve'),
generer_url_ecrire("breves_edit","new=oui"),
"breve-24.png","new", false);
        } else {
```

```
            $ajout = icone_horizontale
(_T('icone_breves'), generer_url_ecrire("breves",""),
"breve-24.png", "", false);
        }
        $texte = str_replace("</tr></table>",
"<td>$ajout</td></tr></table>", $texte);
    }
    return $texte;
}
```

accueil_informations

Ce pipeline permet d'ajouter des informations sur les objets éditoriaux dans la navigation latérale de la page d'accueil.

```
$res = pipeline('accueil_informations', $res);
```

Il prend un texte en paramètre qu'il peut compléter et qu'il retourne.

Exemple

Le plugin « breves », s'il existait, pourrait l'utiliser pour ajouter le nombre de brèves en attente de validation :

```
function breves_accueil_informations($texte){
    include_spip('base/abstract_sql');
    $q = sql_select("COUNT(*) AS cnt, statut",
'spip_breves', '', 'statut', '','', "COUNT(*)<>0");
    // traitements sur le texte en fonction du resultat
    // ...
    return $texte;
}
```

affichage_final

Ce pipeline est appelé au moment de l'envoi du contenu d'une page au navigateur. Il reçoit un texte (la page HTML le plus souvent) qu'il peut compléter. Les modifications ne sont pas mises en cache par SPIP.

```
echo pipeline('affichage_final', $page['texte']);
```

C'est un pipeline fréquemment utilisé par les plugins permettant une grande variété d'actions. Cependant, comme le résultat n'est pas mis en cache, et que le pipeline est appelé à chaque page affichée, il faut être prudent et le réserver à des utilisations peu gourmandes en ressources.

Exemple

Le plugin « XSPF », qui permet de réaliser des galeries multimédias, ajoute un javascript uniquement sur les pages qui le nécessitent, de cette façon :

```
function xspf_affichage_final($page) {
    // on regarde rapidement si la page a des classes
player
    if (strpos($page, 'class="xspf_player"')===FALSE)
        return $page;
    // Si oui on ajoute le js de swfobject
    $jsFile = find_in_path('lib/swfobject/swfobject.js');
    $head = "<script src='$jsFile' type='text/
javascript'></script>";
    $pos_head = strpos($page, '</head>');
    return substr_replace($page, $head, $pos_head, 0);
}
```

Le plugin « target » lui, ouvre les liens extérieurs dans une nouvelle fenêtre (oui, c'est très mal !) :

```
function target_affichage_final($texte) {
    $texte = str_replace('spip_out"', 'spip_out"
target="_blank"', $texte);
    $texte = str_replace('rel="directory"',
'rel="directory" class="spip_out" target="_blank"',
$texte);
    $texte = str_replace('spip_glossaire"',
'spip_glossaire" target="_blank"', $texte);
    return $texte;
}
```

affiche_droite

Ce pipeline permet d'ajouter du contenu dans la colonne « droite » (qui n'est d'ailleurs pas forcément à droite – c'est en fonction des préférences et de la langue de l'utilisateur) des pages « exec » de l'espace privé. Cette colonne contient généralement des liens de navigation transversale en relation avec le contenu affiché, comme le cadre « dans la même rubrique » qui liste les derniers articles publiés.

```
echo pipeline('affiche_droite', array(
    'args'=>array(
        'exec'=>'naviguer',
        'id_rubrique'=>$id_rubrique),
    'data'=>''));
```

Ce pipeline reçoit le nom de la page « exec » affichée ainsi que, s'il y a lieu, l'identifiant de l'objet en cours de lecture, comme ici « id_rubrique ».

Exemple

Le plugin « odt2spip » qui permet de créer des articles SPIP à partir de documents Open Office Texte (extension `.odt`) utilise ce pipeline pour ajouter un formulaire dans la vue des rubriques afin d'envoyer le fichier `odt` :

```
function odt2spip_affiche_droite($flux){
    $id_rubrique = $flux['args']['id_rubrique'];
    if ($flux['args']['exec']=='naviguer' AND
$id_rubrique > 0) {
        $icone =
icone_horizontale(_T("odtspip:importer_fichier"), "#",
"", _DIR_PLUGIN_ODT2SPIP . "images/odt-24.png", false,
"onclick='$(\"#boite_odt2spip\").slideToggle(\"fast\");
return false;'");
        $out = recuperer_fond('formulaires/odt2spip',
array('id_rubrique'=>$id_rubrique, 'icone'=>$icone));
        $flux['data'] .= $out;
    }
    return $flux;
}
```

affiche_enfants

Ce pipeline permet d'ajouter ou de modifier le contenu des listes présentant les enfants d'un objet. Il reçoit dans `args` le nom de la page en cours et l'identifiant de l'objet, et dans `data` le code HTML présentant les enfants de l'objet. Ce pipeline n'est appelé actuellement qu'à un seul endroit : sur la page de navigation des rubriques.

```
$onglet_enfants = pipeline('affiche_enfants', array(
    'args'=>array(
        'exec'=>'naviguer',
        'id_rubrique'=>$id_rubrique),
    'data'=>$onglet_enfants));
```

affiche_gauche

Ce pipeline permet d'ajouter du contenu dans la colonne « gauche » des pages de l'espace privé. Cette colonne contient généralement des liens ou des formulaires en relation avec le contenu affiché, tel que le formulaire d'ajout de logo.

```
echo pipeline('affiche_gauche', array(
    'args'=>array(
        'exec'=>'articles',
        'id_article'=>$id_article),
    'data'=>''));
```

Ce pipeline reçoit le nom de la page « exec » affichée ainsi que, s'il y a lieu, l'identifiant de l'objet en cours de lecture, comme ici « id_article ».

Exemple

Le plugin « spip bisous », qui permet d'envoyer des bisous (!) entre les auteurs, utilise ce pipeline pour afficher la liste des bisous reçus et envoyés sur les pages des auteurs :

```
function bisous_affiche_gauche($flux){
    include_spip('inc/presentation');
    if ($flux['args']['exec'] == 'auteur_infos'){
        $flux['data'] .=
```

```
        debut_cadre_relief('',true,'',
_T('bisous:bisous_donnes')) .
        recuperer_fond('prive/bisous_donnes',
array('id_auteur'=>$flux['args']['id_auteur'])) .
        fin_cadre_relief(true) .
        debut_cadre_relief('',true,'',
_T('bisous:bisous_recus')) .
        recuperer_fond('prive/bisous_recus',
array('id_auteur'=>$flux['args']['id_auteur'])) .
        fin_cadre_relief(true);
    }
    return $flux;
}
```

affiche_hierarchie

Le pipeline « affiche_hierarchie » permet de modifier ou compléter le code HTML du fil d'ariane de l'espace privé. Il reçoit un certain nombre d'informations dans `args` : l'objet et son identifiant en cours de lecture s'il y a lieu, éventuellement l'identifiant du secteur.

```
$out = pipeline('affiche_hierarchie', array(
    'args'=>array(
        'id_parent'=>$id_parent,
        'message'=>$message,
        'id_objet'=>$id_objet,
        'objet'=>$type,
        'id_secteur'=>$id_secteur,
        'restreint'=>$restreint),
    'data'=>$out));
```

Exemple

Le plugin « polyhiérarchie » qui autorise une rubrique à avoir plusieurs parents utilise ce pipeline pour lister les différents parents de la rubrique ou de l'article visité :

```
function polyhier_affiche_hierarchie($flux){
    $objet = $flux['args']['objet'];
    if (in_array($objet,array('article','rubrique'))){
```

```
        $id_objet = $flux['args']['id_objet'];
        include_spip('inc/polyhier');
        $parents =
polyhier_get_parents($id_objet,$objet,$serveur='');
        $out = array();
        foreach($parents as $p)
            $out[] = "[->rubrique$p]";
        if (count($out)){
            $out = implode(', ',$out);
            $out = _T('polyhier:label_autres_parents')."
".$out;
            $out = PtoBR(propre($out));
            $flux['data'] .= "<div
id='chemins_transverses'>$out</div>";
        }
    }
    return $flux;
}
```

affiche_milieu

Ce pipeline permet d'ajouter du contenu sur les pages `exec/` de SPIP, après le contenu prévu au centre de la page.

Il est appelé comme ceci :

```
echo pipeline('affiche_milieu',array(

'args'=>array('exec'=>'nom_du_exec','id_objet'=>$id_objet),
    'data'=>''));
```

Exemple

Le plugin « Sélection d'articles » l'utilise pour ajouter un formulaire dans la page des rubriques afin de créer une sélection d'articles :

```
function pb_selection_affiche_milieu($flux) {
    $exec = $flux["args"]["exec"];
```

```
    if ($exec == "naviguer") {
        $id_rubrique = $flux["args"]["id_rubrique"];
        $contexte = array('id_rubrique'=>$id_rubrique);
        $ret = "<div id='pave_selection'>";
        $ret .= recuperer_fond("selection_interface",
$contexte);
        $ret .= "</div>";
        $flux["data"] .= $ret;
    }
    return $flux;
}
```

Le plugin « statistiques » (en développement pour la prochaine version de SPIP) l'utilise pour ajouter un formulaire de configuration dans les pages de configuration de SPIP :

```
function stats_affiche_milieu($flux){
    // afficher la configuration ([des]activer les
statistiques).
    if ($flux['args']['exec'] == 'config_fonctions') {
        $compteur = charger_fonction('compteur',
'configuration');
        $flux['data'] .= $compteur();
    }
    return $flux;
}
```

ajouter_boutons

Ce pipeline permet d'ajouter des boutons dans le menu de navigation de l'espace privé. Il n'a plus vraiment d'utilité depuis la création du tag `<bouton>` dans le fichier `plugin.xml` (voir Définir des boutons (p.193)).

```
$boutons_admin = pipeline('ajouter_boutons', $boutons_admin);
```

Le pipeline « ajouter_boutons » reçoit un tableau associatif « identifiant d'un bouton / description du bouton » (classe PHP Bouton). Un bouton peut déclarer un sous-menu dans la variable « sousmenu » de la classe Bouton. Il faut créer une instance de la classe `Bouton` pour définir celui-ci :

```
function plugin_ajouter_boutons($boutons_admin){
$boutons_admin['identifiant'] =
    new Bouton('image/du_bouton.png', 'Titre du bouton',
'url');
$boutons_admin['identifiant']->sousmenu['autre_identifiant']
=
    new Bouton('image/du_bouton.png', 'Titre du bouton',
'url');
return $boutons_admin;
}
```

Le troisième paramètre `url` de la classe `Bouton` est optionnel. Par défaut ce sera une page « exec » de même nom que l'identifiant donné (`ecrire/?exec=identifiant`).

Exemple

Le plugin « Thelia » qui permet d'interfacer SPIP avec le logiciel Thélia, utilise ce pipeline pour ajouter au menu « Édition » (identifiant « naviguer ») un lien vers le catalogue Thélia :

```
function spip_thelia_ajouter_boutons($boutons_admin) {
    // si on est admin
    if ($GLOBALS['visiteur_session']['statut'] ==
"0minirezo") {
        $boutons_admin['naviguer']-
>sousmenu['spip_thelia_catalogue'] =
        new Bouton(_DIR_PLUGIN_SPIP_THELIA . 'img_pack/
logo_thelia_petit.png', 'Catalogue Th&eacute;lia');
    }
    return $boutons_admin;
}
```

Migration vers la nouvelle écriture

Pour migrer cet exemple dans la nouvelle écriture, il faut séparer 2 choses : la déclaration du bouton et l'autorisation de le voir ou non (ici seulement si l'on est administrateur). La déclaration s'écrit dans le fichier `plugin.xml` :

```
<bouton id="spip_thelia_catalogue" parent="naviguer">
    <icone>smg_pack/logo_thelia_petit.png</icone>
    <titre>chaine de langue du titre</titre>
</bouton>
```

L'autorisation passe par une fonction d'autorisation spécifique (utiliser le pipeline autoriser (p.102) pour la définir) :

```
function
autoriser_spip_thelia_catalogue_bouton_dist($faire,
$type, $id, $qui, $opt) {
    return ($qui['statut'] == '0minirezo');
}
```

ajouter_onglets

Ce pipeline permet d'ajouter des onglets dans les pages `exec` de l'espace privé. Il n'a plus vraiment d'utilité depuis la création du tag `<onglet>` dans le fichier `plugin.xml` (voir Définir des onglets (p.196)).

```
return pipeline('ajouter_onglets',
array('data'=>$onglets,'args'=>$script));
```

Le pipeline « ajouter_onglets » reçoit un tableau associatif « identifiant de l'onglet / description de l'onglet » (classe PHP Bouton), mais aussi un identifiant de barre d'onglet (dans `args`).

```
// ajout d'un onglet sur la page de configuration de SPIP
function plugin_ajouter_onglets($flux){
    if ($flux['args']=='identifiant')
        $flux['data']['identifiant_bouton']= new Bouton("mon/
image.png", "titre de l'onglet"), 'url');
    return $flux;
}
```

Le troisième paramètre `url` de la classe `Bouton` est optionnel. Par défaut ce sera une page « exec » de même nom que l'identifiant donné (`ecrire/?exec=identifiant`).

Dans les pages `exec`, une barre d'outil s'appelle avec deux arguments : l'identifiant de la barre désirée et l'identifiant de l'onglet actif :

```
echo barre_onglets("identifiant barre d'onglet", "identifiant
de l'onglet actif");
echo barre_onglets("configuration", "contenu");
```

Exemple

Le plugin « Agenda » modifie les onglets par défaut du calendrier de SPIP en utilisant ce pipeline :

```
function agenda_ajouter_onglets($flux) {
if($flux['args']=='calendrier'){
    $flux['data']['agenda']= new Bouton(
        _DIR_PLUGIN_AGENDA . '/img_pack/agenda-24.png',
        _T('agenda:agenda'),
        generer_url_ecrire("calendrier","type=semaine"));
    $flux['data']['calendrier'] = new Bouton(
        'cal-rv.png',
        _T('agenda:activite_editoriale'),
        generer_url_ecrire("calendrier",
"mode=editorial&type=semaine"));
}
return $flux;
}
```

Migration vers la nouvelle écriture

Pour migrer cet exemple dans la nouvelle écriture, il faut séparer 2 choses : la déclaration du bouton et l'autorisation de le voir ou non. La déclaration s'écrit dans le fichier `plugin.xml` :

```
<onglet id="agenda" parent="calendrier">
    <icone>img_pack/agenda-24.png</icone>
    <titre>agenda:agenda</titre>
    <url>calendrier</url>
    <args>type=semaine</args>
</onglet>
<onglet id="calendrier" parent="calendrier">
    <icone>cal-rv.png</icone>
    <titre>agenda:activite_editoriale</titre>
```

```
    <url>calendrier</url>
    <args>mode=editorial&type=semaine</args>
</onglet>
```

L'autorisation passe par une fonction spécifique (utiliser le pipeline autoriser (p.102) pour la définir) :

```
function autoriser_calendrier_onglet_dist($faire, $type,
$id, $qui, $opt) {
    return true;
}
function autoriser_agenda_onglet_dist($faire, $type, $id,
$qui, $opt) {
    return true;
}
```

autoriser

Le pipeline « autoriser » est particulier. Il permet simplement de charger des fonctions d'autorisations au tout premier appel de la fonction `autoriser()`. Ce pipeline ne transmet rien et ne reçoit rien.

```
pipeline('autoriser');
```

Grâce à lui, un plugin peut déclarer des autorisations spécifiques, regroupées dans un fichier « prefixePlugin_autorisations.php » et les déclarer, dans `plugin.xml` comme ceci :

```
<pipeline>
    <nom>autoriser</nom>
    <inclure>prefixePlugin_autorisations.php</inclure>
</pipeline>
```

Outre les fonctions d'autorisations, le fichier doit contenir la fonction appelée par tous les pipelines (« prefixePlugin_nomDuPipeline() ») mais elle n'a rien a effectuer. Un exemple :

```
function prefixePlugin_autoriser(){}
```

Exemple

Le plugin « forum » déclare quelques autorisations. Son fichier `plugin.xml` contient :

```
<pipeline>
    <nom>autoriser</nom>
    <inclure>forum_autoriser.php</inclure>
</pipeline>
```

Et le fichier appelé, « forum_autoriser.php » contient :

```
// declarer la fonction du pipeline
function forum_autoriser(){}
function
autoriser_forum_interne_suivi_bouton_dist($faire, $type,
$id, $qui, $opt) {
    return true;
}
function autoriser_forum_reactions_bouton_dist(($faire,
$type, $id, $qui, $opt) {
    return autoriser('publierdans', 'rubrique',
_request('id_rubrique'));
}
// Moderer le forum ?
// = modifier l'objet correspondant (si forum attache a
un objet)
// = droits par defaut sinon (admin complet pour
moderation complete)
function autoriser_modererforum_dist($faire, $type, $id,
$qui, $opt) {
    return autoriser('modifier', $type, $id, $qui, $opt);
}
// Modifier un forum ?  jamais !
function autoriser_forum_modifier_dist($faire, $type,
$id, $qui, $opt) {
    return false;
}
...
```

body_prive

Ce pipeline permet de modifier la balise HTML `body` de l'espace privé ou d'ajouter du contenu juste après cette balise. Il est appelé par la fonction `commencer_page()` exécutée lors de l'affichage des pages privées.

```
$res = pipeline('body_prive',
    "<body class='$rubrique $sous_rubrique " .
_request('exec') . "'"
    . ($GLOBALS['spip_lang_rtl'] ? " dir='rtl'" : "") . '>');
```

boite_infos

Ce pipeline permet de gérer les informations affichées dans l'espace privé dans le cadre d'information des objets SPIP. C'est dans ce cadre par exemple qu'est affiché le numéro d'un article, ainsi que les liens pour changer son statut.

Il reçoit un tableau en paramètre.

- `data` : contient ce qui sera ensuite affiché sur la page,
- `args` contient un tableau avec :
  - `type` : le type d'objet (article, rubrique...)
  - `id` : l'identifiant (8, 12...)
  - `row` : tableau contenant l'ensemble des champs SQL de l'objet et leurs valeurs.

Exemple

Le plugin « Prévisualisation pour les articles en cours de rédaction » (previsu_redac) utilise ce pipeline pour ajouter le bouton « previsualiser » lorsqu'un article est en cours de rédaction (ce lien n'apparaît normalement que lorsque l'article a été proposé à publication) :

```
function previsu_redac_boite_infos(&$flux){
    if ($flux['args']['type']=='article'
      AND $id_article=intval($flux['args']['id'])
      AND $statut = $flux['args']['row']['statut']
      AND $statut == 'prepa'
      AND autoriser('previsualiser')){
        $message = _T('previsualiser');
```

```
        $h = generer_url_action('redirect',
"type=article&id=$id_article&var_mode=preview");
        $previsu =
        icone_horizontale($message, $h, "racine-24.gif",
"rien.gif",false);
        if ($p = strpos($flux['data'],'</ul>')){
            while($q =
strpos($flux['data'],'</ul>',$p+5)) $p=$q;
            $flux['data'] = substr($flux['data'],0,$p+5)
. $previsu . substr($flux['data'], $p+5);
        }
        else
            $flux['data'].= $previsu;
    }
    return $flux;
}
```

declarer_tables_auxiliaires

Ce pipeline sert à déclarer des tables « auxiliaires », c'est à dire qui ne servent essentiellement qu'à réaliser des liaisons avec des tables principales.

Comme le pipeline declarer_tables_principales (p.107), il reçoit la liste des tables, se composant du même tableau.

Exemple

Le plugin « SPIP Bisous » qui permet qu'un auteur (membre inscrit sur le site) envoie un bisou à un autre auteur (l'équivalent d'un *poke* sur certains réseaux sociaux) déclare une table `spip_bisous` liant 2 auteurs avec la date du bisou, avec le code ci-dessous. On remarquera la clé primaire composée de 2 champs.

```
function
bisous_declarer_tables_auxiliaires($tables_auxiliaires){
    $spip_bisous = array(
        'id_donneur' => 'bigint(21) DEFAULT "0" NOT
NULL',
```

```
        'id_receveur' => 'bigint(21) DEFAULT "0" NOT
NULL',
        'date' => 'datetime DEFAULT "0000-00-00 00:00:00"
NOT NULL'
    );

    $spip_bisous_cles = array(
        'PRIMARY KEY' => 'id_donneur, id_receveur'
    );

    $tables_auxiliaires['spip_bisous'] = array(
        'field' => &$spip_bisous,
        'key' => &$spip_bisous_cles
    );

    return $tables_auxiliaires;
}
```

declarer_tables_objets_surnoms

Il permet d'indiquer la relation entre le type d'objet et sa correspondance SQL. Par défaut, un 's' de pluriel est ajouté (l'objet 'article' donne une table SQL 'articles'). Le pipeline reçoit un tableau des correspondances de SPIP.

Appel du pipeline :

```
$surnoms = pipeline('declarer_tables_objets_surnoms',
    array(
        'article' => 'articles',
        'auteur' => 'auteurs',
        //...
    ));
```

Ces correspondances permettent aux fonctions `table_objet()` et `objet_type()` de retrouver l'un et l'autre :

```
// type...
$type = objet_type('spip_articles'); // article
$type = objet_type('articles'); // article
// table...
$objet = table_objet('article'); // articles
```

```php
$table = table_objet_sql('article'); // spip_articles
// id...
$_id_objet = id_table_objet('articles'); // id_article
$_id_objet = id_table_objet('spip_articles'); // id_article
$_id_objet = id_table_objet('article'); // id_article
```

Exemple

Le plugin « jeux » déclare sa relation de la sorte :

```php
function jeux_declarer_tables_objets_surnoms($surnoms) {
    $surnoms['jeu'] = 'jeux';
    return $surnoms;
}
```

declarer_tables_principales

Ce pipeline permet de déclarer des tables ou des champs de tables supplémentaires à SPIP, en indiquant le type SQL de chaque champ, les clés primaires, les clés d'index, parfois des clés de jointures.

Ce pipeline concerne les tables dites « principales » qui contiennent du contenu éditorial, à comparer aux tables dites « auxiliaires » qui contiennent plutôt des tables de liaisons entre les tables principales.

Ces déclarations servent à SPIP pour :

- gérer l'affichage des boucles (mais ce n'est pas indispensable car SPIP sait récupérer les descriptions d'une table même si elle n'est pas déclarée),
- créer les tables (ou les champs manquants) à l'installation de SPIP ou d'un plugin,
- prendre en compte ces tables et ces champs dans les sauvegardes et restaurations faites par le gestionnaire de sauvegarde de SPIP (les *dump*).

La fonction prend en paramètre la liste des tables déjà déclarées et doit retourner ce tableau. Ce tableau liste des tables avec pour chacune un tableau de 2 à 3 clés (`join` est optionnel) :

```
$tables_principales['spip_nom'] = array(
    'field' => array('champ'=>'code sql de creation'),
    'key' => array('type' => 'nom du/des champs'),
    'join' => array('champ'=>'champ de liaison')
);
```

SPIP fait appel à ce pipeline lors de la déclaration des tables utilisées, dans le fichier ecrire/base/serial.php.

Exemple

Le plugin « Agenda » déclare une table « spip_evenements » avec de nombreux champs. Il déclare la clé primaire (`id_evenement`), 3 index (`date_debut`, `date_fin` et `id_article`), ainsi que deux clés préférentielles pour les jointures : `id_evenement` et `id_article` (je crois que l'ordre est important).

Ce plugin déclare aussi un champ "agenda" dans la table `spip_rubriques`.

```
function
agenda_declarer_tables_principales($tables_principales){
    //-- Table EVENEMENTS -------------------
    $evenements = array(
        "id_evenement"  => "bigint(21) NOT NULL",
        "id_article"    => "bigint(21) DEFAULT '0' NOT
NULL",
        "date_debut"    => "datetime DEFAULT '0000-00-00
00:00:00' NOT NULL",
        "date_fin"  => "datetime DEFAULT '0000-00-00
00:00:00' NOT NULL",
        "titre" => "text NOT NULL",
        "descriptif"    => "text NOT NULL",
        "lieu"  => "text NOT NULL",
        "adresse"   => "text NOT NULL",
        "inscription" => "tinyint(1) DEFAULT 0 NOT NULL",
        "places" => "int(11) DEFAULT 0 NOT NULL",
        "horaire" => "varchar(3) DEFAULT 'oui' NOT NULL",
        "id_evenement_source"   => "bigint(21) NOT NULL",
        "maj"   => "TIMESTAMP"
        );
```

```
    $evenements_key = array(
        "PRIMARY KEY"    => "id_evenement",
        "KEY date_debut"     => "date_debut",
        "KEY date_fin"   => "date_fin",
        "KEY id_article"     => "id_article"
        );

    $tables_principales['spip_evenements'] = array(
        'field' => &$evenements,
        'key' => &$evenements_key,
        'join'=>array(
            'id_evenement'=>'id_evenement',
            'id_article'=>'id_article'
        ));

$tables_principales['spip_rubriques']['field']['agenda']
= 'tinyint(1) DEFAULT 0 NOT NULL';
    return $tables_principales;
}
```

editer_contenu_objet

Ce pipeline est appelé au moment de l'affichage d'un formulaire d'édition d'un objet de SPIP. Il permet de modifier le contenu HTML du formulaire. Ce pipeline est appelé comme paramètre de chargement d'un formulaire CVT (p.179) :

```
$contexte['_pipeline'] = array('editer_contenu_objet',
array('type'=>$type, 'id'=>$id));
```

Le pipeline transmet :

- le type (`type`) , l'identifiant de l'objet (`id`) et le contexte de compilation (tableau `contexte`) dans le tableau `args`
- le code HTML dans la clé `data`

Exemple

Le plugin « OpenID » ajoute un champ de saisie dans le formulaire d'édition des auteurs :

```
function openid_editer_contenu_objet($flux){
    if ($flux['args']['type']=='auteur') {
        $openid = recuperer_fond('formulaires/inc-
openid', $flux['args']['contexte']);
        $flux['data'] = preg_replace('%(<li
class="editer_email(.*?)</li>)%is', '$1'."\n".$openid,
$flux['data']);
    }
    return $flux;
}
```

formulaire_charger

Le pipeline `formulaire_charger` permet de modifier le tableau de valeurs envoyé par la fonction `charger` d'un formulaire CVT.

Il reçoit en argument le nom du formulaire ainsi que les paramètres transmis au formulaire dans les fonctions `charger`, `verifier` et `traiter`. Il retourne le tableau des valeurs à charger.

```
$valeurs = pipeline(
    'formulaire_charger',
    array(
        'args'=>array('form'=>$form,'args'=>$args),
        'data'=>$valeurs)
);
```

Exemple

Le plugin « noSpam » se sert de ce pipeline pour ajouter un jeton indiquant une durée de validité sur les formulaires sélectionnés par une variable globale :

```
$GLOBALS['formulaires_no_spam'][] = 'forum';
```

```
//
function nospam_formulaire_charger($flux){
    $form = $flux['args']['form'];
    if (in_array($form,
$GLOBALS['formulaires_no_spam'])){
        include_spip("inc/nospam");
        $jeton = creer_jeton($form);
        $flux['data']['_hidden'] .= "<input type='hidden'
name='_jeton' value='$jeton' />";
    }
    return $flux;
}
```

insert_head

Le pipeline `insert_head` permet d'ajouter des contenus dans la partie `<head>` d'une page HTML :

- au moment de l'appel à `#INSERT_HEAD` si la balise est définie,
- sinon juste avant la fin du header (avant `</head>`) si la fonction `f_insert_head` est définie dans le pipeline affichage_final, par exemple avec ceci dans `mes_options.php` :

```
$spip_pipeline['affichage_final'] .= '|f_insert_head';
```

Le pipeline reçoit le contenu à ajouter et retourne donc un contenu :

```
function prefixPlugin_insert_head($flux){
    $flux .= "<!-- un commentaire pour rien ! -->\n";
    return $flux;
}
```

Exemple

Ajouter un appel à une fonction jQuery, ici pour afficher une barre d'outil sur les balises `textarea` des formulaires de Crayons (avec le plugin Porte Plume) :

```
function documentation_insert_head($flux){
    $flux .= <<<EOF
<script type="text/javascript">
<!--
(function($){
$(document).ready(function(){
    /* Ajouter une barre porte plume sur les crayons */
    function barrebouilles_crayons(){
        $('.formulaire_crayon textarea.crayon-
active').barre_outils('edition');
    }
    barrebouilles_crayons();
    onAjaxLoad(barrebouilles_crayons);
});
})(jQuery);
-->
</script>
EOF;
    return $flux;
}
```

La fonction JavaScript `onAjaxLoad` permet de rappeler la fonction donnée en paramètre lors d'un rechargement AJAX d'un élément de la page.

jquery_plugins

Ce pipeline permet d'ajouter très simplement dans les pages privées et publiques (ayant une balise #INSERT_HEAD (p.36)) des scripts JavaScript.

Ce pipeline reçoit et retourne un tableau d'adresses de fichiers à insérer et est appelé comme suit :

```
$scripts = pipeline('jquery_plugins', array(
    'javascript/jquery.js',
    'javascript/jquery.form.js',
    'javascript/ajaxCallback.js'
));
```

Exemple

Ajouter le script « jquery.cycle.js » sur toutes les pages :

```
function prefixPlugin_jquery_plugins($scripts){
    $scripts[] = "javascript/jquery.cycle.js";
    return $scripts;
}
```

pre_boucle

Le pipeline `pre_boucle` permet de modifier les requêtes SQL servant à générer les boucles. Il agit après la prise en compte des critères (fonctions `critere_NOM()`) par le compilateur et avant l'appel des fonctions `boucle_NOM()`.

Ce pipeline est appelé à la création de chaque boucle au moment de la compilation. Il reçoit un objet `Boucle` en paramètre, contenant les données issues de la compilation concernant la boucle parcourue.

Il est ainsi possible d'agir sur la boucle en fonction des critères qui lui sont passés, par exemple en modifiant les paramètres de sélections ou la condition *where* d'une boucle.

Exemple

Le plugin « mots techniques » ajoute un champ technique sur les groupes de mots de SPIP. Lorsqu'aucun critère `{technique}` n'est ajouté sur la boucle GROUPES_MOTS, la boucle est alors filtrée automatiquement, affichant uniquement les groupes ayant un champ technique vide. Ce fonctionnement pourrait aussi être réalisé en créant une fonction `boucle_GROUPES_MOTS()`.

```
function mots_techniques_pre_boucle($boucle){
    if ($boucle->type_requete == 'groupes_mots') {
        $id_table = $boucle->id_table;
        $mtechnique = $id_table .'.technique';
```

```
        // Restreindre aux mots cles non techniques
        if (!isset($boucle-
>modificateur['criteres']['technique']) &&
            !isset($boucle->modificateur['tout'])) {
                $boucle->where[]= array("'='",
"'$mtechnique'", "'\"\"'");
        }
    }
    return $boucle;
}
```

Le tableau `$boucle->where[]` reçoit comme valeurs des tableaux. Ces tableaux ont 3 entrées : l'opérateur, le champ, la valeur. Ici, on ajoute `$mtechnique=''` par :

```
$boucle->where[]= array("'='", "'$mtechnique'",
"'\"\"'");
```

pre_liens

Le pipeline « pre_liens » permet de traiter les raccourcis typographiques relatifs aux liens tel que `[titre->url]`. Il est appelé par la fonction `expanser_liens()`, elle-même appelée par la fonction `propre()`.

```
$texte = pipeline('pre_liens', $texte);
```

SPIP se sert lui-même de ce point d'entrée pour effectuer des traitements sur le texte reçu en intégrant 3 fonctions dans la définition du pipeline dans le fichier ecrire/inc_version.php, définies dans le fichier ecrire/inc/lien.php :

- `traiter_raccourci_liens` génère automatiquement des liens si un texte ressemble à une URL,
- `traiter_raccourci_glossaire` gère les raccourcis `[?titre]` pointant vers un glossaire (p.200).
- `traiter_raccourci_ancre` s'occupe des raccourcis `[<-nom de l'ancre]` créant une ancre nommée

Exemple

Le plugin « documentation » (qui gère cette documentation), utilise ce pipeline pour ajouter automatiquement un attribut `title` sur les raccourcis de liens internes comme `[->art30]`, le transformant en `[|art30->art30]` (ce pis-aller sert à afficher le numéro de la page relative au lien lorsque l'on exporte le contenu de la documentation au format PDF)

```
function documentation_pre_liens($texte){
    // uniquement dans le public
    if (test_espace_prive()) return $texte;
    $regs = $match = array();
    // pour chaque lien
    if (preg_match_all(_RACCOURCI_LIEN, $texte, $regs,
PREG_SET_ORDER)) {
        foreach ($regs as $reg) {
            // si le lien est de type raccourcis "art40"
            if (preg_match(_RACCOURCI_URL, $reg[4],
$match)) {
                $title = '|' . $match[1] . $match[2];
                // s'il n'y a pas deja ce title
                if (false === strpos($reg[0], $title)) {
                    $lien = substr_replace($reg[0],
$title, strpos($reg[0], '->'), 0);
                    $texte = str_replace($reg[0], $lien,
$texte);
                }
            }
        }
    }
    return $texte;
}
```

rechercher_liste_des_champs

Ce pipeline permet de gérer les champs pris en compte par le moteur de recherche de SPIP pour une table donnée. Ce pipeline reçoit un tableau de noms d'objet SPIP (article, rubrique...) contenant les noms des champs à prendre en compte pour la recherche (titre, texte...) affectés d'un coefficient de pondération du résultat : plus le coefficient est élevé, plus la recherche attribue des points si la valeur cherchée est présente dans le champ.

Exemple

```
function
prefixPlugin_rechercher_liste_des_champs($tables){
    // ajouter un champ ville sur les articles
    $tables['article']['ville'] = 3;
    // supprimer un champ de la recherche
    unset($tables['rubrique']['descriptif']);
    // retourner le tableau
    return $tables;
}
```

rechercher_liste_des_jointures

Ce pipeline utilisé dans ecrire/inc/rechercher.php permet de déclarer des recherches à effectuer sur d'autres tables que la table où la recherche est demandée, pour retourner des résultats en fonction de données extérieures à la table. Grâce à cela, une recherche d'un nom d'auteur sur une boucle ARTICLES retourne les articles associés au nom de l'auteur (via la table AUTEURS).

Ce pipeline reçoit un tableau de tables contenant un tableau de couples table, champ, pondération (comme le pipeline « rechercher_liste_des_champs »).

Exemple

Voici un exemple de modifications pour la table article.

```
function
prefixePlugin_rechercher_liste_des_jointures($tables){
    // rechercher en plus dans la BIO de l'auteur si on
cherche dans un article (oui c'est aberrant !)
    $tables['article']['auteur']['bio'] = 2;
    // rechercher aussi dans le texte des mots cles
    $tables['article']['mot']['texte'] = 2;
    // ne pas chercher dans les documents
    unset($tables['article']['document']);
    // retourner l'ensemble
    return $tables;
}
```

recuperer_fond

Le pipeline « recuperer_fond » permet de compléter ou modifier le résultat de la compilation d'un squelette donné. Il reçoit le nom du fond sélectionné et le contexte de compilation dans `args`, ainsi que le tableau décrivant le résultat dans `data`.

```
$page = pipeline('recuperer_fond', array(
    'args'=>array(
        'fond'=>$fond,
        'contexte'=>$contexte,
        'options'=>$options,
        'connect'=>$connect),
    'data'=>$page));
```

Bien souvent, seule la clé `texte` du tableau `data` sera modifiée. Se reporter à la fonction recuperer_fond() (p.122) pour obtenir une description de ce tableau.

Exemple

Le plugin « fblogin » permet de s'identifier en passant par Facebook. Il ajoute un bouton sur le formulaire d'identification habituel de SPIP. Le pipeline « social_login_links » (du même plugin) renvoie le code HTML d'un lien pointant sur l'identification de Facebook.

```
function fblogin_recuperer_fond($flux){
```

```
    if ($flux['args']['fond'] == 'formulaires/login'){
        $login = pipeline('social_login_links', '');
        $flux['data']['texte'] = str_replace('</form>',
'</form>' . $login, $flux['data']['texte']);
    }
    return $flux;
}
```

styliser

Ce pipeline permet de modifier la façon dont SPIP cherche les squelettes utilisés pour générer une page. Il est possible par exemple, d'aiguiller vers un squelette spécifique en fonction d'une rubrique donnée.

Ce pipeline est appelé comme suit :

```
// pipeline styliser
$squelette = pipeline('styliser', array(
    'args' => array(
        'id_rubrique' => $id_rubrique,
        'ext' => $ext,
        'fond' => $fond,
        'lang' => $lang,
        'connect' => $connect
    ),
    'data' => $squelette,
));
```

Il reçoit donc des arguments connus de l'environnement et retourne un nom de squelette qui sera utilisé pour la compilation. Ainsi, en appelant une page `spip.php?article18`, on recevrait les arguments

- id_rubrique = 4 (si l'article est dans la rubrique 4)
- ext = 'html' (extension par défaut des squelettes SPIP)
- fond = 'article' (nom du fond demandé)
- lang = 'fr'
- connect = '' (nom de la connexion SQL utilisée).

Exemple

Le plugin « SPIP Clear » utilise ce pipeline pour appeler des squelettes spécifiques sur les secteurs utilisés par ce moteur de blog :

```
// définir le squelette a utiliser si on est dans le cas
d'une rubrique de spipClear
function spipclear_styliser($flux){
    // si article ou rubrique
    if (($fond = $flux['args']['fond'])
    AND in_array($fond, array('article','rubrique'))) {

        $ext = $flux['args']['ext'];
        // [...]
        if ($id_rubrique = $flux['args']['id_rubrique'])
{
            // calcul du secteur
            $id_secteur = sql_getfetsel('id_secteur',
'spip_rubriques', 'id_rubrique=' . intval($id_rubrique));
            // comparaison du secteur avec la config de
SPIP Clear
            if (in_array($id_secteur,
lire_config('spipclear/secteurs', 1))) {
                // si un squelette $fond_spipclear existe
                if ($squelette =
test_squelette_spipclear($fond, $ext)) {
                    $flux['data'] = $squelette;
                }
            }
        }
    }
    return $flux;
}
// retourne un squelette $fond_spipclear.$ext s'il existe
function test_squelette_spipclear($fond, $ext) {
    if ($squelette =
find_in_path($fond."_spipclear.$ext")) {
        return substr($squelette, 0, -strlen(".$ext"));
    }
    return false;
}
```

Des fonctions à connaître

SPIP dispose de nombreuses fonctions fort utiles pour son fonctionnement. Certaines sont très utilisées et méritent quelques points d'explications.

Nom	Description
charger_fonction (p.120)	Trouver une fonction
find_all_in_path (p.121)	Trouver une liste de fichiers
find_in_path (p.121)	Trouver un fichier
include_spip (p.122)	Inclure une librairie PHP
recuperer_fond (p.122)	Retourner le résultat du calcul d'un squelette
spip_log (p.124)	Ajouter une information dans les logs
_request (p.125)	Récupérer une variable d'URL ou de formulaire

charger_fonction

Cette fonction `charger_fonction()` permet de récupérer le nom d'une fonction surchargeable de SPIP. Lorsqu'une fonction interne suffixée de `_dist()` est surchargée (en la recréant sans ce suffixe), ou lorsqu'on surcharge l'ensemble d'un fichier contenant une fonction de la sorte, il faut pouvoir récupérer la bonne fonction au moment de son exécution.

C'est cela que fait `charger_fonction()`. Elle retourne le nom de la fonction à exécuter.

```
$ma_fonction = charger_fonction('ma_fonction','repertoire');
$ma_fonction();
```

Principe de recherche

La fonction se comporte comme suit :

- elle retourne si la fonction est déjà déclarée `repertoire_ma_fonction`,
- sinon `repertoire_ma_fonction_dist`,
- sinon tente de charger un fichier `repertoire/ma_fonction.php` puis
- retourne `repertoire_ma_fonction` si existe,
- sinon `repertoire_ma_fonction_dist`,

- sinon renvoie `false`.

Exemple

Envoyer un mail :

```
$envoyer_mail = charger_fonction('envoyer_mail', 'inc');
$envoyer_mail($email, $sujet, $corps);
```

find_all_in_path

Cette fonction est capable de récupérer l'ensemble des fichiers répondant à un critère particulier de tous les répertoires connus des chemins de SPIP.

```
$liste_de_fichiers = find_all_in_path($dir, $pattern);
```

Exemple

SPIP utilise cette fonction pour récupérer l'ensemble des CSS que les plugins ajoutent à l'interface privée via les fichiers « prive/style_prive_plugin_*prefix*.html ». Pour cela SPIP récupère la liste de l'ensemble de ces fichiers en appelant :

```
$liste = find_all_in_path('prive/',
'/style_prive_plugin_');
```

find_in_path

La fonction `find_in_path()` permet de récupérer l'adresse d'un fichier dans le `path` de SPIP. Elle prend 1 à 2 arguments :

- nom ou adresse du fichier (avec son extension)
- éventuellement dossier de stockage

```
$f = find_in_path("dossiers/fichier.ext");
$f = find_in_path("fichier.ext","dossiers");
```

Exemple

Si un fichier `inclusions/inc-special.html` existe, récupérer le résultat de la compilation du squelette, sinon récupérer `inclusions/inc-normal.html`.

```
if (find_in_path("inclusions/inc-special.html")) {
    $html = recuperer_fond("inclusions/inc-special");
} else {
    $html = recuperer_fond("inclusions/inc-normal");
}
```

include_spip

La fonction `include_spip()` permet de charger une librairie PHP, un fichier. C'est l'équivalent du `include_once()` de PHP avec comme détail important le fait que le fichier demandé est recherché dans le `path` de SPIP, c'est à dire dans l'ensemble des dossiers connus, par ordre de priorité.

La fonction prend 1 a 2 arguments et retourne l'adresse du fichier trouvé :

- nom ou adresse du fichier (sans l'extension .php)
- inclure (true par défaut) : inclure le fichier ou seulement retourner son adresse ?

```
include_spip('fichier');
include_spip('dossier/fichier');
$adresse = include_spip('fichier');
$adresse = include_spip('fichier', true); // inclusion non
faite
```

Exemple

Charger le fichier de fonctions de mini présentations pour lancer la fonction `minipres` affichant une page d'erreur.

```
include_spip('inc/minipres');
echo minipres('Pas de chance !', 'Une erreur est survenue
!');
```

recuperer_fond

Autre fonction extrêmement importante de SPIP, `recuperer_fond()` permet de retourner le résultat du calcul d'un squelette donné. C'est en quelque sorte l'équivalent de `<INCLURE{fond=nom} />` des squelettes mais en PHP.

Elle prend 1 à 4 paramètres :

- nom et adresse du fond (sans extension)
- contexte de compilation (tableau clé/valeur)
- tableau d'options
- nom du fichier de connexion à la base de données à utiliser

Utilisation simple

Le retour est le code généré par le résultat de la compilation :

```
$code = recuperer_fond($nom, $contexte);
```

Utilisation avancée

L'option `raw` définie à `true` permet, plutôt que de récupérer simplement le code généré, d'obtenir un tableau d'éléments résultants de la compilation, dont le code (clé `texte`).

Que contient donc ce tableau ? Le texte, l'adresse de la source du squelette (dans « source »), le nom du fichier de cache PHP généré par la compilation (dans « squelette »), un indicateur de présence de PHP dans le fichier de cache généré (dans « process_ins »), divers autres valeurs dont le contexte de compilation (la langue et la date s'ajoutent automatiquement puisqu'on ne les avait pas transmises).

Exemple

Récupérer le contenu d'un fichier `/inclure/inc-liste-articles.html` en transmettant dans le contexte l'identifiant de la rubrique voulue :

```
$code = recuperer_fond("inclure/inc-liste-articles",
array(
    'id_rubrique' => $id_rubrique,
));
```

Option `raw` :
Voici un petit test avec un squelette « ki.html » contenant simplement le texte "hop". Ici, le résultat est envoyé dans un fichier de log (`tmp/test.log`).

```
$infos = recuperer_fond('ki',array(),array('raw'=>true));
spip_log($infos,'test');
```

Résultat dans `tmp/test.log` :

```
array (
  'texte' => 'hop
',
  'squelette' => 'html_1595b873738eb5964ecdf1955e8da3d2',
  'source' => 'sites/tipi.magraine.net/squelettes/
ki.html',
  'process_ins' => 'html',
  'invalideurs' =>
  array (
    'cache' => '',
  ),
  'entetes' =>
  array (
    'X-Spip-Cache' => 36000,
  ),
  'duree' => 0,
  'contexte' =>
  array (
    'lang' => 'en',
    'date' => '2009-01-05 14:10:03',
    'date_redac' => '2009-01-05 14:10:03',
  ),
)
```

spip_log

Cette fonction permet de stocker des actions dans les fichiers de logs (généralement placés dans le répertoire `tmp/log/`).

Cette fonction prend 1 ou 2 arguments. Un seul, elle écrira dans le fichier `spip.log`. Deux, elle écrira dans un fichier séparé et aussi dans `spip.log`.

```
<?php
spip_log($tableau);
spip_log($tableau, 'second_fichier');
spip_log("ajout de $champ dans $table","mon_plugin");
?>
```

Lorsqu'un tableau est transmis à la fonction de log, SPIP écrira le résultat d'un `print_r()` dans le fichier de log. Pour chaque fichier demandé, ici `spip` (par défaut) et `second_fichier`, SPIP créera ou ajoutera le contenu du premier argument, mais pas n'importe où. Si le script est dans l'interface privée, il écrira dans « prive_spip.log » ou « prive_second_fichier.log », sinon dans « spip.log » ou « second_fichier.log ».

Le fichier de configuration `ecrire/inc_version.php` définit la taille maximale des fichiers de log. Lorsqu'un fichier dépasse la taille souhaitée, il est copié sous un autre nom, par exemple `prive_spip.log.n` (`n` s'incrémentant). Ce nombre de fichiers copiés est aussi réglable. Il est aussi possible de désactiver les logs en mettant une de ces valeurs à zéro dans `mes_options.php`.

```
$GLOBALS['nombre_de_logs'] = 4; // 4 fichiers au plus
$GLOBALS['taille_des_logs'] = 100; // de 100ko au plus
```

Une constante `_MAX_LOG` (valant 100 par défaut) indique le nombre d'entrées que chaque appel d'une page peut écrire. Ainsi, après 100 appels de `spip_log()` par un même script, la fonction ne log plus.

_request

La fonction `_request()` permet de récupérer des variables envoyées par l'internaute, soit par l'URL, soit par un formulaire posté.

```
$nom = _request('nom');
```

Principes de sécurité

Ces fonctions ne doivent pas être placées n'importe où dans les fichiers de SPIP, ceci afin de connaître précisément les lieux possibles de tentatives de piratage. Les éléments issus de saisies utilisateurs ne devraient être récupérés que dans

- les fichiers d'actions (dans le répertoire `action/`),
- les fichiers d'affichage privé (dans le répertoire `exec/`),
- pour certaines très rares balises dynamiques (dans le répertoire `balise/`),
- ou dans les fichiers de formulaires (dans le répertoire `formulaires/`).

Il faut en règle générale en plus, vérifier que le type reçu est bien au format attendu (pour éviter tout risque de hack, bien que SPIP effectue déjà un premier nettoyage de ce qui est reçu), par exemple, si vous attendez un nombre, il faut appliquer la fonction `intval()` (qui transformera tout texte en valeur numérique) :

```
if ($identifiant = _request('identifiant')){
    $identifiant = intval($identifiant);
}
```

Récupérer dans un tableau

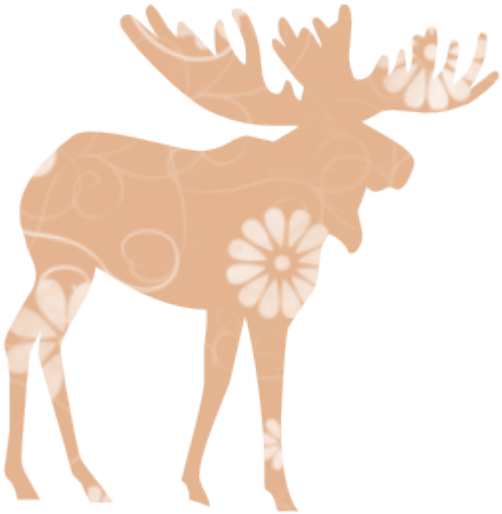
Si vous souhaitez récupérer uniquement parmi certaines valeurs présentes dans un tableau, vous pouvez passer ce tableau en second paramètre :

```
// recupere s'il existe $tableau['nom']
$nom = _request('nom', $tableau);
```

Exemple

Récupérer uniquement parmi les valeurs transmises dans l'URL :

```
$nom = _request('nom', $_GET);
```



Les différents répertoires

Ce chapitre va éclaircir le rôle des différents répertoires de SPIP. Pour certains, il abordera la manière de créer de nouveaux éléments dans ces répertoires.

Action

Le répertoire `action/` a pour but de gérer les actions affectant les contenus de la base de données. Ces actions sont donc la plupart du temps sécurisées.

Contenu d'un fichier action

Un fichier d'action comporte au moins une fonction à son nom. Un fichier `action/rire.php` devra donc déclarer une fonction `action_rire_dist()`.

```
<?php
if (!defined("_ECRIRE_INC_VERSION")) return;
function action_rire_dist(){
}
?>
```

Déroulement de la fonction

En général, la fonction principale est découpée en 2 parties : vérifications des autorisations, puis exécution des traitements demandés.

Les vérifications

Le bon auteur

La plupart des actions de SPIP vérifient uniquement que l'auteur en cours est bien le même que celui qui a cliqué l'action. Cela se fait avec :

```
$securiser_action = charger_fonction('securiser_action',
'inc');
$arg = $securiser_action();
```

La fonction de sécurité tue le script si l'auteur actuel n'est pas celui qui a demandé l'action, sinon elle renvoie l'argument demandé (ici dans `$arg`).

Le bon argument

Ensuite, généralement, la variable `$arg` reçue est vérifiée pour voir si elle est conforme à ce qu'on en attend. Elle prend souvent la forme de « id_objet », parfois « objet/id_objet » ou plus complexe comme ici des éléments de date :

```
if (!preg_match(",^\W*(\d+)\W(\w*)$,", $arg, $r)) {
    spip_log("action_dater_dist $arg pas compris");
    return;
}
```

Et l'autorisation

Certaines actions vérifient en plus que l'auteur a bien l'autorisation d'exécuter cette action (mais en général cette autorisation est déjà donnée en amont : le lien vers l'action n'apparaissant pas pour l'auteur n'en ayant pas les droits). Par exemple :

```
if (!autoriser('modererforum', 'article', $id_article))
    return;
// qui pourrait etre aussi :
if (!autoriser('modererforum', 'article', $id_article)) {
    include_spip('inc/minipres');
    minipres('Moderation',"Vous n'avez pas l'autorisation de
r&eacute;gler la mod&eacute;ration du forum de cet article");
    exit;
}
```

Les traitements

Lorsque toutes les vérifications sont correctes, des traitements sont effectués. Souvent, ces traitements appellent des fonctions présentes dans le même fichier, ou dans une librairie du répertoire `inc/`. Parfois l'action est simplement effectuée dans la fonction principale.

Exemple du réglage de la modération d'un article

```
// Modifier le reglage des forums publics de l'article x
function action_regler_moderation_dist()
{
    include_spip('inc/autoriser');
    $securiser_action = charger_fonction('securiser_action',
'inc');
    $arg = $securiser_action();
    if (!preg_match(",^\W*(\d+)$,", $arg, $r)) {
        spip_log("action_regler_moderation_dist $arg pas
compris");
        return;
```

```
    }
    $id_article = $r[1];
    if (!autoriser('modererforum', 'article', $id_article))
        return;
    // traitements
    $statut = _request('change_accepter_forum');
    sql_updateq("spip_articles", array("accepter_forum" =>
$statut), "id_article=". $id_article);
    if ($statut == 'abo') {
        ecrire_meta('accepter_visiteurs', 'oui');
    }
    include_spip('inc/invalideur');
    suivre_invalideur("id='id_forum/a$id_article'");
}
```

Les traitements effectués modifient la table `spip_articles` dans la base de données pour affecter un nouveau statut de gestion de forum. Lorsqu'un forum est demandé sur abonnement, c'est à dire qu'il faut être logé pour poster, il faut obligatoirement vérifier que le site accepte l'inscription de visiteurs, c'est ce que fait `ecrire_meta('accepter_visiteurs', 'oui');`.

Enfin, un appel à l'invalidation des fichiers du cache est effectué avec la fonction `suivre_invalideur()`. Tout le cache sera recréé (avant SPIP 2.0, cela n'invalidait qu'une partie du cache).

Redirections automatiques

À la fin d'une action, après le retour de la fonction, SPIP redirige la page sur une URL de redirection envoyée dans la variable `redirect`. Les fonctions pour générer les liens vers les actions sécurisées, comme `generer_action_auteur()` ont un paramètre pour recevoir ce lien de redirection.

Forcer une redirection

Certaines actions peuvent cependant forcer une redirection différente, ou définir une redirection par défaut. Pour cela, il faut appeler la fonction `redirige_par_entete()` qui permet de rediriger le navigateur sur une page différente.

Exemple :

Rediriger simplement vers l'URL de redirection prévue :

```
if ($redirect = _request('redirect')) {
    include_spip('inc/headers');
    redirige_par_entete($redirect);
}
```

Actions editer_objet

Les actions d'édition ont une petite particularité. Appelées par les formulaires d'édition des objets SPIP (dans le répertoire `prive/formulaires/`) depuis le fichier ecrire/inc/editer.php, elles ne reçoivent pas d'action de redirection et doivent retourner, dans ce cas là un un couple « identifiant », « erreur ». Le traitement du formulaire (CVT) gérant lui-même la redirection par la suite.

Pour cette raison, les fichiers `action/editer_xx.php` où xx est le type d'objet (au singulier) peuvent retourner un tableau :

```
if ($redirect) {
    include_spip('inc/headers');
    redirige_par_entete($redirect);
} else {
    return array($id_auteur,'');
}
```

Auth

Le répertoire `auth` contient les différents scripts pour gérer la connexion des utilisateurs. Deux sont fournis dans SPIP 2.0 par défaut : SPIP pour une connexion tout à fait normale, et LDAP pour une connexion des utilisateurs via cet annuaire.

Contenu d'un fichier auth

Les différentes authentifications sont appelées au moment du login dans le fichier prive/formulaires/login.php. La première qui valide une authentification permet de loger une personne en train de s'identifier.

La liste des différentes authentifications est décrite par une variable globale : `$GLOBALS['liste_des_authentifications']`.

Cependant, les processus d'authentifications sont relativement complexes faisant entrer de nombreuses sécurités. Aux fonctions de vérifications sont transmises le login et le mot de passe (crypté en md5 couplé à un nombre aléatoire - ou en clair dans le pire des cas lorsqu'il n'est pas possible de poser de cookies).

Fonction principale d'identification

Un fichier `auth/nom.php` doit posséder une fonction `auth_nom_dist()`. Cette fonction retourne un tableau décrivant l'auteur si celui-ci est authentifié.

```
if (!defined("_ECRIRE_INC_VERSION")) return;
// Authentifie et retourne la ligne SQL decrivant
l'utilisateur si ok
function auth_spip_dist ($login, $pass, $md5pass="",
$md5next="") {
...
}
```

Balise

Le répertoire balise stocke les déclarations des balises dynamiques et des balises génériques de SPIP.

Les balises dynamiques

Les balises dynamiques sont des balises qui sont calculées à chaque affichage de la page, contrairement aux balises statiques qui sont calculées uniquement lors du calcul de la page.

Ces balises dynamiques stockent donc dans le cache généré une portion de PHP qui sera exécuté à l'affichage. En principe, elles servent essentiellement pour afficher des formulaires.

Un fichier de balise dynamique peut comporter jusqu'à 3 fonctions essentielles : `balise_NOM_dist()`, `balise_NOM_stat()`, `balise_NOM_dyn()`.

Fonction balise_NOM_dist

La première fonction d'une balise dynamique est la même fonction utilisée pour les balises statiques, c'est à dire une fonction du nom de la balise : `balise_NOM_dist()`.

Cette fonction, au lieu d'insérer un code statique va appeler une fonction générant un code dynamique : `calculer_balise_dynamique()`.

En général, le contenu de la fonction se résume à l'appel du calcul dynamique, comme pour cet exemple de la balise `#LOGIN_PRIVE` :

```
function balise_LOGIN_PRIVE ($p) {
    return calculer_balise_dynamique($p, 'LOGIN_PRIVE',
array('url'));
}
```

La fonction de balise reçoit la variable `$p`, contenant les informations issues de l'analyse du squelette concernant la balise en question (arguments, filtres, à quelle boucle elle appartient, etc.).

La fonction `calculer_balise_dynamique` prend 3 arguments :

- le descriptif `$p`
- le nom de la balise dynamique à exécuter (en général, le même nom que la balise !)
- un tableau d'argument à récupérer du contexte de la page. Ici la balise dynamique demande à récupérer un paramètre `url` issu du contexte (boucle la plus proche ou environnement de compilation du squelette). Si l'on n'a pas de paramètre à récupérer du contexte, il faut donner un `array()` vide.

Fonction balise_NOM_stat()

Si elle existe, la fonction `balise_NOM_stat()` va permettre de calculer les arguments à transmettre à la fonction suivante (`_dyn()`). En son absence, seuls les arguments indiqués dans la fonction `calculer_balise_dynamique()` sont transmis (dans l'ordre du tableau). La fonction `stat`, va permettre de transmettre en plus des paramètres issus d'arguments ou des filtres transmis à la balise.

Le fonction reçoit 2 arguments : `$args` et `$filtres`.

- `$args` contient les arguments imposés par la fonction `calculer_balise_dynamique()`, cumulés avec les arguments transmis à la balise.
- `$filtres` contient la liste des filtres (`|filtre`) passés à la balise. Utilisé au temps ancien où SPIP utilisait des filtres pour passer des arguments (exemple qui n'est plus valable en SPIP 2.0 : `[(#LOGIN_PUBLIC|spip.php?article8)]` remplacé par `#LOGIN_PUBLIC{#URL_ARTICLE{8}}`)

Exemple

Reprenons l'exemple de `#LOGIN_PUBLIC` : elle fonctionne avec 1 ou 2 arguments : le premier est l'URL de redirection après s'être logé, le second est le login par défaut de la personne à loger. Les deux sont optionnels.

On peut donc passer à la balise un argument de redirection : `#LOGIN_PUBLIC{#SELF}` ou `#LOGIN_PUBLIC{#URL_ARTICLE{8}}`, mais en absence d'argument, on souhaite que la redirection soit faite sur un paramètre d'environnement `url` s'il existe. On avait demandé à récupérer cet argument, il se trouve dans `$args[0]`. `$args[1]` lui contient le contenu du premier argument donné à la balise (il s'ajoute dans le tableau `$args` après la liste des arguments automatiquement récupérés). Ceci donne :

```
function balise_LOGIN_PUBLIC_stat($args, $filtres) {
    return array(
        isset($args[1])
            ? $args[1]
            : $args[0],
        (isset($args[2])
            ? $args[2]
            : '')
    );
}
```

Si `$args[1]` est présent on le transmet, sinon `$args[0]`. De même si `$args[2]` est présent, on le transmet, sinon ''.

La fonction `_dyn()` recevra ces 2 arguments transmis :

```
function balise_LOGIN_PUBLIC_dyn($url, $login) {
...
}
```

Fonction balise_NOM_dyn()

Cette fonction permet d'exécuter les traitements à effectuer si un formulaire a été soumis. La fonction peut retourner une chaîne de caractère (qui sera affichée sur la page demandée) ou un tableau de paramètres qui indique le nom du squelette à récupérer et le contexte de compilation.

Les traitements

Pour 2 raisons je n'en parlerai pas :

- je n'ai toujours pas compris comment ça fonctionne,

- ce n'est plus très utile depuis que SPIP intègre un mécanisme plus simple appelé « formulaires CVT » (Charger, Vérifier, Traiter) qui s'appuie aussi sur cette fonction, mais de façon transparente.

L'affichage

Ce que retourne la fonction est alors affiché sur la page. Un tableau indique un squelette à appeler. Il se présente sous cette forme :

```
return array("adresse_du_squelette",
    3600, // duree du cache
    array( // contexte
        'id_article' => $id_article,
    )
);
```

Balises génériques

Un autre mécanisme malin de SPIP est la gestion des balises qu'on peut qualifier de génériques. En fait, il est possible d'utiliser une seule déclaration de balise pour tout un groupe de balises préfixées d'un nom identique.

Ainsi une balise `#PREFIXE_NOM` peut utiliser un fichier `balise/prefixe_.php` et déclarer une fonction `balise_PREFIXE__dist()` qui sera alors utilisée si aucune fonction `balise_PREFIXE_NOM_dist($p)` est présente.

La fonction générique, qui reçoit les attributs de la balise dans la variable `$p`, peut utiliser `$p->nom_champ` pour obtenir le nom de la balise demandée (ici "PREFIXE_NOM"). En analysant ce nom, on peut donc effectuer des actions adéquates.

Exemple

Cet exemple est utilisé par les balises génériques `#FORMULAIRE_NOM`, qui en plus sont des balises dynamiques (fichier ecrire/balise/formulaire_.php).

```
function balise_FORMULAIRE__dist($p) {
```

```
    preg_match(",^FORMULAIRE_(.*)?$,", $p->nom_champ,
$regs);
    $form = $regs[1];
    return
calculer_balise_dynamique($p,"FORMULAIRE_$form",array());
}
```

Récupérer objet et id_objet

Nous allons voir comment récupérer le type (objet) et l'identifiant d'une boucle pour s'en servir dans les calculs d'une balise.

Balise statique

Avec les paramètres de balise `$p`, il est très simple de récupérer `objet` et `id_objet` :

```
function balise_DEMO($p){
    // on prend nom de la cle primaire de l'objet pour
calculer sa valeur
    $_id_objet = $p->boucles[$p->id_boucle]->primary;
    $id_objet = champ_sql($_id_objet, $p);
    $objet = $p->boucles[$p->id_boucle]->id_table;
    $p->code = "calculer_balise_DEMO('$objet', $id_objet)";
    return $p;
}
function calculer_balise_DEMO($objet, $id_objet){
    $objet = objet_type($objet);
    return "Objet : $objet, id_objet : $id_objet";
}
```

Observons les deux fonctions. La première récupère dans la description de la balise le nom de sa boucle parente, le nom de la clé primaire, et demande à récupérer via la fonction `champ_sql()` la valeur de la clé primaire. Attention : ce que l'on récupère dans la variable `$id_objet` est un code qui doit être évalué en PHP (ce n'est pas une valeur numérique encore).

Une fois ces paramètres récupérés, on demande d'ajouter un code PHP à évaluer dans le code généré par la compilation du squelette (ce code sera mis en cache). C'est ce qu'on ajoute dans `$p->code`. Ce code là sera évalué par la suite au moment de la création du cache de la page appelée.

La fonction `calculer_balise_DEMO()` reçoit alors les deux arguments souhaités et retourne un texte qui les affiche sur la page.

```
<BOUCLE_a(ARTICLES){0,2}>
    #DEMO<br />
</BOUCLE_a>
    <hr />
<BOUCLE_r(RUBRIQUES){0,2}>
    #DEMO<br />
</BOUCLE_r>
```

Ce squelette permet alors de voir le résultat, la balise `#DEMO` reçoit des informations différentes en fonction du contexte dans lequel elle se trouve :

```
Objet : article, id_objet : 128
Objet : article, id_objet : 7
----
Objet : rubrique, id_objet : 1
Objet : rubrique, id_objet : 2
```

Balise dynamique

Dans le cas d'une balise dynamique, son fonctionnement même empêche de récupérer facilement le type et l'identifiant de la boucle dans laquelle elle est inscrite.

Lorsqu'on a néanmoins besoin de cela, par exemple pour créer des formulaires CVT qui adaptent leurs traitements en fonction du type de boucle, il faut envoyer à la fonction `_dyn()` (et par conséquent aux fonctions charger, vérifier et traiter de CVT) le type (objet) et l'identifiant de la boucle en cours.

L'appel à `calculer_balise_dynamique()` permet de récupérer des éléments du contexte de compilation. Si l'on demande à récupérer 'id_article', on l'obtiendra bien dans une boucle `ARTICLES`, mais pas dans une boucle `RUBRIQUES`. En étant plus précis, lorsqu'on demande une valeur 'id_article', SPIP fait comme s'il récupérait le résultat d'une balise `#ID_ARTICLE`, il cherche donc la valeur dans la boucle la plus proche, sinon dans le contexte, et aussi en fonction des balises qui ont été déclarées spécifiquement.

On peut demander à calculer `id_objet` facilement, mais `objet` va nécessiter de passer par une balise renvoyant la valeur de l'`objet`. Cette balise n'existant pas par défaut dans SPIP 2.0, il faut en créer une (`DEMODYN_OBJET`), ce qui donne :

```
function balise_DEMODYN($p){
    // cle primaire
    $_id_objet = $p->boucles[$p->id_boucle]->primary;
    return calculer_balise_dynamique(
        $p, 'DEMODYN', array('DEMODYN_OBJET', $_id_objet)
    );
}
function balise_DEMODYN_OBJET($p) {
    $objet = $p->boucles[$p->id_boucle]->id_table;
    $p->code = $objet ? objet_type($objet) :
"balise_hors_boucle";
    return $p;
}
function balise_DEMODYN_dyn($objet, $id_objet){
    return "Objet : $objet, id_objet : $id_objet";
}
```

Exec

Ce répertoire gère l'affichage des pages dans l'espace privé de SPIP.

Il y a deux manières différentes de mettre en place ces pages :

- Dans le répertoire **exec**, on peut les écrire en PHP.
- Dans le répertoire **prive/exec**, on peut les écrire en squelettes SPIP.

Contenu d'un fichier exec (squelette)

L'appel dans l'espace privé d'une page `?exec=nom` charge automatiquement un squelette placé dans `prive/exec/nom.html`.

Dans la majorité des cas, il est recommandé d'utiliser cette méthode plutôt qu'un fichier PHP. L'objectif est que l'espace privé de SPIP soit lui aussi écrit en squelette, donc plus facilement personnalisable. Il est ainsi possible d'utiliser des boucles, inclusions, balises, autorisations, comme dans tout squelette SPIP.

Exemple de squelette d'une page privée vide :

```
<h1>Une page priv&eacute;e directement en squelette</h1>
<p>Du contenu dans la page</p>
<!--#navigation-->
<div class='cadre-info'>
Une information dans une colonne de navigation.
</div>
<!--/#navigation-->
<!--#extra-->
Du contenu en plus dans la colonne extra.
<!--/#extra-->
```

Les encadrements `<!--#navigation-->` et `<!--#extra-->` servent à séparer les blocs principaux de la page. De manière automatique, l'espace privé de SPIP va déplacer chacun de ces blocs dans des balises HTML appropriées.

D'un point de vue technique, ces squelettes sont traités par le fichier ecrire/exec/fond.php. Automatiquement, les pipelines suivants sont ajoutés : affiche_gauche (p.95), affiche_droite (p.93) et affiche_milieu (p.97) en passant en paramètre le nom du paramètre `exec` :

```
echo pipeline('affiche_milieu', array('args' => array('exec'
=> $exec), 'data' => ''));
```

Aussi, le titre de la page privée est calculé en extrayant le contenu de la première balise HTML `<h1>` (ou `<hn>`) rencontrée.

Contenu d'un fichier exec (PHP)

En l'absence de squelette SPIP `prive/exec/nom.html`, l'appel dans l'espace privé d'une page `?exec=nom` charge une fonction `exec_nom_dist()` dans un fichier `exec/nom.php`.

Ces fonctions sont pour la plupart découpées de la même façon : l'appel à un début de page, la déclaration d'une colonne gauche, d'une colonne droite, d'un centre. Des pipelines sont présents pour que des plugins puissent ajouter des informations dans ces blocs.

Exemple de page vide « nom »

```
<?php
if (!defined("_ECRIRE_INC_VERSION")) return;
include_spip('inc/presentation');
function exec_nom_dist(){
    // si pas autorise : message d'erreur
    if (!autoriser('voir', 'nom')) {
        include_spip('inc/minipres');
        echo minipres();
        exit;
    }
    // pipeline d'initialisation
    pipeline('exec_init',
array('args'=>array('exec'=>'nom'),'data'=>''));
    // entetes
    $commencer_page = charger_fonction('commencer_page',
'inc');
    // titre, partie, sous_partie (pour le menu)
```

```
    echo $commencer_page(_T('plugin:titre_nom'), "editer",
"editer");

    // titre
    echo "<br /><br /><br />\n"; // outch ! aie aie aie ! au
secours !
    echo gros_titre(_T('plugin:titre_nom'),'', false);

    // colonne gauche
    echo debut_gauche('', true);
    echo pipeline('affiche_gauche',
array('args'=>array('exec'=>'nom'),'data'=>''));

    // colonne droite
    echo creer_colonne_droite('', true);
    echo pipeline('affiche_droite',
array('args'=>array('exec'=>'nom'),'data'=>''));

    // centre
    echo debut_droite('', true);
    // contenu
    // ...
    echo "afficher ici ce que l'on souhaite !";
    // ...
    // fin contenu
    echo pipeline('affiche_milieu',
array('args'=>array('exec'=>'nom'),'data'=>''));
    echo fin_gauche(), fin_page();
}
?>
```

Boite d'information

Pour ajouter une description de la page, ou une description de l'objet/id_objet en cours de lecture, un type d'encart est prévu : « boite_infos »

Il est souvent utilisé de la sorte, en ajoutant une fonction dans la colonne gauche :

```
// colonne gauche
echo debut_gauche('', true);
echo cadre_nom_infos();
```

```
echo pipeline('affiche_gauche',
array('args'=>array('exec'=>'nom'),'data'=>''));
```

Cette fonction appelle le pipeline et retourne son contenu dans une boite :

```
// afficher les informations de la page
function cadre_champs_extras_infos() {
    $boite = pipeline ('boite_infos', array('data' => '',
        'args' => array(
            'type'=>'nom',
            // eventuellement l'id de l'objet et la ligne SQL
            // $row = sql_fetsel('*', 'spip_nom',
'id_nom='.sql_quote($id_nom));
            'id' => $id_nom,
            'row' => $row,
        )
    ));
    if ($boite)
        return debut_boite_info(true) . $boite .
fin_boite_info(true);
}
```

Le pipeline charge automatiquement un squelette (avec le contexte fourni par le tableau `args`) homonyme au paramètre « type », dans le répertoire prive/infos/ soit `prive/infos/nom.html`. Il faut donc le créer avec le contenu souhaité.

Genie

Le génie gère les tâches périodiques, ce qu'on appelle généralement un `cron`.

Fonctionnement du cron

Les tâches à exécuter sont appelées à chaque consultation de page par un visiteur sur le site. Le passage d'un visiteur n'exécute qu'une seule tâche par page appelée, s'il y en a effectivement à traiter.

Cependant, pour que les tâches soient appelées, la balise `#SPIP_CRON` doit être présente dans le squelette de la page. Cette balise renvoie une image vide mais lance le script de tâches à traiter. Un navigateur texte lance aussi les tâches périodiques si la balise n'est pas présente.

Pour appeler le `cron`, il suffit d'exécuter la fonction `cron()`. Cette fonction peut prendre un argument indiquant le nombre de secondes qui doivent s'écouler avant qu'une autre tâche puisse être lancée, par défaut 60 secondes. Les appels par `#SPIP_CRON` sont mis à 2 secondes comme ceci :

```
cron(2);
```

Déclarer une tâche

Pour déclarer une tâche, il faut indiquer son nom et sa périodicité en secondes via le pipeline `taches_generales_cron` :

```
function monplugin_taches_generales_cron($taches){
    $taches['nom'] = 24*3600; // tous les jours
}
```

Cette tâche sera appelée au moment venu. Les traitements sont placés dans un fichier du répertoire `genie/`, homonyme à la tâche (`nom.php`) et disposant d'une fonction `genie_nom_dist()`.

La fonction reçoit en argument la date à laquelle s'est réalisé le dernier traitement de cette tâche. Elle doit retourner un nombre :

- nul, si la tâche n'a rien à faire
- positif, si la tâche a été traitée

- négatif, si la tâche a commencé, mais doit se poursuivre. Cela permet d'effectuer des tâches par lots (pour éviter des *timeout* sur les exécutions des scripts PHP à cause de traitements trop longs). Dans ce cas là, le nombre négatif indiqué correspond au nombre de secondes d'intervalle pour la prochaine exécution.

Exemple :

Cet exemple est simple, issu des tâches de « maintenance » de SPIP, dans le fichier `genie/maintenance.php`, puisqu'il exécute des fonctions et renvoie toujours `1`, indiquant que l'action a été réalisée.

```
// Diverses taches de maintenance
function genie_maintenance_dist ($t) {
    // (re)mettre .htaccess avec deny from all
    // dans les deux repertoires dits inaccessibles par http
    include_spip('inc/acces');
    verifier_htaccess(_DIR_ETC);
    verifier_htaccess(_DIR_TMP);
    // Verifier qu'aucune table n'est crashee
    if (!_request('reinstall'))
        verifier_crash_tables();
    return 1;
}
```

Gestion d'autorisations

Deux éléments essentiels permettent de gérer les accès aux actions et aux affichages des pages de SPIP : les autorisations, avec la fonction `autoriser()`, et les actions sécurisées par auteur, avec la fonction `securiser_action()`.

La librairie « autoriser »

SPIP dispose d'une fonction extensible `autoriser()` permettant de vérifier des autorisations. Cette fonction admet 5 arguments. Seul le premier est indispensable, les autres étant optionnels.

```
autoriser($faire, $type, $id, $qui, $opt);
```

La fonction renvoie `true` ou `false` en fonction de l'autorisation demandée et de l'auteur connecté (ou l'auteur demandé). Voici à quoi correspondent les différents arguments :

- `$faire` correspond à l'action demandée. Par exemple « modifier » ou « voir »,
- `$type` sert à donner généralement le type d'objet, par exemple « auteur » ou « article »,
- `$id` sert à donner l'identifiant de l'objet demandé, par exemple « 8 »,
- `$qui` permet de demander une autorisation pour un auteur particulier. Non renseigné, ce sera l'auteur connecté. On peut donner comme argument à $qui un `id_auteur`,
- `$opt` est un tableau d'option, généralement vide. Lorsqu'une autorisation nécessite de passer des arguments supplémentaires, ils sont mis dans ce tableau.

Exemple

```
if (autoriser('modifier','article',$id_article)) {
    // ... actions
}
```

La balise #AUTORISER

Une balise `#AUTORISER` permet de demander des autorisations dans un squelette. La présence de cette balise, comme la présence de la balise `#SESSION` crée un cache de squelette par visiteur identifié et un seul cache pour les visiteurs non identifiés.

Cette balise prend les mêmes arguments que la fonction `autoriser()`.

Exemple

```
[(#AUTORISER{modifier,article,#ID_ARTICLE})
    ... actions
]
```

Processus de la fonction autoriser()

Les autorisations par défaut de SPIP sont écrites dans le fichier ecrire/inc/autoriser.php

Lorsque l'on demande à SPIP une autorisation `autoriser($faire, $type)`, SPIP part à la recherche d'une fonction pour traiter l'autorisation demandée. Il recherche dans cet ordre une fonction nommée :

- `autoriser_$type_$faire`,
- `autoriser_$type`,
- `autoriser_$faire`,
- `autoriser_defaut`,
- puis la même chose avec le suffixe `_dist`.

Exemple

```
autoriser('modifier','article',$id_article);
```

Va retourner la première fonction trouvée et l'exécuter. C'est celle-ci :

```
function autoriser_article_modifier_dist($faire, $type,
$id, $qui, $opt){
...
}
```

La fonction reçoit les mêmes paramètres que la fonction `autoriser()`. Dedans, `$qui` est renseigné par l'auteur en cours s'il n'a pas été transmis en argument dans l'appel à `autoriser()`.

Créer ou surcharger des autorisations

Pour créer une autorisation, il suffit de créer les fonctions adéquates.

```
function autoriser_documentation_troller_dist($faire, $type,
$id, $qui, $opt) {
    return false; // aucun troll permis ! non mais !
}
```

Déclarer cette fonction permet d'utiliser la fonction `autoriser('troller','documentation')` ou la balise `#AUTORISER{troller, documentation}`.

Nouvelles fonctions, mais pas n'importe où !

La fonction `autoriser()`, à son premier appel, charge un pipeline du même nom. Cet appel du pipeline « autoriser » (p.102) permet de charger les fichiers d'autorisations pour un dossier squelettes ou un plugin.

Exemple

Dans un squelette :

Dans le fichier `config/mes_options.php` on ajoute l'appel d'une fonction pour nos autorisations :

```
<?php
$GLOBALS['spip_pipeline']['autoriser'] .=
"|mes_autorisations";
function mes_autorisations(){
    include_spip('inc/mes_autorisations');
}
?>
```

Ainsi lorsque le pipeline `autoriser` est appelé, il charge le fichier `inc/mes_autorisations.php`. On peut donc créer ce dossier et le fichier, qui contient les fonctions d'autorisations souhaitées, dans son dossier `squelettes/`.

Dans un plugin :
Pour un plugin, presque de la même façon, il faut déclarer l'utilisation du pipeline dans le `plugin.xml` :

```
<pipeline>
    <nom>autoriser</nom>
    <inclure>inc/prefixePlugin_autoriser.php</inclure>
</pipeline>
```

Et créer le fichier en question en ajoutant absolument, dans le fichier que le pipeline appelle, la fonction `prefixePlugin_autoriser()`.

```
<?php
if (!defined("_ECRIRE_INC_VERSION")) return;
// fonction pour le pipeline, n'a rien a effectuer
function prefixePlugin_autoriser(){}
// declarations d'autorisations
function autoriser_documentation_troller_dist($faire,
$type, $id, $qui, $opt) {
    return false; // aucun troll permis ! non mais !
}
?>
```

Les actions sécurisées

Les actions sécurisées sont un moyen d'être certain que l'action demandée provient bien de l'auteur qui a cliqué ou validé un formulaire.

La fonction `autoriser()` n'est pas suffisante pour cela. Par exemple, elle peut vérifier que tel type d'auteur (administrateur, rédacteur) a le droit d'effectuer telle genre d'action. Mais elle ne peut pas vérifier que telle action a effectivement été demandée par tel individu.

C'est en cela que les actions sécurisées interviennent. En fait, elles vont permettre de créer des URL, pour les liens ou pour les formulaires, qui transmettent une clé particulière. Cette clé est générée à partir de plusieurs informations : un nombre aléatoire regénéré à chaque connexion d'un auteur et stocké dans les données de l'auteur, l'identifiant de l'auteur, le nom de l'action et ses arguments.

Grâce à cette clé, lorsque l'auteur clique sur le lien où le formulaire, l'action appelée peut vérifier que c'est bien l'auteur actuellement connecté qui a demandé d'effectuer l'action (et pas un malicieux personnage à sa place !)

Fonctionnement des actions sécurisées

L'utilisation d'actions sécurisées se passe en deux temps. Il faut d'abord générer un lien avec la clé de sécurité, puis lorsque l'utilisateur clique sur l'action, qui va exécuter une fonction d'un fichier dans le répertoire `action/`, il faut vérifier la clé.

La fonction securiser_action()

Cette fonction `securiser_action`, dans le fichier ecrire/inc/securiser_action.php, crée ou vérifie une action. Lors d'une création, en fonction de l'argument `$mode`, elle créera une URL, un formulaire, ou retournera simplement un tableau avec les paramètres demandés et la clé générée. Lors d'une vérification, elle compare les éléments soumis par GET (URL) ou POST (formulaire) et tue le script avec un message d'erreur et `exit` si la clé ne correspond pas à l'auteur actuel.

Générer une clé

Pour générer une clé, il faut appeler la fonction avec les paramètres corrects :

```
$securiser_action =
charger_fonction('securiser_action','inc');
$securiser_action($action, $arg, $redirect, $mode);
```

Ces quatre paramètres sont les principaux utilisés :

- `$action` est le nom du fichier d'action et de l'action correspondante (`action/nom.php` et fonction associée `action_nom_dist()`)
- `$arg` est un argument transmis, par exemple `supprimer/article/3` qui servira entre autre à générer la clé de sécurité.
- `$redirect` est une URL sur laquelle se rendre une fois l'action réalisée.
- `$mode` indique ce qui doit être retourné :
  - `false` : une URL
  - `-1` : un tableau des paramètres
  - un contenu texte : un formulaire à soumettre (le contenu est alors ajouté dans le formulaire)

Dans une action, vérifier et récupérer l'argument

Dans une fonction d'action (`action_nom_dist()`), on vérifie la sécurité en appelant la fonction sans argument. Elle retourne l'argument (sinon affiche une erreur et tue le script) :

```
$securiser_action =
charger_fonction('securiser_action','inc');
$arg = $securiser_action();
// a partir d'ici, nous savons que l'auteur est bien le bon !
```

Fonctions prédéfinies d'actions sécurisées

Les actions sécurisées sont rarement générées directement en appelant la fonction `securiser_action()`, mais plus souvent en appelant une fonction qui elle, appelle la fonction de sécurisation.

Le fichier ecrire/inc/actions.php contient une grande partie de ces fonctions.

generer_action_auteur()

Particulièrement, la fonction `generer_action_auteur()` appelle directement la fonction `securiser_action` en renvoyant une URL sécurisée par défaut.

redirige_action_auteur()

Cette fonction admet à la place du 3e argument de redirection, 2 paramètres : le nom d'un fichier exec, et les arguments à transmettre. SPIP crée alors l'url de redirection automatiquement.

redirige_action_post()

Identique à la fonction précédente sauf qu'elle génère un formulaire POST par défaut.

Exemple

Générer un lien pour changer les préférences d'affichage dans l'interface privée :

```
$url = generer_action_auteur('preferer',"display:1",
$self);
```

Lancer une action sur l'édition d'une brève, puis rediriger sur la vue de la brève.

```
$href =
redirige_action_auteur('editer_breve',$id_breve,'breves_voir',
"id_breve=$id_breve");
```

Poster un formulaire puis rediriger sur la page « admin_plugin ». `$corps` contient le contenu du formulaire pour activer un plugin.

```
echo redirige_action_post('activer_plugins','activer',
'admin_plugin','', $corps);
```

URL d'action en squelette

Une balise `#URL_ACTION_AUTEUR` permet de générer des URL d'actions sécurisées depuis un squelette.

```
#URL_ACTION_AUTEUR{action,argument,redirection}
```

Exemple

Supprimer le commentaire de forum demandé si l'auteur en a le droit bien sûr (`autoriser('configurer')` est bien vague, mais c'est celle appliquée dans le privé dans ecrire/exec/forum_admin.php]) !

```
[(#AUTORISER{configurer})
<a href="#URL_ACTION_AUTEUR{instituer_forum,#ID_FORUM-
off,#URL_ARTICLE}"><:supprimer:></a>
]
```

Compilation des squelettes

Cette partie explique quelques détails de la transformation d'un squelette par le compilateur.

La syntaxe des squelettes

SPIP 2.0 possède une syntaxe pour écrire des squelettes construite avec un vocabulaire réduit, mais extrêmement riche et modulaire. Cette syntaxe, définie dans le fichier ecrire/public/phraser_html.php contient des éléments tel que :

- la boucle

```
<B_nom>
... avant
<BOUCLE_nom(TABLE){criteres}>
... pour chaque element
</BOUCLE_nom>
... apres
</B_nom>
... sinon
<//B_nom>
```

- le champ (balise)

```
[ avant (#BALISE{criteres}|filtres) apres ]
```

- l'argument (`{args}`, `|filtre` ou `|filtre{args}` sur les balises)
- le critère (`{critere=param}` sur les boucles)
- l'inclusion

```
<INCLURE{fond=nom}>
```

- l'idiome (chaîne de langue)

```
<:type:chaine_langue:>
```

- le polyglotte (`<multi>` utilisé dans un squelette)

```
<multi>[fr]français[en]English</multi>
```

L'analyse du squelette

Lorsque le phraseur (le parseur) de SPIP analyse un squelette, il traduit la syntaxe en un vocabulaire connu et compris du compilateur. On peut donc dire que le phraseur traduit une langue particulière (la syntaxe SPIP 2.0) qu'on nomme « syntaxe concrète » en une langue précise qu'on nomme « syntaxe abstraite ». Elle est définie par des objets PHP dans le fichier ecrire/puclic/interfaces.php

De cette analyse de la page, le phraseur crée un tableau la décrivant, séquentiellement et récursivement, en utilisant le vocabulaire compris du compilateur (les objets Texte, Champ, Boucle, Critere, Idiome, Inclure, Polyglotte).

Pour bien comprendre, observons quel tableau est généré par des petits exemples de squelettes.

Un texte

Squelette :

```
Texte simple
```

Tableau généré : (issu d'un `print_r`)

```
array (
  0 =>
  Texte::__set_state(array(
     'type' => 'texte',
     'texte' => 'Texte simple
',
     'avant' => NULL,
     'apres' => '',
     'ligne' => 1,
  )),
)
```

Le tableau indique que la premier élément lu sur la page (clé 0) est un élément « Texte », débutant sur la ligne 1, et possédant le texte "Texte simple".

Une balise

Squelette :

```
[avant(#VAL)après]
```

On peut comprendre du tableau ci-dessous, que le premier élément lu de la page est un Champ (une balise), que son nom est « VAL », qu'il n'est pas dans une boucle (sinon id_boucle serait défini), que ce qui est dans la partie optionnelle avant la balise est un élément « Texte » dont le texte est « avant ».

Tableau généré :

```
array (
  0 =>
  Champ::__set_state(array(
     'type' => 'champ',
     'nom_champ' => 'VAL',
     'nom_boucle' => '',
     'avant' =>
    array (
      0 =>
      Texte::__set_state(array(
         'type' => 'texte',
         'texte' => 'avant',
         'avant' => NULL,
         'apres' => '',
         'ligne' => 1,
      )),
    ),
     'apres' =>
    array (
      0 =>
      Texte::__set_state(array(
         'type' => 'texte',
         'texte' => 'après',
         'avant' => NULL,
         'apres' => '',
         'ligne' => 1,
      )),
    ),
     'etoile' => '',
     'param' =>
    array (
    ),
     'fonctions' =>
    array (
```

```
    ),
     'id_boucle' => NULL,
     'boucles' => NULL,
     'type_requete' => NULL,
     'code' => NULL,
     'interdire_scripts' => true,
     'descr' =>
    array (
    ),
     'ligne' => 1,
  )),
  1 =>
  Texte::__set_state(array(
     'type' => 'texte',
     'texte' => '
',
     'avant' => NULL,
     'apres' => '',
     'ligne' => 1,
  )),
)
```

Une boucle

Prenons un dernier exemple d'une boucle avec une balise, plus compliqué car il induit une référence circulaire dans le tableau généré. Observons :

Squelette :

```
<BOUCLE_a(ARTICLES){id_article=3}>
#TITRE
</BOUCLE_a>
```

Cette boucle sélectionne l'article 3 et devrait afficher le titre de l'article. Le tableau de la page si on tente de l'afficher finit par générer une erreur de récursion. L'observation montre que le second élément lu dans la boucle est un Champ (balise) nommé "TITRE". Ce champ contient une référence vers la boucle dans laquelle il est (`'boucles'=>array(...)`). Cette boucle contenant la balise qui appartient à la boucle contenant la balise qui appartient à ...

Tableau généré partiel

```
array (
  0 =>
  Boucle::__set_state(array(
     'type' => 'boucle',
     'id_boucle' => '_a',
     'id_parent' => '',
     'avant' =>
    array (
    ),
     'milieu' =>
    array (
      0 =>
      Texte::__set_state(array(
         'type' => 'texte',
         'texte' => '
',
         'avant' => NULL,
         'apres' => '',
         'ligne' => 1,
      )),
      1 =>
      Champ::__set_state(array(
         'type' => 'champ',
         'nom_champ' => 'TITRE',
         'nom_boucle' => '',
         'avant' => NULL,
         'apres' => NULL,
         'etoile' => '',
         'param' =>
        array (
        ),
         'fonctions' =>
        array (
        ),
         'id_boucle' => '_a',
         'boucles' =>
        array (
          '_a' =>
          Boucle::__set_state(array(
             'type' => 'boucle',
             'id_boucle' => '_a',
             'id_parent' => '',
             'avant' =>
            array (
            ),
```

```
             'milieu' =>
            array (
              0 =>
              Texte::__set_state(array(
                 'type' => 'texte',
                 'texte' => '
',
                 'avant' => NULL,
                 'apres' => '',
                 'ligne' => 1,
              )),
              1 =>
              Champ::__set_state(array(
                 'type' => 'champ',
                 'nom_champ' => 'TITRE',
                 'nom_boucle' => '',
                 'avant' => NULL,
                 'apres' => NULL,
                 'etoile' => '',
                 'param' =>
                array (
                ),
                 'fonctions' =>
                array (
                ),
                 'id_boucle' => '_a',
                 'boucles' =>
                array (
                  '_a' =>
                  Boucle::__set_state(array(
...
```

Pourquoi de telles références ?

Tout simplement parce qu'elles servent ensuite dans le calcul des balises. Lorsqu'une balise est calculée, une partie de ce tableau lui est passée en paramètre (le fameux `$p` que l'on recroisera). Cette partie concerne simplement les informations de la balise. Pour obtenir des informations de sa boucle englobante, il suffit, grâce à cette référence, d'appeler le paramètre `$p->boucles[$p->id_boucle]`.

Processus d'assemblage

La production d'une page par le compilateur se réalise dans le fichier ecrire/public/assembler.php.

Ce fichier appelle des fonctions pour analyser ce qui est demandé, récupérer le squelette adapté, le traduire en PHP, retourner le résultat de l'évaluation du code PHP. Le tout en gérant des caches.

SPIP utilise généralement la fonction `recuperer_fond()` pour récupérer le résultat d'un squelette mais il appelle aussi directement la fonction `assembler()` depuis le fichier `ecrire/public.php`.

Cascade d'appels

La fonction `recuperer_fond()` appelle `evaluer_fond()` qui appelle `inclure_page()` qui appelle la fonction `cacher()` du fichier ecrire/public/cacher.php. C'est cette même fonction `cacher()` qu'appelle aussi `assembler()`.

Déterminer le cache

Le fichier ecrire/public/cacher.php permet de gérer les fichiers du cache.

La fonction `cacher()` récupère le nom et la date d'une page en cache si elle existe, en fonction du contexte qui lui est donné. Si l'on transmet en plus une adresse de fichier, alors le fichier cache est créé.

Ainsi, cette fonction peut-être appelée 2 fois :

- la première fois pour déterminer le nom du fichier de cache et pour indiquer si un cache valide existe pour la page demandée.
- Une seconde fois lorsqu'il n'y a pas de cache valide. La page est alors calculée par la fonction `parametrer()`, puis la fonction `cacher()` est alors appelée pour stocker cette fois le résultat en cache.

```
// Cette fonction est utilisee deux fois
$cacher = charger_fonction('cacher', 'public');
// Les quatre derniers parametres sont modifes par la
fonction:
// emplacement, validite, et, s'il est valide, contenu & age
```

```
$res = $cacher($GLOBALS['contexte'], $use_cache,
$chemin_cache, $page, $lastmodified);
```

Paramètres déterminant le nom du squelette

Le fichier ecrire/public/parametrer.php permet de créer des paramètres qui seront nécessaires pour récupérer le nom et les informations du squelette à compiler via `styliser()` puis demander son calcul via `composer()`.

Ainsi la fonction `parametrer()` calcule la langue demandée ainsi que le numéro de la rubrique en cours si cela est possible.

Ces paramètres permettent alors de trouver le nom et l'adresse du squelette correspondant à la page demandée. Cela est fait en appelant la fonction `styliser()` qui reçoit les arguments en question.

Déterminer le fichier de squelette

Le fichier ecrire/public/styliser.php détermine le nom et le type de squelette en fonction des arguments qui lui sont transmis.

```
$styliser = charger_fonction('styliser', 'public');
list($skel,$mime_type, $gram, $sourcefile) =
    $styliser($fond, $id_rubrique_fond,
$GLOBALS['spip_lang'], $connect);
```

Un 5e argument permet de demander un phraseur (une syntaxe concrète) et par conséquent une extension des fichiers de squelettes différents. Par défaut, le phraseur, donc l'extension utilisée, est `html`.

La fonction cherche un squelette nommé `$fond.$ext` dans le *path* de SPIP. S'il n'existe pas, elle renvoie une erreur, sinon elle tente de trouver un squelette plus spécifique dans le même répertoire que le squelette trouvé, en fonction des paramètres d'`id_rubrique` et `lang`.

Styliser cherche alors des fichiers comme `nom=8.html`, `nom-8.html`, `nom-8.en.html` ou `nom.en.html` dans l'ordre :

- `$fond=$id_rubrique`

- `$fond-$id_rubrique`
- `$fond-$id_rubrique_parent_recursivement`
- puis ce qu'il a trouvé (ou non) complété de `.$lang`

La fonction retourne alors un tableau d'éléments de ce qu'elle a trouvé `array($squelette, $ext, $ext, "$squelette.$ext")` :

- 1er paramètre : le nom du squelette,
- 2e : son extension
- 3e : sa grammaire (le type de phraseur)
- 4e : le nom complet.

Ces paramètres servent au composeur et sa fonction `composer()`.

Une belle composition

Le fichier ecrire/public/composer.php a pour but de récupérer le squelette traduit en PHP et de l'exécuter avec le contexte demandé.

Si le squelette a déjà été traduit en PHP, le résultat est récupéré d'un fichier cache et utilisé, sinon SPIP appelle sa fonction de compilation `compiler()` pour traduire la syntaxe concrète en syntaxe abstraite puis en code exécutable par PHP.

Le fichier `composer.php` charge aussi les fonctions nécessaires à l'exécution des fichiers PHP issus de la compilation des squelettes.

La compilation

Le compilateur de SPIP, dans le fichier ecrire/public/compiler.php est appelé avec la fonction `compiler()` depuis la fonction `parametrer()`.

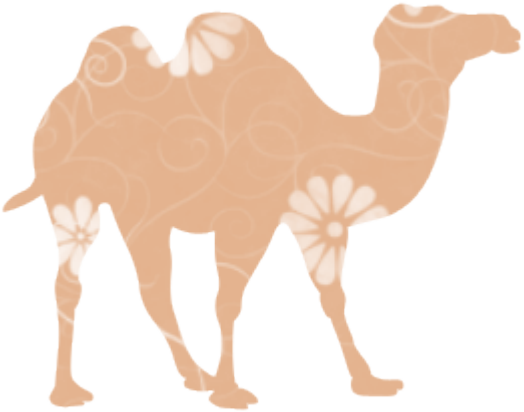
La compilation commence par appeler le phraseur approprié en fonction de la grammaire demandée (la syntaxe concrète du squelette). C'est donc le phraseur `phraser_html()` qui est appelé, dans le fichier ecrire/public/phraser_html.php. Il transforme la syntaxe du squelette en un tableau (`$boucles`) de listes d'objets PHP formant la syntaxe concrète que va analyser la fonction de compilation.

Pour chaque boucle trouvée, SPIP effectue un certain nombre de traitements en commençant par retrouver à quelles tables SQL elle correspond et quelles jointures sont déclarées pour ces tables.

Il calcule ensuite les critères appliqués sur les boucles (déclarés dans ecrire/public/criteres.php ou par des plugins), puis le contenu des boucles (dont les balises définies pour certaines dans ecrire/public/balises.php). Il calcule ensuite les éléments d'un squelette extérieur à une boucle.

Il exécute enfin les fonctions de boucles qui sont déclarées dans le fichier ecrire/public/boucles.php. Le résultat de tout cela construit un code PHP exécutable avec une fonction PHP par boucle, et une fonction PHP générale pour le squelette.

C'est ce code exécutable que retourne le compilateur. Ce code sera mis en cache puis exécuté par le composeur avec les paramètres de contexte transmis. Le résultat est le code de la page demandé, qui sera mis en cache (par appel de la fonction `cacher()` une seconde fois, dans le fichier `assembler.php`) puis qui sera envoyé au navigateur (ou si c'est une inclusion, ajouté à un fragment de page). Il peut encore contenir du PHP lorsque certaines informations doivent s'afficher en fonction du visiteur, comme les formulaires dynamiques.



Formulaires

SPIP dispose d'un mécanisme assez simple et puissant pour gérer les formulaires, dit CVT (Charger, Vérifier, Traiter) qui décompose un formulaire en 4 parties :

- une vue, qui est un squelette SPIP affichant le code HTML du formulaire, correspondant au fichier `formulaires/{nom}.html`,
- et 3 fonctions PHP pour charger les variables du formulaire, vérifier les éléments soumis et enfin traiter le formulaire, déclarées dans le fichier `formulaires/{nom}.php`.

Structure HTML

Les formulaires sont stockés dans le dossier `formulaires/`. Pour faciliter la réutilisation et la personnalisation graphique des formulaires, une syntaxe HTML est proposée.

Afficher le formulaire

Un fichier `formulaires/joli.html` s'appelle dans un squelette par `#FORMULAIRE_JOLI` qui affiche alors le formulaire.

Le HTML du formulaire suit une écriture standard pour tous les formulaires SPIP. Les champs du formulaire sont encadrés dans une liste d'éléments `ul/li`.

```
<div class="formulaire_spip formulaire_demo">
<form action="#ENV{action}" method="post"><div>
    #ACTION_FORMULAIRE{#ENV{action}}
    <ul>
        <li class="editer_la_demo obligatoire">
            <label for="la_demo">La demo</label>
            <input type='text' name='la_demo' id='la_demo'
value="#ENV{la_demo}" class="text" />
        </li>
    </ul>
    <p class="boutons"><input type="submit" class="submit"
value="<:pass_ok:>" /></p>
</div></form>
</div>
```

Pour le bon fonctionnement du formulaire, l'attribut `action` doit être renseigné par la variable `#ENV{action}` calculée automatiquement par SPIP. De même, la balise `#ACTION_FORMULAIRE{#ENV{action}}` doit être présente, elle calcule et ajoute des clés de sécurité qui seront vérifiées automatiquement à la réception du formulaire.

Quelques remarques :

- Les balises `<li>` reçoivent des classes CSS `editer_xx` où xx est le nom du champ, et éventuellement `obligatoire` pour indiquer (visuellement) que ce champ est obligatoirement à remplir.

- Les balises `input` ont une classe CSS nommée comme leur type (pour pallier à une déficience d'Internet Explorer en CSS qui ne comprenait pas `input[type=text]`)
- Les boutons de soumission sont encadrés d'une classe CSS `boutons`

Gerer le retour d'erreurs

La fonction `verifier()` du formulaire peut retourner des erreurs si les champs soumis ne sont pas corrects ; nous le verrons plus tard. Pour afficher ces erreurs dans le HTML du formulaire, des classes CSS et un nommage sont proposés :

En tête du formulaire, des erreurs (ou des messages de réussite) généraux :

```
[<p class="reponse_formulaire
reponse_formulaire_erreur">(#ENV*{message_erreur})</p>]
[<p class="reponse_formulaire
reponse_formulaire_ok">(#ENV*{message_ok})</p>]
```

Pour chaque champ, un message et une classe CSS sur l'item de liste pour marquer visuellement l'erreur. On calcule le message du champ grâce à la variable `#ENV{erreurs}` qui recense toutes les erreurs des champs :

```
#SET{erreurs,#ENV**{erreurs}|table_valeur{xxx}}
<li class="editer_xxx obligatoire[
(#GET{erreurs}|oui)erreur]">
    [<span class='erreur_message'>(#GET{erreurs})</span>]
</li>
```

Ceci donne, au complet avec le formulaire précédent :

```
<div class="formulaire_spip formulaire_demo">
[<p class="reponse_formulaire
reponse_formulaire_erreur">(#ENV*{message_erreur})</p>]
[<p class="reponse_formulaire
reponse_formulaire_ok">(#ENV*{message_ok})</p>]
<form action="#ENV{action}" method="post"><div>
    #ACTION_FORMULAIRE{#ENV{action}}
    <ul>
        #SET{erreurs,#ENV**{erreurs}|table_valeur{la_demo}}
```

```
        <li class="editer_la_demo obligatoire[
(#GET{erreurs}|oui)erreur]">
            <label for="la_demo">La demo</label>
            [<span
class='erreur_message'>(#GET{erreurs})</span>]
            <input type='text' name='la_demo' id='la_demo'
value="#ENV{la_demo}" />
        </li>
    </ul>
    <p class="boutons"><input type="submit" class="submit"
value="<:pass_ok:>" /></p>
</div></form>
```

Séparation par fieldset

Lorsqu'un formulaire possède de nombreux champs, on le divise généralement en différents blocs nommés `fieldset` en HTML.

Comme précédemment, une écriture est proposée pour de tels blocs, toujours s'appuyant sur des listes `ul/li` :

```
[...]
<form method="post" action="#ENV{action}"><div>
#ACTION_FORMULAIRE{#ENV{action}}
<ul>
    <li class="fieldset">
        <fieldset>
            <h3 class="legend">Partie A</h3>
            <ul>
                <li> ... </li>
                <li> ... </li>
                ...
            </ul>
        </fieldset>
    </li>
    <li class="fieldset">
        <fieldset>
            <h3 class="legend">Partie B</h3>
            <ul>
                <li> ... </li>
                <li> ... </li>
                ...
```

```
            </ul>
        </fieldset>
    </li>
</ul>
    <p class="boutons"><input type="submit" class="submit"
value="<:pass_ok:>" /></p>
</div></form>
```

Il y a donc deux listes imbriquées ici, le premier `<li>` possédant la classe CSS "fieldset". En lieu et place des balises HTML `<legend>` est proposée une écriture `<h3 class="legend">` qui offre plus de possibilités de décoration en CSS.

Champs radio et checkbox

Pour afficher des listes d'éléments de type radio ou checkbox, une syntaxe est proposée en encadrant les éléments d'une `<div class="choix"></div>`. Cette écriture permet d'avoir le bouton avant le label, d'avoir la liste radio en horizontal (via CSS).

```
<li class="editer_syndication">
    <div class="choix">
        <input type='radio' class="radio" name='syndication'
value='non' id='syndication_non'[
(#ENV{syndication}|=={non}|oui)checked="checked"] />
        <label
for='syndication_non'><:bouton_radio_non_syndication:></label>
    </div>
    <div class="choix">
        <input type='radio' class="radio" name='syndication'
value='oui' id='syndication_oui'[
(#ENV{syndication}|=={oui}|oui)checked="checked"] />
        <label
for='syndication_oui'><:bouton_radio_syndication:></label>
    </div>
</li>
```

Passer la liste en horizontal :

Cela se réalise en CSS pour cet exemple par :

```
.formulaire_spip .editer_syndication .choix {display:inline;}
```

Expliquer les saisies

Il est souvent nécessaire de donner une explication pour remplir correctement une saisie de formulaire. SPIP propose une écriture pour cela, à insérer comme classe CSS dans une balise <p> ou :

- **explication** (avec <p>) permet d'écrire une explication plus détaillée que le label du champ souhaité
- **attention** (avec) met en exergue un descriptif proposé. À utiliser avec modération !

Ces deux descriptions complètent donc les autres options déjà citées `erreur` et `obligatoire`.

Exemple

```
#SET{erreurs,#ENV**{erreurs}|table_valeur{nom}}
<li class="editer_nom obligatoire[
(#GET{erreurs}|oui)erreur]>
    <label
for="nom"><:titre_cadre_signature_obligatoire:></label>
    [<span class='erreur_message'>(#GET{erreurs})</span>]
    <p class='explication'><:entree_nom_pseudo:></p>
    <input type='text' class='text' name='nom' id='nom'
value="[(#ENV**{nom})]" />
</li>
```

Affichage conditionnel

Les fonctions `charger()` ou `traiter()` peuvent indiquer dans leur réponse que le formulaire est éditable ou non. Cela se traduit par la réception d'un paramètre `editable` dans le squelette, qui peut servir à masquer ou non le formulaire (mais pas les messages d'erreur ou de réussite).

Il s'utilise comme ceci `[(#ENV{editable}) ... contenu de <form> ... ]` :

```
<div class="formulaire_spip formulaire_demo">
    [<p class="reponse_formulaire
reponse_formulaire_ok">(#ENV*{message_ok})</p>]
    [<p class="reponse_formulaire
reponse_formulaire_erreur">(#ENV*{message_erreur})</p>]
    [(#ENV{editable})
        <form method='post' action='#ENV{action}'><div>
        #ACTION_FORMULAIRE{#ENV{action}}
        <ul>
        ...
        </ul>
        <p class='boutons'><input type='submit'
class='submit' value='<:bouton_enregistrer:>' /></p>
        </div></form>
    ]
</div>
```

En cas de boucles dans le formulaire

Si une boucle SPIP est présente à l'intérieur de l'écriture `[(#ENV{editable}) ... ]` (ou tout autre balise), le compilateur SPIP renvoie une erreur (ou n'affiche pas correctement la page) car cela n'est pas prévu par le langage actuel de squelettes.

Pour pallier à cela, il faut :

- soit mettre la boucle dans une inclusion appelée alors par `<INCLURE{fond=mon/inclusion} />`
- soit utiliser le plugin Bonux et sa boucle CONDITION comme ceci :

```
<div class="formulaire_spip formulaire_demo">
    [<p class="reponse_formulaire
reponse_formulaire_ok">(#ENV*{message_ok})</p>]
    [<p class="reponse_formulaire
reponse_formulaire_erreur">(#ENV*{message_erreur})</p>]
    <BOUCLE_editable(CONDITION){si #ENV{editable}}>
        <form method='post' action='#ENV{action}'><div>
        #ACTION_FORMULAIRE{#ENV{action}}
        <ul>
        ...
        </ul>
```

```
        <p class='boutons'><input type='submit'
class='submit' value='<:bouton_enregistrer:>' /></p>
        </div></form>
    </BOUCLE_editable>
</div>
```

Traitements PHP

Les fichiers `formulaires/{nom}.php` contiennent les trois fonctions essentielles des formulaires CVT de SPIP :

- `formulaires_{nom}_charger_dist`,
- `formulaires_{nom}_verifier_dist` et
- `formulaires_{nom}_traiter_dist`.

Passage d'arguments aux fonctions CVT

Les fonctions `charger()`, `verifier()` et `traiter()` ne reçoivent par défaut aucun paramètre.

```
function formulaires_x_charger_dist(){…}
function formulaires_x_verifier_dist(){…}
function formulaires_x_traiter_dist(){…}
```

Pour que les fonctions reçoivent des paramètres, il faut soumettre les arguments explicitement dans l'appel de formulaire.

```
#FORMULAIRE_X{argument, argument, …}
```

Les fonctions PHP reçoivent les paramètres dans le même ordre :

```
function formulaires_x_charger_dist($arg1, $arg2, …){…}
function formulaires_x_verifier_dist($arg1, $arg2, …){…}
function formulaires_x_traiter_dist($arg1, $arg2, …){…}
```

À noter qu'une possibilité complémentaire en utilisant les fonctions des balises dynamiques permet de transmettre automatiquement des paramètres.

Exemple

Le plugin « Composition » dispose d'un formulaire qui nécessite un type et un identifiant. Il est appelé comme cela :

```
[(#FORMULAIRE_EDITER_COMPOSITION_OBJET{#ENV{type},
#ENV{id}})]
```

Les fonctions de traitement reçoivent donc ces deux paramètres :

```
function
formulaires_editer_composition_objet_charger($type,
$id){…}
```

Charger les valeurs du formulaire

La fonction `charger()` permet d'indiquer quels champs doivent être récupérés lorsque le formulaire est soumis et permet aussi de définir les valeurs par défaut de ces champs.

Cette fonction renvoie tout simplement un tableau associatif « nom du champ » / « valeur par défaut » :

```
function formulaire_nom_charger_dist() {
    $valeurs = array(
        "champ" => "valeur par défaut",
        "autre champ" => "",
    );
    return $valeurs;
}
```

Toutes les clés qui sont indiquées seront envoyées dans l'environnement du squelette HTML du formulaire. On récupère alors ces données par `#ENV{champ}`. Dès que le formulaire est posté, ce sont les valeurs envoyées par l'utilisateur qui sont prioritaires sur les valeurs par défaut.

Il n'est pas utile de protéger les valeurs envoyées contenant des guillemets, SPIP s'en chargeant automatiquement. Ceci dit, les champs commençants par un souligné « _ » ne subissent pas ce traitement automatique, ce qui peut être utile pour transmettre des variables complexes.

Autoriser ou non l'affichage du formulaire

Le formulaire est affiché par défaut, cependant il est possible de restreindre cet affichage en fonction d'autorisations données.

Deux possibilités :

- soit on ne veut pas du tout afficher le formulaire, on retourne alors `false` :

```
function formulaire_nom_charger_dist() {
    $valeurs = array();
    if (!autoriser("webmestre")) {
        return false;
    }
    return $valeurs;
}
```

- soit simplement une partie du formulaire est cachée (souvent la partie éditable) en utilisant la variable « editable », gérée alors dans le squelette du formulaire :

```
function formulaire_nom_charger_dist() {
    $valeurs = array();
    if (!autoriser("webmestre")) {
        $valeurs['editable'] = false;
    }
    return $valeurs;
}
```

Exemple

Le plugin « Accès restreint » dispose d'un formulaire pour affecter des zones à un auteur ; il envoie dans l'environnement des champs à récupérer et leurs valeurs par défaut : l'identifiant de zone, l'auteur connecté et l'auteur qui sera affecté à la zone. En plus, si l'auteur n'a pas les droits suffisants, la variable « editable » est passée à faux.

```
function
formulaires_affecter_zones_charger_dist($id_auteur){
    $valeurs = array(
        'zone'=>'',
        'id_auteur'=>$id_auteur,
        'id'=>$id_auteur
    );
    include_spip('inc/autoriser');
    if (!autoriser('affecterzones','auteur',$id_auteur)){
        $valeurs['editable'] = false;
```

```
    }
    return $valeurs;
}
```

Autres options de chargement

Différents autres paramètres spéciaux peuvent être envoyés dans le formulaire lors de son chargement pour modifier son comportement d'origine :

message_ok, message_erreur

Le message de succès est en principe fourni par la fonction `traiter` ; le message d'erreur par la fonction `verifier` ou `traiter`. Il est néanmoins possible de les fournir par la fonction `charger` de manière dérogatoire.

action

Cette valeur précise l'URL sur laquelle est posté le formulaire. C'est par défaut l'URL de la page en cours ce qui permet de ré-afficher le formulaire en cas d'erreur. Pour des usages très particuliers, cette URL peut-être modifiée.

_forcer_request

Lorsqu'un formulaire est soumis, SPIP l'identifie pour permettre d'avoir plusieurs formulaires du même type dans une page, et ne traiter que celui qui a été soumis. Cette vérification est basée sur la liste des arguments passés à la balise #FORMULAIRE_XXX.

Dans certains cas où ces arguments changent suite à la saisie, SPIP peut se tromper et croire que la saisie vient d'un autre formulaire.

Passer `_forcer_request` à `true` indique à SPIP qu'il ne doit pas faire cette vérifcation et traiter la saisie dans tous les cas.

_action

Si le traitement du formulaire doit faire appel à une fonction du répertoire `actions/` protégée par `securiser_action()`, il est utile d'indiquer le nom de l'action afin que SPIP fournisse automatiquement le hash de protection correspondant.

_hidden

La valeur de ce champ sera ajoutée directement dans le HTML du formulaire généré. Elle est souvent utilisée pour y ajouter des input de type « hidden » qui devront être écrits explicitement :

```
$valeurs['_hidden'] = "<input type='hidden' name='secret'
value='chut !' />";
```

Pipelines au chargement

formulaire_charger

Ce pipeline permet de modifier le tableau de valeurs renvoyées par la fonction `charger` d'un formulaire. Il est décrit dans le chapitre sur les pipelines : formulaire_charger (p.110)

paramètre _pipeline

Ce paramètre permet de modifier le code HTML envoyé en lui faisant traverser un pipeline donné. Cette information, envoyée dans le tableau de chargement, permet d'indiquer le nom d'un pipeline et des arguments à lui transmettre. Il sera appelé au moment de l'affichage du texte du formulaire.

Exemple

SPIP utilise ce paramètre de manière générique en faisant passer tous les formulaires d'édition qui appellent la fonction `formulaires_editer_objet_charger()` dans un pipeline nommé `editer_contenu_objet`. Ce pipeline est décrit dans le chapitre consacré : editer_contenu_objet (p.109).

```
$contexte['_pipeline'] = array('editer_contenu_objet',
array('type'=>$type, 'id'=>$id));
```

Le plugin CFG utilise ce paramètre pour faire passer tous les formulaires CFG écrits comme des formulaires CVT dans le pipeline `editer_contenu_formulaire_cfg`

```
$valeurs['_pipeline'] =
array('editer_contenu_formulaire_cfg',
    'args'=>array(
        'nom'=>$form,
        'contexte'=>$valeurs,
        'ajouter'=>$config->param['inline'])
);
```

Pipeline que CFG utilise alors pour enlever du contenu non nécessaire dans le HTML transmis :

```
// pipeline sur l'affichage du contenu
// pour supprimer les parametres CFG du formulaire
function cfg_editer_contenu_formulaire_cfg($flux){
    $flux['data'] = preg_replace('/(<!-- ([a-
z0-9_]\w+)(\*)?=)(.*?)-->/sim', '', $flux['data']);
    $flux['data'] .= $flux['args']['ajouter'];
    return $flux;
}
```

Vérifier les valeurs soumises

La fonction `verifier()` permet d'analyser les valeurs postées et de retourner éventuellement des erreurs de saisie. Pour cela, la fonction retourne un tableau associatif « champ » / « message d'erreur » pour les champs incriminés, ainsi éventuellement qu'un message d'erreur plus général pour l'ensemble du formulaire sur la clé « message_erreur ».

La fonction de traitement du formulaire sera appelée uniquement si le tableau retourné est vide. Dans le cas contraire, le formulaire est réaffiché avec les différents messages d'erreurs transmis.

```
function formulaire_nom_verifier_dist() {
    $erreurs = array();
    foreach(array('titre','texte') as $champ) {
        if (!_request($champ)) {
            $erreurs[$champ] = "Cette information est
obligatoire !";
        }
    }
```

```
    if (count($erreurs)) {
        $erreurs['message_erreur'] = "Une erreur est présente
dans votre saisie";
    }
    return $erreurs;
}
```

Exemple

Le plugin « Amis » dispose d'un formulaire pour inviter des personnes à devenir son ami ! La fonction `verifier()` vérifie que l'adresse mail de la personne à inviter est correcte :

```
function formulaires_inviter_ami_verifier_dist(){
    $erreurs = array();
    foreach(array('email') as $obli)
        if (!_request($obli))
            $erreurs[$obli] =
(isset($erreurs[$obli])?$erreurs[$obli]:'') .
_T('formulaires:info_obligatoire_rappel');
    if ($e=_request('email')){
        if (!email_valide($e))
            $erreurs['email'] =
(isset($erreurs['email'])?$erreurs['email']:'') .
_T('formulaires:email_invalide');
    }
    return $erreurs;
}
```

Accès SQL

SPIP 2.0 peut lire, écrire et fonctionner à partir de gestionnaires de bases de données MySQL, PostGres et SQLite.

Bien que leur syntaxe d'écriture des requêtes soit différente, SPIP, grâce à un jeu de fonctions spécifiques d'abstraction SQL permet de coder des interactions avec la base de données indépendantes de celles-ci.

Adaptation au gestionnaire SQL

SPIP s'appuie essentiellement sur le standart SQL, mais comprendra une grande partie des spécificités MySQL qu'il traduira alors si nécessaire pour les gestionnaires SQLite ou PostGres.

SPIP n'a besoin d'aucune déclaration particulière (hormis la présence du fichier de connexion adéquat pour la base de données souhaitée) pour lire et extraire des informations des bases de données, dès lors qu'on utilise soit des squelettes, soit, en php, les fonctions d'abstractions SQL prévues et préfixées de `sql_`.

Déclarer la structure des tables

Dans certains cas, particulièrement pour les plugins qui ajoutent des tables dans la base de données, ou des colonnes dans une table, il est nécessaire de déclarer la structure SQL de la table, car c'est à partir de ces déclarations que SPIP construit la requête de création ou de mise à jour des tables.

SPIP tentera alors d'adapter la déclaration au gestionnaire de données utilisé, en convertissant certaines écritures propres à MySQL.

Ainsi, si vous déclarez une table avec "auto-increment" sur la primary key à la façon de SPIP (comme dans ecrire/base/serial.php et ecrire/base/auxiliaires.php en utilisant les pipelines spécifiques SPIP 2 `declarer_tables_principales` et `declarer_tables_auxiliaires`), SPIP traduira l'écriture « auto-increment » pour qu'elle soit prise en compte lorsqu'on utilise PostGres ou SQLite.

De la même manière, une déclaration de champ "ENUM" spécifique à Mysql sera tout de même fonctionnelle sous PG ou SQLite. L'inverse par contre n'est pas valable (des déclarations spécifiques PostGres ne seront pas comprises par les autres).

Mises à jour et installation des tables

Lorsque SPIP s'installe, il utilise des fonctions pour installer ou mettre à jour ses tables. Les plugins peuvent aussi utiliser ces fonctions dans leur fichier d'installation.

Ces fonctions sont déclarées dans le fichier ecrire/base/create.php

Créer les tables

La fonction `creer_base($connect='')` crée les tables manquantes dans la base de données dont le fichier de connexion est donné par `$connect`. Par défaut, la connexion principale.

Cette fonction crée les tables manquantes (il faut évidemment qu'elles aient été déclarées), mais ne va rien modifier sur une table existante. Si la table est déclarée en tant que table principale (et non auxiliaire), et si la clé primaire est un entier, alors SPIP affectera automatiquement un type 'auto-increment' à cette clé primaire.

Mettre à jour les tables

La fonction `maj_tables($tables, $connect='')` met à jour des tables existantes. Elle ne fera que créer les champs manquants ; aucune suppression de champ ne sera effectuée. Il faut indiquer le nom de la table (chaîne) ou des tables (tableau) à la fonction. Là encore, on peut indiquer un fichier de connexion différent de la base principale.

Si une table à mettre à jour n'existe pas, elle sera créée, suivant le même principe que `creer_base()` pour l'auto-increment.

Exemples :

```
include_spip('base/create');
creer_base();
maj_tables('spip_rubriques');
maj_tables(array('spip_rubriques','spip_articles'));
```

Développer des plugins

Les plugins sont un moyen de proposer des extensions pour SPIP. Ils sont généralement fournis sous forme d'un dossier compressé (au format ZIP) à décompresser dans le dossier « plugins » (à créer au besoin) ou à installer directement en donnant l'adresse du fichier compressé via l'interface privée dans la page d'administration des plugins.

Principe des plugins

Les plugins ajoutent des fonctionnalités à SPIP, ce peut être un jeu de squelettes, une modification du fonctionnement, la création de nouveaux objets éditoriaux...

Ils ont l'avantage de permettre de gérer des tâches à accomplir au moment de leur installation ou désinstallation et d'être activables et désactivables. Ils peuvent gérer des dépendances à d'autres plugins.

Tous les dossiers et les éléments surchargeables de SPIP peuvent être recréés dans le dossier d'un plugin comme on le ferait dans son dossier « squelettes ». La différence essentielle est la présence d'un fichier XML décrivant le plugin nommé `plugin.xml`.

plugin.xml minimum

Le fichier `plugin.xml` doit être créé à la racine de votre plugin. Il contient la description de celui-ci et permet de définir certaines actions.

Le minimum pourrait être cela (les caractères hors ASCII sont échappés) :

```
<plugin>
    <nom>Porte plume - Une barre d'outil pour bien
&eacute;crire</nom>
    <auteur>Matthieu Marcillaud</auteur>
    <licence>GNU/GLP</licence>
    <version>1.2.1</version>
    <description>
    "Porte plume" est une barre d'outil g&eacute;niale pour
SPIP [...]
    </description>
    <etat>stable</etat>
    <prefix>porte_plume</prefix>
</plugin>
```

Ces attributs sont simples à comprendre, mais décrivons-les :

- `nom` : nom du plugin,
- `auteur` : auteur(s) du plugin,
- `licence` : licence(s) du plugin,

- `version` : version du plugin. Ce nommage est affiché dans l'espace privé lorsqu'on demande des informations sur le plugin, il sert aussi à gérer les dépendances entre plugins, couplé avec le préfixe. Un autre attribut à ne pas confondre est 'version_base' qui sert lorsque le plugin crée des tables ou des champs dans la base de données,
- `description` : c'est assez évident !
- `etat` : état d'avancement du plugin, peut être « dev » (en développement), « test » (en test) ou stable
- `prefix` : préfixe unique distinguant ce plugin d'un autre. Pas de chiffre, écrit en minuscule.

plugin.xml, attributs courants

Options et fonctions

Les fichiers d'options et de fonctions d'un plugin sont déclarés directement dans le fichier `plugin.xml`, avec les attributs `options` et `fonctions` :

```
    <options>porte_plume_options.php</options>
    <fonctions>inc/barre_outils.php</fonctions>
    <fonctions>autre_fichier.php</fonctions>
```

Plusieurs fichiers de fonctions peuvent être chargés si besoin, en les indiquant successivement.

Lien de documentation

L'attribut `lien` permet de donner une adresse de documentation du plugin :

```
    <lien>http://documentation.magraine.net/-Porte-
Plume-</lien>
```

Icone du plugin

L'attribut `icon` permet d'indiquer une image à utiliser pour présenter le plugin :

```
        <icon>imgs/logo-bugs.png</icon>
```

Gestion des dépendances

Les plugins peuvent indiquer qu'ils dépendent de certaines conditions pour fonctionner. Deux attributs indiquent cela : `necessite` et `utilise`. Dans le premier cas, la dépendance est forte : un plugin qui nécessite quelque chose (une certaine version de SPIP ou d'un plugin) ne pourra pas s'activer si celui-ci n'est pas présent et actif. Une erreur sera générée si l'on tente d'activer le plugin s'il ne vérifie pas sa dépendance. Dans le second cas, la dépendance est faible, le plugin peut s'activer et fonctionner même si la dépendance n'est pas présente.

Necessite

```
<necessite id="prefixe" version="[version_min;version_max]"
/>
```

- `id` est le nom du préfixe du plugin, ou "SPIP" pour une dépendance directe à SPIP,
- `version` optionnellement peut indiquer la version minimum et/ou la version maximum d'un plugin. Les crochets sont utilisés pour indiquer que la version indiquée est comprise dedans, les parenthèses pour indiquée que la version indiquée n'est pas comprise.

Utilise

Utilise permet donc de déclarer des dépendances optionnelles, exactement avec la même syntaxe que `necessite`.

Exemple

```
// necessite SPIP 2.0 minimum
<necessite id="SPIP" version="[2.0;)" />
// necessite SPIP < 2.0
<necessite id="SPIP" version="[;2.0)" />
// necessite SPIP >= 2.0, et <= 2.1
<necessite id="SPIP" version="[2.0;2.1]" />

// spip_bonux 1.2 minimum
<necessite id="spip_bonux" version="[1.2;]" />
```

Certains plugins peuvent indiquer qu'il est possible de modifier leur configuration si le plugin CFG est présent (mais il n'est pas indispensable au fonctionnement du plugin) :

```
// plugin de configuration
<utilise id="cfg" version="[1.10.5;]" />
```

Installer des librairies externes

Les plugins peuvent aussi demander à télécharger des librairies externes dont ils dépendent. Cela nécessite plusieurs choses : une déclaration spécifique dans le fichier `plugin.xml`, et la présence d'un répertoire `/lib` accessible en écriture à la racine de SPIP dans lequel sera téléchargée la librairie (ou mise manuellement).

```
<necessite id="lib:nom" src="adresse du fichier zip" />
```

- `nom` indique le nom du dossier décompressé du zip
- `src` est l'adresse de l'archive de la librairie, au format .zip

Exemple

Un plugin « loupe photo » utilise une librairie javascript qu'il installe en tant que librairie (fournie en dehors du plugin donc) de cette façon :

```
<necessite id="lib:tjpzoom" src="http://valid.tjp.hu/
tjpzoom/tjpzoom.zip" />
```

Dans le plugin, il retrouve le nom des fichiers qu'il utilise comme ceci :

```
$tjp = find_in_path('lib/tjpzoom/tjpzoom.js');
```

Le plugin « Open ID » utilise aussi une librairie externe au plugin. Il la télécharge de la même façon :

```
<necessite id="lib:php-openid-2.1.2"
src="http://openidenabled.com/files/php-openid/packages/
php-openid-2.1.2.zip" />
```

Et l'utilise ainsi :

```
// options
if (!defined('_DIR_LIB')) define('_DIR_LIB', _DIR_RACINE
. 'lib/');
define('_DIR_OPENID_LIB', _DIR_LIB . 'php-
openid-2.1.2/');
// utilisation (c'est plus complexe !)
function init_auth_openid() {
    // ...
    $cwd = getcwd();
    chdir(realpath(_DIR_OPENID_LIB));
    require_once "Auth/OpenID/Consumer.php";
    require_once "Auth/OpenID/FileStore.php";
    require_once "Auth/OpenID/SReg.php";
    chdir($cwd);
    // ...
}
```

Utiliser les pipelines

Pour utiliser les pipelines de SPIP ou d'un plugin, il faut explicitement déclarer leur utilisation dans le fichier `plugin.xml` :

```
<pipeline>
    <nom>nom_du_pipeline</nom>
    <action>nom de la fonction a charger</action>
    <inclure>repertoire/fichier.php</inclure>
</pipeline>
```

Le paramètre `action` est optionnel, par défaut, il vaut le même nom que le pipeline. Cette déclaration indique de charger un fichier particulier au moment de l'appel du pipeline (déterminé par `inclure`) et de charger une fonction `prefixPlugin_action()`. Notons que le paramètre `action` est rarement renseigné.

On indique plusieurs pipelines en les listant de la sorte :

```
<pipeline>
    <nom>nom_du_pipeline</nom>
    <inclure>repertoire/fichier.php</inclure>
</pipeline>
<pipeline>
    <nom>autre nom</nom>
    <inclure>repertoire/fichier.php</inclure>
</pipeline>
```

Exemple

Le pipeline insert_head (p.111) ajoute du contenu dans le <head> des pages publiques. Le plugin « Messagerie » (ayant « messagerie » comme préfixe) s'en sert pour ajouter des styles CSS :

```
<pipeline>
    <nom>insert_head</nom>
    <inclure>messagerie_pipelines.php</inclure>
</pipeline>
```

Et dans le fichier `messagerie_pipelines.php` :

```
function messagerie_insert_head($texte){
    $texte .= '<link rel="stylesheet" type="text/css"
href="'.find_in_path('habillage/messagerie.css').'"
media="all" />'."\n";
    return $texte;
}
```

Définir des boutons

Pour ajouter des boutons dans l'espace privé il suffit de renseigner un attribut `bouton` dans le fichier `plugin.xml`, de la sorte :

```
<bouton id="identifiant" parent="nom de l'identifiant
parent">
    <icone>chemin de l'icone</icone>
    <titre>chaine de langue du titre</titre>
```

```
    <url>nom de l'exec</url>
    <args>arguments transmis</args>
</bouton>
```

Description :

- `id` reçoit l'identifiant unique du bouton, qui sert entre autre aux sous-menus à indiquer le nom de leur bouton parent. Souvent, le nom du fichier `exec` (servant à afficher la page) est le même que le nom de l'identifiant,
- `parent` : optionnel, permet d'indiquer que le bouton est un sous élément d'un bouton parent. On renseigne donc l'identifiant du bouton parent. En son absence, c'est un élément de premier niveau qui sera créé (comme les boutons « À suivre, Édition, ... »,
- `icone` : optionnel aussi, pour indiquer le chemin de l'icone,
- `titre` : texte du bouton qui peut-être une chaîne de langue « plugin:chaine »,
- `url` indique le nom du fichier exec qui est chargé si l'on clique sur le bouton. S'il n'est pas indiqué, c'est le nom de l'identifiant qui est utilisé.
- `args`, optionnel, permet de passer des arguments à l'url (exemple : `<args>critere=debut</args>`).

Autorisations

Les boutons sont affichés par défaut pour toutes les personnes connectées à l'interface privée. Pour modifier cette configuration, il faut créer des autorisations spécifiques pour les boutons (et donc utiliser le pipeline d'autorisation pour charger les autorisations nouvelles du plugin) :

```
function autoriser_identifiant_bouton_dist($faire, $type,
$id, $qui, $opt) {
    return true; // ou false
}
```

Exemple

Les statistiques de SPIP 2.1 – en cours de développement – seront dans un plugin séparé. Il reproduit actuellement les boutons comme ceci :

```
<pipeline>
    <nom>autoriser</nom>
    <inclure>stats_autoriser.php</inclure>
</pipeline>
<bouton id="statistiques_visites">
    <icone>images/statistiques-48.png</icone>
    <titre>icone_statistiques_visites</titre>
</bouton>
<bouton id='statistiques_repartition'
parent='statistiques_visites'>
    <icone>images/rubrique-24.gif</icone>
    <titre>icone_repartition_visites</titre>
</bouton>
<bouton id='statistiques_lang'
parent='statistiques_visites'>
    <icone>images/langues-24.gif</icone>
    <titre>onglet_repartition_lang</titre>
</bouton>
<bouton id='statistiques_referers'
parent='statistiques_visites'>
    <icone>images/referers-24.gif</icone>
    <titre>titre_liens_entrants</titre>
</bouton>
```

Les autorisations sont définies dans un fichier spécifique :

```
<?php
function stats_autoriser(){}
// Lire les stats ? = tous les admins
function autoriser_voirstats_dist($faire, $type, $id,
$qui, $opt) {
    return (($GLOBALS['meta']["activer_statistiques"] !=
'non')
            AND ($qui['statut'] == '0minirezo'));
}
// autorisation des boutons
function
autoriser_statistiques_visites_bouton_dist($faire, $type,
$id, $qui, $opt) {
    return autoriser('voirstats', $type, $id, $qui,
$opt);
}
```

```
function
autoriser_statistiques_repartition_bouton_dist($faire,
$type, $id, $qui, $opt) {
    return autoriser('voirstats', $type, $id, $qui,
$opt);
}
function autoriser_statistiques_lang_bouton_dist($faire,
$type, $id, $qui, $opt) {
    return ($GLOBALS['meta']['multi_articles'] == 'oui'
            OR $GLOBALS['meta']['multi_rubriques'] ==
'oui')
        AND autoriser('voirstats', $type, $id, $qui,
$opt);
}
function
autoriser_statistiques_referers_bouton_dist($faire,
$type, $id, $qui, $opt) {
    return autoriser('voirstats', $type, $id, $qui,
$opt);
}
?>
```

Définir des onglets

Déclarer des onglets pour les pages `exec` de l'espace privé reprend exactement la même syntaxe que les boutons. Le nom du parent par contre est obligatoire et correspond à un paramètre transmis dans la fonction d'appel de l'onglet dans le fichier `exec` :

```
<onglet id='identifiant' parent='identifiant de la barre
onglet'>
    <icone>chemin</icone>
    <titre>chaine de langue</titre>
    <url>nom du fichier exec</url>
    <args>arguments</args>
</onglet>
```

Comme pour les boutons, si l'url n'est pas renseignée, c'est le nom de l'identifiant qui est utilisé comme nom du fichier à charger.

Autorisations

Encore comme les boutons, une autorisation permet de gérer l'affichage ou non de l'onglet.

```
function autoriser_identifiant_onglet_dist($faire, $type,
$id, $qui, $opt) {
    return true; // ou false
}
```

Exemple

Le plugin « Champs Extras 2 » ajoute un onglet dans la page de configuration, sur la barre d'onglets nommée très justement « configuration ». Voici ses déclarations dans le fichier `plugin.xml` :

```
<pipeline>
    <nom>autoriser</nom>
    <inclure>inc/iextras_autoriser.php</inclure>
</pipeline>
<onglet id='iextras' parent='configuration'>
    <icone>images/iextras-24.png</icone>
    <titre>iextras:champs_extras</titre>
</onglet>
```

Les autorisations sont définies dans le fichier `inc/iextras_autoriser.php`. L'onglet s'affichera uniquement si l'auteur est déclaré « webmestre ».

```
<?php
if (!defined("_ECRIRE_INC_VERSION")) return;
// fonction pour le pipeline, n'a rien a effectuer
function iextras_autoriser(){}
// declarations d'autorisations
function autoriser_iextras_onglet_dist($faire, $type,
$id, $qui, $opt) {
    return autoriser('configurer', 'iextras', $id, $qui,
$opt);
}
function autoriser_iextras_configurer_dist($faire, $type,
$id, $qui, $opt) {
```

```
    return autoriser('webmestre', $type, $id, $qui,
$opt);
}
?>
```

Enfin, dans le fichier `exec/iextras.php`, la barre d'onglet est appelée comme ci-dessous. Le premier argument est l'identifiant de la barre d'onglet souhaitée, le second l'identifiant de l'onglet en cours.

```
echo barre_onglets("configuration", "iextras");
```

Exemples

Un chapitre pour présenter quelques exemples concrets de petits scripts.

Adapter tous ses squelettes en une seule opération

Grâce à des points d'entrées spécifiques, il est possible d'agir simplement sur l'ensemble de ses squelettes pour modifier le comportement d'un type de boucle particulier, en utilisant le pipeline pre_boucle (p.113). Pour chaque boucle RUBRIQUES, quel que soit le squelette, cacher le secteur 8 :

```
$GLOBALS['spip_pipeline']['pre_boucle'] .=
'|cacher_un_secteur';
function cacher_un_secteur($boucle){
  if ($boucle->type_requete == 'rubriques') {
    $secteur = $boucle->id_table . '.id_secteur';
    $boucle->where[] = array("'!='", "'$secteur'", "8");
  }
  return $boucle;
}
```

À noter que le plugin « Accès Restreint » permet aussi d'offrir cette fonction de restriction d'accès à du contenu.

Afficher un formulaire d'édition, si autorisé

Des balises spéciales #AUTORISER permettent de gérer finement l'accès à certains contenus, à certains formulaires. Ci-dessous, si le visiteur a des droits de modifications sur l'article, afficher un formulaire pour l'éditer, qui, une fois validé, retourne sur la page de l'article en question :

```
[(#AUTORISER{modifier, article, #ID_ARTICLE})
 #FORMULAIRE_EDITER_ARTICLE{#ID_ARTICLE, #ID_RUBRIQUE,
#URL_ARTICLE}
]
```

Ajouter un type de glossaire

Il est possible d'ajouter des liens vers des glossaires externes dans SPIP via le raccourci [?nom]. Par défaut, c'est wikipédia qui est utilisé. Pour créer un nouveau lien de glossaire, la syntaxe [?nom#typeNN] existe.

- type est un nom pour le glossaire
- NN, optionnellement un identifiant numérique.

Une simple fonction `glossaire_type()` permet de retourner une url particulière. 2 paramètres sont transmis : le texte et l'identifiant.

Exemple :

Un lien vers la source des fichiers trac de SPIP 2.0 :

```
<?php
@define('_URL_BROWSER_TRAC', 'http://trac.rezo.net/trac/spip/
browser/branches/spip-2.0/');
/*
 * Un raccourci pour des chemins vers trac
 * [?ecrire/inc_version.php#trac]
 * [?ecrire/inc_version.php#tracNNN] // NNN = numero de ligne
 */
function glossaire_trac($texte, $id=0) {
    return _URL_BROWSER_TRAC . $texte . ($id ? '#L'.$id :
'');
}
?>
```

Appliquer un tri par défaut sur les boucles

Il est possible de trier le résultat des boucles avec le critère `{par}`. Ce squelette de documentation utilise pour toutes ses boucles `ARTICLES` et `RUBRIQUES` un tri `{par num titre, titre}`.

Plutôt que de le répéter pour toutes les boucles, appliquons-le une fois pour toute si aucun tri n'est déjà demandé. Pour cela, on utilise le pipeline `pre_boucle` et on ajoute dessus une sélection SQL `ORDER BY` :

Plugin.xml :

```
<pipeline>
    <nom>pre_boucle</nom>
    <inclure>documentation_pipelines.php</inclure>
</pipeline>
```

documentation_pipelines.php :

```
function documentation_pre_boucle($boucle){
```

```
    // ARTICLES, RUBRIQUES : {par num titre, titre}
    if (in_array($boucle->type_requete,
array('rubriques','articles'))
    AND !$boucle->order) {
        $boucle->select[] = "0+" . $boucle->id_table .
".titre AS autonum";
        $boucle->order[]  = "'autonum'";
        $boucle->order[]  = "'" . $boucle->id_table .
".titre'";
    }
    return $boucle;
}
```

De cette manière, les boucles sont triées par défaut :

```
// tri auto {par num titre, titre} :
<BOUCLE_a1(ARTICLES){id_rubrique}>...
// tri différent :
<BOUCLE_a2(ARTICLES){id_rubrique}{!par date}>...
```

Quelques détails

Le pipeline reçoit un objet PHP de type « boucle » qui peut recevoir différentes valeurs. La boucle possède notamment des variables `select` et `order` qui gèrent ce qui va être mis dans la clause `SELECT` et `ORDER BY` de la requête SQL générée. Le nom de la table SQL (`spip_articles` ou `spip_rubriques` dans ce cas là) est stocké dans `$boucle->id_table`.

Lorsqu'on met un numéro sur les titres des articles de SPIP (qui n'a pas encore de champ `rang` dans ses tables alors que le code est déjà prévu pour le gérer !), on l'écrit comme cela : « 10. Titre » (numéro point espace Titre). Pour que SQL puisse trier facilement par numéro, il suffit de forcer un calcul numérique sur le champ (qui est alors converti en nombre). C'est à ça que sert le « 0+titre AS autonum » qui crée un alias `autonum` avec ce calcul numérique qu'il est alors possible d'utiliser comme colonne de tri dans le `ORDER BY`.

Prendre en compte un nouveau champ dans les recherches

Si vous avez créé un nouveau champ dans une table SPIP, il n'est pas pris en compte par défaut dans les recherches. Il faut le déclarer aussi pour cela. Le pipeline rechercher_liste_des_champs (p.115) est ce qu'il vous faut, appelé dans le fichier ecrire/inc/rechercher.php

Il reçoit un tableau `table/champ = coefficient`, le coefficient étant un nombre donnant des points de résultats à la recherche. Plus le coefficient est élevé, plus le champ donnera des points de recherches si le contenu recherché est trouvé dedans.

Exemple

Vous avez un champ "ville" dans la table SQL "spip_articles" que vous souhaitez prendre en compte ? Il faut ajouter la déclaration du pipeline, puis :

```
function
prefixPlugin_rechercher_liste_des_champs($tables){
    $tables['article']['ville'] = 3;
    return $tables;
}
```

Glossaire

Définition des termes techniques employés.

Argument

On appelle « argument » en programmation le contenu envoyé lors de l'appel d'une fonction. Des fonctions peuvent utiliser plusieurs arguments. Les arguments envoyés peuvent être issus de calculs. On différenciera les « arguments » (ce qui est envoyé) des « paramètres » (ce que reçoit la fonction). On trouvera en PHP :

```
nom_de_la_fonction('argument', $argument, ...);
nom_de_la_fonction($x + 4, $y * 2); // 2 arguments calcules
envoyes.
```

Et en SPIP, pour les balises et les filtres :

```
#BALISE{argument, argument, ...}
[(#BALISE|filtre{argument, argument})]
```

Paramètre

Les « paramètres » d'une fonction, c'est à dire ce qu'elle reçoit quand on l'appelle, sont décrits dans sa déclaration. Cette déclaration peut préciser le type de valeur attendue (entier, tableau, chaîne de caractères…), une valeur par défaut, et surtout indique le nom de la variable où est stocké le paramètre utilisable dans le code de la fonction. On écrit en PHP :

```
function nom($param1, $param2=0){}
```

Cette fonction « nom » recevra deux « paramètres » lorsqu'elle sera appelée, stockés dans les variables locales `$param1` et `$param2` (qui a la valeur `0` par défaut). On peut alors appeler cette fonction avec 1 ou 2 « arguments » :

```
nom('Extra'); // param2 vaudra 0
nom('Extra', 19);
```

Récursivité

En programmation, on appelle « récursion » un algorithme (un code informatique) qui s'exécute lui-même. On parle aussi d'« auto-référence ». Les fonctions PHP peuvent s'appeler récursivement, comme ci-dessous une fonction qui somme les x premiers entiers (juste pour l'exemple, car mathématiquement cela vaut `x*(x+1)/2`).

```
// calcul de : x + (x-1) + ... + 3 + 2 + 1
function somme($x) {
    if ($x <= 0) return 0;
    return $x + somme($x-1);
}
// appel
$s = somme(8);
```

SPIP permet aussi d'écrire des boucles récursives (p.17) dans les squelettes.

Index

Symboles

A

B

C

D

E

F

G

H

I

R

S

Table des matières