Programmer avec SPIP

DOCUMENTATION À L'USAGE DES DÉVELOPPEURS ET WEBMESTRES

Version SPIP 2.1.0+ / mai 2010

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

Préface

L'histoire de ce livre commence fin 2008. Matthieu commençait alors cette documentation pour les développeurs, et nous disait : « ... surtout dans une licence libre pour que ça puisse être récupéré derrière moi et réutilisé évidemment ... et peut-être mis sur papier ». Un livre SPIP, un livre libre : ce rêve, caressé depuis des années, avait fini par germer. L'idée a fait son chemin, les technologies ont mûri.

L'autoéditeur trouve désormais sur le net des outils d'impression à la demande simples et efficaces. C'est vraiment formidable de se dire que, pour une somme modique, on peut recevoir un exemplaire unique d'un texte quel qu'il soit. Et l'émotion quand on reçoit pour la première fois l'exemplaire papier, l'imaginez-vous ?

La première version de ce livre sort dans ce lieu magique des Troglos, c'est un signe. On peut le voir comme la fin d'une aventure : la version 1.0 du livre. Mais c'est aussi le début d'une autre. Tous les éléments techniques sont en place pour éditer d'autres bouquins. Il n'y a plus qu'à les rédiger, dans un site SPIP bien sûr ! Ajouter une couverture, et envoyer le tout chez l'imprimeur.

SPIP SPIP HOURRA !

Ben.

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 quel que 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 et en espagnol. 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 2.1 nécessite a minima PHP 5.x (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 dans sa véritable casse, 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){critere}>
    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. 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 `id_chat` et `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 » (le premier `$table_des_traitements['BALISE'][]` rencontré) définit un traitement quelle que soit la table concernée,
- une clé « nom_de_la_table » (`$table_des_traitements['BALISE']['nom_de_la_table']` sans le préfixe de table) 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 particularités 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.31)	Tester des autorisations
#CACHE (p.31)	Définir la durée du cache
#CHEMIN (p.32)	Retrouver l'adresse d'un fichier
#DESCRIPTIF_SITE_SPIP (p.32)	Retourner le descriptif du site
#EDIT (p.33)	Éditer du contenu (avec le plugin « crayons »)
#ENV (p.33)	Récupérer une variable dans l'environnement
#EVAL (p.34)	Évaluer une expression via PHP
#EXPOSE (p.35)	Mettre en évidence l'élément en cours de lecture (dans une liste, un menu)
#GET (p.36)	Récupérer une valeur stockée par `#SET`
#INCLURE (p.37)	Inclure un squelette
#INSERT_HEAD (p.38)	Balise d'insertion de scripts dans le `<head>` pour SPIP ou des plugins
#INSERT_HEAD_CSS (p.38)	Balise d'insertion de CSS dans le `<head>` pour des plugins.
#INTRODUCTION (p.38)	Afficher une introduction
#LANG (p.39)	Obtenir le code de langue
#LANG_DIR (p.40)	Retourner le sens d'écriture
#LESAUTEURS (p.41)	Afficher les auteurs d'un article
#MODELE (p.41)	Insère un modèle de mise en page
#NOTES (p.42)	Afficher les notes crées avec le raccourci SPIP `[[]]`
#REM (p.44)	Mettre un commentaire dans le code
#SELF (p.44)	Retourne l'URL de la page courante
#SESSION (p.44)	Récupère une information de session
#SESSION_SET (p.45)	Définir des variables de session

Nom	Description
#SET (p.45)	Stocker une valeur, récupérable avec #GET
#VAL (p.46)	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 (Autorisations (p.194)) 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 a 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.100).

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.24)) – 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.45).

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. 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. Pour ajouter des CSS, il est préférable d'utiliser `#INSERT_HEAD_CSS` et le pipeline insert_head_css (p.161).

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

#INSERT_HEAD_CSS

`#INSERT_HEAD_CSS` placé entre les balises HTML `<head>` et `</head>` permet à des plugins d'ajouter des scripts CSS en utilisant le pipeline insert_head_css (p.161). Si cette balise n'est pas présente dans le squelette, `#INSERT_HEAD` ajoutera le contenu du pipeline elle-même.

Dans les squelettes par défaut de SPIP, cette balise est insérée avant le fichier CSS `habillage.css` dans squelettes-dist/inc-head.html. Ainsi, des thèmes graphiques qui surchargent ce fichier `habillage.css` peuvent également surcharger, en CSS, les déclarations ajoutées par les plugins auparavant.

#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.36).

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.

```
<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}`.

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 selon 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, utilisés 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]'}> ...
```

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>
```

Critères optionnels avec opérateurs

Un cas particulier des critères optionnels est que l'on peut utiliser des opérateurs sous certaines conditions. Il faut que la valeur testée dans l'environnement porte le même nom que le critère tel que le critère `X` dans : `{X ?operateur #ENV{X}}`. Tous les opérateurs peuvent s'appliquer, il suffit d'accoler le `?` à l'opérateur voulu.

Le test est ainsi effectué uniquement si la valeur désignée est présente dans l'environnement, sinon le critère est ignoré.

```
<BOUCLEx(TABLES){nom ?operateur #ENV{nom}}>
<BOUCLEx(TABLES){nom ?== ^#ENV{nom}$}>
<BOUCLEx(TABLES){nom ?!IN #ENV{nom}}>
<BOUCLEx(TABLES){nom ?LIKE %#ENV{nom}%}>...
```

Exemple

Sélectionner les 10 derniers articles ayant une date de rédaction inférieure à celle donnée dans l'environnement, sinon simplement les 10 derniers articles :

```
<ul>
<BOUCLE_art(ARTICLES){date_redac ?<
#ENV{date_redac}}{!par date}{0, 10}>
<li>#TITRE</li>
</BOUCLE_art>
<ul>
```

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 boucle, 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 les sources d'un article traduit, c'est à dire celui servant de référence aux traductions. Tous les articles sources :

```
<BOUCLE_sources(ARTICLES) {origine_traduction}>
#TITRE (#LANG)<br />
</BOUCLE_sources>
```

La traduction d'origine d'un article :

```
<BOUCLE_article(ARTICLES){id_article}>
  <BOUCLE_origine(ARTICLES) {traduction}{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})]
```

Les différents répertoires

Ce chapitre explique le rôle des différents répertoires de SPIP.

Liste des répertoires

Nom	Description
config (p.86)	Identifiants de connexion à la base de donnée et options du site.
ecrire/action (p.89)	Gérer les actions affectant les contenus de la base de données.
ecrire/auth (p.89)	Gérer l'authentification des utilisateurs.
ecrire/balise (p.89)	Déclarations des balises dynamiques et des balises génériques.
ecrire/base (p.90)	API en relation avec la base de données et déclaration des tables SQL.
ecrire/charsets (p.90)	Traducteurs d'encodages de caractères.
ecrire/configuration (p.90)	Éléments de configuration de l'espace privé de SPIP.
ecrire/exec (p.90)	Vue des pages dans l'espace privé (au format PHP).
ecrire/genie (p.91)	Tâches périodiques pour le génie (`cron`).
ecrire/inc (p.91)	Librairies et APIs diverses.
ecrire/install (p.91)	Procédure d'installation de SPIP
ecrire/lang (p.91)	Fichiers de localisation (langue)
ecrire/maj (p.91)	Procédures de mises à jour
ecrire/notifications (p.91)	Fonctions de notifications et contenus des mails de notifications
ecrire/plugins (p.92)	Concerne l'installation et la gestion des plugins
ecrire/public (p.92)	Compilateur et gestion du cache
ecrire/req (p.92)	Pilotes de bases de données
ecrire/typographie (p.92)	Correction typographiques
ecrire/urls (p.93)	Jeux d'écriture d'URL
ecrire/xml (p.93)	Parseur et vérificateur de XML
extensions (p.86)	Répertoire de plugins non désactivables
IMG (p.86)	Stockage des documents du site

Nom	Description
lib (p.86)	Librairies externes ajoutées par des plugins
local (p.86)	Stockage des caches d'images, CSS et Javascript
plugins (p.87)	Répertoire des plugins
prive/contenu (p.94)	Squelette de vue des objets dans l'interface privée
prive/editer (p.94)	Squelettes qui appellent les formulaires d'édition des objets
prive/exec (p.94)	Vue des pages dans l'espace privé (en squelettes SPIP)
prive/formulaires (p.94)	Formulaires d'édition des objets éditoriaux
prive/images (p.94)	Images de l'espace privé
prive/infos (p.95)	Squelettes des cadres d'information des objets de l'espace privé
prive/javascript (p.95)	Scripts Javascript
prive/modeles (p.95)	Modèles de base fournis par SPIP
prive/rss (p.95)	Squelettes générant les RSS du suivi éditorial dans l'espace privé
prive/stats (p.95)	Squelettes en rapport avec les statistiques
prive/transmettre (p.95)	Squelettes en rapport avec les export CSV
prive/vignettes (p.96)	Icônes des extensions de documents
squelettes (p.87)	Personnalisations des fichiers et squelettes.
squelettes-dist (p.87)	Jeu de squelettes par défaut
tmp (p.87)	Fichiers temporaires et de cache

config

Le répertoire `config` stocke des informations de configuration relatifs au site, tel que les identifiants de connexion à la base de données (`connect.php`) ou à des bases externes, le fichier `mes_options.php` relatif au site ou encore l'écran de sécurité (`ecran_securite.php`) permettant de combler rapidement d'éventuelles failles découvertes.

extensions

Le répertoire `extensions` permet de définir les plugins installés, actifs et non désactivables, dès l'installation de SPIP. Il suffit de placer les plugins souhaités dans ce répertoire.

Dans la distribution de SPIP, des plugins sont présents par défaut :

- « Compresseur », pour compresser les Javascript, CSS et HTML,
- « Filtres images et couleurs », donnant accès aux traitements graphiques et typographiques,
- « Porte Plume », fournissant une barre d'outils d'édition des raccourcis SPIP,
- « SafeHTML », pour nettoyer les forums et les flux de syndication d'éléments indésirables.

IMG

Le répertoire `IMG/` contient l'ensemble des documents éditoriaux ajoutés par les rédacteurs du site, classés (par défaut) par extension dans des sous répertoires.

lib

Ce répertoire (non présent par défaut) permet à des plugins de partager des librairies externes, qui sont alors téléchargées et extraites dans ce répertoire.

local

Ce répertoire stocke les caches crées par les images typographiques, les redimensionnements d'images, les traitements graphiques, les compressions CSS ou JavaScript ; c'est à dire tous les caches nécessitant un accès HTTP.

Lire sur ces sujets :

- Caches CSS et Javascript (p.220)
- Cache des traitements d'image (p.220)

plugins

Le répertoire `plugins` permet de placer les plugins qui seront activables et désactivables dans la page de configuration des plugins de l'espace privé. La présence d'un répertoire `plugins/auto` accessible en écriture autorise les administrateurs webmestres à télécharger automatiquement des plugins depuis l'interface.

squelettes

Le répertoire `squelettes`, non présent par défaut, permet de surcharger les fichiers d'origine de SPIP et de plugins, dont les squelettes par défaut. Ce répertoire permet aussi de créer ses propres squelettes et de placer tous les fichiers spécifiques à son site.

squelettes-dist

Ce répertoire contient le jeu de squelettes fourni avec SPIP. Il contient aussi des formulaires publics et des modèles.

tmp

Ce répertoire contient les fichiers temporaires, de caches et de log, non accessibles par HTTP. On retrouve dedans un dossier spécifique

- pour le cache (`cache`),
- un autre pour les sauvegardes (`dump`),
- pour les sessions des visiteurs enregistrés (`sessions`),

- pour les documents envoyés par FTP (`upload`)
- ou encore pour calculer les statistiques des visites (`visites`)

ecrire

Ce répertoire contient tout ce qui fait fonctionner SPIP !

ecrire/action

Ce répertoire a pour but de réaliser les modifications les contenus dans la base de données. La plupart des actions sont sécurisées, de sorte qu'elles vérifient à la fois :

- que l'auteur effectuant l'action a l'autorisation de l'effectuer,
- que c'est bien la personne connectée qui a demandée cette action en son nom.

À la fin des traitements, une redirection est effectuée vers une URL indiquée en général dans l'appel de l'action. Se référer au chapitre sur les actions et traitements (p.202) pour plus de détail.

ecrire/auth

Le répertoire `ecrire/auth` contient les différents scripts pour gérer la connexion des utilisateurs. Un fichier gère l'authentification via la procédure SPIP, un autre via un annuaire LDAP.

Les processus d'authentifications sont relativement complexes faisant entrer de nombreuses sécurités. Une API définit les différentes étapes de l'authentification et de la création de nouveaux utilisateurs. Se référer au chapitre sur l'authentification (p.206) pour plus de précisions.

ecrire/balise

Le répertoire `ecrire/balise` permet de définir

- des balises dynamiques, c'est à dire effectuant un calcul à chaque appel de page,
- des balises génériques, c'est à dire commençant par le même préfixe et effectuant des actions communes (`#URL_`, `#FORMULAIRE_`, ...)

Les balises statiques, elles, sont déclarées dans le fichier ecrire/public/balises.php, ou pour les plugins, dans des fichiers de fonctions.

Se reporter au chapitre sur les balises (p.179) pour des explications détaillées.

ecrire/base

Ce dossier contient ce qui est en rapport avec la base de données : les description des tables, les fonctions d'abstraction SQL, les fonctions de création et de mise à jour des tables SQL.

Un chapitre complet s'intéresse à la base de données : Accès SQL (p.251).

ecrire/charsets

Ce répertoire contient des fichiers servant aux traductions d'encodages de caractères, généralement appelés par le fichier `ecrire/inc/charsets.php`.

ecrire/configuration

Ce répertoire contient les éléments des pages de configuration dans l'espace privé de SPIP. Chaque fichier correspond à un cadre particulier de configuration.

ecrire/exec

Le répertoire `ecrire/exec` stocke les fichiers PHP permettant d'afficher des pages dans l'espace privé avec le paramètre `?exec=nom`. De plus en plus est utilisé maintenant un affichage utilisant des squelettes SPIP pour ces pages, installés dans le répertoire prive/exec (p.94).

Une explication détaillée est donnée dans le chapitre sur la création de pages pour l'espace privé (p.186).

ecrire/genie

Le répertoire `ecrire/genie` stocke les fonctions à executer de manière périodiques par le génie (ce qu'on appelle un `cron`), à raison, bien souvent d'un fichier par tâche à effectuer.

Lire le chapitre à ce sujet : la programmation de tâches périodiques (p.224).

ecrire/inc

Ce répertoire contient la plupart des librairies PHP créées pour SPIP. Certaines sont chargées systématiquement. C'est le cas de `ecrire/inc/utils.php` qui contient les fonctions de base et de démarrage, ou encore `ecrire/inc/flock.php` s'occupant des accès aux fichiers.

ecrire/install

Le répertoire `ecrire/install` contient tout ce qui concerne l'installation de SPIP. Les différents fichiers construisent les étapes d'installation et sont appelés via le fichier `ecrire/exec/install.php`.

ecrire/lang

Le répertoire `ecrire/lang` contient les différentes traductions de l'interface de SPIP. Ces traductions sont fournies grâce à 3 fichiers par langue (`xx` étant le code de langue) :

- `public_xx.php` traduit des éléments des squelettes publics,
- `ecrire_xx.php` traduit l'interface privée,
- `spip_xx.php` traduit... le reste ?!

ecrire/maj

Ce répertoire contient les procédures de mise à jour de la base de données au fil des versions de SPIP. Pour des versions anciennes, il contient aussi la structure de la base de données d'origine. Cela permet de réimporter des sauvegardes SPIP de version antérieure (jusqu'à SPIP 1.8.3) normalement sans difficulté.

ecrire/notifications

Ce répertoire contient les différentes fonctions appelées par l'API de notifications de SPIP présente dans le fichier `ecrire/inc/notifications.php`. Les notifications permettent (par défaut) d'envoyer des emails suite à des évènements survenus dans SPIP, comme l'arrivée d'un nouveau message sur un forum.

Ce répertoire contient aussi certains squelettes SPIP construisant des textes de mails envoyés lors de notifications.

ecrire/plugins

Le répertoire `ecrire/plugins` contient tout ce qui a un rapport avec les plugins de SPIP, ainsi que les extensions (plugins non désactivables) et les librairies externes (répertoire `lib/`). On trouve donc de quoi lister les plugins présents, déterminer leur dépendances, analyser les fichiers `plugin.xml`, gérer les différents caches en rapport avec les plugins (lire à ce sujet le cache des plugins (p.218))...

ecrire/public

Le mal nommé répertoire `ecrire/public` contient les différents fichiers relatifs à la recherche, l'analyse, la compilation, le débuggage des squelettes SPIP, la création des pages issues de squelettes squelettes et la gestion des caches correspondants.

Des précisions sur le fonctionnement du compilateur (p.207) sont présentes dans le chapitre correspondant.

ecrire/req

Le répertoire `ecrire/req` contient les traducteurs entre les fonctions d'abstraction SQL de SPIP et les moteurs de base de données correspondants.

Quatre pilotes sont fournis : MySQL, PostGres, SQLite 2 et SQLite 3.

ecrire/typographie

Ce répertoire contient les corrections typographiques pour le français et l'anglais, appliquées lors de l'appel à la fonction `typo`.

ecrire/urls

Le répertoire `ecrire/urls` contient les différents jeux d'URL proposés par SPIP (propre, html, arborescent...). L'API de ces jeux d'URL permet de construire une URL à partir d'un contexte donné et inversement de retrouver un objet et son identifiant à partir d'une URL.

ecrire/xml

Ce répertoire contient des fonctions pour analyser des chaines XML et les transformer en tableau PHP. Un outil de vérification est aussi présent et permet de calculer les erreurs de DTD d'une page.

prive

Le répertoire `prive` stocke tous les squelettes qui concernent l'interface privée de SPIP ainsi que certaines CSS relatives à cet espace privé.

prive/contenu

Le répertoire `prive/contenu` contient les squelettes servant à afficher le contenu d'un objet de SPIP, tel qu'un article (fichier `article.html`) dans l'interface privée.

prive/editer

Le répertoire `prive/editer` contient les squelettes qui appellent les formulaires d'édition de SPIP.

prive/exec

Le répertoire `prive/exec` stocke les squelettes SPIP permettant d'afficher des pages dans l'espace privé avec le paramètre `?exec=nom`. Ce répertoire n'est pas présent dans le cœur de SPIP, mais des plugins peuvent l'utiliser.

Une explication détaillée est donnée dans le chapitre sur la création de pages pour l'espace privé (p.186).

prive/formulaires

Le répertoire `prive/formulaires` contient les formulaires CVT d'édition des objets éditoriaux de SPIP.

prive/images

Ce répertoire contient les images et icônes utilisées dans l'espace privé et dans la procédure d'installation.

prive/infos

Le répertoire `prive/infos` contient les squelettes des cadres d'information des objets de l'espace privé, cadres contenant le numéro de l'objet, le statut, et quelques statistiques (nombre d'articles dans le cas d'une rubrique, nombre de visites pour un article...).

prive/javascript

Ce répertoire contient les scripts JavaScript dont jQuery utilisés dans l'espace privé et pour certains appelés par le pipeline jquery_plugins (p.162) dans le site public également.

prive/modeles

Ce répertoire contient les modèles utilisables dans les textes pour les rédacteurs, tel que `<imgXX>`, `<docXX>` ou dans les squelettes avec la balise `#MODELE`.

prive/rss

Squelettes générant les RSS du suivi éditorial dans l'espace privé, appelés via le fichier `prive/rss.html` avec une URL construite par la fonction `bouton_spip_rss` (déclarée dans `ecrire/inc/presentation.php`).

prive/stats

Squelettes en rapport avec les statistiques...

prive/transmettre

Le répertoire `prive/transmettre` contient les squelettes générant des données CSV, appelés depuis le squelette `prive/transmettre.html`.

prive/vignettes

Ce répertoire stocke les différentes icônes relatifs aux extensions de documents. La balise `#LOGO_DOCUMENT` retourne cette icône si aucune vignette de document n'est attribué. D'autres fonctions concernant ces vignettes se trouvent dans `ecrire/inc/documents.php`.

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

Lorsqu'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) {
...
}
```

Fonctions à connaître

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

Nom	Description
charger_fonction (p.103)	Trouver une fonction
find_all_in_path (p.104)	Trouver une liste de fichiers
find_in_path (p.104)	Trouver un fichier
include_spip (p.105)	Inclure une librairie PHP
recuperer_fond (p.106)	Retourner le résultat du calcul d'un squelette
spip_log (p.108)	Ajouter une information dans les logs
trouver_table (p.108)	Donne la description d'une table SQL.
_request (p.111)	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.

```php
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 :

```php
$code = recuperer_fond($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.

trouver_table

La fonction `trouver_table()` (`base_trouver_table_dist`) est déclarée dans ecrire/base/trouver_table.php et permet d'obtenir une description d'une table SQL. Elle permet de récupérer la liste des colonnes, des clés, des jointures déclarées, et d'autres informations.

En tant que fonction surchargeable, elle s'utilise avec charger_fonction (p.103) :

```
$trouver_table = charger_fonction('trouver_table', 'base');
$desc = $trouver_table($table, $serveur);
```

Ses paramètres sont :

1. `$table` : le nom de la table ('spip_articles' ou 'articles')
2. `$serveur` : optionnel, le nom de la connexion SQL utilisée, qui est par défaut celle de l'installation de SPIP.

Le tableau `$desc` retourné est de cette forme :

```
array(
    'field' => array('colonne' => 'description'),
    'key' => array(
        'PRIMARY KEY' => 'colonne',
        'KEY nom' => 'colonne' // ou 'colonne1, colonne2'
    ),
    'join' => array('colonne' => 'colonne'),
    'table' => 'spip_tables'
    'id_table' => $table,
    'connexion' => 'nom_connexion',
    'titre' => 'colonne_titre AS titre, colonne_langue AS
lang'
);
```

- La clé `field` est un tableau associatif listant toutes les colonnes de la table et donnant leur description SQL,
- `key` est un autre tableau listant les clés primaires et secondaires,
- `join` liste les colonnes de jointures possibles, si déclarés dans la description des tables principales ou auxiliaires
- `table` est le véritable nom de la table (hors préfixe : si le préfixe des tables est différent de « spip », c'est « spip_tables » qui sera tout de même retourné),

- `id_table` est le paramètre `$table` donné,
- `connexion` est le nom du fichier de connexion, si différent de celui d'installation,
- `titre` est une déclaration SQL de SELECT indiquant où est la colonne titre ou où est la colonne langue (sert entre autre pour calculer les URLs) ; exemples : « `titre, lang` », « `nom AS titre, '' AS lang` »

Cette fonction met en cache (p.218) le résultat de l'analyse afin d'éviter des accès intempestifs au serveur SQL. Pour forcer un recalcul de ce cache, il faut appeler la fonction avec une chaine vide :

```
$trouver_table = charger_fonction('trouver_table', 'base');
$desc = $trouver_table('');
```

Note : Lorsqu'une table est demandée sans préfixe « spip_ », c'est le nom de la table avec le préfixe donné pour le site qui sera retourné (pour peu que la table soit déclarée à SPIP). Demander une table « spip_tables » cherchera l'existence véritable de cette table (le préfixe n'est pas remplacé par celui utilisé pour le site). Dans l'avenir, une option sera probablement ajoutée à la fonction `trouver_table()`, comme pour sql_showtable (p.297) afin de pouvoir modifier automatiquement le préfixe.

Exemple

La fonction `creer_champs_extras()` du plugin « Champs Extras » permet de créer les colonnes SQL décrites par les instances d'objets « ChampExtra » transmises (`$c->table` est le nom de la table SQL, `$c->champ` celui de la colonne). La fonction renvoie `false` si une colonne n'a pas été créée :

```
function creer_champs_extras($champs) {
    // la fonction met a jour les tables concernées avec
maj_tables()
    // [...]
    // Elle teste ensuite si les nouveaux champs sont
bien crees :
    // pour chaque champ a creer, on verifie qu'il existe
bien maintenant !
```

```
    $trouver_table =
charger_fonction('trouver_table','base');
    $trouver_table(''); // recreer la description des
tables.
    $retour = true;
    foreach ($champs as $c){
        if ($table = table_objet_sql($c->table)) {
            $desc = $trouver_table($table);
            if (!isset($desc['field'][$c->champ])) {
                extras_log("Le champ extra '" . $c->champ
. "' sur $table n'a pas ete cree :(", true);
                $retour = false;
            }
        } else {
            $retour = false;
        }
    }
    return $retour;
}
```

_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 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 (p.321) permet de faire transiter un code entre plusieurs intermédiaires (des fonctions) pour le compléter ou le modifier.

Déclaration dans un plugin

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()`).

Déclaration hors plugin

Une utilisation d'un pipeline en dehors du cadre d'un plugin reste possible. Il faut alors déclarer son utilisation directement dans le fichier `config/mes_options.php` :

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

function nom_de_la_fonction($flux) {
    return $flux .= "Ce texte sera ajoute";
}
```

La fonction appelée doit être connue au moment de l'appel du pipeline, le plus simple étant de la déclarer comme ici pour la fonction `nom_de_la_fonction` dans le fichier d'option.

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.119)	Ajouter du contenu au centre de la page d'accueil.
accueil_gadget (p.119)	Ajouter des raccourcis en haut du contenu de la page d'accueil.
accueil_informations (p.120)	Informer sur les statistiques des objets éditoriaux sur la page d'accueil.
affichage_entetes_final (p.121)	Modifier les entêtes des pages envoyées
affichage_final (p.122)	Effectue des traitements juste avant l'envoi des pages publiques.
afficher_config_objet (p.123)	Ajouter des éléments dans un cadre de configuration des objets éditoriaux
afficher_contenu_objet (p.124)	Modifier ou compléter la vue d'un objet dans l'interface privée.
afficher_fiche_objet (p.125)	Ajouter du contenu dans la vue des objets éditoriaux
affiche_droite (p.125)	Ajouter du contenu dans la colonne « droite » de l'espace privé.
affiche_enfants (p.126)	Modifier ou compléter le contenu des listes présentant les enfants d'un objet dans l'espace privé
affiche_gauche (p.127)	Ajouter du contenu dans la colonne « gauche » de l'espace privé.
affiche_hierarchie (p.128)	Modifier le code HTML du fil d'ariane de l'espace privé.
affiche_milieu (p.129)	Ajouter du contenu au centre de la page sur les pages privées.
ajouter_boutons (p.130)	Ajouter des boutons dans le menu de l'espace privé.
ajouter_onglets (p.132)	Ajouter des onglets dans les pages de l'espace privé.
alertes_auteur (p.134)	Ajouter des alertes à l'auteur connecté dans l'espace privé.

Nom	Description
autoriser (p.135)	Charger des fonctions d'autorisations.
body_prive (p.137)	Insérer du contenu après `<body>` dans l'espace privé.
boite_infos (p.137)	Afficher des informations sur les objets dans les boîtes infos de l'espace privé.
compter_contributions_auteur (p.138)	Compter les contributions d'un auteur
declarer_tables_auxiliaires (p.140)	déclarer des tables SQL « auxiliaires »
declarer_tables_interfaces (p.141)	Déclarer des informations tierces sur les tables SQL (alias, traitements, jointures, ...)
declarer_tables_objets_surnoms (p.148)	Indiquer la relation entre le type d'objet et sa correspondance SQL.
declarer_tables_principales (p.149)	Déclarer des tables SQL « principales »
declarer_url_objets (p.151)	Permettre des URL standard sur un nouvel objet éditorial
definir_session (p.152)	Définir les paramètres distinguant les caches par visiteur
delete_statistiques (p.154)	Trigger au moment d'un effacement des tables de statistiques.
delete_tables (p.154)	Trigger au moment d'un effacement de la base de donnée.
editer_contenu_objet (p.155)	Modifier le contenu HTML des formulaires.
formulaire_charger (p.156)	Modifier le tableau retourné par la fonction `charger` d'un formulaire CVT.
formulaire_traiter (p.157)	Modifier le tableau retourné par la fonction `traiter` d'un formulaire CVT ou effectuer des traitements supplémentaires.
formulaire_verifier (p.158)	Modifier le tableau retourné par la fonction `verifier` d'un formulaire CVT.

Nom	Description
header_prive (p.159)	Ajouter des contenus dans le <head> privé.
insert_head (p.160)	Ajouter des contenus dans le <head> public.
insert_head_css (p.161)	Ajouter des CSS dans l'espace public
jquery_plugins (p.162)	Ajouter des scripts JavaScript (espace public et privé).
lister_tables_noerase (p.163)	Liste des tables à ne pas vider avant d'une restauration.
lister_tables_noexport (p.163)	Liste des tables SQL à ne pas sauvegarder
lister_tables_noimport (p.164)	Liste des tables SQL à ne pas importer.
optimiser_base_disparus (p.164)	Nettoyer des éléments orphelins dans la base de données
post_typo (p.165)	Modifier le texte après les traitements typographiques
pre_boucle (p.166)	Modifier les requêtes SQL servant à générer les boucles.
pre_insertion (p.167)	Ajouter des contenus par défaut au moment d'une insertion dans la base de données
pre_liens (p.168)	Traiter les raccourcis typographiques relatifs aux liens
pre_typo (p.170)	Modifier le texte avant les traitements typographiques
rechercher_liste_des_champs (p.171)	Définir des champs et des pondérations pour les recherches sur une table.
rechercher_liste_des_jointures (p.171)	Définir des jointures pour les recherches sur une table.
recuperer_fond (p.172)	Modifie le résultat de compilation d'un squelette
rubrique_encours (p.173)	Ajouter du contenu dans l'espace « Proposés à publication » des rubriques
styliser (p.174)	Modifier la recherche des squelettes utilisés pour générer une page.

Nom	Description
taches_generales_cron (p.175)	Assigner des tâches périodiques
trig_supprimer_objets_lies (p.176)	Supprimer des liaisons d'objets au moment de la suppression d'un objet
... et les autres (p.177)	Ceux qui restent à documenter

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_entetes_final

Ce pipeline, appelé sur chaque page publique de SPIP au moment de son affichage, reçoit un tableau contenant la liste des entêtes de la page. Il permet donc de modifier ou de compléter ces entêtes. Il est appelé peu avant le pipeline affichage_final (p.122) qui lui reçoit uniquement le texte envoyé.

Ce pipeline est appelé dans ecrire/public.php prend et retourne un tableau contenant les différents entêtes de page :

```
$page['entetes'] = pipeline('affichage_entetes_final',
$page['entetes']);
```

Exemple

Une utilisation de ce pipeline est de pouvoir gérer les statistiques du site, car en connaissant les entêtes envoyées (donc le type de page) et certains paramètres d'environnement on peut renseigner une table de statistiques de visites (le code réel a été simplifié et provient du plugin « Statistiques ») :

```
// sur les envois html, compter les visites.
function stats_affichage_entetes_final($entetes){
    if (($GLOBALS['meta']["activer_statistiques"] !=
"non")
```

```
    AND preg_match(',^\s*text/html,', $entetes['Content-
Type'])) {
        $stats = charger_fonction('stats', 'public');
        $stats();
    }
    return $entetes;
}
```

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;
}
```

afficher_config_objet

Ce pipeline permet d'ajouter des éléments dans un cadre de configuration des objets de SPIP.

Il est appelé comme cela dans ecrire/exec/articles.php :

```
$masque = pipeline('afficher_config_objet',
    array('args' => array('type'=>'type objet',
'id'=>$id_objet),
    'data'=>$masque));
```

Il s'applique pour le moment uniquement sur les articles et ajoute son contenu dans le cadre « Forum et Pétitions ».

Exemple

Le plugin « Forum » ajoute les réglages de modération (pas de forum, forum sur abonnement, forum libre...) pour chaque article comme ceci :

```
function forum_afficher_config_objet($flux){
    if (($type = $flux['args']['type']) == 'article'){
```

```
        $id = $flux['args']['id'];
        if (autoriser('modererforum', $type, $id)) {
            $table = table_objet($type);
            $id_table_objet = id_table_objet($type);

            $flux['data'] .= recuperer_fond( "prive/
configurer/moderation", array($id_table_objet => $id));
        }
    }
    return $flux;
}
```

afficher_contenu_objet

Ce pipeline permet de modifier ou compléter le contenu des pages de l'interface privée présentant des objets, tel que la page de visualisation d'un article.

Il est appelé dans la vie de chaque objet de d'espace privé, en transmettant le type et d'identifiant de l'objet, dans `args`, et le code HTML de la vue de l'objet dans `data` :

```
$fond = pipeline('afficher_contenu_objet',
    array(
    'args'=>array(
        'type'=>$type,
        'id_objet'=>$id_article,
        'contexte'=>$contexte),
    'data'=> ($fond)));
```

Exemple

Le plugin « Métadonnées Photos » ajoute un histogramme des photos et des données EXIF sous le descriptif des images JPG jointes aux objets de la sorte :

```
function photo_infos_pave($args) {
    if ($args["args"]["type"] == "case_document") {
        $args["data"] .= recuperer_fond("pave_exif",
```

```
            array('id_document' => $args["args"]["id"]));
    }
    return $args;
}
```

afficher_fiche_objet

Ce pipeline permet d'ajouter du contenu dans les pages de vues des objets éditoriaux de l'espace privé. Il est appelé comme ceci :

```
pipeline('afficher_fiche_objet', array(
    'args' => array(
        'type' => 'type_objet',
        'id' => $id_objet),
    'data' => "<div class='fiche_objet'>" . "...contenus..."
. "</div>");
```

Il est pour l'instant déclaré sur les pages « articles » et « navigation » (rubriques).

Exemple

Le plugin « Forum » l'utilise pour ajouter les boutons pour discuter d'un article. Il ajoute donc un forum au pied de l'article :

```
function forum_afficher_fiche_objet($flux){
    if (($type = $flux['args']['type'])=='article'){
        $id = $flux['args']['id'];
        $table = table_objet($type);
        $id_table_objet = id_table_objet($type);
        $discuter = charger_fonction('discuter', 'inc');
        $flux['data'] .= $discuter($id, $table,
$id_table_objet, 'prive', _request('debut'));
    }
    // [...]
    return $flux;
}
```

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.307)).

```
$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.135) 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.310)).

```
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.135) 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;
}
```

alertes_auteur

SPIP sait envoyer des messages d'alerte pour différentes occasions plus ou moins urgentes :

- Crash de la base de données
- Crash d'un plugin
- Erreur d'activation d'un plugin
- Avertissement pour prévenir d'un message dans la messagerie

Ce pipeline, appelé dans ecrire/inc/commencer_page.php par la fonction `alertes_auteur()`, permet de compléter le tableau contenant ces alertes.

```
$alertes = pipeline('alertes_auteur', array(
        'args' => array(
            'id_auteur' => $id_auteur,
           'exec' => _request('exec'),
        ),
        'data' => $alertes
    )
);
```

Il reçoit un tableau en paramètre.

- `data` : contient un tableau de texte des différentes alertes,
- `args` contient un tableau avec :

- `id_auteur` est l'auteur actuellement connecté,
- `exec` est le nom de la page affichée.

Exemple

S'il existait un plugin « Attention aux bananes » qui indiquerait aux gens qu'ils risquent de marcher sur une banane et se casser la figure, alors il pourrait faire :

```
function bananes_alertes_auteur($flux){
    $alertes = $flux['data'];

    // S'il y a une banane devant cet auteur
    if (tester_banane($flux['args']['id_auteur'])) {
        // On ajoute une alerte
        $alertes[] = "<strong>Attention ! Une banane
!</strong>";
    }

    // On retourne le tableau des alertes
    return $alertes;
}
```

Heureusement que le plugin était là.

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;
}
```

compter_contributions_auteur

Ce pipeline permet de compléter, sur la page qui liste les auteurs, le nombre de leurs contributions.

Il est appelé comme ceci dans ecrire/inc/formater_auteur.php :

```
$contributions = pipeline('compter_contributions_auteur',
array(
    'args' => array('id_auteur' => $id_auteur, 'row' =>
$row),
```

```
    'data' => $contributions));
```

Exemple

Le plugin « Forum » ajoute le nombre de messages qu'a écrit un auteur :

```
function forum_compter_contributions_auteur($flux){
    $id_auteur = intval($flux['args']['id_auteur']);
    if ($cpt = sql_countsel("spip_forum AS F",
"F.id_auteur=".intval($flux['args']['id_auteur']))){
        // manque "1 message de forum"
        $contributions = ($cpt>1) ? $cpt . '  '.
_T('public:messages_forum') : '1 ' .
_T('public:message');
        $flux['data'] .= ($flux['data']?", ":"") .
$contributions;
    }
    return $flux;
}
```

configurer_liste_metas

Ce pipeline permet de compléter (ou de modifier) les valeurs par défaut des paramètres de configurations de SPIP. Il reçoit en paramètre un tableau de couples « nom / valeur » et le retourne.

Ce pipeline est appelé dans ecrire/inc/config.php :

```
return pipeline('configurer_liste_metas', array(
    'nom_site' => _T('info_mon_site_spip'),
    'adresse_site' => preg_replace(",/$,", "",
url_de_base()),
    'descriptif_site' => '',
    //...
));
```

La fonction `config()` permet de compléter les paramètres encore absents dans SPIP mais ayant une valeur par défaut définie par le pipeline. Elle est appelée notamment par les formulaires de configurations natifs de SPIP.

```
$config = charger_fonction('config', 'inc');
$config();
```

Exemple

L'extension « Compresseur » l'utilise pour définir les options par défaut du système de compression des pages.

```
function compresseur_configurer_liste_metas($metas){
    $metas['auto_compress_js']='non';
    $metas['auto_compress_closure']='non';
    $metas['auto_compress_css']='non';
    return $metas;
}
```

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.149), 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_interfaces

Ce pipeline sert à déclarer des informations relatives aux tables SQL ou à certains champs de ces tables. Il permet de compléter les informations données par ecrire/public/interfaces.php

La fonction prend en paramètre le tableau des éléments déclarés, souvent nommé `$interface` et doit le retourner. Ce tableau est composé de différents éléments eux aussi tabulaires :

- `table_des_tables` déclare des alias de tables SQL,
- `exceptions_des_tables` attribue des alias de colonne SQL sur une table donnée,
- `table_titre` indique la colonne SQL d'un objet servant à définir le titre pour certains types de jeux d'URL
- `table_date` indique une colonne SQL de type date pour une table SQL donnée, permettant d'utiliser dessus des critères spécifiques (age, age_relatif, ...)
- `tables_jointures` explicite des jointures possibles entre tables SQL
- `exceptions_des_jointures` crée des alias de colonnes SQL issus d'une jointure
- `table_des_traitements` indique des filtres à appliquer systématiquement sur des balises SPIP.

table_des_tables

Déclare des alias pour des tables SQL, relatifs aux déclarations données dans les tables principales ou auxiliaires.

En général, tout plugin proposant un nouvel objet éditorial déclare également un alias identique au nom de l'objet. Cela permet d'écrire des boucles `<BOUCLEx(NOM)>`, identiques à `<BOUCLEx(spip_nom)>` (qui indique simplement le nom de la table SQL).

```
// 'nom_declare' = 'spip_rubriques', mais sans le prefixe
'spip_'
$interface['table_des_tables']['alias'] = 'nom_declare';
// exemples
$interface['table_des_tables']['articles'] = 'articles'; //
boucles ARTICLES sur spip_articles
$interface['table_des_tables']['billets'] = 'articles'; //
boucles BILLETS sur spip_articles
```

exceptions_des_tables

De la même manière que la déclaration d'alias de table SQL, il est possible de déclarer des alias de colonnes SQL. Ces alias peuvent aussi forcer une jointure sur une autre table.

```
// balise #COLONNE_ALIAS ou critere {colonne_alias} dans la
boucle concernee
$interface['exceptions_des_tables']['alias']['colonne_alias']
= 'colonne';
$interface['exceptions_des_tables']['alias']['colonne_alias']
= array('table', 'colonne');
// exemples
$interface['exceptions_des_tables']['breves']['date'] =
'date_heure';
$interface['exceptions_des_tables']['billets']['id_billet'] =
'id_article';
$interface['exceptions_des_tables']['documents']['type_document']
= array('types_documents'
, 'titre');
// permet d'utiliser les criteres racine, meme_parent,
id_parent
$interface['exceptions_des_tables']['evenements']['id_parent']
= 'id_evenement_source';
$interface['exceptions_des_tables']['evenements']['id_rubrique']
= array('spip_articles', 'id_rubrique');
```

table_titre

Indique quel champ sera utilisé pour générer des titres pour certains jeux d'URL (propre, arborescent...). La chaîne transmise est une déclaration de sélection SQL (SELECT), qui doit renvoyer 2 colonnes (ou alias de colonne SQL) : « titre » et « lang ». Lorsque l'objet n'a pas de champ « lang » correspondant, on doit donc renvoyer `'' AS lang`.

```
$interface['table_titre']['alias']= "colonne_titre AS titre,
colonne_lang AS lang";
// exemples
$interface['table_titre']['mots']= "titre, '' AS lang";
$interface['table_titre']['breves']= 'titre , lang';
```

Lorsqu'un objet a déclaré son titre, le générateur d'URL peut créer de belles URL automatiquement (en fonction du jeu d'URL utilisé par le site).

table_date

Cette information permet de déclarer certaines colonnes SQL comme des entrées de type date. Le compilateur de SPIP pourra alors appliquer des critères spécifiques à ces types de champs, tel que « age », « age_relatif », « jour_relatif »... Il n'y a qu'un seul champ déclaré de type date possible par table.

```
$interface['table_date']['alias'] = 'colonne';
// exemples
$interface['table_date']['articles']='date';
$interface['table_date']['evenements'] = 'date_debut';
```

tables_jointures

Ces déclarations permettent au compilateur de déterminer explicitement les jointures possibles lorsqu'une boucle sur une table demande un champ (balise ou critère) inconnu.

Le compilateur sait retrouver de façon implicite certaines jointures (sans les déclarer donc) en recherchant la colonne demandée dans les autres tables SQL qu'il connait. Le compilateur ne cherche pas sur toutes les tables, mais uniquement sur celles ayant des colonnes spécifiques communes :

- même nom que la clé primaire,
- même nom qu'une colonne déclarée en jointure potentielle dans le descriptif `join` des tables principales ou auxiliaires.

Dans de nombreux cas, il est utile et préférable de déclarer de façon explicite au compilateur les jointures qu'il peut tenter lorsqu'un champ inconnu sur une table se présente à lui. C'est le but de cette déclaration. L'ordre des déclarations a parfois son importance, puisque dès que le compilateur trouvera le champ recherché dans une des tables possibles à joindre, il liera alors cette table. Même si le champ voulu se trouvait dans la table déclarée ensuite.

```
$interface['tables_jointures']['spip_nom'][] = 'autre_table';
$interface['tables_jointures']['spip_nom']['colonne'] =
'autre_table';
// exemples
// {id_mot} sur ARTICLES
$interface['tables_jointures']['spip_articles'][]=
'mots_articles';
$interface['tables_jointures']['spip_articles'][]= 'mots';
// jointures evenements (plugin agenda) sur les mots ou
articles
$interface['tables_jointures']['spip_evenements'][]= 'mots';
// a placer avant la jointure sur articles
$interface['tables_jointures']['spip_evenements'][] =
'articles';
$interface['tables_jointures']['spip_evenements'][] =
'mots_evenements';
// jointures articles vers evenements
$interface['tables_jointures']['spip_articles'][] =
'evenements';
```

La plupart du temps, aidé aussi de la description « exceptions_des_jointures » expliqué ensuite, cela suffit pour qu'une boucle SPIP sache calculer les jointures qui lui sont nécessaires pour afficher les différentes balises demandées. Si ce n'est toujours pas suffisant, ne pas oublier que les squelettes aussi peuvent indiquer les jointures qu'ils souhaitent avec les boucles et critères (cf. Forcer des jointures (p.79)).

exceptions_des_jointures

Cette définition permet d'attribuer un alias de colonne créant une jointure sur une table pour récupérer un autre champ, si la jointure est réalisable. C'est un peu le pendant de « exception_des_tables » déclarant une jointure, mais non spécifique à une table donnée. On pourra alors utiliser cet alias comme balise SPIP ou comme critère de boucle.

Notons que lorsqu'on utilise ces jointures uniquement comme critère de boucle tel que `{titre_mots=xx}`, il est préférable d'écrire `{mots.titre=xx}`, plus générique et qui ne nécessite pas de déclaration.

```
$interface['exceptions_des_jointures']['colonne_alias'] =
array('table', 'colonne');
// exemples
$interface['exceptions_des_jointures']['titre_mot'] =
array('spip_mots', 'titre');
```

Un cas particulier existe : un troisième argument peut être donné contenant le nom de la fonction qui va créer la jointure. C'est un usage rare, utilisé par le plugin « Forms & Tables »

```
// cas particulier
$interface['exceptions_des_jointures']['forms_donnees']['id_mot']
= array('spip_forms_donnees_champs', 'valeur',
'forms_calculer_critere_externe');
```

table_des_traitements

Cette description est très utile ; elle permet de définir des traitements systématiques (filtres) sur certaines balises de SPIP. L'étoile (`#BALISE*`) désactive ces traitements.

Concrètement, pour chaque balise, ou couple balise / boucle, les fonctions indiquées seront exécutées. `%s` sera remplacé par le contenu réel que retourne la balise.

Deux constantes sont à disposition pour les usages les plus fréquents :

```
// traitements typographiques
define('_TRAITEMENT_TYPO', 'typo(%s, "TYPO", $connect)');
// traitements des raccourcis SPIP ([->artXX], <cadre>, {{}},
...)
define('_TRAITEMENT_RACCOURCIS', 'propre(%s, $connect)');
```

```
$interface['table_des_traitements']['BALISE'][]=
'filtre_A(%s)';
$interface['table_des_traitements']['BALISE'][]=
'filtre_B(filtre_A(%s))';
```

```
$interface['table_des_traitements']['BALISE'][]=
_TRAITEMENT_TYPO;
$interface['table_des_traitements']['BALISE'][]=
_TRAITEMENT_RACCOURCIS;
$interface['table_des_traitements']['BALISE']['boucle']=
_TRAITEMENT_TYPO;
// exemples dans SPIP
$interface['table_des_traitements']['BIO'][]=
_TRAITEMENT_RACCOURCIS;
$interface['table_des_traitements']['CHAPO'][]=
_TRAITEMENT_RACCOURCIS;
$interface['table_des_traitements']['DATE'][]=
'normaliser_date(%s)';
$interface['table_des_traitements']['ENV'][]=
'entites_html(%s,true)';
// exemples dans le plugin d'exemple "chat"
$interface['table_des_traitements']['RACE']['chats'] =
_TRAITEMENT_TYPO;
$interface['table_des_traitements']['INFOS']['chats'] =
_TRAITEMENT_RACCOURCIS;
```

Un exemple très souvent utile est la suppression automatique des numéros sur des titres de rubriques. Cela peut être réalisé comme ceci dans son fichier `config/mes_options.php` (ou en utilsant ce pipeline dans un plugin évidemment !) :

```
// version simple
$GLOBALS['table_des_traitements']['TITRE'][]=
'typo(supprimer_numero(%s), "TYPO", $connect)';
// version complexe (ne pas ecraser la definition existante)
if (isset($GLOBALS['table_des_traitements']['TITRE'][0])) {
    $s = $GLOBALS['table_des_traitements']['TITRE'][0];
} else {
    $s = '%s';
}
$GLOBALS['table_des_traitements']['TITRE'][0] =
str_replace('%s', 'supprimer_numero(%s)', $s);
```

Exemple

Prenons l'exemple complexe du plugin Agenda, qui déclare une table `spip_evenements`, une table de liaison `spip_mots_evenenents` et une seconde table de liaison `spip_evenements_participants`.

Un alias est posé pour boucler sur les évènements. Des jointures explicites sont déclarées, un champ date et des traitements également. Il y a presque tout !

```
function agenda_declarer_tables_interfaces($interface){
    // 'spip_' dans l'index de $tables_principales

$interface['table_des_tables']['evenements']='evenements';

    //-- Jointures ------------------------------------
--------------
    $interface['tables_jointures']['spip_evenements'][]=
'mots'; // a placer avant la jointure sur articles
    $interface['tables_jointures']['spip_articles'][]=
'evenements';
    $interface['tables_jointures']['spip_evenements'][] =
'articles';
    $interface['tables_jointures']['spip_mots'][]=
'mots_evenements';
    $interface['tables_jointures']['spip_evenements'][] =
'mots_evenements';
    $interface['tables_jointures']['spip_evenements'][] =
'evenements_participants';
    $interface['tables_jointures']['spip_auteurs'][] =
'evenements_participants';
    $interface['table_des_traitements']['LIEU'][]=
'propre(%s)';

    // permet d'utiliser les criteres racine,
meme_parent, id_parent

$interface['exceptions_des_tables']['evenements']['id_parent']='id_evenem

$interface['exceptions_des_tables']['evenements']['id_rubrique']=array('s
'id_rubrique');
```

```
    $interface['table_date']['evenements'] =
'date_debut';
    return $interface;
}
```

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
$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 :

```
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 potentielles 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;
}
```

declarer_url_objets

Ce pipeline permet d'utiliser des URLs standard de SPIP sur les objets indiqués et de calculer la correspondance entre une URL standard et l'objet correspondant. Ces URLs peuvent être de la forme :

- `spip.php?objetXX` (spip.php ?article12)
- `?objetXX` (?article12)
- pareil avec `.html` à la fin.

Avec le fichier `.htaccess` fournit par SPIP et activé, les URL peuvent aussi être :

- `objetXX` (article12)
- `objetXX.html` (article12.html)

L'URL calculée lorsqu'on utilise les fonctions de calcul d'URL de SPIP (balise `#URL_` ou fonction `generer_url_entite`) dépend du jeu d'URL sélectionné dans la configuration de SPIP.

Ce pipeline est appelé dans ecrire/inc/urls.php avec une liste d'objets prédéfinie. Il prend et retourne un tableau de la liste des objets pouvant servir d'URL :

```
$url_objets = pipeline('declarer_url_objets',
array('article', 'breve', 'rubrique', 'mot', 'auteur',
'site', 'syndic'));
```

La balise `#URL_nom` renvoie une URL pour un objet et un identifiant donnée (pas besoin de déclaration pour cela). Ce pipeline est utilisé pour décoder une URL standard et retrouver l'objet et l'identifiant de l'objet sur laquelle elle s'applique. Une fois déclaré un objet « nom », `?nomXX` dans l'URL permet à SPIP de calculer que l'objet est « nom » ; que l'identifiant « id_nom » vaut « XX » et que par conséquent, il doit essayer de charger le squelette `nom.html` avec l'identifiant trouvé.

L'usage de ce pipeline peut aller de pair avec la déclaration de « table_titre » dans le pipeline declarer_tables_interfaces (p.141). Cela indique sur quelle colonne SQL de l'objet puiser pour créer une URL signifiante.

Exemple

Le plugin « Grappes » utilise ce pipeline permettant de créer des URLs pour cet objet. `#URL_GRAPPE` crée une URL adaptée à l'objet. SPIP saura alors à quel squelette se référer lorsque cette URL est appelée : `grappe.html`.

```
function grappes_declarer_url_objets($array){
    $array[] = 'grappe';
    return $array;
}
```

Le pipeline d'interface déclare aussi le champ de titre pour des URL signifiantes :

```
function grappes_declarer_tables_interfaces($interface){
    // [...]
    // Titre pour url
    $interface['table_titre']['grappes'] = "titre, '' AS
lang";
    return $interface;
}
```

definir_session

Lorsque dans un squelette est utilisée `#AUTORISER`, `#SESSION` ou toute balise demandant à créer un cache différent par session, un identifiant spécifique est calculé avec les informations de session connues du visiteur par la fonction `spip_session`. Cet identifiant est utilisé pour nommer les fichiers de cache. Lorsqu'aucune information n'est connue du visiteur, l'identifiant retourné est nul.

Le pipeline `definir_session` permet de compléter les informations servant à créer cet identifiant. Il est donc possible de composer des caches uniques s'appuyant sur d'autres paramètres que les données du visiteur.

Le pipeline reçoit et retourne une chaîne de caractères. Il est appelé de la sorte dans le fichier ecrire/inc/utils.php :

```
$s = pipeline('definir_session',
    $GLOBALS['visiteur_session']
        ? serialize($GLOBALS['visiteur_session'])
        . '_' . @$_COOKIE['spip_session']
    : ''
);
```

Remarque : les informations de session pouvant intervenir très tôt dans le fonctionnement de SPIP, il vaut mieux déclarer pour un plugin la fonction du pipeline directement dans un fichier d'options. La déclaration dans le fichier `plugin.xml` ne nécessite pas de définir la balise XML `<inclure>` dans ce cas là :

```
<options>prefixPlugin_options.php</options>
<pipeline>
    <nom>definir_session</nom>
</pipeline>
```

Exemple

Le plugin « FaceBook Login » définit un cache dépendant aussi de l'authentification à FaceBook si elle est validée :

```
function fblogin_definir_session($flux){
```

```
    $flux .= (isset($_SESSION['fb_session']) ?
serialize(isset($_SESSION['fb_session'])) : '');
    return $flux;
}
```

Le plugin « Forms & Tables » également définit un cache spécifique lorsque des cookies liés à ses formulaires sont trouvés :

```
function forms_definir_session($session){
    foreach($_COOKIE as $cookie=>$value){
        if (strpos($cookie,'cookie_form_')!==FALSE)
            $session .= "-$cookie:$value";
    }
    return $session;
}
```

On notera que la balise dynamique `#FORMS` de ce plugin demande à créer un cache par session en mettant à `true` l'option `session` de la balise :

```
function balise_FORMS ($p) {
    $p->descr['session'] = true;
    return calculer_balise_dynamique($p, 'FORMS',
array('id_form', 'id_article',
'id_donnee','id_donnee_liee', 'class'));
}
```

delete_statistiques

Ce pipeline est appelé juste après avoir effectué une action de suppression des statistiques dans l'interface privée, sur la page `ecrire/?exec=admin_effacer`. C'est un trigger : un pipeline qui informe simplement d'un évènement, sans passer de paramètre. A ce titre, ce pipeline sera peut être renommé `trig_delete_statistiques` dans l'avenir.

```
pipeline('delete_statistiques', '');
```

Il n'y a pas encore d'utilisation dans les plugins de SPIP Zone. Ce pipeline doit servir à supprimer des tables SQL de statistiques ajoutées par d'autres plugins.

delete_tables

Ce pipeline est appelé juste après avoir effectué une action de suppression totale des tables de la base de donnée dans l'interface privée, sur la page `ecrire/?exec=admin_effacer`. C'est un trigger : un pipeline qui informe simplement d'un évènement, sans passer de paramètre. A ce titre, ce pipeline sera peut être renommé `trig_delete_tables` dans l'avenir.

```
pipeline('delete_tables', '');
```

Il n'y a pas d'application intéressante dans les plugins de SPIP Zone. Il serait possible de l'utiliser pour effectuer des traitements sur une bases de données externe si un site SPIP est réinitialisé de cette manière, ou encore pour envoyer des notifications de l'action à certains destinataires.

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.239) :

```
$contexte['_pipeline'] = array('editer_contenu_objet',
array('type'=>$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 est donc appelé lors de l'affichage d'un formulaire dans le fichier ecrire/balise/formulaire_.php

Il reçoit en argument le nom du formulaire ainsi que les paramètres transmis 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;
}
```

formulaire_traiter

Ce pipeline est appelé dans ecrire/public/aiguiller.php après les traitements (p.241) d'un formulaire CVT. Il permet de compléter le tableau de réponse ou d'effectuer des traitements supplémentaires.

Il reçoit les mêmes arguments que les pipelines formulaire_charger (p.156) ou formulaire_verifier (p.158). Il retourne le tableau d'informations résultant du traitement (message d'erreur, de réussite, redirection, formulaire éditable de nouveau...).

```
$rev = pipeline(
    'formulaire_traiter',
    array(
        'args' => array('form'=>$form, 'args'=>$args),
        'data' => $rev)
);
```

Exemple

Le plugin « Licence » qui offre la possibilité d'attribuer une licence d'utilisation aux articles l'utilise pour enregistrer la valeur de la licence par défaut enregistrée dans la configuration, au moment de la création d'un nouvel article :

```
function licence_formulaire_traiter($flux){
    // si creation d'un nouvel article lui attribuer la
licence par defaut de la config
    if ($flux['args']['form'] == 'editer_article' AND
$flux['args']['args'][0] == 'new') {
        $id_article = $flux['data']['id_article'];
```

```
        $licence_defaut = lire_config('licence/
licence_defaut');
        sql_updateq('spip_articles', array('id_licence'
=> $licence_defaut), 'id_article=' .
intval($id_article));
    }
    return $flux;
}
```

Notes :

- la fonction PHP `lire_config()` appartient au plugin de configuration « CFG ».
- en SPIP 2.1, il serait plus pertinent d'utiliser le pipeline pre_insertion (p.167) pour ce cet exemple précis.

formulaire_verifier

Ce pipeline est appelé dans ecrire/public/aiguiller.php au moment de la vérification des données soumises d'un formulaire CVT. Il permet de compléter le tableau d'erreurs renvoyé par la fonction verifier (p.240) du formulaire en question.

Il reçoit les mêmes arguments que le pipeline formulaire_charger (p.0), à savoir le nom du formulaire ainsi que les paramètres transmis dans les fonctions `charger`, `verifier` et `traiter`. Il retourne le tableau des erreurs.

```
$verifier =
charger_fonction("verifier","formulaires/$form/",true);
$post["erreurs_$form"] = pipeline('formulaire_verifier',
array(
    'args' => array(
        'form'=>$form,
        'args'=>$args),
    'data'=>$verifier
        ? call_user_func_array($verifier, $args)
        : array()));
```

Exemple

Le plugin « OpenID » se sert de ce pipeline pour vérifier, lorsqu'un auteur édite ses informations, que l'URL OpenID indiquée est correcte et dans le cas contraire indique une erreur sur le champ en question.

```
function openid_formulaire_verifier($flux){
    if ($flux['args']['form'] == 'editer_auteur'){
        if ($openid = _request('openid')){
            include_spip('inc/openid');
            $openid = nettoyer_openid($openid);
            if (!verifier_openid($openid))
                $flux['data']['openid'] =
_T('openid:erreur_openid');
        }
    }
    // [...]
    return $flux;
}
```

header_prive

Le pipeline `header_prive` permet d'ajouter des contenus dans la partie `<head>` des pages de l'espace privé. Il fonctionne comme le pipeline insert_head (p.160).

Le pipeline reçoit le contenu du HEAD et le retourne :

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

Exemple

Le plugin « Notations » se sert de ce point d'entrée pour ajouter une CSS dans l'espace privé et dans l'espace public (avec `insert_head`) :

```
function notation_header_prive($flux){
    $flux = notation_insert_head($flux);
    return $flux;
}
function notation_insert_head($flux){
    $flux .= '<link rel="stylesheet" href="' .
_DIR_PLUGIN_NOTATION  .'css/notation.v2.css" type="text/
css" media="all" />';
    return $flux;
}
```

Le plugin « Open Layers » permettant d'utiliser des cartes Open Street Map l'utilise pour charger les javascripts nécessaires :

```
function openlayer_insert_head_prive($flux){
    $flux .= '<script type="application/javascript"
src="http://www.openlayers.org/api/
OpenLayers.js"></script>
    <script type="application/javascript" src="' .
_DIR_PLUGIN_OPENLAYER . 'js/openlayers.js"></script>
    <script type="application/javascript"
src="http://openstreetmap.org/openlayers/
OpenStreetMap.js"></script>';
    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 (p.122), par exemple avec ceci dans `mes_options.php` :

```
$GLOBALS['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.

insert_head_css

Le pipeline `insert_head_css` permet à des plugins d'insérer les fichiers CSS qui leur sont nécessaires, au moment de l'appel de la balise `#INSERT_HEAD_CSS` si présente, sinon en début du résultat de `#INSERT_HEAD`. Cela permet à un squelette d'indiquer l'emplacement des CSS supplémentaires chargées.

Il est appelé simplement par :

```
return pipeline('insert_head_css', '');
```

Exemple

L'extension « Porte Plume » l'utilise (en simplifiant) pour ajouter deux fichiers CSS, le second étant d'ailleurs un squelette SPIP :

```
function porte_plume_insert_head_css($flux) {
    $css = find_in_path('css/barre_outils.css');
    $css_icones =
generer_url_public('barre_outils_icones.css');
    $flux .= "<link rel='stylesheet' type='text/css'
media='all' href='$css' />\n"
        .   "<link rel='stylesheet' type='text/css'
media='all' href='$css_icones' />\n";
    return $flux;
}
```

jquery_plugins

Ce pipeline permet d'ajouter très simplement dans les pages privées et publiques (ayant une balise #INSERT_HEAD (p.38)) 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;
}
```

lister_tables_noerase

Ce pipeline permet d'indiquer les tables SQL à ne pas vider juste avant une restauration.

Il est appelé par la fonction `lister_tables_noerase` du fichier ecrire/base/dump.php. Il prend et retourne un tableau de la liste des tables à ne pas purger :

```
$IMPORT_tables_noerase = pipeline('lister_tables_noerase',
$IMPORT_tables_noerase);
```

lister_tables_noexport

Ce pipeline permet de déclarer les tables SQL qui ne seront pas intégrées dans les sauvegardes SPIP.

Il est appelé dans la fonction `lister_tables_noexport` du fichier ecrire/base/dump.php. Il prend et retourne un tableau de la liste des tables à ne pas sauvegarder :

```
$EXPORT_tables_noexport = pipeline('lister_tables_noexport',
$EXPORT_tables_noexport);
```

Par défaut, certaines tables de SPIP sont déjà exclues : les tables de statistiques, de recherche et les révisions.

Exemple

Le plugin « Géographie » utilise ce pipeline pour décider de ne pas exporter ses tables SQL contenant des données géographiques :

```
function geographie_lister_tables_noexport($liste){
    $liste[] = 'spip_geo_communes';
    $liste[] = 'spip_geo_departements';
    $liste[] = 'spip_geo_regions';
    $liste[] = 'spip_geo_pays';
    return $liste;
}
```

lister_tables_noimport

Ce pipeline permet d'indiquer les tables SQL à ne pas importer lors d'une restauration.

Il est appelé par la fonction `lister_tables_noimport` de fichier ecrire/base/dump.php. Il prend et retourne un tableau de la liste des tables à ne pas sauvegarder :

```
$IMPORT_tables_noimport = pipeline('lister_tables_noimport',
$IMPORT_tables_noimport);
```

optimiser_base_disparus

Appelé depuis ecrire/genie/optimiser.php, il permet de compléter le nettoyage des objets orphelins, en supprimant des éléments lors des taches périodiques.

```
$n = pipeline('optimiser_base_disparus', array(
        'args'=>array(
            'attente' => $attente,
            'date' => $mydate),
        'data'=>$n
));
```

Il reçoit la durée d'attente entre 2 optimisations, ainsi que la date de péremption correspondante. Dans l'argument « data » se stocke le nombre d'éléments supprimés. La fonction `optimiser_sansref()` permet de gérer la suppression des éléments en donnant 3 arguments :

- la table,
- la clé primaire,
- un résultat de requête SQL contenant uniquement une colonne « id » définissant les identifiants à effacer.

Exemple

Pour supprimer les forums appartenant à une rubrique disparue, le plugin « Forum » l'utilise comme ceci :

```
function forum_optimiser_base_disparus($flux){
    $n = &$flux['data'];
    # les forums lies a une id_rubrique inexistante
   $res = sql_select("forum.id_forum AS id",
            "spip_forum AS forum
               LEFT JOIN spip_rubriques AS rubriques
                 ON
forum.id_rubrique=rubriques.id_rubrique",
            "rubriques.id_rubrique IS NULL
                AND forum.id_rubrique>0");
    $n+= optimiser_sansref('spip_forum', 'id_forum',
$res);
    // [...]
    return $flux;
}
```

post_typo

Le pipeline `post_typo` permet de modifier le texte après que SPIP ait effectué les traitements typographiques prévus, et donc après le pipeline pre_typo (p.170) également. Il est appelé par la fonction `corriger_typo()` de ecrire/inc/texte.php, fonction qui est appelée lors de l'utilisation des fonctions `propre()` ou `typo()`.

```
$letexte = pipeline('post_typo', $letexte);
```

Exemple

Le plugin « Typo Guillemets » remplace dans un texte les guillemets " par l'équivalent adapté à la langue comme « et » pour le français. Il analyse pour cela le texte une fois les raccourcis typographiques appliqués comme cela :

```
function typo_guillemets_post_typo($texte) {
    // ...
    switch ($GLOBALS['spip_lang']) {
        case 'fr':
            $guilles="&laquo; $2 &raquo;";
//LRTEUIN
            break;
        // ...
    }
    // on echappe les " dans les tags ;
    // attention ici \01 est le caractere chr(1), et \$0
represente le tag
    $texte = preg_replace(',<[^>]*"[^>]*(>|$),msSe',
"str_replace('\"','\01', \"\$0\")", $texte);
    // on corrige les guill restants, qui sont par
definition hors des tags
    // Un guill n'est pas pris s'il suit un caractere
autre que espace, ou
    // s'il est suivi par un caractere de mot (lettre,
chiffre)
    $texte = preg_replace('/(^|\s)"\s?([^"]*?)\s?"(\W|$)/
S', '$1'.$guilles.'$3', $texte);
    // et on remet les guill des tags
    return str_replace("\01", '"', $texte);
}
```

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_insertion

Il permet d'ajouter des contenus par défaut au moment de la création d'un nouvel élément éditorial dans la base de données.

Lorsqu'on enregistre un élément éditorial, si celui ci n'a pas encore d'identifiant (il est donc nouveau), un identifiant est crée pour cet élément, via les fonctions `insert_xx` où xx est le nom de l'objet souhaité. Cette insertion a simplement pour objectif de retourner un identifiant et d'enregistrer les valeurs par défaut de l'élément. Ce pipeline est appelé dans ces fonctions `insert_xx`.

Une fois l'identifiant connu, les taches de modifications normales sont effectuées, via les fonctions `xx_set` et `modifier_contenu` qui appelle les pipelines `pre_edition` et `post_edition`. Ce sont elles qui enregistreront les données postées par l'utilisateur, et cela donc, même pour un nouvel élément.

Le pipeline transmet le nom de la table et un tableau des champs et valeurs par défaut à insérer :

```
$champs = pipeline('pre_insertion',
    array(
        'args' => array(
            'table' => 'spip_rubriques',
        ),
        'data' => $champs
    )
);
```

Exemple

Le plugin « Forum » ajoute au moment d'une insertion la valeur du statut des forums d'un article comme ceci :

```
function forum_pre_insertion($flux){
    if ($flux['args']['table']=='spip_articles'){
        $flux['args']['data']['accepter_forum'] =
substr($GLOBALS['meta']['forums_publics'], 0, 3);
    }
    return $flux;
}
```

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.314).
- `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;
}
```

pre_typo

Le pipeline `pre_typo` permet de modifier le texte avant d'effectuer les traitements typographiques prévus par SPIP. Il est appelé par la fonction `corriger_typo()` de ecrire/inc/texte.php, fonction qui est appelée lors de l'utilisation des fonctions `propre()` ou `typo()`.

```
$letexte = pipeline('pre_typo', $letexte);
```

Les modifications proposées doivent s'occuper uniquement de traitement pour des éléments qui pourront être affichés sur une seule ligne (*inline*). Pour des traitements qui modifient ou créent des blocs ou paragraphes, il faudra utiliser le pipeline `pre_propre`.

Exemple

Le plugin « Enluminures Typographiques » modifie automatiquement quelques écritures de caractères, par exemple pour transformer « (c) » en « © » :

```
function typoenluminee_pre_typo($texte) {
    // ...
    $chercher_raccourcis = array(
        // ...
        /* 19 */    "/\(c\)/Si",
        /* 20 */    "/\(r\)/Si",
        /* 21 */    "/\(tm\)/Si",
        /* 22 */    "/\.\.\./S",
```

```
    );
    $remplacer_raccourcis = array(
        // ...
        /* 19 */    "&copy;",
        /* 20 */    "&reg;",
        /* 21 */    "&trade;",
        /* 22 */    "&hellip;",
    );
    // ...
    $texte = preg_replace($chercher_raccourcis,
$remplacer_raccourcis, $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.106) 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;
}
```

rubrique_encours

Il permet d'ajouter du contenu dans le cadre « Proposés à publication » des rubriques. Ce cadre s'affiche uniquement si au moins un élément (article, site, brève...) dans la rubrique est proposé à publication.

Il est appellé dans ecrire/exec/naviguer.php :

```
pipeline('rubrique_encours', array(
    'args' => array('type' => 'rubrique', 'id_objet' =>
$id_rubrique),
    'data' => $encours));
```

Exemple

Le plugin « Forum » l'utilise pour ajouter une phrase suggérant de commenter les articles proposés à la publication :

```
function forum_rubrique_encours($flux){
    if (strlen($flux['data'])
      AND $GLOBALS['meta']['forum_prive_objets'] !=
'non')
        $flux['data'] =
_T('texte_en_cours_validation_forum') . $flux['data'];
    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;
}
```

taches_generales_cron

Ce pipeline permet de déclarer des fonctions exécutées de manière périodique par SPIP. Il est appelé dans le fichier ecrire/inc/genie.php par la fonction `taches_generales`, prend et retourne un tableau associatif ayant pour clé le nom de la fonction à exécuter et pour valeur la durée en seconde entre chaque exécution.

```
return pipeline('taches_generales_cron', $taches_generales);
```

Lire le chapitre sur le Génie (p.224) pour plus de renseignements.

Exemple

Un plugin quelconque peut déclarer une fonction de nettoyage à exécuter toutes les semaines :

```
function carte_postale_taches_generales_cron($taches){
    $taches['nettoyer_cartes_postales'] = 7*24*3600; //
toutes les semaines
    return $taches;
}
```

Cette fonction est contenue dans le fichier `genie/nettoyer_cartes_postales.php` . Elle supprime tous les fichiers d'un répertoire donné âgés de plus de 30 jours, grâce à la fonction `purger_repertoire` :

```
function genie_nettoyer_cartes_postales_dist($t){
    // Purge des cartes postales agees de 30 jours
    include_spip('inc/invalideur');
    purger_repertoire(_DIR_IMG . 'cartes_postales/',
array(
        'atime' => (time() - (30 * 24 * 3600)),
    ));
    return 1;
}
```

trig_supprimer_objets_lies

Ce pipeline est un trigger (il ne retourne rien) appelé au moment de la suppression de certains objets. Il permet de supprimer des informations contenues dans les tables de liaisons en même temps que l'objet supprimé. Il reçoit un tableau des différentes suppressions (contenant le type et l'identifiant de l'objet supprimé).

```
pipeline('trig_supprimer_objets_lies', array(
        array('type'=>'mot', 'id'=>$id_mot)
));
```

Ce pipeline est appelé au moment de la suppression d'un mot-clé et d'un message.

Exemple

Le plugin « Forum » l'utilise pour supprimer les liens avec les messages de forums liés à un mot-clé supprimé ou à un message (de messagerie) supprimé :

```
function forum_trig_supprimer_objets_lies($objets){
    foreach($objets as $objet){
        if ($objet['type']=='message')
            sql_delete("spip_forum", "id_message=" .
sql_quote($objet['id']));
        if ($objet['type']=='mot')
            sql_delete("spip_mots_forum", "id_mot=" .
intval($objet['id']));
    }
    return $objets;
}
```

… et les autres

Il reste un certain nombre de pipelines à documenter. Voici leur noms :

1. affiche_formulaire_login
2. afficher_nombre_objets_associes_a
3. afficher_revision_objet
4. arbo_creer_chaine_url

5. agenda_rendu_evenement
6. base_admin_repair
7. calculer_rubriques
8. exec_init
9. formulaire_admin
10. libelle_association_mots
11. mots_indexation
12. nettoyer_raccourcis_typo
13. notifications
14. objet_compte_enfants
15. page_indisponible
16. post_boucle
17. post_image_filtrer
18. pre_propre
19. post_propre
20. pre_edition
21. post_edition
22. pre_syndication
23. post_syndication
24. pre_indexation
25. propres_creer_chaine_url
26. requete_dico
27. trig_calculer_prochain_postdate
28. trig_propager_les_secteurs

Balises

Explications, fonctionnement, création de balises statiques, dynamiques ou 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";
}
```

Créer des pages dans l'espace privé

Les pages de l'espace privé peuvent être complétée en créant de nouveaux fichiers pour y accéder. 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 :

```
<!--#hierarchie-->
<ul id="chemin">
    <li>Une liste de pages constituant un chemin</li>
</ul>
<!--/#hierarchie-->

<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 `<!--#hierarchie-->`, `<!--#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.

Si le squelette ne renvoie que du vide, alors SPIP génèrera automatiquement une erreur d'autorisation.

D'un point de vue technique, ces squelettes sont traités par le fichier ecrire/exec/fond.php. Automatiquement, les pipelines suivants sont ajoutés : affiche_hierarchie (p.128), affiche_gauche (p.127), affiche_droite (p.125) et affiche_milieu (p.129) en passant en argument 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.

Exemple

Le plugin « Formidable » utilise les squelettes SPIP pour construire les pages de l'espace privé. Pour afficher les réponses à un formulaire, il fait :

```
<BOUCLE_formulaire(FORMULAIRES){id_formulaire}>
<BOUCLE_autoriser(CONDITION){si #AUTORISER{voir,
formulaires_reponse}}>

<!--#hierarchie-->
<ul id="chemin">
    <li>
        <a href="#URL_ECRIRE{formulaires_tous}"
class="racine"><:formidable:formulaires_tous:></a>
    </li>
    <li>
        <span class="bloc">
            <em>&gt;</em>
            <a class="on"
href="[(#URL_ECRIRE{formulaires_voir}
```

```
                |parametre_url{id_formulaire,
#ID_FORMULAIRE})]">#TITRE</a>
        </span>
    </li>
</ul>
<!--/#hierarchie-->

<div class="fiche_objet">
    <a href="[(#URL_ECRIRE{formulaires_voir}
        |parametre_url{id_formulaire, #ID_FORMULAIRE})]"
class="icone36" style="float:left;">
        <img width="24" height="24" src="#CHEMIN{images/
formulaire-24.png}" />
        <span><:retour:></span>
    </a>

    <:formidable:voir_reponses:>
    <h1>#TITRE</h1>
    <div class="nettoyeur"></div>
</div>

<INCLURE{fond=prive/liste/
formulaires_reponses}{id_formulaire}
    {titre=<:formidable:reponses_liste_publie:>}{ajax} />

<!--#navigation-->
<div class="cadre infos cadre-info">
    <div class="numero">
        <:formidable:voir_numero:>
        <p>#ID_FORMULAIRE</p>
    </div>
    <div class="hover">
        <a href="#SELF" class="cellule-h">
            [<img src="(#CHEMIN{images/formulaire-
reponses-24.png})" style="vertical-align:middle;" alt=""
/>]
            <span style="vertical-
align:middle;"><:formidable:reponses_liste:></span>
        </a>
    </div>
    <div>
        <a href="[(#URL_ECRIRE{formulaires_analyse}
```

```
            |parametre_url{id_formulaire,
#ID_FORMULAIRE})]" class="cellule-h">
            [<img src="(#CHEMIN{images/formulaire-
analyse-24.png})" style="vertical-align:middle;" alt=""
/>]
            <span style="vertical-
align:middle;"><:formidable:reponses_analyse:></span>
        </a>
    </div>
</div>
<!--/#navigation-->
</BOUCLE_autoriser>
</BOUCLE_formulaire>
```

Remarques :

- L'ensemble est entouré d'une boucle testant l'existence du formulaire : s'il n'existe pas le squelette ne renverra rien et produira une erreur.
- De la même façon il est entouré d'un test avec #AUTORISER (p.194) pour vérifier que la personne a le droit de voir les réponses. On utilise ici la boucle `CONDITION` du plugin « Bonux », afin de pouvoir continuer à écrire des boucles SPIP à l'intérieur de la condition.
- Le bloc `<!--#hierarchie-->` affiche un chemin pertinent parmi les pages privées du plugin.

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
<?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.

Fonctionnalités

Ce chapitre développe quelques fonctionnements de SPIP en détail ; autorisations, actions, authentifications, cache, compilateur...

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.135) 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.

Génerer 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>
]
```

Actions et traitements

Le répertoire `ecrire/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,'');
}
```

Authentifications

Le répertoire `auth` contient les différents scripts pour gérer la connexion des utilisateurs. Couplé avec l'API disponible dans le fichier ecrire/inc/auth.php, l'ensemble permet de définir de nouveaux modes d'authentification et de création d'utilisateurs de site SPIP. Deux modes d'authentification sont fournis dans SPIP :

- SPIP pour une connexion tout à fait normale,
- 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 sha256 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 si ok retourne le tableau de la ligne SQL
de l'utilisateur
// Si risque de secu repere a l'installation retourne False
function auth_spip_dist ($login, $pass, $serveur='') {
...
}
```

Compilateur

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.

Cache

L'usage de différents caches est une partie intrinsèque de SPIP permettant de générer les différentes pages aux visiteurs plus rapidement, dans une optique de performance : on garde à portée de main les données qui sont souvent accédées, ou longues à calculer.

Cache des squelettes

Il existe différents caches dans SPIP, d'autres pouvant aussi être fournis par des plugins tel que « Mémoïsation », « Fastcache » ou encore « Cache Cool ».

Un des caches essentiels est celui des squelettes : le résultat de la compilation d'un squelette, donc le code PHP généré, est mis en cache dans le répertoire `tmp/cache/skel`. Ce cache a une durée de validité illimitée. Il sera recréé, pour un squelette donné, uniquement si :

- le squelette d'origine est modifié (en se basant sur la date du fichier sur le disque),
- le fichier `mes_options.php` ou `mes_fonctions.php` est modifié,
- le paramètre `var_mode=recalcul` est passé dans l'URL,
- le cache est manuellement vidé.

Cache des pages

Un second niveau de cache est celui des pages demandées par les visiteurs du site. Leur résultat est sauvegardé, dans les répertoires `tmp/cache/0 à f/` avec une durée de validité. Ces fichiers sont répartis dans plusieurs dossiers car dans un seul, leur nombre pourrait devenir trop important et avoir un impact sur les performances du système de fichiers du serveur. À noter que les fichiers de plus de 16ko sont automatiquement compressés (gz) si PHP dispose de la fonction `gzcompress()`.

Ce cache est recréé lorsque :

- la durée de validité a expirée (définie dans les squelettes par `#CACHE` ou en son absence par la constante `_DUREE_CACHE_DEFAUT`),
- le contenu éditorial de la base de donnée a été modifié. SPIP s'appuie sur la date de dernière modification pour le déterminer (`$GLOBALS['meta']['derniere_modif']`) renseignée par la fonction `suivre_invalideur()` de ecrire/inc/invalideur.php,

- le paramètre `var_mode=calcul` est passé dans l'URL.

Cache SQL

SPIP met en cache certains éléments de la base de données pour éviter des appels intempestifs au serveur SQL et pour que l'affichage des pages publiques déjà en cache puisse fonctionner même si le serveur de base de donnée est indisponible. Deux caches sont ainsi créés.

Cache des métas

Le premier est un export complet de la table SQL `spip_meta`. Cette table stocke des paramètres de configuration ou de fonctionnement de SPIP. Ces informations sont à la fois déposées dans la globale `$GLOBALS['meta']` et, à l'exception des données sensibles servant à l'authentification, dans le fichier `tmp/meta_cache.php`. Ce fichier a une durée de validité définie par `_META_CACHE_TIME`. Il est réécrit lors de l'appel à `ecrire_meta()` ou `effacer_meta()`. La fonction `lire_metas()` elle recalcule `$GLOBALS['meta']` avec les informations de la base de données.

Cache des descriptions SQL

Le second cache concerne la description des tables SQL des bases de données. Ces descriptions sont stockées dans les fichiers `tmp/cache/sql_desc*.txt`, avec un fichier par connecteur de base de données. Ce fichier est créé et utilisé par la fonction `base_trouver_table_dist()` qui sert de base à diverses fonctions PHP en rapport avec les descriptions SQL comme `table_objet()`, `id_table_objet()`, `objet_type()`.

Pour recréer ce fichier de cache, il faut explicitement appeler la fonction `trouver_table()` sans argument :

```
$trouver_table = charger_fonction('trouver_table','base');
$trouver_table();
```

Cache des plugins

Des fichiers de cache spécifiques aux plugins sont aussi créés dans `tmp/` ou dans `tmp/cache/`.

plugin_xml.cache

Le résultat de l'analyse des différents fichiers `plugin.xml` est mis en cache dans le fichier `tmp/plugin_xml_cache.gz`.

Ce fichier est recréé lors de l'écriture de la liste des plugins actifs via la fonction `ecrire_plugin_actifs()` qui appelle la fonction `plugins_get_infos_dist()` de ecrire/plugins/get_infos.php gérant la récupération des informations d'un plugin. Le fichier est aussi supprimé, comme de nombreux fichiers de cache lors des mises à jour de la structure de base de données.

Fichiers de chargement des plugins

Les plugins déclarent des fichiers d'options, de fonctions et des actions à effectuer sur des appels de pipelines. L'ensemble des fichiers à charger est compilé dans 3 fichiers, recalculés lors d'un passage sur la page de gestion des plugins `ecrire/?exec=admin_plugin`, d'un vidage du cache ou d'une mise à jour de la structure de la base de données :

- `tmp/cache/charger_plugins_options.php` contient la liste des fichiers d'options à charger,
- `tmp/cache/charger_plugins_fonctions.php` contient la liste des fichiers de fonctions,
- `tmp/cache/charger_plugins_pipelines.php` celle des fonctions à exécuter pour chaque pipeline.

Cache des chemins

SPIP utilise différents dossiers pour rechercher les fichiers qui lui sont nécessaires. Lire à ce sujet La notion de chemin (p.100). Lorsqu'il cherche un fichier via la fonction `find_in_path` — fonction qui sert de base à `include_spip`, `charger_fonction`, `recuperer_fond`, aux inclusions de squelettes ou à la balise `#CHEMIN` —, l'ensemble des chemins est parcouru jusqu'à trouver dedans le fichier recherché. L'ensemble de ces recherches crée de nombreux accès disques qu'il est bon de limiter.

SPIP met pour cela en cache, dans le fichier `tmp/cache/chemin.txt`, l'ensemble des correspondances entre un fichier demandé et son emplacement réel trouvé dans un des chemins.

Ainsi, lorsqu'un fichier est demandé, SPIP cherche si le chemin est en cache. Si ce n'est pas encore le cas, il calcule son emplacement et enregistre le tableau de correspondance enrichi du nouveau fichier.

Ce fichier de cache est recréé par l'appel du paramètre `var_mode=recalcul` dans l'URL, ou par une vidange manuelle du cache.

Caches CSS et Javascript

L'extension « Compresseur » présente dans SPIP permet de compacter les différents éléments CSS et Javascript pour limiter le nombre d'appels sur le serveur et la taille des fichiers à obtenir.

Cette compression est active par défaut dans l'espace privé, désactivable via la constante `_INTERDIRE_COMPACTE_HEAD_ECRIRE`.

```
define('_INTERDIRE_COMPACTE_HEAD_ECRIRE', true);
```

Cette compression peut s'activer sur l'espace public selon la configuration choisie. SPIP créera un fichier CSS compacté par type de média (screen, print...), et un fichier Javascript compacté pour tous les scripts externes connus dans le HEAD de la page HTML.

Ces fichiers sont mis en cache dans `local/cache-js/` et `local/cache-css/`. Ces caches sont recalculés si le paramètre `var_mode=recalcul` est passé dans l'URL.

Cache des traitements d'image

SPIP dispose d'une librairie de filtres graphiques permettant par défaut de pouvoir redimensionner des images facilement. Ces fonctions sont définies dans ecrire/inc/filtres_images_mini.php. L'extension « Filtres Images et Couleurs » active par défaut offre de nombreux autres filtres, comme créer des images typographiques ou utiliser masques, fusionner des images, extraire des couleurs...

Afin d'éviter de recalculer plusieurs fois des traitements extrêmement gourmands, SPIP stocke les résultats des calculs effectués dans les répertoires `local/cache-gd2` et `local/cache-vignettes`.

Ces images en cache ne seront effacées que lorsque le cache des images est vidé depuis l'interface de SPIP ou lorsque le paramètre `var_mode=images` est transmis dans l'URL.

Actualisation du cache

Lors d'une utilisation normale de SPIP, avec des visites, des nouveaux articles publiés, le cache et l'actualisation des données est correctement géré. Par défaut (mais des plugins pourraient modifier ce comportement), dès que SPIP a connaissance de modifications des contenus éditoriaux dans la base de donnée, il invalide tout le cache des pages. Une page demandée sera alors calculée de nouveau avant - ou après avec le plugin « Cache Cool » - d'être servie au visiteur.

Il est souvent nécessaire de vider le cache manuellement lorsqu'on effectue des modifications directement sur les fichiers, particulièrement en mettant à jour une feuille de style ou un script Javascript calculés par des squelettes SPIP si les options de compressions sont actives.

Se rappeler que :

- `var_mode=calcul` dans l'URL actualise le cache de la page
- `var_mode=recalcul` (pour des administrateurs) dans l'URL recompile le squelette puis actualise le cache de la page.
- passer sur la page de gestion des plugins `ecrire/?exec=admin_plugin` recalcule les fichiers de cache `tmp/cache/charger_*.php` des plugins, soit les listes de fichiers d'options, de fonctions et de pipelines.
- le navigateur a son propre cache, que ce soit pour les pages ou pour les éléments AJAX. Il faut aussi penser à le vider ; ce n'est pas forcément SPIP qui ne retourne pas les contenus attendus, mais peut être le navigateur qui retourne son cache.

Configurer le cache

Différents paramètres permettent de gérer plus finement le cache des pages de SPIP.

Durée du cache

Il est inutile de renseigner pour chaque squelette SPIP une durée de cache en utilisant la balise #CACHE. Cette balise est utile pour définir une durée de validité différente de la valeur par défaut. Concrètement, une inclusion listant des actualités issues de syndications d'autres sites peut avoir un cache rafraichi plus souvent, peut être toutes les heures, que le reste du site.

Dans la plupart des cas, il vaut mieux utiliser une durée de cache assez longue par défaut, puisque SPIP rend obsolète le cache si des modifications des contenus sont effectuées.

Le cache des pages est défini à une journée, qu'il est possible de changer avec la constante _DUREE_CACHE_DEFAUT, par exemple pour mettre un mois de validité :

```
define('_DUREE_CACHE_DEFAUT', 24*3600*30);
```

Taille du cache

SPIP s'arrange pour que le cache ait une taille ne dépassant pas une certaine valeur, qui est de 10Mo par défaut. La variable globale $GLOBALS['quota_cache'] permet de changer cette valeur, en mettant par exemple 100Mo :

```
$GLOBALS['quota_cache'] = 100;
```

Validité du cache

Uniquement pour du développement ou pour débugguer, il peut être utile de modifier le fonctionnement ou l'usage du cache. Une constante _NO_CACHE (ou via un plugin « NoCache ») permet cela :

```
// ne jamais utiliser le cache
// ni meme creer  les fichiers cache
define('_NO_CACHE', -1);
// ne pas utiliser le fichier en cache,
// mais stocker le resultat du calcul dans le fichier cache
define('_NO_CACHE', 1);
```

```
// toujours prendre les fichiers en cache s'ils existent
// s'ils n'existent pas, les calculer
define('_NO_CACHE', 0);
```

Tâches périodiques (cron)

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
    return $taches;
}
```

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;
}
```

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 :

- Le formulaire est encadré d'une classe CSS `formulaire_spip` et d'une autre de son propre nom, `formulaire_demo` ici. Le nom peut être récupéré plus agréablement par le contexte `#ENV{form}` (ou `#FORM`

directement), ce qui peut donner : `<div class="formulaire_spip formulaire_#FORM">`.

- 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`

Utiliser AJAX facilement

Entourer la balise formulaire d'une classe CSS ajax indique à SPIP d'utiliser AJAX permettant de ne recharger que le formulaire et non toute la page.

```
<div class="ajax">
#FORMULAIRE_JOLI
</div>
```

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. Il est proposé, pour de tels blocs, de les encadrer également dans 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>
```

Le premier `<li>` posséde une classe CSS « fieldset ». En lieu et place des balises HTML `<legend>` est suggéré 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, on utilise un bloc `<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>
```

Pour passer la liste en horizontal en CSS, il suffit d'indiquer que le bloc « choix » doit s'afficher `inline` :

```
.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. Pour cela, deux classes CSS à insérer dans une balise `<p>` ou `<em>` peuvent être utilisées :

- **explication** (avec `<p>`) permet d'écrire une explication plus détaillée que le label du champ souhaité
- **attention** (avec `<em>`) 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 formulaires_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.156)

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.155).

```
$contexte['_pipeline'] = array('editer_contenu_objet',
array('type'=>$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 formulaires_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;
}
```

Le pipeline formulaire_verifier (p.158) permet de compléter les erreurs retournées.

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;
}
```

Effectuer des traitements

Lorsque la fonction de vérification (p.240) ne renvoie aucune erreur, le formulaire passe alors à la fonction `traiter()`. C'est dans celle-ci qu'il faudra effectuer les opérations voulues avec les données issues du formulaire (envoi par courriel, modification de la base de données, etc.).

La fonction doit renvoyer un tableau associatif :

```php
function formulaires_nom_traiter(){
    // Effectuer des traitements

    // Valeurs de retours
    return array(
        'message_ok' => 'Excellent !', // ou bien
        'message_erreur' => 'Et mince, une erreur.'
    );
}
```

Valeurs importantes

Voici quelques valeurs que l'on renvoie souvent :

- **message_ok** permet de retourner un agréable message à l'utilisateur, lui indiquant que tout s'est bien passé.
- **message_erreur**, inversement, permet de renvoyer un message d'erreur, lorsque le traitement n'a pas fonctionné.
- **editable**, comme au chargement, est utilisée pour afficher ou non la partie éditable du formulaire. Par défaut elle vaut `false` mais vous pouvez lui indiquer `true` si votre formulaire peut être utilisé plusieurs fois à la suite.
- **redirect** est une URL qui permet d'indiquer à SPIP vers quelle page il redirigera le visiteur après les traitements du formulaire. Par défaut la page boucle sur elle-même.

Pipeline formulaire_traiter

Une fois la fonction `formulaires_nom_traiter` effectuée, le pipeline formulaire_traiter (p.157) est exécuté, permettant à d'autres plugins de compléter les traitements de ce formulaire.

Traitement sans AJAX

Si un formulaire est appelé en AJAX mais qu'il redirige sur une autre page à la fin du traitement, cela oblige à des contorsions Javascript (gérées par SPIP) pour capturer la redirection et renvoyer effectivement le navigateur sur une autre URL au retour de la réponse.

Lorsqu'une redirection est certaine, il est possible d'interdire AJAX sur le traitement du formulaire tout en le conservant pour la partie de vérification. De cette manière, seul le formulaire sera rechargé en cas d'erreur dans `verifier()`, mais si le traitement s'effectue, toute la page sera directement rechargée.

Pour cela, il faut appeler la fonction `refuser_traiter_formulaire_ajax()` **au tout début** des traitements :

```
function formulaires_nom_traiter(){
    // Empecher le traitement en AJAX car on sait que le
formulaire va rediriger autre part
    refuser_traiter_formulaire_ajax();

    // Effectuer des traitements

    // Valeurs de retours
    return array(
        'redirect' => 'Une autre URL'
    );
}
```

Exemples

La gestion de formulaires CVT mérite bien quelques exemples dédiés.

Calcul de quantième

Cet exemple court va permettre de calculer et d'afficher le quantième (le numéro d'un jour dans l'année) à partir d'une date saisie dans un formulaire.

Ce formulaire sera nommé « calculer_quantieme » et pourra donc être appelé dans un squelette par `#FORMULAIRE_CALCULER_QUANTIEME` ou dans le texte d'un article par en utilisant le code `<formulaire|calculer_quantieme>`.

Mise en place

Les deux fichiers nécessaires seront créés comme ceci :

- `formulaires/calculer_quantieme.html` pour la partie HTML
- `formulaires/calculer_quantieme.php` pour les fonctions CVT d'analyse et de traitement PHP.

Squelette HTML

Le fichier `formulaires/calculer_quantieme.html` contient le code suivant, respectant la structure HTML et classes CSS préconisées :

```
<div class="formulaire_spip formulaire_#FORM">
[<p class="reponse_formulaire
reponse_formulaire_ok">(#ENV*{message_ok})</p>]
[<p class="reponse_formulaire
reponse_formulaire_erreur">(#ENV*{message_erreur})</p>]
[(#ENV{editable}|oui)
<form name="formulaire_#FORM" action="#ENV{action}"
method="post"><div>
    #ACTION_FORMULAIRE{#ENV{action}}
    <ul>
    <li class="editer_date_jour obligatoire[
(#ENV**{erreurs}|table_valeur{message}|oui)erreur]">
        <label for="champ_date_jour">Date (jj/mm/aaaa)
:</label>
        [<span
class='erreur_message'>(#ENV**{erreurs}|table_valeur{message})</span>]
```

```
        <input type="text" id="champ_date_jour"
name="date_jour" value="[(#ENV{date_jour})]" />
    </li>
    </ul>
    <p class="boutons">
        <input type="submit" name="ok" value="Trouver" />
    </p>
</div></form>
]
</div>
```

À noter que le plugin « Saisies » permet d'écrire les champs de formulaire grâce à une balise `#SAISIE` en indiquant le type de saisie et le nom de la variable utilisée, puis les autres paramètres optionnels. Son utilisation pourrait donner (partie entre `<ul>` et `</ul>`) :

```
<ul>
[(#SAISIE{input, date_jour, obligatoire=oui, label="Date (jj/
mm/aaaa) :"})]
</ul>
```

Chargement, vérifications et traitements

Ce fichier `formulaires/calculer_quantieme.php` contient les trois fonctions suivantes :

La fonction « charger » liste les variables qui seront envoyées dans l'environnement du squelette et initialise leurs valeurs par défaut. Ici aucune date par défaut n'est définie, mais il serait possible d'en indiquer une.

```
function formulaires_calculer_quantieme_charger_dist (){
    $valeurs = array(
        'date_jour' => ''
    );
    return $valeurs;
}
```

La fonction « vérifier » teste si tous les champs obligatoires sont renseignés et vérifie que le format de date semble correct :

```
function formulaires_calculer_quantieme_verifier_dist (){
    $erreurs = array();
```

```
    // champs obligatoires
    foreach(array ('date_jour') as $obligatoire) {
        if (!_request($obligatoire)) $erreurs[$obligatoire] =
'Ce champ est obligatoire';
    }
    // format de date correct
    if (!isset($erreurs['date_jour'])) {
        list($jour, $mois, $annee) = explode('/',
_request('date_jour'));
        if (!intval($jour) or !intval($mois) or
!intval($annee)) {
            $erreurs['date_jour'] = "Ce format de date n'est
pas reconnu.";
        }
    }
    if (count($erreurs)) {
        $erreurs['message_erreur'] = 'Votre saisie contient
des erreurs !';
    }
    return $erreurs;
}
```

Si les vérifications sont correctes (aucune erreur), la fonction « traiter » est exécutée. Le formulaire est déclaré ré-éditable, ce qui permet de saisir une nouvelle valeur de date aussitôt après la validation.

```
function formulaires_calculer_quantieme_traiter_dist (){
    $date_jour    = _request('date_jour');
    $retour = array('editable' => true);
    if ($quantieme = calcule_quantieme($date_jour)) {
        $retour['message_ok'] = "Le quantième de $date_jour
est $quantieme";
    } else {
        $retour['message_erreur'] = "Erreur lors du calcul du
quantième !";
    }
    return $retour;
}
```

Bien sûr il manque la fonction qui permet de trouver le quantième, mais deux petites lignes de PHP suffiront. Cette fonction peut être mise dans le même fichier que les trois fonctions précédentes :

```
function calcule_quantieme($date_jour) {
    list($jour, $mois, $annee) = explode('/', $date_jour);
    if ($time = mktime( 0, 0, 0, $mois, $jour, $annee)) {
        return date('z', $time);
    }
    return false;
}
```

Traducteur de blabla

Cet autre exemple simple va créer un petit formulaire demandant à un service externe de traduire un contenu qui lui est envoyé. Le résultat sera affiché sous le texte saisi.

Le formulaire sera nommé « traduire_blabla » et pourra donc être appelé dans un squelette par la balise `#FORMULAIRE_TRADUIRE_BLABLA` ou dans un article par `<formulaire|traduire_blabla>`.

Il fonctionne comme la plupart des formulaires CVT avec deux fichiers :

- `formulaires/traduire_blabla.html` pour la partie HTML
- `formulaires/traduire_blabla.php` pour les fonctions d'analyse et de traitement PHP.

Squelette HTML

Le squelette du formulaire recevra deux champs de saisie de type `textarea` : le premier pour écrire le contenu à traduire, le second pour afficher le résultat de la traduction une fois le calcul effectué. Ce second champ n'est affiché que s'il est renseigné.

```
<div class="formulaire_spip formulaire_#FORM">
[<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{traduire})]})]
```

```
        <li class="editer_traduire obligatoire[
(#GET{erreurs}|oui)erreur]">
            <label for="traduire">Traduire</label>
            [<span
class='erreur_message'>(#GET{erreurs})</span>]
            <textarea name='traduire'
id='champ_traduire'>#ENV{traduire}</textarea>
        </li>
        [

[(#SET{erreurs,[(#ENV**{erreurs}|table_valeur{traduction})]})]
        <li class="editer_traduction[
(#GET{erreurs}|oui)erreur]">
            <label for="traduction">Traduction</label>
            [<span
class='erreur_message'>(#GET{erreurs})</span>]
            <textarea name='traduction'
id='champ_traduction'>(#ENV{traduction})</textarea>
        </li>
        ]
    </ul>
    <p class="boutons"><input type="submit" class="submit"
value="Traduire" /></p>
</div></form>
</div>
```

Les deux champs se nomment « traduire » et « traduction ». Le même squelette pourrait écrire avec le plugin « Saisies » le contenu entre `<ul>` et `</ul>` de la sorte :

```
<ul>
    [(#SAISIE{textarea, traduire, obligatoire=oui,
label=Traduire})]

    [(#ENV{traduction}|oui)
        [(#SAISIE{textarea, traduction, label=Traduction})]
    ]
</ul>
```

Chargement, vérifications et traitements

La fonction « charger » du formulaire, déclarée dans le fichier `formulaires/traduire_blabla.php` doit indiquer qu'elle ajoute ces deux champs « traduire » et « traduction » dans le contexte du squelette :

```
function formulaires_traduire_blabla_charger_dist() {
    $contexte = array(
        'traduire' => '',
        'traduction' => '',
    );
    return $contexte;
}
```

La fonction « vérifier » a simplement besoin de tester si du contenu a bien été saisi dans le champ « traduire » et dans le cas contraire retourner une erreur :

```
function formulaires_traduire_blabla_verifier_dist() {
    $erreurs = array();
    if (!_request('traduire')) {
        $erreurs['message_erreur'] = "Vous avez oublié
d'écrire ! Votre clavier est cassé ?";
        $erreurs['traduire'] = "C'est là dedans qu'on écrit
son texte !";
    }
    return $erreurs;
}
```

C'est à partir de la fonction « traiter » que les choses se compliquent un peu. Il va falloir envoyer le contenu à un service distant (on utilisera *Google Translate* dans cet exemple), récupérer et traiter l'information retournée, puis la faire afficher dans le formulaire.

Pour ce faire, le script commence par calculer l'URL du service distant basée sur l'API de celui-ci. On utilise la fonction PHP de SPIP `parametre_url` pour ajouter proprement des variables à l'URL du service. Grâce à une autre fonction, `recuperer_page` qui permet de récupérer le code retourné par l'appel d'une URL, le retour du service est stocké dans la variable `$trad`.

Le service retournant des données formatées en JSON, il faut les décortiquer (fonction `json_decode`). En fonction du retour, la traduction est réussie ou non. Le message est adapté en conséquence.

```
// http://ajax.googleapis.com/ajax/services/language/
translate?v=1.0&q=hello%20world&langpair=en%7Cit
define('URL_GOOGLE_TRANSLATE', "http://ajax.googleapis.com/
ajax/services/language/translate");
```

```php
function formulaires_traduire_blabla_traiter_dist() {
    // creer l'url selon l'api google
    $texte = _request('traduire');
    $url = parametre_url(URL_GOOGLE_TRANSLATE, 'v', '1.0',
'&');
    $url = parametre_url($url, 'langpair', 'fr|en', '&');
    $url = parametre_url($url, 'q', $texte, '&');
    // chargement du texte traduit par google (retour : json)
    include_spip('inc/distant');
    $trad = recuperer_page($url);
    // attention : PHP 5.2
    $trad = json_decode($trad, true); // true = retour array
et non classe
    // recuperation du resultat si OK
    if ($trad['responseStatus'] != 200) {
        set_request('traduction', '');
        return array(
            "editable" => true,
            "message_erreur" => "Pas de chance, faux retour
de l'ami Google !"
        );
    }
    // envoi au charger
    set_request('traduction',
$trad['responseData']['translatedText']);
    // message
    return array(
        "editable" => true,
        "message_ok" => "Et voilà la traduction !",
    );
}
```

La fonction `set_request()` force le stockage d'une valeur de variable qui pourra être récupérée par `_request()`. Ainsi le prochain chargement du formulaire peut récupérer la valeur du champs « traduction » pour l'envoyer dans le contexte du squelette.

Note : Il est possible qu'une méthode plus propre soit développée dans de prochaines versions de SPIP pour faire transiter des données entre le traitement et le chargement via un nouveau paramètre au tableau de retour du traitement.

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 standard 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 declarer_tables_principales (p.149) et declarer_tables_auxiliaires (p.140)), 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'));
```

API SQL

Les fonctions d'abstraction SQL de SPIP forment une API dont voici les fonctions :

Nom	Description
Éléments communs (p.256)	Paramètres et options systématiques.
sql_allfetsel (p.257)	Retourne un tableau de l'ensemble des résultats d'une sélection
sql_alltable (p.259)	Retourne un tableau des tables SQL présentes
sql_alter (p.259)	Modifier la structure d'une table SQL
sql_count (p.261)	Compte le nombre de ligne d'une ressource de sélection
sql_countsel (p.262)	Compter un nombre de résultat
sql_create (p.263)	Crée une table selon le schéma indiqué.
sql_create_base (p.265)	Crée une base de donnée
sql_create_view (p.265)	Crée une vue
sql_date_proche (p.266)	Retourne une expression de calcul de date
sql_delete (p.267)	Supprime des éléments.
sql_drop_table (p.268)	Supprime une table !
sql_drop_view (p.269)	Supprime une vue.
sql_errno (p.269)	Retourne le numéro de la dernière erreur SQL
sql_error (p.270)	Retourne la dernière erreur SQL
sql_explain (p.270)	Explicite comment le serveur SQL va traiter une requête
sql_fetch (p.270)	Retourne une ligne d'une ressource de sélection
sql_fetch_all (p.273)	Retourne un tableau de tous les résultats

Nom	Description
sql_fetsel (p.273)	Sélectionne et retourne la première ligne des résultats.
sql_free (p.274)	Libère une ressource
sql_getfetsel (p.275)	Récupère l'unique colonne demandée de la première ligne d'une sélection.
sql_get_charset (p.276)	Demande si un codage de nom donné est disponible sur le serveur.
sql_get_select (p.276)	Retourne la requête de sélection
sql_hex (p.278)	Retourne une expression numérique d'une chaine hexadécimale
sql_in (p.279)	Construit un appel à l'opérande IN.
sql_insert (p.280)	Insérer du contenu
sql_insertq (p.281)	Insérer du contenu (protégé automatiquement).
sql_insertq_multi (p.282)	Permet d'insérer plusieurs lignes en une opération.
sql_in_select (p.283)	Effectue un `sql_in` sur le résultat d'un `sql_select`.
sql_listdbs (p.285)	Listes les bases de données disponibles sur une connexion donnée
sql_multi (p.285)	Extrait un contenu multilingue.
sql_optimize (p.286)	Optimise une table donnée.
sql_query (p.287)	Exécute une requête donnée.
sql_quote (p.287)	Protège une chaine.
sql_repair (p.289)	Répare une table endomagée.
sql_replace (p.289)	Insère ou modifie une entrée
sql_replace_multi (p.290)	Insérer ou remplacer plusieurs entrées.
sql_seek (p.291)	Place une ressource de sélection sur le numéro de ligne désigné.
sql_select (p.291)	Sélectionne des contenus

Nom	Description
sql_selectdb (p.294)	Sélectionne la base de données demandée.
sql_serveur (p.295)	Fonction principale et transparente de l'API
sql_set_charset (p.296)	Demande d'utiliser le codage indiqué.
sql_showbase (p.296)	Retourne une ressource de la liste des tables
sql_showtable (p.297)	Retourne une description de la table.
sql_update (p.298)	Met à jour un enregistrement.
sql_updateq (p.299)	Actualise un contenu (et protège les données).
sql_version (p.300)	Retourne la version du gestionnaire de base de données

Éléments communs

Dans le jeu de fonctions `sql_*` certains paramètres sont présents systématiquement et signifient la même information. Voici donc ces paramètres pour ne pas répéter maintes fois les mêmes choses :

- `$serveur` (ou `$connect`) est le nom du fichier de connexion SQL (dans le répertoire `config/`. Non renseigné ou vide, c'est le fichier de connexion défini à l'installation de SPIP qui est utilisé. C'est en général l'avant dernier paramètre des fonctions d'abstractions SQL.
- `$options` vaut `true` par défaut et permet d'indiquer un caractère optionnel par son intermédiaire. Ce paramètre est en général le dernier des fonctions d'abstractions SQL. Il admet :
  - `true` : toute fonction dans l'API SQL et non trouvée sur le jeu d'instruction SQL du serveur demandé provoquera une erreur fatale.
  - `'continue'` : pas d'erreur fatale si la fonction n'est pas trouvée.
  - et `false` : la fonction du jeu SQL n'exécute pas la requête quelle aura calculée, mais doit la retourner (on obtient donc une chaine de caractère qui est une requête SQL valide pour le gestionnaire de base de données demandé).

D'autres paramètres sont parfois présents d'une fonction à l'autre, en particulier pour toutes les fonctions qui s'apparentent à `sql_select()` en reprenant tout ou partie de ses paramètres :

- `$select`, tableau des colonnes sql à récupérer,
- `$from`, tableau des tables SQL à utiliser,
- `$where`, tableau de contraintes sur les colonnes où chaque élément du tableau sera agrégé avec `AND`,
- `$groupby`, tableau les groupements des résultats,
- `$orderby`, tableau définissant l'ordonnancement des résultats,
- `$limit`, chaine indiquant le nombre de résultat à obtenir,
- `$having` tableau de post-contraintes pour les fonctions d'agrégation.

Dans les fonctions de modification de contenu, on rencontre un paramètre commun :

- `$desc`, est un tableau de description des colonnes de la table SQL utilisée. S'il est omis, la description sera calculée automatiquement si les fonctions de portage en ont besoin.

Principes d'écriture

Un grand nombre de paramètres sont tolérants dans le type d'argument qui leur est envoyé, acceptant tableaux ou chaînes de caractères. C'est le cas par exemple des paramètres de `sql_select()`. Son premier paramètre est `$select` correspondant à la liste des colonnes SQL à récupérer. Voici 4 écritures fonctionnelles pour ce paramètre :

```
// 1 element
sql_select('id_article', 'spip_articles');
sql_select(array('id_article'), 'spip_articles');
// 2 elements
sql_select('id_article, titre', 'spip_articles');
sql_select(array('id_article', 'titre'), 'spip_articles');
```

Par convention, qui n'a rien d'obligatoire, on préférera utiliser l'écriture tabulaire dès qu'il y a plus d'un élément, écriture plus facilement analysable par les fonctions traduisant l'écriture abstraite en requête SQL.

sql_allfetsel

La fonction `sql_allfetsel()` récupère un tableau de l'ensemble des résultats d'une sélection. Elle a les mêmes paramètres que `sql_select()` et est un raccourcis de la combinaison `sql_select()` + `sql_fetch_all()`. Stockant tous les résultats dans un tableau PHP, il faut faire attention à ne pas dépasser la taille mémoire allouée à PHP si l'on traite de volumineux contenus.

Elle possède 9 paramètres :

1. `$select`,
2. `$from`,
3. `$where`,
4. `$groupby`,
5. `$orderby`,
6. `$limit`,
7. `$having`,
8. `$serveur`,
9. `$option`.

La fonction `sql_allfetsel()` s'utilise comme cela :

```
$all = sql_allfetsel('colonne', 'table');
// $all[0]['colonne'] est la colonne de la premiere ligne
recue
```

Exemple

Sélectionner tous les couples `objet` / `id_objet` liés à un document donné :

```
if ($liens = sql_allfetsel('objet, id_objet',
'spip_documents_liens', 'id_document=' . intval($id))) {
    foreach ($liens as $l) {
        // $l['objet'] et $l['id_objet']
    }
}
```

Le plugin « Contact avancé » sélectionne tous les emails des destinataires d'un message comme ceci :

```
// On recupere a qui ça va etre envoye
$destinataire = _request('destinataire');
if (!is_array($destinataire)) {
    $destinataire = array($destinataire);
}
$destinataire = array_map('intval', $destinataire);
$mail = sql_allfetsel('email', 'spip_auteurs',
sql_in('id_auteur', $destinataire));
```

sql_alltable

La fonction `sql_alltable()` retourne un tableau listant les différentes tables SQL présentes dans la base de données. Elle prend les mêmes paramètres que sql_showbase (p.296) :

1. `$spip` vide par défaut, il permet de ne lister que les tables utilisant le préfixe défini pour les tables SPIP. Utiliser `'%'` pour lister toutes les tables,
2. `$serveur`,
3. `$option`.

Utilisation :

```
$tables = sql_alltable();
sort($tables);
// $tables[0] : spip_articles
```

sql_alter

La fonction `sql_alter()` permet d'envoyer une requête de type `ALTER` au serveur de base de données pour modifier la structure de la base.

La fonction a 3 paramètres :

1. `$q` est la requête (sans le terme "ALTER") a effectuer
2. `$serveur`,
3. `$option`

Note : Cette fonction prenant directement une requête au format SQL, il est prudent de bien respecter les standards SQL. Il est possible que pour des prochaines versions de SPIP, le paramètre `$q` accepte un tableau plus structuré en entrée pour faciliter les différents portages.

La fonction s'utilise comme suit :

```
sql_alter("TABLE table ADD COLUMN colonne INT");
sql_alter("TABLE table ADD colonne INT"); // COLUMN est
optionnel
sql_alter("TABLE table CHANGE colonne colonne INT DEFAUT
'0'");
sql_alter("TABLE table ADD INDEX colonne (colonne)");
sql_alter("TABLE table DROP INDEX colonne");
sql_alter("TABLE table DROP COLUMN colonne");
sql_alter("TABLE table DROP colonne"); // COLUMN est
optionnel
// possibilite de passer plusieurs actions, mais attention
aux portages :
sql_alter("TABLE table DROP colonneA, DROP colonneB");
```

La fonction `sql_alter()` sert particulièrement lors de la phase de mise à jour de plugins dans les fonctions `{nom}_upgrade()` des différents plugins.

Exemple

Ajouter une colonne « composition » sur la table `spip_articles` (plugin « Composition ») :

```
sql_alter("TABLE spip_articles ADD composition
varchar(255) DEFAULT '' NOT NULL");
```

Ajouter une colonne « css » sur la table « spip_menus » (plugin « Menus ») :

```
sql_alter("TABLE spip_menus ADD COLUMN css tinytext
DEFAULT '' NOT NULL");
```

Le plugin « TradRub » dans sa procédure d'installation ajoute une colonne « id_trad » sur la table `spip_rubriques` en utilisant la fonction `maj_tables()` prévue, puis ajoute un index sur cette même colonne avec `sql_alter()` :

```
function tradrub_upgrade($nom_meta_base_version,
$version_cible){
    $current_version = 0.0;
    if (
(!isset($GLOBALS['meta'][$nom_meta_base_version]) )
        || (($current_version =
$GLOBALS['meta'][$nom_meta_base_version]) !=
$version_cible))
    {
        include_spip('base/tradrub');
        if ($current_version==0.0){
            include_spip('base/create');
            maj_tables('spip_rubriques');
            // index sur le nouveau champ
            sql_alter("TABLE spip_rubriques ADD INDEX
(id_trad)");
            ecrire_meta($nom_meta_base_version,
$current_version=$version_cible, 'non');
        }
    }
}
```

sql_count

La fonction `sql_count()` retourne le nombre de ligne d'une ressource de sélection obtenue avec `sql_select()`.

Elle possède 3 paramètres :

1. `$res` est la ressource d'une sélection,
2. `$serveur`,
3. `$option`.

Elle s'emploie ainsi :

```
$res = sql_select('colonne', 'table');
```

```
if ($res and sql_count($res)>2) {
    // il y a au moins 3 lignes de resultat !
}
```

Exemple

Usage possible : afficher un compteur sur le nombre total d'élément.

```
if ($res = sql_select('titre', 'spip_rubriques',
'id_parent=0')) {
    $n = sql_count($res);
    $i = 0;
    while ($r = sql_fetch($res)) {
        echo "Rubrique " . ++$i . " / $n : $r[titre]<br
/>";
        // Rubrique 3 / 12 : La fleur au vent
    }
}
```

sql_countsel

La fonction `sql_countsel()` retourne le nombre de lignes d'une sélection demandée. C'est un raccourci d'écriture à peu près équivalent à `sql_select('COUNT(*)', ...)`.

Elle prend les mêmes arguments que `sql_select()` moins le premier :

1. `$from`,
2. `$where`,
3. `$groupby`,
4. `$orderby`,
5. `$limit`,
6. `$having`,
7. `$serveur`,
8. `$option`.

Elle s'utilise comme ceci :

```
$nombre = sql_countsel("table");
```

Exemple

Compter le nombre de mots d'un groupe de mot donné :

```
$groupe = sql_countsel("spip_mots",
"id_groupe=$id_groupe");
```

Retourner `false` s'il y a des articles dans une rubrique :

```
if (sql_countsel('spip_articles', array(
    "id_rubrique=$id_rubrique",
    "statut <> 'poubelle'"
))) {
    return false;
}
```

Si la table `spip_notations_objets` du plugin « Notations » ne contient pas encore d'entrée pour l'identifiant d'objet indiqué, on effectue une insertion dans la base, sinon une mise à jour :

```
// Mise a jour ou insertion ?
if (!sql_countsel("spip_notations_objets", array(
    "objet=" . sql_quote($objet),
    "id_objet=" . sql_quote($id_objet),
))) {
    // Remplir la table de notation des objets
    sql_insertq("spip_notations_objets", ...);
    // ...
} else {
    // Mettre ajour dans les autres cas
    sql_updateq("spip_notations_objets", ...);
    // ...
}
```

sql_create

La fonction `sql_create()` permet de créer une table SQL selon le schéma indiqué.

Elle accepte 7 paramètres :

- `$nom` est le nom de la table à créer
- `$champs` est un tableau de description des colonnes
- `$clefs` est un tableau de description des clefs
- `$autoinc` : si un champ est clef primaire est numérique alors la propriété d'autoincrémentation sera ajoutée. `false` par défaut.
- `$temporary` : est-ce une table temporaire ? par défaut : `false`
- `$serveur`,
- `$option`

Elle s'utilise comme ceci :

```
sql_create("spip_tables",
    array(
        "id_table" => "bigint(20) NOT NULL default '0'",
        "colonne1"=> "varchar(3) NOT NULL default 'oui'",
        "colonne2"=> "text NOT NULL default ''"
    ),
    array(
        'PRIMARY KEY' => "id_table",
        'KEY colonne1' => "colonne1"
    )
);
```

En règle général, pour un plugin, il vaut mieux déclarer la table SQL un utilisant les pipelines adéquats declarer_tables_principales (p.149) et declarer_tables_auxiliaires (p.140), et utiliser les fonctions `creer_base()` ou `maj_tables('spip_tables')` lors de l'installation du plugin, qui appelleront si besoin `sql_create()`. Lire à ce titre le passage « Mises à jour et installation des tables (p.252) ».

Exemple

Exemple de création d'une table « spip_mots_tordus » qui serait une liaison avec « spip_tordus ». Remarquer la clé primaire composée des 2 colonnes :

```
sql_create("spip_mots_tordus",
    array(
        "id_mot" => "bigint(20) NOT NULL default '0'",
```

```
        "id_tordu"=> "bigint(20) NOT NULL default '0'"
    ),
    array(
        'PRIMARY KEY' => "id_tordu,id_mot"
    )
);
```

sql_create_base

La fonction `sql_create_base()` tente de créer une base de données dont le nom est donné. La fonction retourne `false` en cas d'erreur.

Elle possède 3 paramètres :

- `$nom` est le nom de la base à créer,
- `$serveur`,
- `$option`

Cette fonction sert uniquement lors de l'installation de SPIP pour créer une base de données demandée pour un gestionnaire de base de données :

```
sql_create_base($sel_db, $server_db);
```

Dans le cas de SQLite, le nom de la base de données correspond au nom du fichier sans l'extension (`.sqlite` sera ajouté) et le fichier sera enregistré dans le répertoire défini par la constante `_DIR_DB` qui est par défaut `config/bases/`

sql_create_view

La fonction `sql_create_view()` crée une vue à partir d'une requête de sélection donnée. La vue pourra donc être utilisée par des boucles SPIP ou par de nouvelles requêtes de sélection.

Elle admet 4 paramètres :

1. `$nom` est le nom de la vue créée,
2. `$select_query` est la requête de sélection,
3. `$serveur`,

4. `$option`.

On peut l'utiliser couplée à la fonction sql_get_select (p.276) pour obtenir la sélection voulue :

```
$selection = sql_get_select('colonne', 'table');
sql_create_view('vue', $selection);
// utilisation
$result = sql_select('colonne', 'vue');
```

Note : Lorsqu'une colonne de sélection utilise une notation `'nom.colonne'`, il faut impérativement déclarer un alias pour la colonne sinon certains portages (SQLite notamment) ne créent pas la vue attendue, par exemple `'nom.colonne AS colonne'`.

Exemple

Ce petit exemple montre le fonctionnement, en créant une table (bien inutile) à partir de 2 colonnes d'une rubrique :

```
$select = sql_get_select(array(
        'r.titre AS t',
        'r.id_rubrique AS id'
    ), array(
        'spip_rubriques AS r'
    ));
// creer la vue
sql_create_view('spip_short_rub', $select);
// utiliser :
$titre = sql_getfetsel('t', 'spip_short_rub', 'id=8');
```

Dans un squelette, la vue pourrait aussi être utilisée :

```
<BOUCLE_vue(spip_short_rub) {id=8}>
    <h3>#T</h3>
</BOUCLE_vue>
```

sql_date_proche

La fonction `sql_date_proche()` permet de retourner une expression de condition d'une colonne par rapport à une date.

Elle prend 5 paramètres :

1. `$champ` est la colonne SQL à comparer,
2. `$interval` est la valeur de l'interval de comparaison : -3, 8, ...
3. `$unite` est l'unité de référence ('DAY', 'MONTH', 'YEAR', ...)
4. `$serveur`,
5. `$option`.

Elle s'utilise comme ceci :

```
$ifdate = sql_date_proche('colonne', -8, 'DAY');
$res = sql_select('colonne', 'table', $ifdate);
```

Exemple

Une autre utilisation dans une clause de sélection comme ci-dessous, est de stocker le résultat booléen dans un alias. L'alias `ici` indique si oui ou non un auteur s'est connecté les 15 derniers jours :

```
$row = sql_fetsel(
    array("*", sql_date_proche('en_ligne', -15, 'DAY') .
" AS ici"),
    "spip_auteurs",
    "id_auteur=$id_auteur");
// $row['ici'] : true / false
```

sql_delete

La fonction `sql_delete()` permet de supprimer des entrées dans une table SQL et retourne le nombre de suppressions réalisées.

Elle possède 4 paramètres :

1. `$table` est le nom de la table SQL,
2. `$where`,
3. `$serveur`,

4. `$option`.

Elle s'utilise comme ceci :

```
sql_delete('table', 'id_table = ' . intval($id_table));
```

Exemple

Supprimer la liaison entre des rubriques et un mot donné :

```
sql_delete("spip_mots_rubriques", "id_mot=$id_mot");
```

Une des taches périodiques de SPIP supprime les vieux articles mis à la poubelle comme ceci :

```
function optimiser_base_disparus($attente = 86400) {
    $mydate = date("YmdHis", time() - $attente);
    // ...
    sql_delete("spip_articles", "statut='poubelle' AND
maj < $mydate");
}
```

sql_drop_table

La fonction `sql_drop_table()` supprime une table SQL de la base de données. Elle retourne `true` en cas de réussite, `false` sinon.

Elle accepte 4 paramètres :

1. `$table` est le nom de la table,
2. `$exist` permet de demander à ajouter une vérification sur l'existence de la table lors de la suppression (cela se traduit par l'ajout de `IF EXISTS` sur la requête). Par défaut `''`, mettre `true` pour vérifier,
3. `$serveur`,
4. `$option`.

Cette fonction `sql_drop_table()` s'écrit :

```
sql_drop_table('table');
```

```
sql_drop_table('table', true);
```

Exemple

Les plugins utilisent souvent cette fonction lors de la suppression complète (données comprises) d'un plugin, comme le plugin « Géographie » :

```
function geographie_vider_tables($nom_meta_base_version)
{
    sql_drop_table("spip_geo_pays");
    sql_drop_table("spip_geo_regions");
    sql_drop_table("spip_geo_departements");
    sql_drop_table("spip_geo_communes");
    effacer_meta($nom_meta_base_version);
    ecrire_metas();
}
```

sql_drop_view

La fonction `sql_drop_view()` supprime une vue. Elle prend les mêmes paramètres que `sql_drop_table()` et retourne `true` en cas de succès et `false` sinon.

Ses 4 paramètres sont :

1. `$table` est le nom de la table,
2. `$exist` permet de demander à ajouter une vérification sur l'existence de la table lors de la suppression (cela se traduit par l'ajout de `IF EXISTS` sur la requête). Par défaut `''`, mettre `true` pour vérifier,
3. `$serveur`,
4. `$option`.

La fonction `sql_drop_view()` s'utilise ainsi :

```
sql_drop_view('vue');
sql_drop_view('vue', true);
```

sql_errno

La fonction `sql_errno()` retourne le numéro de la dernière erreur SQL rencontrée. Cette fonction est utilisée par SPIP pour remplir automatiquement des logs d'incidents relatifs à SQL, centralisés dans la fonction `spip_sql_erreur()` de ecrire/base/connect_sql.php

sql_error

La fonction `sql_error()` retourne la dernière erreur SQL rencontrée. Cette fonction est utilisée par SPIP pour remplir automatiquement des logs d'incidents relatifs à SQL, centralisés dans la fonction `spip_sql_erreur()` de ecrire/base/connect_sql.php

sql_explain

La fonction `sql_explain()` permet de retourner une explication de comment le serveur SQL va traiter une requête. Cette fonction est utilisée par le débusqueur (le mode debug) pour donner des informations sur les requêtes générées.

La fonction accepte 3 paramètres :

1. `$q` est la requête,
2. `$serveur`,
3. `$option`.

Un usage peut être :

```
$query = sql_get_select('colonne', 'table');
$explain = sql_explain($query);
```

sql_fetch

La fonction `sql_fetch()` retourne une ligne, sous forme d'un tableau associatif, d'un résultat d'une sélection. Elle retourne `false` s'il n'y a plus de ligne à afficher.

Elle admet 3 paramètres, seul le premier est indispensable :

1. `$res` est la ressource obtenue avec sql_select(),
2. `$serveur`,
3. `$option`.

Elle s'utilise en partenariat étroit avec `sql_select()`, souvent employé dans cette association :

```
if ($res = sql_select('colonne', 'table')) {
    while ($r = sql_fetch($res)) {
        // utilisation des resultats avec $r['colonne']
    }
}
```

Exemple

Lister les articles proposés à publication :

```
$result = sql_select("id_article, id_rubrique, titre,
statut", "spip_articles", "statut = 'prop'", "", "date
DESC");
while ($row = sql_fetch($result)) {
    $id_article=$row['id_article'];
    if (autoriser('voir', 'article', $id_article)) {
        // actions
    }
}
```

Le plugin « Contact avancé » peut enregistrer des messages dans la table `spip_messages`. Au moment de la suppression d'un de ces messages, il supprime les éventuels documents qui lui sont liés comme ceci :

```
function action_supprimer_message() {
    $securiser_action =
charger_fonction('securiser_action', 'inc');
    $id_message = $securiser_action();
    // Verifions si nous avons un document
    if ($docs = sql_select('id_document',
'spip_documents_liens', 'id_objet=' . intval($id_message)
. ' AND objet="message"')) {
        include_spip('action/documenter');
```

```
        while ($id_doc = sql_fetch($docs)) {

supprimer_lien_document($id_doc['id_document'],
"message", $id_message);
        }
    }
    sql_delete("spip_messages", "id_message=" .
sql_quote($id_message));
    sql_delete("spip_auteurs_messages", "id_message=" .
sql_quote($id_message));
}
```

La fonction `calculer_rubriques_publiees()` dans `ecrire/inc/rubriques.php` permet de recalculer les statuts et dates de rubriques pour savoir lesquelles ont le statut « publié ». Dedans, une partie sélectionne les rubriques qui ont des documents publiés (la rubrique l'est alors aussi) et attribue à une colonne temporaire le nouveau statut et la nouvelle date. Une fois toutes les mises à jour faites, la colonne temporaire est enregistrée dans la véritable colonne :

```
// Mettre les compteurs a zero
sql_updateq('spip_rubriques', array(
    'date_tmp' => '0000-00-00 00:00:00',
    'statut_tmp' => 'prive'));
// [...]
// Publier et dater les rubriques qui ont un *document*
publie
$r = sql_select(
    array(
        "rub.id_rubrique AS id",
        "max(fille.date) AS date_h"),
    array(
        "spip_rubriques AS rub",
        "spip_documents AS fille",
        "spip_documents_liens AS lien"),
    array(
        "rub.id_rubrique = lien.id_objet",
        "lien.objet='rubrique'",
        "lien.id_document=fille.id_document",
        "rub.date_tmp <= fille.date",
        "fille.mode='document'", "rub.id_rubrique"));
while ($row = sql_fetch($r)) {
```

```
      sql_updateq('spip_rubriques',
        array(
            'statut_tmp'=>'publie',
            'date_tmp'=>$row['date_h']),
        "id_rubrique=" . $row['id']);
}
// [...]
// Enregistrement des modifs
sql_update('spip_rubriques', array(
    'date'=>'date_tmp',
    'statut'=>'statut_tmp'));
```

sql_fetch_all

La fonction `sql_fetch_all()` retourne un tableau contenant toutes les lignes d'une ressource de sélection. Comme tous les résultats seront présents en mémoire il faut faire attention à ne pas sélectionner un contenu trop volumineux.

La fonction `sql_fetch_all()` prend 3 paramètres :

1. `$res` est la ressource obtenue avec sql_select(),
2. `$serveur`,
3. `$option`.

Elle s'emploie ainsi :

```
$res = sql_select('colonne', 'table');
$all = sql_fetch_all($res);
// $all[0]['colonne'] est le premier resultat
```

Cependant cette fonction est peu utilisée au profit de la fonction `sql_allfetsel()` qui effectue la même opération directement avec les paramètres de sélection :

```
$all = sql_allfetsel('colonne', 'table');
// $all[0]['colonne'] est le premier resultat
```

sql_fetsel

La fonction `sql_fetsel` retourne le premier résultat d'une sélection. Elle prend les mêmes paramètres que `sql_select()` et est un raccourcis de la combinaison `sql_select()` + `sql_fetch()`.

Ses paramètres sont donc :

1. `$select`,
2. `$from`,
3. `$where`,
4. `$groupby`,
5. `$orderby`,
6. `$limit`,
7. `$having`,
8. `$serveur`,
9. `$option`.

Elle s'utilise ainsi :

```
$r = sql_fetsel('colonne', 'table');
// $r['colonne']
```

Exemple

Sélectionner les colonnes « id_trad » et « id_rubrique » d'un article donné :

```
$row = sql_fetsel("id_trad, id_rubrique",
"spip_articles", "id_article=$id_article");
// $row['id_trad'] et $row['id_rubrique']
```

Sélectionner toutes les colonnes d'une brève donnée :

```
$row = sql_fetsel("*", "spip_breves",
"id_breve=$id_breve");
```

sql_free

La fonction `sql_free()` permet de libérer une ressource SQL issue de `sql_select()`. Idéalement cette fonction devrait être appelée après chaque fin d'usage d'une ressource.

Elle possède 3 paramètres :

1. `$res` est la ressource d'une sélection,
2. `$serveur`,
3. `$option`.

La fonction `sql_free()` s'utilise ainsi :

```
$res = sql_select('colonne', 'table');
// traitements utilisant la fonction sql_fetch($res) ...
// puis clore la ressource
sql_free($res);
```

À noter que des fonctions de l'API appellent cette fonction automatiquement. C'est le cas de :

- `sql_fetsel` (et `sql_getfetsel`),
- `sql_fetch_all` (et `sql_allfetsel`),
- `sql_in_select`.

sql_getfetsel

La fonction `sql_getfetsel()` récupère l'unique colonne demandée de la première ligne d'une sélection. Elle a les mêmes paramètres que `sql_select()` et est un raccourcis de la combinaison `sql_fetsel()` + `array_shift()`.

Ses paramètres sont :

1. `$select` nommant la colonne souhaitée,
2. `$from`,
3. `$where`,
4. `$groupby`,
5. `$orderby`,
6. `$limit`,
7. `$having`,

8. `$serveur`,
9. `$option`.

Elle s'utilise comme ceci :

```
$colonne = sql_getfetsel('colonne', 'table', 'id_table=' .
intval($id_table));
```

Notons que déclarer un alias fonctionne aussi :

```
$alias = sql_getfetsel('colonne AS alias', 'table',
'id_table=' . intval($id_table));
```

Exemple

Obtenir le secteur d'une rubrique :

```
$id_secteur = sql_getfetsel("id_secteur",
"spip_rubriques", "id_rubrique=" . intval($id_rubrique));
```

Le plugin « Job Queue » qui gère une liste de tâche planifiées obtient la date d'une prochaine tâche à effectuer comme cela :

```
$date = sql_getfetsel('date', 'spip_jobs', '', '',
'date', '0,1');
```

sql_get_charset

La fonction `sql_get_charset()` permet de vérifier que l'utilisation d'un codage de caractère donné est possible sur le serveur de base de données.

`sql_get_charset()` admet trois paramètres dont seul le premier est indispensable :

1. `$charset` est le type de charset souhaité, tel que « utf8 »
2. `$serveur`,
3. `$options`.

sql_get_select

La fonction `sql_get_select()` retourne la requête de sélection demandée. C'est un alias de la fonction `sql_select()` mais qui envoie l'argument `$option` à `false`, de sorte la requête SQL au lieu d'être exécutée est retournée.

Elle prend les mêmes arguments que `sql_select()` hormis le dernier qui est renseigné par la fonction :

1. `$select`,
2. `$from`,
3. `$where`,
4. `$groupby`,
5. `$orderby`,
6. `$limit`,
7. `$having`,
8. `$serveur`

Elle s'utilise comme ceci :

```
$requete = sql_get_select('colonne', 'table');
// retourne "SELECT colonne FROM table" (avec MySQL)
```

On récupère ainsi une requête SQL valide pour le gestionnaire de base de données utilisé. Comme cette requête est propre, elle peut être utilisée directement par la fonction `sql_query()`, mais plus souvent, elle sert à créer des sous requêtes en association avec `sql_in()` :

```
// liste d'identifiants
$ids = sql_get_select('id_table', 'tableA');
// selection en fonction de cette selection
$resultats = sql_select('titre', 'tableB', sql_in('id_table',
$ids)));
```

Exemple

Pour obtenir tous les titres de rubriques dont les identifiants d'articles sont supérieurs à 200, une des méthodes possibles (on pourrait aussi utiliser une jointure) est d'utiliser `sql_get_select()` :

```
// creer la requete de selection donnant la liste des
rubriques
$ids = sql_get_select('DISTINCT(id_rubrique)',
'spip_articles', array('id_article > 200'));
// selectionner ces rubriques
$res = sql_select('titre', 'spip_rubriques',
sql_in('id_rubrique', $ids));
while ($r = sql_fetch($res)) {
    // afficher le titre.
    echo $r['titre'] . '<br />';
}
```

De façon bien plus complexe, on trouve des exemples dans certaines fonctions de critères, par exemple dans le critère `{noeud}` du plugin « SPIP Bonux » qui crée une sous requête pour récupérer la liste des objets ayant des enfants.

```
function critere_noeud_dist($idb, &$boucles, $crit) {
// [...]
// cette construction avec IN fera que le compilateur
demandera
// l'utilisation de la fonction sql_in()
$where = array("'IN'", "'$boucle->id_table." .
"$primary'", "'('.sql_get_select('$id_parent',
'$table_sql').')'");
if ($crit->not)
    $where = array("'NOT'", $where);
$boucle->where[]= $where;
}
```

sql_hex

La fonction `sql_hex()` retourne une expression numérique d'une chaine hexadécimale, transformant `09af` en `0x09af` (avec MySQL et SQLite). Cela sert essentiellement pour écrire un contenu hexadécimal dans une colonne SQL de type numérique.

Elle prend 3 paramètres :

1. `$val` est la chaîne à traduire,
2. `$serveur`,

3. `$option`.

Utilisation :

```
$hex = sql_hex('0123456789abcdef');
sql_updateq('table', array('colonne'=>$hex), 'id_table=' .
$id_table);
```

sql_in

La fonction `sql_in()` permet de créer une contrainte sur une colonne utilisant le mot clé `IN`. Elle se compose de 5 paramètres :

1. `$val` est le nom de la colonne,
2. `$valeurs` est la liste des valeurs, sous forme de tableau ou d'une chaine d'éléments séparés par des virgules. Elles seront protégées par `sql_quote` automatiquement,
3. `$not` permet de définir la négation. Vaut `''` par défaut ; mettre `'NOT'` pour réaliser un `NOT IN`,
4. `$serveur`,
5. `$option`.

On peut l'utiliser ainsi :

```
$vals = array(2, 5, 8);
// ou $vals = "2, 5, 8";
$ids = sql_in('id_table', $vals);
if ($res = sql_select('colonne', 'table', $in)) {
    // ...
}
```

Exemple

Le plugin « Tickets » utilise `sql_in()` pour obtenir le titre d'un ticket uniquement si celui-ci a un statut parmi ceux indiqués :

```
function
inc_ticket_forum_extraire_titre_dist($id_ticket){
    $titre = sql_getfetsel('titre', 'spip_tickets',
array(
```

```
        'id_ticket = ' . sql_quote($id_ticket),
        sql_in('statut', array('ouvert', 'resolu',
'ferme'))
    ));
    return $titre;
}
```

sql_insert

La fonction `sql_insert()` permet d'insérer du contenu dans la base de données. Les portages SQL peuvent rencontrer des problèmes sur l'utilisation de cette fonction et à ce titre, il faut utiliser la fonction `sql_insertq()` à la place. Cette fonction est présente uniquement pour assurer le support d'une restauration de vieilles sauvegardes et la transition d'anciens scripts.

La fonction admet 6 paramètres :

1. `$table` est la table SQL,
2. `$noms` est la liste des colonnes impactées,
3. `$valeurs` est la liste des valeurs à enregistrer,
4. `$desc`,
5. `$serveur`,
6. `$option`.

Utilisation :

```
sql_insert('table', '(colonne)', '(valeur)');
```

Exemple

Insérer une liaison d'un mot avec un article :

```
$id_mot = intval($id_mot);
$article = intval($article);
sql_insert("spip_mots_articles", "(id_mot, id_article)",
"($id_mot, $article)");
```

Exemple de migration vers `sql_insertq()` :

```
sql_insertq("spip_mots_articles", array(
    "id_mot" => $id_mot,
    "id_article" => $article));
```

sql_insertq

La fonction `sql_insertq()` permet de réaliser une insertion dans la base de données. Les valeurs transmises non numériques seront protégés par des fonctions adaptées à chaque gestionnaire de bases de données pour gérer les apostrophes. La fonction retourne si possible le numéro de l'identifiant de clé primaire inséré.

La fonction admet 5 paramètres :

1. `$table` est le nom de la table SQL,
2. `$couples` est un tableau associatif nom / valeur,
3. `$desc`,
4. `$serveur`,
5. `$option`.

Elle s'utilise simplement comme ceci :

```
$id = sql_insertq('table', array('colonne'=>'valeur'));
```

Exemple

Les fonctions `insert_xx()` tel que `insert_article()` présente dans ecrire/action/editer_article.php permettent de créer des insertions en base de données pour les objets concernés, en gérant les valeurs par défaut et en appelant le pipeline pre_insertion (p.167) concerné. Ces fonctions retournent l'identifiant alors créé.

Ces fonctions exécutent donc, après le pipeline `pre_insertion` la fonction `sql_insertq()`. Puis dans la foulée, si un auteur est identifié, l'article est lié à cet auteur :

```
$id_article = sql_insertq("spip_articles", $champs);
// controler si le serveur n'a pas renvoye une erreur
if ($id_article > 0 AND
$GLOBALS['visiteur_session']['id_auteur']) {
    sql_insertq('spip_auteurs_articles', array(
        'id_auteur' =>
$GLOBALS['visiteur_session']['id_auteur'],
        'id_article' => $id_article));
}
```

sql_insertq_multi

La fonction `sql_insertq_multi()` permet d'insérer en une opération plusieurs éléments au schéma identique dans une table de la base de données. Lorsque les portages le permettent, ils utilisent d'ailleurs une seule requête SQL pour réaliser l'ajout. Plus précisément une requête par lot de 100 éléments pour éviter des débordements de mémoire.

La fonction a les mêmes 5 paramètres que `sql_insertq()` mais le second paramètre est un tableau de tableau de couples et non les couples directement :

1. `$table` est le nom de la table SQL,
2. `$couples` est un tableau de tableau associatif nom / valeur,
3. `$desc`,
4. `$serveur`,
5. `$option`.

Les colonnes utilisées doivent impérativement être les mêmes pour toutes les insertions. Elle s'utilise comme ceci :

```
$id = sql_insertq_multi('table', array(
    array('colonne' => 'valeur'),
    array('colonne' => 'valeur2'),
    array('colonne' => 'valeur3'),
);
```

Exemple

Les recherches effectuées via SPIP stockent dans une table `spip_resultats` quelques éléments utilisés comme cache, en prenant soin de l'appliquer sur la connexion SQL en court. `$tab_couples` contient l'ensemble des données à insérer :

```
// inserer les resultats dans la table de cache des
resultats
if (count($points)){
    $tab_couples = array();
    foreach ($points as $id => $p){
        $tab_couples[] = array(
            'recherche' => $hash,
            'id' => $id,
            'points' => $p['score']
        );
    }
    sql_insertq_multi('spip_resultats', $tab_couples,
array(), $serveur);
}
```

Le plugin « Polyhierarchie » l'utilise aussi pour insérer la liste des rubriques nouvellement liées à un objet donné :

```
$ins = array();
foreach($id_parents as $p){
    if ($p) {
        $ins[] = array(
            'id_parent' => $p,
            'id_objet' => $id_objet,
            'objet' => $objet);
    }
    if (count($ins)) {
        sql_insertq_multi("spip_rubriques_liens", $ins,
"", $serveur);
    }
```

sql_in_select

La fonction `sql_in_select()` effectue un `sql_in` sur le résultat d'un `sql_select`.

Elle prend les mêmes arguments que `sql_select` avec un premier en plus :

1. `$in` est le nom de la colonne sur laquelle s'appliquera le `IN`,
2. `$select`,
3. `$from`,
4. `$where`,
5. `$groupby`,
6. `$orderby`,
7. `$limit`,
8. `$having`,
9. `$serveur`,
10. `$option`.

On peut l'exploiter ainsi :

```
$where = sql_in_select("colonne", "colonne", "tables",
"id_parent = $id_parent"));
// $where : colonne IN (3, 5, 7)
if ($res = sql_select('colonne', 'autre_table', $where)) {
    // ...
}
```

Cette fonction actuellement calcule les valeurs à intégrer dans le `IN` et ne crée donc pas une sous-requête. Dans l'avenir, elle sera peut être modifiée : si le gestionnaire de base de données utilisé accepte les sous-requêtes, alors c'est cela qui serait retourné de préférence, de sorte qu'on obtiendrait d'équivalent de :

```
$where = sql_in("colonne", sql_get_select( "colonne",
"tables", "id_parent = $id_parent"));
// $where : colonne IN (SELECT colonne FROM tables WHERE
id_parent=3)
```

Exemple

Supprimer toutes les liaisons entre un article et les mot-clés d'un groupe de mot donné :

```
sql_delete("spip_mots_articles", array(
        "id_article=" . $id_article,
         sql_in_select("id_mot", "id_mot", "spip_mots",
"id_groupe = $id_groupe"));
```

sql_listdbs

La fonction `sql_listdbs()` liste les différentes bases de données disponibles pour une connexion donnée. Elle retourne une ressource de sélection ou directement un tableau PHP des différentes bases de données (cas de SQLite).

Elle reçoit 2 paramètres :

1. `$serveur`,
2. `$option`.

SPIP se sert de cette fonction au moment de l'installation pour permettre de choisir lorsque c'est possible, une base de données parmi celles autorisées par le gestionnaire de base de données.

```
$result = sql_listdbs($server_db);
```

sql_multi

La fonction `sql_multi()` applique une expression SQL sur une colonne contenant un polyglotte (p.68) (`<multi>`) pour y extraire la partie correspondant à la langue indiquée. Elle retourne une chaine du type : `expression AS multi`. Cette opération sert essentiellement pour demander simultanément un tri sur cette colonne.

Elle accepte 4 paramètres :

1. `$sel` est le nom de la colonne,
2. `$lang` est le code de langue ('fr', 'es', ...),

3. `$serveur`,
4. `$option`

Elle s'utilise ainsi :

```
$multi = sql_multi('colonne', 'langue');
$select = sql_select($multi, 'table');
```

Notons que dans un squelette, le critère de boucle `{par multi xx}` où `xx` est le nom de la colonne à trier appelle aussi cette fonction pour trier selon la langue en cours.

Exemple

SPIP utilise cette fonction pour trier des listes selon le titre d'un élément, dans la langue du visiteur :

```
$select = array(
    'id_mot', 'id_groupe', 'titre', 'descriptif',
    sql_multi ("titre", $GLOBALS['spip_lang']));
if ($results = sql_select($select, 'spip_mots',
"id_groupe=$id_groupe", '', 'multi')) {
    while ($r = sql_fetch($results)) {
        // $r['titre'] $r['multi']
    }
}
```

Le plugin « Grappes » également :

```
$grappes = sql_allfetsel("*, ".sql_multi ("titre",
"$spip_lang"), "spip_grappes", "", "", "multi");
foreach ($grappes as $g) {
    // $g['multi']
}
```

sql_optimize

La fonction `sql_optimize()` permet d'optimiser une table SQL. Cette fonction est appelée par la fonction `optimiser_base_une_table()` qui est exécutée périodiquement par le cron. Se référer aux commandes `OPTIMIZE TABLE` ou `VACUUM` des gestionnaires SQL pour comprendre les détails des opérations réalisées.

La fonction a 3 paramètres :

1. `$table` est le nom de la table à optimiser,
2. `$serveur`,
3. `$option`.

Utilisation :

```
sql_optimize('table');
```

Note : SQLite, ne peut pas optimiser table par table, mais optimise toute sa base de donnée d'un coup. Dans son cas, si la fonction `sql_optimize()` est appelée plusieurs fois sur un même hit, l'opération n'est effectuée qu'au premier appel.

sql_query

La fonction `sql_query()` exécute la requête qui lui est transmise. C'est la fonction la moins portable des instructions SQL ; il faut donc éviter de l'utiliser au profit des autres fonctions plus spécifiques.

Elle a 3 paramètres :

1. `$ins` est la requête,
2. `$serveur`,
3. `$option`.

Utilisation :

```
$res = sql_query('SELECT * FROM spip_meta');
// mais on preferera :
$res = sql_select('*', 'spip_meta');
```

sql_quote

La fonction `sql_quote()` sert à protéger du contenu (apostrophes) afin d'éviter toute injection SQL. Cette fonction est très importante et doit être utilisée dès qu'un contenu provient d'une saisie utilisateur. Les fonctions `sql_insertq`, `sql_updateq`, `sql_replace` effectuent automatiquement cette protection pour les données insérées (mais pas pour les autres paramètres comme `$where` qu'il faut tout de même protéger).

Elle accepte 3 paramètres :

1. `$val` est l'expression à protéger,
2. `$serveur`,
3. `$type` optionnel, est le type de valeur attendu. Peut valoir `int` pour un entier.

Elle s'utilise comme cela :

```
$oiseau = sql_quote("L'oiseau");
$champ = sql_quote($champ);
sql_select('colonne', 'table', 'titre=' . sql_quote($titre));
sql_updateq('table', array('colonne'=>'valeur'), 'titre=' .
sql_quote($titre));
```

Lorsqu'un identifiant numérique est attendu, c'est souvent le cas des clés primaires, la protection peut être de simplement appliquer la fonction PHP `intval()` (La valeur zéro sera retournée si le contenu passé n'est pas numérique) :

```
$id_table = intval(_request('id_table'));
sql_select('colonne', 'table', 'id_table=' . intval($id));
```

Exemple

La fonction `url_delete()` supprime des URLs de la table SQL stockant les URLs des objets éditoriaux de SPIP. Elle protège des chaines avec `sql_quote()` et utilise `intval()` sur l'identifiant :

```
function url_delete($objet, $id_objet, $url=""){
    $where = array(
        "id_objet=" . intval($id_objet),
```

```
        "type=" . sql_quote($objet)
    );
    if (strlen($url)) {
        $where[] =  "url=" . sql_quote($url);
    }

    sql_delete("spip_urls", $where);
}
```

sql_repair

La fonction `sql_repair()` sert à réparer une table SQL endomagée. Elle est appelée par SPIP lorsqu'un administrateur tente de réparer une base de donnée sur la page `ecrire/?exec=admin_tech`.

Elle dispose de 3 paramètres : La fonction accepte 3 paramètres :

1. `$table` est la table à tenter de réparer,
2. `$serveur`,
3. `$option`.

Utilisation :

```
sql_repair('table');
```

Note : PostGres et SQLite ignorent cette instruction.

sql_replace

La fonction `sql_replace()` insère où met à jour une entrée d'une table SQL. La clé ou les cles primaires doivent être présentes dans les données insérés. La fonction effectue une protection automatique des données.

Il faut préférer les fonctions spécifiques `sql_insertq()` et `sql_updateq()` à cette fonction pour être plus précis, lorsque cela est possible.

Ses 5 paramètres sont :

1. `$table` est la table SQL utilisée,

2. `$couples` contient les couples colonne/valeur à modifier,
3. `$desc`,
4. `$serveur`,
5. `$option`.

On l'utilise tel que :

```
sql_replace('table', array(
    'colonne' => 'valeur',
    'id_table' => $id
));
```

sql_replace_multi

La fonction `sql_replace_multi()` permet d'insérer ou de remplacer plusieurs lignes (de même schéma) d'une table SQL en une opération. Les valeurs sont automatiquement protégées. Il est nécessaire que les colonnes des couples insérées contiennent la ou les clés primaires de la table.

Il faut préférer les fonctions spécifiques `sql_insertq_multi()` et `sql_updateq()` à cette fonction pour être plus précis, lorsque cela est possible.

Elle a les même 5 paramètres que sql_replace (p.289) :

1. `$table` est la table SQL utilisée,
2. `$couples` est un tableau des couples colonne/valeur à modifier,
3. `$desc`,
4. `$serveur`,
5. `$option`.

Elle s'utilise ainsi :

```
sql_replace_multi('table', array(
    array(
        'colonne' => 'valeur1',
        'id_table' => $id1
    ),
    array(
        'colonne' => 'valeur2',
        'id_table' => $id2
```

```
    )
));
```

sql_seek

La fonction `sql_seek()` place une ressource de sélection issu de `sql_select()` sur le numéro de ligne désigné.

Elle prend 4 paramètres :

1. `$res`, la ressource,
2. `$row_number`, le numéro de ligne,
3. `$serveur`,
4. `$option`.

Elle s'utilise ainsi :

```
if ($res = sql_select('colonne', 'table')) {
    if (sql_seek($res, 9)) { // aller au 10e
        $r = sql_fetch($res);
        // $r['colonne'] du 10e resultat
    }
    // replacer au tout debut
    sql_seek($res, 0);
}
```

sql_select

La fonction `sql_select()` sélectionne des contenus dans la base de données et retourne une ressource SQL en cas de réussite, ou `false` en cas d'erreur.

Elle dispose de 9 paramètres, les 2 premiers sont indispensables, placés dans l'ordre de description d'une requête SQL standard. Ils prennent en entrée (de préférence) un tableau, mais acceptent des chaînes de caractères dont les éléments sont séparés par des virgules :

1. `$select`,
2. `$from`,
3. `$where`,

4. `$groupby`,
5. `$orderby`,
6. `$limit`,
7. `$having`,
8. `$serveur`,
9. `$option`.

La fonction `sql_select()` est souvent couplée à `sql_fetch()` comme ceci :

```
// selection
if ($resultats = sql_select('colonne', 'table')) {
    // boucler sur les resultats
    while ($res = sql_fetch($resultats)) {
        // utiliser les resultats
        // $res['colonne']
    }
}
```

Les paramètres `$select` et `$from` acceptent de déclarer des alias. On peut donc imaginer :

```
if ($r = sql_select(
    array(
        'a.colonne AS colA',
        'b.colonne AS colB',
        'SUM(b.nombre) AS somme'
    ),
    array(
        'tableA AS a',
        'tableB AS b'
    ))) {
    while ($ligne = sql_fetch($r)) {
        // on peut utiliser :
        // $ligne['colA']  $ligne['colB']  $ligne['somme']
    }
}
```

Exemple

Sélectionner les rubriques racines (id_parent=0) de la table « spip_rubriques » triés par rang [1 (p.294)] puis par ordre alphanumérique, et en demander toutes les colonnes (sélection totale avec '*') :

```
$result = sql_select('*', "spip_rubriques",
"id_parent=0", '', '0+titre,titre');
while ($row = sql_fetch($result)){
    $id_rubrique = $row['id_rubrique'];
    // ...
}
```

Sélectionner les chats mais pas les chiens (dans le titre) pour les articles du secteur 3 :

```
$champs = array('titre', 'id_article', 'id_rubrique');
$where = array(
    'id_secteur = 3',
    'titre LIKE "%chat%" ',
    'titre NOT LIKE "%chien%"'
);
$result = sql_select($champs, "spip_articles", $where);
```

Sélectionner les titres et extensions connues pour les documents, et stocker cela dans un tableau :

```
$types = array();
$res = sql_select(array("extension", "titre"),
"spip_types_documents");
while ($row = sql_fetch($res)) {
    $types[$row['extension']] = $row;
}
```

Cette sélection pourrait aussi s'écrire :

```
$res = sql_select("extension, titre",
"spip_types_documents");
```

Sélectionner les documents liés à une rubrique, avec le titre de la rubrique en question, triés par date antichronologique :

```
$result = sql_select(
    array(
        "docs.id_document AS id_doc",
        "docs.extension AS extension",
        "docs.fichier AS fichier",
        "docs.date AS date",
        "docs.titre AS titre",
        "docs.descriptif AS descriptif",
        "R.id_rubrique AS id_rub",
        "R.titre AS titre_rub"),
    array(
        "spip_documents AS docs",
        "spip_documents_liens AS lien",
        "spip_rubriques AS R"),
    array(
        "docs.id_document = lien.id_document",
        "R.id_rubrique = lien.id_objet",
        "lien.objet='rubrique'",
        "docs.mode = 'document'"),
     "",
    "docs.date DESC");
while ($row=sql_fetch($result)) {
    $titre=$row['titre'];
    // ...
    // et avec le tableau précédent :
    $titre_extension =
$types[$row['extension']]['titre'];
}
```

[1 (p.0)] Un jour, un jour il y aura une véritable colonne dédiée !

sql_selectdb

La fonction `sql_selectdb()` permet de sélectionner pour une connexion à un serveur de base de données donné une base à utiliser. La fonction renvoie `true` si l'opération est réussie, `false` sinon.

La fonction `sql_selectdb()` a 3 paramètres :

1. `$nom` correspond au nom de la base à utiliser,

2. `$serveur`,
3. `$option`.

Cette fonction est utilisé par SPIP lors de l'installation pour essayer de pré-sélectionner le nom de la base de données à utiliser, en tentant de sélectionner une base du même nom que le login.

```
$test_base = $login_db;
$ok = sql_selectdb($test_base, $server_db);
```

sql_serveur

La fonction `sql_serveur()` permet à la fois de se connecter au serveur de base de données si ce n'est pas encore fait et d'obtenir le nom véritable de la fonction qui sera exécutée pour un traitement demandé. Cette fonction est appelée de façon transparente par des alias. Il est donc normalement inutile de l'utiliser.

`sql_serveur()` admet trois paramètres dont seul le premier est indispensable :

1. `$ins_sql` est le nom de la fonction désiré dans la liste des fonctions que connait l'API tel que « select », « update », « updateq »... Volontairement vide, il indique alors à simplement se connecter au serveur de base de données si ce n'est pas encore fait.
2. `$serveur`,
3. `$continue` définit ce qui se passe lorsque l'instruction de l'API SQL n'est pas trouvée pour le gestionnaire de base de données demandée. Par défaut à `false`, le système retourne une erreur fatale, mais il est possible de poursuivre en forçant ce paramètre à la valeur `true`.

Cette fonction s'utilise de la sorte :

```
// calcul du nom de la fonction
$f = sql_serveur('select');
// execution de la fonction selon l'API prevue
$f($arg1, $arg2, ... );
```

Si on demande l'instruction `select` dans le jeu d'instruction prévu pour MySQL et présent dans le fichier ecrire/req/mysql.php, la variable `$f` vaudra `spip_mysql_select`. La corrélation entre les l'instructions et la fonction est défini dans le même fichier par une globale `spip_mysql_functions_1` (mysql est le type de serveur, 1 est la version du jeu d'instruction).

Des alias pour simplifier

Pratiquement toutes les fonctions de l'API `sql_*` sont des alias calculant une fonction via `sql_serveur` et l'exécutant. Ainsi, appeler la fonction `sql_select` effectue (à quelques détails près) la même opération que le code précédent. Ce sont ces instructions là qu'il faut utiliser :

```
sql_select($arg1, $arg2, ...);
```

sql_set_charset

La fonction `sql_set_charset()` demande d'utiliser le codage indiqué pour les transactions entre PHP et le gestionnaire de base de données.

`sql_set_charset()` admet trois paramètres. Seul le premier est requis :

1. `$charset` est le type de charset souhaité, tel que « utf8 »
2. `$serveur`,
3. `$options`.

Cette fonction est appelée suite à chaque connexion au serveur de base de données afin d'indiquer le charset à utiliser. Ce choix encodage est d'ailleurs défini par la méta `charset_sql_connexion` créé à l'installation de SPIP.

sql_showbase

La fonction `sql_showbase()` permet d'obtenir une ressource utilisable avec `sql_fetch()` des tables présentes dans la base de données.

Elle admet 3 paramètres :

1. `$spip` vide par défaut, il permet de ne lister que les tables utilisant le préfixe défini pour les tables SPIP. Utiliser `'%'` pour lister toutes les tables,

2. `$serveur`,
3. `$option`.

Utilisation :

```
if ($q = sql_showbase()) {
    while ($t = sql_fetch($q)) {
        $table = array_shift($t);
        // ...
}
```

La fonction sql_alltable (p.259) est en général plus adapté, retournant directement un tableau PHP des différentes tables.

sql_showtable

La fonction `sql_showtable()` retourne une description d'une table SQL dans un tableau associatif listant les colonnes et leurs descriptions SQL (« field ») et listant les clés (« key »). Lorsqu'une déclaration de jointure est présente pour la table déclarée dans `tables_principales` ou `tables_auxiliaires`, le tableau la retourne également dans la clé « join ».

Ses paramètres sont :

1. `$table` est le nom de la table à interroger,
2. `$table_spip` permet de remplacer automatiquement « spip » par le vrai préfixe de table ; il vaut `false` par défaut,
3. `$serveur`,
4. `$option`

Utilisation :

```
$desc = sql_showtable('spip_articles', true);
// $desc['field']['id_article']  = "bigint(21) NOT NULL
AUTO_INCREMENT"
// $desc['key']['PRIMARY KEY']  = "id_article"
// $desc['join']['id_article']  = "id_article"
```

Dans la plupart des situations, il vaut mieux utiliser directement la fonction trouver_table (p.108), qui possède un cache sur la structure des données, utilise cette fonction `sql_showtable()` et ajoute des informations supplémentaires.

```
$trouver_table = charger_fonction('trouver_table', 'base');
$desc = $trouver_table('spip_articles');
```

sql_update

La fonction `sql_update()` met à jour un ou des enregistrements dans une table SQL. Les éléments transmis ne sont pas protégés automatiquement comme avec `sql_updateq()`, il faut dont faire attention aux injections SQL et utiliser les fonctions `sql_quote()` pour protéger les contenus quand cela est nécessaire.

La fonction admet 6 paramètres :

1. `$table` est la table SQL utilisée,
2. `$exp` contient les modifications à réaliser,
3. `$where`,
4. `$desc`,
5. `$serveur`,
6. `$option`.

Cette fonction est utilisée principalement pour modifier des valeurs en utilisant la valeur même d'une colonne, tel que :

```
// ajoute 1 a la colonne
sql_update('table', array('colonne' => 'colonne + 1'));
```

Lorsque des données ajoutées avec cette fonction sont susceptibles d'avoir des apostrophes ou proviennent de saisies utilisateur, il est important de protéger l'insertion avec `sql_quote()` :

```
sql_update('table', array('colonne' => sql_quote($valeur)));
```

Exemple

Actualiser la colonne « id_secteur » avec l'identifiant des rubriques n'ayant pas de parent :

```
// fixer les id_secteur des rubriques racines
sql_update('spip_rubriques',
array('id_secteur'=>'id_rubrique'), "id_parent=0");
```

Ajouter un nombre identique de visites aux statistiques de certains articles :

```
$tous = sql_in('id_article', $liste);
sql_update('spip_visites_articles',
    array('visites' => "visites+$n"),
    "date='$date' AND $tous");
```

sql_updateq

La fonction `sql_updateq()` sert à mettre à jour du contenu d'une table SQL. Le contenu transmis à la fonction est protégé automatiquement.

Ses 6 paramètres sont les mêmes que `sql_update()` :

1. `$table` est la table SQL utilisée,
2. `$exp` contient les modifications à réaliser,
3. `$where`,
4. `$desc`,
5. `$serveur`,
6. `$option`.

Elle s'utilise ainsi :

```
sql_updateq('table', array('colonne' => $valeur), 'id_table='
. intval($id_table));
```

Exemple

La fonction `modifier_contenu()` de ecrire/inc/modifier.php est appelée lorsqu'un objet éditorial est modifié et se charge d'appeler les pipelines `pre_edition` et `post_edition` et utilise `sql_updateq()` pour mettre à jour les données collectées :

```
sql_updateq($spip_table_objet, $champs,
"$id_table_objet=$id", $serveur);
```

sql_version

La fonction `sql_version()` retourne simplement le numéro de version du gestionnaire de base de données.

Elle dispose de 2 paramètres facultatifs :

1. `$serveur`,
2. `$option`.

Utilisation :

```
$x = sql_version();
echo $x;
// en fonction du type de serveur, on peut recevoir :
// avec MySQL :   5.1.37-1ubuntu5.1
// avec SQLite2 :  2.8.17
// avec SQLite3 :  3.6.16
```

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`.

`utilise` et `necessite` permettent aussi, par conséquent, de surcharger les fichiers du plugin qu'ils indiquent (en étant prioritaire dans le chemin).

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.160) 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.166). 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.171) 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.

AJAX

Le terme AJAX, accronyme de « *Asynchronous JavaScript and XML* », désigne un ensemble de technologies utilisées pour créer des interactions clients / serveur asynchrones.

Ces constructions, qui permettent de faire transiter au retour du serveur uniquement une partie de la page (ou de quoi reconstruire un élément de la page), diminuent fortement le volume des données à transporter et rendent souvent l'application plus réactive aux yeux de l'utilisateur.

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})]
```

Cache

Un cache est un stockage qui sert à accélérer l'accès aux données. Des caches sont présents à tous les niveaux d'un ordinateur, dans les microprocesseurs, les disques durs, logiciels, fonctions PHP... Ils permettent qu'une donnée qui a été retrouvée ou calculée soit accessible plus rapidement si on la demande à nouveau, cela en contrepartie d'un espace de stockage souvent volatile (comme la mémoire RAM) ou parfois rémanent (comme un disque dur).

Un cache a souvent une durée de vie limitée, par exemple, le temps de l'exécution d'un logiciel, ou le temps d'un traitement d'un appel à PHP. Une durée de validité peut aussi être fixée lorsque le support de stockage est rémanent ; une page web peut ainsi dire au navigateur combien d'heures une page sera valide si celui-ci la garde dans son cache.

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);
```

Pipeline

Le terme pipeline employé dans SPIP est à considérer au sens UNIX. Le pipeline exécute une série de fonctions dont le résultat de l'une sert d'argument à la suivante. De cette manière, chaque fonction d'un pipeline peut utiliser les données qui lui sont transmise, les modifier ou les utiliser, et les retourne. Ce résultat entrant alors comme argument de la fonction suivante et ainsi de suite jusqu'à la dernière.

Lors de l'appel d'un pipeline, il est très souvent transmis à la première fonction des données, ou au moins une valeur par défaut. Le résultat du chaînage des différentes fonctions est ensuite exploité ou affiché en fonction des situations.

Certains appels spécifiques de pipelines dans SPIP sont à considérer comme des déclencheurs (triggers) dans le sens où ils déclarent simplement un évènement, mais n'attendent aucun résultat des différentes fonctions qu'appellera le pipeline. La plupart de ces déclencheurs ont un nom préfixé de `trig_`.

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.19) dans les squelettes.

Index

Symboles

A

E

F

G

H

I

J

L

M

N

S

T

U

V

W

X

_

Table des matières