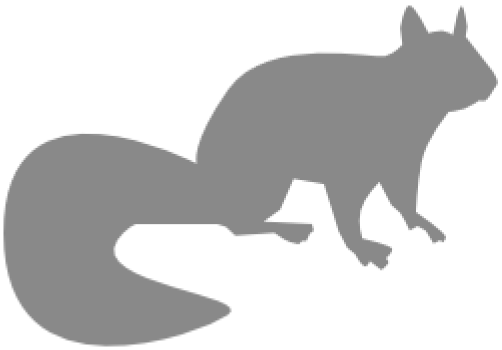
Programming with SPIP

DOCUMENTATION TO BE USED BY DEVELOPERS AND WEBMASTERS



SPIP's version 2.1.0+ / August 2010

SPIP is both a publication system and a development platform. After a quick tour of SPIP's features, we will describe how it works and explain how to develop with it using helpful examples wherever possible.

This documentation targets webmasters with knowledge of PHP, SQL, HTML, CSS and JavaScript.

Contents

Preface

The beginnings of this book date back to the late 2008. Matthieu was starting his work on this documentation for developers, and said to us: “... whatever we do, let’s do it with an open licence so that other people will be able to take over after I’ve done my part and then take it even further ... perhaps one day it could even be published in hard copy”. A SPIP book with an open licence: a dream nurtured for years was now showing its first signs of taking solid form. The idea had become firmly sounded out and recent new technologies were available to help make it a reality.

Anyone who wants to print a book can now find all kinds of simple tools on the internet to help with the task. It’s great to think that at a modest cost you can order a one-off copy of any text. And the feeling when you first receive the printed copy — can you imagine?

The first version of this book is coming out in the magical atmosphere of the Troglos meeting, and that is a positive sign. Version 1.0 of the book can be seen as the end of one adventure, but it is also the beginning of another. All the technical elements required are now in place for other books. They only need to be written — on a SPIP site, of course! Add a cover, and send it all to the printer.

SPIP SPIP HOORAY!

Ben.

Notes about this documentation

License and rights

Continuous improvements

This work - still in progress - has been subject to numerous proofreadings but is certainly not guaranteed exempt from any error. Please don't hesitate to offer improvements or point out mistakes by using the suggestion form available on the documentation internet site (http://programmer.spip.org). You may also discuss the organisation (of the content or technical presentation) and the translations by using the discussion list "spip-programmer" (requires subscription).

Write a chapter

If you feel motivated by this project, you may offer to write a chapter about a subject that you have mastered, or rework an existing chapter to make it clearer or more complete. We will do our best to accommodate your efforts and support you in such activities.

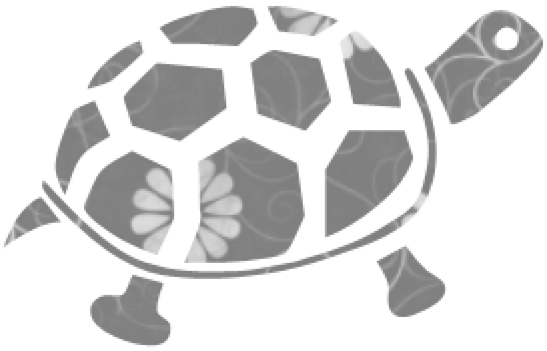
Translations

You may also contribute to the translation of this documentation into English or Spanish. The site's private zone (http://programmer.spip.org) is used for discussing the translations that are currently being prepared. Having said that, we're not planning to translate the documentation into other languages until such time as the organisation of the various chapters has been stabilised, which might yet take several months.

Computer code and properties of languages

With the aim of retaining compatibility, the computer code segments which serve as examples only contain ASCII characters. This means, among other things, that you will not find any language diacritic marks in the comments that accompany the examples anywhere in the documentation, a matter normally of considerable importance in French, and almost none in English. Therefore, we ask that you expect such absences to occur and ignore them.

Happy reading!



Introduction

An introduction to SPIP and a presentation of its general principles

What is SPIP?

SPIP 2.x is a free software package distributed under the GNU/GPL3 licence. Originally a Content Management System, it has gradually become a development platform making it possible to create maintainable and extensible interfaces independently of the structure of the managed data.

What can SPIP be used for?

SPIP is particularly suitable for websites and portals with regular community contributions, but it can also be used for a blog, a wiki, or a social network. More generally, SPIP can manage the storage and presentation of any data stored in MySQL, Postgres or SQLite databases. Extensions are available which also offer interaction with XML.

Requirements and basic description

SPIP 2.1 requires PHP version 5.0 or higher and at least 10MB of memory. It also requires a database (MySQL, Postgres or SQLite are supported).

The public website (front-office) is visible to all visitors by default, but it can be restricted to certain users, by section, if required. The private interface (back-office) is accessible only to persons authorised to manage the software or site content.

The templates (aka squelettes)

The whole website and its private area are calculated from templates that are made of static code (mainly HTML) and SPIP elements. These templates are, by default, stored in the squelettes-dist (for the private area) and prive (for the public-facing pages) directories.

For example, when a visitor makes a request for the home page, SPIP creates an HTML page based on the template named `sommaire.html`. Each type of editorial object in SPIP has a default template to display it, such as `article.html` for articles and `rubrique.html` for sections (or "rubriques" in French).

SPIP analyses and compiles these templates into PHP code which it stores in a cache. These cached files are then used to produce the HTML pages — which are themselves also cached — and then returned to the site visitor.

Quick overview

SPIP transforms templates into static pages. Templates are mainly composed of loops (`<BOUCLE>`) which select elements, typically sets of records, and tags (`#TAG`) which display the properties and values of those elements, typically fields from the records or system-wide functions and preconfigured values.

List the 5 most recently published articles:

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

In this first example, the `<BOUCLE_art()>` loop performs a selection from the `ARTICLES` table in the database. It sorts the data in reverse chronological order and returns the first five elements.

For each article that the loop selects, the `#URL_ARTICLE` tag is replaced with the URL calculated for that article's page, and the `#TITRE` tag is replaced with its title as stored in the database.

Typical HTML resulting from this kind of loop:

```
<ul>
    <li><a href="Recursion">Recursion</a></li>
    <li><a href="Parameter">Parameter</a></li>
    <li><a href="Argument">Argument</a></li>
    <li><a href="Modifying-all-of-your-templates-
in">Modifying all of your templates in one hit</a></li>
    <li><a href="Display-an-authoring-form-if">Display an
authoring form, if authorised</a></li>
</ul>
```

The templates

SPIP generates `HTML` pages from `templates`, which each contain a mixture of `HTML`, SPIP `loops` and `criteria`, and SPIP `tags` and `filters`. The major strength of SPIP is its ability to easily extract database content using a simple and understandable language.

Loops

A `loop` typically displays content retrieved from the database. It generates an optimised SQL query that extracts the desired content.

The syntax of loops

A loop declares both a database table from which to extract information as well the criteria for selection.

```
<BOUCLE_loopname(TABLE){criterion1}{criterion2}>
 ... for each object ...
</BOUCLE_loopname>
```

Every loop has a name (which must be unique within the template file), this name is used together with the word “BOUCLE” (English: "loop") to mark the start and the end of the loop. In this example, the loop’s name is “_loopname”.

The table is specified either by an alias (written in capital letters) or by the real name of the table (matching the case), for example “spip_articles”. The example uses the “TABLE” alias.

The next components of a loop are the criteria, each of which are always written enclosed in braces. For example, the criterion `{par nom}` will sort the results according to the “nom” column of the database table.

Example

This loop lists all of the images stored in the database. It draws its data from the database using the `DOCUMENTS` alias, and the criterion `{extension IN jpg,png,gif}` selects only those files which have a filename extension matching one in the given list.

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

The `#FICHIER` tag contains the URL of the image document. A filter named “image_reduire” is applied to the tag. This will resize the image to be at most 300 pixels (in height AND width) and returns an HTML `<img>` tag for the new scaled image.

The complete syntax of loops

Loops, like tags, have a syntax which allow them to deliver content in various combinations. Optional parts are displayed only once (not for each element) and only if the loop returns some content. An additional alternative part is displayed only if the loop does NOT return any actual content. Here is the complete syntax of a loop (x is the loop’s name):

```
<Bx>
    Display this once, before the loop's content
<BOUCLEx(TABLE){criteria}>
    Display something for each matching element
</BOUCLEx>
    Display this only once, after the loop's content
</Bx>
    Display this if there are no matching element results
<//Bx>
```

Example

This loop selects the five most recently published articles on the site. In this example, the `<ul>` and `</ul>` HTML tags will be displayed only once, and only if the loop criteria match some elements. If there are no matching elements, then these optional parts will not be output. This prevents the resulting HTML file from containing the `<ul>` and `</ul>` pair of tags without any constituent `<li>...</li>` child elements.

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

```
</B_latest_articles>
```

The `#DATE` tag displays the publication date of the article and the `affdate` filter ensures that it is in the correct language and is properly formatted.

Result:

```
<ul>
  <li>Content of exec file (squelette), <em>13 October
2009</em></li>
  <li>AJAX links, <em>1st October 2009</em></li>
  <li>Force language according to visitor, <em>27
September 2009</em></li>
  <li>Definition, <em>27 September 2009</em></li>
  <li>List of current pipelines, <em>27 September
2009</em></li>
</ul>
```

Nested loops

It is often useful to nest loops within each other to display more complicated elements. Nesting loops in this way makes it possible to use values from the first, outer, loop as selection criteria for the second, inner, loop.

```
<BOUCLEx(TABLE){criteria}>
    #ID_TABLE
    <BOUCLEy(SECOND_TABLE){id_table}>
        ...
    </BOUCLEy>
</BOUCLEx>
```

Example

Here we list all of the articles contained in the first two sections of the site. We use the `{racine}` criteria to select only the top-level sections, which are usually call "sectors" in SPIP:

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

The `ARTICLES` loop uses a sorting key `{par titre}` (“by title”) and a criterion `{id_rubrique}`. This last criterion instructs SPIP to select articles that belong to the current section — in this case, the one that is currently selected by the `RUBRIQUES` loop.

This will output:

```
<ul class='rubriques'>
    <li>fr
    </li>
    <li>en
        <ul class='articles'>
            <li>Notes about this documentation</li>
            <li>Another article</li>
        </ul>
    </li>
</ul>
```

Recursive loops

A common concept in many programming languages, an algorithm (a data-processing code) which makes reference calls to itself is described as being "recursive". Here, a recursive loop (n), contained in a parent loop (x), makes it possible to execute the loop (x) again, by automatically passing it the required arguments. Therefore, inside the loop (x), the same loop (x) is called with different parameters. This is what we refer to as recursion. This process will be repeated as many times as the recursive loop returns results.

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

When a site has many sub-sections, or many forum messages, it often uses recursive loops. This makes it possible to display identical elements very easily.

Example

We can use a recursive loop to display a list of all of the sections in the site. To do so, we loop for the first time on the sections, with a criterion that selects the sub-sections of the current section: `{id_parent}`. We also sort by numbers (assigned to the sections so that we can display them in a particular order), and then by titles (in case the numbers are absent or repeated for sections within a given parent section).

```
<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>
```

In the first iteration of the loop, `{id_parent}` will list the sections at the root of the site. These "root" sections all have an id_parent field equal to zero. When the first section is displayed, the recursive loop is called and so SPIP calls the loop “_rubs” again. This time the `{id_parent}` criterion selects a different set of sections because it will list the sub-sections of the current section. If there are sub-sections, the first one is displayed. Then the “_rubs” loop is called again, but now for *this* sub-section. As long as there are deeper sub-sections to display, this recursive process will repeat.

Result:

```
<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>Template syntax
            <ul>
                <li>Loops</li>
                <li>Tags</li>
                <li>Loops criteria</li>
                ...
            </ul>
        </li>
        ...
    </ul>
</li>
</ul>
```

More

Understanding the principals of recursive programming is not easy. If this explanation has left you confused, you may find it helpful to read the article about recursive loops on SPIP.net: http://www.spip.net/en_article2090.html

Loops with missing tables

When we ask SPIP to use a table which does not exist, it displays an error on the page. These error messages help administrators to fix problems with the site, but other users get to see them as well.

Sometimes, we don't care if a table is missing and want to ignore it silently, for example if we reference a table for a plug-in which might not be currently active. In these cases, we can place a question mark before the end of the brackets to indicate that the absence of the table is tolerated:

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

Example:

If a template uses the "Agenda" plug-in (which includes an EVENEMENTS table for events), but which must still function even in the absence of that plug-in, it is possible to write its loops this way:

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

Tags

Tags are most often used to display content, and sometimes just to calculate it without displaying (outputting) anything at all. Such content can originate from various sources:

- The template's own environment, i.e. some parameters that have been passed to the template, known as the compilation context.
- Content from an SQL table, by using a loop.
- Another specific source. In this case, the tags and their actions must be explicitly declared to SPIP, while the first 2 sources can be calculated automatically.

Tag syntax, the definitive version

Just like loops (boucles), tags often have optional components, and can sometimes also accept parameter arguments. Asterisks can be applied to bypass certain processes that are normally executed automatically for that tag.

```
#TAG
#TAG{argument}
#TAG{argument, argument, argument}
#TAG*
#TAG**
[(#TAG)]
[(#TAG{argument})]
[(#TAG*{argument})]
[ before (#TAG) after ]
[ before (#TAG{argument}|filter) after ]
[ before (#TAG{argument}|filter{argument}|filter) after ]
...
```

How the brackets work

The full syntax, with parentheses and square brackets, becomes mandatory as soon as one of the tag's arguments also uses parentheses and square brackets, or when the tag contains a filter.

```
// risk of bad surprises:
#TAG{[(#TAG|filter)]}
// always correctly interpreted:
[(#TAG{[(#TAG|filter)]})]
// although this works in SPIP 2.0, results are not
guaranteed:
```

```
#TAG{#TAG|filter}
// using a filter means you MUST also use parentheses and
square brackets:
[(#TAG|filter)]
```

For details as to the meaning of the square brackets and parentheses, please refer to the article in the official SPIP documentation: SPIP tag syntax.

Example

Display a link to the home page:

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

Display an HTML `<div>` tag and the contents of the `#SOUSTITRE` (subtitle) field if it exists:

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

The #ENV environment

We use the word environment to define the combined collection of variables that are passed to a particular template. We may also sometimes speak about a compilation context.

For example, when a visitor requests to see article 92, the article identifier (92) is passed to the `article.html` template. Within that particular "squelette" template, it is possible to retrieve the value of that variable using a special tag: `#ENV`. In this way, `#ENV{id_article}` would display "92".

Some parameters are automatically passed to the template, like the current date (when the page is actually generated or refreshed), which can then be displayed using `#ENV{date}`. Similarly, if we call a template with arguments using the page’s URL, they will also be passed into the environment.

Example

The URL `spip.php?page=albums&type=classic` will load up the `albums.html` template. Within that template, `#ENV{type}` enables you to retrieve the value that was passed, which in this case is "classic".

The contents of loops (boucles)

The content extracted from the selection made by a SPIP loop is displayed by using tags. Systematically, whenever a table has an SQL field "x", SPIP is able to provide/display the contents of that field by using the syntax `#X`.

```
<BOUCLE_myloop(TABLES)>
#X - #NAME_OF_SQL_FIELD - #NONEXISTENT_FIELD<br />
</BOUCLE_myloop>
```

SPIP will not create an SQL query for ALL of the table's columns (`SELECT * ...`) in order to retrieve the requested data, but, will, at each occasion, issue a specific data request – in this case, it would be `SELECT x, name_of_sql_field FROM spip_table`.

Whenever a field does not exist in the SQL table, like "nonexistent_field" in our example above, SPIP will not insert it into the table query, but will attempt to recover a value for the field from a surrounding loop – if there are any. If there is no parent loop with such a field, then SPIP looks into the environment, just as if you had written `#ENV{nonexistent_field}` instead.

Example

Let's take an example of an SQL table named "cats" which contains 5 columns: "id_cat", "race", "name", "age", "colour". We can list the contents of that table with:

```
<B_cats>
  <table>
    <tr>
      <th>Name</th><th>Age</th><th>Race</th>
    </tr>
    <BOUCLE_cats(CATS){par name}>
      <tr>
        <td>#NAME</td><td>#AGE</td><td>#RACE</td>
      </tr>
    </BOUCLE_cats>
  </table>
</B_cats>
```

When automatically analysing the squelette template, SPIP will understand that it should retrieve the `name`, `age` and `race` fields from the SQL table called `cats`. However, it will not retrieve the fields that it does not need (in this case `id_cat` and `colour`), thereby nominally reducing the load on the database server.

Contents of parent loops

Sometimes it's useful to retrieve contents from a loop which is a parent of the current loop, just by using an ordinary SPIP tag. SPIP offers a special syntax to do this explicitly with the # symbol, by simply identifying which loop you wish to retrieve the data from (where n below is the identifier of the targeted loop):

```
#_n:TAG
```

Example

Display the title of the section (rubrique) beside the title of the current article:

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

Predefined tags

We have already seen that we can use tags to extract data from the environment or from an SQL table. There are also some other tags which have their own explicitly defined functions.

In such cases, these functions are declared (within SPIP) either in the ecrire/public/balises.php file, or in the ecrire/balise/ directory

Here are a few examples:

- `#NOM_SITE_SPIP` : returns the name of the site
- `#URL_SITE_SPIP` : returns the site URL (without the final /)
- `#CHEMIN` : returns the path of a file `#CHEMIN{javascript/jquery.js}`
- `#CONFIG` : enables site configuration data to be retrieved (often stored in the "spip_meta" SQL table) e.g. `#CONFIG{version_installee}`
- `#SPIP_VERSION` : displays the current version of SPIP
- ...

We will see many more such examples in the following articles.

Generic tags

SPIP offers some powerful tools for creating special tags that reflect the context of the page, the current loop, or simply the tag's name.

It is possible to declare a set of tags with the same prefix, which will then all inherit a common processing.

These types of tags are declared in the ecrire/balise/ directory. They are stored as `SOMETHING_.php` files.

For example:

- `#LOGO_` to display the logos of an article, a section, etc.: `#LOGO_ARTICLE`, `#LOGO_RUBRIQUE`, etc.
- `#URL_` to determine the URL for a SPIP object, such as `#URL_MOT` within a `MOTS` loop
- `#FORMULAIRE_` to display a form defined in the `/formulaires` directory, like the one for `#FORMULAIRE_INSCRIPTION`

Automatic tag processes

Most SPIP tags, especially those that involve reading data from the database, perform some automatic processes to block malicious code that might have been added by article editors when they write their articles (PHP code or JavaScript scripts).

As well as these standard processes, there are others that can be defined for any SQL field in order to systematically apply those processes to the field in question. These operations are defined in the ecrire/public/interfaces.php file using a global table called `$table_des_traitements`. The key to this table is the tag name, the value being an associated table:

- its "0" key (the first `$table_des_traitements['BALISE'][]` encountered) defines a process regardless of the table in question,
- a "table_name" key (`$table_des_traitements['BALISE']['table_name']` without the table prefix) defines a process for a tag in that particular table.

The processes are provided by entering a character string `fonction(%s)` that explicitly reference the functions to be applied. Within that function call, "%s" will be replaced by the contents of the field.

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

Two common usages of automatic filters, which have additionally been defined with constants, can be reused:

- `_TRAITEMENT_TYPO` applies the SPIP typographical processes (applying bold, for example),
- `_TRAITEMENT_RACCOURCIS` applies the typographical processes and translations of SPIP links (adding the html attribute class="spip_out", for example).

Example

The `#TITRE` and `#TEXTE` tags are processed automatically regardless of which or how many loops they are used in, and these processes are defined as follow:

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

The `#FICHIER` tag performs a special process only within a documents loop:

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

Interrupting the automatic tag processes

The security processes and defined processes apply automatically to the tags, but it is possible to turn them off for special cases within a given template file. In essence, this delivers the tag content in a more "raw" format. Adding the "asterisk" suffix to a tag has this effect:

```
// execute all processes
#BALISE
// avoid the specifically defined processes
#BALISE*
// avoid even the security processes
#BALISE**
```

Example

To delay the application of typographical adjustments and the conversion of SPIP links for the text on a page (the `propre` filter is normally applied automatically) so that you can interpose your own custom filter before applying the "normal" filter again afterwards, you could do this:

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

Useful tags to know

Among the many tags that SPIP contains by default, some are used frequently enough to deserve special mention here.

Name	Description
#AUTORISER (p.31)	Checks authorisations
#CACHE (p.31)	Defines the duration of the cache
#CHEMIN (p.32)	Retrieves the path to a file
#DESCRIPTIF_SITE_SPIP (p.32)	Returns the site's description
#EDIT (p.33)	Edits content (using the "crayons" plugin)
#ENV (p.33)	Retrieves an environment variable's value
#EVAL (p.34)	Evaluates an expression using PHP
#EXPOSE (p.35)	Emphasises the element currently being read (in a list or menu)
#GET (p.36)	Retrieves a value stored with a `#SET`
#INCLURE (p.37)	Includes a template
#INSERT_HEAD (p.38)	Tag for inserting scripts into the HTML `<head>` section for SPIP or its plugins
#INSERT_HEAD_CSS (p.38)	CSS insertion tag within the HTML `<head>` section for plugins
#INTRODUCTION (p.38)	Displays an introduction
#LANG (p.39)	Finds out the language code
#LANG_DIR (p.40)	Returns the writing direction
#LESAUTEURS (p.41)	Displays the authors of an article
#MODELE (p.41)	Inserts a layout model
#NOTES (p.42)	Displays the footnotes created using the `[[]]` SPIP link
#REM (p.44)	Inserts a comment in the code
#SELF (p.44)	Returns the URL of the current page
#SESSION (p.44)	Recovers data about the session
#SESSION_SET (p.45)	Defines session variables

Name	Description
#SET (p.45)	Stores a value, retrievable with #GET
#VAL (p.46)	Returns a value

#AUTORISER

#AUTORISER is used to check access authorisations to particular content in order to tailor specific display for certain visitors. A specific chapter (Authorisations (p.198)) has been dedicated just to this topic.

```
[(#AUTORISER{action,objet,identifier}) I am authorised ]
```

The existence of this tag, as with the #SESSION tag, generates a different cache entry for every identified site visitor, and another cache entry for unauthenticated visitors combined.

Example

Check if a visitor has the right

- to view a particular article
- to modify a particular article

```
[(#AUTORISER{voir,article,#ID_ARTICLE}) I am authorised
to view the article]
[(#AUTORISER{modifier,article,#ID_ARTICLE}) I am
authorised to modify the article]
```

#CACHE

#CACHE{duration} is used to define the duration that the cache will be valid for after calculation of a template, expressed as a number of seconds. When this duration is exceeded, the template will be freshly calculated the next time it is requested.

This tag is generally inserted at the top of template files. In its absence, by default, the validity duration of a page is for 24 hours (defined by the constant _DUREE_CACHE_DEFAUT).

Example

Define a cache validity of one week:

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

#CHEMIN

#CHEMIN{directory/file.ext} returns the relative address of a file within the SPIP hierarchy. This topic is discussed in full in the article titled: The concept of path (p.104).

Example

Return the address of the "habillage.css" file. If it exists in the squelettes/ folder, that address will be provided, otherwise it will be the address of the file present in the squelettes-dist/ directory.

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

The squelettes-dist/inc-head.html file uses it to load the corresponding stylesheet into the <head> section of the HTML code. If the file is found, the HTML <link> tag is displayed.

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

Note that the direction_css filter is used to invert the whole CSS stylesheet (left to right and vice versa) if the contents of the site are written in a human language that is written from right to left.

#DESCRIPTIF_SITE_SPIP

`#DESCRIPTIF_SITE_SPIP` returns the description of the site as defined on the configuration page in the private area (back office).

Example

In the `<head>` section of the HTML code, this makes it possible to define the value for the HTML "description" meta tag using this SPIP tag, which is particularly useful on the site's home page (the `sommaire.html` file).

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

In this example, the `couper{150}` filter trims the contents to the first 150 characters (whilst still avoiding cutting any given word in half); and then the `textebrut` filter removes any HTML mark-up.

#EDIT

`#EDIT{name_of_the_field}`: this tag, by itself, does nothing and returns nothing… But when coupled with the "crayons" plugin, it makes it possible to edit content on the public site if the current visitor is authorised to do so. In this case, it returns the names of the CSS classes which will be used by a jQuery script supplied by that plugin.

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

Example

To be able to edit the "title" field:

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

#ENV

#ENV{parameter} – which we addressed in (The #ENV environment (p.24)) – retrieves environment variables passed into the template. A second argument is used to return a default value if the parameter requested does not exist in the environment or if its contents are empty.

```
#ENV{parameter, default value}
```

The parameter value retrieved is automatically filtered through entites_html, which converts the text into an HTML entity (a < character thereby becoming &lt;). To avoid this conversion, we can apply an asterisk to the tag:

```
#ENV*{parameter, default value}
```

Finally, the #ENV tag just by itself returns a sequential table of all of the environment parameters.

Example

Retrieve an article identifier, otherwise the string "new":

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

Display all of the environment variables (useful for debugging):

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

#EVAL

#EVAL{expression}, although quite seldom used, makes it possible to display the results of the PHP evaluation of the expression passed.

Example

```
#EVAL{3*8*12}
```

```
#EVAL{_DIR_PLUGINS}
#EVAL{$GLOBALS['meta']}
```

#EXPOSE

#EXPOSE is used to Emphasise one particular entry within a list. When we loop on a table and the #ENV{id_table} tag has a value within the current environment, or there is an #ID_TABLE in a higher level loop, then #EXPOSE will return a special code when the loop hits the same value as the identifier.

Its syntax is:

```
#EXPOSE{text if yes}
#EXPOSE{text if yes, text if no}
// expose with no parameters returns 'on' if yes or the empty
string '' if no
#EXPOSE
```

Example

List the articles in the current section, and assign the CSS class "on" for the current article.

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

Results:

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

#GET

`#GET{variable}` is used to retrieve the value of a local variable that was stored using a `#SET{variable, value}`. See also #SET (p.0).

A second argument is used to return a default value if the parameter requested does not exist or if its content is empty.

```
#GET{variable, default value}
```

Example

If "use_documentation" equals "yes", make it known:

```
#SET{use_documentation,yes}
[(#GET{use_documentation}|=={yes}|oui)
    We use documentation!
]
```

Display a link to the site’s home page over an image called "my_logo.png" if there is one, otherwise use "logo.png", otherwise use the site logo:

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

Differentiate the absence of an element in the environment: define `#ENV{default}` as a default value when `#ENV{activer}` does not exist. To do this, the `is_null` filter allows us to test that `#ENV{activer}` is not defined. If `#ENV{activer}` exists but is empty, it will be used. We can thereby differentiate the case of sending an empty value into a form, as shown below when the value sent is that of the "champ_activer_non" input field

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

#INCLURE

`#INCLURE` is used to add the results of an inclusion into the current template. We call this a "static" inclusion since the results of the compilation are added to the current template, into the same cache file. This tag is therefore different from a "dynamic" inclusion using `<INCLURE.../>` which creates a separate cache file (with its own specific cache duration).

```
// preferred format
[(#INCLURE{fond=template_name, argument, argument=xx})]
// other format, best to avoid
[(#INCLURE{fond=template_name}{argument}{argument=xx})]
```

From the perspective of the visible results, using `<INCLURE>` or `#INCLURE` will result in identical contents, but causes quite different operations from the internal management point of view. Dynamic inclusion using `<INCLURE>` will generate more autonomous cache files. Static inclusion using `#INCLURE` creates less files, but all of them are larger since their content is duplicated in each page of the cache.

Example

For the current template, add the content resulting from the compilation of the "inc-navigation.html" template, to which we will pass "id_rubrique" as a context argument:

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

Note: the inclusions `inc-head` and `inc-navigation` in SPIP's default templates are called using dynamic inclusions, and not static ones as are shown in this example.

#INSERT_HEAD

`#INSERT_HEAD` entered between the HTML `<head>` and `</head>` markup tags is used to automatically add in certain JavaScript scripts. Some scripts are already added in by default by SPIP (jQuery, for example), and others are inserted by various plugins. Refer to the sections about the insert_head (p.0) and jquery_plugins (p.0) pipelines which expand on adding in such scripts. To add in additional CSS code, it is better to use the `#INSERT_HEAD_CSS` tag and the insert_head_css (p.166) pipeline.

In SPIP's default templates, this tag is inserted at the end of the template named squelettes-dist/inc-head.html.

#INSERT_HEAD_CSS

`#INSERT_HEAD_CSS` inserted between the `<head>` and `</head>` HTML Tags enables plugins to add CSS scripts by using the insert_head_css (p.166) pipeline. If this tag does not exist in the template file, then `#INSERT_HEAD` will add the contents of the pipeline itself.

In SPIP's default template files, this tag is inserted just before the `habillage.css` CSS file in squelettes-dist/inc-head.html. This means that graphical themes that overwrite this `habillage.css` file can likewise be overwritten themselves, using CSS directives or include references, by declarations added by the corresponding installed plugins.

#INTRODUCTION

The `#INTRODUCTION` tag displays an extract of the contents of an SQL "text" field (if the table has such a field). For articles, this extract is drawn from the "Brief description" field if it has a value, else from the "Standfirst introduction" field appended by the "Text" field. The extract can also be explicitly defined when writing the content, by framing it with `<intro>` and `</intro>` mark-up tags.

An argument can be passed to define the maximum length of the introduction (the default is 600 characters):

```
#INTRODUCTION{length}
```

Example

Provide the HTML meta tag "description" with an introductory text about the article pages (example in squelettes-dist/article.html) :

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

Display the 10 most recent articles with an introduction of their contents:

```
<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 displays the language code taken from the element that is closest to the tag. If the tag is located within a loop, #LANG will return the SQL "lang" field from the loop if it exists, otherwise it will return that of the (#ENV{lang}) environment, and failing that the language code for the site's main language (#CONFIG{langue_site}).

#LANG* is used to return only the language of the loop or the environment. If none is defined, then the tag doesn't return anything (that is, it doesn't even return the site's main language).

Example

Define the language in the HTML tag for the page:

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

Define the language in an RSS feed (an example from 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 returns the writing direction for a text depending on its language, being either "ltr" (for "left to right"), or "rtl" (for "right to left"). As with the #LANG tag, the language is taken from the closest loop containing a "lang" field, otherwise from the environment, otherwise from the site's main language. This tag is very useful for multi-lingual sites that mix languages having different writing directions (like English and Arabic, for instance).

Example

Display the text for a section in the direction that it should be in:

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

#LESAUTEURS

#LESAUTEURS displays the list of authors of an article (or syndicated article), separated by commas. When the SQL field "lesauteurs" does not exist for the table requested, as is the case with articles table, then this tag loads a pre-formatted model squelettes-dist/modeles/lesauteurs.html.

Example

Display the authors from inside an ARTICLES loop:

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

#MODELE

#MODELE{name} inserts the results of a template contained in the modeles/ directory. The identifier of the parent loop is passed by default with the "id" parameter to this code inclusion.

Additional arguments can be passed:

```
// preferred format
[(#MODELE{name, argument=xx, argument})]
// alternative format, to be avoided
[(#MODELE{name}{argument=xx}{argument})]
```

These inclusions may also be called within the body text of an article (with the correct syntax), which means that article authors can optionally include particular models as and when they choose:

```
// XX is the identifier of the object to pass.
<nameXX>
// arguments using | (pipes)
<nameXX|argument=xx|argument2=yy>
```

Example

List the various translations of an article, with a link to each of them:

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

#NOTES

`#NOTES` displays the footnotes (collected at the bottom of the page) which have been calculated during the display of the preceding tags. Whenever a tag, for which we calculate SPIP links using the `propre` filter or using another automatic process, contains some footnotes, these notes will be displayed by the `#NOTES` tag once they have all been calculated.

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

Details about footnotes

It is the `traiter_raccourcis()` function called by the `propre` filter which executes the (`inc_notes_dist()` function in the ecrire/inc/notes.php file, which temporarily stores the notes in memory. When the `#NOTES` tag is actually called, these notes are returned and then emptied from memory.

Say there is some text in the "Standfirst introduction" and more in the "Text" body of a given article as shown below:

```
// Introductory text:
In the intro, there is one [[Note A]] and another note [[Note
B]]
// Main text:
In the text, there is one [[Note C]] and another note [[Note
D]]
```

When the template is displayed, the two code examples below will generate different contents. The first will display all of the notes together, numbered from 1 to 4, after the contents of the text:

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

In the second example below, the notes of the "intro" are displayed first (numbered from 1 to 2), immediately after the contents of the `#CHAPO` tag, and then the notes from the text (also numbered from 1 to 2), are displayed after the contents of the `#TEXTE` tag:

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

Example

Calling the notes is often done after all the other fields for an article, which will take into consideration all the notes that have been calculated. Displaying them is quite simple:

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

#REM

#REM is used for commenting code within the templates.

```
[(#REM) This is NOT a pipe! It's just a comment]
```

Note: The code contained in the tag is nonetheless interpreted by SPIP, but nothing is displayed. A filter found on the tag will effectively be called (which is not necessarily what you would like to happen):

```
[(#REM|filter)]
[(#REM) [(#TAG|filter)] ]
```

#SELF

#SELF returns the URL of the current page.

If you wish to use this tag inside some included code, then the URL can only be correctly calculated inside that inclusion if the `self` or `env` parameter is passed so that a different cache is created for each of the URLs.

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

#SESSION

`#SESSION{parameter}` displays data about a connected visitor. A session may be considered as a set of some data, stored on the server whilst the visitor is still connected. As such, this data can be retrieved and re-used when the visitor changes pages by navigating through the site.

The existence of this tag, as with the `#AUTORISER` tag, generates a different cache for each authenticated visitor on the site, and one more additional cache for the non-authenticated visitors.

Example

Display the name of the visitor if it is known:

```
#SESSION{nom}
```

Display a notice if the visitor is authenticated, that is, if the visitor has an `id_auteur` value:

```
[(#SESSION{id_auteur}|oui) You have been authenticated]
```

#SESSION_SET

The `#SESSION_SET{parameter, value}` tag is used to define session variables for a visitor, which can then be retrieved using `#SESSION{parameter}`.

Example

Define a flavour as vanilla, and then retrieve it.

```
#SESSION_SET{flavour,vanilla}
#SESSION{flavour}
```

#SET

`#SET{variable,value}` is used to store values locally within a template. They are retrievable, within the same template, using `#GET{variable}`. See also #GET (p.36).

Example

Store the value of a colour that already exists in the environment, otherwise use a default colour:

```
#SET{light,##ENV{light_colour,edf3fe}}
#SET{dark,##ENV{dark_colour,3874b0}}
<style class="text/css">
#contenu h3 {
    color:[(#GET{light})];
}
</style>
```

#VAL

`#VAL{value}` is used to return the value passed as an argument. This tag is mainly used to send a first argument to some existing filters.

```
#VAL{This text will be returned}
```

Example

Return a character using the PHP function `chr`:

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

Sometimes the SPIP compiler gets confused between the square brackets that we want to show as text characters, and the opening and closing square brackets used for our SPIP tags. A common example is sending a table parameter to a form (`name="field[]"`), when the field is included inside a loop:

```
// problem: the ] in the text field[] confuses the SPIP
compiler
// with the closing of the #ENV tag
[(#ENV{afficher}|oui)
<input type="hidden" name="field[]" value="valeur" />
]
// no problem for the SPIP compiler with the code shown
below
[(#ENV{afficher}|oui)
<input type="hidden"
name="field[(#VAL{91}|chr)][(#VAL{93}|chr)]"
value="valeur" />
]
```

Loops Criteria

Use criteria in loops to specify simple or complex selection conditions.

Criteria syntax

The loops criteria for are written between curly brackets just after the tables names.

```
<BOUCLE_name(TABLE){criterion1}{criterion2}...{criterion n}>
```

Any SQL field in a table can become a selection criterion separated by an operator. But other criteria can be created when necessary. They are defined in the ecrire/public/criteres.php file.

Some tags can also be used as criteria parameters, but it is not possible to use their optional components. In general, the use of square brackets is not possible:

```
<BOUCLE_name(TABLE){id_table=#TAG}> OK
<BOUCLE_name(TABLE){id_table=(#TAG|filter)}> OK
<BOUCLE_name(TABLE){id_table=[(#TAG)]}> will fail
```

Example

This ARTICLES loop has 2 criteria. The first selects the articles where the "id_rubrique" SQL field in the "spip_articles" SQL table equals 8. The second criterion specifies that the results should be sorted in ascending order of those articles’ titles.

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

Criteria shortcuts

A criterion can sometimes be written in a simplified form: `{criterion}`. In such cases, SPIP normally translates this as `{criteria=#CRITERIA}` (unless a special function has been defined for this particular criterion in ecrire/public/criteres.php).

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

Example

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

In this example, `{id_article}` makes the selection `{id_article=#ID_ARTICLE}`. Just as with any SPIP tag, `#ID_ARTICLE` is retrieved, if present, from the closest containing loops, otherwise it is retrieved from the environment, as if you had written `#ENV{id_article}`.

Simple operators

All of the criteria that perform selections based on SQL field values have a certain number of operators available for their use.

```
{field operator value}
```

Here is a list of the simple operators:

- = : equality operator `{id_rubrique=8}` selects records with "id_rubrique" equal to 8.
- > : strictly greater than operator. `{id_rubrique>8}` selects records with "id_rubrique" greater than 8.
- >= : greater or equal operator. `{id_rubrique>=8}` selects records with "id_rubrique" greater than or equal to 8.
- < : strictly lesser than operator . `{id_rubrique<8}` selects records with "id_rubrique" less than 8.
- <= : lesser or equal operator. `{id_rubrique<=8}` selects records with "id_rubrique" less than or equal to 8.

- `!=` : not equal operator `{id_rubrique!=8}` selects records with "id_rubrique" different from 8.

The IN operator

There are some other operators that allow for more specific selections. The `IN` operator selects records based on a list of possible matching values for a field element. This list can either be determined by a comma-separated list of characters, by an array table returned by a tag, or by a tag or tag filter.

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

The inverse operator `!IN` selects records that have field values that do not match any of those listed after the operator.

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

Example

Select the images linked to an article:

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

Select the sections, except for 3 specific ones:

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

The == operator

The == operator (or its inverse !==) is used for making record selections based on regular expressions. They can therefore enable extremely specific selection criteria, but may also be quite resource-intensive for the database manager.

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

Example

Select articles with a title that starts with "The" or "the":

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

Select article texts that do not contain the word "carnival":

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

Select article texts that do contain the word "carnival", but only if followed by the word "Venice" within 20 characters.

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

The "!" operator

Conditional criteria of simple negation, when operating on fields that are external to the table (fields accessed by a join to another table), do not always do what one might think at first.

As an example, the criteria `{titre_mot!=rose}` selects, for an ARTICLES loop, all the articles which are not linked to the keyword "rose". However, the type of SQL join created selects only articles linked to at least one keyword, and where at least one of those keywords is not "rose".

But in most cases, we would simply be trying to display all articles that do not have the keyword "rose", regardless of whether they had any other keywords or not. That is the result produced by using code with a `{!criterion}`. The code below generates a double SQL query:

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

First, articles with keyword X are selected, then they are removed from the main SQL record selection by use of a `NOT IN (selection criteria)` on the actual SQL database query.

This syntax is equally valid when you want to force a join field, which could be written as follows:

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

Example

Select the sections which have no article whose title starts with an "L" or an "l". Note, however, that this query uses a regular expression (`^[Ll]`) which will require more calculation time from the database manager.

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

Optional criteria

Sometimes it's useful to make a selection only if the environment contains the requested tag. For example, we might hope to filter the loops based on a particular search, but only if a search has been performed, otherwise display everything. In such a case, a trailing question mark is used to request such an action:

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

Example

Display either all the articles of the site (if there is no `id_article`, `id_rubrique` or `recherche` variable apparent), or perform a selection based on the criteria that are present. In this way, if we call the template with the `id_rubrique=8` and `recherche=extra` parameters, the loop will select only the articles that match these criteria. This allows us to reuse the loop code in different ways within a model or with SPIP template includes.

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

Optional criteria with operators

The use of optional criteria may be combined with the use of operators under certain specific conditions. In particular, it is necessary for the variable which is being tested in the environment to have the same name as the criteria; for example, X in:
`{X ?operator #ENV{X}}`. Any operator can be used here, and you only need to affix a ? to the selected operator (leaving no space between the ? and the operator).

In the following examples, the test is performed only if the variable is present in the environment. Otherwise the criterion is ignored.

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

Example

To select the 10 most recent articles but with an "earlier publishing date" prior to the one in the current environment, or, failing that, simply the 10 most recent articles, use this loop:

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

Tag filters

Applying filters allows you to change the output generated by SPIP tags.

Filter syntax

Filters are applied to tags by using the pipe ("|") character. Their effect is to call a PHP function, either one which is in the standard PHP library or one which has been declared within SPIP.

```
[(#TAG|filter)]
[(#TAG|filter{argument2, argument3, ...})]
```

Whenever a filter "x" is requested, SPIP looks for a function called "filtre_x". If it does not find one, it looks for "filtre_x_dist", and then "x". It then runs the function that it has found, passing any arguments. It is important to understand that the first argument sent to the filter (and therefore to the PHP function) is the result of the component to the immediate left of that filter. Thus the example above shows the filter's parameter list as argument2, argument3, etc.

Example

Insert a `title` attribute on a link. To do this, we use the `|couper` filter, which allows us to cut a text down to a requested length, and the `|attribut_html` filter, which allows us to apply escape sequence characters to apostrophes that might cause problems with the generated HTML code (example: `title='David's book'` would cause a problem because of the embedded apostrophe.).

The `|couper` filter is applied to the result of the `#TITRE` tag, and the `|attribut_html` filter is applied to the result of the `|couper` filter. This shows how filters can be chained.

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

Filters derived from PHP classes

A less well-known coding technique makes it possible to also execute a PHP class method. When requested to process a filter written as “x::y”, SPIP will look for a “filter_x” PHP class with an executable “y” method. If it doesn’t find one, it will then look for a class “filtre_x_dist”, and then finally for a class “x”.

```
[(#TAG|class::method)]
```

Example

Let’s imagine a PHP class which has been defined as shown below. It contains a (recursive) function which calculates factorials (`x! = x*(x-1)*(x-2)*...*3*2*1`).

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

This new function could be called within SPIP as follows:

```
[(#VAL{9}|Math::factorial)]
// returns 362880
```

Comparison filters

Just like the criteria used for loops, comparison filters can also be applied to tags with the following "pipe" syntax:

```
[(#TAG|operator{value})]
```

The list of applicable operators includes:

- == (confirms equality)
- `!=`
- `>`

- >=
- <
- <=

Example

```
[(#TITRE|=={Chocolate}|oui)
    Some chocolate!
]
[(#TEXTE|strlen|>{200}|oui)
    This text is longer than 200 characters!
]
```

`[(#TITRE|=={Chocolate}) Some chocolate!]` would, if the test evaluates to true, display “1 Some chocolate!” (since 1 indicates a *true* value in PHP). But adding the `|oui` filter (French for yes) allows you to hide the results of the test.

Search and replace filters

There are some filters that allow you to perform comparisons or searches for components. This is the case for the "|match" and "|replace" filters.

- `match` is used to test if the argument passed verifies a regular expression passed as the filter’s second argument.
- `replace` is used to replace text, and is also followed by a regular expression.

```
[(#TAG|match{text})]
[(#TAG|replace{text,other text})]
```

Example

```
// displays "text yes"
[(#VAL{A good text}|match{text}) yes ]
// displays "yes"
[(#VAL{A good text}|match{text}|oui) yes ]
```

```
// doesn't display anything
[(#VAL{A good house}|match{text}) non ]
// displays "yes"
[(#VAL{A good house}|match{text}|non) yes ]

// displays "A fine cat"
[(#VAL{A fine house}|replace{house,cat})]
```

Test filters

There are several filters used for tests and logical operations. These are the filters "?", "sinon" (else in French), "oui", "non", "et", "ou", "xou" which are generally used in most cases.

- `|?{vrai,faux}` returns "faux" (false in French) if what is input to the filter is empty or null, otherwise it returns "vrai" (true in French) - this might be better interpreted in English as "boolean_does_this_thing_have_a_value".
- `|sinon{this text}` returns "this text" only if what is input to the filter is empty, otherwise it simply returns that same input - this might be better interpreted in English as "but_if_empty_put_this_instead".
- `|oui` returns either a space or nothing. It is equivalent to `|?{' ',''}` `or |?{' '}` and is used to return a non-empty content (a space) to indicate that the optional parts of the tags should be displayed.
- `|non` is the opposite of `|oui` and is equivalent to `|?{'',' '}`
- `|et` is used to confirm the existence of 2 elements (logical AND)
- `|ou` is used to confirm the existence of either 1 or 2 elements (logical OR)
- `|xou` is used to confirm the existence of one, and only one, of the two elements (logical XOR).

In addition, SPIP will also understand the English equivalents for these: "yes", "not", "or", "and" and "xor".

Example

```
// display the short description if it exists, otherwise
the beginning of the text
[(#CHAPO|sinon{#TEXTE|couper{200}})]
 // displays "This title is long" only if the title is
longer than 30 characters
[(#TITRE|strlen|>{30}|yes) This title is long ]

[(#CHAPO|no) There is no short description ]
[(#CHAPO|and{#TEXTE}) There is a short description, and a
text ]
[(#CHAPO|and{#TEXTE}|non) The two do not exist at the
same time ]
[(#CHAPO|or{#TEXTE}) There is either a short description,
a text, or both ]
[(#CHAPO|or{#TEXTE}|non) There is neither one nor the
other ]
[(#CHAPO|xor{#TEXTE}) There is one, or the other, but not
both, and not neither ]
[(#CHAPO|xor{#TEXTE}|non) Neither or both, but not just
one of the two ]
```

Includes

To facilitate the maintenance of generated code, it is important to be able to share re-usable code. This is achieved through the use of included code segments.

Includes within the templates

Creating and using includes — reusable code segments — makes it easier to maintain your templates. In general practice, certain segments of the HTML pages on your site are identical, regardless of the type of page. This is often the case when displaying a portfolio of images, for a navigation menu, for keywords attached to a section or an article, for meta tags in the HTML body, or footer text and links at the bottom of each page.

Any existing SPIP template code can be included within another using the following syntax:

```
<INCLURE{fond=file_name}{passed parameters} />
```

Typically, the only parameters passed are the current or a specific id_rubrique, id_article or similar identifier, the current or a specific language code, or the keyword “doublons” to permit a data context to recognise an overlapping environmental data duplication.

Passing parameters to includes

You can pass one or more parameters to code segments that have been included in a template. By default, nothing is passed to included code except the processing date. To pass parameters to the compilation context of the template, they must be explicitly declared when calling the include:

```
<INCLURE{fond=include_template}{parameter} />
<INCLURE{fond=include_template}{parameter=value} />
```

The first example with {parameter} only retrieves the value of #PARAMETER and passes it to the compilation context in the variable parameter. The second example assigns a specific value to that parameter variable. In both cases, within the included code, we can retrieve the value by reference using #ENV{parameter}.

Passing the entire current context

The {env} parameter can be used to pass the entire template compilation context to the code that is being included.

Example

```
// file A.html
<INCLURE{fond=B}{type}{key=# newt} />
// file B.html
<INCLURE{fond=C}{env}{colour=red} />
// file C.html
Type : #ENV{type} <br />
Keyword : #ENV{key} <br />
Colour : #ENV{colour}
```

If we call the page spip.php?page=A&type=animal, that would pass the type and key parameters to the B.html template segment. This third example passes everything it has received and adds another parameter colour when it calls the C.html template segment.

Within the C.html template, we then see that it is possible to retrieve all of the parameters that have been passed.

Ajax

SPIP allows you to easily reload parts of a page using AJAX.

AJAX paginations

Includes which have the `{ajax}` criteria are used to reload only the part of the page that has been included. Most of the time, you must also include the `{env}` criteria whenever there is a pagination mechanism within the included code.

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

When we combine this include criteria with the `#PAGINATION` tag, the pagination links will then automatically become AJAX links. More specifically, all of the links in the included template code are contained within a CSS class named `pagination`.

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

Example

List the five most recent articles. This include lists the most recent articles in groups of 5, and displays a pagination block.

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

The file `modeles/list_recent_articles.html` uses:

```
<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>
```

Results: Ajax pagination, in groups of 5...

```
<a id="pagination_art" name="pagination_art"/>
<ul>
    <li><a href="Recursion,369"
title="art369">Recursion</a></li>
    <li><a href="Parameter,368"
title="art368">Parameter</a></li>
    ...
</ul>
<p class="pagination">
    <strong class="on">0</strong>
    <span class="separator">|</span>
    <a rel="nofollow" class="link_pagination noajax"
href="Paginations-AJAX?debut_art=5#pagination_art">5</a>
    <span class="separator">|</span>
    <a rel="nofollow" class="link_pagination noajax"
href="Paginations-
AJAX?debut_art=10#pagination_art">10</a>
    <span class="separator">|</span>
    ...
    <a rel="nofollow" class="link_pagination noajax"
href="Paginations-
AJAX?debut_art=205#pagination_art">...</a>
</p>
```

AJAX links

In addition to the includes that contain a pagination mechanism, it is possible to specify links to be reloaded using AJAX by adding the CSS class `ajax` to those links.

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

Example

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

The file `modeles/list_articles.html`: Shows or hides the introduction to 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,''})]">Hide
the introductions</a>
]
[(#ENV{afficher_introduction}|=={oui}|non)
    <a class="ajax"
href="[(#SELF|parametre_url{afficher_introduction,oui})]">Show
the introductions</a>
]
```

Linguistic elements

The management and the creation of multilingual content is always a delicate thing to organise. We will see in this section how to manage the multilingual interface.

For the management of interface texts (as distinguished from editorial content), SPIP has two elements: language strings that are known as "idioms", and a multilingual tag called a "polyglot".

The syntax of language strings

Language-specific strings, known as "idioms" within SPIP, are the codes assigned to the existing translations in the files stored in the `lang/` directories in SPIP, plugins or in specific template files.

To reference a language string, you only need to know its corresponding code:

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

The general syntax is:

```
<:key:>
<:prefix:key:>
```

Language files

The language files are stored in the `lang/` directories. These are PHP files named with a prefix and a language code: `prefix_xx.php`.

Content of the files

These PHP files each declares a mapping table. Each key has its corresponding value. Any and all problematic language characters are transcribed using the HTML entities (for accented letters, for example), and some languages have the values written in unicode html entities (e.g. for Japanese, Arabic, etc.).

```
<?php
```

```
$GLOBALS[$GLOBALS['idx_lang']] = array(
    'key' => 'value',
    'key2' => 'value2',
    // ...
);
```

Example

Here is an extract from the French language file for the template of this site (`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 !',
    //...
);
```

The equivalent extract from the English version (`documentation_en.php`)would look like this:

```
<?php
$GLOBALS[$GLOBALS['idx_lang']] = array(
    //C
    'choisir'=>'Select...',
    'conception_graphique_par'=>'Graphical theme based on
',
    //E
    'en_savoir_plus' => 'Find out more!',
    //...
);
```

Using the language codes

Any language idiom (externally defined character strings) can be referenced in a SPIP template file using this syntax:

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

Looking for a code in several files

It is possible to search for a code in several language files. By default, if the prefix has not been supplied, SPIP will look in the `local_xx.php` files, then the `spip_xx.php` files, and finally the `ecrire_xx.php` files. If it does not find the code in the language requested, it then looks in the French language. If it still does not find the code, it will display the language code itself (but replacing underscore characters with spaces).

You can specify that the search should operate over several files with the following syntax:

```
<:prefix1/prefix2/.../prefixN:choisir:>
```

Overwriting a language file

To overwrite the language items already present in a SPIP language file, for example `ecrire/lang/spip_xx.php` or in a plugin language file `lang/plugin_prefix_xx.php`, you only need to create a `squelettes/local_xx.php` file and insert any modified or new items into that file. SPIP automatically uses such "local" files as taking precedence over the others mentioned above.

Such an operation is often used for locale specific overrides - for instance, in France there are regional divisions known as "départements" and "régions", whereas in Switzerland, it might be more appropriate to rename that same field as a "canton".

Example

Select the right documentation!

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

If `bouton_ajouter` is not found in the "documentation" language file, then look for it in the "spip" language file, and failing that, in the "ecrire" language file:

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

The complete syntax of language codes

The complete syntax is as shown below:

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

Parameters

The language codes can receive parameters which will be inserted into the values at the time of translation. The parameters are then written in the language files between at (@) signs.

A language code might therefore be:

```
'creer_fichier'=>'Create the @fichier@ file?',
```

Calling with parameters

We could call a parameter as below:

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

Filtering language codes

It's not a commonly used practice, but it is possible to pass language codes through filters just as if they were SPIP tags, for example:

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

Using language codes in PHP

A function has been created in PHP to retrieve the translations of the language codes: `_T`.

It is used very simply as shown below:

```
_T('code');
_T('prefix:code');
_T('prefix1/.../prefixN:code');
_T('prefix:code', array('param'=>'value'));
```

Character strings during development

You may sometimes run into the _L function, which is used to signify: "Character string to be assigned a language code when development is nearly finished". The general idea, is that during development of SPIP or plugin functionality, the language strings may change quite frequently. In order to distinguish strings which have already been translated in the language files from those that have just recently been created, we generally apply the _L function.

```
_L('This text will need to be codified and translated!');
```

When the code development has stabilised, a search through the code for uses of the "_L" function makes it easy to replace such character strings with appropriate language codes (and then use the _T function instead).

Example

The "Tickets" plugin has a language file named `lang/tickets_fr.php` which contains (amidst other code):

```
$GLOBALS[$GLOBALS['idx_lang']] = array(
    // ...
    'ticket_enregistre' => 'Ticket saved',
);
```

When someone creates a new ticket, the feedback form indicates that it has actually been saved by sending the language string to the `message_ok` parameter of the ticket writing form:

```
$message['message_ok'] = _T('tickets:ticket_enregistre');
// being = "Ticket enregistr&eacute;" if it were in
French.
```

Polyglots (multi tags)

A `<multi>` tag (in the HTML sense of the word), usable both for templates and in content written by editors, makes it possible to select a particular piece of text based on the requested (or currently default) language.

It’s syntax is:

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

This means that multilingual elements can easily be written within templates without needing to use language codes and strings as discussed in previous articles in this section.

Usage by content editors

This syntax is mostly used by content editors (or through means of a data-entry plugin that does the work automatically!) to translate a site when there are only a few (2 or 3) languages to be translated. `<multi>` is therefore more generally used on the content creation side rather than in construction of the templates. If such a `<multi>` block does not contain an entry for a specified language, then the first entry within that block will be used as the default.

Multilingualism

SPIP is designed to manage a multilingual website. We can distinguish several possibilities for what a multilingual website might mean:

- Having the language of the interface that adapts to the visitor (for example, displaying dates or writing words in the correct direction),
- Having content in multiple languages, not just the interface (for example a version of the site is in French, another in English),
- Or why not a mixture of both (interface in Arabic with text in French ...)

SPIP has various tools and syntaxes to achieve all these ends.

Multilingual possibilities

There are a number of ways of developing a multilingual site with SPIP, for example:

- create a sector (a section at the root of the site) for each language, with completely autonomous content,
- create the site in the main language and define translations of the articles in the various language(s) desired,
- or even define the language for each section of the site or for each article.

Each of these solutions has its own advantages and disadvantages, and the webmaster's choice of which method to use will largely not affect the construction of the template files (squelettes). In the following pages, we will review some of the tools used by the page templates for delivering multilingual content.

More

An excellent discourse on multilingualism was compiled by Alexandra Guiderdoni for the SPIP Party in Clermont-Ferrand in 2007. Reading it will benefit anyone who wishes to understand the subtleties or who needs to ask themselves the right questions during the construction of a multilingual site (in French only): http://www.guiderdoni.net/SPIP-et-l...

The environment's language

SPIP passes the language requested by the site visitor to the first template, which can be retrieved by using the #ENV{lang} function within a template. By default, this will be the main language of the site, but it can be changed using the #MENU_LANG form, which lists the predetermined languages for your site's multilingual content.

Whenever you use the #MENU_LANG form, the language selected is saved in a cookie and a redirection is made for the current page with the lang URL parameter assigned to the selected language. The lang parameter that is passed will then be accessible by SPIP. It will then also be possible to use the cookie later to force the display language.

The language may otherwise be specified explicitly within a template file, by using the lang criterion:

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

The language of an object

Some editable objects in SPIP, such as sections and articles , have a language field stored in their corresponding SQL tables, which makes it possible to specify in which language they have been written (or to which language they belong).

We can find out the language of the current section or article by using the #LANG tag within a RUBRIQUES or ARTICLES loop.

When the current section does not have a specific language assigned, then that of its parent section is returned, and failing that, the main language of the site.

Example

Display the articles and the languages of the first 2 sections in the site:

```
Your language: #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>
```

Results:

```
Your language : 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>
```

Special language criteria

Some special loop criteria make it possible to retrieve articles in a specifically desired language.

lang

First of all, the quite simple `{lang}` criterion enables us to select the visitor’s language or a specific language:

```
// language of the visitor
<BOUCLE_art(ARTICLES){lang}> ... </BOUCLE_art>
// English language (en)
<BOUCLE_art(ARTICLES){lang=en}> ... </BOUCLE_art>
```

traduction

The `{traduction}` criterion enables us to list the various translations of an article:

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

In this case, all the translations of an article will be displayed (including the current article, which could be excluded by specifically adding the `{exclus}` criterion).

origine_traduction

This criterion enables us to retrieve the original source article for a particular translated article, that being the one that serves as the source reference to the other translations. To show all of the source articles, use:

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

To show the original translation for an article (the one in the current context):

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

Example

Display an article in the visitor’s language where possible, and if not, then in the main language.

We start by listing, for a section, all the articles which serve as sources for the creation of translations. We then continue by checking if a translation exists in the language requested by the visitor. Depending on the result, we display the title of the translated article or that of the source article.

```
<BOUCLE_art1(ARTICLES){id_rubrique}{origine_traduction}>
    <BOUCLE_art2(ARTICLES){traduction}{lang=#ENV{lang}}>
      // if a translation does exist
      <li>#TITRE</li>
    </BOUCLE_art2>
      // otherwise use the original article's title
      <li>#TITRE</li>
    <//B_art2>
</BOUCLE_art1>
```

Forcing the language of the visitor’s choice

The parameter `forcer_lang`

The `#MENU_LANG` form stores the selected language in a cookie. This cookie can then be used to re-display the site in the language that the user has chosen. One of the ways of doing this is to define the `forcer_lang` global variable in the options file.

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

Specifying this parameter indicates to SPIP that it should systematically redirect a requested page by adding the `lang` URL parameter with the language cookie value if there is one, and if not, then with the code of the site’s main language.

This `forcer_lang` global code, however, also has another action: at the same time, it specifies that the language strings of the interface should display in the visitor’s language, and not in the language assigned to the articles and sections.

Another use of the cookie

Another possibility is to use the user’s preference, but not to force the redirection through the `lang` URL parameter, but rather by using SPIP’s `set_request` function to add a calculated `lang` parameter that SPIP will later reuse when it calls the `_request` function.

Example

The example below, used in an options file, calculates the language to be used. This calculation is made in two steps:

- check if the URL is of the form `http://name.domain/language/rest_of_the_url`, where "language" is one of the languages defined for the site ("fr", "en" or "es" for example) and in such a case, use the language thus discovered,
- otherwise, the `utiliser_langue_visiteur()` function uses the cookie language, otherwise use the language of the browser.

And finally, if the language calculated is different from the cookie, then the cookie is recreated.

```
// Systematically add the context original language.
if (!$langue = _request('lang')) {
    include_spip('inc/lang');
    $langues = explode(',',
$GLOBALS['meta']['langues_multilingue']);
    // if the language is defined in the url (en/ or fr/
), then use it
    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'];
        }
    }
    // store it in $_GET
    set_request('lang', $langue);
}
// store the language as a cookie...
if ($langue != $_COOKIE['spip_lang']) {
    include_spip('inc/cookie');
    spip_setcookie('spip_lang', $langue);
}
```

Choosing the navigation language

By default, when you navigate to view an English article, the interface components are translated into English.

By using the language selection form `#MENU_LANG`, this will change the interface elements by default and for the articles in the selected language.

Unless we are already in an article for a specific language, English for example, and therefore already using the English interface and with the language menu displaying "English", then if we request to display that page in French by using the language menu, the URL for the page will now add a parameter `lang=fr`, but in fact nothing actually changes for the site visitor, with both the article itself and its interface remaining in English: what is actually happening, is that it is the article’s own context which has taken priority over the visitor’s request.

We could also see the opposite case though, displaying the interface in French, but reading the article in English anyway. In order to make the interface behave independently to the current article’s or current section’s language, you will need to define the global variable `forcer_lang` in the mes_options file:

```
// enforce the language of the visitor
$GLOBALS['forcer_lang']=true;
```

Forcing a change in the interface language

As a final note of importance regarding multilingualism, some people may want to have a mix of languages between the interface and the content, yet still wish to maintain some consistency. More specifically, many would like to display the articles in the source languages if they have not as yet been translated into the requested language. In such cases, you will need to activate the `forcer_lang` setting.

Nonetheless, when displaying an article, it is possible to list the other various existing translations, as is done in the SPIP model code `modeles/articles_traductions.html`, where the generated link does not change the interface language, given that `forcer_lang` maintains the visitor's language.

If you would prefer that clicking on a translation link implies changing the interface language as well (into the same language as that of the translated article), then you will need to edit the model code for `articles_traductions.html` or create a new version. We then use the "converser" action enabling the generation of a special link which redirects to the desired article in the desired interface language:

```
[(#VAL{converser}
    |generer_url_action{[redirect=(#URL_ARTICLE
        |parametre_url{var_lang,#LANG})]})]
```

Example of a complete (and complex!) model:

This is a model that lists the various translations of an article. If it is not the translation currently being viewed, a link is proposed indicating the translation language.

```
<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>
```

SQL joints between tables

A joint in SQL is what allows information to be retrieved from multiple tables combined in a single query. It is possible to perform certain joints using SPIP's loop syntax.

Automatic joins

Whenever a loop is requested to use a criteria which does not belong to the loop's own primary table, SPIP automatically tries to find a linked table which contains the requested field.

SPIP has two methods to find linked tables: either through links that are declared explicitly, or ones that are calculated.

Example

Retrieve the documents which are inserted into the body text of articles or other editorial content (as with a `<docXX>` type model), and which are not just linked to that object. The `vu` field belongs to the `spip_documents_liens` table, so a join is created to that table to obtain the desired result.

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

Explicit join declarations

The links between tables are declared within SPIP in the ecrire/public/interfaces.php file. Further declarations can be added with the "declarer_tables_interfaces" pipeline.

Such a declaration might look like:

```
// suggest a join between sections and documents
$tables_jointures['spip_rubriques'][]= 'documents_liens';
```

```
// suggest a join between articles and authors, specifying
the join field explicitly
$tables_jointures['spip_articles']['id_auteur']=
'auteurs_articles';
```

This shows the links that are possible between tables, When 2 tables can have several fields to link between them, the example above shows how to specify the linking field precisely.

Exceptions

It is even possible to create joins by calling non-existent fields, as demonstrated by the sample criterion `{titre_mot=yy}`, which can lead to a join on the "spip_mots" table, even though the "titre_mot" SQL field does not exist in that table. This is done as shown below:

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

Automating joins

When they have not been explicitly declared to SPIP, joins are calculated where they are possible. To do this, SPIP compares the names of the fields of the various tables.

For example, if an `AUTEURS` loop looks for a criteria field that is not in its table, perhaps `{prenom=Daniel}` for example, SPIP will go and look in the other tables that it knows of and which have fields with the same names as the auteur table (like the `id_auteur` key field), to see if such tables have the "prenom" field being requested. If one of the tables does have that field, then a join will be made between these two tables.

For example, if a custom table `AUTEURS_ELARGIS` (extended_authors) exists (as it does for the "Inscription 2" plugin) with both of the fields "id_auteur" and "prenom", a join would be made to enable the previously mentioned loop criterion to operate correctly.

object, id_object
SPIP 2.0 introduced a new method of searching for joins. The primary keys of one table, in this case "id_auteur" for the `spip_auteurs` table, as well as being searched for in the field names of the other tables, are also searched for in tables that have the pair of fields "object" and "id_object", where "objet=auteur" in our example. In the current SPIP standard database schema, this is actually the case for the `spip_documents_liens` table.

Forcing joins

SPIP's automatic detection capabilities are sometimes limited, and so two syntax variants are offered for forcing table joins or the fields of the tables to be used.

```
// forcing a particular table
<BOUCLE_table(TABLE1 table2 tablen){...}>
// forcing a field in a table
<BOUCLE_table(TABLE){table.field}>
```

Example

These two loops select articles where an author has a name containing "na" (like "Diana", "Joanna", etc.).

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

Note, however, there is a considerable difference between these examples: at present, only the first one declaring all of the tables will make it possible to display a field `#FIELD` from another table. Therefore, `#NOM` will only be provided in the first loop.

Accessing multiple databases

SPIP can easily read any MySQL, Postgres or SQLite databases, and displays their content within structures defined in the SPIP templates.

Declaring another database

In order to access another database, SPIP needs to have access codes to that database. At the time of writing, secondary declared databases are correctly handled in read mode. Writing to such databases, however, is not yet handled by SPIP 2.

To declare another database, there are two possible solutions:

- use the standard graphical interface defined for that purpose (Configuration > Site maintenance > Declare another database)
- write your own connector file following the syntax defined for that purpose, and store it in the `config/` directory (or the directory defined by the constant `_DIR_CONNECT`).

The connector file `config/xx.php`

For a connector file called `tarabiscote.php`, its content would be:

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

We would then call the `spip_connect_db()` function using the following arguments in that order:

1. the address of the sql server
2. the connecting port number, if necessary
3. the username
4. the password
5. the database name
6. the server type (mysql, pg, sqlite2, sqlite3...)
7. the table prefix
8. are users connected using ldap ?

Accessing a declared database

Every additionally declared database can be accessed using SPIP loops as follows:

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

The **name** parameter corresponds to the name of the connector file.

Example

In testing with WordPress some time ago, the author was able to establish a functional database link. By creating a `wordpress.php` connector file, it was possible to recover the last 5 published articles with the code shown below:

```
<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>
```

The "connect" URL parameter

When it has not been specified explicitly with a connector file for use within loops, SPIP uses the default connector file (often named `connect.php`).

For all of these loops, we can request a specific connection that will then be applied by using the URL parameter `connect=name`.

Example

Say you have 2 SPIP sites with different templates (site A and site B). By copying the connector file for site A over to site B (and renaming it as `A.php`) and vice versa for site B, you can then navigate the sites in various combinations:

- `http://A/` (the contents of site A appear using template A)

- `http://B/` (the contents of site B appear using template B)
- `http://A/?connect=B` (the contents of site B appear using template A)
- `http://B/?connect=A` (the contents of site A appear using template B)

In summary, passing `connect=name` in the URL makes it possible to use the "name" connector file for all the loops in the templates that do not have an explicit connector defined, such as `<BOUCLE_a(ARTICLES)>`.

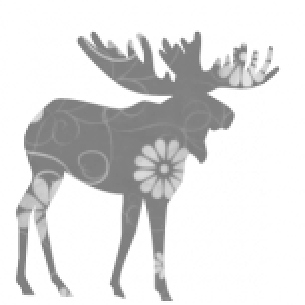
Inclure with a connector parameter

It is possible to pass a particular connection as a parameter when using an include:

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

An include does not automatically pass the parent connection: to propagate a connection, you need to specify it as a criterion in the include itself:

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



Contents of the directories

This chapter will clarify the purpose of the various directories used by SPIP. In some cases, it will address the manner in which new elements are created in these directories.

The list of directories

Name	Description
config (p.90)	Database connection identifiers and site options
ecrire/action (p.93)	Handles the actions which affect the contents of the database
ecrire/auth (p.93)	Manage user authentications
ecrire/balise (p.93)	Declarations of dynamic tags and generic tags
ecrire/base (p.94)	APIs relating to the database and SQL table declarations
ecrire/charsets (p.94)	Character encoding translation sets
ecrire/configuration (p.94)	Configuration components for SPIP's private (back-end) area
ecrire/exec (p.94)	Viewing pages in the private area (PHP code)
ecrire/genie (p.94)	Periodic tasks to be run by the (`cron`) "wizard"
ecrire/inc (p.95)	Libraries and various APIs
ecrire/install (p.95)	SPIP's installation procedures
ecrire/lang (p.95)	The localisation (language) files
ecrire/maj (p.95)	The database update procedures
ecrire/notifications (p.95)	Functions for notifications and contents of notification emails
ecrire/plugins (p.96)	Code relating to the installation and management of plugins
ecrire/public (p.96)	The compiler and cache manager

Name	Description
ecrire/req (p.96)	The database drivers
ecrire/typographie (p.96)	Typographical corrections
ecrire/urls (p.97)	URL rewriting conventions
ecrire/xml (p.97)	XML parser and verifier
extensions (p.90)	The directory for plugins which can not be deactivated
IMG (p.91)	Storage of site documents
lib (p.91)	External libraries added by plugins
local (p.91)	Storage location for caches of images, CSS and Javascript files
plugins (p.91)	The plugins directory
prive/contenu (p.98)	Templates for viewing objects in the private area
prive/editer (p.98)	Templates used for the editing forms for SPIP objects
prive/exec (p.98)	Viewing pages in the private area (coded as SPIP template files)
prive/formulaires (p.98)	Editing forms for editorial objects
prive/images (p.98)	Image files used in the private area
prive/infos (p.99)	Templates for the information panels displayed for SPIP objects in the private area
prive/javascript (p.99)	JavaScript scripts
prive/modeles (p.99)	The standard model "snippets" provided by SPIP
prive/rss (p.99)	Templates that generate the RSS feeds for monitoring editorial changes made in the private zone

Name	Description
prive/stats (p.99)	Templates used for displaying site statistics
prive/transmettre (p.100)	Templates used for CSV exports
prive/vignettes (p.100)	Icon images for attached documents
squelettes (p.92)	Customisations of templates and other standard files.
squelettes-dist (p.92)	The default set of site templates
tmp (p.92)	Temporary files and cache files

config

The `config` directory stores configuration details for the site, such as identifiers for the standard SPIP database connection (in `connect.php`) or for other external databases, the `mes_options.php` file used to define site options, and the security screen (`ecran_securite.php`) which makes it possible to rapidly recover from some system failures that have been observed.

extensions

The `extensions` directory is used to define plugins which are pre-installed, pre-activated and which cannot be later deactivated, as part of the standard SPIP installation. All that is required is to store the desired plugins in this directory.

In the SPIP standard distribution, some plugins are included by default:

- "Compresseur", to compress Javascript, CSS and HTML files,
- "Filtres images et couleurs" (Image and colour filters), providing functions for graphical and typographical manipulation,
- "Porte Plume", offering an editor’s toolbar to insert SPIP shortcuts,
- "SafeHTML", for cleaning out unwanted or dangerous items from forum contributions and syndicated site content feeds (RSS).

IMG

The IMG/ directory contains all of the editorial documents added to the site, classified (by default) according to their file extensions in to various sub-directories. It retains the image-related directory name from earlier days when additional "documents" for SPIP only involved additional image files.

lib

This directory (which is not created in the default install) enables plugins to share external common libraries, which are therefore to be downloaded and extracted into this directory.

local

This directory houses the caches that are generated for typographical images, image resizing, graphics manipulations, and CSS and JavaScript compressions; that is, all the caches that require HTTP access.

For more information, please refer to the following articles:

- The CSS and JavaScript caches (p.223)
- The image processing cache (p.223)

plugins

The plugins directory is used for installing plugins which will be activated or deactivated from the private area’s plugin configuration administration page. Plugins are typically downloaded using FTP and extracted into this directory, one sub-directory per plugin. The existence of a directory called plugins/auto that is write accessible will enable webmasters to automatically download plugins from within the private area interface without the need to use FTP or to UNZIP the plugins manually.

The most recent collection of plugins is located at http://files.spip.org/spip-zone/, but other plugins may be available from other sites. As with any externally contributed code, be very careful what you install, and if in doubt, only use plugins from SPIP’s own collection.

squelettes

The `squelettes` directory is not created by the SPIP default installation. Once created manually, it makes it possible for the webmaster to override the original files included in SPIP and any installed plugins, these mainly being the default templates for SPIP. This directory is also used to create your own special template files and to store any files that are specific to your site. The advantage of using this directory for customisation is to prevent your changes being overwritten any time that you update SPIP itself or any of the plugins that you might be using.

Just as much as you need to create this directory manually to store customised versions of files that are normally located in the squelettes directory, so must you also create sub-directories for any overriding files you create that would normally exist in sub-directories of the standard squelettes (or plugin) directory.

squelettes-dist

This directory contains the set of template files supplied with the standard SPIP installation. It also contains sub-directories of files for the public forms and models (snippets). Do not delete or overwrite these files if you can avoid doing so, as any changes you make to these files will be lost if you update your version of SPIP - you are advised to using the squelettes (p.0) directory for your customisations.

tmp

This directory contains any files of a temporary nature, including those for caches and logs, and which are not accessible via HTTP. It contains sub-directories specially created for:

- the cache (`cache`),
- any backups made through the admin interface (`dump`),
- sessions for registered users (`sessions`),
- documents sent by FTP (`upload`)
- visits statistics (`visites`)

ecrire

This directory contains all of the SPIP code to make the engine tick!

ecrire/action

This directory is intended to store code used to modify the contents of the database. Most of the actions are secured, in such a way as to confirm both that:

- the author performing the action is authorised to do so, and
- the action is actually being requested by the person currently logged in.

Upon completion of these processes, a redirection is made to a URL that generally makes reference to the recent action call. Please refer to the section on actions and processes (p.205) for further details.

ecrire/auth

The `ecrire/auth` directory contains the various scripts used to manage user connections. One file manages authentication using the SPIP methodology, and the other works for an LDAP directory.

The authentication processes are relatively complex as they involve numerous security checks. An API defines the various stages of authentication and the creation of new users. Please refer to the section on authentication (p.209) for further details.

ecrire/balise

The `ecrire/balise` directory is used to define

- dynamic tags, meaning those that involve calculations for every page reference and generation, and
- generic tags, meaning those starting with the same prefix and performing shared actions (`#URL_`, `#FORMULAIRE_`, ...)

The static tags, however, are declared in the ecrire/public/balises.php file, or when they belong to plugins, within the function definition files for each plugin.

Please refer to the section on tags (p.183) for further details.

ecrire/base

This folder contains code related to the database: the table definitions, SQL abstraction functions, and functions used for creating and updating the SQL tables.

A complete chapter is devoted to the database functions: SQL access (p.253).

ecrire/charsets

This directory contains the files used for translating character encodings, generally called by the `ecrire/inc/charsets.php` file.

ecrire/configuration

This directory contains components used in the configuration pages of SPIP's private zone. Each file corresponds to one particular configuration frame.

ecrire/exec

The `ecrire/exec` directory is used to store the PHP files used to display pages in the private zone using the `?exec=name` parameter. However, it is becoming increasingly common to use "regular" SPIP template files for these pages, which are installed in the prive/exec (p.98) directory.

A detailed explanation is provided in the section devoted to Creating pages in the private zone (p.190).

ecrire/genie

The `ecrire/genie` directory is used to store functions to run periodically by the "génie" aka wizard (which are similar to but not quite the same as `cron` tasks), each of which normally has a dedicated file to be run for each such task.

Please refer to the section on this subject: Periodic tasks (cron) (p.227).

ecrire/inc

This directory contains most of the PHP libraries created for use with SPIP. Some of these libraries are systematically loaded for every SPIP site. This is the case for `ecrire/inc/utils.php`, which contains the core and start-up functions, and also for `ecrire/inc/flock.php`, which takes care of file locking and access.

ecrire/install

The `ecrire/install` directory contains everything required for SPIP installation. The various files in this directory comprise the installation steps and are called from the `ecrire/exec/install.php` file.

ecrire/lang

The `ecrire/lang` directory contains the various translation files for the SPIP public and private interfaces. These translations are provided by the use of 3 files for each language (where `xx` is a specific language code):

- `public_xx.php` translates text strings appearing in the public template files,
- `ecrire_xx.php` translates text strings in the private zone (action, exec),
- `spip_xx.php` translates... the others (inc,prive,formulaires,modeles) ?!

ecrire/maj

This directory contains the update routines for the database as it progresses through different versions of SPIP. For older versions, it also contains the structure of the original database. This makes it possible to re-import SPIP backups from previous versions (back to SPIP 1.8.3) normally without any problems.

ecrire/notifications

This directory contains the various functions called by SPIP's notifications API in the `ecrire/inc/notifications.php` file. The notifications make it possible (by default) to send emails after certain events occur within SPIP, such as the arrival of a new message in a forum or the proposal of a new article for publication.

This directory also stores certain SPIP templates that are used to build the email text messages included in these notifications.

ecrire/plugins

The `ecrire/plugins` directory contains all of the code used for SPIP's plugins, as well as code for the extensions (plugins which can not be deactivated) and external libraries (the `lib/` directory). This naturally includes the code for listing the plugins, determining their dependencies, processing their `plugin.xml` files, and managing the various caches used by the plugins (please read the information about the plugins cache (p.221))...

ecrire/public

The rather badly named `ecrire/public` directory contains the various files involved in searching, analysing, compilation and debugging of the SPIP templates, the creation of the pages generated by the templates, and the management of their corresponding caches.

Some further details on the compilation of the templates (p.210) are available in the corresponding section.

ecrire/req

The `ecrire/req` directory contains the translators used to transform SPIP's SQL abstraction functions and queries for the corresponding database engines.

Four drivers are available: MySQL, PostGres, SQLite 2 and SQLite 3.

ecrire/typographie

This directory contains the typographical corrections for French and English, applied by calling the `typo` function. This processing is required to resolve typographical differences between how text is entered and how it is displayed, such as the use of different character sequences for left and right quotation marks in different languages, for example.

ecrire/urls

The `ecrire/urls` directory contains the code used to drive the URL rewriting systems offered by SPIP (propre, html, arborescent...). The API for these URL rewriting systems makes it possible to construct URLs based on a given context and, in complementary fashion, to identify an object and its identifier based on a user-requested URL.

Any custom-built objects added to SPIP may need to be addressed by whatever URL rewriting method is selected for the site.

ecrire/xml

This directory contains the functions used for analysing XML strings and transforming them into PHP arrays. A verification tool is also available so that DTD page errors can be identified.

prive

The `prive` directory stores all of the templates used in SPIP's private zone, as well as certain CSS stylesheets applied to the private zone.

prive/contenu

The `prive/contenu` directory contains the templates used to **display** the contents of SPIP objects, such as for articles (the `article.html` file) in the private zone.

prive/editer

The `prive/editer` directory contains the templates of the forms used for **editing** the SPIP objects.

prive/exec

The `prive/exec` directory is used to store SPIP template files used for displaying pages in the private zone using the `?exec=name` parameter. This directory is not used by the core of SPIP, but some plugins may use it.

A detailed explanation is provided in the section on creating pages in the private zone (p.190).

prive/formulaires

The `prive/formulaires` directory contains the CVT editing forms for SPIP editorial objects.

prive/images

This directory stores all of the images and icons that are used in the private zone and those used during the installation procedures.

prive/infos

The `prive/infos` directory contains the templates for the information (and sometimes action button) panels for SPIP objects in the private zone. These panels typically include the object identifier, the object status, and some statistics (e.g. the number of articles in a section, the number of times the article has been visited, etc.).

prive/javascript

This directory contains the JavaScript scripts, including jQuery, that are used in the private zone and for certain calls from the jquery_plugins (p.166) pipeline from the public site as well.

prive/modeles

This directory contains the reusable SPIP code "snippets" that can be used within object texts by site contributors, such as `<imgXX>` and `<docXX>`. They can also be used within other customised template files by using the `#MODELE` tag.

prive/rss

These templates generate the RSS feeds for monitoring site changes in the private zone, and are called by the `prive/rss.html` file with a URL constructed by the `bouton_spip_rss` function (declared in `ecrire/inc/presentation.php`).

prive/stats

Templates used for displaying the statistics maintained in SPIP's internal statistics files.

prive/transmettre

The `prive/transmettre` directory contains the templates used to generate CSV data, called from the `prive/transmettre.html` template file.

prive/vignettes

This directory stores the various images that each correspond to an extension for a class of attached documents. The `#LOGO_DOCUMENT` tag returns the applicable icon if no specific icon has been assigned to an individual document. Other functions related to these image vignettes are found in `ecrire/inc/documents.php`.

Extending SPIP

One long-term goal of SPIP has been adaptability. There are many ways to refine and extend it according to the requirements of each particular web site, or to create new functionality not included in the core modules.

This section explains the ways that programmers can use to extend SPIP.

Introduction

Templates, plug-ins, access paths, the `_dist()` functions and how to use and override them... This section explains it all.

Templates or plug-ins?

Use the "squelettes" folder

The `squelettes/` folder is used to store all the files required for the operation of your site and to customise its graphic design: templates (or “squelettes”, images, JavaScript and CSS files, PHP libraries, ...).

Or create a plug-in

A plug-in, stored in a folder like `plugins/name_of_the_plugin/`, can *also* contain any or all of the files that your site might require, just like the `squelettes/` folder. Additionally, a plug-in supports some additional actions, essentially those required to install and uninstall the plug-in.

So, is it best to write a plug-in or simply use the `squelettes` folder?

Generally speaking, the `squelettes/` folder is used to store everything that is specific to a particular site. Only when a piece of code is generic and reusable does it makes sense to package it as a plug-in.

Declaring options

When a visitor requests a page (whether or not it is in the cache), SPIP carries out a number of actions, one of which is to load the “options” files. In these files we can, for example, define new constants or modify global variables that control the way SPIP operates.

These options can be created in the file `config/mes_options.php` or in any plug-in by declaring the name of the file in `plugin.xml` like this: `<options>pluginprefix_options.php</options>`.

All options files (those of the site, and then those of all the plugins) are loaded every time a page request is made in the public zone or the private zone, so they should be as simple and as small as possible.

This example, from a contribution called “switcher”, will change the set of templates used by the site (or, strictly speaking, the name of the templates folder) depending on the value of the `var_skel` parameter in the URL.

```
<?php
// 'name' => 'template path'
$squelettes = array(
  '2008'=>'squelettes/2008',
  '2007'=>'squelettes/2007',
);
// If a particular set of templates are requested (and
exist), set a cookie, otherwise delete the cookie
if (isset($_GET['var_skel'])) {
  if (isset($squelettes[$_GET['var_skel']]))
    setcookie('spip_skel', $_COOKIE['spip_skel'] =
$_GET['var_skel'], NULL, '/');
  else
    setcookie('spip_skel', $_COOKIE['spip_skel'] = '',
-24*3600, '/');
}
// If a particular template path is permitted, define it as
the templates folder
if (isset($_COOKIE['spip_skel']) AND
isset($squelettes[$_COOKIE['spip_skel']]))
  $GLOBALS['dossier_squelettes'] =
$squelettes[$_COOKIE['spip_skel']];
?>
```

Declaring new functions

The “_fonctions” files are loaded automatically by SPIP, but — unlike the “_options” files (p.102) — only when it needs to evaluate a template to generate a new page.

These files make it possible, for example, to define new filters that can be used in templates. If you create a `squelettes/mes_fonctions.php` file containing the following code, then you will be able to use the `hello_world` filter in your templates (useless though it is!):

```
<?php
function filtre_hello_world($v, $add){
    return "Title:" . $v . ' // Followed by: ' . $add;
}
?>
```

```
[(#TITRE|hello_world{this text is added afterwards})]
```

(displays "Title:title of the article // Followed by: this text is added afterwards")

To create such files in a plug-in, you need to add the name of the file in your `plugin.xml` like so: `<fonctions>pluginprefix_fonctions.php</fonctions>`. Each plug-in may contain any number of these declarations (and files).

Functions for specific templates

Sometimes, filters are specific to a single template. It is not always desirable to load all such functions for each and every page. SPIP thus makes it possible to load certain functions only when calculating a particular template.

Such a file should be created in the same folder as the template and named after it, but with `_fonctions.php` instead of `.html`.

Consider the example from above again. If the file named `squelettes/world.html` contains the code `[(#TITRE|hello_world{this text is added afterwards})]`, then the `hello_world` function could be declared in the `squelettes/world_fonctions.php` file. This file will only be loaded when SPIP is generating a page based on the `squelettes/world.html` template.

The concept of path

SPIP uses a large number of functions and templates, contained in various folders. When a script needs to open a file to load a function or to read a template, SPIP will search for it in one of a number of folders. The first matching file found in one of these will be loaded and used.

The folders are perused in the order defined by the constant `SPIP_PATH` and, optionally, using the global variable `$GLOBALS ['dossier_squelettes']`.

The default search path is, in order:

- `squelettes/`
- the plug-in `plugin_B/` (which depends on “plugin A”)
- the plug-in `plugin_A/`
- `squelettes-dist/`
- `prive/`
- `ecrire/`
- `./`

Overriding a file

One of the first possibilities to modify SPIP’s behaviour is to copy one of its files from `ecrire/` into a folder with higher priority (p.104) — a plug-in or `squelettes/` folder, for example — while preserving the folder hierarchy.

Thus, one could modify the way in which SPIP manages the cache by copying `ecrire/public/cacher.php` to `squelettes/public/cacher.php` and then modifying this copy. It is this modified copy which would be loaded by SPIP as it — being in `squelettes/` — has a higher priority than the original.

This technique must be used with full knowledge of the facts. While this technique is very powerful, it is also very sensitive to changes in SPIP. If you use this method, you may find it difficult or impossible to upgrade your site to future versions of SPIP.

Overloading a _dist function

Many of the functions in SPIP are designed to be overridden. These functions have the extension “_dist” in their name. All the `balises` (“tags”), `boucles` (“loops”), and `criteres` (“criteria”) are named like this and can thus be overridden by declaring (perhaps in the file `mes_fonctions.php`) the same function, but without the suffix “_dist” in the name.

For example, the ecrire/public/boucles.php file contains a function called `boucle_ARTICLES_dist`. It can be overloaded by declaring a function like this:

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

Some functions you should know

SPIP contains many extremely useful PHP functions. Some are used more frequently than others and deserve a bit more explanation.

Name	Description
charger_fonction (p.107)	Finds a function
find_all_in_path (p.108)	Finds a list of files
find_in_path (p.108)	Finds a function
include_spip (p.109)	Includes a PHP library
recuperer_fond (p.110)	Returns the results of compiling a template
spip_log (p.112)	Outputs additional data to the logs
trouver_table (p.113)	Provides the description of an SQL table
_request (p.115)	Retrieves a variable from the URL or a form.

charger_fonction

This `charger_fonction()` (translation: `load_function()`) function is used to retrieve the name of an overloadable SPIP function. Whenever an internal function with a `_dist()` suffix is overloaded (by recreating it without that suffix), or whenever all of a file that contains such a function is overloaded, then the correct function to be run must be retrievable at the time that that function is to be executed.

This is what the `charger_fonction()` does. It returns the correct name of the function to be executed.

```
$ma_fonction = charger_fonction('my_function','directory');
$ma_fonction();
```

The searching principle

The function operates as follows:

- if the `directory_my_function` function has already been declared, then that function is returned,
- else `directory_my_function_dist`,
- else try to load a file called `directory/my_function.php` then

- return `directory/my_function` if it exists,
- else `directory/my_function_dist`,
- else return `false`.

Example

Send an email:

```
$envoyer_mail = charger_fonction('envoyer_mail', 'inc');
$envoyer_mail($email_address, $subject, $text_body);
```

find_all_in_path

`find_all_in_path()` returns the list of files that match a specific pattern. Like `find_in_path()` (p.108), these files are searched for in all the directories defined in the SPIP file path.

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

Example

SPIP uses this function to get all the CSS files that the plugins add to the private interface using the files named "prive/style_prive_plugin_*prefix*.html". To do so, it uses the following line of PHP code:

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

find_in_path

The function `find_in_path()` returns the path of a particular function. This function is searched for in the "SPIP path" (p.104).

It accepts 1 or 2 arguments:

- the name or relative path of a file (with its extension)

- optionally, the directory where it is stored.

```
$f = find_in_path("directory/file.ext");
$f = find_in_path("file.ext","directory");
```

Example

If the `pattern/inc-special.html` file exists, calculate $html as the result of compiling this template. Otherwise $html is the result of compiling `pattern/inc-normal.html`.

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

include_spip

The function `include_spip()` includes a PHP file. The difference from PHP's normal `include_once()` is that the file is searched for in the SPIP `path` (p.104), that is, in all the known directories and in the search priority order specified in the SPIP path.

`include_spip()` accepts 1 or 2 arguments:

- the name or relative path of the file (without its .php extension)
- a flag (true by default) that indicates if the file is actually to be included, or if only the path of the file is to be returned.

```
include_spip('fichier');
include_spip('dossier/fichier');
$address = include_spip('fichier');
$address = include_spip('fichier', false); // inclusion is
not performed
```

Example

```
// loads the file containing the functions used on
// the installation pages or error pages
include_spip('inc/minipres');
echo minipres('Bad luck!', 'An error has occurred!');
exit;
```

recuperer_fond

Another function which is extremely important within SPIP, `recuperer_fond()`, is used to return the results of compiling a given template. This is sort of the same as for `<INCLURE{fond=name} />` used in templates but in PHP.

It accepts from 1 to 4 parameters:

- the name and address of the source code file (without extension)
- the compilation context (key/value table)
- a table of options
- the name of the connection file for the database to be used

Simple usage

The data returned is the code generated by the compilation output:

```
$code = recuperer_fond($name, $context);
```

Advanced usage

The `raw` option set to `true` will provide, rather than just the generated code, a table of items calculated by the compilation, which also includes the code (with the key `texte`).

What does this table contain then? The text, the address of the template source (tagged "source"), the filename of the PHP cache generated by the compilation (tagged "squelette"), an indicator of the presence of PHP in the generated cache file (tagged "process_ins"), and various other values included in the compilation context (the language and data are automatically added since they have not been passed as parameters).

Example

Retrieve the contents of a file `/inclure/inc-liste-articles.html` by passing the identifier of the desired section (rubrique) in the context:

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

Using the `raw` option:
Here is a small test with a template called "ki.html" containing only the text "hop". In this example, the results are output to a log file called (`tmp/test.log`).

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

These are the results that will be output to `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

This function is used to record actions out to the log files (generally located in the `tmp/log/` directory).

This function accepts 1 or 2 arguments. With one argument, it will write out to just the `spip.log` file. With two arguments it will write out to both a separate log file and also to the `spip.log`.

```
<?php
spip_log($tableau);
spip_log($tableau, 'second_file');
spip_log("adding field $champ into the $table
table","my_plugin");
?>
```

When a table is passed to the log function, SPIP will write out the output from `print_r()` into the log file. For each file requested, in this case `spip` (by default) and `second_file`, SPIP will create or add the contents of the first argument, but not just anywhere. If the script is run from the private interface, it will write out to "prive_spip.log" or to "prive_second_file.log", otherwise it will write to "spip.log" or "second_file.log".

The configuration file `ecrire/inc_version.php` defines the maximum size of the log files. When a given log file exceeds this pre-determined file size, it is renamed `prive_spip.log.n` (n will automatically increment). The number of such files that may exist is also configurable. It is also possible to deactivate the logs by setting one of these specified values to zero within the `mes_options.php` file.

```
$GLOBALS['nombre_de_logs'] = 4; // maximum 4 log files
$GLOBALS['taille_des_logs'] = 100; // maximum 100 KB each
```

There is also a _MAX_LOG constant (set to 100 by default) which specifies the number of entries that each call from a given page may write to a log file. With this default setting, after 100 calls are made to spip_log() from any particular script, the log function will refuse to write any further content for that script.

trouver_table

The trouver_table() function (base_trouver_table_dist) is declared in ecrire/base/trouver_table.php and is used to obtain a description for an SQL table. It provides a mechanism to retrieve the list of columns, keys, declared joins and some other information details.

As an overloadable function, it is used with charger_fonction (p.107):

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

Its parameters are:

1. $table: the name of the table ('spip_articles' or 'articles')
2. $serveur: optional, the name of the SQL connection, which is by default the same as that for the SPIP installation itself.

The $desc table returned is structured as follows:

```
array(
    'field' => array('column' => 'description'),
    'key' => array(
        'PRIMARY KEY' => 'column',
        'KEY name' => 'column' // or 'column1, column2'
    ),
    'join' => array('column' => 'column'),
    'table' => 'spip_tables'
    'id_table' => $table,
    'connexion' => 'connection_name',
    'titre' => 'column_title AS titre, column_language AS 
lang'
);
```

- The field key is an associative table listing all of the table's columns and their SQL descriptions,

- `key` is another table listing the primary and secondary keys,
- `join` lists the columns of any joins, if declared in the descriptions of the principal or auxiliary tables
- `table` is the actual name of the table (without prefix: if the table prefix is different from "spip", then it will be "spip_tables" that will be returned),
- `id_table` is the given `$table` parameter,
- `connexion` is the name of the connection file, if different from that of the installation,
- `titre` is an SQL SELECT declaration indicating where is the column title or where is the column language (used amongst other things to calculate the URLs); e.g. "`titre, lang`", or "`name AS title, '' AS lang`"

This function caches (p.221) the result of the analysis in order to avoid repetitive disruptive access to the SQL server. To force a recalculation of this cache, the function must be called with an empty string:

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

Note: Whenever a table is requested without the "spip" prefix, it is the name of the table with the prefix assigned for the site that will be returned (so long as the table is declared in SPIP). Requesting a "spip_tables" table will look for the real existence of that table (the prefix is not replaced by that used for the site). In the future, an option will probably be added to the `trouver_table()` function, as there is already for sql_showtable (p.299) in order to be able to automatically modify the prefix.

Example

The `creer_champs_extras()` function from the "Champs Extras" plugin is used to create SQL columns described by the "ChampExtra" object instances passed (`$c->table` is the name of the SQL table, `$c->champ` is that of the column). The function returns `false` if a column has not been created:

```
function creer_champs_extras($champs) {
    // the function updates the tables in question using
maj_tables()
    // [...]
    // It then tests if the new fields have actually been
created:
    // for each column to be created, check that is
actually exists now!
    $trouver_table =
charger_fonction('trouver_table','base');
    $trouver_table(''); // recreate the description of
the 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

The `_request()` function is used to retrieve the values of variables sent by the site visitor, either through a URL or through a posted form.

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

Security principles

These functions must not be located just anywhere amongst the SPIP files, in order to be able to carefully restrict the possible locations likely to be targeted for pirating. The elements provided by user input must only be retrievable from

- action files (in the `action/` directory),
- the private zone display files (in the `exec/` directory),
- some very rare dynamic tag functions (in the `balise/` directory), or
- in the files that process web forms (in the `formulaires/` directory).

As an additional general rule, it is necessary to verify that the variable type received is indeed in the expected format (to eliminate any risk of hacking, even if SPIP already performs a first level cleaning of input data): for example, if you expect a number, then you must apply the `intval()` function (which will transform any text into its numeric value):

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

Retrieval from a table

If you want to retrieve only certain specific values that exist in a table, you can pass that table as a second parameter:

```
// retrieve if there is a $table['name']
$name = _request('name', $table);
```

Example

Retrieve only from the values that were passed in the URL:

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

Pipelines

Some parts of the code define “pipelines”. They provide one of the best ways to modify or adapt the behaviour of SPIP.

Definition

A pipeline (p.323) is used to pass code through one or more intermediary functions to complete or modify that code.

Declaration within a plugin

Any plugin can use an existing pipeline. To do so, it declares it in the `plugin.xml` file as illustrated here below:

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

- Nom: specifies the name of the pipeline to be used,
- Inclure: specifies the name of the file that contains the function to be executed when calling the pipeline (`prefixPlugin_PipelineName()`).

Declaration without using a plugin

One usage of a pipeline outside that of plugins does remain possible. In this case, it must be declared directly in the `config/mes_options.php` file:

```
$GLOBALS['spip_pipeline']['name_of_the_pipeline'] .=
"|name_of_the_function";
// Example of adding into the "insert_head" pipeline:
$GLOBALS['spip_pipeline']['insert_head'] .=
"|name_of_the_function";

function name_of_the_function($flux) {
    return $flux .= "This text will be appended";
}
```

The function called must be known at the time that the pipeline is called, with the simplest solution being to declare it as above with a `name_of_the_function` function defined in the options file.

List of current pipelines

The default pipelines defined in SPIP are listed in the file ecrire/inc_version.php. However, plugins are able to create new ones.

There are several types of pipelines: some of them deal with typographical modifications, others deal with database modifications, or pages that are only displayed in the private area, etc.

Declaring a new pipeline

The pipeline must first be declared in a global options file like this:
`$GLOBALS['spip_pipeline']['newPipelineName'] = '';`

The name of this pipeline must be a key of the associative array `$GLOBALS['spip_pipeline']`.

Then, the pipelines must be called from somewhere, either in a template or a PHP file:

- Templates: `#PIPELINE{newPipelineName, initial content}`
- PHP: `$data = pipeline("newPipelineName", "initial content");`.

The `#PIPELINE` tag and the `pipeline()` function both use the same arguments. The first argument is the name (in our example, it's "newPipelineName"). The other one is the data that is sent to the hook.

The pipeline is a channel by which information is transmitted sequentially. Each plugin that has declared this pipeline is party to this channel, and so can complete or modify the input data, and transmit the result to the next part. The result of the pipeline is the result of the last process that has been applied.

Contextual pipelines

It is often necessary to pass contextual arguments to the pipeline on top of the data returned by the pipeline. This is possible by using a table with at least 2 keys, named `"args"` and `"data"`.

When the last function of the pipeline chain is called, only the value of `data` is returned.

```
$data = pipeline('newPipeline',array(
    'args'=>array(
        'id_article'=>$id_article
    ),
    'data'=>"initial content"
);
```

```
[(#PIPELINE{newPipeline,
    [(#ARRAY{
        args,[(#ARRAY{id_article,#ID_ARTICLE})],
        data,initial content
    })]})]
```

Pipeline details

This section describes the use of some of SPIP's pipelines.

Name	Description
rechercher_liste_des_champs (p.122)	Defines the fields and weightings to apply for searches in a table
accueil_encours (p.123)	Adds content to the centre of the home page
accueil_gadget (p.123)	Adds links above the content of the home page
accueil_informations (p.124)	Provides statistics about editorial objects on the home page
affichage_entetes_final (p.125)	Modifies the returned page headers
affichage_final (p.126)	Performs processing just before publishing public web pages
afficher_config_objet (p.127)	Adds elements to the configuration panels for editorial objects
afficher_contenu_objet (p.128)	Modifies or adds to the view form of an object in the private interface
afficher_fiche_objet (p.129)	Adds content to the view screens of editorial objects
affiche_droite (p.130)	Adds content to the "right-hand side" column in the private area
affiche_enfants (p.131)	Modifies or adds to the contents of the lists showing the children of an object in the private area
affiche_gauche (p.131)	Adds contents to the "left-hand side" column in the private area
affiche_hierarchie (p.132)	Modifies the HTML code of the breadcrumb path in the private area
ajouter_boutons (p.134)	Adds buttons to the menu bar in the private area
ajouter_onglets (p.136)	Adds tabs to the pages of the private area
alertes_auteur (p.138)	Adds warnings to the author logged into the private area

Name	Description
autoriser (p.139)	Loads the authorisation functions
body_prive (p.141)	Inserts content after the `<body>` section in the private area
boite_infos (p.141)	Display data about objects in the info boxes within the private zone
compter_contributions_auteur (p.143)	Counts an author's contributions
declarer_tables_auxiliaires (p.145)	Declares "auxiliary" SQL tables
declarer_tables_interfaces (p.146)	Declares additional data in SQL tables (alias, processes, joins, ...)
declarer_url_objets (p.155)	Enables standard URLs for a new editorial object
definir_session (p.157)	Defines the parameters that identify the visitor specific caches
delete_statistiques (p.159)	Triggered when the statistics tables are purged
delete_tables (p.159)	Triggered during database purges
editer_contenu_objet (p.159)	Modifies the HTML content of forms
formulaire_charger (p.160)	Modifies the table of values returned by the `charger` function for a CVT form
formulaire_traiter (p.161)	Modifies the table returned by the `traiter` function for a CVT form or perform some added processes
formulaire_verifier (p.162)	Modifies the array returned by the `verifier` function for a CVT form
header_prive (p.163)	Adds content to the `<head>` section of private area pages
insert_head_css (p.166)	Adds CSS code for the public site
lister_tables_noerase (p.167)	Lists the tables that are not to be purged before a backup restore
lister_tables_noexport (p.167)	Lists the SQL tables that are not to be backed up

Name	Description
lister_tables_noimport (p.168)	Lists the SQL tables that are not to be imported
optimiser_base_disparus (p.168)	Cleans out orphan records from the database
post_typo (p.170)	Modifies text after the typographical processes have been applied
pre_insertion (p.172)	Adds default content when a database insert is executed
pre_liens (p.173)	Processes typographical shortcuts relating to links
pre_typo (p.174)	Modifies text before the typographical processes are applied
recuperer_fond (p.176)	Modifies the results of a template compilation
rubrique_encours (p.177)	Adds content to the "Submitted for evaluation" area for sections
taches_generales_cron (p.179)	Sets up periodic tasks
trig_supprimer_objets_lies (p.180)	Deletes the links for an object when an object is deleted
... and the rest of them (p.181)	Those that are yet to be documented

rechercher_liste_des_champs

This pipeline specifies the fields to be considered when a search is performed on a given table.

It manipulates a 2-dimensional associative array composed like this:

- the first key is the name of a SPIP object (article, rubrique...).
- the other key is the name of a field (titre, texte...) to take into account for the search.
- the value is the weighting coefficient: the higher this value is, the more points are attributed to a result found in the corresponding field.

Example

```
function
prefixPlugin_rechercher_liste_des_champs($tables){
    // add a field 'town' to the articles
    $tables['article']['town'] = 3;
    // hide a field from the search process
    unset($tables['rubrique']['descriptif']);
    return $tables;
}
```

accueil_encours

This pipeline is used to add content to the centre of the home page in the private zone, e.g. to display new articles proposed for publication.

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

This pipeline accepts a text string as argument and returns the supplemented text as output.

Example

The "breves" plugin, if it existed, might use this pipeline to add the list of recently proposed news items:

```
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

This pipeline is used to add links above the content of the home page in the private zone, within the frame that lists the various actions available (create a section, an article, a news item, etc.).

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

This pipeline accepts a text argument and returns the supplemented text as output.

Example

The "breves" plugin, if it existed, would use this pipeline to add a link at the top to allow the user to create or view the list of news item depending on the status of the author currently connected:

```
function breves_accueil_gadgets($texte){
    if ($GLOBALS['meta']['activer_breves'] != 'non') {
        // create, otherwise view
        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

This pipeline is used to add statistical data about the editorial objects into the side navigation panel on the home page.

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

It accepts text as a parameter that it may complete and return as output.

Example

The "breves" plugin, if it existed, might use this pipeline to add the number of news items awaiting validation for publication:

```
function breves_accueil_informations($texte){
    include_spip('base/abstract_sql');
    $q = sql_select("COUNT(*) AS cnt, statut",
'spip_breves', '', 'statut', '','', "COUNT(*)<>0");
    // processes operating on the text depending on the
resulting output
    // ...
    return $texte;
}
```

affichage_entetes_final

This pipeline, called for every SPIP public page when it is displayed, accepts a table parameter containing the list of page headers. It then allows the modification of or addition to those headers. It is called just before the affichage_final (p.126) pipeline, which receives the text string output by this function.

This pipeline is called in ecrire/public.php, taking and returning a table parameter containing the various page headers:

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

Example

One usage of this pipeline is to enable site statistics generation, since by knowing the headers sent out (and therefore the page type) and certain other environmental parameters, we can make entries into a visitor statistics table (the action code has been simplified for reference purposes here and comes from the "Statistiques" plugin):

```
// for html pages generated, count the visits.
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

This pipeline is called at the time that the contents of a page are being sent back to the visitor’s browser. It accepts a text argument (most commonly the HTML page) that it may edit or add to. The modifications are not stored in the cache by SPIP.

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

This is a pipeline frequently used by plugins that enable a wide range of actions. Nonetheless, since the results of the pipeline are not stored in the cache, and this pipeline is called for every page displayed, it would be wise to limit its usage to functions that are not too resource intensive.

Example

The "XSPF" plugin, which is used to generate multimedia galleries, adds a JavaScript component only to pages that require it, as shown below:

```
function xspf_affichage_final($page) {
    // check to see if the page has any "player" class
components
    if (strpos($page, 'class="xspf_player"')===FALSE)
        return $page;
    // If so, add the swfobject js
    $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);
}
```

The "target" plugin opens external links in a new window, (oh, yes, even if that’s not a terribly popular idea these days!), and so it systematically changes "outward" links so that they have an external target attribute:

```
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

This pipeline is used to add elements into the configuration panels for SPIP objects.

It is called as demonstrated in ecrire/exec/articles.php:

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

As of writing, it only applies to articles and adds its content into the "Forum and Petitions" panel.

Example

The "Forum" plugin adds moderation control settings (no forum, registration required, post-moderation...) for each article, using the following code:

```
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

This pipeline is used to modify or complete the contents of the pages in the private interface that are used to display objects, such as the page for viewing an article.

It is called during the life of any object in the private zone, by passing the type and identifier of the object in the `args` parameter, and the HTML code for the object view in the `data` parameter:

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

Example

The "Métadonnées Photos" (photo metadata) plugin adds a photo usage graphic and the EXIF data underneath the description of the JPG images which are attached to the current object, using the code shown below:

```
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

This pipeline is used to add content into the view pages for editorial objects in the private zone. It is called as demonstrated below:

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

As of writing, it is used for adding elements to the "articles" and "navigation" (sections) pages.

Example

The "Forum" plugin uses this pipeline to add buttons enabling discussion of an article. It does this by adding a forum reference to the footer of the article page:

```
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

This pipeline is used to add content into the "right-hand" column (which is not necessarily actually on the right hand side, depending on the user’s preference settings and language) on the "exec" pages in the private zone. This column normally contains "horizontal" navigation links related to the currently displayed contents, such as in the "In the same section" panel which lists recently published articles in the same section as the current article.

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

This pipeline accepts the "exec" page name displayed as a parameter, as well as an optional identifier for the object currently being read, e.g. "id_rubrique".

Example

The "odt2spip" plugin, used to create SPIP articles based on OpenOffice text documents (with the `.odt` file extension), employs this pipeline to add a form to the section view screen in order to enter an `odt` filename:

```
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

This pipeline is used to add to or modify the contents of the lists showing the children of an object. The `args` parameter accepts the name of the current page and the object identifier, and its `data` parameter accepts the HTML code showing the object's children. This pipeline is actually only called from a single location: on the section navigation page.

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

affiche_gauche

This pipeline is used to add content to the "left-hand" column in the private zone pages. This column generally contains links or forms relating to the currently displayed content, like the form for adding a logo for the current section/article.

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

This pipeline accepts the name of the currently displayed "exec" page as an argument, as well as the possible identifier for the object currently being displayed, such as the "id_article".

Example

The "spip bisous" plugin, which is used to send kisses(bisous) amongst site authors, employs this pipeline to display the list of kisses received and sent for the author pages:

```
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

The "affiche_hierarchie" pipeline is used to modify or add to the HTML code for the breadcrumb path in the private zone. It accepts a certain number of data items in the `args`: the subject of its current identifier, if there is one, and possibly the identifier of the sector.

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

Example

The "polyhiérarchie" plugin, which enables a section or article to have multiple parents, uses this pipeline to list the various parents for the section or article currently displayed:

```
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

This pipeline is used to add some content to SPIP’s `exec/` pages. The new content is inserted after the content of the middle part of the page.

It is called as follows:

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

Examples

The plugin "Sélection d’articles" uses it to add a form to the sections page to offer a selection of 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;
}
```

The plugin "statistiques" adds a configuration form inside SPIP’s configuration pages.

```
function stats_affiche_milieu($flux){
    // displays the configuration ([de]activate the
statistics).
    if ($flux['args']['exec'] == 'config_fonctions') {
        $compteur = charger_fonction('compteur',
'configuration');
        $flux['data'] .= $compteur();
    }
    return $flux;
}
```

ajouter_boutons

This pipeline is used to add buttons to the private zone navigation menu. It is not really so useful since the creation of the <bouton> tag in the `plugin.xml` file (see Defining buttons (p.309)).

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

The "ajouter_boutons" pipeline accepts a parameter table of "button identifer / button description" couples (with a PHP class of Bouton(Button)). A button can declare a sub-menu in the "submenu" variable of the Bouton(Button) class. You must create an instance of the `Bouton` class to define this:

```
function plugin_ajouter_boutons($boutons_admin){
$boutons_admin['identifier'] =
    new Bouton('image/du_bouton.png', 'Button title', 'url');
$boutons_admin['identifier']->sousmenu['other_identifier'] =
    new Bouton('image/du_bouton.png', 'Button title', 'url');
return $boutons_admin;
}
```

The third `url` parameter of the `Bouton` class is optional. By default, it will be an "exec" page with the same name as the identifier provided (`ecrire/?exec=identifier`).

Example

The "Thelia" plugin, which makes it possible to interface SPIP with the Thélia software package, uses this pipeline to add a link to the Thélia catalogue to the "Édition" menu (with the "naviguer" identifier):

```
function spip_thelia_ajouter_boutons($boutons_admin) {
    // if you are admin
    if ($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 to the new system

To rewrite this example to the new system, two things would need to be separated: the button declaration, and the authorisation to view it or not (in this case, authorisation is only for administrators). The declaration is written in the `plugin.xml` file:

```
<bouton id="spip_thelia_catalogue" parent="naviguer">
    <icone>smg_pack/logo_thelia_petit.png</icone>
    <titre>title language string</titre>
</bouton>
```

The authorisation component is built with a special authorisation function (use the autoriser (p.139) pipeline to define this):

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

ajouter_onglets

This pipeline is used to add tabs to the `exec` pages in the private zone. It is not so nearly useful since the creation of the `<onglet>` tag in the `plugin.xml` file (see Defining page tabs (p.312)).

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

The "ajouter_onglets" pipeline accepts a table of couples of "tab identifier / tab description" (PHP class of Bouton), but also an identifier for the tab toolbar (in `args`).

```
// add a tab to SPIP's configuration page
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;
}
```

The third `url` parameter for the `Bouton` class is optional. By default it will be an "exec" page with the same name as the provided identifier (`ecrire/?exec=identifier`).

In the `exec` pages, a toolbar is called with two arguments: the identifier of the desired toolbar and the identifier of the active tab:

```
echo barre_onglets("tab toolbar identifier", "active tab
identifier");
echo barre_onglets("configuration", "contents");
```

Example

The "Agenda" plugin modifies the default tabs for SPIP’s calendar by using this 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 to the new system
To rewrite this example in the new system, 2 things need to be separated: the declaration of the button, and the authorisation to see it or not. The declaration is made in the `plugin.xml` file:

```
<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>
```

The authorisation is relocated into a special function (use the autoriser (p.139) pipeline to define it):

```
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 can send warning messages for various events that may be more or less considered as being urgent:

- A database crash
- A plugin crash
- A plugin activation error
- A notification that there is a message in the mailbox

This pipeline, called in ecrire/inc/commencer_page.php by the `alertes_auteur()` function, is used to populate the table containing such warnings.

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

It receives an array as a parameter.

- `data`: contains an array of the various warnings,
- `args` contains an array with:
    - `id_auteur` being the currently logged-in author,
    - `exec` is the name of the displayed page.

Example

Suppose that there is a plugin called "Watch out for llamas", which tells people that they are at risk of encountering a fearsome llama, then we could provide this as follows:

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

    // If there is a llama in front of this author
    if (tester_llama($flux['args']['id_auteur'])) {
        // We add a warning
        $alertes[] = "<strong>Watch out! There's a 
llama!</strong>";
    }

    // We return the table of warnings
    return $alertes;
}
```

A most fortuitous and beneficent plugin indeed!

autoriser

The "autoriser" pipeline is a special one. It is simply used to load the authorisation functions the first time that the `autoriser()` function is called. This pipeline neither accepts arguments nor returns output.

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

With this pipeline, a plugin can declare its own special authorisations, regrouped in a file named "PluginPrefix_autorisations.php" and declare them in the `plugin.xml` file as in this example:

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

In addition to authorisation functions, the file must contain the function called by all of the pipelines ("PluginPrefix_PipelineName()") but it has nothing to execute, e.g.:

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

Example

The "forum" plugin declares several new authorisations. Its `plugin.xml` file contains:

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

And the file which is called "forum_autoriser.php" contains:

```
// declare the pipeline function
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, $opt) {
    return autoriser('publierdans', 'rubrique',
_request('id_rubrique'));
}
// Moderate the forum?
// = modify the corresponding object (if there is a forum
for this object)
// = default rights else (full admin for full moderation
rights)
function autoriser_modererforum_dist($faire, $type, $id,
$qui, $opt) {
    return autoriser('modifier', $type, $id, $qui, $opt);
}
// Modify a forum ?  never !
function autoriser_forum_modifier_dist($faire, $type,
$id, $qui, $opt) {
    return false;
}
...
```

base_admin_repair

This pipeline is placed at the end of a repair process (for example to repair documents).

It has been created by the changeset [14262]

body_prive

This pipeline is used to modify the HTML body tag in the private zone, or to add content just after this tag. It is called by the `commencer_page()` function that is executed during the display of private zone pages.

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

boite_infos

This pipeline modifies the information block of the objects in SPIP's private zone. As an example, this is the block that contains the number of an article and the links used to change its status.

It accepts an associative array defined like this:

- `data`: what will be displayed on the page,
- `args`: another associative array of 3 keys:
    - `type`: the object type (article, rubrique...)
    - `id`: the object id (8, 12...)
    - `row`: array containing all the SQL fields of the object and their values.

Example

The plugin "Prévisualisation pour les articles en cours de rédaction" (previsu_redac) adds the button "Preview" when an article is still in the editing process (normally this link appears only when an article has been submitted for evaluation):

```
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;
}
```

calculer_rubriques

With this pipeline, plugins can change the status of a section (e.g. each section is published at its creation).

This pipeline can do everything but in order to modify the `status/dates` fields, it must modify the `statut_tmp/date_tmp` fields like this:

```
sql_updateq('spip_rubriques', array('date_tmp' =>
'0000-00-00 00:00:00', 'statut_tmp' => 'prive'));
```

Because SQL queries aren't transactional in SPIP, these temporary fields are necessary in order to be sure that the database won't be broken during the calculation process.

This pipeline is called here: http://trac.rezo.net/trac/spip/brow...

compter_contributions_auteur

This pipeline is used to insert content onto the author list page showing the volume of each author's contributions.

It is called as shown below from ecrire/inc/formater_auteur.php:

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

Example

The "Forum" plugin adds the number of messages written by an author:

```
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

This pipeline is used to supplement (or modify) SPIP's default configuration parameter values. It accepts a parameter consisting of a table of "name / value" pairs and returns the same as output.

This pipeline is called in 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' => '',
    //...
));
```

The `config()` function is used to supplement the parameters still missing from SPIP but which have a default value defined by the pipeline. It is specifically called from SPIP's native configuration forms.

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

Example

The "Compresseur" extension uses this pipeline to define the default options for the page compression system.

```
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

This pipeline declares the "auxiliary" tables, which are mainly used to create joins between principal tables.

It accepts the same arguments as the pipeline declarer_tables_principales (p.153).

Example

The plugin "SPIP Bisous" enables an author to send a *poke* to another author. It declares a table `spip_bisous` linking 2 members with the poke’s date using code as shown below. Note that the primary key is composed of 2 separate fields.

```
function
bisous_declarer_tables_auxiliaires($auxiliary_tables){
    $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_key = array(
        'PRIMARY KEY' => 'id_donneur, id_receveur'
    );

    $auxiliary_tables['spip_bisous'] = array(
        'field' => &$spip_bisous,
        'key' => &$spip_bisous_key
    );

    return $auxiliary_tables;
}
```

declarer_tables_interfaces

This pipeline is used to declare information relating to SQL tables or for certain fields in those tables. It makes it possible to supplement the information provided by ecrire/public/interfaces.php

The function accepts a parameter which is the array of declared elements, often called `$interface`, which must also be returned as output from the function. This array consists of the various elements, each of which are also arrays:

- `table_des_tables` declares the alias names of SQL tables,
- `exceptions_des_tables` assigns aliases to SQL columns for a given table,
- `table_titre` specifies the SQL column of an object used to define the title for certain types of URL naming conventions,
- `table_date` specifies an SQL data type column for a given SQL table which can be used for certain specific selection criteria (age, age_relatif, ...),
- `tables_jointures` defines the possible joins between SQL tables,
- `exceptions_des_jointures` creates aliases for SQL columns resulting from a join,
- `table_des_traitements` specifies filters to be systematically applied on SPIP tags.

table_des_tables

Declares alias names for SQL tables, relating to the declaration provided in either the principal or join tables.

In general, any plugin offering a new editorial object also declares an identical alias as the object name. This makes it possible to write loops like `<BOUCLEx(NAME)>`, in exactly the same way as `<BOUCLEx(spip_name)>` (which simply specifies the name of the SQL table).

```
// 'name_declare' = 'spip_rubriques', but without the 'spip_'
prefix
$interface['table_des_tables']['alias'] = 'name_declare';
// examples
$interface['table_des_tables']['articles'] = 'articles'; //
ARTICLE loops on spip_articles
$interface['table_des_tables']['billets'] = 'articles'; //
BILLET loops on spip_articles
```

exceptions_des_tables

Just as with declaration of aliases for SQL tables, it is also possible to declare aliases for SQL columns. These aliases can also force a join to another table.

```
// the tag #COLUMN_ALIAS or criteria {column_alias} applied
to the correct loop
$interface['exceptions_des_tables']['alias']['column_alias']
= 'column';
$interface['exceptions_des_tables']['alias']['column_alias']
= array('table', 'column');
// examples
$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');
// allows for the use of criteria like racine (root),
meme_parent (same 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

Specifies which field will be used to generate the titles for certain URL naming conventions (propre, arborescent...). The character string passed is an SQL selection declaration (SELECT), which must return 2 columns as output (or the SQL column alias(s)) : "title" and "lang". When the object has no corresponding "lang" field, then it must return `'' AS lang` instead.

```
$interface['table_titre']['alias']= "title_column AS titre,
lang_column AS lang";
// examples
$interface['table_titre']['mots']= "titre, '' AS lang";
$interface['table_titre']['breves']= 'titre , lang';
```

Whenever an object has declared its title, the URL generator can then create meaningful URL's automatically (depending on the URL naming convention chosen for the web site).

table_date

This information is used to declare certain SQL columns as date type fields. The SPIP compiler can then apply certain kinds of criteria to these fields, such as "age", "age_relatif", "jour_relatif"... Only one single date type field can be declared for any given table.

```
$interface['table_date']['alias'] = 'column_name';
// examples
$interface['table_date']['articles']='date';
$interface['table_date']['evenements'] = 'start_date';
```

tables_jointures

These declarations are used by the compiler to explicitly determine the possible joins whenever a loop on a table requests an unknown field (tag or criteria).

The compiler knows implicitly how to make certain joins (without declaring them) by looking for the column requested in the other SQL tables that it knows about. The compiler does not search through all tables, but only in those that have specific columns in common:

- same name as the primary key,
- same name as a column declared as a potential join in its `join` description in the principal or join tables.

In many cases, it is useful and preferable to explicitly declare to the compiler which joins that it can try to make when it is presented with an unknown field in a table. That is the explicit purpose of these kinds of declaration. The order of the declarations is sometimes important, since it will effect which join the compiler will find when it looks for the field in another table. Even if the field sought after would be found declared for the table anyway.

```
$interface['tables_jointures']['spip_nom'][] = 'other_table';
$interface['tables_jointures']['spip_nom']['column'] =
'other_table';
// examples
// {id_mot} for ARTICLES
$interface['tables_jointures']['spip_articles'][]=
'mots_articles';
$interface['tables_jointures']['spip_articles'][]= 'mots';
// event joins (for the plugin agenda) on keywords or
articles
$interface['tables_jointures']['spip_evenements'][]= 'mots';
// inserted before the articles join
$interface['tables_jointures']['spip_evenements'][] =
'articles';
$interface['tables_jointures']['spip_evenements'][] =
'mots_evenements';
// articles joins to events (evenements)
$interface['tables_jointures']['spip_articles'][] =
'evenements';
```

Most of the time, by also using the "exceptions_des_jointures" description explained below, it will be sufficient for a SPIP loop to know how to calculate the joins that it will need to display the various tags requested. If that is not always sufficient, don't forget that joins can also be specified in the loops and criteria themselves (cf. Forcing joins (p.82)).

exceptions_des_jointures

This definition is used to assign a column alias that creates a join with another table to retrieve another field, so long as the join is possible. It's a bit like the "exception_des_tables" which declare a join, but is not specific to a given table. We can then use this alias as a SPIP tag or as a loop criteria.

Note that when we use these joins only as loop criteria like `{titre_mots=xx}`, it is preferable to write this as `{mots.titre=xx}`, which is a more generic style and does not require a declaration.

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

One special scenario also exists: a third argument can be provided that contains the name of the function which will create the join. This is a rare circumstance, one use of which is employed by the "Forms & Tables" plugin

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

table_des_traitements

These descriptions are very useful; they make it possible to define standardised processes (filters) for certain SPIP tags. Using an asterisk (i.e. #TAG*) will deactivate any such processes.

In concrete terms, for each tag, or each tag/loop pair, the functions specified will be executed. %s will be replaced by the actual contents that the tag returns.

Two constants are available for the most common usages:

```
// typographical processing
define('_TRAITEMENT_TYPO', 'typo(%s, "TYPO", $connect)');
// SPIP shortcut processing ([->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;
// examples in 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;
```

An example which is often very useful is the automatic deletion of the numbers used as prefixes in section titles. This can be implemented using this method in the `config/mes_options.php` file (or by using this pipeline in a plugin, of course!) :

```
// simple version
$GLOBALS['table_des_traitements']['TITRE'][]=
'typo(supprimer_numero(%s), "TYPO", $connect)';
// complex version (do not overwrite the existing definition)
if (isset($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);
```

Example

Take the complex example of the Agenda plugin, which declares a table called `spip_evenements`(events), a linkage table called `spip_mots_evenenents` (keyword events) and a second linkage table called `spip_evenements_participants` (event participants).

An alias is defined to loop over the events. Explicit joins are declared, along with a date field and special processes. It uses nearly all of the features defined above!

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

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

    //-- Joins ------------------------------------------
----------
    $interface['tables_jointures']['spip_evenements'][]=
'mots'; // to be inserted before the join on 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)';

    // used for critiria such as 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

This pipeline creates a relationship between an object type and its corresponding SQL table. By default, an ’s’ is added to the end of the object type name (e.g. the ’article’ object maps to a table called ’articles’).

Pipeline call:

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

These relationships enable the functions `table_objet()` and `objet_type()` to work together:

```
// 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
```

Example

The "jeux" plugin uses:

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

declarer_tables_principales

This pipeline declares additional tables or fields. The SQL type of each field is specified, along with the primary keys and sometimes the secondary keys (used for joins between tables).

The tables in question are the "principal" ones because they mainly concern editorial content, whereas "auxiliary" tables (p.145) relate to links between those principal tables.

These declarations are used by SPIP to:

- manage the display of loops (even if it's optional because SPIP can get the SQL description of a table that hasn't been declared),
- create tables (or new fields) during the installation of SPIP or a plugin,
- backup and restore these tables with the default backup manager in SPIP's private area.

The function receives as arguments the list of the tables already declared and returns this same array, now supplemented. In this array, each table is declared with an associative array of 3 keys (the join key is actually optional):

```
$tables_principales['spip_name'] = array(
    'field' => array('champ'=>'SQL creation code'),
    'key' => array('type' => 'name(s) of the field(s)'),
    'join' => array('champ'=>'join field')   // Optional key
);
```

SPIP uses this pipeline in the last part of the declaration of the tables that will be used

Example

The "Agenda" plugin declares a table of events, "spip_evenements", with a number of fields. It declares the primary key (`id_evenement`), 3 indices (`date_debut`, `date_fin` and `id_article`), as well as two possible keys for use in joins: `id_evenement` and `id_article` (the order of these declared keys determines their priority when establishing joins).

It also declares an "agenda" field in the `spip_rubriques` table:

```
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

This pipeline is used to generate standard SPIP URLs for the specified objects, and to calculate the correspondence between a standard URL and its matching object. These URLs may take the form:

- `spip.php?objcetXX` (spip.php?article12)
- `?objectXX` (?article12)
- or the same with `.html` at the end.

With the `.htaccess` file as supplied with SPIP and activated, URLs may also be like:

- `objectXX` (article12)
- `objectXX.html` (article12.html)

The URL calculated whenever we use SPIP's URL calculation functions (the `#URL_` tag or the `generer_url_entite` function) depend on the URL options selected within the SPIP configuration pages.

This pipeline is called in ecrire/inc/urls.php with a list of predefined objects. It accepts input parameters and produces output of a table of the list of the objects that can be used in a URL:

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

The `#URL_nom` tag returns a URL for a given object type and specific object identifier (no need for declarations to do this). This pipeline is used to decode a standard URL and to identify the object type and object odentifier to which it applies. Once an object "name" has been declared , `?nameXX` in the URL will enable SPIP to calculate that the object type is "name"; that the "id_name" identifier is equal to "XX", and that SPIP should therefore try to load the `name.html` template for the identifier in question.

The use of this pipeline can be coupled with the declaration of "table_title" in the declarer_tables_interfaces (p.146) pipeline. This indicates which SQL column of the object should be relied on to create a meaningful URL.

Example

The "Grappes" plugin uses this pipeline making it possible to create URLs for the new object. #URL_GRAPPE creates a URL modifed for the object type. SPIP will then know which template to refer to when such a URL is requested: grappe.html.

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

The interface pipeline also declares the title field for meangingful URLs:

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

definir_session

Whenever a template requests to use #AUTORISER, #SESSION or any other tag which requires the creation of a different cache for each session, a special identifier is calculated with the session information known about the visitor by the spip_session function. This identifier is used to name the cache files. When no information is known about the visitor, the identifier returned is null.

The definir_session pipeline is used to complete the information used to create this identifier. It is also possible to compose unique caches relying on other parameters rather than data relating to the visitor.

The pipeline receives and returns a character string. It is called as in the file ecrire/inc/utils.php:

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

Remarks: the session data can be required very early on in SPIP's operations, so it is best to declare the the pipeline function for a plugin directly in the options file. The declaration in the `plugin.xml` file does not need to define the XML tag `<inclure>` in such circumstances:

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

Example

The "FaceBook Login" plugin defines a cache name which is also dependent on the Facebook authentication if that has been validated:

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

The "Forms & Tables" plugin also defines a specific cache when cookies linked to its forms are discovered:

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

We should note that the `#FORMS` dynamic tag for this plugin requests the creation of a cache per session by assigning `true` to the `session` option of the tag:

```
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

This pipeline is called just before executing the operation to delete the statistics from within the private zone on the `ecrire/?exec=admin_effacer` page. This is a trigger: a pipeline which only reports an event without passing any parameters. As such, this pipeline might be renamed as `trig_delete_statistiques` in the future.

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

It has not yet been used in any plugin available on the SPIP Zone. This pipeline should be used to delete the SQL statistics tables that might be added by any other plugins.

delete_tables

This pipeline is called just before executing the function that totally deletes the database tables from within the private zone via the `ecrire/?exec=admin_effacer` page. It is a trigger: a pipeline which simply takes note of an event, without any parameters being passed. As such this pipeline might be renamed as `trig_delete_tables` in the future.

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

There isn't any particularly interesting application of this pipeline within the plugins available on SPIP Zone. It might be possible to use it to execute processes on an external database when a SPIP site is reinitialised using the admin page, or also to send notifications of the purge action (and perhaps the current admin user's connection details) to certain nominated recipients as an audit.

editer_contenu_objet

This pipeline is called during the display of a back-end form for a SPIP object. It is used to change the HTML content of that form. This pipeline is called as a CVT form loading parameter (p.241) :

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

The pipeline passes:

- the type (`type`) , the object identifier (`id`) and the compilation context (the `contexte` table) using the `args` table
- the HTML code in the `data` key

Example

The "OpenID" plugin adds a data entry field into the author creation form:

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

formulaire_charger

The `formulaire_charger` pipeline is used to modify the table of values that passed from the `charger` function for a CVT form. It is therefore called when displaying a form from the ecrire/balise/formulaire_.php file.

It is passed a parameter of the form name as well as the parameters passed to the form in the `charger`, `verifier` and `traiter` functions. It returns the table of values to be loaded.

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

Example

The "noSpam" plugin uses this pipeline to add a token indicating a validity period for the forms nominated in a global variable:

```
$GLOBALS['formulaires_no_spam'][] = 'forum';
//
function nospam_formulaire_charger($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

This pipeline is called in ecrire/public/aiguiller.php after the processes (p.0) have been run for a CVT form. It is used to supplement the response table or to perform any additional processes.

It accepts the same arguments as the formulaire_charger (p.160) or formulaire_verifier (p.162) pipelines. It returns the table of data that are the results of processing (error message, success message, redirection, editable form refresh...).

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

Example

The "Licence" plugin, which offers the opportunity to assign a usage licence to articles, uses this pipeline to save the default licence value in the configuration details whenever a new article is created:

```
function licence_formulaire_traiter($flux){
    // if creating a new article, assign it the
configured default licence
    if ($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:

- the `lire_config()` PHP function belongs to the configuration plugin "CFG".
- in SPIP 2.1, it will be more relevant to use the pre_insertion (p.172) pipeline for this specific example.

formulaire_verifier

This pipeline is called from ecrire/public/aiguiller.php during the verification of data submitted from a CVT form. It is used to complete the array of errors returned by the verifier (p.242) function for the form in question.

It is passed the same argument parameters as the formulaire_charger (p.0) pipeline, those being the form name as well as the parameters passed in the the `charger`, `verifier` and `traiter` functions. It returns the array of errors as output.

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

Example

The "OpenID" plugin uses this pipeline to verify that the provided OpenID URL is valid when an author edits his details, and if not, it provides an error message tagged for the field in 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

The `header_prive` pipeline is used to add content into the HTML <head> section of pages in the private zone. It works like the insert_head (p.164) pipeline.

The pipeline accepts a parameter and returns as output the contents of the HEAD section:

```
function prefixPlugin_header_prive($flux){
    $flux .= "<!-- a comment for no reason at all! -->\n";
    return $flux;
}
```

Example

The "Notations" plugin uses this hook to add CSS declarations for both private and public pages (it also uses `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;
}
```

The "Open Layers" plugin enables the use of ’Open Street Map’ maps and uses this function to load the necessary JavaScript code:

```
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

The `insert_head` pipeline adds some content into the <head> section of an HTML page:

- wherever the `#INSERT_HEAD` tag has been used,
- otherwise just before `</head>` if the function `f_insert_head` is called in the affichage_final (p.126) pipeline - for example with this line in `mes_options.php` :

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

The pipeline accepts the contents to be added as arguments and returns the completed contents:

```
function prefixPlugin_insert_head($flux){
    $flux .= "<!-- A comment that does nothing ! -->\n";
    return $flux;
}
```

Example

Add in a jQuery function call, in this case, to display a toolbar for `textarea` tags in the Crayons forms (with the plugin "Porte Plume"):

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

The `onAjaxLoad` JavaScript function is used here to provide the given function as a parameter during the AJAX load of a page element.

insert_head_css

The `insert_head_css` pipeline is used by plugins to insert the CSS files that they need to operate correctly into the section of the SPIP template that includes the `#INSERT_HEAD_CSS` tag if there is one, and if not then at the start of the code included using the `#INSERT_HEAD` tag. This allows a template to indicate a specific location for additionally loaded CSS code.

It is called quite simply by using:

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

Example

The "Porte Plume" extension uses it in a simplified manner to add two CSS files, the second being a SPIP template file:

```
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

This pipeline makes it easy to add JavaScript code which will be inserted into every public and private page (which uses the tag #INSERT_HEAD (p.38)).

It receives and returns an array that contains the paths (these paths will be completed by the function find_in_path() (p.108)) of the files to be inserted:

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

Example

Add the script "jquery.cycle.js" to every page:

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

lister_tables_noerase

This pipeline is used to specify the SQL tables not to be emptied just before a restore.

It is called by the `lister_tables_noerase` function in the ecrire/base/dump.php file. It accepts as parameter and returns as output an array containing the list of database tables not to be purged:

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

lister_tables_noexport

This pipeline is used to declare SQL tables which will not be included in the SPIP back ups.

It is called from the `lister_tables_noexport` function in the ecrire/base/dump.php file. It accepts a parameter and returns as output an array containing the list of database tables not to be backed up:

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

By default, certain SPIP tables are already excluded, these being the tables used for statistics, searches and revisions.

Example

The "Géographie" plugin uses this pipeline to nominate not to export its SQL tables that contain the geographical data (these are very large):

```
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

This pipeline is used to specify the SQL tables not to be imported during the restore of an internal SPIP backup.

It is called by the `lister_tables_noimport` function in the ecrire/base/dump.php file. It accepts as parameter and returns as output an array containing the list of database tables not to be imported:

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

optimiser_base_disparus

Called from ecrire/genie/optimiser.php, this pipeline is used to supplement the cleaning operations for orphaned items, by deleting such items during standard periodic task scheduling.

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

As parameters, it receives the expected inter-operation delay (attente) as well as the corresponding expiry date. The "data" argument array stores the number of items deleted. The `optimiser_sansref()` function is used to manage the deletion of the records by providing 3 argument parameters:

- the table,
- the primary key,
- an SQL query result containing only an "id" column listing the identifiers targeted for deletion.

Example

To delete forums that belong to an obsoleted section, the "Forum" plug uses this pipeline as shown below:

```
function forum_optimiser_base_disparus($flux){
    $n = &$flux['data'];
    # forums linked to a non-existent id_rubrique
(section)
   $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

The `post_typo` pipeline is used to modify text after SPIP has applied its normal typographical processes, and therefore also after the pre_typo (p.174) pipeline. It is called by the `corriger_typo()` function in ecrire/inc/texte.php, a function which itself is called when using the `propre()` or `typo()` functions.

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

Example

The "Typo Guillemets" plugin replaces quotation marks " in a piece of entered text with the appropriate equivalent depending on the language code, such as using « and » for French texts. To do this, it analyses the text for typographical short-cuts that have been applied as shown below:

```
function typo_guillemets_post_typo($texte) {
    // ...
    switch ($GLOBALS['spip_lang']) {
        case 'fr':
            $guilles="&laquo;&nbsp;&nbsp;&raquo;";
//LRTEUIN
            break;
        // ...
    }
    // escape any " found in the tags;
    // note <!--extra--> is the character chr(1), and <!-
-extra--> represents the tag
    $texte = preg_replace(',<[^>]*"[^>]*(>|$),msSe',
"str_replace('\"','<!--extra-->', \"<!--extra-->\")",
$texte);
    // We correct any remaining quotes, which are by
definition not within tags
    // A quote is not processed if it follows a non-space
character, or
    // if it is followed by a word (letter, digit)
    $texte = preg_replace('/(^|\s)"\s?([^"]*?)\s?"(\W|$)/
S', '<!--extra-->'.$guilles.'', $texte);
    // and put back the quotes in any tags
    return str_replace("<!--extra-->", '"', $texte);
}
```

pre_boucle

The pipeline `pre_boucle` modifies the SQL queries that result from the interpretation of the loops of SPIP. It is called at each compilation phase, after the compiler has already taken into account the selection criteria (the `critere_NAME()` functions), and before the call to the `boucle_NAME()` functions.

It receives as argument a "Boucle" object that contains the data issued from the previous compilation steps for the current loop.

It is therefore possible to take actions based on the criteria that are passed to the loop, like modifying the selection parameters or the "where" condition for the loop's SQL query.

Example

The "mots techniques" plugin adds a technical field to the groups of keywords of SPIP.

When there is no `{technique}` criteria passed to the loop GROUPE_MOTS, the loop automatically filters its results, returning only those where the field `{technique}` is empty. This same feature could also be implemented by creating a function called boucle_GROUPES_MOTS().

```
function mots_techniques_pre_boucle($loop){
    if ($loop->type_requete == 'groupes_mots') {
        $table_name = $loop->id_table;
        $technical_kw = $table_name .'.technique';
        // Select only the loop without the "technical"
keyword
        if (!isset($loop-
>modificateur['criteres']['technique']) &&
            !isset($loop->modificateur['tout'])) {
                $loop->where[]= array("'='",
"'$technical_kw'", "'\"\"'");
        }
    }
    return $loop;
}
```

The array `$loop->where[]` contains arrays with 3 entries: successively being the operator, the field and the value. Here, we add to the query the string `{$table_name}.technique=''` with:

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

pre_insertion

This pipeline is used to add default content when a new editorial element is being inserted into the database.

When an editorial item is requested to be saved, it has not yet been allocated a unique identifier (implying it is a new item), so an identifier is automatically created for that item using the `insert_xx` functions, where xx is the name of the intended object. This insertion pipeline has the simple goal of returning an identifier and saving the item's default values. The pipeline is called from these `insert_xx` functions.

Once the identifier has been established, the normal modification tasks are performed using the `xx_set` and `modifier_contenu` functions which call the `pre_edition` and `post_edition` pipelines. Those tasks are the ones that will save the data entered by the user, and which will therefore do so even for new items.

This pipeline passes the table name and an array of fields and default values to be inserted:

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

Example

The "Forum" plugin adds the forum status value for an article when it is created using the code below:

```
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

The "pre_liens" pipeline is used to process typographical shortcuts relating to links of the form`[title->url]`. It is called by the `expanser_liens()` (expand_link) function, which is itself called by the `propre()` function.

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

SPIP itself makes use of this entry point to execute processes that include 3 functions in the definition of the pipeline in the ecrire/inc_version.php file, defined within ecrire/inc/lien.php :

- `traiter_raccourci_liens` automatically generates links for a piece of text that looks like a URL,
- `traiter_raccourci_glossaire` generates `[?title]` shortcuts pointing to a glossary (p.316).
- `traiter_raccourci_ancre` takes care of `[<-anchor name]` shortcuts that create a named anchor point

Example

The "documentation" plugin (which manages this same documentation), uses this pipeline to automatically add a `title` attribute on internal link shortcuts of the form `[->art30]`, transforming them into `[|art30->art30]` (this workaround serves to display the page number relating to the link when exporting the contents of the documentation in PDF format)

```
function documentation_pre_liens($texte){
    // only for the public site
    if (test_espace_prive()) return $texte;
    $regs = $match = array();
    // for each link
    if (preg_match_all(_RACCOURCI_LIEN, $texte, $regs,
PREG_SET_ORDER)) {
        foreach ($regs as $reg) {
            // if the shortcut is of the form "art40"
            if (preg_match(_RACCOURCI_URL, $reg[4],
$match)) {
                $title = '|' . $match[1] . $match[2];
                // if this title doesn't already exist
                if (false === strpos($reg[0], $title)) {
                    $lien = substr_replace($reg[0],
$title, strpos($reg[0], '->'), 0);
                    $texte = str_replace($reg[0], $lien,
$texte);
                }
            }
        }
    }
    return $texte;
}
```

pre_typo

The `pre_typo` pipeline is used to modify the text before the typographical processes envisaged by SPIP are executed. It is called by the `corriger_typo()` function in ecrire/inc/texte.php, a function which is itself called when using the `propre()` or `typo()` functions.

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

The modifications proposed must only apply processes to the elements that can be displayed on a single (*inline*) line. For processes that modify or create blocks or paragraphs, you must use the `pre_propre` pipeline.

Example

The "Enluminures Typographiques" plugin automatically modifies how some character strings are displayed, e.g. transforming "(c)" into "©":

```
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_jointures

This pipeline, used in ecrire/inc/rechercher.php, is used to declare the searches that should be executed on other tables rather than the table explicitly referenced in a loop. For example, a search for an author name for an ARTICLES loop returns the articles that this author has written (by searching in the `spip_auteurs_articles` table).

This pipeline receives an array of tables containing an array of table, field, weighting triplets (like the pipeline "rechercher_liste_des_champs" (p.122)).

Example

Some modifications for the `spip_articles` table:

```
function
pluginPrefix_rechercher_liste_des_jointures($tables){
    // search in the BIO field of authors when we search
in the articles
    $tables['article']['auteur']['bio'] = 2;
    // search in the text of the keywords
    $tables['article']['mot']['texte'] = 2;
    // do not search in the documents
    unset($tables['article']['document']);
    return $tables;
}
```

In SPIP, this pipeline is used to search for elements using their linked keywords or authors

recuperer_fond

The "recuperer_fond" pipeline is used to add to or modify the compilation results of a given template file. As input, it accepts the name of the selected "fond", or model template, and the compilation context within the `args` table, as well as the table describing the results in the `data` table.

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

Very often, only the `texte` key in the `data` table will be modified. Please refer to the recuperer_fond() (p.110) article for a full description of this table.

Example

The "fblogin" plugin is used to identify visitors with their Facebook credentials. It adds a button to SPIP's normal identification form. The "social_login_links" pipeline (in the same plugin) returns the HTML code for a link pointing to the Facebook identification page.

```
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

This is used to add contents into the "Proposed for publication" panel displayed for sections. This panel will only be displayed when there is at least one element (article, site, news item...) in that section that has been proposed for publication.

It is called from ecrire/exec/naviguer.php:

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

Example

The "Forum" plugin uses this pipeline to add a phrase encouraging comments for the articles proposed for 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

This pipeline modifies the way in which SPIP searches for the template to use to compute a page - and for example, to change it for a specific section.

You can use it like this :

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

It receives some arguments found in the environment context and returns the name of the template that will be used by the compilation.

If the url is `spip.php?article18`, the arguments will be :

- id_rubrique = 4 (if the article is in section number 4)
- ext = ’html’ (the default extension for templates)
- fond = ’article’ (name of the template initially used)
- lang = ’fr’

- connect = ’’ (SQL connection name).

Example :

The plugin "Spip-Clear" uses this pipeline to call some specific templates for the different branches of the blog:

```
// defines the template to use for a section of Spip-Clear
function spipclear_styliser($flux){
    // article or section ?
    if (($fond = $flux['args']['fond'])
    AND in_array($fond, array('article','rubrique'))) {

        $ext = $flux['args']['ext'];
        // [...]
        if ($section_id = $flux['args']['id_rubrique']) {
            // calculates the branch
            $branch_id = sql_getfetsel('id_secteur',
'spip_rubriques', 'id_rubrique=' . intval($section_id));
            // comparison of the branch with the config of
Spip-Clear
            if (in_array($branch_id, lire_config('spipclear/
secteurs', 1))) {
                // if the template $fond_spipclear exists
                if ($template =
test_squelette_spipclear($fond, $ext)) {
                    $flux['data'] = $template;
                }
            }
        }
    }
    return $flux;
}
// returns a template $fond_spipclear.$ext when it exists
function test_squelette_spipclear($fond, $ext) {
    if ($template = find_in_path($fond."_spipclear.$ext")) {
        return substr($template, 0, -strlen(".$ext"));
    }
    return false;
}
```

taches_generales_cron

This pipeline is used to declare functions that will be periodically executed by SPIP. It is called in the ecrire/inc/genie.php file by the `taches_generales` function, accepting a parameter and returning output of a keyed array, using function names as the key and the time between execution runs as the value.

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

Please read the section on the Wizard (p.227) for further information.

Example

Any plugin whatsoever could declare a cleaning function to be run every week:

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

This function is contained in the `genie/nettoyer_cartes_postales.php` file. It deletes all the files in a given directory that are older than 30 days, by using the `purger_repertoire` function:

```
function genie_nettoyer_cartes_postales_dist($t){
    // Purge postcards that are older than 30 days
    include_spip('inc/invalideur');
    purger_repertoire(_DIR_IMG . 'cartes_postales/',
array(
        'atime' => (time() - (30 * 24 * 3600)),
    ));
    return 1;
}
```

trig_supprimer_objets_lies

This pipeline is a trigger (returns no output) which is called when certain objects are deleted. It makes it possible to delete data stored in linkage tables at the same time as an object is deleted. It is passed an array of the various deletions to be made (containing the deleted object type and identifier).

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

This pipeline is called for deletion of a keyword and for a message.

Example

The "Forum" plugin uses this pipeline to delete links with forum messages that are associated with a deleted keyword or message (from the mailbox):

```
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;
}
```

... and the rest of them

There are a handful of pipelines that have not yet been documented. They are listed here for information purposes only:

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

Tags

The balise directory stores the declarations of dynamic tags and SPIP's generic tags.

Dynamic tags

Dynamic tag are tags which are recalculated **every time** that the page is displayed, as opposed to static tags which are calculated only when the page is recalculated, either manually by a site administrator or automatically because the page cache has exceeded its expiry date/time.

These dynamic tags therefore store in the generated cache files a section of PHP which will be executed when the page is displayed. In principle, they are essentially used for displaying web forms.

A dynamic tag file may contain up to 3 essential functions: `balise_NAME_dist()`, `balise_NAME_stat()`, `balise_NAME_dyn()`.

The balise_NAME_dist function

The first function for a dynamic tag is the same function used for static tags, that is, a function with that tag's actual name: `balise_NAME_dist()`.

This function, instead of inserting a static code, will call a function generating a dynamic code: `calculer_balise_dynamique()`.

Generally speaking, the contents of the function contine with calling the dynamic calculation, as with this following tag example for `#LOGIN_PRIVE` :

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

The tag function is passed the `$p` variable containing the data originating from the analysis of the template in question (arguments, filters, to which loop it belongs, etc.).

The `calculer_balise_dynamique` function accepts 3 or 4 arguments:

- the `$p` description
- the name of the dynamic tag to execute (normally the same name as the tag!)
- an array of arguments to be retrieved from the page context. At this point, the dynamic tag requests the retrieval of a `url` parameter originating from the context (the closest loop or the template compilation environment). If there is no parameter to be retrieved from the context, then it must be passed an empty `array()`.
- the optional 4th argument is used to pass an array of elements that will be passed to the following function (`balise_NOM_stat`), thereby completing the `$context_compil` array. This then allows the calculation of the elements in the `balise_NOM_dist()` function and to pass them on.

The balise_NAME_stat() function

If it exists, the `balise_NAME_stat()` function will make it possible to calculate the arguments to be passed to the following (`_dyn()`) function. In its absence, only the arguments specified in the `calculer_balise_dynamique()` function are passed (in the order of the array). The `stat` function will make it possible to additionally pass the parameters originating from arguments or filters passed to the tag.

The function is passed 2 arguments: `$args` and `$context_compil`.

- `$args` contains the arguments required by the `calculer_balise_dynamique()` function, in addition to the arguments passed to the tag.
- `$context_compil` is an array of data about the completed compilation, containing 5 entries (template name, compiled file name, the name of the loop where the tag is used, line number, language), possibly followed by optional array elements supplied by the principal function for dynamic tags (the 4th argument of the `balise_dynamique` function).

Example

Referring again to the #LOGIN_PUBLIC example: it works with either 1 or 2 arguments: the first is the redirection URL after being connected, the second is the default user name for the person to be connected. Both of these are optional.

We can therefore pass a redirection argument to the tag: #LOGIN_PUBLIC{#SELF} or #LOGIN_PUBLIC{#URL_ARTICLE{8}}, but in the absence of an argument, we would like that the redirection be made to an environment URL parameter if there is one. Once having requested to retrieve this argument, it is found in $args[0]. with $args[1] storing the contents of the first argument passed to the tag (it adds itself into the $args array after the list of arguments automatically retrieved). This ends up with:

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

If $args[1] exists, it is passed, otherwise $args[0]. In the same manner, if $args[2] exists, it is also passed, otherwise ''.

The _dyn() function will be passed these 2 arguments:

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

The balise_NAME_dyn() function

This function is used to execute the processes to be performed when a form has been submitted. The function may return a character string (which will be displayed on the requesting page) or a parameter array which indicates the name of the template to retrieve and the compilation context.

The processes

This article will not address these operations for 2 reasons:

- the original author of this article (and the translator) does not completely understand how it works,
- it is not very useful since SPIP includes a simpler mechanism called "CVT forms" (Charger, Vérifier, Traiter), in English: (Load, Verify, Process) which also relies on this function but in a transparent manner.

The display

Whatever the function returns is then displayed on the page. An array indicates a template to be called. It generally looks like this:

```
return array("template_address",
    3600, // cache duration
    array( // context
        'id_article' => $id_article,
    )
);
```

Generic tags

Another clever SPIP mechanism is the provisioning of tags that might be termed as generic. In fact, it is possible to use a single tag declaration for a whole group of tags prefixed with an identical name.

As such, a tag named `#PREFIX_NAME` can use a file called `balise/prefix_.php` and delcare a function `balise_PREFIX__dist()` which will then be used if there is no `balise_PREFIX_NAME_dist($p)` function present.

The generic function, which accepts tag attributes in the `$p` variable, can use `$p->name_field` to obtain the name of the requested tag (in this case "PREFIX_NAME"). By analysing this name, it can then execute the appropriate actions.

Example

This example is used by the generic tags `#FORM_NAME`, which are also dynamic fields (in the file ecrire/balise/formulaire_.php).

```
function balise_FORM__dist($p) {
    preg_match(",^FORM_(.*)?$,", $p->name_field, $regs);
    $form = $regs[1];
    return
calculer_balise_dynamique($p,"FORM_$form",array());
}
```

Retrieving the object and id_object

This article will show how to retrieve the type (object) and identifier of a loop, so that they can be used in the calculations of a tag.

Static tags

With the parameters for the tag $p, it is very simple to retrieve both `object` and `id_object` :

```
function balise_DEMO($p){
    // take the name of the object's primary key to calculate
its value
    $_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";
}
```

Note that there are two functions here. The first uses the description of the tag to retrieve the name of its parent loop and the name of the primary key, and requests to retrieve the value of the primary key using the `champ_sql()` function. Note: what is retrieved in the `$id_object` variable is a code which must be evaluated using PHP (which is no longer a numeric value).

Once these parameters have been retrieved, we then add a PHP code to be evaluated in the code generated by the template compilation (this code will be cached). This is what is added into `$p->code`. That code will then next be evaluated during the creation of the called page cache.

The `calculer_balise_DEMO()` function is then passed the two desired arguments and returns a text which displays them on the page.

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

This template then enables the result to be seen, the `#DEMO` tag is passed the various data depending on the context in which it is found:

```
Object : article, id_object : 128
Object : article, id_object : 7
----
Object : rubrique, id_object : 1
Object : rubrique, id_object : 2
```

Dynamic tags

For a dynamic tag, its operation even prevents the simple retrieval of the type and identifier of the loop in which it has been written.

Even so, when it is needed, for example in creation of CVT forms which modify their processes depending on the type of loop, it is necessary to pass the object type and current loop identifier to the `_dyn()` function (and consequently to CVT's load, verify and process functions).

The call `tocalculer_balise_dynamique()` makes it possible to retrieve the compilation context elements. If we ask to retrieve 'id_article', we will certainly get one from within an `ARTICLES` loop, but not if we are in a `RUBRIQUES` loop. To be more specific, when we request an 'id_article' value, SPIP acts as if it is retrieving the result from an `#ID_ARTICLE` tag, so it then looks for the value in the closest loop, otherwise it looks in the context, and it also depends on the tags which have been specifically declared.

We could ask to calculate `id_object` quite easily, but `object` will require passing a tag returning the `object` value. This tag does not exist by default within SPIP 2.x, so it must be created with a (`DEMODYN_OBJET`), which gives us:

```
function balise_DEMODYN($p){
    // primary key
    $_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";
}
```

Creating pages in the private zone

The pages in the private zone can be supplemented by creating new files to alter them.

There are two different ways to install such pages:

- In the **exec** directory, where they can be written using PHP.
- In the **prive/exec** directory, where they can be written using SPIP template code.

The contents of a (template) exec file

A call from within the private zone of a `?exec=name` page automatically loads a template located in `prive/exec/name.html`.

In most cases, it is recommended to use this method rather than a PHP file. The objective being that SPIP's private zone itself also be written as a template, and therefore be easier to customise. This then makes it possible to use loops, includes, tags, and authorisations just like any other regular SPIP template.

Example of an empty private page template:

```
<!--#hierarchie-->
<ul id="chemin">
    <li>A list of pages constituting a breadcrumb path</li>
</ul>
<!--/#hierarchie-->

<h1>A private page directly coded in a template file</h1>
<p>Some page content</p>
<!--#navigation-->
<div class='cadre-info'>
Some information in a navigation column.
</div>
<!--/#navigation-->
<!--#extra-->
Some extra content in the extra column.
<!--/#extra-->
```

The `<!--#hierarchie-->`, `<!--#navigation-->` and `<!--#extra-->` frame tags serve to separate the page's major sections. SPIP's private zone will automatically relocate each of these sections into the appropriate HTML tags.

If the template only returns an empty result, then SPIP will automatically generate an authorisation error.

From a technical point of view, these templates are processed by the ecrire/exec/fond.php file. The following pipelines are automatically added: affiche_gauche (p.131), affiche_droite (p.130) et affiche_milieu (p.133) by passing the `exec` parameter name as a parameter:

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

In addition, the private page title is calculated by extracting the contents of the first HTML`<h1>` (or `<hn>`) tag that is found.

Example

The "Formidable" plugin uses SPIP templates to construct its pages for the private zone. To display responses in a form, it uses the following template code:

```
<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>
```

Notes:

- All of this is included within a loop that checks for the existence of the form: if the form does not exist, the template then returns nothing and provides an error message instead.
- In the same manner, it is surrounded with an #AUTORISER (p.198) test to check that the current person has the rights to see the responses. In this case we use the `CONDITION` loop from the "Bonux" plugin in order to be able to continue to read SPIP loops that lie inside the condition.
- The `<!--#hierarchie-->` code section displays a suitable path from amongst the private pages of the plugin.

The contents of a (PHP) exec file

In the absence of a `prive/exec/name.html` SPIP template file, a call from the private zone to a `?exec=name` page loads a `exec_name_dist()` function in a `exec/name.php` code file.

Such functions are mostly broken down as follows: the call to the start of the page, the declaration of a left column, a right column and a page centre. There are some pipelines that exist so that plugins will be able to add data to these page blocks.

Example of an empty "name" page

```
<?php
if (!defined("_ECRIRE_INC_VERSION")) return;
include_spip('inc/presentation');
function exec_nom_dist(){
    // if not authorised: error message
    if (!autoriser('voir', 'nom')) {
        include_spip('inc/minipres');
        echo minipres();
        exit;
    }
    // initialisation pipeline
    pipeline('exec_init',
array('args'=>array('exec'=>'nom'),'data'=>''));
    // headers
    $commencer_page = charger_fonction('commencer_page',
'inc');
    // titre, partie, sous_partie (pour le menu)
    echo $commencer_page(_T('plugin:titre_nom'), "editer",
"editer");

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

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

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

    // centre
    echo debut_droite('', true);
    // contents
    // ...
    echo "display whatever you want to here!";
    // ...
    // end of contents
    echo pipeline('affiche_milieu',
array('args'=>array('exec'=>'nom'),'data'=>''));
    echo fin_gauche(), fin_page();
}
?>
```

The information panel

To add a page description, or a description of the object/id_object currently being shown, a type of insert panel has been envisaged: "boite_infos" (info_box)

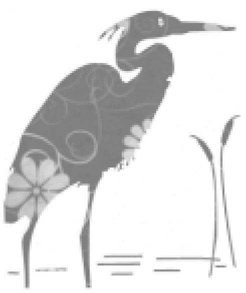
It is often used in a way to add a function into the left column:

```
// left column
echo debut_gauche('', true);
echo cadre_nom_infos();
echo pipeline('affiche_gauche',
array('args'=>array('exec'=>'nom'),'data'=>''));
```

This function calls the pipeline and returns its contents in a panel:

```
// display the page information
function cadre_champs_extras_infos() {
    $boite = pipeline ('boite_infos', array('data' => '',
        'args' => array(
            'type'=>'nom',
            // possibly the object's ID and the SQL line
            // $row = sql_fetsel('*', 'spip_nom',
'id_nom='.sql_quote($id_nom));
            'id' => $id_nom,
            'row' => $row,
        )
    ));
    if ($boite)
        return debut_boite_info(true) . $boite .
fin_boite_info(true);
}
```

The pipeline automatically loads a template (with the context supplied by the `args` array) of the same name to the "type" parameter in the prive/infos/ directory i.e. `prive/infos/nom.html`. It must then be created with the desired content.



Functionalities

This chapter explains some of SPIP's functionalities in further detail: authorisations, actions, authentication, the cache, the compiler...

Authorisations

Two essential elements make it possible to manage access to SPIP's actions and pages: the authorisations with the `fonction autoriser()` function, and actions secured by author with the `fonction securiser_action()` function.

The "autoriser" library

SPIP has an extendable `autoriser()` function enabling the verification of authorisations. This function accepts 5 arguments. Only the first is necessary, and the others are all optional.

```
autoriser($faire, $type, $id, $qui, $opt);
```

The function returns `true` or `false` depending on the authorisation requested and the user(editor) who is connected (or the requested user passed as an explicit parameter). Here are what the different arguments are used for:

- `$faire` corresponds to the action requested. For example "modifier" (modify) or "voir" (view),
- `$type` is generally used to define the object type, for example "auteur" (author) or "article",
- `$id` is used to provide the identifier of the requested object, for example "8" as an article number,
- `$qui` is used to enquire or assign authorisation for a specific author. When not provided, it assumes the currently connected author. The argument is typically an `id_auteur` number,
- `$opt` is an array of options, usually empty. When an authorisation requires additional arguments to be passed, they are entered in this array.

Example

```
if (autoriser('modifier','article',$id_article)) {
    // ... actions
}
```

The #AUTORISER tag

The `#AUTORISER` tag is used to request authorisations within a template. The existence of this tag, as with the existence of the `#SESSION` tag, creates a template cache for each identified visitor and a single cache for all unidentified visitors.

This tag accepts the same arguments as the `autoriser()` function.

Example

```
[(#AUTORISER{modifier,article,#ID_ARTICLE})
        ... actions
]
```

Processes in the autoriser() function

SPIP's default authorisations are made using the ecrire/inc/autoriser.php file.

When SPIP is requested for an `autoriser($faire, $type)` type authorisation, it goes to look for a function to handle this requested authorisation. It looks for the named function in the following order:

- `autoriser_$type_$faire`,
- `autoriser_$type`,
- `autoriser_$faire`,
- `autoriser_defaut`,
- and then the same list with the `_dist` suffix attached.

Example

```
autoriser('modifier','article',$id_article);
```

Will return the first function found and execute it. This being:

```
function autoriser_article_modifier_dist($faire, $type,
$id, $qui, $opt){
```

```
...
}
```

The function is passed the same parameters as the `autoriser()` function. Inside it, `$qui` is passed the current author if an author was not passed as an argument in the call to `autoriser()`.

Creating or overloading the authorisations

To create an authorisation, you only need to create the supporting functions.

```
function autoriser_documentation_troller_dist($faire, $type,
$id, $qui, $opt) {
    return false; // no trolls permitted! and no exceptions!
}
```

Declaring this function makes it possible to use the `autoriser('troller','documentation')` function or the `#AUTORISER{troller, documentation}` tag.

New functions, but not everywhere!

The `autoriser()` function, when first called, loads a pipeline with the same name. This call to the "autoriser" pipeline (p.139) is used to load the authorisation files for a template directory or a plugin.

Example

In a template: In the `config/mes_options.php` file, we add the call to a function for our authorisations:

```
<?php
$GLOBALS['spip_pipeline']['autoriser'] .=
"|my_authorisations";
function my_authorisations(){
    include_spip('inc/my_authorisations');
}
?>
```

So then when the `autoriser` pipeline is called, it loads the `inc/my_authorisations.php` file. We can then create this directory and file, which contains the intended authorisation functions in its `squelettes/` directory.

In a plugin: For a plugin, it's almost exactly the same: you have to declare the use of the pipeline inside your `plugin.xml` :

```
<pipeline>
    <nom>autoriser</nom>
    <inclure>inc/prefixePlugin_autoriser.php</inclure>
</pipeline>
```

And create the file in question and absolutely make sure to add in the `pluginprefix_autoriser()` function into the file that the pipeline calls.

```
<?php
if (!defined("_ECRIRE_INC_VERSION")) return;
// function for the pipeline, nothing to do
function pluginprefix_autoriser(){}
// declarations of authorisations
function autoriser_documentation_troller_dist($faire,
$type, $id, $qui, $opt) {
    return false; // no trolls permitted! and no
exceptions!
}
?>
```

Secured actions

Secured actions provide a method of ensuring that the requested action indeed originates from the author who clicked or validated a form.

The `autoriser()` function does not provide this functionality. For example, it can verify what type of author (administrator, editor) has the right to perform which actions. But it can not verify which action has been effectively requested by which individual.

This is where secured actions are applied. What they do in fact, is make it possible to create URLs for links or forms which pass a special key. This key is generated based on several data: a random number generated on each connection by an author and stored alongside the author's personal data, the author identifier, the name of the action and arguments of that action if there are any.

Using this passed key, when the author clicks on the link or the form, the action being called can confirm that it is actually the currently connected author who has requested the action to be performed (and not some malicious individual or robot executing an HTML query with stolen credentials!).

How secured actions work

Using secured actions is a 2-step process. You must first generate a link with the security key, and then later verify that key when the user clicks on the action that will execute a file function in the `action/` directory.

The securiser_action() function

This `securiser_action` function, stored in the ecrire/inc/securiser_action.php file, creates or verifies an action. During creation, depending on the `$mode` argument, it will create a URL, a form or simply return an array with the requested parameters and the generated key. During verification, it compares the elements submitted with a GET (URL) or POST (form) and kills the script with an error message and `exits` if the key does not match the current author.

Generating a key

To generate a key, you need to call the function with the right parameters:

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

These four parameters are the main ones used:

- `$action` is the name of the action file and the corresponding action(`action/name.php` and the associated function `action_name_dist()`)

- `$arg` is a passed argument, for example `supprimer/article/3` which will be used, among other things, to generate the security key.
- `$redirect` is a URL for redirection after the action has been performed.
- `$mode` indicates what should be returned:
    - `false`: a URL
    - `-1`: an array of parameters
    - a content text: a form to be submitted (the content is then added into the form)

Inside an action, verifying and retrieving the argument

Within an action function (`action_name_dist()`), we verify the security key by calling the function without an argument. It returns the argument (otherwise displays an error and kills the script):

```
$securiser_action =
charger_fonction('securiser_action','inc');
$arg = $securiser_action();
// from here on, we know that the author is the right person!
```

Secured actions' predefined functions

Secured actions are rarely directly generated by calling the `securiser_action()` function, but more frequently by calling a function which itself then calls the security function.

The ecrire/inc/actions.php file contains a large number of these functions.

generer_action_auteur()

In particular, the `generer_action_auteur()` function directly calls the `securiser_action` function, passing a secured URL by default.

redirige_action_auteur()

This function takes two parameters instead of the 3rd redirection argument: the name of an exec file, and the arguments to be passed. SPIP then creates the redirection URL automatically.

redirige_action_post()

Same as the previous function except that it generates a POST form by default.

Example

Generate a link to change the display preferences in the private area:

```
$url = generer_action_auteur('preferer',"display:1",
$self);
```

Run an action when editing a news item, then redirect to the news item view.

```
$href =
redirige_action_auteur('editer_breve',$id_breve,'breves_voir',
"id_breve=$id_breve");
```

Post a form, then redirect to the "admin_plugin" page. `$corps` contains the contents of the form to activate a plugin.

```
echo redirige_action_post('activer_plugins','activer',
'admin_plugin','', $corps);
```

Action URLs in a template

The `#URL_ACTION_AUTEUR` tag is used to generate secured action URLs from inside a template file.

```
#URL_ACTION_AUTEUR{action,argument,redirection}
```

Example

Deleting the forum comment requested if the author actually has the (`autoriser('configurer')` rights is certainly vague, but it is applied in the private area in ecrire/exec/forum_admin.php]) !

```
[(#AUTORISER{configurer})
<a href="#URL_ACTION_AUTEUR{instituer_forum,#ID_FORUM-
off,#URL_ARTICLE}"><:supprimer:></a>
]
```

Actions and processes

The `ecrire/action/` directory is intended for handling the actions afffecting the contents of the database . These actions are therefore mostly secured.

The contents of an action file

An action file provides at least one function matching its own filename. A file called `action/laugh.php` should therefore declare a function called `action_laugh_dist()`.

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

Operation of the function

In general, the main function is divided into 2 sections: verification of authorisations, then execution of the requested process.

The verifications

The right author

Most SPIP actions only verify that the current author is indeed the same as the one who clicked for the action. This is done with:

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

The security function kills the script if the current author is not the one requesting the action, otherwise it will return the requested argument (in this case through `$arg`).

The right argument

Then, generally speaking, the `$arg` variable received is verified to see if it is conformant with what was expected. It often takes the form "id_object", sometimes "object/id_object" or more complex ones like date elements:

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

And authorisation

Some actions also verify that the author is actually approved to execute that action (but in general, this authorisation has already been confirmed before: the link that fires the action will not normally be visible if the author does not have the appropriate rights). For example, checking to see if the current author has the right to moderate the forum for the given article:

```
if (!autoriser('modererforum', 'article', $id_article))
    return;
// which could also be written with a debug-type message:
if (!autoriser('modererforum', 'article', $id_article)) {
    include_spip('inc/minipres');
    minipres('Moderation',"You do not have the rights to
manage moderations on the forum for this article");
    exit;
}
```

The processes

When all the verifications have been made, the processes are then executed. Very often, these processes call functions that exist in the same file, or functions in a library in the `inc/` directory. Sometimes the action will be to simply execute the file's own main function.

Example of assigning moderation rights to an article

```
// Modifier le reglage des forums publics de l'article x
// Modify the moderation rights for the public form on
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 - processes
    $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'");
}
```

The processes executed modify the `spip_articles` table in the database to assign a new management status for forum management. When a forum is requested on subscription, which means you must be logged in to post, we must absolutely be sure that the site actually accepts visitor registrations, which is checked by calling `ecrire_meta('accepter_visiteurs', 'oui');`.

And finally, a call to invalidate the cached files is executed by calling the `suivre_invalideur()` function. All of the cache will be recreated (note that from SPIP 2.0 onwards, which was not the case previously, only the relevant section of the cache was invalidated).

Automatic redirections

At the end of an action, after the return of the function, SPIP redirects the page to a redirection URL passed in the `redirect` variable. The functions to generate the links to the secured actions, like `generer_action_auteur()`, have a parameter to receive this redirection link.

Forcing a redirection

Some actions, however, can force a different redirection, or define a default redirection. To do this, you must call the `redirige_par_entete()` function, which enables redirecting the browser to a different page.

Example:

Simply redirect to the redirection URL requested:

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

editer_objet actions

Actions which write data have a special peculiarity. Called by forms that write data for SPIP objects (in the `prive/formulaires/` directory) from the ecrire/inc/editer.php file, they do not receive redirection instructions and must return, in such cases, a data pair of "identifier", "error". The (CVT) form process itself manages the subsequent redirection.

For this reason, the `action/editer_xx.php` files, where xx is the object type (in the singular), may return an array:

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

Authentications

The auth directory contains the various scripts used to generate the user connections. Added to the API available in the file ecrire/inc/auth.php, the whole set can define new ways of authentication and user creation. SPIP provides two ways of authentication:

- SPIP for an ordinary connection
- LDAP to connect users authenticated using such a directory

The contents of an auth file

The various authentication checks are called during login through the prive/formulaires/login.php file. The first, which validates an authentication, makes it possible to accommodate someone who is in the process of identifying themselves.

The list of the various authentications is defined by a global variable: `$GLOBALS['liste_des_authentifications']`.

Nonetheless, the authentication processes are relatively complex requiring several safety checks. The user login and password are passed to the verification functions (encrypted with sha56 paired with a random number - or in the clear in the worst of cases when it is not possible to store cookies).

The primary identification function

A `auth/nom.php` file must have a `auth_nom_dist()` function. This function returns a table describing the author if that author is authenticated.

```
if (!defined("_ECRIRE_INC_VERSION")) return;
// Authenticates and if ok, returns the array for the user's
SQL row
// If a security risk affects the installation, return False
function auth_spip_dist ($login, $pass, $md5pass="",
$md5next="") {
...
}
```

The compiler

This section explains some of the details on how a template is compiled.

The syntax of the templates

SPIP uses a syntax to write templates which has a limited vocabulary but which is also extremely rich and modular in nature. This syntax, defined explicitly in the ecrire/public/phraser_html.php files, contains elements such as:

- the loop ("boucle" in French)

```
<B_loopname>
... before content
<BOUCLE_loopname(TABLE){criteria}>
... for each matching element
</BOUCLE_loopname>
... after content
</B_loopname>
... else content
<//B_loopname>
```

- the field or tag ("champ" and "balise" in French)

```
[ before (#TAG{criteria}|filters) after ]
```

- the argument (`{args}`, `|filter` or `|filter{args}` on tags)
- criteria (`{criteria=param}` used on loops)
- code inclusion

```
<INCLURE{fond=included_code_segment_name}>
```

- placeholders ("idiome" in French) (language specific character strings)

```
<:type:string_code_name:>
```

- polyglots ("polyglotte" in French) (`<multi>` used throughout templates and in user text)

```
<multi>[fr]français[en]English</multi>
```

Analysing a template

When SPIP's parser analyses a template, it translates the syntax into a vocabulary known and understood by the compiler. We might then say that the parser is translating a particular language (the SPIP syntax), that we refer to as a "concrete syntax", into a precise language that we refer to as an "abstract syntax". This is defined by PHP objects in the ecrire/puclic/interfaces.php file.

With this page analysis, the parser creates a table describing it, sequentially and recursively, by using the vocabulary included by the compiler (the objects containing Text, Fields, Loops, Criteria, Placeholders, Includes, Polyglots).

To make things a little clearer, let's look at what table is generated by a few template examples.

A text

Template :

```
Simple text
```

Generated table: (output by a `print_r`)

```
array (
  0 =>
  Texte::__set_state(array(
     'type' => 'texte',
     'texte' => 'Simple text
',
     'avant' => NULL,
     'apres' => '',
     'ligne' => 1,
  )),
)
```

The table specifies that the first element read on the page (key 0) is a "Texte" element, starting on line 1, and holding the text string "Simple text".

A tag

Template:

```
[before(#VAL)after]
```

We can read from the generated table below, that the first element read on the page is a Field ("champ" in French) (a tag), that it’s name is "VAL", that it is not within a loop (otherwise the id_loop would be defined), and that what is in the optional section before the tag is a "Texte" element with the text string being "before".

Generated table:

```
array (
  0 =>
  Champ::__set_state(array(
     'type' => 'champ',
     'nom_champ' => 'VAL',
     'nom_boucle' => '',
     'avant' =>
    array (
      0 =>
      Texte::__set_state(array(
         'type' => 'texte',
         'texte' => 'before',
         'avant' => NULL,
         'apres' => '',
         'ligne' => 1,
      )),
    ),
     'after' =>
    array (
      0 =>
      Texte::__set_state(array(
         'type' => 'texte',
         'texte' => 'after',
         '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,
   )),
)
```

A loop

Let’s look at one more example of a loop using a tag, which is a little more complicated since it implies a circular reference in the generated table. Look at this simple template segment:

Template:

```
<BOUCLE_a(ARTICLES){id_article=3}>
#TITRE
</BOUCLE_a>
```

This loop selects article 3 and should display the title of the article. The page table, if we were to try to display it, would end up generating a recursion error. The illustration shows that the second element read in the loop is a Field ("champ" in French) or tag named "TITRE". This field contains a reference to the loop which it is defined within (`'boucles'=>array(...)`). This loop contains the tag which belongs to the loop containing the tag which belongs to the loop ...

Excerpt of the generated table

```
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(
...
```

Why use such references?
Quite simply because they are then used for calculating the tags. When a tag is calculated, a part of this table is passed as a parameter (the famous $p that we will meet often). This part simply relates to the tag's properties. To retrieve properties from the enclosing loop, all that is required (thanks to these references) is to call the parameter `$p->boucles[$p->id_boucle]`.

The assembly processes

The production of a page by the compiler is performed in the ecrire/public/assembler.php file.

This file calls functions to analyse what has been requested, retrieve the modified template, translate it into PHP, and return the results of the PHP code evaluations. And do all this whilst managing the various file caches.

SPIP generally uses the `recuperer_fond()` function to retrieve the result of a template, but it also directly calls the `assembler()` function from the `ecrire/public.php` file.

Function call sequence

The `recuperer_fond()` function calls `evaluer_fond()` which calls `inclure_page()` which calls the `cacher()` function in the ecrire/public/cacher.php file. This is the same `cacher()` function which also calls `assembler()`.

Determining the cache

The ecrire/public/cacher.php file is used for managing the files stored in the cache.

The `cacher()` function retrieves the name and date of a cached page if it exists, depending on the context that has been provided. If it is also passed a file address, then the cache file is created.

As such, this function can be called in 2 ways:

- the first time to determine the name of the cache file and to indicate if a valid cache exists for the requested page.
- a second time when there is no valid cache. The page is then calculated by the `parametrer()` function, and then the `cacher()` function is called, this time to store the results in the cache.

```
// This function is used twice
$cacher = charger_fonction('cacher', 'public');
// The last four parameters are modified by the function:
// location, validity, and, if valid, contents & age
$res = $cacher($GLOBALS['contexte'], $use_cache,
$chemin_cache, $page, $lastmodified);
```

Parameters determining the name of the template

The ecrire/public/parametrer.php file makes it possible to create the parameters which will be necessary to retrieve the name and details of the template to be compiled using `styliser()`, and then request its calculation using `composer()`.

This is how the `parametrer()` function calculates the requested language as well as the number of the current section (rubrique) if that is possible.

These parameters then enable the name and address corresponding to the requested page to be determined. This is done by calling the `styliser()` function which is passed the arguments in question.

Determining the template file

The ecrire/public/styliser.php file determines the name and type of the template depending on the arguments which are passed to it.

```
$styliser = charger_fonction('styliser', 'public');
list($skel,$mime_type, $gram, $sourcefile) =
    $styliser($fond, $id_rubrique_fond,
$GLOBALS['spip_lang'], $connect);
```

A 5th argument makes it possible to request a parser (a concrete syntax) and then consequently an extension for the various template files. By default, the parser (and therefore the extension applied) is `html`.

The function searches for a template named `$fond.$ext` in SPIP's *path*. If it does not exist, it returns an error, otherwise it atempts to find a more specific template in the same directory as the template found, depending on the `id_rubrique` and `lang` parameters.

Styliser then searches files like nom`=8.html`, `nom-8.html`, `nom-8.en.html` or `nom.en.html` in the following order:

- `$fond=$id_rubrique`
- `$fond-$id_rubrique`
- `$fond-$id_rubrique_parent_recursivement`

- then whichever it has found (or not) terminated with `.$lang`

The function then returns a table of elements of what it has found `array($squelette, $ext, $ext, "$squelette.$ext")` :

- 1st parameter: the name of the template,
- 2nd: its extension
- 3rd: its grammar (the type of parser)
- 4th: the full name.

These parameters are used by the composer and its `composer()` function.

A clean composition

The ecrire/public/composer.php file is intended to retrieve the template translated into PHP and to execute it in the requested context.

If the template has already been translated into PHP, then the result is retrieved from a cache file and used, otherwise SPIP calls its compilation function `compiler()` to translate the concrete syntax into abstract syntax and then into code that is executable by PHP.

The `composer.php` file also loads the functions necessary for executing the PHP files output by the compilation of the templates.

The compilation

The SPIP compiler, in the ecrire/public/compiler.php file, is called using the `compiler()` function from within the `parametrer()` function.

Compilation starts by calling the appropriate parser depending on the grammar requested (the concrete syntax of the template). So it is the `phraser_html()` parser which is called in the ecrire/public/phraser_html.php file. It transforms the syntax of the template into a table (`$boucles`) of lists of PHP objects forming the concrete syntax that the compilation function will analyse.

For each loop found, SPIP performs a certain number of processes, starting by looking for which SQL tables match and which joins have been declared for these tables.

It then calculates the criteria applied on the loops (declared in ecrire/public/criteres.php or via plugins), and then the content of the loops (which have tags defined for some of them in ecrire/public/balises.php). It then proceeds to calculate the template elements that are outside of any loop.

Finally, it runs the loop functions that are dec-ared in the ecrire/public/boucles.php file. The result of all this builds a PHP-coded executable with a PHP function for each loop, and an overall PHP function for the whole template.

It is then this executable code which the compiler returns. This code will be stored in the cache then executed by the composer with the contextual parameters that have been passed. The result is the code for the requested page, which will be stored in cache (by calling the `cacher()` function a second time, in the `assembler.php` file) and then sent out to the browser (or if it is an inclusion, added to a page fragment). It may still contain PHP when certain details must display depending on the person visiting the page, such as with dynamic forms.

The cache

Using the various caches is an intrinsic component of SPIP that allows the various pages to be generated faster for the site visitor, thereby improving site response and performance. Any data that is frequently accessed, or which take longer times to be processed, are kept at the ready so they do not need to be generated "on the fly".

The template cache

There are various different caches used within SPIP, and others are also provided with various plugins, such as "Mémoisation", "Fastcache", "Expresso" or "Cache Cool".

One of the most important caches is the one maintained for the templates: whenever a template file is compiled, and the resultant PHP code generated, then these results are stored in cache in the `temp/cache/skel` directory. This cache is configured to have a validity period that is unlimited. Files in this cache are only regenerated for each template file when:

- the original template file has been modified (which is based on the file's storage date on disk),
- either of the files `mes_options.php` or `mes_fonctions.php` have been modified,
- the cache is emptied manually by an administrator.

The page cache

A second level of caching is maintained for pages requested by site visitors. These page results are saved in a series of directories named `tmp/cache/0` through `tmp/cache/f` and each cache file has its own validity period.

These cache files are created when:

- the validity period has expired and the page is requested anew (the period is defined in the templates using the `#CACHE` tag, or absent such a specific tag, through the system constant `_DUREE_CACHE_DEFAUT`),
- the editorial content in the database has been changed. SPIP relies on the registered date of last modification to determine if this has happened: (`$GLOBALS['meta']['derniere_modif']`) as provided by the function `suivre_invalideur()` de ecrire/inc/invalideur.php,

- the parameter `var_mode=calcul` is explicitly passed to SPIP in the URL, such as is the case when using the "Refresh this page" button on the public site when you are currently logged in.

The SQL cache

SPIP stores certain database elements in a cache in order to prevent over-working the SQL server, and so that the public pages already cached can be displayed if the database server doesn’t work for some reason. There are two such caches created for these purposes.

Cache of the meta data

The first cache is a complete export of the SQL table `spip_meta`. This table stores the parameters used for configuring and running SPIP. These data are stored both in the global variable `$GLOBALS['meta']` and, except for sensitive data used for authentication, in the file named `tmp/meta_cache.php`. This file has a validity period defined by `_META_CACHE_TIME`. It is rewritten when calls are made to `ecrire_meta()` or `effacer_meta()`. The function `lire_metas()` recalculates the contents of this cache `$GLOBALS['meta']` using the current data in the database.

Cache of SQL descriptions

The second cache concerns the descriptions of the database SQL tables. These descriptions are stored in the `tmp/cache/sql_desc*.txt` files, along with a file for each database connector. This file is created and used by the function `base_trouver_table_dist()`, which is used by various PHP functions for SQL descriptions like `table_objet()`, `id_table_objet()`, and `objet_type()`.

To create this cache file, it is necessary to explicitly call the function `trouver_table()` without any arguments:

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

The plugins cache

There are some cache files specific to plugins which are also created in `tmp/` or in `tmp/cache/`.

plugin_xml.cache

The results from analysing the various `plugin.xml` files is stored in a cache in the `tmp/plugin_xml_cache.gz` file.

This file is created when the list of active plugins is changed by the function `ecrire_plugin_actifs()`, which calls the function `plugins_get_infos_dist()` from ecrire/plugins/get_infos.php to manage the retrieval of data for a plugin. The file can also be deleted, as for numerous cache files when updates are made to the database structure.

Plugin load files

Plugins typically declare files for options, functions and actions to be executed for pipeline calls. All of the files to be loaded are compiled into 3 files, recalculated whenever the plugin management page is accessed at `ecrire/?exec=admin_plugin`, when the cache is manually emptied, or when there is an update to the database structure:

- `tmp/cache/charger_plugins_options.php` contains the list of option files to be loaded,
- `tmp/cache/charger_plugins_fonctions.php` contains the list of function files,
- `tmp/cache/charger_plugins_pipelines.php` contains those files used for the functions to be executed for each pipeline.

The path cache

SPIP uses various folders to look for the files that it needs to operate. More on this subject here: The concept of path (p.104). When it uses the function `find_in_path` — a function which is essential for three other functions: `include_spip`, `charger_fonction`, `recuperer_fond`, to look for a file to be included by a template or if it uses the `#CHEMIN` tab —, then all of the paths are searched through until the sought-after file is located. These numerous searches create frequent, repetitive disk accesses which would be better to be restricted in number if possible.

For this reason, SPIP uses the `tmp/cache/chemin.txt` file to cache all of the matches between a requested file and its actual logical location as found in one of the path's sequenced directories.

With this accomplished, whenever a file is requested, SPIP first checks to see if the path for that file is in the cache. If it's not already there, then SPIP proceeds to calculate its location and updates the correspondence table with a new entry for the newly located file.

This cache file is recreated when calling with the `var_mode=recalcul` parameter in the URL, or as a direct consequence of manually emptying the cache in the administrative interface.

The CSS and JavaScript caches

The "Compresseur" extension in SPIP is used to compress the various CSS and Javascript elements to restrict the number of calls to the server and the size of the generated files.

This compression is active by default in the private zone, and can be deactivated using the constant `_INTERDIRE_COMPACTE_HEAD_ECRIRE`.

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

This compression can be activated on the public zone depending on the configuration selected. SPIP will create a compressed CSS file for each media type (screen, print...), and a compressed JavaScript file for all of the external scripts defined in the HEAD of the HTML page.

These files are cached in `local/cache-js/` and `local/cache-css/`. These caches are recalculated whenever the `var_mode=recalcul` parameter is passed in the URL.

The image processing cache

SPIP has a library of graphical filters that are used by default to easily help resize images. These functions are defined in detail in ecrire/inc/filtres_images_mini.php. The "Filtres Images et Couleurs" extension, which is active by default, offers numerous other filters as well, like creating text images or using masks, merging images, colour manipulation, etc.

In order to avoid recalculating the same very time-consuming processes over and over, SPIP stores the results of these kinds of processes in the `local/cache-gd2` and `local/cache-vignettes` directories.

These cached images will only be deleted when the image cache is manually emptied from SPIP's administration module, or when the parameter `var_mode=images` is included in the URL.

Refreshing the cache

During normal usage of a SPIP site with public visitors and new articles being published regularly, the cache and the updating of data is handled correctly. By default (although some plugins are able to alter this behaviour), as soon as SPIP becomes aware of changes to the editorial content in the database, it invalidates the whole page cache. A requested page will then be calculated again from before - or after if using the "Cache Cool" plugin - being served up to the site visitor.

It is often necessary to manually empty the cache when making modifications directly on content files, especially when updating a stylesheet or a JavaScript script which is calculated by a SPIP template if the compression options have been activated.

Remember that:

- `var_mode=calcul` in the URL updates the page's cache
- `var_mode=recalcul` (for administrators) in the URL recompiles the template and then updates the page cache.
- entering the plugin management page `ecrire/?exec=admin_plugin` recalculates the cache files `tmp/cache/charger_*.php` for the plugins, which may be lists of files for options, functions or pipelines.
- the browser has its own private cache, which may be for whole pages or for AJAX elements. The site visitor and any administrator should also

think about emptying this cache - it is not necessarily SPIP which is returning a particular unexpected result, as it may be the browser returning data from its own cache. Various instructions for emptying a browser's local cache are specific to each browser and each platform - please consult your appropriate user guide.

Configuring the cache

There are several parameters used to finely tune SPIP's page cache.

Cache longevity

It is essentially a useless act to provide every SPIP template a specific cache duration by using the #CACHE markup tag. This tag is, however, useful for defining a validity duration that is different from SPIP's default value. In concrete terms, a piece of include code that lists news items from other syndicated sites will benefit from being refreshed more regularly than the default for the rest of the site, perhaps once every hour, for example.

In most cases, it's better to use a longer cache duration by default, since SPIP will automatically obsolete the cache when changes have been made to any content.

The page cache is defined as 24 hours, which can be modified by changing the constant `_DUREE_CACHE_DEFAUT`, as in this example where it is assigned to one month (30 days):

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

Cache size

SPIP organises itself so that the cache does not exceed a certain predetermined maximum size, set to 10 MB by default. The global variable `$GLOBALS['quota_cache']` is used to change this default value, as shown in this example which sets the upper limit to 100 MB:

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

Cache validity

A final facility is provided for development or debugging reasons, which can be used to modify the operation or usage of the cache. A constant named _NO_CACHE is used for this purpose (or simply use the "NoCache" plugin):

```
// never use the cache
// and don't even create the cache files
define('_NO_CACHE', -1);
// do not use the cache file,
// but store the results of the calculation in the file cache
define('_NO_CACHE', 1);
// always use the cache files if they exist
// if they don't, then calculate them
define('_NO_CACHE', 0);
```

Periodic tasks (cron)

The genie directory, French for wizard among other things, is used to store the periodic tasks, more generally referred to as cron tasks.

How cron jobs are run

It must be stressed that such cron jobs will not run at all if no-one ever visits the pages that have the `#SPIP_CRON` tag embedded - they are not cron jobs scheduled on the server, as might be assumed, they are simply procedures that are run intermittently and triggered by the activity of visits to the website pages themselves.

The tasks to be executed are called each time a site visitor views the page. A visitor's viewing of a page only executes a single cron task for each page called, if there is actually one to be processed.

However, for tasks to be called, the `#SPIP_CRON` tag must be present in the page template. This tag returns an empty image, but will run the task processing script. A text browser also runs the periodic tasks if the tag is not present.

To call the `cron`, you only need to execute the `cron()` function. This function takes an argument specifying the number of seconds which must elapse before another task can be launched, 60 seconds by default. Calls using `#SPIP_CRON` are applied every 2 seconds with the following code:

```
cron(2);
```

Declaring a cron task

To declare a task, you need to specify its name and frequency in seconds using the `taches_generales_cron` pipeline:

```
function myplugin_general_cron_jobs($taches){
    $taches['nom'] = 24*3600; // once per day, every day
}
```

This task will be called at the appropriate time. The processes are placed in a file in the `genie/` directory, with the same name as the (`name.php`) task and including a `genie_name_dist()` function.

The function is passed the date at which it last performed that task as an argument. It must return a number:

- null, if the task has nothing to do
- positive, if the task has been run
- negative, if the task started but could not complete. This makes it possible to run tasks in batches (to avoid *timeouts* on PHP script executions because the processes run too long). In such cases, the negative number indicated corresponds to the number of seconds of interval before the next scheduled task run.

Example

This example is simple, originating from SPIP’s "maintenance" tasks in the `genie/maintenance.php` file, since it executes functions and always returns `1`, indicating that the action has been run.

```
// Various maintenance tasks
function genie_maintenance_dist ($t) {
    // (re)set .htaccess with deny from all
    // for the two nominated directories inaccessible
through http
    include_spip('inc/acces');
    verifier_htaccess(_DIR_ETC);
    verifier_htaccess(_DIR_TMP);
    // Confirm that neither table crashed
    if (!_request('reinstall'))
        verifier_crash_tables();
    return 1;
}
```

Forms

SPIP provides a simple and powerful process to manage forms, called CVT (Charger, Vérifier, Traiter i.e. Load, Verify, Process). It breaks down a form into four parts:

- a view, which is basically a SPIP template containing the HTML code of the form corresponding to `formulaires/{nom}.html`,
- and three PHP functions to load the form's variables, verify the submitted elements and process the form declared in the `formulaires/{nom}.php` file.

HTML structure

Forms are stored in the `formulaires/` directory. A special HTML syntax allows easy customisation and reuse of forms.

Displaying the form

A file `formulaires/joli.html` is called from a template file using the syntax `#FORMULAIRE_JOLI`, which then calls and displays the form.

The HTML of the form follows a standard format for all SPIP forms. The fields of the form are surrounded in a list of elements using `ul/li` HTML markup.

```
<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>
```

For the form to work properly, the `action` attribute must be provided by the `#ENV{action}` variable which is automatically calculated by SPIP. In the same fashion, the `#ACTION_FORMULAIRE{#ENV{action}}` tag must be present, as it calculates and adds the security key which will be automatically verified when the form is received.

A few comments: The form is surrounded by a CSS class called `formulaire_spip` and by a second with its own name, in this case `formulaire_demo`. The name can be recovered more easily using the context function: `#ENV{form}` (or directly with #FORM), which could generate: `<div class="formulaire_spip formulaire_#FORM">`. The `<li>` mark-up tags are assigned CSS classes of `editer_xx`, where xx is the field name, and possibly the obligatoire class to indicate (visually) that this field must be provided before submitting the form.

- The input tags are assigned a CSS class appropriate to the each field type (to remediate a deficiency in Internet Explorer with CSS that does not understand input[type=text])
- The submission buttons are framed with a CSS class of boutons

Easily employing AJAX

Surrounding the form's tag with an "ajax" CSS class tells SPIP to use AJAX, thereby making it possible to reload only the form and not the whole page.

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

Handling errors returned

The verifier() function of the form can return errors if the submitted field contents are not correct; which we will see in more detail later. To display these errors in the form's HTML, there are CSS classes and a naming system which are employed:

At the top of the form there are general errors (or success messages):

```
[<p class="reponse_formulaire
reponse_formulaire_erreur">(#ENV*{message_erreur})</p>]
[<p class="reponse_formulaire
reponse_formulaire_ok">(#ENV*{message_ok})</p>]
```

For each field, there is a message and a CSS class on the list item to visually tag the error. The field message is calculated using the #ENV{erreurs} variable which provides all the field errors:

```
#SET{erreurs,#ENV**{erreurs}|table_valeur{xxx}}
<li class="editer_xxx obligatoire[
(#GET{erreurs}|oui)erreur]">
    [<span class='erreur_message'>(#GET{erreurs})</span>]
</li>
```

This combines with the previous form to give:

```
<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>
```

Field separation using fieldset

When a form contains a large number of fields, they are generally broken up into various blocks, each known as a `fieldset` in HTML.

Such blocks of fields are sequenced into `ul/li` type lists:

```
[...]
<form method="post" action="#ENV{action}"><div>
#ACTION_FORMULAIRE{#ENV{action}}
<ul>
    <li class="fieldset">
        <fieldset>
            <h3 class="legend">Section A</h3>
            <ul>
                <li> ... </li>
                <li> ... </li>
                ...
            </ul>
        </fieldset>
    </li>
    <li class="fieldset">
        <fieldset>
            <h3 class="legend">Section 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>
```

This example shows that you can have embedded lists, with the first `<li>` having the CSS class of "fieldset". Instead of providing HTML `<legend>` tags, a different format is provided using `<h3 class="legend">`, which offers a lot more opportunity for CSS tag styling.

Radio and checkbox fields

To display element lists of radio or checkbox controls, the syntax provided for wrapping the elements uses a `<div class="choix"></div>`. This formatting makes it possible to have buttons display before the labels, or to provide a horizontal radio list (using CSS statements).

```
<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>
```

To make the list display in horizontal mode using CSS, just make the "choix" divs display as `inline`:

```
.formulaire_spip .editer_syndication .choix {display:inline;}
```

Explaining input fields

It is often necessary to provide an explanation so that the user knows how to correctly fill out particular fields in a form. SPIP offers 2 formats to do this, by inserting a <p> or <em> tag with a special CSS class:

- **explication** (on a <p> element) is used to provide a more detailed explanation than the label of the field in question
- **attention** (on an <em> element) highlights a description that has been provided. To be used with moderation!

These two descriptions are therefore additional to the other options already provided for an `erreur` (error) or an `obligatoire` (mandatory) field.

Example

```
#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>
```

Conditional displays

The `charger()` or `traiter()` functions can indicate if the form is editable or not in their responses. This provides a means of receiving an `editable` parameter in the template, which can be used to hide or display the form as desired (but not the error or success messages).

It is used like this `[(#ENV{editable}) ... contents of the <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>
```

For any loops in the form

If there is a SPIP loop inside the code `[(#ENV{editable}) ... ]` (or any other tag), the SPIP compiler returns an error (or incorrectly displays the page) since this feature has not been envisaged in the current version of the template language.

To remediate this, you need to either:

- put the loop in an include, and then call it using an `<INCLURE{fond=mon/inclusion} />`
- or use the Bonux plugin and its CONDITION loop as follows:

```
<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>
```

PHP processing

The `formulaires/{nom}.php` files contain the three core functions related to the CVT forms in SPIP:

- `formulaires_{nom}_charger_dist` (loading),
- `formulaires_{nom}_verifier_dist` (verifying), and
- `formulaires_{nom}_traiter_dist` (processing).

Passing arguments to the CVT functions

The `charger()`, `verifier()` and `traiter()` functions do not receive any parameters by default.

```
function formulaires_x_charger_dist(){…}
function formulaires_x_verifier_dist(){…}
function formulaires_x_traiter_dist(){…}
```

For these functions to receive parameters, they need to be submitted as arguments explicitly when calling the form.

```
#FORMULAIRE_X{argument, argument, …}
```

The PHP functions receive the parameters in the same order:

```
function formulaires_x_charger_dist($arg1, $arg2, …){…}
function formulaires_x_verifier_dist($arg1, $arg2, …){…}
function formulaires_x_traiter_dist($arg1, $arg2, …){…}
```

Note that there is a supplementary possibility to use the functions for dynamic tags, which make it possible to pass parameters automatically.

Example

The "Composition" plugin contains a form which requires a type and an identifier. It is called as follows:

```
[(#FORMULAIRE_EDITER_COMPOSITION_OBJET{#ENV{type},
#ENV{id}})]
```

The processing functions therefore receive these two parameters:

```
function
formulaires_editer_composition_objet_charger($type,
$id){…}
```

Loading values into the forms

The `charger()` function makes it possible to specify which fields should be retrieved when the form is submitted, and also makes it possible to define the default values for such fields.

This function quite simply returns a paired table of "field name" / "default value" pairs:

```
function formulaire_nom_charger_dist() {
    $valeurs = array(
        "field" => "default value",
        "another field" => "",
    );
    return $valeurs;
}
```

All the keys specified will be passed into the form's HTML template environment. These data are then retrieved using `#ENV{field}` references. As soon as the form is posted, it will be the values entered by the user which take priority over the default values.

There is no need to protect the system from values entered that contain quotation marks, as SPIP already takes care of these automatically. Nonetheless, fields starting with an underscore "_" are not subject to this automatic processing, which makes them useful for passing complex variables.

Authorise the display or hiding of a form

Forms are displayed by default, but it is possible to restrict this display depending on certain assigned authorising data.

Two possibilities exist:

- either you don’t want to display the formula at all, so then return a `false` :

```
function formulaire_nom_charger_dist() {
    $valeurs = array();
    if (!autoriser("webmestre")) {
        return false;
    }
    return $valeurs;
}
```

- or simple hide a part of the form (often the editable part) by using the "editable" variable, which is then handled in the form template:

```
function formulaire_nom_charger_dist() {
    $valeurs = array();
    if (!autoriser("webmestre")) {
        $valeurs['editable'] = false;
    }
    return $valeurs;
}
```

Example

The "Accès restreint" (limited access) plugin has a form for assigning zones to an author; it passes the fields to be retrieved and their default values into the environment: the zone identifier, the connected author, and the author to be assigned to the zone. In addition, if the author does not have adequate rights, the "editable" variable is passed as false.

```
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;
}
```

Other preloading options

Various other special parameters can be sent to the form when it is loaded to modify its original behaviour:

message_ok, message_erreur

The success message is generally supplied by the `traiter` function; the error message is supplied by the `verifier` or `traiter` functions. It is nonetheless possible to supply them using the `charger` function in exceptional circumstances.

action

This value specifies the URL to which the form is posted. By default, it is the same URL as the current page, which makes it possible to redisplay the form if errors are detected. For other very special use cases, this URL can be altered.

_forcer_request

When a form is submitted, SPIP identifies it so as to be able to have several forms of the same type on a single page, and to only process the one that has been submitted. This verification is based on the list of arguments passed to the #FORMULAIRE_XXX tag.

In some cases where these arguments change after data entry, SPIP can make a mistake and assume that the data comes from another form.

Sending `_forcer_request` as `true` indicates to SPIP that it should not perform this verification and ought to process the entry data in every circumstance.

_action

If the processing of the form must call a directory function `actions/` protected by `securiser_action()`, it is useful to specify the name of the action so that SPIP automatically supplies the corresponding protection hash.

_hidden

The value of this field will be added directly to the HTML of the generated form. it is often used to add "hidden" type input fields which should be written out explicitly:

```
$valeurs['_hidden'] = "<input type='hidden' name='secret'
value='shhhhh !' />";
```

Pipelines used for loading

formulaire_charger

This pipeline makes it possible to modify the table of values returned by the `charger` function for a form. It is more fully described in the chapter about pipelines: formulaire_charger (p.160)

paramètre _pipeline

This parameter makes it possible to modify the HTML code published by making it pass through a specific pipeline. This data, sent in the loading table, makes it possible to specify the name of a pipeline and the arguments to be passed to it. It will be called at the time the form text is displayed.

Example

SPIP uses the parameter in a generic fashion by making all publication forms that call the `formulaires_editer_objet_charger()` function pass through a pipeline named `editer_contenu_objet`. This pipeline is described in its own special article: editer_contenu_objet (p.159).

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

The CFG plugin uses this parameter to make all the CFG forms written as CVT forms pass through the `editer_contenu_formulaire_cfg` pipeline.

```
$valeurs['_pipeline'] =
array('editer_contenu_formulaire_cfg',
    'args'=>array(
        'nom'=>$form,
        'contexte'=>$valeurs,
        'ajouter'=>$config->param['inline'])
);
```

The pipeline that CFG then uses to collect the contents not necessary in the transmitted HTML:

```
// pipeline for the display of content
// to delete the CFG parameters from the form
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;
}
```

Checking the submitted values

The `verifier()` function is used to analyse the values posted and return errors that may exist concerning the data entered. To do this, the function returns a paired "field" / "error message" array of the offending fields, and also possibly a general message for the whole of the form using the "message_erreur" key.

The form processing function will be called on ONLY if the table returned is empty. If it is not, the form is redisplayed with the various error messages that have been passed.

```
function formulaire_nom_verifier_dist() {
    $erreurs = array();
    foreach(array('titre','texte') as $champ) {
        if (!_request($champ)) {
            $erreurs[$champ] = "This data is mandatory!";
        }
    }
    if (count($erreurs)) {
        $erreurs['message_erreur'] = "An error occured in
your data entry";
```

```
    }
    return $erreurs;
}
```

The formulaire_verifier (p.162) pipeline is used to supplement the list of returned errors.

Example

The "Amis" (Friends) plugin has a form for inviting people to become your friend! The `verifier()` function checks that the mail address of the person being invited is correctly formatted:

```
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;
}
```

Executing the processes

Whenever the verification function (p.242) doesn't return an error, the form then moves on to the `traiter()` (processing) function. It is in this function that the desired operations should be performed with the data from the form (send an email, update the database, etc.).

The function must return an associative table:

```
function formulaires_nom_traiter(){
    // Execute the processes

    // Return values
    return array(
        'message_ok' => 'Excellent !', // or perhaps
        'message_erreur' => 'Sorry, an error has occurred.'
    );
}
```

Important values

Here are some of the values frequently returned:

- **message_ok** is used to return a pleasant message to the user indicating that everything processed normally.
- **message_erreur**, on the other hand, is used to return an error message when the processing didn't work correctly.
- **editable**, as for loading, this is used to display or hide the editable portion of the form. By default it is set to `false`, but you may assign it a value of `true` if your form can be used several times in a row.
- **redirect** is a URL which is used to tell SPIP which page it should redirect to after processing the form. By default, the page will loop back to itself.

The formulaire_traiter (form_process) pipeline

Once the `formulaires_nom_traiter` function has completed, the formulaire_traiter (p.161) pipeline is executed, thereby enabling other plugins to complete the processes for this form.

Processing without AJAX

If a form is called using AJAX but then redirects to another page after finishing its processes, this would require Javascript tricks (managed by SPIP) to capture that redirection and effectively send the browser to another URL instead of the normal response.

Whenever a redirection is certain, it is possible to prevent AJAX for the form's processing, while still maintaining it for the verification phase. This means that the form would be reloaded in the event of an error in `verifier()`, but if the processing is executed, then the whole page will be immediately reloaded.

To do this, you must call the `refuser_traiter_formulaire_ajax()`function **right at the start** of the processes:

```
function formulaires_nom_traiter(){
    // Prevent AJAX processing since we know that the form
will redirect elsewhere
    refuser_traiter_formulaire_ajax();

    // Execute the processes

    // Return values
    return array(
        'redirect' => 'Another URL'
    );
}
```

Examples

Management of the CVT forms deserves some dedicated examples of its own.

Translate anything

This simple example will create a small form that calls an external translation service to translate the content entered on that form. The result will be displayed underneath the source text that was entered.

The form will be called "translate_anything" and can then be called in a regular SPIP template file using the tag `#FORMULAIRE_TRANSLATE_ANYTHING` or within an article by using `<formulaire|translate_anything>`.

As with most CVT forms, it operates using two files:

- `formulaires/translate_anything.html` for the HTML section
- `formulaires/translate_anything.php` for the PHP analysis and processing functions.

The HTML template

The template for the form will receive two data entry fields of the `textarea` type: the first for writing the content to be translated, and the second to display the results of the translation once the calculation has been performed. This second field is only displayed when it actually has some content.

```
<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">Source text</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">Translated text</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="Translate" /></p>
</div></form>
</div>
```

The two fields named "traduire" and "traduction" (source and destination for the translation). The same template could be written using the "Saisies" plugin with the content between `<ul>` and `</ul>` represented as follows:

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

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

Loading, verifying and processing

The "loading" of the form, declared in the `formulaires/translate_anything.php` file, must specify that is is adding the two "traduire" and "traduction" fields into the template’s context:

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

The "verify" function simply needs to test if there has actually been some content entered into the "traduire" field and return an error is there hasn’t:

```
function formulaires_translate_anything_verifier_dist() {
    $erreurs = array();
    if (!_request('traduire')) {
        $erreurs['message_erreur'] = "You have not entered
any text to translate - is your keyboard broken?";
        $erreurs['traduire'] = "Normally that is how you
enter text, isn't it?";
    }
    return $erreurs;
}
```

It is with the "process" function that things now get a little complicated. The content needs to be sent to a remote service (we use *Google Translate* in this example), the return data retrieved and processed, and then displayed on our form.

To do all this, the script starts by calculating the URL for the remote service based on that service’s published API. We use SPIP’s `parametre_url` PHP function to cleanly add the variables to the service’s URL. Thanks to another function, `recuperer_page` which is used to retrieve the code returned by a call to an URL, the service’s returned data is stored in the `$trad` variable.

The service returns the data formatted in JSON format, so it must be extricated using the `json_decode` function. depending on the information returned, the translation will be determined as having been successful or not. The message is adapted depending on this outcome.

```
// 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");
function formulaires_translate_anything_traiter_dist() {
    // create the google api URL
    $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, '&');
    // load the text as translated by google (returned as
JSON code)
    include_spip('inc/distant');
    $trad = recuperer_page($url);
    // warning: uses PHP 5.2
    $trad = json_decode($trad, true); // true = retour array
et non classe
    // retrieve the results if OK
    if ($trad['responseStatus'] != 200) {
        set_request('traduction', '');
        return array(
            "editable" => true,
            "message_erreur" => "Bad luck, Google couldn't
help!"
        );
    }
    // send the data to be loaded
    set_request('traduction',
$trad['responseData']['translatedText']);
    // message
    return array(
        "editable" => true,
        "message_ok" => "And here's the translation!",
    );
}
```

The `set_request()` functions forces the saving of a variable value that can then later be retrieved using `_request()`. This allows the next loading of the form to retrieve the value of the "traduction" field to send it into the template's context.

Note: It is possible that a cleaner method could be developed for future versions of SPIP in order to transit the data between the processing and loading phases using a new parameter in the processing return table.

Calculating the day-of-the-year

This short example makes it possible to calculate and display the day of the year for a date entered on a form.

This form will be named "calculate_doy", and can then be called from within a SPIP template file with `#FORMULAIRE_CALCULATE_DOY` or within the text of an article by using `<formulaire|calculate_doy>`.

Implementation

The two files necessary will be created as follows:

- `formulaires/calculate_doy.html` for the HTML section
- `formulaires/calculate_doy.php` for the PHP analysis and processing the CVT functions.

The HTML template file

The `formulaires/calculate_doy.html` file contains the following code, respecting the recommended HTML structure and CSS classes:

```
<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 (dd/mm/yyyy)
:</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="Calculate" />
    </p>
</div></form>
]
</div>
```

Note that the "Saisies" plugin can be used to write the form's fields using a `#SAISIE` tag, and specifying the type and name of the variable used, followed by whichever optional parameters are useful. Doing so would produce (the code section between `<ul>` and `</ul>`):

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

Loading, verifying and processing

The `formulaires/calculate_doy.php` file contains the three following functions:

The "loading" file lists the variables which will be passed into the template environment and initialises their default values. There is no default date here, but it would be possible to specify one if you wanted.

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

The "verify" function checks to make sure the compulsory fields are entered and that the date format appears to be correct:

```
function formulaires_calculate_doy_verifier_dist (){
    $erreurs = array();
    // compulsory fields
    foreach(array ('date_jour') as $obligatoire) {
        if (!_request($obligatoire)) $erreurs[$obligatoire] =
'This field is compulsory';
    }
    // correct date format
    if (!isset($erreurs['date_jour'])) {
        list($jour, $mois,U $annee) = explode('/',
_request('date_jour'));
        if (!intval($jour) or !intval($mois) or
!intval($annee)) {
            $erreurs['date_jour'] = "Unknown date format.";
        }
    }
    if (count($erreurs)) {
        $erreurs['message_erreur'] = 'Your data contains
errors!';
    }
    return $erreurs;
}
```

If the verifications are correct (no errors found), then the "process" function is executed. The form is declared as re-editable, which means that a new date value can be entered again immediately after the validation.

```
function formulaires_calculate_doy_traiter_dist (){
    $date_jour   = _request('date_jour');
    $retour = array('editable' => true);
    if ($doy = calculate_doy($date_jour)) {
        $retour['message_ok'] = "The day of the year for
$date_jour is $doy";
    } else {
        $retour['message_erreur'] = "DOY calculation error!";
    }
    return $retour;
}
```

Of course, this still omits the function used to calculate the day-of-the-year, but a few simple lines of PHP will fix that. This function can be implemented in the same file as the three previous functions:

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

SQL access

SPIP 2 can read, write and use the following database management systems: MySQL, PostGres and SQLite.

Although their query syntax is not the same, thanks to a set of special SQL abstract functions, SPIP allows the development of database interactions independent of the systems.

Modification of the SQL manager

SPIP essentially applies the SQL standards, but will also understand a large portion of the MySQL particularities, that it will then translate for the SQLite or PostGres database managers when necessary.

SPIP does not need any special declaration (other than the presence of the connection file necessary for the database in question) in order to read and extract data from such databases, regardless of whether this is through the use of templates or, via PHP, through the SQL abstraction functions envisaged and prefixed with `sql_`.

Declaring table structures

In certain cases, particularly for plugins which add tables into the database, or add columns into a table, it is necessary to declare the SQL structure of the table, since it is based on these declarations that SPIP constructs its queries to create or update the tables.

SPIP will therefore attempt to modify the declaration to the database manager being used, by converting certain syntax that is particular to MySQL.

As such, if you declare a table with an "auto-increment" on the primary key as proscribed by SPIP (as in ecrire/base/serial.php and ecrire/base/auxiliaires.php by using the SPIP 2 specific pipelines declarer_tables_principales (p.153) and declarer_tables_auxiliaires (p.145), SPIP will then translate the "auto-increment" syntax so that it is appropriately accommodated when using PostGres or SQLite.

In the same fashion, a declaration for an "ENUM" field specific to MySQL will have the same functionality under PG or SQLite. The inverse, on the other hand, is not true (PostGres specific declarations will not be understood by the other databases).

Table updates and installation

When SPIP installs itself, it uses particular functions to install or update its tables. Plugins may also use these functions in their own installation routines.

These functions are declared in the ecrire/base/create.php file.

Creating tables

The `creer_base($connect='')` function creates tables missing in the database which has the connection file specified in `$connect`. By default, this is the principal connection.

This function creates the missing tables (of course, they must have already been defined), but does nothing for modifying an existing table. If the table is declared as a principal table (and not an auxiliary table), and if the primary key is an integer, then SPIP will automatically assign an 'auto-increment' type to this primary key.

Updating tables

The `maj_tables($tables,  $connect='')` function updates existing tables. It will only create fields that are missing; no field deletion will be performed. The table name (character string) or list of table names (table) must be provided to the function. There again, it is possible to specify a different connection file other than the principal database.

If a table to be updated does not exist, it will be created, following the same principle as `creer_base()` does for the auto-increment.

Examples:

```
include_spip('base/create');
creer_base();
maj_tables('spip_rubriques');
maj_tables(array('spip_rubriques','spip_articles'));
```

The SQL API

SPIP's SQL abstraction functions constitute an API which contains the following functions:

Name	Description
Common elements (p.258)	System parameters and options
sql_allfetsel (p.259)	Returns an array with all of the results from a selection
sql_alltable (p.261)	Returns an array of the existing SQL tables
sql_alter (p.261)	Modifies the structure of an SQL table
sql_count (p.263)	Counts the number of rows in a selection resource
sql_countsel (p.264)	Counts the number of results
sql_create (p.265)	Creates a table according to the schema provided
sql_create_base (p.267)	Creates a database
sql_create_view (p.267)	Creates a view
sql_date_proche (p.269)	Returns a date comparison expression based on a date calculation
sql_delete (p.269)	Deletes database records
sql_drop_table (p.270)	Deletes a table!
sql_drop_view (p.271)	Deletes a view
sql_errno (p.272)	Returns the number code for the last SQL error
sql_error (p.272)	Returns the last SQL error
sql_explain (p.272)	Explains how the SQL server will process a request
sql_fetch (p.273)	Returns a row from a selection resource
sql_fetch_all (p.275)	Returns a table with all the results from a query
sql_fetsel (p.276)	Selects and returns the first row of results
sql_free (p.277)	Releases a resource

Name	Description
sql_getfetsel (p.277)	Retrieves the single column requested from the first row in the selection
sql_get_charset (p.278)	Requests if a particular character encoding is available on the server
sql_get_select (p.279)	Returns the selection query
sql_hex (p.280)	Returns a numeric value for a hexadecimal character string
sql_in (p.281)	Constructs a condition using the IN operator
sql_insert (p.282)	Inserts content into the database
sql_insertq (p.283)	Inserts content into a database (automatically filtered)
sql_insertq_multi (p.284)	Used to insert several database rows in a single operation
sql_in_select (p.286)	Returns an `sql_in` condition from the results of an `sql_select`
sql_listdbs (p.287)	Lists the databases available for a given connection
sql_multi (p.287)	Extracts multilingual content
sql_optimize (p.288)	Optimises a given table
sql_query (p.289)	Executes a specific query
sql_quote (p.289)	Filters (or escapes) an SQL parameter
sql_repair (p.291)	Repairs a damaged table
sql_replace (p.291)	Inserts or modifies a record
sql_replace_multi (p.292)	Inserts or replaces several records
sql_seek (p.293)	Positions a selection resource at the designated row number
sql_select (p.293)	Selects content
sql_selectdb (p.296)	Selects the requested database
sql_serveur (p.297)	The API's principal transparent function

Name	Description
sql_set_charset (p.298)	Requests the use of the specified character encoding
sql_showbase (p.298)	Returns a resource of the list of database tables
sql_showtable (p.299)	Returns a description of the table
sql_update (p.300)	Updates a database record
sql_updateq (p.301)	Updates database content (and filters the data against SQL injection attacks)
sql_version (p.302)	Returns the version number of the database manager

Common elements

Within the set of `sql_*` functions, certain parameters are systematically available and are used to denote the same information. These parameters are all described here, principally so that they are not repeated ad infinitum in multiple articles:

- `$serveur` (or `$connect`) is the name of the SQL connection file (in the `config/` directory. When not defined or empty, then the connection file defined during SPIP installation will be used. Normally it is the penultimate (last but one) parameter for the SQL abstraction functions.
- `$options` equals `true` by default and is used to specify an optional character with its usage. This parameter is normally the last for the SQL abstraction functions. It may have the following values:
  - `true`: any function in the SQL API and not found in the SQL instruction set of the requested server will cause a fatal error.
  - `'continue'`: no fatal error if the function is not found.
  - and `false`: the SQL set function does not run the query which has been calculated, but should return it instead (we therefore obtain a text string that is a valid SQL query for the database manager requested).

Some other parameters are often present from one function to another, particularly so for all functions which are similar to `sql_select()` by reusing all or some of its parameters:

- `$select`, table of SQL columns to be retrieved,
- `$from`, table of SQL tables to be used,
- `$where`, table of column constraints where each element in the table will be combined with a logical AND,
- `$groupby`, table of groupings of the results,
- `$orderby`, table defining the ordering of the results,
- `$limit`, string indicating the maximum number of results to return,
- `$having` table of post-constraints for the aggregation functions.

For functions used to modify content, there is another common parameter:
- `$desc`, which is a table of column descriptions for the SQL table employed. If it is omitted, the description will be automatically calculated if the calling functions have need of it.

Coding principles

A large number of parameters are tolerant in respect of the type of argument which is passed to them, often accepting tables or text strings. This is the case, for example, for the `sql_select()` parameters. Its first parameter is `$select`, which corresponds to the list of SQL columns to be retrieved. Here are the 4 functional coding methods for this parameter:

```
// 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');
```

Out of convention, which imposes no obligations on anyone, we generally prefer to use the tabular (array) form whenever there is more than one element, a coding method which is easier to analyse by the functions which translate these abstracted coding methods into SQL queries.

sql_allfetsel

The `sql_allfetsel()` functions retrieves an array with all of the results of a selection. It accepts the same parameters as the `sql_select()` function and is a combined short-cut to replace calling `sql_select()` and `sql_fetch_all()`. As it stores all of the results in a PHP array, be careful not to exceed the memory limit allowed to PHP if you are dealing with large data volumes.

It accepts 9 parameters:

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

The `sql_allfetsel()` function is used as shown below:

```
$all = sql_allfetsel('column', 'table');
// $all[0]['column'] is the column in the first line
retrieved
```

Example

Select all of the `objet` / `id_objet` pairs associated with a particular document:

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

The "Contact avancé" plugin selects all of the emails for the recipients of a message as shown below:

```
// Retrieve who it was sent to
$destinataire = _request('destinataire');
if (!is_array($destinataire)) {
    $destinataire = array($destinataire);
}
$destinataire = array_map('intval', $destinataire);
$mail = sql_allfetsel('email', 'spip_auteurs',
sql_in('id_auteur', $destinataire));
```

sql_alltable

The `sql_alltable()` function returns an array listing the various SQL tables that exist in the database. it accepts the same parameters as sql_showbase (p.298):

1. `$spip` empty by default, the parameter is used to list only the tables using the prefix defined for SPIP tables. Use `'%'` instead if you want to list ALL tables,
2. `$serveur`,
3. `$option`.

Usage:

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

sql_alter

The `sql_alter()` function is used to send an `ALTER` type SQL command to the database server to modify the structure of the database.

The function accepts 3 parameters:

1. $q is the query string (without the term "ALTER") to be executed
2. `$serveur`,
3. `$option`

Note: This function directly assumes an SQL formatted command, so it is important to respect the SQL standards. It is possible in future versions of SPIP, that the `$q` parameter will accept a more structured table as input in order to simplify porting to other systems.

The function is used as shown in this example:

```
sql_alter("TABLE table ADD COLUMN column_name INT");
sql_alter("TABLE table ADD column_name INT"); // COLUMN is an
optional keyword for this SQL command
sql_alter("TABLE table CHANGE column_name column_name INT
DEFAULT '0'");
sql_alter("TABLE table ADD INDEX column_name (column_name)");
sql_alter("TABLE table DROP INDEX column_name");
sql_alter("TABLE table DROP COLUMN column_name");
sql_alter("TABLE table DROP column_name"); // COLUMN is an
optional keyword for this command
// You may pass several actions, but be careful about other
DBMS ports:
sql_alter("TABLE table DROP column_nameA, DROP
column_nameB");
```

The `sql_alter()` function is particularly used during updates for plugins in the `{plugin_name}_upgrade()` functions for the various plugins you may have installed.

Example

Add a "composition" column to the `spip_articles` table (plugin "Composition"):

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

Add "css" to the "spip_menus" table (plugin "Menus"):

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

The "TradRub" plugin includes in its installation procedure an instruction to add the "id_trad" column to the `spip_rubriques` table by using the `maj_tables()` function provided for such a purpose, then adds an index on that same column using `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 on the new field
            sql_alter("TABLE spip_rubriques ADD INDEX
(id_trad)");
            ecrire_meta($nom_meta_base_version,
$current_version=$version_cible, 'non');
        }
    }
}
```

sql_count

The `sql_count()` function returns the number of rows for a selection resource opened with `sql_select()`.

It accepts 3 parameters:

1. `$res` is the resource identifier for a selection,
2. `$serveur`,
3. `$option`.

It is used as shown below:

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

```
if ($res and sql_count($res)>2) {
    // checks to see if there are at least 3 rows in the
results!
}
```

Example

Possible application: display a count of the total number of elements.

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

sql_countsel

The `sql_countsel()` functions returns the number of rows for a desired selection. It is more-or-less a short way of writing `sql_select('COUNT(*)', ...)`.

It accepts the same arguments as `sql_select()` except for the first (normally the columns):

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

It is used as shown in this example:

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

Example

Count the number of keywords in a given keyword group:

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

Return `false` if a section has any articles NOT in the trash:

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

If the `spip_notations_objets` table in the "Notations" table does not yet have any entry for the object identifier specified, then perform a database insert, otherwise perform an update:

```
// Update or insert?
if (!sql_countsel("spip_notations_objets", array(
    "objet=" . sql_quote($objet),
    "id_objet=" . sql_quote($id_objet),
))) {
    // Insert a record for the object notation
    sql_insertq("spip_notations_objets", ...);
    // ...
} else {
    // Update if there already is a record
    sql_updateq("spip_notations_objets", ...);
    // ...
}
```

sql_create

The `sql_create()` function is used to create an SQL table according to the schema provided.

It accepts 7 parameters:

- `$nom` is the name of the table to create
- `$champs` is an array of column descriptions
- `$clefs` is an array of key descriptions
- `$autoinc`: if a field is to be a primary key and is numeric, then the auto-increment property will be added. `false` by default.
- `$temporary`: is this a temporary table? Default value: `false`
- `$serveur`,
- `$option`

It is used as shown below:

```
sql_create("spip_tables",
    array(
        "id_table" => "bigint(20) NOT NULL default '0'",
        "column1"=> "varchar(3) NOT NULL default 'oui'",
        "column2"=> "text NOT NULL default ''"
    ),
    array(
        'PRIMARY KEY' => "id_table",
        'KEY column1' => "column1"
    )
);
```

As a general rule, plugins should declare their SQL tables using the pipelines intended for the purpose: declarer_tables_principales (p.153) and declarer_tables_auxiliaires (p.145), and use the `creer_base()` or `maj_tables('spip_tables')` functions during installation of each plugin, which will call the `sql_create()` function when necessary. Read more on this topic here: "Table updates and installation (p.254)".

Example

Example of creating a "spip_mots_tordus" table which will be a link with "spip_tordus". Note that the primary key is composed from 2 columns:

```
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

The `sql_create_base()` function attempts to create a database with the name provided. The function returns `false` if an error occurs.

It accepts 3 parameters:

- `$nom` is the name of the database to create,
- `$serveur`,
- `$option`

This function is only used during the installation of SPIP to create a database as requested for a given database manager:

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

When using SQLite, the database name corresponds to the file name without the file type extension (`.sqlite` will be added automatically) and the file will be stored in the directory defined by the `_DIR_DB` constant, which by default is set to `config/bases/`.

sql_create_view

The `sql_create_view()` function creates a view for the selection query provided. The view can then be used by SPIP loops or by other selection commands.

It accepts 4 parameters:

1. `$nom` is the name of the view created,
2. `$select_query` is the selection query,
3. `$serveur`,
4. `$option`.

It can be used in conjunction with the sql_get_select (p.279) function to retrieve the desired selection:

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

`Note:` Whenever a selection column uses the `'name.column'` notation, you absolutely must declare an alias for the column, otherwise certain database ports (SQLite in particular) will not create the expected view, e.g. `'name.column AS column'`.

Example

This small example demonstrates this function by creating a (rather useless) table from 2 columns in a section:

```
$select = sql_get_select(array(
        'r.titre AS t',
        'r.id_rubrique AS id'
    ), array(
        'spip_rubriques AS r'
    ));
// create the view
sql_create_view('spip_short_rub', $select);
// use it:
$titre = sql_getfetsel('t', 'spip_short_rub', 'id=8');
```

The view could also be used within a SPIP template file, as in:

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

sql_date_proche

The `sql_date_proche()` function is used to return a conditional expression for a column in relation to a date.

It accepts 5 parameters:

1. `$champ` is the SQL column to be compared,
2. `$interval` is the comparison interval value: -3, 8, ...
3. `$unite` is the units of reference ('DAY', 'MONTH', 'YEAR', ...)
4. `$serveur`,
5. `$option`.

It is used as shown below:

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

Example

Another use for a selection query such as illustrated below, is to store the boolean result in an alias. The alias `recently` indicates whether or not an author has logged in during the last 15 days:

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

sql_delete

The `sql_delete()` function is used to delete records from an SQL table and returns the number of records that were all deleted.

It has 4 parameters:

1. `$table` is the name of the SQL table,
2. `$where`,
3. `$serveur`,
4. `$option`.

It is used as shown below:

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

Example

Delete the link between all sections and a given keyword:

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

One of SPIP's standard periodical tasks is to delete old articles that have been put in the dustbin (poubelle), as detailed below:

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

sql_drop_table

The `sql_drop_table()` function deletes an SQL table from the database. It returns `true` if successful, and `false` if not.

It accepts 4 parameters:

1. `$table` is the name of the table,

2. `$exist` is used to request verification of the table's existence for the deletion (which translates into adding `IF EXISTS` to the SQL command). By default, `''`, it passes `true` to confirm the table is there before trying to delete it,
3. `$serveur`,
4. `$option`.

This `sql_drop_table()` function is used as shown below:

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

Example

Plugins often use this function for complete removal (data included) of a plugin when so requested by the administrator, as shown in this example from the "Géographie" plugin:

```
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

The `sql_drop_view()` function deletes a database view. It accepts the same parameters as `sql_drop_table()` and returns `true` if successful and `false` if not.

Its 4 parameters are:

1. `$table` is the name of the view,

2. `$exist` used to request verification of the existence of the view before deletion (this translates into the addition of `IF EXISTS` to the SQL command). By default `''`, it includes `true` to request the verification,
3. `$serveur`,
4. `$option`.

The `sql_drop_view()` function is used as follows:

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

sql_errno

The `sql_errno()` functions returns the number code for the most recent SQL error that has occurred. This function is used in SPIP to automatically record the details in the incident logs generated for SQL actions, which are centrally managed by the `spip_sql_erreur()` function in ecrire/base/connect_sql.php

sql_error

The `sql_error()` function returns the most recent SQL error that has occurred. This function is used within SPIP to automatically record details for the incident logs generated for SQL actions, which are centrally managed by the `spip_sql_erreur()` function in ecrire/base/connect_sql.php

sql_explain

The `sql_explain()` function is used to return an explanation of how the SQL server will process a request. This function is used by the debugger to provide information relating to the generated SQL commands.

The function accepts 3 parameters:
1. `$q` is the SQL query,
2. `$serveur`,
3. `$option`.

One possible usage might be:

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

sql_fetch

The `sql_fetch()` function returns a row, in the form of an associative array, from the results of a selection. It returns `false` if there are no more rows to be retrieved.

It accepts 3 parameters, only the first of which is mandatory:

1. `$res` is the resource generated by an sql_select(),
2. `$serveur`,
3. `$option`.

This function is used in strict conjunction with `sql_select()`, often used in the following manner:

```
if ($res = sql_select('column', 'table')) {
    while ($r = sql_fetch($res)) {
        // use the results with $r['column']
    }
}
```

Example

List the articles proposed for 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
    }
}
```

The "Contact avancé" plugin can save messages in the `spip_messages` table. When one of these messages is deleted, it also deletes any documents that may be linked to it:

```
function action_supprimer_message() {
    $securiser_action =
charger_fonction('securiser_action', 'inc');
    $id_message = $securiser_action();
    // Check if we have any documents
    if ($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));
}
```

The `calculer_rubriques_publiees()` function within `ecrire/inc/rubriques.php` is used to recalculate the statuses and dates for sections in order to find out which have the status of "publié" (published). Within the function, a code segment selects the sections which have published documents (and therefore so does the section) and assigns a temporary column for the new status and new date. Once the updates are completed, the temporary column is saved into the real column:

```
// Set the counters to zero
sql_updateq('spip_rubriques', array(
    'date_tmp' => '0000-00-00 00:00:00',
    'statut_tmp' => 'prive'));
// [...]
// Publish and date the sections which have a published
*document*
$r = sql_select(
    array(
        "rub.id_rubrique AS id",
        "max(fille.date) AS date_h"),
    array(
        "spip_rubriques AS rub",
        "spip_documents AS fille",
        "spip_documents_liens AS lien"),
    array(
        "rub.id_rubrique = lien.id_objet",
        "lien.objet='rubrique'",
        "lien.id_document=fille.id_document",
        "rub.date_tmp <= fille.date",
        "fille.mode='document'", "rub.id_rubrique"));
while ($row = sql_fetch($r)) {
      sql_updateq('spip_rubriques',
        array(
            'statut_tmp'=>'publie',
            'date_tmp'=>$row['date_h']),
        "id_rubrique=" . $row['id']);
}
// [...]
// Save the modifications
sql_update('spip_rubriques', array(
    'date'=>'date_tmp',
    'statut'=>'statut_tmp'));
```

sql_fetch_all

The `sql_fetch_all()` function returns an array containing all of the rows for a selection resource. Since all of the results will be stored in current memory, you should be careful not to select too much content at once.

The `sql_fetch_all()` function accepts 3 parameters:

1. `$res` is the resource obtained using an sql_select(),
2. `$serveur`,
3. `$option`.

It is used as in the example below:

```
$res = sql_select('column', 'table');
$all = sql_fetch_all($res);
// $all[0]['column'] is the first row
```

However, this function is not often used, since the `sql_allfetsel()` function can execute much the same operation but also with selection parameters:

```
$all = sql_allfetsel('column', 'table');
// $all[0]['column'] is the first row
```

sql_fetsel

The `sql_fetsel` function returns the first row of results for a selection. It accepts the same parameters as the `sql_select()` function and is a short-cut for the combined call of `sql_select()` and `sql_fetch()`.

Its parameters are:

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

It is used as shown below:

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

Example

Select the "id_trad" and "id_rubrique" columns of a given article:

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

Select all the columns for a given news item:

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

sql_free

The `sql_free()` function is used to release an SQL resource opened using a call to the `sql_select()` function. Ideally this function ought to be called after finishing using each resource.

It accept 3 parameters:

1. `$res` is the resource for a selection,
2. `$serveur`,
3. `$option`.

The `sql_free()` function is therefore used as shown below:

```
$res = sql_select('column', 'table');
// operations using the sql_fetch($res) and similar functions
...
// then close the resource
sql_free($res);
```

Note that the API functions call this function automatically. This is the case for:

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

sql_getfetsel

The `sql_getfetsel()` function retrieves the single column requested from the first row of the selection. It accepts the same parameters as the `sql_select()` function and is a short-cut for the combination of calling `sql_fetsel()` and `array_shift()`.

Its parameters are:

1. `$select` nominating the desired column,
2. `$from`,

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

It is used as shown below:

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

Note that an alias can also be defined as shown here:

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

Example

Find out the sector for a section (rubrique):

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

The "Job Queue" plugin manages a list of scheduled tasks, so we can find out the date of the next task to be performed with this code:

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

sql_get_charset

The `sql_get_charset()` function is used to check if the usage of the particular character encoding is available on the database server.

`sql_get_charset()` accepts three parameters, with only the first being mandatory:

1. `$charset` is the charset being requested, such as "utf8"
2. `$serveur`,
3. `$options`.

sql_get_select

The `sql_get_select()` function returns the query for the requested selection. This is an alias for the `sql_select()` function but which passes the `$option` argument set to `false`, so that the SQL query is returned rather than being executed.

It accepts the same arguments as `sql_select()` except for the last, which is provided by the function:

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

It is applied as shown in this example:

```
$request = sql_get_select('column', 'table');
// returns "SELECT column FROM table" (for a MySQL database)
```

This function therefore returns a SQL query which is valid for the database manager in use. As this query is clean, it can be directly used by the `sql_query()` function, but more often than not, it is used to create more complex queries in conjunction with `sql_in()` :

```
// list of identifiers
$ids = sql_get_select('id_table', 'tableA');
// selection based on that prior selection
$results = sql_select('titre', 'tableB', sql_in('id_table',
$ids)));
```

Example

To find out the titles of the sections which have article identifiers greater than 200, one of the possible methods (we could also use a join) is to use `sql_get_select()`:

```
// create the selection query to find the list of
sections
$ids = sql_get_select('DISTINCT(id_rubrique)',
'spip_articles', array('id_article > 200'));
// select the titles of those sections
$res = sql_select('titre', 'spip_rubriques',
sql_in('id_rubrique', $ids));
while ($r = sql_fetch($res)) {
    // display each title.
    echo $r['titre'] . '<br />';
}
```

Considerably more complicated, we could search for examples in certain criteria functions, for example with the `{noeud}` criteria of the "SPIP Bonux" plugin which creates a sub-query to retrieve the list of objects which have child records.

```
function critere_noeud_dist($idb, &$boucles, $crit) {
// [...]
// this construction with IN will make the compiler
request
// the use of the sql_in() functions
$where = array("'IN'", "'$boucle->id_table." .
"$primary'", "'('.sql_get_select('$id_parent',
'$table_sql').')'");
if ($crit->not)
    $where = array("'NOT'", $where);
$boucle->where[]= $where;
}
```

sql_hex

The `sql_hex()` function returns a numeric value for a hexadecimal expression, transforming `09af` into `0x09af` (for MySQL and SQLite). This is principally used to write hexadecimal content into a numerically-typed SQL column.

It accepts 3 parameters:

1. `$val` is the character string to be translated,
2. `$serveur`,
3. `$option`.

Usage:

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

sql_in

The `sql_in()` function is used to create a column condition using the `IN` SQL keyword.

It employs 5 parameters:

1. `$val` is the name of the column,
2. `$valeurs` is the list of values, in the form of an array or a comma-separated sequence of strings. These values will be automatically filtered using `sql_quote`,
3. `$not` is used to provide negation. By default it is empty `''`; assign `'NOT'` to execute a `NOT IN` condition,
4. `$serveur`,
5. `$option`.

It can be used as follows:

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

Example

The "Tickets" plugin uses `sql_in()` to obtain the title of a ticket only if it has a status matching one of those listed:

```
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

The `sql_insert()` function is used to insert content into a database. The SQL ports may experience problems when using this function, and if so, they should use the `sql_insertq()` function instead. This function is described here only to ensure support for restoring old backups and for transitioning old scripts.

The function accepts 6 parameters:

1. `$table` is the SQL table,
2. `$noms` is the list of columns affected,
3. `$valeurs` is the list of values to be stored,
4. `$desc`,
5. `$serveur`,
6. `$option`.

Usage example:

```
sql_insert('table', '(column)', '(value)');
```

Example

Insert a link to a keyword for an article:

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

Example of migrating to `sql_insertq()`:

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

sql_insertq

The `sql_insertq()` function is used to perform a record insert into the database. Non-numeric values will be filtered using functions modified for each database manager in order to correctly handle apostrophes. When possible, the function returns the identifying number for the inserted primary key.

The function accepts 5 parameters:

1. `$table` is the name of the SQL table,
2. `$couples` is an array table of (name / value) pairs,
3. `$desc`,
4. `$serveur`,
5. `$option`.

It is used as shown below:

```
$id = sql_insertq('table', array('column1'=>'value1',
'column2'=>'value2'));
```

Example

The `insert_xx()` functions like `insert_article()` described in ecrire/action/editer_article.php are used to create database inserts for the objects in question, by managing the default values and calling the pre_insertion (p.172) appropriate pipeline. These functions return the identifier of the created record.

These functions therefore run the `sql_insertq()` function after the `pre_insertion` pipeline. If an author is identifiable during the process, then the article is linked to that author:

```
$id_article = sql_insertq("spip_articles", $champs);
// check that the server doesn't return an error
if ($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

The `sql_insertq_multi()` function is used to insert, in one single action, several elements with identical schemas into a database table. If the database manager port allows it, it will then use a single SQL command to implement the insert. More specifically, a single SQL command for each batch of 100 elements is used in order to avoid memory congestion.

The function has the same 5 parameters as `sql_insertq()` , but the second parameter for this function is a table of a table of pairs, and not just directly a table of pairs:

1. `$table` is the name of the SQL table,
2. `$couples` is a table of associative tables of name / value pairs,
3. `$desc`,
4. `$serveur`,
5. `$option`.

The columns used in this command absolutely must be the same set for all of the inserts. The command is used as shown below:

```
$id = sql_insertq_multi('table', array(
    array('column' => 'value'),
    array('column' => 'value2'),
    array('column' => 'value3'),
);
```

Example

Searches made using SPIP use the `spip_resultats` table to store some elements used as a cache, by taking care to use the table for the SQL connection. `$tab_couples` contains all of the data to be inserted:

```
// insert the results into the results cache table
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);
}
```

The "Polyhierarchie" plugin also uses this function for inserting the list of sections just recently linked to a given object:

```
$ins = array();
foreach($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

The function `sql_in_select()` returns a `sql_in` from the result of a `sql_select`.

It accepts the same arguments as `sql_select` plus one additional parameter in first place:

1. `$in` is the name of the column on which the `IN` will be applied,
2. `$select`,
3. `$from`,
4. `$where`,
5. `$groupby`,
6. `$orderby`,
7. `$limit`,
8. `$having`,
9. `$serveur`,
10. `$option`.

You can use it like this:

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

Example

Delete every link between an article and the keywords of a given keyword group:

```
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

The `sql_listdbs()` function lists the various databases that are available for a particular connection. It returns a selection resource or directly a PHP array of the various databases (as is the case for SQLite).

It accepts 2 parameters:

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

SPIP uses this function during the installation routine to permit the selection, when one can be made, of which database to use from those authorised by the database manager.

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

sql_multi

The `sql_multi()` function applies an SQL expression to a column that contains a multi-lingual expression (p.0) (`<multi>`) in order to extract from it the portion corresponding to a nominated language. It returns a character string typed as: `expression AS multi`. This operation is essentially used to simultaneously request a sort on this column.

It accepts 4 parameters:

1. `$sel` is the name of the column,
2. `$lang` is the language code (’fr’, ’es’, ...),
3. `$serveur`,
4. `$option`

It is used as shown below:

```
$multi = sql_multi('column', 'language');
```

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

Note that in a template file, the loop criteria `{par multi xx}` where `xx` is the name of the column to be sorted, will also call this function in order to sort according to the current language.

Example

SPIP uses this function to sort the lists according to the title of an element and according to the site visitor nominated language:

```
$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']
    }
}
```

In similar fashion, the "Grappes" plugin uses it here:

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

sql_optimize

The `sql_optimize()` function is used to optimise an SQL table. This function is called by the `optimiser_base_une_table()` function which is periodically called by the cron mechanism. Please refer to the `OPTIMIZE TABLE` or `VACUUM` commands for the appropriate SQL database manager to understand the details of what is being executed by these commands.

The function accepts 3 parameters:

1. `$table` is the name of the table to be optimised,
2. `$serveur`,
3. `$option`.

Usage:

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

Note: SQLite can not optimise one table at a time, but optimises the entire database in one hit. In this case, if the `sql_optimize()` function is called multiple times in a row, then the operation will actually only be performed just once for the first call.

sql_query

The `sql_query()` function executes the query passed to it as a parameter. It is the least portable of the SQL instruction command set; it should therefore be avoided wherever possible in preference to the other more specific SQL API functions.

It accepts 3 parameters:

1. `$ins` is the SQL query,
2. `$serveur`,
3. `$option`.

Usage:

```
$res = sql_query('SELECT * FROM spip_meta');
// but we would prefer you used this instead:
$res = sql_select('*', 'spip_meta');
```

sql_quote

The `sql_quote()` function is used to secure or filter data content (with apostrophes) in order to avoid SQL injection attacks. This function is very important and must be used whenever content is provided by user data entry. The `sql_insertq`, `sql_updateq`, and `sql_replace` functions automatically apply this filtering for any inserted data (but not for the other parameters like `$where` which ought to be filtered nonetheless anyway).

It accepts 3 parameters:

1. `$val` is the expression to be filtered,
2. `$serveur`,
3. `$type` optional, is the type of value expected. This would equal `int` for an integer value.

It is used as shown below:

```
$charstring = sql_quote("David's car");
$fieldname = sql_quote($fieldname);
sql_select('column', 'table', 'titre=' . sql_quote($titre));
sql_updateq('table', array('column'=>'value'), 'titre=' .
sql_quote($titre));
```

Whenever a numeric identifier is expected, which is often the case for primary keys, the filtering may simply apply the PHP `intval()` function (the value zero will be returned if the content passed is not numeric):

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

Example

The `url_delete()` function deletes URLs from the SQL table that stores the URLs for SPIP editorial objects. It filters the strings using `sql_quote()` and uses `intval()` on the identifier:

```
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

The `sql_repair()` function is used to repair a damaged SQL table. It is called by SPIP when an administrator attempts to repair a database using the `ecrire/?exec=admin_tech` page.

It accepts 3 parameters:

1. `$table` is the table which is requested to be repaired,
2. `$serveur`,
3. `$option`.

Usage:

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

Note: PostGres and SQLite database managers ignore this instruction.

sql_replace

The `sql_replace()` function inserts or updates a record in an SQL table. The primary key(s) must exist amongst the inserted data. The function automatically secures the data.

It is recommended to use the specific `sql_insertq()` and `sql_updateq()` instead of this function to be more precise, at least where such is possible.

Its 5 parameters are:

1. `$table` is the SQL table in question,
2. `$couples` contains the column/value pairs to be modified,

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

It is used as shown below:

```
sql_replace('table', array(
    'column' => 'value',
    'id_table' => $id
));
```

sql_replace_multi

The `sql_replace_multi()` function is used to insert or replace several rows (which have the same schema) for an SQL table in a single operation. The values are automatically filtered against SQL injection attacks. It is necessary that the columns of inserted pairs contain the primary key(s) for that table.

It is recommended to use the specific functions `sql_insertq_multi()` and `sql_updateq()` instead of this function to be more precise, at least where such is possible.

It has the same 5 parameters as sql_replace (p.291) :

1. `$table` is the SQL table in question,
2. `$couples` is a table of column/value pairs to be modified,
3. `$desc`,
4. `$serveur`,
5. `$option`.

It is used as shown below:

```
sql_replace_multi('table', array(
    array(
        'column' => 'value1',
        'id_table' => $id1
    ),
    array(
        'column' => 'value2',
        'id_table' => $id2
    )
));
```

sql_seek

The `sql_seek()` function positions a selection resource originating from a `sql_select()` at the designated row number.

It accepts 4 parameters:

1. `$res`, the resource,
2. `$row_number`, the row number,
3. `$serveur`,
4. `$option`.

It is used as shown below:

```
if ($res = sql_select('column', 'table')) {
    if (sql_seek($res, 9)) { // go to number 10
        $r = sql_fetch($res);
        // $r['column'] of the 10th result
    }
    // return back to the start
    sql_seek($res, 0);
}
```

sql_select

The `sql_select()` function selects content form the database and returns an SQL resource when successful or `false` in the event of an error.

It accepts up to 9 parameters, the first 2 being mandatory, and sequenced in the same descriptive order as a standard SQL query. Each parameter will (preferably) accept an array as input data, but will also accept character strings with elements separated by commas:

1. `$select`,
2. `$from`,
3. `$where`,
4. `$groupby`,
5. `$orderby`,

6. `$limit`,
7. `$having`,
8. `$serveur`,
9. `$option`.

The `sql_select()` function is often coupled with an `sql_fetch()`, such as shown here below:

```
// selection
if ($resultats = sql_select('column', 'table')) {
    // loop on the results
    while ($res = sql_fetch($resultats)) {
        // use the results
        // $res['column']
    }
}
```

The `$select` and `$from` parameters accept the declaration of aliases. This offers the following type of construction:

```
if ($r = sql_select(
    array(
        'a.column AS colA',
        'b.column AS colB',
        'SUM(b.number) AS btotal'
    ),
    array(
        'tableA AS a',
        'tableB AS b'
    ))) {
    while ($ligne = sql_fetch($r)) {
        // we now have access to:
        // $ligne['colA']  $ligne['colB']  $ligne['btotal']
    }
}
```

Example

Select the root sections (id_parent=0) in the "spip_rubriques" table sorted by rank [1 (p.296)], then in alphanumeric order, and request all of the columns (total selection with '*') :

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

Select cats but not dogs (in the title) for articles in sector 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);
```

Select the titles and extensions recognised for documents, and store the result in a table:

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

This selection could also be written as:

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

Select the documents linked to a section, with the title of the section in question, and sort in reverse date order:

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

[1 (p.0)] Maybe one of these days there will be a genuinely dedicated column for this!

sql_selectdb

The `sql_selectdb()` function is used to select a connection to a database server that offers a database for use. The function returns `true` of the operation is successful, otherwise it returns `false`.

The `sql_selectdb()` function has 3 parameters:

1. `$nom` being the name of the database to use,
2. `$serveur`,
3. `$option`.

This function is used by SPIP during the installation routine to try to pre-select the name of the database to be used, by means of attempting to select a database with the same name as the login.

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

sql_serveur

The `sql_serveur()` function is used to both connect to the database server if that has not yet already been done, and to obtain the real name of the function that will be executed for a requested transaction. This function is called transparently by means of aliases. It is therefore normally not a useful operation to employ it directly.

`sql_serveur()` accepts three parameters, with only the first being critical:

1. `$ins_sql` is the name of the function requested from amongst the list of functions that the API understands, such as "select", "update", "updateq"... When left deliberately empty, it is then simply requesting that a connection be made to the database server if such has not already been done.
2. `$serveur`,
3. `$continue` defines what should happen whenever the SQL API instruction is not found by the requested database manager. Set by default to `false`, the system returns a fatal error, but it is possible to continue programme execution by setting this parameter’s value to be `true`.

This function is typically used as below:

```
// calculate the function name
$f = sql_serveur('select');
// execution of the function as per the determined API
$f($arg1, $arg2, ... );
```

If you are requesting the `select` instruction in the instruction set determined for MySQL and existing in the ecrire/req/mysql.php file, then the `$f` variable will equal `spip_mysql_select`. Correlation between the instructions and the function is defined in that same file with a global variable: `spip_mysql_functions_1` (MySQL is the type of server, 1 is the version of the instruction set).

Using aliases to make things simple
Practically all of the `sql_*` API functions are aliases which that calculate a function using `sql_serveur` and then execute it. In this way, calling the `sql_select` function performs (more or less) exactly the same operation as the previous code. It is these instructions that ought to be used:

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

sql_set_charset

The `sql_set_charset()` function requests the usage of the specified encoding for transactions between PHP and the database manager.

`sql_set_charset()` accepts three parameters. Only the first is required:

1. `$charset` is the requested charset, such as "utf8"
2. `$serveur`,
3. `$options`.

This function is called immediately after each connection to the database server in order to specify the charset to be employed. This encoding selection is defined elsewhere in the `charset_sql_connexion` meta variable created during the installation of SPIP.

sql_showbase

The `sql_showbase()` function is used to obtain a resource that can be used with `sql_fetch()` detailing the tables that exist in the database.

It accepts 3 parameters:

1. `$spip` empty by default, the parameter is used to list only the tables using the prefix defined for SPIP tables. Use `'%'` instead if you want to list ALL tables,
2. `$serveur`,
3. `$option`.

Usage:

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

The sql_alltable (p.261) function is generally easier to use, since it directly returns a PHP array listing the various database tables.

sql_showtable

The `sql_showtable()` functions returns a description of an SQL table in an associative array that lists the columns and their SQL "field" descriptions and also listing the keys. Whenever a join declaration exists for the table declared in `tables_principales` or `tables_auxiliaires`, the array will also include an entry for the "join" key.

Its parameters are:

1. `$table` is the name of the table to investigate,
2. `$table_spip` is used to automatically replace "spip" by the table's real prefix; it equals `false` by default,
3. `$serveur`,
4. `$option`

Usage:

```
$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"
```

In most situations, it would be better to use the trouver_table (p.113) function, which has a cache on the data structure, use the `sql_showtable()` function and add some supplementary information.

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

sql_update

The `sql_update()` function updates one or several records in an SQL table. The elements passed are not automatically filtered against SQL injection attacks as with `sql_updateq()`, so you must watch out for SQL injection attacks and use `sql_quote()` functions to secure the content when necessary.

The function accepts 6 parameters:

1. `$table` is the SQL table in question,
2. `$exp` contains the modifications to be made,
3. `$where`,
4. `$desc`,
5. `$serveur`,
6. `$option`.

This function is principally used to modify values which use the same value as the column being updated, e.g.

```
// increment the column by 1
sql_update('table', array('column' => 'column + 1'));
```

Whenever data added with this function are likely to include apostrophes or originate from user data entry, it is important to secure the insert with the use of the `sql_quote()` function:

```
sql_update('table', array('column' => sql_quote($value)));
```

Example

Update the "id_secteur" column with the identifier for sections that don’t have a parent:

```
// assign the id_secteur value for root sections
sql_update('spip_rubriques',
array('id_secteur'=>'id_rubrique'), "id_parent=0");
```

Add a set number of visits to the statistical data for certain articles:

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

sql_updateq

The `sql_updateq()` function is used to update content in an SQL table. The content passed to the function is automatically filtered.

Its 6 arguments are the same as for `sql_update()`:

1. `$table` is the SQL table in question,
2. `$exp` contains the modifications to be made,
3. `$where`,
4. `$desc`,
5. `$serveur`,
6. `$option`.

It is used as shown below:

```
sql_updateq('table', array('column' => $value), 'id_table=' .
intval($id_table));
```

Example

The `modifier_contenu()` function in ecrire/inc/modifier.php is called when an editorial object is modified, and takes care of calling the `pre_edition` and `post_edition` pipelines, using the `sql_updateq()` function to update the collected data:

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

sql_version

The `sql_version()` function simply returns the version number of the database manager.

It accepts 2 optional parameters:

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

Usage:

```
$x = sql_version();
echo $x;
// depending on the type of server, we might see:
// for MySQL:   5.1.37-1ubuntu5.1
// for SQLite2:  2.8.17
// for SQLite3:  3.6.16
```

Creating your own plugins

Plugins are a convenient way to add extensions to SPIP. They usually come as a compressed folder (in ZIP format) and have to be extracted in the "plugins" directory (to be created if need be) or to be installed directly by entering the compressed file’s URL in the plugins administration page in the private area.

The basic principle of plugins

Plugins add features or functions to SPIP, which might be a set of standardised template files, a modification of existing functionality, creation of new editable database objects,...

They have the advantage of enabling the management of tasks to be carried out when they are installed or uninstalled, activated or deactivated. They can also handle interdependencies with other plugins.

All of the SPIP folders and elements that can be overloaded can be recreated in the folder of a plugin, in the same fashion as is done in your own private "squelettes" folder. The essential difference is the existence of an XML file which describes the plugin, uniformly named `plugin.xml`.

The minimal plugin.xml

The `plugin.xml` file must be created in the root directory of your plugin. It contains the description of the plugin and allows it to define certain actions.

The minimum content of the plugin file might be as follows (non-ASCII characters are "escaped"):

```
<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>
```

These attributes are easy to understand, but are described below nonetheless:

- `nom`: name of the plugin,
- `auteur`: author(s) of the plugin,
- `licence`: license(s) for the plugin,

- `version`: version of the plugin. This detail is displayed in the private area when requesting information about the plugin, and it also serves for handling dependencies between plugins, when coupled with the prefix. Another attribute not to be confused with this one is the ’version_base’ which is used when the plugin creates tables or fields in the database,
- `description`: pretty obvious! But note that this description and the plugin name are often written using multilingual "idioms" as text placeholders that will be replaced by the appropriate string for the current language, defined in the "lang" files for the plugin.
- `etat`: the state of development of the plugin, perhaps "dev" (in development), "test" (under testing) or stable
- `prefix`: a unique prefix distinguishing this plugin from any others. No numerals are permitted here, and it must be in lower case.

plugin.xml, other common attributes

Options and functions

The files for the options and functions provided by the plugin are declared directly within the `plugin.xml` file using the `options` and `fonctions` attributes:

```
    <options>porte_plume_options.php</options>
    <fonctions>inc/barre_outils.php</fonctions>
    <fonctions>autre_fichier.php</fonctions>
```

Several function files may be loaded if necessary by listing them in succession.

Documentation link

The `lien` attribute is used to provide an address for documentation about the plugin:

```
    <lien>http://documentation.magraine.net/-Porte-
Plume-</lien>
```

Plugin icon

The `icon` attribute is used to specify an image to be used to visually represent the plugin:

```
        <icon>imgs/logo-bugs.png</icon>
```

Handling dependencies

Plugins can indicate if they depend upon certain conditions in order for them to work correctly. Two attributes are used to specify this: `necessite` and `utilise`. In the first case, the dependency is a strong one: a plugin that requires a resource (a certain version of SPIP or another particular plugin) can not be activated if that resource is not present and active. An error will be generated if we try to activate the plugin without that dependency being fulfilled. In the second case, the dependency is weak, and the plugin can be activated and perhaps even work even if that dependency is not fulfilled.

Necessite

```
<necessite id="prefixe" version="[version_min;version_max]"
/>
```

- `id` is the name of the plugin's prefix, or "SPIP" for a direct dependency on SPIP itself,
- `version` used optionally can indicate the mininum and/or maximum version of a plugin. Square brackets are used to indicate that the version as specified is included, parentheses to indicate that the version specified is not included.

Utilise

"Utilise" is therefore used to declare optional dependencies, with exactly the same syntax as for `necessite`.

`utilise` and `necessite` also therefore make it possible to override the files for the plugin that they refer to (as they have priority in the file path).

Example

```
// requires at least SPIP 2.0
<necessite id="SPIP" version="[2.0;)" />
// requires SPIP < 2.0
<necessite id="SPIP" version="[;2.0)" />
// requires SPIP >= 2.0, and <= 2.1
<necessite id="SPIP" version="[2.0;2.1]" />
```

```
// requires spip_bonux at least at version 1.2
<necessite id="spip_bonux" version="[1.2;]" />
```

Certain plugins may indicate that it is possible to modify their configurations if the CFG plugin is loaded (but without actually being an indispensable requirement for the plugin to work):

```
// configuration plugin
<utilise id="cfg" version="[1.10.5;]" />
```

Installing external libraries

Plpugins may also require external libraries that they are dependent upon to be downloaded. This requires several things: a specific declaration in the `plugin.xml` file, and the existence of a `/lib` directory that is write accessible in the SPIP root directory, into which the library will be automatically (or manually) loaded.

```
<necessite id="lib:nom" src="address of the zip file" />
```

- nom specifies the name of the zip’s uncompressed folder
- `src` is the address of the library archive in zip format

Example

A plugin called "loupe photo"" uses a javascript library that it installs as a library (therefore outside of the plugin itself) in this manner:

```
<necessite id="lib:tjpzoom" src="http://valid.tjp.hu/
tjpzoom/tjpzoom.zip" />
```

In the plugin, the names of the files that the plugin uses are listed like this:

```
$tjp = find_in_path('lib/tjpzoom/tjpzoom.js');
```

The "Open ID" plugin also uses a library that is external to the plugin. It loads it in the following manner:

```
<necessite id="lib:php-openid-2.1.2"
src="http://openidenabled.com/files/php-openid/packages/
php-openid-2.1.2.zip" />
```

And then uses that library as below:

```
// options
if (!defined('_DIR_LIB')) define('_DIR_LIB', _DIR_RACINE
. 'lib/');
define('_DIR_OPENID_LIB', _DIR_LIB . 'php-
openid-2.1.2/');
// usage (somewhat more complicated!)
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);
    // ...
}
```

Using pipelines

To use the pipelines of SPIP or of a plugin, their usage must be explicitly defined in the `plugin.xml` file:

```
<pipeline>
    <nom>name_of_the_pipeline</nom>
    <action>name of the function to load</action>
    <inclure>directory/file.php</inclure>
</pipeline>
```

The `action` parameter is optional, and by default, it has the same name as the pipeline. This declaration indicates a particular file to load when calling the pipeline (determined by the `inclure`) and loading a function like `pluginprefix_action()`. Note that the `action` parameter is only rarely provided.

Several pipelines can be specified by listing them as demonstrated below:

```
<pipeline>
    <nom>name_of_the_pipeline</nom>
    <inclure>directory/file.php</inclure>
</pipeline>
<pipeline>
    <nom>another_name</nom>
    <inclure>directory/file.php</inclure>
</pipeline>
```

Example

The pipeline insert_head (p.164) adds content into the `<head>` section of published pages. The "Messagerie" plugin (using "messagerie" as a prefix) uses it for adding CSS styles:

```
<pipeline>
    <nom>insert_head</nom>
    <inclure>messagerie_pipelines.php</inclure>
</pipeline>
```

And in the `messagerie_pipelines.php` file:

```
function messagerie_insert_head($texte){
    $texte .= '<link rel="stylesheet" type="text/css"
href="'.find_in_path('habillage/messagerie.css').'"
media="all" />'."\n";
    return $texte;
}
```

Defining buttons

To add buttons into the private zone, all that is needed is to provide a `bouton` attribute in the `plugin.xml` files as follows:

```
<bouton id="identifier" parent="name of parent identifier">
    <icone>icon path</icone>
    <titre>title language description</titre>
    <url>name of the exec</url>
    <args>arguments passed</args>
</bouton>
```

Description:

- `id` holds the unique identifier of the button, which is used (amongst other things) by sub-menus to indicate the name of their parent button. Quite often, the name of the `exec` file (used to display the page) is the same as the identifier name,
- `parent`: optional, used to specify that the button is a sub element of a parent button. It therefore stores the identifier of the parent button. Absent any value, it is a top level element that will be created (alongside the "Launch pad" and "Site edit" buttons),
- `icone`: also optional, to specify the icon path,
- `titre`: button text, may also be a placeholder "plugin:placeholdername",
- `url` specifies the exec file name that is loaded when you click on the button. If not indicated, it will be the identifier name that is used.
- `args`, optional, used to pass arguments to the URL (example: `<args>criteria=start</args>`).

Authorisations

The buttons are displayed by default for all persons connecting to the private zone. To change this configuration, specific authorisations must be created for the buttons (and then use the authorisation pipeline to load the new plugin authorisations):

```
function autoriser_identifiant_bouton_dist($faire, $type,
$id, $qui, $opt) {
    return true; // or false
}
```

Example

Statistics for SPIP 2.1 – currently under development – will be in a separate plugin. At present, it reproduces the buttons as below:

```
<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>
```

The authorisations are defined in a specific file:

```
<?php
function stats_autoriser(){}
// View the stats ? = all admins
function autoriser_voirstats_dist($faire, $type, $id,
$qui, $opt) {
    return (($GLOBALS['meta']["activer_statistiques"] !=
'non')
            AND ($qui['statut'] == '0minirezo'));
}
// Button authorisation
```

```
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);
}
?>
```

Defining page tabs

Declaring the tabs for the `exec` pages in the private zone follows exactly the same syntax as for the buttons. The name of the parent, however, is mandatory here and corresponds to a parameter passed in the call function for the tab in the `exec` file:

```
<onglet id='identifier' parent='tab bar identifier'>
    <icone>icon_path</icone>
    <titre>placeholder title</titre>
    <url>exec filename</url>
    <args>arguments</args>
</onglet>
```

As for the buttons, if the URL is not provided, then the identifier name is used as the name of the file to be loaded.

Authorisations

Again as with the buttons, an authorisation is used to manage whether the tab is displayed or not.

```
function autoriser_identifiant_onglet_dist($faire, $type,
$id, $qui, $opt) {
    return true; // or false
}
```

Example

The "Champs Extras 2" (Extra fields v2) plugin adds a tab into the configuration page, into the toolbar quite appropriately named "configuration". Here are the declarations used in its `plugin.xml` file:

```
<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>
```

Authorisations are defined in the `inc/iextras_autoriser.php` file. The tab only displays if the author is declared as a "webmestre (webmaster)".

```
<?php
if (!defined("_ECRIRE_INC_VERSION")) return;
// function for the pipeline, nothing to do
function iextras_autoriser(){}
// authorisation declarations
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);
}
?>
```

Finally, in the `exec/iextras.php` file, the toolbar is called as shown below. The first is the identifier of the requested toolbar, the second is the identifier of the current tab.

```
echo barre_onglets("configuration", "iextras");
```

Examples

A chapter to present a few practical examples of small scripts.

Adding a type of glossary

It is possible to add links to external glossaries from SPIP using the `[?nom]` shortcut. By default, the links are made to wikipedia. To create a new glossary link, there is the `[?nom#typeNN]` syntax available.

- type is a word for the glossary
- NN is an optional numeric identifier.

A simple function called `glossaire_type()` is used to return a particular URL. 2 parameters are passed: the text and the identifier.

Example:

A link to the trac source files for SPIP 2.1:

```
<?php
@define('_URL_BROWSER_TRAC', 'http://trac.rezo.net/trac/spip/
browser/branches/spip-2.1/');
/*
 * A link pointing to trac files
 * [?ecrire/inc_version.php#trac]
 * [?ecrire/inc_version.php#tracNNN] // NNN = line number
 */
function glossaire_trac($texte, $id=0) {
    return _URL_BROWSER_TRAC . $texte . ($id ? '#L'.$id :
'');
}
?>
```

Applying a default sort sequence to the loops

It is possible to sort the output of loops using the `{par}` criteria. The template for the documentation you are currently reading has the same sorting criteria of `{par num titre, titre}` for all of its `ARTICLES` et `RUBRIQUES` loops.

Rather than repeat this in the code for all of the loops, we can apply it just once for all the loops if there is no other sorting criteria specified for a given loop. To do this, we use the `pre_boucle` pipeline and insert an `ORDER BY` for the SQL select queries.

Plugin.xml:

```
<pipeline>
    <nom>pre_boucle</nom>
    <inclure>documentation_pipelines.php</inclure>
</pipeline>
```

documentation_pipelines.php:

```
function documentation_pre_boucle($boucle){
    // ARTICLES, SECTIONS : {par num titre, titre}
    if (in_array($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;
}
```

Doing this means that the loops are sorted by default:

```
// auto sort {par num titre, titre} :
<BOUCLE_a1(ARTICLES){id_rubrique}>...
// different sort:
<BOUCLE_a2(ARTICLES){id_rubrique}{!par date}>...
```

A few details

The pipeline receives a "boucle" (loop) type PHP object that may have various values. The loop notably has some `select` and `order` variables which handle what will be entered into the `SELECT` and `ORDER BY` clauses of the generated SQL query. The SQL table name (`spip_articles` or `spip_rubriques` in the current case) is stored in `$boucle->id_table`.

When we assign a number within the titles of SPIP articles (which do not have any `ranking` field in their tables even though the code has already been envisaged to handle it!), we write it like this: "10. Title" (number point space Title). In order for SQL to be able to easily sort by number, all that is needed is to force a numerical evaluation of the field (which is then converted into a number). This is why the code "0+titre AS autonum", which creates an alias column called autonum holding this numeric calculation value in it, is then able to be used as a sort column in the ORDER BY clause.

Consideration of new fields in table searches

If you have created a new field in one of the SPIP tables, it will not be considered by default by the search functions. It must also be declared explicitly for that to occur. The rechercher_liste_des_champs (p.122) pipeline has what you need called from the ecrire/inc/rechercher.php file.

It accepts a parameter table listing table/field = coefficient couples, where the coefficient is a number specifying the number of points to assign for a successful search in that field on that table. The higher the coefficient, the more points that field will credit to a total score for any searches that match that field's contents.

Example

You have a field "town" in the SQL table "spip_articles" that you would like to include in searches - it must be declared as an additional field in the pipeline:

```
function
pluginprefix_rechercher_liste_des_champs($tables){
    $tables['article']['town'] = 3;
    return $tables;
}
```

Display an authoring form, if authorised

There are special #AUTORISER tags that make it possible to manage access to certain content and/or certain forms on a fine-grained scale. As shown below, if the visitor has the rights to modify the article, then a form can be displayed to edit that article, which, once validated, will return to the article page in question:

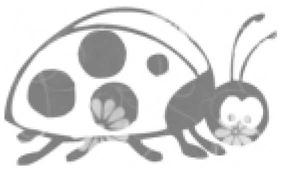
```
[(#AUTORISER{modifier, article, #ID_ARTICLE})
 #FORMULAIRE_EDITER_ARTICLE{#ID_ARTICLE, #ID_RUBRIQUE,
#URL_ARTICLE}
]
```

Modifying all of your templates in one hit

Thanks to some special hooks, it is possible to use a single simple operation on a complete set of template files to modify the behaviour of a particular loop or type of loop, just by using the pre_boucle (p.171) pipeline. For example, every `RUBRIQUES` loop, regardless of which template file it is stored in, can have sector 8 omitted from its search criteria:

```
$GLOBALS['spip_pipeline']['pre_boucle'] .= '|hide_a_sector';
function hide_a_sector($boucle){
  if ($boucle->type_requete == 'rubriques') {
    $secteur = $boucle->id_table . '.id_secteur';
    $boucle->where[] = array("'!='", "'$secteur'", "8");
  }
  return $boucle;
}
```

Note that the plugin "Accès Restreint" also offers this function to restrict access to specific content.



Glossary

Definitions of some of the key technical terms used in the documentation.

AJAX

The term AJAX, an acronym for "*Asynchronous JavaScript and XML*", is used to describe a collection of technologies used to create asynchronous client-server interactions.

These constructions, which make it possible to only request a partial page update from the server (or partial element update), can significantly reduce the data volumes that need to be transmitted and often make an application appear more responsive to its users.

Argument

In programming, the term "argument" is used for content passed when making a function or procedure call. Functions can use several arguments. Arguments can be the results of other calculations. We differentiate "arguments" (the input data) from "parameters" (what the function receives). In PHP we have:

```
function_name('argument', $argument, ...);
function_name($x + 4, $y * 2); // 2 calculated arguments are
sent.
```

And in SPIP, for tags and filters:

```
#TAG{argument, argument, ...}
[(#TAG|filter{argument, argument})]
```

Cache files

A cache is a store of files that is used to accelerate data access. There are caches used internally in almost every part of a computer: in the microprocessors, on hard drives, in software, in PHP functions, etc. They make it possible for a given piece of data to be retrieved or calculated faster in the event that it is requested more than just a single time, whether it be a highly volatile storage system (like RAM memory), or a more permanent resource (like a hard drive).

A cache often has a limited life span, as for example, the time that it takes for a programme to run, or the time required to process a PHP function call. A validity period can also be assigned when the storage device delivers data that is more persistent - a web page can thereby tell a browser programme for how many hours a page will remain valid if that page is being held in the browser's local cache.

Parameter

The "parameters" of a function, that is, what is received when the function is called, are described in the declaration of that function. This declaration may specify the type of value expected (integer, table, character string…), a default value, and most importantly the name of the variable where the usable parameter is stored within the function's code. In PHP we have:

```
function name($param1, $param2=0){}
```

This "name" function will receive two "parameters" when it is called, stored in the local variables `$param1` and `$param2` (which will have a value of 0 by default). We may then call this particular function with either 1 or 2 "arguments":

```
name('Extra'); // param2 will equal 0
name('Extra', 19);
```

Pipeline

The term pipeline is used within SPIP in the UNIX sense of the word. The pipeline executes a series of functions for which the result of one such function is used as input for the next. In this way, each function in a pipeline can use the data that are passed to it, modify them, use them, and return them. The results then act as arguments for the next function, and the next and so on until the last such function.

When calling a pipeline, the first function is very often passed data, or at least a default value. The results of the chaining of the various functions is then used or displayed depending on the situation at hand.

Certain particular calls on pipelines in SPIP are to be considered as triggers, in the sense that they simply declare an event, but do not expect any result to be returned from the various functions that the pipeline will call. Most of these triggers have a name that uses the prefix `trig_`.

Recursion

In programming, we use the term "recursion" for an algorithm (some computer code) that is able to call itself. We also speak of "self-referencing". PHP functions can call themselves recursively, like the example below which adds up the first X integers (just as an example, as this can be computed faster with `x*(x+1)/2`).

```
// calculation of : x + (x-1) + ... + 3 + 2 + 1
function sum($x) {
    if ($x <= 0) return 0;
    return $x + sum($x-1);
}
// call it
$s = sum(8);
```

SPIP also allows you to write recursive loops (p.19) within the templates.

Index

Table of contents