Programming with SPIP

DOCUMENTATION TO BE USED BY DEVELOPERS AND WEBMASTERS

SPIP's version 2.0.10+ / 10% translated / November 2009

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

Introduction

Introduction to SPIP and presentation of its general principles

What is SPIP?

SPIP 2.0 is a free software package distributed under the licence GNU/GPL3. 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 be done with SPIP?

SPIP is particularly suitable for publishing portals, but it can also be used for a blog, wiki, or social network. More generally, SPIP can manage any data stored in MySQL, Postgres or SQLite databases. Extensions also offer interaction with XML.

Requirements and basic description

SPIP 2.0 requires PHP version 4.1 or higher and at least 10MB of memory. It also requires a database (MySQL, Postgres or SQLite are accepted).

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

The templates

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

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 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 transforms these templates into PHP code which it stores in a cache. These cached files are then used to produce the HTML pages — which may also be cached — and then returned to each visitor.

Quick overview

SPIP transforms templates into static pages. Templates are mainly composed of loops (`<BOUCLE>`) which select elements, and tags (`#TAG`) which display the properties and values of those elements.

List of 5 last 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 example, this `<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 for its page, and the `#TITRE` tag is replaced with its title.

This loop displays the following results:

- Secured actions (p.38)
- How secured actions work (p.38)
- Secured actions' predefined functions (p.39)
- Action URLs in a template (p.40)
- Contextual pipelines (p.28)

Snippets and Ajax

Templates can include other templates, more or less independently.

Here is a snippet that lists the latest articles five at a time and displays a block of pagination controls.

File `models/list_last_articles.html`:

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

You can re-use this template anywhere with the `<INCLURE>` markup. For the pagination to be effective, it requires at least the `{env}` parameter that contains the contextual variables.

This snippet can be reloaded using AJAX by adding the parameter `{ajax}` to the inclusion. In this case, only the block element, and not the whole page, will be modified when the visitor clicks on links with the CSS class name `ajax` or links included within DOM elements with the class `pagination`.

```
<INCLURE{fond=models/list_last_articles}{env}{ajax} />
```

Result: An AJAX pagination, from 5 to 5...

- Secured actions (p.38)
- How secured actions work (p.38)
- Secured actions' predefined functions (p.39)
- Action URLs in a template (p.40)
- Contextual pipelines (p.28)

0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 |...

Secure and extensible

Adapt all templates in a single operation

Thanks to SPIP's "pipelines", it is possible and easy to change the behaviour of a particular type of loop in every template.

Example: For each RUBRIQUES loop, regardless of the template, hide the sector 8:

```
$GLOBALS['spip_pipeline']['pre_boucle'] .= '|hide_a_sector';
function hide_a_sector($loop){
  if ($loop->type_requete == 'rubriques') {
    $sector = $loop->id_table . '.id_secteur';
    $loop->where[] = array("'!='", "'$sector'", "8");
  }
  return $loop;
}
```

Show only if it is allowed

The special tag #AUTORISER can restrict access to some content or forms.

Example: If the visitor is authorised to change the current article, the code below displays a form to edit it. When the form is submitted, it will return the user to the article's page. The [] and () markup is a simple example of SPIP's if-then-else syntax for dynamic code inclusion.

```
[(#AUTORISER{modifier, article, #ID_ARTICLE})
 #FORMULAIRE_EDITER_ARTICLE{#ID_ARTICLE, #ID_RUBRIQUE,
#URL_ARTICLE}
]
```

The templates

SPIP generates `HTML` pages from `templates`, containing a mixture of `HTML`, `loops` and `criteria`, `tags` and `filters`. Its strength is the ability to extract database content using a simple and understandable language.

Loops

A `loop` displays some content stored in database. It generates an optimized SQL query that extracts the desired content.

The syntax of loops

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

```
<BOUCLE_nom(TABLE){critere}{crirere}>
 ... for each object ...
</BOUCLE_nom>
```

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. Here, the name is “_nom”.

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

The next components of a loop are the criteria, which are written between 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 the images on the site. It draws its data from the database using the `DOCUMENTS` alias, and the criterion `{extension IN jpg,png,gif}` selects only those files with a filename extension in the given list.

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

The tag `#FICHIER` contains the address of the document, which is modified with a filter named “image_reduire”. This will resize the image to be at most 300 pixels and return an HTML `<img>` tag for the new image.

The complete syntax of loops

Loops, like tags, have a syntax allowing of multiple compositions. Optional parts are displayed only once (not for each element) and only if the loop returns some content. An alternative part is displayed only if the loop does not return any content. Here is the syntax (x is the name of the loop):

```
<Bx>
    Display once, before the loop content
<BOUCLEx(TABLE){critère}>
    For each element
</BOUCLEx>
    Display once, after the loop content
</Bx>
    Display if there is no result
<//Bx>
```

Example:

This loop selects the five latest articles published on the site. Here, `<ul>` and `</ul>` HTML tags will be displayed once, if the loop criteria match some elements. If there are no matching elements, then these optional parts will not be displayed.

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

Result:

- Secured actions, *20 November 2009*
- How secured actions work, *20 November 2009*
- Secured actions' predefined functions, *20 November 2009*
- Action URLs in a template, *20 November 2009*
- Contextual pipelines, *20 November 2009*

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.

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 we usually call "sectors":

```
<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}. The latter instructs SPIP to select the articles belonging to the current section — in this case, the one chosen by the RUBRIQUES loop.

- en
- fr
    - Notes sur cette documentation

Recursive loops

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.

In programming, an algorithm (a data-processing code) which calls itself 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 transmitting the required arguments. Therefore, inside the loop (x), the same loop (x) is called with different parameters. This is what we call recursion. This process will be repeated as long as the recursive loop returns results.

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

Example:

We can use a recursive loop to display a list of all of the sections in the site. For that, we loop for the first time on the sections, with a criterion to select the sub-sections of the current section: `{id_parent}`. We also sort by number (given to the sections so that we can display them in a particular order), and then by title.

```
<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. They have an id_parent field of zero. When the first section is displayed, the recursive loop is called and SPIP calls the loop “_rubs” again. This time the `{id_parent}` criterion selects different sections because it lists the sub-sections of the current section. If there are sub-sections, the first is displayed. Then the “_rubs” loop is called again, but in *this* sub-section. As long as there are sub-sections to display, this recursive process starts again.

Result:

- en
    - Introduction
    - The templates
        - Loops
    - Extending SPIP
        - Introduction
        - Pipelines
        - Some pipelines
        - The functions you should know
    - Access control
- fr
    - Introduction
    - Écriture des squelettes
        - Boucles
        - Balises
        - Des balises à connaître
        - Critères de boucles
        - Filtres de balises
        - Inclusions
        - Ajax
        - Éléments linguistiques
        - Multilinguisme
        - Liaisons entre tables (jointures)
        - Accéder à plusieurs bases de données
    - Étendre SPIP
        - Généralités
        - Les pipelines
        - Liste des pipelines
        - Des fonctions à connaître
    - Les différents répertoires
        - Action
        - Auth
        - Balise
        - Exec
        - Genie
    - Gestion d’autorisations
    - Compilation des squelettes
    - Formulaires
        - Structure HTML
        - Traitements PHP

- Accès SQL
- Développer des plugins
- Exemples
- Glossaire

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 use a table of a plug-in which might not be 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 ?){criteres}>
   ...
</BOUCLE>
```

Example:

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

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

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 web site, and to create new functionality.

This part explains the ways that programmers can extend SPIP.

Introduction

Templates, plug-ins, access paths, functions called `_dist()`... This section explains it all.

Templates or plug-ins?

Use the "squelettes" folder

The `squelettes/` folder allows to store all the files required for the operation of your site and of 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.

Then, plug-in or simple `squelettes` folder?

Generally, we will use the `squelettes/` folder 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>`.

As these files of options are loaded with each request, they must 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'
$squelettes = array(
  '2008'=>'squelettes/2008',
  '2007'=>'squelettes/2007',
);
// If a particular set of template are requested (and exist),
set a cookie, otherwise delete the cookie
if (isset($_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.22) — 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, 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.24) — 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”), `boucle`s (“loops”), and `critere`s (“criteria”) are named like this and can thus be overridden by declaring (perhaps in the file `mes_fonctions.php`) the same function, 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) {
    // ...
}
```

Pipelines

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

Definition

Pipelines are provided by SPIP to allow your plugin to ‘hook into’ SPIP code; that is, to call functions in your plugin at specific moments, and thereby set your plugin in motion.

The declaration file `plugin.xml` of the plugin must contain these lines :

```
<cadre class="xml">
<prefix>pluginPrefix</prefix>
<pipeline>
  <nom>pipelineName</nom>
  <inclure>cfg_pipeline.php</inclure>
</pipeline>
</cadre>
```

- <nom> : the pipeline name,
- <inclure> : indicates the file that declares the function to execute when the pipeline is triggered (this function is always named thus : `pluginPrefix_pipelineName()`).

The file `config/mes_options.php` (and other "XX_options" files (p.22)) also allows code execution after a pipeline, with this line :

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

List of current pipelines

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

There are several types of pipelines: some of them deal with typographical modifications, others deal with databases modifications, or pages display in the private area, etc.

Declaring a new pipeline

The pipeline must be declared in an option file like this :
`$GLOBALS['spip_pipeline']['newPipelineName'] = '';`
The name of this pipeline must be a key of the associative array `$GLOBALS['spip_pipeline']`.

Then, you can use it in a template or a PHP file :

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

The tag `#PIPELINE` and the function `pipeline()` 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 declaring this pipeline is part of 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 treatment that has been done.

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
    })]})]
```

Some pipelines

This part describes the use of some of SPIP’s pipelines.

affiche_milieu

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

You call it like this:

```
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 page of the sections to select some 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 the pages of SPIP’s configuration.

```
function stats_affiche_milieu($flux){
    // displays the configuration ([un]activate the
statistics).
```

```
    if ($flux['args']['exec'] == 'config_fonctions') {
        $compteur = charger_fonction('compteur',
'configuration');
        $flux['data'] .= $compteur();
    }
    return $flux;
}
```

declarer_tables_auxiliaires

This pipeline declares the «auxiliary» tables, which are mainly used to create joints between main tables.

It receives the same arguments than the pipeline declarer_tables_principales (p.0).

Example :

The plugin "SPIP Bisous" enables an author to send a *poke* to another author. It declares a table spip_bisous linking 2 members to the poke's date:

```
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;
}
```

In this case, the primary key is composed of two fields.

declarer_tables_objets_surnoms

This pipeline creates a relationship between an object type and its corresponding SQL table. By default, a ’s’ is added to the end of the object type name.

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 plugin "jeux" uses:

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

jquery_plugins

This pipeline makes it easy to add some Javascript code wich will be inserted into every pages.

It receives and returns an array that contains the path [1 (p.33)] of the files to insert :

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

Example :

Add the script "jquery.cycle.js" to every pages :

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

[1 (p.0)] This path will be completed by the function find_in_path() (p.36)

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 issues from the 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 the section number 4)
- ext = 'html' (default extension of the 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 branchs 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));
            // comparaison 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;
}
```

The functions you should know

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

find_in_path

The function `find_in_path()` returns the path of a function. This function is searched for in the "SPIP path" (p.24).

It takes 1 or 2 arguments :

- name or relative path of a file (with its extension)
- eventually, the directory where it is stored.

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

Example :

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

If a file `pattern/inc-special.html` exists, $html is the result of the compilation of this template. Else $html is the result of the compilation of `pattern/inc-normal.html`.

Access control

Two essential elements make it possible to manage access to the actions and pages displayed from SPIP: the authorisations with the `fonction autoriser()` function, and actions secured by author with the `fonction securiser_action()` function.

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. In fact, they make is possible to create URLs for links or forms which pass a particular 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()` funciton, 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>
]
```

Index

Table of contents