Development guide

Create your own templates with Bluo CMS

Introduction

In Bluo the content is separated from the graphics. The graphics of your site are contained in files called templates. Bluo comes with 5 customizable templates and by reading this document you could easily add many more.

Steps in creating a template

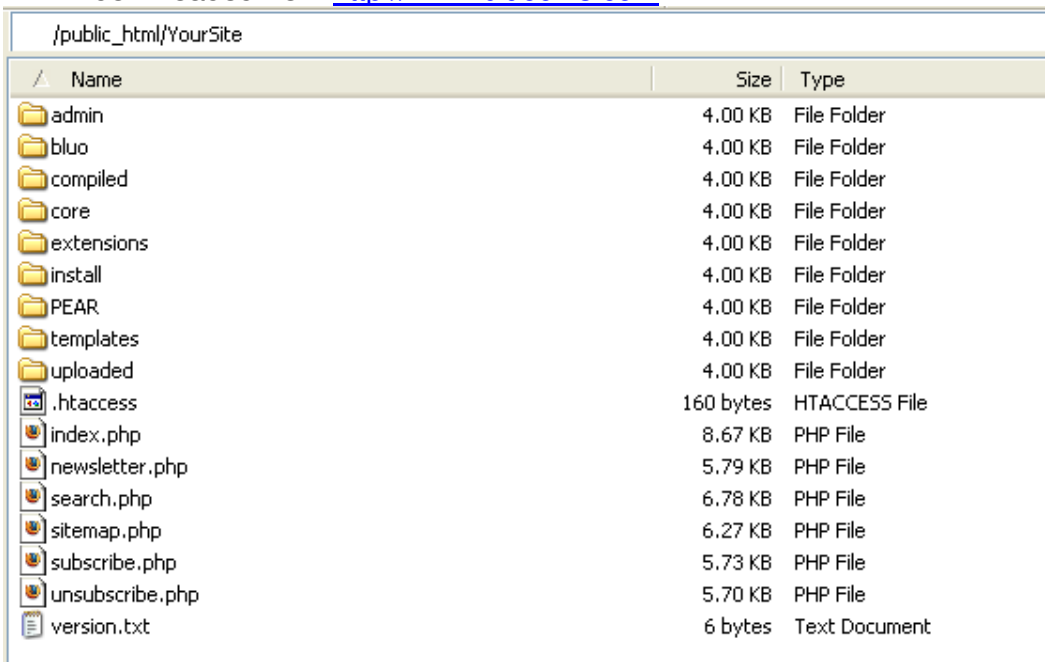
To build up a Bluo template, you should follow these steps:

1. The first thing you need is a design.

2. Once you like the design you have (or the design you create), you will be able to build up the html file. We highly recommend you that during building it up, you write all the classes in a CSS file.

3. Once the two files are ready: the html file and the CSS file, you can start creating the template itself.

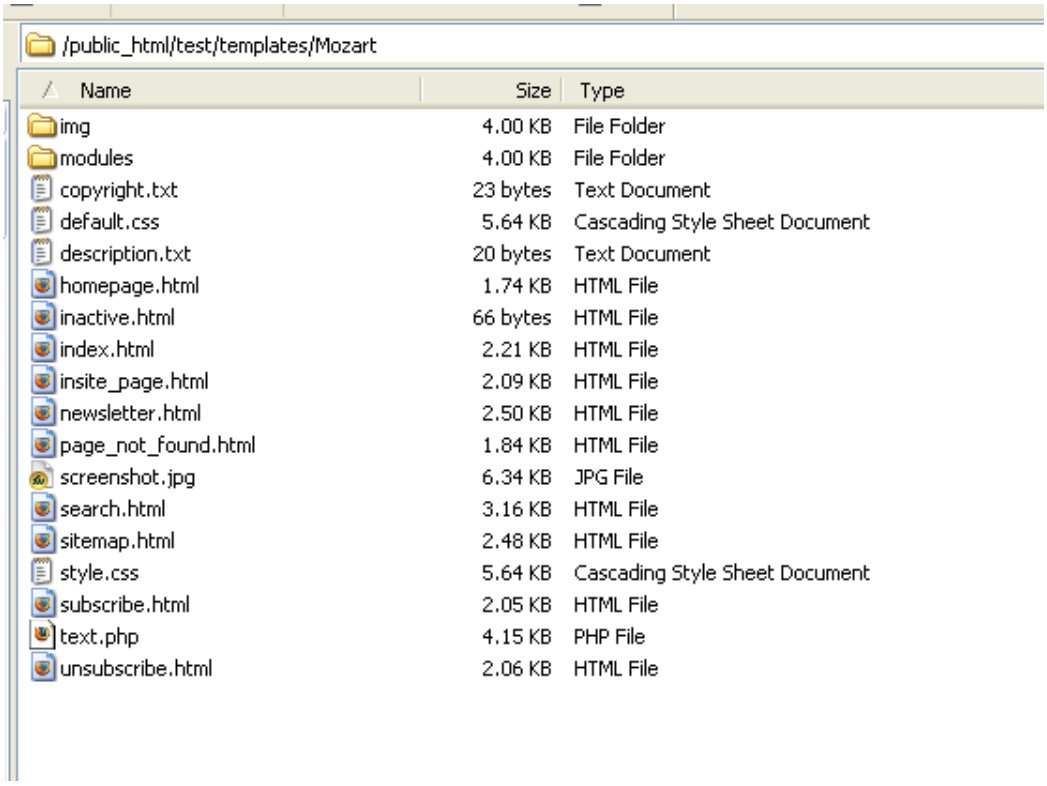
4. The directory containing your site[1] must have the structure in the image below. Otherwise, extract one more time the archive you have downloaded from http://www.bluocms.com [2]



From the files you see in the image above, you are interested in the file called *templates*. This is actually the file which contains all the templates available for your version of Bluo.

5. This second step consists in creating a new directory in the directory templates, whose name will also be given to your template. This directory should have the following structure of files and directories:

[1] If you do not have Bluo, get on http://www.bluocms.com and download the version you are interested in.
[2] If you do not have it anymore you must download Bluo again from http://www.bluocms.com



6. After having made sure that the directory containing the name of your template has the above depicted configuration, you have to copy the images included in the html file you created in the *img* directory. After that, you have to copy the content from the file containing the styles in *style.css*.

7. You write the content of the *copyright.txt* and *description.txt* files and you replace the screenshot.jpg image with the image you want to be assigned to your template. Make sure that the name of this image remains *screenshot.jpg*[1].

8. You can now begin creating the template by filling in the html files. The templates: *homepage.html*, *index.html*, *inactive.html*, *insite_page*.html and *page_not_found.html* are compulsory. The rest of the templates are not necessary unless your site imposes it. Within their content you can use a series of modules[2] which will considerably ease up your work.

9. After having created these templates you can modify or not the style.css file, as mentioned in the CSS chapter.

[1] It is important that the extension be .jpg
[2] I will present to you these modules in the chapter specially dedicated to them.

10. Only after being sure that the *style.css* file contains the default[1] styles for your site, can you fill in the *default.cs*s file with the content from the *style.css*.

11. You open the administration area and select from *Settings -> Templates* your template in order to visualize it.

12. You check whether the template is well done by visualizing the front end[2]. You make correct the problems of formatting, if there are any.

13. THAT'S IT!

[1] You can obtain the custom styles as a result of the modifications in the admin area in the Settings -> Design options section.
[2] This checking can be performed beginning from step 8.

Files

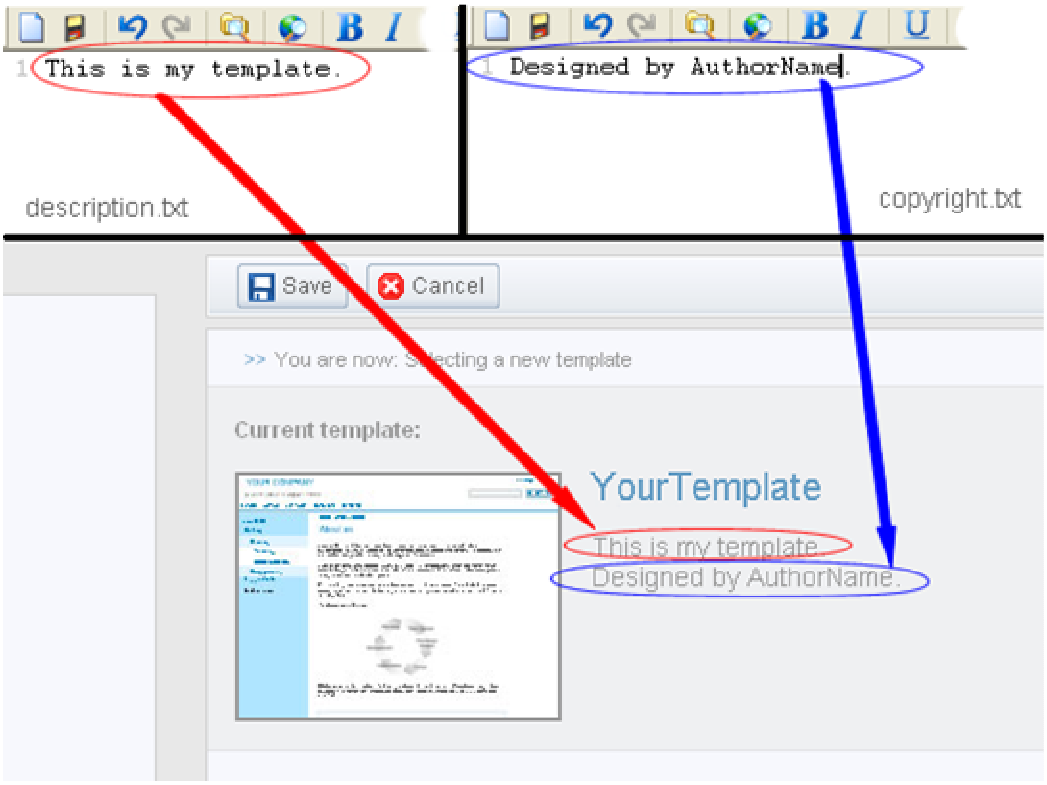
Observation: The templates: *homepage.html*, *index.html*, *inactive.html*, *insite_page*.html and *page_not_found.html* are compulsory. The rest of the templates are not necessary unless your site imposes it.

Let me show you what the meaning of each directory/file is:

- **Img** – is the directory where all the images appearing on your site are stocked, except for those images which appear in the content section of the page of a page (see bellow for more details about the structure of a template). These last ones are being stocked in the *uploaded* file which appears in the image at step 4. Also, all the images uploaded from the administration area using the editor are saved in this folder.

- **Modules** – in this directory you will find all the templates needed for the *RenderHTML*, *RenderBreadcrumb* and *RenderMenu* modules (more on the modules bellow). I will tell you more on the content of this directory at the chapter dedicated to explaining the modules of Bluo.

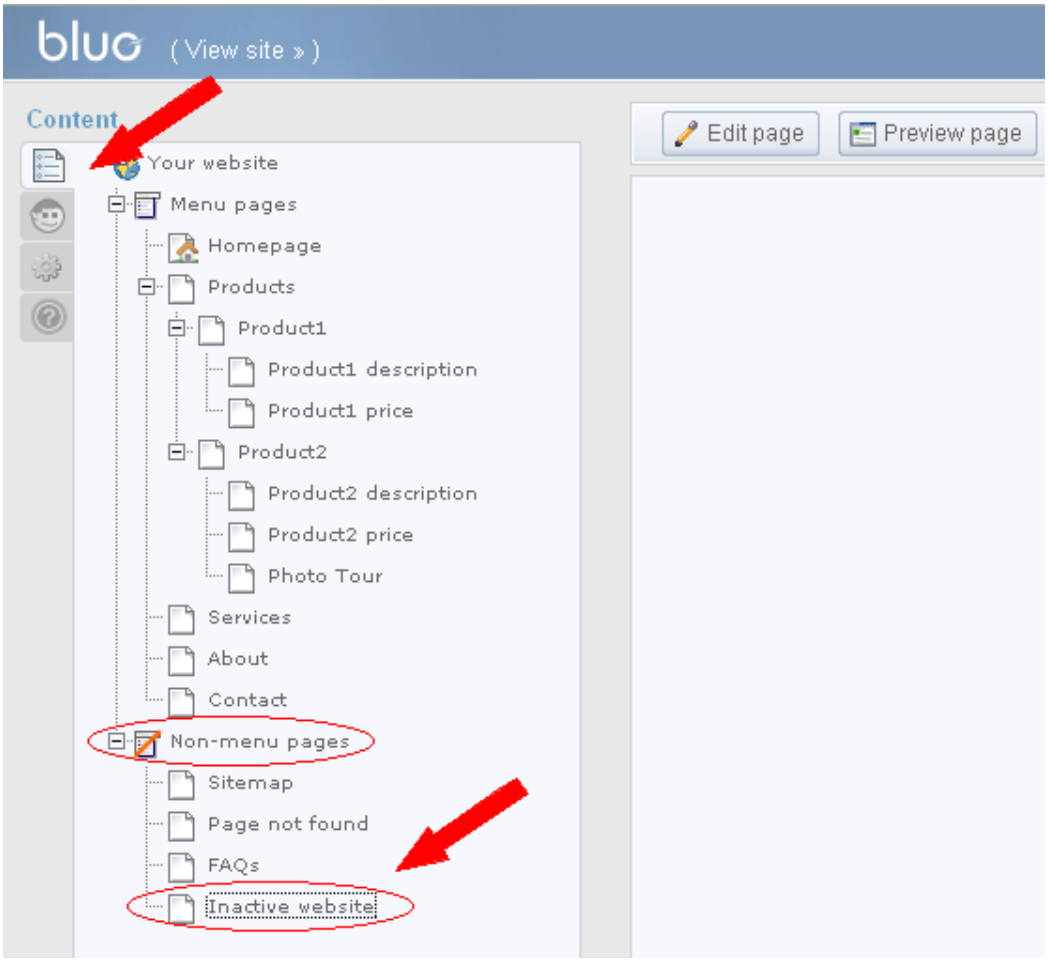
- **copyright.txt** and **description.txt** are two complementary files. *Description.txt* contains the description of the template, and *copyright.txt* contains the name of template creator. The content of these two files appears in the administration area, in the *Settings> Templates* section.



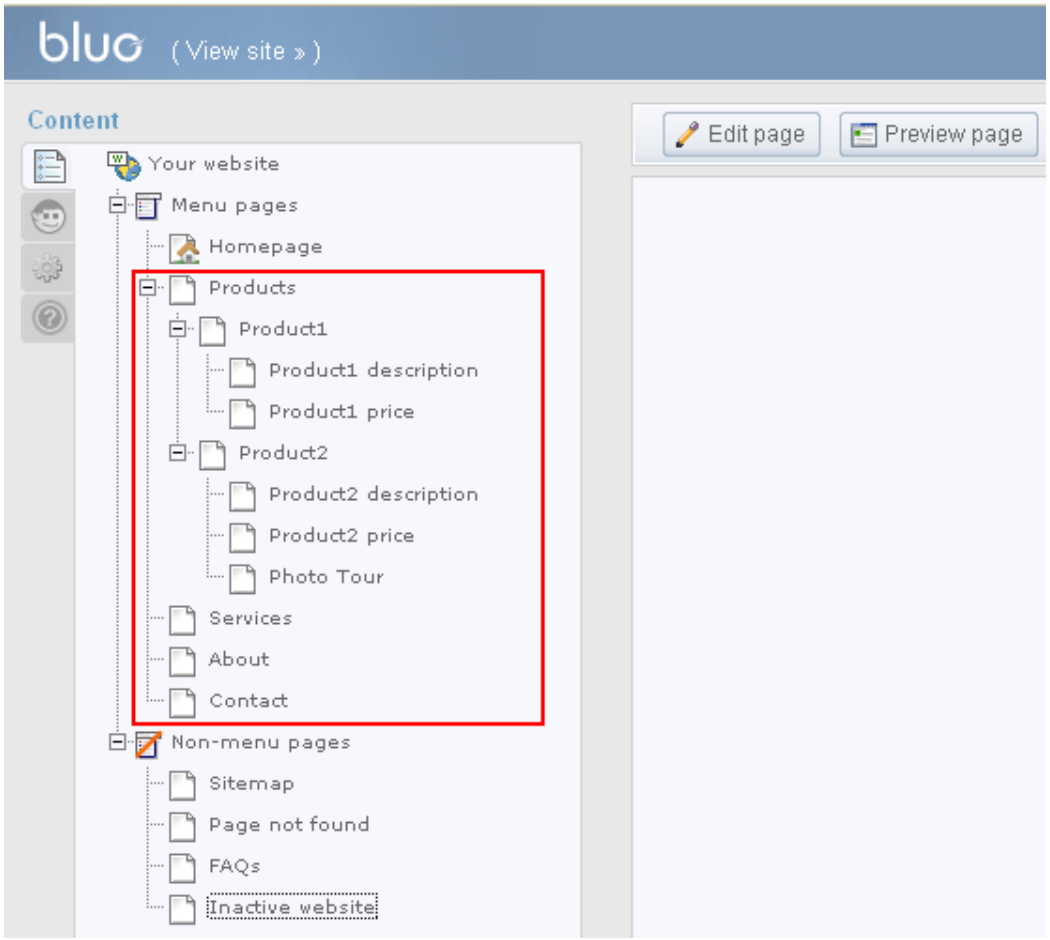
- **default.css** - is the file containing the default stiles for your template. This means that when you click the *Restore* to default button in section *Setting> Design* option in the administration area, the content of the *style.css* file will automatically be replaced by the content in *default.css*. You must not mistake *default.css* with *style.css*. We will talk about this at the right moment.

- **homepage.html** – is the template for the starting page of your website.

- **Inactive.html** – is the template for the Inactive Website page, which will be posted in case you deactivate your website. You can find the Inactive Website page in the tree in the administration area, at *Non-Menu Pages* section.



- **Index.html** – is the template for all the pages existing in the Menu Pages section, except for Homepage. Check out the image bellow to see which the pages for which the index.html template applies are.

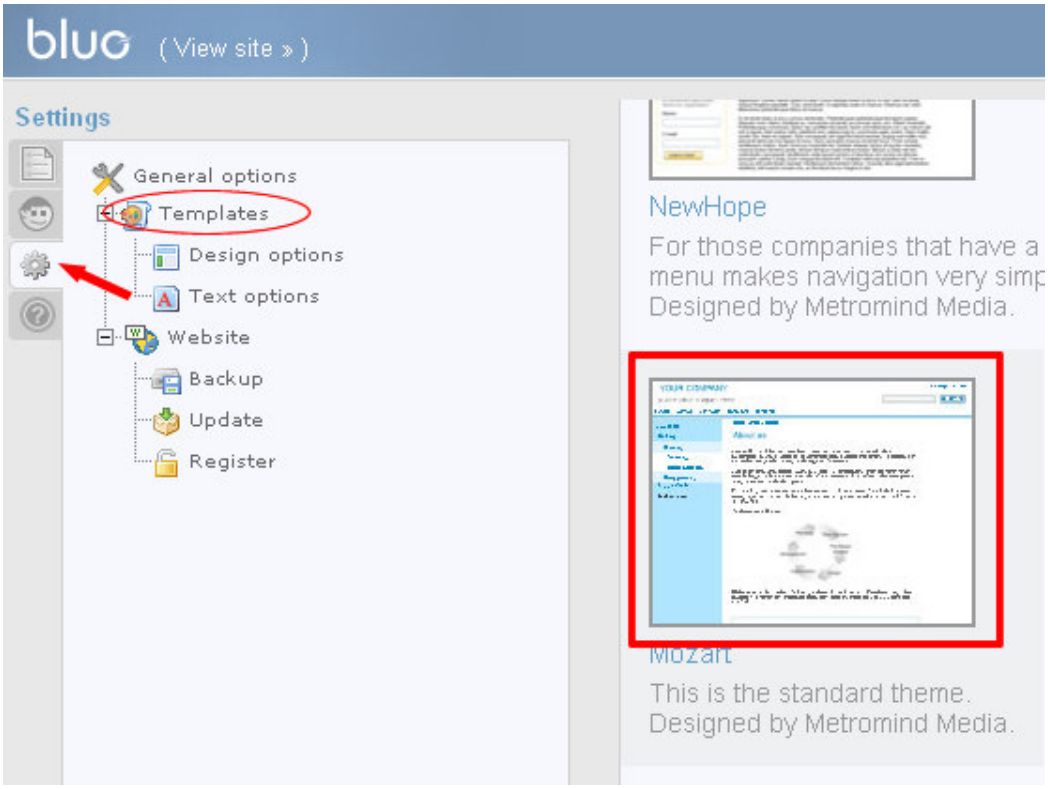


- **Insite_page.html** is the template available for all the pages in the Non-menu pages section, pages which are different from Inactive Website, *Page not found* and *Sitemap*. For these last pages you can have different templates (see bellow)

- **newsletter.html** is the template which is being used the moment a visitor of your site enters his contact data to the newsletter box.

- **page_not_found.html** is the template for the *Page not found* page, which you can find in the administration area at the *Non-menu pages* section.

- **screenshot.jpg** is the image designated to your template; it will appear in the administration area in the *Settings->Templates*. For example, for the Mozart template , the image from *screenshot.jpg* is framed in the red square you can see below:



- **search.html** is the template for the Search Results page which posts the search results.

- **sitemap.html** is the template for the Sitemap page appearing in the Non-menu pages section.

- **style.css** is the file where you can find the styles for your template. I will tell you more about the content of this file in the chapter dedicated to it

- **subscribe.html** is the template which is being posted the moment a visitor of your site confirms his subscription to the newsletter.

- **text.php** is a file containing two arrays: *text* and *help*, which you will fill in according to the prescriptions presented in the *RenderText* module

- **unsubscribe.html** is the template for the page appearing when a visitor of your site unsubscribe from the newsletter.

Flexy

Flexy is a PEAR scripting platform above PHP. We will use it to create our templates.

Here you are some notions about flexy that will be useful to you:

`{variableName}` – Writes the value of the variable `variableName`
`{variableName:h}` – Writes the value of the variable `variableName` in html code

`{nameFunction(param1,param2,…)}` – Runs the nameFunction function with the parameters param1, …

`{runModule(#numeModul#,#param1#,#param2#,…..)}` – Runs the nameModule module with the parameters param1, param2, …

The if structure

```
{if:variable}
    HTML_Code_Branch_If
{else:}
    HTML_Code_Branch_Else
{end:}

{if:functionBoolean()}
    HTML_Code_Branch_If
{else:}
    HTML_Code_Branch_Else
{end:}
```

The foreach struture

```
{foreach:nameArray,key,value}
    HTML_Code
{end:}
```

```
<tr flexy:foreach=" nameArray,key,value "> <!—insertinf
the foreach structure within an HTML tag -->
```

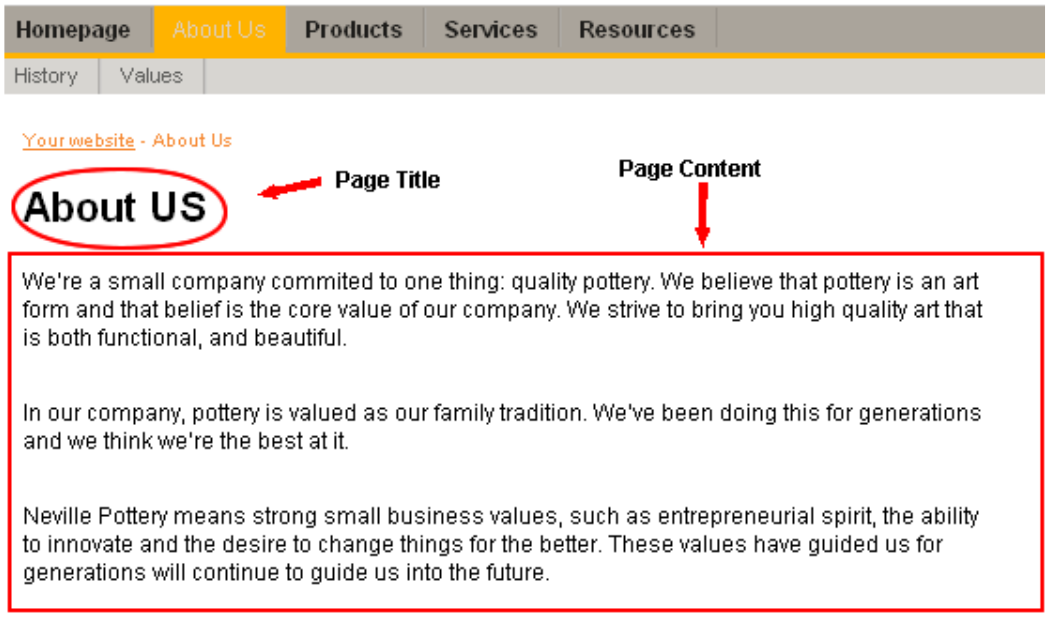
More info: http://pear.php.net/package/HTML_Template_Flexy/

Template structure

We are interested in the way the structural parts of a template look in front end and in the way those components are generated.

We can distingue the follow sections of a Bluo template:

- Sections included using modules (more in the following chapter)
- Page content that the user can insert using the admin area
- Page title, also introduced in the admin area.



The way you put title and content section in your template is very important. You can use 2 variables for that:

pageContent – this is the variable that holds the HTML content edited in the admin area.

You can use in the HTML part of your template like this:

```
{pageContent:h}
```

pageTitle – this is the variable that holds the page title of the page

You can use in the HTML part of your template like this:

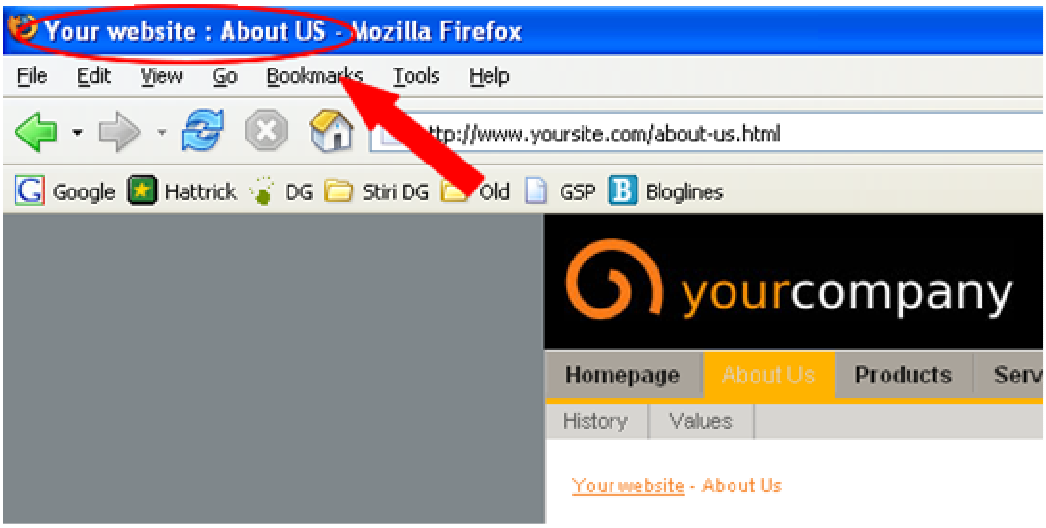
```
{pageTitle}
```

```
<td id="siteBorder">
    <div id="top">
        {runModule(#RenderMenu#,#false#,#1#,#1#,#all#,#TopMenu.html#)}
        {runModule(#RenderHTML#,#Header.html#)}
    </div>
    <div class="titleBox">
        <div class="leftTD">
            <h1>{pageTitle}</h1>
        </div>
    </div>
    <div class="topContent">
        {pageContent:h}
    </div>

    <div id="bottom"></div>
</td>
>

{runModule(#RenderMenu#,#true#,#1#,#2#,#all#,#BottomMenu.html#)}
```

Let me also tell you about variable that introduces page title in browser title bar.



It is called *head*.

head – its value is the code contained *<head>* and *</head>* tags. It contains all the *<meta>* tags, the *<title>* tag and also a reference to *style.css* file.

You can use it this way:

```
{head:h}
```

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "ht
2 <html xmlns="http://www.w3.org/1999/xhtml">
3 <head>
4 <!-- HEAD -->
5 {htmlHead:h}
6 <!-- /HEAD -->
7 </head>
8
```

Here's how the variable looks like for a template:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtm
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<!-- HEAD -->
        <title>Your website : About US</title>

        <meta name="description" content="">
        <meta name="keywords" content="">
        <meta name="author" content="">
        <meta name="rating" content="General">
        <meta name="expires" content="never">

        <meta name="language" content="english">
        <meta name="charset" content="ISO-8859-1">
        <meta name="robots" content="all">
        <meta name="spiders" content="all">
        <meta name="revisit-after" content="7 Days">
        <meta name="email" content="contact@yoursite.ro">
        <meta name="authors" content="yoursite.com">
        <meta name="copyright" content="Copyright 2006 - yoursite.com">
        <meta name="distribution" CONTENT="global">

        <meta content="PHP" name="CODE_LANGUAGE">
        <meta content="JavaScript" name="vs_defaultClientScript">
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">

        <link href="http://www.metromind.ro/test/templates/BeautifulDawn/style.css"
                rel="stylesheet" type="text/css">
<!-- /HEAD -->
</head>
```

We will speak about the other modules in the following chapters.

Modules

Modules are functional units that allow you to add functionality to the HTML files of a template. For example, in order to create a dynamic menu you could recall a special module. Modules are created using Flexy and HTML, and are recalled in your templates using `runModule`() function.

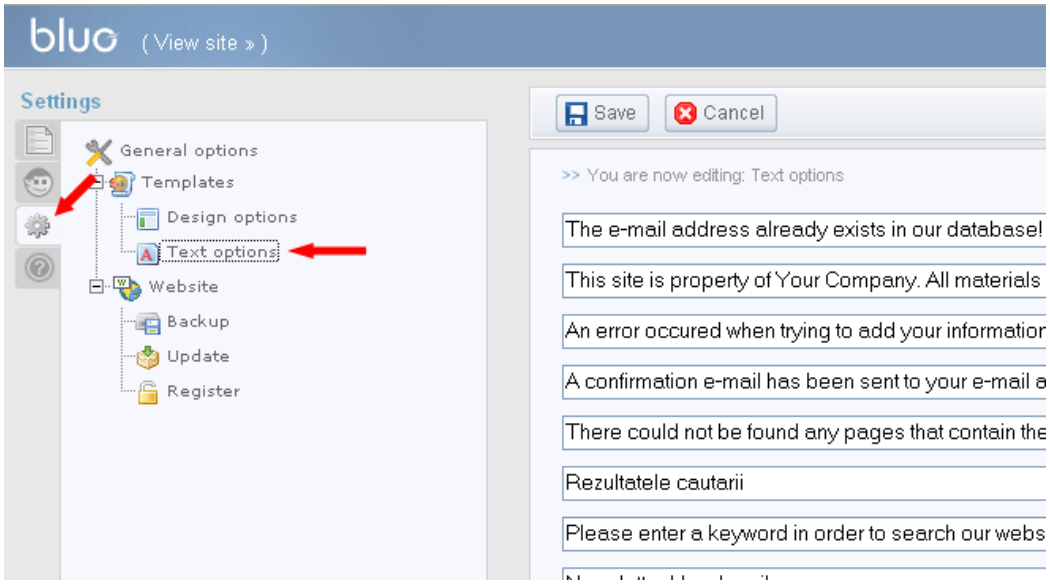
RenderText

Role

You can use the *RenderText* module if you want to insert within the page some texts different from those you can change from the Content section in the administration area, such as Page Name, Page Title etc.

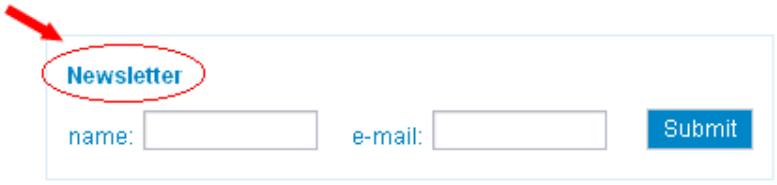
Using this module, you can modify the inserted text from the *Settings->Text options* section in the admin area.

If you did not manage yet to identify this section, the next image will certainly be of help to you.



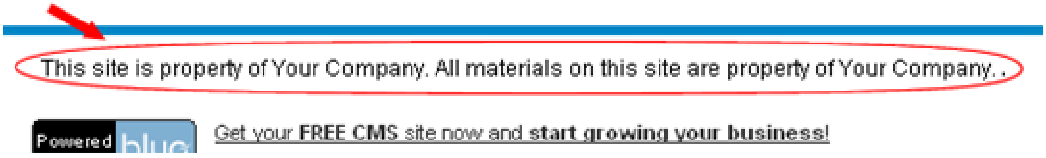
Let's take an example of a text which will prove the usefulness of the *Render Text* module:

Suppose that there appears a section of the newsletter within a page of your template, and you have to assign a name to it. Let's call it Newsletter.



Nevertheless, after some time, you might want to name it Newsletter Subscription. Unless you use the *RenderText* module to insert the name, you will have to modify all the template pages where this text appears. But if you insert the text using this module, your work will be considerably reduced, and all you will have to do will be a few clicks in the admin area.

Another example of a text which could appear on your site and for which I recommend you to use the *RenderText* module to insert it, is the text appearing in the footer, where you mention issues relates to copyright and the ownership of the site.



Running procedure

All you need to know in order to insert a text sequence using the *RenderText* is how it runs:

```
{runModule(#RenderText#,#TextName#)}
```

This code line has to be inserted in the template page right on the spot of the text that you want to make editable from the admin area.

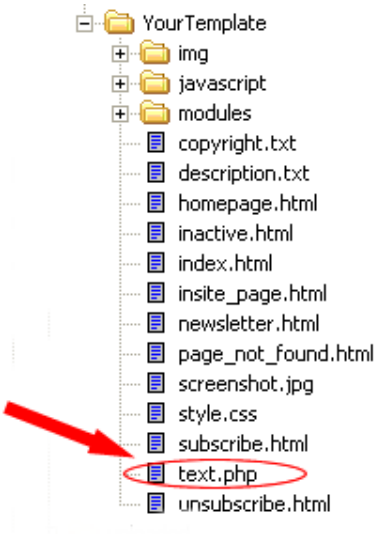
How it works

To understand better how this module operates, we must begin by its running procedure:

```
{runModule(#RenderText#,#TextName#)}
```

With this code line the *RenderText* module is recalled and *TextName* is send as a parameter for this one.

The *TextName* parameter is used as a key to search for the text that has to appear in the respective place. Where is this text being looked for? There exists a file named *text.php* which you can find in the root of the template and which contains an array with all the texts inserted with the *RenderText* module.



Why is the text taken out from this file precisely? Because you can operate upon this text from the admin area and that way you can easy make adjustments.

Pay attention! In the admin area, the key by which the search is operated within the *RenderText* module is not visible. This is the reason for which, if you look closely, the *text.php* file contains another array named *help*, with which you can assign tips for each text situated in the text array. These tips can be seen in the admin area, as you can see in the image below:

Here's an example for filling in the help vector:

Let us suppose that you want to insert in help a tip for the NewsletterTitle text that we were talking about at the first example of running the module. This way, the *text.php* file will look like this:

```
$text = array(
                "NewsletterTitle" => "Newsletter",
                "OtherTextName1" => "text1",
                "OtherTextName2" => "text2",
        );

$help = array(
                "NewsletterTitle" => "Hint for NewsletterTitle text",
                "OtherTextName1" => "Hint for OtherTextName1 text",
                "OtherTextName2" => "Hint for OtherTextName2 text",
        );
```

Available Functions and Variables

For this module there is no need for functions or variables.

Examples

Newsletter Title

We have the following code sequence:

```
siteLink)/newsletter.php" method="POST" id="Newsletter" onSubmit="retu
 <div class="newsletter">
         <div class="newsletterTitle">Newsletter</div>

         <div class="newsletterForm">
                 name:<input type="text" size="15" class="newsletterIn
                 email:<input type="text" size="15" id="newsletterEmai
                 <input type="submit" value="Submit" class="searchButt
         </div>

 </div>
```

If we want to introduce the "Newsletter" text using the *RenderText*, than here it is what you have to modify in the html file:

```
etter.php" method="POST" id="Newsletter" onSubmit="return newsletterV
wsletter">
lass="newsletterTitle">{runModule(#RenderText#,#NewsletterTitle#)}</d

lass="newsletterForm">
  name:<input type="text" size="15" class="newsletterInput" id="newsl
  email:<input type="text" size="15" id="newsletterEmail" class="news
  <input type="submit" value="Submit" class="searchButton Change_main
```

After that, we modify the text.php file. For example, above we chose *NewsletterTitle* as name for the string of characters "Newsletter". In *text.php* we will have to indicate this thing:

```
$text = array(
                "NewsletterTitle" => "Newsletter",
                "OtherTextName1" => "text1",
                "OtherTextName2" => "text2",
        );

$help = array(
                "NewsletterTitle" => "Hint for NewsletterTitle text"
                "OtherTextName1" => "Hint for OtherTextName1 text",
```

Submit Button Value

We have the following code sequence in a template-page[1]:

```
<form action="{siteLink}/search.php" method='POST' onSubmit="return
        <input class="Change_mainFont" id="searchInput" type="text"
        <input type="submit" value="Search" class="searchButton" />
</form>
```

We want to introduce the text appearing on the button, that is, the submit type value of the input, using the *RenderText* module. What we must modify in the html file is:

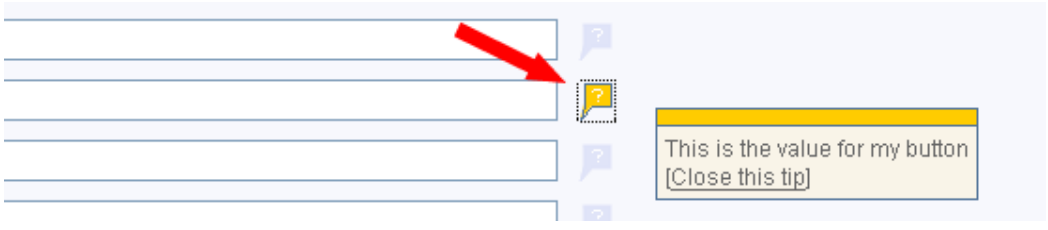
```
tion="{siteLink}/search.php" method='POST' onSubmit="return searchVal
<input class="Change_mainFont" id="searchInput" type="text" size="20"
<input type="submit" value="{runModule(#RenderText#,#ButtonValue#)}"
```

As you can see, the name *ButtonValue was assigned to the Search* string of characters. Next, we have to modify the *text.php* file situated in the root directory of the template, by adding a new line:

```
$text = array(
                "ButtonValue" => "Search",
                "OtherTextName1" => "text1",
                "OtherTextName2" => "text2",
        );

$help = array(
                "ButtonValue" => "This is the value for my button",
                "OtherTextName1" => "Hint for OtherTextName1 text",
                "OtherTextName2" => "Hint for OtherTextName2 text",
        );
```

The help array was also completed with the code line indicated by the second arrow. This results in the appearance in the admin area, near the input where the text *Search* is posted, of a tool tip containing the explanation "*This is the value for my button"*, as indicated in the image bellow:



[1] I remind you that by template-page we understand one of the html files existing in the root directory of the template.

RenderHTML

Role

You can use this module in order to insert an html code sequence, which is repeated in more than one file within your template.

All you have to do is to practically create an html file within the modules directory[1], where you insert the repeating code sequence. Instead of running it, you will run the *RenderHTML* module, together with the corresponding parameters.

You might wonder why need such a module when you can simply leave the sequence in the original file, without wasting time on moving it? Here's the reason:

It is highly probable that after having finished the template, you will want to modify part of the code appearing in more than one template-page. In this case, if you did not use the *RenderHTML* module, you ill have to do the same procedure FOR EACH AND EVERY FILE. On the other hand, if you have inserted the respective sequence using this module, all you have to do is to open the file you created in the modules directory and which contains the code sequence and to modify it A SINGLE TIME.

To better understand what this is all about, here there are some situations where the use of the *RenderHTML* module is suited:

1. Installing the header

Usually, a site has one header, no matter if the visited page is generated using the *homepage.html*, *index.html* templates or any other template-page. This is reason enough to choose to use the *RenderHTML* module instead of the alternative of writing in each template-page the code sequence for the header.

2. The newsletter subscription form

It is highly recommended that you use the *RenderHTML* module in order to insert in a template-page the newsletter subscription form, because this form will appear identical on more template-pages.

Running procedure

Running the *RenderHTML* module in a template-page can be done using the following code:

[1] The modules directory is situated in the root directory of the template.

```
{runModule(#RenderHTML#,#FileName.html#)}
```

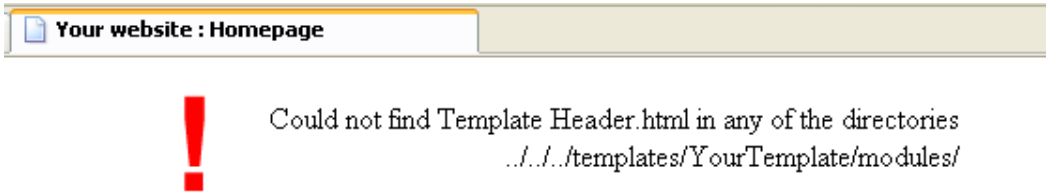
Where *FileName* is the name of the file you created in the nodules directory.

Pay attention! You will introduce this code in the exact spot of the code sequence which has been inserted in the *FileName.html* file from the modules directory.

How It Works

To understand better the way this module operates let us part from the running sequence described above. You can interpret that code line as follows: the *RenderHTML* module is run and the *FileName.html* string of characters is transmitted as parameter. This parameter is used in order to identify the html file from the modules directory containing the necessary code.

Localizing *the FileName.html* is very important. This file HAS to be found in the modules directory. In case the address of the file you entered is wrong, you will be returned in the front end an error similar to the one in the image bellow:



On the other hand, if you created the html file in the right place, the *RenderHTML* module will insert the html code in the template-page, precisely in the place of the running line.

Simply put, we can say that all that *RenderHTML* does is to include in the template-page the file whose name is specified as parameter.

Functions and constants

For this module there are available some functions and variables you can use in the *FileName.html* file[1].

I have to mention that all the functions and all the variables can be used only with the Flexy language.

[1] We cnsidered that in the tempate-page the following sequence was used in order to introduce the RenderHTML module: {runModule(#RenderHTML#,#FileName.html#)}

siteLink – is the constant that contains the link to the start page of your site.

For example:

```
siteLink = http://www.example.com
```

If you want to see precisely how this constant looks like, all you have to do is to open the *settings.php* file situated at the *root/core/settings/settings.php*. The value of this constant is indicated by *SITE_LINK* as you can see in the image below:

settings.php

```
define('SITE_NAME','Your Site Name');
define('SITE_ALIAS','Your Site Alias');
define('SITE_LINK','http://www.yoursite.com');
define('BUILD_PAGES_MODE','write_mode');
```

The running mode can be easily intuited if you would firstly read the introduction to the flexy language described above. Wthin the html code, the siteLink constant is introduced with the following code:

```
{siteLink}
```

If you try to implement a template for Bluo CMS you will notice how important this constant is and how much does it help you in writing the code.
Here are some examples for its use:

1. Writing the source of an image:

```
{else:}
        <img src="{siteLink}/templates/YourTemplate/imagesDirectory/imageName.jpg" id='logo' alt='' />
{end:}
```

2. Inserting a link towards the start page

---

bluo_template – is the constant containing the name of the current template used by Bluo CMS. By current template I understand the last template having been set up as active from the admin area.
For example:

```
bluo_template= YourTemplate
```

You might be wondering how you can realize what is the name of a template. Very simple! The name of a template is given when you choose a name for the root directory of a template. Therefore, if your template is located in the MyTemplate directory, then the name of the template will be MyTemplate. The moment this becomes active, that s, this template is selected in the admin section in the *Settings->Templates* section, the *bluo_template* constant will take its name, so it will contain the value “MyTemplate”.

If you want to see how precisely does this constant looks, all you have to do is to open the settings.php file from: *root/core/settings/settings.php*. The value of this variable is indicated as it appears in the image bellow:

```
//page default options
define('DEFAULT_AUTHOR','');
define('DEFAULT_DESCRIPTION','');
define('DEFAULT_KEYWORDS','');

define('TEMPLATE','YourTemplate');

define('LOGO','');
```

settings.php

The running procedure for this constant is:

```
{bluo_template}
```

Just as *siteLink*, this variable is important in writing the address toward different files within the root directory of the template.
Here are some examples for its use:

1. Writing the source of an image from within the root directory of the template

2. Writing the address towards a certain file from the root directory of the template:

---

checkLogo() – is a function with which you verify whether a new logo has been uploaded in the admin area, at the *Settings-> General Options* section. The function returns true if the 2 conditions are simultaneously fulfilled:

1. the LOGO constant is set in the settings[1] file, that is, the constant is not void;

2. if LOGO is not void, then there is the image located at the address root/uploaded/LOGO[2]

If one of these two constraints is not true then the *checkLogo()* function returns false; in this case the default logo has to be posted, that is, the image assigned as logo, for which the location address is known.

I have to mention that, when a new logo is selected in the admin area, the LOGO constant from *setting.php* will automatically receive as value the name of

[1] Located at the address MyTemplate/core/settings/settings.php
[2] Where LOGO is replaced by its value

the selected image[1], and this image is saved in the uploaded file located in the template's root.

In order to be able to write the path to the new image-logo the *logo*[2] variable is available to you; this will return precisely the value indicated by the LOGO constant, that is, the name and the extension of the respective image. The *logo* variable is usually used only together with the *checkLogo()* function.

Here's an example of using the function:

```
{if:checkLogo()}
    <a href='{siteLink}'>
        <img src='{siteLink}/uploaded/{logo}?{time}' alt='' id='logo' />
    </a>
{else:}
    <a href='{siteLink}'>
        <img src='{siteLink}/templates/{bluo_template}/img/logo.jpg?{time}' alt='' />
    </a>
{end:}
```

Above, you can notice that if the *checkLogo()* function returns true, that is, a new image has been uploaded as logo, the respective image will be posted. Otherwise, an image located in the *img*[3] directory, specially created for stocking images, is posted.

You can also notice the appearance of a variable *{time}* in the name of the image. This variable has been introduced in order to avoid the memorizing in the cache of the browser of the name of the logo-image. This would have led to posting in the front end of the precedent image, although this has been replaced from the admin area. The introduction of this time related variable makes the browser interpret two images with the same name as distinct one from another.

Examples

Insert Header

Inserting the header is an example of an obvious use of the *RenderHTML* module.
Let us consider that the template-page you are modifying contains the following code sequence:

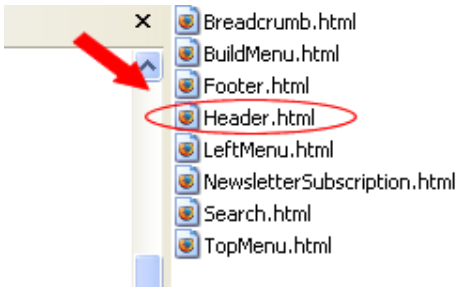
[1] If the image's name is logo.png then LOGO = 'logo.png'. It is important that you remember that the name of an image is saved together with its extension; this helps when you want to write the source of the image for the uploaded logo.
[2] The running mode for this one is {logo}
[3] The img directory is located in the root directory of the template.

```
<div id="header">
    {if:checkLogo()}
        <a href='{siteLink}'>
            <img src='{siteLink}/uploaded/{logo}?{time}' alt='' id='logo' />
        </a>
    {else:}
        <a href='{siteLink}'>
            <img src='{siteLink}/templates/{bluo_template}/img/logo.jpg?{time}' alt='' />
        </a>
    {end:}
    {runModule(#RenderHTML#,#Search.html#)}
</div>
```

Practically, you use this code to insert the logo in the web page.
If you choose to use the *RenderHTML* module in order to introduce the code from the header, then this page will look like this:

```
<div id="header">
    {runModule(#RenderHTML#,#Header.html#)}
    {runModule(#RenderHTML#,#Search.html#)}
</div>
```

All we did so far was to recall the module we are talking about. We now have to create the html file whose name was send as parameter in the running line. Pay attention, nevertheless, to the creation address! The structure of the modules directory has to look like this:



modules

The content of the html file is the following:

```
{if:checkLogo()}
    <a href='{siteLink}'>
        <img src='{siteLink}/uploaded/{logo}?{time}' alt='' id='logo' />
    </a>
{else:}
    <a href='{siteLink}'>
        <img src='{siteLink}/templates/{bluo_template}/img/logo.jpg?{time}' alt='' />
    </a>
{end:}
```

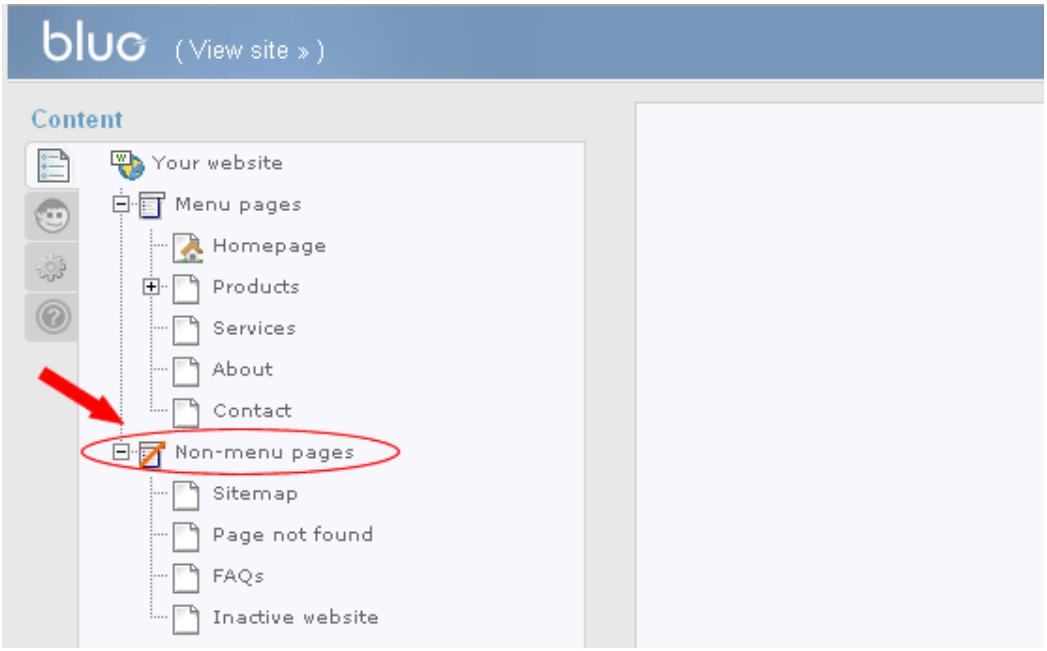
Header.html

RenderInfo

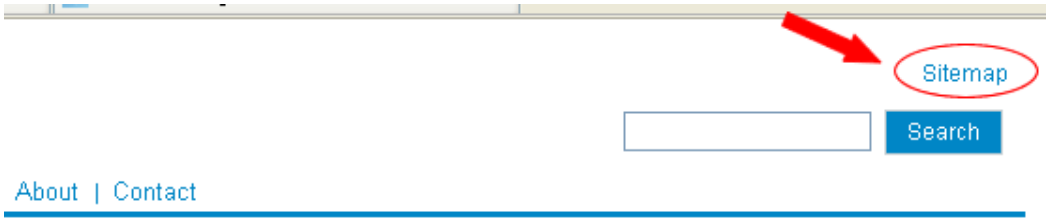
Role

You use *RenderInfo* when you want to introduce the name or the link of a single page in your site within one of the template's sections, different from the section where you introduce the content of the page[1].

You can notice the usefulness of this module when you have to introduce a link towards a page such as *SiteMap*, *FAQs*, *Terms of use*, which are not usually integrated into any of the site's menus. The fact that they do not appear in any of the site's menu implies that they will be located in the *Non-menu pages* category, and they can be found in the tree in the admin area:



Let us suppose you want to introduce the link in the image bellow:



In order to do this, you need to know two elements:

- the link towards the *SiteMap* page
- the name of the page

[1] Within the content of the page one can insert a link using the editor.

You might be wondering why you have to name the page using the *RenderInfo* module when you know that this is called *SiteMap*. But think from the following angle: the moment you want to change the name of this page, you will have to intervene into the code and make the change. On the other hand, if you use *RenderInfo*, the moment you have changed the name of this page in the admin area, the same change will happen automatically in the front end of your site.

Pay attention! In order to use this module you have to know the id of the page whose link you want to find out. How do you get this id? Very simple! You log in the admin area, you right click on the respective page and you click on *Preview*. In the address bar you will have something like this:

http://www.yoursite.com/index.php?**id=*idValue***&admin=...

The id you need is actually *idValue*.

Running procedure

You can run the *RenderInfo* module in a template-page by using the following code:

```
{runModule(#RenderInfo#,#idValue#,#infoName#)}
```

where:

- *idValue* is the id of the page for which the information is asked
- *infoName* is the name of the information. This parameter can take two distinct values:
    1. **name** – in this case, running the RenderInfo module will result in the name of the page having *idValue* as id.
    2. **link** – in this case, running the RenderInfo module will result in the link towards the page having *idValue* as id.

Pay attention! You will type this code line in the exact place where you want the respective name/link to be inserted.

How it works

Just as in the case of the other described modules, I will explain to you the way the RenderInfo module works :

```
{runModule(#RenderInfo#,#idValue#,#infoName#)}
```

This code line can be “translated” such as this: the RenderInfo module is run and *idValue* and *infoName* are are sent as paramenters. They will be used in order to obtain the wanted information.

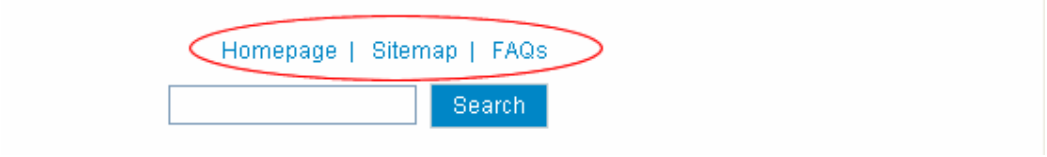
Available Functions and Variables

For this module no function or variable is needed.

Examples

Creating a menu

Even if the *RenderMenu* module will also be available to you in order to create menus, and which will be described later on, there are some cases in which, using the *RenderInfo* module, you will be able to create a menu:



Here it is how the html code for the section in the above picture looks like before using any Bluo module:

```
<div id='topRightMenu'>
    <a href="{siteLink}/faqs.html" class='topRightLink'>FAQs</a>
    <span class='topRightSpacer'>::</span>
    <a href='{siteLink}/sitemap.php' class='topRightLink'>SiteMap</a>
</div>
```

The code that you have to type in the template-page is the following one:

```
<div id='topRightMenu'>
    <a href="{runModule(#RenderInfo#,#306#,#link#)}" class='topRightLink'>
        {runModule(#RenderInfo#,#306#,#name#)}
    </a>
    <span class='topRightSpacer'>::</span>
    <a href='{siteLink}/sitemap.php' class='topRightLink'>
        {runModule(#RenderInfo#,#103#,#name#)}
    </a>
</div>
```

Inserting a link towards a page in the Menu Pages section

You must **not** understand that this module can be used only in reference to pages in the *Non-Menu Pages* section. Here's an example of a link towards a page from the *Menu Pages* category.
Let us suppose that you do not want that the home page be included in the main menu of your site[1] and that you want to insert a link towards this page in a distinct section.

Let us consider the image bellow:

[1] This is possible by correctly using the RenderMenu module. For more details see the section dedicated to this module.

The initial html code the image above, without using any Bluo module, is:

```
<div id='topRightMenu'>
    <a href='{siteLink}' class='topRightLink'>
        Home
    </a>
</div>

{runModule(#RenderHTML#,#Header.html#)}
```

In the template page you must write the following code:

```
<div id='topRightMenu'>

    <a href="{runModule(#RenderInfo#,#1#,#link#)}" class='topRightLink'>
        {runModule(#RenderInfo#,#1#,#name#)}
    </a>

</div>
```

RenderBreadcrumb

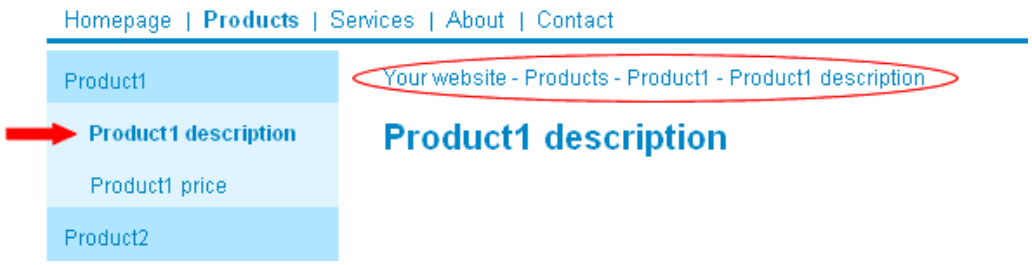
Role

The role of this section is to help you generate a breadcrumb type section.

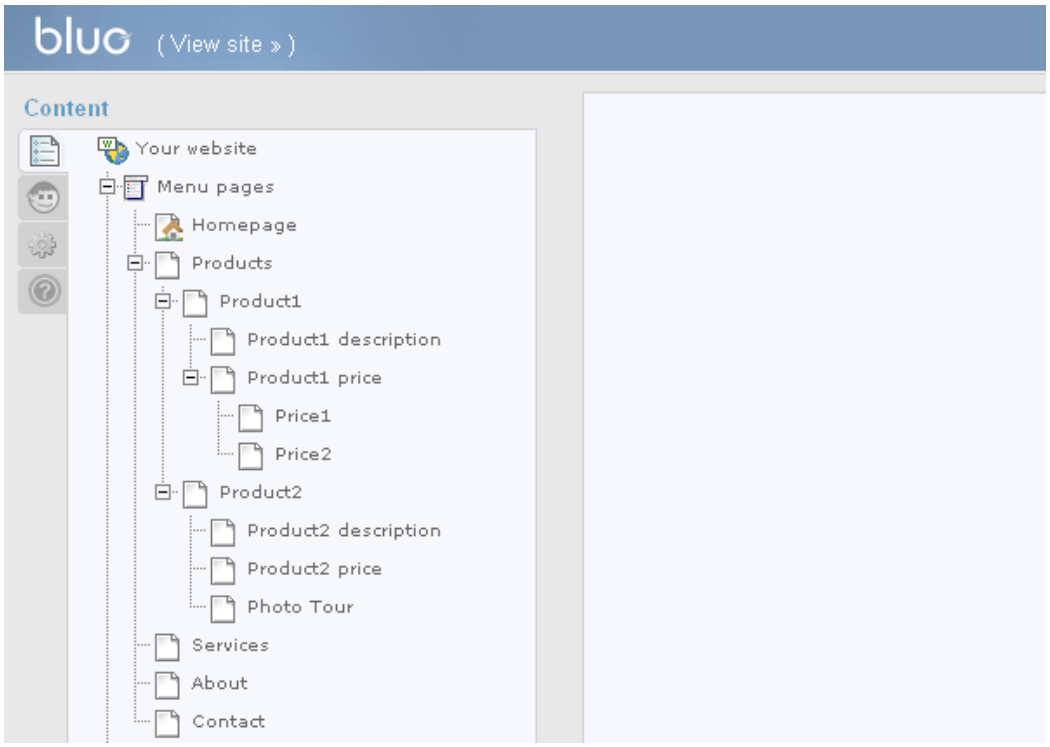
What is a breadcrumb? Here's how you can define this notion:

Breadcrumb is a link to all levels of the hierarchy above the current location, showing the route you have taken on the site, and the context of the current page. Breadcrumbs allow you to backtrack and to move up the hierarchy.

If you did not understand what it is all about, here's an eloquent image with this type of a sequence:



Practically, the breadcrumb describes the level path until the current page. Here's how the tree structure looks like for the situation above:



If your site contains such a section named Breadcrumb, than you ought to use the *RenderBreacrumb* module.

Running procedure

In case your site contains a breadcrumb section, than all you have to do in order to create it is to insert the following code in the html file from the template-page on the place where you want it to appear in the front end of the site:

```
{runModule(#RenderBreadcrumb#,#FileName.html#)}
```

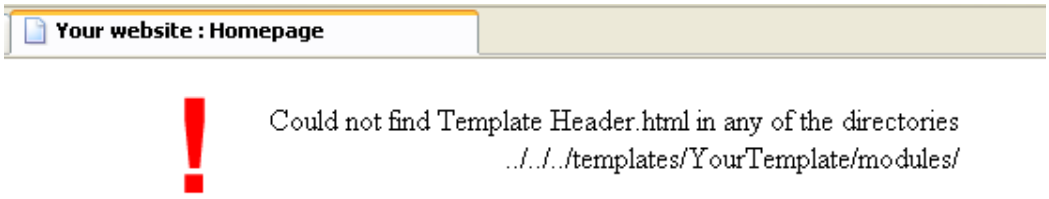
Where the *FileName* is the name of the file you created in the modules directory.

This running mode, as well as the fact that you work with a file from the modules directory, might make you think that this module does the exact same thing as the *RenderHTML* module. Nonetheless, you will discover there are many differing elements between them, the distinction being mostly based on the different functions and variables made available by each of the two modules.

How it works

To understand better the way this module operates, let us leave from the code sequence described above. That code line can be interpreted as it follows: the *RenderBreadcrumb* module is run and the NumeFisier.html string of characters is sent as a parameter. This parameter will be used to identify the html file from the modules directory which contains the code needed for creating the breadcrumb section.

Localizing the *NumeFisier.html* file is very important. This must NECESSARILY be located in the modules directory. In case you entered the wrong address for creating the file, an error similar to the one in the following image will be returned in the front end:



On the other hand, in case you created the html file in the right place, the *RenderHTML* module will insert for you the html code in the template-page.

More simply put, we can say that all that *RenderBreadcrumb* does is to include the file whose name is specified as parameter in the template-page.

The content of the *FileName.html* file is very important for the RenderBreadcrumb module, because this is where the code which generates the elements of the breadcrumb has to be written. I said elements because, as I have already mentioned in the **Role** chapter, the breadcrumb is composed by more links.

Functions and Variables

breadcrumbItems – is a vector containing the names of all the pages needed to build the breadcrumb, except for the start page.

It is of utmost importance that you know that all the elements are inserted within this array in the exact order. Therefore, in order to correctly display the breadcrumb, you must not perform any other changes, but simply publish the *breadcrumbItems* array.

The vector is indexed by the id of the page, and its value is the name of the page:

```
breadcrumbItems [pageId] = namePage;
```

To understand better the way you have to run it, follow the example from the Examples chapter.

root_name – is a constant containing the name you have chosen for your site. You can find out the value of this constant by opening the *settings.php* file located at *root/core/settings/settings.php*. The constant is pointed by *SITE_ALIAS* as you can also see in the image below:

```
12    define('DB','bluocms_site');
13    define('DSN',"mysql://".USER.":".PASS."@".HOST."/".DB);
14
15    define('SITE_NAME','www.yoursite.com');
16    define('SITE_ALIAS','Your Site');
17    define('SITE_LINK','http://www.yoursite.com');
18    define('BUILD_PAGES_MODE','write_mode');
19
20    //users options
21    define('USERS_PER_PAGE','15');
22    define('NEWSLETTER_FROM','admin@yoursite.com');
```

settings.php

Running procedure

```
{root_name}
```

---

root – is the constant containing the link towards the start page of your site, equivalent to *siteLink*, already explained at the RenderHTML module description.

For example:

```
root = http://www.example.com
```

If you want to see how precisely this constant looks like, all you have to do is to open the *settings.php* file located at *root/core/settings/settings.php*. The value of this constant s given by *SITE_LINK*, as you can see in the image below:

settings.php

```
define('SITE_NAME','Your Site Name');
define('SITE_ALIAS','Your Site Alias');
define('SITE_LINK','http://www.yoursite.com');
define('BUILD_PAGES_MODE','write_mode');
```

Running procedure
Within the html code, you will insert the root constant using the following code:

```
{root}
```

This constant is very important because, as I have already mentioned, in the *breadcrumbItems* array, the start page is not included. And this is a page that the breadcrumb has to contain.

---

bluo_template – is the constant containing the name of the current template used by Bluo CMS. By the current template you will understand the last template which has been set on as active in the admin area**.**

For example:

```
bluo_template= YourTemplate
```

If you want to see how precisely this variable looks at one moment in time, all you have to do is to open the *settings.php* file located at *root/core/settings/settings.php*. The value of this variable is identical to the one indicated in the *TEMPLATE*, as it appears in the image below:

```
//page default options
define('DEFAULT_AUTHOR','');
define('DEFAULT_DESCRIPTION','');
define('DEFAULT_KEYWORDS','');

define('TEMPLATE','YourTemplate');

define('LOGO','');
```

settings.php

The running procedure for this variable is:

```
{bluo_template}
```

This variable is important in the description of the address towards different files within the root directory of the template.

---

compare(pageId) – is a function which receives an *id* as parameter and verifies if this parameter is or is not equal to the id of the current page. The function returns the true value if the parameter coincides to the id of the current page. Otherwise, it will return the value false.

Why do you need this function? Because the name of the current page appears within the breadcrumb, but it does not appear as a link, but as a simple text, and this is very important. Therefore, you will need a function in order to verify which element within the *breadcrumbItems* must not appear as a link.

To run this function you will use an IF structure.

```
{if:compare(id)}
```

where *id* is the id of the page for which you make the check out.

---

getLink(pageId*)* – is a function which receives an id of a page as parameter and which returns the link of this page.

To run this function you will proceed as follows:

```
{getLink(id)}
```

where *id* is the id of the page whose link you want to know.

---

isEmpty() – is a function with which you will check if the *breadcrumbItems* array contains any element, that is, if it is necessary for the breadcrumb to be displayed or not.

The function returns true if the array is void and false if this array contains at least one element.

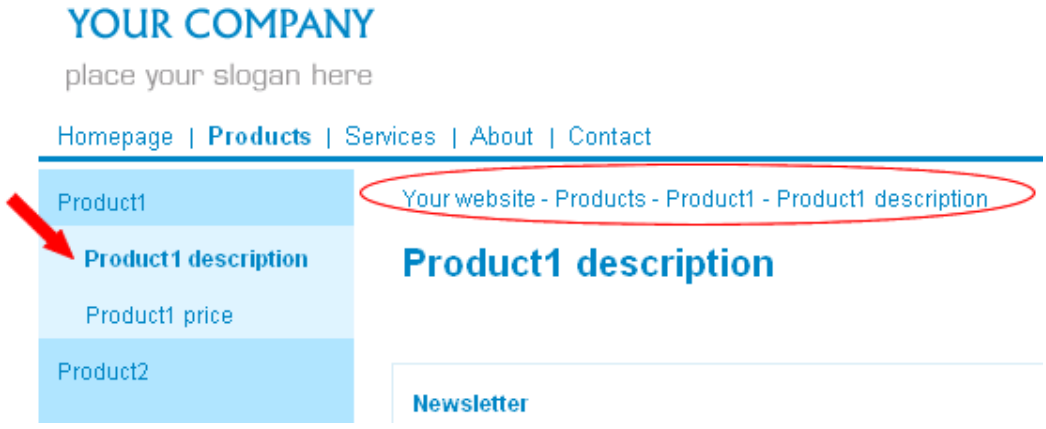
I mentioned before that the *breadcrumbItems* array does not contain the start page; this means that if the *isEmpty()* function returns true the visited page is quite the start page, and in this case the appearance of the breadcrumb section will not be needed.

The running procedure for this function is similar to that of the *compare(id)* function, because an IF structure is also used here. And because you are interested in the case where the breadcrumb can appear, that is, the case where the *isEmpty()* function returns false, you will prefer recalling it by negating the result of the function, that is:

```
{if:!isEmpty()}
```

Example

Let us consider the breadcrumb from the image below:



We want to write the code needed to generate this breadcrumb.

The code that we have to write in the template of the page where it will appear is the following::

```
{runModule(#RenderBreadcrumb#,#Breadcrumb.html#)}
```

Having only one parameter, this cod is very easy to be written. The only decision you have to take concerns the name of the file where you will write the code for the breadcrumb.

Breadcrumb.html looks as follows:

```
{if:!isEmpty()}
<a class="breadcrumbLink" href="{root}" title="{root_name}">
    {root_name}
</a> -
{foreach:breadcrumbItems,pos,name}
    {if:compare(pos)}
        <span class="breadcrumbLink">{name}</span>
    {else:}
        <a class="breadcrumbLink" href="{getLink(pos)}" title="{name}">
            {name}
        </a> -
    {end:}
{end:}
{end:}
```

Let us now comment a little upon the code from Breadcrumb.html
The first thing to be done is to verify whether the *breadcrumbItems* array is or is not void. In case there are elements in this array, the link towards the start page (about which I mention before that is not included in the *breadcrumbItems*) will be shown.

The next step consists of going through the array and writing all its elements. Because of the fact that the name of the current page must not be a link, another checking is being made before any element of the vector is published; for this, the *compare(pageId)* function is used.
You must pay special attention to correctly close the flexy structures; for this, you must correctly position the {end:} tag.

RenderMenu

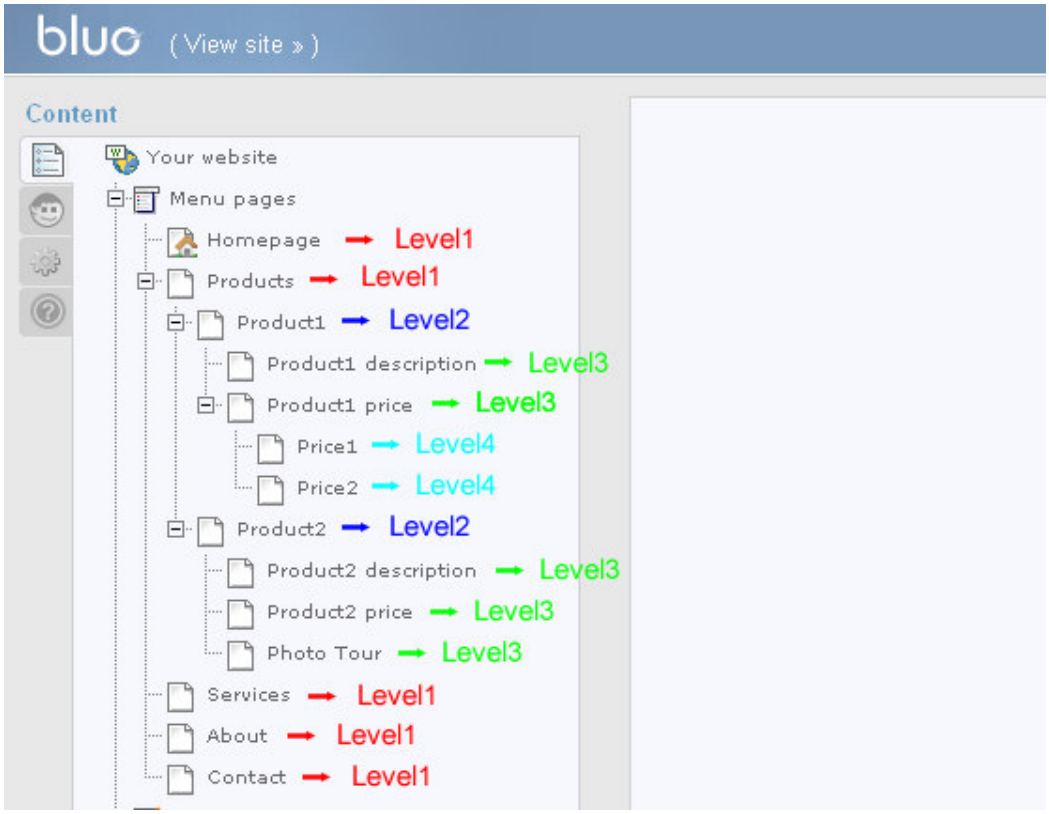
Role

For any web site the navigation menu is a very important element!

The RenderMenu module is useful precisely because of this thing: it helps you create different menus that your site will make available to the users.

As you well know, there can be menus where only pages from level 1 are published, or only pages from level 2, or menus which contain all the pages from level 1 and 2. Of course, you can extend this assertion to n levels, depending on the way the pages are organized on your site.

If you are not familiar to the numbering way of the levels, the following image will be of great help to you:



Practically the RenderMenu does not do anything else but to generate a menu depending on several parameters I will describe to you next..

Running procedure

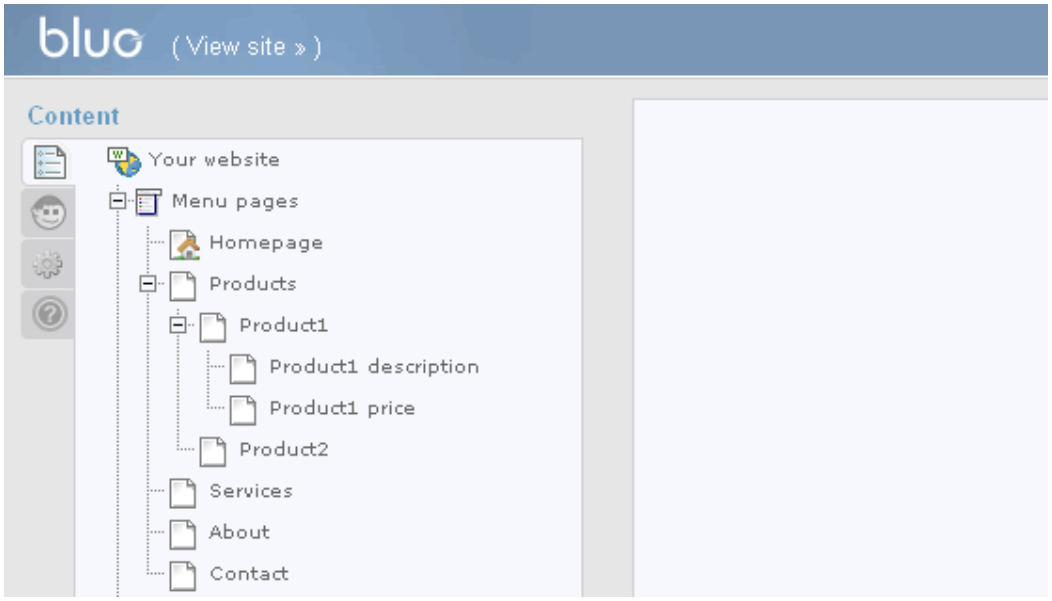
In order to create a menu with RenderMenu, you have to insert in the template-page the following code line:

```
{runModule(#RenderMenu#,#includeHomepage#,#start#,#end#,#expand#,#NumeFisier.html#)}
```

Where:

- the first parameter represents the name of the module and will always receive the value *RenderMenu* in order to run this module;
- **includeHomepage** is a parameter which can take the true or false values, depending on whether you wish or not to include the start page among the elements of the menu.
- **start** is the parameter which indicates the level from which you started to create the menu. It can take the following values:
    - 1,2,3 … - that is, indicating the number of the wanted level as the start level
    - CURRENT, CURRENT+1, CURRENT+2 … - which means that the starting level is chosen in reference to the level where the current page is situated. Therefore, if the *start* parameter receives the value CURRENT+1, and the visited page is located at the 2nd level, than the generation of the module will start from 2+1. In other words, he constant CURRENT actually indicates the level of the current page.
- **end** is a similar parameter to *start*, and it indicates the level until which you will create the menu. It can take the following values:
    - 1,2,3 … - that is, indicating the number of the wanted level as the finish level
    - CURRENT, CURRENT+1, CURRENT+2 … - which means that the level until which the menu will be created is chosen in reference to the level where the current page is situated. Therefore, if the *end* parameter receives the value CURRENT+1, and the visited page is located at the 3nd level, than the ending level of the menu will be 3+1.
- **FileName.html** – is the name of the file where you will type the html code for the menu; this file has to be located in the modules directory from the root directory of your template.
- **expand** is a parameter which can take the following values:
    - *all* – which means that your menu will contain all the pages located between the *start* and the *end* levels, inclusively.
    - *selected* – which is a particular expand-all. That is: you leave from the start level and you expand the menu only on the selected branch

To understand this better, let us consider the following consideration of the pages:



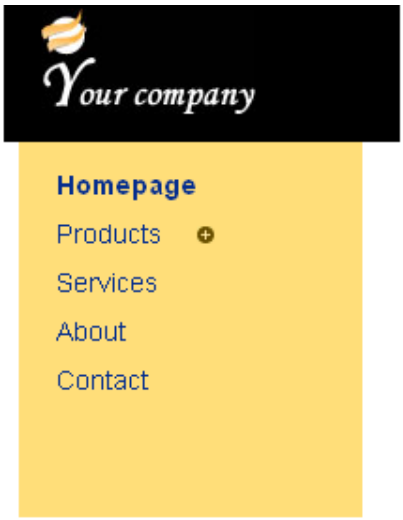
You will run the *RenderMenu* module as follows:

```
{runModule(#RenderMenu#,#true#,#1#,#3#,#selected#,#FileName.html#)}
```

The name of the html file is of no importance to us for the moment. What I would like to explain to you now is the selection mode of the pages in case the parameter expand receives the value selected.

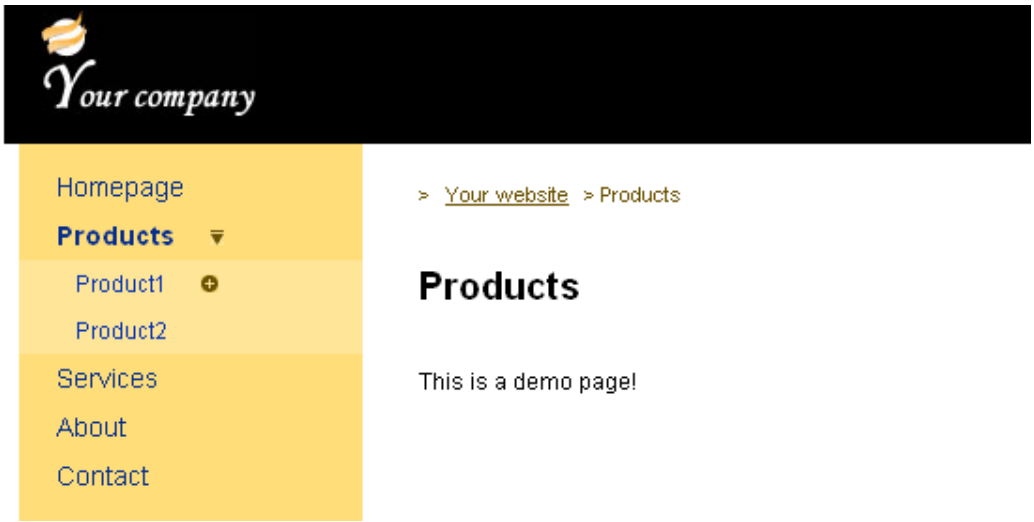
This way, if the current page[1] is Homepage, than when the menu is generated, all the pages from the same level as Homepage will be taken into consideration. Also, there will be taken into consideration all the pages directly linked with the selected page (in our case, there are no such pages). In the front end the menu will actually look like this:

[1] By current page you must understand the visited page



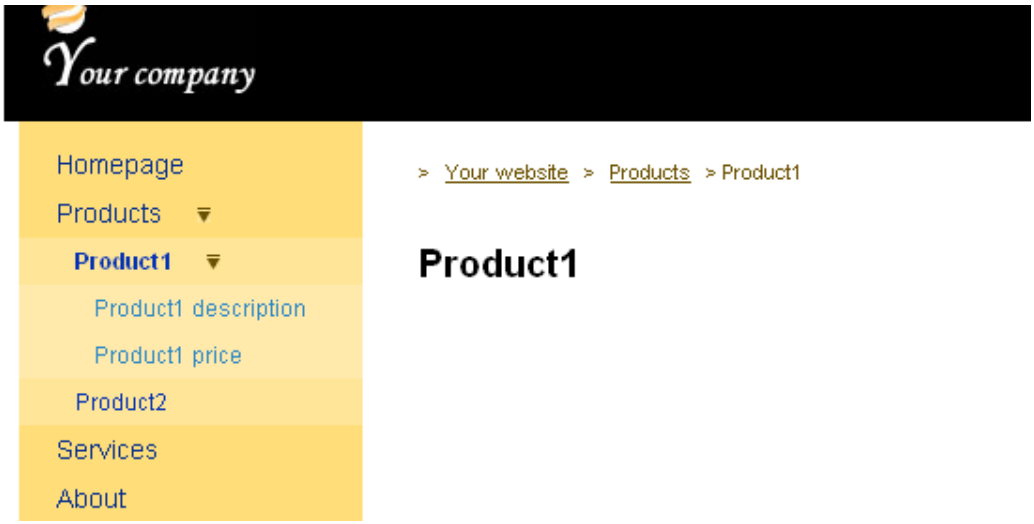
The menu will have the same structure in case the selected page is one of the following: Services, About or Contact.

The situation will change when the Products page is selected. The menu will change as follows:

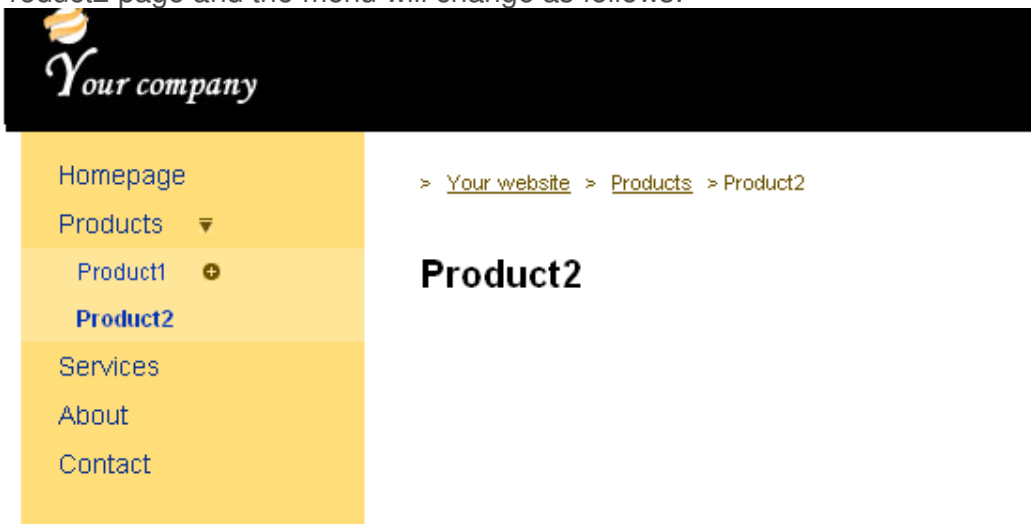


As you can notice, level 2 was expanded only for the selected page, that is, the Products page.

Moving on, if we select the Product1 page, the menu will have the following configuration:



At this stage (when the Product1 page is the current page) we select the Product2 page and the menu will change as follows:



So, in case the expand parameter receives selected as value, I conclude that the menu is composed by all the pages from the start level plus all the pages from the selected branch until level n, where n is equal to:

- *end* in case the visited page is located on the *end* level;
- N+1 (where N is the level where the selected page is located) in case N<*end*.

How it works

The running line for the *RenderMenu* module is the following:

```
{runModule(#RenderMenu#,#includeHomepage#,#start#,#end#,#expand#,#FileName.html#)}
```

The *RenderMenu* module is run, and *includeHomepage*, the starting level(indicated by *start*), the ending level (indicated by *end*), the expanding mode (all – all the page are expanded; selected – only the branch containing the selected page is expanded) and the file where the menu is formatted (indicated by *FileName.html*) are transmitted as parameters.

Pay attention! The *FileName.html* file has to be located, as I mention before, in the modules directory. Otherwise, the following error will appear in the front end:

In the case above the *FileName.html* parameter received the value *Menu.html*. The error above was a result of the fact that no html file under the name *Menu.html* was found in the modules directory.

In other words, in this file it is written in html code the mode in which the menu generated using the parameters sent while running (that is *includeHomepage*, *start*, *end*, *expand*, *NumeFisier.html*) will appear. Generating the menu consists of populating an array which will be gone through in the *FileName.html* file and which will contain all the pages that have to be displayed. More about this array in the next section.

Functions and Variables

The set of functions and variables you can use within the *FileName.html* file in order to display your menu are the following:

nodes – is the vector containing all the names of the pages in the menu, whose selection was based on the set parameters The indexation of this array is to be done by the pages ids:

```
nodes [pageId] = pageName;
```

You will run this array in foreach[1] structure, because in order for the menu to be published it is necessary for it to be gone through. I will show you some examples of where to use *nodes* in the Examples section**.**

---

bluo_template – is the constant containing the name of the current template used by Bluo CMS. By current template you will understand the last template which has been set on as active in the admin area.
For example:

```
bluo_template= YourTemplate
```

If you want to see how precisely this variable looks at some point, all you have to do is to open the *settings.php* file located at *root/core/settings/settings.php*. The value of this variable is identical to the one indicated in the *TEMPLATE*, as it appears in the image below:

```
//page default options
define('DEFAULT_AUTHOR','');
define('DEFAULT_DESCRIPTION','');
define('DEFAULT_KEYWORDS','');

define('TEMPLATE','YourTemplate');

define('LOGO','');
```

settings.php

The running procedure for this variable is:

```
{bluo_template}
```

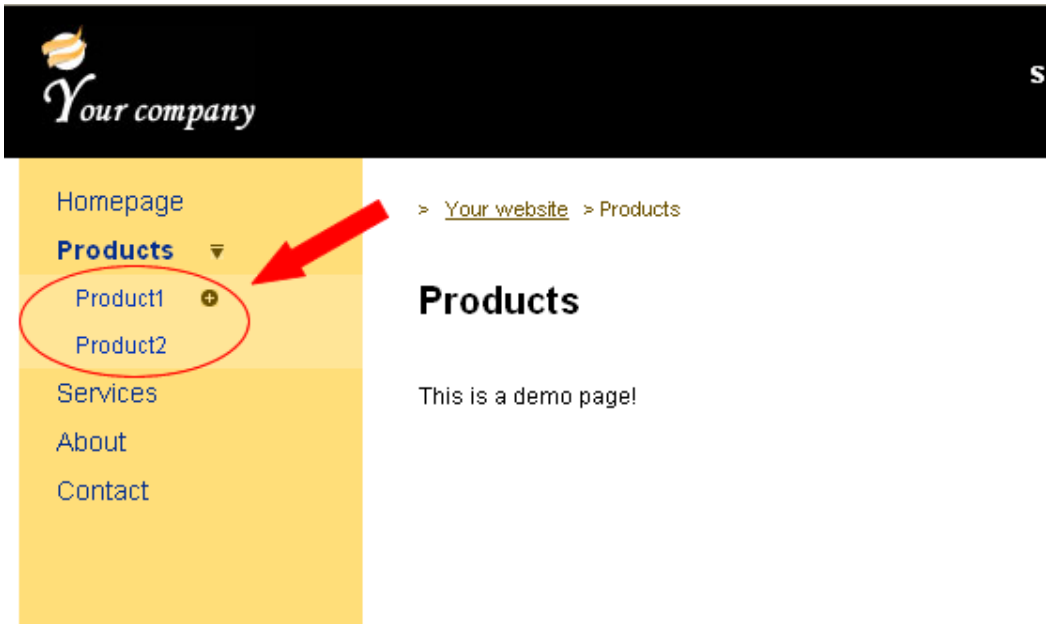
This variable is important in writing the address towards different files within the root directory of the template.

---

hasChildren(pageId*)* – is a function with which you can check if the page with the id *pageId* has or does not have children, that is, directly linked pages towards it. The function returns true if the page checked has at list one child, and false if this page has no children.

To understand better the notion of a child page, in the image below there are pointed out the 1st order children for the Products page.

[1] See the introductive chapter on the Flexy language.



You will run the *hasChildren(pageId)*function in an IF structure, as it follows:

```
{if:hasChildren(pageId)}
```

---

isFirst(pageId*)* – – is a function with which you can check if the page with the *pageId* id is the first one contained in the *nodes* array; in other words, you can check if it is the first element in the menu.

The function returns true if the page with the id *pageId* is the first element in the vector. Otherwise, it returns false.

One example of running this function is:

```
{if:isFirst(pageId)}
```

---

getLink(pageId*)* – is a function which receives an id of a page as parameter and returns the link towards this page

To run this function you will type:
`{getLink(id)}`

where *id* is the id of the page for which you want to know the link.

---

isSelected(pageId) – is a function which gives you the opportunity to check if a page is or is not visited during the moment when the menu is generated.

The function returns true if the page with the id *pageId* is the current page. Otherwise, it returns false.

The running mode for this function is:

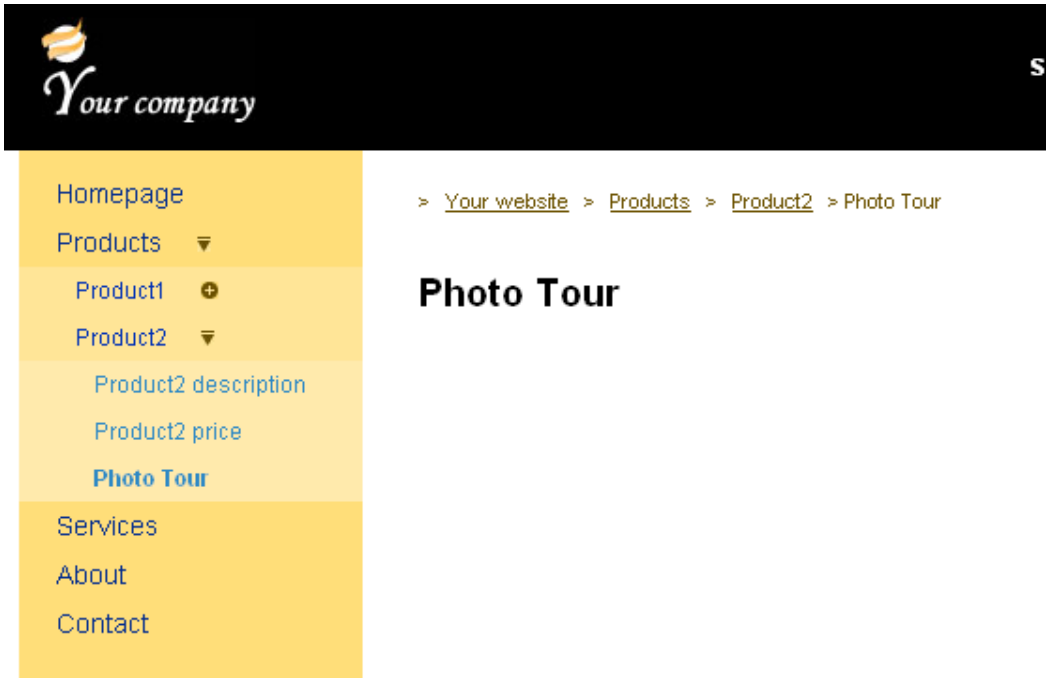
```
{if:isSelected(pageId)}
```

isSelected(pageId) is useful in case you want to apply other styles to the element in the menu which indicates the current page.

---

isActive(pageId) – is a function which returns true if the page with the id *pageId* is the selected page or if it is located on the child-parent path from the selected page to the corresponding page on the top level of the menu.

Let us consider the case in the image below:



As you can notice, the selected page is Photo Tour. The child-parent path from this page till the starting level is:

Photo Tour -> Product2 -> Products. Therefore, *the isActive(pageId)* will return true if the page with the id *pageId* will indicate the id of one of the pages mentioned above.

If you look closely to the image, the child-parent path I mentioned above can also be identified in the breadcrumb[1], only written in the reverse order.

The running mode for this function is:

[1] The breadcrumb appears in the right of the image.

`{if:isActive(pageId)}`

We can observe that *isActive* behaves just as *isSelected* in case *pageId* indicates the value of the selected page's id.

---

checkLogo() – is a function with which you can check if a new logo has been uploaded from the admin area, in the *Settings-> General Options* section. The function returns true if the following two constraints are simultaneously fulfilled:

1. in the settings[1] file, the LOGO constant is set on, that is, it is not void
2. if LOGO is not void, than there exists the image located at root/uploaded/LOGO[2]

If one of this conditions is not true , than the *checkLogo()* function will return false; in this case the default logo will be posted, which is the image assigned as logo whose address is known.

I have to mention that once a new logo is selected from the admin area, the constant LOGO from the *settings.php* will automatically receive as value the name of the selected image[3]; this image is saved in the uploaded directory located in the root of the template.

In order for you to write the path towards the new image-logo, the *logo*[4] variable is available to you. It will return precisely the value indicated by the *LOGO* constant, that is, the name and the extension of the respective image. The *logo* variable will usually be used only together with the *checkLogo()* function.

Here's an example of using this function:

```
{if:checkLogo()}
    <a href='{siteLink}'>
        <img src='{siteLink}/uploaded/{logo}?{time}' alt='' id='logo' />
    </a>
{else:}
    <a href='{siteLink}'>
        <img src='{siteLink}/templates/{bluo_template}/img/logo.jpg?{time}' alt='' />
    </a>
{end:}
```

In the example above you may notice that if the *checkLogo()* function returns true, that is, a new image has been uploaded as logo, the respective new image will be published. Otherwise, an image located in the *img*[5] directory, specially created for stocking images, will be published.

---

[1] You can find it at MyTemplate/core/settings/settings.php.
[2] Where LOGO is replaced by its value.
[3] If the image is called logo.png then LOGO = 'logo.png'. It is important that you remember that the name of the image is saved together with its extension; this will be of great value when you want to write the source of the image for the uploaded logo.
[4] Its running mode is {logo}.
[5] The img directory can be found in the root directory of the template.

In the same time, you may notice the appearance of a variable, *{time},* in the name of the image. This variable was inserted in order to avoid an eventual memorizing in the cache of the browser of the name of the image-logo. This would mean that in the front end the preceding image would be published, whereas this has been replaced in the admin area. The appearance of this variable, depending on the time, makes the browser interpret as distinct two images that have practically the same name.

---

compareShift(pageId,level) – is a function which returns true if the page with the id *pageId* is located on the same level. Otherwise, it returns false.

What I must mention here is that *level* takes values starting with 0. Therefore, in case you want to test if the page with the id *pageId* is on the top level in the tree, you must run the function as it follows:

```
{if:compareShift(pos,0)}
```

In order to do the same thing for the next level, the level parameter needs to take the value 1.

The running mode for this function is the one indicated above.

This function is important in case your menu contains pages for more than one level, as it will help you make a distinction between pages located on different levels.

---

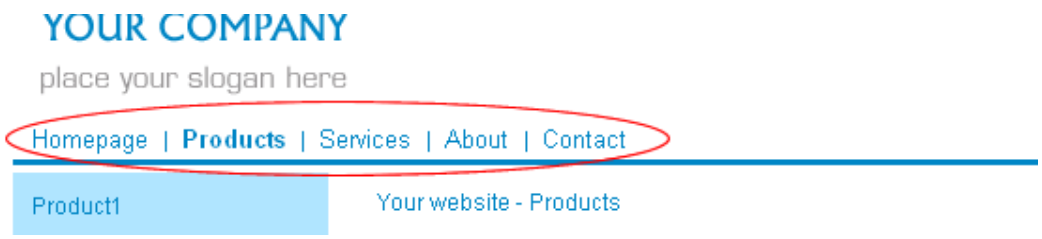
getShift(pageId) – is a function which returns the level where the page with the id *pageId* is located.

Its **running procedure**:

```
{getShift(pageId)}
```

Examples

Let us consider the menu below:



You want to write the code needed to generate this module.

Observations:

- all the included pages are located on the first level of the tree in the admin area; therefore, the start parameter will be equal to the end parameter, both of them showing the common level of the pages.
- homepage is part of the menu so the *includeHomepage* parameter will be true

You have to insert in the template of the page where you want this menu to appear the following code:

```
{runModule(#RenderMenu#,#true#,#1#,#1#,#all#,#TopMenu.html
#)}
```

I chose to name the file where I will write the code for the menu *TopMenu.html*. Instead of this name, any other name can be chosen, provided that it respects the extension.

TopMenu.html looks like this:

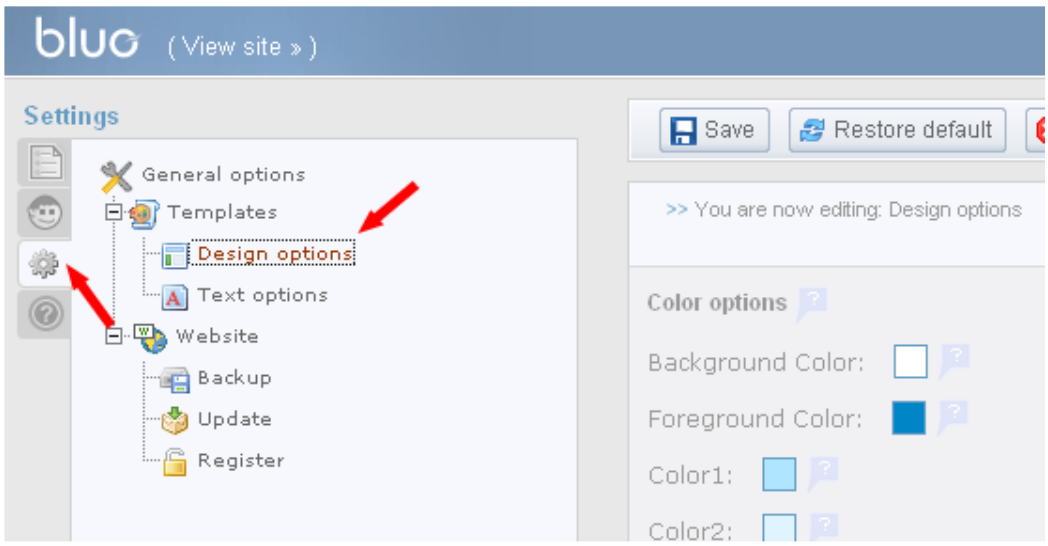
```
{if:hasPages()}
{foreach:nodes,pos,name}
    {if:isFirst(pos)}
        {if:isActive(pos)}
            <a href="{getLink(pos)}" title="{name}" class="topMenuLinkSel">
                <b>{name}</b>
            </a>
        {else:}
            {if:isSelected(pos)}
                <a href="{getLink(pos)}" title="{name}" class="topMenuLinkSel">
                    <b>{name}</b>
                </a>
            {else:}
                <a href="{getLink(pos)}" title="{name}" class="topMenuLink">
                    {name}
                </a>
            {end:}
        {end:}
    {else:}
        {if:isActive(pos)}
            |   <a href="{getLink(pos)}" title="{name}" class="topMenuLinkSel">
            <b>{name}</b>
            </a>
        {else:}
            {if:isSelected(pos)}
                |   <a href="{getLink(pos)}" title="{name}" class="topMenuLinkSel">
                        <b>{name}</b>
                    </a>
            {else:}
                |   <a href="{getLink(pos)}" title="{name}" class="topMenuLink">
                        {name}
                    </a>
            {end:}
        {end:}
    {end:}
{end:}
{end:}
```

The only thing from this code that has to be explained is the way the elements from the menu are extracted from the *nodes* array.. In this way, using the foreach structure from the flexy language, a loop through the elements of the array will be executed. The elements of the menu are present in the array in the precise order they must be published.

The CSS

This chapter is dedicated entirely to the way in which you have to create the *style.css* file. You might be wondering why this file needs to have a certain structure as long as it contains the styles for your template that you can sort, create and use as you please.

It is important that you follow a certain structure in order to have access to some facilities offered by Bluo. In this way, you will be able to use some styles from *style.css* within the editor in the admin area of the site. Or you can change some colors, some styles for the fonts, some borders of your site, without modifying the *style.css* file. Instead, by simply clicking a few times in the admin area, more precisely in the *Settings->Design* options section you can obtain all these changes.



In order for your template to be modifiable in respect to the colors, the fonts or the borders, you have to follow certain structures of declaring the classes which set these elements.

The elements of a class you created in *styles.css* become modifiable from the admin area only if the respective class has been declared according to the following structure:

```
/* category | classTitle | classDescription */
.Change_className{
     styles
}
```

where:

category – is the name of the category containing the respective class

You declare a category at the beginning of the *style.css* as it follows:

```
/* categoryName: categoryDescription */
```

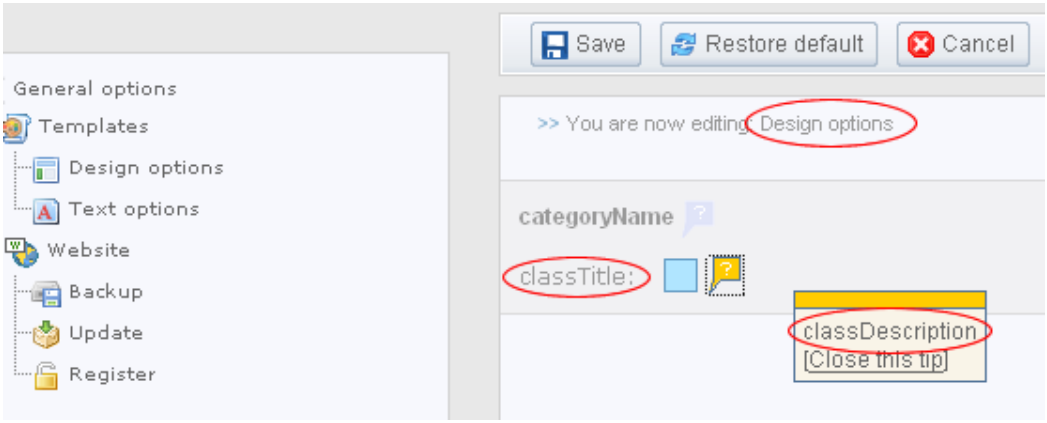
categoryName and *categoryDescription* are the parameters you chose according to your needs. The values you choose for them can be visualized in the admin area, in the *Settings->Design options* section.

classTitle – is the name the class takes in the admin area.

classDescription – is the description of the respective class for the admin area.

className – is a pseudo-name for the class; the pseudo-name is appended with the string of characters “Change_” and there results the name of the class.
Pay attention!!! In the html files this class will be run under the name *Change_className*.

classTitle and *classDescription* are visible in the admin area as it follows:



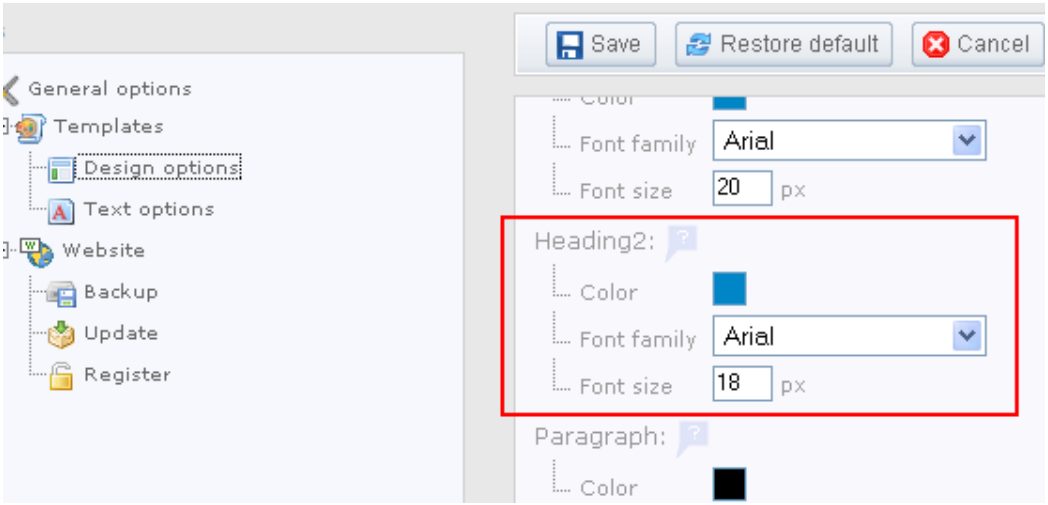
styles – represents the styles which become modifiable.

Pay attention!!! Not all the styles included in the class are necessarily modifiable. As I mentioned before, in this category there enter only the properties related to *border*, *color*, *font-size*, *font-family* and *background color*.

In other words, if we have a class declares as it follows:

```
/* Text options | Heading2 | Sets the text style for h2 */
.Change_heading2{
      color: #0086C6;
      font-size: 18px;
      font-family: Arial, sans-serif;
      text-align: left;
      margin: 0px;
      padding-left: 23px;
      padding-top: 0px;
}
```

from these properties there become modifiable only the following: *color*, *font-size* an *font-family*, the rest of them remaining as they are. These last ones will not be included in the main area either, which means that you will be able to visualize from that class only the following elements in the *Settings->Design options* section:



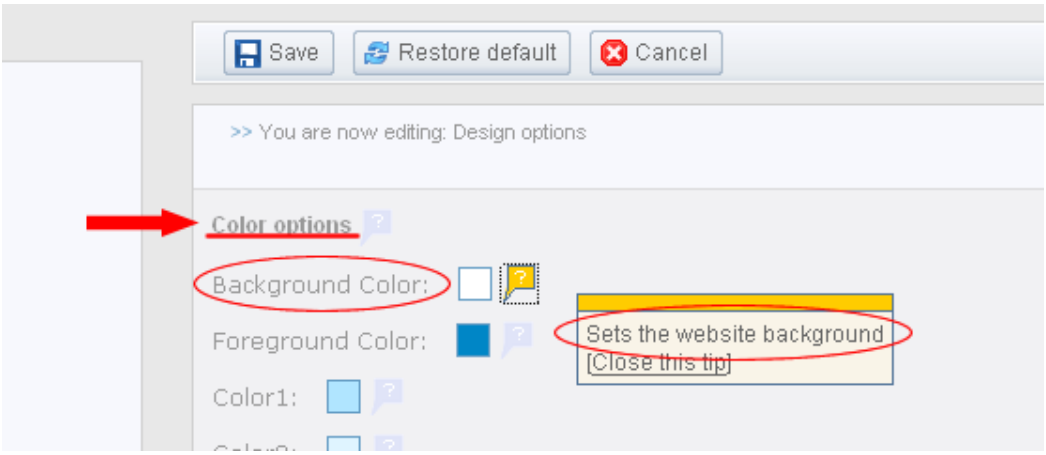
Let us see an example of the beginning part of a *style.css* file.

```
/* Color options: You can use these options to modify the colors from the website */
/* Text options: You can use these options to modify the text from your website */
/* Color options | Background Color | Sets the website background */
.Change_siteBackground{
    background-color: #ffffff;
}
/* Text options | Main Font | Sets the main font of text */
.Change_mainFont{
    font-size: 12px;
    font-family: Arial, sans-serif;
}
/* Color options | Foreground Color | Sets the default color of the text */
.Change_textColor{
    color: #0086C6;
}
/* Color options | Color1 | Sets the background color for the menu items from level 1 */
.Change_color1{
    background-color: #b0e5ff;
}
/* Color options | Color2 | Sets the background color for the menu items from level 2 */
.Change_color2{
    background-color: #dff4ff;
}
```

The first and the second line contain the declaration part of the categories where the classes containing the modifiable elements are included.

The next, third, line sets "**Backgroung color"** class and the "**Sets the website background"** description as part of the **Color options** category.

After this line there follows the definition of the class, without overlooking using the proper syntax for the name of the class; this means that you must begin it with the "*Change_*" string of characters. Following this declaration, in the admin area the following elements will be visible:



Regarding the import of certain classes from *style.css* in the editor in the admin area, you must divide the classes in:

- classes for the elements p, h1, h2 etc[1]
- custom classes you have created.

Integrating the classes for the p, h1, h2 etc elements is done automatically. That means that if I have the following class in the *style.css*

[1] Any class that is assigned to a container (p, div, ul, etc) and which is declared using only the name of that container.

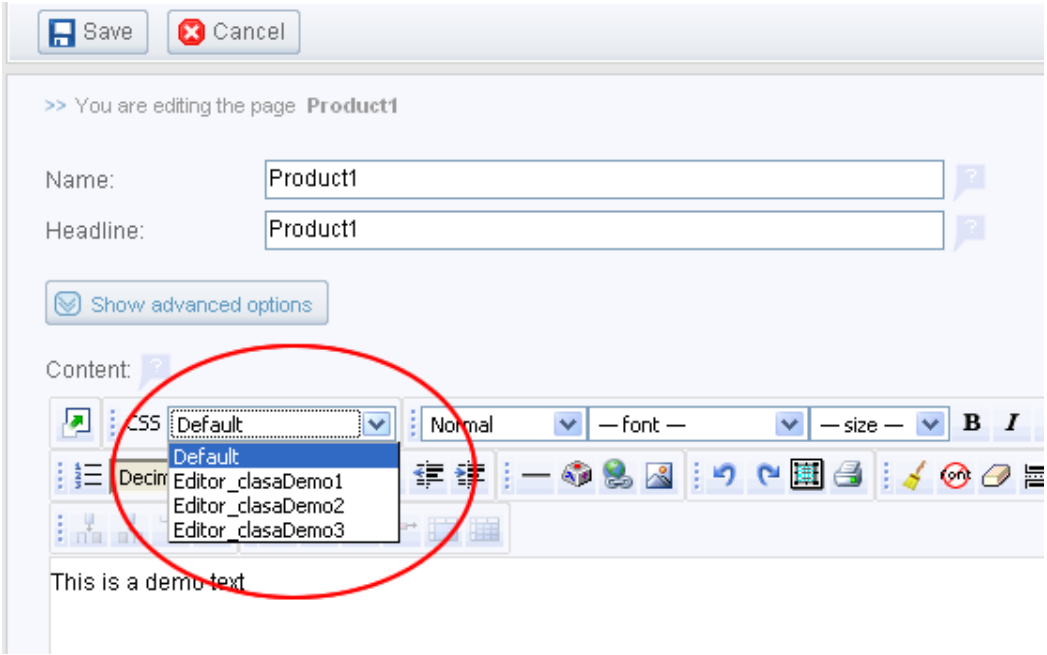
```
p{
      font-size: 12px;
}
```

then any paragraph from the editor will have this class as the default class. The same is for the other elements.

The custom classes you have created can be imported in the editor by modifying their name taking into account the following aspect:
Any class which contains within its name the string of characters "*Editor_*" is imported in the editor.
All you have to do is to actually take each class that you need in the editor and include "*Editor_*" in its name.

Pay attention!!! Do not forget to modify the name of the respective classes in the html files where you will use them also.

The classes thus imported in the editor will be included into a dropdown which appears in the menu of the editor, as indicated in the following image:



The classes above have appeared because of the following code from the *style.css* file:

```
.Editor_clasaDemo1{

}
.Editor_clasaDemo2{

}
```

```
.Editor_clasaDemo3{

}
```