Displaying content on the front office

Displaying content on the front office

As it is, the module does not do much. In order to display something on the front office, we have to add support for a few hooks. This is done by implementing the hooks' methods, and that was actually done in the `install()` method we wrote earlier, using the `registerHook()` method:

```
public function install()
{
  if (Shop::isFeatureActive())
    Shop::setContext(Shop::CONTEXT_ALL);

  return parent::install() &&
    $this->registerHook('leftColumn') &&
    $this->registerHook('header') &&
    Configuration::updateValue('MYMODULE_NAME', 'my friend');
}
```

As you can see, we make it so that the module is hooked to the "`leftColumn`" and "`header`" hooks. In addition to this, we will add code for the "`rightColumn`" hook.

Attaching code to a hook requires a specific method for each:

As you can see, we make it so that the module is hooked to the "`leftColumn`" and "`header`" hooks. In addition to this, we will add code for the "`rightColumn`" hook.

Attaching code to a hook requires a specific method for each:

- `hookDisplayLeftColumn()`: will hook code into the left column – in our case, it will fetch the MYMODULE_NAME module setting and display the module's template file, `mymodule.tpl`, which must be located in the `/views/templates/hook/` folder.
- `hookDisplayRightColumn()`: will simply do the same as `hookDisplayLeftColumn()`, but for the right column.
- `hookDisplayHeader()`: will add a link to the module's CSS file, `/css/mymodule.css`.

```
public function hookDisplayLeftColumn($params)
{
  $this->context->smarty->assign(
      array(
          'my_module_name' => Configuration::get('MYMODULE_NAME'),
          'my_module_link' => $this->context->link->getModuleLink('mymodule', 'display')
      )
  );
  return $this->display(__FILE__, 'mymodule.tpl');
}

public function hookDisplayRightColumn($params)
{
  return $this->hookDisplayLeftColumn($params);
}

public function hookDisplayHeader()
{
  $this->context->controller->addCSS($this->_path.'css/mymodule.css', 'all');
}
```

We are using the Context (`$this->context`) to change a Smarty variable: Smarty's `assign()` method makes it possible for us to set the template's name variable with the value of the `MYMODULE_NAME` setting stored in the configuration database table.

The header hook is not part of the visual header, but enables us to put code in the `<head>` tag of the generated HTML file. This is very useful for JavaScript or CSS files. To add a link to our CSS file in the page's `<head>` tag, we use the `addCSS()` method, which generates the correct `<link>` tag to the CSS file indicated in parameters.

Save your file, and already you can hook your module's template into the theme, move it around and transplant it (even though there is not template file for the moment): go to the "Positions" page from the "Modules" menu in the back office, then click on the "Transplant a module" button (top right of the page).

In the transplantation form:

- Find "My module" in the "Module" drop-down list.
- Choose "(displayLeftColumn) Left column blocks" in the "Hook into" drop-down list.
- Click "Save".

It is useless to try to attach a module to a hook for which it has no implemented method.

The "Positions" page should reload, with the following message: "Module transplanted successfully to hook" (or maybe "This module has already been transplanted to this hook. "). Congratulations! Scroll down the "Positions" page, and you should indeed see your module among the other modules in the "Left column blocks" list. Move it to the top of the list by drag'n'dropping the module's row.

The module is now attached to the left column… but without any template to display, it falls short of doing anything useful: if you reload the homepage, the left column simply displays a message where the module should be, saying "No template found for module mymodule".

Displaying content

Now that we have access to the left column, we should display something there for the customer to see.

The visible part of the module is defined in `.tpl` files placed in specific View folders:

- `/views/templates/front/`: front office features.
- `/views/templates/admin/`: back office features.
- `/views/templates/hook/`: features hooked to a PrestaShop (so can be displayed either on the front office or the back office).

Template files can have just about any name. It there is only one such file, it is good practice to give it the same name as the folder and main file: `mymodule.tpl`.

In the case of this tutorial, the module will be hooked to the left column. Therefore, the TPL files that are called from the column's hook should be placed in `/views/templates/hook/` in order to work properly.

As said earlier, the content to be displayed in the theme should be stored in `.tpl` template files placed in a specific folder: `/views/templates/front/`. Template files can have just about any name. It there is only one such file, it is good practice to give it the same name as the folder and main file: `mymodule.tpl`.

We will create the `mymodule.tpl` file, which was passed as a parameter of the `display()` method in our module's code, in the `hookDisplayHome()` method. When calling a template from within a hook, PrestaShop looks for that template file in the `/views/templates/hook/` folder (in the module's folder), which you must create yourself.

In PrestaShop 1.4, the module's template files were to be placed at the root of the module's folder.

For compatibility reasons, template files can still reside in the root folder in PrestaShop 1.5 and 1.6, although the sub-folders of `/views/templates/` are now the recommended locations. If you intend your module to also work in PrestaShop 1.4, you should keep your files at the root.

Here is our template file, located at `/views/templates/hook/mymodule.tpl`:

```
<!-- Block mymodule -->
<div id="mymodule_block_home" class="block">
  <h4>Welcome!</h4>
  <div class="block_content">
    <p>Hello,
       {if isset($my_module_name) && $my_module_name}
           {$my_module_name}
       {else}
           World
       {/if}
       !
    </p>
    <ul>
      <li><a href="{$my_module_link}" title="Click this link">Click me!</a></li>
    </ul>
  </div>
</div>
<!-- /Block mymodule -->
```

This is just regular HTML code… except for a few Smarty calls:

- The `{l s='xxx' mod='yyy'}` call is PrestaShop-specific method that enables you to register the string in the module's translation panel. The s parameter is the string, while the mod parameter must contain the module's identifier (in the present case, "mymodule"). We only use this method once here for readability reasons, but in practice it should be used on all of the template's strings.
- The `{if}`, `{else}` and `{/if}` statements are Smarty conditionals. In our example, we check that the `$my_module_name` Smarty variable exists (using PHP's `isset()` function, which considered as trusted by Smarty) and that it is not empty. If it goes well, we display the content of that variable; if not, we display "World", in order to have "Hello World".
- The `{$my_module_link}` variable in the link's `href` attribute: this is a Smarty variable that we will create later on, which will point to PrestaShop's root directory.

In addition to that, we are going to create a CSS file, and save it as /css/mymodule.css in the module's folder (or any sub-folder you like to keep you CSS in):

```
div#mymodule_block_home p {
  font-size: 150%;
  font-style:italic;
}
```

Save the template file in the module's `/views/templates/hook/` folder and the CSS file in the module's `/css/` folder, reload your shop's homepage: the content of the template should appear on top of the left column, right below the shop's logo (if you have indeed moved it at the top of the "Left Column" hook during the transplanting part).

As you can see, the theme applies its own CSS to the template we added:

- Our <h4> title becomes the block's header, styled the same way as the other block titles.
- Our `<div class="block_content">` block has the same style as the other blocks on the page.

It is not pretty, but it works the way we want it to.

Disabling the cache

If you’ve followed this tutorial to the letter and still do not see anything appearing in the theme’s left column, it might be because PrestaShop has cached the previous templates, and is still serving these to you. Hence, you see the original version of the theme, without your changes.

Smarty caches a compiled version of the homepage, for performance reasons. This is immensely helpful for production sites, but is useless for tests sites, where you may load the front-page very regularly in order to see the impact of your changes.

When editing or debugging a theme on a test site, you should always disable the cache, in order to force Smarty to recompile templates on every page load. To that end, go to the “Advanced Parameters” menu, select the “Performance” page, then, in the “Smarty” section:

- Template cache. Choose “Disable the cache”.
- Cache. Disable it.
- Debug console. You can also open the console if you want to learn more about Smarty’s internals.

Do NOT disable the cache or enable the debug console on a production site, as it severely slows everything down! You should always perform all your tests in a test site, ideally on your own computer rather than online.

Embedding a template in the theme

The link that the module displays does not lead anywhere for now. Let’s create the `display.php` file that it targets, with a minimal content, and put it in the module’s root folder.

```
Welcome to this page!
```

Click the “Click me!” link: the resulting page is just that raw text, without anything from the theme. We would like to have this text embedded in the theme, so let’s see how to do just that.

As you would expect, we have to create a template file in order to use the theme’s style. Let’s create the `display.tpl` file, which will contain the basic “Welcome to my shop!” line, and will be called by `display.php`. That display.php file will be rewritten into a front-end controller in order to properly embed our basic template within the theme’s header, footer, columns, etc.

NOTE: You should strive to use explicit and recognizable names for your template files, so that you can find them quickly in the back office – which is a must when using the translation tool.

Here are our two files:

```
display.php
<?php
class mymoduledisplayModuleFrontController extends ModuleFrontController
{
  public function initContent()
  {
    parent::initContent();
    $this->setTemplate('module:mymodule/views/templates/front/display.tpl');
  }
}
```

```
display.tpl
Welcome to my shop!
```

Let’s explore `display.php`, our first PrestaShop front-end controller, stored in the /controllers/front folder of the module’s main folder:

- A front-end controller must be a class that extends the ModuleFrontController class.
- That controller must have one method: initContent(), which calls the parent class' initContent() method...
- ...which then calls the setTemplate() method with our display.tpl file.

`setTemplate()` is the method that will take care of embedding our one-line template into a full-blown page, with proper header, footer and sidebars.

Until PrestaShop 1.4, developers who wanted to embed a template file into the site’s theme had to use PHP’s include() calls to include each portion of the page. Here is the equivalent code for display.php:

```
display.php
<?php
// This file must be placed at the root of the module's folder.
global $smarty;
include('../../config/config.inc.php');
include('../../header.php');

$smarty->display(dirname(__FILE__).'/display.tpl');

include('../../footer.php');
?>
```

As you can see, this is not necessary anymore since PrestaShop 1.5: you can and should use a front-end controller, and both the controller (Controller) and its template (View) should share the same name: `display.php` is tied to `display.tpl`.

Save both files in their respective folders, and reload your shop’s homepage, then click on the “Click me!”, and voilà ! You have your link. With just a few lines, the end result is already much better, with the “Welcome” line neatly placed between header, footer and columns!

It is only a first step, but this gives you an idea of what is possible if you follow the templating rules.
Using Smarty

Smarty is a PHP template engine, and is used by PrestaShop’s theming system. It is a free and open-source projet, hosted at http://www.smarty.net/.

It parses template `.tpl` files, looking for dynamic elements to replace with their contextual equivalents, then send the generated result to the browser. Those dynamic elements are indicated with curly brackets: `{ ... }`. Programmers can create new variables and use them in TPL files; PrestaShop adds its own set of variables.

For instance, we can create the $my_module_message variable in PHP right in the `hookDisplayLeftColumn()` method, and have it displayed by our template file:

```
mymodule.php
public function hookDisplayLeftColumn($params)
{
    $this->context->smarty->assign(
        array(
            'my_module_name' => Configuration::get('MYMODULE_NAME'),
            'my_module_link' => $this->context->link->getModuleLink('mymodule', 'display'),
            'my_module_message' => $this->l('This is a simple text message') // Do not forget to enclose your
strings in the l() translation method
        )
    );

    return $this->display(__FILE__, 'mymodule.tpl');
}
```

From there on, we can ask Smarty to display the content of this variable in our TPL file.

```
 mymodule.tpl
{$my_module_message}
```

PrestaShop adds its own set of variables. For instance, `{$hook_left_column}` will be replaced with the content for the left column, meaning the content from all the modules that have been attached to the left column's hook.

All Smarty variables are global. You should therefore pay attention not to name your own variable with the name of an existing Smarty variable, in order to avoid overwriting it. It is good practice to avoid overly simple names, such as `{products}`, and to prefix it with your module's name, or even your own name or initials, such as: `{$henryb_mymodule_products}`.

Here is a list of Smarty variables that are common to all pages:

File / folder	Description
img_ps_dir	URL for PrestaShop's image folder.
img_cat_dir	URL for the categories images folder.
img_lang_dir	URL for the languages images folder.
img_prod_dir	URL for the products images folder.
img_manu_dir	URL for the manufacturers images folder.
img_sup_dir	URL for the suppliers images folder.
img_ship_dir	URL for the carriers (shipping) images folder.
img_dir	URL for the theme's images folder.
css_dir	URL for the theme's CSS folder.
js_dir	URL for the theme's JavaScript folder.
tpl_dir	URL for the current theme's folder.
modules_dir	URL the modules folder.
mail_dir	URL for the mail templates folder.
pic_dir	URL for the pictures upload folder.
lang_iso	ISO code for the current language.
come_from	URL for the visitor's origin.
shop_name	Shop name.
cart_qties	Number of products in the cart.

cart	The cart.
currencies	The various available currencies.
id_currency_cookie	ID of the current currency.
currency	Currency object (currently used currency).
cookie	User cookie.
languages	The various available languages.
logged	Indicates whether the visitor is logged to a customer account.
page_name	Page name.
customerName	Client name (if logged in).
priceDisplay	Price display method (with or without taxes…).
roundMode	Rounding method in use.
use_taxes	Indicates whether taxes are enabled or not.

There are many other contextual hooks. If you need to display all of the current page's Smarty variables, add the following call:

```
{debug}
```

Comments are based on asterisk:

```
{* This string is commented out *}

{*
This string is too!
*}
```

Unlike with HTML comments, commented-out Smarty code is not present in the final output file.