

# How to create a simple module for PrestaShop

By Matthew Morek (aka Sandfighter). Digital property of [www.sandfighter.net](http://www.sandfighter.net). All rights reserved.



Presumably many of you have wondered how to unleash the full power of **PrestaShop**, a bright light in the dark area of e-commerce platforms. Let's have a look at the **modules** subject for PrestaShop and try to create one almost from scratch. This tutorial is of medium difficulty (OOP).

The advantage that PrestaShop has over its competition is definitely a modular structure that allows developers to create their own modules and themes that are scalable and portable. Using **SMARTY** template system makes it easier to get access to certain variables that certainly are likely to become a ground for our imagination.

The only problem with PrestaShop is its lack of solid documentation. Forum is bursting with questions to which developers are very unlikely to give answers because they simply don't have time. Recently they have been upgrading the shop, fixing all the bugs that have been reported so it's not their fault entirely. I hope that one day somebody will release a good info on PrestaShop, but until then I would like you to get cracking with creating your own modules after this tutorial.

**Warning! This tutorial requires not only basic knowledge of PHP but also Object Oriented Programming skills (OOP)!**

## The back-end module file in PrestaShop

Modules in PrestaShop are like extenders to a basic (or should I say complex) shop's functionality. Each module is stored within "**modules**" directory in its own folder. In general PrestaShop requires you to create only two files to make the module work:

- **back-end** processing script (PHP class-file);
- **front-end** display template file (for front-end modules only);

The back-end script is nothing more than a simple class file that is being recognized by the core as a backbone and all the instructions here are processed automatically to certain point. Basically when PS encounters a module file (has to have the same name as folder which the class-file is stored in) it searches for two main functions within it:

1. `__construct();`
2. `install();`

**\_\_construct()** defines all the information about the module that will be displayed in the admin panel like version number, a name or description, etc. `install()` function aims to register the basic hook the one of the parts of the page. In the front-end of PS it is possible to use these hooks:

- **hookHeader** (hooks content of the template file to heading of the page between `<head></head>` tags);
- **hookTop** (hooks content of the template file to the top of the page, above the main content);
- **hookLeftColumn** or **hookRightColumn** (hooks content of the template file to one of the side panels);
- **hookHome** (displays the content on the homepage only);

- **hookFooter** (hooks content of the file to the footer);

The structure of the module is very simple; it looks like a simple class file:

```
01.class functionName extends Module {
02.    function __construct()
03.    {
04.        $this->name = 'SampleModule';
05.        $this->tab = 'Blocks';
06.        $this->version = '1.0';
07.        Parent::__construct();
08.        $this->displayName = $this->l('displayName');
09.        $this->description = $this->l('description');
10.    }
11.    function install()
12.    {
13.        //registers the basic hook
14.    }
15.    function (hookName)($params)
16.    {
17.        //hooks the mod
18.    }
19.}
```

Below this code you can add your own functions (well, you have to), so the module would come to life. A sample of the hook function is just here:

```
01.function hookHome($params)
02.    {
03.        global $smarty;
04.        $category = new Category(1);
05.        $nb = intval(Configuration::get('HOME_FEATURED_NBR'));
06.        $products = $category->getProducts(intval($params['cookie']->id_lang), 1, ($nb ? $nb : 10),
'date_add', 'DESC');
07.        $smarty->assign(array(
08.            'allow_buy_when_out_of_stock' => Configuration::get('PS_ORDER_OUT_OF_STOCK', false),
09.            'max_quantity_to_allow_display' => Configuration::get('PS_LAST_QTIES'),
10.            'category' => $category,
11.            'products' => $products,
12.            'currency' => new Currency(intval($params['cart']->id_currency)),
13.            'lang' => Language::getIsoById(intval($params['cookie']->id_lang)),
14.            'productNumber' => sizeof($products)
15.        ));
16.        return $this->display(__FILE__, 'ourcontent.tpl');
17.    }
```

As you can see we can call PS configuration variables and use them within smarty to define variables for HTML template file so we won't have needs for using PHP directly. If you'd like to hook the content to different areas but there would be no change in the content just create a second function with different hook that refers to our main hook, like:

```
1.function hookRightColumn($params)
2.{
3.    return $this->hookHome($params);
4.}
```

To display a configuration page within admin panel we need to use a different pre-defined function called **displayForm()**. This is a function that should render our desired content on the config page for our module. It might look like this one:

```
01.public function displayForm()
02.    {
03.        $output = '
04.<form action="'.$_SERVER['REQUEST_URI'].'" method="post">
05.        <fieldset><legend>'.$this-
>l('Settings').</legend>
06.
07.'. $this->l('In order to add products to your homepage, just add them to the "home" category.').'
```

```

08.
09.         <label>'.${this->l('Number of product displayed')}.'</label>
10.<div class="margin-form">
11.<input name="nbr" size="5" value="'.Tools::getValue('nbr',
Configuration::get('HOME_FEATURED_NBR')).'" type="text">
12.
13.'.${this->l('The number of products displayed on homepage (default: 10)')}.'</div>
14.<input class="button" name="submitHomeFeatured" value="'.${this->l('Save')}.'" type="submit">
15.         </fieldset>
16.     </form>
17.
18. ';
19.     return $output;
20. }

```

In order to process the content of the config form we need to use another pre-defined function, that is **getContent()**. It will make sure that the form will be processed:

```

01.public function getContent()
02.    {
03.        $output = '
04.<h2>'.${this->displayName}.'</h2>
05.
06. ';
07.        if (Tools::isSubmit('submitHomeFeatured'))
08.            {
09.                $nbr = intval(Tools::getValue('nbr'));
10.                if (!$nbr OR $nbr <= 0 OR !Validate::isInt($nbr))                $errors[] = $this-
>l('Invalid number of product');
11.                else
12.                    Configuration::updateValue('HOME_FEATURED_NBR', $nbr);
13.                if (isset($errors) AND sizeof($errors))
14.                    $output .= $this->displayError(implode('
15.', $errors));
16.                else
17.                    $output .= $this->displayConfirmation($this->l('Settings updated'));
18.            }
19.        return $output.$this->displayForm();
20.    }

```

## TPL template module display file in PrestaShop

To conclude this tutorial it will be wise to display the content of our back-end script on the website. Since we hooked the content to the homepage the only thing we need to do is create a file with content called **“ourcontent.tpl”**:

```

01.<!-- Sample display module -->
02.<div class="sample-display">
03.<h2>Content:</h2>
04.
05.{if isset($products) AND $products}
06.<h3>{$product.name|escape:'htmlall':'UTF-8'}</h3>
07.
08.{$product.description_short|strip_tags}
09.    {/if}</div>
10.
11.<!-- /Sample Display Module -->

```

Now it's just a matter of saving the files, uploading the whole module to the server and testing it until it works. Later on I will try to come up with the tutorial about using smarty variables in your modules and themes, as this is a very important issue that needs to have some light shed on it.

Until then good luck with your modules!