

Creating a PrestaShop module

Table of contents

- Creating a PrestaShop module
 - What is a PrestaShop module
 - Technical principles behind a module
 - Modules' operating principles
 - Module file tree
 - Basic structure of a module
 - Hooking a module
 - Displaying content
 - Using Smarty
 - Module translation
 - Creating the module's back-office tab, and its class
 - A list of PrestaShop hooks
 - Troubleshooting

Creating a PrestaShop module

What is a PrestaShop module

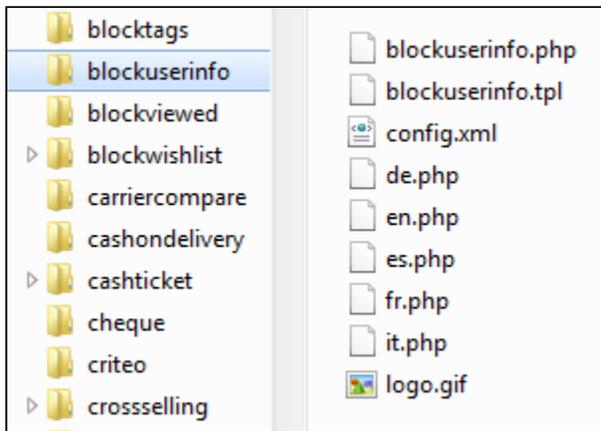
PrestaShop's extensibility revolves around modules, which are small programs that make use of PrestaShop's functionality and changes them or add to them in order to make PrestaShop easier to use or more customized.

Technical principles behind a module

A PrestaShop module consists of:

- A root folder, named after the module, which will hold all of the module's files, and will reside in PrestaShop's `/modules` folder.
- A main PHP file, named after the module, located in that root folder. This PHP file should have the same name as its root folder.
- An icon file, named `logo.gif`, representing this module.
- Optional: some `.tpl` files, containing the module's theme.
- Optional: language files, if the module or its theme have text to display (and therefore, that should be translatable).
- Optional: in a `/themes/modules` folder, a folder with the same name as the module, containing `.tpl` and language files if necessary. This last folder is essential during modifications of existing module, so that you can adapt it without having to touch its original files. Notably, it enables you to handle the module's display in various ways, according to the current theme.

Let's see an example with PrestaShop's **blockuserinfo** module:



Any PrestaShop module, once installed on an online shop, can interact with one or more "hooks". Hooks enable you to "hook" your code to the current View at the time of the code

parsing (i.e., when displaying the cart or the product sheet, when displaying the current stock...). Specifically, a hook is a shortcut to the various methods available from the Module object, as assigned to that hook.

Modules' operating principles

Modules are the ideal way to let your talent and imagination as a developer express themselves, as the creative possibilities are many.

They can display a variety of content (blocks, text, etc.), perform many tasks (batch update, import, export, etc.), interface with other tools...

Modules can be made as configurable as necessary; the more configurable it is, the easier it will be to use, and thus will be able to address the need of a wider range of users.

One of the main interest of a module is to add functionalities to PrestaShop without having to edit its core files, this making it easier to perform an update without having the transpose all core changes.

That is way you should always strive to stay away from core files when building a module, even though this can prove hard to do in some situations...

Module file tree

All PrestaShop modules are found in the `/modules` folder, which is at the root of the PrestaShop main folder. This is true for both default modules (provided with PrestaShop) and 3rd-party modules that are subsequently installed.

Each modules has its own sub-folder inside the `/modules` folder: `/bankwire`, `/birthdaypresent`, etc.

Basic structure of a module

All modules use the same basic structure, which makes it easier to learn by observing existing modules' source code.

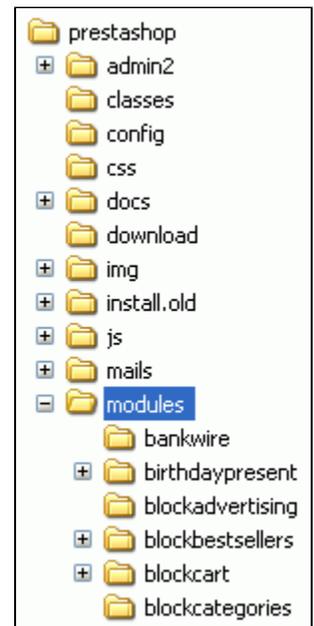
Let's create a simple first module; this will enable us to better describe its structure. We'll call it "My module".

Let's first create the module folder. It should have the same name as the module, with no space, only alphanumerical characters, the hyphen and the underscore, all in lowercase: `/mymodule`.

This folder must contain a PHP file of the same name, which will handle most of the processing: `mymodule.php`.

That is enough for a very basic module, but obviously more files and folders can complement it.

The front-office part of the module is defined in a `.tpl` file placed at the root of the module's



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

The `mymodule.php` file must start with the following test:

```
if (!defined('_PS_VERSION_'))
    exit;
```

This checks for the existence of a PHP constant, and if it doesn't exist, it quits. The sole purpose of this is to prevent visitors to load this file directly.

The file must also contain the module's class. PrestaShop uses Object-Oriented programming, and so do its modules.

That class must bear the same name as the module and its folder, in **CamelCase**: `MyModule`. Furthermore, that class must extend the `Module` class, and thus inherits all methods and attributes. It can just as well extend any class derived from `Module`: `PaymentModule`, `ModuleGridEngine`, `ModuleGraph`...

mymodule.php

```
<?php
if (!defined('_PS_VERSION_'))
    exit;

class MyModule extends Module
{
    public function __construct()
    {
        $this->name = 'mymodule';
        $this->tab = 'Test';
        $this->version = 1.0;
        $this->author = 'Firstname Lastname';
        $this->need_instance = 0;

        parent::__construct();

        $this->displayName = $this->l('My module');
        $this->description = $this->l('Description of my module. ');
    }

    public function install()
    {
        if (parent::install() == false)
            return false;
        return true;
    }
}
?>
```

Let's examine each line from our `MyModule` object...

```
public function __construct()
```

Defines the class' constructor.

```
$this->name = 'mymodule';  
$this->tab = 'Test';  
$this->version = 1.0;  
$this->author = 'PrestaShop';
```

This section assign a handful of attributes to the class instance (`this`):

- A 'name' attribute. This is an internal identifier, so make it unique, without special characters or spaces, and keep it lower-case.
- A 'tab' attribute. This is the title for the table that shall contain this module in PrestaShop's back-office modules list. You may use an existing name, such as `Products`, `Blocks` or `Stats`, or a custom, as we did here. In this last case, a new table will be created with your title.
- Version number for the module, displayed in the modules list.
- An 'author' attribute. This is displayed in the PrestaShop modules list.

```
$this->need_instance = 0;
```

The `need_instance` flag indicates whether to load the module's class when displaying the "Modules" page in the back-office. If set at 0, the module will not be loaded, and therefore will spend less resources to generate the page module. If your modules needs to display a warning message in the "Modules" page, then you must set this attribute to 1.

```
parent::__construct();
```

Calling the parent's constructor. This must be done before any use of the `$this->l()` method, and after the creation of `$this->name`.

```
$this->displayName = $this->l('My module');
```

Assigning a public name for the module, which will be displayed in the back-office's modules list. The `l()` method is part of PrestaShop translation's tools, and is explained further below.

```
$this->description = $this->l('Description of my module.');
```

Assigning a public description for the module, which will be displayed in the back-office's modules list.

```
public function install()  
{  
    return (parent::install());  
}
```

In this first and extremely simplistic incarnation, this method is useless, since all it does is check the value returned by the Module class' install() method. Moreover, if we hadn't created that method, the superclass' method would have been called instead anyway, making the end result identical.

Nevertheless, we must mention this method, because it will be very useful once we have to perform checks and actions during the module's installation process: creating SQL tables, copying files, creation configuration variables, etc.

Likewise, the module should contain an `uninstall()` method, so as to have a custom uninstallation process. This method could look as such:

```
public function uninstall()
{
    if (!parent::uninstall())
        Db::getInstance()->Execute('DELETE FROM `'.$_DB_PREFIX_.'mymodule`');
    parent::uninstall();
}
```

To put the finishing touch to this basic module, we can add an icon, which will be displayed next to the module's name in the back-office modules list.

The icon file must respect these requirements:

- 16*16 image.
- named `logo.gif`.
- placed on the module's main folder.

You can find an excellent set of free icons to pick from on [the FamFamFam website](#).

Now that all basics are in place, put the module's folder in the `/modules` folder of your PrestaShop test install, open PrestaShop, and in the "Modules" tab, under "Other Modules", you should find your module. Install it in order to be able to manage it for the rest of this guide.



PrestaShop automatically creates a small `config.xml` file in the module's folder, which stores a few configuration information. You should **NEVER** edit it by hand.

On installation, PrestaShop also adds a line to the `ps_module` SQL table.

<input type="checkbox"/>			51	statssearch	1
<input type="checkbox"/>			52	statscheckup	1
<input type="checkbox"/>			53	mymodule	1

Hooking a module

Displaying data, starting a process at a specific time: in order for a module to be "attached" to a location on the front-office or the back-office, you need to give it access to one of the many PrestaShop hooks, described earlier in this guide.

To that effect, we are going to change your module's code, and add these lines:

mymodule.php (partial)

```
public function install()
{
    if (parent::install() == false OR !$this->registerHook('leftColumn'))
        return false;
    return true;
}

...

public function hookLeftColumn( $params )
{
    global $smarty;
    return $this->display(__FILE__, 'mymodule.tpl');
}

public function hookRightColumn($params)
{
    return $this->hookLeftColumn($params);
}
```

Let's explore these new or changed lines.

```
if (parent::install() == false OR !$this->registerHook('leftColumn'))
    return false;
return true;
```

We changed the original line to add a second test.

This code checks:

- the boolean value returned by the Module class' `install()` method: if `true`, the module is installed and can be used.
- the boolean value returned by `registerHook()` for the `leftColumn` hook: if `true`, the module is indeed registered to the hook it needs, and can be used.

If any of these two boolean values is `false`, `install()` returns `false` too, and the module cannot be installed. Both values have to be `true` for the module to be considered installed.

Therefore, this line now reads this way: if installation or hooking fail, we inform PrestaShop.

```
public function hookLeftColumn($params)
{
    global $smarty;
    return $this->display(__FILE__, 'mymodule.tpl');
}
```

The `hookLeftColumn()` method makes it possible for the module to hook into the theme's left column.

`$smarty` is the global variable for the Smarty template system, which PrestaShop uses, and which we need to access.

The `display()` method returns the content of the `mymodule.tpl` template file, if it exists.

```
public function hookRightColumn($params)
{
    return $this->hookLeftColumn($params);
}
```

Likewise, `hookRightColumn()` gives access to the theme's right column. In this example, we simply call the `hookLeftColumn()` method, in order to have the very same display, whatever the column.

Save your file, and already you can hook it into the theme, move it around and transplant it: go to the "Positions" sub-tab for the "Modules" tab in the back-office, then click on the "Transplant a module" link.

In the transplantation form, find "My module" in the modules drop-down menu, then choose "Left column blocks" in the "Hook into" drop-down menu.

Transplant a module

Module : *

Hook into : *

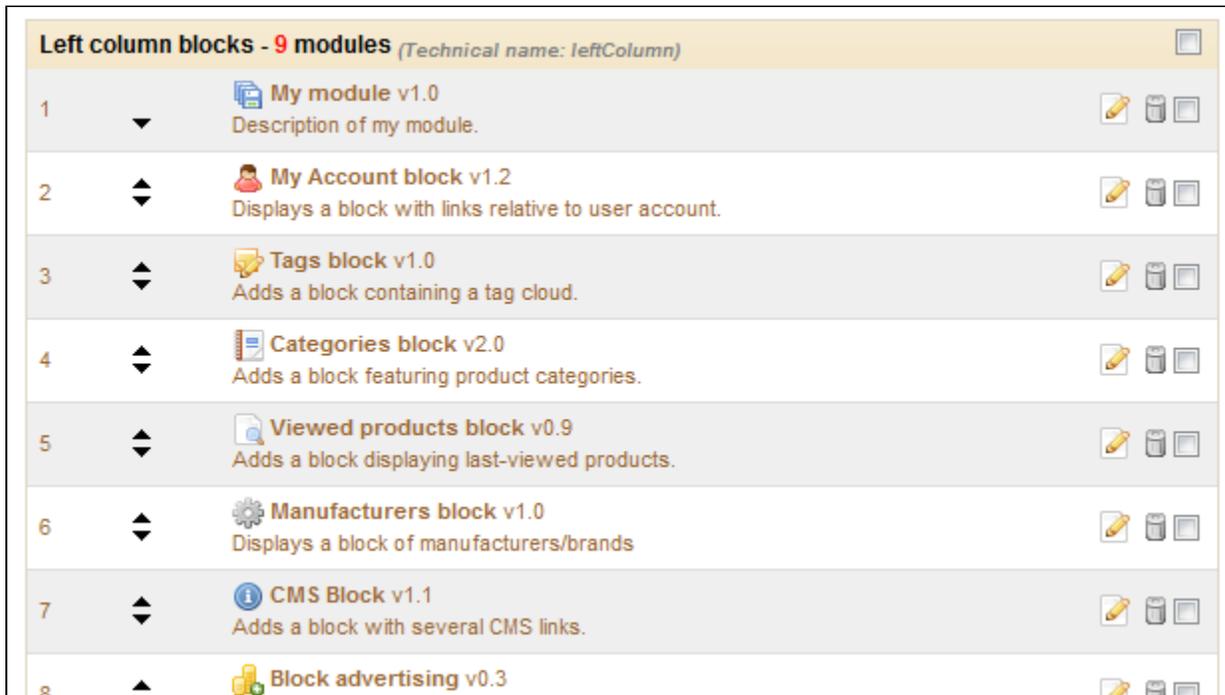
Exceptions :

Ex: identity.php, history.php, order.php, product.php

Please specify those files for which you do not want the module to be displayed. These files are located in your base directory, e.g., identity.php. Please type each filename separated by a comma.

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

Save. The "Positions" page should reload, with the following message: "Module transplanted successfully to hook". Congratulations! Scroll down, and you should indeed see your module among the other modules from the "Left column blocks" list. Move it to the top of the list.



Displaying content

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

As said earlier, the content to be displayed in the theme should be stored in `.tpl` files. We will create the `mymodule.tpl` file, which was passed as a parameter of the `display()` method in our module's code.

So, let's create the `mymodule.tpl` file, and add some lines of code to it.

```
mymodule.tpl
<!-- Block mymodule -->
<div id="mymodule_block_left" class="block">
  <h4>Welcome!</h4>
  <div class="block_content">
    <ul>
      <li><a href="{ $base_dir }modules/mymodule/mymodule_page.php" title="Click this link">Click me!</a></li>
    </ul>
  </div>
</div>
<!-- /Block mymodule -->
```

Save the file in the module's root folder, reload your shop's homepage: it should appear on top of the left column, right below the shop's logo.



The displayed link doesn't lead anywhere for now. If you need to test it, add the needed `mymodule_page.php` file in the module's folder, with a minimal content, such as "Welcome to my shop!" The resulting page will be very raw, so let's see if we can use the theme's style instead.

As you would expect, we have to create a TPL file in order to use the theme's style. Let's create the `mymodule_page.tpl` file, which will contain the basic line, and call that file from `mymodule_page.php`, which will add the theme's header, footer, etc.

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

`mymodule_page.tpl`

```
Welcome to my shop!
```

`mymodule_page.php`

```
<?php
global $smarty;
include('../..../config/config.inc.php');
include('../..../header.php');

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

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

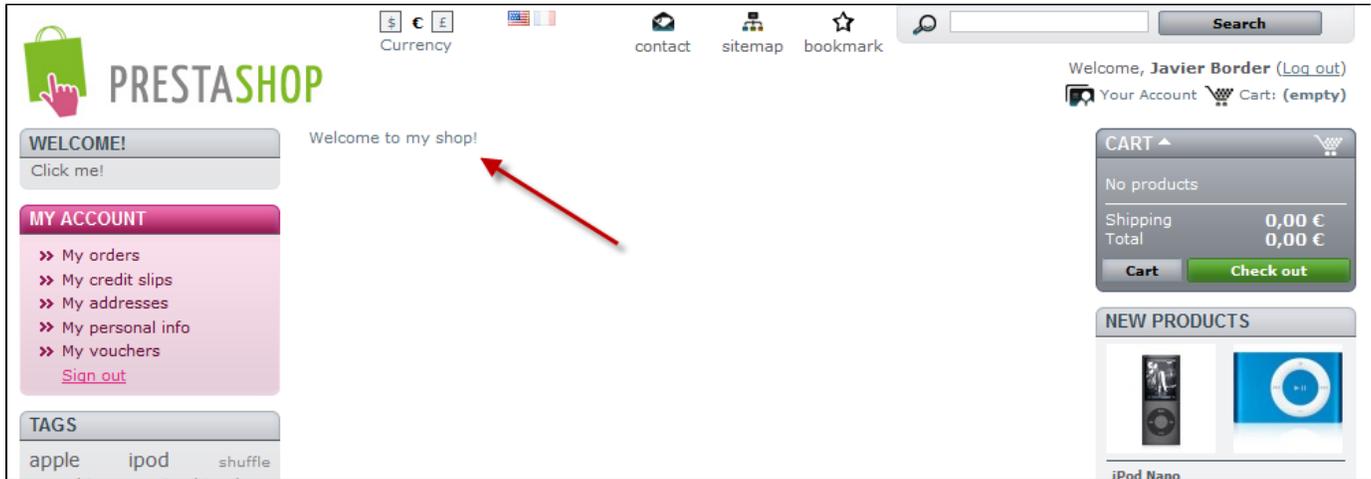
We first load the current Smarty instance. This must be done before any call to the `display()` method.

The various `include()` calls in the file enable us to load:

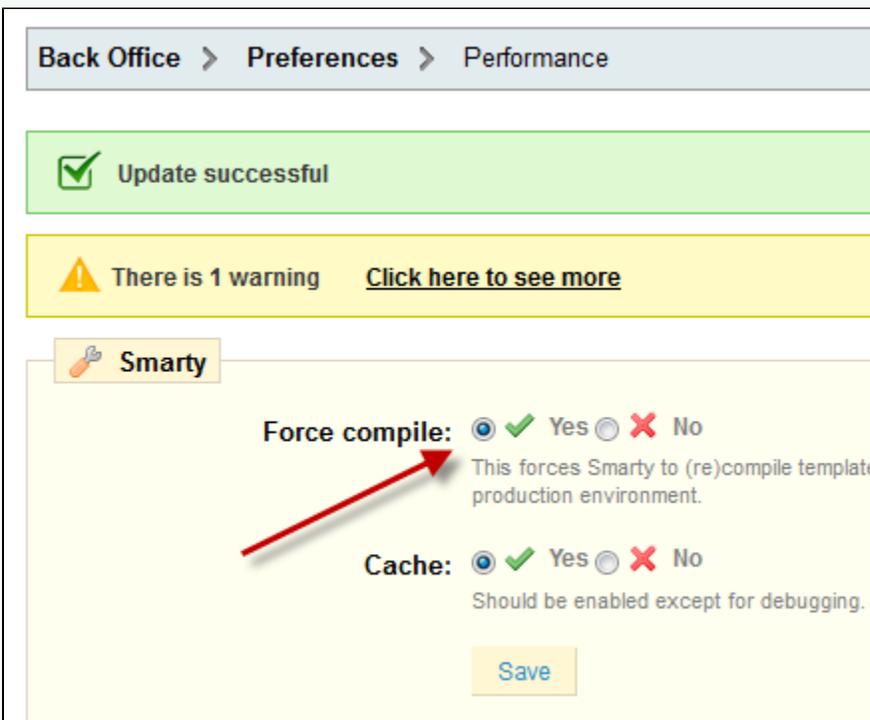
- The current PrestaShop configuration.
- The theme's header file (through `header.php`, which acts as a load file).
- the theme's footer file (through `footer.php`, which acts as a load file).

In the middle of these, we place your custom template file, whose single action will be to display the "Welcome to my shop!" line.

Save all files and reload your shop's homepage: with just a few lines, the end result is so much better, with our "Welcome" line neatly placed between header, footer and columns!



If you make multiple changes and reloads to your homepage, it may seem said changes do not apply. This is because Smarty caches a compiled version of the homepage. In order to force Smarty to recompile templates on every invocation, you must go to "Preferences" tab, its "Performance" sub-tab, and choose "Yes" for the "Force recompile" option.



Do not force recompilation on production sites, as it severely slows everything down!

Using Smarty

Smarty is a PHP template engine, and is used by PrestaShop's theming system.

It parses TPL files, looking for dynamic elements to replace by their data equivalents, then displays the generated result. Those dynamic elements are indicated with curly brackets : { ... }. The programmer may create new variables and use them in TPL files.

For instance, in our `mymodule_page.php`, we can create such a variable:

```
mymodule_page.php
<?php
global $smarty;

include('../..../config/config.inc.php');
include('../..../header.php');

$mymodule = new MyModule();
$message = $mymodule->l('Welcome to my shop!');
$smarty->assign('messageSmarty', $message); // creation of our variable
$smarty->display(dirname(__FILE__).' /mymodule_page.tpl');

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

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

```
mymodule_page.tpl
{$messageSmarty}
```

PrestaShop includes a number 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, such as: `{$mark_mymodule_product}`.

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

File / folder	Description
<code>img_ps_dir</code>	URL for the PrestaShop image folder.
<code>img_cat_dir</code>	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_qtyies	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.

If you need to have all of the current page's Smarty variables displayed, add the following function:

```
{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.

Module translation

Our module's text strings are written in English, but we might want French shop owners to use our module too. We therefore have to translate those strings into French, both front-office and back-offices ones. This could be a tedious task, but Smarty and PrestaShop's own translation make it far easier.

Strings in PHP files will need to be displayed through the `l()` method, from the `Module.php` abstract class.

mymodule.php (partial)

```
...
$this->displayName = $this->l('My module');
$this->description = $this->l('Description of my module.');
```

Strings in TPL files will need to be turned into dynamic content, which Smarty will replace by the translation for the chosen language. In our sample module, this file:

mymodule.tpl (partial)

```
<li>
  <a href="{ $base_dir }modules/mymodule/mymodule_page.php" title="Click this
link">Click me!</a>
</li>
```

...becomes:

mymodule.tpl (partial)

```
<li>
  <a href="{ $base_dir }modules/mymodule/mymodule_page.php" title="{ l s='Click this
link' mod='mymodule' }">{ l s='Click me!' mod='mymodule' }</a>
</li>
```

...and this one:

mymodule_page.tpl

```
<h4>Welcome!</h4>
...
Click me!
```

...becomes:

mymodule.tpl

```
<h4>{ l s='Welcome!' mod='mymodule' }</h4>
...
{ l s='Click me!' mod='mymodule' }
```

The translation tool needs the `mod` parameter in order to match the string to translate with its translation.

Strings are delimited with single quotes. If a string contains single quotes, they should be escaped using a backslash (`\`).

This way, strings can be directly translated inside PrestaShop: go to the "Tools" tab, its "Translations" sub-tab, and in the "Modify translations" drop-down menu, choose "Module translations", then click the French flag in order to translate modules into French.

The next page displays all the strings for all the currently-installed modules. Modules that have all their strings already translated have their fieldset closed, whereas if at least one string is missing in a module's translation, its fieldset is expanded.

In order to translate your module's strings (the ones that were "marked" using the `l ()` method), simply find your module in the list (use the browser's in-page search), and fill the empty fields.

Module: *mymodule*

default - mymodule - 4 expressions (4)

My module	=	<input type="text"/>	
Description of my module.	=	<input type="text"/>	
Click this link	=	<input type="text"/>	
Click me!	=	<input type="text"/>	

default - mymodule_page - 1 expressions (1)

Welcome to my shop!	=	<input type="text"/>	
---------------------	---	----------------------	---

Once all strings for your module are correctly translated, click on the "Update translation" button, either at the top or the bottom of the page.

Each field has an icon on its right. This enables you to get an suggestion from Google Translate. You can hover the mouse over it to see the translation, and click it to fill the field with the translation.

Automatic translation are not always accurate; use with caution.

The translations are saved in a new file, `fr.php` (or `languageCode.php`, which is generated by PrestaShop and looks like this:

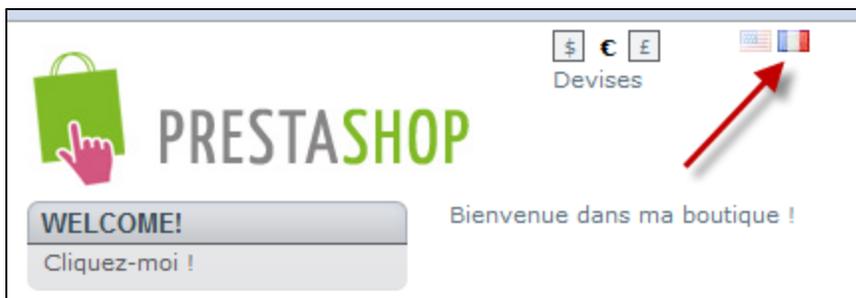
mymodule.tpl

```
<?php

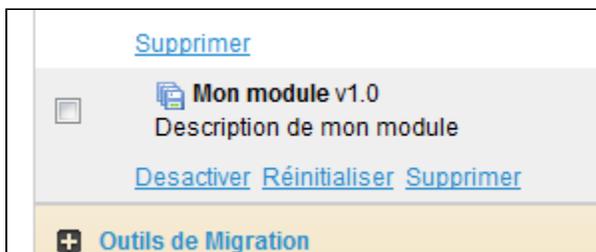
global $_MODULE;
$_MODULE = array();
$_MODULE['<{mymodule}prestashop>mymodule_2dddc2a736e4128ce1cdfd22b041e7f'] = 'Mon
module';
$_MODULE['<{mymodule}prestashop>mymodule_d6968577f69f08c93c209bd8b6b3d4d5'] =
'Description de mon module';
$_MODULE['<{mymodule}prestashop>mymodule_c66b10fbf9cb6526d0f7d7a602a09b75'] = 'Cliquez
sur ce lien';
$_MODULE['<{mymodule}prestashop>mymodule_f42c5e677c97b2167e7e6b1e0028ec6d'] =
'Cliquez-moi \!';
$_MODULE['<{mymodule}prestashop>mymodule_page_c0d7cf0105851272f83d5c1fe63a1c'] =
'Bienvenue dans ma boutique \!';
```

⚠ This file **must not** be edited manually! It can only be edited through the PrestaShop translation tool.

Now that we have a translation, we can click on the French flag in the front-office (provided the language has indeed been installed), and get the expected result: the module's strings are now in French.



They are also translated in French when the back-office is in French.



The translated strings can only be taken into account by the PrestaShop tool, the PHP and TPL files have to be located at the root of the module's folder.

Creating the module's back-office tab, and its class

In this section you will learn how to give your module its own tab or sub-tab, in a matter of

minutes.

Follow these steps:

1. Add a new table to your PrestaShop database, named `ps_test`. Give it two fields:
 - `id_test` (INT 11)
 - `test` (VARCHAR 32)
2. Create a blank file named `Test.php` in PrestaShop's `/classes` folder.
3. Add the following lines to that file:

Test.php

```
<?php
class Test extends ObjectModel
{
    /** @var string Name */
    public $test;

    protected $fieldsRequired = array('test');
    protected $fieldsSize = array('test' => 64);
    protected $fieldsValidate = array('test' => 'isGenericName');
    protected $table = 'test';
    protected $identifier = 'id_test';

    public function getFields()
    {
        parent::validateFields();
        $fields['test'] = pSQL($this->test);
        return $fields;
    }
}
?>
```

1. Create a blank file named `AdminTest.php` in PrestaShop's `/admin/tabs`.
2. Add the following lines to that file:

AdminTest.php

```
<?php
include_once( PS_ADMIN_DIR.'../classes/AdminTab.php' );

class AdminTest extends AdminTab
{
    public function __construct()
    {
        $this->table = 'test';
        $this->className = 'Test';
        $this->lang = false;
        $this->edit = true;
        $this->delete = true;
        $this->fieldsDisplay = array(
            'id_test' => array(
                'title' => $this->l('ID'),
                'align' => 'center',
                'width' => 25),
            'test' => array(
                'title' => $this->l('Name'),
```

```

        'width' => 200)
    );

    $this->identifier = 'id_test';

    parent::__construct();
}

public function displayForm()
{
    global $currentIndex;

    $defaultLanguage = intval(Configuration::get('PS_LANG_DEFAULT'));
    $languages = Language::getLanguages();
    $obj = $this->loadObject(true);

    echo '
        <script type="text/javascript">
            id_language = Number('.$defaultLanguage.');
        </script>';

    echo '
        <form action="' . $currentIndex . '&submitAdd' . $this->table . '=1&token=' .
    $this->token . '" method="post" class="width3">
            ' . ($obj->id ? '<input type="hidden" name="id_' . $this->table . '" value="'
    . $obj->id . '" />' : '').'
            <fieldset><legend>' .
    $this->l('Profiles') . '</legend>
            <label>'.$this->l('Name:').' </label>
            <div class="margin-form">;
            foreach ( $languages as $language )
                echo '
                    <div id="name_' . $language['id_lang']|'id_lang'] . '" style="display: ' .
    ($language['id_lang']|'id_lang'] == $defaultLanguage ? 'block' : 'none') . '; float:
    left;">
                        <input size="33" type="text" name="name_' . $language['id_lang']|'id_lang']
    . '" value="' . htmlentities( $this->getFieldValue( $obj, 'name', intval(
    $language['id_lang']|'id_lang'] ) ), ENT_COMPAT, 'UTF-8' ) . '" /><sup>*</sup>
                    </div>;
                $this->displayFlags( $languages, $defaultLanguage, 'name', 'name' );
            echo '
                <div class="clear"></div>
            </div>
            <div class="margin-form">
                <input type="submit" value="' . $this->l('Save')."
    name="submitAdd' . $this->table . '" class="button" />
            </div>
            <div class="small"><sup>*</sup> '.$this->l('Required field')." </div>
            </fieldset>
        </form> ';
}

```

```

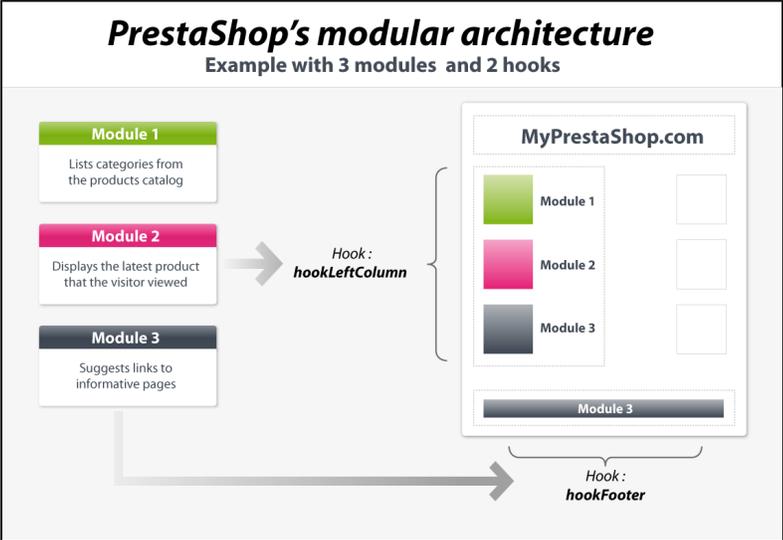
}
?>

```

Put the files online, then create the tab by going to the "Employee" tab, then its "Tabs" sub-tab. Click the "Add new" button, and fill-in the fields with the class' name, "AdminTest". Do not confuse "class" with "modules"! Choose an icon (like one from the [FamFamFam pack](#)), choose where the tab should go, and save. You're set! Now start customizing it to your needs!

A list of PrestaShop hooks

Here's a recap of PrestaShop's module architecture:



When one of the site's page is loaded, the PrestaShop engine check which are the modules to call for each of the hooks that make up the page.

Here is a list of 53 hooks, available in PrestaShop.

Front-office

Homepage and general website items

Hook name	File location	Visible	Description
header	header.php	No	Called between the HEAD tags. Ideal location for adding JavaScript and CSS files.
top	header.php	Yes	Called in the page's header.
leftColumn	header.php	Yes	Called when loading the left column.
rightColumn	footer.php	Yes	Called when loading the right column.
footer	footer.php	Yes	Called in the page's footer.
home	index.php	Yes	Called at the center of the homepage.

Product sheet

Hook name	File location	Visible	Description
extraLeft	product.php	Yes	Called right before the "Print" link, under the picture.
extraRight	product.php	Yes	Called right after the block for the "Add to Cart" button.
productActions	product.php	Yes	Called inside the block for the "Add to Cart" button, right after that button.
productOutOfStock	product.php	Yes	Called inside the block for the "Add to Cart" button, right after the "Availability" information.
productfooter	product.php	Yes	Called right before the tabs.
productTab	product.php	Yes	Called in tabs list, such as "More info", "Data sheet", "Accessories"... Ideal location for one more tab, the content of which is handled by the <code>productTabContent</code> hook.
productTabContent	product.php	Yes	Called when a tab is clicked. Ideal location for the content of a tab that has been defined using the <code>productTab</code> hook.

Cart

Hook name	File location	Visible	Description
cart	Class: Cart.php	No	Called right after a cart creation or update.
shoppingCart	order.php	Yes	Called right below the cart items table.
shoppingCartExtra	order.php	Yes	Called after the cart's table of items, right above the navigation buttons.
createAccountTop	authentication.php	Yes	Called within the client account creation form, right above the the "Your personal information" block.
createAccountForm	authentication.php	Yes	Called within the client account creation form, right before the "Register" button.
createAccount	authentication.php	No	Called right after the client account creation.
customerAccount	my-account.php	Yes	Called on the client account homepage, after the list of available links. Ideal location to add a link to this list.
myAccountBlock	Module: blockmyaccount.php	Yes	Called within the "My account" block, in the left column, below the list of available links. Ideal location to add a link to this list.
authentication	authentication.php	No	Called right after the client identification, only if the authentication is valid (e-mail address and password are both OK).

Search

Hook name	File location	Visible	Description
search	Class: Search.php	No	Called after a search is performed. Ideal location to parse and/or handle the search query and results.

Carrier choice

Hook name	File location	Visible	Description
extraCarrier	order.php	Yes	Called after the list of available carriers, during the order process. Ideal location to add a carrier, as added by a module.

Payment

Hook name	File location	Visible	Description
payment	order.php	Yes	Called when needing to build a list of the available payment solutions, during the order process. Ideal location to enable the choice of a payment module that you have developed.
paymentReturn	order-confirmation.php	Yes	Called when the user is sent back to the store after having paid on the 3rd-party website. Ideal location to display a confirmation message or to give some details on the payment.
orderConfirmation	order-confirmation.php	Yes	A duplicate of <code>paymentReturn</code> .
backBeforePayment	order.php	No	Called when displaying the list of available payment solutions. Ideal location to redirect the user instead of displaying said list (i.e., 1-click PayPal checkout)..

Merchandise Returns

Hook name	File location	Visible	Description
orderReturn	order-follow.php	No	Called when the customer request to send his merchandise back to the store, and if now error occurs.
PDFInvoice	Class: PDF.php	Yes	Called when displaying the invoice in PDF format. Ideal location to display dynamic or static content within the invoice.

Back-office

General

Hook name	File location	Visible	Description
backOfficeTop	header.inc.php	Yes	Called within the header, above the tabs.

backOfficeHeader	header.inc.php	No	Called between the HEAD tags. Ideal location for adding JavaScript and CSS files.
backOfficeFooter	footer.inc.php	Yes	Called within the page footer, above the "Power By PrestaShop" line.
backOfficeHome	index.php	Yes	Called at the center of the homepage.

Orders and order details

Hook name	File location	Visible	Description
newOrder	Class: PaymentModule.php	No	Called during the new order creation process, right after it has been created.
paymentConfirm	Class: Hook.php	No	Called when an order's status becomes "Payment accepted".
updateOrderStatus	Class: OrderHistory.php	No	Called when an order's status is changed, right before it is actually changed.
postUpdateOrderStatus	Class: OrderHistory.php	No	Called when an order's status is changed, right after it is actually changed.
cancelProduct	AdminOrders.php	No	Called when an item is deleted from an order, right after the deletion.
invoice	AdminOrders.php	Yes	Called when the order's details are displayed, above the Client Information block.
adminOrder	AdminOrders.php	Yes	Called when the order's details are displayed, below the Client Information block.
orderSlip	AdminOrders.php	No	Called during the creation of a credit note, right after it has been created.

Products

Hook name	File location	Visible	Description
addproduct	AdminProducts.php	No	Called when a product is created or duplicated, right after said creation/duplication.
updateproduct	AdminProducts.php	No	Called when a product is update with a new picture, right after said update.
deleteproduct	Class: Product.php	No	Called when a product is deleted, right before said deletion..
updateQuantity	Class: PaymentModule.php	No	Called during an the validation of an order, the status of which being something other than "canceled" or "Payment error", for each of the order's items.
updateProductAttribute	Class: Product.php	No	Called when a product declination is updated, right after said update.

watermark	AdminProducts.php	No	Called when an image is added to an product, right after said addition.
-----------	-------------------	----	---

Statistics

Hook name	File location	Visible	Description
GraphEngine	Class: ModuleGraph.php	Yes	Called when a stats graph is displayed.
GridEngine	Module: GridEngine.php	Yes	Called when the grid of stats is displayed.
AdminStatsModules	AdminStatsTab.php	Yes	Called when the list of stats modules is displayed.

Clients

Hook name	File location	Visible	Description
adminCustomers	AdminCustomers.php	Yes	Called when a client's details are displayed, right after the list of the clients groups the current client belongs to.

Carriers

Hook name	File location	Visible	Description
updateCarrier	AdminCarriers.php	No	Called during a carrier's update, right after said update.

Troubleshooting

If your module does not work as expected, here are a few ways to find help.

PrestaShop official forum

Join our forum at <http://www.prestashop.com/forums/>, and search for an answer using the relevant keywords. If your search needs refining, use the advanced search form. And if your search doesn't yield anything useful, create a new thread, where you can be as wordy as necessary when writing your question; you will need to registered first.

Some forums keep certain threads pinned on top of all threads; they contain some useful information, so be sure to read them through.

Our bug-tracker

If it turns out your issue stems from a PrestaShop bug rather than your code, please do submit the issue in the PrestaShop bug-tracker: <http://forge.prestashop.com/> (you will need to register). This enables you to discuss the issue directly with the PrestaShop developers.

Official PrestaShop websites

URL	Description
http://www.prestashop.com	Official website for the PrestaShop tool, its community, and the company behind it.
http://addons.prestashop.com	Marketplace for themes and modules
http://www.prestabox.com	Host your shop with us!