

What is a PrestaShop module?

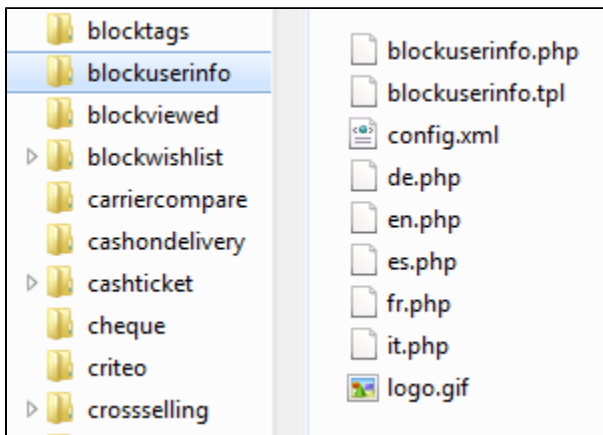
What is a PrestaShop module?

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 main PHP file with as many other PHP files as needed, and all the images and template (.tpl) files necessary to display the information.

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/attach 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, etc.). 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, and much much more.

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 advantages of a module is to add functionalities to PrestaShop without having to edit its core files, thus making it easier to perform an update of PrestaShop without having to transpose all core changes. Indeed, you should always strive to stay away from core files when building a module, even though this may seem necessary in some situations.

The PrestaShop file structure

The PrestaShop developers have done their best to clearly and intuitively separate the various parts of the software.

Here is how the files are organized:

- `/admin` (the name is customized on installation): contains all the PrestaShop files pertaining to the back-office. When accessing this folder with your browser, you will be asked to provide proper identification, for security reasons.
Important: you should make sure to protect that folder with a `.htaccess` or `.htpasswd` file!
- `/cache`: contains temporary folders that are generated and re-used in order to alleviate the server's load.
- `/classes`: contains all the files pertaining to PrestaShop's object model (some are used for the front-office, others for the back-office). Each file represents (and contains) a PHP class, and its methods/properties.
- `/config`: contains all of PrestaShop's configuration files. Unless asked to, you should **never** edit them, as they are directly handled by PrestaShop's installer and back-office.
- `/controllers`: contains all the files pertaining to PrestaShop controllers – as in Model-View-Controller (or MVC), the software architecture used by PrestaShop. Each file controls a specific part of PrestaShop.
- `/css`: contains all the CSS files that are not attached to themes – hence, these are mostly used by the PrestaShop back-office.

- `/docs`: contains some documentation, the licenses, and the sample import files.
Note: it should be deleted in a production environment.
- `/download`: contains your virtual products, which can be downloaded by the user who paid for it. Files are stored with a md5 filename.
- `/img`: contains all of PrestaShop's default images, icons and picture files – that is, those that do not belong to the theme. This is where you can find the pictures for product categories (`/c` sub-folder), those for the products (`/p` sub-folder), and those for the back-office itself (`/admin` sub-folder).
- `/install`: contains all the files related to PrestaShop's installer. You will be required to delete it after installation, in order to increase security.
- `/js`: contains all JavaScript files that are not attached to themes. Most of them belong to the back-office. This is also where you will find the jQuery framework.
- `/localization`: contains all of PrestaShop's localization files – that is, files that contain local information, such as currency, language, tax rules and tax rule groups, states and the various units in use in the chosen country (i.e., volume in liter, weight in kilograms, etc.).
- `/log`: contains the log files generated by PrestaShop at various stages, for instance during the installation process.
- `/mails`: contains all HTML and text files related to e-mails sent by PrestaShop. Each language has its specific folder, where you can manually edit their content if you wish. PrestaShop contains a tool to edit your e-mails, located in the back-office, in the Localization > Translation page.
- `/modules`: contains all of PrestaShop's modules, each in its own folder. If you wish to definitely remove a module, first uninstall it from the back-office, then only should you delete its folder.
- `/override`: this is a special folder that appeared with PrestaShop 1.4. By using PrestaShop's regular folder/filename convention, it is possible to create files that override PrestaShop's default classes or controllers. This enables you to change PrestaShop core behavior without touching to the original files, keeping them safe for the next update.
- `/pdf`: contains all the template files (`.tpl`) pertaining to the PDF file generation (invoice, delivery slips, etc.). Change these files in order to change the look of the PDF files that PrestaShop generates.
- `/themes`: contains all the currently-installed themes, each in its own folder.
- `/tools`: contains external tools that were integrated into PrestaShop. For instance, this were you'll find Smarty (template/theme engine), TCPDF (PDF file generator), Swift (mail sender), PEAR XML Parser (PHP tool), etc.
- `/translations`: contains a sub-folder for each available language. However, if you wish to change the translation, you must do so using the PrestaShop internal tool, and **not** edit them directly in this folder.
- `/upload`: contains the files that would be uploaded by clients for customizable products (for instance, a picture that a client wants printed on a mug).
- `/webservice`: contains files that enable third-party applications to access PrestaShop through its API.

Modules folder

PrestaShop's 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.

Modules can also be part of a theme if they are really specific to it. In that case, they would be in the theme's own `/modules` folder, and therefore under the following path: `/themes/[my-theme]/modules`

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

About the cache

The `/cache/class_index.php` file contains the link between the class and the declaration file. If there is a caching issue, this file can safely be deleted.

The `/config/xml` folder contains the list of all the base modules:

- `default_country_modules_list.xml`
- `must_have_modules_list.xml`
- `tab_modules_list.xml`

When the store's front-end doesn't quite reflect your changes and emptying the browser's cache is not effective, you should try emptying the following folders:

- `/cache/smarty/cache`
- `/cache/smarty/compile`

Creating a first module

Creating a first module

- Creating a first module
 - File structure for a PrestaShop module
 - The PrestaShop coding convention
 - Creating a first module
 - The constant test
 - The main class
 - The constructor method
 - Building the install() and uninstall() methods
 - The install() method
 - The uninstall() method
 - The Configuration object
 - The main methods
 - Retrieving external values from the ps_configuration data table
 - The Shop object
 - The icon file
 - Installing the module

File structure for a PrestaShop module

A module is made of a lot of files, all stored in a folder that bears the same name as the module, that folder being in turn stored in the `/modules` folder at the root of the main PrestaShop folder: `/modules/name_of_the_module/`.

Here are the possible files and folders for a PrestaShop 1.6 module:

File/folder name	Description	Details
<code>name_of_the_module.php</code>	Main file.	The main PHP file should have the same name as the module's root folder. For instance, for the BlockCMS module: <ul style="list-style-type: none">• Folder name: <code>/modules/blockcms</code>• Main file name: <code>/modules/blockcms/blockcms.php</code>
<code>config.xml</code>	Cache configuration file.	If it does not exist yet, this file is automatically generated by PrestaShop when the module is first installed.
<code>logo.gif</code> or <code>logo.jpg</code> (up to v1.4) <code>logo.png</code> (v1.5+)	Icon files representing this module in the back-office.	PrestaShop 1.4: 16*16 pixels Gif or Jpeg file. PrestaShop 1.5: 32*32 pixels PNG file. If your module works on both PrestaShop 1.4 and PrestaShop 1.5+, you should have both a <code>logo.gif</code> AND a <code>logo.png</code> file.
<code>/views</code>	This folder contains the View files.	
<code>/views/templates</code>	This folder contains your module's template files (<code>.tpl</code>).	If the module needs to work with PrestaShop 1.4, the template files should be placed either directly at the root of the module's folder, or in a <code>template</code> folder at the root of the module's folder.
<code>/views/templates/admin</code>	Sub-folder for template files used by the module's administration controllers.	
<code>/views/templates/front</code>	Sub-folder for template files used by the module's front-office controllers.	
<code>/views/templates/hook</code>	Sub-folder for template files used by the module's hooks.	
<code>/css</code>	Sub-folder for CSS files used.	

/js	Sub-folder for JavaScript files.	
/img	Sub-folder for image files.	
/controllers	This folder contains the Controller files.	You can use the same sub-folder paths as for the View files. For instance, <code>/modules/bankwire/controllers/front/payment.php</code> .
/override	Sub-folder for the class-overriding code.	This is very useful when you need to change some of the default PrestaShop code. Since you must not do so, you can override the default code. For instance, <code>/modules/gsitemap/override/classes/Shop.php</code> extends the default <code>ShopCore</code> class.
/translations	Sub-folder for the translation files.	<code>fr.php</code> , <code>en.php</code> , <code>es.php</code> , etc.
/themes/[theme_name]/modules	Sub-folder for overriding .tpl files and languages files, if necessary.	This folder is essential during modifications of an existing module, so that you can adapt it without having to touch its original files. Notably, it enables you to handle the module's template files in various ways, depending on the current theme.
/upgrade	Sub-folder for upgrade files	When releasing a new version of the module, the older might need an upgrade of its data or files. This can be done using this folder.

Only the three first are necessary for a basic module: the main file, the cache configuration file (which is autogenerated by PrestaShop anyway) and the icon file. All the other ones can be used if necessary, but a module can work without them. The module can also as many other files and folders as necessary: `/css`, `/img`, `/js`, etc.



If wish to use an external library, it should be put in a dedicated folder.

That folder can use one of these names: 'lib', 'libs', 'libraries', 'sdk', 'vendor', 'vendors'.

Choose the most appropriate one for your library (indeed, 'libraries' doesn't not have the same meaning as 'sdk'). You can have more than one such folder, for instance `/lib` and `/sdk`.

The PrestaShop coding convention

Before you start writing code for your PrestaShop module, you should be aware that the PrestaShop team uses a specific set of coding convention (or coding standards, coding norm, etc.).

As Wikipedia puts it, "Coding conventions are a set of guidelines for a specific programming language that recommend programming style, practices and methods for each aspect of a piece program written in this language", and "Good procedures, good methodology and good coding standards can be used to drive a project such that the quality is maximized and the overall development time and development and maintenance cost is minimized." (See http://en.wikipedia.org/wiki/Coding_conventions).

PrestaShop's own standards is available at this page: <http://doc.prestashop.com/display/PS16/Coding+Standards>. You should read it in order to make sure that the code you produce fits correctly with the overall code of the PrestaShop project. The PHPCodeSniffer can help you make sure you follow the standard properly.

Creating a first module

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

First, create the module's folder, in the `/modules` 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 the main file, a PHP file of the same name as the folder, which will handle most of the processing: `mymodule.php`.



That is enough for a very basic module, but obviously more files and folders can be added later.

The constant test

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

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

This checks for the existence of an always-existing PrestaShop constant (its version number), and if it does not exist, it stops the module from loading. The sole purpose of this is to prevent malicious visitors to load this file directly.

Note that, as required by the PrestaShop Coding Standards (see above), we do not use a PHP closing tag.

The main class

The main file must contain the module's main class (along with other classes if needed). 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 (see <http://en.wikipedia.org/wiki/CamelCase>). In our example: MyModule.

Furthermore, that class must extend the `Module` class, in order to inherit all its methods and attributes.

mymodule.php

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

class MyModule extends Module
{
}
```



It can just as well extend any class derived from `Module`, for specific needs: `PaymentModule`, `ModuleGridEngine`, `ModuleGraph`, etc.

At this stage, if you place the module's folder on the `/modules` folder, the module can already be seen in the "Modules" page in the back-office, in the "Other modules" section – albeit with no real name nor thumbnail.

The constructor method

Now, let's fill the class' code block with the essential constructor lines. A constructor is a function in a class that is automatically called when you create a new instance of a class with `new`. In the case of a PrestaShop, the constructor class is the first method to be called when the module is loaded by PrestaShop. This is therefore the best place to set most of its details.

mymodule.php

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

class MyModule extends Module
{
    public function __construct()
    {
        $this->name = 'mymodule';
        $this->tab = 'front_office_features';
        $this->version = '1.0.0';
        $this->author = 'Firstname Lastname';
        $this->need_instance = 0;
        $this->ps_versions_compliancy = array('min' => '1.6', 'max' => _PS_VERSION_);
        $this->bootstrap = true;

        parent::__construct();

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

```
        $this->confirmUninstall = $this->l('Are you sure you want to uninstall?');

        if (!Configuration::get('MYMODULE_NAME'))
            $this->warning = $this->l('No name provided');
    }
}
```

Let's examine each line from this first version of the MyModule class...

```
public function __construct()
```

This line defines the class' constructor function.

```
$this->name = 'mymodule';
$this->tab = 'front_office_features';
$this->version = '1.0';
$this->author = 'Firstname Lastname';
```

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

- **'name' attribute.** This attribute serves as an internal identifier. The value **MUST** be the name of the module's folder. Do not use special characters or spaces, and keep it lower-case.
- **'tab' attribute.** The title for the section that shall contain this module in PrestaShop's back-office modules list. You may use an existing name, such as `seo`, `front_office_features` or `analytics_stats`, or a custom one. In this last case, a new section will be created with your identifier. We chose `"front_office_features"` because this first module will mostly have an impact on the front-end.

Here is the list of available "Tab" attributes, and their corresponding section in the "Modules" page:

"Tab" attribute	Module section
administration	Administration
advertising_marketing	Advertising & Marketing

analytics_stats	Analytics & Stats
billing_invoicing	Billing & Invoices
checkout	Checkout
content_management	Content Management
dashboard	Dashboard
emailing	E-mailing
export	Export
front_office_features	Front Office Features
il18n_localization	I18n & Localization
market_place	Market Place
merchandizing	Merchandizing
migration_tools	Migration Tools
mobile	Mobile
others	Other Modules
payments_gateways	Payments & Gateways
payment_security	Payment Security
pricing_promotion	Pricing & Promotion
quick_bulk_update	Quick / Bulk update
search_filter	Search & Filter
seo	SEO
shipping_logistics	Shipping & Logistics
slideshows	Slideshows
smart_shopping	Smart Shopping
social_networks	Social Networks

- **'version' attribute.** The version number for the module, displayed in the modules list. It is a string, so that you may use such variation as "1.0b", "3.07 beta 3" or "0.94 (not for production use)".
- **'author' attribute.** This is displayed as-is in the PrestaShop modules list.

Let's continue with the next line in this block of code:

```
$this->need_instance = 0;
$this->ps_versions_compliancy = array('min' => '1.5', 'max' => '1.6');
$this->bootstrap = true;
```

This section handles the relationship with the module and its environment (namely, PrestaShop):

- **need_instance.** 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 "Modules" page. If your module needs to display a warning message in the "Modules" page, then you must set this attribute to 1.
- **ps_versions_compliancy.** Indicates which version of PrestaShop this module is compatible with. In the example above, we explicitly write that this module will only work with PrestaShop 1.5.x, and no other major version.
- **bootstrap.** Indicates that the module's template files have been built with PrestaShop 1.6's bootstrap tools in mind – and therefore, that PrestaShop should not try to wrap the template code for the configuration screen (if there is one) with helper tags.

Next, we call the constructor method from the parent PHP class:

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

This will trigger a lot of actions from PrestaShop that you do not need to know about at this point.

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

The next section deals with text strings, which are encapsulated in PrestaShop's translation method, `l()`:

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



```
$this->confirmUninstall = $this->l('Are you sure you want to uninstall?');
```



```
if (!Configuration::get('MYMODULE_NAME'))
    $this->warning = $this->l('No name provided.');
```

These lines respectively assign:

- A name for the module, which will be displayed in the back-office's modules list.
- A description for the module, which will be displayed in the back-office's modules list.
- A message, asking the administrator if he really does want to uninstall the module. To be used in the installation code.
- A warning that the module doesn't have its `MYMODULE_NAME` database value set yet (this last point being specific to our example, as we will see later).

The constructor method is now complete. You are free to add more to it later if necessary, but this the bare minimum for a working module.

Now go to your back-office, in the Modules page: the module is visible in the modules list, with its information displayed – and no icon for now.

You can install the module, but it does not do anything yet.



When you click on the "Install" button for your module, it will display a module window saying that your module is Untrusted.

The only way to make your module Trusted is to distribute it through the PrestaShop Addons marketplace (with a unique identifying key), or to become a PrestaShop partner. Other trusted modules are the native ones.

To install the module, click the "Proceed with installation" on this screen.

Building the install() and uninstall() methods

Some modules have more needs than just using PrestaShop's features in special ways. Your module might need to perform actions on installation, such as checking PrestaShop's settings or to registering its own settings in the database. Likewise, if you changed things in the database on installation, it is highly recommended to change them back (or remove them) when uninstalling the module.

The `install()` and `uninstall()` methods make it possible to control what happens when the store administrator installs or uninstalls the module. They must be included in the main class' block of code (in our example, the `MyModule` class) – at the same level as the constructor method.

The install() method

Here is the bare minimum for the `install()` method:

```
public function install()
{
    if (!parent::install())
        return false;
    return true;
}
```

In this first and extremely simplistic incarnation, this method does the minimum needed: `return true` returned by the `Module` class' `install()` m

ethod, which returns either `true` if the module is correctly installed, or `false` otherwise. As it is, if we had not 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.

So for example how you can expand the `install()` method to perform installation checks. In the following example, we perform the following tasks during installation:

- Check that the Multistore feature is enabled, and if so, set the current context to all shops on this installation of PrestaShop.
- Check that the module parent class is installed.
- Check that the module can be attached to the `leftColumn` hook.
- Check that the module can be attached to the `header` hook.
- Create the `MYMODULE_NAME` configuration setting, setting its value to "my friend".

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

    if (!parent::install() ||
        !$this->registerHook('leftColumn') ||
        !$this->registerHook('header') ||
        !Configuration::updateValue('MYMODULE_NAME', 'my friend')
    )
        return false;

    return true;
}
```

If any of the lines in the testing block fails, the method returns `false` and the installation does not happen.

The uninstall() method

Here is the bare minimum for the `uninstall()` method:

```
public function uninstall()
{
    if (!parent::uninstall())
        return false;
    return true;
}
```

Building on this foundation, we want an `uninstall()` method that would delete the data added to the database during the installation (`MYMODULE_NAME` configuration setting). This method would look like this:

```
public function uninstall()
{
    if (!parent::uninstall() ||
        !Configuration::deleteByName('MYMODULE_NAME')
    )
        return false;

    return true;
}
```

The Configuration object

As you can see, our three blocks of code (`__construct()`, `install()` and `uninstall()`) all make use of a new object, `Configuration`.

This is a PrestaShop-specific object, built to help developers manage their module settings. It stores these settings in PrestaShop's database without require to use SQL queries. Specifically, this object handles data from the `ps_configuration` database table.

The main methods

So far, we've used three methods, to which we'll add a fourth one in the list below:

- `Configuration::get('myVariable')`: retrieves a specific value from the database.
- `Configuration::getMultiple(array('myFirstVariable', 'mySecondVariable', 'myThirdVariable'))`: retrieves several values from the database, and returns a PHP array.
- `Configuration::updateValue('myVariable', $value)`: updates an existing database variable with a new value. If the variable does not yet exist, it creates it with that value.
- `Configuration::deleteByName('myVariable')`: deletes the database variable.

There are many more, such as `getInt()` or `hasContext()`, but these four are the ones you will use the most.

Note that when using `updateValue()`, the content of `$value` can be anything, be it a string, a number, a serialized PHP array or a JSON object. As long as you properly code the data handling function, anything goes. For instance, here is how to handle a PHP array using the `Configuration` object:

```
// Storing a serialized array.
Configuration::updateValue('MYMODULE_SETTINGS', serialize(array(true, true, false)));

// Retrieving the array.
$configuration_array = unserialize(Configuration::get('MYMODULE_SETTINGS'));
```

As you can see, this is a very useful and easy-to-use object, and you will certainly use it in many situations. Most native modules use it too for their own settings.



Multistore

By default, all these methods work within the confines of the current store context, whether PrestaShop is using the multistore feature or not.

However, it is possible to work outside of the current context and impact other known stores. This is done using three optional parameters, which are not presented in the list above:

- `id_lang`: enables you to force the language with which you want to work.
- `id_shop_group`: enables you to indicate the shop group of the target store.
- `id_shop`: enables you to indicate the id of the target store.

By default, these three parameters use the values of the current context, but you can use them to target other stores.

Note that it is not recommended to change the default values of these variables, even more so if the module you are writing is to be used on other stores than your own. They should only be used if the module is for your own store, and you know the id and shop group of all of your shops.

Retrieving external values from the `ps_configuration` data table

You are not limited to your own variables: PrestaShop stores all its own configuration settings in the `ps_configuration` table. There are literally hundreds of settings, and you can access them just as easily as you would access your own. For instance:

- `Configuration::get('PS_LANG_DEFAULT')`: retrieves the ID for the default language.
- `Configuration::get('PS_TIMEZONE')`: retrieves the name of the current timezone, in standard TZ format (see: http://en.wikipedia.org/wiki/List_of_tz_database_time_zones).
- `Configuration::get('PS_DISTANCE_UNIT')`: retrieves the default distance unit ("km" for kilometers, etc.).
- `Configuration::get('PS_SHOP_EMAIL')`: retrieves the main contact e-mail address.
- `Configuration::get('PS_NB_DAYS_NEW_PRODUCT')`: retrieves the number of days during which a newly-added product is considered "New" by PrestaShop.

Dive into the `ps_configuration` table in order to discover many other settings!

The Shop object

Another of `install()`'s lines is thus:

```
if (Shop::isFeatureActive())
    Shop::setContext(Shop::CONTEXT_ALL);
```

As said earlier, here we check that the Multistore feature is enabled, and if so, set the current context to all shops on this installation of PrestaShop.

The Shop object helps you manage the multistore feature. We will not dive in the specifics here, but will simply present the two methods that are used in this sample code:

- `Shop::isFeatureActive()`: This simply checks whether the multistore feature is active or not, and if at least two stores are presently activated.
- `Shop::setContext(Shop::CONTEXT_ALL)`: This changes the context in order to apply coming changes to all existing stores instead of only the current store.

The Context is explained in more details in the "Using the Context Object" chapter of this Developer Guide.

The icon file

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

In case your module is made for a prominent service, having that service's logo visible brings trust.





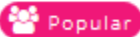
Make sure you do not use a logo already used by one of the native modules, or without authorization from the owner of the logo/service.

The icon file must respect these requirements:

- It must be placed on the module's main folder.
- 32*32 PNG image.
- Named `logo.png`.
- Tip: There are many free 32*32 icon libraries available. Here are a few: <http://www.fatcow.com/free-icons> (very close to the FamFamFam one) or <http://www.iconarchive.com/show/danish-royalty-free-icons-by-jonas-rask.html> (Danish Royalty Free),

Installing the module






















Now that all basics are in place, reload the back-office's "Modules" pages, in the "Front-office features" section, you should find your module. Install it (or reset it if it is already installed).

	<p>Front Office Features</p> <p>My Account block v1.3 - by PrestaShop</p> <p>Displays a block with links relative to a user's account.</p>
	<p>Front Office Features</p> <p>My account block for your website's footer v1.4 - by PrestaShop</p> <p>Displays a block with links relative to user accounts.</p>
	<p>Front Office Features</p> <p>My module v1.0 - by Firstname Lastname</p> <p>Description of my module.</p>
	<p>Advertising and Marketing</p> <p>Newsletter & Statistics v4.0.22 - by PrestaShop - </p> <p>There's no better way to keep in touch with customers and up your conversion rates and increase customer Newsletter and Statistics module! This brilliant module makes it easy</p> <p>Read more</p>

During the module's installation, PrestaShop automatically creates a small `config.xml` file in the module's folder, which stores the configuration information. You should be very careful when editing by hand.

PrestaShop also adds a row to the `ps_module` SQL table during the module installation:

+ Options

<div><div>←T→</div><div></div></div>				id_module	name	active	version
<input type="checkbox"/>	 Edit	 Copy	 Delete	61	statssearch	1	1
<input type="checkbox"/>	 Edit	 Copy	 Delete	62	statsstock	1	1
<input type="checkbox"/>	 Edit	 Copy	 Delete	63	statsvisits	1	1
<input type="checkbox"/>	 Edit	 Copy	 Delete	64	autoupgrade	1	0.9.4
<input type="checkbox"/>	 Edit	 Copy	 Delete	65	blocklayered	1	1.8.9
<input type="checkbox"/>	 Edit	 Copy	 Delete	68	mymodule	1	1.0
<div><div>↑</div><div>Check All / Uncheck All</div><div>With selected:</div><div> Change</div><div> Delete</div><div> Export</div></div>							

About the config.xml file

About the config.xml file

The `config.xml` file makes it possible to optimize the loading of the module list in the back-office.

```
<?xml version="1.0" encoding="UTF-8" ?>
  <module>
    <name>mymodule</name>
    <displayName><![CDATA[My module]]></displayName>
    <version><![CDATA[1.0]]></version>
    <description><![CDATA[Description of my module.]]></description>
    <author><![CDATA[Firstname Lastname]]></author>
    <tab><![CDATA[front_office_features]]></tab>
    <confirmUninstall>Are you sure you want to uninstall?</confirmUninstall>
    <is_configurable>0</is_configurable>
    <need_instance>0</need_instance>
    <limited_countries></limited_countries>
  </module>
```

A few details:

- `is_configurable` indicates whether the module has a configuration page or not.
- `need_instance` indicates whether an instance of the module must be created when it is displayed in the module list. This can be useful if the module has to perform checks on the PrestaShop configuration, and display warning message accordingly.
- `limited_countries` is used to indicate the countries to which the module is limited. For instance, if the module must be limited to France and Spain, use `<limited_countries>fr,es</limited_countries>`.

Adding a configuration page

Adding a configuration page

Your module can get a "Configure" link in the back-office module list, and therefore let the user change some settings. This "Configure" link appears with addition of the `getContent()` method to your main class. This is a standard PrestaShop method: its sole existence sends a message to the back-office, saying "there's a configuration page in this module, display the configuration link".

But having a `getContent()` public method in the `MyModule` object does only make the "Configure" link appear; it does not create the configuration page out of nowhere. We are going to explain how to create one, where we will be able to edit the content of the `MYMODULE_NAME` variable that we stored in the `ps_configuration` data table.

The `getContent()` method

First, here is the complete code for the `getContent()` method:

```
public function getContent()
{
    $output = null;

    if (Tools::isSubmit('submit'.$this->name))
    {
        $my_module_name = strval(Tools::getValue('MYMODULE_NAME'));
        if (!$my_module_name
            || empty($my_module_name)
            || !Validate::isGenericName($my_module_name))
            $output .= $this->displayError($this->l('Invalid Configuration value'));
        else
        {
            Configuration::updateValue('MYMODULE_NAME', $my_module_name);
            $output .= $this->displayConfirmation($this->l('Settings updated'));
        }
    }
    return $output.$this->displayForm();
}
```

The `getContent()` method is the first one to be called when the configuration page is loaded. Therefore, we use it to first update any value that might have been submitted by the form that the configuration page contains.

Here is a line by line explanation:

1. `Tools::isSubmit()` is a PrestaShop-specific method, which checks if the indicated form has been validated.
In this case, if the configuration form has not yet been validated, the whole `if()` block is skipped and PrestaShop will only use the last line, which displays the configuration with the current values, as generated by the `displayForm()` method.
2. `Tools::getValue()` is a PrestaShop-specific method, which retrieve the content of the `POST` or `GET` array in order to get the value of the specified variable.
In this case, we retrieve the value of the `MYMODULE_NAME` form variable, turn its value into a text string using the `strval()` method, and stores it in the `$my_module_name` PHP variable.
3. We then check for the existence of actual content in `$my_module_name`, including the use of `Validate::isGenericName()`.
The `Validate` object contains many data validation methods, among which is `isGenericName()`, a method that helps you keep only strings that are valid PrestaShop names – meaning, a string that does not contain special characters, for short.
4. If any of these checks fail, the configuration will open with an error message, indicating that the form validation failed.
The `$output` variable, which contains the final rendition of the HTML code that makes the configuration page, thus begins with an error message, created using PrestaShop's `displayError()` method. This method returns the correct HTML code for our need, and since that code is first in `$output`, this means the configuration will open with that message.
5. If all these checks are successful, this means we can store the value in our database.
As we saw earlier in this tutorial, the `Configuration` object has just the method we need: `updateValue()` will store the new value for `MYMODULE_NAME` in the configuration data table.
To that, we add a friendly message to the user, indicating that the value has indeed been saved: we use PrestaShop's `displayConfirmation()` method to add that message as the first data in the `$output` variable – and therefore, at the top of the page.
6. Finally, we use the custom `displayForm()` method (which we are going to create and explain in the next section) in order to add

content to `$output` (whether the form was submitted or not), and return that content to the page.

Note that we could have included the code for `displayForm()` right within `getContent()`, but chose to separate the two for readability and separation of concerns.

This form-validation code is nothing new for PHP developers, but uses some of the PrestaShop methods that you will very regularly use.

Displaying the form

The configuration form itself is displayed with the `displayForm()` method. Here is its code, which we are going to explain after the jump:

```
public function displayForm()
{
    // Get default language
    $default_lang = (int)Configuration::get('PS_LANG_DEFAULT');

    // Init Fields form array
    $fields_form[0]['form'] = array(
        'legend' => array(
            'title' => $this->l('Settings'),
        ),
        'input' => array(
            array(
                'type' => 'text',
                'label' => $this->l('Configuration value'),
                'name' => 'MYMODULE_NAME',
                'size' => 20,
                'required' => true
            )
        ),
        'submit' => array(
            'title' => $this->l('Save'),
            'class' => 'button'
        )
    );

    $helper = new HelperForm();

    // Module, token and currentIndex
    $helper->module = $this;
    $helper->name_controller = $this->name;
    $helper->token = Tools::getAdminTokenLite('AdminModules');
    $helper->currentIndex = AdminController::$currentIndex.'&configure='.$this->name;

    // Language
    $helper->default_form_language = $default_lang;
    $helper->allow_employee_form_lang = $default_lang;

    // Title and toolbar
    $helper->title = $this->displayName;
    $helper->show_toolbar = true;          // false -> remove toolbar
    $helper->toolbar_scroll = true;       // yes -> Toolbar is always visible on the
top of the screen.
    $helper->submit_action = 'submit'.$this->name;
    $helper->toolbar_btn = array(
        'save' =>
            array(
                'desc' => $this->l('Save'),
                'href' =>
AdminController::$currentIndex.'&configure='.$this->name.'&save'.$this->name.
```

```
        '&token='.Tools::getAdminTokenLite('AdminModules'),
    ),
    'back' => array(
        'href' =>
AdminController::$currentIndex.'&token='.Tools::getAdminTokenLite('AdminModules'),
        'desc' => $this->l('Back to list')
    )
);

// Load current value
$helper->fields_value['MYMODULE_NAME'] = Configuration::get('MYMODULE_NAME');
```



```

    return $helper->generateForm($fields_form);
}

```

While this might look like a huge block of code for a single value to change, this block actually uses some of PrestaShop's method to make it easier to build forms, most notably the `HelperForm` object.

Diving in displayForm()

Let's run down that method:

1. Using the `Configuration::get()` method, we retrieve the value of the currently chosen language ("PS_LANG_DEFAULT"). For security reasons, we cast the variable into an integer using `(int)`.
2. In preparation for the generation of the form, we must build an array of the various titles, textfields and other form specifics. To that end, we create the `$fields_form` variable, which will contain a multidimensional array. Each of the arrays it features contains the detailed description of the tags the form must contain. From this variable, PrestaShop will render the HTML form as it is described. In this example, we define three tags (`<legend>`, `<input>` and `<submit>`) and their attributes using arrays. The format is quite easy to get: the legend and submit arrays simply contain the attributes to each tag, while the input contains as many `<input>` tags are needed, each being in turn an array which contains the necessary attributes. For instance:

```



```

...generates the following HTML tags:

```

<label>Configuration value </label>
<div class="margin-form">
    <input id="MYMODULE_NAME" class="" type="text" size="20" value="my friend"
name="MYMODULE_NAME">
    <sup>*</sup>
</div class="clear"></div>

```

As you can see, PrestaShop is quite clever, and generates all the code that is needed to obtain a useful form.

Note that the value of the main array is actually retrieved later in the form generation code.

3. We then create an instance of the `HelperForm` class. This section of the code is explained in the next section of this chapter.
4. Once the `HelperForm` settings are all in place, we generate the form based on the content of the `$fields_form` variable.

Using HelperForm

`HelperForm` is one of the helper methods that were added with PrestaShop 1.5, along with `HelperOptions`, `HelperList`, `HelperView` and `HelperHelpAccess`. They enable you to generate standard HTML elements for the back-office as well as for module configuration pages. You can get more information about Helper classes in the "Helpers" chapter of this developer guide, with a page dedicated to `HelperForm`.

Here is our sample code, as a reminder:

```

$helper = new HelperForm();

// Module, Token and currentIndex
$helper->module = $this;
$helper->name_controller = $this->name;
$helper->token = Tools::getAdminTokenLite('AdminModules');
$helper->currentIndex = AdminController::$currentIndex.'&configure='.$this->name;

// Language
$helper->default_form_language = $default_lang;
$helper->allow_employee_form_lang = $default_lang;

// title and Toolbar
$helper->title = $this->displayName;
$helper->show_toolbar = true;           // false -> remove toolbar
$helper->toolbar_scroll = true;        // yes -> Toolbar is always visible on the top
of the screen.
$helper->submit_action = 'submit'.$this->name;
$helper->toolbar_btn = array(
    'save' =>
        array(
            'desc' => $this->l('Save'),
            'href' =>
AdminController::$currentIndex.'&configure='.$this->name.'&save'.$this->name.
            '&token='.Tools::getAdminTokenLite('AdminModules'),
        ),
    'back' => array(
        'href' =>
AdminController::$currentIndex.'&token='.Tools::getAdminTokenLite('AdminModules'),
        'desc' => $this->l('Back to list')
    )
);

// Load current value
$helper->fields_value['MYMODULE_NAME'] = Configuration::get('MYMODULE_NAME');

return $helper->generateForm($fields_form);

```

Our example uses several of HelperForm's attributes: they need to be set before we generate the form itself from the \$fields_form variable:

- \$helper->module: requires the instance of the module that will use the form.
- \$helper->name_controller: requires the name of the module.
- \$helper->token: requires a unique token for the module. getAdminTokenLite() helps us generate one.
- \$helper->currentIndex:
- \$helper->default_form_language: requires the default language for the shop.
- \$helper->allow_employee_form_lang: requires the default language for the shop.
- \$helper->title: requires the title for the form.
- \$helper->show_toolbar: requires a boolean value – whether the toolbar is displayed or not.
- \$helper->toolbar_scroll: requires a boolean value – whether the toolbar is always visible when scrolling or not.
- \$helper->submit_action: requires the action attribute for the form's <submit> tag.
- \$helper->toolbar_btn: requires the buttons that are displayed in the toolbar. In our example, the "Save" button and the "Back" button.
- \$helper->fields_value[]: this is where we can define the value of the named tag.

Finally, after all is set and done, we can call the generateForm() method, which will take care of putting it all together and, as its name says, generate the form that the user will use to configure the module's settings.

Here is the rendition of the form as it is presently written – which you can see by yourself by clicking on the "Configure" link for the module in the back-office:

Change the value to whichever you like, click on the "Save" button, then go reload the homepage: your module is indeed updated with the new string!

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:

- `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:

1. Find "My module" in the "Module" drop-down list.
2. Choose "(displayLeftColumn) Left column blocks" in the "Hook into" drop-down list.
3. 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. If 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. If 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`:

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 is 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 your 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.



Disable the cache

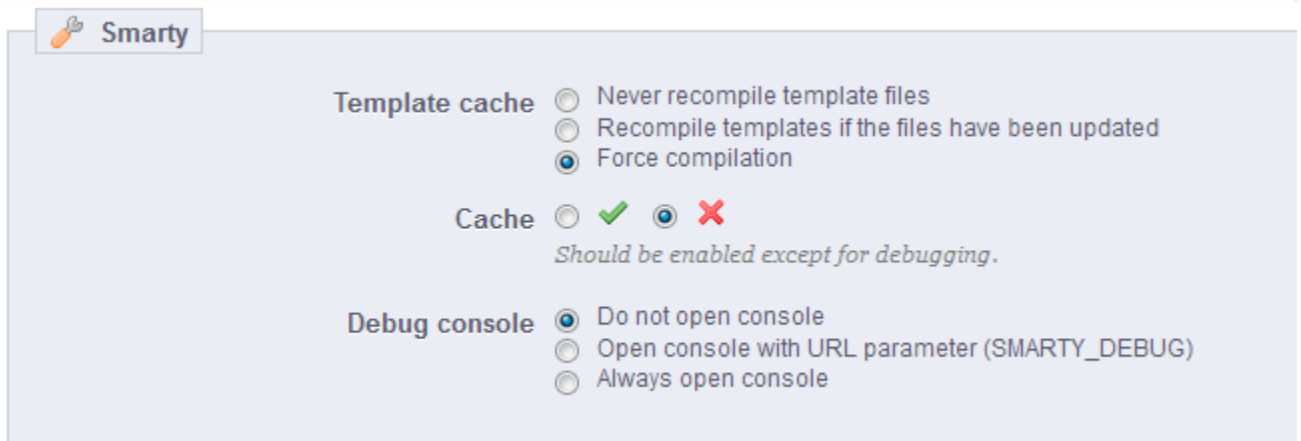
If you've followed this tutorial to the letter and still do not see anything appearing in the theme's left column, it might be because PrestaShop has cached the previous templates, and is still serving these to you. Hence, you see the original version of the theme, without your changes.

Smarty caches a compiled version of the homepage, for performance reasons. This is immensely helpful for production sites, but is useless for test 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.




The screenshot shows the Smarty configuration interface. It has a header with a wrench icon and the word "Smarty". Below the header, there are three sections: "Template cache", "Cache", and "Debug console". Each section has radio buttons for different options. "Template cache" has three options: "Never recompile template files", "Recompile templates if the files have been updated", and "Force compilation". "Cache" has two options: "Cache" (with a green checkmark) and "Disable cache" (with a red X). Below the "Cache" options is the text "Should be enabled except for debugging.". "Debug console" has three options: "Do not open console", "Open console with URL parameter (SMARTY_DEBUG)", and "Always open console".

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.



The screenshot shows the content of the `display.php` file. It contains the text "Welcome to this page!".

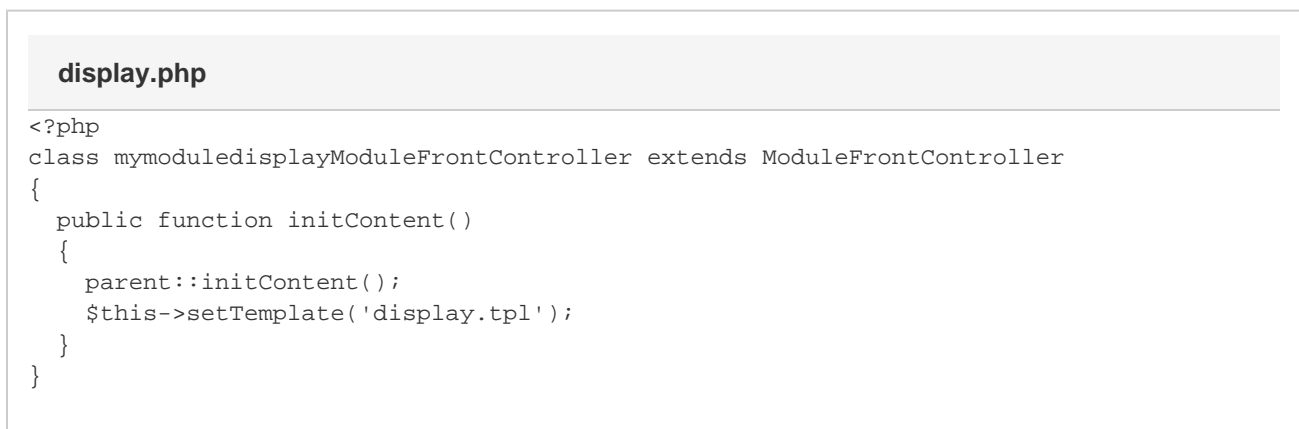
Click the "Click me!" link: the resulting page is just that raw text, without anything from the theme. We would like to have this text embedded in the theme, so let's see how to do just that.

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.



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:



The screenshot shows the content of the `display.php` file. It contains the following PHP code:

```
<?php
class mymoduledisplayModuleFrontController extends ModuleFrontController
{
    public function initContent()
    {
        parent::initContent();
        $this->setTemplate('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

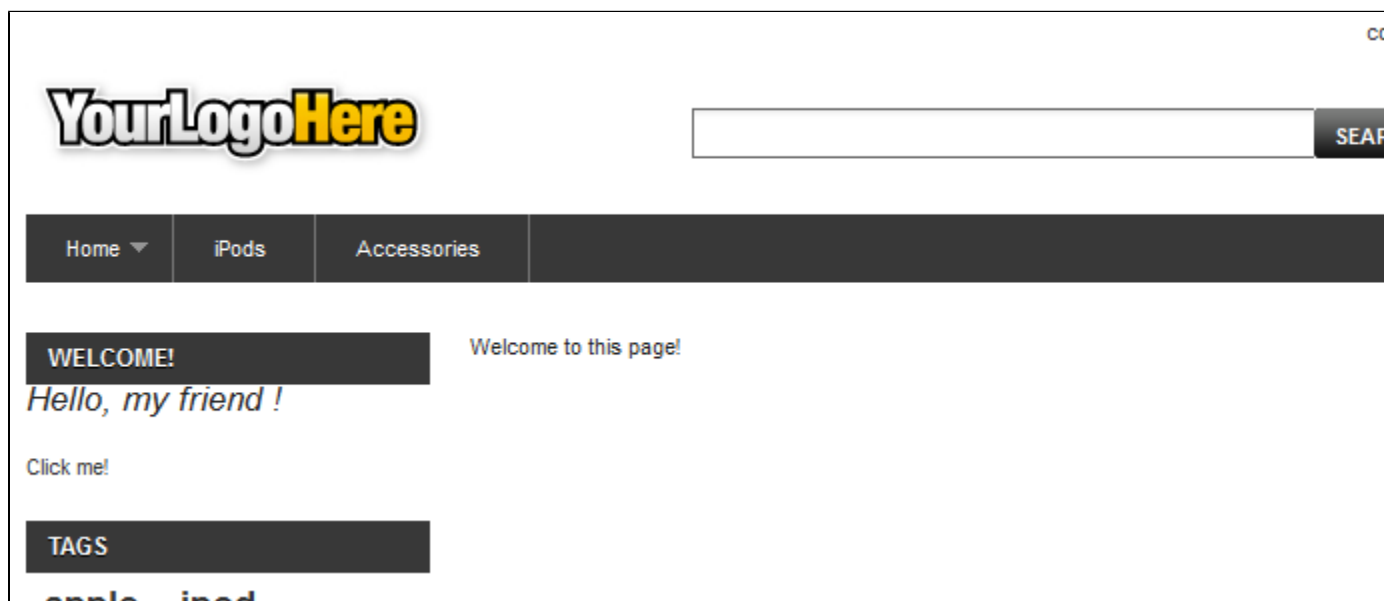
```
<?php
// This file must be placed at the root of the module's folder.
global $smarty;
include('...../config/config.inc.php');
include('...../header.php');

$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.

Module translation

Module translation

The module's text strings are written in English, but you might want French, Spanish or Polish shop owners to use your module too. You therefore have to translate those strings into those languages, both the front-office and the back-offices strings. Ideally, you should translate your module in all the languages that are installed on your shop. This could be a tedious task, but Smarty and PrestaShop's own translation tool make it far easier.

In short, PrestaShop implements its own translation mechanism, through the use of the `l` (lowercase L) method, used to encapsulate the strings to be translated.. This method is applied in a different way depending of the file type.

Strings in PHP files will need to be displayed through the `$this->l('My string.')` method call, which comes from the `Module` abstract class, and thus is available in all modules.

mymodule.php (partial)

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



There are specific context where `$this->l()` will not work: when your module has a front-end controller, that controller's strings must be put in a `$this->module->l('My string', 'filename')` method call.

For instance, in the `/bankwire/controllers/front/validation.php` file:

```
die($this->module->l('This payment method is not available.', 'validation'));
```

This is a very specific case, and you should not often have to use it. Keep to `$this->l()`, unless your code breaks because of it when in a `FrontController` context.

Strings in TPL files will need to be turned into dynamic content using the `{l s='My string' mod='modulename'}` function call, which Smarty will replace by the translation for the chosen language. In our sample module, the `mymodule.tpl` file...

mymodule.tpl (partial)

```
<li>
  <a href="{ $base_dir}modules/mymodule/mymodule_page.php" title="Click this
link">Click me!</a>
</li>
<!-- Block mymodule -->
<div id="mymodule_block_left" class="block">
  <h4>{l s='Welcome!' mod='mymodule'}</h4>
  <div class="block_content">
    <p>Hello,
      {if isset($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 -->
```

...becomes:

mymodule.tpl (partial)

```
<li>
  <a href="{ $base_dir}modules/mymodule/mymodule_page.php" title="{l s='Click this
link' mod='mymodule'}">{l s='Click me!' mod='mymodule'}</a>
</li>
<!-- Block mymodule -->
<div id="mymodule_block_left" class="block">
  <h4>{l s='Welcome!' mod='mymodule'}</h4>
  <div class="block_content">
    <p>
      {if !isset($my_module_name) || !$my_module_name}
        {capture name='my_module_tempvar'}{l s='World' mod='mymodule'}{/capture}
        {assign var='my_module_name' value=$smarty.capture.my_module_tempvar}
      {/if}
      {l s='Hello %1$s!' sprintf=$my_module_name mod='mymodule'}
    </p>
    <ul>
      <li><a href="{ $my_module_link}" title="{l s='Click this link'
mod='mymodule'}">{l s='Click me!' mod='mymodule'}</a></li>
    </ul>
  </div>
</div>
<!-- /Block mymodule -->
```

...and the display.tpl file:

display.tpl

Welcome to this page!

...becomes:

display.tpl

```
{l s='Welcome to this page!' mod='mymodule'}
```



Translating complex code

As we can see, the basis of template file translation is to enclose them in the `{l s='The string' mod='name_of_the_module'}`. The changes in `display.tpl` and in `mymodule.tpl`'s link and title texts are thus easy to understand. But added a trickier block of code for the "Hello World!" string: an if/else/then clause, and a text variable. Let's explore this code:

Here is the original code:

```
Hello,
    {if isset($my_module_name) && $my_module_name}
        {$my_module_name}
    {else}
        World
    {/if}
!
```

As you can see, we need to get the "Hello World" string translatable, but also to cater for the fact that there is a variable. As explained in the "Translations in PrestaShop 1.5" chapter, variables are to be marked using `sprintf()` markers, such as `%s` or `%1$s`.

Making "Hello %s!" translatable words in easy: we just need to use this code:

```
{l s='Hello %s!' sprintf=$my_module_name mod='mymodule'}
```

But in our case, we also need to make sure that the `%s` is replaced by "World" in case the `"my_module_name"` value does not exist... and we must make "World" translatable too. This can be achieved by using Smarty `{capture}` function, which collects the output of the template between the tags into a variable instead of displaying, so that we can use it later on. We are going to use it in order to replace the variable with the translated "World" if the variable is empty or absent, using a temporary variable. Here is the final code:

```
{if !isset($my_module_name) || !$my_module_name}
    {capture name='my_module_tempvar'}{l s='World' mod='mymodule'}{/capture}
    {assign var='my_module_name' value=$smarty.capture.my_module_tempvar}
{/if}
{l s='Hello %s!' sprintf=$my_module_name mod='mymodule'}
```

Notice that we always use the `mod` parameter. This is used by PrestaShop to assert which module the string belongs to. The translation tool needs it in order to match the string to translate with its translation. This parameter is **mandatory** for module 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 "Translations" page under the "Localization" menu,
- In the "Modify translations" drop-down menu, choose "Module translations",

- Click the flag of the country of which language you want to translate the module into. The destination language must already be installed to enable translation in it.

The page that loads 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.

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

PrestaShop then saves the translations in a new file, named using the `languageCode.php` format (for instance, `/mymodule/fr.php`). The translation file looks like so:

fr.php

```
<?php
global $_MODULE;
$_MODULE = array();
$_MODULE['<{mymodule}prestashop>mymodule_2dddc2a736e4128ce1cdfd22b041e7f'] = 'Mon
module';
$_MODULE['<{mymodule}prestashop>mymodule_d6968577f69f08c93c209bd8b6b3d4d5'] =
'Description du module.';
$_MODULE['<{mymodule}prestashop>mymodule_533937acf0e84c92e787614bbb16a7a0'] =
'Êtes-vous certain de vouloir désinstaller ce module ? Vous perdrez tous vos réglages
!';
$_MODULE['<{mymodule}prestashop>mymodule_0f40e8817b005044250943f57a21c5e7'] = 'Aucun
nom fourni';
$_MODULE['<{mymodule}prestashop>mymodule_fe5d926454b6a8144efce13a44d019ba'] = 'Valeur
de configuration non valide.';
$_MODULE['<{mymodule}prestashop>mymodule_c888438d14855d7d96a2724ee9c306bd'] =
'Réglages mis à jour';
$_MODULE['<{mymodule}prestashop>mymodule_f4f70727dc34561dfde1a3c529b6205c'] =
'Réglages';
$_MODULE['<{mymodule}prestashop>mymodule_2f6e771db304264c8104cb7534bb80cd'] = 'Valeur
de configuration';
$_MODULE['<{mymodule}prestashop>mymodule_c9cc8cce247e49bae79f15173ce97354'] =
'Enregistrer';
$_MODULE['<{mymodule}prestashop>mymodule_630f6dc397fe74e52d5189e2c80f282b'] = 'Retour
à la liste';
$_MODULE['<{mymodule}prestashop>display_86e88cbccafa83831b4c6685501c6e58'] =
'Bienvenue sur cette page !';
$_MODULE['<{mymodule}prestashop>mymodule_9a843f20677a52ca79af903123147af0'] =
'Bienvenue !';
$_MODULE['<{mymodule}prestashop>mymodule_f5a7924e621e84c9280a9a27e1bcb7f6'] = 'Monde';
$_MODULE['<{mymodule}prestashop>mymodule_3af204e311ba60e6556822eac1437208'] = 'Bonjour
%s !';
$_MODULE['<{mymodule}prestashop>mymodule_c66b10fbf9cb6526d0f7d7a602a09b75'] = 'Cliquez
sur ce lien';
$_MODULE['<{mymodule}prestashop>mymodule_f42c5e677c97b2167e7e6b1e0028ec6d'] =
'Cliquez-moi !';
```



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

Now that we have a French translation, we can click on the French flag in the front-office, 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.

Enabling the Auto-Update

Enabling the Auto-Update

Since PrestaShop 1.5, it is possible to have your module auto-update: once a new version is available on Addons, PrestaShop suggests an "Update it!" button to the user. Clicking this button will trigger a series of methods, each leading closer to the latest version of your module.

In order to bring auto-update support to your module, you need three main things:

- Clearly indicate the module's version number in its constructor method: `$this->version = '1.1';`
- Create an `/upgrade` sub-folder in the module's folder.
- Add an auto-update PHP script for each new version.

For instance:

```
/*
 * File: /upgrade/Upgrade-1.1.php
 */
function upgrade_module_1_1($module) {
    // Process Module upgrade to 1.1
    // ....
    return true; // Return true if success.
}
```

...and then:

```
/*
 * File: /upgrade/Upgrade-1.2.php
 */
function upgrade_module_1_2($module) {
    // Process Module upgrade to 1.2
    // ....
    return true; // Return true if succes.
}
```

Each method should bring the necessary changes to the module's files and database data in order to reach the latest version.

For instance, here is the `install-1.4.9.php` file from the gamification module:

```
<?php
if (!defined('_PS_VERSION_'))
    exit;
function upgrade_module_1_4_9($object)
{
    return Db::getInstance()->execute(
        'CREATE TABLE IF NOT EXISTS `'.$_DB_PREFIX_.'tab_advice` (
            `id_tab` int(11) NOT NULL,
            `id_advice` int(11) NOT NULL,
            PRIMARY KEY (`id_tab`, `id_advice`)
        ) ENGINE=\'._MYSQL_ENGINE_.\' DEFAULT CHARSET=utf8;');
}
```

The homeslider module's `install-1.2.1.php` file does even more:

```

<?php
if (!defined('_PS_VERSION_'))
    exit;
function upgrade_module_1_2_1($object)
{
    return Db::getInstance()->execute('
UPDATE `._DB_PREFIX_`.homeslider_slides_lang SET
    `._homeslider_stripslashes_field('title')`,
    `._homeslider_stripslashes_field('description')`,
    `._homeslider_stripslashes_field('legend')`,
    `._homeslider_stripslashes_field('url')
    `);
}
function homeslider_stripslashes_field($field)
{
    $quotes = array('"\\\\"', '\\\'');
    $dquotes = array('\\"\\\\"', '\\\'');
    $backslashes = array('\\\\\\\\\\\\\\\\', '\\\\\\\\\\\\\\');
    return '`._bqSQL($field).` = replace(replace(replace(`._bqSQL($field).`,
    `._$quotes[0].`, `._$quotes[1].`), `._$dquotes[0].`, `._$dquotes[1].`),
    `._$backslashes[0].`, `._$backslashes[1].`);
}

```

PrestaShop will then parse all of these scripts one after the other, sequentially. It is therefore highly advised to number your module's versions sequentially, and to only use numbers – because the upgrade code uses PHP's [version_compare\(\)](#) method.



If the new version of your module adds or update its hooks, you should make sure to update them too

Indeed, since the hooks are (usually) defined when the module is installed, PrestaShop will not install the module again in order to include the new hooks' code, so you have to use the upgrade methods:

For instance, here's the `install-1.2.php` file from the `blockbestseller` module:

```

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

function upgrade_module_1_2($object)
{
    return ($object->registerHook('addproduct')
        && $object->registerHook('updateproduct')
        && $object->registerHook('deleteproduct')
        && $object->registerHook('actionOrderStatusPostUpdate'));
}

```


Development Troubleshooting

Development Troubleshooting

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

Make sure your code is valid!

PrestaShop has built a special online tool to help you discover possible issues within your code.

You can access it here: <https://validator.prestashop.com/auth>

Make sure to validate your module before you submit it to the Addons marketplace! Indeed, the Addons team will refuse to accept a module that is not at least valid.

PrestaShop official forum

Join our forums 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 register 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!

1.6-specific Developer Documentation

Developer documentation specific to PrestaShop 1.6

Version 1.6 of PrestaShop is an evolution of version 1.5: while a lot has changed on the surface, the core of the software remains the same – only tightened and sturdier.

Hence, any module that works in version 1.5 should work as-is in version 1.6 – though it really helps if it uses the Helper class to build its interface rather than custom HTML.

That being said, version 1.6 does bring a slew of improvement to the module API, most notably with the adjunction of Bootstrap to the default interface and theme. These new features and practices in the following chapters:

- [Creating a Dashboard Module](#)
- [Making your module work with Bootstrap](#)

Miscellaneous developer documentation

Miscellaneous developer documentation

Here are listed some additional documentation, for those who want to keep on learning with specific subjects.

- [Specifics of multistore module development](#)
- [The Backward Compatibility Toolkit: Making your 1.5+ module compatible with PrestaShop 1.4](#)
- [Creating a payment module](#)
- [Creating a carrier module](#)
- [Tying your module to your Addons account](#)

Creating a Dashboard Module

Table of contents

- [Creating a Dashboard Module](#)
 - [Differences with a regular PrestaShop module](#)
 - [The specifics](#)
 - [The name](#)
 - [The constructor method](#)
 - [The install\(\) method](#)
 - [The zones](#)
 - [The templates](#)
 - [The data flow](#)
 - [Building hookDashboardData\(\)](#)
 - [The data types](#)
 - [Naming and getting the values displayed](#)
 - [You own data types](#)
 - [The configuration form](#)

Creating a Dashboard Module

Version 1.6 of PrestaShop brings a brand new Dashboard which features blocks of content that can be reorganized and, more importantly to you, added to.

Indeed, you can create your own dashboard modules, which you can make available for all to download or buy on PrestaShop Addons.

Differences with a regular PrestaShop module

A PrestaShop dashboard module is essentially the same thing as a regular PrestaShop module, with a few specifics. It is located in the `/modules` folder, and it can make use of PrestaShop's controllers, just like any module does.

The specifics

The name

Your dashboard module must have a unique name. As a convention, dashboard modules should all use the "dash" prefix: "dashproduct", "dashactivity", "dashgoals" are the names of some of the default dashboard modules

The constructor method

As for any PrestaShop module, the main PHP file must contain a `__construct()` method, which declares the usual variables: `name`, `displayName`, `version`, etc.

A few new variables are necessary for a proper dashboard module:

- `$this->push_filename`: the directory where your push data should be stored.
- `$this->allow_push`: a boolean indicating whether your module should have data be pushed to it. or not.
- `$this->push_time_limit`: the numbers of seconds between two data pushes.

Here is a sample `__construct()` method, taken from the Dashactivity module:

```

public function __construct()
{
    $this->name = 'dashactivity';
    $this->displayName = 'Dashboard Activity';
    $this->tab = '';
    $this->version = '0.1';
    $this->author = 'PrestaShop';
    $this->push_filename = _PS_CACHE_DIR_.'push/activity';
    $this->allow_push = true;
    $this->push_time_limit = 180;

    parent::__construct();
}

```

Note that the `tab` variable is empty. It is not needed as of today, but might be in future versions of PrestaShop.

The install() method

As for any PrestaShop module, the main PHP file must contain a `install()` method, which declares the usual hooks, such as `displayBackOfficeHeader`.

PrestaShop 1.6 implements several new hooks dedicated to dashboard modules:

- `dashboardData`: how you want your data to be handled.
- `dashboardZoneOne`: enables you to display your content in the left column of the dashboard zone.
- `dashboardZoneTwo`: enables you to display your content in the central column of the dashboard zone.

Here is a sample `install()` method, taken from the Dashproducts module:

```

public function install()
{
    if (!parent::install()
        || !$this->registerHook('dashboardZoneTwo')
        || !$this->registerHook('dashboardData'))
        return false;
    return true;
}

```

The zones

The dashboard has two available zones for your modules:

- Zone One: The left column of the dashboard.
- Zone Two: The central column of the dashboard.

Depending of the hooks your module is registered with, you might be able to display your content either on one of the two columns, or in both if need be.



The right column of the dashboard is not available for module.

The user cannot move a module from one column to the other, so you shouldn't hook your module to both zones.

Each hook must call on a template file in order to display your content.

Here is a sample `hookDashboardZoneTwo()` method, taken from the Dashproducts module:

```
public function hookDashboardZoneTwo($params)
{
    $this->context->smarty->assign(array(
        'date_from' => Tools::displayDate($params['date_from']),
        'date_to' => Tools::displayDate($params['date_to']));
    return $this->display(__FILE__, 'dashboard_zone_two.tpl');
}
```

The templates

You should name your template after the zone it should be displayed in, as a handy reminder.

The template file should be stored in a module-specific folder: `/modules/dashmyodule/views/templates/hook/dashboard_zone_two.tpl`

The template itself is a classic PrestaShop template, with Smarty tags in HTML tags.

As you saw in the hook method, you can create new Smarty tags when your module is hooked to a zone. For instance, this code (inspired from the Dashactivity module):

```
public function hookDashboardZoneOne($params)
{
    $this->context->smarty->assign(array_merge(array(
        'dashactivity_config_form' => $this->renderConfigForm(),
        'date_subtitle' => $this->l('(from %s to %s)'),
        'date_format' => $this->context->language->date_format_lite
    ), $this->getConfigFieldsValues()));
    return $this->display(__FILE__, 'dashboard_zone_one.tpl');
}
```

This code declares three new tags:

- `dashactivity_config_form`: displays the content of the module's `renderConfigForm()` method.
- `date_subtitle`: displays a localized string ("from %s to %s").
- `date_format`: displays the date in "light" format (day.month.year).

From there on, you can call these tags directly in the template. For instance, `{$dashactivity_config_form}` displays the form in `renderConfigForm()`.

The data flow

The point of having a dashboard module is to display useful content to your user. When the Dashboard first loads, and every time you change a time setting (at the top of the Dashboard), PrestaShop triggers a call to the `dashboardData` hook in order to load data through Ajax requests.

In order to hook code the `dashboardData` hook, you must create a `hookDashboardData()` method, which must contain the SQL requests necessary for your data. This method must then return an array of the standard data types, each containing an array of the values in that data type.

For instance, here is the `hookDashboardData()` method from Dashproduct:

```

public function hookDashboardData($params)
{
    $table_recent_orders = $this->getTableRecentOrders();
    $table_best_sellers = $this->getTableBestSellers($params['date_from'],
$params['date_to']);
    $table_most_viewed = $this->getTableMostViewed($params['date_from'],
$params['date_to']);
    $table_top_10_most_search = $this->getTableTop10MostSearch($params['date_from'],
$params['date_to']);
    return array(
        'data_table' => array(
            'table_recent_orders' => $table_recent_orders,
            'table_best_sellers' => $table_best_sellers,
            'table_most_viewed' => $table_most_viewed,
            'table_top_10_most_search' => $table_top_10_most_search,
        )
    );
}

```

Building hookDashboardData()

Your hookDashboardData() method can be as simple or complex as is necessary for your dashboard module. See for instance the way the Dashactivity module implements it: <https://github.com/PrestaShop/PrestaShop/blob/c7fe91a0a9e02ca21183cc1c09e3e565d4de7265/modules/dashactivity/dashactivity.php#L95> The important thing is that it should return the various values as an array of arrays, with the data types as the root arrays.

See for instance the bottom of Dashactivity's hookDashboardData() method:

```

return array(
    'data_value' => array(
        'pending_orders' => $pending_orders,
        'return_exchanges' => $return_exchanges,
        // etc.
    ),
    'data_trends' => array(
        'orders_trends' => array('way' => 'down', 'value' => 0.42),
    ),
    'data_list_small' => array(
        'dash_traffic_source' => $this->getReferer($params['date_from'],
$params['date_to']),
    ),
    'data_chart' => array(
        'dash_trends_chart1' => $this->getChartTrafficSource($params['date_from'],
$params['date_to']),
    ),
);

```

PrestaShop retrieves this data using an Ajax request, in JSON format:

```
{ "dashactivity": {
  "data_value": { "pending_orders": "0", "return_exchanges": "0" },
  "data_trends": { "orders_trends": { "way": "down", "value": 0.42 } },
  "data_list_small": { "dash_traffic_source": { "Direct link": 0 } },

  "data_chart": { "dash_trends_chart1": { "chart_type": "pie_chart_trends", "data": [ { "key": "Direct link", "y": 0 } ] } }
}
```

Note that you must set the values within an array corresponding to the data type:

- All Value arrays must be in the `data_value` array
- All Trends arrays must be in the `data_trends` array
- etc.

The data types

There are 4 data types that your module can use:

- Value: a single value, which can have any data type (string, number, boolean, etc.)
- Trends: a specific type, available as an array two values:
 - 'way': either 'up' or 'down', depending on the trend
 - 'value': the difference between the past and current value.
- List Small: an array of values.
- Chart: an array of two values:
 - 'chart_type': the type of chart to be used (ie.: 'pie_chart_trends').
 - 'data': an array of arrays:
 - key: the data key.
 - y: the data value.

Naming and getting the values displayed

The key for each value is very important, as this is how PrestaShop will update the display of your module: using JavaScript, the Dashboard will find the HTML tag which correspond to a value, and update its content.

For instance, this code in the Dashactivity's `dashboard_zone_one.tpl` file:

```
<span class="data_value size_l">
  <span id="pending_orders"></span>
</span>
```

...is tied to the JSON's `pending_orders` value through PrestaShop's JavaScript code – which you have yourself set in the PHP code for your `hookDashboardData()` method.

You must therefore pay attention to the way you name your HTML elements and your data key, as there is a clear correlation between the two within PrestaShop.

You own data types

In addition to the default ones, **you can create your own data type!** For instance, you could feel the need for a Slider data type, or a clear boolean. You return code would therefore be:


```
return array(
    'data_boolean' => array(
        'is_enabled' => getModuleState(),
        'must_auto_reload' => $auto_reload,
        // etc.
    ),
);
```

Which would give you the following JSON data response:

```
{ "dashmymodule": {
    "data_boolean": { "is_enabled": true, "must_auto_reload": false },
}
```

The default data types have their data loaded and placed by specific JavaScript functions. In order for the data of your custom data types to still be loaded and correctly placed, you must add your own JavaScript function, which is called automatically when `hookDashboardData()` send its JSON back.

That JavaScript function should be in your own `.js` file, which you should add to the theme's header using the `$this->context->controller->addjs('js/mymethods.js', 'all');` method.

It should take this simple form:

```
function data_boolean(widget_name, data)
{
    // Here, the code that takes the data array sent by hookDashboardData, and handles
    their display in the right location.
}
```

To get an example of such JavaScript methods, check the ones for `data_value`, `data_trends` and the other default data type in the file <https://github.com/PrestaShop/PrestaShop/blob/e71094283395b092ccd0b0a0bb0a0fdfe25cbabc/js/admin-dashboard.js#L112>.

The configuration form

Dashboard modules can have their own configuration form, which the user can access at the click of a button.

To declare the configuration form, you simply have to:

1. Declare the PHP function that will render this form.
2. Assign that function to a Smarty tag.
3. Call that Smarty tag in the template.

Here is how the Dashactivity module does it.

1 - Declaration:

```

public function renderConfigForm()
{
    $fields_form = array(
        'form' => array(
            'id_form' => 'step_carrier_general',
            'input' => array(),
            'submit' => array(
                'title' => $this->l(' Save '),
                'class' => 'btn btn-default submit_dash_config',
                'reset' => array(
                    'title' => $this->l('Cancel'),
                    'class' => 'btn btn-default cancel_dash_config',
                )
            )
        ),
    );

    $sub_widget = array(
        array('label' => $this->l('Show Pending'), 'config_name' =>
'DASHACTIVITY_SHOW_PENDING'),
        array('label' => $this->l('Show Notifications'), 'config_name' =>
'DASHACTIVITY_SHOW_NOTIFICATION'),
        array('label' => $this->l('Show Clients'), 'config_name' =>
'DASHACTIVITY_SHOW_CUSTOMERS'),
        array('label' => $this->l('Show Newsletters'), 'config_name' =>
'DASHACTIVITY_SHOW_NEWSLETTER'),
        array('label' => $this->l('Show Traffic'), 'config_name' =>
'DASHACTIVITY_SHOW_TRAFFIC'),
    );

    // etc.
}

```

For the full `renderConfigForm()` code, see <https://github.com/PrestaShop/PrestaShop/blob/c7fe91a0a9e02ca21183cc1c09e3e565d4de7265/modules/dashactivity/dashactivity.php#L297>.

2 - Assignment to a Smarty tag:

```

public function hookDashboardZoneOne($params)
{
    $this->context->smarty->assign(array_merge(array(
        'dashactivity_config_form' => $this->renderConfigForm(),
    ), $this->getConfigFieldsValues()));
    return $this->display(__FILE__, 'dashboard_zone_one.tpl');
}

```

3 - Use within the template:

```

<section id="dashactivity_config" class="dash_config hide">
    <header><i class="icon-wrench"></i> {l s='Configuration' mod='dashactivity'}</header>
    {$dashactivity_config_form}
</section>

```

Note that the template code should be exactly as presented here, so as to make sure that it fits well within the dashboard's design.

Making your module work with Bootstrap

Making your module work with Bootstrap

Version 1.6 of PrestaShop brings a whole new design to the default theme and the software itself. These designs are technically based on the Bootstrap 3 CSS framework (<http://getbootstrap.com/>), which enables designers and developers to rely on its tools and templates in order to create great and responsive designs.

As a module developer, you should strive to update your module to use Bootstrap, so that they integrate will into the new design.

It's all in the helpers

In itself, it is not complicated: most of the work is handled by PrestaShop's Helper methods, which have been upgrade to use Bootstrap the way it should be. Therefore, the hardest part for you is to move your module's interface code from the old way, where you defined your forms directly in HTML, into the new way that was introduced with PrestaShop 1.5, and which makes the Helpers methods do the heavy lifting.

Before / after

Here is an example of how your module can make use of the Bootstrap framework, through PrestaShop's Helper methods. This example is taken from the blockcart module, and concentrates on two methods: `getContent()` (which PrestaShop calls in order to display the module's configuration page), and `displayForm()` (which is replaced by `renderForm()`).

You can see the difference right on Github:

- Old blockcart module: <https://github.com/PrestaShop/PrestaShop/blob/62ff976d69f4f5efd3413227f20bed429705e7b7/modules/blockcart/blockcart.php>
- "Bootstrapped" blockcart module: <https://github.com/PrestaShop/PrestaShop/blob/8144935d764d39d9ed809a1d16c8f452dd9f5591/modules/blockcart/blockcart.php>
- The commit where all the work happens: <https://github.com/PrestaShop/PrestaShop/commit/c7ebf5ba5daaf54e7c1579c39f9d1d929f0259aa#diff-643b105ace43ea459d923f319583a84c>

First, `getContent()`.

Before

```
public function getContent()
{
    $output = '<h2>'.$this->displayName.'</h2>';
    if (Tools::isSubmit('submitBlockCart'))
    {
        $ajax = Tools::getValue('cart_ajax');
        if ($ajax != 0 && $ajax != 1)
            $output .= '<div class="alert error">'.$this->l('Ajax : Invalid choice.').'</div>';
        else
            Configuration::updateValue('PS_BLOCK_CART_AJAX', (int)($ajax));
        $output .= '<div class="conf confirm">'.$this->l('Settings updated').'</div>';
    }
    return $output.$this->displayForm();
}
```

After

```
public function getContent()
{
    $output = '';
    if (Tools::isSubmit('submitBlockCart'))
    {
        $ajax = Tools::getValue('cart_ajax');
        if ($ajax != 0 && $ajax != 1)
            $output .= $this->displayError($this->l('Ajax : Invalid choice.'));
        else
            Configuration::updateValue('PS_BLOCK_CART_AJAX', (int)($ajax));
        $output .= $this->displayConfirmation($this->l('Settings updated'));
    }
    return $output.$this->renderForm();
}
```

The changes are minimal, the most important one being that we do not use direct HTML code anymore to output content, but rather use PrestaShop's `displayError()` for error messages and `displayConfirmation()` for success messages. It is no longer necessary to include `displayName()` in the output since this is taken into account by PrestaShop methods.

As you can see, `getContent()` no longer calls on `displayForm()`, but rather `renderForm()`. We could simply rewrite `displayForm()`'s code to use the HelperForm methods, but changing the name of the method too helps to make the step to the new way of building forms since PrestaShop 1.5.

Let's first get a reminder of what `displayForm()` looked like.

```
public function displayForm()
{
    return '
    <form action="'.Tools::safeOutput($_SERVER['REQUEST_URI']).'" method="post">
    <fieldset>
        <legend>'. $this->l('Settings').</legend>

        <label>'. $this->l('Ajax cart').</label>
        <div class="margin-form">
            <input type="radio" name="cart_ajax" id="ajax_on" value="1" '
                .(Tools::getValue('cart_ajax',
Configuration::get('PS_BLOCK_CART_AJAX')) ? 'checked="checked" ' : '').</div>
            <label class="t" for="ajax_on"> l('Enabled').</label>
            <input type="radio" name="cart_ajax" id="ajax_off" value="0" '
                .(!Tools::getValue('cart_ajax',
Configuration::get('PS_BLOCK_CART_AJAX')) ? 'checked="checked" ' : '').</div>
            <label class="t" for="ajax_off"> l('Disabled').</label>
            <p class="clear">'. $this->l('Activate AJAX mode for cart (compatible with the
default theme)').</p>
        </div>

        <center><input type="submit" name="submitBlockCart" value="'. $this->l('Save').'"
class="button" /></center>
    </fieldset>
    </form>';
}
```

Now, here is the `renderForm()` method, which makes pretty much the same thing as `displayForm()`, but in a cleaner, more portable and now responsive way. This is how you should build forms from now on.



For a complete presentation of `HelperForm`, see this documentation page: <http://doc.prestashop.com/display/PS15/HelperForm>

```
public function renderForm()
{
    $fields_form = array(
        'form' => array(
            'legend' => array(
                'title' => $this->l('Settings'),
                'icon' => 'icon-cogs'
            ),
            'input' => array(
                array(
                    'type' => 'switch',
                    'label' => $this->l('Ajax cart'),
                    'name' => 'PS_BLOCK_CART_AJAX',
                    'is_bool' => true,
                    'desc' => $this->l('Activate AJAX mode for cart (compatible with the default theme)'),
                    'values' => array(
                        array(
                            'id' => 'active_on',
                            'value' => 1,
                            'label' => $this->l('Enabled')
                        ),
                        array(
                            'id' => 'active_off',
                            'value' => 0,
                            'label' => $this->l('Disabled')
                        )
                    )
                ),
            ),
            'submit' => array(
                'title' => $this->l('Save'),
                'class' => 'btn btn-default pull-right'
            ),
        ),
    );

    $helper = new HelperForm();
    $helper->show_toolbar = false;
    $helper->table = $this->table;
    $lang = new Language((int)Configuration::get('PS_LANG_DEFAULT'));
    $helper->default_form_language = $lang->id;
    $helper->allow_employee_form_lang =
        Configuration::get('PS_BO_ALLOW_EMPLOYEE_FORM_LANG') ?
        Configuration::get('PS_BO_ALLOW_EMPLOYEE_FORM_LANG') : 0;
    $this->fields_form = array();

    $helper->identifier = $this->identifier;
    $helper->submit_action = 'submitBlockCart';
    $helper->currentIndex = $this->context->link->getAdminLink('AdminModules', false)
        . '&configure=' . $this->name . '&tab_module=' . $this->tab . '&module_name=' . $this->name;
    $helper->token = Tools::getAdminTokenLite('AdminModules');
    $helper->tpl_vars = array(
```

```
'fields_value' => $this->getConfigFieldsValues(),  
'languages' => $this->context->controller->getLanguages(),  
'id_language' => $this->context->language->id  
);
```

```
return $helper->generateForm(array($fields_form));
}
```

Important details

The 'bootstrap' variable

One last thing to watch out for: if your module uses a bootstrapped controller, you must add the `bootstrap` variable to the module's constructor method.

Indeed, helpers do most of the hard work, but as long as you do not indicate that you are using Bootstrap, your controller will be surrounded by "classic" CSS classes, whereas a single line makes PrestaShop use the "bootstrapped" CSS classes:

```
public function __construct()
{
    $this->bootstrap = true;
    $this->display = 'view';
    $this->meta_title = $this->l('Your Merchant Expertise');
    parent::__construct();
}
```

This MUST be placed in your module's `__construct()` method to work – if you use bootstrapped controllers.

If you are not use a bootstrapped controller, then PrestaShop will wrap it with a specific class, which will do its best to handle the controller as effectively as possible, thus ensuring a certain level of retrocompatibility.

Text field width

If you want to choose the width of your text fields, just add a class on the input, directly in the array described in your `HelperForm`.

For instance:

```
array (
    'type' => 'text',
    'label' => $this->l('Field name'),
    'name' => 'field_name',
    'class' => 'fixed-width-xs',          // Add this line.
    'required' => true
),
```

In this example, "xs" is used to choose the width of the field.

There are several available sizes you can use:

- xs: extra small.
- sm: small.
- md: medium.
- lg: large.
- xl: extra large.
- xxl: extra extra large.