New Developers Features In PrestaShop 1.5

Table of content

New Developers Features In PrestaShop 1.5

The new version of PrestaShop brings a lot of new and updated features to improve online shopping for both customers and shop owners, and the API have been dutifully updated and expanded in order to help developers and designers leverage these new possibilities.

The technical documentation itself has been updated to reflect the new state of working with PrestaShop. This chapter describes all the changes between v1.4 and v1.5, in order to help you get up to date quickly.

PrestaShop Core

Coming soon...

Module API

There are six main aspects of PrestaShop's Module API which received a significant overhaul for version 1.5: dynamic hooks, evolution of the override system, new file-tree and operations, multistore feature, Addons webservice, and the update system. We will explore these one by one.

Hooks

Hooks are a fundamental part of PrestaShop's modules API. They enable modules to 'hook into' the rest of PrestaShop, and call functions at specific times, thereby setting your own code in motion.

There are two families of hooks:

- **"Action" hooks**. For example, to send out a mail when a client creates a user account.
- **"Display" hooks**. For example, to display a module in a column.

New hooks

Read the dedicated page: Hooks in PrestaShop 1.5.

Dynamic hooks

PrestaShop brings the ability of defining dynamic hooks. In effect, there is now a hook before and after each action of each controller: PrestaShop's `ObjectModel` and `AdminController` base objects have been updated so that this new feature is automatically applied to existing code. They make it possible for your code to hook to any object or back-office action.

```
public function add($autodate = true, $null_values = false)
{
        // @hook actionObject*AddBefore
        Hook::exec('actionObjectAddBefore', array('object' => $this));
        Hook::exec('actionObject'.get_class($this).'AddBefore', array('object' => $this));
}
```

This way, you can for instance ask for a module to hook itself to the action that is triggered before the creation of manufacturer. This will result in `hookActionObjectManufacturerAddBefore`.

Here is the list of available dynamic hooks:

- actionObjectAddBefore
- actionObject**ObjectName**AddBefore
- actionObjectAddAfter
- actionObject**ObjectName**AddAfter
- actionObjectUpdateBefore
- actionObject**ObjectName**UpdateBefore
- actionObjectUpdateAfter
- actionObject**ObjectName**UpdateAfter
- actionObjectDeleteBefore
- actionObject**ObjectName**DeleteBefore
- actionObjectDeleteAfter
- actionObject**ObjectName**DeleteAfter

There are more dynamic hooks in `AdminControllers`, and new ones will be added in the future.

Here is the list of available dynamic hooks:

- actionAdmin**ActionName**Before
- action**ControllerNameActionName**Before
- actionAdmin**ActionName**After
- action**ControllerNameActionName**After
- actionAdminControllerSetMedia

Override

PrestaShop's override system, which was introduced with PrestaShop 1.4, makes it easier for third party developers to include changes to the core methods without changing the core files.

Until now, you could only override PrestaShop's classes and controllers by placing the appropriate file (s) in either the `/override/classes` folder or the `/override/controllers` one. The module API in PrestaShop 1.5 now enables you to override classes and controllers directly from a module.

For instance, should you need to override the `Cart.php` class, you could do this from a module: instead of placing your customized `Cart.php` file in the root's `/override/classes` folder, now you can simply put your `Cart.php` file directly in the `/override/classes` of your module's folder. PrestaShop will automatically merge of the module's overriding code and the overriding code that would already exist in the root `/override` folder.

The advantages are numerous, the major ones being:

- Easier to manage, install, and share along with the module itself.
- Multiple modules can make changes to the core classes, as long as they do not attempt to override the same functions.

New File-tree And Operations

In previous version of PrestaShop, there could be situations when you would have to call one of the module's file directly. For instance, when paying with the Cheque module, you would have to reach such a URL: http://myprestashop.com/modules/cheque/payment.php

It is now recommended to avoid directly calling files from the module. While it will still work in v1.5, we strongly advise you to refrain from it, and use this safer URL path: http://myprestashop.com/index.php?fc=module&module=cheque&controller=payment

This is achieved by using `FrontController` classes in your module. In the example of the Cheque module, the following file has been place in the module's file-tree: `/modules/cheque/controllers/front/payment.php`

That "class" name must be built using the following syntax: `class ModuleNamePageName extends ModuleFrontController`.

Here is an example:

```
class ChequePaymentModuleFrontController extends ModuleFrontController
{
  public $display_column_left = false;
  public $ssl = true;

  /**
  * @see FrontController::initContent()
  */
  public function initContent()
  {
    parent::initContent();

    $cart = $this->context->cart;
    if (!$this->module->checkCurrency($cart))
      Tools::redirect('index.php?controller=order');

    $this->context->smarty->assign(array(
      'nbProducts'    => $cart->nbProducts(),
      'cust_currency' => $cart->id_currency,
      'currencies'    => $this->module->getCurrency((int)$cart->id_currency),
      'total'         => $cart->getOrderTotal(true, Cart::BOTH),
      'isoCode'       => $this->context->language->iso_code,
      'chequeName'    => $this->module->chequeName,
      'chequeAddress' => Tools::nl2br($this->module->address),
      'this_path'     => $this->module->getPathUri(),
      'this_path_ssl' => Tools::getShopDomainSsl(true, true).__PS_BASE_URI__.'modules/'.$this->module->name.'/'
    ));

    $this->setTemplate('payment_execution.tpl');
  }
}
```

The `payment_execution.tpl` template file, called at the end of this controller, must be placed in the following path: `/modules/cheque/views/templates/front/payment_execution.tpl`

With this in place, when calling this URL: `http://myprestashop.com/index.php?fc=module&module=cheque&controller=payment`, PrestaShop will automatically load the controller, provided it is correctly placed and named.

Generally speaking, a PrestaShop 1.5 module should now use the following file-tree:

- `/config.xml`: Automatically created by PrestaShop when the module is loaded.
- `/mymodule.php`
- `/logo.jpg`: Logo file, 16*16 pixels. For versions of PrestaShop up to v1.4.
- `/logo.png`: Logo file, 32*32 pixels. For PrestaShop 1.5 and later.
- `/controllers/`
- `/controllers/front/myfile.php`: As explained above.
- `/upgrade/`: Upgrade files. See Update System.
- `/translations/`

- `/translations/fr.php`: French language file, created automatically when you start translating the module in your back-office.
- `/translations/de.php`: German (deutsch) language file, created automatically when you start translating the module in your back-office.
- `/translations/...`: ...and so on...
- `/views/`
- `/views/css/`: For CSS files.
- `/views/js/`: For JavaScript files.
- `/views/templates/admin`
- `/views/templates/admin/folder_name`: For template files used by the module's admin controllers. (e.g. "`sample_data`" for the "`SampleData`" admin controller)
- `/views/templates/front/`: For template files used by the module's controllers.
- `/views/templates/hooks/`: For template files directly used by the module. For backward compatibility purposes, this type of files can also be placed in the root folder of the module.

Multistore Feature

One of the major features of version 1.5 is the multistore feature, which enable shop owners to manage more than one shop with a single installation of PrestaShop. However, this has deep implications in the code of PrestaShop, and while all modules written for v1.4 should work as-is with v1.5, most will not work as expected in the multistore context.

If your 1.4 module only stores its data using `Configuration::get` and `Configuration::updateValue`, it should have no problem working with multiple shops.
On the other hands, if the module stores its data within its own database tables, you will have to update these tables: add a `id_shop` row in the table where you will store the shop's ID when saving data.

Failing that, your 1.4 module will work perfectly fine, but will have the very same configuration for every shops, which might not be what the module's users want.

Addons Webservice

With PrestaShop 1.5, many native modules are not physically part of the original package, such as the PayPal module. They are in the module list, but their files are nowhere to be found in the `/modules` folder.

These missing modules have country-specific features, and therefore are downloaded from the PrestaShop Addons website (http://addons.prestashop.com/en/) depending on your shop's country, as set during the installation process. This way, shop owner can only access modules that are relevant to their country.

In effect, when clicking on a module's "Install" button, PrestaShop automatically downloads the most recent version of the module, further enabling you to benefit from the latest features for this module without having to update it right away.

Update System

PrestaShop has new module update system. If an installed module has a new version available on the Addons website (http://addons.prestashop.com/en/), PrestaShop will display an "Update" button, and the user will be able to perform the update with a single click.
The "Update" button also appears when the module has been installed manually (by uploading the module's zip file), and PrestaShop detects that a new version is available.

You can also add an update file to your module: create an `/upgrade` folder in your module's folder, and put your update files in it, using the `install-ModuleVersion.php` name norm.

For instance, the `install-1.8.0.php` file should be built like so:

```
<?php
// Sample file for module update

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

// object module ($this) available
function upgrade_module_1_8_0($object)
{
  // Your code to upgrade from version 1.8.0 of the module
}
?>
```

When updating, PrestaShop will call all your update files one after the other, from the current version to the latest version. For instance, let's say the module's `/upgrade` folder contains the following files:

- `install-1.8.0.php`
- `install-1.8.1.php`
- `install-1.8.2.php`
- `install-1.8.3.php`

If you are updating from version 1.8.1 to version 1.8.3 of the module, PrestaShop will only load the two last files, `install-1.8.2.php` and `install-1.8.3.php`.

Theme API

See Changes in version 1.5 which impact theme development