Overriding default behaviors

Table of contents

Overriding default behaviors

PrestaShop enables you to override various default components and behaviors. This system consists of two major points:

1. Overriding PrestaShop's own behavior (class files and controller files) in order to target a specific section of the required components.
2. Overriding the visible parts of modules (Templates, JavaScript, style sheet language...) so that the themes can adapt better to them.

Keep overrides for your own shop

Overrides in PrestaShop are exclusive. This means that if your module overrides one of PrestaShop's behaviors, another module will not be able to use that behavior properly, or override it in an predictable way.

Therefore, **overrides should only be used for your own local modules**, when you have a specific need that cannot be applied without it.

It is **not recommended to use an override in a module that you intend to distribute** (for instance through the PrestaShop Addons marketplace), and **they are forbidden in partner modules**.

Overriding PrestaShop's classes and controllers

Overriding is a way to "override" class files and controller files. PrestaShop's ingenious class auto-loading function makes the "switch" to other files fairly simple. Thanks to PrestaShop's fully object-oriented code, you can rely on object inheritance to modify and add new behaviors, using the properties and methods of the various existing classes.

Classes and controllers are usually built following a certain norm. Here is the Product class and controller:

- `/classes/Product.php`
  This class would be called `ProductCore`.
- `/controllers/front/ProductController.php`
  This controller would be called `ProductControllerCore`.

You will need to create a PHP file and place it either of the override folders, depending on whether this file is part of a module, or distributed as-is. Indeed, since PrestaShop 1.5, there are two locations where you can place your overriding files: either at the root of the PrestaShop install, or within a module.

Overriding a class

In order to override the `Product` class, your file needs to be called `Product.php` and must feature a `Product` class that then extends `ProductCore` class.

The file can be placed in either of these locations:

- `/override/classes/Product.php`
- `/modules/my_module/override/classes/Product.php`

Overriding a controller

In order to override the `ProductController` class, your file needs to be called `ProductController.php` and must feature a `ProductController` class that then extends `ProductControllerCore` class.

The file can be placed in either of these locations:

- `/override/controllers/front/ProductController.php`
- `/modules/my_module/override/controllers/front/ProductController.php`

Overriding other behaviors

PrestaShop has certain files you can use to override elements such as displaying redirections (`Tools.php`) and measuring hook execution time (`_Module.php`), etc. you can enable them by removing the "_" prefix. For instance, rename `_Tools.php` into `Tools.php`. If there already exists a `Tools.php` override, you will have to merge it with yours.

Overriding a module's behavior

The modules are usually in the following format:

- `/modules/my_module/my_module.tpl`
- `/modules/my_module/my_module.css`
- `/modules/my_module/my_module.js`

⚠ Since PrestaShop 1.5, they can and should also be in the following format:

- `/modules/my_module/views/templates/front/my_module.tpl`
- `/modules/my_module/views/templates/front/my_module.css`
- `/modules/my_module/views/templates/front/my_module.js`

PrestaShop allows you to override or replace certain front office module files with new ones within the same theme. The override is governed by the theme: once it contains a `/modules` folder (or more!), PrestaShop will browse its content for files which have the same name and path as those of existing modules, and replace these with the new ones.

This means, for PrestaShop 1.4-compatible modules:

- `/themes/my_theme/modules/my_module/my_module.tpl`
- `/themes/my_theme/css/modules/my_module/my_module.css`
- `/themes/my_theme/js/modules/my_module/my_module.js`

Since PrestaShop 1.5, the path is slightly longer

- `/themes/my_theme/modules/my_module/views/templates/front/my_module.tpl`
- `/themes/my_theme/css/modules/my_module/views/templates/front/my_module.css`
- `/themes/my_theme/js/modules/my_module/views/templates/front/my_module.js`

In general, the proper path to override a `.tpl`, `.js` or `.css` file depends on the module's own path. That is the reason why if PrestaShop has to work with a module without a `view` folder, it will need the same override path.
In short, you can keep overriding code in 1.6 just as you did in 1.4.

The new files will be used when the customer loads your shop.

> ⚠ Contrary to the override code that is to be placed manually in the `/override` folder, module overrides are enabled as soon as the module is installed. During installation, overriding code is merge with those already in place (if any), otherwise they are copied to the `/override` folder at the root of the PrestaShop folder.

Manipulating the override code manually

Modules and themes may add an override to a default behavior, and PrestaShop takes care of reseting the `/cache/class_index.php` file.

But sometimes you need to add that overriding code yourself, manually uploading the file to your server. In that case, you need to trigger the regeneration of the /cache/class_index.php file yourself. This is done simply by deleting the file: if PrestaShop cannot find the file, it will regenerate it, taking all the overrides into account.

It is the same when manually removing an override: in order to reinstate the default behavior, you must delete the `/cache/class_index.php` file.

Sample code

Example 1

Using the `MySQL.php` data class is simply impossible while trying to enter data into a different database from PrestaShop's on the same MySQL Server. (Really!)

The solution is to use the following override of the `MySQLCore` class:

```
<?php
class MySQL extends MySQLCore
{
        public function __construct($server, $user, $password, $database, $newlink = false)
        {
                $this->_server = $server;
                $this->_user = $user;
                $this->_password = $password;
                $this->_type = _DB_TYPE_;
                $this->_database = $database;

                $this->connect($newlink);
        }

        public function connect($newlink = false)
        {
                if (!defined('_PS_DEBUG_SQL_'))
                        define('_PS_DEBUG_SQL_', false);

                if ($this->_link = mysql_connect($this->_server, $this->_user, $this->_password, $newlink) )
                {
                        if (!$this->set_db($this->_database))
                                die(Tools::displayError('The database selection cannot be made.'));
                }
                else
                        die(Tools::displayError('Link to database cannot be established.'));

                /* UTF-8 support */
                if (!mysql_query('SET NAMES \'utf8\'', $this->_link))
                        die(Tools::displayError('PrestaShop Fatal error: no utf-8 support. Please check your
server configuration.'));
                return $this->_link;
        }
}
?>
```

To use it you have to instantiate the class as follows:

- For local connection: `new MySQL(`*`DB_SERVER`*`, `*`DB_USER`*`, `*`DB_PASSWD`*`, 'DB_name', true);`
- For remote connection: `new MySQL(`*`DB_SERVER`*`, `*`DB_USER`*`, `*`DB_PASSWD`*`, 'DB_name', true);`

The last parameter forces the creation of a MySQL connection.

Example 2

```
/*
 * Create a cron task to make periodical database backup
*/
class AdminTab extends AdminTabCore{
        public function ajaxProcess()
        {
                // Here we call the same thing as if we did the old way
                // + with "if": maybe we want to limit its use to only adding backup
                // note: find yourself a way to get the file link if you want to send it by mail!
                if (isset($_REQUEST['addbackup']))
                        return $this->postProcess();
        }

        public function displayAjax()
        {
                if (sizeof($this->_errors) > 0)
                {
                        // handle errors
                        // for example, send mail with all error msg
                        $content = '';
                        foreach($this->_errors as $errorMsg)
                         $content .= $errorMsg;

                         $lang = Configuration::get('PS_LANG_DEFAULT');
                         // here we send a mail to give the result of the process
                         // notice: you have to create template mails files
                         Mail::Send($lang, 'backuptaskdone', '[autobackup] report backup error', array
('backup_link'=>)), $to);
                }
                else
                {
                        // no error, but maybe we want a mail?
                        if(Configuration::get('PS_NOTICE_SUCCEED_BACKUP'))
                        {
                                // fileAttachment available, see 9th param of Send() method in classes/Mail.php
                                // + we can add a condition "if (Configuration::get('PS_AUTOBACKUP_SEND_FILE'))"
                                Mail::Send($lang, 'backuptaskerror', '[autobackup] report backup error', array
('vars to use in tpl'), $to);
                        }
                }
        return true;
        }
}
```

```
/*
 * This override allows you to use ajax-tab.php to make any admin action.
 * Use it for crontask for example
*/
class AdminTab extends AdminTabCore {
        public function ajaxProcess()
        {
                return $this->postProcess();
        }
}
```

Example 3

```
/*
 * With this override, you have a new Smarty variable called "currentController" available in header.tpl
 * This allows you to use a different header if you are on a product page, category page or home.
 */
class FrontController extends FrontControllerCore {
        public function initHeader()
        {
                self::$smarty->assign('currentController', get_class($this));
                return parent::initHeader();
        }
}
```