Modules, Classes and Controller Override

Modules, Classes and Controller Override

This article was written by Julien Breux, and was first published on August 10th, 2011, on the PrestaShop blog.

Introduction

PrestaShop allows you to override various components and behaviors. PrestaShop version 1.4 consists of two major points:

- The first is overriding the visible parts of modules (Templates, JavaScript and style sheet language) so the themes can adapt better to them.
- The second is overriding software behavior (class files and controller files) to target a specific section of the required components.

Module override

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

PrestaShop allows you to override or replace certain visible module files with new ones with the same theme. It couldn't be simpler, just do the following:

- /themes/prestashop/modules/my_module/my_module.tpl
- /themes/prestashop/css/modules/my_modules/my_module.css
- /themes/prestashop/js/modules/my_modules/my_module.js

The new files will be used when you display your website.

Class override

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.
Use inheritance to modify and add new behavior using the various classes' properties and methods.

They are usually constructed the following way (product example):

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

You will need to create a file in the "override/classes/" folder to "override" the resulting class model such as:

- /override/classes/Product.php

This class would be called "Product" and spans the "ProductCore" class. Just a flick of PrestaShop's magic wand and the spell is working!

You can also use this principle with the controllers and "override" in the following way:

- /override/controllers/ProductController.php

This controller would be called "ProductController" and spans the "ProductControllerCore" class.

PrestaShop has certain folders you can use to override elements such as displaying redirections (`_Tools.php`) and measuring hook execution time (`_Module.php`) etc.

Example 1

Using data class `MySQL.php` is simply impossible while trying to type data into a database different from PrestaShop 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.'));
    // removed SET GLOBAL SQL_MODE : we can't do that (see PSCFI-1548)
    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

```
/*
 * 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

```
/*
 * Create a cron task to make periodical database backup
 * (please test before use, I didn't tested it yet)
*/
class AdminTab extends AdminTabCore{
    public function ajaxProcess()
    {
        // here we call the same thing as if we do 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;
}
}
```

Example 4

```
/*
 * with this override, you have a new Smarty variable "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 displayHeader()
  {
    self::$smarty->assign('currentController',get_class($this));
    return parent::displayHeader();
  }
}
```

Conclusions

Core files are not modified by overriding. This technique allows you to personalize your PrestaShop boutique and monitor how the software evolves. Updates are facilitated.