

Secteur Tertiaire Informatique
Filière « Etude et développement »

Séquence « Mettre en œuvre une solution
e-commerce »

Développer des Modules - PrestaShop

Apprentissage

Mise en pratique

Evaluation

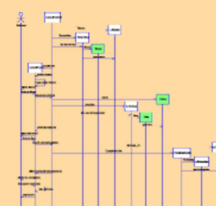
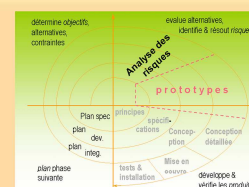
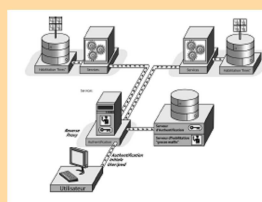


TABLE DES MATIERES

Table des matières	2
1. Introduction	4
2. Architecture générale de PrestaShop.....	4
2.1 Noyau PrestaShop et héritage de classes.....	4
2.2 Structure d'une boutique PrestaShop.....	5
2.3 Rappels sur les templates Smarty.....	7
3. Comprendre les modules PrestaShop.....	10
3.1.1 Notion de 'surchargé' de classe PrestaShop.....	10
3.1.2 Structure d'un module standard Prestashop	11
3.1.3 Comment s'y retrouver ?.....	12
4. Développement de modules PrestaShop	14
4.1 Structure-type d'un module	14
4.2 Ecriture d'un premier module PrestaShop.....	14
4.2.1 Créer un premier module	15
4.2.2 Les méthodes install() et uninstall()	20
4.2.3 La classe Configuration	21
4.2.4 Le fichier d'icônes	22
4.2.5 Installation du module.....	23
4.2.6 Implémenter des hooks	23
4.2.7 Afficher du contenu.....	25
4.2.8 Désactiver le cache PrestaShop.....	27
4.2.9 Intégrer un template à un thème ; les contrôleurs	28
4.2.10 Bien utiliser Smarty.....	29
4.2.11 Ajouter une page de configuration au Back-office.....	32
4.2.12 La méthode getContent().....	32
4.2.13 Afficher le formulaire.....	34
4.2.14 Utiliser HelperForm.....	36
4.3 Bilan de l'exercice	39
4.4 La classe Db et l'accès aux bases de données	40

Objectifs

A l'issue de cette séance, le stagiaire sera capable d'intervenir sur des modules existants et de créer de nouveaux modules intégrés à une boutique PrestaShop.

Pré requis

Avoir pris en main le Front-office et le Back-office de PrestaShop.

Maîtriser les techniques de base du développement Web, HTML, CSS et programmation PHP orientée objet.

Outils de développement

Un navigateur doté d'un débogueur complet.

Un éditeur de code source non-WYSIWYG, adapté au développement en techniques Internet (NotePad ++, CodeLobster, Brackets, Eclipse...).

Méthodologie

Ce support propose une avancée progressive dans la problématique du développement de modules PrestaShop.

Il renvoie fréquemment à la documentation de référence de PrestaShop (disponible en ligne sur <http://doc.prestashop.com>) et il la complète.

NB : les apports donnés ici se basent sur la version 1.6.xx du logiciel.

Mode d'emploi

Symboles utilisés :



Renvoie à des supports de cours, des livres ou à la documentation en ligne constructeur.



Propose des exercices ou des mises en situation pratiques.



Point important qui mérite d'être souligné !

Ressources

La documentation de référence PrestaShop : <http://doc.prestashop.com>

Lectures conseillées

1. INTRODUCTION

L'utilisation courante du Back-office PrestaShop permet de se familiariser avec la notion de module et d'en administrer certains directement dans les pages du Back-office. C'est déjà un premier niveau de travail sur les modules plus spécialement destiné aux non-informaticiens.

Si le besoin de personnalisation ne concerne pas les traitements mais uniquement la présentation des résultats, l'informaticien développeur pourra intervenir sur le composant de présentation du module concerné, le '*template*', sans risque d'effet de bord sur le reste du module et sans risque de perturbation de la boutique. C'est tout l'intérêt de l'architecture '3 tiers' adoptée par les auteurs du logiciel.

Mais comment tout cela est-il agencé ? Où et comment intervenir en tant que développeur ? Comment créer de nouveaux modules intégrés au logiciel ? C'est bien l'enjeu de la suite de ce document.

2. ARCHITECTURE GENERALE DE PRESTASHOP

2.1 NOYAU PRESTASHOP ET HERITAGE DE CLASSES

Le cœur de PrestaShop est composé d'un ensemble complexe de **classes PHP** dont les **classes 'Métier'** qui représentent et gèrent les données (client, produit, catégorie...) et les **classes 'Contrôleur'** qui gèrent tous les traitements. Il s'agit bien là de code 100% PHP.

Les auteurs ont tout d'abord identifié les données et traitements communs pour définir des **classes de base** qu'ils ont ensuite **spécialisées par dérivation**, selon les bonnes pratiques de la programmation orientée objet.

Ainsi, la classe Métier `ProductCore` dérive de la classe `ObjectModel` tout comme la classe `CategoryCore` et `ProductControllerCore` dérive de `FrontController` tout comme `CategoryControllerCore`.

Le développeur peut à loisir prolonger ces dérivations de classes pour ses propres besoins, par exemple créer une classe `MyProduct` qui dérive de `ProductCore`. Grâce au mécanisme d'héritage, cette nouvelle classe dispose immédiatement de tous les attributs et méthodes de la classe `ProductCore` et de la classe `ObjectModel`. Il suffit d'y ajouter les éléments spécifiques à cette classe particulière selon les besoins. Ce processus de programmation par dérivation de classes est à la base du développement à l'aide de Framework, cas où le développeur doit tout créer lui-même.

La solution de e-commerce PrestaShop propose au développeur des moyens supplémentaires pour intervenir sur l'existant, comme la '**surcharge de contrôleur**' qui assure l'évolution future de PrestaShop. C'est bien l'objet de cette étude et il est ici question d'avancer pas à pas dans la compréhension du fonctionnement et de la structure des traitements PrestaShop.

2.2 STRUCTURE D'UNE BOUTIQUE PRESTASHOP

Une boutique PrestaShop est donc constituée d'une base de données MySQL et d'une multitude de fichiers et dossiers stockés dans un dossier publié par le serveur Web :

Nom	
admin5134	← les composants du back-office
cache	
classes	← les classes 'Métier' représentant les données PrestaShop (le Modèle' de MVC)
config	
controllers	← les classes 'Contrôleur' assurant la production des pages de la boutique
css	← les feuilles de styles globales (communes aux différents thèmes)
docs	
download	
img	← les images globales de la boutique (communes aux différents thèmes)
js	← les scripts JavaScript globaux (communs aux différents thèmes)
localization	
log	
mails	
modules	← les composants des modules installés (contrôleur et vue de MVC)
override	← les 'surcharges' de classes
pdf	
themes	← les composants des thèmes installés
tools	← une boîte à outils contenant des scripts PHP très utiles au développeur
translations	
upload	
webservice	
.htaccess	
{ } CONTRIBUTING.md	
{ } CONTRIBUTORS.md	
error500.html	
footer.php	
header.php	
images.inc.php	
index.php	← le 'contrôleur principal' qui distribue les traitements à réaliser ('dispatcher')
init.php	
modules.txt	
{ } README.md	
sitemap.xml	

⚡ Toutes les demandes de pages sont adressées au contrôleur principal `index.php` qui redirige la requête vers le bon contrôleur.

Chaque contrôleur PrestaShop se charge de générer une page-type de la boutique et il fait appel au code PHP (= contrôleur du modèle MVC) des différents modules greffés sur la page.

Développer des modules - PrestaShop


Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

Par exemple, l'URL suivante :

`http://localhost:8080/prestashop16/index.php?id_category=14&controller=category&id_lang=1`
appelle le contrôleur `CategoryController.php` avec les paramètres supplémentaires de langue et de numéro de catégorie demandée.

Le script `CategoryController.php` définit une **classe contrôleur** `CategoryControllerCore` (dérivée de la classe contrôleur de base pour Front-office) :

```
class CategoryControllerCore extends FrontController
{
```

 Ainsi, selon les principes de la programmation orientée objet, en *héritant* de cette classe de base qui fait partie du noyau de PrestaShop, le contrôleur 'sait' déjà faire des choses (grâce aux 'méthodes' héritées) et 'connaît' des données (grâce aux attributs hérités).

Un contrôleur ne s'occupe que de la 'logique applicative', c'est-à-dire des **traitements** à effectuer et fait le lien avec les composants de mise en page (les '*templates*') et les éventuels scripts CSS et JavaScript ; dans l'exemple ci-dessous, la méthode `setMedia()` fait le lien avec les CSS spécifiques à l'affichage des catégories.

Extrait de la classe `CategoryControllerCore` :

```
public function setMedia()
{
    parent::setMedia();

    if (!$this->useMobileTheme())
    {
        //TODO : check why cluetip css is include without is file
        $this->addCSS(array(
            _THEME_CSS_DIR_.'scenes.css' => 'all',
            _THEME_CSS_DIR_.'category.css' => 'all',
            _THEME_CSS_DIR_.'product_list.css' => 'all',
        ));
    }
}
```

NB : il existe aussi une méthode `addJS()` permettant de relier du script JavaScript à une page.

Et la méthode `initContent()` fait le lien avec le template stocké dans le dossier du thème courant :


```
public function initContent()
{
    parent::initContent();


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

NB : Les templates Smarty ont par défaut une extension `.tpl`.

NB : `_PS_THEME_DIR_` et `_THEME_CSS_DIR_` sont des constantes globales PrestaShop qui définissent des chemins d'accès communs.

Dans chacune de ces méthodes publiques, la classe dérivée utilise les services de la classe de base en appelant sa méthode correspondante comme dans : `parent::initContent();`


 Grâce au mécanisme d'héritage de classes du noyau PrestaShop, un contrôleur ne se préoccupe que de ses traitements spécifiques et utilise les services du noyau PrestaShop.

 Toute la présentation (vue de MVC) est reléguée dans les scripts du template et des feuilles de styles CSS.

2.3 RAPPELS SUR LES TEMPLATES SMARTY

PrestaShop a choisi, pour assurer la présentation des pages, le '*moteur de template*' Smarty. Il s'agit d'un logiciel complémentaire (gratuit) installé sur le serveur Web qui génère le code final HTML/JavaScript/CSS à envoyer au navigateur à partir de modèles de pages (templates) et de variables PHP reçues par le template.

Pour assurer la fusion des variables dans le gabarit, Smarty propose un langage spécifique relativement simple, très proche du PHP, et assez puissant. Il faut donc aussi se familiariser avec le fonctionnement et le langage Smarty pour intervenir sur les affichages fournis ou en créer de nouveaux ; c'est l'objet de la séance consacrée à au développement de gabarits de mise en page.

 Dans PrestaShop, la mise en page est assurée par des templates Smarty constitués d'une imbrication complexe d'éléments HTML `<div>` (bien souvent optionnels selon le paramétrage du Back-office) et d'instructions Smarty permettant de fusionner des variables PHP gérées par les contrôleurs et modules. Chaque template ne s'occupe que de sa ou ses `<div>` spécifiques.

Pour résumer, un template Smarty :

- contient le **code HTML constant** à adresser au navigateur,
- contient des **instructions de contrôle** (test, boucle) en langage Smarty écrites entre `{...}`,
- peut être découpé en plusieurs scripts assemblés par l'instruction Smarty `{include file='...'}`
- peut manipuler des variables PHP,
- **insère très simplement le contenu d'une variable PHP** `{$...}` (similaire à l'ordre PHP `echo`),
- peut faire appel à d'autres scripts PHP par appel direct,
- **peut inclure des chaînes de caractères à traduire** selon la langue courante de l'utilisateur `{ls='...' mod='...'}`,
- peut contenir du commentaire HTML (qui sera adressé au navigateur) `<!-- ... -->` ou même du commentaire Smarty spécifique (non adressé au navigateur) `{* ... *}`

De plus cette variante de Smarty offre de nombreuses variables d'environnement adaptées aux données manipulées par PrestaShop (images de produits...) ainsi que des ordres de formatage des données ('filtres') ou d'aide à la mise au point. Par exemple :

- {`$maVariable` | `truncate :120` | `escape :'htmlall'` } ne prend que les 120 premiers caractères de la variable et transforme les caractères spéciaux HTML comme le signe <,
- {`$monTableau` | `@count` } restitue le nombre d'occurrences du tableau,
- {`$maVariable` | `@print_r` } et {`$maVariable` | `@debug_print_var` } insèrent des informations de débogage dans la page.
- {`debug`} insère dans le code HTML envoyé au navigateur l'ensemble des variables connues de la page en cours.

Le moteur Smarty peut accéder aux variables globales PHP (et donc aux variables globales PrestaShop) mais il est préférable de transmettre les données sous forme de **paramètres**, depuis le contrôleur ou le module vers le template.

C'est le rôle de la méthode `assign()` de l'objet smarty du contexte des contrôleurs. Ainsi dans la méthode `initContent()` du contrôleur `CategoryController.php` :

```
$this->context->smarty->assign(array(
    'category' => $this->category,
    'description_short' => Tools::truncateString($this->category->description, 350),
    'products' => (isset($this->cat_products) && $this->cat_products) ? $this->cat_p
    'id_category' => (int)$this->category->id,
    'id_category_parent' => (int)$this->category->id_parent,
    'return_category_name' => Tools::safeOutput($this->category->name),
    'path' => Tools::getPath($this->category->id),
    'add_prod_display' => Configuration::get('PS_ATTRIBUTE_CATEGORY_DISPLAY'),
    'categorySize' => Image::getSize(ImageType::getFormattedName('category')),
    'mediumSize' => Image::getSize(ImageType::getFormattedName('medium')),
    'thumbSceneSize' => Image::getSize(ImageType::getFormattedName('m_scene')),
    'homeSize' => Image::getSize(ImageType::getFormattedName('home')),
    'allow_oosp' => (int)Configuration::get('PS_ORDER_OUT_OF_STOCK'),
    'comparator_max_item' => (int)Configuration::get('PS_COMPARATOR_MAX_ITEM'),
    'suppliers' => Supplier::getSuppliers(),
    'body_classes' => array($this->php_self.'-'.$this->category->id, $this->php_self
));
```

Dès lors, le template Smarty peut manipuler les variables `$category`, `$description_short`, `$products`... comme on peut le voir dans l'extrait de code du template `category.tpl` :

```
{include file="$tpl_dir./errors.tpl"}
{if isset($category)}
    {if $category->id AND $category->active}
        {if $scenes || $category->description || $category->id_image}
            <div class="content_scene_cat">
                {if $scenes}
                    <div class="content_scene">
                        <!-- Scenes -->
```

Pour un module PrestaShop, le processus reste similaire car un module dispose aussi de son contexte qui contient une référence sur le buffer de transmission de variables au moteur Smarty. Ainsi, depuis la version 1.6, on peut écrire dans une méthode de module :

```
$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')
    )
);
```



La méthode **\$this->l()** des contrôleurs et modules permet d'utiliser le système de traduction de PrestaShop. Le texte de base écrit dans le code PHP **'This is a simple text message'** pourra être traduit dans le Back-office de PrestaShop et restitué sur le Front-office dans la langue de l'utilisateur.

Par compatibilité avec les versions précédentes, qui offraient une référence sur ce buffer directement depuis l'objet module, on peut aussi écrire :

```
if ((Tools::getValue('id_product') || Tools::getValue('id_category')) && isset($this->context->cookie->last_visited_category)) {
    $category = new Category($this->context->cookie->last_visited_category, $this->context->link->getModuleLink('mymodule', 'display'));
    if (Validate::isLoadedObject($category)) {
        $this->smarty->assign(array('currentCategory' => $category, 'currentCategoryLink' => $this->context->link->getModuleLink('mymodule', 'display')));
    }
}

$this->smarty->assign('isDhtml', Configuration::get('BLOCK_CATEG_DHTML'));
```




Finalement, la programmation dans PrestaShop revient essentiellement à dériver une/des classes, à effectuer le lien avec les composants de présentation (templates, feuilles CSS), à leur transmettre les données variables, à établir le/les feuilles de styles et gabarits de mise en page correspondants aux pages ou portions de page HTML dynamiques Smarty en y identifiant les variables à fusionner.

Reste à comprendre où trouver les scripts à modifier, où stocker les nouveaux scripts et comment construire un nouveau module.

3. COMPRENDRE LES MODULES PRESTASHOP

3.1.1 Notion de 'surcharge' de classe PrestaShop

Un grand principe, et un gros avantage des versions actuelles de PrestaShop, réside dans la notion de '**surcharge de classe**', **appliquée aux classes Métier et aux contrôleurs** :

 Dans PrestaShop, tout contrôleur peut être réécrit par duplication sans intervenir sur le code initial. Dès lors, la mise à jour du contrôleur d'origine lors du changement de version n'a aucune incidence sur le code réécrit.

La surcharge de classe signifie que le développeur peut à loisir redéfinir complètement une classe, qu'elle soit livrée en standard ou écrite par lui-même, de manière à modifier son comportement, et ce, en préservant le code d'origine. Ainsi, pour les composants standards de PrestaShop, la mise à jour d'une version n'aura aucun effet sur les classes surchargées par le développeur.

Reprenons l'exemple du contrôleur des catégories déjà évoqué plus haut et dans la séance consacrée au développement des gabarits de mise en page.

Le contrôleur des catégories est appelé par le contrôleur principal `index.php` lorsque l'utilisateur choisit une catégorie dans le Front-office ; la demande de page est alors de type :

`http://localhost:8080/prestashop16/index.php?id_category=14&controller=category&id_lang=1`

Le contrôleur d'origine est bien situé dans le dossier `controllers/front` de la boutique.

Pour surcharger ce contrôleur, il suffit de :

- recopier le fichier `CategoryController.php` dans le dossier `override/controllers/front` de la boutique.
- ouvrir cette copie à l'aide de l'éditeur de code pour modifier uniquement la déclaration de classe :

```
class CategoryController extends FrontController { ... }
```

Il suffit donc de supprimer le suffixe `Core` et d'enregistrer dans le dossier `/override` pour surcharger le contrôleur !

Restera ensuite à modifier le corps de ce nouveau contrôleur afin d'assurer le comportement voulu.

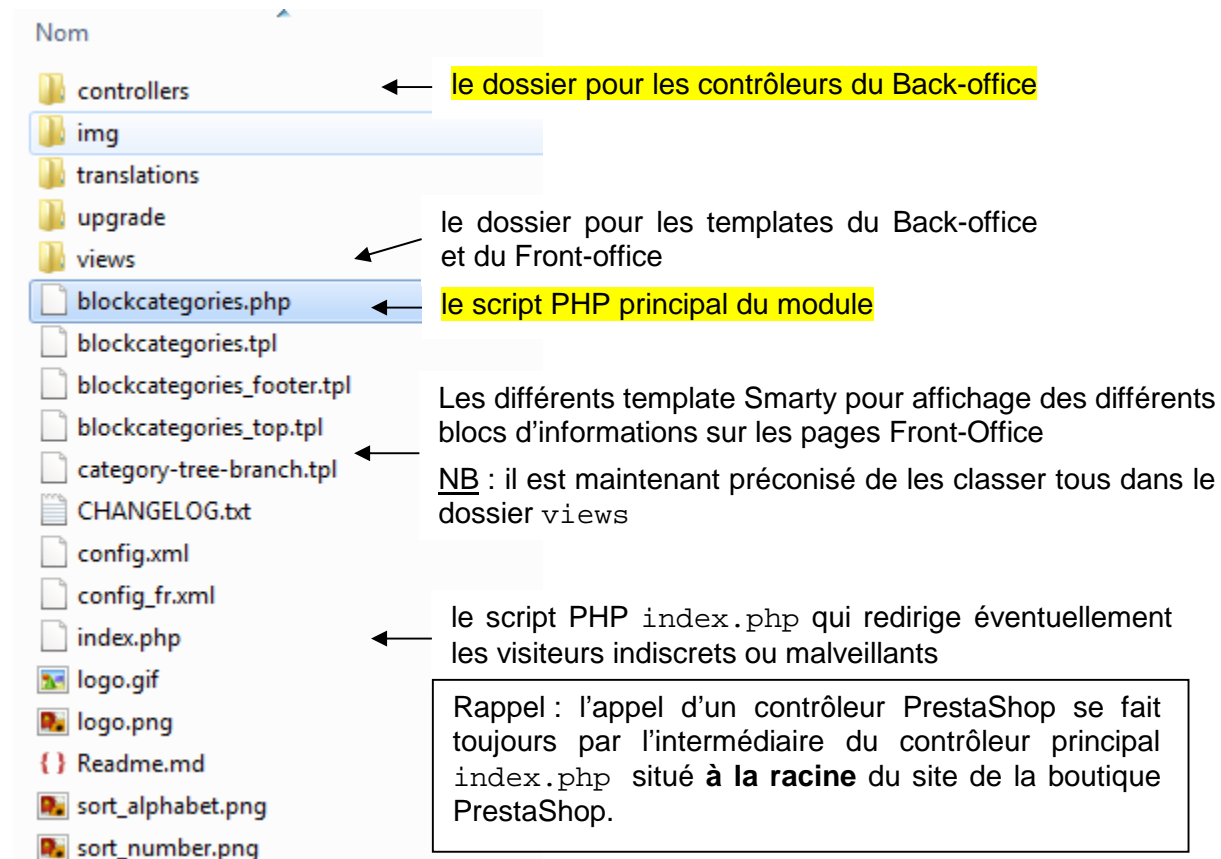
Voilà la surcharge du contrôleur effectuée ; c'est immédiat, il suffit de recharger la page d'accueil et de choisir une catégorie pour constater le nouveau comportement.

Pour rétablir le fonctionnement initial, il suffit de supprimer le contrôleur surchargé. Grâce au mécanisme de surcharge de PrestaShop, rien n'a été perdu !

3.1.2 Structure d'un module standard Prestashop

Un module PrestaShop assure un traitement élémentaire aboutissant ou non à un affichage sur une page de la boutique. Ainsi, les modules `blockcategories` ou `blockcurrencies` affichent des informations en Front-Office et le module `ExpressMailing` réalise les campagnes d'emailing depuis le Back-Office.

Un module PrestaShop est constitué d'un ensemble de scripts stockés dans un sous-dossier du dossier `/modules` de la boutique. Ainsi pour le module `blockcategories` qui affiche les catégories de produits :



NB : script PHP des modules et Contrôleurs PrestaShop constituent la couche Contrôleur du modèle d'architecture MVC ; templates et feuilles CSS représentent les Vues MVC.

NB : PrestaShop a déjà beaucoup évolué (et évoluera encore) ; en particulier, l'organisation des composants se structure de plus en plus depuis la version 1.5. Par exemple, il est vivement préconisé de stocker tous les templates dans les sous-dossiers du dossier `views`. Mais les modules livrés en standards ne sont pas tous réécrits selon cette nouvelle convention ; c'est pourquoi, dans cet exemple, les templates sont stockés 'en vrac' dans le dossier principal... Un modèle de structure de module sera préconisé plus loin pour les développements futurs.

3.1.3 Comment s'y retrouver ?

Quand on y regarde de plus près, on constate de multiples template et feuilles de styles et même scripts PHP de modules pour une même fonctionnalité, portant le même nom mais situé dans des dossiers différents.

Pour ce module `blockcategories`, les feuilles de styles CSS sont stockées dans les sous-dossiers `css` du dossier `modules` de chacun des thèmes de la boutique.

De plus, une multitude de modules d'affichage assurant chacun leur petite fonctionnalité à l'intérieur des pages pilotées par les contrôleurs.

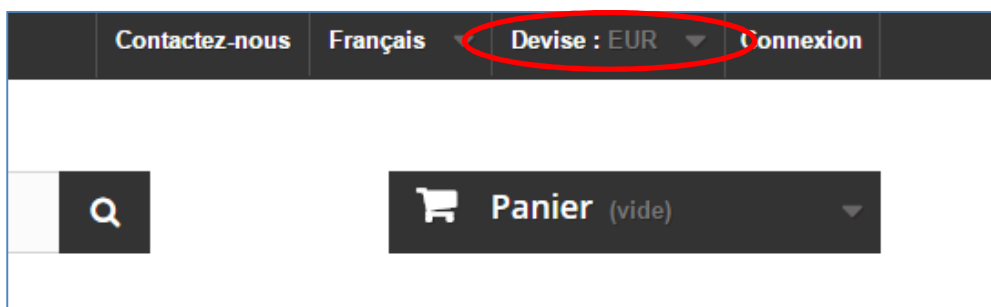
Comment s'y retrouver pour savoir où intervenir ?

Là encore, une bonne méthode est d'utiliser le **débogueur du navigateur** qui permet de consulter la structure HTML de la page, et ses **commentaires par blocs**, donc le **nom du module** qui génère chacun d'eux.

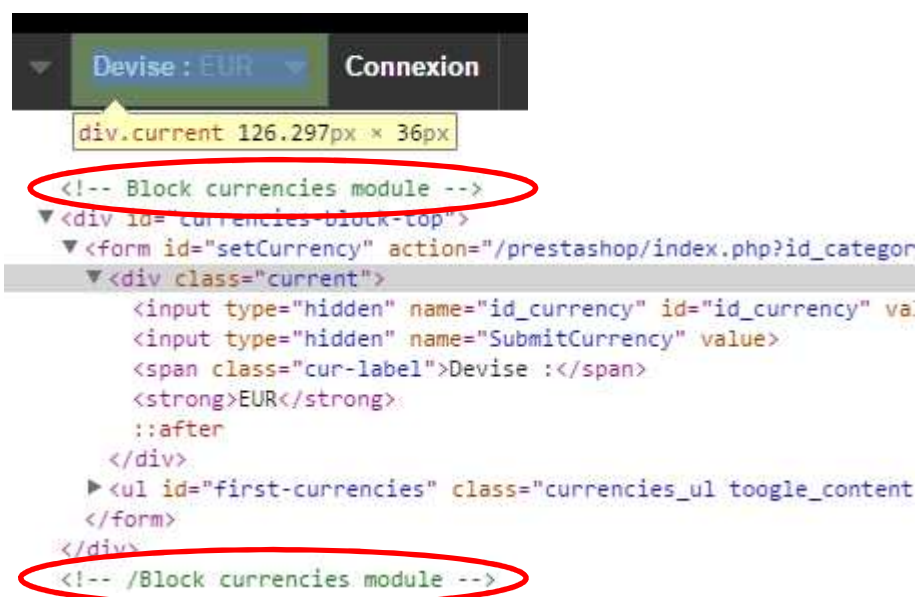
NB : Les fonctionnalités de mise en page du Back-office (Live Edit, Positions des modules) peuvent aussi aider à cette identification.

Exemple avec le débogueur pour le module d'affichage des devises :

A l'écran du navigateur :

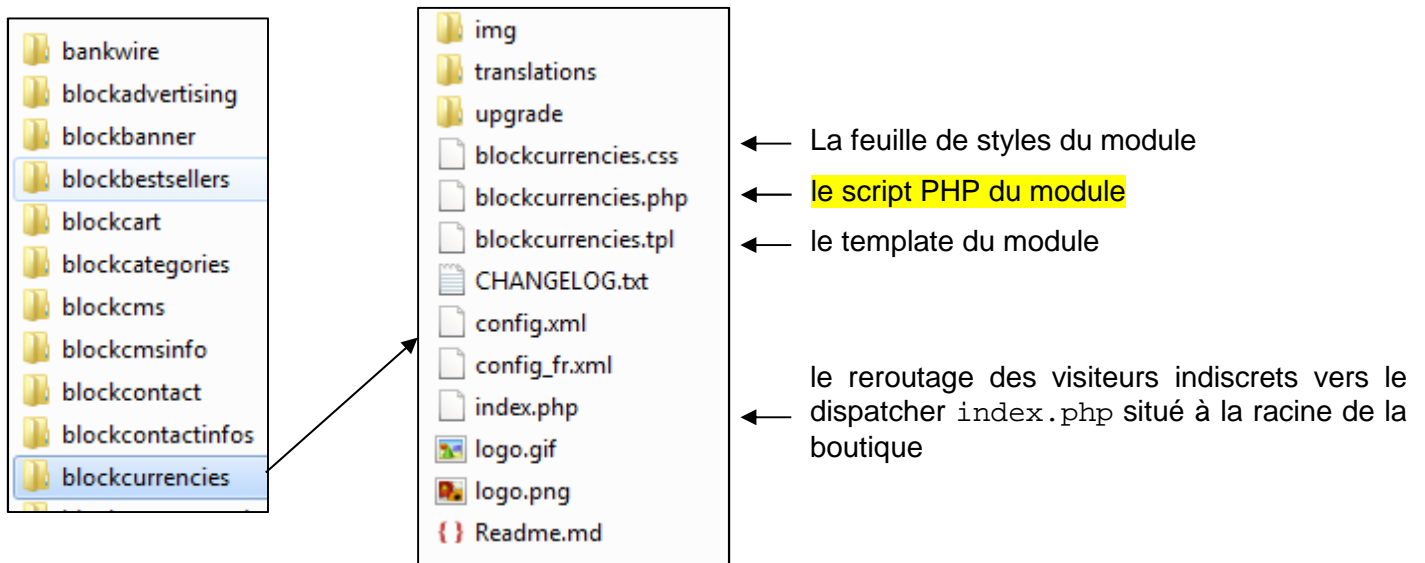


Dans le débogueur :



Développer des modules - PrestaShop

Dans le dossier `/modules` de la boutique, on retrouve bien le dossier du module correspondant :



***NB :** on retrouve ici une variante 'historique' moins bien classée de la structure d'un module PrestaShop.*

C'est bien là que ça se passe : les traitements du module sont décrits dans le script PHP `blockcurrencies.php` ; la présentation est assurée par template et feuille de styles CSS.

On sait déjà que la présentation peut être déclinée pour chaque thème si l'on place des fichiers de template et de CSS dans le dossier du thème.

Il en va de même pour le code PHP des modules : un thème contient un sous-dossier `.../modules` dans lequel le développeur peut placer les éventuelles variantes du module selon le thème.

Voilà pour l'architecture et le fonctionnement généraux des traitements dans PrestaShop. La suite du document va permettre d'apprendre à écrire de nouveaux modules.

4. DEVELOPPEMENT DE MODULES PRESTASHOP

4.1 STRUCTURE-TYPE D'UN MODULE

Partons sur de bonnes bases. Les fichiers et dossiers par défaut d'un module PrestaShop 1.6 sont maintenant :

- Le fichier de démarrage : `nom_du_module.php`
- Le fichier de configuration du cache : `config.xml` (généré par PrestaShop)
- Les contrôleurs spécifiques au module, stockés dans le dossier `/controllers`
- Les fichiers de vue (templates, JavaScript, CSS). Ils peuvent être placés dans les sous-dossiers du dossier `/views` du module :
 - dossier `/views/css` pour les fichiers CSS (si le module doit être compatible avec PrestaShop 1.4, les fichiers CSS doivent être placés à la racine du module, dans un dossier `/css`). On peut en trouver des variantes par thème dans les dossiers modules des différents thèmes installés.
 - dossier `/views/img` pour les fichiers image (si le module doit être compatible avec PrestaShop 1.4, les fichiers image doivent être placés à la racine du module, dans un dossier `/img`).
 - dossier `/views/js` pour les fichiers JavaScript (si le module doit être compatible avec PrestaShop 1.4, les fichiers JS doivent être placés à la racine du module, dans un dossier `/js`).
 - dossier `/views/templates/admin` pour les fichiers utilisés par les contrôleurs d'administration du module.
 - dossier `/views/templates/front` pour les fichiers utilisés par les contrôleurs Front-office du module.
 - dossier `/views/templates/hook` pour les fichiers utilisés par les *hooks* du module.
- Logo du module en 32x32 : `logo.png` (format PNG)
- Fichiers éventuels de traduction : `fr.php`, `en.php`, `es.php`, etc. À partir de la version 1.5, tous ces fichiers peuvent être placés dans le dossier `/translations`.

Nous respecterons donc cette structure pour les nouveaux modules à créer.


4.2 ECRITURE D'UN PREMIER MODULE PRESTASHOP

PrestaShop impose une structure aux différentes classes PHP nécessaires car elles dérivent en général d'une classe de base fournie.

Le mieux est encore de réaliser un premier module de manière totalement guidée. Cet exemple est tiré de la documentation de référence 1.5 ; pour plus de clarté, il est simplifié et corrigé de quelques coquilles.

4.2.1 Créer un premier module

Créons donc un premier module simple ; il nous permettra de mieux apprécier sa structure. Nous l'appellerons "My module".

 Tout d'abord, créez le dossier du module comme sous-dossier du dossier `module` de la boutique. Il doit avoir le même nom que le module, sans aucun espace, composé seulement des caractères alphanumériques, du tiret et du caractère souligné "_", tous en minuscule : `module/mymodule`.

Ce dossier doit contenir le fichier de démarrage, un fichier PHP du même nom que le dossier du module, qui s'occupera de la plupart des traitements : `mymodule.php`.

C'est suffisant pour un module très simple, mais de toute évidence de nombreux autres fichiers et dossiers peuvent être ajoutés.

Par exemple, la partie visible (Front-office) du module est définie dans des fichiers template `.tpl` placés dans un dossier spécifique : `/views/templates/front/`. Les fichiers template peuvent prendre n'importe quel nom. S'il n'y en a qu'un seul, une bonne pratique consiste à lui donner le même nom que le dossier et le fichier principal : `mymodule.tpl`.

Dans le cadre de ce tutoriel, le module sera attaché à la colonne de gauche via un *hook*. De fait, les fichiers templates appelés par ce *hook* devront être placés dans le dossier `/views/templates/hook/` afin de fonctionner.

Pour des raisons de compatibilité, les fichiers template peuvent toujours se trouver à la racine de ce dossier, mais les sous-dossiers de `/views/template` sont désormais à favoriser.

Le script PHP principal, `mymodule.php`, doit commencer avec le test suivant :

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

Le but est de tester ici l'existence d'une constante propre à PrestaShop, et de stopper l'exécution en cas d'absence. Son seul intérêt est donc d'éviter qu'un visiteur peu scrupuleux puisse lancer le script PHP directement.

Ce fichier doit également, et surtout, contenir la classe principale du module. Cette classe doit porter le même nom que le module et son dossier, en écriture '*CamelCase*' : `MyModule`.

De plus, cette classe doit étendre la classe de base `Module`, afin d'hériter de ses méthodes et attributs. Elle peut tout aussi bien étendre l'une des classes dérivées de `Module` pour des besoins spécifiques : `PaymentModule`, `ModuleGridEngine`, `ModuleGraph...`

Code provisoire de mymodule.php

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

class MyModule extends Module
{
}
?>
```

Remplissons maintenant notre classe avec les lignes de démarrage essentielles :

Code de 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';
        $this->author = 'Prénom Nom';
        $this->need_instance = 0;
        $this->ps_versions_compliancy = array('min' => '1.5', 'max' => _PS_VERSION_);

        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');
    }
}
?>
```



Structure minimale d'une classe PHP de module PrestaShop, à reproduire et adapter pour chaque nouveau module.

Examinons chaque ligne de cette première version de la classe `MyModule`.

```
public function __construct()
```

Cette ligne déclare la fonction '*constructeur*' de notre classe et assigne une poignée d'attributs à l'instance de la classe (`$this`).

```
$this->name = 'mymodule';
```

```
$this->tab = 'front_office_features';
```

```
$this->version = '1.0';
```

```
$this->author = 'Prénom Nom';
```

- **attribut 'name'**. Cet attribut sert d'identifiant interne, donc faites en sorte qu'il soit unique, sans caractères spéciaux ni espaces, et gardez-le en minuscule. Dans les faits, la valeur DOIT être le nom du dossier du module.
- **attribut 'tab'**. Cet attribut donne l'identifiant de la section de la liste des modules du Back-office de PrestaShop où devra se trouver ce module. Vous pouvez utiliser un nom existant, tel que `front_office_features` ou `analytics_stats`, ou un identifiant personnalisé. Dans ce dernier cas, une nouvelle section sera créée avec votre identifiant. Nous avons choisi "`front_office_features`" parce que ce module aura surtout un impact sur le front-end.

Voici la liste des attributs "Tab" et leurs sections respectives dans la page "Modules" :

Attribut "tab"	Section du module
administration	Administration
advertising_marketing	Publicité et marketing
analytics_stats	Statistiques & analyses
billing_invoicing	Facturation
checkout	Processus de commande
content_management	Gestion de contenu
emailing	Envoi d'e-mails
export	Export
front_office_features	Fonctionnalités front-office
il8n_localization	Internationalisation et localisation
market_place	Places de marché
merchandizing	Merchandising

Attribut "tab"	Section du module
migration_tools	Outils de migration
mobile	Mobile
others	Autres modules
payments_gateways	Paielements
payment_security	Sécurité des paiements
pricing_promotion	Prix & promotions
quick_bulk_update	Modifications rapides / de masse
search_filter	Recherche et filtres
seo	Référencement - SEO
shipping_logistics	Transporteur & logistique
slideshows	Diaporamas
smart_shopping	Guides d'achat
social_networks	Réseaux sociaux

- **attribut 'version'.** Le numéro de version du module, affiché dans la liste des modules. C'est une chaîne, donc vous pouvez utiliser des variations comme "1.0b", "3.07 beta 3" ou "0.94 (not for production use)".
- **Attribut 'author'.** Le nom de l'auteur est affiché dans la liste des modules de PrestaShop.

Continuons avec le bloc de code suivant :

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

```
$this->ps_versions_compliancy = array('min' => '1.5', 'max' => _PS_VERSION_);
```

Cette section gère les relations entre le module et son environnement (donc, PrestaShop) :

- Le drapeau `need_instance` indique s'il faut charger la classe du module quand celui-ci est affiché dans la page "Modules" du back-office. S'il est à 0, le module n'est pas chargé, et il utilisera donc moins de ressources. Si votre module doit afficher un avertissement dans la page "Modules", alors vous devez passer ce drapeau à 1.
- `ps_version_compliancy` est un nouveau drapeau de PrestaShop 1.5. Il permet d'indiquer clairement les versions de PrestaShop avec lesquelles le module est compatible. On peut indiquer explicitement que ce module ne fonctionnera qu'avec la version 1.5 par exemple, et aucune autre.

Développer des modules - PrestaShop

Nous faisons ensuite appel au constructeur de la classe de base (parent) :

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

Cet appel doit être fait après la création de `$this->name` et avant toute utilisation de la méthode `$this->l()`.

La section suivante met en place les premières chaînes de caractères du module, encapsulées dans la **fonction de traduction PrestaShop 1()**:

```
$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');
```

Chacune de ces lignes assigne un élément :

- un nom pour le module, qui sera affiché dans la liste des modules du Back-office.
- une description pour le module, qui sera affiché dans la liste des modules du Back-office.
- un message demandant à l'administrateur s'il souhaite réellement désinstaller ce module.
- un avertissement que le module n'a pas défini sa variable `MYMODULE_NAME` (ce dernier point est spécifique à notre exemple).

Notre méthode constructeur est maintenant complète, et le module est visible dans la page "Module" du Back-office avec quelques premières informations. Vous pouvez déjà effectuer les traductions nécessaires par le Back-office (il faudra y revenir plusieurs fois, au fur et à mesure des développements...) :

Avant d'installer le module, il reste des méthodes à ajouter dans le code PHP du module.

4.2.2 Les méthodes `install()` et `uninstall()`

Ces deux méthodes vous permettent de contrôler ce qui se passe quand l'administrateur de la boutique installe ou désinstalle le module, pour par exemple vérifier les réglages de PrestaShop ou enregistrer ses propres réglages dans la base de données. Elles doivent faire partie de la classe principale du module (dans notre exemple, la classe `MyModule`).

Code provisoire de la fonction `install()` :

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

Dans cette première incarnation très simple, cette méthode fait le minimum requis : renvoyer la valeur retournée par la méthode `install()` de la classe de base `Module`, qui retourne soit `true` si le module est correctement installé, soit `false` dans le cas contraire. Notez que si nous n'avons pas créé cette méthode, la méthode `install()` de la classe parente `Module` aurait de toute façon été appelée, ce qui aurait donné le même résultat dans notre cas simplifié. Cependant, nous devons mentionner cette méthode, car elle se montrera très utile une fois que nous aurons à mettre en place des tests et actions lors du processus d'installation du module : création de tables SQL, création de variables de configuration, etc.

Il est possible d'ajouter autant des lignes de code à `install()` que nécessaire. Dans l'exemple suivant, nous lançons les tâches suivantes en cours de l'installation :

- vérifier que le module est installé.
- lier le module au `hook` `leftColumn`.
- lier le module au `hook` `header`.
- créer la variable de configuration `MYMODULE_NAME`, en lui donnant la valeur "my friend".

Code de la fonction `install()` :

```
public function install()
{
    /* ajout Version 1.5 pour la gestion multiboutiques */
    if (Shop::isFeatureActive())
        Shop::setContext(Shop::CONTEXT_ALL);
    /* rattacher aux 2 hook et déclarer une var de config */
    return parent::install() &&
        $this->registerHook('leftColumn') &&
        $this->registerHook('header') &&
        Configuration::updateValue('MYMODULE_NAME', 'my friend');
}
```

Si une seule de ces lignes échoue, l'installation n'a pas lieu.

L'objet `Shop` est une nouveauté de PrestaShop 1.5, qui vous permet de gérer la fonctionnalité multi boutique. Sans plonger dans les détails, voici les deux méthodes qui sont utilisées dans ce code d'exemple :

- `Shop::isFeatureActive()` : cette ligne teste simplement si la fonctionnalité multi boutique est activée ou non, et si au moins deux boutiques sont actuellement activées.
- `Shop::setContext(Shop::CONTEXT_ALL)` : cette ligne modifie le contexte pour appliquer les changements qui suivent à toutes les boutiques existantes plus qu'à la seule boutique actuellement utilisée.

De son côté, la méthode de désinstallation supprime simplement la variable de configuration `MYMODULE_NAME`.

```
public function uninstall()  
{  
    return parent::uninstall() && Configuration::deleteByName('MYMODULE_NAME');  
}
```

Si votre module crée effectivement des données SQL, alors il faut impérativement que vous mettiez en place un processus de suppression de ces données dans la méthode `uninstall()`. Cette méthode pourrait alors ressembler à ceci :

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

NB : voir plus loin les informations sur la classe `Db`.

4.2.3 La classe Configuration

Comme vous pouvez le voir, nos trois blocs de code utilisent une nouvelle classe, `Configuration`. Il s'agit d'une classe propre à PrestaShop, conçue pour aider les développeurs à stocker les réglages dans la base de données de PrestaShop sans devoir gérer leurs propres tables. Plus précisément, cet objet utilise la table `ps_configuration`.

Jusqu'ici nous avons utilisé trois de ses méthodes, auxquelles nous allons ajouter une quatrième :

- `Configuration::get('maVariable')` : récupère une valeur spécifique depuis la base de données.
- `Configuration::getMultiple(array('maPremiereVariable', 'maSecondeVariable', 'maTroisiemeVariable'))` : récupère plusieurs valeurs de la base de données, et renvoie un tableau PHP.

- `Configuration::updateValue('maVariable', $value)` : met à jour une variable existant dans la base de données en lui donnant une nouvelle valeur. Si cette variable n'existe pas déjà, elle sera créée pour l'occasion.
- `Configuration::deleteByName('myVariable')` : supprime la variable de la base de données.

Comme vous pouvez le voir, c'est une classe très utile et très facile à utiliser, et vous y aurez certainement recours dans de nombreuses situations. La plupart des modules s'en servent pour leurs propres réglages.

Vous n'êtes pas limité à vos propres variables : PrestaShop stocke également tous ses réglages de configuration dans la table `ps_configuration`. Des centaines de réglages s'y trouvent, et vous pouvez y accéder aussi facilement que vous accédez aux vôtres. Par exemple :

- `Configuration::get('PS_LANG_DEFAULT')` : récupère l'identifiant de la langue par défaut.
- `Configuration::get('PS_TIMEZONE')` : récupère le nom du fuseau horaire.
- `Configuration::get('PS_DISTANCE_UNIT')` : récupère l'unité de distance par défaut ("km" pour les kilomètres, etc.).
- `Configuration::get('PS_SHOP_EMAIL')` : récupère l'adresse e-mail principale de contact.
- `Configuration::get('PS_NB_DAYS_NEW_PRODUCT')` : récupère le nombre de jours durant lesquels un produit récemment ajouté est considéré comme étant "Nouveau" par PrestaShop.

Notez que lorsque vous utilisez `updateValue()`, le contenu du paramètre `$value` peut être ce que vous voulez : une chaîne, un nombre, un tableau PHP sérialisé ou un objet JSON. À partir du moment où votre code peut gérer correctement ce type de données, tout conviendra.



La classe PrestaShop `Configuration` permet de gérer automatiquement les paramètres durables des modules dans la table MySQL de configuration de PrestaShop.

4.2.4 Le fichier d'icônes

Pour parfaire ce premier module, nous pouvons ajouter une icône, qui sera affichée à côté du nom du module dans la liste des modules au Back-office. Assurez-vous de ne pas utiliser un logo déjà utilisé par un module natif.

Le fichier d'icône doit respecter le format suivant :

- placé dans le dossier principal du module.
- Pour fonctionner avec PrestaShop 1.4 :
 - Image en 16*16.
 - Format GIF.
 - Nommé `logo.gif`.

- Pour fonctionner avec PrestaShop 1.5 et suivants :
 - o Image en 32*32.
 - o Format PNG.
 - o Nommé `logo.png`.

4.2.5 Installation du module

Maintenant que toutes les bases sont en place, rechargez la page "Modules" du Back-office. Vous y trouverez votre module (dans la section "Fonctionnalités Front-office") car PrestaShop vient de scanner les dossiers de modules de votre boutique. Installez-le (ou réinitialisez-le s'il est déjà installé) de manière à déclencher l'exécution de sa méthode `install()`.

Lors de l'installation, PrestaShop crée automatiquement un fichier `config.xml` dans le dossier du module, qui stocke les informations de configuration. *Vous ne devez pas le modifier.*

4.2.6 Implémenter des hooks

En tant que tel, notre module ne fait pas grand-chose. Afin d'afficher du contenu sur le Front-office, nous devons ajouter le support pour quelques *hooks*, ce qui se fait dans la méthode `install()`... Comme nous l'avons donné en exemple un peu plus tôt dans ce chapitre, avec la méthode `registerHook()` :

```
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');
}
```

Comme vous pouvez le voir, nous faisons en sorte que le module soit lié aux *hooks* "leftColumn" et "header". En plus de cela, nous allons ajouter du code pour le *hook* "rightColumn".

Le fait d'attacher ce code à un *hook* requiert des méthodes spécifiques pour chaque *hook* :

- `hookDisplayLeftColumn()` : attachera du code à la colonne de gauche – dans notre cas, nous allons récupérer le réglage de module `MYMODULE_NAME` et afficher le résultat du fichier `template mymodule.tpl`, qui se trouve dans le dossier `/views/templates/hook/`.
- `hookDisplayRightColumn()` : fera de même que `hookDisplayLeftColumn()`, mais en l'appliquant à la colonne de droite (NB : le module doit être rattachable à ce *hook* dans sa procédure d'installation).

- `hookDisplayHeader()` : ajoutera un lien vers le fichier CSS du module, `/css/mymodule.css`

Implémentation des *hooks* dans le code du module `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')
        )
    );
    return $this->display(__FILE__, 'mymodule.tpl');
}

public function hookDisplayRightColumn($params)
{ /* dans cet exemple, réalise la même chose que pour la left column */
    return $this->hookDisplayLeftColumn($params);
}

public function hookDisplayHeader()
{
    $this->context->controller->addCSS($this->_path.'css/mymodule.css', 'all');
}
```

En plus de cela, nous utiliserons le *Contexte* pour définir une variable Smarty utilisable dans le template `mymodule.tpl` : la méthode `assign()` ci-dessus permet d'affecter la valeur du paramètre de configuration `MYMODULE_NAME` à la variable `my_module_name`. Dès lors, vous pourrez accéder directement à cette variable dans le template `mymodule.tpl` sous le nom `$my_module_name`

Le hook `header` n'est pas un entête visuel, mais nous permet d'ajouter du code entre les balises `<head>` du code HTML résultant. C'est surtout utile pour inclure les références aux fichiers JavaScript et CSS. Pour ajouter dans l'en-tête un lien vers notre fichier CSS, nous utilisons la méthode `addCSS()`, qui génère la bonne balise HTML `<link>` pointant vers le fichier indiqué en paramètre. Un module dispose aussi d'une méthode `addJS()` permettant d'ajouter les liens adéquats vers des fichiers de scripts JavaScript.

Enregistrez votre fichier, et déjà vous pouvez accrocher le template de votre module au thème, le déplacer et le greffer (voir la page "Position" du menu "Modules" du Back-office).

Dans le formulaire de greffe :

- Trouvez "My module" dans la liste déroulante "Module".
- Choisissez "Left column blocks" dans la liste déroulante "Greffer le module sur".
- Enregistrez vos modifications.

GREFFER UN MODULE

* Module: Mon module

* Greffer le module sur: displayLeftColumn (Left column blocks)

Exceptions: Fichiers pour lesquels le module ne sera pas affiché



NB : Il est impossible d'attacher un module à un hook pour lequel le module ne dispose pas de méthode implémentée.

La page "Positions" devrait alors se recharger, en affichant le message de succès. Faites défiler la page, et vous devriez effectivement voir votre module parmi ceux affichés dans la liste "Left column blocks". Déplacez-le en haut de la liste en faisant un glisser-déposer.

Le module est maintenant attaché à la colonne de gauche... mais sans aucun modèle à afficher, nous sommes loin d'avoir quoi que ce soit d'utile sur la page d'accueil...



L'implémentation d'un hook associé au module nécessite l'appel de la méthode `registerHook()` dans la méthode `install()` du module et l'écriture de la méthode `hookDisplayXXX()` correspondante.

4.2.7 Afficher du contenu

Maintenant que nous avons accès à la colonne de gauche, nous pouvons y afficher ce que nous voulons.

Comme dit plus haut, le contenu à afficher dans le thème doit être défini dans un fichier de template `.tpl`. Nous allons donc créer le fichier `mymodule.tpl`, qui sera passé en paramètre de la méthode `display()` dans le code de notre module, avec la méthode `hookDisplayLeftColumn()`. Lorsque l'on appelle un template depuis un *hook*, PrestaShop cherche ce fichier template dans le dossier de templates `/views/templates/hook/` (dans le dossier du module) ; **vous devez créer vous-même cette arborescence de dossiers.**

Voici notre fichier template, situé dans `/views/templates/hook/mymodule.tpl` :

Code de `mymodule.tpl`

```
<!-- Block mymodule -->
<div id="mymodule_block_left" class="block">
  <h4>Welcome!</h4>
  <div class="block_content">
    <p>{l s='Hello,' mod='mymodule'}
    {if isset($my_module_name) && $my_module_name}
      {$my_module_name}
    {else}
      World
    {/if}
  </div>
</div>
```

Les 2 classes de styles utilisées ici correspondent à l'affichage standard d'un bloc en colonne de gauche (comme pour les catégories ou les déjà vus)


```

</p>
<ul>
    <li><a href="{ $my_module_link}" title="Click this link">Click me!</a></li>
</ul>
</div>
</div>
<!-- /Block mymodule -->

```

C'est tout simplement du code HTML... avec quelques appels Smarty :

- L'appel `{l s='xxx' mod='yyy'}` est une méthode propre à PrestaShop qui enregistre la chaîne dans le panneau de **traduction du module**. Le paramètre `s` est pour la chaîne, tandis que le paramètre `mod` doit contenir l'identifiant du module (dans le cas présent, "mymodule"). Nous n'utilisons cette méthode qu'une fois dans cet exemple pour des raisons de lisibilité, mais dans les faits ***il faudrait l'appliquer à absolument toutes les chaînes pour assurer une bonne traduction***.
- Les instructions Smarty `{if}`, `{else}` et `{/if}` expriment des conditions. Dans notre exemple, nous vérifions que la variable Smarty `$my_module_name` existe bien (grâce à la fonction PHP `isset()`) et qu'elle n'est pas vide. Si tout est comme attendu, nous affichons le contenu de cette variable. Sinon, nous affichons "World", afin d'obtenir le message par défaut "Hello World!"
- La variable `{ $my_module_link }` dans l'attribut HTML `href` : il s'agit d'une variable transmise à Smarty par le code PHP du module et que nous allons bientôt créer ; elle donnera accès au dossier racine de PrestaShop.

 **NB** : c'est une bonne pratique de toujours rédiger le code source en langue anglaise puis de définir les traductions dans le Back-office PrestaShop (y compris pour l'anglais !).

En parallèle, nous allons créer un fichier CSS, et allons l'enregistrer sous le nom **mymodule.css** :

```

div#mymodule_block_left p {
    font-size: 150%;
    font-style: italic;
}

```

Enregistrez le template dans le dossier `/views/templates/hook/` **et le fichier CSS dans le dossier** `/css/`, recherchez la page d'accueil de votre boutique : le contenu de votre template devrait apparaître en haut de la colonne de gauche, juste sous le logo de la boutique (si vous avez effectivement déplacé le module en première position du *hook* "Left Column" lors de sa greffe).

Comme vous pouvez le voir, le thème applique ses propres CSS au template que nous avons créé :

- Notre titre `<h4>` devient l'en-tête du bloc, stylé de la même manière que pour les autres titres.
- Notre bloc `<div class="block_content">` a le même style que les autres blocs sur la page.

Ce n'est pas forcément très joli, mais cela fonctionne !

4.2.8 Désactiver le cache PrestaShop

Si vous avez suivi ce tutoriel à la lettre et que vous ne voyez toujours rien s'afficher dans la colonne de gauche de votre thème, cela peut être dû au mécanisme de cache de PrestaShop, qui conserve les versions précédentes de vos templates et continue de vous les servir telles quelles. C'est pourquoi vous voyez toujours la version originale du thème, sans vos changements.

Smarty met en cache une version compilée de vos pages, pour des questions de performance. C'est extrêmement utile pour les sites en production, mais gênant pour les sites en phase de test, où vous êtes amené à recharger très régulièrement les pages afin de voir l'impact de vos modifications.

Lorsque vous modifiez ou déboguez des éléments du thème d'un site de test, vous devriez toujours désactiver le cache, afin de forcer Smarty à recompiler les templates à chaque chargement. Veillez aussi à désactiver le cache de votre navigateur.

Pour ce faire, rendez-vous dans le menu "Paramètres avancés", ouvrez la page "Performances", et dans la section "Smarty" choisissez les options suivantes :



Les affichages nécessaires aux modules se font par le biais de templates Smarty. Le module fait le lien avec le template grâce à sa méthode `display()` et il transmet les données à fusionner grâce à la méthode `assign()` de l'objet Smarty de son contexte. Le rendu final est assuré grâce aux feuilles de styles CSS.

4.2.9 Intégrer un template à un thème ; les contrôleurs

Le lien que le module affiche sur le Front-office ne mène nulle part pour le moment. Créons le fichier `display.php` qu'il cible, avec un contenu minimal, et mettons-le à la racine du dossier du module.

Code provisoire du contrôleur `display.php`

```
Welcome to this page!
```

Si vous cliquez sur le lien "Click me!", la page résultante ne sera que du texte brut, sans rien qui rappelle le thème de la boutique. Nous aimerions pourtant que ce texte soit affiché au sein du thème, donc faisons quelques modifications.

Comme l'on peut s'y attendre, nous devons créer un fichier template afin d'utiliser la mise en forme du thème. Créons donc `display.tpl`, qui contiendra notre simple ligne "Welcome to my shop!", et sera appelé par le fichier `display.php`. Nous allons par ailleurs réécrire ce fichier `display.php` pour en faire un contrôleur front-end, afin d'intégrer correctement notre template très basique au milieu des éléments du thème : en-tête, pied de page, colonnes, etc.

Voici nos deux fichiers :

Code du contrôleur `display.php`

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

Code du template `display.tpl`

```
Welcome to my shop!
```

Explorons `display.php`, notre premier contrôleur front-end PrestaShop, à stocker dans le sous-dossier `/controllers/front` du dossier du module :

- Un contrôleur permet de générer une page complète de la boutique PrestaShop.
- Un contrôleur front-end doit être une classe qui étend la classe `ModuleFrontController`.
- Ce contrôleur doit avoir une méthode : `initContent()`, qui doit appeler la méthode `initContent()` de la classe parente...
- ...qui appelle ensuite la méthode `setTemplate()` en précisant quel fichier template utiliser (`display.tpl`).

La méthode `setTemplate()` s'occupera d'intégrer notre template ne contenant qu'une ligne, et d'en faire une page complète, avec entête, pied de page et colonnes.

Jusqu'à PrestaShop 1.4, les développeurs qui souhaitaient intégrer un fichier template dans le thème du site devaient utiliser des appels `include()` pour chaque portion de la page. Maintenant, tout est automatisé grâce à `setTemplate()`.

Enregistrez les deux fichiers dans leurs dossiers respectifs, et rechargez la page d'accueil, puis cliquez sur le lien "Click me!", et vous pouvez voir que votre lien fonctionne comme attendu. Avec juste quelques lignes, le résultat final est déjà beaucoup mieux, la ligne "Welcome..." étant maintenant placée au milieu de la page d'accueil.



Bien entendu, pour un affichage en Français, il est nécessaire de passer par une étape de traduction dans le Back-office !

Ce n'est là qu'une première étape, mais elle vous donne une idée de ce qu'il est possible de faire si vous suivez les règles de templates.



Un module peut appeler un contrôleur de manière à enchaîner sur une nouvelle page complète de la boutique. Il peut s'agir d'un contrôleur PrestaShop existant ou d'un nouveau contrôleur développé par dérivation d'une classe de base (`Controller`, `FrontController`, `ModuleFrontController`...).

4.2.10 Bien utiliser Smarty

Smarty est le moteur de template utilisé par le système de thème de PrestaShop. Il parcourt les fichiers `.tpl`, à la recherche d'éléments dynamiques à remplacer par leurs équivalents contextuels, pour envoyer au navigateur le résultat personnalisé sous forme de code HTML pur. Ces éléments dynamiques sont marqués par des accolades : `{ ... }`. Les développeurs peuvent créer leurs variables.

Par exemple, nous pouvons créer la variable par PHP `$my_module_message` directement dans la méthode `hookDisplayLeftColumn()`, et la faire s'afficher par notre fichier template.

Code `mymodule.php` modifié

```
public function hookDisplayLeftColumn($params)
{
    $this->context->smarty->assign(
        array(
            .....
```

```

        'my_module_message' => $this->l('This is a simple text message')
        // Ne pas oublier de mettre la chaîne dans la méthode de traduction l()
    )
};

return $this->display(__FILE__, 'mymodule.tpl');
}

```

Partant de là, nous pouvons demander à Smarty d'afficher le contenu de cette variable dans notre fichier TPL.

Code du template `mymodule.tpl` modifié

```

<div class="block_content">
    <p>{l s='Hello' mod='mymodule'}
        {if isset($my_module_name) && $my_module_name}
            {$my_module_name}
        {else}
            {l s='World' mod='mymodule'}
        {/if}
    !
</p>
    <p>{$my_module_message}
</p>

```

Après avoir saisi toutes les traductions nécessaires à ce module, on obtient :



PrestaShop ajoute ses propres variables aux variables standard Smarty. Par exemple, `{hook_left_column}` sera remplacé par le contenu de la colonne de gauche, ce qui signifie *le contenu de tous les modules* qui ont été attachés au *hook* de la colonne de gauche.

Toutes les variables de Smarty sont globales. Vous devriez donc faire attention à ce que vos propres noms de variables n'utilisent pas le nom d'une variable Smarty existante, afin d'éviter qu'elle ne la remplace. Une bonne pratique consiste à éviter les noms trop simples, comme

Développer des modules - PrestaShop

`{products}`, et de les préfixer avec le nom de votre module, voire même avec votre propre nom ou vos initiales : `{henryb_mymodule_products}`.

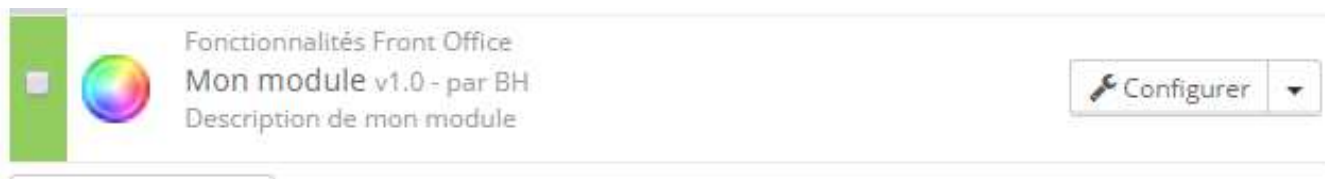
Voici une liste de variables Smarty communes à toutes les pages :

Fichier / dossier	Description
img_ps_dir	Adresse du dossier image de PrestaShop.
img_cat_dir	Adresse du dossier des images des catégories.
img_lang_dir	Adresse du dossier des images des langues.
img_prod_dir	Adresse du dossier des images des produits.
img_manu_dir	Adresse du dossier des images des fabricants.
img_sup_dir	Adresse du dossier des images des fournisseurs.
img_ship_dir	Adresse du dossier des images des transporteurs.
img_dir	Adresse du dossier des images du thème.
css_dir	Adresse du dossier des fichiers CSS du thème.
js_dir	Adresse du dossier des fichiers JavaScript du thème.
tpl_dir	Adresse du dossier du thème actuel.
modules_dir	Adresse du dossier des modules.
mail_dir	Adresse du dossier des modèles d'e-mails.
pic_dir	Adresse du dossier des images mises en ligne.
lang_iso	Code ISO de la langue actuelle.
come_from	Adresse d'origine du visiteur.
shop_name	Nom de la boutique.
cart_qties	Nombre de produits dans le panier.
cart	Le panier.
currencies	Les différentes devises disponibles.
id_currency_cookie	Identifiant de la devise actuelle.
currency	Objet Currency (la devise actuellement utilisée).
cookie	Cookie de l'utilisateur.
languages	Les différentes langues disponibles.
logged	Indique si le visiteur est connecté à un compte utilisateur.

Fichier / dossier	Description
page_name	Nom de la page.
customerName	Nom du client (s'il est connecté).
priceDisplay	Méthode d'affichage du prix (avec ou sans les taxes...).
roundMode	Méthode d'arrondi utilisée.
use_taxes	Indique si les taxes sont activées ou non.

4.2.11 Ajouter une page de configuration au Back-office

Votre module peut avoir un lien "Configurer" dans la liste des modules du Back-office, et donc permettre à l'utilisateur administrateur de modifier quelques paramètres (c'est bien ce qui manque au module standard des devises par exemple !). Ce lien "Configurer" apparaît avec l'addition de la méthode `getContent()` dans la classe principale du module. C'est une méthode PrestaShop standard : sa seule présence envoie au Back-office un message indiquant qu'il y a une page de configuration, et donc que le lien "Configurer" devrait être affiché.



Mais le fait d'avoir une méthode `getContent()` publique dans la classe `MyModule` ne fait qu'afficher ce lien ; elle ne crée pas la page de configuration elle-même. Nous allons voir comment la créer, afin d'y placer un formulaire permettant de modifier la valeur de la variable de configuration `MYMODULE_NAME` qui est stockée dans la table PrestaShop `ps_configuration`.

4.2.12 La méthode `getContent()`

Tout d'abord, voici le code complet pour la méthode `getContent()` à ajouter au module `mymodule.php` :

```
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();
}

```

La méthode `getContent()` est la première à être appelée quand la page de configuration est chargée. Ainsi, nous l'utilisons tout d'abord pour **mettre à jour toute valeur qui aurait pu être soumise par le biais du formulaire** contenu dans la page :

Voici une explication ligne à ligne :

- `Tools::isSubmit()` est une méthode statique propre à PrestaShop, qui vérifie si le formulaire indiqué a bien été saisi et validé (PrestaShop utilise ici le principe de page dynamiques *'réentrante'*, qui se rappelle elle-même). Si le formulaire de configuration a bien été validé, le bloc `if()` entier est sauté et PrestaShop ne lit que la dernière ligne, qui affiche le formulaire de configuration avec les valeurs actuelles, tel que généré par la méthode `displayForm()`.
- `Tools::getValue()` est une méthode statique propre à PrestaShop, qui **récupère le contenu du tableau POST ou GET** pour en tirer la valeur de la variable indiquée. Ici, nous récupérons la valeur de `MYMODULE_NAME` du formulaire, transformons sa valeur en une chaîne à l'aide de la fonction PHP `strval()`, et la stockons dans la variable PHP `$my_module_name`.
- Nous vérifions ensuite l'existence de véritable contenu dans `$my_module_name`, notamment en utilisant `Validate::isGenericName()`. La classe `Validate` contient de nombreuses méthodes statiques de validation, parmi lesquelles se trouve `isGenericName()`, une méthode qui vous aide à vérifier qu'une chaîne est bien un nom de variable PrestaShop valide (c'est-à-dire qu'elle ne contient pas de caractères spéciaux, pour simplifier).
- Si ces tests échouent, le formulaire de configuration s'ouvre avec un message d'erreur, indiquant que la validation du formulaire a échoué.
La variable `$output`, qui contient le rendu final du code HTML qui compose la page de configuration, affiche en premier lieu un message d'erreur, créé éventuellement par la méthode `displayError()`. Cette méthode renvoie le code HTML nécessaire à nos besoins, et comme le message d'erreur est le premier inséré dans `$output`, la page commencera par ce message.
- Si ces tests réussissent, cela signifie que nous pouvons stocker la valeur dans la base de données.
Comme nous l'avons vu plus tôt dans ce tutoriel, la classe `Configuration` a exactement la méthode statique dont nous avons besoin : `updateValue()` stockera la nouvelle valeur pour le paramètre `MYMODULE_NAME` dans la table de configuration.
Nous ajoutons une notification à l'utilisateur, indiquant que la valeur a bien été mise à jour : nous utilisons la méthode `displayConfirmation()` de PrestaShop, qui place le message en premier dans la variable `$output` – et donc, en haut de la page.

- Enfin, nous utilisons la méthode `displayForm()` (que nous allons créer et expliquer dans la section suivante) pour ajouter du contenu à `$output` (que le formulaire ait été validé ou non), et renvoyons ce contenu dans la page.

Notez que nous aurions pu inclure le code de `displayForm()` directement dans `getContent()`, mais que nous avons choisi de séparer les deux pour des questions de lisibilité et de séparation des intérêts (donc pour faciliter et sécuriser la maintenance).

Ce type de code de validation de formulaire n'est pas une nouveauté pour les développeurs PHP, mais il utilise certaines des méthodes PrestaShop que vous utiliserez le plus.

4.2.13 Afficher le formulaire

Le formulaire de configuration lui-même est affiché par la méthode `displayForm()`. Voici son code, que nous expliquerons ensuite.

```
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 page
    $helper->submit_action = 'submit'.$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);
}

```

À première vue, il s'agit surtout d'un énorme bloc de code pour ne changer qu'une seule valeur. Mais ce bloc utilise plusieurs des méthodes de PrestaShop qui vous aideront à construire vos formulaires, notamment un objet `HelperForm`.

Parcourons cette méthode `displayForm()`:

- À l'aide de la méthode `Configuration::get()`, nous récupérons la valeur de la langue par défaut ("PS_LANG_DEFAULT"). Pour des questions de sécurité, nous faisons en sorte que la variable soit un entier en utilisant le 'cast' (int).

- Afin de préparer la génération du formulaire, nous construisons un tableau des titres, champs et autres spécificités de formulaire.

Pour ce faire, nous créons la variable `$fields_form`, qui contient un tableau multidimensionnel. Chaque tableau qu'il contient présente une description détaillée des balises de saisie que le formulaire contient. À partir de cette variable `$fields_form`, PrestaShop pourra générer le formulaire HTML tel que décrit.

Dans notre exemple, nous définissons trois balises HTML (`<legend>`, `<input>` et `<submit>`) et leurs attributs à l'aide de tableau. Le format est simple à comprendre : les tableaux de la légende et du bouton contiennent les attributs de chaque balise, tandis que le champ texte contient autant de champs `<input>` que nécessaire, chacun étant à son tour décrit en tableau contenant les attributs nécessaires.

Par exemple :

```
'input' => array(
    array(
        'type' => 'text',
        'label' => $this->l('Configuration value'),
        'name' => 'MYMODULE_NAME',
        'size' => 20,
        'required' => true
    )
)
```

génère les balises HTML suivantes :

```
<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>
```

- Comme vous pouvez le voir, PrestaShop utilise tout cela intelligemment, et génère tout le code HTML nécessaire à l'obtention d'un formulaire utile. Notez que la valeur du tableau principal est utilisée plus loin dans le code, au sein du code permettant de générer le formulaire.
- Nous instancions ensuite un objet `HelperForm`. Cette section du code est expliquée dans la prochaine section de ce chapitre.
- Une fois que les réglages de `HelperForm` sont en place, nous générons le formulaire à partir de la variable `$fields_form`.

4.2.14 Utiliser HelperForm

`HelperForm` est l'un des outils d'aide ajoutés à PrestaShop, en même temps que `HelperOptions`, `HelperList`, `HelperView` et `HelperHelpAccess`. Ces outils permettent de générer des éléments HTML standards pour le Back-office, ainsi que les pages de configuration des modules.

Vous pouvez obtenir plus d'information sur les classes Helper dans le chapitre "Helper" du guide du développeur, qui dispose d'une page dédiée à `HelperForm` (<http://doc.prestashop.com/display/PS16/Using+the+Helper+classes>)

Développer des modules - PrestaShop

Pour rappel, voici notre code :

```
$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 page
$helper->submit_action = 'submit'.$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');

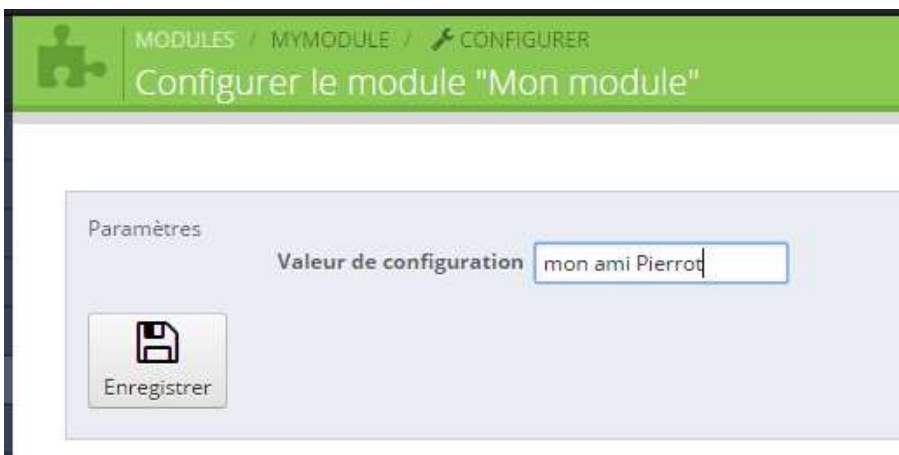
return $helper->generateForm($fields_form);
```

Notre exemple utilise plusieurs attributs de `HelperForm` : ils doivent être mis en place avant que nous ne lancions la génération du formulaire à partir de la variable `$fields_form` :


- `$helper->module` : définit l'instance du module qui utilisera les données du formulaire.
- `$helper->name_controller` : définit le nom du module.
- `$helper->token` : définit un jeton (*token*) unique et propre au module. La méthode statique `getAdminTokenLite()` en génère un pour vous.
- `$helper->default_form_language` : récupère la langue par défaut de la boutique.
- `$helper->title` : définit le titre du formulaire.
- `$helper->show_toolbar` : définit un booléen indiquant si la barre d'outils est affichée ou non.
- `$helper->toolbar_scroll` : définit un booléen indiquant si la barre d'outils est visible lors du défilement ou non.
- `$helper->submit_action` : définit l'attribut d'action de la balise `<submit>` du formulaire.
- `$helper->toolbar_btn` : définit les boutons qui seront affichés dans la barre d'outils. Dans notre exemple, les boutons "Save" et "Back".
- `$helper->fields_value[]` : c'est ici que nous pouvons définir la valeur de la balise de saisie HTML nommée.

Enfin, après avoir mis tout ceci en place, nous pouvons appeler la méthode `generateForm()`, qui se chargera de combiner toutes ces informations et, comme son nom l'indique, de générer le formulaire que l'utilisateur emploiera pour modifier les réglages du module dans le Back-office.

Voici le rendu du formulaire tel qu'actuellement écrit – comme vous pouvez le voir vous-même en cliquant sur le lien "Configurer" du module dans le Back-office :



Vous pouvez maintenant saisir un texte dans le Back-office de PrestaShop et observer le résultat dans la colonne de gauche des pages du Front-office.

 Pour rendre un module paramétrable dans le Back-office, il suffit de lui ajouter la méthode `getContent()` et de dessiner le form HTML correspondant. La classe `HelperForm` aide grandement à la génération du code HTML nécessaire.

4.3 BILAN DE L'EXERCICE

Cet exercice guidé a permis de placer les grands principes de l'écriture de modules dans PrestaShop :

- Un module est matérialisé par une classe de code PHP qui dérive de la classe de base `Module`.
- Un module doit contenir un constructeur (méthode `__construct()`) qui définit ses paramètres généraux et appelle systématiquement le constructeur de la classe de base (`parent::__construct()` ;).
- Un module doit disposer des méthodes `install()` et `uninstall()` de manière à pouvoir être géré dans le Back-office. La méthode `install()` déclare à quels *hooks* ce module peut être greffé (méthode `registerHook()`).
- Un module doit être accompagné d'une icône normalisée destinée au Back-office.
- Pour qu'un module affiche quelque chose sur une page de la boutique, il doit être associé à un ou plusieurs *hooks* et doit contenir les méthodes d'affichage correspondantes de type `hookDisplayXXX()`.
- Ces méthodes transmettent les données au moteur Smarty grâce à la méthode `assign()` de l'objet Smarty du contexte du module :
`$this->context->smarty->assign(...)`
- ... et font le lien avec le template Smarty grâce à la méthode `display()` du module.
- Les liens éventuels avec les feuilles de styles CSS et scripts JavaScript doivent être réalisés par la méthode `hookDisplayHeader()` du module grâce aux méthodes `addCSS()` et `addJS()` du contrôleur qui pilote la page :
`$this->context->controller->addCSS(...)`.
- Un module peut faire le lien vers une nouvelle page qui doit alors être pilotée par un contrôleur, classe PHP dérivée de la classe `ModuleFrontController` ; cette classe doit contenir le lien vers son template principal grâce à sa méthode `setTemplate()` appelée au sein de son autre méthode `initContent()`.
- Un template de module est constitué du code nécessaire à la production de l'extrait de page HTML (ensemble d'éléments `<div>` imbriqués) et des instructions Smarty nécessaires à la fusion des variables transmises par le module au moteur Smarty.
- Il faut privilégier la traduction automatique des messages en utilisant (systématiquement) l'instruction Smarty `{l s='...' mod='...'}`.
- Les fichiers CSS du module ne contiennent que les règles CSS correspondant aux affichages spécifiques. Dans le code HTML, il faut privilégier l'appel des classes de styles standards utilisées par PrestaShop de manière à assurer une bonne harmonie avec le reste du thème de la boutique.
- Pour offrir un paramétrage du module dans le Back-office, le module doit contenir la méthode `getContent()` qui définit les modalités de récupération et mise à jour de ces paramètres. Les paramètres peuvent être aisément gérés par la classe `PrestaShopConfiguration` et ses méthodes statiques `Configuration::get()` et `Configuration::updateValue()`.
- PrestaShop propose un générateur de formulaires HTML avec la classe `HelperForm` qui facilite le design et la bonne harmonie des formulaires de saisie de données.

4.4 LA CLASSE DB ET L'ACCES AUX BASES DE DONNEES

Au-delà de la classe `Configuration` qui permet de stocker des valeurs dans la table de paramètres de PrestaShop, le développement d'un module peut nécessiter l'accès aux tables MySQL de PrestaShop ou même à des tables spécifiques au module.

Quand une nouvelle table est nécessaire, il est préférable de stocker cette table dans une base de données MySQL distincte de la base de la boutique, afin d'assurer les mises à jour futures.

Pour coder les appels à la base de données, le développeur peut utiliser :

- Les instructions `mysqli_xxx()` et écrire toute la logique nécessaire ;
- Le framework complémentaire PDO qui permet de s'affranchir de la syntaxe SQL ;
- La classe PrestaShop `Db` et ses dérivées spécialisées pour MySQLi, pour PDO... qui offrent ensemble encore plus de services que PDO pour tous les besoins courants.

Lors de l'installation de la boutique PrestaShop, l'assistant a paramétré le mode d'accès à la base de données de la boutique (type de SGBD, informations de connexion, nom de la base de données...) ; ces paramètres sont enregistrés dans un fichier de configuration lu dès le démarrage de la boutique et sont utilisés par les classes PrestaShop d'accès aux données. Pour le développeur, il n'est même plus nécessaire de préciser les modalités d'accès au SGBD : il suffit d'invoquer les méthodes statiques de la classe `Db` :

- `Db::getInstance()->insert(...)` pour insérer des données dans une table
- `Db::getInstance()->update(...)` pour modifier des données dans une table
- `Db::getInstance()->delete(...)` pour supprimer des enregistrements dans une table
- `Db::getInstance()->executeS(...)` pour sélectionner des données depuis les tables de la base
- `Db::getInstance()->getRow(...)` pour sélectionner la première ligne uniquement du résultat d'une requête de sélection
- `Db::getInstance()->getValue(...)` pour sélectionner uniquement la première valeur de la première ligne du résultat d'une requête de sélection
- `Db::getInstance()->AffectedRows(...)` pour connaître le nombre d'enregistrements affectés par la dernière méthode de mise à jour invoquée

Et encore quelques autres...

L'usage de ces méthodes de la classe `Db` est assez intuitif et plutôt bien documenté. Pour en savoir plus, étudier la documentation de référence :

<https://www.prestashop.com/blog/fr/les-bonnes-pratiques-de-la-classe-db-sur-prestashop-1-5/>

Ce support d'apprentissage vous a permis de comprendre le fonctionnement et la structure d'un module PrestaShop ainsi que les moyens de mémoriser durablement des données au sein d'un module.

Il ne reste maintenant qu'à acquérir un peu de pratique...

CREDITS

ŒUVRE COLLECTIVE DE L'AFPA

Sous le pilotage de la DIIP et du centre d'ingénierie sectoriel Tertiaire-Services

Equipe de conception (IF, formateur, mediatiseur)

Benoit Hézard - formateur

Chantal Perrachon – Ingénieure de formation

Date de mise à jour : 29/02/16

Reproduction interdite

Article L 122-4 du code de la propriété intellectuelle.

« Toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droits ou ayants cause est illicite. Il en est de même pour la traduction, l'adaptation ou la reproduction par un art ou un procédé quelconque. »

Développer des modules - PrestaShop

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »