Créer un module PrestaShop

Table des matières

- Créer un module PrestaShop
  - Qu'est-ce qu'un module PrestaShop ?
    - Les principes techniques derrière un module
    - Principes opératoires du module
    - La structure des fichiers de PrestaShop
    - Dossier du module
    - À propos du cache
  - Organiser votre module
    - Créer un premier module
    - Les méthodes install() et uninstall()
    - L'objet Configuration
    - L'objet Shop
    - Le fichier d'icônes
    - Apparence dans la page des modules
  - À propos du fichier config.xml
  - Implémenter des hooks
    - Afficher du contenu
    - Intégrer un template à un thème
  - Utiliser Smarty
  - Ajouter une page de configuration
    - La méthode getContent()
    - Afficher le formulaire
  - Traduction du module
  - En cas de problème
    - Forums officiels de PrestaShop
    - Notre bug-tracker
    - Sites officiels de PrestaShop

Créer un module PrestaShop

Qu'est-ce qu'un module PrestaShop ?

L'extension se base sur les modules, qui sont des petits programmes qui exploitent les fonctionnalités de PrestaShop et les modifient ou les étendent afin de rendre PrestaShop plus facile à utiliser, plus complet ou plus personnalisé.

Les principes techniques derrière un module

Un module PrestaShop comprend :

- Un dossier racine, portant le même nom que le module, qui contient tous les fichiers du module et se trouve dans le dossier `/modules` de PrestaShop.
- Un fichier PHP principal, portant le nom du module, placé à la racine de ce dossier. Ce fichier PHP doit avoir le même nom que le dossier du module.
- Deux fichiers icônes, afin de représenter le module dans le back-office :
  - Un fichier pour PrestaShop 1.4 (si besoin) : `logo.gif`, 16*16 pixels.
  - Un fichier pour PrestaShop 1.5 : `logo.png`, 32*32 pixels.
- Facultatif : un fichier template `.tpl`, contenant le thème du module.
- Facultatif : un fichier de langue, si le module ou son thème affichent des chaînes de texte (qui doivent, de fait, être traduites).
- Facultatif : dans un dossier `/themes/modules`, un dossier avec le même nom que le module, contenant les fichiers `.tpl` et de traduction si nécessaire. Ce dernier dossier est essentiel pendant les modifications d'un module existant, afin que vous puissiez l'adapter sans jamais toucher à ses fichiers originaux. Notamment, il vous permet de gérer l'affichage d'un module de différentes manières en fonction du thème actuel.

Voyons un exemple avec le module PrestaShop **blockuserinfo** :

Tout module PrestaShop, une fois installé sur une boutique en ligne, peut interagir avec des "hooks", ou "point d'accroche". Un hook vous permet "d'accrocher" du code à votre Vue actuelle au moment de l'analyse du code (ex. : lors de l'affichage du panier ou de la fiche produit, lors de l'affichage du stock actuel...). Plus spécifiquement, un hook est un raccourci vers les différentes méthodes disponibles au sein de l'objet Module, telles qu'assignées à ce hook.

Principes opératoires du module

Les modules sont la voie idéale pour laisser votre talent et votre imagination de développeur s'exprimer, tant les possibilités créatives sont nombreuses.

Ils peuvent afficher une grande variété de contenu (graphiques, textes, animations...), lancer de nombreuses tâches (mise à jour par lot, import, export, etc.), s'interfacer avec d'autres outils.

Les modules peuvent être aussi configurables que nécessaire ; plus ils sont configurables, plus ils seront faciles à utiliser, et seront donc en mesure de correspondre à un plus grand nombre d'utilisateurs.

L'un des principaux intérêts des modules et d'ajouter des fonctionnalités à PrestaShop sans devoir modifier ses fichiers coeur, ce qui simplifie grandement la mise à jour du logiciel sans devoir recopier toutes ces modifications. De fait, vous devriez faire en sorte de ne jamais modifier les fichiers de PrestaShop lorsque vous concevez un module, même si cela peut se révéler compliqué dans certaines situations.

La structure des fichiers de PrestaShop

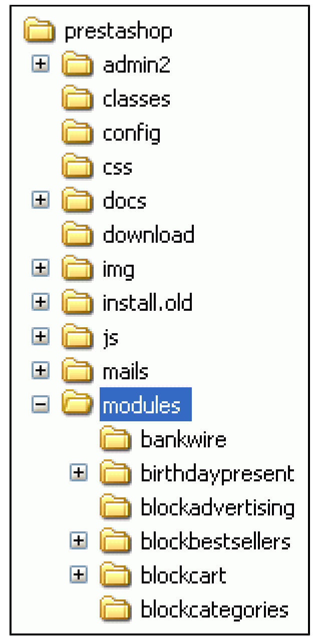
Les développeurs de PrestaShop ont fait de leur mieux pour séparer les différentes parties du logiciel de manière claire et intuitive.

Voici comment les dossiers sont organisés :

- `/admin` : contient tous les fichiers de PrestaShop relatifs au back-office. Si vous cherchez à accéder à ce dossier à l'aide de votre navigateur, il vous sera demandé de vous authentifier, pour des raisons de sécurité. **Important** : faites en sorte que ce dossier reste protégé par des fichiers `.htaccess` et `.htpasswd`.
- `/cache`: contient des dossiers temporaires qui sont générés et réutilisés afin d'alléger la charge du serveur.
- `/classes` : contient tous les fichiers relatifs au modèle Objet de PrestaShop. Chaque fichier représente (et contient) une classe PHP, et ses méthodes et propriétés.
- `/config` : contient tous les fichiers de configuration de PrestaShop. Ne modifiez **jamais** ces fichiers, sauf si on vous le demande expressément, car ils sont gérés directement par l'installateur et le back-office de PrestaShop.
- `/controllers` : contient tous les fichiers relatifs aux contrôleurs de PrestaShop (dans le cadre Modèle-Vue-Contrôleur (ou MVC), l'architecture logicielle sur laquelle repose PrestaShop. Chaque fichier contrôle une partie précise de PrestaShop.
- `/css` : contient tous les fichiers CSS qui ne sont pas liés aux thèmes ; de fait, ils sont la plupart du temps utilisés par le back-office de PrestaShop.

- `/docs` : contient un peu de documentation. **Note** : vous devriez les effacer dans un environnement de production.
- `/download` : contient tous vos produits numériques, pouvant être téléchargés : fichiers PDFs, MP3s, etc.
- `/img` : contient toutes les images par défaut de PrestaShop, c'est à dire celles qui n'appartiennent pas au thème. C'est ici que vous trouverez les sous-dossiers des images pour les catégories de produits (`/c`, celles des produits (sous-dossier `/p`) et celle pour le back-office lui-même (sous-dossier `/admin`).
- `/install` : contient tous les fichiers relatifs à l'installateur de PrestaShop. Vous devriez l'effacer après installation, afin d'améliorer la sécurité.
- `/js` : contient tous les fichiers JavaScript qui ne sont pas liés aux thèmes. La plupart appartiennent au back-office. C'est également ici que vous trouverez le framework jQuery.
- `/localization` : contient tous les fichiers de localisation de PrestaShop – c'est à dire, les fichiers qui contiennent des informations locales, comme la monnaie, la langue, les règles de taxes et les groupes de règles de taxes, les états et autres unités utilisées dans le pays choisi (par exemple, le volume en litre, le poids en kilogramme, etc.).
- `/log` : contient les fichiers de log générés par PrestaShop lors de diverses étapes, par exemple pendant le processus d'installation.
- `/mails` : contient tous les fichiers HTML et textes relatifs aux e-mails envoyés par PrestaShop. Chaque langue dispose de son propre dossier, où vous pouvez modifier manuellement ce que vous souhaitez.
- `/modules` : contient tous les modules de PrestaShop, chacun dans son propre dossier. Si vous souhaitez enlever définitivement un module, commencez par le désinstaller depuis le back-office, puis effacez son dossier à la main.
- `/override` : il s'agit d'un dossier particulier, apparu à partir de la version 1.4 de PrestaShop. En utilisant la convention de nommage de dossier et fichiers de PrestaShop, il devient possible de créer des fichiers qui supplantent les classes et contrôleurs par défaut de PrestaShop. Cela vous permet de modifier le comportement fondamental de PrestaShop sans toucher aux fichiers originaux, ce qui les garde intacts en prévision de la prochaine mise à jour.
- `/themes` : contient tous les thèmes actuellement installés, chacun dans son propre dossier.
- `/tools` : contient les outils externes qui ont été intégré à PrestaShop. Par exemple, c'est ici que vous trouverez Smarty (moteur de thème), FPDF (générateur de fichiers PDF), Swift (expéditeur d'e-mails), PEAR XML Parser (outil PHP).
- `/translations` : contient un sous-dossier pour chaque langue. Cependant, si vous souhaitez modifier les traductions, vous devez utiliser l'outil interne de PrestaShop, et **surtout ne pas** les modifier directement dans ce dossier.
- `/upload` : contient les fichiers qui ont été mis en ligne par les clients pour personnaliser vos produits (par exemple, une image à imprimer sur un mug).
- `/webservice` : contient les fichiers qui permettent aux applications tierces de se connecter à PrestaShop, au travers de son API.

Dossier du module



Les modules de PrestaShop sont le plus souvent situés dans le dossier /modules, qui se trouve à la racine du dossier principal de PrestaShop. C'est le cas tant pour les modules par défaut (fournis avec PrestaShop) que pour les modules tiers qui seront installés par la suite.
Les modules peuvent également être inclus dans un thème s'ils lui sont vraiment spécifiques. Dans ce cas, ils se trouveraient dans le sous-dossier `/modules` du dossier de ce thème, et seraient donc localisés dans le chemin `/themes/nom-du-theme/modules`.

Chaque module dispose de son propre sous-dossier au sein du dossier /modules: `/bankwire`, `/birthdaypresent`, etc.

À propos du cache

Le fichier `/cache/Class_index.php` contient le lien entre la classe et le fichier de déclaration. Il peut être effacé sans crainte.

Le dossier `/cache/xml` contient la liste de tous les modules de base.

Lorsque le front-end de la boutique ne correspond pas à vos modifications, et que le fait de vider le cache du navigateur n'améliore pas les choses, vous pouvez essayer de vider ces dossiers :

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

Organiser votre module

Un module peut être composé de nombreux fichiers, tous stockés dans le dossier qui porte le même nom que le module, ce dossier étant à son tour situé dans le dossier `/modules` à la racine du dossier de PrestaShop.

Les fichiers et dossiers par défaut d'un module PrestaShop 1.5 sont :

- Le fichier de démarrage : `nom_du_module.php`
- Le fichier de configuration du cache : `config.xml`
- Les contrôleurs spécifiques au module, stockés dans le dossier `/controllers`
- Les classes de surcharge, stockées dans le dossier `/override` (installation et désinstallation automatique par copie ou par fusion du code)
- Les fichiers de vue (templates, JavaScript, CSS, etc.). Ils peuvent être placés dans ces dossiers 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`.
  - 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 admin du module.
  - dossier `/views/templates/front` pour les fichiers utilisés par les contrôleurs front du module.
  - dossier `/views/templates/hook` pour les fichiers utilisés par les hooks du module.

  À partir de la v1.5, les fichiers JavaScript et CSS peuvent être placés dans ces sous-dossiers :
  - dossier `/views/css` pour les fichiers CSS.
  - dossier `/views/js` pour les fichiers JavaScript.v

    Vous pouvez placer vos fichiers CSS, JavaScript et images dans n'importe lequel des dossiers autorisés. Efforcez-vous surtout d'être cohérent, et en cas d'overload, de toujours utiliser le même chemin que le code original.
- Logo du module en 16x16 : `logo.jpg` (format JPG ou GIF)
- Logo du module en 32x32 : `logo.png` (format PNG)
- Fichiers de traduction : `fr.php`, `en.php`, `es.php`, etc. À partir de la v1.5, tous ces fichiers peuvent être placés dans le dossier `/translations`.

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. Il doit avoir le même nom que le module, avec aucun espace, seulement des caractères alphanumériques, le tiret et le caractère souligné "_", tous en minuscule : `/mymodule`.

Ce dossier doit contenir le fichier de démarrage ("bootstrap file"), 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 `.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.

Dans PrestaShop 1.4, les fichiers templates du module pouvaient être placés à la racine du dossier du module.

Pour des raisons de compatibilité, les fichiers template peuvent toujours se trouver à la racine de ce dossier avec PrestaShop, mais les sous-dossiers de `/views/templates` sont désormais à favoriser. Si vous souhaitez que votre module fonctionne également pour PrestaShop 1.4, vous devriez conserver ces fichiers à la racine du dossier.

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

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

Nous testons ici l'existence d'une constante propre à PrestaShop, et stopper l'exécution en cas d'absence. Son seul intérêt est donc d'éviter qu'un visiteur peu scrupuleux puisse lancer le fichier directement.

Ce fichier doit également, et surtout, contenir la classe principale du module. PrestaShop utilise la programmation orientée Objet, et ses modules également.

Cette classe doit porter le même nom que le module et son dossier, en CamelCase (lire http://fr.wikipedia.org/wiki/CamelCase) : `MyModule`.
De plus, cette classe doit étendre la classe `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`...

mymodule.php

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

class MyModule extends Module
{
}
?>
```

Vous pouvez dès maintenant placer le dossier du module dans le dossier /modules de PrestaShop ; votre module sera déjà visible dans la page "Modules" du back-office, dans la section "Autres modules".

Remplissons maintenant notre classe avec les lignes de démarrage essentielles :

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 = 'Firstname Lastname';
    $this->need_instance = 0;
    $this->ps_versions_compliancy = array('min' => '1.5', 'max' => '1.5');
    $this->dependencies = array('blockcart');

    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');
  }
}
?>
```

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

```
public function __construct()
```

Cette ligne déclare la fonction constructeur de notre classe.

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

Cette fonction assigne une poignée d'attributs à l'instance de la classe (`this`) :

- **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 `seo`, `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
`i18n_localization`	Internationalisation et localisation
`market_place`	Places de marché
`merchandizing`	Merchandising
`migration_tools`	Outils de migration
`mobile`	Mobile
`others`	Autres modules
`payments_gateways`	Paiements
`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' => '1.5');
$this->dependencies = array('blockcart');
```

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. Dans l'exemple ci-dessus, nous indiquons explicitement que ce module ne fonctionnera qu'avec la version 1.5, et aucune autre.
- `dependencies` est un nouveau drapeau de PrestaShop 1.5. Il permet d'indiquer clairement que le module a besoin de l'activation d'un autre module afin de fonctionner correctement. Votre module peut avoir besoin de fonctionnalités mises en place par l'autre module, ou il peut simplement être un ajout utile qui n'aurait aucun sens sans cet autre module. Utilisez le dossier du module comme identifiant. Dans notre exemple, notre module requiert l'activation du module Blockcart.

Nous faisons ensuite appel au parent du constructeur :

```
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 du module, encapsulées dans la fonction de traduction `l()`:

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

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

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

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" quand vous la rechargez, avec quelques premières informations. Cependant, en cliquant sur le bouton "Install", rien ne se passe. Il reste des méthodes à ajouter.

Les méthodes install() et uninstall()

Ces deux méthodes vous permettent de contrôler ce qu'il 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`).

```
public function install()
{
  return (parent::install());
}

// Ceci marche également et est plus pérenne :

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 renvoyée par la méthode `install()` de la classe `Module`, qui renvoie soit `true` si le module est correctement installé, soit `false` dans le cas contraire. Notez que si nous n'avions pas créé cette méthode, la méthode `install()` de la classe parente aurait de toute façon été appelée, ce qui donne le même résultat.
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 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".

```
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');
  }
}
```

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

De son côté, la méthode d'installation supprime simplement la variable de configuration `MYMODULE_NAME`.

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

// ...qui est simplement une manière plus courte d'écrire ceci :

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

Si votre module crée effectivement des données SQL, que ce soit dans ses propres tables MySQL ou dans l'une des tables `ps_` existantes, alors il faut impérativement que vous mettiez en place un processus de suppression de ces données dans la méthodes `uninstall()`. Cette méthode pourrait alors ressembler à ceci :

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

L'objet Configuration

Comme vous pouvez le voir, nos trois blocs de code utilisent un nouvel objet, `Configuration`. Il s'agit d'un objet propre à PrestaShop, conçu 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.

Il y en a de nombreuses autres, comme `getInt()` ou `hasContext()`, mais les quatre présentées sont celles que vous utiliserez le plus souvent.
Comme vous pouvez le voir, c'est un objet 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.

Multiboutique

Par défaut, toutes ces méthodes fonctionnent dans le contexte de la boutique actuelle, que PrestaShop soit en mode multiboutique ou non.

Cependant, il est possible de les faire travailler en dehors du contexte en cours, et d'avoir un impact sur les autres boutiques connues. Cela peut se faire à l'aide de trois paramètres optionnels (non présentés ci-dessus) :

- `id_lang` : permet de forcer la langue avec laquelle on souhaite travailler ;
- `id_shop_group` : permet de préciser le groupe de boutiques de la boutique cible ;
- `id_shop` : permet de préciser l'identifiant de la boutique cible.

Par défaut, ces trois paramètres utilisent les valeurs du contexte en cours, mais vous pouvez leur faire cibler une autre boutique.

Notez bien qu'il est déconseillé de modifier les valeurs par défaut de ces variables, surtout si le module que vous créez est destiné à être utilisé sur d'autres boutiques que la vôtre. Elles ne devraient être utilisées que si le module est conçu pour votre propre installation de PrestaShop, et que vous connaissez les identifiants et groupes de boutiques de toutes les autres boutiques.

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, au format standard TZ (voir : http://en.wikipedia.org/wiki/List_of_tz_database_time_zones).
- `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 de contact principale.
- `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.

Plongez-vous dans la table `ps_configuration` afin de découvrir de nombreux autres réglages.

Notez que lorsque vous utilisez `updateValue()`, le contenu de `$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. Par exemple, voici comme gérer un tableau PHP avec l'objet `Configuration` :

```
// Stocker un tableau sérialisé.
Configuration::updateValue('MYMODULE_SETTINGS', serialize(array(true, true, false)));

// Récupérer le tableau.
$configuration_array = unserialize(Configuration::get('MYMODULE_SETTINGS'));
```

L'objet Shop

Deux autres lignes de la méthode `install()` sont comme suit :

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

L'objet Shop est une nouveauté de PrestaShop 1.5, qui vous permet de gérer la fonctionnalité multiboutique. Sans plonger dans les détails, nous allons simplement vous présenter les deux méthodes qui sont utilisées dans ce code d'exemple :

- `Shop::isFeatureActive()` : cette ligne teste simplement si la fonctionnalité multiboutique 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.

Le contexte est une autre fonctionnalité apparue avec PrestaShop 1.5. Il est expliqué en détail dans le chapitre "Utiliser l'objet Context".

Le fichier d'icônes

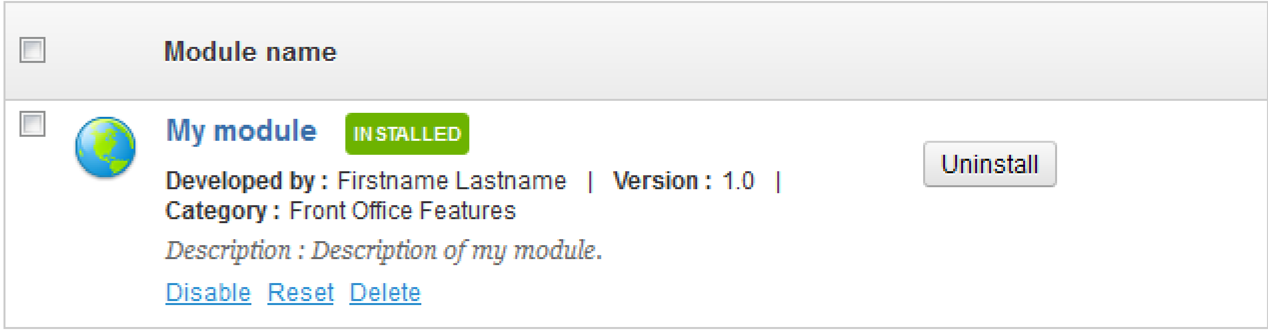
Pour parfaire ce premier module, nous pouvons ajouter une icône, qui sera affichée à côté du nom du module dans la liste des modules.
Dans le cas où le module utilise officiellement un service web connu, le fait d'utiliser le logo de ce service comme icône apporte une plus grande confiance en votre module.
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`.
    - Suggestion : le jeu d'icône gratuit Silk de FamFamFam est probablement ce qui se fait de mieux : http://www.famfamfam.com/lab/icons/silk/.
- Pour fonctionner avec PrestaShop 1.5 :
    - Image en 32*32.
    - Format PNG.
    - Nommé `logo.png`.
    - Suggestion : il existe de nombreuses bibliothèques d'icônes 32*32 gratuites. En voici quelques-unes : http://www.fatcow.com/free-icons (très proche de celle de FamFamFam) ou http://www.jonasraskdesign.com/downloads/downloads.html (Danish Royalty Free).

Apparence dans la page des modules

Maintenant que toutes les bases sont en place, rechargez la page "Modules" du back-office et trouvez votre module dans la section "Fonctionnalités front-office". Installez-le (ou réinitialisez-le s'il est déjà installé).



Lors de l'installation, PrestaShop crée automatiquement un fichier `config.xml` dans le dossier du module, qui stocke les informations de configuration. Vous devez être très prudent lorsque vous décidez de le modifier.

PrestaShop ajoute également une ligne dans la table `ps_module`.

+ Options

←T→				id_module	name	active	version
☐	Edit	Copy	Delete	61	statssearch	1	1
☐	Edit	Copy	Delete	62	statsstock	1	1
☐	Edit	Copy	Delete	63	statsvisits	1	1
☐	Edit	Copy	Delete	64	autoupgrade	1	0.9.4
☐	Edit	Copy	Delete	65	blocklayered	1	1.8.9
☐	Edit	Copy	Delete	68	mymodule	1	1.0

Check All / Uncheck All *With selected:* Change Delete Export

À propos du fichier config.xml

Le fichier `config.xml` permet d'optimiser le chargement du module dans la liste des modules.

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

Quelques détails:

- `is_configurable` indique que le module a une page de configuration ou non.
- `need_instance` indique qu'une instance du module doit être créée lorsque s'affiche la liste des modules. Cela peut se révéler utile lorsque le module doit réaliser des tests sur la configuration de PrestaShop, et afficher des notifications en fonction du résultat.
- `limited_countries` est utilisé pour indiquer les pays auxquels le module est limité. Par exemple, si le module doit être limité à la France et l'Espagne, utilisez <limited_countries>fr,es</limited_countries>.

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

```
public function hookDisplayLeftColumn($params)
{
  $this->context->smarty->assign(
      array(
          'my_module_name' => Configuration::get('MYMODULE_NAME'),
          'my_module_link' => $this->context->link->getModuleLink('mymodule', 'display')
      )
  );
  return $this->display(__FILE__, 'mymodule.tpl');
}

public function hookDisplayRightColumn($params)
{
  return $this->hookDisplayLeftColumn($params);
}

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

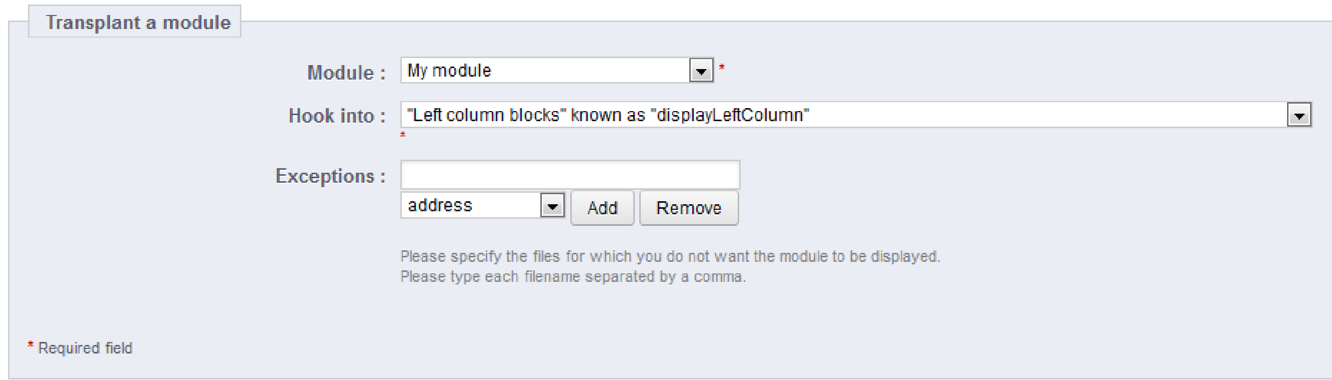
En plus de cela, nous utiliserons le Contexte pour changer une variable Smarty : la méthode `assign()` permet de donner une valeur à la variable `MYMODULE_NAME` stockée dans la table de configuration de la base de données.

Le hook `header` n'est pas un entête visuel, mais nous permet d'ajouter du code entre les balises `<head>` du fichier HTML résultant. C'est surtout utile pour les 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 `<link>` vers le fichier indiqué en paramètre.

Enregistrez votre fichier, et déjà vous pouvez accrocher le template de votre module au thème, le déplacer et le transplanter : aller dans la page "Position" du menu "Modules" du back-office, et cliquez sur le bouton "Greffer un module" (en haut à droite de la page).

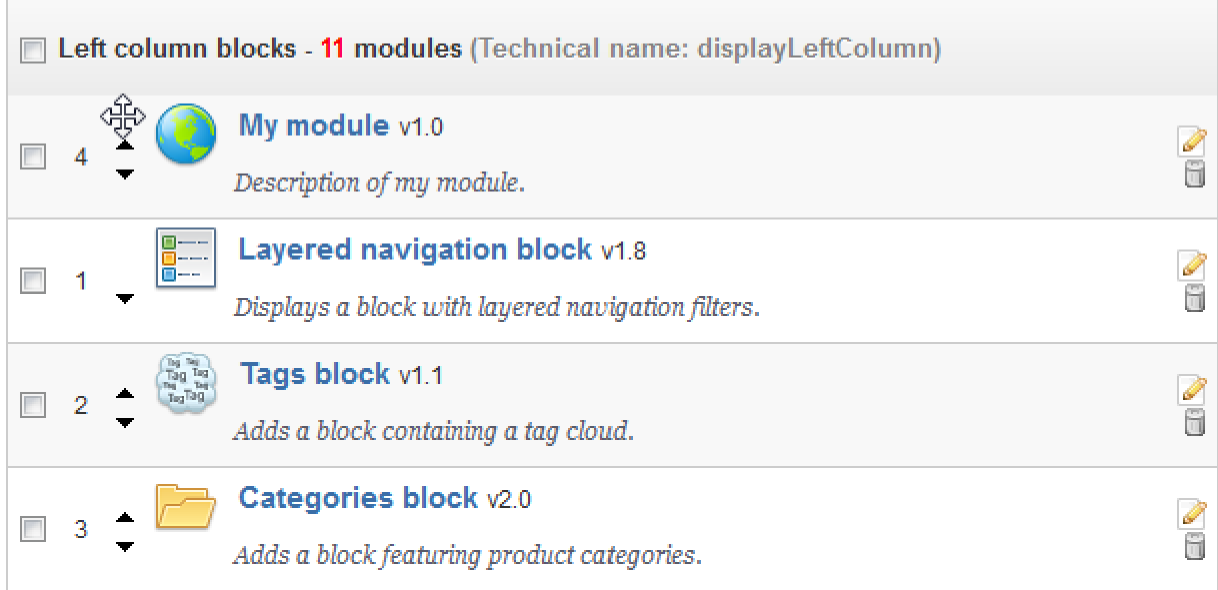
Dans le formulaire de greffe :

1. Trouvez "Mon module" dans la liste déroulante "Module".
2. Choisissez "Left column blocks" dans la liste déroulante "Greffer le module sur".
3. Enregistrez vos modifications.

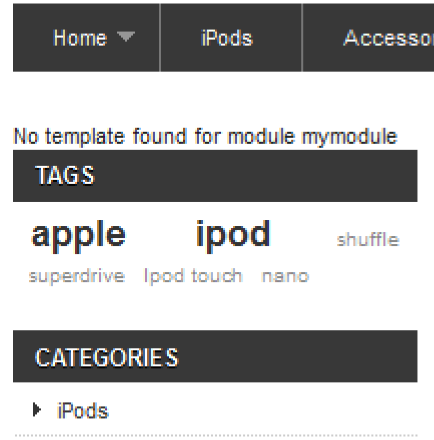


Il est inutile d'attacher un module à un hook pour lequel le module ne dispose pas de méthode implémentée.

La page "Positions" devrait alors se recharger, en affichant le message suivant "Le module a bien été greffé sur le point d'accroche". Félicitations ! Faites défiler la page, et vous devriez effectivement voir votre module parmi ceux affichés dans la liste "Left column blocks". Déplacez-le vers le haut de la liste en faisant un glisser-déposer de la ligne du module.



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 : si vous la rechargez, vous noterez que la colonne de gauche affiche simplement un message là où le module devrait se trouver, disant "No template found for module mymodule".



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 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), que vous devez créer vous-même.

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

mymodule.tpl

```
<!-- Block mymodule -->
<div id="mymodule_block_left" class="block">
  <h4>Welcome!</h4>
  <div class="block_content">
    <p>Hello,
       {if isset($my_module_name) && $my_module_name}
           {$my_module_name}
       {else}
           World
       {/if}
       !
    </p>
    <ul>
      <li><a href="{$my_module_link}" title="Click this link">Click me!</a></li>
    </ul>
  </div>
</div>
<!-- /Block mymodule -->
```

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

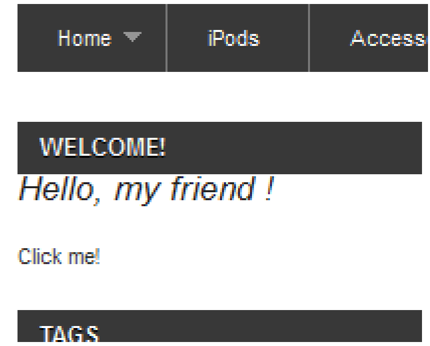
- L'appel `{l s='xxx' mod='yyy'}` est une méthode propre à PrestaShop qui enregistre la chaîne dans le panneau de traduction du module. Le paramètre `s` est pour la chaîne, tandis que le paramètre `mod` doit contenir l'identifiant du module (dans le cas présent, "mymodule"). Nous n'utilisons cette méthode qu'une fois dans cet exemple pour des raisons de lisibilité, mais dans les faits il faudrait l'appliquer à absolument toutes les chaînes.
- Les déclarations `{if}`, `{else}` et `{/if}` sont des conditions Smarty. Dans notre exemple, nous vérifions que la variable Smarty `$my_module_name` existe bien (grâce à la méthode Smarty `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 "Hello World!"

- La variable `{$my_module_link}` dans l'attribut `href` : il s'agit d'une variable Smarty que nous allons bientôt créer, et qui donne accès au dossier racine de PrestaShop.

En parallèle, nous allons créer un fichier CSS, et allons l'enregistrer en tant que `/css/mymodule.css` dans le dossier du module (ou tout autre sous-dossier où vous souhaitez placer vos fichiers 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 très joli, mais cela fonctionne comme nous le souhaitons.

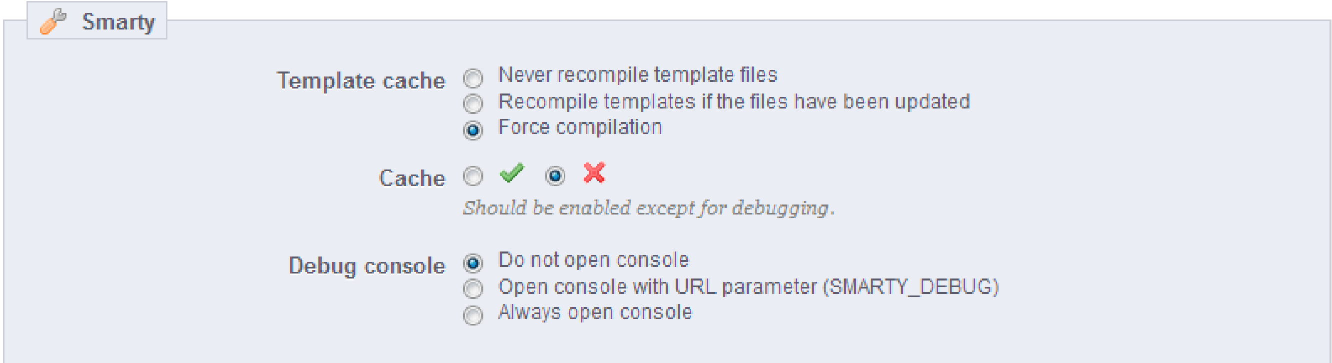
Désactiver le cache

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 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 votre page d'accueil, 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 la page d'accueil afin de voir l'impact de vos modifications.

Lorsque vous modifiez ou déboguez le thème d'un site de test, vous devriez toujours désactiver le cache, afin de forcer Smarty à recompiler les templates à chaque chargement.
Pour ce faire, rendez-vous dans le menu "Paramètres avancés", ouvrez la page "Performances", et dans la section "Smarty" :

- **Cache des templates**. Choisissez "Forcer la compilation à chaque appel".
- **Cache**. Désactivez-le.
- **Console de débogage**. Vous pouvez également faire s'ouvrir la console si vous voulez en apprendre plus sur le fonctionnement interne de Smarty.



Ne désactivez jamais le cache sur un site en production, ni n'activez la console de débogage, car cela ralentirait tout !
Vous devriez toujours réaliser vos tests sur un site à part, idéalement installé sur votre propre ordinateur plutôt qu'en ligne.

Intégrer un template à un thème

Le lien que le module affiche 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.

display.php

```
Welcome to this page!
```

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.

Vous devriez vous assurer de donner des noms explicites à vos fichiers templates, afin de pouvoir les retrouver facilement dans votre back-office – ce qui est indispensable une fois que vous devez utiliser l'outil de traduction.

Voici nos deux fichiers :

display.php

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

display.tpl

```
Welcome to my shop!
```

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

- 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()` avec notre fichier `display.tpl`.

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

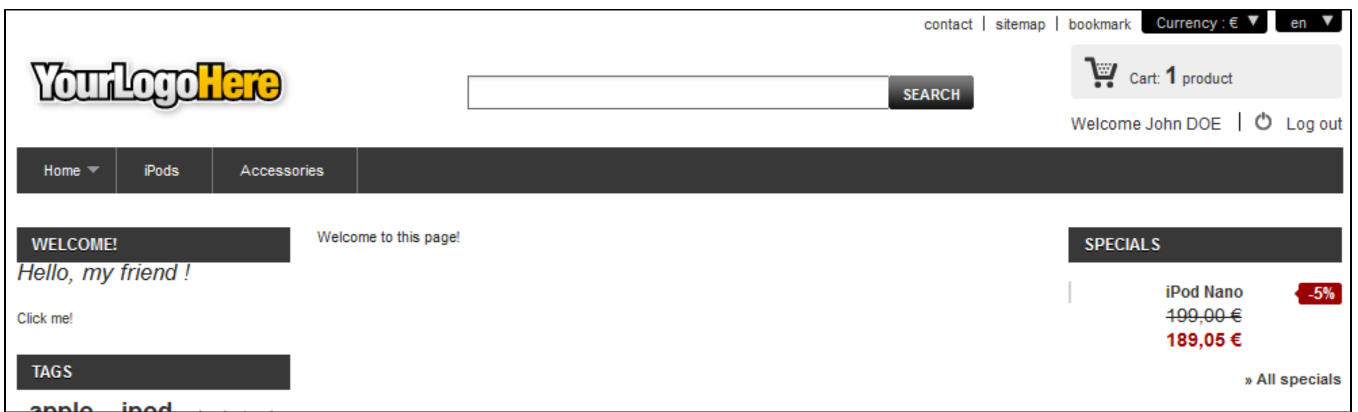
Jusqu'à PrestaShop 1.4, les développeurs qui souhaitaient intégrer un fichier template dans le thème du site devaient utiliser des appels `include()` pour chaque portion de la page. Voici le code 1.4 de notre `display.php` ci-dessus :

display.php

```
<?php
// Ce fichier doit être placé à la racine du dossier du module
global $smarty;
include('../../config/config.inc.php');
include('../../header.php');

$smarty->display(dirname(__FILE__).'/display.tpl');

include('../../footer.php');
?>
```

Comme vous pouvez le voir, cela n'est plus nécessaire avec PrestaShop 1.5 : vous pouvez et même devriez utiliser un contrôleur front-end, et le contrôleur (Contrôleur) et son template (Vue) devraient avoir le même nom : `display.php` est lié à `display.tpl`.

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 l'interface.



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.

Utiliser Smarty

Smarty est un moteur de template. C'est celui utilisé par le système de thème de PrestaShop. Il s'agit d'un projet libre et open-source, hébergé à http://www.smarty.net/.

Il parcourt les fichiers `.tpl`, à la recherche d'éléments dynamiques à remplacer par leurs équivalents contextuels, pour envoyer au navigateur le résultat générique. Ces éléments dynamiques sont marqués par des accolades : `{ ... }`. Les développeurs peuvent créer leurs propres variables. PrestaShop ajoute ses propres variables.

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

mymodule.php

```
public function hookDisplayLeftColumn($params)
{
    $this->context->smarty->assign(
        array(
            'my_module_name' => Configuration::get('MYMODULE_NAME'),
            'my_module_link' => $this->context->link->getModuleLink('mymodule', 'display'),
            'my_module_message' => $this->l('This is a simple text message') // 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.

mymodule.tpl

```
{$my_module_message}
```

PrestaShop ajoute ses propres variables. 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 `{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.
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.

Il y a de nombreux hooks contextuels. Si vous avez besoin d'afficher toutes les variables de la page en cours, utiliser l'appel suivant :

```
{debug}
```

Les commentaires sont signalés par des astérisques :

```
{* This string is commented out *}

{*
This string is too!
*}
```

Au contraire des commentaires HTML, le code Smarty commenté n'apparaît pas dans le fichier final.

Ajouter une page de configuration

Votre module peut avoir un lien "Configurer" dans la liste des modules du back-office, et donc permettre à l'utilisateur de modifier quelques réglages. Ce lien "Configurer" apparaît avec l'addition de la méthode `getContent()` dans la classe principale. 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 donc vous expliquer comme la créer, afin d'y placer un formulaire permettant de modifier la variable de configuration `MYMODULE_NAME` qui est stockée dans la table `ps_configuration`.

La méthode getContent()

Tout d'abord, voici le code complet pour notre méthode `getContent()` :

```
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 :

1. `Tools::isSubmit()` est une méthode propre à PrestaShop, qui vérifie si le formulaire indiqué a bien été validé.
   Dans ce cas, si le formulaire de configuration, 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()`.
2. `Tools:getValue()` est une méthode propre à PrestaShop, qui récupère le contenu du tableau `POST` ou `GET` pour en tirer la valeur de la variable indiquée.
   Dans ce cas, nous récupérons la valeur de `MYMODULE_NAME` du formulaire, transformons sa valeur en une chaîne à l'aide de la méthode `strval()`, et la stockons dans la variable PHP `$my_module_name`.
3. Nous vérifions ensuite l'existence de véritable contenu dans `$my_module_name`, notamment en utilisant `Validate::isGenericName()`.
   L'objet `Validate` contient de nombreuses méthodes 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.
4. 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 à l'aide de la méthode `displayError()`. Cette méthode renvoie le code HTML nécessaire à nos besoins, et étant qu'il est le premier inséré dans $output, cela signifie que la page s'ouvrira avec ce message.

5. 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, l'objet `Configuration` a exactement la méthode dont nous avons besoin : `updateValue()` stockera la nouvelle valeur pour `MYMODULE_NAME` dans la table de configuration.
   Pour cela, 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 `$output` – et donc, en haut de la page.
6. Enfin, nous utilisons la méthode `displayForm()` (que nous allons créer et expliquer dans la section suivante) pour ajouter du contenu à `$outpu`
   (que le formulaire ait été validé ou non), et renvoyons ce contenu dans la page.
   Notez que 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.

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.

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 screen.
    $helper->submit_action = 'submit'.$this->name;
    $helper->toolbar_btn = array(
        'save' =>
        array(
            'desc' => $this->l('Save'),
            'href' => AdminController::$currentIndex.'&configure='.$this->name.'&save'.$this->name.
            '&token='.Tools::getAdminTokenLite('AdminModules'),
        ),
        'back' => array(
            'href' => AdminController::$currentIndex.'&token='.Tools::getAdminTokenLite('AdminModules'),
            'desc' => $this->l('Back to list')
        )
    );

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

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

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 l'objet `HelperForm`.

Plonger dans displayForm()

Parcourons cette méthode :

1. 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 `(int)`.
2. 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 que le formulaire contient. À partir de cette variable, PrestaShop pourra générer le formulaire HTML tel que décrit.
   Dans notre exemple, nous définissons trois balises (`<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 champ <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 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.
3. Nous créons ensuite la classe `HelperForm`. Cette section du code est expliquée dans la prochaine section de ce chapitre.
4. Une fois que les réglages de `HelperForm` sont en place, nous générons le formulaire à partir de la variable `$fields_form`.

Utiliser HelperForm

`HelperForm` est l'une des méthodes d'aide ajoutée à PrestaShop, en même temps que `HelperOptions`, `HelperList`, `HelperView` et `HelperHelpAccess`. Ces méthodes permettent de générer des éléments HTML standard 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`.

Pour rappel, voici notre code :

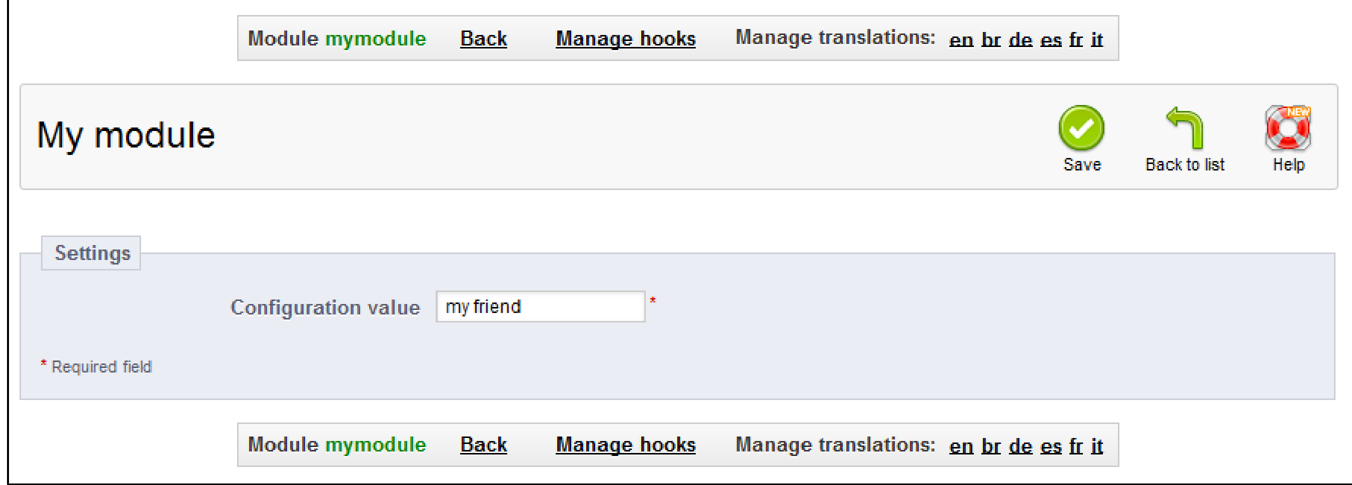
```
$helper = new HelperForm();

// 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 screen.
$helper->submit_action = 'submit'.$this->name;
$helper->toolbar_btn = array(
    'save' =>
    array(
        'desc' => $this->l('Save'),
        'href' => AdminController::$currentIndex.'&configure='.$this->name.'&save'.$this->name.
        '&token='.Tools::getAdminTokenLite('AdminModules'),
    ),
    'back' => array(
        'href' => AdminController::$currentIndex.'&token='.Tools::getAdminTokenLite('AdminModules'),
        'desc' => $this->l('Back to list')
    )
);

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

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

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` :

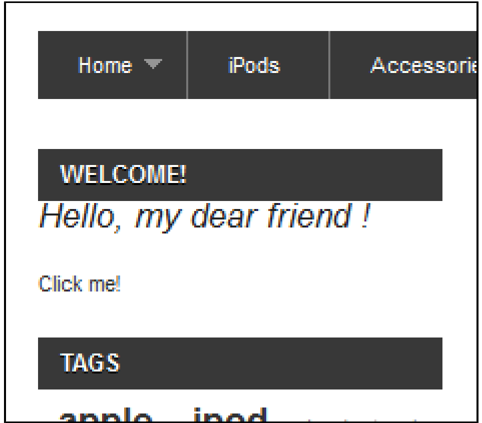
- `$helper->module` : requiert une instance du module qui utilisera les données du formulaire.
- `$helper->name_controller` : requiert le nom du module.
- `$helper->token` : requiert un jeton (*token*) unique et propre au module. `getAdminTokenLite()` en génère un pour vous.
- `$helper->currentIndex`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

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 :



Modifiez la valeur comme vous le souhaitez, cliquez sur le bouton "Save" et rechargez la page d'accueil : votre module devrait effectivement afficher une chaîne différente !



Traduction du module

Les chaînes du module sont écrites en anglais, vous pourriez vouloir que les propriétaires de boutiques français, espagnols ou polonais puissent également utiliser le module. Vous devez donc traduire ces chaînes dans ces différentes langues, que ce soit pour les chaînes du back-office comme celles du front-office. Idéalement, vous devriez traduire votre module dans l'ensemble des langues disponibles sur votre boutique. Cela peut s'avérer être une tâche laborieuse, mais Smarty et l'outil de traduction de PrestaShop vous simplifient au maximum la tâche.

Les chaînes des fichiers PHP doivent être affichées à l'aide de la méthode `l()`, qui provient de la classe abstraite Module, et est donc disponible au sein de tous les modules.

mymodule.php (partial)

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

Les chaînes des fichiers TPL doivent être transformées en contenu dynamique, que Smarty remplacera par la traduction dans la langue choisie. Dans notre exemple, le fichier `mymodule.tpl`...

mymodule.tpl (partial)

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

...devient...

mymodule.tpl (partial)

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

...et le fichier `display.tpl`...

display.tpl

```
Welcome to this page!
```

...devient...

display.tpl

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

Traduire du code complexe

Comme vous pouvez le voir, la base de la traduction d'un fichier template consiste à placer ses chaînes dans le bon appel Smarty : `{l s='The string' mod='name_of_the_module'}`. Les modifications des textes pour les titres et liens des fichiers `display.tpl` et `mymodule.tpl` sont donc faciles à comprendre. Mais nous avons ajouté un bloc de code plus conséquent pour la chaîne "Hello World" : une condition `if/else/then`, et une variable de texte. Explorons ce code.

Voici le code original :

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

Comme vous pouvez le voir, nous devons rendre la chaîne "Hello World!" traduisible, mais également prendre en compte le fait qu'il y a une variable. Comme expliqué dans le chapitre "Les traductions dans PrestaShop 1.5", les variables doivent être marquées à l'aide de marqueurs `sprintf()`, tels que %s ou %1$s.

Il est facile de rendre "Hello %s!" traduisible : nous devons simplement mettre ce code en place :

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

Mais dans notre cas, nous devons également faire en sorte que %s soit remplacé par "World" dans le cas où "my_module_name" n'existe pas... et nous devons rendre "World" également traduisible. Cela peut se faire en utilisant la fonction `{capture}` de Smarty, qui récupère la valeur de sortie au lieu de l'afficher, afin de l'utiliser plus tard. Nous l'utiliserons pour remplacer la variable avec le mot "World" s'il se trouvait que cette même variable était vide ou absente, à l'aide d'une variable temporaire. Voici le code final :

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

Notez que nous utilisons toujours le paramètre `mod`. Il est utilisé par PrestaShop pour vérifier à quel module une chaîne appartient. L'outil de traduction en a besoin pour faire correspondre une chaîne à sa traduction. Ce paramètre est **obligatoire** pour les chaînes de modules.

Les chaînes sont délimitées par des apostrophes droites. Si une chaîne contient des apostrophes, elles devraient être échappées par un antislash (`\`).

Ainsi, les chaînes peuvent être traduites directement dans PrestaShop :

- Rendez-vous dans la page "Traductions", dans le menu "Localisation".
- Dans la liste déroulante de la section "Modifier la traduction", choisissez "Traduction des modules installés"
- Cliquez sur le drapeau du pays dont la langue correspond à celle dans laquelle vous voulez traduire le module. La langue de destination doit être déjà installée dans PrestaShop.

La page qui se charge alors affiche les chaînes de tous les modules actuellement installés. Les modules qui ont déjà leurs chaînes traduites ont leurs sections fermées, tandis que ceux qui ont au moins une chaîne non traduite ont leur section ouverte.
Pour traduire les chaînes de votre module (celles que vous avez marqué avec la méthode `l()`), trouvez simplement votre module dans la liste (utilisez la recherche de votre navigateur) et remplissez les champs vides.

Une fois toutes les chaînes de votre module correctement traduites, cliquez sur le bouton "Enregistrer les modifications" en haut de la page.

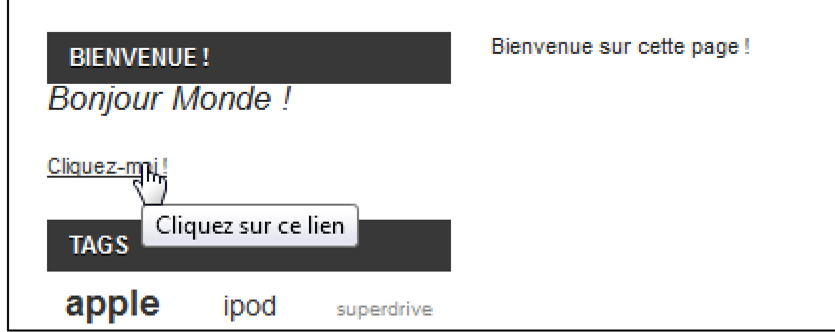
PrestaShop enregistre alors votre traduction dans un nouveau fichier, nommé sous la forme `codeDeLaLangue.php` (par exemple, `/mymodule/fr.php`). Le fichier de traduction ressemble à ceci :

fr.php

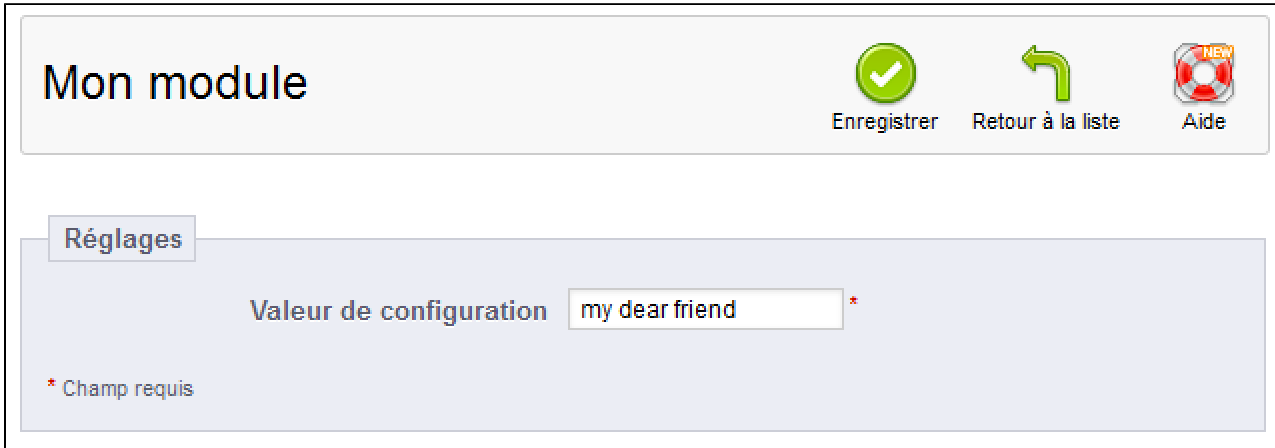
```
<?php
global $_MODULE;
$_MODULE = array();
$_MODULE['<{mymodule}prestashop>mymodule_2ddddc2a736e4128ce1cdfd22b041e7f'] = 'Mon module';
$_MODULE['<{mymodule}prestashop>mymodule_d6968577f69f08c93c209bd8b6b3d4d5'] = 'Description du module.';
$_MODULE['<{mymodule}prestashop>mymodule_533937acf0e84c92e787614bbb16a7a0'] = 'Êtes-vous certain de vouloir
désinstaller ce module ? Vous perdrez tous vos réglages !';
$_MODULE['<{mymodule}prestashop>mymodule_0f40e8817b005044250943f57a21c5e7'] = 'Aucun nom fourni';
$_MODULE['<{mymodule}prestashop>mymodule_fe5d926454b6a8144efce13a44d019ba'] = 'Valeur de configuration non
valide.';
$_MODULE['<{mymodule}prestashop>mymodule_c888438d14855d7d96a2724ee9c306bd'] = 'Réglages mis à jour';
$_MODULE['<{mymodule}prestashop>mymodule_f4f70727dc34561dfde1a3c529b6205c'] = 'Réglages';
$_MODULE['<{mymodule}prestashop>mymodule_2f6e771db304264c8104cb7534bb80cd'] = 'Valeur de configuration';
$_MODULE['<{mymodule}prestashop>mymodule_c9cc8cce247e49bae79f15173ce97354'] = 'Enregistrer';
$_MODULE['<{mymodule}prestashop>mymodule_630f6dc397fe74e52d5189e2c80f282b'] = 'Retour à la liste';
$_MODULE['<{mymodule}prestashop>display_86e88cbccafa83831b4c6685501c6e58'] = 'Bienvenue sur cette page !';
$_MODULE['<{mymodule}prestashop>mymodule_9a843f20677a52ca79af903123147af0'] = 'Bienvenue !';
$_MODULE['<{mymodule}prestashop>mymodule_f5a7924e621e84c9280a9a27e1bcb7f6'] = 'Monde';
$_MODULE['<{mymodule}prestashop>mymodule_3af204e311ba60e6556822eac1437208'] = 'Bonjour %s !';
$_MODULE['<{mymodule}prestashop>mymodule_c66b10fbf9cb6526d0f7d7a602a09b75'] = 'Cliquez sur ce lien';
$_MODULE['<{mymodule}prestashop>mymodule_f42c5e677c97b2167e7e6b1e0028ec6d'] = 'Cliquez-moi !';
```

Ce fichier ne doit pas être modifié à la main ! Elle ne peut être modifiée qu'en passant par l'outil de traduction de PrestaShop.

Maintenant que nous avons une traduction française, nous pouvons cliquer sur le drapeau français, et obtenir les résultats attendus : les chaînes du module sont maintenant en français.



Elles sont également traduites en français dans le back-office.



En cas de problème

Si le module ne fonctionne pas comme attendu, voici quelques possibles solutions.

Forums officiels de PrestaShop

Rejoignez nos forums à l'adresse http://www.prestashop.com/forums/, et cherchez-y une réponse à votre question en tapant les mots-clefs. Si vous ne trouvez rien, utilisez le formulaire de recherche avancé. Et si ici encore la recherche ne donne rien, créez une nouvelle discussion, dans laquelle vous pourrez donner autant de détails que nécessaire en écrivant votre question. Notez que vous devrez être enregistré pour créer une discussion.

Certains forums ont des discussions situés en haut de la page : elles contiennent de précieuses informations, lisez-les attentivement.

Notre bug-tracker

S'il se trouve que votre problème vient d'un bug de PrestaShop plutôt que d'une erreur de votre côté, signalez le problème au bug-tracker de PrestaShop : http://forge.prestashop.com/ (vous devrez d'abord créer un compte). Cela vous permettra de discuter du problème directement avec les développeurs de PrestaShop.

Sites officiels de PrestaShop

URL	Description
http://www.prestashop.com	Site officiel du logiciel PrestaShop, de sa communauté et de la société qui s'en occupe.
http://addons.prestashop.com	Place de marché de thèmes et modules.
http://www.prestabox.com	Faites héberger votre boutique par PrestaShop !