Chapter No. 1
"Creating a New Module"
In this package, you will find:

The author’s biography

A preview chapter from the book, Chapter no.1 "Creating a New Module"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Fabien Serny is a former core developer at PrestaShop. He has 10 years of experience in web development and e-commerce. He has worked for several big e-commerce companies in France, and then created his own company named 23Prod in late 2010. In 2014, along with two other former core developers from PrestaShop, he launched Froggy Commerce, a platform that sells simple and powerful modules for PrestaShop based on the needs of e-tailers. You can visit his websites http://www.23prod.com and http://www.froggy-commerce.com.
I'd like to thank my lovely wife, Camille, who supported and pushed me to write this book.

I'd also like to thank my co-workers, Gregoire Poulain and Alain Folletete from Froggy Commerce; Bruno Leveque, founder of PrestaShop, who encouraged me to write this book; and some of my former colleagues at PrestaShop: Franck (the wise), Raphaël, Vincent, François, Sabrina, Cécile, Mareva, Julien, Tony, and the rest of the development team and the company.

Lastly, I would like to thank my family (my mum, Claude, Philou, BreeBree, Greg, Milo, Ludovic, Oliver—the boot, Astrid, Enzo, my grandma, and so on) and friends (Nico, Sophie, Caro, Romain alias "The Master", Elodie, Tom, Alain, Seb, Mélo, Micka, Gaël, Gégé, Ludo, Béré, Elo Lobster, Jérome, and so on); I would like to mention them all but it would be a very long list. However, I would not like to thank the great caravan that made me sleep during my trip in America instead of letting me work on this book.

I would also like to specially thank Benjee, Moni, and Julien, who helped me design the cover of this book; my editors Joanne, Arun, and Humera, who did an incredible job helping me on this book; and my reviewers Romain, Altaf, Sandeep, Ratko, Tom, and Bart.

For More Information:  
www.packtpub.com/web-development/prestashop-module-development
PrestaShop Module Development

This book covers how to develop different modules for PrestaShop 1.6. It will walk you through the widely used best practices and empower you to use these methods while developing various features in PrestaShop. This book will help you create powerful and scalable modules, thereby improving the performance of your shop.

Throughout the course of this book, you will be introduced to a number of clear, practical examples of PrestaShop concepts (such as hooks) in the form of a tutorial. You will also get acquainted with the best practices of PrestaShop, which will greatly reduce the probability of introducing errors in your future modules.

By the end of this book, you will be accustomed with PrestaShop best practices and the important concepts to make a great module.

What This Book Covers

Chapter 1, Creating a New Module, covers the first step of module creation. This chapter introduces you to a small module example with a simple configuration form.

Chapter 2, Hooks, presents the concept of hooks in PrestaShop, including how they work and how they can be used to interact with the software. The use of the database class is also covered with practical examples.

Chapter 3, Using Context and its Methods, introduces you to the many methods available in PrestaShop, such as the multilingual function or JS/CSS manager.

Chapter 4, Building Module Updates, introduces you to the native module update system. Some extra functional tips are also revealed.

Chapter 5, Front Controllers, Object Models, and Overrides, presents the main design patterns of PrestaShop and explains how to use them to construct a well-organized application.

Chapter 6, Admin Controllers and Hooks, explains how to create new administration tools, use the helper to build forms or an object list, and how to use the hooks available in the administration panel.

Chapter 7, The Carrier Module, introduces you to the creation of a module carrier. Some advanced features such as delivery point handling are also covered.

Chapter 8, The Payment Module, introduces you to the creation of a payment module. An example of creating a payment API is covered.

For More Information:
www.packtpub.com/web-development/prestashop-module-development
Chapter 9, Multistore, covers the main guidelines to make your module compliant with the multistore native PrestaShop feature.

Chapter 10, Security and Performance, explains the best practices to make your module secure and efficient.

Appendix, Native Hooks, gives a description of all the hooks that PrestaShop offers.

For More Information:
www.packtpub.com/web-development/prestashop-module-development
Creating a New Module

To learn how to code a PrestaShop module, it is always easier to work on a practical case. So, throughout the next chapters, we will develop a module that will permit customers to grade and comment on products.

In this first chapter, we will see how to:

- Create the Bootstrap of an installable module
- Add a configuration form to the module using Smarty templates
- Register the module configuration in a database

First steps

First of all, we have to choose a public and technical name for the module. The technical name must be in lowercase, contain only letters and numbers, and begin with a letter. Let’s call the module publicly My Module of product comments and technically mymodcomments.

You can choose whatever names you want as long as you stick to them for the rest of the book. However, since the technical name will be the directory name of your module, you won't be able to have two modules with the same technical name in a PrestaShop. Moreover, your technical name has to be unique if you want to sell it on marketplaces. One trick is to prefix it with your company name (in my case, my company name is Froggy Commerce, so I prefixed all my modules with froggy).

On the other hand, the public name has no restriction, so you can write whatever you want for the merchant.

For More Information:
www.packtpub.com/web-development/prestashop-module-development
Creating a New Module

Now, we will begin to create our first module.

Open the modules directory in the root of the PrestaShop directory, then create a new directory and name it with the technical name we chose: mymodcomments. Once done, in this new directory, create a new blank PHP file and name it with the technical name as well (in our case, mymodcomments.php). This is the main file of the module. The following screenshot depicts the file and folder structure:

Now open the mymodcomments.php file, and write the class of your module. You must name it with your technical name. To make it easier to read, you can use CamelCase. The class must extend PrestaShop's Module class. This class contains all the needed methods to make a module work; without it, your module won't work. In our case, the class of the module will be:

```php
<?php
class MyModComments extends Module {
}
```

In order to have a working module, we just have to add the __construct method. In this method, you must write the following three mandatory lines:

- **Set the technical name**: Without this, the module won't be installable. This variable is used by PrestaShop to build the install, uninstall, and configure links of the module:
  ```php
  $this->name = 'mymodcomments';
  ```

For More Information:

www.packtpub.com/web-development/prestashop-module-development
• **Set the public name:** This line is used to display the module name for the merchant in the modules list of the back office:

```php
$this->displayName = 'My Module of product comments';
```

• **Calling the parent `__construct` method:** This line is mandatory since important initializations are made in this function:

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

You can add some of the following optional lines to give information about the module:

• **Set the module category:** This makes the module search easier. If you don't set it or set a wrong value, then the module will automatically be associated with the Other Modules category. You should set one of the values shown in the following table:

<table>
<thead>
<tr>
<th>administration</th>
<th>advertising_marketing</th>
<th>analytics_stats</th>
<th>billing_invoicing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkout</td>
<td>content_management</td>
<td>export</td>
<td>emailing</td>
</tr>
<tr>
<td>front_office_features</td>
<td>i18n</td>
<td>localization</td>
<td>merchandizing</td>
</tr>
<tr>
<td>migration_tools</td>
<td>payments_gateways</td>
<td>payment_security</td>
<td>pricing_promotion</td>
</tr>
<tr>
<td>quick_bulk_update</td>
<td>search_filter</td>
<td>seo</td>
<td>shipping_logistics</td>
</tr>
<tr>
<td>slideshows</td>
<td>smart_shopping</td>
<td>market_place</td>
<td>social_networks</td>
</tr>
<tr>
<td>others</td>
<td>mobile</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These possible values are associated with the module's categories search filter in the back office of your PrestaShop:

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

• **Set the module version:** This variable will not only be used to display the module version in the module's list of the back office but also to check if updates are available (we'll cover this in more detail later):

```php
$this->version = '0.1';
```

• **Set the author name:** This line is used to display the author name for the merchant in the module's list of the back office. It can also be used to search for modules:

```php
$this->author = 'Fabien Serny';
```

For More Information:

www.packtpub.com/web-development/prestashop-module-development
Creating a New Module

- **Set the module description**: This helps the merchant learn what the module is used for:
  
  ```php
  $this->description = 'With this module, your customers will be able to grade and comments your products';
  ```
  
  Here's what your code should look like now:

  ```php
  <?php
  class MyModComments extends Module
  {
    public function __construct()
    {
      $this->name = 'mymodcomments';
      $this->tab = 'front_office_features';
      $this->version = '0.1';
      $this->author = 'Fabien Serny';
      $this->displayName = 'My Module of product comments';
      $this->description = 'With this module, your customers will be able to grade and comments your products.';
      parent::__construct();
    }
  }
  ```

  At this point, you should be able to see your module in the back office of the modules section, as shown in the following screenshot:

  ![](module.png)

  **The question mark icon**

  This is the default logo for all modules. If you want a custom picture, you just have to add a `logo.png` picture of 32 x 32 pixels and a `logo.gif` picture of 16 x 16 pixels (for PrestaShop 1.4 compliancy) at the root of your module directory.

The module is now a working module; you can install it by clicking on the **Install** button. Here is what you should see:

For the moment, your module does nothing and doesn't have any configuration options. The only actions available are:

- **Uninstall**: This option uninstalls the module and deletes the specific configurations, if there are some.
- **Disable**: This is an alternative to the uninstall action but permits you to keep module configurations. The module is still installed and will simply be ignored by PrestaShop.
- **Reset**: This option uninstalls and reinstall the module.
- **Delete**: This option will uninstall the module and then delete all the module files.

All these core functions are handled by the `Module` class and can be overridden by the module (we will see this later).

## Adding the module configuration

We will now add configuration options to our module. To do so, we just have to add the `getContent` function to our module. The return value of this function will be the content displayed on your screen.

For More Information:

www.packtpub.com/web-development/prestashop-module-development
Creating a New Module

In our case, we will write the following code:

```php
public function getContent()
{
    return 'My name is Raphael, I am a tourist';
}
```

The sentence returned is only an example (and a private joke from one of the PrestaShop core developers).

When the `getContent` function is placed in a module's class, PrestaShop automatically displays a `Configure` link in the back office. When the `Configure` link is clicked, this function is called and the return value is displayed.

So, if you refresh the modules list in the back office and your module is installed, you should now see a `Configure` button:

If you click on the configuration link, you will see the translations block (automatically generated by PrestaShop software), and the sentence that we wrote in the function:

You must avoid writing HTML in PHP code (very bad practice). That's why we will use the Smarty template (the template engine used in PrestaShop). If you are not familiar with this library, or with template engines in general, I invite you to read the official documentation (the link is in the introduction of this book). However, do not panic; using Smarty templates is quite easy!

For More Information:
www.packtpub.com/web-development/prestashop-module-development
We will start by creating the templates directory in the root of the module's directory: /views/templates/hook/. All of the templates used in the module's class must be placed in this directory.

One of the best practices is to name the templates with the name of the method in which they are used. So, we will create a template named getContent.tpl. Let's write our previous text in this template:

My name is Raphael, I am still a tourist

Then, in the getContent method, we just have to change the return line to the following:

```php
    return $this->display($__FILE__, 'getContent.tpl');
```

In this case, the display method will automatically use the getContent.tpl template in the /views/templates/hook/ directory. If you refresh the page, you'll see that your display has changed (we added one word). Now, your view is separated from your code. The following screenshot displays the file architecture you should have now:

```
logo.gif
logo.png
mymodcomments.php
views
templates
hook
getContent.tpl
```

Making a simple configuration form

We will now make a small configuration form with two options: one to enable grades and the other one to enable comments on a product.

Let's fill in the getContent.tpl file we created previously with a small form.

Since PrestaShop 1.6, the back office now uses the CSS Framework Bootstrap. You can either write basic HTML (and make the display compliant with an older version of PrestaShop), or write Bootstrap templates.

If you choose to use Bootstrap (as I'll do later), you have to set the Bootstrap flag in your module constructor first.
Creating a New Module

In the __construct method of your module's mymodcomments.php main class, add the following line before parent::__construct:

```php
$this->bootstrap = true;
```

If you do not set this flag to true (and if you are using PrestaShop 1.6), PrestaShop will include a retrocompatibility CSS file to make the template that you used in PrestaShop 1.5 compliant with the PrestaShop 1.6 display. Here is the Bootstrap HTML code that we will use to display the configuration:

```html
<fieldset>
  <h2>My Module configuration</h2>
  <div class="panel">
    <div class="panel-heading">
      <legend><img src="../img/admin/cog.gif" alt="" width="16" />
        Configuration</legend>
    </div>
    <form action="" method="post">
      <div class="form-group clearfix">
        <label class="col-lg-3">Enable grades:</label>
        <div class="col-lg-9">
          <input type="radio" id="enable_grades_1" name="enable_grades" value="1" />
          <label class="t" for="enable_grades_1">Yes</label>
          <input type="radio" id="enable_grades_0" name="enable_grades" value="0" />
          <label class="t" for="enable_grades_0">No</label>
        </div>
      </div>
      <div class="form-group clearfix">
        <label class="col-lg-3">Enable comments:</label>
        <div class="col-lg-9">
          <input type="radio" id="enable_comments_1" name="enable_comments" value="1" />
          <label class="t" for="enable_comments_1">Yes</label>
          <input type="radio" id="enable_comments_0" name="enable_comments" value="0" />
          <label class="t" for="enable_comments_0">No</label>
        </div>
      </div>
    </form>
  </div>
</fieldset>
```

For More Information:

http://www.packtpub.com/web-development/prestashop-module-development
The HTML tags used here are the tags generally used in the native module, but you have no real limitations except your imagination.

I will not go further in my explanation of this code since HTML basics are beyond the scope of this book.

If you refresh your browser, you will see the form appear, but for the moment, submitting the form will do nothing. So, we need to take care of saving the configuration options chosen by the merchant when he submits the form.

In order to not overload the `getContent` method, let's create a new method dedicated to handling the submission of the module configuration form. The name `processConfiguration` seems a good name to me for such a function. Since `getContent` is the only method called by PrestaShop when we enter in a module's configuration, we still have to call the newly created `processConfiguration` method in the `getContent` method:

```php
public function processConfiguration()
{
}

public function getContent()
{
    $this->processConfiguration();
    return $this->display(__FILE__, 'getContent.tpl');
}
```

In the `processConfiguration` method, we will check whether the form has been sent with the `Tools::isSubmit` method:

```php
if (Tools::isSubmit('mymod_pc_form'))
{
}
```

As you might have noticed, we set the `mymod_pc_form` name attribute on the Submit button. PrestaShop's `Tools::isSubmit` method checks whether a key matches the attribute name in the POST and GET values. If it exists, we will know that the form has been sent for sure.

For More Information:
www.packtpub.com/web-development/prestashop-module-development
Creating a New Module

We will now save the data of the form sent. We will use the following two methods for this action:

- **Tools::getValue**: This function will retrieve the POST value of the key passed in the parameter. If the POST value does not exist, it will automatically check for the GET value. We can set a second parameter (optional), which will correspond to the default value if neither POST nor GET value is found.

- **Configuration::updateValue**: This function is used to save simple value in the configuration table of PrestaShop. You just have to set the key as the first parameter and the value as the second. The updateValue function makes a new entry in the configuration table if the provided configuration is not found, and will update the configuration value if it is already in the configuration table. This method can have other parameters, but we will cover them later since we do not need them for the moment.

So in practice, we will have this:

```
if (Tools::isSubmit('mymod_pc_form'))
{
    $enable_grades = Tools::getValue('enable_grades');
    $enable_comments = Tools::getValue('enable_comments');
    Configuration::updateValue('MYMOD_GRADES', $enable_grades);
    Configuration::updateValue('MYMOD_COMMENTS', $enable_comments);
}
```

It is a PrestaShop best practice to set the key in uppercase. Since any module can add configuration value, it's also a best practice to add a prefix to avoid conflict between modules. That's why I added `MYMOD_`.

Now, I invite you to refresh the configuration page in your browser, change the value of one or both options, and then submit the form.

Apparently, nothing has changed. However, if you go into your phpMyAdmin (or any administration tool), and see the content of PrestaShop's configuration table (if you kept the default database prefix, its name should be `ps_configuration`), you will see that the last two entries of the table are your values. You can play with the form and submit it again; the value will be updated.

We still have to display a confirmation message to let the merchant know that their configuration has been saved. For this, we just have to assign a confirmation variable to Smarty in the isSubmit condition just after the Configuration::updateValue calls.
The Smarty object is available in the $this->context->smarty variable in your module class. We will use the assign method of Smarty:

```php
$this->context->smarty->assign('confirmation', 'ok');
```

This function takes two parameters: the name of the variable in which the value will be stored, and the value.

If you want more information about this, you can check the official Smarty documentation at http://www.smarty.net/docsv2/en/api.assign.tpl.

In the Smarty template, you will now have a variable named $confirmation, which contains the value ok.

We will see later exactly what the Context object contains. It is not important at this point.

If the $confirmation variable has been assigned to Smarty, then we display a confirmation message in getContent.tpl:

```html
{if isset($confirmation)}
  <div class="alert alert-success">Settings updated</div>
{/if}
```

You can add these lines wherever you want in the template. However, you should add it just before the fieldset since this is a PrestaShop standard.

Now, you can submit the form again. If you did it right, the confirmation message should appear, as shown in the following screenshot:

For More Information:
www.packtpub.com/web-development/prestashop-module-development
Creating a New Module

We've almost finished the first version of the configuration form. However, you might have noticed that if you close the configuration form and reopen it, the options you set to Yes won't be preselected correctly. This will give the impression to the merchant that their configuration has not been saved properly.

To fix this problem, we just have to retrieve the configuration values and assign them to Smarty. We will use another function from the Configuration class: Configuration::get.

This function takes the key of the configuration wanted as a parameter, and returns the associated value:

```php
$enable_grades = Configuration::get('MYMOD_GRADES');
$enable_comments = Configuration::get('MYMOD_COMMENTS');
```

Then, to assign these variables to Smarty, we will use the same method we saw previously to set the confirmation message flag:

```php
$this->context->smarty->assign('enable_grades', $enable_grades);
$this->context->smarty->assign('enable_comments', $enable_comments);
```

To keep a short getContent method, my advice is to create a new method that we will call assignConfiguration, and write the code in it. So, at the end, we will have something like this:

```php
public function assignConfiguration()
{
    $enable_grades = Configuration::get('MYMOD_GRADES');
    $enable_comments = Configuration::get('MYMOD_COMMENTS');
    $this->context->smarty->assign('enable_grades', $enable_grades);
    $this->context->smarty->assign('enable_comments', $enable_comments);
}
```

```php
public function getContent()
{
    $this->processConfiguration();
    $this->assignConfiguration();
    return $this->display($__FILE__, 'getContent.tpl');
}
```
We just have to add some Smarty code in `getContent.tpl` to make it work. I will take the Enable grades option as an example, but it will be the same for Enable comments. If the Yes option is enabled, we will precheck the radio button whose value is 1. If the option value is not set or is set to 0, we will precheck the other radio button. The following is the Smarty code:

```html
<img src="../img/admin/enabled.gif" alt="" />
<input type="radio" id="enable_grades_1" name="enable_grades"
value="1" {if $enable_grades eq '1'}checked{/if} />
<label class="t" for="enable_grades_1">Yes</label>
<img src="../img/admin/disabled.gif" alt="" />
<input type="radio" id="enable_grades_0" name="enable_grades"
value="0" {if empty($enable_grades) || $enable_grades eq
'0'}checked{/if} />
<label class="t" for="enable_grades_0">No</label>
```

If we close the configuration form and reopen it, you will now see that the configuration values are prechecked correctly.

Congratulations, you finished the first step of your module!

If you don't know Smarty well yet, take time to read the official documentation to learn about functions of the Smarty object and template syntax. This will save you time for future developments.

**Summary**

In this chapter, we saw how to create the Bootstrap of an installable module with just one file. We also learned how to use Smarty templates in a module by building a small configuration form, and how to register merchant choices in the configuration table of the PrestaShop database.

In the next chapter, we will talk about the concept of hooks: what they're for and how to use them. We will also build the first interaction between the module and the front office.

For More Information:

Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.