Estándares de Codificación

Estándares de Codificación

La consistencia es importante, más aún cuando se trata de escribir código abierto (open source), puesto que el código pertenece a millones de globos oculares, y la corrección de errores recae sobre éstos a la hora de localizar todos los errores actualmente presentes y entender cómo resolverlos.

Por esta razón, al crear cualquier cosa para PrestaShop, ya sea un tema, un módulo o un parche para el núcleo o corazón del software, debe esforzarse por seguir las siguientes pautas. They are the ones that the PrestaShop developers adhere to, and following them is the surest way to have your code be elegantly integrated in PrestaShop.

In short, having code consistency helps keeping the code readable and maintainable.

If use an IDE, you can use the CodeSniffer code validator to help you write better code.

Starting with version 1.6.1.0, the PrestaShop Core codebase has switched to the PSR-1 coding standard and PSR-2 coding style guide. See the reasons why on the announcement article on the Build PrestaShop deblog.

Existing modules and themes are not required to switch to PSR-1 and PSR-2. PrestaShop's own modules and any newly-created community module are expected to adopt these guidelines.

If you want to update your PHP code to the PSR-1 and PSR-2 guidelines, you can use the PHP Coding Standards Fixer, which fixes most issues automatically.

For reference's sake, the old PrestaShop coding standards is kept in this page: Pre-1.6.1.0 PHP Coding Standards. Please do not use it.

Table of contents

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

PSR-1 - Basic Coding Standard

1. Overview

- Files MUST use only `<?php` and `<?=` tags.
- Files MUST use only UTF-8 without BOM for PHP code.
- Files SHOULD *either* declare symbols (classes, functions, constants, etc.) *or* cause side-effects (e.g. generate output, change .ini settings, etc.) but SHOULD NOT do both.
- Namespaces and classes MUST follow an "autoloading" PSR: [PSR-0, PSR-4].
- Class names MUST be declared in `StudlyCaps`.

- Class constants MUST be declared in all upper case with underscore separators.
- Method names MUST be declared in `camelCase`.

2. Files

2.1. PHP Tags

PHP code MUST use the long `<?php ?>` tags or the short-echo `<?= ?>` tags; it MUST NOT use the other tag variations.

2.2. Character Encoding

PHP code MUST use only UTF-8 without BOM.

2.3. Side Effects

A file SHOULD declare new symbols (classes, functions, constants, etc.) and cause no other side effects, or it SHOULD execute logic with side effects, but SHOULD NOT do both.

The phrase "side effects" means execution of logic not directly related to declaring classes, functions, constants, etc., *merely from including the file*.

"Side effects" include but are not limited to: generating output, explicit use of `require` or `include`, connecting to external services, modifying ini settings, emitting errors or exceptions, modifying global or static variables, reading from or writing to a file, and so on.

The following is an example of a file with both declarations and side effects; i.e, an example of what to avoid:

```
<?php
// side effect: change ini settings
ini_set('error_reporting', E_ALL);

// side effect: loads a file
include "file.php";

// side effect: generates output
echo "<html>\n";

// declaration
function foo()
{
    // function body
}
```

The following example is of a file that contains declarations without side effects; i.e., an example of what to emulate:

```
<?php
// declaration
function foo()
{
    // function body
}

// conditional declaration is *not* a side effect
if (! function_exists('bar')) {
    function bar()
    {
        // function body
    }
}
```

3. Namespace and Class Names

Namespaces and classes MUST follow an "autoloading" PSR: [PSR-0, PSR-4].

This means each class is in a file by itself, and is in a namespace of at least one level: a top-level vendor name.

Class names MUST be declared in `StudlyCaps`.

Code written for PHP 5.3 and after MUST use formal namespaces.

For example:

```
<?php
// PHP 5.3 and later:
namespace Vendor\Model;

class Foo
{
}
```

Code written for 5.2.x and before SHOULD use the pseudo-namespacing convention of `Vendor_` prefixes on class names.

```
<?php
// PHP 5.2.x and earlier:
class Vendor_Model_Foo
{
}
```

4. Class Constants, Properties, and Methods

The term "class" refers to all classes, interfaces, and traits.

4.1. Constants

Class constants MUST be declared in all upper case with underscore separators. For example:

```
<?php
namespace Vendor\Model;

class Foo
{
    const VERSION = '1.0';
    const DATE_APPROVED = '2012-06-01';
}
```

4.2. Properties

This guide intentionally avoids any recommendation regarding the use of `$StudlyCaps`, `$camelCase`, or `$under_score` property names.

Whatever naming convention is used SHOULD be applied consistently within a reasonable scope. That scope may be vendor-level, package-level, class-level, or method-level.

4.3. Methods

Method names MUST be declared in `camelCase()`.

PSR-2 - Coding Style Guide

1. Overview

- Code MUST follow a "coding style guide" PSR [PSR-1].
- Code MUST use 4 spaces for indenting, not tabs.
- There MUST NOT be a hard limit on line length; the soft limit MUST be 120 characters; lines SHOULD be 80 characters or less.
- There MUST be one blank line after the `namespace` declaration, and there MUST be one blank line after the block of `use` declarations.
- Opening braces for classes MUST go on the next line, and closing braces MUST go on the next line after the body.
- Opening braces for methods MUST go on the next line, and closing braces MUST go on the next line after the body.
- Visibility MUST be declared on all properties and methods; `abstract` and `final` MUST be declared before the visibility; `static` MUST be declared after the visibility.
- Control structure keywords MUST have one space after them; method and function calls MUST NOT.
- Opening braces for control structures MUST go on the same line, and closing braces MUST go on the next line after the body.
- Opening parentheses for control structures MUST NOT have a space after them, and closing parentheses for control structures MUST NOT have a space before.

1.1. Example

This example encompasses some of the rules below as a quick overview:

```php
<?php
namespace Vendor\Package;

use FooInterface;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;

class Foo extends Bar implements FooInterface
{
    public function sampleFunction($a, $b = null)
    {
        if ($a === $b) {
            bar();
        } elseif ($a > $b) {
            $foo->bar($arg1);
        } else {
            BazClass::bar($arg2, $arg3);
        }
    }

    final public static function bar()
    {
        // method body
    }
}
```

2. General

2.1 Basic Coding Standard

Code MUST follow all rules outlined in PSR-1.

2.2 Files

All PHP files MUST use the Unix LF (linefeed) line ending.

All PHP files MUST end with a single blank line.

The closing ?> tag MUST be omitted from files containing only PHP.

2.3. Lines

There MUST NOT be a hard limit on line length.

The soft limit on line length MUST be 120 characters; automated style checkers MUST warn but MUST NOT error at the soft limit.

Lines SHOULD NOT be longer than 80 characters; lines longer than that SHOULD be split into multiple subsequent lines of no more than 80 characters each.

There MUST NOT be trailing whitespace at the end of non-blank lines.

Blank lines MAY be added to improve readability and to indicate related blocks of code.

There MUST NOT be more than one statement per line.

2.4. Indenting

Code MUST use an indent of 4 spaces, and MUST NOT use tabs for indenting.

⚠ Using only spaces, and not mixing spaces with tabs, helps to avoid problems with diffs, patches, history, and annotations. The use of spaces also makes it easy to insert fine-grained sub-indentation for inter-line alignment.

2.5. Keywords and True/False/Null

PHP keywords MUST be in lower case.

The PHP constants `true`, `false`, and `null` MUST be in lower case.

3. Namespace and Use Declarations

When present, there MUST be one blank line after the `namespace` declaration.

When present, all `use` declarations MUST go after the `namespace` declaration.

There MUST be one `use` keyword per declaration.

There MUST be one blank line after the `use` block.

For example:

```
<?php
namespace Vendor\Package;

use FooClass;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;

// ... additional PHP code ...
```

4. Classes, Properties, and Methods

The term "class" refers to all classes, interfaces, and traits.

4.1. Extends and Implements

The `extends` and `implements` keywords MUST be declared on the same line as the class name.

The opening brace for the class MUST go on its own line; the closing brace for the class MUST go on the next line after the body.

```
<?php
namespace Vendor\Package;

use FooClass;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;

class ClassName extends ParentClass implements \ArrayAccess, \Countable
{
    // constants, properties, methods
}
```

Lists of `implements` MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one interface per line.

```
<?php
namespace Vendor\Package;

use FooClass;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;

class ClassName extends ParentClass implements
    \ArrayAccess,
    \Countable,
    \Serializable
{
    // constants, properties, methods
}
```

4.2. Properties

Visibility MUST be declared on all properties.

The `var` keyword MUST NOT be used to declare a property.

There MUST NOT be more than one property declared per statement.

Property names SHOULD NOT be prefixed with a single underscore to indicate protected or private visibility.

A property declaration looks like the following.

```
<?php
namespace Vendor\Package;

class ClassName
{
    public $foo = null;
}
```

4.3. Methods

Visibility MUST be declared on all methods.

Method names SHOULD NOT be prefixed with a single underscore to indicate protected or private visibility.

Method names MUST NOT be declared with a space after the method name. The opening brace MUST go on its own line, and the closing brace MUST go on the next line following the body. There MUST NOT be a space after the opening parenthesis, and there MUST NOT be a space before the closing parenthesis.

A method declaration looks like the following. Note the placement of parentheses, commas, spaces, and braces:

```
<?php
namespace Vendor\Package;

class ClassName
{
    public function fooBarBaz($arg1, &$arg2, $arg3 = [])
    {
        // method body
    }
}
```

4.4. Method Arguments

In the argument list, there MUST NOT be a space before each comma, and there MUST be one space after each comma.

Method arguments with default values MUST go at the end of the argument list.

```
<?php
namespace Vendor\Package;

class ClassName
{
    public function foo($arg1, &$arg2, $arg3 = [])
    {
        // method body
    }
}
```

Argument lists MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one argument per line.

When the argument list is split across multiple lines, the closing parenthesis and opening brace MUST be placed together on their own line with one space between them.

```
<?php
namespace Vendor\Package;

class ClassName
{
    public function aVeryLongMethodName(
        ClassTypeHint $arg1,
        &$arg2,
        array $arg3 = []
    ) {
        // method body
    }
}
```

4.5. `abstract`, `final`, and `static`

When present, the `abstract` and `final` declarations MUST precede the visibility declaration.

When present, the `static` declaration MUST come after the visibility declaration.

```
<?php
namespace Vendor\Package;

abstract class ClassName
{
    protected static $foo;

    abstract protected function zim();

    final public static function bar()
    {
        // method body
    }
}
```

4.6. Method and Function Calls

When making a method or function call, there MUST NOT be a space between the method or function name and the opening parenthesis, there MUST NOT be a space after the opening parenthesis, and there MUST NOT be a space before the closing parenthesis. In the argument list, there MUST NOT be a space before each comma, and there MUST be one space after each comma.

```
<?php
bar();
$foo->bar($arg1);
Foo::bar($arg2, $arg3);
```

Argument lists MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one argument per line.

```
<?php
$foo->bar(
    $longArgument,
    $longerArgument,
    $muchLongerArgument
);
```

5. Control Structures

The general style rules for control structures are as follows:

- There MUST be one space after the control structure keyword
- There MUST NOT be a space after the opening parenthesis
- There MUST NOT be a space before the closing parenthesis
- There MUST be one space between the closing parenthesis and the opening brace
- The structure body MUST be indented once
- The closing brace MUST be on the next line after the body

The body of each structure MUST be enclosed by braces. This standardizes how the structures look, and reduces the likelihood of introducing errors as new lines get added to the body.

5.1. `if`, `elseif`, `else`

An `if` structure looks like the following. Note the placement of parentheses, spaces, and braces; and that `else` and `elseif` are on the same line as the closing brace from the earlier body.

```
<?php
if ($expr1) {
    // if body
} elseif ($expr2) {
    // elseif body
} else {
    // else body;
}
```

The keyword `elseif` SHOULD be used instead of `else if` so that all control keywords look like single words.

5.2. `switch`, `case`

A `switch` structure looks like the following. Note the placement of parentheses, spaces, and braces. The `case` statement MUST be indented once from `switch`, and the `break` keyword (or other terminating keyword) MUST be indented at the same level as the `case` body. There MUST be a comment such as `// no break` when fall-through is intentional in a non-empty `case` body.

```
<?php
switch ($expr) {
    case 0:
        echo 'First case, with a break';
            break;
    case 1:
        echo 'Second case, which falls through';
                // no break
    case 2:
    case 3:
    case 4:
        echo 'Third case, return instead of break';
        return;
    default:
        echo 'Default case';
                break;
}
```

5.3. `while`, `do while`

A `while` statement looks like the following. Note the placement of parentheses, spaces, and braces.

```
<?php
while ($expr) {
    // structure body
}
```

Similarly, a `do while` statement looks like the following. Note the placement of parentheses, spaces, and braces.

```
<?php
do {
    // structure body;
} while ($expr);
```

5.4. `for`

A `for` statement looks like the following. Note the placement of parentheses, spaces, and braces.

```php
<?php
for ($i = 0; $i < 10; $i++) {
    // for body
}
```

5.5. `foreach`

A `foreach` statement looks like the following. Note the placement of parentheses, spaces, and braces.

```php
<?php
foreach ($iterable as $key => $value) {
    // foreach body
}
```

5.6. `try`, `catch`

A `try catch` block looks like the following. Note the placement of parentheses, spaces, and braces.

```php
<?php
try {
    // try body
} catch (FirstExceptionType $e) {
    // catch body
} catch (OtherExceptionType $e) {
    // catch body
}
```

6. Closures

Closures MUST be declared with a space after the `function` keyword, and a space before and after the `use` keyword.

The opening brace MUST go on the same line, and the closing brace MUST go on the next line following the body.

There MUST NOT be a space after the opening parenthesis of the argument list or variable list, and there MUST NOT be a space before the closing parenthesis of the argument list or variable list.

In the argument list and variable list, there MUST NOT be a space before each comma, and there MUST be one space after each comma.

Closure arguments with default values MUST go at the end of the argument list.

A closure declaration looks like the following. Note the placement of parentheses, commas, spaces, and braces:

```php
<?php
$closureWithArgs = function ($arg1, $arg2) {
    // body
};

$closureWithArgsAndVars = function ($arg1, $arg2) use ($var1, $var2) {
    // body
};
```

Argument lists and variable lists MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one argument or variable per line.

When the ending list (whether or arguments or variables) is split across multiple lines, the closing parenthesis and opening brace MUST be placed together on their own line with one space between them.

The following are examples of closures with and without argument lists and variable lists split across multiple lines.

```
<?php
$longArgs_noVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
) {
    // body
};

$noArgs_longVars = function () use (
    $longVar1,
    $longerVar2,
    $muchLongerVar3
) {
    // body
};

$longArgs_longVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
) use (
    $longVar1,
    $longerVar2,
    $muchLongerVar3
) {
            // body
};

$longArgs_shortVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
) use ($var1) {
    // body
};

$shortArgs_longVars = function ($arg) use (
    $longVar1,
    $longerVar2,
    $muchLongerVar3
) {
    // body
};
```

Note that the formatting rules also apply when the closure is used directly in a function or method call as an argument.

```
<?php
$foo->bar(
    $arg1,
    function ($arg2) use ($var1) {
        // body
    },
    $arg3
);
```

SQL Guidelines

Table names

1. Table names must begin with the PrestaShop "`_DB_PREFIX_`" prefix.

   ```
   ... FROM `'. _DB_PREFIX_.'customer` ...
   ```

2. Table names must have the same name as the object they reflect: "`ps_cart`".
3. Table names have to stay singular: "`ps_order`".
4. Language data have to be stored in a table named exactly like the object's table, and with the "`_lang`" suffix: "`ps_product_lang`".

SQL query

1. Keywords must be written in uppercase.

   ```
   SELECT `firstname`
   FROM `'._DB_PREFIX_.'customer`
   ```

2. Back quotes ("`") must be used around SQL field names and table names.

   ```
   SELECT p.`foo`, c.`bar`
   FROM `'._DB_PREFIX_.'product` p, `'._DB_PREFIX_.'customer` c
   ```

3. Table aliases have to be named by taking the first letter of each word, and must be lowercase.

   ```
   SELECT p.`id_product`, pl.`name`
   FROM `'._DB_PREFIX_.'product` p
   NATURAL JOIN `'._DB_PREFIX_.'product_lang` pl
   ```

4. When conflicts between table aliases occur, the second character has to be also used in the name.

   ```
   SELECT ca.`id_product`, cu.`firstname`
   FROM `'._DB_PREFIX_.'cart` ca, `'._DB_PREFIX_.'customer` cu
   ```

5. A new line has to be created for each clause.

   ```
   $query = 'SELECT pl.`name`
   FROM `'._DB_PREFIX_.'product_lang` pl
   WHERE pl.`id_product` = 17';
   ```

6. It is forbidden to make a `JOIN` in a `WHERE` clause.

About the code validator (PHP CodeSniffer)

The CodeSniffer configuration file is not yet available. Thank you for your patience!