

Invision Power Services Coding Standards

Version 4.0

Terminology

Throughout this document, the following terms will be used:

MUST / MUST NOT

Indicates that the practice is a requirement and failure to do this shall be considered to be a bug. In instances where there is a valid technical reason for it not being possible to do so, this must be discussed internally between all developers and documented as a comment by the relevant code.

SHOULD / SHOULD NOT

Indicates that the practice is a strong recommendation. In instances where there is a valid technical reason for it not being possible or feasible due to other considerations, this should be documented with clear reasoning as a comment by the relevant code.

MAY

Indicates that a practice is allowed, but optional, and there is no preference as to doing so or not.

Third party libraries included in our projects are exempt from all standards.

Your Editor

1. You **MUST** set your editor settings to the following values and ensure that all committed code is in line with these settings:
 - 1.1. Text encoding is UTF-8
 - 1.2. All line endings are Unix style (LF)
 - 1.3. Tabs are used to indent code, not spaces

General Standards

1. All pages **MUST** conform to proper HTTP standards, meaning they **MUST**:
 1. Send an appropriate HTTP response code.
 2. Send a HTTP header specifying an appropriate content-type.
 3. Send other HTTP headers as necessary depending on the content being displayed (for example, for pages which are for file downloading, Content-Length will be sent).
2. All functionality **MUST** work on the following desktop browsers:
 1. Internet Explorer 9 and above.
 2. Firefox 4 and above.
 3. Google Chrome, all versions.
 4. Safari 4 and above.
 5. Opera 10 and above.
3. All functionality **SHOULD** work on older versions of the above versions, though a lesser user experience is acceptable.
4. All functionality **MUST** work for any resolution above the following values wide, assuming the browser is using that entire width:
 1. For front-end user functionality, 320px
 2. For other functionality (control panels, install scripts, etc.), 1280px.

Third-Party Libraries

Third-party libraries included with our projects:

1. **MUST** be approved by management to ensure their licensing terms are conformed with.
2. **MUST** be acknowledged in the Credits.txt file (even if their licensing terms do not require it).
3. **MUST NOT** be modified, except through appropriate plugin systems.
4. **MUST** be updated with security issue fixes (and such instances shall be treated as per the same procedures if a security vulnerability is found in our own code).
5. **SHOULD** be updated with bug fixes.
6. To ensure that rules 4 and 5 are adhered to, where the library developer provides a mailing list to notify of new releases, our “all developer” email address shall be subscribed.

HTML Standards

1. HTML **MUST** be valid HTML5. Meaning they **MUST**:
 1. Specify a HTML5 Doctype.
 2. Validate as valid markup according to the W3C Markup Validation Service.
 3. Use appropriate semantic elements as appropriate (e.g. <header>, <footer>, <video>, <time>, etc.)
 4. Not use self closing tags (i.e.
 not
)
2. HTML **MUST** specify the UTF-8 character set.
3. HTML **MUST NOT** include inline CSS or JavaScript, meaning they **MUST NOT** include:
 1. <style> elements
 2. Elements with *style* attributes
 3. <script> elements (other than those with a *src* attribute to point to an external .js file, or as per JavaScript standards rule 1)
 4. Elements with any JavaScript attributes, such as *onclick*, *onload*, etc.
4. Rich snippets in the form of microdata **SHOULD** be used whenever appropriate - see <http://support.google.com/webmasters/bin/answer.py?hl=en&answer=99170>

CSS Standards

1. CSS **MUST** be included in .css files (see HTML Standards rule 2)
2. CSS **MUST** be valid CSS3 and validate as such according to the W3C CSS Validation Service.
3. CSS **SHOULD** be used instead of JavaScript when there is an option as to which technology can be used (for example, for animations) (see Javascript Standards rule 2).
4. CSS styles **SHOULD** be placed in modules grouping similar styles.
5. Modules **MUST** be named using camelCase.
6. All classes and elements **SHOULD** be named thus:
 - One of the following prefixes:
 - “ips” for framework classes and elements.
 - “c” for other classes.
 - “el” for other elements.
 - The name of the module.
 - Optionally, an underscore and an additional description named using camelCase.Examples:
 - .ipsMenu
 - .ipsTabs_item
 - #ipsLayout_mainArea
 - .ipsComment_hasChildren
 - .cUserLink
 - #elNewTopic
7. When defining classes, curly braces **SHOULD** be on the same line as the selector.
8. When defining classes, related styles **MAY** be indented like so:

```
#somethingSomething {
    background: #fff;
    color: #000;
}

#somethingSomething > li {
    background: #000;
    color: #fff;
}
```

JavaScript Standards

1. General

1. JavaScript **MUST** be included in .js files (see HTML Standards rule 2) except a single block, within the HTML <head> element, to set variables specific to that session.
2. JavaScript **SHOULD NOT** be used instead of CSS when there is an option as to which technology can be used (for example, for animations) (see CSS Standards rule 3).
3. All JavaScript code **SHOULD** be written using the jQuery library.
4. All JavaScript functionality **SHOULD** have appropriate fallbacks for users with JavaScript disabled.
5. When implementing fallbacks for users without JavaScript with regards to AJAX functionality, the same methods in the PHP code **SHOULD** be used, distinguishing between the types by examining the HTTP Accept header sent by the client.
6. JavaScript code **SHOULD NOT** log anything to the console, unless a variable to enable logging has been specifically enabled.
7. Variables **MUST NOT** be leaked into the global scope, unless there is a specific need (therefore, variables **MUST** be declared with the var statement).

1. Syntax

9. Each script file **MUST** be entirely wrapped thus:

```
;( function($, _, undefined){  
  "use strict";  
  // Code goes here  
}(jQuery, _));
```

2. JavaScript **SHOULD** be documented using the JSDoc syntax (<http://usejsdoc.org>). Docblock comments **SHOULD** only be used for JSDoc comments; single-line comments (//) should be used for other purposes.
3. Semi-colons **SHOULD** be used at the end of every statement, even when functions/ object literals are being assigned.
4. When assigning values in an object, trailing commas **MUST NOT** be used.
5. `ips.getAjax()(url)` **SHOULD** be used over `$.ajax` and callbacks **SHOULD** be assigned using chained methods over parameters in the data object.

Example:

```
ips.getAjax()( url )  
  .done( function (response) {  
    // Do something with the response  
  })  
  .fail( function () {  
    // There's an error  
  });
```

6. All variable, method, property, etc. names **SHOULD** be named using camelCase.
7. Lines **SHOULD** be shorter than 120 characters, splitting immediately after a comma or operator, indented appropriately.
8. Curly braces **SHOULD** be on the same line as the statement (not a new line).
9. The left parenthesis **SHOULD** sit immediately next to the statement except where a function is being declared.

```
if(...  
for(...  
function test (...
```


10. Spaces should surround all parameters inside parenthesis - except when the parameter is a simple string.

```
$( 'foo' );  
$( var );  
$( 'foo' + 'bar' );
```

11. When using daisy-chaining, each method **SHOULD** be placed on it's own line, indented appropriately.

Example:

```
$(...)  
    .find(...)  
      .hide()  
    .end()  
    .find(...)  
      .css(...)  
    .end()  
    .append $(...).attr( {  
      ...  
    } );
```

PHP Standards

1. General

1. PHP files **MUST** start with `<?php` on line 1 (short tags **MUST NOT** be used) unless the code is intended to be eval'd rather than included, in which case it **SHOULD** start with `//<?php` (so that text editors will use syntax highlighting properly).
2. PHP files **MUST NOT** end with a closing `?>`
3. PHP files **MUST NOT** contain HTML, CSS or JavaScript code.
4. All used internal PHP functions, classes and features **MUST** be available on a default, non-customised version of PHP 5.3.0 except for features specifically only available in developer mode. Extensions which are enabled by default but may be disabled may be used, however, extensions which need to be enabled when configuring PHP **MUST NOT** be used, except the following extensions which are part of our requirements.
 - 4.1. gd
 - 4.2. mbstring
5. PHP code **SHOULD NOT** raise any errors, including notices. It is recommended that you set error reporting to `E_ALL`.
6. The following functions **MUST NOT** be used, except for the purposes of upgrading data from older versions:
 - 6.1. `escapeshellarg`
 - 6.2. `escapeshellcmd`
 - 6.3. `exec`
 - 6.4. `ini_alter`
 - 6.5. `parse_ini_file`
 - 6.6. `passthru`
 - 6.7. `pcntl_exec`
 - 6.8. `popen`
 - 6.9. `proc_*`
 - 6.10. `serialize` (use a safer encoding function like `json_encode/decode` instead)
 - 6.11. `shell_exec`
 - 6.12. `show_source`
 - 6.13. `symlink`
 - 6.14. `system`
 - 6.15. `unserialize` (use a safer encoding function like `json_encode/decode` instead)
7. Multibyte functions **MUST** be used instead of normal string functions where it is possible that the string will contain multibyte characters.
8. Whenever curly braces are used, the indentation of the code **SHOULD** increase by one tab:

```
// ...

if ( foo )
{
    // ...
}

// ...
```

2. Classes

1. Classes **MUST** be located in their own file - only one class per file is permitted.
2. Classes **MUST** be located and named appropriately so as to be autoloaded. This means:
 - In the /system directory for core framework classes.
 - In the /sources directory of an application for classes belonging to that application.
 - In the /modules_* directories of an application for controllers.
 - In the /extensions directory of an application for extensions.
3. Classes **MUST** be prefixed with an underscore so they can be extended by the hooks system.
4. Classes **SHOULD** be declared abstract where appropriate.
5. Classes **MUST** only be named with alphabetic characters, using PascalCase. Class names **MUST NOT** contain numbers or underscores (other than the initial underscore).
6. Classes **MUST** contain at least one non-static method, unless it is defined as abstract. Classes **MUST NOT** be used as generic containers for methods (such as with the *IPSLib* class in IP.Board 3).
7. Classes **MUST** contain at least one non-abstract method (otherwise an interface should be used).
8. The class declaration **SHOULD** be all on one line, except in cases where the declaration is very long, in which case, linebreaks may be used to separate the parent class and each interface, using tabs to indent like so:

```
class _MyClass
    extends someOtherClass
    implements      anInterface,
                   anOtherInterface
{
```

9. The curly braces **SHOULD** be defined on their own lines, with the starting brace on the line immediately following the declaration, except if the class has no contents, in which case they may be on the same line separated by a space:

```
class _MyEmptyClass { }
```

Example declaration:

```
abstract class _MyClass extends \IPS\Foo\Bar
{
    //...
}
```

3. Functions and Methods

1. Functions Methods **MUST** only be named with alphabetic characters (and underscores as per rules 2 and 3), using camelCase.
2. Function and Method names **MUST NOT** start with a double underscore except for magic methods.
3. Function and Method names **MUST NOT** contain an underscore except for:
 - Getters and setters which should be named get_*() and set_*() and may contain underscores if the property name contains underscores (for example, if the class is an Active Record, the property names will contain underscores as per SQL Standards).

- Methods within controllers which should not be executed automatically if the “do” parameter matches the method names. In this case, the method name should begin with a single underscore.
4. Methods **MUST** declare their visibility, but **MUST NOT** be declared private as it interferes with the usage of hooks.
 5. Abstract methods **SHOULD** be declared where appropriate.
 6. If it is desired for a hook to not be able to override a method, it **SHOULD** be declared final.
 7. As with classes, the curly braces **SHOULD** be defined on their own lines, with the starting brace on the line immediately following the declaration, except if the function or method has no contents, in which case they may be on the same line separated by a space, or if the function or method has lots of arguments as per rule 9.7.
 8. Pass-by-reference is allowed in the method declaration, but call-time pass-by-reference **MUST NOT** be used.
 9. Arguments **SHOULD** be declared with:
 - 9.1. One space (or linebreak and tabs as per rule 9.7) after the opening parenthesis, unless there are no arguments.
 - 9.2. No whitespace before the opening parenthesis.
 - 9.3. Once space (or linebreak and tabs as per rule 9.7) before the closing parenthesis, unless there are no arguments.
 - 9.4. One space (or linebreak and tabs as per rule 9.7) after each comma.
 - 9.5. No whitespace between the argument name, the assignment operator and the default value.
 - 9.6. No whitespace between the pass-by-reference operator and the argument.


```
public function myFunction( $foo, &$bar, $baz=NULL )
public function myFunction()
```
 - 9.7. For methods that accept a large number of arguments, the declaration **SHOULD** be broken into multiple lines, putting each argument on it's own line. In this case the curly brace **SHOULD** go on the same line as the closing parenthesis:


```
public function myFunction(
    $foo,
    &$bar,
    $baz=NULL
) {
```
 10. Type-hinting **SHOULD NOT** be used.

4. Properties and Variables

1. Properties and variables **MUST** only contain alphanumerical characters, (and underscores as per rule 2), using camelCase where appropriate.
2. Properties and variables **MUST NOT** contain underscores, except for properties in a class which uses getter and setter methods (for example, an Active Record), in which case, properties which are outside the scope of the getters and setters may start with an underscore to prevent conflicts.
3. Properties **MUST NOT** be declared private as it interferes with the usage of hooks.
4. When assigning a variable, there **SHOULD** be a single space either side of the assignment operator, or, if defining more than one variable, tabs instead of the space before the assignment operator to align the assignment operator on each line:

```
$var = 'foo';
```

```
$var1      = 'foo';
```

```
$var2      = 'foo';
```

5. When casting a variable, there **SHOULD** be no spaces in the parentheses, and a single space between the case and the variable:

```
$var = (string) $var;
```

6. When casting a variable to a type which can be cast by more than one keyword, the following **SHOULD** be used:
 - 6.1. int, not integer
 - 6.2. bool, not boolean
 - 6.3. float, not double or real
7. Properties **SHOULD** specify a default value (even if it is NULL).

5. Constants

1. Constants (both class constants and namespace constants) **MUST** be in SCREAMING_SNAKE_CASE.
2. Constants **MUST** not be declared in the global namespace.
3. When using PHP's internal constants which can be used in upper or lowercase (for example, NULL, TRUE, FALSE), uppercase **SHOULD** be used.

6. Interfaces

1. Interfaces **SHOULD** only be used when an abstract class is not appropriate.
2. Interfaces **MUST** follow all the same naming conventions as classes.

7. Strings

1. Strings **SHOULD** use single-quotes where there is no variable substitution and the string does not contain apostrophes. It is preferential to use double-quotes when the string contains apostrophes rather than escaping them because it's easier to read.
2. Variable substitution **SHOULD** be done using curly braces around the entire variable:-
Correct:

```
$foo = "bar {$var} baz";
```


Incorrect:

```
$foo = "bar $var baz";  
$foo = "bar ${var} baz";
```
3. Where the string concatenation operator is used, it **SHOULD** be separated with spaces either side:

```
$foo = 'bar' . Class::method() . 'baz';
```

8. Arrays

1. Arrays **SHOULD** be declared with one space (or line-break or tabs as per rule-2):
 - 1.1. After the opening parenthesis, unless the array is empty.
 - 1.2. Before the closing parenthesis, unless the array is empty.
 - 1.3. After each comma.
 - 1.4. Either side of the fat comma.

```
$array = array( 1, 2, 3 );
```

```
$array = array( 1 => 'foo', 2 => 'bar' );
$array = array();
```

2. For long array declarations, a linebreak **SHOULD** be used instead of the space for rules 8.1.1-3, and tabs **SHOULD** be used instead of the space for rule 8.1.4. When using tabs, there **SHOULD** be an appropriate number to align the values, with the indent increasing with each level of depth in a multi-dimensional array. For each array within a multi-dimensional arrays, whether to use the syntax in rules 8.1 or 8.2 is assessed for each individual array.

```
$array = array(
    'foo'      => array(
        'bar'      => array( 'foo', 'bar', 'baz' ),
        'baz'      => array(
            'foo'    => 'foo',
            'bar'    => 'bar',
            'baz'    => 'baz'
        ),
    ),
    'moo'      => array(),
);
```

9. Control Statements

1. Control statements **SHOULD** have a single space (or line-breaks or tabs as per rule 2):
 - 1.1. After the opening parenthesis
 - 1.2. Before the closing parenthesis
 - 1.3. Either side of all operators within the statement

```
if( $foo != $bar and $foo != $baz )
```

2. If the control statement is very large, the control statement **SHOULD** be broken into multiple lines as appropriate for the given circumstance.
3. Curly braces **MUST** be used, even if it is not necessary.
4. As with classes, methods and functions, the curly braces **SHOULD** be defined on their own lines, with the starting brace on the line immediately following the declaration.
5. The identical (===) and not identical operators (!==) **SHOULD** be used instead of the equal (==) and not equal (!=) operators when appropriate.
6. A control statement **SHOULD** be declared in such a way that the precedence of operators is obvious. For example, consider the following statements. Though the first two produce identical results, and the third will produce something different, one may not immediately know that without checking the PHP manual. Therefore, the the second statement should be used instead of the first.

```
if ( $foo == $bar ?: $baz )
if ( $foo == ( $bar ?: $baz ) )
if ( ( $foo == $bar ) ?: $baz )
```

7. The logical operators *and* and *or* **SHOULD** be used instead of && and ||. In circumstances where the precedence of && and || is necessary, it is preferential to use parenthesis to denote the precedence as per rule 6.
8. The shorthand ternary operator **SHOULD** be used when appropriate, in which case there **SHOULD** be no whitespace between the ? and the :.
9. In *switch* statements, the break/return statement **SHOULD** be indented to the same level of the code (one deeper than the case statement). If there is no need for a break or return so as to fall through to the next case, a comment indicating this intention **SHOULD** be added so there is confusion that the lack of it is a mistake.

PHP Documentation

General

1. Tags **SHOULD NOT** be used in the documentation which hint at information already indicated by class or method declarations. This includes, but is not limited to:
 - 1.1. @extends
 - 1.2. @public
 - 1.3. @private
 - 1.4. @protected
 - 1.5. @abstract
 - 1.6. @interface
 - 1.7. @implements
2. Where dates are specified, they **SHOULD** be specified in the format “j M Y” - e.g. “1 Jan 2000”.
3. Markdown **MAY** be used in the descriptions.
4. To reference a webpage, a HTML <a> tag **SHOULD** be used.

File Header

```
/**
 * @brief      Description of file
 * @author     <a href='http://www.invisionpower.com'>Invision
Power Services, Inc.</a>
 * @copyright  (c) 2001 - SVN_YYYY Invision Power Services, Inc.
 * @license    http://www.invisionpower.com/legal/standards/
 * @package    IPS Social Suite
 * @subpackage Application
 * @since      1 Jan 2000
 * @version    SVN_VERSION_NUMBER
 */
```

1. All files **MUST** start with the above documentation block.
2. The documentation block **MUST** be on the line immediately following the opening <?php tag.
3. The @brief tag **MUST** indicate a description of the file.
4. The @since tag **MUST** indicate the date the file was created. If a file is being based on a file from a previous version, that file’s creation date **SHOULD** be used, unless it is not known, in which case the date that the new file is being created **MUST** be used.
5. The @subpackage **MUST** be one of the following values if the file belongs to an application (except the “core” application), and **MUST NOT** if the file belongs to the system framework, or the core application.
 - Blog
 - Board (note not “Forums”)
 - Chat
 - Content
 - Downloads
 - Gallery

- Nexus

Classes and Interfaces

```
/**
 * Class description
 */
```

1. All classes and interfaces **MUST** have a documentation block on the line immediately preceding the declaration.
2. Generally additional tags are not necessary, however, if additional tags are included, the description **MUST** start with the @brief tag.

Properties

```
/**
 * @brief Property description
 */
```

1. All properties **MUST** have a documentation block on the line immediately preceding the declaration containing at least a @brief tag.

Methods

```
/**
 * Method Description
 *
 * @code
     MyClass::myMethod();
 * @endcode
 * @todo      Something we need to do
 * @deprecated
 * @note      Something to note
 * @see       <a href='http://www.invisionpower.com'>External
Link</a>
 * @see       SomeOtherClass::$someProperty
 * @param     string      $var1      Description
 * @param     string|array $var2      Description
 * @param     mixed       $var3      Description
 * @throws    Exception
 * @li ERROR1
 * @li ERROR2
 * @return    void
 */
```

1. All methods **MUST** have a documentation block on the line immediately preceding the declaration containing at least a @return tag.
2. The documentation block **MUST** specify a description, followed by a blank line, then the documentation tags. The description **MAY** span more than one line.

3. The `@code` tag **SHOULD** be used to provide a code example of how to use the method if the documentation block would not otherwise provide enough information, for example, if a complex array is passed as a parameter. When used, the `*` is not used at the beginning of the line.
4. The `@deprecated` tag **SHOULD** be used to mark deprecated methods.
5. The `@note` and `@see` tags **MAY** be used to specify additional notes. The `@see` tag can either reference an other classes, property or method, or a webpage.
6. The `@param` tags **MUST** be in the format specified in the above example.
 - 6.1. The type **MUST** be specified as either:
 - 6.1.1. A single type, if only a single type should be used.
 - 6.1.2. Multiple types, separated by a pipe, if multiple types (but not all types) may be used.
 - 6.1.3. The keyword “mixed” if multiple types (including, but not necessarily all types) may be used and the list would be too long for separating each type by a pipe would be feasible.
 - 6.2. The pseudo-type “number” **MAY** be used in place of “int|float”.
 - 6.3. The type **SHOULD** use the same formatting as PHP Standards Rule 4.6 for specifying a type which has more than one keyword.
 - 6.4. The variable name **MUST** be present, and be as it is defined in the declaration.
 - 6.5. The description **MUST** remain on the same line.
7. If calling the method might call an exception to be thrown by that method, a `@throws` tag **SHOULD** be provided, specifying the name of the exception class which may be thrown. Where appropriate (i.e. the exception is not dynamically generated or returned by an API), a list of messages that may be thrown **SHOULD** be listed using `@li` tags.
8. The `@return` tag **MUST** be specified, even if the method does not return a value, in which case it should specify “void”.

SQL Standards

1. All used SQL code **MUST** work on a default, non-customised version of MySQL 5.0.3.
2. All used SQL code **MUST** work when strict mode is enabled (it is recommended this be enabled in development).
3. Tables and views **MUST** be prefixed with the key of the application they belong to.
4. Table, view, column and index names **MUST** be named using snake_case.
5. New columns **MUST**:
 - 5.1. Be defined using appropriate datatypes with particular considerations to the appropriate size and, for numerical values, if UNSIGNED can be used. Timestamps are exempt from this rule for legacy reasons and must be an integer rather than DATETIME.

Examples of datatypes you might use are listed below. These examples are suggestions of what might be appropriate. In practice, the contents of each column must be considered carefully to decide the best datatype.

- For boolean values: BIT(1)
 - For auto-incrementing ID numbers: BIGINT UNSIGNED
 - For content titles: VARCHAR(255)
 - For content: LONGTEXT
 - For date/times: BIGINT UNSIGNED
 - For currencies: DECIMAL(64, 2)
 - For multiple possible values: ENUM or SET (we currently do things like use TINYINT and accept 0, 1, 2 with different meanings - this is incorrect)
 - For md5 hashes: CHAR(32)
- 5.2. Define whether NULL is permitted or not appropriately (for example, it is not acceptable for a table that stores content to have a NULL value for the title. However, it is permitted for the members table to have a NULL value for the Facebook ID) and make the PHP code use NULL appropriately.
 - 5.3. If NULL is not permitted, define an appropriate default value.
 6. Old columns **SHOULD** be updated to adhere to the standards in rule 5 where it is feasible. If this is done, the upgrader **MUST** update existing installations - it is not acceptable for a new installation and an old upgraded installation to be using different schemas.
 7. Columns should include a comment which explains the usage of that column for the benefit of other developers.
 8. Text columns **MUST** use the UTF-8 collation.