Style Guide

Overview

This document provides guidelines for code formatting and documentation to individuals and teams contributing to HUBzero CMS.

Topics covered:

- PHP File Formatting
- PHP and Database Naming Conventions
- PHP, CSS Coding Style
- PHP Inline Documentation

PHP Coding Styles

Code Demarcation

PHP code must always be delimited by the full-form, standard PHP tags:

```
<?php

?>
```

Short tags are never allowed.

For files that contain only PHP code, the closing tag (”?>”) is never permitted. It is not required by PHP, and omitting it prevents the accidental injection of trailing white space into the response.

Indention

Indentation should consist of 1 tab per indentation level. Spaces are not allowed.

Line Length

The target line length is 120 characters. Longer lines are acceptable as long as readability is maintained.

Line Termination

Line termination follows the Unix text file convention. Lines must end with a single linefeed (LF) character. Linefeed characters are represented as ordinal 10, or hexadecimal 0×0A.

Note: Do not use carriage returns (CR) as is the convention in Apple OS’s (0×0D) or the carriage return – linefeed combination (CRLF) as is standard for the Windows OS (0×0D, 0×0A).

Strings

String Literals

When a string is literal (contains no variable substitutions), the apostrophe or “single quote” should always be used to demarcate the string:

```
$a = 'Example String';
```

String Literals Containing Apostrophes

When a literal string itself contains apostrophes, it is permitted to demarcate the string with quotation marks or “double quotes”. This is especially useful for SQL statements:

```
$sql = "SELECT `id`, `name` from `people` "
     . "WHERE `name`='Fred' OR `name`='Susan'";
```

This syntax is preferred over escaping apostrophes as it is much easier to read.

Variable Substitution

Variable substitution is permitted using either of these forms:

```
$greeting = "Hello $name, welcome back!";

$greeting = "Hello {$name}, welcome back!";
```

For consistency, this form is not permitted:

```
$greeting = "Hello ${name}, welcome back!";
```

String Concatenation

Strings must be concatenated using the “.” operator. A space must always be added before and after the “.” operator to improve readability:

```
$company = 'HUBzero' . ' ' . 'content management system';
```

When concatenating long strings with the “.” operator, it is encouraged to break the statement into multiple lines to improve readability. In these cases, each successive line should be padded with white space such that the “.”; operator is aligned under the “=” operator:

```
$sql = "SELECT `id`, `name` FROM `users` "
     . "WHERE `name` = 'Jim' "
     . "ORDER BY `name` ASC ";
```

Arrays

Numerically Indexed Arrays

Negative numbers are not permitted as indices.

An indexed array may start with any non-negative number, however all base indices besides 0 are discouraged.

When declaring indexed arrays with the Array function, a trailing space must be added after each comma delimiter to improve readability:

```
$sampleArray = array(1, 2, 3, 'HUBzero');
```

It is permitted to declare multi-line indexed arrays using the “array” construct. In this case, each successive line must be indented to the same level as first line and then padded with spaces such that beginning of each line is aligned:

```
$sampleArray = array(1, 2, 3, 'HUBzero',
                     $a, $b, $c,
                     56.44, $d, 500);
```

Alternately, the initial array item may begin on the following line. If so, it should be padded at one indentation level greater than the line containing the array declaration, and all successive lines should have the same indentation; the closing paren should be on a line by itself at the same indentation level as the line containing the array declaration:

```
$sampleArray = array(
    1, 2, 3, 'HUBzero',
    $a, $b, $c,
```

```
    56.44, $d, 500,
);
```

When using this latter declaration, we encourage using a trailing comma for the last item in the array; this minimizes the impact of adding new items on successive lines, and helps to ensure no parse errors occur due to a missing comma.

Associative Arrays

When declaring associative arrays with the Array construct, breaking the statement into multiple lines is encouraged. In this case, each successive line must be padded with white space such that both the keys and the values are aligned:

```
$sampleArray = array('firstKey'  => 'firstValue',
                     'secondKey' => 'secondValue');
```

Alternately, the initial array item may begin on the following line. If so, it should be padded at one indentation level greater than the line containing the array declaration, and all successive lines should have the same indentation; the closing paren should be on a line by itself at the same indentation level as the line containing the array declaration. For readability, the various “=>” assignment operators should be padded such that they align.

```
$sampleArray = array(
    'firstKey'  => 'firstValue',
    'secondKey' => 'secondValue',
);
```

When using this latter declaration, we encourage using a trailing comma for the last item in the array; this minimizes the impact of adding new items on successive lines, and helps to ensure no parse errors occur due to a missing comma.

Classes

- Classes must be named according to HUBzero’s naming conventions.
- The brace should always be written on the line underneath the class name.
- Every class must have a documentation block that conforms to the PHPDocumentor

standard.
- All code in a class must be indented with a single tab.
- Only one class is preferred in each PHP file. Additional classes are permitted but strongly discouraged.
- Placing additional code in class files is permitted but discouraged.

The following is an example of an acceptable class declaration:

```
/**
* Documentation Block Here
*/
class SampleClass
{
    // all contents of class
    // must be indented
}
```

Classes that extend other classes or which implement interfaces should declare their dependencies on the same line when possible.

```
class SampleClass extends FooAbstract implements BarInterface
{
}
```

If as a result of such declarations, readability suffers due to line length, break the line before the “extends” and/or “implements” keywords, and pad those lines by one indentation level.

```
class SampleClass
    extends FooAbstract
    implements BarInterface
{
}
```

If the class implements multiple interfaces and the declaration covers multiple lines, break after each comma separating the interfaces, and indent the interface names such that they align.

```
class SampleClass
```

```
    implements BarInterface,
               BazInterface
{
}
```

Class Member Variables

Member variables must be named according to HUBzero's variable naming conventions.

Any variables declared in a class must be listed at the top of the class, above the declaration of any methods.

The var construct is permitted but discouraged. Member variables should declare their visibility by using one of the private, protected, or public modifiers. Giving access to member variables directly by declaring them as public is permitted but discouraged in favor of accessor methods (set & get).

Functions

Declaration

Functions must be named according to HUBzero's function naming conventions.

Methods inside classes must always declare their visibility by using one of the private, protected, or public modifiers.

As with classes, the brace should always be written on the line underneath the function name. Space between the function name and the opening parenthesis for the arguments is not permitted.

Functions in the global scope are strongly discouraged.

The following is an example of an acceptable function declaration in a class:

```
/**
* Documentation Block Here
*/
class Foo
{
    /**
     * Documentation Block Here
     */
```

```
    public function bar()
    {
        // all contents of function
        // must be indented four spaces
    }
}
```

In cases where the argument list affects readability, you may introduce line breaks. Additional arguments to the function or method must be indented one additional level beyond the function or method declaration. The following is an example of one such situation:

```
/**
* Documentation Block Here
*/
class Foo
{
    /**
     * Documentation Block Here
     */
    public function bar($arg1, $arg2, $arg3,
        $arg4, $arg5, $arg6)
    {
        // all contents of function
        // must be indented four spaces
    }
}
```

Note: Pass-by-reference is the only parameter passing mechanism permitted in a method declaration.

```
/**
* Documentation Block Here
*/
class Foo
{
    /**
     * Documentation Block Here
     */
    public function bar(&$baz)
    {
```

```
    }
}
```

Call-time pass-by-reference is strictly prohibited.

The return value must not be enclosed in parentheses. This can hinder readability, in additional to breaking code if a method is later changed to return by reference.

```
/**
* Documentation Block Here
*/
class Foo
{
    /**
     * WRONG
     */
    public function bar()
    {
        return($this->bar);
    }

    /**
     * RIGHT
     */
    public function bar()
    {
        return $this->bar;
    }
}
```

Function and Method Usage

Function arguments should be separated by a single trailing space after the comma delimiter. The following is an example of an acceptable invocation of a function that takes three arguments:

```
threeArguments(1, 2, 3);
```

Call-time pass-by-reference is strictly prohibited. See the function declarations section for the proper way to pass function arguments by-reference.

In passing arrays as arguments to a function, the function call may include the “array” hint and may be split into multiple lines to improve readability. In such cases, the normal guidelines for writing arrays still apply:

```
threeArguments(array(1, 2, 3), 2, 3);

threeArguments(array(1, 2, 3, 'HUBzero',
                     $a, $b, $c,
                     56.44, $d, 500), 2, 3);

threeArguments(array(
    1, 2, 3, 'HUBzero',
    $a, $b, $c,
    56.44, $d, 500
), 2, 3);
```

Control Statements

If/Else/Elseif

Control statements based on the if and else if constructs must have a single space before the opening parenthesis of the conditional.

Within the conditional statements between the parentheses, operators must be separated by spaces for readability. Inner parentheses are encouraged to improve logical grouping for larger conditional expressions.

The opening brace is written on the line after the conditional statement. The closing brace is always written on its own line. Any content within the braces must be indented using 1 tab.

```
if ($a != 2)
{
    $a = 2;
}
```

If the conditional statement causes the line length to affect readability and has several clauses, you may break the conditional into multiple lines. In such a case, break the line prior to a logic operator, and pad the line such that it aligns under the first character of the conditional clause.

The closing paren in the conditional will then be placed on a line with the opening brace, with one space separating the two, at an indentation level equivalent to the opening control statement.

```
if (($a == $b)
    && ($b == $c)
    || (Foo::CONST == $d))
{
    $a = $d;
}
```

The intention of this latter declaration format is to prevent issues when adding or removing clauses from the conditional during later revisions.

For if statements that include else if or else, the formatting conventions are similar to the if construct. The following examples demonstrate proper formatting for if statements with else and/or {else if constructs:

```
if ($a != 2)
{
    $a = 2;
}
else
{
    $a = 7;
}

if ($a != 2)
{
    $a = 2;
}
elseif ($a == 3)
{
    $a = 4;
}
else
{
    $a = 7;
}

if (($a == $b)
    && ($b == $c)
    || (Foo::CONST == $d))
```

```
{
    $a = $d;
}
elseif (($a != $b)
          || ($b != $c))
{
    $a = $c;
}
else
{
    $a = $b;
}
```

PHP allows statements to be written without braces in some circumstances. This is not permitted; all if, else if or else statements must use braces.

Switch

Control statements written with the switch statement must have a single space before the opening parenthesis of the conditional statement and after the closing parenthesis.

All content within the switch statement must be indented one indention level. Content under each case statement must be indented using an additional indention level.

```
switch ($numPeople)
{
    case 1:
        break;

    case 2:
        break;

    default:
        break;
}
```

The construct default should not be omitted from a switch statement.

Note: It is sometimes useful to write a case statement which falls through to the next case by not including a break or return within that case. To distinguish these cases from bugs, any case

statement where break or return are omitted should contain a comment indicating that the break was intentionally omitted.

Inline Documentation

Format

All documentation blocks (“docblocks”) must be compatible with the phpDocumentor format. Describing the phpDocumentor format is beyond the scope of this document. For more information, visit: [1]

All class files must contain a “file-level” docblock at the top of each file and a “class-level” docblock immediately above each class.

Files

Every file that contains PHP code must have a docblock at the top of the file that contains these phpDocumentor tags at a minimum:

```
/**
 * @package     hubzero-cms
 * @author      Joe Smith <joesmith@hubzero.org>
 * @copyright   Copyright 2005-2011 Purdue University. All rights rese
rved.
 * @license     http://www.gnu.org/licenses/lgpl-3.0.html LGPLv3
 *
 * Copyright 2005-2011 Purdue University. All rights reserved.
 *
 * This file is part of: The HUBzero(R) Platform for Scientific Collab
oration
 *
 * The HUBzero(R) Platform for Scientific Collaboration (HUBzero) is f
ree
 * software: you can redistribute it and/or modify it under the terms
of
 * the GNU Lesser General Public License as published by the Free Soft
ware
 * Foundation, either version 3 of the License, or (at your option) an
y
 * later version.
 *
 * HUBzero is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU Lesser General Public License for more details.
```

```
 *
 * You should have received a copy of the GNU Lesser General Public Li
cense
 * along with this program.  If not, see <http://www.gnu.org/licenses/
>.
 *
 * HUBzero is a registered trademark of Purdue University.
 */
```

Classes

Every class must have a docblock that contains these phpDocumentor tags at a minimum:

```
/**
 * Short description for class
 *
 * Long description for class (if any)...
 *
 * @package     hubzero-cms
 * @subpackage  com_members
 * @copyright   Copyright 2005-2011 Purdue University. All rights rese
rved.
 * @license     http://www.gnu.org/licenses/lgpl-3.0.html LGPLv3
 * @version     Release: @package_version@
 * @since       Class available since Release 1.5.0
 * @deprecated  Class deprecated in Release 2.0.0
 */
```

Functions

Every function, including object methods, must have a docblock that contains at a minimum:

- A description of the function
- All of the arguments
- All of the possible return values

It is not necessary to use the “@access” tag because the access level is already known from the “public”, “private”, or “protected” modifier used to declare the function.

If a function or method may throw an exception, use @throws for all known exception classes:

```
@throws exceptionclass [description]
```

SQL Queries

SQL keywords are to be written in uppercase, while all other identifiers (which the exception of quoted text) is to be in lowercase.

```
$sql = "SELECT `id`, `name` from `people` "
     . "WHERE `name`='Fred' OR `name`='Susan'";
```

PHP Naming Conventions

Classes

HUBzero Library

HUBzero Core Library uses the PSR-0 class naming convention whereby the names of the classes directly map to the directories in which they are stored. The root level directory ofHUBzero's standard library is the "Hubzero/" directory. All HUBzero core library classes are stored hierarchically under these root directories.

Class names may only contain alphanumeric characters. Numbers are permitted in class names but are discouraged in most cases. Underscores are only permitted in place of the path separator; the filename "Hubzero/User/Profile.php" must map to the class name "HubzeroUserProfile".

If a class name is comprised of more than one word, the first letter of each new word must be capitalized. Successive capitalized letters are not allowed, e.g. a class "HubzeroPDF" is not allowed while "HubzeroPdf" is acceptable.

Note: Code that must be deployed alongside Hubzero and Joomla libraries but is not part of the standard or extras libraries (e.g. application code or libraries that are not distributed by Hubzero) must never start with "Hubzero".

Extensions

Classes should be given descriptive names. Avoid using abbreviations where possible. Class names should always begin with an uppercase letter and be written in CamelCase even if using traditionally uppercase acronyms (such as XML, HTML).

Namespaced

While namespacing an extension is not required, it is encouraged.

Controllers

For components, such as the Blog in the Administrator, the convention is Components[Component]Controllers[Name].

```
namespace ComponentsBlogControllers;

use HubzeroComponentAdminController;

class Entries extends AdminController
{
    // Methods
```

```
}
```

Models

The naming convention is Components[Component]Models[Name].

```
namespace ComponentsBlogModels;

use HubzeroBaseModel;

class Entry extends Model
{
    // Methods
}
```

Plugins

Currently, plugin naming must follow the pseudo-namespace conventions.

non/pseudo-Namespaced

These conventions define a pseudo-namespace mechanism for extensions in the framework. Third-party developers are to avoid beginning names with ‘Hubzero’ as it is reserved. It is advisable for developers to name classes with their own unique prefix.

Controllers

For single controller components, the naming convention is [Component]Controller.

```
class ContentController extends HubzeroComponentSiteController
{
    // Methods
}
```

For a multi-controller components, such as the Blog in the Administrator, the convention is [Component]Controller[Name].

```
class BlogControllerEntries extends HubzeroComponentAdminController
```

```
{
    // Methods
}
```

Models

The naming convention is [Component]Model[Name].

```
class BlogModelEntry extends HubzeroBaseModel
{
    // Methods
}
```

Plugins

The naming convention is plg[Folder][Element]

```
class plgContentPagebreak extends HubzeroPluginPlugin
{
    // Methods
}
```

Filenames

Only alphanumeric characters, underscores, and the dash character (”-”) are permitted. Spaces are strictly prohibited.

Any file that contains PHP code should end with the extension “.php”. The following examples show acceptable filenames:

```
Hubzero/Session/Helper.php
```

```
Hubzero/View/Helper/Html.php
```

Controllers

For single controller components, the naming convention of

Components[Component][Client]Controller will map to a file name of controller.php and be located in the component folder.

```
com_content
.. /site
.. .. controller.php
```

For a multi-controller components, the convention of Components[Component][Client]Controllers[Name] will map to files located in a /controllers folder under the component folder. The file names will reflect the name of the controller.

```
com_blog
.. /site
.. .. /controllers
.. .. .. entries.php
.. .. .. media.php
```

Models

The naming convention of [Component]Model[Name] will map to a similar file structure. The files will be located in a /models folder under the component folder. The file names will reflect the name of the model.

```
com_blog
.. /models
.. .. entry.php
.. .. comment.php
```

Layouts

Components may support different Layouts to render the data supplied by a View and its Models. A Layout file usually contains markup and some PHP code for display logic only: no functions, no classes.

A Layout consists of at least one .php file and an equally named .xml manifest file located in the /tmpl/ folder of a View, both reflect the internal name of the Layout. The standard Layout is called “display”.

```
com_blog
.. /site
.. .. /views
.. .. .. /entries
.. .. .. .. /tmpl
.. .. .. .. .. display.php
.. .. .. .. .. display.xml
.. .. .. .. .. edit.php
.. .. .. .. .. edit.xml
.. .. .. .. .. entry.php
.. .. .. .. .. entry.xml
```

Functions and Methods

Function names may only contain alphanumeric characters. Underscores are not permitted except as a prefix to indicate protected or private methods. Numbers are permitted in function names but are discouraged in most cases.

Function names must always start with a lowercase letter. When a function name consists of more than one word, the first letter of each new word must be capitalized. This is commonly called “camelCase” formatting.

Verbosity is generally encouraged. Function names should be as verbose as is practical to fully describe their purpose and behavior.

These are examples of acceptable names for functions:

```
filterInput()

getElementById()

widgetFactory()

_myPrivateMethod()
```

For object-oriented programming, accessors for instance or static variables should always be prefixed with “get” or “set”. In implementing design patterns, such as the singleton or factory patterns, the name of the method should contain the pattern name where practical to more thoroughly describe behavior.

For methods on objects that are declared with the “private” or “protected” modifier, the first

character of the method name must be an underscore. This is the only acceptable application of an underscore in a method name. Methods declared “public” should never contain an underscore.

Functions in the global scope (a.k.a “floating functions”) are permitted but discouraged in most cases. Consider wrapping these functions in a static class.

Variables

Variable names may only contain alphanumeric characters. Underscores and numbers are permitted in variable names but are discouraged in most cases.

For instance variables that are declared with the “private” or “protected” modifier, the first character of the variable name must be a single underscore. Member variables declared “public” should never start with an underscore.

As with function names (see above) variable names must always start with a lowercase letter and follow the “camelCaps” capitalization convention.

Verbosity is generally encouraged. Variables should always be as verbose as practical to describe the data that the developer intends to store in them. Terse variable names such as “$i” and “$n” are discouraged for all but the smallest loop contexts. If a loop contains more than 20 lines of code, the index variables should have more descriptive names.

Names should be descriptive, but concise. We don’t want huge sentences as our variable names, but typing an extra couple of characters is always better than wondering what exactly a certain variable is for.

```
namespace HubzeroBase;

class Example
{
    private $_status = null;

    protected $_fieldName = null;

    protected function _sortNames()
    {
        $someNames = array();
    }
}
```

Constants

Constants may contain both alphanumeric characters and underscores. Numbers are permitted in constant names.

All letters used in a constant name must be capitalized, while all words in a constant name must be separated by underscore characters.

For example, EMBED_SUPPRESS_EMBED_EXCEPTION is permitted but EMBED _SUPPRESSEMBEDEXCEPTION is not.

Prefix constant names with the uppercase name of the class/package they are used in. For example, the constants used by the JError class all begin with “JERROR_”.

Constants must be defined as class members with the “const” modifier. Defining constants in the global scope with the “define” function is permitted but strongly discouraged.

CSS Coding Styles

Terminology

Concise terminology used in these standards:

```
selector {
    property: value;
}
```

Selectors

Selectors should:

- be on a single line
- end in an opening brace
- be closed with a closing brace on a separate line

A blank line should be placed between each group, section, or block of multiple selectors of logically related styles.

Were appropriate, blocks of related styles should be commented to facilitate understanding of their use.

```
/* Book Navigation */
    .book-navigation .page-next {
    }
    .book-navigation .page-previous {
    }

/* Book Forms */
    .book-admin-form {
        border: 1px solid #000;
    }
```

Note: Indentation is optional but encouraged when commenting blocks of related styles.

Multiple selectors

Multiple selectors should each be on a single line, with no space after each comma:

```
#forum td.posts,
#forum td.topics,
#forum td.replies,
#forum td.pager {
}
```

Properties

All properties should be on the following line after the opening brace. Each property should:

- be on its own line
- be indented one tab relative to the selector line
- have a colon immediately after (no spaces permitted) the property name
- have a single space after the property and before the property value
- end in a semi-colon

```
#forum .description {
    color: #EFEFEF;
    font-size: 0.9em;
    margin: 0.5em;
}
```

Multiple values

Where properties can have multiple values, each value should be separated with a space.

```
font-family: helvetica, sans-serif;
```

Database Schema Conventions

Table Names

Table names have all lowercase letters and underscores between words, also all table names need to be plural, e.g. invoice_items, orders.

If the table name contains serveral words, only the last one should be plural:

```
applications
application_functions
application_function_roles
```

Field Names

Field names will be lowercase, generally singular case, and words are separated by underscores, e.g. order_amount, first_name

Foreign Keys

The foreign key is named with the singular version of the target table name with _id appended to it, e.g. order_id in the items table where we have items linked to the orders table.

Many-To-Many Link Tables

Tables used to join two tables in a many to many relationship is named using the model names they link, with the table names in alphabetical order, for example item_order.

Indexes

Indexes should follow the naming pattern of idx_{column name}. For example, an index for the column created_by on a table would have an indexed named idx_created_by.

```
ALTER TABLE `#__my_table` ADD INDEX `idx_created_by` (`created_by`);
```

For indexes that use multiple columns, list each column by order of cardinality.

```
ALTER TABLE `#__my_table` ADD INDEX `idx_category_referenceid` (`categ
ory`, `referenced`);
```

Unique Indexes

Unique indexes follow the same pattern as above but should start with uidx_.

```
ALTER TABLE `#__my_table` ADD UNIQUE `uidx_alias` (`alias`);
```

Fulltext Indexes

Fulltext indexes follow the same pattern as above but should start with ftidx_.

```
ALTER TABLE `#__my_table` ADD FULLTEXT `ftidx_content` (`content`);
```

Commit Messages

GIT commit messages are generally comprised of three sections: a short summary, a longer explanation, and links to references. Starting with this structure forces you to answer why the change is necessary before outlining the changes that you have made.

- The summary should be no longer than 50 characters
- The summary should be tagged with what type of commit it is. For example, "[feat] Added a new feature" or "[fix] Fixed a bug".
- Each line of the explanation should wrap at 72 characters
- If a commit fixes an issue, it should be denoted with Fixes: {url}
- If a commit references another commit, outside source, etc, it should be denoted with Refs: {url}

The summary is required but the longer explanation and refs may not always be necessary or apply to the situation at hand.

Template

You can tell GIT to use a template for commit messages with this structure. This is done by setting commit.template in ~/.gitconfig:

```
[commit]
  template = ~/.gitmessage
```

Then create ~/.gitmessage with the following:

```
#----------------------50-----------------------|
#Summary (50 characters)

#----------------------------------72-----------------------------------
-|
#Explanation (wrap around 72 characters (80 - short indent on either s
ide))

#----------------------------------72-----------------------------------
-|

#----------------------------------72-----------------------------------
-|
#Refs:
#Fixes:
```

```
#-----------------------------------72-----------------------------------
-|
# Type can be
#    feat     (new feature)
#    fix      (bug fix)
#    refactor (refactoring production code)
#    style    (formatting, missing semi colons, etc; no code change)
#    docs     (changes to documentation)
#    test     (adding or refactoring tests; no production code change)
```