DotKernel Coding Standard for PHP

DotKernel is an application built on top of Zend Framework.

DotKernel borrowed the coding standard from Zend Framework: ZF Coding Standard with some exceptions.

B.2. PHP FILE FORMATTING

B.2.1. GENERAL

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.

B.2.2. INDENTATION

It makes the indent with tabs, and not with spaces (***not like in Zend Framework, where tabs are not allowed***)

B.2.3. MAXIMUM LINE LENGTH

The target line length is 80 characters. That is to say, DK developers should strive keep each line of their code under 80 characters where possible and practical. However, longer lines are acceptable in some circumstances. The maximum length of any line of PHP code is 120 characters.

B.2.4. 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 0x0A.

Note: Do not use carriage returns (CR) as is the convention in Apple OS's (0x0D) or the carriage return/linefeed combination (CRLF) as is standard for the Windows OS (0x0D, 0x0A).

B.3. NAMING CONVENTIONS

Camel naming convention.

B.3.1. CLASSES

Starts with **Dot_** (e.g. Dot_Templates).

DotKernel standardizes on a class naming convention whereby the names of the classes directly map to the directories in which they are stored. The root level directory of the DK standard library is the "Dot" directory.

All Dot Kernel 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 "Dot/Db/Table.php" must map to the class name "Dot_Db_Table".

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 "Dot_PDF" is not allowed while "Dot_Pdf" is acceptable.

These conventions define a pseudo-namespace mechanism for DotKernel. DotKernel will adopt the PHP namespace feature like:

Example of loader and register the **Dot** namespace

```
require once 'Zend/Loader/Autoloader.php';
$loader = Zend_Loader_Autoloader::getInstance();
$loader->register Namespace('Dot_');
```

See the class names in the standard and extras libraries for examples of this classname convention. *IMPORTANT:* Code that must be deployed alongside DK libraries but is not part of the standard or extras libraries (e.g. application code or libraries that are not distributed by Dot) must never start with "Dot_".

B.3.2. INTERFACES

Ends with the string “Interface”(e.g. Dot_Db_Interface).

B.3.3. FILENAMES

All php files will have the extension “.php” **.**

For all other files, 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 for DotKernel classes:

```
Dot/Db.php
Dot/Controller/Front.php
Dot/View/Helper/FormRadio.php
```

File names must map to class names as described above.

B.3.4.FUNCTIONS AND METHODS

Function names may only contain alphanumeric characters. Underscores are not permitted. 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()
```

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.

B.3.4.5 VARIABLES

Variable names may only contain alphanumeric characters. Underscores are not permitted. 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. This is the only acceptable application of an underscore in a variable name. Member variables declared "public" should never start with an underscore.

As with function names (see section 3.3) variable names must always start with a lowercase letter and follow the "camelCaps" capitalization convention.

```
class SomeClassName
{
    public     $varNameOne;
    private    $_varNameTwo;
    protected $_varNameThree;
}

$someVar = “some text”;
```

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.

B.3.4.5 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***.

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.

B.4. CODING STYLE

B.4.1 PHP CODE DEMARCATION

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

```
<?php
?>
```

Short tags are never allowed. For files containing only PHP code, the closing tag must always be omitted (See Section B.2.1, “General”).

B.4.2. STRINGS

B.4.2.1. 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';
```

B.4.2.2. 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.

B.4.2.3. 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!";
```

B.4.2.4.String Concatenation

Strings must be concatenated using the "." operator. (***not like in Zend Framework, where space must not be added before and after the "." operator***)

```
$company = 'DK' .' '.'Technologies';
```

When concatenating 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 `people` "
    . "WHERE `name` = 'Susan' "
    . "ORDER BY `name` ASC ";
```

B.4.3. ARRAYS

B.4.3.1. Numerical 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, 'DK', 'Studio');
```

It is permitted to declare multi-line indexed arrays using the "array" construct. In this case, each successive line must be padded with spaces such that beginning of each line is aligned:

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

B.4.3.2. 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');
```

B.4.4. CLASSES

B.4.4.1. Class Declaration

Classes must be named according to DotKernel'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 PHP Documentor standard.

All code in a class must be indented with 2 tabs.

Only one class is permitted in each PHP file.

Placing additional code in class files is permitted but discouraged. In such files, two blank lines must separate the class from any additional PHP code in the class file.

The following is an example of an acceptable class declaration:

```
/**
 * Documentation Block Here
 */
class SampleClass
{
  // all contents of class
    // must be indented 2 tabs
}
```

B.4.4.2. Class Members Variables

Member variables must be named according to DotKernel'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 not permitted. Member variables always 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 accessors methods (set/get).

B.4.5. FUNCTIONS AND METHODS

B.4.5.1. Functions And Methods Declaration

Functions must be named according to the DotKernel 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 2 tabs
      }
}
```

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;
  }
}
```

B.4.5.2 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, 'DK', 'Studio',
                $a, $b, $c,
              56.44, $d, 500), 2, 3);
```

B.4.6. CONTROL STATEMENTS

B.4.6.1 IF /ELSE/ELSEIF

Every ***starting curly brace {*** after a statement starts on a new line, end it's ***closing curly brace }*** will be on a new line too. The start and end braces must be on the same column (for better indentation of the code).

e.g:

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

```
if ($a != 2)
{
   $a = 2;
   if($a == 2)
   {
     $c = 3;
   }
}
```

Control statements based on the ***if*** and ***elseif*** constructs must have a single space before the opening parenthesis of the conditional and a single space after the closing parenthesis.

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 next line after the conditional statement. The ***closing brace*** is always written on its own line. Any content within the braces must be indented using `2 tabs`:

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

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

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

PHP allows statements to be written without braces in some circumstances. This coding

standard makes no differentiation- all ***"if", "elseif" or "else"*** **statements must use braces.**

Use of the ***"elseif"*** construct is permitted but strongly discouraged in favor of the ***"else if"*** combination.

B.4.6.2. 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 using `2 tabs`. Content under each "case" statement must be indented using an additional `2 tabs`.

```
switch ($numPeople)
{
  case 1:
  break;
  case 2:
  break;
  default:
```

The construct `default` should never 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.

B.4.7. INLINE DOCUMENTATION

B.4.7.1. 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:http://phpdoc.org/

All class files must contain a "file-level" docblock at the top of each file and a "class-level" docblock immediately above each class. Examples of such docblocks can be found below.

B.4.7.2. 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:

```
/**
 * Short description for file
 *
 * Long description for file (if any)...
 *
 * DotKernel
 * NOTICE OF LICENSE
 *
 * This source file is subject to the Open Software License (OSL 3.0)
 * that is bundled with this package in the file LICENSE.txt.
 * It is also available through the world-wide-web at this URL:
 * http://opensource.org/licenses/osl-3.0.php
 * If you did not receive a copy of the license and are unable to
 * obtain it through the world-wide-web, please send an email
```

```
 * to license@dotkernel.com so we can send you a copy immediately.
 *
 *
 * @category   DotKernel
 * @package    DotKernel
 * @copyright  Copyright (c) 2009 DotBoost  Technologies (http://www.dotboost.com)
 * @license    http://opensource.org/licenses/osl-3.0.php  Open Software License (OSL 3.0)
 * @version   $Id:$
 * @since     File available since Release 1.5.0
*/
```

B.4.7.3. 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)...
 *
* DotKernel
 * NOTICE OF LICENSE
 *
 * This source file is subject to the Open Software License (OSL 3.0)
 * that is bundled with this package in the file LICENSE.txt.
 * It is also available through the world-wide-web at this URL:
 * http://opensource.org/licenses/osl-3.0.php
 * If you did not receive a copy of the license and are unable to
 * obtain it through the world-wide-web, please send an email
 * to license@dotkernel.com so we can send you a copy immediately.
 *
 * @category   DotKernel
 * @package    DotKernel
 * @copyright  Copyright (c) 2009 DotBoost  Technologies (http://www.dotboost.com)
 * @license    http://opensource.org/licenses/osl-3.0.php  Open Software License (OSL 3.0)
 * @version   Release: @package_version@
 * @since     Class available since Release 1.5.0
 * @deprecated Class deprecated in Release 2.0.0
 */
```

B.4.7.4. 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/method may throw an exception, use @throws for all known exception classes:

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