General Style and Syntax

The following page describes the coding rules use adhere to when developing CodeIgniter.

Table of Contents

File Format

Files should be saved with Unicode (UTF-8) encoding. The BOM should *not* be used. Unlike UTF-16 and UTF-32, there's no byte order to indicate in a UTF-8 encoded file, and the BOM can have a negative side effect in PHP of sending output, preventing the application from being able to set its own headers. Unix line endings should be used (LF).

Here is how to apply these settings in some of the more common text editors. Instructions for your text editor may vary; check your text editor's documentation.

TextMate

1. Open the Application Preferences
2. Click Advanced, and then the "Saving" tab
3. In "File Encoding", select "UTF-8 (recommended)"
4. In "Line Endings", select "LF (recommended)"
5. *Optional:* Check "Use for existing files as well" if you wish to modify the line endings of files you open to your new preference.

BBEdit

1. Open the Application Preferences
2. Select "Text Encodings" on the left.
3. In "Default text encoding for new documents", select "Unicode (UTF-8, no BOM)"
4. *Optional:* In "If file's encoding can't be guessed, use", select "Unicode (UTF-8, no BOM)"
5. Select "Text Files" on the left.
6. In "Default line breaks", select "Mac OS X and Unix (LF)"

PHP Closing Tag

The PHP closing tag on a PHP document **?>** is optional to the PHP parser. However, if used, any whitespace following the closing tag, whether introduced by the developer, user, or an FTP application, can cause unwanted output, PHP errors, or if the latter are suppressed, blank pages. For this reason, all PHP files should **OMIT** the closing PHP tag, and instead use a comment block to mark the end of file and it's location relative to the application root. This allows you to still identify a file as being complete and not truncated.

```
INCORRECT:
<?php

echo "Here's my code!";

?>

CORRECT:
<?php

echo "Here's my code!";

/* End of file myfile.php */
/* Location: ./system/modules/mymodule/myfile.php */
```

Class and Method Naming

Class names should always have their first letter uppercase, and the constructor method should match identically. Multiple words should be separated with an underscore, and not CamelCased. All other class methods should be entirely lowercased and named to clearly indicate their function, preferably including a verb. Try to avoid overly long and verbose names.

```
INCORRECT:
class superclass
class SuperClass

CORRECT:
class Super_class
```

Notice that the Class and constructor methods are identically named and cased:

```
class Super_class {

        function Super_class()
        {

        }
}
```

Examples of improper and proper method naming:

```
INCORRECT:
function fileproperties()          // not descriptive and needs underscore separator
function fileProperties()          // not descriptive and uses CamelCase
function getfileproperties()            // Better!  But still missing underscore separator
function getFileProperties()            // uses CamelCase
function get_the_file_properties_from_the_file()       // wordy

CORRECT:
function get_file_properties()     // descriptive, underscore separator, and all lowercase letters
```

Variable Names

The guidelines for variable naming is very similar to that used for class methods. Namely, variables should contain only lowercase letters, use underscore separators, and be reasonably named to indicate their purpose and contents. Very short, non-word variables should only be used as iterators in for() loops.

```
INCORRECT:
$j = 'foo';        // single letter variables should only be used in for() loops
$Str               // contains uppercase letters
$bufferedText           // uses CamelCasing, and could be shortened without losing semantic meaning
$groupid           // multiple words, needs underscore separator
$name_of_last_city_used  // too long

CORRECT:
for ($j = 0; $j < 10; $j++)
$str
```

```
$buffer
$group_id
$last_city
```

Commenting

In general, code should be commented prolifically. It not only helps describe the flow and intent of the code for less experienced programmers, but can prove invaluable when returning to your own code months down the line. There is not a required format for comments, but the following are recommended.

DocBlock style comments preceding class and method declarations so they can be picked up by IDEs:

```
/**
 * Super Class
 *
 * @package     Package Name
 * @subpackage        Subpackage
 * @category    Category
 * @author      Author Name
 * @link    http://example.com
 */
class Super_class {
```

```
/**
 * Encodes string for use in XML
 *
 * @access      public
 * @param       string
 * @return      string
 */
function xml_encode($str)
```

Use single line comments within code, leaving a blank line between large comment blocks and code.

```
// break up the string by newlines
$parts = explode("\n", $str);

// A longer comment that needs to give greater detail on what is
// occurring and why can use multiple single-line comments.  Try to
// keep the width reasonable, around 70 characters is the easiest to
// read.  Don't hesitate to link to permanent external resources
// that may provide greater detail:
//
// http://example.com/information_about_something/in_particular/

$parts = $this->foo($parts);
```

Constants

Constants follow the same guidelines as do variables, except constants should always be fully uppercase. *Always use CodeIgniter constants when appropriate, i.e. SLASH, LD, RD, PATH_CACHE, etc.*

```
INCORRECT:
myConstant     // missing underscore separator and not fully uppercase
N          // no single-letter constants
S_C_VER        // not descriptive
$str = str_replace('{foo}', 'bar', $str);     // should use LD and RD constants

CORRECT:
MY_CONSTANT
NEWLINE
SUPER_CLASS_VERSION
$str = str_replace(LD.'foo'.RD, 'bar', $str);
```

TRUE, FALSE, and NULL

TRUE, **FALSE**, and **NULL** keywords should always be fully uppercase.

```
INCORRECT:
if ($foo == true)
$bar = false;
function foo($bar = null)

CORRECT:
if ($foo == TRUE)
$bar = FALSE;
function foo($bar = NULL)
```

Logical Operators

Use of **||** is discouraged as its clarity on some output devices is low (looking like the number 11 for instance). **&&** is preferred over **AND** but either are acceptable, and a space should always precede and follow **!**.

```
INCORRECT:
if ($foo || $bar)
if ($foo AND $bar)  // okay but not recommended for common syntax highlighting applications
if (!$foo)
if (! is_array($foo))

CORRECT:
if ($foo OR $bar)
if ($foo && $bar) // recommended
if ( ! $foo)
if ( ! is_array($foo))
```

Comparing Return Values and Typecasting

Some PHP functions return FALSE on failure, but may also have a valid return value of "" or 0, which would evaluate to FALSE in loose comparisons. Be explicit by comparing the variable type when using these return values in conditionals to ensure the return value is indeed what you expect, and not a value that has an equivalent loose-type evaluation.

Use the same stringency in returning and checking your own variables. Use **===** and **!==** as necessary.

```
INCORRECT:
// If 'foo' is at the beginning of the string, strpos will return a 0,
// resulting in this conditional evaluating as TRUE
if (strpos($str, 'foo') == FALSE)

CORRECT:
if (strpos($str, 'foo') === FALSE)
```

```
INCORRECT:
function build_string($str = "")
{
        if ($str == "")   // uh-oh!  What if FALSE or the integer 0 is passed as an argument?
        {

        }
}

CORRECT:
function build_string($str = "")
{
        if ($str === "")
        {

        }
}
```

See also information regarding typecasting, which can be quite useful. Typecasting has a slightly different effect which may be desirable. When casting a variable as a string, for instance, NULL and boolean FALSE variables become empty strings, 0 (and other numbers) become strings of digits, and boolean TRUE becomes "1":

```
$str = (string) $str;   // cast $str as a string
```

Debugging Code

No debugging code can be left in place for submitted add-ons unless it is commented out, i.e. no var_dump(), print_r(), die(), and exit() calls that were used while creating the add-on, unless they are commented out.

```
// print_r($foo);
```

Whitespace in Files

No whitespace can precede the opening PHP tag or follow the closing PHP tag. Output is buffered, so whitespace in your files can cause output to begin before CodeIgniter outputs its content, leading to errors and an inability for CodeIgniter to send proper headers. In the examples below, select the text with your mouse to reveal the incorrect whitespace.

INCORRECT:

```

<?php
	// ...there is whitespace and a linebreak above the opening PHP tag
	// as well as whitespace after the closing PHP tag
?>
```

CORRECT:

```
<?php
	// this sample has no whitespace before or after the opening and closing PHP tags
?>
```

Compatibility

Unless specifically mentioned in your add-on's documentation, all code must be compatible with PHP version 4.3+. Additionally, do not use PHP functions that require non-default libraries to be installed unless your code contains an alternative method when the function is not available, or you implicitly document that your add-on requires said PHP libraries.

Class and File Names using Common Words

When your class or filename is a common word, or might quite likely be identically named in another PHP script, provide a unique prefix to help prevent collision. Always realize that your end users may be running other add-ons or third party PHP scripts. Choose a prefix that is unique to your identity as a developer or company.

```
INCORRECT:
class Email			pi.email.php
class Xml		ext.xml.php
class Import			mod.import.php

CORRECT:
class Pre_email		pi.pre_email.php
class Pre_xml		ext.pre_xml.php
class Pre_import		mod.pre_import.php
```

Database Table Names

Any tables that your add-on might use must use the 'exp_' prefix, followed by a prefix uniquely identifying you as the developer or company, and then a short descriptive table name. You do not need to be concerned about the database prefix being used on the user's installation, as CodeIgniter's database class will automatically convert 'exp_' to what is actually being used.

```
INCORRECT:
email_addresses			// missing both prefixes
pre_email_addresses		// missing exp_ prefix
exp_email_addresses		// missing unique prefix

CORRECT:
exp_pre_email_addresses
```

NOTE: Be mindful that MySQL has a limit of 64 characters for table names. This should not be an issue as table names that would exceed this would likely have unreasonable names. For instance, the following table name exceeds this limitation by one character. Silly, no?
exp_pre_email_addresses_of_registered_users_in_seattle_washington

One File per Class

Use separate files for each class your add-on uses, unless the classes are *closely related*. An example of CodeIgniter files that contains multiple classes is the Database class file, which contains both the DB class and the DB_Cache class, and the Magpie plugin, which contains both the Magpie and Snoopy classes.

Whitespace

Use tabs for whitespace in your code, not spaces. This may seem like a small thing, but using tabs instead of whitespace allows the developer looking at your code to have indentation at levels that they prefer and customize in whatever application they use. And as a side benefit, it results in (slightly) more compact files, storing one tab character versus, say, four space characters.

Line Breaks

Files must be saved with Unix line breaks. This is more of an issue for developers who work in Windows, but in any case ensure that your text editor is setup to save files with Unix line breaks.

Code Indenting

Use Allman style indenting. With the exception of Class declarations, braces are always placed on a line by themselves, and indented at the same level as the control statement that "owns" them.

INCORRECT:
```
function foo($bar) {
	// ...
}

foreach ($arr as $key => $val) {
	// ...
}

if ($foo == $bar) {
	// ...
} else {
	// ...
}

for ($i = 0; $i < 10; $i++)
	{
	for ($j = 0; $j < 10; $j++)
		{
		// ...
		}
	}
```

```
CORRECT:
function foo($bar)
{
	// ...
}

foreach ($arr as $key => $val)
{
	// ...
}

if ($foo == $bar)
{
	// ...
}
else
{
	// ...
}

for ($i = 0; $i < 10; $i++)
{
	for ($j = 0; $j < 10; $j++)
	{
		// ...
	}
}
```

Bracket and Parenthetic Spacing

In general, parenthesis and brackets should not use any additional spaces. The exception is that a space should always follow PHP control structures that accept arguments with parenthesis (declare, do-while, elseif, for, foreach, if, switch, while), to help distinguish them from functions and increase readability.

```
INCORRECT:
$arr[ $foo ] = 'foo';

CORRECT:
$arr[$foo] = 'foo'; // no spaces around array keys


INCORRECT:
function foo ( $bar )
{

}

CORRECT:
function foo($bar) // no spaces around parenthesis in function declarations
{

}


INCORRECT:
foreach( $query->result() as $row )

CORRECT:
```

```
foreach ($query->result() as $row) // single space following PHP control structures, but not in interior parenthesis
```

Localized Text in Control Panel

Any text that is output in the control panel should use language variables in your module's lang file to allow localization.

```
INCORRECT:
return "Invalid Selection";

CORRECT:
return $LANG->line('invalid_selection');
```

Private Methods and Variables

Methods and variables that are only accessed internally by your class, such as utility and helper functions that your public methods use for code abstraction, should be prefixed with an underscore.

```
convert_text()        // public method
_convert_text()       // private method
```

PHP Errors

Code must run error free and not rely on warnings and notices to be hidden to meet this requirement. For instance, never access a variable that you did not set yourself (such as $_POST array keys) without first checking to see that it isset().

Make sure that while developing your add-on, error reporting is enabled for ALL users, and that display_errors is enabled in the PHP environment. You can check this setting with:

```
if (ini_get('display_errors') == 1)
{
	exit "Enabled";
}
```

On some servers where display_errors is disabled, and you do not have the ability to change this in the php.ini, you can often enable it with:

```
ini_set('display_errors', 1);
```

NOTE: Setting the display_errors setting with ini_set() at runtime is not identical to having it enabled in the PHP environment. Namely, it will not have any effect if the script has fatal errors

Short Open Tags

Always use full PHP opening tags, in case a server does not have short_open_tag enabled.

```
INCORRECT:
<? echo $foo; ?>

<?=$foo?>

CORRECT:
<?php echo $foo; ?>
```

One Statement Per Line

Never combine statements on one line.

```
INCORRECT:
$foo = 'this'; $bar = 'that'; $bat = str_replace($foo, $bar, $bag);

CORRECT:
$foo = 'this';
$bar = 'that';
$bat = str_replace($foo, $bar, $bag);
```

Strings

Always use single quoted strings unless you need variables parsed, and in cases where you do need variables parsed, use braces to prevent greedy token parsing. You may also use double-quoted strings if the string contains single quotes, so you do not have to use escape characters.

```
INCORRECT:
"My String"                             // no variable parsing, so no use for double quotes
"My string $foo"                        // needs braces
'SELECT foo FROM bar WHERE baz = \'bag\''     // ugly

CORRECT:
'My String'
"My string {$foo}"
"SELECT foo FROM bar WHERE baz = 'bag'"
```

SQL Queries

MySQL keywords are always capitalized: SELECT, INSERT, UPDATE, WHERE, AS, JOIN, ON, IN, etc.

Break up long queries into multiple lines for legibility, preferably breaking for each clause.

```
INCORRECT:
// keywords are lowercase and query is too long for
// a single line (... indicates continuation of line)
$query = $this->db->query("select foo, bar, baz, foofoo, foobar as raboof, foobaz from exp_pre_email_addresses
...where foo != 'oof' and baz != 'zab' order by foobaz limit 5, 100");

CORRECT:
```

```
$query = $this->db->query("SELECT foo, bar, baz, foofoo, foobar AS raboof, foobaz
                            FROM exp_pre_email_addresses
                            WHERE foo != 'oof'
                            AND baz != 'zab'
                            ORDER BY foobaz
                            LIMIT 5, 100");
```

Default Function Arguments

Whenever appropriate, provide function argument defaults, which helps prevent PHP errors with mistaken calls and provides common fallback values which can save a few lines of code. Example:

```
function foo($bar = '', $baz = FALSE)
```

Overlapping Tag Parameters

Avoid multiple tag parameters that have effect on the same thing. For instance, instead of **include=** and **exclude=**, perhaps allow **include=** to handle the parameter alone, with the addition of "not", e.g. **include="not bar"**. This will prevent problems of parameters overlapping or having to worry about which parameter has priority over another.