**Drupal Development Guidelines**



**ver 1.0**

Table of contents

[1.Coding Standard 4](#_Toc447022830)

[1.1 Indenting and Whitespace 4](#_Toc447022831)

[1.1.1 Operators 4](#_Toc447022832)

[1.1.2 Control Structures 4](#_Toc447022833)

[1.1.3 Function Calls & Declarations 5](#_Toc447022834)

[1.1.4 String Concatenation 5](#_Toc447022835)

[1.1.5 Short array syntax 5](#_Toc447022836)

[1.2 Naming Convention 6](#_Toc447022837)

[1.2.1 Constants 6](#_Toc447022838)

[1.2.2 Global Variables 6](#_Toc447022839)

[1.3 Including File 6](#_Toc447022840)

[1.4 PHP Code 7](#_Toc447022841)

[2. Drupal Core 8](#_Toc447022842)

[2.2 Don’t modify Drupal Core Files 8](#_Toc447022845)

[2.3 Upgrade Drupal Modules 8](#_Toc447022846)

[2.4 Unused Modules and themes 9](#_Toc447022847)

[3. Module 10](#_Toc447022848)

[3.2 Integrate with Views, when possible 10](#_Toc447022852)

[3.3 Dependent modules 10](#_Toc447022853)

[3.4 Use Drupal.behaviors 10](#_Toc447022854)

[3.5 Use .install file 10](#_Toc447022855)

[3.6 Coder Module 10](#_Toc447022856)

[3.7 Module Packages 11](#_Toc447022857)

[4. Twig: Templates 12](#_Toc447022858)

[5. Writing secure code 13](#_Toc447022859)

[6. Database Access 14](#_Toc447022860)

[6.2 Use parameterized query to prevent SQL injection 14](#_Toc447022869)

[7. Commenting Framework 15](#_Toc447022870)

[7.2 General Notes on API documentation 15](#_Toc447022872)

[7.3 Hook implementation documentation 16](#_Toc447022873)

[7.4 Drupal API documentation standards for classes and namespaces 17](#_Toc447022874)

[8. Security and Permissions 18](#_Toc447022875)

[8.2 Anonymous User 18](#_Toc447022877)

[8.3 Registered User 18](#_Toc447022878)

[8.4 Validate user input and URL arguments 18](#_Toc447022879)

[9. Performance and scalability 19](#_Toc447022880)

[9.2 Examine the alter hooks that the module uses 19](#_Toc447022881)

[9.3 Limit your use of front-end JavaScript code 19](#_Toc447022882)

[9.4 Avoid generating notices for "missing" information 19](#_Toc447022883)

[9.5 Be aware of the use of display:none 19](#_Toc447022884)

# Coding Standard

The Drupal Coding Standards apply to code within Drupal and its contributed modules. This document is loosely based on the PEAR Coding standards.  All new code should follow the current standards.

## Indenting and Whitespace

Use an indent of 2 spaces, with no tabs.

Lines should have no trailing whitespace at the end.

## Operators

All binary operators (operators that come between two values), such as +, -, =, !=, ==, >, etc. should have a space before and after the operator, for readability.

|  |  |
| --- | --- |
| 1  2  3  4  5 | //Wrong  $foo=$bar;  //Right  $foo = $bar; |

## Control Structures

Control statements should have one space between the control keyword and opening parenthesis, to distinguish them from function calls.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | if (condition1 || condition2) {  action1;  }  elseif (condition3 && condition4) {  action2;  }  else {  defaultaction;  } |

(Note: Don't use "else if" -- always use elseif.)

## Function Calls & Declarations

Functions should be called with no spaces between the function name, the opening parenthesis, and the first parameter; spaces between commas and each parameter, and no space between the last parameter, the closing parenthesis, and the semicolon.

|  |  |
| --- | --- |
| 1  2  3 | $var = foo($bar, $baz, $quux); |

## String Concatenation

Always use a space between the dot and the concatenated parts to improve readability.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <?php   $string = 'Foo' . $bar;   $string = $bar . 'foo';   $string = bar() . 'foo';   $string = 'foo' . 'bar'; ?> |

## Short array syntax

["apples", "oranges", "bananas"]

instead of

array("apples", "oranges", "bananas")

## Naming Convention

## Constants

* Constants should always be all-uppercase, with underscores to separate words. (This includes pre-defined PHP constants like TRUE, FALSE, and NULL.)
* In Drupal 8 and later, constants should be defined using the [const PHP language keyword](http://us3.php.net/const) (instead of define()), because it is better for performance:

|  |  |
| --- | --- |
| 1  2  3 | <?php  Const CACHE\_TEMPORARY = -1;  ?> |

## Global Variables

If you need to define global variables, their name should start with a single underscore followed by the module/theme name and another underscore.

* + 1. Classes
* Classes and interfaces should use UpperCamel naming.
* Methods and class properties should use lowerCamel naming. In Drupal 8, properties of configuration entities are exempt of these conventions. Those properties are allowed to use underscores.
* Classes should not use underscores in class names unless absolutely necessary to derive names inherited class names dynamically. That is quite rare, especially as Drupal does not mandate a class-file naming match.
* Names should not include "Drupal".
* Class names should not have "Class" in the name.
* Interfaces should always have the suffix "Interface".
* Protected or private properties and methods should not use an underscore prefix.

## Including File

Anywhere you are unconditionally including a class file, use require\_once(). Anywhere you are conditionally including a class file (for example, factory methods), use include\_once(). Either of these will ensure that class files are included only once.

When including code from the same directory or a sub-directory, use DRUPAL\_ROOT

|  |  |
| --- | --- |
| 1  2  3 | require\_once DRUPAL\_ROOT . ‘/’ . variable\_get(‘cache\_inc’, ‘includes/cahe.inc’); |

## PHP Code

We dont use closing tag ?> in Drupal. Omitting the closing tag prevents the accidental injection of trailing white space into the response.

# Drupal Core

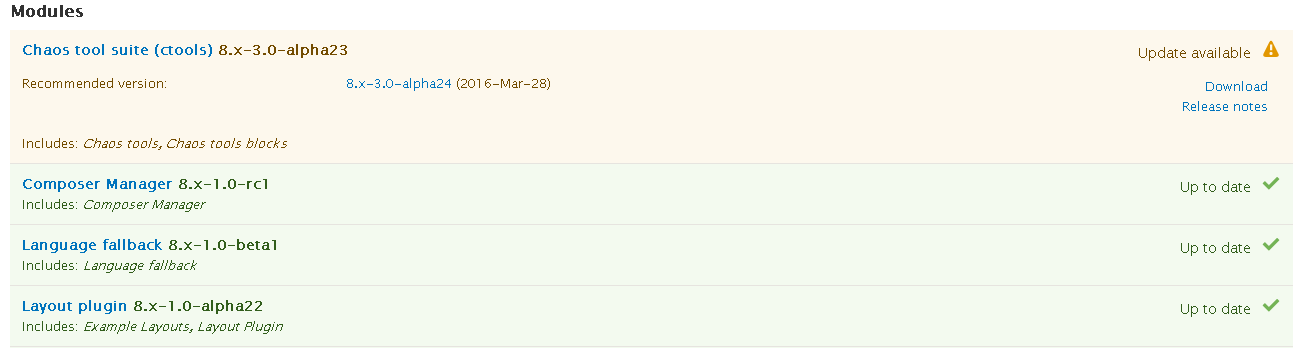


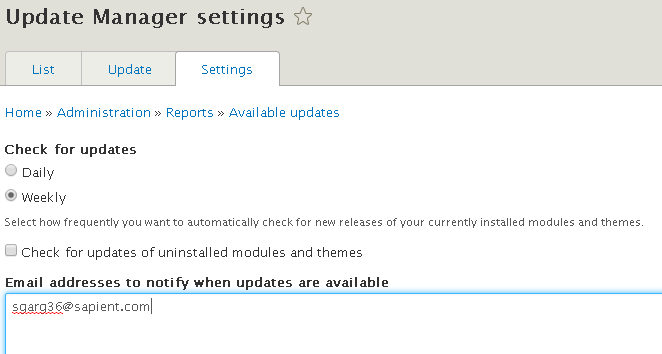
## Don’t modify Drupal Core Files

Never ever modify Drupal core files. Changing Drupal’s core files makes your website impossible to maintain and upgrade because you must keep track of and reapply your changes as you apply security and other updates.

## Upgrade Drupal Modules

On weekly basis validate and upgrade your modules and themes. “Available updates” report in Drupal displays list of information about available updates for your installed modules and themes





## Unused Modules and themes

Disable unused and non-essential modules and themes. Loading additional code unnecessarily will consume additional server resources.

# Module

For creating a custom module your should follow some important rules for selecting a machine name:

* It must start with a letter.
* It must contain only lower-case letters and underscores.
* It must be unique. Your module may not have the same short name as any other module, theme, or installation profile you will be using on the site.
* It may not be any of the reserved terms: src, lib, vendor, assets, css, files, images, js, misc, templates, includes, fixtures, Drupal.
* The API module treats files with the following extensions as PHP: .php, .module, .inc, .install, .engine., .theme, .profile, and .test.



## Integrate with Views, when possible

If it makes sense to expose your data to Views, it reduces a lot of development effort.

## Dependent modules

Ensure that module dependencies are validated before enabling dependent modules. If your module makes calls to functions defined in another module, the module needs to be added as a dependency in your module's .info file. If, instead, your module simply implements the hooks of another module, you don't need to add a dependency for the module.

## Use Drupal.behaviors

Use Drupal.behaviors instead of $(document).ready()

In any .js that you're calling with your module, you can attach anything you'd normally put in document.ready() to Drupal.behaviors. The documentation regarding [Drupal.behaviors](https://drupal.org/node/304258#drupal-behaviors) explains that using Drupal.behaviors allows content loaded using AJAX to continue to have page elements attached to them.

## Use .install file

Ensure that the .install file does everything needed to set up the module.

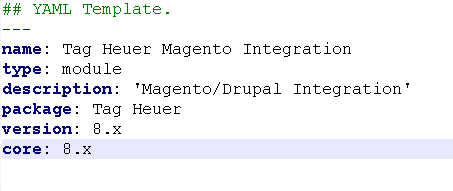
## Coder Module

The Coder module checks against the Drupal coding standards and provides suggestions on what should be changed or replaced. Be sure to compare any displayed warnings against the coding standards.

## Module Packages

Module .info.yml file should use package name.

**Tag Heuer** package should used for all custom modules built for CMS Implementation.



# Twig: Templates

When writing Twig templates, we recommend you to follow these official twig coding standards:

* Put one (and only one) space after the start of a delimiter ({{, {%, and {#) and before the end of a delimiter (}}, %}, and #}):

|  |  |
| --- | --- |
| 1  2  3 | {{ foo }}  *{# comment #}*  {% if foo %}{% endif %} |

* Use lower cased and underscored variable names

|  |  |
| --- | --- |
| 1  2 | {% set foo = 'foo' %}  {% set foo\_bar = 'foo' %} |

* Indent your code inside tags
* Dont include any PHP code in twig template.

# Writing secure code

Whether you are writing a PHP snippet or an entire module, it is important to keep your code secure. Use check functions on output to prevent cross site scripting attacks. No piece of user-submitted content should ever be placed as-is into HTML.

* When outputting plain-text, you need to pass it through check\_plain() before it can be put inside HTML. This will convert quotes, ampersands and angle brackets into entities, causing the string to be shown literally on screen in the browser.
* No piece of user-submitted content should ever be placed as-is into HTML. If you are unsure of whether this is the case, you can always test it by submitting a piece of text like <u>xss</u> into your module's fields. If the text comes out underlined or mangles existing tags, you know you have a problem.

|  |  |
| --- | --- |
| 1  2  3 | //Wrong  <?php print '<tr><td>$title</td><td>'; ?>  <?php print '<a href="/..." title="$title">view node</a>'; ?> |

|  |  |
| --- | --- |
| 1  2  3 | //RIGHT  <?php print '<tr><td>'. check\_plain($title) .'</td></tr>'; ?>  <?php print '<a href="/..." title="'. check\_plain($title) .'">view node</a>'; ?> |

|  |  |
| --- | --- |
|  | //WRONG  <?php print l(check\_plain($title), 'node/'. $nid); ?>  //RIGHT ([l()](https://api.drupal.org/api/drupal/includes%21common.inc/function/l/7) already contains a [check\_plain()](https://api.drupal.org/api/drupal/includes%21bootstrap.inc/function/check_plain/7) call by default):  <?php print l($title, 'node/'. $nid); ?> |

|  |  |
| --- | --- |
| 1  2  3 | //WRONG  <?php print '<a href="/$url">'; ?>  <?php print '<a href="/'. check\_plain($url) .'">'; ?> |

|  |  |
| --- | --- |
| 1  2  3 | //WRONG  //RIGHT (URLs must be checked with [check\_url()](https://api.drupal.org/api/drupal/includes%21common.inc/function/check_url/7)):  <?php print '<a href="/'. check\_url($url) .'">'; ?> |

# Database Access

Always use functions provided by Drupal to access the database to guard against SQL injections attacks.



## Use parameterized query to prevent SQL injection

Use dynamic query functions wherever possible. For inserting / updating db\_insert, db\_update should be used instead of db\_query.

* %d – integers
* %f - floats
* %s - strings, enclose in ''
* %b - binary data, do not enclose in ''
* %% - replaced with %

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <?php  db\_query("SELECT n.nid FROM {node} n WHERE n.nid > %d", $nid);  db\_query("SELECT n.nid FROM {node} n WHERE n.type = '%s'", $type);  db\_query("SELECT n.nid FROM {node} n WHERE n.nid > %d AND n.type = '%s'", $nid, $type);  db\_query("SELECT n.nid FROM {node} n WHERE n.type = '%s' AND n.nid > %d", $type, $nid);  ?> |

Use dynamic query functions wherever possible. For inserting / updating db\_insert, db\_update should be used instead of db\_query.

Source: https://www.drupal.org/dynamic-queries

# Commenting Framework

The Drupal project's standards for API documentation and comments in PHP code, whose purpose is that the API module can parse/display the documentation, that programmers looking at the PHP files can read/understand the documentation, and that integrated developer environments (IDEs) can work successfully with the code and documentation.

The API module parses documentation that is in special documentation blocks (known as "docblocks" in the rest of this document).

|  |  |
| --- | --- |
| 1  2  3  4  5 | /\*\* \* Sample summary line.  \* \* Next paragraph. Etc. \*/ (code being documented goes here, such as a function declaration, class, etc.) |



## General Notes on API documentation

* The special documentation tags described in this document are only recognized within special PHP comment blocks that start with /\*\*. These are known in this document as*docblocks*.
* In-code comment lines starting with // and comment blocks starting with /\* are not recognized as docblocks.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // Some other comment here.  // @todo Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam  // nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat,  // sed diam voluptua.  // We explicitly delete all comments.  comment\_delete($cid); |

* Docblocks normally have \* at the beginning of each line, and the API module strips these out when formatting the documentation (see example above).
* To make a paragraph break in a docblock, leave a blank line (see example above). Some tags also trigger paragraph breaks: @param, @return, @see, @var. The API module does not support blank lines within a single tag's documentation (for example, within a single parameter's documentation with @param.
* The API module itself does not care about line length.
* The first paragraph of a docblock is known as the *summary*.
* To document a function, class, etc., the docblock must appear directly before the item being documented, with no blank line in between (see example above). There are also a few "free-standing" docblocks (for documenting files and making topics -- see tag reference for details).
* The API module automatically turns the names of functions, classes, etc. that it recognizes in documentation text into links to the documentation of those items.

## Hook implementation documentation

|  |  |
| --- | --- |
| 1  2  3  4 | /\*\*  \* Implements hook\_help().  \*/ function blog\_help($section) { |

## Drupal API documentation standards for classes and namespaces

* All classes and all of their methods (including private methods) must be documented.
* If a class has a method that is overriding a method from a parent class/interface, and the documentation is identical, use this short form for the documentation:

|  |  |
| --- | --- |
| 1  2  3  4 | /\*\*  \* {@inheritdoc}  \*/  public function ... |

* Use a third person verb to start the summary of a class, interface, or method. For example: "Represents a ..." or "Provides...".
* Document exceptions with @throws.
* If you use a namespace on a class anywhere in documentation, always make sure it is a fully-qualified namespace (beginning with a backslash).
* Immediately after an @tag (@param, @return, @ver, etc.), class and interface names must always include the fully-qualified namespace.
* Other namespace-related standards for Drupal are under discussion and have not yet been adopted permanently.

# Security and Permissions



## Anonymous User

Ensure that Anonymous permissions are appropriately disabled. Always conduct a unit test on module with anonymous permissions.

## Registered User

Test the module as a normal user account instead of admin account.

## Validate user input and URL arguments

Ensure that your module validates user input from both submitted values and URL parameters before it depends on those values.

# Performance and scalability

## Examine the alter hooks that the module uses

Overusing alter hooks can cause performance issues. The best practice is to use the most specific alter hook required for the function (for example, hook\_form\_FORM\_ID\_alter instead ofhook\_form\_alter).

## Limit your use of front-end JavaScript code

We recommend that you use CSS instead of JavaScript, especially for user interface items. This is because CSS can be smaller and faster than JavaScript. If you use JavaScript, put your code into the footer region for faster page loads.

## Avoid generating notices for "missing" information

Check watchdog reports while unit testing to avoid any notices and warnings.

## Be aware of the use of display:none

If there are issues when you are developing websites, using display:none with entities can be difficult to diagnose. The best practice is to simply ensure that the entity is not displayed.