Written by Chris Graham (ocProducts)

« Return to Code Book table of contents

Introduction

This document documents many coding standards. Agencies may wish to integrate this into their own assessment spreadsheet for projects.

Ideally you should try and learn to apply all these standards naturally, so going through this as a checklist will be a quick final formality when you're finished coding.

What standards to apply

If you are writing code that will be included in future Composr versions (often code for client sites by ocProducts is, if it's a nice clean reusable feature), you must meet the full standards.

For client work you can afford to meet only a subset of standards, as you control the environment much more, and a limited set of programmers will need to work with the code. Team members can agree on the standards used on a per-project basis. So you don't necessarily need to do standards marked as "Product development only".

"Composr development +" means it's needed for Composr code (i.e. code for redistribution to end users), and advisable to do in general (it's best to try and learn to do it by habit if it's something that doesn't make things slower to do).

Automatic application

Any standard not marked in red has some kind of automatic way of being checked, assuming all the recommended tools are used.

The mentioned Composr development mode runs automatically if you are working out of a git repository.

The Code Quality Checker (described in the Code Book, and its **`readme.txt`** file) has various options, including pedantic mode. When we refer to pedantic mode here we might actually mean one of the other non-default options: check to see what's available.

To automatically format code for PSR-2 you can either use the "PHP-CS-Fixer" open source project, or ideally PHP-Storm. PHP-CS-Fixer has some bugs and can't clean everything (especially argument spacing) but generally does a serviceable job.

The optimum PHP-CS-Fixer command is:

Code (Bash)

```
php-cs-fixer fix --fixers=indentation,phpdoc_params,visibility,braces,lowercase_constants,function_declaration,c
```

It is a good idea to use an editor/IDE with support for the **`.editorconfig`** standard. Most editors/IDEs have a plugin for it. **`.editorconfig`** will help you automatically set correct tab settings (amongst other things) as you move between different projects.

Standards

General programming

Good programming practices

Standard	Reason	Scope	Test method	Notes
1) DRY (Don't Repeat Yourself)	Various (esp. maintainability)	All development	Manually	Don't copy & paste big blocks of code (exceptions: see below), or the same snippet of code time-and-time-again. If you spot a reusable chunk of code, consider making both usages work via a new shared function. Sometimes if you see the same **pattern** of code time-and-time-again it makes sense to abstract that too.
2) Only ever copy & paste if you need to, and use care	Increases programmer morale	All development	Manually	If you do copy & paste a chunk of code, e.g. if implementing a new importer and using a pre-existing one as a base – then make sure you at least adjust your copied code to make sense. Don't leave comments in there that refer to the old context (e.g. refer to the wrong importer in the pasted code).
3) Know what you are doing in all your code	Avoids unforeseen problems	All development	Manually	Understand all the code you write. Don't just copy some existing pattern into a new context.
4) Write well structured code	Easier to unit test, easier code re-use	All development	Code Quality Checker can check function length (pedantic mode)	Divide into functions into appropriate units (sensible modules interacting over reasonable interfaces). Where possible create functions and classes that are isolated from specific screens, as these can then have unit tests written for them. Try and make your code flow from top to bottom roughly represent execution order.
5) Keep your code tidy	Increases programmer morale	All development	Manually	Just generally try and keep your code neat. Don't misspell in comments and variable names. Tidy code

				encourages a mindset of zero tolerance for messiness in other areas, and increases general team morale such as to keep a high level of standards. See Broken Windows Theory .
6) Separate code chunks with blank lines	Easier for other programmers to understand	Composr development +	Manually	A chunk of code is usually a few lines long and does a common purpose. Usually the chunk will start with a short comment, and end with a single blank line or a function's closing brace. Make your blank lines mean something (i.e. the end of a code chunk), don't just add them to space things out randomly.
7) Keep track of things you intend to fix later	Stops things being forgotten	All development	Manually	If you have to do some code that really should be cleaner, put a comment by it starting **HACKHACK**. If you haven't finished something put a comment starting **TODO**, so it is easy for someone to later pick up if you forgot to finish it.
8) Don't write cryptic code	Increases programmer morale	All development	Manually	Don't write code that is too "clever" or relies on too deep an understanding, unless there's good reason to. For example, don't assume the programmer knows the precedence order of some of the less common operators so put in explicit brackets to make things clear.
9) Use software engineering best practices	Various (esp. maintainability)	Composr development +	Manually	Use software engineering best practices (while also keeping things consistent with how Composr is designed, sometimes unconventional decisions may have been made for a reason).
10) Write well modularised code	Performance, Easier to modify, Can be split up	Composr development +	Manually	Group things into separate files where appropriate. For example, an API acting for a certain addon would be in a number of code files named after that addon. CSS for an addon would be in a file named after that addon.
11) Use meaningful function / class / variable / file / identifier names	Easier for other programmers to understand	Composr development +	Manually	Use sensible function / class / variable / file / identifier / token names that make full intuitive sense in whatever context they are used in; this is particularly important for global functions (you should have an idea what specifically the function is for without having to look it up).
12) Comment your code, but not too much	Easier for other programmers to understand	Composr development +	Code Quality Checker can check comment density (pedantic mode)	Write comments for non-obvious code. Don't comment obvious things ("this is a loop", "this runs if $foo equals $bar"). Write comments that explain your algorithms, and comments that give a high-level overview of what chunks of code do. Very roughly one of two comments per 7 lines, but don't feel bound to match that.
13) Use constants instead of "magic numbers"	Easier for other programmers to understand	Composr development +	Manually	Don't use "magic numbers". For example, instead of using the numbers 1, 2, and 3 with special meanings throughout your code, define constants and reference those instead.
14) Everything should be configurable from within Composr	User friendliness for webmasters	Product development only	Manually	No FTP/file-manager should ever be necessary to operate, configure, or extend an installed Composr website.
15) Use our standard code header	General quality	Product development only	Manually	We have a standard comment format at the top of files (different depending on the file type). Use it consistently.
16) Any legacy code must be marked as such	Code tidiness and performance	Product development only	Manually	Legacy code, due to supporting data from old Composr versions, must be marked with a **LEGACY** comment. This will allow us to clean it out in future versions, should we want to break old compatibility.

Back-end programming

General

Standard	Reason	Scope	Test method	Notes
17) Use PSR-2 coding standard	Consistent code across all projects	Composr development +	Code Quality Checker	Use PSR-2, e.g. the bracing style, 4-spaces not tabs. Our only exceptions are relating to class and namespace structure, as we are not an OOP-heavy codebase.
18) Spaces around operators	Easier for other programmers to understand	Composr development +	Code Quality Checker	Do like **$a + $b** not **$a+$b**. This makes the code easier to read and understand for some programmers, and largely consistent with what other projects do.

19) Use Composr APIs wherever appropriate	Secure, stable, designed to meet all standards	All development	Manually	If is very easy to make mistakes when writing new code. Our functions have been tuned over years. For example, our **http_download_file** function works on any server and supports lots of options.
20) Use the **type** parameter for your modules	Allows SEO URLs to work automatically	All development	Manually	All modules should be coded to use **type** as the internal specifier of what 'screen' to show. The **run** function should decide what function to call based on this parameter, and typically also the **run_pre** function will be there to load meta-data for the screen (allows HTML to start streaming while the screen code runs). The default entry point for a module must either not exist, or be named **browse**. Do not hardcode defaults other than **browse** as it will interfere with the page-link permission security and the URL scheme support.
21) Remember to apply sorting wherever appropriate	So live sites are tidy	All development	Manually	Remember to sort things. Sometimes this needs doing in PHP code, and sometimes it needs doing in your queries (**ORDER BY**). If doing it in PHP code consider you usually will want to do a "natural" sort. Typically you should order data records by date, not by ID. You should never assume that data comes out of the database in the same order as it goes in.
22) Remember to apply pagination wherever appropriate	Performance	All development	Manually	Don't forget to paginate your browse screens using Composr's **pagination** or back/next system.
23) Use the **temporal** API, not PHP date/time functions	Internationalisation / consistency	All development	Manually	Don't forget about users potentially being in different timezones. Rarely if ever use PHP's inbuilt date/time functions – use Composr's temporal API.
24) Consider performance	Performance on real live sites	All development	Manually	Don't do silly things that affect performance like running queries inside a loop/recursion.
25) Never start a file **<?**	Portability	All development	Code Quality Checker	PHP **short_tags** option may be off.
26) Never include **<?** within your PHP source	Portability	All development	Code Quality Checker	PHP **short_tags** option may be on. This string is used within XML so is fairly common to type in; if you need it append two strings together to form it so that it doesn't have to be literally written into the code.
27) Never end a PHP file with **?>**	Stability	All development	Code Quality Checker	It is not needed, and if a blank line ends up after it accidentally (e.g. added by FTP) then cookies won't save.
28) Use unique function and class names	You may need to load up both at some point	All development	Code Quality Checker	No two classes or functions may have the same name, regardless of where they are defined or used. It should be possible for the whole set of Composr PHP files to be simultaneously loaded.
29) Don't rely on "Security through obscurity"	Security	All development	Manually	Always assume that people will try and hack the website, and that they have the full source code, so don't rely on them not guessing URL parameters etc. For example, if you implement some kind of permission scheme for a feature, don't just limit how the link to the feature is shown, also limit the interface and actualiser functions for the feature.
30) Be careful with: **header()**, **eval()**, **preg_replace()**	Security	All development	Manually, Code Quality Checker pedantic mode will help	If you're putting variables into the header function consider data could contain **\r** or **\n** symbols, so escape via the **escape_header** function. If you're using eval be **extremely** careful to validated input; there's rarely a reason to use it anyway. If you're using **preg_replace** make sure to use **preg_quote** and to manually escape the regular expression encapsulation character (usually **#** or **/**).
31) Avoid using **eval**, **$$**, **create_function** or **preg_replace** with **/e**	Security	Product development only	Manually	It's hackerish and prone to security issues. There's rarely, if ever, a good reason to do it.
32) Long running requests should be tasks	Stability / Performance	All development	Manually	If a page request may take more than 30 seconds to complete, it should be codified as a task so that it can execute out of the task queue in an orderly manner.
33) Any page should be accessible as standalone	General quality	All development	Manually	You shouldn't add modules that are never intended to be used independently, because they will come up on the sitemap. If you intend to write some kind of helper script that may be called by a module then either use an "entry-point script" for that or build the functionality into the module file itself (you are

				allowed to **exit** out of a module with your own HTTP headers etc rather than returning Tempcode).
34) Make your code robust against accessed deleted content	Stability	All development	Manually	Imagine some content was added, then later deleted. Imagine that someone or a web crawler calls up a URL associated with the old content. It should produce an intentional error message (usually our **MISSING_RESOURCE** error), not a PHP error.
35) **underlining_for_names**	Consistent code across all projects	Composr development +	Code Quality Checker	Use **underlining_for_names** in PHP variables, PHP function names, PHP arrays, database tables, database fields, and filenames. Not for JavaScript or CSS though.
36) **UPPER_CASE** should be used for globals and constants	Easier for other programmers to understand	Composr development +	Code Quality Checker	
37) Support minimum PHP version	Portability	Product development only	Code Quality Checker	The version listed in Composr's minimum requirements must be supported.
38) Do not use **$_GET** or **$_POST** directly	Implements security filtering	Product development only	Code Quality Checker, pedantic mode	Instead use one of the set of **get_param_*/post_param_*** functions. Some exceptions are allowed, e.g. looping over all input to find what input exists.
39) Write appropriate hooks for new content types	Feature consistency	Composr development +	Manually	When adding new content types remember to write all appropriate hooks for them (e.g. search hooks).
40) Communicate upload limits	User friendliness	Product development only	Manually	Show the appropriate maximum-upload size on all screens that support file uploads, unless the size is very unlikely to be reached by the user (e.g. just uploading an avatar). Composr has a standard mechanism to do this, which can be automated within CRUD modules. For code examples search the code for **get_upload_limit_config_url**.
41) Use our MVC model as described in the Code Book	Cleaner code, easier to change and maintain	Composr development +	Manually, Code Quality Checker will help	Composr enforces a strict separation between content, presentation and code. This is referred to as the MVC (model, view, controller) design pattern. No HTML is allowed in the PHP code except for very select exceptions, such as **<option>** (for efficiency) and hardcoded emergency error pages. You can usually assume the output will be some kind of HTML though (you don't need to code so abstractly that output could be a speech API, for example).
42) Don't assume PHP extensions or PEAR are installed	Portability	Composr development +	Code Quality Checker	Never assume that any extensions are installed unless they are listed in Composr's minimum requirements. If you do use something else (necessarily so, and only for auxiliary behaviour) you must cleanly handle the situation where it is not present using either code branching or helpful error messages.
43) Use only white-listed functions	Portability, PHP **disabled_functions** option	Product development only	Code Quality Checker	In fact, don't assume any rarely-used function is available unless it is listed in the **sources/phpstub.php** file (Code Quality Checker will check, no need to manually check). We have avoided many functions where web hosts often block using the PHP **disable_functions** option. Use our **php_function_allowed** function.
44) Do not require cookies to be available on visitor machines	Compatibility with user computers	Product development only	Manually	Cookies must not be required for functionality to work (except session cookies), unless said functionality is not needed to operate the system (i.e. alternatives exist).
45) Template GUIDs must be unique	Allows people to theme sites without PHP code	Product development only	Release scripts	Don't copy and paste GUIDs in template calls. If you copy and paste a line with a GUID (**'_GUID' => 'blahblahblah'**), remove it completely from your new line. Our automated scripts can put one back in later. The reason: these are used to allow different uses of shared templates to be differentiated via Tempcode. The GUID is a unique identifier for a specific template call.
46) Put standard **index.html** and **.htaccess** files in new directories	Extra security, stops probing/tunnelling	Product development only	Automated test	Almost all directories should contain an empty **index.html** (to stop directory browsing on web servers that have it enabled). Most directories should also contain the small **.htaccess** file (copy it from another deep directory) that we use to block direct file access. If you strictly need to block access to a directory for security reasons you should also update **web.config** to reflect that (for IIS users).
47)	Allows installation /	Product development	Automated test	Every file intended to be distributed with Composr

All files must be associated with an addon	uninstallation / integrity-scans	only		must be registered in one addon. To do this list it in the obvious position within one of the **sources/hooks/systems/addon_registry** files. If it is a PHP file it should have a matching **@package** line at the top of the file.
48) Manage any interdependencies between addons	So addons may be uninstalled freely	Product development only	Manually, Automated test helps a bit	Dependencies may be properly handled via hooks, using the **get_dependencies** function in the **addon_registry** hook (i.e. say addons only work if some other addon is also there), or via code that uses the **addon_installed** function to guard off the dependency. For example, if you used the **galleries** addon in some feature in the **downloads** addon, you would make sure the **downloads** addon did not break if the **galleries** addon was uninstalled (instead you'd write code to turn off the feature if the dependency was not met).
49) Don't define too many functions in a single file	Portability: PHP **memory_limit** setting	Composr development +	Composr development mode sets a low **memory_limit**	Instead put the script into code files that are loaded only when needed.
50) Pass in extra useful template parameters	To give webmasters more themeing power	Product development only	Manually	Pass in any additional unused parameters to the template where they might be useful; raw timestamps, IDs, etc.
51) Functions should have proper API documentation	Easier for other programmers to understand, and it allows the Code Quality Checker to check more	Composr development +	Code Quality Checker, function signatures feature	Use the "phpdoc"-style syntax correctly, and never adlib on its syntax (it gets parsed). Make sure the data types specified are accurate. Where **null** is allowed for a value make sure that is specified (via **?** before the data type) and its meaning explained. Do not copy and paste comments from elsewhere without changing it to be fully correct for where you've added it.
52) Remember that a site can have multiple themes/languages	Stability	Product development only	Manually	Remember that a site can have multiple themes and languages – be careful that you do not cache things with implicit assumptions that should not have been made (e.g. instead of caching something in English, you should cache stuff per-language).
53) Support multi-site-networks (i.e. site db != forum db)	Stability	Product development only	Composr development mode	Remember that Conversr may be run over an M.S.N. (i.e. different URL, not just from the **forum** zone) and also from a separate database – don't make false assumptions about databases, tables (e.g. using **SITE_DB** instead of **FORUM_DB** and hence wrongly assuming the forum and site always share a database) and file-storage/URLs (e.g. using **get_base_url** instead of **get_forum_base_url**). Use the correct APIs for referencing Conversr data and files. As a general rule, any table starting **f_** is an Conversr table and therefore should be accessed through **FORUM_DB**. If you are running development mode you will get an error ("Using Conversr queries on the wrong driver") if you use the wrong database object.
54) Handle errors in failure-prone function calls (e.g. **fopen**)	Stability	Product development only	Code Quality Checker, pedantic mode	
55) Always respect/handle character-sets properly	Internationalisation / stability	Composr development +	Manually	Don't assume the character set is **utf-8**, or **ISO-8859-1** for input data or for output data. Use appropriate conversions when interfacing with things that come-from/go-to a potentially alien character set.
56) Don't put loose code (code outside functions) in any PHP file	Security – stops hackers "jumping in"	All development	Code Quality Checker	The only loose code allowed is our standard boot code in the entry point files.
57) Don't add more than a few lines of code to an "entry script"	So Composr code-overriding can work on it	Composr development +	Composr development mode / Code Quality Checker	Instead of directly placing code in the entry-point file, put it in a sources file and require it in then call it as a function.
58) Make choice of **GET** or **POST** carefully	So bookmarking works and web accelerators don't break things	Composr development +	Manually	HTTP **GET** requests should not change database state in any major way (it can cause problems with web spiders, problems with web accelerator systems, and XSS vulnerabilities). **POST** requests should not be used for screens that might be reasonably bookmarked (e.g. search result screens, or result filtering screens). Composr has specific features in its API to allow you to choose whether forms use **GET** or **POST** requests.
59) Code as if PHP was "strict typed".	Allows easier bug finding and code portability	Composr development +	Code Quality Checker / ocProducts PHP version	This is hard for many PHP programmers to grasp so is explained in a separate section in our Code Book.

60) If you use cookies, use the Composr cookie API	Portability	Composr development + 	Code Quality Checker, pedantic mode	If you do use cookies, use the Composr cookie functions to read/set/delete them, as these are written to respect the webmaster's defined cookie settings (cookie domain, cookie path).
61) Don't write code that'll output PHP 'notice' errors	Stability	Composr development + 	Composr makes all PHP errors generate stack dumps	
62) E-mail subject lines must be unique, or the e-mail sender must be a specific user (not the website itself)	Usability	Composr development + 	Manually	This convention makes emails easier to scan in an inbox and stops incorrect automatic threading.

Files

Standard	Reason	Scope	Test method	Notes
63) Use the **`fix_permissions`** function after making a new file	Makes the file deletable via FTP	Product development only	ocProducts PHP version	
64) Run the **`sync_file`** function after making any file changes	Allows a site to run on a server farm	Product development only	ocProducts PHP version	
65) Be clever about text line-endings	Portability	Product development only	Manually	Multi-line textual input (from the web browser, or from the file-system) should go through the **`unixify_line_format`** function, to make sure line-terminations are converted to the right format that the server operating-system needs.
66) Generally use binary mode	Internationalisation / portability	Product development only	Manually	We almost always open text files as binary, not text (**`t`** mode). This is because we don't benefit from line-ending auto-conversion and we prefer consistency across platforms. We use our **`unixify_file_format`** function instead to enforce consistency.
67) 1 KB = 1024 bytes, etc	Consistency	Product development only	Manually	We don't say KiB or use 1000s. People don't understand KiB (the term never caught on), and 1024 is the long term convention in computing.
68) Make sure you use appropriate file read/write locking	Stability	Composr development + 	Manually	Files are never locked automatically, and it can cause big problems if you don't manually do locking – writes can mess up each other, or partial files can read in and cache. The **`cms_file_put_contents_safe`** will handle locking well.

URLs and file paths

Standard	Reason	Scope	Test method	Notes
69) Use the **`build_url`** function and **`PAGE_LINK`** symbol	Portability / SEO URLs	All development	Composr development mode	Do not hard-code URLs – use **`build_url()`** correctly. This function will convert to whatever URL scheme is in place, as well as propagate **`keep_`** fields. Never use relative URLs.
70) Use **`build_url`**'s **`$skip_keep`** parameter for e-mailed Comcode	Needed for accurate URLs	All development	Manually	If you are using a URL in an e-mail, make sure the **`$skip_keep`** parameter is set to true, as **`keep_`** parameters should not go out in e-mails.
71) Don't append to URLs in templates	URL schemes involve different URL structures	All development	Manually	Do not try and appending something to a URL inside a template, unless it is a URL to a helper script (rather than a screen) or if you have explicitly disabled SEO for the URL. Pass in all the URLs you need to the template so you don't need to deploy hacks.
72) Fix broken user input	Users often miss out **`http://`**	All development	Manually	If you write functionality that accepts URLs into the system via a form interface then use the **`remove_url_mistakes`** function to make sure the URLs start **`http://`** (users will often miss it out). It should add that on the front if the URL does not already contain **`://`**.
73) Avoid relative paths	Code may run from different relative paths	All development	ocProducts PHP version	Don't assume any current directory (either URL or filesystem): use full paths (using **`get_file_base`**

				or **get_custom_file_base** to start off the full path).
74) Don't assume where a module is located when you link	Webmasters may move modules around	Product development only	Manually	Don't assume which zone a module is located in, except in the case of admin modules. Use the **get_module_zone** function or **_SEARCH** token to break down such assumptions.
75) Don't link to screens using non-canonical URLs	SEO (canonical URLs, avoid duplicate content)	Product development only	Manually	For example, the **forumview** module doesn't pass through a **type** if it is **browse**. Most other modules do. Any **browse** screen doesn't use an ID if the ID would be **db_get_first_id()**.
76) Don't assume where an entry-point is located when you link	Webmasters may move entry-point scripts around	Product development only	Manually	Use the **FIND_SCRIPT** symbol or **find_script** function to find zone-based scripts. Put non zone-based scripts in **data** (or **data_custom** if for an unofficial addon).
77) Use the ***_custom** filing system properly	So the override system can work correctly	Product development only	Manually	Never put custom/user-files/logs in a non-custom directory, and never put code files in a custom directory if they are going to be distributed with Composr. Generally don't put files in places where they don't belong.
78) Use the **get_custom_base_url**, **get_custom_file_base**, **get_base_url** and **get_file_base** distinctions properly	So shared installs work (multiple sites, one Composr install) / GAE	Product development only	Manually	Use a **custom_*** function if referencing custom/user-data, otherwise don't. Core Composr files that can't be changed from inside Composr are not custom, other files are. i.e. non-custom files are those that would be shared between sites.
79) Don't assume directories already exist to write into	So shared installs work (multiple sites, one Composr install) / GAE	Product development only	Manually	If running a blank custom directory, none of the bundled directory structure will be initially present. You must therefore detect and auto-create missing directories, using **make_missing_directory**.
80) Use / as a directory separator, even on Windows	Simplicity	Composr development +	Manually	If we use multiple directory separators it gets messy. / will work on Windows for all PHP library functions.

Databases

Standard	Reason	Scope	Test method	Notes
81) Avoid writing SQL	Removes security and compatibility errors	All development	Code Quality Checker, pedantic mode	Wherever possible just use our APIs, don't write SQL by hand. Use **query_insert**, **query_update** and **query_delete**. The only common cases where you need to write SQL by hand completely are if you need to do **OR** queries, or do wildcard searches.
82) When you need to write SQL, write it securely	Security	All development	Code Quality Checker, pedantic mode	Never ever write SQL without thinking about security.
83) Use **db_escape_string** not **mysql_escape_string** / **addslashes**	Cross database compatibility	All development	Manually	Avoids assumption about how escaping is done (you might not know this but the MySQL escaping method is non-standard and the Microsoft one is actually "correct"!)
84) Be type-strict	Cross database compatibility	All development	MySQL-strict-mode and XML database driver	For example, do not do **WHERE validated='1'** as **validated** is an integer field and thus must be referred to as such. Most database are type-strict, MySQL is an exception so it's a big mistake to get things wrong here.
85) Use sensible database indices	Performance on real live sites	All development	Manually	Create database indexes if you regularly query a table using non-key field(s).
86) Use the most appropriate Composr database field types	So higher-level meta functions can work	All development	Manually	When you define database schemas via **create_table** use the closest Composr field type you can. For example, make use of **AUTO_LINK** and **TIME** and **USER** rather than always using **INTEGER**. Use **URLPATH** for URLs. This will help Composr understand its own database better (which it does need to do in some situations such as broken URL scanning). If a field needs to support Comcode, define it as **SHORT_TRANS__COMCODE** or **LONG_TRANS__COMCODE** (never run **comcode_to_tempcode** on-the-fly).
87) Avoid complex SQL	Cross database compatibility	Product development only	Manually	You cannot use SQL syntax that is not widely supported such as the MySQL substring functions, unless the **db_function** function supports it.
88)	Cross database	Product development	Composr development	Make use of the from/max parameters to the query

Don't use `LIMIT` manually, use the API	compatibility (MS Access)	only	mode	functions rather than hard-coding an SQL `LIMIT` clause (this is required for MS Access compatibility).
89) Don't use ` symbols in your SQL	They are MySQL-specific	Product development only	Manually	You will have been forced to choose field names that don't require it anyway.
90) In queries use `<>` rather than `!=`	Cross database compatibility	Product development only	Manually	Not all databases support the latter syntax.
91) Use `db_string_equal_to`	Compatibility with Oracle	Product development only	Manually	Do not build string equalities into WHERE clauses directly (e.g. `WHERE foo='bar'`), use the `db_string_equal_to` function (e.g. `WHERE ' . db_string_equal_to('foo', 'bar')`).
92) Keep your queries as simple as possible	Cross database compatibility	Product development only	Manually	
93) Do not assume the first record in a table is `1`	Cross database compatibility	Product development only	Manually	This is the case for MySQL but does not necessarily hold true. Use `db_get_first_id()` to find the number to use.
94) Use upper case SQL keywords (e.g. `WHERE` not `where`)	Consistent code across all projects	Product development only	Manually	If you must write your own SQL, write SQL keywords in upper case (e.g. `SELECT`, `FROM`, `UPDATE`, `WHERE`, `LIKE`, `AND`, `OR`). This makes it easier to scan over them, and makes it look distinct from PHP code.
95) Use the correct forum API	Composr's forum compatibility support	Product development only	Manually	Do not assume you are developing for Conversr (unless you are coding Conversr directly), or other forum driver. In other words, use the forum driver API properly and do not assume anything about the forum and member software beyond that.
96) Use table prefixes	Portability	Composr development +	Manually	Do not assume any database table prefix (so don't assume `cms_` is at the start of the table names – use `db_get_table_prefix` to find out what to use).
97) Apply Composr field naming standards	Consistent code across all projects	Composr development +	Composr development mode	Use the same coding standards for database field naming as for Composr (lower case, with underscores to separate words).
98) Avoid processing full row sets	Performance	Composr development +	Manually	For many kinds of table there could be 10's of thousands of rows, and it is very easy to forget this scaling problem when devising algorithms during development.
99) Don't require write queries on normal views	Stability	Product development only	Manually	Some web hosts will restrict SQL write permission if a database size limit is exceeded, so any routine write queries should have the `$fail_ok` option set so that it doesn't take the whole site down.

Templates and Interfaces

Standard	Reason	Scope	Test method	Notes
100) Use existing conventions and features and re-use templates where possible	Consistent UI, user-friendly, tighter product, easier themeing	All development	Manually	Generally try and replicate the interface conventions of existing modules, unless there is a strong argument that you are dealing with a special case. For example, don't design new ways of previewing things, use the standard previewing mechanism. Another example, use the `pagination` function instead of implementing new previous/next navigation code. There are almost always API functions or templates for you to use, so it's actually easier to use conventions than re-implementing from scratch. Some are listed further down.
101) Define breadcrumbs in the standard way	Consistent UI, user-friendly	All development	Manually	Use `breadcrumb_set_self` / `breadcrumb_set_parents` as required to generate appropriate breadcrumb chains. Try to do it at the start of screen code, so it'll be set if an error happens during screen generation.
102) Inline edit links	Consistent UI, user-friendly	Composr development +	Manually	For viewing content: if someone has permission to edit something, then for the template that shows the content, pass in a parameter called `EDIT_URL` (typically), containing the URL to edit the content. If the user doesn't have edit permission, then pass in `EDIT_URL` empty. Use the standard `STAFF_ACTIONS` include method for displaying any actions within the template (see the `DOWNLOAD_SCREEN` template for an example).

These guidelines apply for admin modules...

Standard	Reason	Scope	Test method	Notes
103) Write CRUD (add/edit/delete) modules	Consistent UI, user-friendly	Composr development +	Manually	Write content/data management modules as a Composr CRUD module (add/edit/delete).
104) Use standard approaches after an action is performed	Consistent UI, user-friendly	Composr development +	Manually	CMS/Admin "completion" screens should either be redirects to an appropriate location (e.g. the **view** screen for what was just added/edited), do-next managers, or very rarely, message screens. Follow the conventions other modules use. If you do do a redirect, use the proper API (the **redirect_screen** function) so that the 'success' messages show up on the destination screen for the redirect.
105) Give admin modules a do-next manager for the **browse** screen	Consistent UI, user-friendly	Product development only	Manually	A CMS/Admin module's functions should generally be divided by a welcoming do-next manager (i.e. a set of icons). Exceptions include: when adding only involves specifying a single field (in which case you can put add and edit on the same screen), and also when adding is for power-users only (in which case the add form would be hidden behind a button). Follow the conventions other modules use.
106) Minimise number of interfaces, mark-up jump links consistently	Consistent UI, user-friendly	Composr development +	Manually	Don't duplicate interface elements in Composr, unless the duplicate is clearly marked as some kind of shortcut/dupe/jump. For example, if there is a shortcut link between two modules (a duplication in navigation), use the standard Composr convention for informing the user they are jumping between modules (for an example, see how the link from the "Manage themes" screen to the "Zones" screen works).
107) Populate the helper panel	Consistent UI, user-friendly	Composr development +	Manually	Wherever appropriate define helper-panel tutorial links, and inline help, for Admin Zone and CMS Zone modules. There are functions for setting them.

Here are some guidelines for designing/naming templates...

Standard	Reason	Scope	Test method	Notes
108) Templates should be named in upper case	Portability	All development	Composr development mode	Case sensitive file systems mean we want a strong convention.
109) Use **_SCREEN** templates	Consistent UI, user-friendly	All development	Composr development mode	All screens should ultimately be wrapped with some kind of template suffixed **_SCREEN** that includes a **{TITLE}**. This might be a standard screen template that already exists, such as **INDEX_SCREEN** or it might be custom one such as **DOWNLOAD_SCREEN**.
110) Implement screen previews for new templates and maintain them with template changes	Testability of themes	Product development only	Unit test	Previews allow themers to confirm a theme works on all screens, and allow us to do mass-checks of HTML.
111) Nest your templates	Easier for a web designer to work with	Composr development +	Manually	Templates should nest, not be split up into start/end pairs.
112) Use filename prefixes to group templates	Easy to relate templates to features	Composr development +	Manually	For example, the download addon's templates mostly start **DOWNLOAD_**. There's no strict rule, be sensible.
113) Prefix templates used by blocks with **BLOCK_**	Easy to relate templates to features	Composr development +	Manually	Do this unless the template is also used by a module too.
114) Prefix mail templates (Comcode) with **MAIL_**	Easy to relate templates to features	Composr development +	Manually	
115) Prefix other templates that are Comcode with **_FCOMCODE**	Easy to relate templates to features	Composr development +	Manually	Any template that is for Comcode must contain the filename substring **_MAIL** or **_FCOMCODE**. Any template named like this must be in Comcode format.
116) Store templates in the correct directory	Cleanliness	Composr development +	Manually	XML and text (including Comcode) templates must be in their respective directories.

Here are some specific templates to use where appropriate...

Standard	Reason	Scope	Test method	Notes
117) ...**MAP_TABLE**	Consistent UI	Product development only	Manually	A part of a screen that is for viewing title/value data in a table. There are also some functions to help with map tables.
118) ...**COLUMNED_TABLE**	Consistent UI	Product development only	Manually	Templates for rendering a plain table of something.
119) ...**INDEX_SCREEN** / **INDEX_SCREEN_ENTRY**	Consistent UI	Product development only	Manually	A screen having a shortish paragraphed sequence of things to choose.
120) ... **INDEX_SCREEN_FANCIER_SCREEN** / **INDEX_SCREEN_FANCIER_ENTRY**	Consistent UI	Product development only	Manually	A screen having a shortish tabled/described list of things to choose.
121) ...**CONFIRM_SCREEN**	Consistent UI	Product development only	Manually	Show a preview and confirm it's okay.
122) ...**PAGINATION_***	Consistent UI	Product development only	Manually	Very simple browsing through multiple screens. Usually you'll want to use pagination instead.
123) ...**SCREEN_BUTTON** / **SCREEN_ITEM_BUTTON**	Consistent UI	Product development only	Manually	Show a standard shaped button on the screen.

Here are some specific interfaces to use where appropriate...

Standard	Reason	Scope	Test method	Notes
124) ...**pagination** function	Consistent UI	All development	Manually	An interface for pagination.
125) ...**results_table** function	Consistent UI	All development	Manually	An interface for pagination of a table.

When showing errors use inform / error messaging APIs / conventions, correctly. There are 6 top-level kinds of messaging output...

Standard	Reason	Scope	Test method	Notes
126) Where appropriate, Fatal/Warn/Info attached message	Consistent UI, user-friendly	Composr development +	Manually	Use the **attach_message** function for this. Use the appropriate type of this for any kind of response based messaging, unless you feel compelled to give a full screen message.
127) Where appropriate, Fatal/Warn/Info exit/screen	Consistent UI, user-friendly	Composr development +	Manually	Use the **whatever_screen** or **whatever_exit** function. Use the appropriate type of message for when you feel compelled to do a full screen message, or if the message is truly terminal (in the 'fatal' case here).
128) Where appropriate, include the **WARNING_BOX** template	Consistent UI, user-friendly	Composr development +	Manually	Places an on-screen warning about the action or content that a screen is providing.
129) Where appropriate, inline errors	Consistent UI, user-friendly	Composr development +	Manually	Use one of the standard CSS classes, such as **nothing_here** or **red_alert** or **inline_wip_message** to mark the error using standard Composr styles.
130) Where appropriate, access-denied screens	Consistent UI, user-friendly	Composr development +	Manually	Use the **access_denied** function as appropriate. It can be passed a language code, or a string. For Composr development, it should be a language code, so that the logs may be internationalised.

Specific guidance for input forms...

Standard	Reason	Scope	Test method	Notes
131) Use **FORM_SCREEN** for forms	Consistent UI, user-friendly	All development	Manually	Use a standard form template such as **FORM_SCREEN** (most cases) or **FORM** (if the form is just part of a complex screen). These templates provide automatic client-side field validation; server-side field validation is provided automatically also,

				via the standard field retrieval functions (e.g. **post_param_string**).
132) Use the form field API to generate input fields	Consistent UI, user-friendly	All development	Manually	Use our most appropriate standard input field types (e.g. **form_input_integer**), or creating a new type if you have a special case.
133) Use the standard method to split up large forms	Consistent UI, user-friendly	All development	Manually	Split complex forms into groups by using the **FORM_SCREEN_FIELD_SPACER** template to separate them. Hide groups of advanced functionality that is not often needed (often this functionality can fit nearly into a single "Advanced" group).
134) Use the standard method for showing groups of alternative fields	Consistent UI, user-friendly	All development	Manually	Use the **alternate_fields_set__start** system to mark alternative fields (i.e. the case where users may enter something in a field, or something in a different field, but never in both fields). Look at examples in existing code to see how it works.
135) Use **handle_max_file_size ($hidden)** when using upload fields so that your form has the right file-size limits on it	Stops nasty server-side upload-limit errors	All development	Manually	

Modifying core classes

Standard	Reason	Scope	Test method	Notes
136) Object orientated principles	Architectural integrity	Product development only		Use object-orientated principles properly where Composr does already. For example, forum drivers all have a database object as a property, rather than referencing the global **FORUM_DB** object. This is because they provide a core service and cannot make assumptions about the environment (e.g. it must be possible to have multiple simultaneous forum drivers running).

Blocks

Standard	Reason	Scope	Test method	Notes
137) List block parameters	User friendliness for webmasters	Product development only	Automated test	Define all block parameters in the block code itself (in the **info** function for the block) then document all block parameters via inserting the language strings in language files for the addon the block is a part of (see how it's done for existing blocks for examples).
138) Test blocks that have caching with and without it	Stability	All development	Manually	If a block has caching and you are only testing with caching disabled (the norm in development mode) it is possible that your blocks caching is completely misconfigured, which will cause stack traces.

Attachments

Standard	Reason	Scope	Test method	Notes
139) Remember posted Comcode might be invalid	Stability	All development	Manually	If you are supporting Composr attachments in a field, run the **check_comcode** function on the Comcode that references the attachments before the main row for the content is added to the database. This ensures the Comcode is valid, so you can proceed to add the content row (using the integer **0** temporarily as the field value for where the Comcode will be placed). The attachment Comcode can then be fully parsed (given the ID of the content row, so it can store its security properly) and then the content row updated with the new language string ID for that parsed Comcode.

Comcode parser/renderer code

Standard	Reason	Scope	Test method	Notes
140) No context-sensitivity	Stability	Product development only	Manually	When programming the Comcode system, never allow Comcode to become context-sensitive (e.g. behaving differently when run from different URL paths).
141) Don't use **`require_javascript`** or **`require_css`**	Cache safety	Product development only	Manually	When implementing tags, CSS must all be in **`global.css`**, and any JavaScript must be by using a symbol to load the JavaScript file – this is because any inclusion code put in the PHP rendering code only runs when the Comcode is first compiled, and not when it is loaded out of the cache.

Regular expressions

If you are writing regular expressions to extract data from HTML, you should anticipate...

Standard	Reason	Scope	Test method	Notes
142) ...there may be extra whitespace (there may be extra spaces inside tag definitions)				
143) ...whitespace may come in other forms than the **`\t`** symbol (there may be tabs)				
144) ...HTML used instead of XHTML. (there may be **` `** instead of **` `**)				
145) ...that tags may be written in upper-case				
146) ...attribute quotes can be given as single-quotes as well as double quotes, and attribute contents may span multiple lines				
147) ...there may be whitespace around tag-contents				
148) ...tag contents may not be properly entity-encoded				
149) ...quoted data may contain quotes of the other-type (there may be **`<a title="He's saying hello">`**)				

You **may** assume...

Standard	Reason	Scope	Test method	Notes
150) ...a tag opener/closer stays on one line				
151) ...attribute values are quoted				
152) ...the text is in either well-formed SGML (HTML) or well-formed XML (XHTML)				

Installation code

Standard	Reason	Scope	Test method	Notes

153) Installation code must respect the multi-language system	Internationalisation	Product development only	Manually	Make sure the **`lang_code_to_default_content`** function is used where appropriate in installation code. This function puts in language strings to the database such that they get stored automatically for all translations available on the installation.
154) Upgrades must be handled correctly	Stability	Product development only	Manually	Make sure that modules upgrade properly between major versions of Composr, and preferably between all versions. Do not assume a module will never be upgraded: if your module has undergone database structure revisions then you must make sure there is upgrade code to apply these revisions. See how existing code does it for examples.
155) Reinstalls must be handled correctly	Stability	Product development only	Automated test	If a module is reinstalled, it should not give errors about existing tables or privileges. For this to work, the uninstall function must fully uninstall what the install function creates.

Front-end programming

Spelling/grammar/lexicon

(examples of what not to do are in *italics*)

Standard	Reason	Scope	Test method	Notes
156) Don't spell things *rong*	User friendliness	All development	Use the **`.ini`** files with Microsoft Word's checking tools	
157) *Do Not Capitalise words just for the Effect*	User friendliness	All development	Manually	Capitalisation is for proper nouns (e.g. Charlie or Microsoft, but not box), for new sentences, or sometimes for sentence-like standalone phrases.
158) *Wrong to miss out words*	User friendliness	All development	Manually + use the .ini files with Microsoft Word's checking tools	In this example, "It is" is missing from the start of the sentence.
159) Be careful where you put your *apostrophe's*	User friendliness	All development	Manually + use the .ini files with Microsoft Word's checking tools	Apostrophes are not used for plurals.
160) Check readability from an outsider's perspective	User friendliness	All development	Manually	Triple check everything you write to make sure it is completely clear, as concise as reasonably possible, and cannot be misinterpreted.
161) Where appropriate use attractive HTML entities, not just ASCII	User friendliness	All development	Manually	Use HTML entities where possible, in particular **`ndash`**, **`mdash`**, **`hellip`**, **`ldquo`**, **`rdquo`**, **`lsquo`** and **`lrquo`**. These make the output that much nicer, and grammatically more correct (doing **`-`** instead of **`&ndash;`** is technically wrong, as a hyphen is not a dash).
162) Use terminology with consistency	User friendliness	All development	Automated test	Apply consistency to your terminology. Sometimes multi-word proper nouns are given hyphens (e.g. "multi-word" , sometimes concatenation is used (e.g. "multiword" , sometimes abbreviation is used by just using the first letters placed together as capitals (e.g. "MW" , and sometimes each word is given in capitals (e.g. "Multi Word" . All this inconsistency is ingrained in English, but for any single term, reference it in a uniform way. We do not usually use the antiquated form of abbreviations using dots (e.g. "M.W." .
163) Don't pollute **`global.ini`**	Performance on real live sites	All development	Manually	If phrases are not general to the rest of Composr, then put them in a language file that can be individually included by a module.
164) Most lang strings support HTML, some do not: be aware	Stability	Product development only	Automated test	You may assume language strings can contain HTML. There are only a few exceptions where this is not true. Theoretically the burden usually lies on code removing HTML formatting that is there and can't be used, rather than the other-way-around. Practically, there are some cases where HTML entities are allowed but not HTML tags (e.g. if a language string is used for a **`title`** attribute), and you will just need to use your judgement and test if you change an existing language string to include HTML tags. The flip-side to HTML being allowed is that you must encode things like the ampersand symbol as entities.

165) When needed, add comments to the language files to define context	Internationalisation	Product development only	Manually	For an example of how to do this, see **`global.ini`**.
166) Minimise adding new language strings	Internationalisation	Product development only	Manually	Try to reuse strings as much as possible. Don't add new strings unless it is an improvement, as they need to be translated into potentially dozens of languages. To facilitate this, phrases should be made as general as reasonable… i.e. avoid using context where possible (e.g. use **`Added entry {1}`** rather than **`Added download {1}`**). Don't do this if it creates a usability problem though.
167) Don't use the name "Composr", use the phrase "the software"	De-branding	Product development only	Automated test	
168) Use **`<kbd>`** within language strings as appropriate	User friendliness	Composr development +	Manually	Use this for something typed, or for code.
169) When referencing key names, refer to them like Ctrl or Shift or Option or Alt: capitalise the first letter, don't use quotes, and don't use **`tt`**/**`kbd`**	Consistent UI	Product development only	Manually	
170) Don't use Composr-specific terms on the main website	Consistent UI	Composr development +	Manually	Composr 'brands' or unique terminology ('Zone', 'Comcode', …) should not be seen on the main website.
171) Use our standard terminology	Consistent UI	Composr development +	Manually	Use correct terminology like 'Codename', 'Option', 'Title', 'Description'.
172) End form field descriptions with a full stop	Consistent UI	Composr development +	Composr development mode	

Respect the Queen of England…

Standard	Reason	Scope	Test method	Notes
173) Do not use American spellings such as *color* or *center* or *license*	Consistent UI	Product development only	Automated test	Composr originates from the UK. We have special processing to support American English automatically.
174) Use British English standard for mixing full stops and brackets	Consistent UI	Product development only	Automated test	Full stops and brackets mix so that the full stop follows the closing of the bracket. For example: example (example).
175) Say "tick (check)" rather than just "tick" or just "check"	Internationalisation	Product development only	Automated test	The word "tick" in British English is "check" in American English (referring to the square 'tickbox'/'checkbox' on HTML forms). Due to this disparity, write both like "tick (check)".

Language for submit buttons…

Standard	Reason	Scope	Test method	Notes
176) For an add button, use the title text ("Add xxx")	Consistent UI	Product development only	Manually	
177) For an edit button, use the language string **`SAVE`** ("Save")	Consistent UI	Product development only	Manually	Only use the language string **`EDIT`** ("Edit") when you are distinguishing it against the language string **`DELETE`** ("Delete").
178) For a delete button use the language string **`DELETE`** ("Delete")	Consistent UI	Product development only	Manually	Usually these are actually ticks for an edit screen, but sometimes we do have them. If it's more of a removal than a deletion, the language string **`REMOVE`** ("Remove").
179) For an intermediary button, use the language string **`PROCEED`** ("Proceed")	Consistent UI	Product development only	Manually	
180) For sorting, use the language string **`SORT`** ("Sort")	Consistent UI	Product development only	Manually	
181)	Consistent UI	Product development	Manually	For changing the number of items shown, use the

Use **CHANGE** and **PER_PAGE** where appropriate		only		language string **CHANGE** ("Change"), or use the language string **PER_PAGE** ("Per page") if you are doing it as a sentence.

Markup/Tempcode

Standard	Reason	Scope	Test method	Notes
182) Use tabs for HTML code	Smaller streaming size	Composr development +	**.editorconfig**	HTML code (e.g. in templates) should be written using tab for indentations rather than spaces.

183)
Take into account that content may not all be filled in
	Tidy UI	All development	Manually	Remember that if something may be blank (e.g. if it is possible for a field to not be filled in, or for there to be no entries in a list) you need to accommodate for that possibility. Sometimes no changes are required, but often if there are headings or boxes used to contain it then you'll want to wrap around the **IF_NON_EMPTY** directive so as to avoid empty headings/boxes. For example to stop an empty paragraph element producing an accessibility problem when **SOMETHING** is blank, guard it with **IF_NON_EMPTY** as follows:
				Code
				`{+START,IF_NON_EMPTY,{SOMETHING}}`
				`  <p>{SOMETHING*}</p>`
				`{+END}`

184) Make templates beautiful	Easier for other programmers to understand	All development	Manually	Indent Tempcode HTML neatly. Each multi-line Tempcode directive or HTML tag should have its contents indented an extra level. Indentation is for the purposes of people reading/modifying the templates, not the final HTML, so indent with this in mind (i.e. Tempcode indenting is fine even though it'll make the final HTML look messier). The browser DOM inspector is how people view outputted HTML nowadays and that does its own automatic indentation.
185) Use the Tempcode escaping syntax correctly	Security	All development	ocProducts PHP version	For example, if you are outputting a textual parameter, make sure to reference it as **{PARAM*}** rather than just **{PARAM}**. Give big thought to this if you are outputting the variable into JavaScript code: usually you'll want to use the **%** escaper which locks down data very tightly.
186) Meet WCAG (Web Content Accessibility Guidelines)	Accessibility	Composr development +	Code Quality Checker	Meet WCAG (Web Content Accessibility Guidelines) 1 level 3, and WCAG 2 (read them!). Write semantic markup. Always use the **<p>** element to mark true paragraphs if they are not alone in a table cell, and if they are always going to be just one single paragraph. Images should not be wrapped in paragraphs. Short lines may be wrapped in paragraphs as/if you wish.
187) No deprecated tags or attributes	Accessibility	Composr development +	Code Quality Checker	Don't use deprecated HTML attributes or tags (e.g. the **align** attribute, the **b** tag). Use a combination of CSS and semantic tags instead (e.g. **text-align** and the **strong** tag). This is part of WCAG, but needed re-iterating.
188) Write to XHTML5	Accessibility	Product development only	Firefox HTML validator extension / Code Quality Checker / Template previews combined with the unit test that validates them all	We don't actually output XHTML mime-type, or write JavaScript as XHTML-compatible, but we do have the actual markup be XHTML-compatible for better readability and tooling-parsablity.
189) Don't make assumptions about word order by coding things like **{!GO} {!HERE}**	Internationalisation	Product development only	Manually	Word order in languages may vary.
190) Don't use native language directly in templates or PHP code	Internationalisation	Product development only	Manually	However, colons and so forth may be used in templates in a visual way (even though strictly it is an Englishism).
191) Use the **alt** and **title** attributes correctly on images	Accessibility	Composr development +	Manually	If you want to have a tooltip for an image (perhaps the image meaning is not obvious), use the title tag. The alt tag is intended for alternate text when the image itself is not viewable. This means, if you want

				a tooltip, you must have both alt and title attributes (**`alt`** is always required for accessibility).
192) Avoid the **`style`** attribute ("inline styles")	Easier for a web designer to work with	Product development only	Code Quality Checker	Don't use the HTML style attribute except where you need to pass in styles contingent on template parameters.
193) Use ASCII characters only – HTML entities for the rest	Internationalisation	Product development only	Code Quality Checker	Don't use any extended-ASCII characters in templates, as you cannot assume what character set will be in use.
194) Only forcibly open up links in new windows in certain conditions	Accessibility	Product development only	Manually	Only forcibly open up links in new windows in these conditions: ▪ The action is strictly auxiliary to an ongoing action (e.g. opening up documentation to read while performing the action, or opening a small pop-up window) ▪ An ongoing action is being performed that will spawn many followup actions that are not going to flow in sequence ▪ It is a link to an external website on the main website placed in a prominent position (e.g. link to RSS news story) To be clear, do not forcibly open up links in these conditions (or any other not mentioned above): ▪ When it's a link to an external site not in a prominent position on the main website ▪ The action is arguably auxiliary but often may not be ▪ When moving between non-strongly related resources (e.g. clicking on a username in the forum-view) Do not provide a choice of "open in new-window" and "open in same-window" links to the user – their web browser already gives them this choice.
195) Use **`{!LINK_NEW_WINDOW}`**	Accessibility	Composr development +	Code Quality Checker	When links are sent to new windows, remember to use **`{!LINK_NEW_WINDOW}`** in the link title attribute. The purpose of the link should precede **`{!LINK_NEW_WINDOW}`**, so that the link title makes sense when read out alone.
196) Don't use tooltips for critical information	Mobile	Composr development +	Manually	Touch screen devices can't see them.
197) Either place checkboxes before labels, or place a colon after the label	Accessibility	Product development only	Code Quality Checker	Otherwise users may be confused as to how things are laid out.

Guidance for tables...

Standard	Reason	Scope	Test method	Notes
198) Don't use layout tables	Accessibility	All development	Manually	If you are showing tabular data – e.g. a map table or a columned table, it's always correct to use a table – if it can have sensible **`th`** tags, it's a proper table.
199) Use the **`results_table`** CSS class on most tables	Consistent UI	Composr development +	Manually	**`standard_border`** is formalised for other kinds of "Map table" or "Columned table", or just about any other kind of table.
200) Use the **`th`** tag properly, and also **`thead`** / **`tbody`**	Accessibility	Composr development +	Manually	Use the **`th`** tag to define headers. If you have a case where there are top-level and secondary-level headers, put the top-level header in **`thead`** and also make use of **`tbody`**.

CSS

Standard	Reason	Scope	Test method	Notes
201) Follow naming and layout standards	Consistency	Product development only	Manually	Use the same tabbing style, bracing style, naming style, use of quotes, and way of layout out attributes, as existing CSS code broadly is using.
202) Avoid setting heights	Accessibility	All development	Manually	If you set heights then if people increase their font size (or a translation uses longer/more words) then

				it's likely text will overflow and look awful. There's rarely a true reason to set a height anyway: if you need to clear floats use **`float_surrounder`** (described elsewhere in this document).
203) If possible avoid setting widths or heights	Easy of making changes / Compatibility	All development	Manually	Do not assign widths or heights or min-widths or min-heights unless necessary (e.g. to make sure you align with a background's size). It makes it very hard to adapt designs later if the spacings change and it makes pixel-level browser bugs cause larger problems.
204) If you mix percentages and pixels, use **`box-sizing`**	Stability	All development	Manually	An element with **`100%`** width and **`2px`** padding will actually use more than 100% of the available space. Don't hack around this problem by setting percentages to be slightly smaller: instead use **`box-sizing: border-box`** to change this behaviour to what you'd expect).
205) Perform browser testing	Compatibility	All development	Manually	Test on all the browsers that we support.
206) Extra CSS files should only define specific stylings	Stability	All development	Manually	CSS defined outside **`global.css`** and **`nocache.css`** must not have wide-sweeping effects (it must be modular and target specific elements that would not normally exist on arbitrary screens).
207) Do not float everything	Stability	All development	Manually	Do not use floats unless doing it to make stuff sit next to something else. a. do not float stuff just to make it not overlap with another float b. do not float stuff to try and work around IE bugs Use a **`float_surrounder`** CSS class like Composr does (i.e. **`overflow: hidden`**) around stuff that is floated and what it is floated next to, to contain them without affecting other content. It is a much better stable solution.
208) Do not use background images unless actually a background	Compatibility	All development	Manually	Do not use background images unless an image really is in the background. It is harder to implement (needs CSS widths/heights setting, which is inflexible) and we can't assign alt text. Use **`img`** tags instead.
209) Do not use a CSS reset	Compatibility	All development	Manually	Composr does not use a CSS reset in its default templates so you would need to re-theme all Composr screens to re-apply default spacings.
210) Use meaningful naming for CSS classes	Easier for a web designer to work with	Product development only	Manually	Name CSS classes according to what they are for and not what they do (e.g. **`big_and_red_page_icon`** is a terrible name, but **`warning_icon`** is a good one).

211)
Don't use the **`clear`** property
	Needed for the way our default theme is built	Composr development +	Manually	We know this is a very common technique, but we don't use it in Composr. Instead, wrap all your floats inside...
				Code (HTML)
				`<div class="float_surrounder">`
				`  (floats)`
				`</div>`
				It works a lot better, is easier to understand, and the way our theme is built, it's necessary.

212) Put your CSS neatly into the most appropriate CSS file	Easier for a web designer to work with, and performance	Composr development +	Manually	Don't put all your code into **`global.css`**. If you have lots of styles relating to a new feature, give a neat comment formatted exactly the same as other formats in the file.
213) Keep CSS minimal and simple	Easier for a web designer to work with, and performance, and stability	Composr development +	Manually	The key to writing good CSS is understanding. Understand your way through how browsers render CSS, and the quirks of particular browsers. Never ever write CSS by adding or hacking with properties until it works.
214) Don't leave cruft lying around	Easier for a web designer to work with, and performance	Composr development +	Automated test, or Firefox "Dustme selectors" addon	Don't leave old unused styles in the CSS files.
215) Don't break the Theme Wizard	The Theme Wizard needs to keep working	Product development only	Automated test	If colours are added to the CSS then a proper Theme Wizard colour equation should also be added (see how it works for other colours for examples). The best thing to do is usually to re-use an existing colour definition – not only does that make the

				colour palette consistent, it stops you needing to calculate a new equation. We made our calculations using Photoshop, by adding white/black layers above the **`seed`** color and adjusting its opacity.
216) Never allow default border colours	The Theme Wizard needs to keep working	Product development only	Manually	The default border colour is black, which doesn't look great on our default theme, but may look far worse on other themes. Always set a colour for borders and follow the above rule so that the Theme Wizard can set the colour. To find a border colour, just grab the code from an existing bit of CSS that performs a similar styling to what you want.
217) Use relative font sizes	Accessibility	Composr development +	Code Quality Checker	We want output to be easily scalable, so write sizes using **`em`**. It's really easy: if you want something smaller, use something like **`0.85em`** and something bigger like **`1.1em`**. The advantage to scaling is a themer can easily knock the whole theme's font sizing up, and also it is possible to proportional down-scale content for embed contexts. There is a tool to convert the default theme to **`px`** units for designers who prefer it (professional designers have good reason to want to do pixel-perfect designs).
218) Designs must work on 1024x768 with 2 panels, and up	Compatibility	Composr development +	Manually	
219) Front-end designs should work on mobile mode	Mobile	Product development only	Manually	

JavaScript

Standard	Reason	Scope	Test method	Notes
220) Follow framework standards	Consistency	Product development only	Manually	The JavaScript framework defines a way of writing objects, hooking in functionality (mainly via data attributes), querying the DOM tree, and naming variables – follow the existing standard of the code. We broadly follow the same kind of conventions other JavaScript developers do, but with a focus on modern ways of doing things wherever possible (as JavaScript is historically a very messy language). You do not need to define jsdoc. We do not use any JavaScript preprocessors or pre-minification (automatic minification is done by Composr).
221) Use official DOM (so no **`document.all`**, or **`window.images`**)	Standards compliancy	Product development only	Code Quality Checker	**`document.all`**, **`form.foo`**, etc, are all non-standard (even though they're commonplace) – use **`getElementById`** etc.
222) Don't use any deprecated methods (so no **`window.escape`**)	Standards compliancy	Product development only	Code Quality Checker	Instead of window.escape, use **`window.encodeURIComponent`**.
223) Do not use inline JavaScript	Security	Composr development +	Code Quality Checker	We use CSP to avoid XSS vulnerabilities, so inline JS is blocked.
224) Fix any strict warnings	Standards compliancy / Future proofing	Product development only	JavaScript files should have **`use strict;`** in	Ideally also don't make assumptions about **`undefined`** and **`null`** being the same as **`false`**, explicit code is easier to read; don't reference undefined variables without using **`typeof`** as otherwise browsers with strong notice reporting will give out noise.
225) Don't use the browser **`alert`** / **`prompt`** / **`confirm`** / **`showModalDialog`** UI's – use Composr's **`fauxmodal_*`** APIs	User friendliness	Composr development +	Manually	Better styling, no issues with popup blockers, no issues on full-screen UIs and tablets with poor window management.
226) Don't rely on **`window.focus`** / **`window.print`** / **`window.scrollTo`** / **`window.scrollBy`** / **`window.resizeTo`** / **`window.resizeBy`** / **`window.moveTo`** / **`window.moveBy`** existing	Stability	Composr development +	Manually	Popup blockers may remove this functionality, setting them to null. You must catch exceptions over the calls.
227)	Standards compliancy	Product development	Manually	A minor point, but if we capitalise our technology

It's written JavaScript not Javascript		only		names incorrectly we look less professional.

CSS and JavaScript

Standard	Reason	Scope	Test method	Notes
228) Use comment encapsulation	Syntax highlighting	Product development only	Unit test	Tempcode directives should use comment encapsulation so that syntax highlighting in text editors works.

Images

Standard	Reason	Scope	Test method	Notes
229) Do not refer to theme images directly by URL	The webmaster may change image mappings	All development	Manually	Bundled images should be referred to via the theme image system, not by direct URL.
230) Usually use PNG files	Performance	All development	Manually	Use PNG images in most cases, GIF when animation is needed, or JPEG if there is zero chance of the image needing any transparency and never likely to be edited again (i.e. only use it at a post-development optimisation stage) and it compresses better as JPEG.
231) Compress PNG images and strip gamma	Performance and browser compatibility	All development	Automated test checks compression ratios	Pass PNG images through a compression program such as PNGGauntlet or PNGThing, and strip gamma settings in the process (needed for colour compatibility on some browsers).
232) Use inlining correctly	Performance	Composr development +	Manually	If an image is referenced in the CSS and used very commonly, consider using **{$IMG_INLINE,...}** so that it loads without any additional HTTP request.

Miscellany

Testing

Standard	Reason	Scope	Test method	Notes
233) Design tests first or write automated unit tests after	If we re-factor code, we know what to re-test	All development	Manually	Before writing any code, you could design/document the tests you are going to run to make sure your written code works. Think of how it might break under situations you might have otherwise forgotten about (e.g. what if a guest user is running your module) and add in tests to test that it doesn't break. Use your judgement.

Documentation

Standard	Reason	Scope	Test method	Notes
234) Provide documentation for new features	So people know how to use the features	Product development only	Manually	After your code is written and tested make sure that the user documentation for Composr is updated (edit a **tut_*** file in **docs/pages/comcode_custom/EN**).
235) List new features	So we can communicate changes with users	Product development only	Manually	If you've added a non-trivial feature it will need to be added to the Composr website's feature-list eventually (**composr_homesite_featuretray.php**), and also announced.

General guidelines

Standard	Reason	Scope	Test method	Notes
236) Think of Composr as a whole, keep	Keeps everything strong, as a consistent whole	Product development only	Manually	For example, Chris once found that **XMLHttpRequest** on Internet Explorer had a

it consistent				caching bug... so instead of just throwing in a solution to where it affected him, he considered the problem architecturally, making sure all AJAX-facing PHP scripts have a common approach to solving the problem. This kind of approach is essential.
237) Update credits file	Legal / Ethics	Product development only	Manually	If you include some third party code you must update **`licence.txt`** to reference it properly. This file is shown when users install.
238) Use text-based file formats	Revision control	Composr development +	Manually	Use text-based file formats rather than binary documents (like **`xlsx`** or **`docx`**). Revision management and global search and replace works much better with text files.

Agency standards

Standard	Reason	Scope	Test method	Notes
239) Add new CSS/templates/images into the default theme **`*_custom`** dirs	Deployment, collaboration	Projects	Manually	If you decide to switch themes, or preview from the Admin Zone, you want to know your functionality will still work.
240) Use a staging server system	Deployment, collaboration	Projects	Manually	The Staging Servers tutorial lays out processes for deploying in a professional way.
241) Use a good upgrade and maintenance process	Deployment, collaboration	Projects	Manually	The Professional upgrading tutorial lays out a good process for maintaining and upgrading websites.

Feedback

Please rate this tutorial:

Have a suggestion? Report an issue on the tracker.