BLACKBOARD OPEN LMS APPROVED PLUGIN PROGAM

Code Review Guidelines

SEPTEMBER 26, 2018

I. Introduction

In order to provide the most stable and secure infrastructure for its clients, the Blackboard Open LMS Development Team reviews all third party code before it can be included into the central codebase used by all Blackboard Open LMS hosted instances. Blackboard has designed and developed these Code Review Guidelines to assist client developers in creating custom code that will have a high probability of passing Blackboard Open LMS code review for eventual inclusion to the central codebase. This document describes each of the main areas of the review process, covering in as much technical detail as possible, any pitfalls for development. Below is a list of the topic areas that Blackboard uses to review code:

- Introduction
- Structure and coding practices
- PHP Tags
- Security
- Performance
- Moodle Core codebase changes
- Continual Integration
- Installation and run time
- Language files
- Styles
- Markup
- Accessbility
- File system interactions
- Moodle Cron
- Key customization files
- Recommended sample size
- Deprecated Function Removal
- Glossary of terms

II. Structure and coding practices

The most common way a code review to fail is to not follow the general Moodle development guidelines found at http://docs.moodle.org/en/Development:Coding. The customization structure should follow the proper guideline laid out by Moodle at http://docs.moodle.org/en/Development:Plugins. Customizations not properly modularized will automatically fail review and most likely will not install on the Blackboard Open LMS hosted instance. The customization should also follow Moodle's coding style found at http://docs.moodle.org/en/Development:Coding_style.

A great way to start a customization to Moodle is to use the plugin skeleton generator.

https://docs.moodle.org/31/en/Plugin_skeleton_generator

The worst way to make a customization to Moodle is to hack core code - it is very likely that a customization will not be approved if it employs core hacks.

Moodle has an excellent tutorial on module development found at http://docs.moodle.org/en/Development:NEWMODULE_Documentation as well as a development course located at http://dev.moodle.org/.

PHP naming collisions should be avoided by:

- Using namespaces - https://docs.moodle.org/dev/Automatic_class_loading#Namespaces
- Encapsulating your functions in classes - e.g. as opposed to having a list of functions in a locallib.php file, create a classes/local.php file with your functions inside the 'local' class.
- Only using the lib.php file in your plugin for api functions, which are typically prefixed by the plugin name - e.g.

```
<?php
function collaborate_add_instance(stdClass $collaborate, mod_collaborate_mod_form
$mform = null) {
```

Javascript naming collisions should be avoided by:

- Using AMD modules
- Scoping globals at the module or function level (global variables should not be used unless absolutely necessary)

CSS naming collisions should be avoided by:

- Name spacing via the body path class - e.g. .path-mod-mymod .users. Note that this will work fine for module specific pages but not for plugins that output markup outside of their own pages.
- Prefixing via the plugin name - e.g. .filter_imageopt_container

Follow the JavaScript guidelines carefully. (REF: https://docs.moodle.org/dev/Javascript_FAQ and https://docs.moodle.org/dev/Javascript/Coding_Style)

Note: at the time of writing, the coding style document on moodle.org is out of date and refers to YUI modules. Please note, this is out of date and bad advice.

Almost all JavaScript code should be in separate .js files. There should be the smallest possible amount of JavaScript inline in the HTML code of pages, ideally there should be no inline JavaScript. These .js

files should not be included directly, but should utilize $PAGE->requires. The recommend way of doing this is by writing them as an AMD module

```
$initvars = [$PAGE->context->id, $COURSE->shortname];
$PAGE->requires->js_call_amd('mod_mymod/myamdmodule',
'function_within_amd_module', $initvars);
```

Note that all new code should use AMD modules, all old code using YUI modules or scripts that are included directly should be converted to use AMD modules.

As it is really easy to pass parameters into your AMD modules, there should be little to no reason for using inline javascript.

Any 3rd party javascript libraries that are shipped with core (e.g. requires / jQuery) should not be duplicated in your module or referenced via a linking script tag.

The ideal way to include jquery in a script is via an AMD module - e.g:

```
/**
 * Main responsive video function.
 */
define(['jquery'], function($) {
```

Or, to support non AMD scripts / inline scripts:

```
$PAGE->requires->jquery();
```

III. PHP Tags

First and foremost, PHP short tags like '<?=' are not allowed. See short_open_tag in http://php.net/manual/en/ini.core.php

As per the coding style guidelines PHP files should start with "<?php" and should not end with a "?>" as it can cause white space problems. If at all possible, remove trailing "?>" from code before accepting. If

this is not possible, you must verify that there are no white space problems. Running the following command will find any potentially problematic PHP close tags:

```
# Change the first . to a directory path to narrow searching, EG:./path/to/plugin
or just cd to the correct directory.
> find . -iname '*.php' | xargs pcregrep -M '\?>\s\s+\Z'
```

An example of a bad closing PHP tag is a "?>" followed by two new lines. Any bad ones **should** be fixed. If any of the bad ones are in or included by primary core files (block class file, lib.php file, lang file, etc) then it **must** be fixed prior to accepting the code.

In addition, if any files have any white space before the opening PHP tag, then this **must be fixed** as well. Running the following command will find any potentially problematic PHP opening tags:

```
# Change the first . to a directory path to narrow searching, EG: ./path/to/plugin
or just cd to the correct directory.
> find . -iname '*.php' | xargs pcregrep -Mn '^\s+<\?php'
```

Note that the above command has the -n option enabled. This will print out the line number of the matching line. This helps because we only care about ones near the top of the file.

Reasons why code would be rejected for violating these rules:

1. It is breaking the coding style guidelines set by Moodle.
2. These new lines can break critical functions in Moodle, like downloading files, AJAX requests, disrupt the sending of HTTP headers, etc. These breakages can even happen outside of the offending plugin. Example: Moodle commonly loads all lib.php files for hooks. If an offending plugin has a broken lib.php, then this can affect global operations of Moodle.

IV. Security

Customized code should follow the security guidelines Moodle has defined at http://docs.moodle.org/en/Development:Security. Blackboard specifically reviews code to determine the following:

1. Cleaning of input parameters
2. Permissions
3. Exposure of data

4. Variable preparation

CLEANING OF INPUT PARAMETERS

Code should not directly access the $_POST, $_GET or $_REQUEST PHP variables or use them in regular expression searches without properly cleaning the information submitted. If code doesn't clean data being submitted then the program becomes vulnerable to cross scripting attacks and SQL injection attacks. Moodle has developed several standard functions that clean input parameters for the developer. These are:

- required_param()
- required_param_array()
- optional_param()
- optional_param_array()
- clean_param()
- clean_param_array()

Each of these functions check the $_POST, $_GET or $_REQUEST PHP variables for the specific variable the developers is looking for and then cleans based on the format provided. If a developer is expecting a number to be passed for a variable then required_param(), optional_param(), and clean_param() will only return a number as a value for the variable requested. These functions should be used directly after the config.php file is included in a script.

Ex: view.php

```
<?php

require_once(../../config.php);

$id = require_param('id',PARAM_INT);

$user = optional_param('user','joe',PARAM_TEXT);

// note that ?> is intentionally left off in Moodle 3.X Code
```

PROPER USE OF PARAMETERS IN THE DATABASE LAYER

Moodle's database layer mimics prepared statements in order to add parameters safely to your query. When this is not used, then you run the risk for SQL injection attacks. Example of how **not to do it**:

```
<?php
```

```
// Do setup, etc.
$DB->execute('SELECT * FROM {course} WHERE shortname = '.$shortname);
```

So, the above is subject to an SQL injection attack because the $shortname might contain something dangerous and is not properly escaped. Here is the **right way**:

```
<?php
// Do setup, etc.
$DB->execute('SELECT * FROM {course} WHERE shortname = ?', array($shortname));

// Or with named parameters.
$DB->execute('SELECT * FROM {course} WHERE shortname = :shortname',
array('shortname' => $shortname));
```

AVOID PARAM_RAW

PARAM_RAW means no cleaning is done and should be avoided unless there is a very good reason for its use.
Explanations should be written into comments if such is the case.

PERMISSIONS

When it comes to permissions there are three areas that Blackboard reviews for code violations:

1. Require login
2. Proper use for session key
3. Validation of capabilities

If not implemented properly, the customization could allow users to access data they should not be allowed to.

Require login

Blackboard Open LMS requires code to validate that the user is logged into the system before the user is given acces to add, update or delete data in Moodle. Moodle provides the `function require_login()` to simplify the process of determining if the has logged in previously. If they user has not logged in the the Moodle login page is shown rather than the form or data the user was looking to access. This

function should be included after parameters have been cleaned or the course module is obtained, but before any data is displayed.

Typical example:

```
<?php
require_once("../../config.php");
// more libs required as necessary ....

$id = required_param('id', PARAM_INT);    // Course Module ID
$action = optional_param('action', '', PARAM_ALPHA);
$edit = optional_param('edit', -1, PARAM_BOOL);

if (!$cm = get_coursemodule_from_id('sampleactivity', $id)) {
    print_error('Course Module ID was incorrect');
}
if (!$course = $DB->get_record('course', array('id'=> $cm->course))) {
    print_error('course is misconfigured');
}
if (!$certificate = $DB->get_record('sampleactivity', array('id'=> $cm-
>instance))) {
    print_error('course module is incorrect');
}

require_login($course->id, false, $cm);

// Obtain Context ...

$context = context_module::instance($cm->id); // use the proper context classes
for each context
// Ex. context_course::instance($courseid), context_coursecat::instance($catid)
```

```
require_capability('mod/sampleactivity:view', $context);


// ... rest of code ....
```

Proper use of session key

Moodle provides a session key that is linked to the user's login for the duration user is accessing the system. Blackboard Open LMS requires that code always send the session key using the `sesskey()` function in HTML forms, and interfaces that can modify data in Moodle. Blackboard Open LMS also requires that the code validate the session key using the functions `confirm_sesskey()` or `require_sesskey()` before allowing the user to add, update or delete data from Moodle. Moodle's moodleform class automatically adds the sesskey to it's rendered form and validates it on submit.

Example of `sesskey()` function

```
// A URL
echo "<a href=\"$CFG-
>wwwroot/some/script.php?sesskey=".sesskey().'">'.get_string('delete').'</a>';


// As part of a form that is being manually created
echo '<input type="hidden" name="sesskey" value="'.sesskey().'" />';
```

Example of `confirm_sesskey()`

```
<?php
    require_once('../../config.php');


    require_login($SITE->id);


    if (confirm_sesskey()) {
        // Execute code that requires a valid sesskey in the request...
```

```
    } else {
        // Execute code when invalid sesskey is passed, typically something like
the following line:
        throw new moodle_exception('confirmsesskeybad', 'error');
    }

    $context = context_system::instance();
```

Example of `require_sesskey()`

```
<?php
    require_once('../../config.php');

    require_login($SITE->id);
    require_sesskey();

    $context = context_system::instance();
```

Validation of capabilities

Blackboard will review any new capabilities or changes to existing capabilities to determine the level of access granted and use in the customization. Blackboard will review that the code is properly calling Moodle's capability functions `has_capability()` and `require_capability()`.

Example of `require_capability()`

```
<?php
    require_once('../../config.php');

    require_login($SITE->id);
    require_sesskey();
```

```
    $context = context_system::instance();
    require_capability('moodle/legacy:admin', $context);
```

Example of `has_capability()`

```
<?php
class block_blockname extends block_list {
    function get_content() {
        global $CFG, $COURSE;

        if ($this->content !== NULL) {
            return $this->content;
        }

        if (empty($this->instance)) {
            return $this->content;
        }

        $this->content = new stdClass;
        $this->content->items  = array();
        $this->content->icons  = array();
        $this->content->footer = '';

   $context = context_course::instance($COURSE->id);

        if (has_capability('block/mpower:edit', $context)) {
```

```
        $this->content->items[] ='<a href="'.$CFG-
>wwwroot.'/blocks/blockname/edit.php?'.

$type.'='.$id.'">'.get_string('edit','block_blockname').'</a>';
            $this->content->icons[] = '<img src="'.$CFG->pixpath.'/i/edit.gif"
class="icon" alt="'.
                                        get_string('edit','block_blockname').'" />';
    }
     }
}
```

Moodle allows developers to inform administrators of the security risk a new capability can create if a user is granted the permission. Blackboard will verify that these "risks bits" properly inform administrators of the dangers of the capability. Moodle defines the following risk bits (http://docs.moodle.org/en/Development:Hardening_new_Roles_system):

- **RISK_SPAM:** user can add visible content to site, send messages to other users; originally protected by !isguest()
- **RISK_PERSONAL:** access to private personal information - ex: backups with user details, non-public information in profile (hidden email), etc.; originally protected by isteacher()
- **RISK_XSS:** user can submit content that is not cleaned (both HTML with active content and unprotected files); originally protected by isteacher()
- **RISK_CONFIG:** user can change global configuration, actions are missing sanity checks
- **RISK_MANAGETRUST:** manage trust bitmasks of other users
- **RISK_DATALOSS:** can destroy large amounts of information that cannot easily be recovered.

Example of proper use of `riskbitmask`

```
<?php
$blockname_capabilities = array(

    'blockname/permissionname' => array(
        'riskbitmask' => RISK_PERSONAL,
```

```
        'captype' => 'read',
        'contextlevel' => CONTEXT_COURSE,
        'legacy' => array(
            'teacher' => CAP_ALLOW,
            'editingteacher' => CAP_ALLOW,
            'admin' => CAP_ALLOW
        ),
    )
);
```

VARIABLE PREPARATION

Blackboard reviews all code to verify that data being inserted into the database is properly sanitized and all strings are escaped. Proper sanitization and escaping of data saved in the database removes the possibility of SQL injection attacks. Moodle, by default, will sanitize and clean Moodle forms if properly created. To sanitize data in a Moodle form, use the `setType()` function.

```
<?php
require_once($CFG->dirroot.'/course/moodleform_mod.php');


class mod_modulename_mod_form extends moodleform_mod {


    function definition() {
   $mform    =& $this->_form;


        $mform->addElement('text', 'name', get_string('topic', 'mldulename'),
array('size'=>'64'));
        $mform->setType('ending', PARAM_TEXT);
    }
}
```

Not all activities in Moodle will sanitize data before the data is saved in the database. Because of this, Blackboard Open LMS recommends always sanitizing data before displaying to the user. Blackboard validates that the developer properly uses the `format_string()` and `format_text()` functions to sanitize display data. These functions not only sanitize the string before printing, they also properly execute the Moodle filters before the string or text is displayed. This helps to keep a consistent level of user functionality within all features of Moodle, while making sure users are not entering malicious code into their submissions.

```
<?php
$mycourses = "<li><a $linkcss title=\"" . format_string($course->shortname) . "\"
".
             "href=\"$CFG->wwwroot/course/view.php?id=$course->id\">" .
             format_string($course->fullname) . "</a></li>";
```

V. Performance

Blackboard reviews all code for performance issues in order to verify that with the installation of the code, Blackboard will maintain its service level agreement with clients. When reviewing performance issues, Blackboard focuses on the following topic areas:

1. Database communication
2. JavaScript requirements

DATABASE COMMUNICATIONS

Interactions with the database are one of the main causes for slow performance of a server. When reviewing code, Blackboard checks whether any of the following questions are positive:

- **Does the code excessively query the database?**

  Excessive queries are usually design flaws, and Blackboard Open LMS developers will suggest either alternative designs to decrease the the number of requests or an alternative sql that will provide the same data. When reviewing code, look for loops with sql queries inside and test all code with a large volume of data in tables being queried. Blackboard recommends testing code with a large volume of data; see Recommended Sample Sizes.

- **Does it make a very large DB call?**

Blackboard looks for single queries of a large number of tables, queries that don't properly use tables' indexes, or queries that return a large number of columns or rows. These types of queries can often times perform acceptably with a small data set, but not when run with a larger production data set. When reviewing code, look for queries that link to more than 3 tables, return more than 30 columns and require processing of more than 100 records in a single instance. Moodle offers database functions to limit rows returned from a query and display data in a paged format. These should be used whenever displaying more than 50 rows of data to a user. Blackboard recommends testing code with a large volume of data; see Recommended Sample Sizes. For optimal performance, the function get_recordset should be used as opposed to get_records. The function get_records should only be used where it is guaranteed or highly likely that the result set will be small.

- **Does it make proper use of table indexes?**

Developers are able to add indexes to tables used by the code. Blackboard reviews all SQL code to determine if indexes need to be added to improve query performance. Most ID fields need to be indexed in order to do quick look ups or to combine tables. It is recommended that any select query should have the fields in the where clause properly indexed.

- **Does the code directly query the database and not use Moodle database functions?**

Moodle provides several functions to request data from the database. These functions should be used instead of executing the PHP MySQL functions. Moodle abstracts the database layer in order to allow for greater database vendor support, which in turn allows Blackboard to be flexible in its server construction and design.

JAVASCRIPT DOM MANIPULATION

- **Does the JavaScript excessively modify/process the HTML code?**

JavaScript code can be used to modify existing HTML code or add new HTML code to an already loaded web page. Searching and changing the HTML code after the browser has loaded the page, if done too often, can delay the load of the page, as the user's computer is generating the HTML code. Javascript which manipulates the DOM should be called once the DOM is ready, ideally by using the jquery function $(document).ready().

VI. Moodle Core codebase changes

Blackboard Open LMS does not accept third party code that has modified the Core Moodle codebase, unless no other way exists to get the customization to work and the modifications to core files is minimal. Changes to the Core Moodle codebase require ongoing work to maintain in each new release

of Moodle and have the potential to require complete rewrites for full version upgrades. Core Moodle codebase changes also can have unconsidered consequences on other Moodle features.

A Core Moodle modification is any change to any code within the released code found within the zip at http://download.moodle.org/. An example of a Core Moodle change would be modifying the course/lib.php to add a new function that your course format needs, upgrading the version of Yahoo User interface within the the lib directory, or modifying a line in user/tabs.php to add a new user tab when the user views their profile.

When reviewing Core Moodle changes, Blackboard answers the following questions:

1. Does the code require changes to Core Moodle to work?I f yes, how extensive/invasive are those changes?
2. How would the change affect other clients not using the code?
3. Can the changes be moved into a plugin (e.g. a block)?

Blackboard considers minimal changes to Core Moodle to be 2 or less patched files with a minimal sized diff (less than 15 lines). Extensive changes to Core Moodle would be 1 or more patched files to Core with a large diff (15 or more lines). In general, Blackboard does not review any code requiring a change to Core code.

VII. Continual Integration

Moodle plugins should contain PHP Unit tests and Behat tests. PHP Unit tests are located in the plugin's /tests folder and behat tests in the plugin's /tests/behat folder.

For every class / library in a plugin, there should be an accompanying php unit test file which tests each of the functions available in the class or library.

https://docs.moodle.org/dev/PHPUnit

Behat tests should be used to describe and test the operation of the plugin.

https://docs.moodle.org/dev/Acceptance_testing

https://docs.moodle.org/dev/Behat_integration

If your plugin is hosted on github, you can use the following tool to check the coding standards of your plugin and run any unit tests / behat tests via Travis. The README.md file in the following repository explains how to set up and use the tool.

https://github.com/Blackboard Open LMS/moodle-plugin-ci

There is also a code checker plugin for Moodle that can be run from the command line or integrated into your IDE:

https://moodle.org/plugins/local_codechecker

https://docs.moodle.org/dev/CodeSniffer

VIII. Installation and run time

Blackboard always installs code on an alpha instance during the code review process. Blackboard evaluates the code to determine if it:

- Installs correctly.
- Produces errors, notices or warnings while Moodle debugging is set to maximum.
- Should not insert user visible content into the database. This is usually done in the plugin's db/install.php. Examples would be a module adding an instance of itself to the site course or adding new profile fields to user profile page. Basically if the plugin has installed and it is CP disabled, there shouldn't be any significant change to the end users.

If the code fails to install or produces errors, then the code review is an automatic failure. Blackboard Open LMS will report all errors found to the client. If the code is being updated, Blackboard Open LMS also tests if the upgrade executes properly.

Upgrade and Install Plugin Settings must have Defaults
The plugin must install without any human interaction. (Ex. Admin would be able to just click "Save Changes" without having to modify any required fields.) This ensures that the plugin can be updated on many sites without intervention.

Notices/Warnings are not show stoppers, but Blackboard prefers they are fixed to reduce the information displayed when our developers are debugging.

Use of ini_set is restricted within plugins. Code reviews without sufficient documentation justifying the usage of ini_set will be an automatic violation.

During this process, code is reviewed for any licensing issues.

Moodle 3.x Code Note
The proper way to call rebuild_course_cache in the db/install.php for 3.x is to pass the true flag in the second param, for example:

rebuild_course_cache(SITEID, true);

DOES PROPER UPGRADE CODE EXIST?

For example modules in 3.x with db/upgrade.php must use the upgrade_mod_savepoint functions to properly upgrade.

```
function xmldb_modname_upgrade($oldversion=0) {
    // oldest version to upgrade from....
    if ($oldversion < YYYYMMDDXX) {
   // XMLDB upgrade path code...
            upgrade_mod_savepoint(true, YYYYMMDDXX, 'modname');
     }

    // newer version to upgrade from
    if ($oldversion < YYYYMMDDXX) {
   // XMLDB upgrade path code...
        upgrade_mod_savepoint(true, YYYYMMDDXX, 'modname');
    }
     ....etc...

     ....
     return true;
}
```

Note that in 3.x the version.php file must exist and be set to a version higher than the last save point so as not to trap the upgrade service in an infinite loop.

It is not acceptable to expect to delete all previous instances of a plugin in order to upgrade the plugin. (If there is a previous version code may not rely on db/index.xml to perform it's task alone) Upgrades **must** preserve/convert or otherwise gracefully handle previous instances on a site with the older version.

Consider looking into using the XMLDB Editor to auto generated some xmldb update code. (http://docs.moodle.org/dev/XMLDB_editor)

PLUGIN VERSION STRING

The plugin version string must follow the Moodle version string guidelines. (Many functions in Moodle rely on comparisons of this integer stamp.) Use the $plugin->release for a human readable release number. Read more here http://docs.moodle.org/dev/version.php

IX. Language files

It's very important for language files to be encoded correctly. To check if a file is encoded properly, run this inside of the plugin:

```
> find . -path './lang/*php' -exec file -I {} \;
```

Example output:

- `./lang/ca/turnitintool.php: text/x-php; charset=utf-8`
  - This file is correct, it has utf-8 charset.
- `./lang/sv/block_attendance.php: text/x-php; charset=iso-8859-1`
  - This file is **not** correct, it has iso-8859-1 (Latin 1) charset.
- `./lang/de/block_attendance.php: text/x-php; charset=us-ascii`
  - This is common, even in the Moodle code base.

Any charset of `iso-8859-1` must be fixed. The file must be opened and re-saved with the proper encoding and line endings. If these files are not fixed, then any UTF-8 characters inside of them will not display properly in browsers and tools in Moodle like the LANGUAGE CUSTOMIZATION will not work for that language.

If you must fix this yourself, one way of doing this is with `iconv`:

```
> pwd
/path/to/mod/modname/lang/fr
> iconv -f ISO-8859-1 -t UTF-8 modname.php > modname.php
```

Once you do that, do your best to verify the contents of the language file. An IDE might be helpful in comparing changes. Also, if it is a very foreign language, you can download language packs and compare to see if it looks even remotely similar. Note that this process doesn't always work so it might require re-translation or removal of the lang file.

X. Styles

Blackboard reviews all CSS styles added in `styles.css or in sass/less folders`. These CSS styles are included in the theme's primary `styles.css` and therefore have a wide ranging effect. Blackboard Open LMS specifically focuses on unique naming conventions for the CSS styles. Style's class names should be specific to the plugin or written to reduce its scope - both of the following examples achieve this:

Specific class:

```
.mod-widget-foo-bar
{
    color: #FFFFFF;
}
```

Limiting scope:

```
.mod-widget .foo-bar
{
    color: #FFFFFF;
}
```

XI. Markup

Wherever possible pre-existing markup should be utilised. It is available via the Moodle core renderers located in lib/outputrenderers.php or mustache templates located in lib/templates. This level of consistency makes it much easier for theme designers to achieve a consistent look and feel across Moodle. If bespoke markup is necessary, it should ideally be implemented through a renderable, renderer and mustache template.

RENDERABLE

A renderable is a class which describes data, in a similar way to a model in an MVC framework. It is good practice to have a renderable class corresponding to each component of markup you wish to output, in this way the data associated and markup are separated.

RENDERER

A renderer is a class containing functions for outputting markup located in \[plugindir\]/renderer.php. Renderer functions should not contain $DB calls and complicated logic. For each renderable class there will typically be a corresponding renderer function -

e.g. A renderable of 'person' should require a corresponding 'render_person' function in the renderer.

MUSTACHE TEMPLATE

A mustache template provides an easy way to mix variables and limited logic with html. As it doesn't contain PHP, a front end designer with no knowledge of PHP should be able to design markup around the template variables.

https://docs.moodle.org/dev/Output_API

https://docs.moodle.org/dev/Templates

BOOTSTRAP

In Moodle 3.2, the main Moodle theme is based on Bootstrap 4.

https://docs.moodle.org/dev/Bootstrap

XII. Accessibility

- All markup should be created with accessibility in mind. Accessibility must exceed the WCAG 2.0 Level AA standards as well as the Section 508 standard.

- Markup should be semantic and not rely on purely visual aspects (e.g. colour, boldness) to convey meaning.
- It should be possible to navigate the user interface using the keyboard and successfully complete all tasks necessary to fully utilize each feature.
- It should also be possible to use a screen reader to complete all tasks and fully utilize each feature.
- All images should have alt text unless they are purely decorative.
- All buttons should be defined as such, either by using the <button> html element or the aria-role="button" attribute.
- All headings should be headings (*not* bold text / styled via CSS) and should be semantically laid out - i.e main heading 'h1', sub heading 'h2', etc...
- All form fields should be logically labelled
- All links should appear as such - i.e. have an underline or be styled consistently in a fashion which differentiates them from the main text.
- Links should be self describing and should only occur once on the page.

XIII. File system interactions

Blackboard reviews all code for file system interactions - reading, writing, moving, copying and deleting of files in both the Moodle data directory and the general file system. Blackboard determines if the code excessively accesses the file system. Excessive reading of and writing to the file system can cause server performance and storage issues. A common example of excessive access to file systems is when a plugin writes to a log file often, instead of saving all log reports and writing once at the end of the process.

Because Blackboard Open LMS runs all code in a clustered environment, all temporary data must be written to a location in the Moodle data directory instead of the server's temporary directories, such as `/tmp`.

Because Blackboard Open LMS runs all code in a clustered environment, all temporary data must be written to a location in the Moodle data directory instead of the server's temporary directories, such as `/tmp`.

Blackboard Open LMS uses the regular expression below to determine what file system interactions 3rd party code is using:

```
(basename\s*\()|(chdir\s*\()|(check_dir_exists\s*\()|
(check_potential_filename\s*\()|(chgrp\s*\()|
```

```
(chmod\s*\()|(chown\s*\()|(chroot\s*\()|
(clearstatcache\s*\()|(closedir\s*\()|(copy\s*\()|
(delete\s*\()|(dir\s*\()|(dirname\s*\()|
(diskfreespace\s*\()|(disk_free_space\s*\()|
(disk_total_space\s*\()|(download_file_content\s*\()|
(fclose\s*\()|(feof\s*\()|(fflush\s*\()|
(fgetc\s*\()|(fgetcsv\s*\()|(fgets\s*\()|(fgetss\s*\()|
(file\s*\()|(fileatime\s*\()|(filectime\s*\()|
(filegroup\s*\()|(fileinode\s*\()|(filemtime\s*\()|
(fileowner\s*\()|(fileperms\s*\()|(filesize\s*\()|
(filetype\s*\()|(file_exists\s*\()|(file_get_contents\s*\()|
(file_put_contents\s*\()|(finfo_buffer\s*\()|
(finfo_close\s*\()|(finfo_file\s*\()|
(finfo_open\s*\()|(finfo_set_flags\s*\()|
(flock\s*\()|(fnmatch\s*\()|(fopen\s*\()|(fpassthru\s*\()|
(fputcsv\s*\()|(fputs\s*\()|(fread\s*\()|
(fscanf\s*\()|(fseek\s*\()|(fstat\s*\()|(ftell\s*\()|
(ftruncate\s*\()|(fulldelete\s*\()|
(fwrite\s*\()|(getcwd\s*\()|(get_directory_list\s*\()|
(get_records_csv\s*\()|(get_user_directories\s*\()|
(glob\s*\()|(is_dir\s*\()|(is_executable\s*\()|(is_file\s*\()|
(is_link\s*\()|(is_readable\s*\()|(is_uploaded_file\s*\()|
(is_writable\s*\()|(is_writeable\s*\()|
(lchgrp\s*\()|(lchown\s*\()|(link\s*\()|(linkinfo\s*\()|
(lstat\s*\()|(make_mod_upload_directory\s*\()|
(make_upload_directory\s*\()|(make_user_directory\s*\()|
(mime_content_type\s*\()|(mkdir\s*\()|(move_uploaded_file\s*\()|
(opendir\s*\()|(parse_ini_file\s*\()|
(parse_ini_string\s*\()|(pathinfo\s*\()|(pclose\s*\()|
```

```
(popen\s*\()|(put_records_csv\s*\()|(readdir\s*\()|
(readfile\s*\()|(readfile_chunked\s*\()|(readlink\s*\()|
(realpath\s*\()|(remove_dir\s*\()|(rename\s*\()|(resolve_filename_collisions\s*\()
|
(rewind\s*\()|(rewinddir\s*\()|(rmdir\s*\()|(scandir\s*\()|
(send_file\s*\()|(send_temp_file\s*\()|(send_temp_file_finished\s*\()|
(set_file_buffer\s*\()|(stat\s*\()|(symlink\s*\()|(tempnam\s*\()|
(tmpfile\s*\()|(touch\s*\()|(umask\s*\()|(unlink\s*\()|
(unzip_cleanfilename\s*\()|(unzip_files\s*\()|(valid_uploaded_file\s*\()|
(zip_files\s*\()|(check_and_create_moddata_dir\s*\()|
(check_dir_exists\s*\()|(\$CFG->dataroot)
```

XIV. Moodle Cron

Blackboard reviews the code to determine how the code injects itself into the Moodle administrative cron process. All tasks required to run on schedule should use the task API. It is undesirable for tasks to block other tasks or make excessive usage of system resources and this will be considered during the review process.

https://docs.moodle.org/dev/Task_API

XV. Key customization files

Certain files are more important than others in a Moodle customization because they are included globally even if the plugin is hidden in Moodle's administrative control panel. This means that these files will potentially affect all Blackboard Open LMS clients even if they aren't using the customization. These files are listed below organized by customization type:

- Blocks
  - block_blockname.php
- Any plugin with a lib.php file
- Any plugin with a settings.php file
- Any files within a plugins db directory
- Any plugin with a version.php file (All should have one now)
- Any files within a plugins lang directory

XVI. Recommended sample size

- **Courses:** 500
- **Users:** 1,000
- **Enrollments:** 3,000
- **Users per course:** 30-50
- **Activity Instances:** 5-10
- **Block Instances:** 5-10

XVII. Deprecated Function Removal

Code should not be using any deprecated functions. Some of these functions are listed in lib/deprecatedlib.php, some are class methods in various classes throughout Moodle. To find the deprecated class methods in Moodle you can scan for the @deprecated PHP doc - for example:

CLI command (tested on OSX and Ubuntu)

```
find . -type f -name \*.php -exec grep -E -a4 '(\* @deprecated)(.*)(function)' {} \+
```

A plugin with PHP Unit tests covering the majority of code will fail on deprecated functions that call debugging functions.

INVALID FUNCTION USAGES

- **get_list_of_plugins**: After Moodle 2.6, this function can only be used on non-standard plugins. Example violations are `get_list_of_plugins ('mod')` or `get_list_of _plugins ('course/format')`. Correct usage of this function would be some path to any non-standard plugin directory, like `get_list_of_plugins('backup/converter')`. These invalid usages **must be fixed**.

XVIII. Glossary of terms

- **Core Moodle:** This refers to any code that is released as part of the open source Moodle project found at http://download.moodle.org/. At the time of this writing, the current version of Moodle is 3.2.1.