

# Development

- [Moodle Versions in Git](#)
- [Set the Default Date on the Moodle Date Picker](#)
- [XMLDB constants](#)
- [Convert URL object to string](#)
- [Turn off Javascript Caching](#)
- [Adding Navigation to a Local Plugin](#)
- [Custom Email Signup Registration Form](#)
- [Plugin Dependencies](#)
- [Move the Download Options Below a Table](#)
- [Debug SQL generated by \\$DB methods](#)
- [Getting the Moodle Base/Root Directory](#)
- [Make a Select Form Field Required](#)
- [Unknown Error Upgrading Plugin to Version](#)
- [XMLDB Editor and Reserved Names](#)
- [Set User IDs that will always see debug messages](#)
- [Debug Log Level Values](#)
- [Upgrading Database Schema for a Plugin](#)
- [Moodle Git Branches and Tags](#)
- [Get Records SQL giving incorrect result](#)
- [Setting up a Scheduled Task for your Plugin](#)
- [User Profile Field Select Options](#)
- [Upgrading between Moodle Stable Versions with Git](#)

# Moodle Versions in Git

All versions of Moodle are appropriately tagged in the Moodle git repository (

`git://git.moodle.org/moodle.git`) so you can use the following to show the commits that relate to each version. Major versions are branched using the convention `MOODLE_XX_STABLE` where `XX` is the version e.g. `MOODLE_36_STABLE` for Moodle 3.6.

```
git show-ref --tags
```

This gives you something like this:

```
...30e069a061296bef321b31b791a4366a953cae23 refs/tags/v3.6.1
008baea92aa20ee7ff619d37cc6568b076cda00c refs/tags/v3.6.2
ec902921430745139548095e45380ed017acd770 refs/tags/v3.6.3
```

Here you can see that the `v3.6.1` tag has the commit hash `30e069a061296bef321b31b791a4366a953cae23` (the tag is just an alias for that hash).

The key to working with the Moodle git repository is the `version.php` file. This file is changed on every major commit, so you can use it to track down a specific moodle version in the repository (use this in conjunction with branches and version tags).

```
git show v3.6.1:version.php
```

To see what's changed on this file including the file changes themselves use:

```
git whatchanged -p version.php
```

This is handy, but chances are you are searching for a specific version. Eg. say I want version `2012062503.02`. How do I track down the commit(s) that relate to that version? Use the command above but add the `-S` search parameter:

```
git whatchanged -p -S2012062503.02 version.php
```

This gives you the commit details (including the hash) you need. If you get multiple results, you probably want the latest commit.

Once you have the correct commit, use `git checkout` with the `branch` option to make/set your branch. E.g.

```
git checkout 5f1d8f2 -B moodle
```

Now my moodle git repository is on the exact commit I need. Very useful for comparing a code base to a vanilla Moodle to find any customisations.

Sometimes you want to know what files changed between versions, you can use `git diff` with the `--stat` option for that:

```
git diff v3.6.2 v3.6.3 --stat
```

If you use a diff/merge tool (e.g. Araxis Merge) you can use the `--dir-diff` to get a full directory comparison of changes between tags as well. Very handy.

```
git difftool --dir-diff v3.6.1 v3.6.2
```

One other useful thing is to look at the log between two tags like so:

```
git log --oneline v3.6.1..v3.6.2
```

Handy to see what Moodle fixes were added between tags (`MDL-`) which you can check against the [Moodle Tracker](#)



# Set the Default Date on the Moodle Date Picker

You can set the default date on the Moodle Date Picker using the `setDefault` method for example:

```
$mform->setDefault('date_field', date('U', strtotime("+14 days")));
```

The following code changes the default value of the date picker from today's date to 14 days in the future. The second parameter returns the date in Unix format 14 days from now.

# XMLDB constants

Moodle uses [XMLDB](#) as an abstraction layer to support multiple database platforms and schema changes. You'll need to know how this works to work with the database. In particular you'll need to know the relevant constants.

They are all defined under and prefixed with XMLDB

```
lib//xmlldb/xmlldb_constants.php
```

For example, you can get the database field types like so:

```
cat lib/xmlldb/xmlldb_constants.php | grep XMLDB_TYPE
```

If you need some examples of the syntax, check out the core moodle install and upgrade files under

```
lib/db.
```

# Convert URL object to string

There's a method in the moodle\_url class in `lib/weblib.php` called `__toString()` which will convert a Moodle URL object back to a string if you just need the URL string itself.

```
$PAGE->url->__toString()
```

# Turn off Javascript Caching

When developing in Moodle, Javascript caching can be a real pain as you generally need to purge caches & reload the page to get the javascript to refresh.

There's an easy way to turn it off though:

Site Administration > Appearance > AJAX and Javascript

Cache Javascript ( `cachejs` ) which is set to YES by default. Turn this off.

You can also find and update the `cachejs` entry in the `mdl_config` table or set it in the `config.php` file. Just remember to purge your cache to ensure it takes effect.



# Adding Navigation to a Local Plugin

Here's how to add your local plugin to the following navigation through the `settings.php` file:

Site Administration > Plugins > Local Plugins

```
// Add the settings page to the navigation block
$settings = new admin_settingpage(
    'local_yourplugin',
    get_string('pluginname', 'local_yourplugin')
);
$ADMIN->add('localplugins', $settings);
```

The final line `$ADMIN->add('localplugins', $settings);` determines where the plugin will be placed so you can change `localplugins` to other common locations have a look at `/admin/settings/plugins.php` for some other examples.

# Custom Email Signup Registration Form

You can modify the custom `auth/email` signup registration page with the following steps (note this is a minor customisation to core code):

In `auth/email/auth.php` add the following code to call a custom signup form file:

```
function signup_form() {  
    global $CFG;  
    require_once($CFG->dirroot.'/auth/email/signup_form.php') return new login_signup_form(null,  
    null, 'post', '', array('autocomplete'=>'on'));  
}
```

This creates a hook to open the new file you will be creating `auth/email/signup_form.php` thereby minimising customisations to the existing `auth/email` plugin code.

Copy `signup_form.php` from `login/` into `auth/email`.

**NOTE:** you can put this file anywhere, but it seems logical to keep it with the auth/email plugin that will be using it right?

Modify `auth/email/signup_form.php` accordingly.

# Plugin Dependencies

Plugin dependencies can be added using the `$plugin->dependencies` attribute in `version.php`.

There are quite a few examples in the code base, for example:

`filter_data` depends on `mod_data` so the following appears in the `filter/data/version.php`:

```
$plugin->dependencies = array('mod_data' => 2014050800);
```

# Move the Download Options Below a Table

If you are working with flexible tables ( `lib/tablelib.php` ) in Moodle and have table download enabled, you can move the “Download table data as” select and button below the table by adding the following line before finishing your table setup:

```
$table->show_download_buttons_at(array(TABLE_P_BOTTOM));
```

By default they appear above the table but this doesn't always look so good and takes up screen space.

# Debug SQL generated by \$DB methods

A handy debugging tip, if you need to debug the SQL generated by any of the `$DB` methods (see [DML API](#)), you can use the following around the statement(s) to display the SQL:

```
$DB->set_debug(true);  
//...  
$DB->set_debug(false);
```

# Getting the Moodle

## Base/Root Directory

The moodle root directory `$CFG->dirroot` is established in the file `lib/setup.php` along with other values in the global `$CFG` object which combines configuration from `config.php` as well as the entries in the database table `mdl_config`.

The value is set as follows:

```
$CFG->dirroot = dirname(dirname(__FILE__));
```

And you can use it reliably establish the base/root moodle directory absolute path and use it for things like file requires in your plugin code.

Some other useful entries in `lib/setup.php` from the code:

- `$CFG->wwwroot` = Path to moodle index directory in url format.
- `$CFG->dataroot` = Path to moodle data files directory on server's filesystem.
- `$CFG->dirroot` = Path to moodle's library folder on server's filesystem.
- `$CFG->libdir` = Path to moodle's library folder on server's filesystem.
- `$CFG->tempdir` = Path to moodle's temp file directory on server's filesystem.
- `$CFG->cachedir` = Path to moodle's cache directory on server's filesystem (shared by cluster nodes).
- `$CFG->localcachedir` = Path to moodle's local cache directory (not shared by cluster nodes).

There's lots more in there have a look around in `lib/setup.php`.

# Make a Select Form Field Required

When adding a form rule for select (or multiple select) fields, these need to be applied on the client side (not server side).

Here's an example of how to add a rule for a select field:

```
$mform->addRule(  
    'selectfield',  
    get_string("selectfieldrequiredmessage", "local_yourplugin"),  
    'required',  
    '',  
    'client'  
);
```

The key is the 5th parameter above which is set to `client` instead of the default of server.

For more help, see the [Moodle Forms Library Documentation](#)



# Unknown Error Upgrading Plugin to Version

If you are getting an error like this when attempting to upgrade a plugin:

```
!!! Unknown error upgrading <pluginname> to version 2017042400, can not continue. !!!  
!!  
Error code: upgradeerror !!!! Stack trace: * line 340 of /lib/upgradelib.php: upgrade_exception  
thrown  
* line 549 of /lib/upgradelib.php: call to upgrade_plugin_savepoint()* line 1630 of  
/lib/upgradelib.php: call to upgrade_plugins()* line 171 of /admin/cli/upgrade.php: call to  
upgrade_noncore()  
!!
```

Then it might be simply due to a missing line in your `db/upgrade.php` file.

Check that in your function `xmlldb_<type>_<pluginname>_upgrade($oldversion)` you have a final line that returns a true value.

That is: `return true` at the end of the function.

# XMLDB Editor and Reserved Names

One (of the many) good reasons to use the XMLDB editor in Moodle rather than hacking the `db/install.xml` file directly is that it warns you if you use a reserved name.

For example:

|        |        |      |        |          |                |                    |
|--------|--------|------|--------|----------|----------------|--------------------|
| column | [Edit] | [Up] | [Down] | [Delete] | [XML] Reserved | char (64) not null |
|--------|--------|------|--------|----------|----------------|--------------------|

The red Reserved text after `[XML]` in the screenshot is indicating that the word `column` is reserved and shouldn't be used. It would be nice if it stopped you from saving such a change (it doesn't) but at least it indicates a problem.

The Reserved link points to the following location if you ever want to look up the currently used reserved words:

```
<http://www.yourmoodle.com>/admin/tool/xmldb/index.php?action=view_reserved_words
```

This link also shows you potential existing problems in your XMLDB that may need to be fixed.

# Set User IDs that will always see debug messages

There's a handy feature in Moodle configuration called `$CFG->debugusers = '<moodle_user_ids>'` that allows you to set certain user IDs to always see debug messages regardless of the setting for site under:

Site administration > Development > Debugging

This can be a comma separated list of Moodle User Ids (from the user table). Set this in the `config.php` or see `config-dist.php` for more details.

Useful if you need to troubleshoot an error but don't want to increase debug level for the entire system and catch messages for the other users in the system too.

# Debug Log Level Values

These are the values you can set for the debug level under:

Site administration > Development > Debugging

For Debug message ( `debug` ). You can also query this from the `mdl_config` table looking at the key `debug`.

- `0 = NONE` : Do not show any errors or warnings
- `5 = MINIMAL` : Show only fatal errors
- `15 = NORMAL` : Show errors, warnings and notices
- `30719 = ALL` : Show all reasonable PHP debug messages
- `32767 = DEVELOPER` : extra Moodle debug messages for developers

Those are values you can set in the database for the debug entry in `mdl_config` if you need to check or change this on the database. You need to purge your cache as well.

**NOTE:** even on production sites, leave the setting at `MINIMAL` instead of `NONE` to help with troubleshooting as some errors are very time/user specific and hard to replicate. If your PHP error is filling up on this setting, it indicates an underlying issue.

These are defined with PHP error constants in `lib/setuplib.php` as follows:

- `DEBUG_NONE = 0`
- `DEBUG_MINIMAL = E_ERROR | E_PARSE`
- `DEBUG_NORMAL = E_ERROR | E_PARSE | E_WARNING | E_NOTICE`
- `DEBUG_ALL = E_ALL & ~E_STRICT`
- `DEBUG_DEVELOPER = E_ALL | E_STRICT`

These constants may vary depending on Moodle versions.

# Upgrading Database Schema for a Plugin

If you need to upgrade your database schema for a custom plugin, e.g. adding a new table or changing the properties of an existing table, you should do it through `db/upgrade.php` per the suggestions in the [Upgrade API](#) in Moodle Docs.

This isn't a bad idea to do even when iterating through development as it isn't a lot of work to keep things synchronised through the XMLDB editor and you won't miss something later like you can if you are manually altering database schema.

Use the Moodle XMLDB editor to adjust your schema:

Site administration > Development > XMLDB editor

Copy the relevant changes into your `install.xml` and `upgrade.php` straight from the editor then update your plugin version and test.

The Moodle XMLDB editor actually makes this very easy.

- Open the XMLDB editor
- Find your plugin (assuming it is already installed etc). If you haven't managed to get this far, at least install what you have so it will show up in XMLDB editor. Note you do not need to have anything DB related defined for this to work.
- Load the DB for your plugin or if nothing is defined using Create.
- Follow the UI to create your table, fields, keys and indexes
- When you are ready use View PHP code to get the code to go into `db/upgrade.php`

Go all the way back to the main XMLDB editor screen (where you can see all plugin databases) and use save to save the XML to `db/install.xml` as well.

See this [article](#) if you are getting a save error.

# Moodle Git Branches and Tags

Moodle uses the convention of the branch name `MOODLE_XX_STABLE` for each stable release e.g. Moodle 3.7 is in the `MOODLE_37_STABLE` branch.

All minor version upgrades then go into this branch until the next stable release and are tagged accordingly.

There's heaps more information about this over at Moodle Docs in [Git for Developers](#).

As a shortcut you can use the following to see all the stable branches:

```
git ls-remote git://git.moodle.org/moodle.git | grep head
```

This returns a bunch of hashes and the `refs/heads/{MOODLE_XX_STABLE}` branches. You can guess these, so this is just to check what they are without having to clone the entire repository. Plus you'll know when a new stable branch is available in the repository.

To clone a specific branch (e.g. say I only want Moodle 3.2), use this command in the relevant target directory (note the `-t` to include tags so you can go to specific tags like `v3.2.2`):

```
git clone -b MOODLE_32_STABLE git://git.moodle.org/moodle.git
```

To see all the tags:

```
git ls-remote -t git://git.moodle.org/moodle.git | grep -v \\^\\{\\}
```

The grep at the end `grep -v \\^\\{\\}` removes the tag with the `^{} at the end, which seems to come through otherwise e.g. you get both refs/tags/v3.7.1 and refs/tags/v3.7.1^{}.`

You can then just filter down to your required Moodle stable version with another grep e.g. for just v3.6.x tags:

```
git ls-remote -t git://git.moodle.org/moodle.git | grep -v \\^\\{\\} | grep v3\\.6
```

One other handy thing is to use show origin to see what you are currently tracking to make sure you have all the latest branches from the Moodle Git (remote) repository and if any of your local branches may be out of date with remote.

```
git remote show origin
* remote origin
Fetch URL: git://git.moodle.org/moodle.git
Push URL: git://git.moodle.org/moodle.git
HEAD branch: master
Remote branches:
MOODLE_13_STABLE tracked
MOODLE_14_STABLE tracked
MOODLE_15_STABLE tracked
MOODLE_16_STABLE tracked
MOODLE_17_STABLE tracked
MOODLE_18_STABLE tracked
MOODLE_19_STABLE tracked
MOODLE_20_STABLE tracked
MOODLE_21_STABLE tracked
MOODLE_22_STABLE tracked
MOODLE_23_STABLE tracked
MOODLE_24_STABLE tracked
MOODLE_25_STABLE tracked
MOODLE_26_STABLE tracked
MOODLE_27_STABLE tracked
```



MOODLE\_28\_STABLE tracked

MOODLE\_29\_STABLE tracked

MOODLE\_30\_STABLE tracked

MOODLE\_31\_STABLE tracked

MOODLE\_32\_STABLE tracked

MOODLE\_33\_STABLE tracked

MOODLE\_34\_STABLE tracked

MOODLE\_35\_STABLE tracked

MOODLE\_36\_STABLE tracked

MOODLE\_37\_STABLE tracked

master tracked

Local branches configured for 'git pull': MOODLE\_36\_STABLE merges with remote

MOODLE\_36\_STABLE

master merges with remote master

Local refs configured for 'git push': MOODLE\_36\_STABLE pushes to MOODLE\_36\_STABLE (local out of date)

master pushes to master (local out of date)

# Get Records SQL giving incorrect result

If you have code that uses `$DB->get_records_sql()` and it is only giving you one row, or an incorrect number of rows, then first just make sure you are using the plural form `get_records_sql()` and not `get_record_sql()`.

If that's not the issue, the other reason this can happen is that Moodle uses the first column in your query as a unique `id` and then filters results down to that `id` field. However, if your first column isn't unique across your data set, you won't get every row.

For example, your query may select user enrolments for a course. But if you put your course id as the first column, you'll only get one row, even if there are 30 enrolments in that course, because all of those enrolments belong to that one course id and that is the unique identifier used.

The fix is simple, find (or if you need to, create) a unique id so that the first column is unique for every row in your query. In the user enrolments example you probably want to use the unique id from `mdl_user_enrolments`.

**NOTE:** in this example, using `user id` isn't a great choice either because a user can have more than one enrolment.

# Setting up a Scheduled Task for your Plugin

There are two parts to setting up a scheduled task in your plugin.

- Create the task class under `classes\task` with an appropriate name using the template shown.
- Define the task and default schedule in `db\tasks.php`

## Creating the task class

First, create the directory structure `classes\task` in your plugin folder. You will need to create at least the `task` directory and perhaps the `classes` one as well.

Here's the basic template for a scheduled task using the example of a local plugin called `yourplugin` as a placeholder. This file would be saved as `example_task.php` to match the name of the class.

```
/**
 * {Example_task} class definition
 *
 * @package    local/yourplugin
 * @author     Your Name * @license    http://www.gnu.org/copyleft/gpl.html GNU GPL v3 or
 * later
 */
namespace local_yourplugin\task;
use core\task\scheduled_task;
class example_task extends scheduled_task {
    /**
     * Get scheduled task name.
    */
}
```

```

*
* @return string
* @throws \coding_exception
*/
public function get_name() {          return get_string("exampletaskname",
"local_yourplugin");
}
/**
* Execute the scheduled task.
*/
public function execute() {
    global $CFG;          require_once($CFG->dirroot .
'/local/yourplugin/locallib.php');
    local_yourplugin_execute_task();
}
}

```

A few things to note about this code:

- You need an entry for `exampletaskname` in your plugin's language pack that is the readable name of the task (used by `get_name()` method) and shown on the scheduled tasks page.
- The `execute()` method includes `$CFG` and includes the `locallib.php` for your plugin where you define the details of the `local_yourplugin_execute_task()` function.

You can change the `local_yourplugin_execute_task()` to anything you like. The goal is to define the logic for this function in the local library (`locallib.php`) and **not** in the task class.

## Define the task schedule

The file `tasks.php` can then be created in the `db\` folder. Here's an example of how this is defined using the `$tasks` array. If your plugin has more than one scheduled task, simply add another child array.

```

/**
* Schedule tasks

```

```

*
* @package      local/yourplugin
* @author       Your Name * @license      http://www.gnu.org/copyleft/gpl.html GNU GPL v3 or
later
*/

$tasks = [
    [
        'classname' => 'local_yourplugin\task\example_task',
        'blocking' => 0,
        'minute' => 0,
        'hour' => 6,
        'day' => '*',
        'month' => '*',
        'dayofweek' => '*',
        'disabled' => 1
    ]
]

```

In this example:

- The namespace to the class definition is used for the `classname` so make sure you have namespaced your task class accordingly.
- The default scheduled is to run every day at 6am
- The task is disabled by default

# User Profile Field Select Options

If you need to get the list of user profile field select options for your plugin, you can query the data from the `mdl_user_info_field` table, looking for the matching `shortname` of the field. The select options are stored in the column `param1`.

Once you have queried these out e.g.

```
global $DB;
$query = '
    select param1
    from {user_info_field}
    where shortname = :shortname
';
$parameters = ['shortname' => '<profile_field_shortname>'];
$result = $DB->get_record_sql($query, $parameters);
```

You can then use the PHP `explode` command to explode the values in `param1` into an array as they are stored in the database as separate lines with a line feed (`\n`) character.

```
$options = explode("\n", $result->param1);
```

# Upgrading between Moodle Stable Versions with Git

Due to the large divergences between moodle stable versions in the Moodle git repository (e.g. `MOODLE_37_STABLE`, `MOODLE_38_STABLE`, `MOODLE_39_STABLE`) it can be a challenge to upgrade between major Moodle versions. Here's one approach you can use if you have minimal customisations (e.g. just additional plugins):

First, add the moodle code tree as an upstream e.g. called Moodle

```
git remote add moodle git://git.moodle.org/moodle.git
```

Checkout a new branch e.g. lets say we are going from Moodle 3.8 to Moodle 3.9, we'll call this branch `upgrade39` and base it from say `master` where we have the current Moodle 3.8 code base.

```
git checkout -b upgrade39
```

Fetch the latest Moodle 3.8 and perform a minor version upgrade to the very latest Moodle 3.8 code. If you hit any conflicts here, they need to be resolved before moving on (e.g. core code customisations).

```
git fetch moodle MOODLE_38_STABLE
git pull moodle MOODLE_38_STABLE
```

Next, fetch the last Moodle 3.9 code from the upstream Moodle code repository:

```
git fetch moodle MOODLE_39_STABLE
```

Now we can attempt the rebase, rebasing our Moodle 3.8 code with the latest Moodle 3.9 stable code on the upgrade 3.9 branch:

```
git rebase --onto moodle/MOODLE_39_STABLE moodle/MOODLE_38_STABLE upgrade39
```

Hopefully the rebase works with any issues. Of course if there are problems you'll need to work through the rebase conflicts accordingly.