Developing A Moodle Block Plugin

Contents:

Setting up the development environment

In order to develop a Moodle plugin, Moodle must first be installed. Moodle has many different versions available, with 3.8 being the latest at the time of this writing. Installation instructions can be found here for the latest version. Legacy versions are also available and can be downloaded here. Both these Web pages contain links to other versions and version documentation. The Moodle main development documentation page contains links to various API pages and other developer resources.

Moodle has a number of minimum requirements and dependencies that must be met, all of which is detailed in the installation documentation and the release notes. Moodle is a Web-based application, and as such, requires a Web server (Apache), database (MySQL, MariaDB, PostgreSQL), and PHP. It is recommended to deploy to a GNU/Linux server, but it is possible to use Windows or Mac OSX. Moodle itself can be downloaded from here, then copied into location. The database must also be set up and Moodle requires a separate data directory. With these prerequisites met, the Moodle installation script can be run, which will continue the process. It is also important to set up cron, which will run scheduled tasks automatically in the background. More information about cron can be found here.

With Moodle installed, configured, and running, there are few settings that can be changed to aid in plugin development. Moodle caches strings and JavaScript by default, but this can be turned off so that changes made to the language file or any JavaScript will take effect without having to manually clear the caches. To change the caching of language strings, log in as administrator and navigate to Site administration -> Language -> Language settings, then look for the "Cache all language strings" checkbox. Similarly, from Site administration -> Appearance -> AJAX and JavaScript, there is a checkbox labeled "Cache JavaScript." It is possible to clear the caches manually from Site administration -> Development -> Purge all caches. There is more information about in the developer mode documentation.

Moodle code is written in English and that is the language plugins get submitted in, but Moodle supports numerous other languages. To enable multiple languages in Moodle, the required language packs must be installed. From Site administration -> Language packs, scroll through the list of available languages on the left and select one. There is a button at the bottom of the list to install the desired language pack. Changing languages in Moodle is done from the language drop-down list located in the top menu bar.

Development of a block plugin

Installing a plugin

To get a plugin started, the skeleton generator plugin can be used. This plugin is designed for developers to quickly get a simple plugin template made with the minimum plugin requirements. To use the skeleton generator plugin, it must first be installed. There are 3 ways to do this. The first is through the Moodle plugin directory. The second is using a downloaded archive of the plugin through the Moodle plugin installation interface. Alternatively, it is possible to copy the plugin source code directly into the Moodle code, then navigating to site administration.

For either of the first two plugin installation methods, log in as administrator and navigate to Site administration -> Plugins -> Install plugins. There is an "Install plugins from the Moodle plugins directory" button that will take you to the plugins directory. You will need to have an account created and log in, then you can browse the plugin directory. Click the plugin you want to install, then follow the "Download" and "Install" buttons and links to install the plugin.

Alternatively, you can browse the plugins directory directly, find a plugin and download it as a zip archive. From Site administration -> Plugins -> Install plugins, look for the option to install from ZIP file. Use the file browser or drag and drop your file to upload it and start the installation process. Provided the plugin passes Moodle's checks, the plugin will be allowed to be installed.

After installation, depending on the method, and to trigger an upgrade after changing the version, an administrator must navigate to their dashboard. This will trigger Moodle to look for discrepancies between installed versions of plugins (in the database) and the version in the version file. When the version has changed, Moodle will upgrade the database, which involves running the upgrade file for the plugin.

Using and configuring a block plugin

With a block plugin installed, it is then necessary to add it to a course. This can be done from within the course home page by clicking the settings icon (top right) and selecting to turn editing on. Then, click the "Add block" button (lower left), which will bring up a menu of installed plugins. Click the plugin you want to install and it will appear in the course page.

With editing on, the block will have its own settings icon, which can be used to configure the block. Each block has default options, but can have custom options as well. The Demo block has added the option to display a question and answer interface in the block, where the question and a response are customizable. These options affect only the block instance installed in that course. Blocks can also have global settings, which are available from Site administration -> Plugins by scrolling down to the "Blocks" section and clicking the name of the plugin.

Plugin file structure

A Moodle block plugin has certain files that are required and a number that are optional. More information on blocks can be found in the blocks documentation. All blocks are located in the blocks directory, where each plugin is contained in its own sub-directory, which is considered the root of the plugin directory structure. The root directory must contain the block definition file (block_name.php) and the version file. There must also be a directory called lang for language files, which must contain an en directory for the English language file (lang/en/block_name.php). A directory named db must contain an access file that indicates user role capabilities.

Among the optional files are a global settings file and an instance options file (edit_form.php). Within the db directory, there may be an install.xml file, which contains the database table definitions, as well as scripts that run during install, uninstall, or upgrade. There may also be a classes directory that contains various other classes the plugin uses, such as for event logging and background tasks. More information on running background tasks is available in the task API documentation. There is also an advanced block documentation page that contains more information of using forms, creating and using a database, and refining the user interface. For more information on various plugin files, see the plugin files documentation,

Frankenstyle naming conventions

The block definition and language files, mentioned earlier, both have a specific naming convention, called Frankenstyle, that must be followed. This involves prefixing certain file names with block, such as in block_name, where name is the name of the plugin. Function and class names are prefixed with block_name, such as block_name_class1, as are database table names, such as block_name_table1. Use of the Frankenstyle naming conventions is required for many files to get recognized and loaded properly, and as protection against name collisions.

Creating database tables

Each Moodle instance has a database that can be accessed. To create tables for your plugin, as administrator, navigate to Site administration -> Development -> XMLDB editor. This page contains a list of all the plugins and other modules that have database tables. Find the name of your plugin in the list, then load and edit its definition file. The interface for editing table definition files is fairly straight forward. It will list any tables already created with links to edit those tables. Alternatively, click the "New table" link to create a new table, then add fields. Remember to save after making changes or the changes will not propagate to the db/install.xml file. More information on creating and editing tables can be found in the XMLDB editor documentation and using XMLDB documentation. Note that the Frankenstyle naming conventions apply to database table names. All table names must start with block_name, where name is the name of the block.

Creating a table using the XMLDB editor only creates a definition for the table and does not actually create the table in the database. The tables will be created on installation, provided the tables are defined in db/install.xml at the time of installation. To create tables after the plugin has been installed, the table should still be created in the XMLDB editor. The editor has a link for viewing the PHP code, which can be used to determine the code for creating or altering a table during upgrade. The code will be placed in the db/upgrade.php file for this purpose. The upgrade script gets run whenever the version changes and an administrator views their dashboard page. The upgrade file provided with the Demo block provides an example of table creation using the upgrade feature.

Using database tables

To access the Moodle database, there is a data manipulation API that is used to interact with the tables. Note that the table's schema can not be changed using this API, as that must be done using the XMLDB editor and the upgrade file. The API has numerous functions that provide a consistent interface to the database without need for the programmer to worry which database the end user may have configured. This API provides functions for simple query tasks such as getting a set of records, deleting records, and altering records. There are also more complicated functions for running more complicated queries involving mixed and or clauses. There are also functions for executing raw SQL queries. However, it is recommended to avoid raw SQL queries as much as possible to avoid cross-database conflict, which can cause errors and break the plugin functionality. Using the data manipulation API offloads the query specific syntax issues to Moodle and ensures the programmer's code works on any Moodle installation.

Outputting content

A block type plugin occupies a space on the side or bottom of a page. Blocks are often found on course pages, but can occur elsewhere within Moodle as well. The block can contain arbitrary HTML, CSS, and JavaScript, which includes links, forms, and images. As there may be other blocks on the same page, as well as other code from the page and from Moodle, it is important encapsulate code as much as possible and practice good naming conventions. For output, the two most important APIs are the output API and page API. The output API controls a renderer and is available through a global variable. But it is also possible to create a custom renderer, which is what some plugins have done, such as the Moodle community block. The page API is used for setting up aspects of a page such as title and page URL, as well as including CSS and JavaScript in a page.

To output HTML, the HTML writer class is used. This class is for generating HTML content for output to the block or to a separate page. It has functions for outputting simple paragraphs and divs, as well as showing images and building tables. Unfortunately, the online documentation for the HTML writer class is limited and perusing the source code is the best way to learn the functions available and their parameters. The html_writer class is found in Moodle's

lib/outputcomponents.php file. There are also many examples of how the class is used in other plugins and Moodle pages.

Using JavaScript with Moodle

The page API, introduced earlier, is used for including JavaScript in the block or in a page. The JavaScript can be a regular script or it can be wrapped as an Asynchronous Module Definition (AMD) module and loaded through require.js. Using AMD modules is the preferred method of including JavaScript code because it encapsulates the code and decreases naming conflicts between different scripts that may be loaded by other plugins. Many third party libraries already come as an AMD module, so this is the best way to include such a library. More information can be found in the JavaScript modules documentation. The Demo block also provides working examples of both plain scripts and modules, as do other plugins.

The page API has a few important functions for including JavaScript that are worth mentioning. These functions use the global PAGE object, which has a chained function call. Loading plain JavaScript can be done with a command such as $PAGE->requires->js('location/of/script.js'), but this will only load the script and not call any functions within the script. Another function, js_init_call(), is required to start the script, where the function name to call is the first parameter of the function. The js_init_call() function is also used to pass data to the script, which is the second argument.

There is also a js_call_amd() function that is used for loading AMD modules. This function takes the name of module to load as the first argument and the function within the module to call as the second argument. If Moodle is configured to cache the JavaScript (see Setting up the development environment), then the JavaScript must be minified. Any modular code resides in the amd directory, which has 2 sub-directories, src and build. The src directory holds the human readable source code and the build directory holds the minified code. When Moodle is not caching JavaScript, then the code from the src folder is used. Moodle recommends using Grunt for linting, minification, and packaging.

There is also a function for passing strings to the JavaScript. The string_for_js() function takes the string identifier and the location of the string as arguments, which is the same as the get_string() function used for pulling strings from the language file. Any strings included using this function are globally available in the JavaScript and can be accessed with M.str.block_name.stringid.

For those who use the JQuery framework, there are also functions for including it and its various plugins. The function $PAGE->requires->jquery() will include the framework. Another function jquery_plugin() takes the name of the plugin as argument and will then include that plugin in the page as well. All these functions and more can be found in the Moodle lib/outputrequirementslib.php file, which also contains some deprecated functions that are shown in some Moodle documentation, but should not actually be used.

Coding standards

Moodle has a variety of coding standards that you are expected to adhere to. There is specific documentation for general coding style, JavaScript coding style, and SQL coding style. The coding standards page, linked to above, provides an overview of the various facets that Moodle has standards for, which include coding style, security measures, accessibility, usability, and performance, among others. Some of these facets will be briefly explored below.

It is worth noting that there are 2 plugins that are helpful in adhering to Moodle's standards. There is a plugin for checking coding style and another for checking documentation. The Moodle general coding style documentation specifically recommends using both these plugins. It is also recommended to use Travis Continuous Integration (Travis CI), which is available through GitHub, that also runs a number of tests against the code every time new code is pushed to the repository. With Travis CI it is possible to run tests against various versions of PHP and Moodle automatically, so is a very useful development tool.

Coding style

There are many different coding style standards that Moodle has. These include not using a closing PHP tag at the end of files, using spaces instead of tab characters for indenting, using 4 space indents, and having a maximum line length of 180 characters (132 recommended maximum). There are also standards for how to indent long lines that have been wrapped to the next line and which type of end of line (EOL) character to use (Unix type LF, not Mac CR or Windows CRLF). There are also file and function naming conventions, which include, but are not limited to the Frankenstyle naming mentioned earlier. It is a good idea to read through the coding style documentation and use the code checker plugin against your own code.

There are also a number of standards for documentation. These include the use of the GNU GPL at the top of each PHP file, followed by a comment block containing a description of the file contents, then package, copyright, and license tags. All classes and functions must be documented with param and return tags. Inline comments are to start with two slashes and a space, then a sentence consisting of a capitalized first word and ending in proper punctuation. It is a good idea to read through the general coding style documentation and to use the documentation checking plugin against your own code.

Security

Moodle has a number of security standards that are important to adhere to. Only a couple of major issues will be mentioned here. When making a page for users to view, there are certain security checks that must be done to ensure only authorized users are able to access the page.

Moodle has a number of user roles such as administrator, teacher, and student, as well as different contexts such as system, course, and module. Together these allow users to have various capabilities in various contexts. It is important to check to see if a user has the required

capability in the current context before allowing them to view a page or do something else. This is accomplished through the functions has_capability() and require_capability(), which take a capability and a context as parameters, then tests these against the capabilities defined in the db/access.php file to determine if the action is permitted. Note that a change in the access file requires a reinstall for the changes to take effect.

It is also important to authenticate the user. When a request is made to execute a PHP script, the user making the request must be a valid user, and since blocks are most often used in courses, the user must be enrolled in the course. For user authentication, the require_login() function is used, which takes a course object as parameter. This function ensures the user is logged in and has permissions to access the course. This function is part of the access API, which contains other similar functions.

Passing data to a PHP script in the URL is normal practice, but it opens up a security vulnerability. The person requesting the script may have manually altered the URL, which is why it is important to verify the received parameters. Moodle has the required_param() and optional_param() functions that take the URL identifier and a filter as arguments. These functions can ensure that the right URL parameters were passed and that they are of the right type, such as integer or Boolean. A full list of parameter filters is available in lib/moodlelib.php.

As many blocks are used from within courses, a course id is often passed to a plugin script. Moodle has a get_course() function, which takes a course id as parameter, and ensures that the course id value is valid. The function returns the course object, which is usually cached, so this function is equivalent to retrieving the course object from the database and passing the must exist flag, but will not query the database unnecessarily.

The last piece of PHP script security is ensuring that the script is called from within Moodle. Moodle defines a constant called MOODLE_INTERNAL, which is tested for at the top of the script with the built in PHP function defined(). Put all together, the head of most plugin scripts looks something like that in store-text.php from the Demo block:

```
require_once(__DIR__.'/../../config.php');

defined('MOODLE_INTERNAL') || die();

$courseid = required_param('cid', PARAM_INT);
$textdata = required_param('data', PARAM_RAW);

require_sesskey();

$course = get_course($courseid);
require_login($course);

$context = context_course::instance($courseid);
require_capability('block/demo:view', $context);

// Do other scripting stuff here...
```

In the above code, all the aforementioned security concerns have been addressed. The first test ensures the script is called from within Moodle. The next 2 lines ensure the cid and data parameters are present in the URL and that cid is an integer. The session key will be discussed shortly. Next, the course id value is verified as belonging to a real course and used to get the course object. The course object is used to ensure the user is logged in and enrolled in this course. Finally, the user is checked to see if they have the required (view) permission to execute this script.

The store-text.php script from the Demo block plugin is not meant to be viewed directly, as it is called via AJAX from the client side to store a string in the plugin database table. As such, there is the cross-site request forgery vulnerability to address. This vulnerability is handled with a session key that is passed to the client, then returned with the AJAX call. The use of a randomized session key ensures that the script request is valid and not a forgery. To get the session to the client, the sesskey() function is used without arguments and returns the current session key. The session key is then passed as data to the client side JavaScript, see Using JavaScript with Moodle for more details and the Demo block code for an example.

Another security issue is that of the ubiquitous SQL injection attack. Never trust input from the user! Ever! Moodle has some simple ways to ensure that input is properly sanitized before being input into the database. As has already been seen, if the data is passed to the script in the URL, the required_param() and optional_param() functions can be used to sanitize input based on the given filter. Whenever possible, use the Moodle forms API for collecting user input to ensure it is sane and valid. Use the data manipulation API rather than raw SQL queries, which pass in data values using an array, so that internal Moodle functions will sanitize the values before insertion. SQL queries should never have values concatenated to the query string, as placeholders should be used, then the data passed through an API function to build the final query. Before storing any text strings in a table, the addslashes() function should be used to ensure all quotation marks in the string are properly escaped and do not cause query failure. It may also be necessary to search the string for semi-colons and remove them to ensure no injection attack is present in the string.

User input can also be malicious when it is output to the page. Whenever user input will be displayed in a page, it must be cleaned. Moodle has some functions such as s(), p(), format_string(), and format_text() that take strings as parameters and output cleaned strings, safe for output to the page. There are also the escapeshellcmd() and escapeshellarg() functions for cleaning input before use in a shell command, if the use of a shell command simply can not be avoided.

Privacy

The introduction of the General Data Protection Regulation (GDPR) in the European Union caused Moodle to implement its privacy API in order to conform to the new law. While not currently mandatory, unless the plugin is targeted at a European audience, the implementation of a privacy provider class is recommended. Canada's privacy laws may soon resemble those of

the EU and other countries will also follow suit. The privacy provider class aims to allow users to request a copy of their data or delete their data. If your plugin does not have any external links, database tables, or user preferences, then it is possible to create a null provider, which has a single function. If you implement a privacy provider and want to test it, Moodle has some privacy utility scripts for this purpose.

Plugin contribution

Moodle has an excellent checklist page for use prior to submitting the plugin to the Moodle plugin directory. There is also a page for common mistakes in submitted plugins, which is another good resource. The plugin contribution page describes the process of submitting a plugin to Moodle, while the plugin documentation page describes how to make the plugin documentation available through its own URL.

Some of the first things to check for are that the plugin installs without a problem from a ZIP archive, see Installing a plugin for more information. All functionality of the plugin should be thoroughly tested with developer mode on and there should be no errors, warnings, or notices. It is also recommended to test the plugin with at both MySQL and PostgreSQL databases to ensure cross-database compatibility. Using the data manipulation API will also help ensure cross-database compatibility. All coding standards and styles should be followed, including comments and license information. See Coding standards for more information.

Another important consideration is that Moodle is an international project and expects everything to be in English. This includes function and variable names, as well as comments. When submitting a plugin to Moodle, only the English language file is included, as other language files will be translated from the English file. The language file itself is to be considered a raw data file rather than a PHP script, so should not use concatenation in any strings. The language file should also not contain comments or any other PHP code.

When the plugin code is up to standard and ready to submit to Moodle, a few more things are needed. The first is a hosting repository, such as GitHub, which provides source code hosting with issue tracking, another requirement for submission. The repository name must follow a standard convention, moodle-block_name. You will also need a short and long description of the plugin, know which Moodle versions your plugin supports, and some illustrative screen shots. All the PHP scripts that interact with the Moodle core APIs must be licensed under the GNU GPL v3 or later, which is included in the boilerplate for each script, and other files must either be GPL or compatible.

Once all the requirements have been met, the plugin can be submitted to the Moodle plugin directory. You will need an account for the Moodle site in order to submit a new plugin. After logging in, there is a navigation link on the right side of the page to "Register a new plugin." Follow the link and the instructions in the submission interface. After the plugin has been submitted, it will be placed in the queue with other new plugins that are awaiting review. Eventually, your plugin will be reviewed and most likely rejected, until it gets some issues

fixed. The reviewer will report the issues you need to correct to the bug tracker URL you submitted with the plugin. Fix the issues and resubmit to restart the review process.

Other resources

The primary other resource to mention are the Moodle forums, which is where you can ask for help if you run into a problem you can not find any other solution for. There are many knowledgeable Moodlers that should be able to answer your question. There is also a page that links to the core APIs, which contains the APIs mentioned here and many more. For block development specifically, there is the blocks FAQ and and an advanced blocks page that can shed further light on block development issues. Another excellent page helps new developers find their way into the code.